Western Graduate&PostdoctoralStudies

Western University
Scholarship@Western

Electronic Thesis and Dissertation Repository

November 2010

Characterizing and Diagnosing Architectural Degeneration of Software Systems from Defect Perspective

Zude Li
University of Western Ontario

Supervisor
Dr. Nazim H. Madhavji
The University of Western Ontario

Graduate Program in Computer Science

A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

© Zude Li 2010

Follow this and additional works at: https://ir.lib.uwo.ca/etd

Part of the Software Engineering Commons

Recommended Citation
Li, Zude, "Characterizing and Diagnosing Architectural Degeneration of Software Systems from Defect Perspective" (2010). *Electronic Thesis and Dissertation Repository*. 30.
https://ir.lib.uwo.ca/etd/30

This Dissertation/Thesis is brought to you for free and open access by Scholarship@Western. It has been accepted for inclusion in Electronic Thesis and Dissertation Repository by an authorized administrator of Scholarship@Western. For more information, please contact tadam@uwo.ca.

CHARACTERIZING AND DIAGNOSING ARCHITECTURAL DEGENERATION OF SOFTWARE SYSTEMS FROM DEFECT PERSPECTIVE

(Spine title: Characterizing and Diagnosing Architectural Degeneration)

(Thesis format: Monograph)

by

Zude Li

Graduate Program in Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

© Zude Li 2010

THE UNIVERSITY OF WESTERN ONTARIO
School of Graduate and Postdoctoral Studies

CERTIFICATE OF EXAMINATION

Supervisor

Dr. Nazim Madhavji

Co-Supervisor

Dr. Mechelle Gittens

Examiners

Dr. Jamie Andrews

Dr. Michael Katchabaw

Dr. Luiz Capretz

Dr. Tim Lethbridge

The thesis by

Zude Li

entitled:

Characterizing and Diagnosing Architectural Degeneration of Software Systems from Defect Perspective

is accepted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Date

Chair of the Thesis Examination Board

Abstract

The architecture of a software system is known to degrade as the system evolves over time due to change upon change, a phenomenon that is termed *architectural degeneration*. Previous research has focused largely on structural "deviations" of an architecture from its baseline. However, another angle to observe architectural degeneration is *software defects*, especially those that are architecturally related. Such an angle has not been scientifically explored until now. Here, we ask two questions: (1) *What do defects indicate about architectural degeneration?* and (2) *How can architectural degeneration be diagnosed from the defect perspective?*

To answer question (1), we conducted an exploratory case study analyzing defect data over six releases of a large legacy system (of size approximately 20 million source lines of code and age over 20 years). The relevant defects here are those that span multiple components in the system (called *multiple-component defects* – MCDs). This case study found that MCDs require more changes to fix and are more persistent across development phases and releases than other types of defects. To answer question (2), we developed an approach (called **D**iagnosing **A**rchitectural **D**egeneration – DAD) from the defect perspective, and *validated* it in another, confirmatory, case study involving three releases of a commercial system (of size over 1.5 million source lines of code and age over 13 years). This case study found that components of the system tend to persistently have an impact on architectural degeneration over releases. Especially, such impact of a few components is substantially greater than that of other components. These results are new and they add to the current knowledge on architectural degeneration. The key conclusions from these results are: (i) analysis of MCDs is a viable approach to characterizing architectural degeneration; and (ii) a method such as DAD can be developed for diagnosing architectural degeneration.

Keywords: architectural degeneration, software architecture, software defect, software change, software maintenance and evolution, case study, tool technology.

Acknowledgements

I thank my supervisor Dr. Nazim H. Madhavji and co-supervisor Dr. Mechelle Gittens, for their continuous support and help throughout my Ph.D. studies, especially for their constructive suggestions on my research. Most of all, they help me in developing the ability necessary of doing research in Software Engineering.

Thanks to the University of Western Ontario and the Department of Computer Science thereof for providing useful research resources. Special thanks to Dr. Jamie Andrews and the Graduate Secretary Janice Wiersma for their help on my studies. For my colleagues in the department who participated, guided and commented my work, thank you very much for your participation and effort. Especially, I thank these colleagues in the research group: Andriy Miranskyy, Syed Shariyar Murtaza, Dr. Remo Ferrari, and Shyamsundar B. Kulkarni. My thesis would not have been possible without their help. I also thank Quazi Abidur Rahman for his help on my studies.

My research has been partially supported by research grants from Natural Science and Engineering Research Council (NSERC) of Canada and the Centre for Advanced Studies (CAS), IBM Canada. Thanks to NSERC and CAS. Special thanks to Andriy Miranskyy, David Godwin, Enzo Cialini, and Calisto Zuzarte at IBM Canada, for their support throughout the research project.

Last but not the least, special thanks to my wife (Linda) and son (little Kui) for their motivation and support throughout my overseas studies. Such support has always been very important for me.

Contents

Certificate of Examination **ii**

Abstract **iii**

Acknowledgements **iv**

Contents **v**

List of Tables **x**

List of Figures **xi**

List of Appendices **xiii**

List of Abbreviations **xiv**

Definitions of Key Terms **xv**

1 Introduction **1**
1.1 Motivation 1
1.2 Research Questions 5
1.3 Research Preview 5
1.3.1 Case Study 1: Multiple-Component Defect (MCD) Concerns 6
1.3.2 Diagnosing Architectural Degeneration (DAD) 7
A Profile of the DAD Approach 8
Case Study 2: Validation of the DAD Approach 9
1.4 Research Contributions 10
1.4.1 Case Study 1: MCD Analysis 11
1.4.2 DAD Approach and Tool 12
1.4.3 Case Study 2: Validation of the DAD Approach 13
1.5 Thesis Structure 15

2 Related Work **16**
2.1 General Background 17
2.1.1 Software Architecture 17

2.1.2 Software Maintenance 18
2.1.3 Architectural Evolution 19
2.2 Software Aging 20
2.2.1 Causes and Properties of Aging 21
2.2.2 Code Decay 22
2.3 Architectural Degeneration 23
2.3.1 Measurement 23
Complexity Perspective 24
Maintainability Perspective 24
2.3.2 Prevention 26
Design for Change 26
Change Process Improvement 27
Reverse Engineering 27
2.3.3 Diagnosis 28
Architectural Deviation Detection 28
Defect-Prone Component Identification 29
Fault and Change Architectures 29
2.3.4 Treatment 31
Active maintainability improvement 31
Re-engineering 32
2.3.5 Analysis of Existing Diagnosis Techniques 33
2.4 Software Defects 34
2.4.1 Defect Distribution 35
2.4.2 Defect Correction Effort 36
2.4.3 Architectural Defects 36
2.4.4 Defect-Prone Components 37
2.4.5 Analysis of Existing Defect Research 38

3 MCDs and Architectural Degeneration 39
3.1 Clarifications of MCDs 39
3.1.1 MCDs vs. Interface Defects 40
3.1.2 MCDs vs. Architectural Defects 40
3.2 Understanding Architectural Degeneration 41
3.2.1 Degeneration vs. Deviation 41
3.2.2 Architectural Degeneration and MCDs 42
3.2.3 Degeneration-Critical Components 43
3.3 Key Points 43

4 Case Study 1: MCD Analysis 44
4.1 Research Questions 45
4.2 Terminology 46
4.3 Case Study Design 49

4.3.1 Description of the System and Data 49
4.3.2 Data Collection, Clean-up, and Analysis Procedures 50
4.3.3 Descriptive System Statistics 52
4.3.4 Case Study Process 53
4.4 Analysis of Data, Results, Interpretation, and Comparisons 54
4.4.1 MCD Distribution (Question (i)) 54
4.4.2 MCD Complexity (Question (ii)) 56
4.4.3 MCD Persistence (Question (iii)) 59
4.4.4 Summary of Findings 62
4.5 Threats to Validity 63
4.5.1 Data Reliability 63
4.5.2 External Validity 64
4.5.3 Conclusion Validity 64
4.6 Implications 65
4.6.1 Software Maintenance 65
4.6.2 Architectural Degeneration Treatment 66
4.6.3 Architectural Methods and Tools 67
4.7 Recap of Case Study 1 68

5 Diagnosing Architectural Degeneration (DAD) 70
5.1 Symptoms for Diagnosis 70
5.2 A Conceptual DAD Framework 72
5.2.1 Step 1: Identification of MCDs and Fix Relationships . . . 73
5.2.2 Step 2: Measurement of Components and Fix Relationships 74
5.2.3 Step 3: Identification of Degeneration-Critical Components and Fix Relationships 75
5.2.4 Step 4: Persistence Evaluation for Components and Fix Relationships 76
5.2.5 Step 5: Architectural Degeneration Evaluation 76
5.2.6 Defect Architecture Construction 76
5.3 A DAD Prototype Tool 77
5.3.1 Main Features 78
5.3.2 Data Input for the Tool 79
5.3.3 Data Processing by the Tool 80
5.3.4 Output of the Tool 81
5.4 Comparison and Discussion 82
5.5 Key Points of the DAD Approach 84

6 Case Study 2: DAD Validation 85
6.1 Research Questions and Metrics 85
6.2 Case Study Design 87
6.2.1 Description of the System and Data 88

6.2.2 Data Collection and Clean-up Procedures 89
6.2.3 Data Analysis Procedures 91
6.2.4 Descriptive Defect Statistics 92
A Basic Profile of the System 92
Defect Distributions 93
MCDs vs. Non-MCDs 95
6.2.5 Case Study Process 96
6.3 Analysis of Data, Results, Interpretation, and Comparisons 97
6.3.1 Components' Contributions (Q1) 98
Components' Measures 98
Correlation Analysis 100
Degeneration-Critical Components 102
6.3.2 Persistence of Components' Contributions (Q2) 104
6.3.3 Fix Relationships' Contributions (Q3) 107
6.3.4 Persistence of Fix Relationships' Contributions (Q4) . . . 109
6.3.5 Architectural Degeneration Trend (Q5) 112
6.3.6 Defect Architectures 114
6.3.7 Summary of Findings 116
6.4 Threats to Validity 117
6.4.1 Data Quality 118
6.4.2 External Validity 118
6.4.3 Construct Validity 119
6.4.4 Conclusion Validity 120
6.5 Implications 121
6.5.1 Methods of Architectural Degeneration Analysis 121
6.5.2 Priority Re-engineering of System Components 122
6.5.3 Empirical Research 122
6.6 Recap of Case Study 2 124

7 Case Study 1 vs. Case Study 2 126
7.1 MCD Identification 126
7.2 MCD Distribution 127
7.3 MCD Complexity Measurement 128
7.4 Further Comparative Analysis 129

8 Critical Assessment 131
8.1 MCDs and Architectural Degeneration 131
8.2 Case Study 1: MCD Analysis 133
8.3 DAD Approach and Tool 134
8.4 Case Study 2: DAD Validation 135

9 Challenges and Lessons Learnt 138
9.1 Data Access 138

9.2 Data Quality 140
9.3 Data Analysis, Risks and Scope 141
9.4 Result Interpretation and Validation 142
9.5 Academic Cycles and Industry Concerns 144
9.6 Key Points of Challenges and Lessons 145

10 Conclusions and Future Work **147**

Bibliography **150**

A DAD with Relation Algebra **161**
A.1 Motivation 161
A.2 Related Algebraic Work 164
A.2.1 Ordinary Relation Algebras 164
A.2.2 Algebras of Components and Connectors 165
A.2.3 Analysis of Existing Algebraic Work 166
A.3 Basic Notions and Notations 166
A.4 Extended Relation Algebra for DAD 168
A.4.1 Architectural Relation 168
A.4.2 AR Lifting and Lowering 170
A.4.3 Extended Architectural Relation 172
A.4.4 Attribute Aggregation 174
A.5 An Example Application 175
A.6 Algebra Implementation in the Tool 178
A.6.1 Complete EAR Construction 179
A.6.2 EAR Structure Implementation 180
A.6.3 EAR Operation Implementation 180
A.7 Discussion and Comparison 182
A.8 Short Conclusion 184

B DAD Prototype Tool Demonstration **185**
B.1 Descriptive System Statistics 186
B.2 Component Measurements 188
B.2.1 Degeneration-Critical Components 188
B.2.2 Architectural Degeneration 190
B.3 Defect Architectures 191
B.4 Short Conclusion 195

Vitae and Thesis-Relevant Publications **196**

List of Tables

4.1 Example defect records. 50
4.2 Basic profile of the subject system of Case Study 1. 52
4.3 Proportions of MCDs and their accompanying changes. 57
4.4 Accompanying changes required for MCDs. 58
4.5 Backward and forward-ratios of MCDs. 60

5.1 Key attributes of the data input. 79

6.1 Example defect-fix records (only key fields). 89
6.2 Basic profile of the subject system of Case Study 2. 92
6.3 Component measures with metrics **M1** ("%MCDs") and **M2** ("#MCDs per KSLOC"). 99
6.4 Component measures with metrics **M3** ("#Components fixed per MCD") and **M4** ("#Code files fixed per MCD"). 100
6.5 Correlations between component measures w.r.t. different metrics. 101
6.6 Cross-phase/release rank correlations (Spearman-values) between component measures. 105
6.7 Means and standard deviations (in brackets) of fix-relationships' measures. 110
6.8 Cross-release rank correlations (Spearman-values) between fix relationships' measures. 111

A.1 Algorithm of constructing complete $\mathsf{EAR}^{\mathsf{P}|\mathsf{C}}$. 179

List of Figures

1.1 Visual trend of the architectural degeneration. 4
1.2 The three main parts of this thesis research and their contributions. 10

3.1 Relationship between architectural degeneration and MCDs. . . . 42

4.1 Distributions of MCDs by components and fix relationships. . . . 55

5.1 A conceptual DAD framework. 73
5.2 Data processing by the DAD prototype tool. 81

6.1 Distribution of defects by number of components spanned. 94
6.2 Distribution of defects by number of code files spanned. 94
6.3 MCDs vs. non-MCDs. 95
6.4 Components' MCD percentage measures across releases. 105
6.5 Persistence of components' "%MCDs" measures across releases. . 106
6.6 Fix relationships' measures in release 1. 108
6.7 Persistence of fix relationships' "%MCDs" measures across releases. 112
6.8 Defect architecture (segment) of release 1 with metric **M1**. 114
6.9 Defect architecture (segment) of release 1 with metric **M3**. 115

A.1 An example architectural graph (segment). 170
A.2 A segment of the $\mathsf{EAR}^{\mathsf{C}}$ diagram for release 1. 177
A.3 EAR structure implementation. 181

B.1 Numbers of defects in the Eclipse Platform. 186
B.2 Numbers of defects in the Eclipse Platform (release 1). 187
B.3 Numbers of code files fixed in the commercial system. 187
B.4 Number of MCDs in components of Eclipse Platform (release 3). . 189
B.5 Component measures with MCD complexity metric **M3** ("#Components fixed per MCD") in the commercial system. 189
B.6 Architectural degeneration trend of the Eclipse Platform across releases, w.r.t. the MCD percentage metric **M1** ("%MCDs"). 191
B.7 Architectural degeneration trend of the commercial system across releases, w.r.t. the MCD quantity metric **M1'** ("%MCDs"). . . . 192

B.8 "Macro" defect architecture (segment) of the Eclipse Platform (release 3) w.r.t. the MCD quantity metric "#MCDs". 193
B.9 "Micro" defect architecture (segment) of component C5 in the commercial system (release 1). 194

List of Appendices

Appendix A DAD with Relation Algebra 161
Appendix B DAD Prototype Tool Demonstration 185

List of Abbreviations

Abbreviation	Full name	First use (page #)
SLOC	Source line of code	1
KSLOC	Thousand SLOC	74
DAD	Diagnose architectural degeneration	7
MCD	Multiple-component defect	3
SCD	Single-component defect	35
MFD	Multiple-file defect	94
SFD	Single-file defect	95
ADD	Attribute-driven design	18
ATAM	Architecture tradeoff analysis method	18
SAAM	Software architecture analysis method	18
CBM	Coupling between modules	24
CBMC	Coupling between module classes	24
UML	Unified modeling language	28
DPC	Defect-prone component	29
SAVE	Software architecture visualization and evaluation	29
ROSE	Re-engineering of software evolution	30
SACPT	Software architecture change propagation tool	30
ODC	Orthogonal defect classification	34
D, T, F	Development, testing, and field phases	46
A	All phases (including D, T and F)	46
BR	Backward ratio	48
FR	Forward ratio	48
Pearson-value	Pearson correlation coefficient	52
Spearman-value	Spearman's rank correlation coefficient	91
SWEBOK	Software engineering body of knowledge	66
RPA	Relation partition algebra	164
CSP	Communicating sequential process	165
AR	Architectural relation	168
AES	Architectural entity set	168
EAR	Extended architectural relation	172
SQL	Structured query language	180
FVT	Functional verification test	187
SVT	System verification test	187
PQA	Performance quality assurance	187

Definitions of Key Terms

Defect: A *defect* is "an incorrect step, process, or data definition in a computer program" (IEEE Std 610.12-1990). Here the terms "defect" and "fault" are used interchangeably.

Component: A *component* is a primary building block (or element) of a software system's architecture which encapsulates a set of closely related functionalities of this system. Note that a component is implemented by code files.

Architecture: The *architecture* of a software system is the structure or structures of the system which comprise components (or elements) and their externally visible properties and relationships (Bass et al., 2003, p. 21).

Architectural Degeneration: *Architectural degeneration* is a phenomenon where a software system's architectural change over time leads to progressive quality decline.

Multiple-Component Defect (MCD): A MCD is a defect that requires changes (fixes) in more than one component in a software system.

Fix Relationship: A *fix relationship* is a relationship among components where fixing a MCD in one component requires changes in the other components in order to fix this MCD.

Defect Architecture: A *defect architecture* is a composition of the components and fix relationships among the components of a software system.

Degeneration-Critical Component: A *degeneration-critical component* is a component in a system which contributes substantially more to the architectural degeneration than other components.

Degeneration-Critical Fix Relationship: A *degeneration-critical fix relationship* is a fix relationship in a system which contributes substantially more to the architectural degeneration than other fix relationships.

Chapter 1

Introduction

The architecture of a software system is known to degrade as the system evolves over time due to change upon change (Lehman, 1980). A degraded architecture can make future changes in the system more costly and erroneous than need be (Stringfellow et al., 2006). The phenomenon where a software system's architectural change over time leads to progressive quality decline is termed *architectural degeneration* (Lindvall et al., 2002). Such architectural degeneration is the focus of this thesis research.

1.1 Motivation

Severe architectural degeneration can trigger system *re-engineering* (Brooks, 1975, p. 123). For example, MacCormack et al. (2006) report that the Mozilla browser's code was "too tightly coupled" for ease of modifiability and was the primary cause of its re-engineering in 1998. This effort consumed five years to rewrite over seven thousand source files and two million source lines of code (SLOC, including comments) (Godfrey and Lee, 2000). Such a scenario could well be repeated in the case of the Firefox browser (Mozilla browser's successor after 2002) because, as we noted in our analysis of this browser, its architecture in 2007 was similar

to that in 1998 in terms of tightly coupled subsystems. Similarly, evolvability problems in terms of incorporating new requirements in the Linux-kernel led to a two-year effort on restructuring release 2.2, which was repeated for release 2.6 (van Gurp and Bosch, 2002). Eick et al. (2001) describe how the *modularity* of the architecture of the AT&T 5ESS telephone switching system had degraded over the period 1989 to 1996. In particular, the probability of a change touching more than one file in the system had increased from less than 2% in 1989 to more than 5% in 1996. There were indicators that systems had reached a state from which further change was not possible.

We can conclude from the above Mozilla, Linux-kernel and 5ESS issues that there is substantial evidence of the negative impact of architectural degeneration on the software business and quality of systems. Therefore, characterizing, diagnosing and treating architectural degeneration is significant for the related system quality and cost concerns in practice.

Previous research on architectural degeneration has largely focused on "deviations" from its baseline (one of its previous forms) (Hochstein and Lindvall, 2005), e.g.: *deviation detection* (Murphy et al., 2001) and *removal* (Tran and Holt, 1999) and *deviation-based degeneration measurement* (Lindvall et al., 2002). Deviations made to an architecture lead to its degeneration (Lindvall et al., 2002). For example, new *components*[1] added to a software system have more potential to contain more defects than old or modified components (Endres, 1975). However, there is a counter argument (Bhattacharya and Perry, 2007) that deviations may not always lead to degeneration because deviations show areas of change in an architecture, which does not necessarily indicate loss of functionality or of the structure of the architecture. For an architecture, deviation analysis focuses on its structural difference against its baseline, but degeneration analysis focuses on its adverse impact on system quality. Therefore, detecting, measuring and removing

[1] A component is an architectural element which is implemented by a number of code files.

deviations in a software architecture could align it better to its baseline (from a *structural* perspective) but may not effectively decrease its degeneration (from a *quality* perspective).

We were thus motivated to pursue a new line of investigation into architectural degeneration – that from the point of view of a system's "defect" quality. Fixing a defect spanning multiple system components (called a *multiple-component defect* or MCD (Li et al., 2009)) is of an architectural nature because MCDs are related to architectural problems (von Mayrhauser et al., 2000). The more the number of changes across the components to fix a defect, the more complex the fix in general. We could see above the concern for architecture degradation and change complexity in the AT&T system (Eick et al., 2001). Also, in (von Mayrhauser et al., 2000), the authors use MCDs as a basis for identifying problematic *change-coupling relationships* among components (called *fix relationships* (Li et al., 2009) – i.e., fixing one component entails fixing other related components) in a software architecture. In complementary analysis of three large software systems, DAmbros et al. (2009) use change-coupling relationships to predict defects in system components.

In this thesis research, we build upon the previous work (e.g., (Eick et al., 2001; D'Ambros et al., 2009; Li et al., 2009)) by using MCDs to characterize: (i) defect quality of components, (ii) fix relationships between components, (iii) the persistence of (i) and (ii) across system development phases and releases, and (iv) architectural degeneration through the analysis of (i) and (ii) across phases and releases. The quantity of MCDs in components, severity in terms of extent of change, and persistence over multiple phases and releases would reflect the degree to which a system's architecture is degenerating.

To better understand the phenomenon of architectural degeneration, we randomly selected 50 MCDs from each of the three successive releases of a commercial legacy system and mapped them on the system's respectively three architectures. The subject system contains 10 components, and *restructuring* was carried out on

release 3 in order to improve the system structure. This is shown in Figure 1.1, where boxes (C0–C9) denote system components and edges denote fix relationships between pairs of components. The thicker the box or edge, more MCDs contained in the component or stronger the fix relationship between the pair of components, respectively.

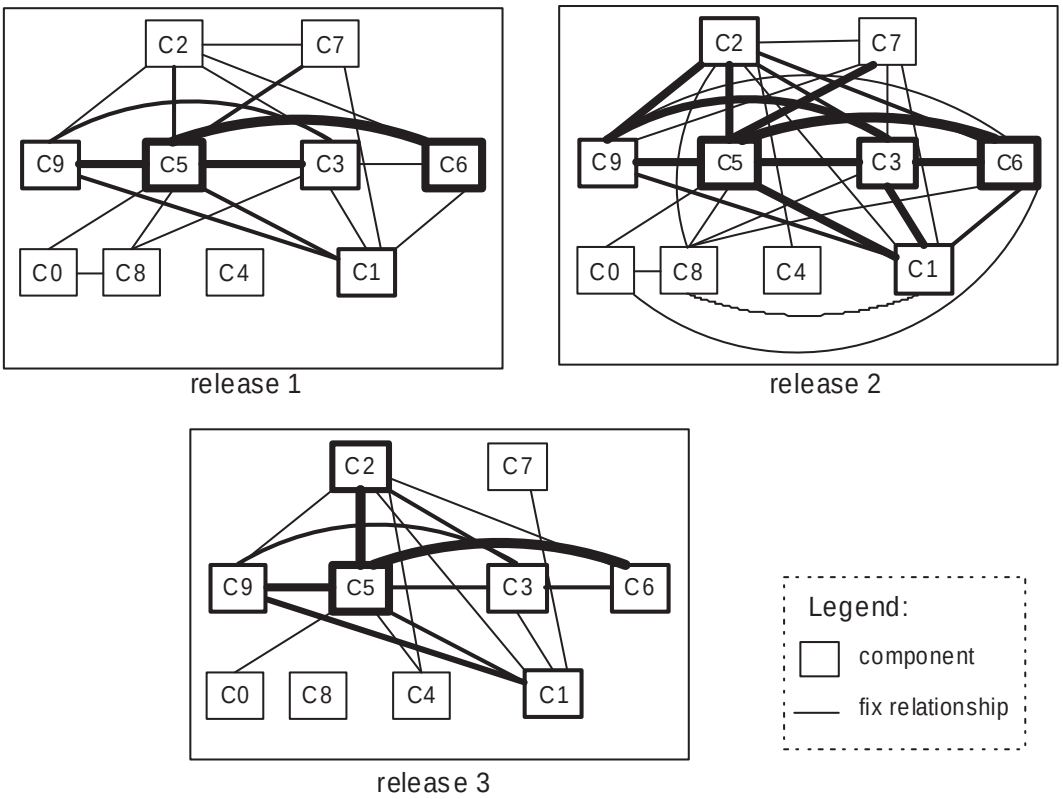


Figure 1.1: Visual trend of the architectural degeneration.

Figure 1.1 shows that MCDs had spread more widely in release 2 than in releases 1 and 3 – there are more "thick" fix relationships in release 2. That is, MCDs were more complex in release 2 than those in releases 1 and 3. This implies that, from the MCD complexity perspective, architectural degeneration of the system increased from release 1 to release 2, but decreased later in release 3 (due to restructuring). This motivated us to investigate into the trend of the system's architectural degeneration across development phases and releases.

1.2 Research Questions

This thesis research focuses on *characterizing* and *diagnosing* architectural degeneration of software systems from the defect perspective. In particular, we ask two relevant, overall questions here:

Question 1: *What do defects indicate about architectural degeneration?*

Question 2: *How can architectural degeneration be diagnosed from the defect perspective?*

Question 1 is concerned with characteristics of defects which are involved in architectural degeneration. The relevant defects here are those defects that span multiple system components (i.e., MCDs). For example, MCDs could be more and more *complex* to fix as the architecture undergoes degeneration. Therefore, the increasing complexity of MCDs could indicate the increase in architectural degeneration. Further, question 2 is concerned with how to measure architectural degeneration in relation to defects, e.g., how to measure the complexity of MCDs. Each of these two overall questions has been decomposed into several specific questions, which are described in the next section.

1.3 Research Preview

The above description of questions 1 and 2 indicates that this thesis research is centered on MCDs and architectural degeneration. MCDs are related to potential crosscutting concerns (Eaddy et al., 2008) or architectural problems (von Mayrhauser et al., 2000). Fixing a MCD requires changes in more than one component, which is a sign of problems with the software architecture (Stringfellow et al., 2006). This thesis research is thus to characterize and diagnose architectural degeneration from the MCD perspective. Here, we present a preview of the main contents of this thesis research.

1.3.1 Case Study 1: Multiple-Component Defect (MCD) Concerns

We propose that from the defect perspective, architectural degeneration manifests[2] itself through MCDs. Therefore, characteristics of MCDs, such as their *quantity* and *complexity*, can reflect the impact of architectural degeneration on software defects. Unfortunately, there is a scarcity of *quantitative* research on architectural degeneration from the MCD perspective. Therefore, the research on question 1 (see Section 1.2) was motivated to characterize the MCDs in a real software system in order to investigate defects that indicate about architectural degeneration. This research is involved in an *exploratory* case study (Case Study 1).

Case Study 1 investigates the defect history (defect records) of a large, legacy system (of size approximately 20 million SLOC and age over 20 years). The system has evolved over nine major releases, numerous minor releases and patches. The defect dataset under investigation covers 17 of over 20 years of the system.

This case study investigates the MCDs in six of the nine major releases of the subject system by answering the following three questions:

(i) *Does the 80:20 Pareto principle fit the MCD distribution?*

(ii) *To what extent are MCDs more complex than other types of defects?*

(iii) *To what extent are MCDs more persistent than other types of defects?*

Question (i) is concerned with the MCD *distribution*: whether approximately 80% of MCDs emanate from 20% of the components of the subject system. Due to the relationship between MCDs and architectural degeneration (as discussed at the beginning of this section), this question is related to the skewed distribution of architectural degeneration over the components. Question (ii) is concerned with the MCD *complexity*. The number of components that are changed in order to

[2]Later in Section 3.2.2, we describe the relationship between architectural degeneration and MCDs (their characteristics such as complexity); also see Figure 3.1.

fix a defect is a measure of complexity[3] (Endres, 1975). Further, question (iii) is concerned with the MCD *persistence* across development phases and releases. MCDs that cross phase or release boundaries are clearly not getting fixed and are being flagged again in subsequent phases or releases, which are thus *harder* to fix then non-persistent defects. The greater the complexity and persistence (difficulty) of fixing MCDs (compared against that for other types of defects), the greater the adverse impact of architectural degeneration on software defects.

Answering the above three questions, (i)–(iii), can build a quantitative profile of MCDs in terms of their (a) distribution by the components, (b) complexity in terms of the number of components changed, and (c) persistence across phases and releases. This profile reflects the impact of architectural degeneration on system defects (thus, addressing question 1 posed in Section 1.2). See Chapter 4 for Case Study 1 and this profile in detail. Moreover, this profile also shows quantitative evidence which can motivate us to create an approach to diagnose architectural degeneration from the defect perspective. This approach is described below.

1.3.2 Diagnosing Architectural Degeneration (DAD)

We propose that there are *degeneration-critical* components and fix relationships in a system which contribute substantially more to architectural degeneration than other components and fix relationships. It is necessary to properly diagnose such degeneration in relation to the degeneration-critical components and fix relationships. Unfortunately, there is a scarcity of effective techniques used to diagnose architectural degeneration. We therefore created an approach (called **D**iagnosing **A**rchitectural **D**egeneration – DAD) from the defect perspective.

This DAD approach answered question 2 posed in Section 1.2. In particular, DAD supports: (a) identification of degeneration-critical components and fix re-

[3]Fixing a defect usually requires changes in a code base. The *change span* (e.g., number of components changed for a defect-fixing request) (Eick et al., 2001) thus indicates the complexity (and the difficulty) of fixing this defect.

lationships in a given system, (b) evaluation of persistence of components and fix relationships in relation to architectural degeneration, and (c) evaluation of the trend in architectural degeneration over time. A profile of this approach is below.

A Profile of the DAD Approach

In order to support architectural degeneration diagnosis, DAD defines a suite of metrics to measure the "contribution" of a component or a fix relationship of a given system to architectural degeneration. These metrics are related to the *quantity* (in forms of proportion and density) and *complexity* of MCDs pertaining to a component or a fix relationship in the system. The number of components or code files changed in order to fix a MCD is a measure of complexity.

By measuring the components with the MCD quantity and complexity metrics, DAD can identify *degeneration-critical* components and fix relationships in the system which are components having *substantially* greater MCD quantity and complexity measures than other components and fix relationships. Based on the component and fix relationship measures, DAD can evaluate the persistence of components and fix relationships in relation to architectural degeneration, and can also evaluate the architectural degeneration of the system over time.

We have developed a prototype tool to facilitate DAD application in real contexts. This prototype tool analyzes the defect-fix history (defect records and change logs) of a given system; and beyond that, it supports automation of DAD application on the system with the help of a Relation Algebra (see Appendix A). The main outputs of this prototype tool are profiles of the system with respect to the defects (mainly MCDs) and architectural degeneration. See a detailed description of the DAD approach and its prototype tool in Chapter 5.

We also applied the DAD approach in the second, *confirmatory*, case study on a commercial software system (of size over 1.5 million SLOC and age over 13 years). The system is actually a core subsystem of the even larger system under investigation of Case Study 1 (MCD analysis). This case study is described below.

Case Study 2: Validation of the DAD Approach

Case Study 2 was conducted on three major, successive releases of the subject system. This study followed the DAD approach to measure MCD quantity and complexity for components and fix relationships of the system. Based on the measurement, this study identified degeneration-critical components and fix relationships, evaluated the persistence of components and fix relationships over time, and discovered architectural degeneration trend for the system.

In particular, Case Study 2 answered five investigative, relevant questions. First of all, there are two questions related to the components in a system:

(a) *Do some components in a system contribute more than other components to the system's architectural degeneration?*

(b) *Do components contribute persistently to architectural degeneration over development phases and releases?*

Question (a) is concerned with the *quantity* of MCDs spread across the system's components. It is also concerned with the number of components or code files changed (a complexity issue) to fix a MCD. Question (b) complements this with its focus on persistence across development phases and system releases. That is, it would highlight components that are tenacious in their defect quality and across phases and releases.

Following this, there are two more questions, (c) and (d) for "fix relationships" analogous to questions (a) and (b). In particular, we are interested in knowing (c) *whether some fix relationships contribute more than others to architectural degeneration*; and (d) *whether fix relationships are persistent over phases and releases*. Based on these four questions, the last question (e) is: *What is the trend in architectural degeneration from the defect perspective?* This question examines architectural degeneration across phases and releases based on measurement of components and fix relationships.

These are clearly important questions to ask about system management; we saw earlier the price to pay due to degenerating architectures (see Section 1.1). However, a key aspect of Case Study 2 is that it shows how we can systematically analyze MCDs to characterize the trend in architectural degeneration (e.g., see Figure 1.1). This complements the deviation perspective taken in the literature. See Chapter 6 for the details of Case Study 2.

1.4 Research Contributions

The above research preview indicates that this thesis research contains three main parts: Case Study 1 (MCD analysis), the DAD approach (and its prototype tool), and Case Study 2 (DAD validation). Following this preview, we discuss the contributions of these three parts, see Figure 1.2 for a profile.

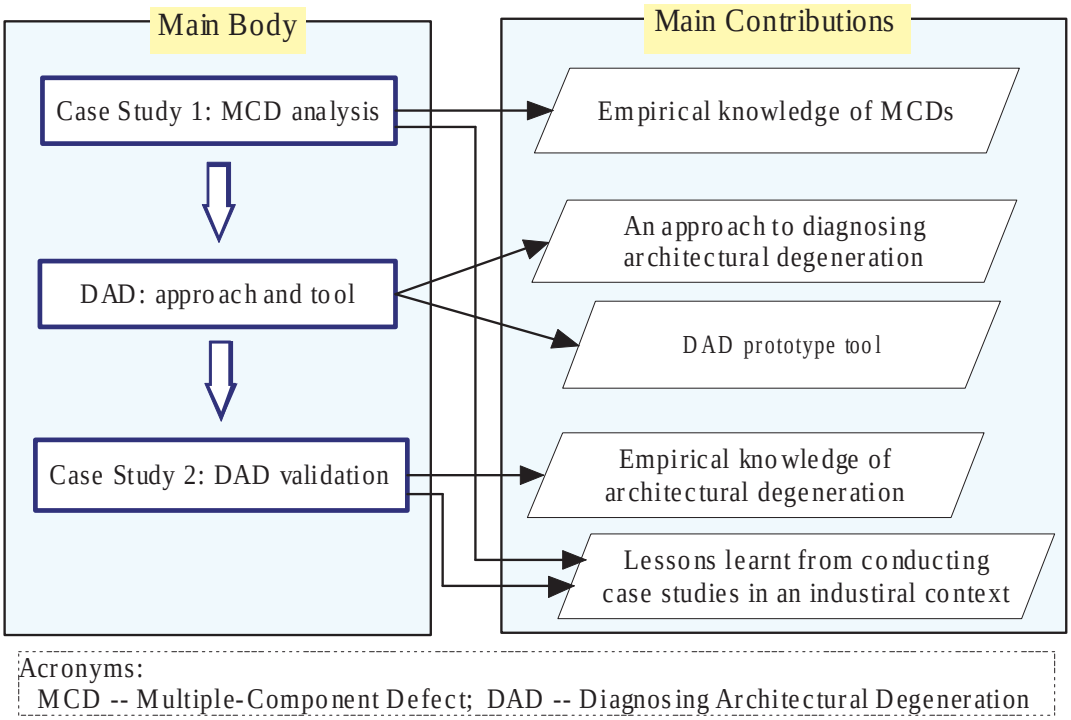


Figure 1.2: The three main parts of this thesis research and their contributions.

The overarching contribution of this research (specifically its three main parts) is new, empirical knowledge on architectural defects (MCDs) and degeneration in

software systems, which can benefit system evolution and quality improvement. The existing knowledge reflects the state of current theory on architectural degeneration, i.e., that the degeneration of an architecture can be determined based on its deviations made against a baseline. This thesis work adds that architectural degeneration can be determined through analysis of MCDs (a quality perspective). The following three subsections give the details.

1.4.1 Case Study 1: MCD Analysis

We note from Section 1.3.1 that Case Study 1 investigates the distribution, complexity and persistence of the MCDs in six releases of the subject system. The relevant quantitative findings are described below.

(1) The Pareto principle fits the MCD distribution by components: over 80% of MCDs emanate from 20% of the components.

(2) MCDs are more complex to fix than non-MCDs. On average, fixing a MCD requires near 3 times changes (based on components) as much as that for fixing a non-MCD.

(3) MCDs are more persistent across development phases and releases than non-MCDs. The proportion of MCDs crossing over from one phase or release to the next is 6.0-8.5 times as much as that for non-MCDs.

The above findings depict a quantitative profile of MCDs in the subject system, which addresses questions (i)–(iii) posed in Section 1.3.1. This profile also has implications for software maintenance and quality improvement. For example, separating MCDs from other types of defects can help focus attention on these hard-to-fix defects, and more correction and testing efforts should be focused on MCDs (because of their persistence) to avoid leaking into successive releases.

This case study is new and there are no similar studies in the scientific literature. Historically, MCDs have been associated with “interface” defects. For example, Endres (1975) defines an interface defect as one that requires “changes”

to more than one module in order to fix it. Previous research has focused on the "extent" of different types of defects, including interface defects, in software systems. It has been found that interface defects account for 5% (Endres, 1975), 11% (Basili and Perricone, 1984), 6%-15% (Basili and Shull, 2005), and 4%-24% (Weiss, 1979) of all defects in software systems. The case study investigates a different line of inquiry focused on MCDs, specifically, their distribution, complexity and persistence over time. These three aspects were not investigated in the previous studies (e.g., (Endres, 1975) and (Basili and Perricone, 1984)). The quantitative findings (as shown above) of these aspects reflect the impact of architectural degeneration on defects, which can thus help understand the architectural degeneration in the system.

1.4.2 DAD Approach and Tool

The DAD approach operationalizes a defect perspective with MCD quantity and complexity metrics to identify degeneration-critical components and fix relationships, to evaluate the persistence of components and fix relationships in relation to architectural degeneration, and to evaluate architectural degeneration over time. In particular, degeneration-critical components are identified specifically as components that have the substantially greater MCD quantity or complexity measures than other components in a given system. Likewise for degeneration-critical fix relationships. Also, the architectural degeneration is claimed to be increased if the system's MCD quantity and complexity measurements increased with time. Otherwise, it is claimed to be decreased or mitigated.

Through applying DAD on a given system, the derived information of architectural degeneration can help improve the system's quality. Existing techniques cannot offer this information. We note from Section 1.1 that previous research has centered on detection (Murphy et al., 2001) and removal (Krikhaar et al., 1999) (Tran and Holt, 1999) of architectural deviations. DAD, on one side, is obviously

different from these deviation-related techniques. On another side, its defect perspective can complement the deviation perspective for architectural degeneration diagnosis (Lindvall et al., 2002) (Bhattacharya and Perry, 2007). For example, the MCD quantity and complexity metrics defined in DAD can be used to measure the deviations made in an architecture. In this situation the deviations that lead to significant increase in the system's MCD quantity or complexity should be identified as the degeneration-critical components for intensive attention.

Moreover, the DAD prototype tool can create profiles of a given system and its defects and diagnose its architectural degeneration over time. Therefore, this tool can complement existing techniques and tools for system quality improvement, such as architectural deviation detection and removal (Lindvall and Muthig, 2008), architectural transformation (Krikhaar et al., 1999) (Fahmy and Holt, 2000), and re-engineering (Chikofsky and Cross, 1990).

1.4.3 Case Study 2: Validation of the DAD Approach

We note from Section 1.3.2 that Case Study 2 applied the DAD approach on a real software system. That is, it identified the degeneration-critical components and fix relationships and evaluated the architectural degeneration over time for the subject system. Beside the DAD application, this case study also addresses the five questions, (a)–(e), posed in Section 1.3.2. The relevant results of this study are summarized below (see Section 6.3).

(1) There are 20% of the system's components and 10% of fix relationships which exhibit over 70% of MCDs. Among these, there are few components and fix relationships that are associated to relatively more complex MCDs. These components and fix relationships are *degeneration-critical*.

(2) The system's components tend to persistently have an impact on architectural degeneration over multiple releases of the system; however, such persistence does not apply to the fix relationships.

(3) As the system evolves, the trend in architectural degeneration may increase or decrease, depending on whether any treatments were made.

These results add to the current knowledge on architectural degeneration, which can aid architectural degeneration treatment and system evolution. For example, the degeneration-critical are the few-but-vital components in the system which have to be fixed and tested more thoroughly than other components. Meanwhile, the components that contribute persistently to the architectural degeneration have to be treated with intensive attention. Overall, we conclude that architectural degeneration can be characterized from the perspective of defects, which complements the characterization from the structural or deviation perspective (Lindvall et al., 2002; Hochstein and Lindvall, 2005).

Historically, there are some studies which address architectural degeneration. For examples, Brooks (1975, p. 123) and Belady and Lehman (1976; 1980) find that any system will eventually require a redesign due to continuing structural deterioration. Eick et al. (2001) then find that improper architectural design is one of the main factors causing code decay (the phenomenon where change to a system becomes more difficult than before). Bhattacharya and Perry (2007) described the structural deviation and functionality loss of a system with time. Lindvall et al. (2002) discussed the problem of system complexity increase due to architectural deviations. However, there are few studies *quantitatively* characterizing, or monitoring, architectural degeneration. For a system, such quantitative characteristics are credible and are important especially when the organization is planning to act for controlling and treating the architectural degeneration. This case study spanned this gap, which measured architectural degeneration in a system.

Moreover, there are challenges and lessons learnt from conducting Case Studies 1 and 2, which are mainly related to the data access, quality, cleaning, analysis, and interpretation, academic cycles, industry jitters, etc. They can benefit future empirical studies conducted in industrial contexts.

Over the two case studies and the DAD approach and its prototype tool, we claim that this thesis research adds to the current knowledge on software defects (e.g., (Endres, 1975) and (Basili and Perricone, 1984)) and the theory of architectural degeneration (mainly, deviation-based evaluation (Lindvall et al., 2002)), and also enriches the methodology and technology of handling (detecting and fixing) software defects and characterizing and diagnosing architectural degeneration. The key conclusions from this work are: (i) analysis of MCDs is a viable approach to characterizing architectural degeneration; and (ii) a method such as DAD can be developed based on MCD characteristics for diagnosing architectural degeneration from the defect perspective.

1.5 Thesis Structure

The thesis is organized as follows. Chapter 2 outlines related work of this thesis research. Chapter 3 introduces the core concepts for this research. Later in Chapter 4, we elaborate Case Study 1. In Chapter 5, we illustrate the DAD approach and its prototype tool. In Chapter 6, we elaborate Case Study 2. Then in Chapter 7, we compare these two case studies. After that, we assess the main achievements and limitations of this thesis research (mainly its three parts shown in Chapters 4–6) in Chapter 8. We then discuss the challenges and lessons learnt from conducting the case studies in Chapter 9. Finally, we conclude the whole thesis and describe the future work in Chapter 10. Appendix A defines the Relation Algebra underlying the DAD prototype tool, which is used to implement the DAD features in the tool prototype. Appendix B demonstrates typical outputs of applying this prototype tool on the open-source Eclipse Platform[4] and the subject commercial system of Case Study 2.

[4]See an introduction to the Eclipse Platform at `http://www.eclipse.org/platform/` (last access in November 2010).

Chapter 2

Related Work

This chapter outlines related work centered on architectural degeneration and its diagnosis. We first describe relevant background knowledge on software architecture and maintenance (see Section 2.1), then dig into three fundamental aspects of the architecture and maintenance research:

(1) understanding the software aging phenomenon (e.g., its causes and properties) which includes architectural degeneration; see Section 2.2;

(2) handling architectural degeneration, including typical techniques for measurement, prevention, diagnosis and treatment of architectural degeneration; see Section 2.3; and

(3) characterizing software defects (e.g., their distribution, complexity and cost measurements, etc.), see Section 2.4.

These three aspects are closely related to the work on characterization and diagnosis of architectural degeneration – the focus of this thesis research. Based on the related work description within these three aspects, we then discuss the specific motivations for the relevant work of this thesis research, which includes Case Study 1 (analysis of multiple-component defects or MCDs – see Chapter 4), the DAD approach (and its prototype tool – see Chapter 5), and Case Study 2 (DAD validation – see Chapter 6).

2.1 General Background

The phenomenon of architectural degeneration is related to software architecture and maintenance. As the size and complexity of software systems rapidly grow, the architecture becomes more important (Garlan and Shaw, 1993) and the maintenance becomes more expensive (Sutherland, 1995). They are thus more and more critical for success of software development projects.

2.1.1 Software Architecture

The *architecture* of a software system is the structure or structures of the system which comprise components (or elements) and their externally visible properties and relationships (Bass et al., 2003, p. 21). The architecture of, especially, a large software system is firmly entrenched as amongst the key information artifacts of a software organization (Bass et al., 2003, p. xi). Its development can galvanize the diverse stakeholders into action towards a common goal of realizing the envisaged system or maintaining the delivered system (Clements et al., 2002, p. 10). In many cases, it can influence, or even dictate, the organization of the various development teams (Booch, 2007).

From a structural perspective, an architecture captures the structure of a system in terms of the components and how they interact (Gorton, 2006). As a conceptual solution, an architecture captures the foundational design decisions made early in the development process (Jansen and Bosch, 2005) (Bass et al., 2003, p. 26). These design decisions typically need to consider the various system qualities (such as performance, reliability, modifiability, usability, and others), which are central to the system's success (Clements et al., 2002, pp. 1-2). Consequently, the architecture is difficult to change much later in the project or after the system's release (Fowler, 2003).

An architecture is generally formed during system design. Such an architecture captures the design decisions made prior to system construction, which is often

termed the *conceptual* (as-designed or as-intended) *architecture* (Bowman and Holt, 1999). There is also an architecture existing in the code, which captures the real structure of the implemented system. Accordingly, such an architecture is termed the *concrete* (as-implemented) *architecture* (Bowman and Holt, 1999).

Research in software architecture focuses on topics such as the choice of architectural drivers, design tactics and patterns (Bass et al., 2003, ch. 5), the use of particular architectural methods (e.g., ADD (Bass et al., 2003, ch. 7), ATAM (Clements and Northrop) and SAAM (Kazman et al., 2002); also see the survey in (Dobrica and Niemela, 2002)), requirements-architecture intertwining (Nuseibeh, 2001), architectural recovery (Biggerstaff, 1989) (Chikofsky and Cross, 1990), transformation (Fahmy and Holt, 2000), evolution (McNair et al., 2007), and visualization (Gallagher et al., 2008). Diagnosis of architectural degeneration – the focus of this thesis research – is closely related to architectural recovery and evolution. Architectural degeneration mostly happens during system evolution, and its diagnosis is usually based on the concrete architecture recovered from a system's code base; see several example techniques for diagnosis in Section 2.3.3.

2.1.2 Software Maintenance

Software maintenance[1] refers to "any work that is undertaken after delivery of a software system" (Cornelius et al., 1989) (see IEEE Std 1219:1998). In particular, it includes *corrective, adaptive, perfective* and *preventive* maintenance (Lientz and Swanson, 1980) (Cornelius et al., 1989) (also see ISO/IEC 14764:2006). The software maintenance costs increase continually over the last 40 years: from approximately 40% of the total software lifetime costs in the early 1970's to 80%-90% in the early 1990's (Pigoski, 1997, fig. 3.1) (also see (Seacord et al., 2003, fig. 1-

[1]In this thesis, the terms "software maintenance" and "software evolution" are usually used interchangeably. However, "software maintenance" is used with implicit meaning of fixing system problems discovered after delivery, and "software evolution" is used with implicit emphasis on system change process over time.

2)) and even over 90% after 2000 (Erlikh, 2000). As Jones[2] reported, software maintenance costs continually increase as the 21st century advances, which has caused more than 50% of the software population to be excessively engaged in maintenance rather than new development.

Software maintenance is mostly a value-adding process (Belady and Lehman, 1976). It keeps the system operating as expected by fixing defects and adding new enhancements. It is also a complexity-increasing process and thus a cost-increasing process. Belady and Lehman's *laws of software evolution* (1976; 1980) indicate that a system under maintenance increases in complexity over time, which leads to increased costs for future development. When the increased cost exceeds the added value, the system could be phased out and new development could be considered as a replacement (Bennett and Rajlich, 2000).

Research in software maintenance focuses on topics such as maintenance classification (Lientz and Swanson, 1980) (Cornelius et al., 1989), change process (Madhavji, 1992) (Sousa and Moreira, 1998), program comprehension (Corbi, 1989) (Padula, 1993), defect analysis (Basili and Perricone, 1984) (Shull et al., 2002), and change impact analysis (Arnold, 1993) (Bohner and Arnold, 1996). The work on architectural degeneration diagnosis is related to defect analysis, especially characteristics of defects spanning multiple components in the architecture. The related work on defect characteristics is discussed in Section 2.4.

2.1.3 Architectural Evolution

A system with a well-designed architecture is relatively easy to change (Hsia et al., 1995). Also the architecture (if documented) could guide system comprehension and change (e.g., ripple effect[3] detection) (Bass et al., 2003, ch. 2). Meanwhile, system change (during evolution) can affect (mostly, "destroy" (Brooks, 1975,

[2]Read Jones' talk in 2007, entitled "Geriatric Issues of Aging Software"; see `http://www.stsc.hill.af.mil/crossTalk/2007/12/0712Jones.html` (accessed in November 2010).

[3]Ripple effect is "effect caused by making a small change to a system which affects many other parts of a system" (Stevens et al., 1974).

p. 122)) the architecture (Williams and Carver, 2010). For example, change in the code base of a system can modify the concrete architecture and cause it to be inconsistent with the conceptual architecture (Perry and Wolf, 1992). Here, change to the architecture form or its properties or constraints is called *architectural change* (Krikhaar et al., 1999). Such architectural change is usually unintentional and is likely to affect the whole system; and its impact is often more adversely than that for change confined in a small area of the system (Nedstam et al., 2004). For example, architectural change tends to destroy the original architecture, leading to major quality problems (Brooks, 1975, p. 122) (Williams and Carver, 2010).

Architectural degeneration is caused only by architectural changes that lead to decline in system quality (Perry and Wolf, 1992), which are called "dirty" architectural changes. Because the term "quality" is usually operationalized with quality attributes such as maintainability, reliability, etc. (see ISO 9126-1 for an example quality model), whether an architectural change is "dirty" or not should be determined with relation to a specific quality attribute. Therefore, diagnosis of architectural degeneration should first clarify the particular (quality) perspective that it focuses upon. Consequently, characteristics of a system with respect to a quality attribute can reflect an aspect of its architectural degeneration. This is the fundamental idea of this thesis research on architectural degeneration diagnosis, see Section 3.2.2 for a detailed discussion.

2.2 Software Aging

Software, like humans, ages with time (Parnas, 1994). Software ages as its value declines but its cost increases. Parnas (1994) terms this phenomenon *software aging* (mirror human aging). Software aging is a more generalized phenomenon of architectural degeneration. We thus give a basic understanding of software aging in this section, mainly covering its causes and properties.

2.2.1 Causes and Properties of Aging

Brooks, Belady and Lehman pioneered the early software aging research in the 1970's. For example, Brooks (1975, p. 123) noted that: "All repairs tend to destroy the structure, to increase the entropy and disorder of the system." Meanwhile, Belady and Lehman's *law of increasing complexity* (1976) states that as a system evolves, its complexity increases unless work is done to maintain or reduce it. Both Brooks (1975, p. 123) and Belady-Lehman (1976) argued that change to software *tends to* degrade its structure; the degraded structure then resists future change and ultimately leads to system redesign. Change to software is risky (Sommerville, 2006, p. 500). Some change degrades the software structure, which is termed "*dirty*" (or "muddy") *change*. Dirty change is the main cause of software aging during maintenance. As Jones said (see footnote 2 in page 19), "maintenance of aging software tends to become more difficult year by year since updates gradually destroy the original structure of the applications."

Beside, software aging can be also caused by "inability to change" (Lyons, 1981, p. 337) or "lack of change" (Parnas, 1994) (i.e., failure of meeting change requirements). As Lyons (1981, p. 337) stated, "software deterioration [aging] results from software's inability to change itself to match the changing decisions processes of the enterprise. Software progressively loses its productive capacity unless it is continually infused with the ongoing changes in the enterprise's decision system." For a system, its use environment and requirements are changing with time. This requires the system to change itself correspondingly. Otherwise, it will age quickly and be phased out soon (Bennett and Rajlich, 2000).

From the literature (e.g., (Parnas, 1994) and (Eick et al., 2001)), we conclude that software aging has the following five properties:

First, aging is inevitable. No matter how great and perfect a software system is now, it will be disconnected from its users eventually.

Second, aging is temporarily resistible and reversible. There are *viable* techniques

which can *temporarily* resist aging and reverse its effect.

Third, software ages diversely. Such aging diversity manifests itself as various aging causes, effects and symptoms in software systems.

Fourth, software that ages may nevertheless increase in value but increase more in cost side by side. Generally, the software under maintenance has passed the peak time of value adding and entered into the time of cost increasing.

Fifth, software aging mainly degrades the system quality, but also affects the functionality enhancement. Software quality decline leads to maintenance cost increase, which thus retards the system functionality change process.

2.2.2 Code Decay

Code decay is a code-level aging phenomenon in which the difficulty and cost of change in code increase with time (Karr et al., 1996). It is pervasive in software systems (Eick et al., 2001). Ohlsson et al. (1998) propose an approach to classify system components based on the amount of decay: *green* (normal evolution), *yellow* (code decay) and *red* ("mission impossible") components. The amount of decay is measured by the *number of defects*, *time* to perform certain maintenance activities, and the *complexity* of the component. Red components require the most attention during system maintenance and evolution.

Graves and Mockus (1998) suggest that the increase rate of *change effort* is "the most prominent indicator of code decay as well as a trigger for maintenance process improvement". Eick et al. (2001) suggest to measure code decay in terms of *change effort* (e.g., number of components changed and number of lines of code added/deleted), *interval* (calendar time required) and *quality*. They find that the code of a large telephone switching system had decayed from 1989 to 1996. For example, the probability that a change touched more than one file in the system increased from less than 2% in 1989 to more than 5% in 1996. Stringfellow et al. (2006) suggest that "increasing difficulty [of change] ... is a sign of code

decay", where the difficulty of change is measured in terms of *change size* (i.e., number of source lines of code added, deleted or modified) and *span* (i.e., number of components or code files that are changed).

Eick et al. (2001) suggest that inadequate architectural design is one of the most critical factors of code decay. We note from Section 2.1.3 above that the architecture of a system could change (mostly, degrade) as the system evolves. The degraded architecture differs from the original architecture (Bowman and Holt, 1999) and usually leads to substantial increase in code decay. We discuss this phenomenon – architectural degeneration – as below.

2.3 Architectural Degeneration

Architectural degeneration (or erosion) is an architecture-level aging phenomenon. In comparison against code decay, it is concerned more with change to high-level system structures (Lindvall et al., 2002) (Hochstein and Lindvall, 2005). Lindvall et al. (2002) suggest that a system is suffering architectural degeneration if its "concrete" architecture has major *deviations* from its "conceptual" architecture. There are some studies in literature which aimed to "measure", "prevent" (resist), "diagnose" and "treat" (reverse) architectural degeneration in real systems; see these four subsections below: 2.3.1–2.3.4.

2.3.1 Measurement

Change to an architecture may lead to *deviation*, which indicates its degeneration (Bhattacharya and Perry, 2007). Existing measurements of architectural degeneration are mainly based on architectural deviations. In particular, we describe two categories of these measurements, which are related to increase in system-structure complexity and decline in system maintainability (mainly modularization) due to architectural deviations. Note that system complexity has a strong relationship with maintainability (Kafura and Reddy, 1987).

Complexity Perspective

As early as the 1970's, Belady and Lehman (1980) noted that the complexity of a system increases as it evolves. There is a wealth of literature which has described the adverse impact of complexity to system quality and costs. For example, Banker et al. (1993) suggest that the maintainability and maintenance costs of a software system are substantially affected by its complexity. Also it has been well recognized that system modules of the highest complexity tend to contain the most defects[4].

Jaktman et al. (1999) suggest that the increase in system complexity indicates architectural degeneration. They define several system-complexity metrics for architectural degeneration measurement. Two examples of such metrics are: average number of components per relative call graph level (hierarchical complexity) and average number of calls per component (structural complexity). Their study on a software system indicates that its structural complexity increased continuously across six releases. Similarly, van Gurp and Bosch (2002) define several architectural complexity metrics such as numbers of packages, classes, functions, or non-commented source statements. They find that the complexity increased substantially over the evolution of a five-release system.

Maintainability Perspective

Lindvall et al. (2002) define two maintainability metrics to measure architectural degeneration: "coupling-between-modules" (CBM) and "coupling-between-module-classes" (CBMC). The CBM metric is the number of non-directional references between modules; and the CBMC metric is the number of non-directional, class-to-class, references between modules. An architecture with low CBM and CBMC measurements is of low degeneration degree. Through a case study on a

[4]See Enerjy's article "McCabe Cyclomatic Complexity: the Proof in the Pudding": `http://www.enerjy.com/blog/?p=198` (last access in November 2010).

two-version system they find that the new version released after restructuring the whole system has lower CBM and CBMC values than the old version.

More interestingly, Andrews et al. (2000) define "defect-cohesion" and "defect-coupling" metrics to measure the "degeneration" of a component or a fix relationship between two components. The defect-cohesion of a component is measured by the total number of code files changed to fix the defects pertaining to the component. Components having the greatest defect-cohesion measurements are the most severely degenerated. The defect-coupling of a fix relationship between two components is measured by the number of code files changed in each of the two components in order to fix the defects spanning the two components. Fix relationships having the greatest defect-coupling measurements are the most severely degenerated. These two metrics are used to derive *fault architectures* from a system's defect history, see Section 2.3.3.

In addition, Bhattacharya and Perry (2007) define metrics to measure architectural deviation from a functional perspective as well as from a structural perspective. Functional deviation measures "how a given version of software deviates from the baseline in terms of the architectural services and data supported". Structural deviation measures "the deviation in terms of architectural characteristics captured in the Form of the architectural description of our abstract architecture model". Examples of such functional or structural deviation metrics are numbers of eroded (deleted, dead or degraded) functionalities, attributes or services of an architecture from its baseline. They also define metrics to measure architectural degeneration based on loss of functionality and architectural structure over time. Application of the deviation and degeneration metrics can help identify the problematic areas (e.g., change-active components) in an architecture.

Measuring the architectural degeneration is the first step towards its diagnosis and treatment, which are described in Sections 2.3.3 and 2.3.4. Note that a deviation to an architecture does not always indicate its degeneration, especially

when, for example, most of defects introduced by this deviation are confined to a very limited area in the system and are not related to any architectural problems. For an architecture, its deviation measurement focuses on its structural difference against its baseline, but its degeneration measurement is based on the impact of architectural change on system quality (e.g., defects).

2.3.2 Prevention

We note that architectural degeneration (as a variant of software aging) is inevitable but resistible (see Section 2.2). Thus the section title "prevention" means not to really avoid architectural degeneration but to resist or postpone it. Next, we outline three usual prevention techniques: *design for change*, *change process improvement*, and *reverse engineering*.

Design for Change

Design for change means designing a software system to *embrace* change in future (Parnas, 1994). Its core principle is *separation of concerns* (or *modularization*) (Parnas, 1979). This means that modules of a system should implement separated concerns, and couplings among modules must be well outlined and keep loose in inter-module coupling. Parnas (1979) states that the most critical step to design an easily changeable system is the design of *use* relationships between software elements (e.g., modules, files, classes, etc.). Ideally, design for change can confine most likely occurred changes to a limited amount of system parts; the consequent, adverse effects on system quality are thus reduced. However, future changes are not always predictable. Some changes (out of prediction) may still span the architecture and inflict adverse impact on system quality. These changes are the so-called "dirty" architectural changes (see Section 2.1.3), which lead to architectural degeneration. This is one of the reasons why architectural degeneration is inevitable during evolution of large software systems.

Change Process Improvement

Some change to a system requires *quick* accomplishment (Sommerville, 2006, p. 500), which thus causes software aging. Improving the change process with *impact analysis* and *regression testing* or *review* can reduce this risk (Rothermel and Harrold, 1997). Impact analysis is to identify what to modify or to identify the potential change consequences (Arnold, 1993); regression testing is to retest the modified software parts, in order to assure that a known defect has been successfully fixed (Leung and White, 1989); change review is used widely for any type of change, in order to assure that the change request has been successfully accomplished without introducing defects (Ciolkowski et al., 2003). These techniques can help reduce the adverse impact of quick-and-dirty change on software quality. Nedstam et al. (2003) proposed a generic architectural change process, which aims at implementing architectural change considering its technical and organizational impacts. However, due to limitations in time, cost and other resources, these techniques may not be used properly in such situations as emergent change, agile development and distributed development and maintenance.

Reverse Engineering

Reverse engineering is a process of creating representations of a software system in another form or at a higher level of abstraction (Chikofsky and Cross, 1990). *Architectural recovery* is a regular form of reverse engineering in practice. Generally, architectural recovery has three steps (Krikhaar, 1997): (i) *extract* basic information from the source code of a software system; (ii) *abstract* a manageable set of architecture-significant information from the basic information; and (iii) *represent* the abstracted information with visual forms (e.g., box-and-arrow graphs (Holt, 1998)). Architectural recovery captures knowledge about the software, which helps in understanding the software and guiding software change. It is also used to detect deviations between architectures; see next Section 2.3.3.

We note from the above discussion that these "prevention" techniques can only resist or postpone, but cannot avoid, architectural degeneration. Although architectural degeneration is inevitable (see Section 2.2.1), these "prevention" techniques can help reduce the damage to system quality (due to architectural change). which is obviously important for system quality and cost concerns. For an architecture that is under degeneration, adequate diagnosis techniques can quantify the adverse impact of its degeneration on system quality.

2.3.3 Diagnosis

Architectural degeneration diagnosis is one of the focuses of this thesis research. It is mainly to identify the most degenerated architectural areas (the so called, "degeneration-critical" components) in a system and to evaluate the architectural degeneration over time for that system. Here, we describe three usual diagnosis techniques: *architectural deviation detection*, *defect-prone component identification*, and *fault* and *change architectures*.

Architectural Deviation Detection

Generally, architectural deviation detection has three steps: (i) *architectural recovery* (as described in Section 2.3.2); (ii) *formal architectural specification*, i.e., describing the architecture with formal languages (e.g., UML[5] (Shin et al., 2006) (Lindvall and Muthig, 2008)) or models (e.g., Murphy et al.'s *software reflexion* model (2001)); and (iii) *architectural comparison*, i.e., identifying deviations between architectural specifications by comparison.

Further, Krikhaar et al. (1999) propose an approach to remove architectural deviations by *transformation*. Examples of architectural transformations are create, delete, and rename a unit, isolate and move a function, etc. This approach has been further extended in Tran and Holt's approach of *forward architectural*

[5]UML stands for Unified Modeling Language; see a detailed description of UML at `http://www.uml.org` (last access in November 2010).

repair (1999), which is to repair the concrete architecture of a system to match the conceptual architecture (the baseline). Lindvall et al. (2007; 2008) have developed a tool named Software Architecture Visualization and Evaluation (SAVE), which is proposed to allow software architects to navigate, visualize, and evaluate architectural deviations occurred over time for a specific system. In this tool, UML notations are used to describe architectures.

Defect-Prone Component Identification

The defect history of a system manifests the components that contain the greatest number of defects in the system, which are often termed *defect-prone components* (DPCs). From a defect perspective, DPCs could be considered as the most degenerated areas in the architecture. Generally, DPCs are identified according to the Pareto-shaped defect distribution: 20% of the components contain 60%-80% of defects in the system (Boehm and Basili, 2001) (Ostrand et al., 2005). As a rule of thumb, DPCs refer to the top α components that contain the greatest number of defects in the system. The threshold α is set to 25% in Ohlsson and Wohlin's study (1998) and 20% in our earlier study (Li et al., 2009). Many studies (e.g., (Ohlsson et al., 1999), (Andrews and Stringfellow, 2001), and (Andersson and Runeson, 2007)) have found that DPCs tend to persist across development phases and releases. This persistence is further discussed in Section 2.4.4.

Fault and Change Architectures

We note from Section 2.3.1 that Andrews et al. (2000) define "defect-cohesion" and "defect-coupling" metrics to measure the degeneration of a component or a fix relationship in a system. Further, von Mayrhauser et al. (2000) propose an approach to derive *fault architectures* from system defect history using these metrics. A fault architecture is a composition of fix relationships among components in a system. Recall that two (or more) components have a *fix relationship* if changes made in one component are coupled by changes in the other(s) in order

to fix an multiple-component defect (MCD) (Zimmermann et al., 2004). As a rule of thumb, the top β fix relationships that involve the greatest number of MCDs are termed *defect-prone* fix relationships. In the literature, β is set to 5%-15% in Stringfellow and Andrews' study (2002) and 10% in our earlier study (Li et al., 2009). Fault architectures can highlight those defect-prone fix relationships over a system's architecture.

Likewise, Stringfellow et al. (2006) propose a similar approach to deriving *change architectures* from system change history. Note that the change architecture partially overlaps with the fault architecture. Related research is related to the *change-coupling* relationships[6] among system components. Zimmermann et al. (2004) propose a prototype tool (ROSE – Re-engineering Of Software Evolution) to determine change-coupling relationships at different levels of granularity (such as directories, modules, files, methods, variables and sections). Ying et al. (2004) use an association mining technique to find potential *change patterns* (pairs or sequences) in source code from the defect histories of the Eclipse[7] and Mozilla[8] projects. Abdelmoez et al. (2004) have developed the tool SACPT (Software Architecture Change Propagation Tool) to measure the likelihood that a change that arises in one component propagates to other components. These approaches focus on changes to related system elements at varying levels of granularity. They are deemed to assist in modification activities.

Overall, diagnosis of architectural degeneration in a system, by identify deviations (Hochstein and Lindvall, 2005) and DPCs (Ohlsson and Wohlin, 1998) and deriving fault and change architectures (von Mayrhauser et al., 2000) (Stringfellow and Andrews, 2002), offers valuable information for the treatment.

---

[6]Two (or more) components in a system have a change-coupling relationship if change made in one component are coupled with changes in the other component(s) in order to complete the intended change (Zimmermann et al., 2004). Fix relationship is a special form of change-coupling relationship in relation to defect correction.

[7]The Eclipse project website: `http://www.eclipse.org` (last access in November 2010).

[8]The Mozilla project website: `http://www.mozilla.org` (last access in November 2010).

2.3.4 Treatment

We note from Section 2.2.1 that software aging is temporarily reversible; so is architectural degeneration. It is thus possible to treat architectural degeneration for software systems, i.e., to reduce or mitigate the impact of architectural degeneration on system quality (e.g., maintainability and reliability). Here, we describe *active maintainability improvement* and *re-engineering* as example treatment techniques. Note that techniques for architectural degeneration removal, such as architectural transformation (Krikhaar et al., 1999) and forward architectural repair (Tran and Holt, 1999) (see Section 2.3.3), also contribute to mitigation of architectural degeneration.

Active maintainability improvement

It is to improve a system's maintainability by repairing its affecting factors (e.g., over-large code files, complex interfaces, too many interactions among components, etc. (Pigoski, 1997, p. 282)) before they affect the system. In contrast, there is *passive* maintainability improvement which is to improve the maintainability after it has affected the software, e.g., *re-engineering* (as described below). Active maintainability improvement has been deployed in many organizations. For example, Microsoft Corporation usually allows about 20% of developers' time immediately after delivery to re-structure or rewrite certain weak parts of software products; this is, so called, "*the 20 percent tax*" (Cusumano and Selby, 1995, pp. 280-281). Their experience indicates that "if you don't pay the 20 percent tax, then you end up in a bad situation." (Cusumano and Selby, 1995, p. 281)

Similarly, Banker et al. (1993) argue that maintaining the structure of a software system often requires up to 25% of the total maintenance cost, and LaToza et al. (2006) report that the effort on making the code more evolvable is near to 15% of the total development effort. Empirical studies find that a more maintainable software system has 32.9% fewer failures (Rombach, 1987), requires

36% fewer defect fixes (Bandi et al., 2003), 16.7% fewer environment changes, and 28.9% fewer requirements changes (Rombach, 1987).

Re-engineering

Re-engineering is a process of reverse engineering a subject system, reconstituting it in a new form, and implementing this new form (Chikofsky and Cross, 1990). Jacobson and Lindström (1991) define that *re-engineering = reverse engineering + change deltas + forward engineering*. Re-engineering is used more often to replace aging software rather than to improve the maintainability (SWEBOK, 2004, ch. 6). Example re-engineering techniques are *transformation* at the architecture level (Grubb and Takang, 2003, p. 144) (see examples in Section 2.3.3) and *component re-engineering* at the design level (Booch, 2008). These techniques could reduce the architectural degeneration for the system.

Here, we focus on a special, code-level, re-engineering technique – *software refactoring* – a light-weight technique of performing changes on a code program to improve its internal structure without changing its external behavior (Fowler, 1999, p. 53). Each change does little but a sequence of changes can restructure the whole system (Mens and Tourw, 2004). However, van Gurp and Bosch (2002) suggest that current refactoring techniques cannot effectively improve the global maintainability, especially when there are complex structural problems which are widely dispersed over multiple components.

Except the adverse impact of architectural degeneration on system quality, there are other concerns which can affect the adaption of architectural degeneration treatment techniques in real system contexts, such as actual costs and benefit of the proposed treatment and related organizational strategies. For example, mitigating the architectural degeneration of a seriously-aging system (via re-engineering) could require even more costs than re-developing the whole system from scratch. As Sneed (1991) states, “In any case one must calculate the expected lifetime of the target system and compare the costs of re-engineering

with the costs of redevelopment starting from scratch. Re-engineering is often only considered a viable alternative, when the re-engineering effort is no more than 50% of the redevelopment effort."

2.3.5 Analysis of Existing Diagnosis Techniques

Section 2.3.3 has outlined three techniques for architectural degeneration diagnosis. We argue that these techniques cannot effectively diagnose architectural degeneration from the defect perspective.

First, deviations to an architecture do not always manifest as degeneration (Bhattacharya and Perry, 2007). Therefore, simply detecting and removing all deviations from an architecture is not the optimal way to diagnose and treat its degeneration (as discussed in Section 1.2).

Second, the defect-prone components (DPCs) of a system are the components containing the greatest number of defects in the system (see Section 2.3.3). However, the DPCs do not always coincide with the components which contribute most to the architectural degeneration (the *degeneration-critical* components). For example, a DPC could be considered degeneration-critical if it contains many more multiple-component defects (MCDs) than other components in the system.

Third, a fault architecture (von Mayrhauser et al., 2000) of a system can manifest a type of degeneration-critical components in the system – high-MCD-quantity components. There are other types of degeneration-critical components (e.g., high-MCD-density or complexity components) which cannot be exposed by the fault architecture. Similar limitations also hold for change architectures (Stringfellow and Andrews, 2002).

We thus created the DAD approach (as previewed in Section 1.3.2) to address these deficiencies. In particular, DAD defines MCD quantity and complexity metrics to identify the degeneration-critical components and fix relationships and

evaluate the architectural degeneration in a given system (see Section 1.4.2). See the details of DAD in Chapter 5.

Further, we were motivated to conduct the second case study to apply this DAD approach on a real system. Another important motivation of conducting this study is rooted in deficiency of scientific knowledge on architectural degeneration. For example, the literature did not identify the persistence of the contribution of a component or a fix relationship to the architectural degeneration, and the quantitative increase trend of the architectural degeneration over time. We thus investigated these aspects in the case study in order to address this deficiency. The findings can aid understanding the architectural degeneration phenomenon of the system. This case study is described in Chapter 6.

2.4 Software Defects

Recall Section 1.3 that Case Study 1 analyzes the defect (MCD) distribution, complexity and persistence. We note that software defects are usually investigated in relation to their aspects (e.g., where the defect was injected, when (which activity) the defect was found, where the defect was fixed, etc.) (Basili and Perricone, 1984). For example, IBM's Orthogonal Defect Classification (ODC) (Chillarege et al., 1992) model defines several aspects including *testing activity* (when the defect was found), *trigger* (the environment or condition that had to exist for defect detection), *impact* (the effect of the defect on the customer), *target* (where to fix the defect), *source* (the origin of the design/code which had the defect), etc. Aside from that, we are not aware of any particular defect logging standard; however, there are several tools for tracking defects, such as Rational Clearquest, Bugzilla, Bug Everywhere, and Fossil[9].

[9]IBM Rational Clearquest: `http://www-01.ibm.com/software/awdtools/clearquest/`; Bugzilla: `http://www.bugzilla.org/`; Bug Everywhere: `http://bugseverywhere.org/be/show/HomePage`; and Fossil: `http://www.fossil-scm.org/index.html/doc/tip/www/index.wiki` (last access in November 2010).

In literature, the defect analysis work that is closely related to this thesis research is about: defect distribution, correction effort, architectural defects, and defect-prone components (DPCs). Here, in the following four subsections (2.4.1–2.4.4), we review typical quantitative findings in these four aspects respectively.

2.4.1 Defect Distribution

Finding 1: Almost all software modules are defective before system delivery, but over half of the modules are defect-free after delivery.

Almost no software modules pass through inspection without defects before delivery (Shull et al., 2002). However, after delivery, defective modules are reduced to 25% (Andrews and Stringfellow, 2001), 26% (Basili and Perricone, 1984), 48% (Endres, 1975), and 50% (Andersson and Runeson, 2007) of modules in the system.

Finding 2: About 60%-80% of defects emanate from 20% of the modules.

Studies have shown that the 80-20 Pareto principle generally fits defect distributions by modules (Boehm and Basili, 2001) (Ostrand et al., 2005) and source files (Gittens et al., 2005). However, it could vary based on different characteristics of system contexts such as development processes used and system quality goals (Shull et al., 2002). For example, Ebert (2001) confirms that 20% of the modules can contain 40% through 80% of defects, depending on product line. Our earlier study (Li et al., 2009) finds that approx. 83% of defects emanate from 20% of the components in a large legacy system.

Finding 3: About 75%-95% of defects are single-component defects.

A defect is a single-component defect (SCD) if it is confined in only one component. Endres (1975) finds that 85% of defects are SCDs. This is supported by the studies described by Basili and Perricone (1984) where the finding stands at 89%, and by Weiss and Basili (1985) where the finding is in the range 76%-96%. Similarly, our earlier study (2009) finds that SCDs account for 88%-94% of all defects in the six releases of a large system.

2.4.2 Defect Correction Effort

Finding 4: Identifying and fixing a critical defect in a large system after delivery is about 100 times more expensive than that before delivery.

For large software systems, a defect effort increase of approximately 100:1 was often supported in literature and experiments (Shull et al., 2002). Hiemann (1974) finds that an effort increase of about 30:1 for defect slippage from code to field. O'Neill (see (Lindner and Tudahl, 1994)) reports an effort increase of about 13:1 for defect slippage from code to test and a further 9:1 increase for slippage from test to field; this means an effort increase of about 117:1 for defect slippage from code, via test, to field.

NASA research (see (Dabney et al., 2004)) shows that a defect introduced in requirements which escapes into design, code, test, integration, and operational phases, may consume the correction-cost factors of 5, 10, 50, 130, and 368, respectively (Boehm, 1976). Boehm (see (McGibbon, 1996)) argues that the 100:1 factor is about right for only critical defects in large systems; for non-critical defects, the factor could be reduced greatly to about 2:1.

2.4.3 Architectural Defects

Finding 5: About 5%-25% of defects are architectural defects.

Some defects in code originate in software architecting phase (Basili and Perricone, 1984), such as defects related to multiple components. We call these "architectural defects". In Yu's study (1998), such defects account for 18.5% of defects in a large system. In Leszak et al.'s study (2000), architectural defects account for approximately 5% of all defects, while consuming approximately 10% of the total correction effort. In Shin et al.'s study (2006), architectural defects are considered as anomalies that occur on the components or their interactions.

Historically, architectural defects have been associated with "*interface*" defects. For example, Endres (1975) defines an interface defect as one that requires

"changes" to multiple modules in order to fix it. With this definition, studies have reported that interface defects account for 5% (Endres, 1975), 11% (Basili and Perricone, 1984), 6%-15% (Basili and Shull, 2005), and 4%-24% (Weiss, 1979) of all defects. In contrast, Basili and Perricone (1984) adopt a stricter definition in that even if a maintainer needs to "examine" (not necessarily "change") more than one module in order to "understand" how to fix it then it is designated as an interface defect. With this definition, the profile of interface defects, however, is in stark contrast: 39% (Basili and Perricone, 1984), 66% (Perry and Evangelist, 1987), and 39.2%-57.5% (Nakajo and Kume, 1991). Note that the use of Basili-Perricone's definition requires subtle, "examination" (or soft), measures, which may not be captured widely in actual software projects.

2.4.4 Defect-Prone Components

Finding 6: Defect-prone components (DPCs) tend to persist across system development phases and releases.

We note from Section 2.3.3 that DPCs are defined as the top α (e.g., 20% (Li et al., 2009)) of components that contain the greatest number of defects in a system. Studies show that DPCs tend to remain defect-prone, from functional testing phases to system testing phases (Yu et al., 1988), from pre-release phases to field phases (Compton and Withrow, 1990), from development phases to testing phases (Andrews and Stringfellow, 2001), and from previous releases to successive releases (e.g., (Ohlsson et al., 1999), (Andrews and Stringfellow, 2001), (Stringfellow and Andrews, 2002), and (Andersson and Runeson, 2007)). However, there are studies which counteract the above findings, indicating that DPCs are not persistent across phases (Andrews and Stringfellow, 2001) or releases (Fenton and Ohlsson, 2000) (Ostrand and Weyuker, 2002). Our earlier study (2009) finds that in a large legacy software system, DPCs also contain the greatest number of MCDs, and over 70% of DPCs persist over multiple development releases.

2.4.5 Analysis of Existing Defect Research

The above overview of existing software defect research (see Findings 1–6) indicates that whereas general software defects have been extensively studied for more than three decades (e.g., (Endres, 1975), (Adams, 1984), and (Basili and Perricone, 1984) in the 1970's and 1980's, (Compton and Withrow, 1990) and (Nakajo and Kume, 1991) in the 1990's, and (Boehm and Basili, 2001), (Leszak et al., 2000), and (Ebert et al., 2005) in the 2000's), software architectures have not been studied as much from the viewpoint of defects during the same period. For example, how architectural defects differ from other types of defects, in terms of characteristics such as distribution, complexity, persistence, etc. In particular, we define architectural defects as multiple-component defects (MCDs) as they are related to potential crosscutting concerns (Eaddy et al., 2008) or architectural problems (von Mayrhauser et al., 2000) (as discussed in Section 1.3).

Intuitively, MCDs are considered more complex and costly to fix than other types of defects. Unfortunately, there are few such empirical findings from the literature (see Section 2.4.5). Our own, pilot, motivational investigations of five open-source software systems (a large application platform, a C++ application-development framework, Mozilla Bugzilla, Mozilla Network Security Suite, and OpenOffice spreadsheet) suggest that in comparison to fixing a non-MCD, fixing a MCD requires 2.3 times the changes, and adds/deletes 2.7 times SLOC (size) in 2.4 times the code files (span). This means that MCDs *are* more complex and costly (in change factors) to fix than non-MCDs.

Clearly, then, scientific studies on MCDs are important for their improved treatment and prevention, which are central to architectural evaluation, improvement and evolution (Rosso, 2006). This seals the initial intention to conduct further investigations on MCDs.

Chapter 3

MCDs and Architectural Degeneration

In this chapter, we introduce the concepts of multiple-component defects (MCDs) and architectural degeneration. In particular, we differentiate MCDs from interface defects and architectural defects (see Section 3.1), and describe relationships between architectural degeneration, and deviations, MCDs, and degeneration-critical components (see Section 3.2). This is fundamental to the focus of this thesis work – characterization and diagnosis of architectural degeneration from the defect perspective (as mentioned in Section 1.2).

3.1 Clarifications of MCDs

A MCD is defined as a defect that requires changes in more than one component in a software system in order to fix this defect (Li et al., 2009). We note from Section 2.4.3 that MCDs are related to interface defects (Endres, 1975) and architectural defects (Leszak et al., 2000). Here, we clarify the relationships among these three types of defects. Clearly separating MCDs from other types of defects can aid understanding and analyzing MCDs, which is important for the later described case studies (see Chapters 4 and 6) in the industrial context.

3.1.1 MCDs vs. Interface Defects

Interface defects are defined as defects that requires "changes" (Endres, 1975) (or "examinations" (Basili and Perricone, 1984)) to multiple modules in order to fix the defects (see Section 2.4.3). MCDs are similar to interface defects; both are related to interacting problems between system elements (components or modules). However, MCDs differ from interface defects because the term "component" defined here is not equivalent to the term "module" (e.g., as per definition in IEEE Std 610.12-1990). A *component* is a primary building block (or element) of a software system's architecture (Bass et al., 2003, p. 21) which encapsulates a set of closely related functionalities of this system. Whereas a *module* is a building block of a system's code base which contains a set of code files or programs (Basili and Perricone, 1984). Simply, components are the parts that make up a conceptual architecture but modules are the parts that make up a "physical" system (IEEE Std 610.12-1990). However, components and modules could overlap to some extent.

3.1.2 MCDs vs. Architectural Defects

Because architectures (or significant parts thereof) encompass multiple, interacting, components, architectural defects typically span more than one component. We thus associate architectural defects with MCDs. Also, due to the "multiple-component" nature, MCDs are related to potential crosscutting concerns (Aversano et al., 2009) and architectural problems (von Mayrhauser et al., 2000). However, architectural defects could be more than MCDs. For example, there could be architectural defects confined to only one component. Furthermore, there is no well-recognized definition of architectural defect in the literature. As we note from Section 2.4.3, in Basili and Perricone's study (1984), architectural defects are associated with interface defects, which account for 39% of all defects in a legacy system. In Leszak et al.'s study (2000), architectural defects refer to defects in code base originating in the early software architecting phase. They find that

architectural defects account for approximately 5% of all defects, while consuming approximately 10% of the total correction effort. And in Shin et al.'s study (2006), architectural defects are considered as anomalies that occur on components or connectors.

3.2 Understanding Architectural Degeneration

To help understand the concept of architectural degeneration, we describe, below, the potential relationships between architectural degeneration, and architectural deviations, MCDs, and system components.

3.2.1 Degeneration vs. Deviation

A well-known perspective to diagnose architectural degeneration is architectural deviation (Lindvall et al., 2002). It is generally recognized that the more deviations made to an architecture from its baseline, the more seriously this architecture has been degenerated (Hochstein and Lindvall, 2005; Bhattacharya and Perry, 2007; Lindvall and Muthig, 2008). In the studies such as (Lindvall et al., 2002) and (van Gurp and Bosch, 2002), deviations are measured from a *structural* perspective. Later, in (Bhattacharya and Perry, 2007), metrics are defined to measure deviations both structural and *functional* perspectives. The functional perspective measures how an architecture deviates from its baseline in terms of the services, functionalities, and data supported by architectural elements.

The concept of architectural degeneration emphasizes the adverse impact of "dirty" architectural change on system *quality*. This has two implications. First, deviations made to an architecture do not always indicate its degeneration. For an architecture, deviation analysis emphasizes its structural difference against its baseline, but degeneration analysis focuses on its adverse impact on system quality. Removing the structural deviations in an architecture may not effectively mitigate its degeneration (from quality perspective). Second, architectural degeneration

can be diagnosed from perspectives of quality attributes such as maintainability, reliability, and performance (see McCall's quality model (1977)). In this thesis research, we diagnose architectural degeneration from only the defect (mainly, MCD) perspective; see the further discussion in the next subsection.

3.2.2 Architectural Degeneration and MCDs

The relationship between architectural degeneration and MCDs is shown in Figure 3.1. We claim that architectural degeneration manifests itself through MCDs. The reasons are: (1) MCDs are related to potential crosscutting concerns (Eaddy et al., 2008; Aversano et al., 2009); and (2) fixing a MCD requires changes in more than one component, which is a sign of problems with the software architecture (von Mayrhauser et al., 2000; Stringfellow et al., 2006). Any change to the architecture of a software system could affect the potential crosscutting concerns or architectural problems and thus could change the profile of MCDs in the system. Therefore, characteristics of MCDs, such as their quantity and complexity, can quantitatively reflect architectural degeneration.

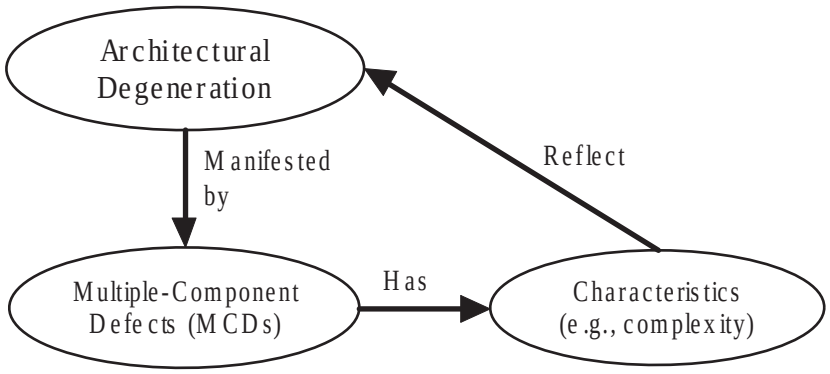


Figure 3.1: Relationship between architectural degeneration and MCDs.

The relationship between architectural degeneration and MCDs (see Figure 3.1) is fundamental to Case Study 1 – investigating the impact of architectural degeneration on defects by characterizing the MCDs in the system, and of the DAD approach – diagnosing architectural degeneration by measuring system components

and fix relationships with MCD quantity and complexity metrics. See Chapters 4 (Case Study 1) and 5 (the DAD approach) for details.

3.2.3 Degeneration-Critical Components

We note from Section 2.4.1 that the defect distribution by components is skewed: 60%-80% of defects emanate from 20% of the components (Boehm and Basili, 2001). Likewise, our earlier study (Li et al., 2009) shows that in a large system, 83% of MCDs emanate from 20% of the components and 75% of MCDs are involved in 10% of the fix relationships. These MCD-prone components and fix relationships should be considered *degeneration-critical*. This motivated us to identify and characterize these components and fix relationships to support effective architectural degeneration treatment. We thus created the DAD approach which can be used to achieve this purpose; see Chapter 5.

3.3 Key Points

We list several key points as mentioned in this chapter:

- MCDs are defects spanning more than one system component (not module) and are considered related to architectural problems (see Section 3.1).
- Architectural degeneration is more related to system *quality* concerns than structural *deviation* concerns (see Section 3.2.1).
- Architectural degeneration manifests itself through MCDs (see Figure 3.1).
- There could be a few *degeneration-critical* components and fix relationships in a specific software system (see Section 3.2.3).

Chapter 4

Case Study 1: MCD Analysis

This chapter describes an exploratory case study (Case Study 1)[1] which analyzed multiple-component defects (MCDs) in a large, commercial, legacy system. As per the relationship between MCDs and architectural degeneration (see Figure 3.1), this case study addresses question 1 posed in Section 1.2 – *What do defects indicate about architectural degeneration*? The answer to this question will allow us to characterize architectural degeneration from the defect perspective, and will further motivate us to create methods for diagnosing architectural degeneration.

In Section 4.1, we describe three specific research questions for Case Study 1. We then define or explain several key terms used in this case study in Section 4.2. In Section 4.3, we describe the case study design, which includes: the subject system and data, the data collection, clean-up, and analysis procedures, and the overall case study process. We then describe and interpret main findings in Section 4.4. In Section 4.5, we discuss threats to validity of the findings. In Section 4.6, we discuss implications of the study. Finally, we briefly recap the study in Section 4.7. Assessment of this study is described in Section 8.2. Challenges and lessons learnt from conducting this study are described in Chapter 9.

[1]This case study has been described in our earlier paper (Li et al., 2009) and its enhanced version accepted by the Empirical Software Engineering journal.

4.1 Research Questions

We note that MCDs, due to the "multiple-component" nature, are related to potential crosscutting concerns or architectural problems in the system (see Section 3.1.2). We also note that software architectures have not been studied much from the viewpoint of MCDs (see Section 2.4.5). Moreover, characterizing MCDs can reflect the negative impact of architectural degeneration on software defects (see Figure 3.1). Therefore, in order to answer question 1 (What do defects indicate about architectural degeneration?) posed in Section 1.2, Case Study 1 addresses three specific questions on a large legacy system (of size approximately 20 million SLOC and age over 20 years):

(i) *Does the 80:20 Pareto principle fit the MCD distribution?*

(ii) *To what extent are MCDs more complex than other types of defects?*

(iii) *To what extent are MCDs more persistent than other types of defects?*

Question (i) is concerned with the MCD distribution: whether approximately 80% of MCDs emanate from 20% of the components in the system. Due to the relationship between MCDs and architectural degeneration (see Figure 3.1), this question is related to the distribution of the impact of architectural degeneration on software defects over the components. This question is addressed in Section 4.4.1. Question (ii) is concerned with the MCD complexity. The number of components that are changed in order to fix a MCD is a measure of complexity (Endres, 1975). Finally, question (iii) is concerned with the MCD persistence across development phases or releases. MCDs that cross phase or release boundaries are clearly not getting fixed and are being flagged again in subsequent phases or releases, which are thus harder to fix then non-persistent defects. The greater the complexity and persistence (difficulty) of MCDs (compared against that for other types of defects), the greater the impact of architectural degeneration on software defects. These two questions, (ii) and (iii), are addressed in Sections 4.4.2

and 4.4.3 respectively. Note that severity of a defect can also be a factor in whether the defect is fixed or not in a given release. This paper performs such analysis.

Overall, answering the above three questions can build a quantitative profile of MCDs in terms of their (a) distribution by the components, (b) complexity in terms of the number of components changed, and (c) persistence across phases and releases. This profile reflects the impact of architectural degeneration on software defects in the subject system. It also adds to the current knowledge on MCDs in relation to architectural degeneration.

4.2 Terminology

A defect is "an incorrect step, process, or data definition in a computer program" (IEEE Std 610.12-1990). Simply, a defect is a flaw in code. We categorize defects into three types based on the phases where they are discovered: *development defects* (type D), *testing defects* (type T), and *field* (i.e., post-delivery) *defects* (type F); found in the phases of development (including design, coding, and unit test), testing (including functional and system test), and field (i.e., after release), respectively. The reason for categorizing defects in this manner is that they can be analyzed for cross-phase boundary transitions. For defect comparisons crossing releases, we add an additional type of defect, named *all defects* (type A), which refers to the sum of defect-types D, T, and F in a release.

In the defect dataset under investigation, two defects are related to each other as *parent* and *child*. For this purpose there are two reference fields: one ("parent reference") to link to the parent defect and other ("children reference") to link to the children defects. A defect may be too complex to fix all at once. Therefore, it is decomposed into multiple simpler defects for piecemeal correction and a defect record created for each child defect and *parent-children relationships* logged. These simpler defects (for the same complex defect) could span different phases (e.g., design, coding and testing) and even releases. The parent and children reference

fields and their relationships are described in the example defect records shown in Table 4.1; see Section 4.3.1 for details.

For example, initial analysis of a new defect (d_1) record indicates that d_1 is a complex defect located in component c_1, which requires fixes in three other components (c_2, c_3, and c_4). A normal routine is to create three new defect (d_2, d_3 and d_4) records indicating new defects d_2, d_3 and d_4 located in components c_2, c_3 and c_4, respectively. In this case, we say that d_1 (as the parent) is decomposed into d_2, d_3 and d_4 (as the three children of d_1), and there is a parent-child relationship between d_1 and each of (d_2, d_3 and d_4), which can be identified by investigating the four defects records (using their parent and children reference values). Also, see example defect records in Section 4.3.1.

A defect could have multiple root causes (Kulkarni, 2008) which are usually located in more than one component. This defect is termed a *multiple-component defect* (MCD). Fixing a MCD thus requires changes to more than one component. For the subject system, a MCD manifests itself as parent-children relationships in the defect-tracking database. For example, if a parent defect d_1 in c_1 has three children defects: d_2 in c_2, d_3 in c_3, and d_4 in c_3, then this indicates that fixing d_1 also requires fixing components c_1, c_2 and c_3. We say that fixing d_1 requires four changes in three components: one is in c_1 (to fix d_1), one is in c_2 (to fix d_2), and the remaining two are in c_3 (to fix d_3 and d_4). We thus also say that fixing d_1 requires three *accompanying* changes (i.e., excluding the change to fix d_1) in the two components: one is in c_2 (to fix d_2), and the remaining two are in c_3 (to fix d_3 and d_4). Number of accompanying changes required to fix a defect is a measure of complexity of this defect.

We identify MCDs based on the parent-children defect relationships; if a parent defect and its children defects are located in multiple components, all these defects are MCDs. An example of MCD identification is described in Section 4.3.1. Further, we say that there is a *fix relationship* among components if fixing a MCD

in one component requires changes to the other components in order to fix this MCD. Fix relationship is an implicit relationship over the architecture, which points to architectural problems (D'Ambros et al., 2009).

If a parent defect and its children defects are found in two different system phases or releases, we say that there is a cross-phase/release parent-children relationship. Such a relationship indicates that the parent defect is alive across phases/releases, indicating that it is more persistent than other defects that do not have such relationships. Further, in a sequence $A - B$, where A and B are two sets of defects found in time-ordered phases/releases (i.e., A is followed by B), we define that: (a) the *backward-ratio* (BR) refers to the number of defects in B that are children of defects in A divided by the size of B; and (b) the *forward-ratio* (FR) refers to the number of parent defects in A that have children defects in B divided by the size of A. They are formally described as below.

$$BR = \frac{\mid \{b \mid b \in B \wedge \exists\, a \in A : b \succ a\} \mid}{|B|} \tag{4.1}$$

$$FR = \frac{\mid \{a \mid a \in A \wedge \exists\, b \in B : a \prec b\} \mid}{|A|} \tag{4.2}$$

Here "$b \succ a$" (or "$a \prec b$") means defect b is a child of defect a, and $|\cdot|$ returns the number of members in a set. The definitions indicate that: (i) the higher the BR the more defects in B are children of defects in A, and (ii) the higher the FR the more defects in A are parents of defects in B. For example, suppose two time-ordered defect sets: $A = \{d_1, d_2, d_3, d_4\}$ and $B = \{d_5, d_6, d_7\}$, where $d_1 \prec d_5$ and $d_2 \prec d_6$. Then, we can derive that, from A to B: $BR = |\{d_5, d_6\}| \ / \ |\{d_5, d_6, d_7\}| = 2 \ / \ 3 \approx 0.67$, and $FR = |\{d_1, d_2\}| \ / \ |\{d_1, d_2, d_3, d_4\}| = 2 \ / \ 4 = 0.5$.

Further, we say that a defect persists across a phase or a release if it has children defects in the later phase or release. The persistence of a type of defect (e.g., MCD) is evaluated with the above BR and FR metrics. For example, the forward-ratio FR of a set of MCDs across two releases is the number of MCDs in

this set that have children defects in the latter release divided by the size of this set. Likewise for the calculation of the defect backward-ratio BR.

4.3 Case Study Design

This section describes the design of Case Study 1, which includes: description of the subject system and data, the data collection, clean-up, and analysis procedures, descriptive system statistics, and the overall case study process.

4.3.1 Description of the System and Data

We had an opportunity to study a large, commercial, legacy system[2] that contains approximately 20 million source lines of C, C++, Java^{TM}, and script code in the latest release, and has evolved over nine major releases and numerous minor releases and patches with fixes in more than 20 years.

The main data upon which this case study is based is the defect history (defect records, not enhancements) of six of the nine major releases (covering 17 years). In the system studied, MCDs are recorded in two situations: *active* and *passive*. The former situation is that a complex defect is actively decomposed (before being fixed) by developers into several simpler ones (for piecemeal fix) and these decomposed defects require fixes in at least one component other than the component where the parent defect is fixed (thereby making the parent defect and each of the children defects a multiple-component defect). The latter situation is that after a defect is "fixed" and the corresponding defect record is closed, it is found that this defect was not fixed properly and requires more fixes. One or more new defect records (called children defects) are thus created to log the remaining problems for the parent defect, meaning that fixing the parent defect requires fixing these new children defects.

[2]Due to the non-disclosure agreement with the sponsoring organisation, we are not permitted to disclose the application type of the system. In our humble opinion, however, this does not affect the understanding of any technical details presented in this paper.

Each defect record includes the following information: (1) the *component*, *phase* and *release* in which this defect was discovered (not injected), (2) the *summary* describing the main problem, (3) the *parent reference* and *children reference* indicating parent-children relationships (as defined in Section 4.2) between defects, and (4) the *severity* indicating the impact of this defect on the use of the system. The case study analyzed these fields of each defect record in the dataset. Two example defect records are shown in Table 4.1.

Table 4.1: Example defect records.

ID	Component	Release _Phase	Parent Reference	Children Reference	Severity	Summary
0010	C1	r1_design	—	0011, ...	2	Data share-space crash
0011	C2	r1_code	0010	—	3	Garbage collection problem

As shown in Table 4.1, the first defect record indicates that the defect 0010 (see column "ID") was discovered in component C1 (see column "Component") when designing release 1 (r1_design – see column "Release_Phase"); its severity level is 2 (considered major – see column "Severity"); and it is a data share-space crash (see column "Summary"). Its "Children Reference" value – "0011, ..." – indicates that the defect 0011 is one of its children defects. The second defect record indicates that the defect 0011 was discovered in component C2 when coding release 1 (r1_code). Its "Parent Reference" value – 0010 – shows the ID of its parent defect (i.e., the first defect shown in the table).

4.3.2 Data Collection, Clean-up, and Analysis Procedures

For the subject system, the defects were recorded when they were discovered in development, testing, and field phases. When a defect is found by a developer or user, it is reported to an appropriate analyst in the development team. A

new record is thus opened to log this defect in the defect-tracking database. The analyst also checks whether it is a rediscovery of a previous defect. If it is a rediscovery, the analyst typically *refers* it to the previous defect. Here, a defect rediscovery indicates that: (a) the original defect was not fixed properly; and (b) the rediscovered defect still requires changes to code in order to fix this defect as well as the original one. Note that this "rediscovery" meaning is different from that in some other contexts where defect rediscoveries do not require further code changes (Kulkarni, 2008). In this case study, this special relationship between the original defect and its "rediscoveries" was defined as parent-children relationship; see Section 4.2. That is, fixing a parent defect requires accompanying changes to fix its children defects (i.e., its "rediscoveries"). The related data quality issues are described in Section 4.5.1.

We wrote programming scripts to query the defect-tracking database and thus gathered the defect records (from the database) for each system release under investigation. We then cleaned up the defect records, mainly removing defects that are still unresolved and removing incomplete or incorrect defect records (e.g., defect records having empty component values or incorrect reference values). For example, we removed the defects which are "invalid" or still "open" in the system. This was carried out based on the *state* field of defect records. In particular, defects records which are not "closed", "integrated", "delivered" and "validated" are excluded from the dataset.

After that, we identified the *development*, *testing*, and *field* phases for the defect records. This step was carried out based on the *phase* field of the defect records. Therefore, the development, testing and field defects were identified accordingly. We also identified the parent-children defect relationships based on the "parent reference" and "children reference" values of defect records. See an example in Table 4.1 that defect 0010 is the parent of defect 0011.

Finally, we used scripts to analyze the defect records. Some statistical methods

(e.g., Pearson correlation coefficient[3] or Pearson-value) and Wilcoxon signed-rank test[4]) were also used for the analysis. Descriptive statistics of the subject system are given below.

4.3.3 Descriptive System Statistics

Table 4.2 depicts the number of components (see column "#Components"), the proportion of defects (see column "%Defects"), and the time range (number of years) covered by the defect history (see column "#Years Covered"), in each of the six releases of the system. Rows "Mean" and "StDev" show the average and standard deviations values across the six releases. The actual release numbers are represented by r1 through r6. For example, release 1 (see row "r1") has 183 components, the defects from this release account for 16% of the all defects, and the dataset covers the defect data recorded in 9.8 years.

Table 4.2: Basic profile of the subject system of Case Study 1.

Release	#Components	%Defects	#Years Covered
r1	183	16%	9.8
r2	206	19%	9
r3	266	14%	9.6
r4	320	22%	9
r5	316	14%	4.3
r6	326	16%	7.5
Mean	270	17%	8.2
StDev	62.3	3%	2.1

The table indicates that the system size (by #Components) increased by about 80% from release 1 to release 6 (across 17 years), the average increase ratio is 12.5% per release. This table also indicates that the defects under investigation are

[3]Pearson correlation coefficient is a correlation measure between two data arrays and its range is from -1 to +1. A value of 1 indicates a positive linear correlation, a value of -1 indicates a negative linear correlation; and a value of 0 or near 0 indicates "no" correlation.

[4]The Wilcoxon signed-rank test is "a non-parametric statistical hypothesis test for the case of two related samples or repeated measures on a single sample" (see `http://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test` (last access in June 2010)). The population for test does need to be normally distributed.

relatively evenly distributed in the six releases (an average of 17% per release); the corresponding standard deviation value is small (3% – row "StDev"). Moreover, this table shows that the defect dataset for each release covers the defect data in an average of 8.2 years (see column "#Years Covered"), indicating the legacy nature of the dataset under investigation. The relevance of these findings to Case Study 1 is described later in Section 4.4.

4.3.4 Case Study Process

Centered on the three questions, (i)–(iii), posed in Section 4.1, Case Study 1 was undertaken in seven key steps:

Step 1: we cleaned up the defect records for the subject system. This data cleaning process was carried out manually for the most part. Some scripts are used to speed up this process. See Section 4.3.2 for the details.

Step 2: we identified parent-children relationships based on the "parent reference" and "children reference" values of defect records, see Section 4.3.2.

Step 3: we identified MCDs with the help of parent-children defect relationships. This step is discussed in Sections 4.2 and 4.4.1.

Step 4: we identified MCDs that are spread across development phases or releases. We examined six main releases of the subject system. The defects data were readily categorized into appropriate phases of each of the six releases. This step is discussed in Section 4.4.3.

Step 5: we analyzed MCD distributions by components and fix relationships in the system, see Section 4.4.1 (addressing question (i)).

Step 6: we compared MCDs against non-MCDs in terms of their complexity (number of changes), see Section 4.4.2 (addressing question (ii)).

Step 7: we compared MCDs against non-MCDs in terms of their persistence across phases and releases, see Section 4.4.3 (addressing question (iii)).

This case study was designed to quantitatively examine the distribution, complexity and persistence over time of MCDs compared against non-MCDs. In particular, the complexity of a defect is measured by the number of accompanying changes required to fix it. The more accompanying changes required for a defect, the more complex this defect is. The persistence over time of a defect is measured by the number of system development phases (and releases) that this defect spans. The more phases or releases a defect spans, the more persistent this defect is. The quantitative findings of MCDs in terms of their distribution, complexity and persistence are presented and interpreted in the next section.

4.4 Analysis of Data, Results, Interpretation, and Comparisons

The findings of Case Study 1 addressing questions (i)–(iii) posed in Section 4.1 are related to the distribution, complexity, and persistence over time of MCDs in the subject system. We describe and interpret the findings using these three aspects. We also compare the findings against related work in the literature.

We note that data interpretation should be based on such principles as: (a) corresponding with the (quantitative or qualitative) findings (Bracey, 2006, p. 32); (b) incorporating the context variables (Basili et al., 2006, p. 68); and (c) reducing subjective judgement (Münch, 2006). Following this, here, we interpret the findings of Case Study 1 (as below) with relation to architectural degeneration in the industrial context.

4.4.1 MCD Distribution (Question (i))

Recall from Section 2.4.1 that defect distribution is usually skewed. Here, Figure 4.1 illustrates the average distributions of all defects and MCDs by components and fix relationships across the six releases of the subject system. There are three

curves showed in this figure: the "Defect:Component" curve represents the distribution of defects by components; the "MCD:Component" curve represents the distribution of MCDs by components; and the "MCD:Fix Relationship" curve represents the distribution of MCDs by fix relationships[5].

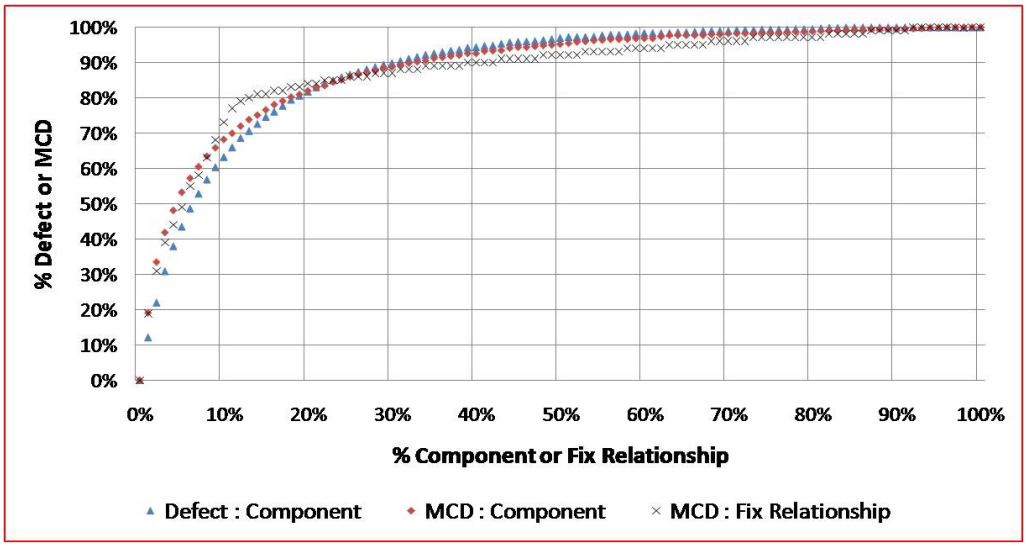


Figure 4.1: Distributions of MCDs by components and fix relationships.

This figure indicates that the Pareto principle (i.e., the 80:20 rule) largely fits the distributions of defects and MCDs by components or fix relationships in the system: over 80% of all defects and MCDs emanate from (i.e., are concentrated in) 20% of the components, and nearly 75% of MCDs involve 10% of the fix relationships. This answered question (i) posed in Section 4.1 – Does the 80:20 Pareto principle fit the MCD distribution?

Further, the MCD distribution reflects the distribution of the impact of architectural degeneration on software defects (recall Figure 3.1). That is, from the MCD quantity perspective, there are 20% of the components which contribute most to the degeneration of the system's architecture. These 20% of components

[5]Recall Section 4.2 that there is a fix relationship between two components if there is at least one MCD pertaining to both these two components. In this case, we also say that that MCD pertains to this fix relationship. Here, the distribution of MCDs by fix relationships is to count the aggregate of MCDs pertaining to the (binary) fix relationships in the system.

should be thus considered *degeneration-critical*, indicating the most problematic areas in the system with respect to architectural degeneration. Identifying these components is thus important for system quality and cost concerns. Therefore, this motivated us to develop an effective solution to identify degeneration-critical components for a given system. Such a solution is the DAD approach (illustrated in Chapter 5). The related issue about degeneration-critical component identification is also investigated in Case Study 2 (see Section 6.3.1).

We note that the Pareto-shaped defect distribution has been studied by several other researchers such as Boehm and Basili (2001), and Ostrand et al. (2005); they find that about 80% of defects emanate from 20% of the components. We also note from Gittens et al.'s study (2005) on the same system that 60% of the defects found in development (type D) and testing (type T) and near 80% of the defects found in field (type F) pertain to only 20% of the system's code. Similar findings are shown in our earlier study (2009): about 80% of the type D and T MCDs emanate from 20% of the components. The findings shown above added that this Pareto principle largely fits the MCD distribution by fix relationships.

4.4.2 MCD Complexity (Question (ii))

If a defect requires one or more accompanying changes (as per definition in Section 4.2), we can assume that this defect is relatively more complex to fix, in some respect, than a defect that does not require any accompanying change. In the study, we find the following profile for defects that require one or more accompanying changes over six releases: 10% defects of type A (all), 8% defects of type D (development), 9% defects of type T (testing), and 9% defects of type F (field)[6]. Due to the "multiple-component" nature, MCDs are generally considered more complex to fix than other types of defects. The complexity of a MCD can be thus

[6]Here, we see that the percentage 10% for type A defects, which is larger than that for types D, T, and F defects. The reason follows. Type A is a set of types D, T and F; and there exist defects requiring accompanying changes across phases (development, testing and field). These defects are considered only when the percentage for type A – 10% – is counted.

measured by the number of accompanying changes required to fix this MCD.

Table 4.3 shows the proportions of MCDs (see column "%MCDs") and their accompanying changes (see column "%Accompanying Changes for MCDs") in the six releases. For example, MCDs account for 7% of all defects but these defects consume 50% of all accompanying changes in release 1 (see row "r1").

Table 4.3: Proportions of MCDs and their accompanying changes.

Release	%MCDs	%Accompanying Changes for MCDs
r1	7%	50%
r2	11%	60%
r3	5%	45%
r4	8%	57%
r5	5%	46%
r6	10%	55%
Mean	8%	52%
StDev	3%	6%

Table 4.3 shows that on average over the six releases, MCDs account for approximately 8% of the defects (with standard deviation 3%), but consume approximately 52% of the all accompanying changes (thus 52% of the all children defects) (with standard deviation 6%). This means that, on average, a MCD consumes many more accompanying changes than a non-MCD, indicating the increased complexity in attempting to fix MCDs in the subject system.

Table 4.3 also describes a near-linear, positive, correlation between the proportion of MCDs and that of accompanying changes for MCDs; the corresponding Pearson-value is 0.93 (over the six releases)[7]. This indicates that the greater the MCD proportion of the system, the greater the proportion of accompanying changes required for fixing the MCDs in the system.

Considering Tables 4.2 and 4.3 together, we find that there is "no" correlation between the system size (see column "#Components" in Table 4.2) and the MCD

[7]The critical value for 6 data pairs (because there are six releases) based Pearson value is 0.622 (see `http://www.une.edu.au/WebStat/unit_materials/c6_common_statistical_tests/test_signif_pearson.html` (last access in November 2010)). Here, obviously, the 0.93 Pearson-value indicates a significant correlation.

proportion (see column "%MCDs" in Table 4.3); the corresponding Pearson-value is -0.15[8]. This indicates that the increase in system size does not substantially affect the MCD proportion of the system.

In particular, Table 4.4 depicts the average numbers of accompanying changes for MCDs and all defects in development phases, testing phases and all phases. Each cell in the "MCDs" and "All Defects" columns is a Mean/Standard deviation (StDev) pair. For example, value "2.10/0.10" in column "MCDs" means that the average number of accompanying changes for a MCD is 2.10 with standard deviation 0.10. Column "MCDs:Non-MCDs" shows the average ratio of the number of accompanying changes for MCDs to that for non-MCDs. The last column "p-value" shows the significance level – the p value[9] – for the difference between the values for MCDs and "All Defects". The p value "< 0.001" (for the testing of defects of types: All, Development and Testing) means that there are statistically significant differences between all defects and MCDs with respect to the quantity of accompanying changes required.

Table 4.4: Accompanying changes required for MCDs.

Defect Type	MCDs	All Defects	MCDs:Non-MCDs	p-value
All	2.10/0.10	0.10/0.03	26.3	< 0.001
Development	2.10/0.17	0.08/0.02	30	< 0.001
Testing	2.15/0.07	0.09/0.03	27	< 0.001

Table 4.4 shows that the average number of accompanying changes for MCDs is about 2.1 (see column "MCDs"), which is 26-30 times (see column "MCDs:Non-MCDs") as much as that for non-MCDs. This also means that a MCD has an average of 2.1 children defects, but a general defect has an average of only 0.1

[8]Because the critical value for 6 data pairs based Pearson value is 0.622 (see the previous footnote), here, the -0.15 Pearson-value thus indicates an insignificant correlation.

[9]In order to determine if there is any significant difference between all defects and MCDs with respect to the accompanying changes, we conducted a Wilcoxon paired signed rank test. Each test here is based on 6 data pairs (because that there are six releases). The p value is "the probability of obtaining a test statistic at least as extreme as the one that was actually observed, assuming that the null hypothesis is true." (see `http://en.wikipedia.org/wiki/P-value` (last access in November 2010))

children defect. This further indicates that, on average, a MCD requires to be decomposed into nearly 3 times as much as the simpler defects for piecemeal correction than a non-MCD (i.e., fixing a MCD requires nearly 3 times changes (based on components) as much as that for fixing a non-MCD.). This confirms the early finding (of Table 4.3) that MCDs are more complex to fix than non-MCDs, which thus answered question (ii) posed in Section 4.1 – To what extent are MCDs more complex than other types of defects?

We note that 20% of defects consume 60%-80% of total correction effort (Ebert et al., 2005, ch. 9). Usually, these 20% of defects are more complex to fix than the remaining defects. We also note that, in (Eick et al., 2001), the authors use *change factors* (including *size* (number of added/deleted SLOC), *span* (number of components spanned), *time interval* (calendar time required), etc.) to measure a change's effort. Fixing a defect usually requires changes in a code base; therefore, these change factors can be used to measure the defect – its "complexity". As shown above, we measured defect (MCD) complexity with the number of accompanying changes. This can complement the previous work (e.g., (Eick et al., 2001) and (Ebert et al., 2005, ch. 9)).

Furthermore, considering the relationship between MCDs and architectural degeneration (see Figure 3.1), we can infer that the architectural degeneration increased the (fix) complexity of defects in the system. This is mainly due to the profile of over 50% of accompanying changes for MCDs (see Table 4.3) and the "26-30" factor of the MCD complexity over that for non-MCDs (see Table 4.4).

4.4.3 MCD Persistence (Question (iii))

We note from Section 4.2 that a defect is persistent if it crosses development phase or release boundaries. We thus say that MCDs are more persistent than non-MCDs if it can be shown that the percentage of persistent MCDs is greater than that for general defects. In this study, the MCD persistence is examined with

the *forward-* and *backward-ratios* (FR and BR – as per definitions in Section 4.2) of MCDs crossing phases and releases.

Table 4.5 shows the average BR and FR values of MCDs and of all defects crossing a phase or a release. Note that r_i and r_{i+1} represent the previous and next releases, respectively. Similar to Table 4.4, the row "p-value"[10] show the p values which indicate the significance level of the difference between MCDs and "All Defects". For example, "0.12/0.05" (in column "D in $r_i \rightarrow T$ in r_i") indicates that the average BR value of all defects from development (D) to testing (T) phases is 0.12 (with standard deviation 0.05). This means that only 12% of the defects found in testing (type T) are forwarded from development (type D) defects. However, there are 83% of testing (type T) MCDs forwarded from development (type D) MCDs (see value "0.83/0.04").

Table 4.5: Backward and forward-ratios of MCDs.

		D in $r_i \rightarrow T$ in r_i	A in $r_i \rightarrow A$ in r_{i+1}
BR	All Defects	0.12/0.05	0.03/0.01
	MCDs	0.83/0.04	0.13/0.10
	MCDs:Non-MCDs	8.3	6.5
	p-value	< 0.001	< 0.001
FR	All Defects	0.10/0.06	0.03/0.03
	MCDs	0.71/0.12	0.15/0.10
	MCDs:Non-MCDs	7.5	6.0
	p-value	< 0.001	< 0.001

By comparing columns "D in $r_i \rightarrow T$ in r_i" and "A in $r_i \rightarrow A$ in r_{i+1}" in Table 4.5, we can infer that the FR and BR values of all defects and MCDs across releases are much smaller than those across phases. However, this table also shows that the cross-phase (or release) FR and BR values of MCDs are 6.0-8.5 times (see rows "MCDs:Non-MCDs") as much as the values for non-MCDs. It

[10]Note that for across-phase comparison (from development to testing), there are 6 data pairs for the testing; but for cross-release comparison (e.g., from release i to release $i+1$; $i = 1..5$), there are only 5 data pairs for the testing. The p values "< 0.001" shown in Table 4.4 mean that there are significant differences between all defects and MCDs with respect to the forward-ratios and backward-ratios across phases and releases.

constitutes strong evidence in support for: MCDs are more persistent than other types of defects across a phase and a release. This answered question (iii) posed in Section 4.1 – To what extent are MCDs more persistent than other types of defects? This further implies that MCDs are harder to fix than non-MCDs and are more likely to escape software development scrutiny.

Note that severity of a defect can also be a factor in whether the defect is fixed or not in a given release. In particular, our data shows that, on average over the six releases, 70% of MCDs and 62% of non-MCDs are of high severity. We further conducted a Wilcoxon paired signed rank test and found that there is no significant difference (at the 0.95 level) between these two percentages of high-severity defects in MCDs and non-MCDs ($0.05 < p < 0.1$) over the six releases. This indicates that the severity does not significantly affect the persistence of MCDs (in comparison against that for non-MCDs). This issue is further discussed in Section 4.5.3.

For the architectural aspect, we can infer that the architecture (specifically the change-coupling problems among components) inflicts more adverse impact on the defect correction. For example, the above findings (see Section 4.4.2) indicate that architectural defects (MCDs) are more complex (by number of accompanying changes) to fix than other types of defects. We can further infer (based on quantitative support) that the architectural degeneration increases the average fix difficulty of defects in the system. This is mainly due to the "6.0-8.5" factor of the MCD persistence over that for non-MCDs (see Table 4.5).

We note from Section 2.4.2 that the longer a defect exists in the system, the much more costly it is to fix this defect. We also note that some defects are rediscovered across development phases and releases in the system (Kulkarni, 2008). As described above, defects that are "rediscovered" across phases and releases are considered persistent and over half of such defects span multiple components (i.e., MCDs – see Table 4.5). This finding adds to the knowledge on defect correction; also see their implications in Section 4.6.

4.4.4 Summary of Findings

There are three key aspects of the findings of this case study: the distribution, complexity and persistence of MCDs. These findings are summarized below.

- The Pareto principle fits the MCD distributions: over 80% of the MCDs emanate from 20% of the components and nearly 75% of the MCDs involve 10% of the fix relationships (see Figure 4.1). This indicates that the MCDs are highly concentrated in a few components in the system.
- The MCDs account for approximately 8% of the defects, but consume approximately 52% of the accompanying changes in the system (see row "Mean" in Table 4.3). In detail, on average, fixing a MCD requires nearly 3 times changes (based on components) as much as that for fixing a non-MCD, indicating the increased complexity in fixing MCDs.
- The proportion of MCDs crossing over from one phase or release to the next is 6.0-8.5 times as much as that for non-MCDs (see row "MCDs: Non-MCDs" in Table 4.5). This indicates that MCDs are more persistent than other defects across phases and releases.

Overall, the above findings are new; and there are no previously published studies similar to this case study. These finding add to the scientific knowledge on architectural defects (MCDs). Based on the relationship between MCDs and architectural degeneration (see Figure 3.1), we claim that: (1) there are a few (20% of) components that contributed most (over 80%) to the architectural degeneration from the MCD quantity perspective; and (2) the architectural degeneration increases the average fix complexity and difficulty of defects in the system. This thus sealed our attention on the DAD approach (see Chapter 5), which can aid mitigating the adverse impact of architectural degeneration on defects (e.g., reducing the fix complexity and difficulty of MCDs).

4.5 Threats to Validity

This section discusses the main threats to validity of the case study findings. We classify these threats into three groups: data reliability, and external and construct validity; see the following three subsections.

4.5.1 Data Reliability

There is a concern with the accuracy of the defect dataset for the legacy system involved in the investigation (see Section 4.3.1). By examining a random sample of 120 (20 from each of the six releases) defect records, we determined that while about 80% of the (parent and children) reference values straightforwardly described the parent-children relationships, about 20% were also used for other purposes, such as representation of separate defects with similar summaries, or introduction of new defects induced by defect fixes. While it was possible to exclude the 20% from the random sample (i.e., the 120 defect recrods), it is clearly not pragmatic to remove such anomalous records from the large defect dataset. Therefore, the identification of parent-children relationships between defects could be affected to some extent in this case study. This introduces some degree of threat to validity of the results and calls for future research to develop techniques to automatically filter out noise from the dataset.

There is also a concern with the completeness of the defect dataset. For example, it is possible that some defects were fixed but were not recorded in the defect-tracking database. Although the developers conveyed their high degree of confidence in the completeness[11] of the defect history representing the 17 years we analyzed (e.g., an average of 8.2 years per release; see Table 4.2), there is no simple way to independently confirm this.

[11] A rigorous routine for the subject system is to first record a defect in the defect-tracking database, then fix this defect, and finally update the fields of the defect record. Thus, it can be said that all the defects fixed in the system have first been recorded in the defect-tracking database. This leads to the statement about the confidence in the completeness of the dataset.

4.5.2 External Validity

External validity is concerned with the generalization of study findings in other contexts (Creswell, 2002). First, let us examine the system under study. It is a large legacy system. While, there are other large legacy systems from diverse application domains, there is no obvious reason to single out the case study system as particular from the point of view of defect types and spread in the system. For example, one would expect that MCDs would also exist in other large systems. However, the way MCDs are recorded in the logs can be specific to the organization (e.g., people culture, habits and process), infrastructure, defect logging tools, and others. This implies that the detailed steps in identifying MCDs (based on the parent-children relationships among defects – Section 4.2) are not obviously generalizable to other contexts.

We note that in some open-source software systems (as mentioned in Section 3.1), the defect histories only partially satisfy the requirements for reproducing this case study. For example, whereas MCDs could be identified from the defect records and change logs of these systems, the MCD-persistence analysis was not possible because the "parent reference" and "children reference" values (see Section 4.3.1) are not recorded in the defect records.

4.5.3 Conclusion Validity

Conclusion validity is concerned the extent to which the conclusions made in the study are reasonable Wohlin et al. (2000). We investigated the complexity and persistence of MCDs by comparing them against other types of defects (see Sections 4.4.2 and 4.4.3). The definition of complexity of a defect is based on the number of accompanying changes required to fix this defect (see Section 4.2). The definition of persistence of a defect is based on the extent to which this defect is leaked into successive phases or releases. The quantitative findings of the MCD complexity and persistence are given in Sections 4.4.2 and 4.4.3.

A possible threat to a conclusion of this study is related to the definition of defect complexity. That is, complexity is based on the changes at the level of components, not number of lines of code added, modified or deleted, or in terms of the amount of time spent, or logical complexity of code, etc. These factors were not considered in this case study because of lack of such detailed data.

Another possible threat to a conclusion of this study is related to the potential impact of defect severity on defect persistence across development phases and releases (see Section 4.4.3). We note from Section 4.4.3 that there is no significant difference ($0.05 < p < 0.1$) between the percentages of high-severity defects in MCDs and non-MCDs (70% vs. 62%); the threat is thus low.

4.6 Implications

In this section, we discuss the implications of the case study findings (see Section 4.4) from the points of views of software maintenance, architectural degeneration treatment, and architectural methods and tools.

4.6.1 Software Maintenance

Because MCDs are highly concentrated in a few components in the system (see Figure 4.1), those MCD-prone components should be separated from other components for effectively discovering and fixing MCDs in the system. Also, because MCDs are more complex than other kinds of defects (see Table 4.4), they require more effort to fix. Moreover, because MCDs are more persistent across phases and releases than other kinds of defects (see Table 4.5), they require more testing effort in order to ensure system qualities such as reliability and user satisfaction.

Therefore, separating MCDs from other defects, during maintenance, can help focus attention on these hard-to-fix defects. Second, it is more difficult to fix a MCD in the system phase or release where this MCD occurred. Third, more

regression-testing efforts are required in order to increase the probability of fixing MCDs in the current phase or release; otherwise, the reliability and maintenance cost of the subsequent system phase or release will be affected. Further, the complexity and persistence of MCDs also reflect the importance of the MCD-prone components for improving defect correction and prevention.

Further, we note that there are a few fix relationships in the system which are involved in a majority of the MCDs (see Figure 4.1). These special fix relationships can help refine strategies of component re-engineering. For example, MCD-prone components having one or more such fix relationships should be re-engineered in higher priority than other components having no such fix relationships. We note that defect classification schemes such as IBM's Orthogonal Defect Classification (ODC) (Chillarege et al., 1992) and Hewlett Packard's Defect Origins, Types and Modes (Grady, 1992, pp. 122-137) do not involve the view of point of multiple-component nature of defects. We also note that SWEBOK[12], IEEE Std 1219-1998[13], and other such literature do not mention how to treat MCDs.

4.6.2 Architectural Degeneration Treatment

Considering the relationship between MCDs and architectural degeneration (see Figure 3.1), the Pareto-shaped MCD distribution (see Figure 4.1) indicates that architectural degeneration is mainly caused by a few (20% of), MCD-prone, components in the system. These MCD-prone components are degeneration-critical from the MCD quantity perspective. Therefore, treating the architectural degeneration should focus only on these components. For example, these components should be re-engineered in priority.

Moreover, the MCD complexity and persistence profile (see Tables 4.3–4.5) re-

[12]SWEBOK stands for the Software Engineering Body of Knowledge (by IEEE); see `www.swebok.org` (last access in November 2010).

[13]IEEE Std 1219-1998 refers to as IEEE's Standard for Software Maintenance; see `http://standards.ieee.org/reading/ieee/std_public/description/se/1219-1998_desc.html` (last access in November 2010).

flects the adverse impact of architectural degeneration on software defects, which thus increases the necessity and significance of architectural degeneration treatment in practice. Typically, treating architectural degeneration should decrease the MCD complexity and persistence. The MCD complexity could be decreased by component re-engineering; however, decreasing the MCD persistence must require defect correction improvement, e.g., enhancing regression testing or post-change code review to avoid leaking defects into next development phases or releases.

4.6.3 Architectural Methods and Tools

It is worth mentioning that while tools for software architecting and maintenance have made significant progress over the last decade (e.g., in the areas of metrics, maintenance history analysis, change and impact analysis, etc.), the work described in this paper is yet another trigger for making further progress in tool-technology. For examples, the ROSE prototype (Zimmermann et al., 2004) can determine the change-coupling relationships at different system-granularity levels, and the SACPT tool (Abdelmoez et al., 2004) can measure the architecture-level change propagation among components. Such tools could benefit from our work on fix relationships.

Both ROSE and SACPT, by integrating the methods of identifying fix relationships, can support detection of inter-connection problems on the architectural level and identification of MCDs on the code level. For example, ROSE could answer questions such as: "Which system elements have fix relationships with the elements under editing?" and "Do these fix relationships cause many MCDs in code?" SACPT could answer questions such as: "Which architectural elements are most likely to contain the fixes that couple the fixes made in a specific architectural element?" and "How many of these fix-coupled architectural elements, given an architectural element?" These questions cannot be easily answered with the ROSE and SACPT tools in the current situation.

4.7 Recap of Case Study 1

Previous research has shown that architectural defects (typically those that span over multiple components and their interactions) can consume twice as much effort in fixing them as that for other defects (Leszak et al., 2000). Thus, any new understanding about these resource-sinking defects can help fix these defects. MCDs are such defects. Due to their "multiple-component" nature, MCDs are related to potential crosscutting concerns (Aversano et al., 2009) and architectural problems (von Mayrhauser et al., 2000).

In Case Study 1 we answered question 1 posed in Section 1.2 (What do defects indicate about architectural degeneration?) in a large legacy system. In particular, this case study answered these three questions (see Section 4.1): (1) Does the 80:20 Pareto principle fit the MCD distribution? (2) To what extent are MCDs more complex than other types of defects? and (3) To what extent are MCDs more persistent than other types of defects?

This case study investigates the defect data of a large system, representing six releases over 17 years (see Section 4.3.1). The qualitative findings of this case study relevant to the questions are: (i) MCDs are highly concentrated in a few components in the system (see Figure 4.1); (ii) MCDs are more complex to fix then non-MCDs (see Tables 4.3 and 4.4); and lastly (iii) MCDs are more persistent than other defects across phases and releases (see Table 4.5). A succinct summary of the quantitative findings can be seen in Section 4.4.4.

According to the relationship between MCDs and architectural degeneration (see Figure 3.1), we can infer from the above MCD profile that: (1) there are a few components and fix relationships that contribute most to the architectural degeneration; and (2) the architectural degeneration increases the average fix complexity and difficulty of defects in the system. This addresses question 1 posed in Section 1.2 – What do defects indicate about architectural degeneration? It also sealed our intention to create an approach for architectural degeneration diagnosis

– the DAD approach (see Chapter 5).

Overall, the above findings are new; and there are no previously published studies similar to this case study. These findings add to the scientific knowledge on architectural defects (MCDs) and degeneration. They also have implications for software maintenance, architectural degeneration treatment, and system quality improvement. For example, because MCDs are more difficult to fix than other types of defects, separating MCDs from other defects, during maintenance, can help focus attention on these hard-to-fix defects. Also, those MCD-prone components should be separated from other components for the purpose of effectively discovering and fixing MCDs as well as architectural degeneration treatment in the system. See Section 4.6 for the details of the implications.

While these findings are new, we should not overlook that these results are from only one, albeit significant, case study. This case study has its own idiosyncratic threats, e.g., specification of raw data, MCD identification, and MCD-persistence analysis (see Section 4.5). There is thus a need to conduct replicated studies involving other systems in order to build a body of knowledge on architectural defects (e.g., MCDs). Later Section 8.2 discusses other related limitations to this case study, and Chapter 9 describes the challenges involved in and lessons learnt from conducting this study in an industrial context.

Chapter 5

Diagnosing Architectural Degeneration (DAD)

In this chapter, we describe the approach to **D**iagnosing **A**rchitectural **D**egeneration from the defect perspective. This approach addresses question 2 posed in Section 1.2 – *How can architectural degeneration be diagnosed from the defect perspective*? In Section 5.1, we describe two plausible symptoms for the diagnosis. In Section 5.2, we illustrate a conceptual DAD framework, mainly the goals of this approach and its procedural steps to achieve the goals. In Section 5.3, we describe a DAD prototype tool, including its main features, data input, processing mechanisms, and output. After that, we compare DAD against related techniques in Section 5.4. Finally, we give a summary of this approach in Section 5.5. A case study that validated DAD on a real system is presented in Chapter 6. Assessment of DAD is discussed in Section 8.3.

5.1 Symptoms for Diagnosis

We note from Figure 3.1 that, from the defect perspective, architectural degeneration manifests itself by multiple-component defects (MCDs). We also note that Case Study 1 examines the distribution, complexity, and persistence of MCDs in

the subject system (see Sections 4.4.1 and 4.4.2) for the purpose of characterizing the architectural degeneration. From this, we claim that the architecture of a specific system has under degeneration if:

(1) *more and more MCDs were discovered in the system*, or

(2) *the MCDs are more and more complex to fix.*

Obviously, these two plausible *symptoms* are concerned with the *quantity* and *complexity* of MCDs in the system. As described in Case Study 1 (see Section 4.1), the *number of components* that are changed in order to fix a MCD is a measure of complexity (Endres, 1975). Likewise, we can also measure the complexity of a MCD by the *number of code files* changed to fix this MCD.

Except these two symptoms (MCD quantity and complexity), there is another symptom – architectural deviation (see Section 2.3.1). If an architecture has deviated from its baseline, it has most likely degenerated (Hochstein and Lindvall, 2005). In addition, some symptoms of code decay (Eick et al., 2001) also manifest architectural degeneration, such as increasingly excessive size and complexity of components. We do not discuss these symptoms here because they are not directly related to software defects.

In a software system, different components could inflict varying impacts (called "contributions" here) on the architectural degeneration. Therefore, based on the skewed distribution of the components' contributions, we can identify the *degeneration-critical components* as the components which contribute *substantially* more to the architectural degeneration than the other components. Likewise for *degeneration-critical fix relationships*. Further, considering the above two symptoms (MCD quantity and complexity) of architectural degeneration, we can identify degeneration-critical components and fix relationships from the MCD quantity and complexity perspectives. For example, the MCD distribution chart in Case Study 1 (see Figure 4.1) shows that over 80% of MCDs are concentrated in 20% of the components and 75% of MCDs are involved in 10% of the fix re-

lationships. From the MCD quantity perspective, these 20% of components and 10% of fix relationships should be considered degeneration-critical in the system.

The DAD approach was thus proposed to measure the contribution of a component or a fix relationship of a specific system to the architectural degeneration, using proper MCD quantity and complexity metrics. The degeneration-critical components and fix relationships can be thus identified (according to a *criterion*) based on the measures of the components and fix relationships. Next, we describe a conceptual framework of this DAD approach.

5.2 A Conceptual DAD Framework

The DAD approach operationalizes the defect perspective (for diagnosing architectural degeneration) with MCD quantity and complexity metrics for components and fix relationships. The three goals that this approach was designed to achieve are: (1) identification of degeneration-critical components and fix relationships in a given system; (2) evaluation of persistence of components and fix relationships in relation to architectural degeneration; and (3) evaluation of architectural degeneration over time for the system.

We designed a conceptual DAD framework (see Figure 5.1) to achieve these three goals: (1)–(3). This framework defines five key steps (see the five boxes in Figure 5.1), which are outlined below.

Step 1: identify MCDs and fix relationships from the defect-fix history (defect records and change logs) of a given system (see Section 5.2.1).

Step 2: measure the system's components and fix relationships with MCD quantity and complexity metrics (see Section 5.2.2).

Step 3: identify degeneration-critical components and fix relationships according to the MCD quantity and complexity measures (see Section 5.2.3).

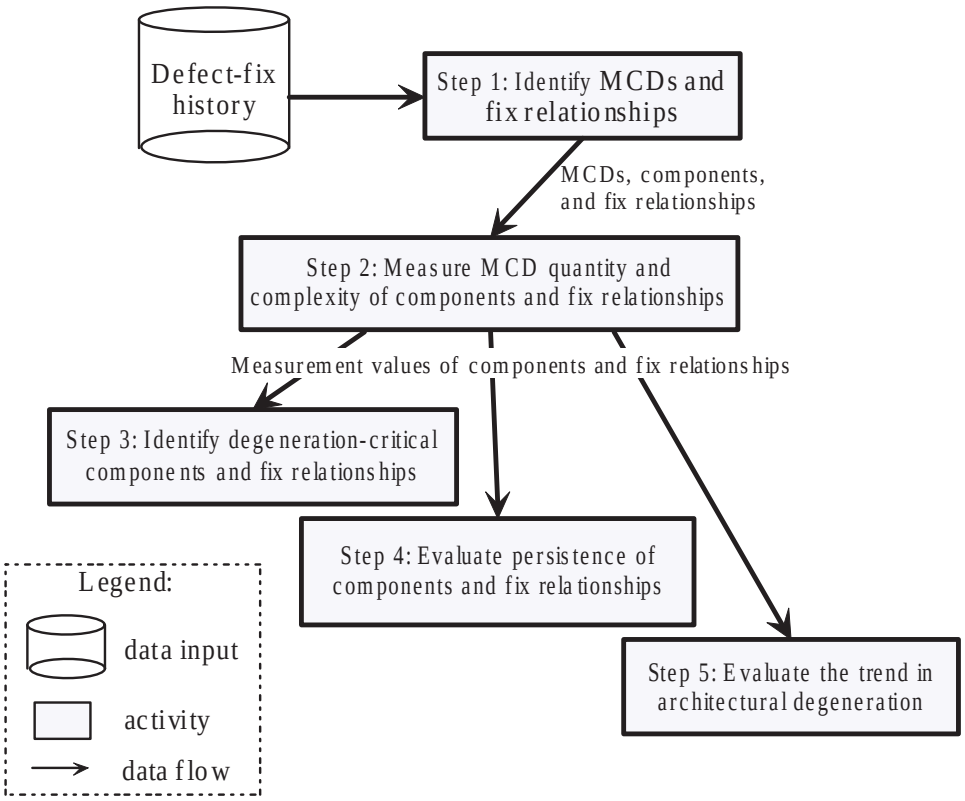


Figure 5.1: A conceptual DAD framework.

Step 4: evaluate the persistence of the components and fix relationships over development phases and releases (see Section 5.2.4).

Step 5: evaluate the architectural degeneration according to the MCD quantity and complexity measures over phases and releases (see Section 5.2.5).

Sections 5.2.1–5.2.5 give the details of these five steps, respectively. After that, in Section 5.2.6, we describe construction of "defect architectures" – the main output of this DAD approach (its five steps).

5.2.1 Step 1: Identification of MCDs and Fix Relationships

This step is to identify MCDs based on the defect-fix history (defect records and change logs) of a given system. In particular, if a defect record is matched

with a set of change logs from more than one component in the system, the corresponding defect is a MCD; consequently, a fix relationship is identified among the components which are changed together to fix this MCD.

Recall that a fix relationship is a relationship among components where fixing a MCD in one component requires changes in the other components in order to fix this MCD. We note that a fix relationship is an implicit relationship over the architecture, which points to architectural problems (D'Ambros et al., 2009).

5.2.2 Step 2: Measurement of Components and Fix Relationships

Step 2 is to measure components and fix relationships of a given system. DAD defines four metrics which are related to the *quantity* and *complexity* (number of components or code files fixed per defect) of MCDs that pertain to a component or a fix relationship in the system. These metrics are below.

M1 ("%MCDs") – the proportion of MCDs pertaining to a component against all MCDs in the system.

M2 ("#MCDs per KSLOC") – the average quantity of MCDs per thousand SLOC (KSLOC) of a given component.

M3 ("#Components fixed per MCD") – the average quantity of components fixed for a MCD in a given component.

M4 ("#Code files fixed per MCD") – the average quantity of code files fixed for a MCD in a given component.

Metrics **M1** ("%MCDs") and **M2** ("#MCDs per KSLOC") are concerned with the *proportion* and *density* of MCDs in a given component. The more the MCDs pertain to a component (**M1**), the greater the contribution of this component to architectural degeneration. Metric **M2** complements **M1** by normalizing the MCD quantity by the component size (in KSLOC). Metrics **M3** ("#Components

fixed per MCD") and **M4** ("#Code files fixed per MCD") are concerned with the *complexity* of MCDs at the *component* and *code file* levels in terms of the number of fixes. The more complex the MCDs pertain to a component, the greater the contribution of this component to architectural degeneration. Note that other similar complexity metrics can be also defined in DAD, for example, number of subsystems or functions that are required to change in order to fix a defect is also a complexity measure.

Metrics **M1** ("%MCDs"), **M3** ("#Components fixed per MCD") and **M4** ("#Code files fixed per MCD") are also used analogously to measure fix relationships[1]. Metric **M1** can be used to measure the proportion of MCDs involving a fix relationship against all MCDs in the system. Analogous interpretations are attributed to metrics **M3** and **M4**.

5.2.3 Step 3: Identification of Degeneration-Critical Components and Fix Relationships

Based on the MCD quantity and complexity measures for the components and fix relationships (see Section 5.2.2), step 3 of DAD is then to identify the degeneration-critical components and fix relationships in the system. In particular, degeneration-critical components and fix relationships are identified as components and fix relationships which have *substantially* greater measures than other components and fix relationships. Determining whether a measure is "substantially" greater than another measure is based on the particular measure distribution. Therefore, DAD does not set up a definite *criterion* for identification of degeneration-critical components and fix relationships. This criterion must be determined in the specific context. An example of such criterion is described in a case study on a commercial legacy software system (see Section 6.3.1).

[1] A fix relationship has no size in SLOC, so we do not measure the MCD density (i.e., the **M2** metric) for fix relationships here.

5.2.4 Step 4: Persistence Evaluation for Components and Fix Relationships

Based on the MCD quantity and complexity measures for the components and fix relationships (see Section 5.2.2), step 4 of DAD is to evaluate the persistence of the components and fix relationships in relation to architectural degeneration. In particular, the components or fix relationships are claimed persistent if their measures keep relatively stable across development phases and releases. In practice, we use Spearman's rank correlation coefficient (or Spearman-value)[2] to evaluate the persistence of the measures of components and fix relationships across development phases and releases.

5.2.5 Step 5: Architectural Degeneration Evaluation

Based on the MCD quantity and complexity measures for the components and fix relationships (see Section 5.2.2), step 5 of DAD is to evaluate the architectural degeneration of a system over phases and releases. In particular, the architectural degeneration is claimed increased if these measures increased with time. Otherwise, it is decreased or mitigated. Accordingly, the architectural degeneration trend of the system can be discovered with the longitudinal evaluations. Note that the evaluations with different MCD quantity or complexity metrics (as defined in Section 5.2.2) could be different.

5.2.6 Defect Architecture Construction

Over the above five steps of the DAD approach, we note that DAD can ultimately derive *defect architectures* for a given system within particular phases and releases. Similar to a fault architecture (von Mayrhauser et al., 2000), a *defect architecture* is defined as a composition of components and fix relationships of a system.

[2]Similar to Pearson correlation coefficient (or Pearson-value), Spearman's rank correlation coefficient (or Spearman-value) is a correlation measure. It is considered as being the Pearson correlation coefficient between two *ranked* data arrays and its range is from -1 to +1.

A defect architecture is generally visualized as a "box-and-arrow" graph (Gorton, 2006, p. 117), where a "box" denotes a component and an "arrow" (actually, an undirected "edge") denotes a fix relationship. An attribute of a box or an arrow can denote its MCD quantity or complexity measure. Thus, defect architectures can be used to highlight degeneration-critical components and fix relationships in a given system over development phases and releases. They also support cross-longitudinal analysis of measures for components and fix relationships, and support evaluation of the architectural degeneration trend over time.

For example, Figure 1.1 (see page 4) shows three charts, each of which is a defect architecture (with MCD quantity measures represented by the thickness of the "boxes" and "edges"). These three defect architectures highlight the degeneration-critical components (e.g., the "box" C5) and fix relationships (e.g., the "edge" between C5 and C6) in the three releases of the commercial system (investigated in Case Study 2; see Section 6.3.1). We also find that some degeneration-critical components and fix relationships persist over the three releases. Especially, by comparing these three defect architectures we can evaluate the architectural degeneration trend of the system over the three releases. In particular, we note that, from the MCD complexity (i.e., metric **M3**) perspective, the architectural degeneration of the system increased as the system evolved from release 1 to release 2 but then decreased in release 3.

5.3 A DAD Prototype Tool

In this section, we describe a DAD prototype tool which implements the conceptual DAD framework (see Figure 5.1). We first present the main features of this tool in Section 5.3.1. We then describe its data input in Section 5.3.2, its data processing steps in Section 5.3.3, and its outputs in Section 5.3.4. Note that this prototype tool was built on an extended Relation Algebra (in order to implement the main features of the DAD approach). We describe this Relation Algebra in

Appendix A. Furthermore, Appendix B demonstrates the application of this DAD prototype tool on a commercial legacy system and the Eclipse Platform.

5.3.1 Main Features

This prototype tool implements the three goals of the DAD approach (see Section 5.2): (i) identification of degeneration-critical components and fix relationships in a given system, (ii) evaluation of persistence of components and fix relationships in relation to architectural degeneration, and (iii) evaluation of architectural degeneration of the system across development phases and releases. There are three basic features upon which these three core features are built: (1) identification of MCDs in a defect-fix dataset, (2) identification of fix relationships among components, and (3) measurement of components and fix relationships with MCD quantity and complexity metrics (see Section 5.2.2).

Except the above features, there are another three complementary visualization features implemented in this DAD prototype tool:

(a) *Defect architecture visualization.* A defect architecture is visualized as a *box-and-arrow* graph (Gorton, 2006, p. 117). It can highlight the degeneration-critical components and fix relationships in the system.

(b) *Visualization for persistence of components and fix relationships.* A persistence view is visualized with a bar or line chart. It aids understanding the obstinate problems in the system leading to architectural degeneration.

(c) *Architectural degeneration trend visualization.* An architectural degeneration trend is visualized with a bar or line chart. It aids evaluating the system's architectural degeneration with time.

We note that an ordinary Relation Algebra (such as Tarski's algebra of binary relations (Tarski, 1941) or Codd's algebra of n-ary relations (Codd, 1972)) has been used by some researchers to facilitate the visualization (Berghammer and

Fronk, 2003), transformation (Krikhaar et al., 1999), abstraction (Holt, 1999), aggregation (Holt, 1999), and analysis (Feijs et al., 1998) of architectures. We thus implemented the above features of the DAD prototype tool based on an extended Relation Algebra (specifically the work by Feijs and Krikhaar (1998) and Holt (1999)). The full algebra is described in Appendix A.

5.3.2 Data Input for the Tool

The main dataset under the investigation of the prototype tool is the defect-fix history (defect records and change logs) of a given software system. The required data attributes are described in Table 5.1. The system structure information is also processed in the prototype tool, which indicates which file belongs to which component and what is the file's size (in KSLOC). This table is self-explained, so we do not describe it here any more.

Table 5.1: Key attributes of the data input.

	Attribute	**Description**
Defect	ID	The unique identity of the defect report
	Release	The release where the defect was discovered
	Phase	The phase where the defect was discovered
	Component	The component where the defect was discovered
	Submit date	The date to submit the defect to the system
	State	The last state of the defect
Change (Fix)	ID	The unique identity of the change log
	Release	The release where the change was made
	Phase	The phase where the change was made
	File	The code file where the change was made
	Defect ID	The identity of the defect fixed by the change
System Structure (part)	Component	The component's name
	File	The file's name
	Size	The file's size

For a usual system, the system structure information is mostly available. Such a system usually contains defects. Fixing a defect requires changes (fixes) to the code base. The discovered defects are usually recorded in a defect-tracking

database (collecting historical defect records, e.g., Bugzilla[3]) and the changes are logged in a version control system (collecting historical changes made in a code base, e.g., Concurrent Versions System or CVS[4]). Therefore, the key attributes of the defect records and change logs (shown in Table 5.1) can be mostly gathered from defect-tracking databases and version control systems. We thus claim that the data required by this prototype tool is widely available.

5.3.3 Data Processing by the Tool

Figure 5.2 illustrates the data processing by the DAD prototype tool. In particular, it specifies the steps used to implement the main features (see Section 5.3.1). We briefly describe these steps below.

- *Map change logs to defect records.* Each defect is associated with a set of changes (in a code base) by matching the "Defect ID" field of change logs to the "ID" field of defect records (see Table 5.1).
- *Locate defects in components.* Each defect is located in component(s) in which at least one code file is changed in order to fix this defect.
- *Identify MCDs*; see Section 5.2.1.
- *Identify fix relationships*; see Section 5.2.1.
- *Measure components and fix relationships with the MCD quantity and complexity metrics*; see Section 5.2.2.
- *Set up criteria and identify degeneration-critical components and fix relationships*; see Section 5.2.3.
- *Create and visualize persistence view for components and fix relationships* by gathering the measures for components or fix relationships over time, which is then visualized as a bar or line chart.

[3]See Bugzilla's web site: `http://www.bugzilla.org/` (last access in November 2010).
[4]See a CVS web site: `http://www.nongnu.org/cvs/` (last access in November 2010).

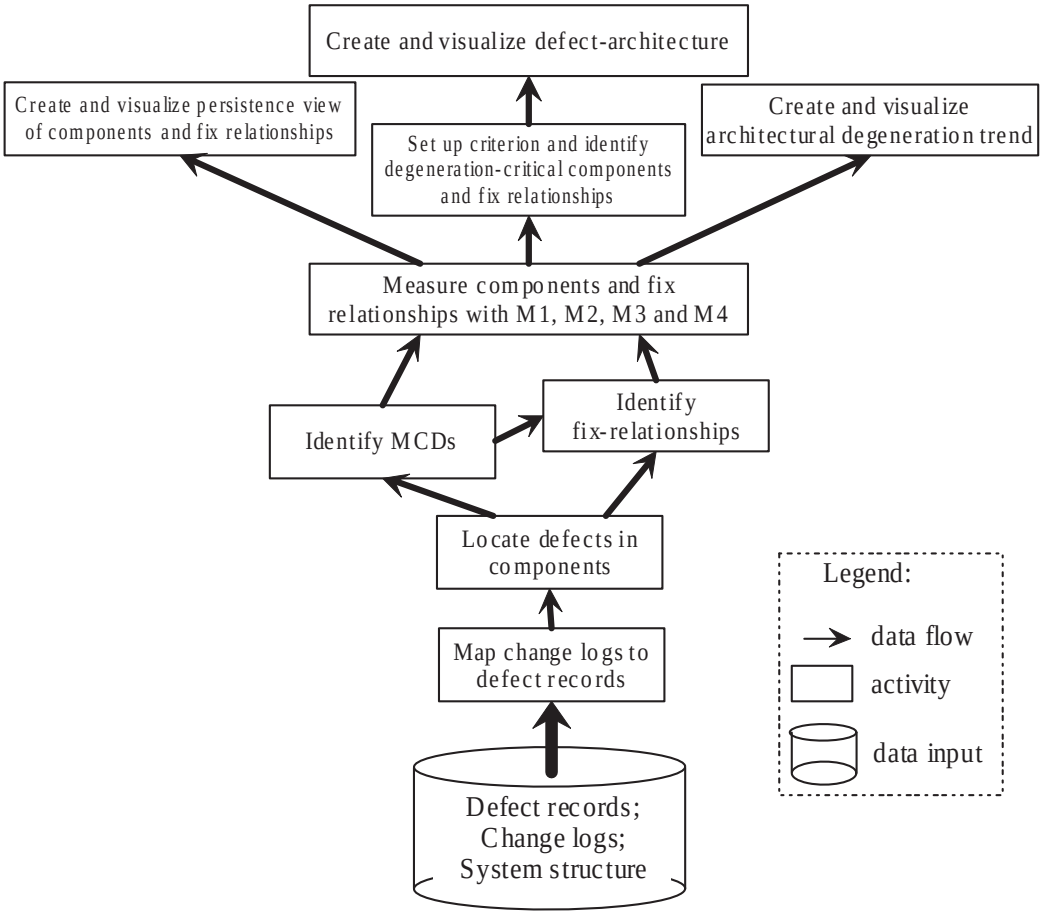


Figure 5.2: Data processing by the DAD prototype tool.

- *Create and visualize architectural degeneration trend* by gathering the average measures of components or fix relationships over time, which is then visualized as a bar or line chart.

- *Create and visualize defect architecture*; see Section 5.2.6.

Overall, the processing steps above posed implement the conceptual DAD framework (see Figure 5.1) in the prototype tool.

5.3.4 Output of the Tool

Following the above description of the data processing steps of in the DAD prototype tool, we describe the outputs of using this tool on a given system:

- Characteristics of defects (including MCDs). The tool creates charts to demonstrates quantity and complexity characteristics of defects (including MCDs) in a given system. See examples in Section B.2.
- Measures of components and fix relationships. The tool creates bar and line charts to show MCD quantity and complexity measures for components and fix relationships of the system. See examples in Section B.2.
- Defect architectures. The tool creates box-and-arrow graphs to visualize defect architectures for the system. See examples in Section B.3.
- Persistence view for components or fix relationships. The tool creates bar and line charts to visualize the measures of components or fix relationships over phases and releases. See examples in Section B.2.1.
- Architectural degeneration trend. The tool creates bar and line charts to visualize architectural degeneration trend over phases and releases for a given system. See examples in Section B.2.2.
- Profile of the system, for example, the size (in SLOC) of a component, and the number of defects and changes occurred in a component. See examples in Section B.1.

These outputs are created by following the data processing steps shown in Figure 5.2. Later in Appendix B, we demonstrate several typical outputs of this prototype tool used on a commercial legacy system.

5.4 Comparison and Discussion

There are some metrics defined in the literature for architectural degeneration measurement. For examples, Jaktman et al. (1999) define complexity metrics (e.g., the average number of calls per component); and Lindvall et al. (2002) define two similar metrics (i.e., "coupling-between-modules" (CBM) and "coupling-between-module-classes" (CBMC)). However, these metrics do not consider defect

characteristics related to architectural degeneration. Therefore, these metrics cannot be used to measure architectural degeneration from defect perspective. DAD defines several MCD quantity and complexity metrics (see Section 5.2.2). These metrics operationalize the defect perspective, which can thus complement the deviation-based measurement (e.g., (Jaktman et al., 1999) and (Lindvall et al., 2002)) for architectural degeneration diagnosis.

Recall Section 2.3.3 where we describe three diagnosis techniques, including architectural deviation detection, defect-prone component (DPC) identification, and fault and change architectures construction. In Section 2.3.5 we discuss the deficiencies of these techniques in diagnosing architectural degeneration from defect perspective. The defect is an indicator of poor system quality. Analysis of defects in history of a given system can uncover potential problems related to the system evolution (von Mayrhauser et al., 2000). Therefore, DAD complements these techniques by offering new information about architectural degeneration.

Furthermore, the DAD approach (and its prototype tool) also complements other related techniques such as reverse engineering (for architectural degeneration prevention) and re-engineering (for architectural degeneration treatment). Reverse engineering can help detect deviations (Murphy et al., 2001) in an architecture against its baseline (Krikhaar, 1997) from structural perspective. DAD offers a defect perspective, which can complement the structural perspective. Re-engineering can greatly improve the structure of a system. However, re-engineering the whole system is usually extremely costly. In some situations, re-engineering the most problematic components in the system is a cost-effective alternative for system quality concerns (Booch, 2008). DAD can identify degeneration-critical components in a system which most likely require re-engineering.

In addition, DAD supports architectural degeneration evaluation over time, which can aid long-term system management (Jacobson and Lindström, 1991) (also see (Sommerville, 2006, p. 506)), such as whether or not, or when, to freeze

the released system, transform the system's architecture to a new form, re-engineer the system, or initiate new-release development.

5.5 Key Points of the DAD Approach

We summarize several key points of the DAD approach (see Section 5.2) and its prototype tool (see Section 5.3) as below.

- DAD operationalizes the defect perspective for diagnosing architectural degeneration with the MCD quantity and complexity metrics.
- The three goals of DAD are: (1) identification of degeneration-critical components and fix relationships in a given system; (2) evaluation of persistence of components and fix relationships in relation to architectural degeneration; and (3) evaluation of architectural degeneration of a given system over development phases and releases.
- A conceptual DAD framework (see Figure 5.1) contains five steps: (1) identification of MCDs and fix relationships; (2) measurement of components and fix relationships; (3) identification of degeneration-critical components and fix relationships; (4) evaluation of persistence of components and fix relationships in relation to architectural degeneration; and (5) evaluation of architectural degeneration across phases and releases.
- The DAD prototype tool implements the DAD approach, which supports visualization of measures of components and fix relationships of a given system of defect architectures for the system (a development or release thereof), and of architectural degeneration trend over development phases and releases. See example outputs of this prototype tool in Appendix B.

Validation of this DAD approach is illustrated by a case study on a commercial legacy system, which is described in the next chapter.

Chapter 6

Case Study 2: DAD Validation

The previous chapter describes the DAD approach. Following that, this chapter describes a confirmatory case study (Case Study 2) which validates the DAD approach to three major, successive, releases of a commercial system (of size over 1.5 million SLOC and age over 13 years). This system is a core subsystem of the even larger system investigated in Case Study 1 (see Section 4.3.1).

In Section 6.1, we describe five research questions for this case study. In Section 6.2, we describe the study design. We present and interpret the study findings in Section 6.3. We then discuss threats to validity of the study findings in Section 6.4 and describe implications in Section 6.5. Finally, we give a summary in Section 6.6. Assessment of this study is described in Section 8.4.

6.1 Research Questions and Metrics

Recall Section 5.2 that the DAD approach (see Figure 5.1) supports: (1) identification of degeneration-critical components and fix relationships in a given system; (2) evaluation of persistence of components and fix relationships in relation to architectural degeneration; and (3) evaluation of architectural degeneration of a given system over time. We thus, in Case Study 2, define five relevant questions (see Section 1.3.2) to address these three aspects on a commercial legacy system.

Q1: *Do some components in a system contribute more than other components to the system's architectural degeneration?*

Q2: *Do components contribute persistently to architectural degeneration over development phases and releases?*

Q3: *Do some fix relationships in a system contribute more than other fix relationships to the system's architectural degeneration?*

Q4: *Do fix relationships contribute persistently to architectural degeneration over development phases and releases?*

Q5: *What is the trend in architectural degeneration from the defect perspective?*

Question **Q1** is concerned with the *quantity* of MCDs spread across the system's components. It is also concerned with the number of components or code files changed (a complexity issue) to fix a MCD. Question **Q2** complements this with its focus on persistence across development phases and system releases. That is, it would highlight components that are tenacious in their defect quality and across phases and releases. Together, **Q1** and **Q2** would give us a handle on the parts of the architecture that need management attention. Following this, we define two more questions (**Q3** and **Q4**) for "fix relationships" analogous to **Q1** and **Q2**. Based on these four questions, question **Q5** examines system degeneration across development phases and releases.

Note that these five questions, **Q1**–**Q5**, are addressed by following the DAD approach in the case study. In particular, **Q1** and **Q3** are related to Step 3 of the DAD approach (see Section 5.2.3); **Q2** and **Q4** are related to Step 4 (see Section 5.2.4); and **Q5** is related to Step 5 (see Section 5.2.5). These are clearly important questions to ask about system management. Earlier we saw the price to pay due to degenerating architectures (see the example Mozilla, Linux-kernel and 5ESS issues in Section 1.1).

The novelty and relevance of these questions were validated through reviews and discussions with collaborating researchers and practitioners from the sponsoring organization, as were the four MCD quantity and complexity metrics defined in the DAD approach (see Section 5.2.2). Here, we list these four metrics below and then briefly discuss their relationships to questions **Q1**–**Q5**.

M1 ("%MCDs") – the proportion of MCDs pertaining to a component against all MCDs in the system.

M2 ("#MCDs per KSLOC") – the average quantity of MCDs per thousand SLOC of a given component.

M3 ("#Components fixed per MCD") – the average quantity of components fixed for a MCD in a given component.

M4 ("#Code files fixed per MCD") – the average quantity of code files fixed for a MCD in a given component.

These four metrics deal with questions **Q1** and **Q2** described above. Meanwhile, metrics **M1** ("%MCDs"), **M3** ("#Components fixed per MCD") and **M4** ("#Code files fixed per MCD") are also used analogously to measure fix relationships (i.e., deal with questions **Q3** and **Q4**). Further, question **Q5** is addressed by looking at the results across questions **Q1** and **Q3**.

Next, we describe the design of Case Study 2 in order to address the above five questions (**Q1**–**Q5**) using these metrics (**M1**–**M4**).

6.2 Case Study Design

This section describes the design of Case Study 2, which includes: the subject system and data, the data collection, clean-up, and analysis procedures, descriptive system statistics, and the overall case study process.

6.2.1 Description of the System and Data

Case Study 1 (see Chapter 4) investigates the defect history of a large legacy system of size over 20 million SLOC and age over 20 years (see Section 4.3.1). This case study (Case Study 2) focused further upon a core subsystem of that system, which is of size over 1.5 million SLOC and age over 13 years. In particular, this study investigates the defect-fix history (defect records and change logs) of three major, successive, releases of this (sub)system.

This system contains 10 components (labeled C0–C9) and each component is implemented by a number of code files (mainly, written in the language C). The system's size increased by 14% from release 1 (about 1.6 million SLOC) to release 2 and then by 7% to release 3. The size growth in releases 2 and 3 were mainly due to enhancements; subsequently, *restructuring* was carried out on release 3 in order to improve the system structure. These three releases are still under active maintenance now.

We focused on the defect-fix history (defect records and change logs) of these three releases, containing approximately 1100, 550 and 600 defect records while spanning approximately six, five and three years respectively. Each defect-fix record includes key information pertaining to: the *release*, *phase* and *component* in which the defect was discovered; the *state* that the defect is currently in (e.g., "working", "validated", "closed", etc.); the *reference* that indicates defect rediscovery (i.e., associates a defect to its previous occurrence); the *submit date* that this defect was submitted to the defect-tracking database; and the *file*(s) and *component*(s) that were changed in order to fix the defect.

Table 6.1 shows four example defect-fix records. For example, defect 0020 (see column "ID") was discovered in component C1 (see column "Component") while "testing" (see column "Phase") release "r1" (see column "Release"), and this defect was fixed in two code files "C1/.../foo1.C" and "C2/.../foo2.C" in components C1 and C2 (see column "Component*") respectively.

Table 6.1: Example defect-fix records (only key fields).

ID	Release	Phase	Component	File	Component*
0020	r1	testing	C1	C1/.../foo1.C	C1
0020	r1	testing	C1	C2/.../foo2.C	C2
0021	r1	field	C3	C3/.../foo3.C	C3
0021	r1	field	C3	C4/.../foo4.C	C4

Note that the values in columns "Component" and "Component*" could be different. For example, Table 6.1 shows that defect 0020 was discovered in component C1 (the "Component" value) but was fixed in components C1 and C2 (the "Component*" values). Likewise for defect 0021 which was discovered in component C3 but was fixed in components C3 and C4. Also note that the "Component*" field does not exist in the collected, raw, defect-fix dataset, which was inserted later during the data cleaning process, described below.

6.2.2 Data Collection and Clean-up Procedures

The collection process for the defect records for this case study is similar to that for the defect dataset under investigation of Case Study 1. We do not repeat this process here; see Section 4.3.2 for details. The collection process for the change logs for this case study is described below. Changes were made to the code base in order to fix each newly recorded defect in the defect-tracking database; the changes were logged in the version control system. We extracted the change logs from the version control system.

Next, we describe the five main steps used to clean up the collected, raw defect-fix dataset, as below. Some of these steps are similar to those for cleaning up the defect dataset in Case Study 1 (see Section 4.3.2). Note that these steps were carried out mainly with programming scripts.

Step 1: we removed defects which are rediscoveries or not closed or validated in the system. This was carried out based on the *state* field of defect records. In particular, defects records which are not "closed", "integrated", "delivered"

and "validated" are excluded from the dataset.

Step 2: we removed change logs where changes were made in non-code files (e.g., documentation files). Here the code files are mainly .c files (in the language C). This was carried out based on the "file" field of change logs.

Step 3: we filled in the "Component*" value for each defect-fix record (see examples in Table 6.1). For each defect record, the "Component*" value is a component name indicating a component that was changed in order to fix this defect. This information is not recorded automatically in the defect-fix database. A simple text analysis technique was used to identify the component name from the "File" field and copy that component name to the "Component*" field. For example, it identified the component name "C1" from the "File" field value "C1/.../foo1.C", so the corresponding "Component*" value is "C1".

Step 4: we identified the *Internal* and *Field* phases for defect records. Note that the internal phase subsumes functional and system testing and performance quality assurance phases. Defect-fix records from other phases (e.g., development[1]) were removed. This step was carried out based on the *phase* field of the defect records (see examples in Table 6.1).

Step 5: we removed "outliers" from the dataset. For example, we find that there is a defect which required fixes in 130 code files while the other defects required fixes in, on average, approximately 2.2 code files (at most 80 code files; see Figure 6.2). This defect was treated as an outlier and was thus excluded from the analysis.

[1]The reason we removed defect-fix records made during the development phases (e.g., design and coding) is that developers could have fixed several defects but recorded them together as one defect and they also could have made changes in the code base which were recorded as fixes to a defect but which were not really fixing that defect.

6.2.3 Data Analysis Procedures

We then wrote programming scripts to analyze the defect-fix records. Statistical methods such as Pearson correlation coefficient (or Pearson-value) and Spearman's rank correlation coefficient (or Spearman-value)[2] were also used to evaluate correlation and persistence for components and fix relationships measures.

The data analysis procedures were carried out based on MCDs identified from the defect-fix dataset. In particular, we identified MCDs based on the "Component*" values (rather than the "Component" values). For example, Table 6.1 indicates that defects 0020 and 0021 are two MCDs. Meanwhile, the fix relationships among components can be consequently identified when a MCD is identified. For example, there is a fix relationship between components C1 and C2 because of MCD 0020. Likewise for the fix relationship between components C3 and C4 (due to MCD 0021). Note that, in this case study, we only identified binary fix relationships in the system. The reason is that the majority of MCDs span only two components, see Figure 6.1 (in Section 6.2.4 below) for details. Fix relationships spanning more than two components were subsequently decomposed into a binary form and thus were used in the study.

Note that, in Case Study 2, the method of MCD identification is different from that for Case Study 1. The former is based on matching change logs to defect records (mainly, "Component*" field information); the latter is based only on defect records (by identifying parent-children relationships – Section 4.2). See a detailed comparison of these two methods in Section 7.1.

The data analysis procedures of this case study were centered on the MCDs in the system. They are incorporated into the case study design; see Section 6.2.5 for details. Before we describe this study design, we first give some descriptive statistics about the defects in the subject system, below.

[2]Similar to Pearson correlation coefficient (or Pearson-value), Spearman's rank correlation coefficient (or Spearman-value) is a correlation measure. It is considered as being the Pearson correlation coefficient between two *ranked* data arrays, ranging from -1 to +1.

6.2.4 Descriptive Defect Statistics

Here, we first describe the components' sizes and defect proportions in the system, then illustrate the defect distributions by number of components or code files in the system. Note that the case study findings (shown in Section 6.3) are related to these descriptive statistics of the system and defects.

A Basic Profile of the System

Table 6.2 gives a basic description of the subject system. It shows the sizes (thousand SLOC – see column "KSLOC") and defect proportions (proportion of defects in the system – see column "%Defects") of the 10 components (written C0–C9) in the three releases (written r1, r2 and r3). For example, component C0 (see row "C0") is of size 40 KSLOC in release 1 (r1), and there are 3% of defects in the subject system which emanate from C0 in r1.

Table 6.2: Basic profile of the subject system of Case Study 2.

Component	KSLOC				%Defects			
	r1	r2	r3	Avg	r1	r2	r3	Avg
C0	40	43	44	42	3%	2%	3%	3%
C1	155	174	188	172	8%	11%	11%	10%
C2	22	36	45	34	5%	8%	9%	7%
C3	337	386	408	377	24%	26%	19%	23%
C4	9	10	11	10	0%	0%	2%	1%
C5	557	633	706	632	35%	44%	51%	44%
C6	275	316	316	302	34%	19%	12%	22%
C7	137	139	140	139	3%	3%	1%	3%
C8	18	20	20	20	2%	2%	1%	2%
C9	129	165	182	159	10%	8%	11%	10%
Mean	168	192	206	189	12%	12%	12%	12%
StDev	176	200	219	198	14%	14%	15%	14%

In Table 6.2, the average size ("KSLOC") and defect proportion ("%Defects") of each component over the three releases are shown in columns "Avg"; and the average and standard-deviation values of the sizes and defect proportions of the components in each release are shown in rows "Mean" and "StDev". For exam-

ple, the average size of a component in release r1 is 168 KSLOC (with standard deviation value 176), the average size of component C0 over the three releases is 42 KSLOC, and the defects pertaining to C0 account for an average of 3% of all defects investigated in the system.

We note from Table 6.2 that the sizes of r1, r2 and r3 are approximately 1680, 1920, and 2060 KSLOC, respectively (see row "Mean"). This indicates that the system's size increased by about 14% from r1 to r2 and then 7% in r3 (as mentioned in Section 6.2.1). We also note from Table 6.2 that the component sizes are not uniform in the system. For example, the size of component C5 is 557 KSLOC (see row "C5") in r1, which accounts for 33% of the system's size; likewise for 33% and 39% in the remaining r2 and r3. Whereas component C4 (see row "C4") is very small: an average of less than 1% of the system's size in the three releases.

Similar to the component size distribution, the defect proportion distribution is also skewed. Across the three releases, component C5 contains an average of 44% of the all defects in the system, but each of components C0, C4, C7 and C8 contains less than 5% of the all defects. Note that each MCD is counted multiple times as it spreads over multiple components. This explains why the average defect proportion of the 10 components is greater than 10% (actually, an average of 12%; see row "Mean").

Defect Distributions

Figure 6.1 illustrates the highly-tailed distribution of defects by number of components spanned. It shows that in the three releases together, 82% of defects are *single-component* defects (SFDs) – defects requiring fixes confined to one component. This means that MCDs account for only 18% of all defects[3]. Further, this figure shows that the MCDs spanning two components account for 77% of the

[3]In particular, 19%, 17% and 17% of all defects are identified as MCDs in the three releases (r1, r2 and r3) of the subject system, respectively; see Figure 6.3 for details.

total MCDs. This indicates that only 4% of defects spanned more than two components in the system. The fix relationships involved in these 4% of defects were subsequently decomposed into a binary form and thus were used in the study.

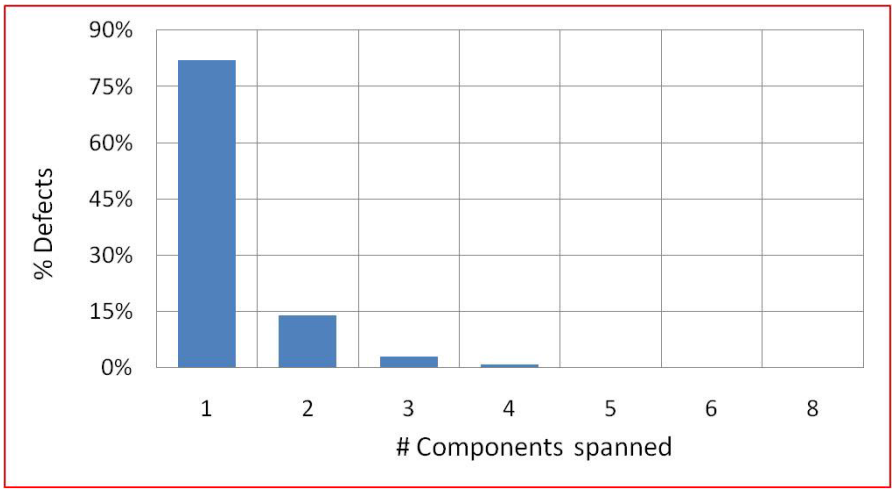


Figure 6.1: Distribution of defects by number of components spanned.

Similar to Figure 6.1, Figure 6.2 illustrates the highly-tailed distribution of defects by number of code files spanned.

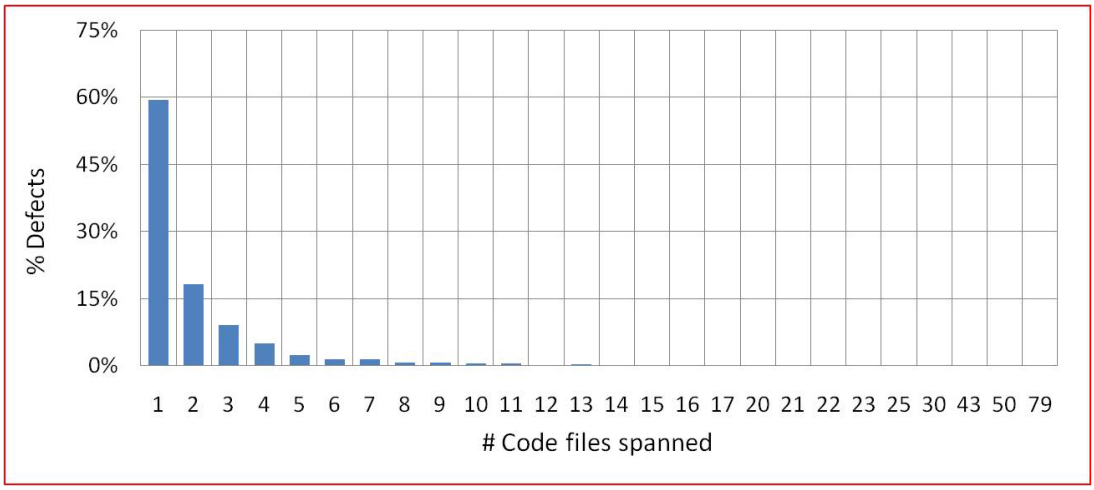


Figure 6.2: Distribution of defects by number of code files spanned.

Figure 6.2 shows that in the three releases together, 41% of defects spanned more than one code file in the system. We call these defects the *multiple-file defects* (MFDs). Correspondingly, the remaining 59% of defects that were confined in

single files are called the *single-file defects* (SFDs). Note that the 18% MCDs (see Figure 6.1) are subsumed in these 41% MFDs because a MCD is always a MFD. Also, Figure 6.2 shows that 85% of MFDs spanned two to five code files. This indicates that there are only 6% of defects which spanned more than five code files in the system.

MCDs vs. Non-MCDs

Figure 6.3 describes two bar charts: the top bar chart shows the proportions of MCDs and non-MCDs in the system (its three releases), and the bottom bar chart shows the average number of changes required to fix a MCD or non-MCD.

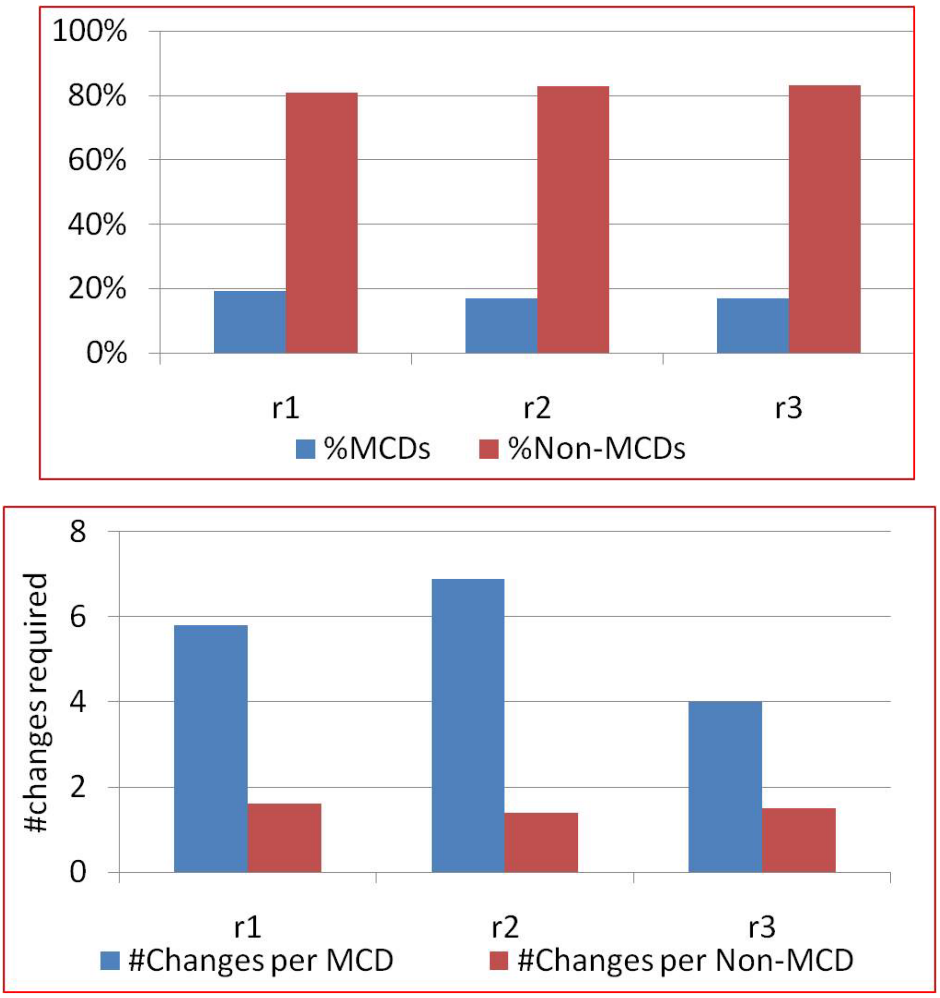


Figure 6.3: MCDs vs. non-MCDs.

We find from Figure 6.3 that approximately 20% of defects are MCDs in the system. In particular, 19%, 17% and 17% of defects are MCDs in the three releases (r1, r2 and r3), respectively (see the "%MCD" bar in the top chart). We can also find that fixing a MCD requires an average of nearly 3.5 times changes as much as that for fixing a non-MCD (see the bottom chart). This is consistent with the relevant finding in Case Study 1 (based purely on the defect dataset) – on average, fixing a MCD requires near 3 times changes (based on components) as much as that for fixing a non-MCD.

6.2.5 Case Study Process

Armed with the questions (**Q1**–**Q5**) and the MCD quantity and complexity metrics (**M1**–**M4**) from Section 6.1, we define these six key steps for conducting the case study, which resulted from the discussions conducted with, and reviews by, the collaborating partners:

Step 1: we cleaned up the defect-fix records for the subject system. Some scripts are used to carry out this process. See Section 6.2.2 for details.

Step 2: we identified MCDs and then identified fix relationships in the system. This step has been discussed in Section 6.2.2.

Step 3: we measured each component in the system with metrics **M1**, **M2**, **M3** and **M4** (see Section 6.1), and then identified the *degeneration-critical* components (addressing question **Q1**), see Section 6.3.1.

Step 4: we measured the persistence of each component's measures across phases and releases (addressing question **Q2**); see Section 6.3.2.

Step 5: we repeated Steps 3 and 4 for the fix relationships in the system (addressing questions **Q3** and **Q4**); see Sections 6.3.3 and 6.3.4.

Step 6: we derived the architectural degeneration trend for the system and also analyzed the impact of system restructuring on architectural degeneration (addressing question **Q5**), see Section 6.3.5.

Steps 1–2 are the preparatory steps for the subsequent steps, 3–6. In Step 3, an appropriate criterion was determined (based on the actual distribution of component measures) for identifying degeneration-critical components of the system. In Step 4, the "persistence" was measured by the strength of rank correlation (Spearman-value)[4] of the measures between different phases and releases. In Step 5, a similar criterion to that in Step 3 was used to identify degeneration-critical fix relationships, and a similar measurement to that in Step 4 was used for the persistence of fix relationships. Step 6 was conducted based on the measures for components and fix relationships (Steps 3 and 5). The main results of the steps 3-6 above are described in the next section.

6.3 Analysis of Data, Results, Interpretation, and Comparisons

In this section, we present and interpret the case study findings related to the questions, **Q1**–**Q5**, posed in Section 6.1. In Section 6.3.1, we describe the components' "contributions" to architectural degeneration (for **Q1**); in Section 6.3.2, we describe the persistence of the components' contributions (for **Q2**); in Sections 6.3.3 and 6.3.4, we describe the fix relationships' contributions (for **Q3**) and their persistence (for **Q4**); and in Section 6.3.5, we describe the architectural degeneration trend over time of the system (for **Q5**). Meantime, we also compare the findings against related work (if any) in the literature. Finally, in Section 6.3.6, we describe example defect architectures derived with the DAD approach for the subject system. These defect architectures can help easily understand the architectural degeneration phenomenon in the system.

[4]Here, the "persistence" has two aspects to its meanings: (a) the cross-longitudinal measures are close for the components; and (b) the *ranks* of the measures change little. Aspect (a) is determined by the actual measures for each phase or release. For aspect (b), we use Spearman-value to measure the rank difference between the measures for different phases and releases. The larger the Spearman-value, the more persistent the measures.

Recall the beginning paragraph of Section 4.4 (for Case Study 1) where we note the fundamental principles for data interpretation, such as: (a) corresponding with the (quantitative or qualitative) findings (Bracey, 2006, p. 32); (b) incorporating the context variables (Basili et al., 2006, p. 68); and (c) reducing subjective judgement (Münch, 2006). Here, we also follow these principles for the interpretation of the findings of Case Study 2, as below.

6.3.1 Components' Contributions (Q1)

Note from Section 6.1 that the contribution of a component to architectural degeneration is measured with the MCD quantity and complexity metrics (**M1**–**M4**; see Section 6.1); the greater the MCD quantity or complexity measures of a component, the more this component contributes to architectural degeneration.

Components' Measures

The MCD percentage (**M1**) and density (**M2**) measures of the components (C0-C9) are shown in Table 6.3. For example (see row "C0"), across releases r1, r2 and r3, component C0 contained MCDs which account for 4%, 7% and 7% of all MCDs; and there are 0.23, 0.16 and 0.16 MCDs per KSLOC.

We note from Table 6.3 that the distribution of MCDs over the components is skewed. For example (see row "C5"), MCDs contained in component C5 account for 64%, 59% and 63% of all MCDs in r1, r2 and r3, respectively. MCDs contained in component C4 (see row "C4") account for an average of only 3% of all MCDs. Likewise for the skewed distribution of MCD-density values over the 10 components. For example, over the three releases, component C2 (see row "C2") has an average of 0.86 MCDs per KSLOC, which is about 300% more than the average value for the 10 components. Component C7 (see row "C7") has an average of 0.08 MCDs per KSLOC, which is only 35% of the average value for the components. Correlation between the measures is described below. One can

Table 6.3: Component measures with metrics **M1** ("%MCDs") and **M2** ("#MCDs per KSLOC").

Component	%MCDs (**M1**)				#MCDs per KSLOC (**M2**)			
	r1	r2	r3	Avg	r1	r2	r3	Avg
C0	4%	7%	7%	6%	0.23	0.16	0.16	0.18
C1	15%	17%	23%	18%	0.20	0.09	0.12	0.14
C2	12%	28%	32%	24%	1.10	0.76	0.71	0.86
C3	37%	38%	22%	32%	0.23	0.09	0.05	0.12
C4	0%	1%	7%	3%	0.23	0.09	0.64	0.25
C5	64%	59%	63%	62%	0.24	0.09	0.09	0.14
C6	50%	50%	28%	43%	0.37	0.15	0.09	0.20
C7	8%	14%	3%	8%	0.12	0.09	0.02	0.08
C8	2%	6%	3%	4%	0.28	0.30	0.15	0.24
C9	36%	26%	32%	31%	0.57	0.15	0.18	0.30
Mean	23%	25%	22%	23%	0.33	0.20	0.22	0.25
StDev	22%	19%	19%	19%	0.31	0.21	0.25	0.22

Note: the rows "Mean" and "StDev" show the average and standard deviation values of the measures across the 10 components in a system release. The "Avg" sub-columns show the average measures of the 10 components across the three releases.

also analyze the trend of MCD quantity across the three releases. We discuss this trend later in Section 6.3.5, where other trend measures are also discussed.

Table 6.4 shows the MCD complexity ("#Components fixed per MCD" (**M3**) and "#Code files fixed per MCD" (**M4**)) measures of the 10 components. For example, fixing a MCD in component C0 (see row "C0") requires, on average, changes in 2.1, 2.3 and 2.3 components and in 3.1, 3.0 and 3.9 code files in the three releases, respectively.

We note from Table 6.4 that the distributions of the components' MCD complexity measures (i.e., their "#Components fixed per MCD" and "#Code files fixed per MCD" values) have a low degree of dispersion. The standard deviation values of these measures (see row "StDev") stays relatively small compared against the average values (see row "Mean"). This is different from the skewed distributions of the MCD quantity measures shown in Table 6.3. The correlation between these component measures is described below.

Table 6.4: Component measures with metrics **M3** ("#Components fixed per MCD") and **M4** ("#Code files fixed per MCD").

Component	#Components fixed per MCD (**M3**)				#Code files fixed per MCD (**M4**)			
	r1	r2	r3	Avg	r1	r2	r3	Avg
C0	2.1	2.3	2.3	2.2	3.1	3.0	3.9	3.3
C1	2.7	3.1	2.4	2.7	8.2	9.7	4.1	7.4
C2	2.5	2.8	2.2	2.5	4.1	6.5	4.9	5.2
C3	2.4	2.7	2.5	2.5	6.1	8.1	4.7	6.3
C4	0.0	2.0	2.0	1.3	0.0	3.0	3.6	2.2
C5	2.3	2.7	2.3	2.4	5.5	6.4	4.4	5.4
C6	2.4	2.6	2.5	2.5	6.2	6.1	4.4	5.6
C7	3.1	2.8	2.7	2.8	13.1	6.9	3.3	7.8
C8	3.0	3.7	2.0	2.9	5.0	10.7	3.0	6.2
C9	2.5	3.1	2.4	2.7	6.2	8.7	3.8	6.2
Mean	2.3	2.8	2.3	2.5	5.8	6.9	4.0	5.6
StDev	0.9	0.5	0.2	0.4	3.4	2.5	0.6	1.7

Correlation Analysis

Table 6.5 describes the Pearson correlation coefficient values (Pearson-values)[5] among the component measures with the MCD quantity and complexity metrics (i.e., **M1**, **M2**, **M3** and **M4**; see Tables 6.3 and 6.4). Meanwhile, we also investigate the correlations with the sizes (column "KSLOC") and defect proportions (column "%Defects") of the components (see Table 6.2). For example, the Pearson-value between the "%MCDs" (**M1**) and "#MCDs per KSLOC" (**M2**) measures is 0.00, indicating "no" correlation between the MCD percentage and density measures of the components in the system.

Table 6.5 indicates that there are two correlation clusters; the measures in each cluster are positively correlated (moderate to large)[6] with each other.

[5]There are 30 data pairs (i.e., 10 components for 3 releases) for each correlation analysis. The critical value for 30-pairs based 0.95-significance for two-tailed test of the Pearson-values is 0.296 (see `http://www.une.edu.au/WebStat/unit_materials/c6_common_statistical_tests/test_signif_pearson.html` (last access in November 2010)). In Table 6.5, the P-values that we focus on are larger than 0.296, indicating that they are statistically significant.

[6]Access to `www.umich.edu/~exphysio/MVS250/PearsonCorr.doc` (last access in November 2010.) for more details about interpretation of Pearson correlation values.

Table 6.5: Correlations between component measures w.r.t. different metrics.

	M2	**M3**	**M4**	KSLOC	%Defects
M1: %MCDs	0.00	0.10	0.30	0.87	0.92
M2: #MCDs per KSLOC		-0.05	0.05	0.28	0.24
M3: #Components fixed per MCD			0.63	0.17	0.08
M4: #Code files fixed per MCD				0.28	0.24
KSLOC (i.e., component size)					0.94

- The first cluster contains the measures with metrics "KSLOC", "%Defects" and "%MCDs" (**M1**). The Pearson-value is 0.94 between "KSLOC" and "%Defects", 0.87 between "KSLOC" and "%MCDs", and 0.92 between "%Defects" and "%MCDs", indicating three near-linear, positive, correlations. This is a well-known fact that does not impart much knowledge here. However, the MCD finding added to the above finding is new. An example component is C5. It is the largest component in the system (see row "C5" in Table 6.2), which also contains the greatest number of defects and MCDs in the three releases, see row "C5" in Table 6.3.

- The second cluster contains the measures with metrics "#Components fixed per MCD" (**M3**) and "#Code files fixed per MCD" (**M4**), between which the Pearson-value is 0.63, indicating a significant, positive correlation. This indicates that the more other components required to fix for a MCD in a given component, the more code files required to fix for a MCD in this component. An example component is C7, which has the greatest **M3** and **M4** measures in release 1; see row "C7" in Table 6.4.

Except the above two correlation clusters, there are no other "significant" correlations among the component measures shown in Table 6.5. Suffice it to say here, this indicates that the MCD quantity (**M1** and **M2**) measure of a component

does not substantially affect its MCD complexity (**M3** and **M4**) measure, and vice versa. Next, we describe the degeneration-critical components that are identified in the system according to the above MCD quantity and complexity measures (see Tables 6.3 and 6.4).

Degeneration-Critical Components

Degeneration-critical components are defined as components in a system which contribute substantially more (based on some defined criteria) to the architectural degeneration than other components. We note from (Li et al., 2009) that 20% of the components contain over 80% of MCDs in a large legacy system. These 20% components could be considered as degeneration-critical in that system; and the (top) "20%" is thus the criterion for identification.

In this case study, we refine this well-known 80-20 criterion further with a focus on architectural degeneration, i.e.: degeneration-critical components are those components that satisfy:

(i) their MCD quantity or complexity measures rank in the top 20%, and

(ii) these measures are *substantially* greater than the others.

We subjectively define that a measure is *substantially* greater than the others if it is at least 50% greater[7] than the average value.

With this new criterion, we first identified component C5 (see row "C5" in Table 6.3) as degeneration-critical in release 1 with respect to (w.r.t.) metric "%MCDs" (**M1**), because C5 was involved in 64% of MCDs, which is the greatest measure and is *substantially* greater than the average value (23%). Likewise, C5 is

[7]The reason we choose "at least 50% greater" as a part of this criterion is described below. As in a plot box, any data point in an array that is greater than the upper quartile is usually considered as an *outlier*. Here, such an outlier refers to a degeneration-critical component. This has been considered in aspect (i) of this criterion – "in the top 20%". However, in case that the data distribution is so even that no degeneration-critical components exist, we must add another aspect to the criterion in order to exclude such "outliers" which actually should not be considered as degeneration-critical components. We thus set up aspect (ii) – "at least 50% greater" – which can achieve this purpose.

also degeneration-critical in releases 2 and 3. Similarly, we identified the following degeneration-critical components (see data from Table 6.3): C6 in releases 1 and 2 w.r.t. "%MCDs" (**M1**) (50% and 50%; see row "C6"), C2 in releases 1, 2 and 3 w.r.t. "#MCDs per KSLOC" (**M2**) (1.10, 0.76 and 0.71; see row "C2"), and C7 in release 1 (13.1; see row "C7") and C8 in release 2 w.r.t. "#Code files fixed per MCD" (**M4**) (10.7; see row "C8").

As a partial validation, we showed the findings (components C5 and C6) to the key developers[8] (by interview) who confirmed from their experience that C5 and C6 were the two topmost problematic components in the subject system, and that they were also among the top problematic components for the entire, much larger, system (mentioned in Section 6.2.1). Unfortunately, there is no "process" data or "historical accounts" in the projects logged for this legacy system to be able to analyse possible underlying reasons for the above component measures (see Tables 6.3 and 6.4). The developers were also enthused by the fact that we had developed a systematic approach to identifying a critical set of components compared to their experimental-based approach which is distributed among the project staff and that is erodes with people turnover and memory lapses over time.

We can conclude from the above discussion that there are a few degeneration-critical components in the system. This addresses question **Q1** posed in Section 6.1 – Do some components in a system contribute more than other components to the system's architectural degeneration? We note Section 2.3.3 where defect-prone components are identified based on the Pareto-shaped defect distribution: 25% (1998) or 20% (Boehm and Basili, 2001; Li et al., 2009) of the components that contain many more defects than other components. These defect-prone components that also contain the most of MCDs are considered degeneration-critical from the MCD quantity perspective. The findings of degeneration-critical

[8]Key developers made their responses based on information known to them and logged in restricted databases accessible only to them. We were not privy to such highly sensitive organizational information. We trust their judgment.

(MCD-prone) components described above can complement the previous work on identification of defect-prone components (e.g., (Ohlsson and Wohlin, 1998)). Likewise, some architectural deviations (see Section 2.3.1) that contain many more MCDs than other deviations are considered degeneration-critical. Next, we continue to examine the persistence of these component measures over time.

6.3.2 Persistence of Components' Contributions (Q2)

We note from Tables 6.3 and 6.4 that the average MCD quantity and complexity measures of the components did not change substantially in the three releases. For example (see row "Mean" in Table 6.3), the MCDs pertaining to a component account for an average of 23%, 25% and 23% of all MCDs in the system, and (see row "Mean" in Table 6.4) the number of components fixed per MCD in a component is an average of 2.3, 2.8 and 2.3. Therefore, we can say that the *average* MCD quantity and complexity measures of the components do persist across releases. If the trend of the average values is on the rise then it indicates that the (MCD) fixes are likely consuming more and more resources over time.

More precisely, we use Spearman's rank correlation (Spearman-value)[9] to explore the rank correlations between the component measures across phases and releases; see Table 6.6. For example, the Spearman-value of the MCD percentage measures (see row "**M1**") is 0.57 between the internal and field phases ("Internal→Field"), 0.96 between release 1 and 2 ("r1→r2), and 0.78 between release 2 and 3 ("r2→r3); indicating three large, positive, rank correlations.

We note from Table 6.6 that these measures persist across internal and field phases (see column "Internal → Field") and across releases 1 and 2 (see column "r1

[9]Note that there are 10 data pairs (due to 10 components) for this correlation analysis. The critical value for 10 data pairs based 0.95-significance for two-tailed test of the Spearman-values is 0.73 (see `http://geographyfieldwork.com/SpearmansRank.htm` (last access in November 2010)). In Table 6.6, we can thus find that the Spearman-values 0.96 (for "%MCDs" crossing r1 and r2), 0.78 (for "%MCDs" crossing r2 and r3), and 0.85 (for "#Components fixed per MCD" crossing r1 and r2) are statistically significant. We thus only focus on these three Spearman values in the following discussion.

Table 6.6: Cross-phase/release rank correlations (Spearman-values) between component measures.

	Internal→Field	r1→r2	r2→r3
%MCDs (**M1**)	0.57	0.96	0.78
#MCDs per KSLOC (**M2**)	0.45	0.55	0.64
#Components fixed per MCD (**M3**)	0.46	0.85	-0.01
#Code files fixed per MCD (**M4**)	0.70	0.49	-0.21

→ r2"); the corresponding Spearman-values are relatively large (ranging between 0.45 and 0.70 – ave. 0.55 for the former, and between 0.49 and 0.96 – ave. 0.71). This suggests that the MCD-proneness of these components is "not" affected by the defect correction process.

For example, Figure 6.4 shows the MCD percentage (**M1**) measures (see column "%MCDs (**M1**)" of Table 6.3) of the 10 components (C0–C9) in the three releases (r1, r2 and r3). This figure indicates the persistence of the MCD percentage measures of each component across the three releases (especially crossing r1 and r2). This coincides with the Spearman-values (0.96 and 0.78) shown in row "%MCDs (**M1**)" of Table 6.6. Similar charts can be created for the "#Components fixed per MCD" measures crossing r1 and r2.

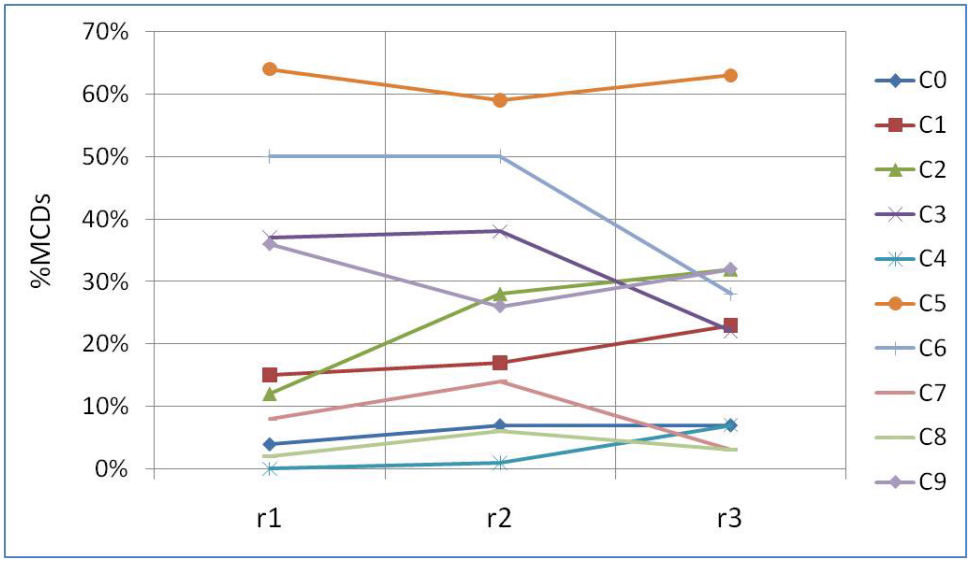


Figure 6.4: Components' MCD percentage measures across releases.

Further, Figure 6.5 illustrates the ranks of the components' MCD percentage ("%MCDs") measures across releases 1 and 2. This figure shows a near-linear curve which indicates that there is a strong rank correlation (with Spearman-value = 0.96; see row "%MCDs (**M1**)" of Table 6.6) between the components' MCD proportions pertaining to the releases 1 and 2. This strengthens the findings (see Figure 6.4 and Table 6.6: the components' MCD quantity and complexity measures do persist across phases and releases.

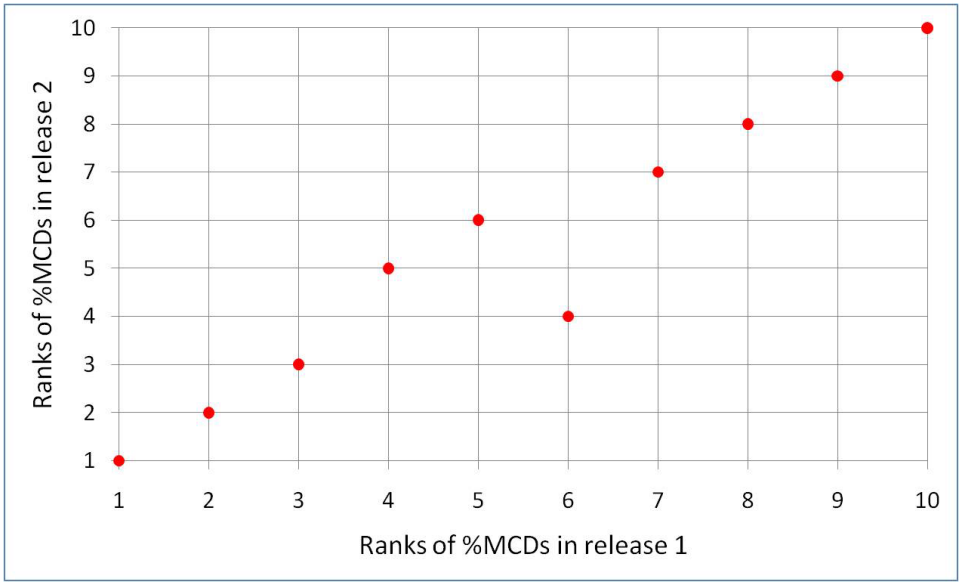


Figure 6.5: Persistence of components' "%MCDs" measures across releases.

We can conclude that the components' contributions to architectural degeneration do persist across interval and field phases and across releases 1 and 2, from the perspectives of MCD quantity and complexity. This addresses question **Q2** posed in Section 6.1 – Do components contribute persistently to architectural degeneration over development phases and releases?

We note from Section 2.4.4 that defect-prone components (DPCs) tend to persist across system development phases and releases (Compton and Withrow, 1990; Ohlsson et al., 1999). This is similar to the persistence of components (MCD quantity and complexity measures thereof) in relation to architectural degenera-

tion shown above. Overall, the above component measures (Tables 6.3 and 6.4) and their persistence analysis (Table 6.6 and Figures 6.4 and 6.5) help in building a quantitative profile of the components (and the complexity of their fixes) in relation to architectural degeneration. Next, we do analogous measurements and persistence analysis for the fix relationships in the system.

6.3.3 Fix Relationships' Contributions (Q3)

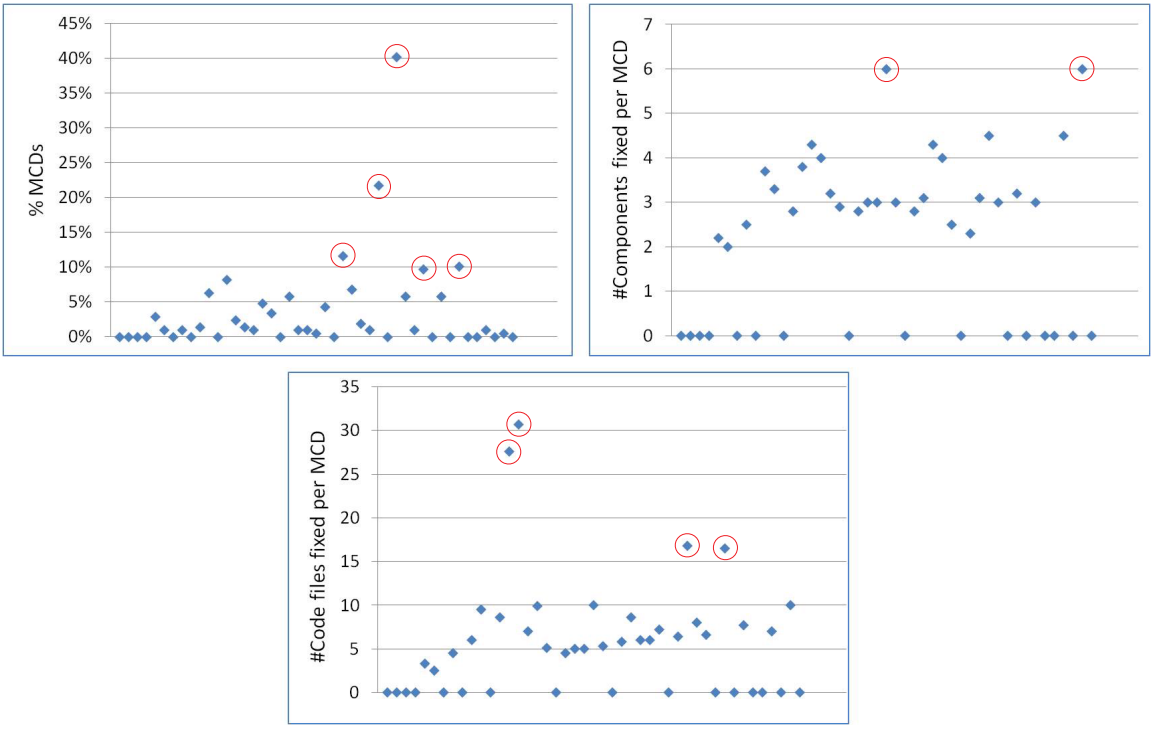
We first describe the MCD quantity and complexity measures of the fix relationships, and then discuss the persistence of these measures across phases and releases. Recall that these measures deal with defects concerning inter-component relationships. Essentially, the more they spread through a system, and the more they persist in components, and across phase and release boundaries, the more the architecture degenerates over time.

We note that there are at most 45 (binary)[10] fix relationships in the system. As an example, Figure 6.6 shows the MCD quantity ("%MCDs" (**M1**) – see the top-left chart) and complexity ("#Components fixed per MCD" (**M3**) and "#Code files fixed per MCD" (**M4**) – see the top-right and bottom charts, respectively) measures of these 45 fix relationships[11] in release 1. Similar charts can be created for the remaining releases 2 and 3.

Figure 6.6 indicates that the three distributions of these measures are skewed. For example, there are five fix relationships (see the red-coloured circles in the top-left chart) which are involved in a greater proportion of MCDs than are other fix relationships. In particular, we note that over 70% of MCDs involve at least one of these five (approximately 10%) fix relationships. This finding coincides with the earlier finding (Li et al., 2009) where nearly 75% of MCDs involved 10% of the fix relationships. Likewise, we find two fix relationships (circled in the

[10]Fix relationships are undirected and there are only 10 components in the system, we can infer that there are at most 45 (i.e., 10×(10-1)/2) fix relationships.

[11]If a fix relationship does not exist in the system, its measures are set to 0.



Note: in each chart, a "diamond" represents a fix relationship, and its index in the vertical axis indicates the measure ("%MCDs" (**M1**), "#Components fixed per MCD" (**M3**), or "#Code files fixed per MCD" (**M4**) in the top-left, top-right or bottom charts, respectively). A "circled" diamond represents a degeneration-critical fix relationship which has "substantially" greater values than other fix relationships (denoted by the non-circled diamonds).

Figure 6.6: Fix relationships' measures in release 1.

top-right chart) that have the greatest "#Components fixed per MCD" (**M3**) measures, and four fix relationships (circled in the bottom chart) that have the greatest "#Code file fixed per MCD" (**M4**) measures.

With a criterion similar to that defined for degeneration-critical component identification (see Section 6.3.1), these 11 (i.e., 5 + 2 + 4 = 11; see Figure 6.6), "circled", fix relationships are considered *degeneration-critical* in the subject system – because they contribute substantially more to architectural degeneration than do other fix relationships. This addresses question **Q3** posed in Section 6.1 – Do some fix relationships in a system contribute more than other fix relation-

ships to the system's architectural degeneration? It also complements the finding from Section 6.3.1: there are a few degeneration-critical components in the system. These degeneration-critical fix relationships should be treated as important as degeneration-critical components in the system.

In the literature, we note (in Section 2.3.3) that von Mayrhauser et al. (2000) propose the fault architecture construction approach to highlight some fix relationships among system components which involve substantially more MCDs than other fix relationships. The work described above extended von Mayrhauser et al.'s approach to measure fix relationships with the MCD quantity and complexity metrics (as defined in the step 2 of the DAD approach – Section 5.2.2).

Further, we note that over half (e.g., 6/11 in release 1) of these degeneration-critical fix relationships are *connected* by the degeneration-critical components (see Section 6.3.1) in the system. Due to lack of detailed data, we cannot investigate the underlying reasons for this inter-relationship between degeneration-critical components and fix relationships. We will continue the discussion on this issue in Section 6.5.3. Suffice it to say here, this finding (about the inter-relationship) can aid in the design of degeneration treatment procedures.

6.3.4 Persistence of Fix Relationships' Contributions (Q4)

Table 6.7 illustrates the mean and standard deviation (in brackets) values of the "%MCDs" (**M1**), "#Components fixed per MCD" (**M3**), and "#Code files fixed per MCD" (**M4**) measures for the fix relationships in the three releases (r1, r2 and r3). For example, the value "4% (7%)" means that each fix relationship is involved in, on average, 4% (with standard deviation 7%) of MCDs in release 1.

We note that (see Table 6.7) these average "%MCDs" (**M1**), "#Components fixed per MCD" (**M3**), and "#Code files fixed per MCD" (**M4**) measures of fix relationships increase from release 1 to release 2 and later decrease in release 3. We will explain this later in relation to architectural degeneration in Sec-

Table 6.7: Means and standard deviations (in brackets) of fix-relationships' measures.

	r1	r2	r3
%MCDs (**M1**)	4% (7%)	5% (6%)	3% (6%)
#Components fixed per MCD (**M3**)	2.2 (1.8)	3.2 (2.6)	1.5 (1.5)
#Code files fixed per MCD (**M4**)	5.7 (6.7)	10.5 (10.2)	2.8 (2.9)

tion 6.3.5. Simply put, this indicates that the (MCD) fixes are likely consuming more resources as the system evolves across releases 1 and 2; whereas that resource consumption decreases in release 3.

We also note that there is a large, positive, correlation between the "#Components fixed per MCD" (**M3**) and "#Code files fixed per MCD" (**M4**) measures for fix relationships; the corresponding average Pearson-value is 0.87 across the three releases. However, there are no significant correlations between the MCD percentage ("%MCDs" or **M1**) and complexity (**M3** and **M4**) measures; the corresponding average Pearson-values are less than 0.25. These findings coincide with the similar findings for the components (see Section 6.3.1), i.e.: fixing a MCD, a fix relationship exhibiting a high number of inter-component changes tends to exhibit a high number of inter-code-file changes.

We note from Table 6.7 that the average MCD percentage (**M1**) and complexity (**M3** and **M4**) measures of fix relationships persist across releases. We further investigate the rank correlations (Spearman-values)[12] of these measures across releases ("r1 → r2" and "r2 → r3"), see Table 6.8. For example, the cell value "0.51" indicates a medium, positive, rank correlation between the "%MCDs" (**M1**) measures of releases 1 and 2.

We note from Table 6.8 that most of the Spearman-values do not indicate

[12]Note that there are 45 data pairs for the Spearman correlation analysis hereon. The critical value for 45 data pairs based 0.95-significance for two-tailed test of the Spearman-values is less than 0.35 (see `http://geographyfieldwork.com/SpearmansRank.htm` (last access in November 2010)). In Table 6.7 we can thus find that the Spearman-values 0.51 ("%MCD" crossing r1 and r2), 0.78 ("%MCD" crossing r2 and r3), 0.44 ("#Components fixed per MCD" crossing r1 and r2), and 0.35 ("#Code files fixed per MCD" crossing r1 and r2) are statistically significant. We thus focus only on these values in the later discussion.

Table 6.8: Cross-release rank correlations (Spearman-values) between fix relationships' measures.

	r1 → r2	r2 → r3
%MCDs (**M1**)	0.51	0.78
#Components fixed per MCD (**M3**)	0.44	0.02
#Code files fixed per MCD (**M4**)	0.35	0.04

large, positive, rank correlations. We thus infer that the across-release persistence of the fix relationships' contributions to the architectural degeneration is *weak*, suggesting that the MCD-proneness of these fix relationships is affected by the defect correction process. This addresses question **Q4** posed in Section 6.1: Do fix relationships contribute persistently to architectural degeneration over development phases and releases? However, note that the above finding of weak persistence of the fix relationships is in contrast to the strong persistence of the components across phases and releases (see Section 6.3.2).

Further, Figure 6.7 illustrates the ranks of the fix relationships' MCD percentage ("%MCDs") measures across releases 1 and 2. This figure shows a curve which indicates that there is a weak rank correlation (with Spearman-value = 0.51; see row "%MCDs (**M1**)" of Table 6.6) between the fix relationships' MCD percentages pertaining to the releases 1 and 2. This strengthens the findings (see Table 6.8: the fix relationships' MCD quantity and complexity measures do not persist across phases and releases.

We note that the fault architecture construction approach by von Mayrhauser et al. (2000) (see Section 2.3.3) can be used to evaluate the persistence of (most frequently occurred) fix relationships across multiple fault architectures. However, this is related only to the MCD quantity perspective as described above. The above identification of degeneration-critical fix relationships extended the fault architectural construction approach with the MCD complexity perspective. The degeneration-critical fix relationships are shown in defect architectures (one of the

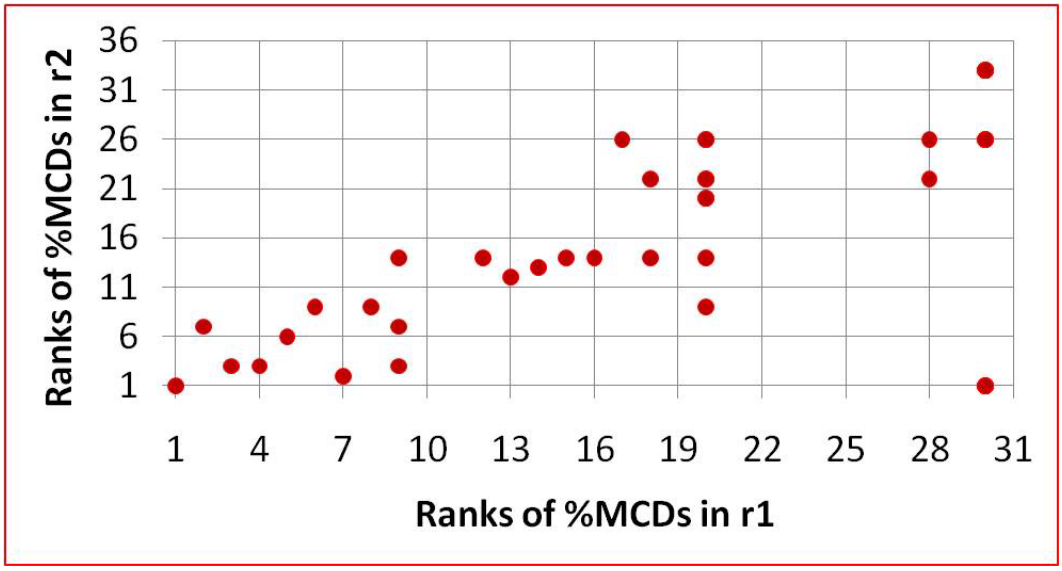


Figure 6.7: Persistence of fix relationships' "%MCDs" measures across releases.

main outputs of the DAD prototype tool), see Figure 1.1 as an example; other examples are shown in Section B.3.

Overall, the above measures and persistence analysis help in building a profile of fix relationships in the system, and complement the profile of components built in Sections 6.3.1 and 6.3.2. These two profiles together describe characteristics of architectural degeneration in the subject system over time. Next, we derive and discuss the architectural degeneration trend of the system based on these profiles of components and fix relationships.

6.3.5 Architectural Degeneration Trend (Q5)

In Sections 6.3.1 and 6.3.3, we described release-specific measures and information for components and fix relationships but deferred trend-analysis of the system's architecture till this subsection. Based on these specific measures, we analyze the trend of architecture degeneration across the three releases.

First of all, we note from Table 6.3 (row "Mean") that the average MCD percentage of the components is 23% in release 1, then increases to 25% in release 2, and finally decreases to 22% in release 3. This indicates that, from the MCD

percentage perspective, the architectural degeneration of the system first increased in release 2 but then decreased in release 3. This "increase-then-decrease" trend is confirmed by the components' MCD complexity measures across the three releases (see Table 6.4). It is also confirmed analogously based on the fix relationships' MCD quantity and complexity measures; see in Table 6.7, the data across the three releases for metrics "%MCDs" (**M1**), "#Components fixed per MCD" (**M3**), and "#Code files fixed per MCD" (**M4**). Recall Figure 1.1 where this trend is also demonstrated. However, this "increase-then-decrease" trend differs from the "increase-only" trend depicted in the AT&T 5ESS case (Eick et al., 2001) discussed in Section 1.1 which, though, is interesting, was surprising as we had naively expected the trend to be "increase-only" in our study as well. Nothing is the dataset and analysis done suggested that the interpretation should be any different than it actually was.

Concurrent to this, Belady and Lehman's *laws of software evolution* (1976; 1980), from 1974, on Increasing Complexity suggests that "as an 'E-type' system evolves its complexity increases unless work is done to maintain or reduce it." Though Belady-Lehman's law does not focus on architectural degeneration, we can draw parallels with it on the assumption that the architecture is likely to degenerate as the system gets more and more complex over time. The trailing part of Belady-Lehman's law ("... unless work is done to maintain or reduce it"), however, supported the idea to present the findings to the developers who immediately, if enthusiastically, attributed this improvement in architecture quality between releases r2 and r3 to "system restructuring" that they had carried out prior to release r3. Among the restructuring process included making the components more cohesive while reducing the coupling amongst the components. For example, this meant grouping related files together to minimize cross-component function calls in release 3. This explains the "increase-then-decrease" trend in the case study system's evolution.

This trend supports Lindvall et al.'s conclusion (Lindvall et al., 2002) that restructuring can decrease architectural degeneration. An interesting difference, however, between our work and Lindvall et al.'s is that whereas they analyzed system architectures based on "deviations" (inter-component interactions) we analyzed system architectures based on quantity and complexity of MCDs. This section addresses question **Q5** posed in Section 6.1: What is the trend in architectural degeneration from the defect perspective?

6.3.6 Defect Architectures

Figure 6.8 illustrates the defect architecture (segment) of release 1 of the subject system with respect to the MCD percentage metric – "%MCDs" (**M1**). It describes the top 2 components (red-coloured nodes) and the top 10 fix relationships which have the greatest **M1** measures (shown in the labels) in release 1. The blue-coloured nodes are shown in the figure because they are connected with these fix relationships. The numeric labeles on each node or edge in Figure 6.8 indicates the "%MCDs" value of the component or fix relationship in the system.

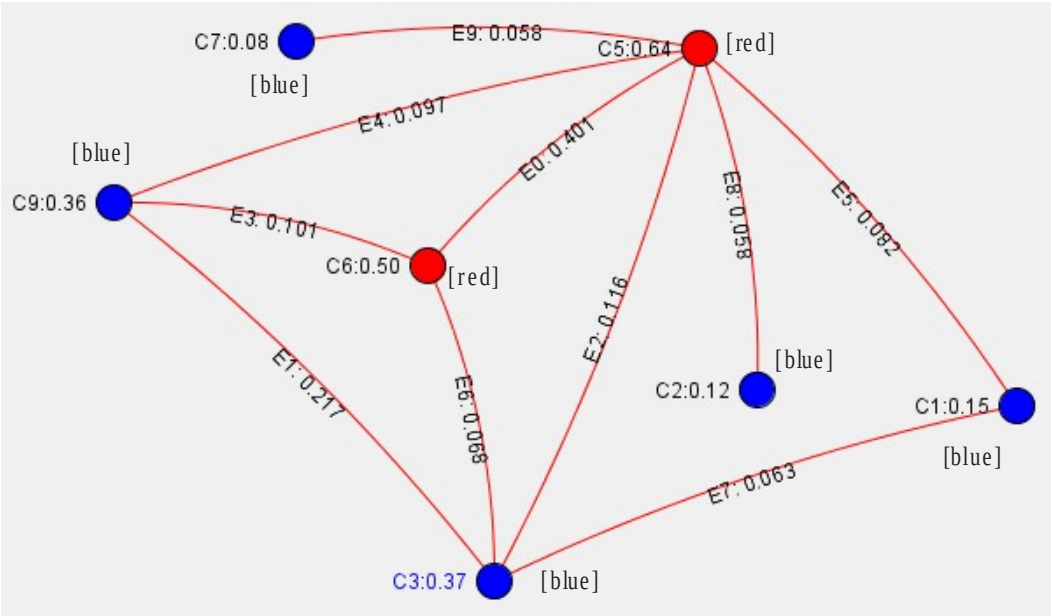


Figure 6.8: Defect architecture (segment) of release 1 with metric **M1**.

As shown in Figure 6.8, there are 64% of MCDs in release 1 that pertained to component C5, there are 50% of MCDs that pertained to component C6 (coinciding with the finding shown in Table 6.3), and there are 40.1% of MCDs that involved the fix relationship between C5 and C6. Moreover, the two "red-coloured" components, C5 and C6, should be considered degeneration-critical as their MCD-percentage (**M1**) measures are obviously greater than that for other components (see blue-coloured nodes). This coincides with the findings of the degeneration-critical components in Section 6.3.1.

Similar to above Figure 6.8, Figure 6.9 illustrates the defect architecture (segment) of release 1 of the subject system with respect to metric **M3** – MCD complexity (i.e., #Components fixed per MCD). It shows the top 2 components (red-coloured nodes) and the top 10 fix relationships which have the greatest **M3** measures (shown in the labels) in release 1.

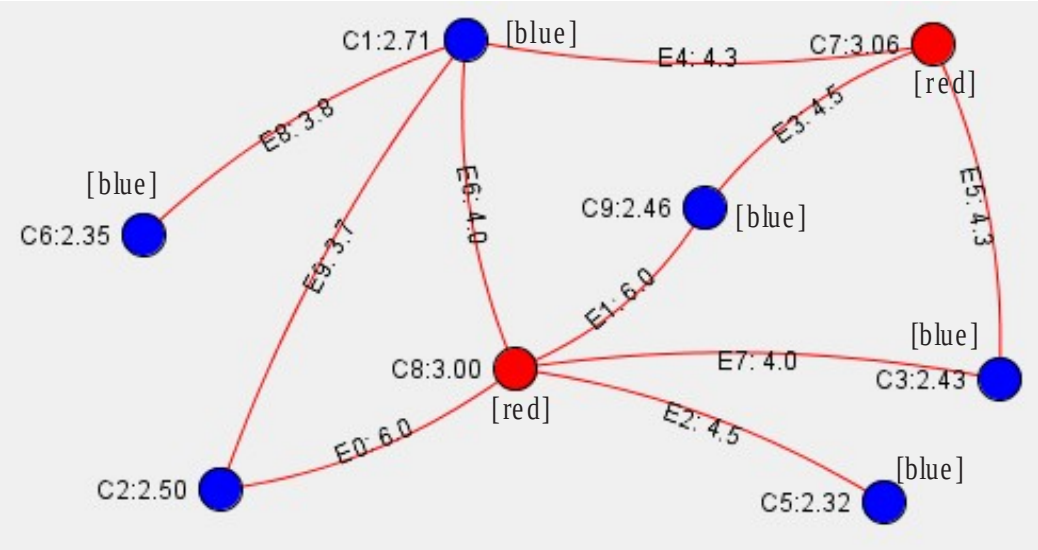


Figure 6.9: Defect architecture (segment) of release 1 with metric **M3**.

In Figure 6.9, the two "red-coloured" components, C7 and C8, should be considered degeneration-critical due to their, substantially great, **M3** measures. This coincides with the findings in Section 6.3.1.

Both Figures 6.8 and 6.9 indicate that the degeneration-critical components

(red-coloured nodes) are connected with degeneration-critical fix relationships than other components (blue-coloured nodes) in the system. In Figure 6.8, the two degeneration-critical components, C5 and C6, have 6 and 3 fix relationships, respectively. However, the other components (the remaining 5 components in the figure) have an average of only 2.2 fix relationships. Also, in Figure 6.9, the two degeneration-critical components, C8 and C7, have 5 and 3 fix relationships, respectively; but the other components (the remaining 6 components in the figure) have an average of only 2 fix relationships. This indicates a strong correlation – the degeneration-critical components tend to correlate with the degeneration-critical fix relationships; see Section 6.3.3.

In addition, Figure 1.1 (in Chapter 1) also describes three defect architectures of the system (with 50 randomly chosen MCDs for each of the three releases), where the thickness of the "boxes" or the "edges" represents the number of MCDs involved in the components and the fix relationships between the components.

6.3.7 Summary of Findings

The key findings which address the five questions, **Q1**–**Q5**, posed in Section 6.1, are summarized below.

(1) There are 20% of the components which contain over 80% of MCDs (see Table 6.3). There are also few components (see Table 6.4) which are associated to relatively more complex MCDs. These components are *degeneration-critical*. This addresses question **Q1**.

(2) The MCD quantity and complexity measures of components persist across development phases and releases (the Spearman-values are, on average, 0.55 for across phases, and 0.71 for across release – see Table 6.6). This indicates that the components tend to persistently contribute to architectural degeneration, which addresses question **Q2**.

(3) There are 10% of the fix relationships which contain over 70% of MCDs (see

Figure 6.6). There are also few fix relationships (see Figure 6.6) which are associated to relatively more complex MCDs. These fix relationships are *degeneration-critical*. This addresses question **Q3**.

(4) The MCD quantity and complexity measures of fix relationships do not persist across phases and releases (most of the Spearman-values are less than 0.5 – Table 6.8). This indicates that the fix relationships do not persistently contribute to architectural degeneration, which addresses question **Q4**.

(5) The average MCD quantity and complexity measures of components increased by approximately 10% (see Table 6.3) and 25% (see Table 6.4) as the system evolved from release 1 to release 2. Likewise, approximately 25% and 50% for fix relationships' measures (see Table 6.7). However, the same measures of components and fix relationships decreased by even higher rate from release 2 to release 3. This indicates an "increase-then-decrease" trend in architectural degeneration of the system, which addresses question **Q5**.

In addition to the above summary of findings, we also note that the MCD complexity measures for components and fix relationships are largely correlated to each other (Pearson-values are close to 1), but there are no substantial correlations between MCD quantity and complexity measures (see Sections 6.3.1 and 6.3.4).

Overall, these findings are new and can be said to add substantially to the body of knowledge on architectural degeneration. These findings also have implications for both software evolution and quality improvement; see Section 6.5.

6.4 Threats to Validity

This section discusses the main threats to the validity of the main findings of this case study (see Section 6.3). We classify these threats into four groups: data quality, and external, construct and conclusion validity.

6.4.1 Data Quality

As described in Section 6.2.1, the dataset examined in this case study is the defect-fix history of three major releases (each of which are six, five and three years old respectively) of an over-13-year-old system. With legacy data, there is a threat regarding the accuracy and completeness of the dataset. For example, could it be possible that some defects were fixed but were not recorded in the defect-tracking database? Likewise, were some changes made in the code base but not recorded in the version control system? And, were some changes recorded in the version control system but not "correct" (can really fix the defect)? In this case study, the developers conveyed their high degree of confidence in the completeness of the dataset, but there is no simple way to independently confirm this. However, from the analysis of the defect-fix dataset we observe that there were no obvious spatial or time gaps in the defect records and in the change logs.

Painfully, we got access to defect data over two years of attempts; there is no "process" data or "historical accounts" in the projects logged for this legacy system to be able to analyse possible underlying reasons for the findings presented in Section 6.3. Staff turnover means that developer memory is faint on such issues and unreliable to include in a scientific study. Yet, the findings are interesting that we think the software engineering community should know about.

6.4.2 External Validity

External validity is concerned with the generalization of the findings of the study in other contexts (Creswell, 2002). The subject system is a (sub)system of an even large legacy system; and the dataset under study is the defect-fix history (defect records and change logs) of three major, successive, releases of this system. For many systems, defect records can be gathered from defect-tracking systems, and change logs can be collected from version control systems. However, the way changes are matched with defects in a software project can be specific to the

organization, people, tools, processes, controls, etc. For example, we noted that several open-source systems do not involve any defect information in change logs, in which case, the change logs cannot be matched with defect records. Without such matches between defects and changes, the "Component*" (indicating the component fixed for a specific defect; see Section 6.2.1) value cannot be determined reliably for each defect. Thus the MCDs cannot be identified from the defect set. This implies that the detailed steps we used in the study (see Step 2 in Section 6.2.5) in identifying MCDs are not easily generalizable to other contexts.

Also, note that the findings of this case study about the components' persistence in the MCD quantity and complexity measures (see Section 6.3.2) and the architectural degeneration trend ("increase-then-decrease"; see Section 6.3.5) should be carefully analyzed when generalizing to other contexts.

6.4.3 Construct Validity

Construct validity is concerned with the actual data gathered in the study and its relationship with the measures defined (Wohlin et al., 2000). In Section 6.2.1, the "complexity" of a defect is measured by the number of components or code files changed in order to fix this defect. There could be other definitions for defect complexity, such as the number of source lines of code that were added, deleted or modified, or the amount of calendar time spent in fixing a defect. In the case study, we chose to use component and file change measures because this was the data that was captured in the logs. Fine-grained measures of code, and effort data, are not logged so such measures could not be incorporated in the analysis of data. Thus, in the case study, the validity of the results has to be viewed from coarse-grained component and file -based complexity measures.

Another issue is the criterion defined for identifying degeneration-critical components (see Section 6.3.1). This criterion is context sensitive (e.g., the component measure distribution and the threshold of "top 20%" chosen by management) and

so using a different threshold could affect the answer to question **Q1** posed in Section 6.2.1 (i.e.: Do some components in a system contribute more than other components to the system's architectural degeneration?). Similar logic applies to the criterion for fix relationships (see Section 6.3.3).

6.4.4 Conclusion Validity

Conclusion validity is concerned the extent to which the conclusions made in the study are reasonable (Wohlin et al., 2000). For this study, we concluded that the components' measures tend to persist over time (see Section 6.3.2), and the architectural degeneration increased as the system evolved (see Section 6.3.5). While the conclusions are traceable to the analysis of the data in the study, we note that we had access to only three releases of the system. Especially, there are three issues to note.

First, not all of the MCD quantity and complexity measures of the components support the architectural degeneration increase trend. For example, the MCD density ("#MCDs per KSLOC" or **M2**) (see row "Mean" in Table 6.3) does not support this trend (see Section 6.3.2).

Second, the four types of the components' measures show varying (not the same) persistence. Table 6.6 shows that the cross-phase/release rank correlations between the components' measures rang between -0.21 and 0.96, indicating great variance. For example, the MCD percentage ("%MCDs" or **M1**) and complexity ("#Components fixed per MCD" or **M3**) measures strongly persist across releases 1 and 2; their Spearman-values are 0.96 and 0.85. However, the other two measures ("#MCDs per KSLOC" (**M2**) and "#Code files fixed per MCD" (**M4**)) exhibit only weak persistence; their Spearman-values are 0.55 and 0.49 (see column "r1 → r2" in Table 6.6).

Third, there was restructuring work carried out on release 3 in order to improve the system structure (see Section 6.2.1), and we found that the restructuring

has substantial impact on the components' persistence in their measures and on the architectural degeneration increase trend (see Section 6.3.5). This thus does not support our conclusion that the architectural degeneration increased as the system evolved.

Considering these three issues, we recommend careful context analysis in the use of the conclusions.

6.5 Implications

This section discusses the implications of the case study findings from the points of views of methods, priority re-engineering, and empirical research.

6.5.1 Methods of Architectural Degeneration Analysis

We note that two of the 10 components of the subject system (e.g., components C5 and C6) are involved in most of the MCDs in the system (see Table 6.3). The 80-20 Pareto rule is well-known in software quality (Boehm and Basili, 2001), where identifying the top 20% defective components is relatively straightforward. Unfortunately, this would not suffice for the analysis of architecture degeneration because it entails more than frequency counting of component defects. In Sections 6.3.1 and 6.3.3, we saw that, in the subject system: (1) there are a few degeneration-critical components and fix relationship which contribute substantially more to the architectural degeneration than other components and fix relationships; and (2) over half of degeneration-critical fix relationships are connected by the degeneration-critical components.

Identification of these "hard core", inter-related, components requires analysis of MCDs, filtering out high density components, mapping out fix relationships, and filtering out "hard core" architectural areas that are degenerating at a faster pace than other architectural areas. Because different software organizations log defect and change data in organization-specific ways and databases

(e.g.: Bugzilla[13], Bugzero[14], and Bugs-Everywhere[15] for software defect tracking and management), it should be possible to create suitable high-level, defect and architectural analysis methods (if not concrete software tools) (e.g., ADD (Bass et al., 2003, ch. 7), ATAM (Clements and Northrop) and SAAM (Kazman et al., 2002); also see the survey in (Dobrica and Niemela, 2002)) with guidelines so that software developing organizations can internalize and customize these methods in their software projects. This would help transition research-oriented case-studies (such as the one described here) to everyday practice.

6.5.2 Priority Re-engineering of System Components

Most values in Table 6.6 show that components contribute persistently to architectural degeneration across internal and field phases and across releases. Creating such a persistence profile can help in determining: (a) whether or not a specific component should be re-engineered (or restructured), and (b) which specific components should be re-engineered with priority. Although re-engineering is costly, priority re-engineering could focus on the most-problematic components in the system (Booch, 2008). Also, this analysis enriches the current re-engineering strategies (e.g., splitting over-large components and reducing inter-component interactions (Tran and Holt, 1999)).

6.5.3 Empirical Research

Based on the study findings, we raise the following two investigative questions that seek to probe deeper into the phenomena observed from this study.

(1) *To what extent are degeneration-critical components and fix relationships inter-related?*

[13]See `http://www.bugzilla.org/` (last access in November 2010).
[14]See `http://www.websina.com/bugzero/` (last access in November 2010).
[15]See `http://bugseverywhere.org/be/show/HomePage` (last access in November 2010).

In Section 6.3.4, we qualitatively observed that over half of the degeneration-critical fix relationships were connected by degeneration-critical components. That is, degeneration-critical components often occur together in terms of MCDs. Despite this potentially interesting finding, we could not scientifically determine the extent to which degeneration-critical components are related to degeneration-critical fix relationships. This was due to the relatively small sample of components investigated in this study; the ten components examined (see Section 6.2.1) were more than adequate for answering the case study research questions, but were too low to provide the statistical power to adequately probe this particular result further. Knowing this relationship would provide critical input into the priority re-engineering process (see above). For example, this would further help isolate an even smaller area of the system for priority re-engineering if it is indeed confirmed that most MCDs are contained within degeneration-critical components that have associated fix relationships, as suggested by the initial finding (see Section 6.3.3).

(2) *What are the underlying reasons for the impact of a defect correction process on the persistence of the components and fix relationships?*

In Section 4.3.2, we observed that fix relationships have "weak" persistence across phases and releases. Similarly, in Section 6.3.2, we observed that components have "strong" persistence. However, in the study, we could not probe deeper into these findings due to a lack of access to the software architecture itself, and its evolutionary data. Recall from Section 6.2.1 that we only had access to the defect and change data, both of which were adequate for investigating the original research questions. The data that is specifically needed is the documentation of the physical interactions between the software components, and how these interactions change over time. Given this data, we could then associate the physical architectural changes with the defect correction process, leading to the underlying technical reasons why the persistence occurred. This characterization of root-cause persistence issues could provide an empirical-based groundwork

for developing further technological support to aid in the defect correction and architectural degeneration treatment processes.

6.6 Recap of Case Study 2

Previous research has far focused on architectural deviations from the baseline as a vehicle to study architectural degeneration. We complement this by focusing on the quantity of MCDs, the spread of changes across the system's components, and fix relationships, and the persistence of these across several releases – as a way to characterize architectural degeneration. In particular, we conducted a case study which followed the DAD approach (see Section 5.2) to characterize the architectural degeneration of a commercial legacy system (of size over 1.5 million SLOC and age over 13 years) using specific metrics.

The main findings of this case study in descriptive terms are: (1) there exist a small set of components and fix relationships that contain most of the MCDs (see Sections 6.3.1 and 6.3.3); (2) the system's components tend to persistently have an impact on architectural degeneration over multiple releases of the system; however, such persistence does not apply to the fix relationships (see Sections 6.3.2 and 6.3.4); and lastly (3) architectural degeneration increases as the system evolves over time; but restructuring can reverse this trend (see Section 6.3.5). A succinct summary of the results is shown in Section 6.3.7. These findings are new and add to the body of knowledge on architectural degeneration. The existing knowledge reflects the state of current theory on architectural degeneration, i.e., the degeneration of an architecture can be determined based on its deviations made against a baseline. This case study (plus Case Study 1) adds that architectural degeneration can be characterized and diagnosed through analysis of MCDs.

Implications of these findings are on: determining degeneration-critical components and fix relationships, priority re-engineering, and empirical research (see Section 6.5). Overall, we conclude that architectural degeneration can be charac-

terized from the defect perspective, which complements the characterization from the structural or deviation perspective (Lindvall et al., 2002).

While these findings are new, one should not overlook that these results are from only one, albeit significant, case study. This study has its own idiosyncratic threats, e.g., defect complexity measure and metric correlation (see Section 6.4). There is thus a need to conduct replicated studies involving other systems in order to (i) validate the DAD approach and (ii) build a body of knowledge on architectural degeneration, components and fix relationships. Later Chapter 9 describes some of the challenges involved and lessons learnt.

Chapter 7

Case Study 1 vs. Case Study 2

Following the description of Case Studies 1 and 2 in Chapters 4 and 6, this chapter compares these two case studies, centered on handling and using multiple-component defects (MCDs). For the comparison, we first note that Case Study 1 was to examine the extent to which architectural degeneration affected software defects, by comparing the complexity and persistence of MCDs against that of other types of defects. It investigates the historical defect records of six major releases of a large legacy software system (see Section 4.3.1). We then note that Case Study 2 had achieved its purpose: application and validation of the DAD approach in a legacy system (which is a part of that large system investigated in Case Study 1). It investigates the defect-fix history (defect records and change logs) of three major, successive, releases of the subject system (see Section 6.2.1).

In the three sections below, 7.1–7.3, we compare the two case studies from three aspects in the subject systems respectively: their MCD identifications, MCD distributions, and MCD complexity measures.

7.1 MCD Identification

We note from Section 6.2.2 that the MCD identification methods are different for the two case studies. They are described below.

- In Case Study 1, each defect record has a "Component" field value (see example defect records in Table 4.1). The parent-children relationships are identified among the defects (as defined in Section 4.2). If a parent defects and its children defects have different "Component" field values, they are identified as MCDs.
- In Case Study 2, each defect-fix record has a "Component*" field value (see example defect-fix records in Table 6.1). A defect having more than one component value recorded in its "Component*" field is identified as a MCD (see Section 6.2.2).

We note from the defect-fix database of Case Study 2 (see Section 6.2.1) that a defect discovered in a component always had one or more fixes in this component. That is, the "Component" value of a defect is always involved in its "Component*" value. Because MCDs are identified based on the "Component" field (as in Case Study 1), we can infer that the set of MCDs identified based on parent-children relationships is included in the set of MCDs identified based on the "Component*" field (as in Case Study 2). In particular, about 8% of defects are identified as MCDs in Case Study 1 (see Table 4.3), but about 18% in Case Study 2 (see Section 6.2.4).

7.2 MCD Distribution

Both Case Studies 1 and 2 investigate the distributions of MCDs in the subject systems. The related findings are described below.

- In Case Study 1, the Pareto principle fits the MCD distribution by components. Figure 4.1 shows that over 80% of MCDs emanate from 20% of the components and about 75% of MCDs involve only 10% of the fix relationships. These 20% of components and 10% of fix relationships should be considered *degeneration-critical*.

- In Case Study 2, the MCD distribution by components is highly tailed. Across the three releases under investigation, the two (out of 10) components, C5 and C6, contain 62% and 43% of MCDs[1] in the system (see column "%MCDs (**M1**)" of Table 6.3). Meanwhile, only a few (5 out of 45, i.e., about 10%) fix relationships are involved in over 10% of MCDs in the system and the majority of MCDs involve at least one of these 10% fix relationships[2] (see Figure 6.6). These two components and a few fix relationships are thus considered degeneration-critical in the system from the MCD percentage perspective.

Therefore, both Case Studies 1 and 2 support that the majority (over 80%) of MCDs are concentrated in a few (about 20% of) components and are involved in a few (about 10% of) fix relationships in the systems.

7.3 MCD Complexity Measurement

We note that both Case Studies 1 and 2 measured the complexity of MCDs. We describe their MCD complexity measures below.

- In Case Study 1, the complexity of a MCD is measured by the number of accompanying changes (as defined in Section 4.2). Table 4.4 indicates that a MCD required an average of 2.1 accompanying changes, indicating 3.1 (2.1+1) changes[3] for a MCD, which is about 2.8 times as much as that for a non-MCD (requiring an average of 1.1 changes).
- In Case Study 2, the complexity of a MCD is measured by the number of components or code files changed in order to fix this MCD (see example metrics "#Components fixed per MCD" (**M3**) and "#Code files fixed per

[1]It is obvious that these 62% and 43% of MCDs are overlapped partially.
[2]This is derived by summing up the MCDs involving these 10% fix relationships.
[3]Note that in Case Study 1, the number of changes is counted based purely on defect records (no change logs). Therefore, as defined in Section 4.2, each non-parent defect has one and only one change. This is different from that in Case Study 2.

MCD" (**M4**) in Section 5.2.2). Table 6.4 indicates that a MCD required an average of 5.5 changes (in an average of 2.3 components) which is about 2.5 times as much as that for a defect (a MCD or not a MCD, requiring an average of only 2.2 changes in 1.2 components).

The above findings (e.g., 2.8 vs. 2.5) indicate that there is no substantial difference[4] in the complexity of the MCDs identified in Case Studies 1 and 2. Therefore, we can further infer that (note that MCDs account for an average of 8% in Case Study 1 but an average of 18% in Case Study 2; see Section 7.1):

(1) For Case Study 1, there are about 10% of defects in the subject system which are not identified as MCDs but have similar complexity measures with the MCDs identified with parent-children relationships.

(2) For Case Study 2, there are about 8% of defects (i.e., about 45% of MCDs) in the subject system which have children defects spanning multiple components and have stronger persistence (see Table 4.5) across development phases and releases than other defects.

(3) The subset of MCDs identified based purely on defect dataset can be used to characterize (in terms of at least their complexity) the whole set of MCDs existing in the system. This indicates an alternative way to investigate MCDs when only defect records are under investigation.

7.4 Further Comparative Analysis

We conclude from the above comparison between Case Studies 1 and 2 that although the MCD identification methods are different (see Section 7.1), both case studies support that MCDs are highly concentrated in a few (about 20% of) components and (about 10% of) fix relationships in the systems (see Section 7.2).

[4]Here, we cannot conduct a statistical significance test for this "2.8 vs. 2.5" comparison because the sizes of the two comparing arrays are obviously not equivalent. However, from the practical perspective, we can declare that there is no substantial difference between 2.8 and 2.5 as the measures of MCD complexity.

Especially, the complexity measures of the MCDs identified with the different methods in the two case studies are similar (see Section 7.3).

The comparison between Case Studies 1 and 2 indicates that the subset of MCDs identified based purely on defect dataset can represent the whole set of MCDs in the system (identified based on defect-fix dataset), in terms of at least their distribution and complexity characteristics. This indicates an alternative way to investigate MCDs when only defect records are under investigation, and that the two case studies are complementary with each other.

Except the differences in MCD identification, distribution, and complexity measurement (see Sections 7.1–7.3), we also note two essential differences between these two studies. First, the research goals are different. Case Study 1 aims at examining MCDs (their complexity and persistence) in order to understand architectural degeneration, but Case Study 2 aims at application of the DAD approach (see Chapter 5) and characterization of architectural degeneration. Therefore, their research questions are different (see Sections 4.1 and 6.1 for details). Second, the formats of the datasets under investigation are different. Case Study 1 investigates defect records but Case Study 2 investigates both defect records and change logs; see Tables 4.1 and 6.1 for the key attributes of the datasets in the two case studies. However, we have faced similar challenges during, and also have learnt similar lessons from, conducting these two case studies; see Chapter 9 for these challenges and lessons.

Chapter 8

Critical Assessment

Recall that Chapters 4, 5 and 6 describe the three main parts of this thesis research: Case Study 1 (analysis of multiple-component defects – MCDs), the DAD approach and its prototype tool, and Case Study 2 (DAD application). Here, we assess these three parts, focusing mainly on their limitations. First of all, we discuss the relationship between MCDs and architectural degeneration (see Figure 3.1), which is fundamental to this thesis research.

8.1 MCDs and Architectural Degeneration

We note from Section 3.1.2 that MCDs, due to their "multiple-component" nature, are related to potential crosscutting concerns (Eaddy et al., 2008) or architectural problems (von Mayrhauser et al., 2000) in the system. As Stringfellow et al. (2006) state, "problems related to interactions between components is a sign of problems with the software architecture of the system and are often costly to fix." We thus propose that architectural degeneration manifests itself through MCDs (see Figure 3.1). Consequently, the characteristics of MCDs such as their quantity and complexity can reflect the impact of architectural degeneration on software defects. Therefore, we can evaluate the architectural degeneration for a specific system by examining the quantity and complexity of MCDs in that

system. This is the fundamental basis of this thesis research – characterization and diagnosis of architectural degeneration. However, there are two issues related to this fundamental basis.

First, we acknowledge that architectural degeneration has impact on not only software defects but also other quality aspects such as maintainability, adaptability, reusability, etc. (see an example quality model in McCall et al.'s study (1977)). Therefore, the defect perspective, as defined in the work, can only characterize one aspect of architectural degeneration; and we should not make cursory decisions on the architectural degeneration of a system based only on the MCD quantity and complexity measures of components and fix relationships.

Second, even from the defect perspective, architectural degeneration could relate to defects confined to only one component (so called *single-component defects* or SCDs). For example, there are SCDs that require only "examinations", but not "physical" changes, to more than one component for their corrections. These defects are obviously not identified as MCDs in the work but fixing these defects needs to consider their potential impact over the architecture.

Basili and Perricone (1984) call these defects (defects requiring examinations in more than one system module) "interface" defects. The literature (e.g., (Basili and Perricone, 1984), (Perry and Evangelist, 1987) and (Nakajo and Kume, 1991)) indicates that interface defects account for about 40%-65% of all defects (see Section 2.4.3). We note that MCDs are included in interface defects (as per Basili-Perricone's definition). However, there could be over 40% of interface defects that are not MCDs, which are thus not considered in the proposal on the relationship between architectural degeneration and MCDs. Therefore, this could affect diagnosis of architectural degeneration from the defect perspective. Unfortunately, the use of Basili-Perricone's interface defect definition requires subtle, "examination" (or soft), measures (as described in Section 2.4.3), which are not captured widely in actual software projects, including the systems under the investigation.

8.2 Case Study 1: MCD Analysis

Built upon the relationship between MCDs and architectural degeneration (see Figure 3.1), Case Study 1 (see Chapter 4) investigates the distribution, complexity and persistence of MCDs in a large legacy system (of size 20 million SLOC), for the purpose of quantifying the extent to which architectural degeneration affects software defects. The defect dataset analyzed for this study covers 17 of over 20 years of the system and six of the nine major releases.

Results indicate that MCD are concentrated in a few components in the system (see Figure 4.1). Results also indicate that MCDs are complex to fix and are persistent across development phases and releases (see Tables 4.3–4.5). Knowing these characteristics can help management and maintenance staff to focus on particular hard-to-fix defects. Moreover, the MCD profile reflects the adverse impact of architectural degeneration on software defects, mainly, in terms of their fix complexity and difficulty. Knowing this can aid understanding the architectural degeneration of the system and can also increase the necessity and significance of treating the architectural degeneration.

We note from Section 2.4.5 that there is clearly little research conducted on characteristics of MCDs and from Section 4.7 that there are no studies in the literature similar to this case study. In particular, the findings on MCDs add to the current knowledge on architectural defects (e.g., the genre defined by Endres (1975) and Basili-Perricone (1984)) and degeneration.

While we have answered the three questions, (i)–(iii), posed in Section 4.1, as yet we do not know whether the defect complexity metric (i.e., the number of accompanying changes required to fix a defect) reflects the real complexity of defects in the system. The defect dataset under the investigation cannot support this validation. Thus, careful thought needs to be considered in the design of such a study in other contexts.

A limitation of this study is that inter-relationship between the MCD-prone

components and the most frequently occurring fix relationships (see Figure 4.1) was not addressed. Such inter-relationship could benefit cost-effective system quality improvement. Another limitation of this case study is the lack of findings about characteristics of the MCD-prone components. For example, we do not know from this study the extent to which these components tend to persist across phases and releases. Such characteristics can help improve the system quality.

8.3 DAD Approach and Tool

The DAD approach (see Chapter 5) aims at: (i) identifying degeneration-critical components and fix relationships, (ii) evaluating persistence of components and fix relationships, and (iii) evaluating architectural degeneration for a given system, using the MCD *quantity* and *complexity* metrics (as defined in Section 5.2.2). A conceptual DAD framework was proposed in order to carry out these three goals (see Figure 5.1). A prototype tool was developed to facilitate DAD application in real system contexts.

Note that the role of DAD and its prototype tool is to operationalize the defect perspective of architecture degeneration. So, in itself, it does not contribute directly to new theories, but it helps automate the process of discovery. The information of architectural degeneration derived with DAD can help treat the architectural degeneration problem in the system, which could lead to increase in system quality and decrease in maintenance costs. DAD can thus complement existing techniques for architectural degeneration diagnosis (see Section 2.3.3), such as architectural deviation detection (Murphy et al., 2001) (Lindvall and Muthig, 2008), defect-prone component (DPC) identification (Ohlsson and Wohlin, 1998) (Li et al., 2009), and fault architecture construction (von Mayrhauser et al., 2000).

In particular, we note that von Mayrhauser et al. (2000) propose an approach to derive fault architectures from system defect history, which can highlight the degeneration-critical fix relationships in the architecture from the MCD quantity

perspective. Fault architectures are similar to defect architectures created with the DAD approach. However, DAD defines both MCD quantity and complexity metrics (see Section 5.2.2) which support creating defect architectures from both MCD quantity and complexity perspectives. Therefore, we can say that fault architecture is a type of defect architecture and there are defect architectures that are not fault architectures.

However, DAD defines only a defect perspective for diagnosing architectural degeneration. It cannot support the diagnosis from other perspectives such as architectural deviation (see Section 2.3.1). Especially, it is obvious that even from the defect perspective, the MCD quantity and complexity metrics (see Section 5.2.2) cannot measure all characteristics of architectural degeneration. For example, these metrics do not involve the *severity* information of defects due to architectural degeneration. It could be that architectural degeneration leads to more severe defects but DAD cannot measure it.

In addition, DAD examines only the architecture-level degeneration of the system. It cannot offer information about, for example, which code files contribute most to the "degeneration" of a component, and which fix relationships among code files frequently occurred across phases and releases. This kind of information can help refine strategies and solutions of treating architectural degeneration.

8.4 Case Study 2: DAD Validation

Case Study 2 (see Chapter 6) applies the DAD approach (see Section 5.2) to identify the degeneration-critical components and fix relationships and evaluate the architectural degeneration in a commercial system (of size over 1.5 million SLOC and age over 13 years). This system is actually a core subsystem of the subject system of Case Study 1 (see Section 4.3.1).

Case Study 2 investigates the defect-fix history of three major, successive releases of the subject system (see Section 6.2.1). Results first show that there are

a few degeneration-critical components and fix relationships in the system which contribute substantially more to the architectural degeneration than other components and fix relationships (see Figure 6.6). Knowing these degeneration-critical components and fix relationships can help management and practitioners to effectively treat the architectural degeneration of the system. Results also show that the components' contributions to the architectural degeneration tend to persist across phases and releases (see Section 6.3.2). Knowing such persistence can help identify the persistent components in the system which should be treated with intensive attention in order to mitigate the architectural degeneration in next phases or releases. Results then show that the architectural degeneration increased as the system evolved across releases and system restructuring could reverse this increase trend (see Section 6.3.5). Knowing this architectural degeneration trend can help management staff in planning system evolution.

We note that researchers such as Brooks (1975, p. 123), Belady and Lehman (1976) have observed and investigated the architectural degeneration phenomena in real software systems. We also note from Section 2.3.1 that the complexity increased substantially over the evolution of a five-release system (see van Gurp and Bosch's study (2002)), and for a two-version system, the new version released after restructuring the whole system has higher maintainability (measured by inter-module interactions) than the old version (see Lindvall et al.'s study (2002)).

These studies use a structural deviation perspective to understand and diagnose architectural degeneration. Our study applies the DAD approach and thus uses a defect perspective, which is obviously different from that structural deviation perspective. However, although there are no previously published studies that operationalized the defect perspective with MCD quantity and complexity metrics, we still need to acknowledge that some of the findings, such as that the architectural degeneration increased during system evolution and system restructuring could reverse this increase trend, coincide with related findings from the

literature (e.g., (Belady and Lehman, 1976), (van Gurp and Bosch, 2002), and (Lindvall et al., 2002)). We also must claim that the findings about the persistence of components' and fix relationships' contributions to the architectural degeneration are new.

While we have answered the five questions, **Q1**–**Q5**, posed in Section 6.1 (also see Section 1.3.2), as yet we do not have a scientific understanding of the interrelationship between degeneration-critical components and fix relationships in the subject system (see a similar discussion in Section 8.2). For example, we do not know the extent to which the degeneration-critical fix relationships are correlated with the components. This can complement existing results on degeneration-critical components and fix relationships.

Chapter 9

Challenges and Lessons Learnt

Conducting a case study of a large legacy system in industry has its challenges. In this chapter, we describe such challenges and lessons learnt from conducting Case Studies 1 and 2 (see Chapters 4 and 6), which are related to: data access (see Section 9.1); data quality (see Section 9.2); data analysis procedures, risks and scope (see Section 9.3); validation of the findings and interpretation (see Section 9.4); and academic cycles and industry concerns (see Section 9.5). The lessons of these aspects can complement existing studies on the management of university-industry collaborations for empirical studies such as (Beckman et al., 1997), (Conradi et al., 2003), and (Lethbridge et al., 2007).

9.1 Data Access

One of the most difficult aspects of conducting a study of a large system in industry is having access to the necessary artifacts of the study. It is the system's defect records in Case Study 1 (see Section 4.3.1) and the system's defect records and change logs in Case Study 2 (see Section 6.2.1). While open-source datasets (such as source code, defect records and change logs) are publicly available, the specific studies that can be conducted using this data depend on the characteristics of the data actually contained in the datasets. For example, we note that the

attribute "reference" (specifically "parent reference" and "children reference" – Section 4.3.1) was non-existent in the defect records of some open-source systems. Thus, the investigation on the persistence of MCDs (across phases and releases) of Case Study 1 (see Section 4.4.3) was not possible for these systems.

Aside from this, it is generally customary to make a non-disclosure agreement (NDA) to have access to proprietary data. Despite having such agreements, it may still be difficult (in some cases) to obtain access to certain type of data. It especially depends on the sensitivity of the data and organizational lines of authority. For example, it is quite probable that the party in the organization authorized to sign NDAs may not have any authority to provide actual access to proprietary data "owned" by product groups. Thus, the NDA in this case serves only as an "enabler" for an external researcher and not necessarily as access rights to project data.

Further, a product group is not usually a homogenous entity; there are technical people in the group of varying seniority and there are one or more levels of management. Thus, even if the technical people are enthusiastic about research collaboration, they may not be able to authorize access to the required data for the case study. It is our experience that jumping over the various organization hurdles can take weeks to months and this should be factored in any desire to conduct a case study in industry.

Beyond signing a NDA with the administrative body, the researcher needs to identify and collaborate with an appropriate product group prior to having any chance to access data for conducting a desirable study. This is also difficult and time consuming because it depends on certain factors such as: knowing which group is amenable to collaboration, which is not obvious at all in large corporations; whether there is a match between the researcher's interests and expertise and the group's needs for investigation; whether there is budget allocated to the research project; and whether sufficient "trust" has been built with the product

group for them to provide the researcher access to data. This can take many months to accomplish, especially for new collaborative relationships. Even for established relationships, obtaining access to a database for a new case study is not to be taken for granted.

In short, data is not given to the researcher "on a silver platter"; it must be "earned over time". This is thus a critical milestone in conducting a case study on industrial data. Clearly, this depiction is also a serious impediment to any plans for an industry-based "replication" of a case study.

9.2 Data Quality

A challenge in conducting a case study in an industry context is in accepting the fact that the term "quality" as understood in the context of a large commercial organization is different from that as understood in the context of academia. In Case Studies 1 and 2 described in Chapters 4 and 6, the defect and change data is logged in corporate databases in the midst of such key elements such as: large-scale system growth, globally distributed development teams, different cultural habits and norms, experience levels of the personnel, software costs and budget cuts, profitability and market competition, need for timely releases of the system, customer satisfaction requirements, etc.

Especially, the defect data for the six releases used in Case Study 1 was captured over a period of 17 years (see Section 4.3.1) and the defect-fix data for the three releases used in Case Study 2 was captured from the over-13-year-old system (see Section 6.2.1). During the period, the organization has gone through a significant changes, e.g.: corporate size and goals, staff turnover (both technical and managerial); technology and infrastructure changes; policy, process, method, procedure and paradigm changes; skill and market changes; customer and needs changes; etc. Thus, "data quality" in industry has been impacted by such factors. Likewise, a lesson to be learnt here is that researcher expectations, predominantly

grown in the relatively idealized visions of development environments, need to be moderated in terms of "cleanliness" and "completeness" of long-lived data. Recall Sections 4.5.1 and 6.4.1 where we discuss the threats on "data reliability".

9.3 Data Analysis, Risks and Scope

Data quality has impact on the analysis procedures and this impact should not be ignored. Given the concerns for proprietary information that are to be expected in the dataset for the case studies, its impact on data analysis procedures should not be ignored. For example, automated tool support for analyzing data may not work well in complicated situations, e.g., when a field of a defect record is used abnormally for unanticipated purposes. We had encountered a number of such cases (see Sections 4.5 and 6.4) and had needed numerous back-and-forth clarification meetings with the developers and had to resort to manual analysis of defect records and change logs. Such situations usually lead to time delays in the study progress.

It is also important to analyze the risk to the two case studies introduced by the quality of the datasets. In Case Study 1, approximately 20% of the defect records had anomalous use of the "reference" (specifically "parent reference" and "children reference") field (see Section 4.5). However, because there were other, key, usable fields in the affected defect records, we decided not to eliminate these 20% records. (Actually, we have no effective ways to eliminate these 20% records due to the large population.) Clearly, there are no firm criteria for deciding when to exclude a defect record from data analysis; this must be judged on a case-by-case basis by examining the issues at hand. In Case Study 2 (see Section 6.4), most of defects are recorded with only one component but a certain percentage (as we find, approximately 22%-25%) of defects span more than one component. Clearly, such multiple-component information cannot be found within the "Component" field of defect records, because that field logs one and only one component name

(see Section 6.2.2). Fortunately, this can be discovered by investigating the change logs. Otherwise, the identification of MCDs was not possible in Case Study 2.

Furthermore, recall Section 4.3.1 where we mention that Case Study 1 utilizes only six of the nine releases of the subject system. Here, data quality was a prime driver in deciding the specific releases we could use for the study. This obviously affects the scope of the study in terms of the volume of usable data. However, in some situations, the impact on the study can even be in terms of the particular research questions that can be investigated, e.g., when the dataset is not adequately clean or when data is missing. For example, analysis of the persistence of MCDs in Case Study 1 (see Section 4.4.3) would be compromised severely if the data were available from only one or two system releases. Likewise for evaluation of the persistence of components and fix relationships in relation to architectural degeneration (see Sections 6.3.2 and 6.3.4) and analysis of the architectural degeneration trend (see Section 6.3.5) of the subject system over phases and releases in Case Study 2.

9.4 Result Interpretation and Validation

In complex case studies in industry, the findings and their interpretation by researchers can be erroneous in some situations even if data analysis was accurately performed. This is, for example, due to hidden reliability problems with the dataset and with the execution of empirical procedures. Recall Section 4.5 for an example of the non-standard use, by developers, of some defect attribute fields. Should such fields be incorrectly understood by the researcher then this could lead to incorrect inputs for data analysis which, in turn, could lead to incorrect findings and their interpretation.

For example, in Figure 1.1 (also see Section 6.3.1), components C5 and C6 are analyzed to be the most problematic ones due to their MCD content. This finding, by itself, even if accurately resultant from data analysis, is still suspect and

requires further validation from the developers (and in some cases even the users of the system under study). Do, for example, developers consider components C5 and C6 to be problematic in their day-to-day experience? If yes, then this would indicate concurrence of real-world experience by the developers with the technical findings from Case Study 2 – and the confidence in the study findings would be high. If no, then this suggests disagreement between the developers' experience (e.g., defect correction) and the study results and so this warrants further investigation, while lowering the confidence in the findings. Similar validation needs to be conducted on any interpretation of the findings from the two case studies. In other words, the "loop" must be closed by validating the findings in the real-world and the appropriateness of the interpretation in specific contexts. Such validation, though was carried out in the case of components C5 and C6 as described in Section 6.3.1, is subtle and subject to accidental omission.

Furthermore, an intriguing question that arose in the two case studies is whether all findings of the studies can be validated in the manner described above. For example, in Case Study 1, Section 4.4.3 presents cross-release defect persistence (see Table 4.5). Could we then realistically expect developers to be aware of defect persistence across releases, from their day-to-day experience, and help in validating the findings of the study? Considering that in this study the six releases span a period of 17 years, during which there have been a "hurricane of changes" in the corporation, the answer is highly likely "no", which is what it turned out to be. The same question and similar answer are for Case Study 2 where the defect-fix dataset covers a period of 13 years.

Therefore, an important lesson to be learnt here is that certain type of findings (e.g., components C5 and C6 quality – see Section 6.3.1) point to "concrete artifacts" which would be in the realm of the developers' day-to-day experience and, accordingly, would be logged in the project's knowledge-base, and would likely be validate-able. However, certain other type of findings (e.g., persistence across

many releases) point to "process" data spanning a long period of time that would be difficult to track, and would likely be tricky to validate in the field.

9.5 Academic Cycles and Industry Concerns

For obvious reasons, it is expected that case studies tackle industry-scale projects as opposed to classroom-scale projects. Graduate students (in the Masters and Doctoral programs) are increasingly involved in conducting such case studies, as was the occurrence in the described two case studies. However, we saw earlier how concerns for proprietary information and time lags can permeate research projects in industry. This can thus pose serious research risks to the unwary, consequences of which can include outright case study modification and even termination. Unfortunately, there are no silver bullets to satisfactorily deal with this situation and it is the price of industrial contribution to the body of knowledge in Software Engineering.

However, an important lesson learnt from this and other case studies in industry is to carefully weigh the risks involved in conducting a "solution-seeking" as opposed to a "knowledge-seeking" study. The former type of study aims at creating a solution to a known problem; whereas, the latter type of study aims at creating new knowledge by examining a current situation. In the former study, however, the resultant solution needs to be validated by applying it in an actual project. This would imply perturbations in the project which are often resisted to by project personnel. This is actually understandably in the practice but it also implies research risk.

Both Case Studies 1 and 2 are of the latter kind – knowledge seeking (note Case Study 2 validated the DAD approach). In the two studies, neither data gathering nor result validation involved any changes in the actual systems, where research risks were thus contained. These issues apply equally well to new case studies and replicated studies in industry.

9.6 Key Points of Challenges and Lessons

The key challenges and lessons learnt from conducting Case Studies 1 and 2 (see Chapters 4 and 6) are summarized below.

- Industrial data is not given to the researcher on a silver platter; it must be earned over time. Specifically, note that: (1) the NDA agreement serves only as an "enabler" for an external researcher and not necessarily as access rights to project data in an industrial context; (2) accessing the industrial data requires to jump over various organization hurdles, which can take weeks to months; and (3) accessing and interpreting the industrial data is difficult and also time consuming.
- The term "quality" as understood in the context of a large, complex, industrial organization is radically different from that as understood in the context of academia. Therefore, researcher expectations need to be moderated with long-lived data from industry.
- Data quality related problems can inflict impact on data analysis procedures and this impact should not be ignored. Thus, it is important to analyze the risk to the studies introduced by the quality of the data.
- Findings and their interpretation of complex industrial studies should be validated in industrial contexts. However, only certain type of findings point to "concrete artifacts" which would likely be validate-able; certain other type of findings point to "process" data would be not validate-able.
- The concerns for proprietary information and time lags occurred during industrial studies could pose serious research risks.

The above mentioned challenges apply equally well to new case studies and replicated studies in industry. Unfortunately, there are no silver bullets to satisfactorily deal with these challenges. However, we believe that the lessons learnt

from the two case studies could help researchers in conducting new or replicated case studies with industrial data in future.

Note that Lethbridge et al. (2007) also discuss benefits, drawbacks, risks, and risk-reducing factors of empirical studies in industrial settings. They further create a checklist of activities that should be involved in planning and management of industry-university collaborations, such as negotiating level and type of commitment, risk management, and access to participants. Other researchers such as Beckman et al. (1997), Mead et al. (1999), Arisholm et al. (1999), Conradi et al. (2003), and Rombach and Achatz (2007) also address similar issues regarding research collaborations. Our lessons learnt, as shown above, can complement these existing studies.

Chapter 10

Conclusions and Future Work

Software architectures degenerate over time (Lehman, 1980), resulting in increased software costs and quality problems (Stringfellow et al., 2006). This phenomenon is termed *architectural degeneration* (Lindvall et al., 2002). The problems described in Section 1.1 concerning the Mozilla browser, Linux-kernel and AT&T 5ESS evolution (e.g., structural degradation and the need for re-engineering) indicate that architectural degeneration can lead to substantial quality and cost problems for software systems.

Previous research has focused on *architectural deviations* from the baseline as a vehicle to study degeneration. However, while detecting (Murphy et al., 2001), measuring (Lindvall et al., 2002) and removing (Tran and Holt, 1999) deviations in an architecture can align it better to its baseline (from a structural perspective) (Hochstein and Lindvall, 2005), this may not improve architectural quality (Bhattacharya and Perry, 2007). In this thesis, we examine architectural degeneration from the perspective of *software defects*. In particular, we characterized and diagnosed architectural degeneration by answering the following two questions (see Section 1.2): (1) *What do defects indicate about architectural degeneration*? and (2) *How can architectural degeneration be diagnosed from the defect perspective*?

To answer question (1), we conducted an *exploratory case study*, Case Study 1

(see Chapter 4), analyzing defect data over six releases of a large legacy system (of size approx. 20 million SLOC and age over 20 years). The relevant defects here are those that span multiple components in the system (called *multiple-component defects* – MCDs (Li et al., 2009)). To answer question (2), we developed an approach (called **D**iagnosing **A**rchitectural **D**egeneration – DAD; see Chapter 5) from the defect perspective, and validated it in another, *confirmatory*, case study (Case Study 2 – Chapter 6) involving three releases of a commercial system (of size over 1.5 million SLOC and age over 13 years).

The key results of this thesis are (see Chapters 4–6):

(1) *The knowledge of MCDs* – On average, fixing a MCD requires nearly 3 times changes (based on components) as much as that for fixing a non-MCD, and MCDs are 6.0-8.5 times more persistent across development phases and releases than other types of defects (see Sections 4.4.2 and 4.4.3).

(2) *The DAD approach*, which defines MCD quantity and complexity metrics (see Section 5.2.2) to measure components and *fix relationships* (a change-coupling relationship among components due to MCDs) of a given system for diagnosing architectural degeneration of the system. A prototype tool has been developed to facilitate the usage of DAD.

(3) *The knowledge of architectural degeneration* – Architectural degeneration is, largely and persistently, due to approx. 20% of components and approx. 10% of fix relationships in a system (see Sections 6.3.1–6.3.4); and it increases (e.g., by 25% in the MCD complexity) as the system evolves from release to release, but it can decrease (by even higher rate) after system restructuring (see Section 6.3.5).

(4) *Lessons learnt from conducting empirical studies in an industrial context* (see Chapter 9), concerning industrial data access, data quality, cleaning and analysis of data, interpretation and validation of the findings, academic cycles, industry jitters, and time lags.

These results are novel and they complement previous research on architectural degeneration (e.g., deviation-based architectural measurement (Lindvall et al., 2002) and construction of "fault architectures" from defect history (von Mayrhauser et al., 2000)). The key conclusions from these results are: (i) analysis of MCDs is a viable approach to characterizing architectural degeneration; and (ii) a method such as DAD can be developed based on MCD characteristics for diagnosing architectural degeneration from the defect perspective.

In the future, we intend to conduct further research from two aspects. The first is to extend the DAD approach (and its prototype tool; see Sections 5.2 and 5.3) to investigate the "degeneration" of a specific component. This can aid understanding a component in relation to its inside defect-related problems. This extension requires the definition of defect-related metrics to measure the "contribution" of a single code file (and a fix relationship between two code files) to the degeneration of the component.

The second aspect is to examine: (i) the inter-relationship between degeneration-critical components and fix relationships, and (ii) the underlying reason for the persistence of varying degrees of "degeneration"-measures of the components and fix relationships over time (see Sections 6.3.2 and 6.3.4). This could form a foundation for developing further technological support to aid in the architectural degeneration treatment process. See questions (1) and (2) in Section 6.5.3 for a detailed discussion.

Bibliography

Walid Abdelmoez, Mark Shereshevsky, Rajesh Gunnalan, Bo Yu, S. Bogazzi, Mustafa Korkmaz, and Hany. H. Ammar. Software architectures change propagation tool (sacpt). In *Proceedings of the 20th International Conference on Software Maintenance (ICSM'04)*, pages 517–517, Chicago, USA, September 2004.

Edward N. Adams. Optimizing preventive service of software products. *IBM Research Journal*, 28(1):2–14, 1984.

Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*, pages 71–80, Sorrento, Italy, May 1994.

Carina Andersson and Per Runeson. A replicated quantitative analysis of fault distribution in complex software systems. *IEEE Transactions on Software Engineering*, 33(5):273–286, 2007.

Anneliese Amschler Andrews and Catherine Stringfellow. Quantitative analysis of development defects to guide testing: A case study. *Software Quality Journal*, 9:195–214, 2001.

Anneliese Amschler Andrews, Magnus C. Ohlsson, and Claes Wohlin. Deriving fault architectures from defect history. *Journal of Software Maintenance: Research and Practice*, 12:287–304, 2000.

Erik Arisholm, Bente Anda, Magne Jørgensen, and Dag I.K. Sjøberg. Guidelines on conducting software process improvement studies in industry. In *Proceedings of the 22nd IRIS Conference (Information Systems Research Seminar in Scandinavia)*, pages 87–102, Keuruu, Finland, 1999.

Robert S. Arnold. *Software Reengineering*. IEEE Computer Society, 1993.

Robert S. Arnold and Shawn A. Bohner. Impact analysis - towards a framework for comparison. In *Proceedings of the Conference on Software Maintenance (CSM'93)*, pages 292–301, Montreal, Canada, September 1993.

Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. The relationship between design patterns defects and crosscutting concern scattering degree: An empirial study. *IET Software*, 3(5):395–409, 2009.

Rajendra K. Bandi, Vijay K. Vaishnavi, and Daniel E. Turk. Predicting maintenance performance using object-oriented design complexity metrics. *IEEE Transactions on Software Engineering*, 29(1):77–87, 2003.

Rajvi D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–94, 1993.

Victor R. Basili and Barry T. Perricone. Software errors and complexity: An empirical investigation. *Communications of the ACM*, 27(1):42–52, 1984.

Victor R. Basili and Forrest Shull. Evolving defect "folklore": A cross-study analysis of software defect behavior. In *Proceedings of the International Software Process Workshop (ISPW'05)*, pages 1–9, Beijing, China, May 2005.

Victor R. Basili, Dieter Rombach, Kurt Schneider, Barbara Kitchenham, Dietmar Pfahl, and Richard W. Selby (ed.). *Empirical Software Engineering Issues: Critical Assessment and Future Directions*. Springer, 2006.

Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice (Second Edition)*. Addison-Wesley Professional, 2003.

Kathy Beckman, Soheil Khajenoori, Neal Coulter, and Nancy R. Mead. Collaborations: Closing the industry-academia gap. *IEEE Software*, 14(6):49–57, 1997.

Laszlo A. Belady and Meir M. Lehman. A model of large program development. *IBM Systems Journal*, 3:225–252, 1976.

Keith Bennett and Václav Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference of the Future of Software Engineering*, pages 73–87, Limerick, Ireland, June 2000.

Rudolf Berghammer and Alexander Fronk. Applying relational algebra in 3d graphical software design. In *Proceedings of the 7th International Seminar on Relational Methods in Computer Science (RelMiCS 7)*, pages 89–96, Malente, Germany, May 2003.

Jan A. Bergstra, Jan Heering, and Paul Klint. Module algebra. *Journal of the Association for Computing Machinery (JACM)*, 37(2):335–372, 1990.

Sutirtha Bhattacharya and Dewayne E. Perry. Architecture assessment model for system evolution. In *Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, pages 8–17, Mumbai, India, January 2007.

Ted J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22(7):36–49, 1989.

Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *IEEE Computer*, 34(1):135–137, 2001.

Barry W. Boehm. Software engineering. *IEEE Transactions on Computers*, C-25 (12):1226–1241, 1976.

Shawn Bohner and Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

Grady Booch. The economics of architecture-first. *IEEE Software*, 24(5):18–20, 2007.

Grady Booch. Nine things you can do with old software. *IEEE Software*, 25(5): 93–94, 2008.

Ivan T. Bowman and Richard C. Holt. Linux as a case study: Its extracted software architecture. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 555–563, Los Angeles, CA, USA, May 1999.

Gerald W. Bracey. *Reading Educational Research: How to Avoid Getting Statistically Snookered*. Heinemann, 2006.

Fred P. Brooks. *The Mythical Man-Month*. Addison Wesley, 1975.

Ettiot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. Orthogonal defect classification - a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, 1992.

Marcus Ciolkowski, Oliver Laitenberger, and Stefan Biffl. Software reviews: The state of the practice. *IEEE Software*, 20(6):46–51, 2003.

Paul Clements and Linda M. Northrop. Software architecture: An executive overview. Technical Report, CMU/SEI-96-TR-003, Feb, 1996.

Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2002.

Edgar F. Codd. Relational completeness of data base sublanguages. *Data Base Systems: Courant Computer Science Symposia Series*, 6:65–98, 1972.

B. Terry Compton and Carol Withrow. Prediction and control of ADA software defects. *Journal of Systems and Software*, 12(3):199–207, 1990.

Reidar Conradi, Tore Dybå, Dag I.K. Sjøberg, and Tor Ulsund. Lessons learned and recommendations from two large norwegian SPI programmes. In *Proceedings of the 9th European Workshop on Software Process Technology (EWSPT'03)*, pages 32–45, Helsinki, Finland, September 2003.

Thomas A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

B. J. Cornelius, M. Munro, and D. J. Robson. An approach to software maintenance education. *Software Engineering Journal*, 4(4):233–136, 1989.

John W. Creswell. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches (2nd edition)*. Sage Publications, 2002.

Michael A. Cusumano and Richard W. Selby. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*. The Free Press, 1995.

James B. Dabney, Gary Barber, and Don Ohi. Estimating direct return on investment of independent verification and validation. In *Proceedings of the IASTED Conference on Software Engineering and Applications*, pages 394–399, Cambridge, MA, USA, November 2004.

Marco D'Ambros, Michele Lanza, and Romain Robbes. On the relationship between change coupling and software defects. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE'09)*, pages 135–144, Antwerp, Belgium, October 2009.

Liliana Dobrica and Eila Niemela. A survey of software architecture analysis methods. *IEEE Transactions on Software Engineering*, 28(7):638–653, 2002.

Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects. *IEEE Transactions on Software Engineering*, 34(4):497–515, 2008.

Christof Ebert. Metrics for identifying critical components in software projects. In *Handbook of Software Engineering and Knowledge Engineering*, 2001.

Christof Ebert, Reiner Dumke, Manfred Bundschuh, and Andreas Schimietendorf. *Best Practices in Software Measurement*. Springer, 2005.

Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

Albert Endres. An analysis of errors and their causes in system programs. *ACM SIGPLAN Notices*, 10(6):327–336, 1975.

Len Erlikh. Leveraging legacy system dollars for e-business. *IT Pro*, 2(3):17–23, 2000.

Hoda Fahmy and Richard C. Holt. Software architecture transformations. In *Proceedings of the 16th International Conference on Software Maintenance (ICSM'00)*, pages 88–96, San Jose, California, USA, October 2000.

Loe M.G. Feijs and R. L. Krikhaar. Relation algebra with multi-relations. *International Journal of Computer Mathematics*, 70(1):57–74, 1998.

Loe M.G. Feijs and Yuechen Qian. Component algebra. *Science of Computer Programming*, 42(2-3):173–228, 2002.

Loe M.G. Feijs, R. L. Krikhaar, and R.C. van Ommering. A relational approach to support software architecture analysis. *Software Practice & Experience*, 28 (4):371–400, 1998.

Norman Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26 (8):797–814, 2000.

Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

Martin Fowler. Who needs an architect? *IEEE Software*, 20(5):11–13, 2003.

Keith Gallagher, Andrew Hatch, and Malcolm Munro. Software architecture visualization: An evaluation framework and its application. *IEEE Transactions on Software Engineering*, 34(2):260–270, 2008.

David Garlan and Mary Shaw. *An Introduction to Software Architecture*. Advances in Software Engineering and Knowledge Engineering, Volume I, World Scientific Publishing Company, 1993.

Mechelle Gittens, Yong Kim, and David Godwin. The vital few versus the trivial many: Examining the pareto principle for software. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, pages 179–185, Edinburgh, Scotland, July 2005.

Michael W. Godfrey and Eric H. S. Lee. Secrets from the monster: Extracting mozilla's software architecture. In *Proceedings of the 2nd Symposium of Constructing Software Engineering Tools (CoSET'00)*, pages 15–23, Limerick, Ireland, June 2000.

Ian Gorton. *Essential Software Architecture*. Springer-Verlag, 2006.

Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall, 1992.

Todd L. Graves and Audris Mockus. Inferring change effort from configuration management databases. In *Proceedings of the 5th International Symposium on Software Metrics (ISSM'98)*, pages 267–273, Bethesda, MD, USA, March 1998.

Penny Grubb and Armstrong A. Takang. *Software Maintenance: Concepts and Practice*. World Scientific Publishing Company, 2003.

Peter Hiemann. A new look at the program development process. In *Proceedings of the 4th Informatic Symposium on Program Methodology*, pages 11–37, Bad Wildbad, Germany, September 1974.

Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

Lorin Hochstein and Mikael Lindvall. Combating architectural degeneration: A survey. *Information and Software Technology*, 47:643–656, 2005.

Richard C. Holt. Structural manipulations of software architecture using tarski relational algebra. In *Proceedings of the 5th IEEE Working Conference on Reverse Engineering (WCRE'98)*, pages 210–219, Honolulu, Hawaii, USA, October 1998.

Richard C. Holt. Software architecture abstraction and aggregation as algebraic manipulations. In *Proceedings of the 9th Conference of the IBM Centre for Advanced Studies on Collaborative Research (CASCON'99)*, pages 5–17, Mississauga, Ontario, Canada, November 1999.

Richard C. Holt. Grokking software architecture. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE'08)*, pages 5–14, Antwerp, Belgium, October 2008.

P. Hsia, G. Gupta, C. Kung, J. Peng, and S. Liu. A study on the effect of architecture on maintainability of object-oriented systems. In *Proceedings of the 3th International Conference on Software Maintenance (ICSM'95)*, pages 4–11, Opio (Nice), France, October 1995.

Ivar Jacobson and Fredrik Lindström. Re-engineering of old systems to an object-oriented architecture. In *Proceedings of the 6th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*, pages 340–350, Phoenix, Arizona, USA, October 1991.

Catherine Blake Jaktman, John Leaney, and Ming Liu. Structural analysis of the software architecture - a maintenance assessment case study. In *Proceedings of the 1st Working IFIP Conference on Software Architecture (WICSA'99)*, pages 455–470, San Antonio, TX, USA, February 1999.

Anton Jansen and Jan Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, pages 109–120, Pittsburgh, Pennsylvania, USA, November 2005.

Michael Jiang, Jing Zhang, Hong Zhao, and Yuanyuan Zhou. Maintaining software product lines c an industrial practice. In *Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM'08)*, pages 444–447, Beijing, China, September 2008.

D. Kafura and R. Reddy. The use of software complexity metrics in software maintenance. *IEEE Transactions on Software Engineering*, 13(3):335–343, 1987.

Alan Karr, Adam Porter, and Lawrence Votta Jr. An empirical exploration of code evolution. In *Proceedings of the 1st International Workshop on Empirical Studies of Software Maintenance (WESS'96)*, Monterey, CA, USA, November 1996.

Rick Kazman, Liam O'Brien, and Chris Verhoef. Architecture reconstruction guidelines, third editions, technical report, cmu/sei-2002-tr-034, 2002.

Rene L. Krikhaar. Reverse architecting approach for complex systems. In *Proceedings of the 1997 International Conference on Software Maintenance (ICSM'97)*, pages 1–11, Bari Italy, September 1997.

Rene L. Krikhaar, Andre Postma, Alex Sellink, Marc Stroucken, and Chris Verhoef. A two-phase process for software architecture improvement. In *Proceedings of the 15th International Conference on Software Maintenance (ICSM'99)*, pages 371–380, Oxford, England, UK, August 1999.

Shyamsundar Kulkarni. Software defect rediscoveries: Causes, taxonomy and signficance. M.sc., The University of Western Ontario, 2008.

Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 492–501, Shanghai, China, May 2006.

Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

Marek Leszak, Dewayne E. Perry, and Dieter Stoll. A case study in root cause defect analysis. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 428–437, Limerick, Ireland, June 2000.

Timothy C. Lethbridge, Steve Lyon, and Peter Perry. The management of univeristy-industry collaborations involving empirical studies of software engineering. In Forrest Shull, Jaince Singer, and Dag I.K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 257–284. Springer, 2007.

Hareton K.N. Leung and Lee White. Insights into regression testing. In *Proceedings of the Conference on Software Maintenance (ICSM'89)*, pages 60–69, Miami, FL, USA, October 1989.

Zude Li, Mechelle Gittens, Syed Shariyar Murtaza, Nazim H. Madhavji, Andriy V. Miranskyy, David Godwin, and Enzo Cialini. Analysis of pervasive multiple-component defects in a large software system. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM'09)*, pages 265–273, Edmonton, Alberta, Canada, September 2009.

Bennet P. Lientz and E. B. Swanson. *Software Maintenance Management: a Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley Publishing, 1980.

Richard J. Lindner and D. Tudahl. Software development at a baldrige winner. In *Proceedings of the International Electro Conference (ELECTRO'94)*, pages 167–180, Boston, USA, May 1994.

Mikael Lindvall and Dirk Muthig. Bridging software architecture gap. *IEEE Computer*, 41(6):98–101, 2008.

Mikael Lindvall, Roseanne Tesoriero, and Patricia Costa. Avoiding architectural degeneration: An evaluation process for software architecture. In *Proceedings of the 8th IEEE Symposium on Software Metrics (METRICS'02)*, pages 77–86, Ottawa, Canada, June 2002.

Michael J. Lyons. Salvaging your software asset: (tools based maintenance). In *Proceedings of the AFIPS Conference on National Computer*, pages 337–341, Chicago, Illinois, USA., May 1981.

Alan MacCormack, John Rusnak, and Carliss Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030, 2006.

Nazim H. Madhavji. Environment evolution: The prism model of changes. *IEEE Transactions on Software Engineering*, 18(5):380–392, 1992.

Jim A. McCall, P. K. Richards, and G. F. Walters. Factors in software quality. *National Technique Information Service*, 1,2,3, 1977.

Thomas McGibbon. Software reliability data summary, data analysis center for software (dacs), 1996.

Andrew McNair, Daniel M. German, and Jens Weber-Jahnke. Visualizing software architecture evolution using change-sets. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE'07)*, pages 130–139, Vancouver, BC., Canada, October 2007.

Nancy Mead, Kathy Beckman, Jimmy Lawrence, George O'Mary, Cynthia Parish, Perla Unpingco, and Hope Walker. Industry/university collaboration: Different perspectives heighten mutual opportunities. *Journal of Systems and Software*, 49:409–423, 1999.

Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

Tom Mens and Tom Tourw. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

Jürgen Münch. Effective data interpretation. In Victor R. Basili, Dieter Rombach, Kurt Schneider, Barbara Kitchenham, Dietmar Pfahl, and Richard W. Selby, editors, *Empirical Software Engineering Issues: Critical Assessment and Future Directions*, pages 83–90. Springer, 2006.

Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

Takeshi Nakajo and Hitoshi Kume. A case history analysis of software error cause-effect relationships. *IEEE Transactions on Software Engineering*, 17(8): 830–838, 1991.

Josef Nedstam, Even-Andre Karlsson, and Martin Host. Experiences from the architectural change process. In *Proceedings of the 2nd International Workshop from Software Requirements to Architectures (STRAW'03)*, Portland, Oregon, USA, May 2003.

Josef Nedstam, Even-Andre Karlsson, and Martin Host. The architectural change process. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE'04)*, pages 27–36, Redondo Beach CA, USA, August 2004.

John Nestor, William Wulf, and David Lamb. IDL - interface description language - formal description (draft revision 2), technical report, computer science department, carnegie-mellon university, 1982.

Bashar Nuseibeh. Weaving the software development process between requirements and architectures. In *Proceedings of the First International Workshop From Software Requirements to Architectures (STRAW'01)*, Toronto, Canada, May 2001.

Magnus C. Ohlsson and Claes Wohlin. Identification of green, yellow and red legacy components. In *Proceedings of the 14th International Conference on Software Maintenance (ICSM'98)*, pages 6–15, Bethesda, Washington D.C., USA, November 1998.

Magnus C. Ohlsson, Anneliese von Mayrhauser, Brian McGuire, and Claes Wohlin. Code decay analysis of legacy software through successive releases. In *Proceedings of the 1999 IEEE Aerospace Conference*, pages 69–81, Snowmass at Aspen, CO, USA, 1999.

Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'02)*, pages 55–64, Rome, Italy, July 2002.

Thomas J. Ostrand, Elaine Weyuker, and Robert M. Bell. Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*, 31(4):340–355, 2005.

Alessandra Padula. Use of a program understanding taxonomy at Hewlett-Packard. In *Proceedings of the 2nd Workshop on Program Comprehension*, pages 66–70, Capri, Italy, July 1993.

David Lorge Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(2):128–138, 1979.

David Lorge Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering (ICSE'94)*, pages 279–287, Sorrento, Italy, May 1994.

Dewayne E. Perry and W. M. Evangelist. An empirical study of software interface faults. In *Proceedings of the 20th Annual Hawaii International Conference on Systems Sciences*, pages 113–126, Hawaii, January 1987.

Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

Thomas M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, Inc., 1997.

D. Li. L. Rees, K. Stephenson, and J. V. Tucker. The algebraic structure of interfaces. *Science of Computer Programming*, 49(1-3):47–88, 2003.

Dieter Rombach and Reinhold Achatz. Research collaborations between academia and industry. In *Proceedings of the Future of Software Engineering 2007, the 29th International Conference on Software Engineering (ICSE'07)*, pages 29–36, Minneapolis, MC, USA, May 2007.

H. Dieter Rombach. A controlled experiment on the impact of software structure on maintainability. *IEEE Transactions on Software Engineering*, 13(3):344–354, 1987.

Kenneth A. Ross and Charles R.B. Wright. *Discrete Mathematics*. Prentice Hall, 1988.

Christian Del Rosso. Continuous evolution through software architecture evaluation: A case study. *Journal of Software Maintenance and Evolution: Research and Practice*, 18:351–383, 2006.

Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2): 173–210, 1997.

Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs*. Springer-Verlag, 1993.

Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices.* Addison-Wesley, 2003.

Michael E. Shin, Yan Xu, Fernando Paniagua, and Jun Hoon An. Detection of anomalies in software architecture with connectors. *Science of Computer Programming*, 61:16–26, 2006.

Forrest Shull, Victor Basili, Barry Boehm, A. Winsor Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What we have learned about fighting defects. In *Proceedings of the 8th IEEE Symposium on Software Metrics (METRICS'02)*, pages 249–258, Ottawa, Canada, June 2002.

Harry M. Sneed. Economics of software re-engineering. *Software Maintenance: Research and Practice*, 3(3):163–182, 1991.

Ian Sommerville. *Software Engineering (8th edition).* Addison Wesley, 2006.

Maria J. C. Sousa and Helena Mendes Moreira. A survey on the software maintenance process. In *Proceedings of the 14th International Conference on Software Maintenance (ICSM'98)*, pages 265–274, Bethesda, Maryland, USA, November 1998.

Wayne G. Stevens, G. Meyers, and Larry Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

William C. Stratton, Deane E. Sibol, Mikael Lindvall, and Patricia Costa. The save tool and process applied to ground software development at JHU/APL: An experience report on technology infusion. In *Proceedings of the 31st Annual IEEE/NASA Software Engineering Workshop (SEW'07)*, pages 187–193, Columbia, Maryland, USA, March 2007.

Catherine Stringfellow and Anneliese Andrews. Deriving a fault architecture to guide testing. *Software Quality Journal*, 10:299–330, 2002.

Catherine Stringfellow, C. D. Amory, D. Potnuri, Anneliese Andrews, and M. Georg. Comparison of software architecture reverse engineering methods. *Information and Software Technology*, 48:484–497, 2006.

Jeff Sutherland. Business objects in corporate information systems. *ACM Computing Surveys*, 27(2):274–276, 1995.

SWEBOK. Guide to the software engineering body of knowledge; see `www.swebok.org`, 2004.

Alfred Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(3): 73–89, 1941.

John B. Tran and Richard C. Holt. Forward and reverse repair of software architecture. In *Proceedings of the 9th Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'99)*, pages 12–20, Mississauga, Ontario, Canada, November 1999.

Jilles van Gurp and Jan Bosch. Design erosion: Problems and causes. *Journal of Systems and Software*, 61(2):105–119, 2002.

Anneliese von Mayrhauser, Magnus C. Ohlsson, and Claes Wohlin. Deriving fault architectures from defect history. *Journal of Software Maintenance: Research and Practice*, 12(5):287–304, 2000.

David M. Weiss. Evaluating software development by error analysis: The data from the architecture research facility. *Journal of Systems and Software*, 1: 57–70, 1979.

David M. Weiss and Victor R. Basili. Evaluating software development by analysis of changes: Some data from the software engineering laboratory. *IEEE Transactions on Software Engineering*, 11(2):157–168, 1985.

Michel Wermelinger and Jose Luiz Fiadeiro. Connectors for mobile programs. *IEEE Transactions on Software Engineering*, 24(5):331–341, 1998.

Byron J. Williams and Jeffrey C. Carver. Characterizing software architecture changes: A systematic review. *Information and Software Technology*, 52:31–51, 2010.

Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, 2000.

Annie T.T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

Tze-Jie Yu, Vincent Y. Shen, and Hubert E. Dunsmore. An analysis of several software defect models. *IEEE Transactions on Software Engineering*, 14(9): 1261–1270, 1988.

Weider D. Yu. A software fault prevention approach in coding and root cause analysis. *Bell Labs Technical Journal*, 3(2):3–21, 1998.

Martin V. Zelkowitz and Loana Rus. Defect evolution in a product line environment. *Journal of Systems and Software*, 70(1-2):143–154, 2004.

Thomas Zimmermann, Peter Weibgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, pages 563–572, Edinburgh, UK, May 2004.

Appendix A

DAD with Relation Algebra

In this chapter, we describe a Relation Algebra implemented in the DAD prototype tool to support expression and manipulation of defect properties of software architectures. In Section A.1, we briefly introduce the motivation of using Relation Algebras for architectural analysis. We then review related algebraic work in Section A.2, centered on expression and manipulation of software architectures. After that, we outline basic notions and their notations in Section A.3. In Section A.4, we describe the Relation Algebra. In Section A.5, we present a simple application of this approach for architectural degeneration measurement. In Section A.6, we elaborate the algebra implementation in the DAD prototype tool. Finally, we compare the approach against existing related techniques in Section A.7.

A.1 Motivation

A software architecture is the structure of a system, comprising of software elements, the externally visible properties of those elements, and the relationships among them (Bass et al., 2003, p. 21). It is often specified informally as a *box-and-arrow* graph (Gorton, 2006, p. 117), termed the *architectural graph* (Holt, 1998). More formally, an ordinary Relation Algebra (such as Tarski's algebra of binary relations (Tarski, 1941) and Codd's algebra of n-ary relations (Codd, 1972)) has

been used by some researchers to facilitate the visualization (Berghammer and Fronk, 2003), transformation (Krikhaar et al., 1999), abstraction (Holt, 1999), aggregation (Holt, 1999), and analysis (Feijs et al., 1998) of software architectures. These are, clearly, important operations on software architectures for the development of software systems.

However, the reality is that software systems can be defective and approximately 5% to 20% of these defects are *architectural defects* (Yu, 1998; Li et al., 2009). Such defects are characterized by: (i) their span across multiple software components (also often referred to as "elements"); (ii) their persistence across multiple development phases or releases; and (iii) their complexity.

Just as a Relation Algebra has been demonstrated to be invaluable for architectural "structure" (e.g., through the described operations) (Holt, 2008), so it can be used fruitfully for the visualisation, aggregation and analysis of architectural "defects". For example, a defect spanning two or more components can be depicted in the algebra as attributes of inter-component relationships. Also, defects pertaining to higher levels of system abstraction (such as system, subsystem and component levels) can be expressed in the algebra as aggregates of defects pertaining to lower levels of system abstraction (such as files, classes and functions). Furthermore, defects persistent across multiple phases or releases can be stated in the algebra through longitudinal analysis of the defects.

Ordinary Relation Algebras such as Tarski's algebra (Tarski, 1941) cannot express and manipulate the described architectural defect-properties by its relational structure and operations. For example, consider that there are three architectural (inter-component) defects d_i, d_j and d_k involving a dependency relationship between components x and y in a software architecture. The architectural relationship can be described as a tuple of a binary relation: $<x, y>$ in the ordinary algebra. However, the three defects cannot be expressed easily within such a binary relation structure; it requires the algebra to deal with different numbers of

defects embodied across the various architectural components. This is also the problem with Codd's algebra (Codd, 1972). Also, the ordinary relational operators cannot manipulate defect-properties of $<x, y>$. Similarly, both Tarski's and Codd's algebras also need to deal with issue of cross-longitudinal defects.

Feijs and Krikhaar (1998) and Holt (1999) have extended the ordinary algebra of binary relations by incorporating a numeric arity with the binary relation (e.g., $<x, y, n>$ where, for example, n can be aggregate size of x and y). While this is useful to capture architectural properties, its power of expression is limited when attempting to address architectural defect-properties already described. We can see from this that there was a need to investigate architectural defect-properties and how these can be expressed in, and manipulated by, a Relation Algebra.

Building upon the work by Feijs and Krikhaar (1998) and Holt (1999), we propose an approach to expressing and manipulating defect-properties of software architectures with a Relation Algebra. Example defect-properties are the quantity of defects pertaining to a system component or involving two or more components, the *complexity* of a defect in terms of, for example, the number of components changed in order to fix this defect, and the *persistence* of a defect crossing development phases and releases.

In our approach, the Relation Algebra is used to support expression of the architectural defects of a software system with ordinary relations, and manipulation of the architectural defects with ordinary relational operations. This approach can therefore support analysis of the architectural defect-properties (e.g., their distribution, complexity and persistence) for a given software system. In particular, it facilitates several requirements for architectural analysis:

(i) specifying the varying number of inter-component defects across the architecture (von Mayrhauser et al., 2000; Li et al., 2009);

(ii) analysis of multiple architectures for persistent defects (e.g., across a product line (Jiang et al., 2008; Zelkowitz and Rus, 2004));

(iii) analysis of the span of architectural defects crossing multiple components (e.g., interface defects (Endres, 1975; Basili and Perricone, 1984));

(iv) tracing defects across longitudinally (phases and releases) for a given system (von Mayrhauser et al., 2000; Li et al., 2009).

With this Relation Algebra, the DAD prototype tool can implement its main features – characterizing MCDs and diagnosing architectural degeneration – for a given system. See Appendix A for a complete description of this Relation Algebra.

A.2 Related Algebraic Work

In this section, we briefly outline existing literature on Relation Algebras and formal specification of architectures, centering on expressing and manipulating defect-properties for software architectures.

A.2.1 Ordinary Relation Algebras

A classical Relation Algebra is Tarski's algebra of binary relations (1941), which has been implemented in a couple of languages (e.g., Relview and Grok - see (Holt, 2008)). For example, Grok is light-weight tool, uses formal approach and supports features such as architectural comparison, abstraction and visualization (1998). Feijs et al. (1998) define a variant of Tarski's algebra, termed Relation Partition Algebra (RPA). This has been implemented in a tool called TEDDY, which shares many features with Grok. A more expressive algebra is Codd's algebra of n-ary relations (1972), which has been implemented in several languages such as GReQL, JGrok and CrocoPat (see (Holt, 2008)). They can be used to manipulate n-ary relationships (e.g., hierarchy and dependency relationships among components).

Feijs and Krikhaar (1998) define a multi-relation theory, where an ordinary relation is bound to a numeric multiplicity arity. The arity is a number which can be used to denote user-defined concepts, such as: the number of instances of a

given tuple in a relation; the size of the related components; defect-count; change-counts; etc. This structure allows more complex relational operators to be defined over relations (e.g., aggregation of component sizes, defects, changes made). Holt (1999) also proposed a similar extension to Tarski's algebra for application to the field of software architectures.

A.2.2 Algebras of Components and Connectors

Complementing the work on Relation Algebra is the formalization of components and connectors for architectural specification. Connectors are communication vehicles (e.g., procedure calls) among the components of a system. There are algebras specifically for the formalization of architectural components and connectors. For example, Bergstra et al. (1990) define a module algebra that specifies export/import resources in first-order logic. Feijs and Qian (2002) similarly define an algebra for component interfaces.

Also, Allen and Garlan (1994) propose an architectural connector theory which defines architectural connectors as explicit semantic interactions between components. In this theory, each connector is specified as a protocol that characterizes the components in an interaction and how they interact. The underlying model is a process algebra (a subset of Hoare's Communicating Sequential Process or CSP (Hoare, 1985)). Similarly, Wermelinger and Fiadeiro (1998) propose a connector algebra which supports connector construction for software design. For example, it supports construction of new connectors from old ones.

In addition, there is other algebraic work for formal specification of architectures (Nestor et al., 1982), where composition of interfaces and import/export relationships amongst the interfaces are prominent (Rees et al., 2003). Furthermore, there are a number of architectural description or modeling languages (e.g., Acme, Wright, and UML – see (Medvidovic and Taylor, 2000)) that also facilitate specification of architectures.

A.2.3 Analysis of Existing Algebraic Work

Just as a Relation Algebra has been demonstrated to be invaluable for architectural "structure" (Holt, 2008), so it could be used fruitfully for the visualization, aggregation and analysis of architectural "defects". However, ordinary Relation Algebras such as Tarski's algebra (1941) and Codd's algebra (1972) cannot express and manipulate architectural defect-properties by its relational structure and operations. For example, consider that there are three architectural (inter-component) defects d_i, d_j and d_k involving a dependency relationship between components x and y in a software architecture. The architectural relationship can be described as a tuple of a binary relation: $<x, y>$ in the ordinary algebra. However, the three defects cannot be expressed easily within such a binary or n-ary relation structure; it requires the algebra to deal with different number of defects embodied across the various architectural components. Similarly, both Tarski's and Codd's algebras also need to deal with issue of cross-longitudinal defects.

Feijs and Krikhaar (1998) and Holt (1999) have extended Tarski's algebra (1941) by incorporating a numeric arity with the binary relation (e.g., $<x, y, n>$ where, for example, n can be aggregate size of x and y). While this is useful to capture architectural properties, its power of expression is limited when attempting to address architectural defect-properties already described.

Analyzing these algebraic techniques and tools indicates that they cannot effectively express and manipulate architectural defect-properties. We were thus motivated to build an enhanced Relation Algebra based on existing algebraic work, in order to implement the DAD features in the prototype tool.

A.3 Basic Notions and Notations

A *set* is a collection of objects. Given set S, a (*binary*) *relation* on S is a subset of the *Cartesian product* $\mathsf{S} \times \mathsf{S}$, written $\mathsf{R}_{\mathsf{S}^2}$ (or R, if S is clear) $\subseteq \{<x, y> \mid x, y \in \mathsf{S}\}$.

We also use an infix notation to present a relation: $x\mathsf{R}y$ for $<x, y> \in \mathsf{R}$. Further, we write EM for the *empty* relation $\{\}$, ID for the *identity* relation $\{<x,\, x> \mid x \in \mathsf{S}\}$, UN for the *universe* relation $\mathsf{S} \times \mathsf{S}$, and $\wp(\mathsf{UN})$ for the *power set* of UN, i.e., $\wp(\mathsf{UN}) = \{\mathsf{R} \mid \mathsf{R} \subseteq \mathsf{UN}\}$.

Several basic set operations are defined as follows (see (Ross and Wright, 1988, ch. 2)). We use $\subseteq$ for *inclusion*, $=$ for *equivalence*, $^{-}$ for *complement*, $^{\top}$ for *transpose*, $\cup$ for *union*, $\cap$ for *intersection*, $-$ for (*asymmetric*) *difference*, and $\circ$ for *composition*. Note that R_i and R_j are two example sets, $\Leftrightarrow$ means "if and only if," and $\equiv$ means "equals to".

- $\mathsf{R}_i \subseteq \mathsf{R}_j \Leftrightarrow \forall x \in \mathsf{R}_i\ [x \in \mathsf{R}_j]$.
- $\mathsf{R}_i = \mathsf{R}_j \Leftrightarrow \mathsf{R}_i \subseteq \mathsf{R}_j \wedge \mathsf{R}_j \subseteq \mathsf{R}_i$.
- $\overline{\mathsf{R}} = \{<x,\, y> \mid <x,\, y> \in \mathsf{UN} \wedge <x,\, y> \notin \mathsf{R}\}$.
- $\mathsf{R}^{\top} = \{<y,\, x> \mid <x,\, y> \in \mathsf{R}\}$.
- $\mathsf{R}_i \cup \mathsf{R}_j \equiv \{<x, y> \mid <x, y> \in \mathsf{R}_i \vee <x, y> \in \mathsf{R}_j\}$.
- $\mathsf{R}_i \cap \mathsf{R}_j \equiv \{<x, y> \mid <x, y> \in \mathsf{R}_i \wedge <x, y> \in \mathsf{R}_j\}$.
- $\mathsf{R}_i - \mathsf{R}_j \equiv \{<x, y> \mid <x, y> \in \mathsf{R}_i \wedge <x, y> \notin \mathsf{R}_j\}$.
- $\mathsf{R}_i \circ \mathsf{R}_j \equiv \{<x, z> \mid <x, y> \in \mathsf{R}_i \wedge <y, z> \in \mathsf{R}_j\}$.

A typical Relation Algebra $(\mathsf{R}, \cup, \cap, ^{-}, \circ, ^{\top})$ (Schmidt and Ströhlein, 1993, p. 271) consists of a nonempty set R of relations, such that: for any x, y, and $\mathsf{z} \in \mathsf{R}$, the following, example, axioms hold:

- $(\mathsf{x} \cup \mathsf{y}) \cup \mathsf{z} = \mathsf{x} \cup (\mathsf{y} \cup \mathsf{z})$, $(\mathsf{x} \cap \mathsf{y}) \cap \mathsf{z} = \mathsf{x} \cap (\mathsf{y} \cap \mathsf{z})$.
- $\mathsf{x} \cup \mathsf{y} = \mathsf{y} \cup \mathsf{x})$, $\mathsf{x} \cap \mathsf{y} = \mathsf{y} \cap \mathsf{x}$.
- $\mathsf{x} \cup (\mathsf{x} \cap \mathsf{y}) = \mathsf{x}$, $\mathsf{x} \cap (\mathsf{x} \cup \mathsf{y}) = \mathsf{x}$.
- $\mathsf{x} \subset \mathsf{y} \Rightarrow \mathsf{x} \cup \mathsf{z} \subset \mathsf{y} \cup \mathsf{z}$, $\mathsf{x} \cap \mathsf{z} \subset \mathsf{y} \cap \mathsf{z}$.
- $\mathsf{x} \circ (\mathsf{y} \circ \mathsf{z}) = (\mathsf{x} \circ \mathsf{y}) \circ \mathsf{z}$.
- If $\mathsf{x} \circ \mathsf{y} = \mathsf{y} \circ \mathsf{x} = \mathsf{x}$ and $\mathsf{x} \circ \mathsf{z} = \mathsf{z} \circ \mathsf{x} = \mathsf{x}$, then $\mathsf{y} = \mathsf{z}$.

- For $\mathsf{R} \neq \mathsf{EM}$: $\mathsf{UN} \circ \mathsf{R} \circ \mathsf{UN} = \mathsf{UN}$.
- $\mathsf{x} \circ \mathsf{y} \subset \mathsf{z} \Leftrightarrow \mathsf{x}^{\top} \circ \overline{\mathsf{z}} \subset \overline{\mathsf{y}} \Leftrightarrow \overline{\mathsf{z}} \circ \mathsf{y}^{\top} \subset \overline{\mathsf{x}}$.

The above mentioned notions and notations are used in following sections for description of the approach with a Relation Algebra.

A.4 Extended Relation Algebra for DAD

In this extended Relation Algebra, we define relational structures used specifically to express architectural defect-properties and operations on the relational structures for manipulation. This work is, in part, built upon the work by Feijs-Krikhaar (1998) and Holt (1999) by utilizing the idea of *attribute* arity. In the subsections below, we first describe the architecture relation and the Lifting and Lowering operations for abstracting architectural elements. After that, we extend the architecture relation with defect attributes. Finally, we define the procedure for aggregating defect-properties. The relational structures and operations support expression and manipulation of architectural structures and defects, which is essential to the DAD prototype tool.

A.4.1 Architectural Relation

A software architecture is a composition of architectural entities and relationships between the entities. Example entities are: components, subsystems, and system. And example relationships are: a component contains another one; changes in one component are related to changes in another; and a component invokes another to satisfy an obligation.

An *architectural relation* (AR) is a special ordinary relation where each tuple denotes a relationship between two entities of an architecture. Formally, an architectural relation, written AR, is a subset of the Cartesian product $\mathsf{AES} \times \mathsf{AES}$, where AES denotes the set of entities of an architecture.

For the purpose of expression and manipulation of architectural defects (such as aggregation (Holt, 1999) and change-impact analysis (Arnold and Bohner, 1993)) with the algebra, we focus on two main AR types: *hierarchical* and *dependence* ARs (also see (Feijs et al., 1998)).

- A hierarchical AR represents a partial-order relation between architectural entities. A hierarchical AR is a *parent-of* relation, written AR^{P}. Given $x, y \in \mathsf{AES}$, $<x, y> \in \mathsf{AR}^{\mathsf{P}}$ (i.e., $x\mathsf{P}y$) if and only if x is a parent of (or contains) y, denoting that a subsystem (x) is the parent of a component (y).
- A dependence AR denotes a dependence relation between architectural entities. There are variant dependence AR types such as function-call, data-reference, change-coupling, user, and so on. A dependence AR is a *change-coupling* relation, written AR^{C}. Given $x, y \in \mathsf{AES}$, $<x, y> \in \mathsf{AR}^{\mathsf{C}}$ (i.e., $x\mathsf{C}y$) if and only if changes (insertion, deletion and modification) in x are related to changes in y.

We assume that any two architectural entities (x, y) having parent-of relationships with the same third-party entity (z) must be involved in a parent-of relationship. This means that for any $x\mathsf{P}z$ and $y\mathsf{P}z$ we have $x\mathsf{P}y$ or $y\mathsf{P}x$. We also assume that hierarchical and dependence ARs are *mutually exclusive* except the identity tuples. For example, considering AR^{P} and AR^{C}: if $x\mathsf{P}y$ or $y\mathsf{P}x$ holds, then $x\mathsf{C}y$ and $y\mathsf{C}x$ do not hold; and vice versa. However, there is an exception: any identity tuple $<x, x>$ belongs to both AR^{P} and AR^{C}.

Example 1: Figure A.1 illustrates an architectural graph segment with parent-of and change-coupling relationships. This segment contains a system node (*sys*), two component nodes (c_1 and c_2), and three source file nodes (f_1, f_2 and f_3). The set of architectural entities is:

$$\mathsf{AES} = \{\ sys,\ c_1,\ c_2,\ f_1,\ f_2,\ f_3\ \}.$$

The tuples of the parent-of and change-coupling ARs are labeled with "P" and "C" in the figure, below:

$\mathsf{AR}^{\mathsf{P}} = \{ <sys, c_1>, <sys, c_2>, <c_1, f_1>, <c_1, f_2>, <c_2, f_3> \}.$

$\mathsf{AR}^{\mathsf{C}} = \{ <f_1, f_2>, <f_2, f_3> \}.$

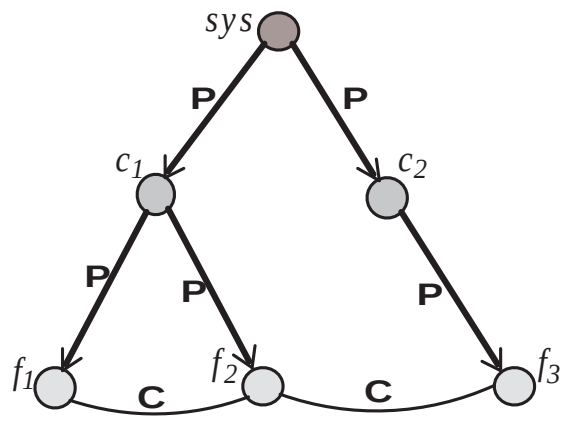


Figure A.1: An example architectural graph (segment).

Later on, we use AR^{P} as a reference hierarchical AR and use AR^{C} as a reference dependence AR. Any operations or constraints applied to AR^{P} or AR^{C} are also applied to hierarchical or dependence ARs, respectively, by default.

On a practical note, we can extract the AR^{P} or AR^{C} tuples by examining the system's architecture and its defect logs. These tuples (that are obtained directly from the source data) are termed *initial* tuples. There is another type, *derived* tuples, that is obtained by applying certain operations on the architecture. These operations and the resultant, derived, tuples are discussed in the next subsection.

A.4.2 AR Lifting and Lowering

Since the parent-of relation AR^{P} is transitive, we can infer that: If $x\mathsf{P}y$ and $y\mathsf{P}z$, then $x\mathsf{P}z$. In a sense, $x\mathsf{P}z$ is a *lifting* (i.e., abstraction) from $y\mathsf{P}z$, and $x\mathsf{P}z$ is also a *lowering* (i.e., refinement) from $x\mathsf{P}y$. For example, considering nodes sys, c_1 and f_1 in Figure A.1. It is known that $sys\mathsf{P}c_1$ and $c_1\mathsf{P}f_1$ (see Example 1). Therefore, we can say that: $c_1\mathsf{P}f_1$ induces $sys\mathsf{P}f_1$ by lifting, and $sys\mathsf{P}c_1$ induces $sys\mathsf{P}f_1$ by lowering. Through iteratively lifting and lowering the initial AR^{P}, a *complete* AR^{P} can be constructed.

This transition based lifting and lowering situation cannot apply to the change-coupling relation $\mathsf{AR^C}$, because it is non-transitive. However, in Relation Partition Algebra (RPA) (Feijs et al., 1998), special Lifting and Lowering operations are defined to transform $\mathsf{AR^C}$ at a certain level to that at a higher or lower level. Likewise, we define these two operations in order to construct a complete $\mathsf{AR^C}$.

Definition 1 (Lifting and Lowering on $\mathsf{AR^C}$) *Lifting* on $\mathsf{AR^C}$ has two types: *left* and *right lifting*. Given $x\mathsf{C}y$:

- Left lifting: If $\exists\ w$, s.t. $<w, x> \in \mathsf{AR^P}$, and $<w, y>$, $<y,\ w> \notin \mathsf{AR^P}$, then $<w, y> \in \mathsf{AR^C}$. The operation of deriving $w\mathsf{C}y$ from $x\mathsf{C}y$ is termed *left lifting*, written ${}^{\uparrow}(x\mathsf{C}y) = w\mathsf{C}y$;
- Right lifting: If $\exists\ z$, s.t. $<z, y> \in \mathsf{AR^P}$, and $<x, z>$, $<z, x> \notin \mathsf{AR^P}$, then $<x, z> \in \mathsf{AR^C}$. The operation of deriving $x\mathsf{C}z$ from $x\mathsf{C}y$ is termed *right lifting*, written $(x\mathsf{C}y)^{\uparrow} = x\mathsf{C}z$.

Similarly, the *Lowering* operation on $\mathsf{AR^C}$ also has two types: *left* and *right lowering*. Given $x\mathsf{C}y$:

- Left lowering: If there exists a non-empty set $\tau = \{w \mid x\mathsf{P}w\}$, then there must exist at least one $w \in \tau$, $<w, y> \in \mathsf{AR^C}$. The operation of deriving $w\mathsf{C}y$ from $x\mathsf{C}y$ is termed *left lowering*, written $w\mathsf{C}y \in {}^{\downarrow}(x\mathsf{C}y)$;
- Right lowering: If there exists a non-empty set $\tau = \{z \mid y\mathsf{P}z\}$, then there must exist at least one $z \in \tau$, $<x, z> \in \mathsf{AR^C}$. The operation of deriving $x\mathsf{C}z$ from $x\mathsf{C}y$ is termed *right lowering*, written $x\mathsf{C}z \in (x\mathsf{C}y)^{\downarrow}$.

Note that the lowering operation is related to the lifting operation. For example, $w\mathsf{C}y \in {}^{\downarrow}(x\mathsf{C}y)$ holds if and only if ${}^{\uparrow}(w\mathsf{C}y) = x\mathsf{C}y$ holds. Likewise, $\{w\mathsf{C}y\} = {}^{\downarrow}(x\mathsf{C}y)$ holds if and only if ${}^{\uparrow}(w\mathsf{C}y) = x\mathsf{C}y$ and $\nexists z : z \neq w \wedge {}^{\uparrow}(z\mathsf{C}y) = x\mathsf{C}y$ hold.

Example 2: Considering three relationships (AR tuples) in Figure A.1: $c_1\mathsf{P}f_2$, $f_2\mathsf{C}f_3$, and $c_2\mathsf{P}f_3$. First, new tuple $c_1\mathsf{C}f_3$ is derived by left lifting $f_2\mathsf{C}f_3$. Second,

new tuple $c_1\mathsf{C}c_2$ is derived by right lifting $c_1\mathsf{C}f_3$. In brief, $\uparrow(f_2\mathsf{C}f_3) = c_1\mathsf{C}f_3$ and $(c_1\mathsf{C}f_3)\uparrow = c_1\mathsf{C}\ c_2$. We can also infer that $\downarrow(c_1\mathsf{C}f_3) = \{f_2\mathsf{C}f_3\}$ and $(c_1\mathsf{C}\ c_2)\downarrow = \{c_1\mathsf{C}f_3\}$. Note that *sys* $\mathsf{C}f_3$ does not hold because of *sys* $\mathsf{P}f_3$.

An architectural relation that contains both initial tuples and all derived tuples is claimed *complete*. Formally:

Definition 2 (AR Completeness) There are two cases:

- AR^{P} is *complete*, if and only if for any $x\mathsf{P}y$ and $y\mathsf{P}z$: $<x, z> \in \mathsf{AR}^{\mathsf{P}}$.
- AR^{C} is *complete*, if and only if for any $x\mathsf{C}y$: $\{^{\uparrow}(x\mathsf{C}y)\} \cup \{(x\mathsf{C}y)^{\uparrow}\} \subseteq \mathsf{AR}^{\mathsf{C}}$.

Note that a complete AR^{C} also indicates that any $x\mathsf{C}y$: $^{\downarrow}(x\mathsf{C}y) \cup (x\mathsf{C}y)^{\downarrow} \subseteq \mathsf{AR}^{\mathsf{C}}$, because, as said above, the AR lowering operation is related to the AR lifting operation.

Example 3: Considering the initial AR^{P} and AR^{C} shown in Figure A.1 (see Example 1), we can construct the corresponding complete ones below. First, the following new parent-of tuples are derived from the initial AR^{P}:

$$sys\ \mathsf{P}f_1;\ \ sys\ \mathsf{P}f_2;\ \text{ and } sys\ \mathsf{P}f_3.$$

These derived tuples, together with the initial ones, comprise the complete AR^{P} over Figure A.1. Second, the following new change-coupling tuples are derived from the initial AR^{C} and the complete AR^{P}:

$$^{\uparrow}(f_2\mathsf{C}f_3) = c_1\mathsf{C}f_3;\ \ (f_2\mathsf{C}f_3)^{\uparrow} = f_2\mathsf{C}\ c_2;\ \text{ then}$$

$$^{\uparrow}(f_2\mathsf{C}\ c_2) = (c_1\mathsf{C}f_3)^{\uparrow} = c_1\mathsf{C}\ c_2.$$

Likewise, the above derived tuples, together with the initial ones, comprise the complete AR^{C} over Figure A.1.

A.4.3 Extended Architectural Relation

The core relational structure of expressing architectural defect-properties is termed the *extended architectural relation* (EAR). An EAR relation adds an *attribute* arity to each tuple in an architectural relation.

Here, an attribute is defined specifically as a *set* of defects pertaining to an architectural entity or relationship. For example, the defect set of a change-coupling relationship between two components x and y is written as $\mathsf{D}_{<x,y>} = \{d_1, \cdots, d_i, \cdots, d_n\}$ where d_i (i=1..n) denotes a defect involving this change-coupling relationship.

Definition 3 (EAR) An EAR built upon an AR and a set Att of defect attributes, written EAR, is a total function: $\mathsf{AR} \rightarrow \mathsf{Att}$. A EAR tuple is thus written as $<x, y, \mathsf{D}_{<x,y>}>$, where $<x, y>$ is an AR tuple and $\mathsf{D}_{<x,y>}$ is the defect set pertaining to this tuple.

In the approach, EARs are used to express architectural structures (AR) and defect attributes (Att) . Later, we define operations for manipulating the architectural structure (AR) and defect attribute (Att) parts of EARs.

As we know *parent-of* and *change-coupling* ARs (see Section A.4.1), we can thus define parent-of and change-coupling EARs based on the according ARs, written $\mathsf{EAR}^{\mathsf{P}}$ and $\mathsf{EAR}^{\mathsf{C}}$, respectively. Further, given AR and its derived EAR, we call AR the *base* of EAR, written $\lfloor \mathsf{EAR} \rfloor = \mathsf{AR}$. Likewise, we write $\lceil \mathsf{AR} \rceil_{\mathsf{Att}}$ (or $\lceil \mathsf{AR} \rceil$, if Att is clear) for EAR (with Att).

Example 4: Supposing that there are two example $\mathsf{EAR}^{\mathsf{C}}$ tuples in Figure A.1:

identity tuple $<f_1, f_1, \{d_1, d_2, d_3\}>$, and

change-coupling tuple $<f_1, f_2, \{d_3\}>$.

The first tuple indicates that file f_1 has three defects: d_1, d_2 and d_3. The second tuple indicates that defect d_3 requires fixes in both files f_1 and f_2, therefore involving tuple $f_1 \mathsf{C} f_2$.

Two EARs can be compared (e.g., inclusion "$\subseteq$" and equivalence "$=$") based on their tuples, as below.

Definition 4 (EAR $\subseteq$ and $=$) Given two EAR tuples: $t_i = <a, b, \mathsf{D}_{<a,b>}>$ and $t_j = <c, d, \mathsf{D}_{<c,d>}>$, we define "*less-than*" ($\preceq$) and *equivalence* ($=$) between t_i and t_j as:

- $t_i \preceq t_j \Leftrightarrow a = c \ \wedge \ b = d \ \wedge \ \mathsf{D}_{<a,b>} \subseteq \mathsf{D}_{<c,d>}$.
- $t_i = t_j \Leftrightarrow a = c \ \wedge \ b = d \ \wedge \ \mathsf{D}_{<a,b>} = \mathsf{D}_{<c,d>}$.

Hence, given two EARs: EAR_i and EAR_j, the EAR comparisons ("$\subseteq$" and "$=$") are defined as follows:

- $\mathsf{EAR}_i \subseteq \mathsf{EAR}_j \Leftrightarrow \forall t_i \in \mathsf{EAR}_i[\exists t_j \in \mathsf{EAR}_j[t_i \preceq t_j]]$.
- $\mathsf{EAR}_i = \mathsf{EAR}_j \Leftrightarrow \mathsf{EAR}_i \subseteq \mathsf{EAR}_j \wedge \mathsf{EAR}_j \subseteq \mathsf{EAR}_i$.

Further, we define the ordinary relational operations (e.g., $\cup$, $\cap$, and $-$) for the EAR structure below.

Definition 5 (Combinational EAR Operations) Given EAR_i and EAR_j, a combinational EAR operation, written $\Box^{\triangle}$, is a combination of two operations $\Box$ and $\triangle$: The first step is $\Box$ between $\lfloor \mathsf{EAR}_i \rfloor$ and $\lfloor \mathsf{EAR}_j \rfloor$. The second step is $\triangle$ between Att_i (i.e., the Att of EAR_i) and Att_j (i.e., the Att of EAR_j). The operation $\Box^{\triangle}$ is defined as:

$$\mathsf{EAR}_i \ \Box^{\triangle} \ \mathsf{EAR}_j \equiv \lceil \ \lfloor \mathsf{EAR}_i \rfloor \Box \lfloor \mathsf{EAR}_j \rfloor \ \rceil_{\mathsf{Att}_i \triangle \mathsf{Att}_j}.$$

For example, two concrete combinational EAR operations, $\cap^{\cup}$ and $\cup^{\cap}$, are below:

- $\mathsf{EAR}_i \cap^{\cup} \mathsf{EAR}_j \equiv \{<x, y, \mathsf{D}_{<x,y>} \cup \mathsf{D}'_{<x,y>}> \mid$
  $<x, y, \mathsf{D}_{<x,y>}> \in \mathsf{EAR}_i \wedge <x, y, \mathsf{D}'_{<x,y>}> \in \mathsf{EAR}_j\}$.
- $\mathsf{EAR}_i \cup^{\cap} \mathsf{EAR}_j \equiv \{<x, y, \mathsf{D}_{<x,y>} \cap \mathsf{D}'_{<x,y>}> \mid$
  $<x, y, \mathsf{D}_{<x,y>}> \in \mathsf{EAR}_i \vee <x, y, \mathsf{D}'_{<x,y>}> \in \mathsf{EAR}_j\}$.

Other combinational operations can be defined similarly, including $\cap^{\cap}$, $\cap^{-}$, $\cup^{\cup}$, $\cup^{-}$, $-^{\cap}$, $-^{\cup}$, and $-^{-}$.

These combinational EAR operations are used to manipulate EARs for varying purposes such as defect aggregation and comparison among architectural elements.

A.4.4 Attribute Aggregation

The attributes of different architectural entities may have some special relationships. For example, we can infer that, for tuple $x\mathsf{P}y$ (x is a parent of y), defects

occurred in y ($\mathsf{D}_{<y,y>}$) also occurred in x ($\mathsf{D}_{<x,x>}$), i.e., $\mathsf{D}_{<y,y>} \subseteq \mathsf{D}_{<x,x>}$. However, it is obvious that this simple reasoning method does not fit change-coupling type tuples such as $x\mathsf{C}y$. To span this deficiency, we define *attribute aggregation* to reason how attributes are aggregated over $\mathsf{AR^P}$ or $\mathsf{AR^C}$ tuples, below.

Definition 6 (Attribute Aggregation) Given EAR ($\mathsf{AR} \rightarrow \mathsf{Att}$), the members in Att pertaining to any tuples in AR are aggregated in two ways:

- Considering $<x, x> \in \mathsf{AR}$, the attribute of $<x, x>$, written $\mathsf{D}_{<x,x>}$, is a *union* ($\cup$) of the attributes of the entities contained by x, i.e., $\mathsf{D}_{<x,x>} = \bigcup_{\forall y[x\mathsf{P}y]} \mathsf{D}_{<y,y>}$.
- Considering $<x, y>$ ($x \neq y$) $\in \mathsf{AR}$, the attribute of $<x, y>$, written $\mathsf{D}_{<x,y>}$, is an *intersection* ($\cap$) of the attributes of the two connecting entities, i.e., $\mathsf{D}_{<x,y>} = \mathsf{D}_{<x,x>} \cap \mathsf{D}_{<y,y>}$.

Example 5: Considering Figure A.1 where component c_1 contains files f_1 and f_2. Supposing that these two EAR tuples hold: $<f_1, f_1, \{d_1, d_2, d_3\}>$ and $<f_2, f_2, \{d_3, d_4\}>$. According to Definition 6, these two EAR tuples hold:

$$<c_1, c_1, \{d_1, d_2, d_3, d_3, d_4\}>; \quad <f_1, f_2, \{d_3\}>.$$

With the above definitions of AR completeness (Definition 2) and attribute aggregation (Definition 6), we can set up the criteria for a complete EAR.

Definition 7 (EAR Completeness) An EAR (no matter of the type) is *complete*, if and only if two conditions hold:

(i) The AR base is complete (see Definition 2);

(ii) Each tuple's attribute satisfies Definition 6.

A.5 An Example Application

Following the description of the approach to expressing and manipulating architectural defect-properties in the previous section, we describe an example application of this approach – architectural degeneration measurement.

The architectural degeneration has many symptoms (e.g., increasingly excessive size and complexity of system entities (Eick et al., 2001), architectural deviation (or departure) (Hochstein and Lindvall, 2005), etc.). Here, we elaborate on the second symptom posed in Section 5.1 – "the MCDs are more and more complex to fix". MCDs are defects requiring fixes in more than one component in the system. In particular, we choose the MCD-complexity metric **M3** – "#Components fixed per MCD" – to measure the components in the subject system of Case Study 2 (see Section 6.2.1).

Note that MCDs involve change-coupling relationships among components (i.e., fix relationships). Therefore, we can count the average *span* of a fix relationship (called SPAN) – the average number of components changed in order to fix a MCD in a specific component – to calculate the **M3** measurement for this component. There are two main steps in this calculation with the algebraic approach described in above Section A.4.

Step 1: *defined the change-coupling extended architectural relation* ($\mathsf{EAR^C}$) *for each release of the system.* This coincides with Step 1 of the DAD approach – defect architecture construction (see Section 5.2.1).

Example 6: Figure A.2 illustrates a component-level segment of the $\mathsf{EAR^C}$ diagram of the first release of the system. In the figure, the weight on a node represents the quantity of MCDs in the component. Likewise, the weight on an edge between two given nodes represents the number of MCDs pertaining to the two connecting components. The two red-colored nodes denote the top two MCD-prone components (C5 and C6) in the system. The figure shows that there are 133 MCDs in component C5, 103 MCDs in component C6, and 83 MCDs involving the fix relationship E0 (between C5 and C6). Thus, for node C5, the corresponding tuple in $\mathsf{EAR^C}$ is defined as:

$$< C5, C5, \{d^1_{<C5,C5>}, \cdots, d^i_{<C5,C5>}, \cdots, d^{133}_{<C5,C5>}\} > ,$$

where $d^i_{<C5,C5>}$ is one of the 133 defects associated with components C5. Likewise, for edge E0, the corresponding tuple is defined as:

$$< C5, C6, \{d^1_{<C5,C6>}, \cdots, d^i_{<C5,C6>}, \cdots, d^{103}_{<C5,C6>}\} > ,$$

where $d^i_{C5,C6}$ is one of the 103 relationship defects.

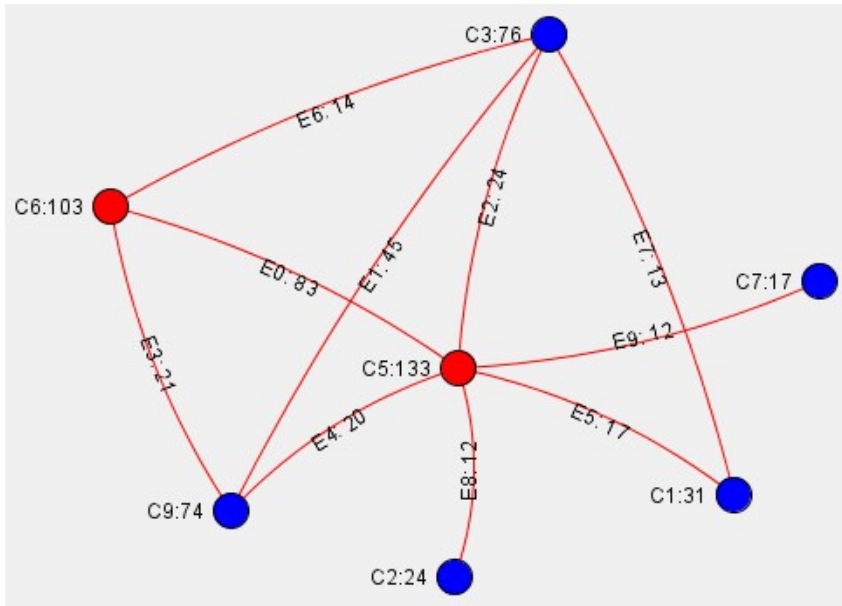


Figure A.2: A segment of the $\mathsf{EAR}^{\mathsf{C}}$ diagram for release 1.

Step 2: *measured the average span of fix relationships in the whole system over time*, as below. This coincides with Step 2 of the DAD approach – defect architecture measurement (see Section 5.2.2).

$$\mathsf{SPAN} = 1 + \frac{1}{m}\sum_{i=1}^{m}\frac{|\bigcup_{1\le j\le k_i}\mathsf{ATT}(c_i\ \mathsf{C}\ c_j)|}{|\mathsf{ATT}(c_i)|},$$

where m is the number of components in the system, k_i is the number of components that have fix relationships with the ith component (i.e., c_i), and $\mathsf{ATT}(c_i\ \mathsf{C}\ c_j)$ denotes the attribute set of tuple $c_i\ \mathsf{C}\ c_j$, likewise for $\mathsf{ATT}(c_i)$. The greater the SPAN value, the greater the average span of fix relationships in the system.

Example 7: Consider Figure A.2 as an example. Two steps of calculating the SPAN value follow. First, calculate the average span of fix relationships for each

component. For example, component C5 has six fix relationships: E0, E2, E4, E5, E8, and E9. The average span of these relationships for C5 is:

$$1 + (83+24+20+17+12+12)/133 \approx 2.26.$$

This means that an average of 2.26 components are involved in fixing a MCD in component C5. Similarly, for component C6, the span is:

$$1 + (83+21+14)/103 \approx 2.14.$$

The span for other components can be calculated similarly[1]. After calculating the average span of fix relationships for each component shown in Figure A.2, we then calculate the average span, SPAN, for all the components in the system, by averaging the span values for each component. This, in fact, gives us the measurement of the span value for one system.

Taking the difference between the span values of the given three releases of the system would identify whether or not the system has degenerated. We found that the SPAN value is 2.28 for release 1, 2.45 for release 2 (increased by about 7%), and 2.21 for release 3 (decreased by about 9%). This coincides with the findings shown in Table 6.4 (row "#Components fixed per MCD (**M3**)"). Further, it indicates that the architectural degeneration increased as the system evolved from release 1 to release 2 but decreased late in release 3, with respect to the average span of a fix relationship in the architecture, see Section 6.3.5 for details.

A.6 Algebra Implementation in the Tool

The approach to expressing and manipulating both architectural structures and defects (shown in Section A.4) has been implemented in the DAD prototype tool.

[1]For a given component (e.g., C2), the total number of MCDs on the fix relationships of that component will be greater than or equal to the number of MCDs in the component. However this figure only shows the relationships among the components that the system contains. The system is a part of the whole, much larger, system. There are other relationships between C2 and other components that are outside the system. These relationships are not shown in Figure A.2. Thus, for C2, the comparison "$12 < 24$" is only a partial view.

Here, we describe the main implementation mechanisms of the approach in the prototype tool, focusing on: (a) construction of complete extended architectural relations (EARs) – Section A.6.1; (b) implementation of EAR tuples in a relational database – Section A.6.2; and (c) implementation of the EAR operations such as Lifting, Lowering, and others – Section A.6.3.

A.6.1 Complete EAR Construction

Constructing complete EARs is critical for analysis of architectural defect-properties. Definition 7 has given the criterion to check if an AR is complete. Here we give an algorithm to construct a complete EAR according to this criterion; see Table A.1.

Table A.1: Algorithm of constructing complete $\mathsf{EAR}^{\mathsf{P|C}}$.

Input: initial $\mathsf{AR}^{\mathsf{P|C}}$
Output: complete $\mathsf{AR}^{\mathsf{P|C}}$ and $\mathsf{EAR}^{\mathsf{P|C}}$
Process:
(1) Do // *for complete* AR^{P}
(2) FOR any two tuples $<x, y>$, $<y, z> \in \mathsf{AR}^{\mathsf{P}}$
(3) add $<x, z>$ into AR^{P};
(4) UNTIL no new tuple can be added to AR^{P}
(5) DO // *for complete* AR^{C}
(6) FOR any tuple $<x, y> \in \mathsf{AR}^{\mathsf{C}}$
(7) add $\uparrow(<x, y>)$ into AR^{C};
(8) add $(<x, y>)\uparrow$ into AR^{C};
(9) UNTIL no new tuple can be added to AR^{C}
(10) FOR any tuple $<x, y> \in \mathsf{AR}^{\mathsf{P|C}}$
(11) add $<x, y, \{\}>$ into $\mathsf{EAR}^{\mathsf{P|C}}$;
// create a placeholder for the attributes of $<x, y>$.
(12) update $\mathsf{D}_{<x,y>}$ for any lowest-level $<x, y>$ in $\mathsf{EAR}^{\mathsf{P|C}}$;
// *this is to initialize* $\mathsf{EAR}^{\mathsf{P|C}}$
(13) Do // *attribute aggregation*
(14) FOR any tuple $<x, y, \mathsf{D}_{<x,y>}> \in \mathsf{EAR}^{\mathsf{P|C}}$
(15) aggregate $\mathsf{D}_{<x,y>}$ according to Definition 6;
(16) UNTIL no new attribute aggregations can occur

Note that initial AR tuples are usually at the lowest level of the architecture. For example, we can find initial change-coupling tuples at the file level; then we

derive new component-level tuples based on these tuples for completeness. The EAR initialization (Line 12) is to fill up the attributes for the lowest-level EAR tuples, which can be extracted from the defect records of the system. The defect properties in upper-level entities such as components and subsystems can be then aggregated from that in the lower-level entities such as source files (Lines 13–16).

A.6.2 EAR Structure Implementation

We describe the implementation of the EAR structure in the DAD prototype tool, see Figure A.3 (a, b and c). Figure A.3a represents the entity hierarchy tree for the system. Figure A.3b shows defect information linked to the respective files in Figure 3a; for example, file f_1 contains three defects (d_1, d_2 and d_3). Figure A.3c shows the defect records. The first defect record in Figure A.3c indicates that defect d_1 was located in file f_1; it was medium-severity and it required 5 changes for the defect to be fixed.

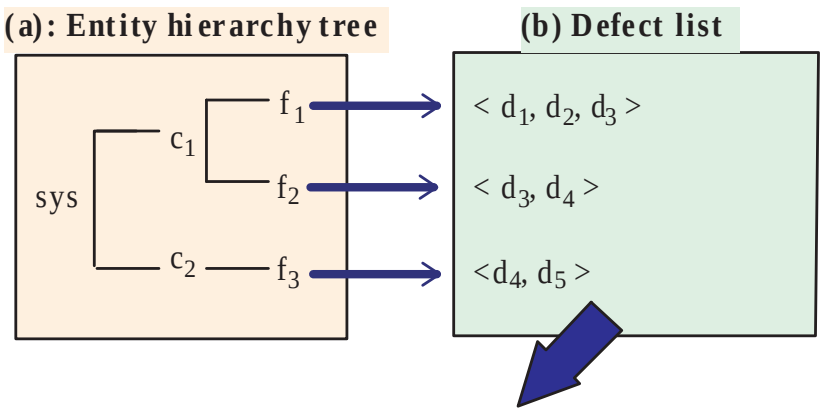
In the DAD prototype tool, this EAR structure was implemented in an ordinary relational database systems MySQL[2]. In MySQL, relational tables can record architectural structures, defect properties, and links between them. The Structured Query Language (SQL) is then used to query and manipulate these relational tables for the support of architectural degeneration diagnosis.

A.6.3 EAR Operation Implementation

Based on the above EAR structure implementation, the combinational operations (see Definition 5) between two EARs could be implemented by combining the corresponding operations on the AR parts and the attribute parts of the EARs. In particular, there are three steps in EAR operation implementation:

Step 1: identify the defect list(s) for each architectural entity involved in an EAR. As shown in Figure A.3, only files (denoted by nodes f_1, f_2 and f_3) have links to

[2]See the website of MySQL: `www.mysql.com` (last access in November 2010).



(c) Defect-fix records (procssed)

Def. ID	Location	Severity	#Changes
d_1	f_1	medium	5
d_2	f_1	medium	1
d_3	f_1, f_2	high	3
d_4	f_2, f_3	low	15
d_5	f_3	high	8

Note: "Def. ID" denotes the defect identity; "Location" denotes where this defect occurs; "Severity" denotes the defect's severity; and "#Changes" denotes the number of changes required to fix the defect.

Figure A.3: EAR structure implementation.

the defect lists. The linked lists of other higher-level entities must be aggregated from those at the lower level. For example, the defect list of c_1 is $\{<d_1, d_2, d_3, d_4\}$, which is the aggregation of the defect lists of f_1 and f_2.

Step 2: Identify the defect list for an EAR tuple by intersecting the defect lists of the respective two entities in the tuple. For example, the defect list of $f_1 \mathsf{C} f_2$ is $\{d_3\}$ (via intersection of f_1 and f_2), specified as $<f_1, f_2, \{d_3\}>$.

Step 3: Operate EARs and their defect lists together. For example, considering two simple EARs:

$$\mathsf{EAR}_i = \{ <f_1, f_2, \{d_3\}>, \ <f_2, f_3, \{d_4, d_5\}> \};$$
$$\mathsf{EAR}_j = \{ <f_1, f_2, \{d_3, d_4\}>, \ <f_2, f_3, \{d_4\}> \}.$$

We can infer that (see Definition 5):

$$\mathsf{EAR}_i \cup^{\cap} \mathsf{EAR}_j = \{ <f_1,\, f_2,\, \{d_3\}>,\ <f_2,\, f_3,\, \{d_4\}> \}.$$

Within MySQL, the ordinary relational operations can be used to implement the above EAR operations. Overall, built upon the implementation of algebraic expression and manipulation of architectural defect-properties, the DAD prototype tool supports diagnosis of the architectural degeneration over time for a give software system. Later Appendix B will demonstrate example outputs of this prototype tool on the subject system of Case Study 2.

A.7 Discussion and Comparison

The approach, as described in Sections A.4 and A.5, uses Relation Algebra to express and manipulate both architectural structures and defect-properties. It utilizes an attribute arity (similar to the work by Feijs-Krikhaar (1998) and Holt (1999)) to express defects for architectural entities (e.g., components) and relationships (e.g., change-coupling). This is supported with the extended architectural relation (EAR) structure; see Definition 3. An EAR is a mapping from a binary relation (representing an architectural structure) to an attribute set (representing a type of architectural defect-property). Further, the EAR operations (see Definition 5) supports manipulation of these defects together with the architectural structures (entities and relationships).

Actually, there are probably more attributes arities bound to an architectural relation because a software architecture can have more attributes than the defect-aspect, e.g., change and correction-cost. Therefore, the EAR structure (see Definition 3) can be generalized to an n-ary ($n > 3$) structure:

$$\mathsf{AR} \rightarrow \mathsf{Att}_1 \times \mathsf{Att}_2 \times \cdots \times \mathsf{Att}_{n-2}.$$

Obviously, this n-ary EAR structure should be more expressive than the current 3-ary structure. And it is viable to define this n-ary EAR structure with

MySQL (just like that for the 3-ary structure; see Section A.6.2).

Feijs and Krikhaar (1998) propose a multi-relation theory by adding a multiplicity (i.e., a numeric attribute) to each tuple in a relation and generalizing ordinary relation operations to multi-relation operations. Such a theory can thus support aggregation of numeric attributes. Holt also defines a numeric arity (termed *attribute*) bound to architectural relations (e.g., $<x, y, n>$). This structure supports attribute aggregation, which is equivalent to that in Feijs-Krikhaar's theory. However, the single numeric arity n in these two techniques was not meant to (and hence cannot) express the set of individual defects $(d_1, \cdots, d_i, \cdots, d_n)$ associated with the components x and y, as represented in the approach. Thus, it is possible to use the representation to perform cross-longitudinal analysis of defects; for example, to assess whether or not defect d_i crosses a phase boundary or exists in more than one component. Such analysis is clearly useful for the diagnosis of architectural degeneration over time.

Note, however, that in the approach the cross-release analysis is carried out procedurally (using the operation "difference" - see Section A.4.3); whereas, cross-component analysis is inherent in the tuple specification (i.e., x and y in a tuple denote two components involved in a defect relationship). One can thus argue the demerits of a procedural operation such as "difference" in the approach. Would it be beneficial, for example, to capture the inter-release properties within a tuple, thereby integrating the difference operation more tightly in the extended algebra itself than is currently possible in the approach? Another interesting question is whether temporal operations can be included in the approach to deal with the timing of the defects (e.g., when secondary defects were introduced). These issues are currently being investigated.

We have compared the work by Feijs-Krikhaar (1998) and Holt (1999) here, but not others such as component algebra (Bergstra et al., 1990) (Feijs and Qian, 2002) and connector algebra (Allen and Garlan, 1994) (Wermelinger and Fiadeiro,

1998) (see Section A.2.2), because RPA (by Feijs-Krikhaar) and Grok (by Holt) are both closely related to the approach. The component algebra and connector algebra focus on algebraic specification of software architectures. The approach complements these works.

A.8 Short Conclusion

Just as a Relation Algebra has proved to be invaluable for operating on the structure of an architecture (e.g., through operations such as abstraction, union and intersection) (Feijs and Krikhaar, 1998) (Holt, 1999), so it can be used fruitfully for the visualization, aggregation and analysis of architectural "defects". This, complementary view, is fundamental to the approach to expressing and manipulating architectural defect-properties with Relation Algebras. This Relation Algebra defines an extended architectural relation structure and corresponding operations to support expression and manipulation of both architectural structures and defects. It has been implemented with MySQL in the DAD prototype tool (see Section 5.3 and Appendix B).

Appendix B

DAD Prototype Tool Demonstration

In this chapter, we demonstrate the DAD prototype tool (as described in Chapter 5.3) on two software systems: (1) the commercial system under investigated in Case Study 2 (see Section 6.2.2); and (2) the open-source Eclipse Platform[1]. In particular, we present several charts created by the prototype tool, which illustrate: descriptive system statistics, component measure, architectural degeneration trend, and defect architectures for the two systems.

Note that because we do not have any process data or historical accounts of the Eclipse Platform, we cannot interpret the relevant findings shown here (with the DAD prototype tool). For example, we cannot explain why the architectural degeneration increased or decreased as the Eclipse Platform evolved release upon release (due to lacking of the data about the historical evolution process of this system). However, for the commercial system, we can interpret some of the relevant findings here in the context of Case Study 2.

[1] "The Eclipse Platform provides the core frameworks and services upon which all plug-in extensions are created. It also provides the runtime in which plug-ins are loaded, integrated, and executed. The primary purpose of the Platform is to enable other tool developers to easily build and deliver integrated tools." (see a detailed introduction to the Eclipse Platform at `http://www.eclipse.org/platform/` (last access in November 2010)).

B.1 Descriptive System Statistics

First of all, this DAD prototype tool can build a descriptive profile for a given system and its components. Such a profile can help understand the system.

First, Figure B.1 shows a bar chart which illustrates the numbers of defects in the seven releases of the Eclipse Platform (releases 1, 2, 2.1, 3, 3.1, 3.2, and 3.3). From this chart, we can easily find that there are over 2,750 defects discovered in release 3, which is many more than the number of defects in any other release. Note that the number of defects discovered in a given release is related to the time range of this release under development, testing and maintenance. For example, our finding shows that the time range is about half year for release 1 but over three years for releases 3, 3.1 and 3.2.

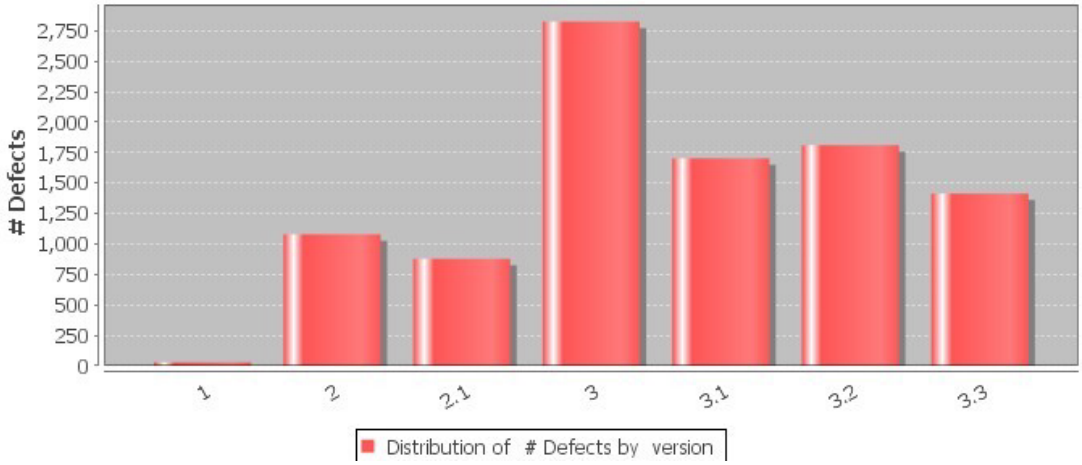


Figure B.1: Numbers of defects in the Eclipse Platform.

Second, Figure B.2 shows a bar chart which illustrates the numbers of code files in the seven components (Ant, Core, CVS, Debug, SWT, Team, and UI) of the Eclipse Platform (release 1). This chart indicates that the defect distribution by the components is highly skewed. Obviously, components SWT and UI contains many more defects than the other components. This is related to the size of each component. For example, our finding shows that 29% (23/79) and 46% (36/79) of code files of release 1 were contained by SWT and UI.

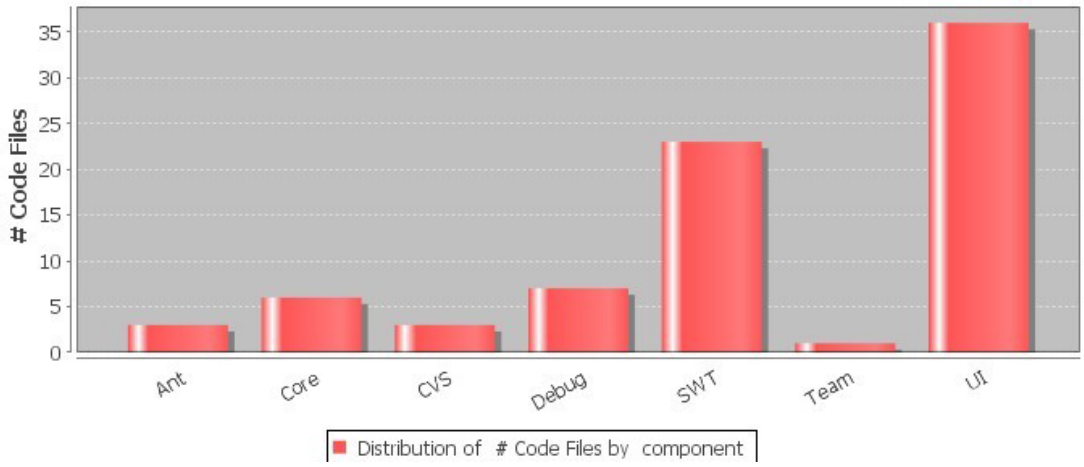


Figure B.2: Numbers of defects in the Eclipse Platform (release 1).

Third, Figure B.3 shows a bar chart which illustrates the numbers of code files fixed in the four phases of the three releases (r1, r2 and r3) of the commercial system of Case Study 2: functional verification testing (FVT), system verification testing (SVT), performance quality assurance (PQA), and field ("_Field") phases. Note that for the subject system, SVT usually succeeds FVT, PQA could happen in parallel with FVT and SVT, but PQA usually succeeds SVT. The FVT, SVT and PQA phases are included in the Internal phase (see Section 6.2.2).

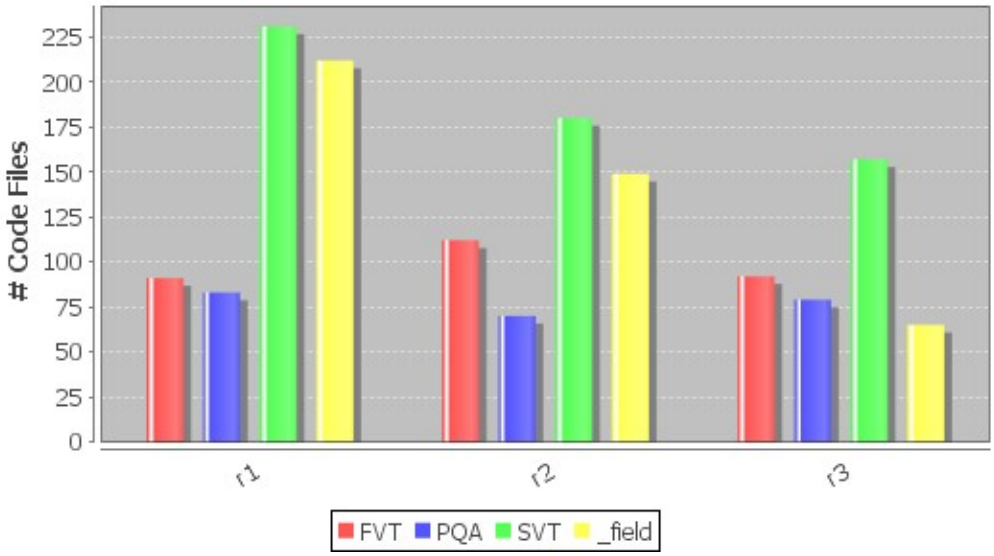


Figure B.3: Numbers of code files fixed in the commercial system.

Figure B.3 shows that in comparison against the FVT and PQA phases, there are more code files fixed during the SVT and field phases (except the field phase of r3). This indicates that defects discovered during the SVT and field phases spread over the system much more widely than defects in the FVT and PQA phases. This figure also shows that as the system evolved from r1 to r3, the numbers of code files fixed in the SVT and field phases decreased substantially but the number of code files fixed in the FVT and PQA phases kept relatively stable. This indicates that the defects discovered during the SVT and field phases tend to be concentrated in a few areas of the system. However, this could be also related to the number of defects (approx. 1100, 550, and 600) investigated in the three releases (see Section 6.2.1).

B.2 Component Measurements

Recall from Section 5.2.2 that the DAD approach defines a suite of metrics to measure components of a given system. Here, we demonstrate examples of component measures in the two subject systems; see Section B.2.1. We also demonstrate the trend in architectural degeneration of the two systems in Section B.2.2.

B.2.1 Degeneration-Critical Components

Recall Sections 5.2.3 and 6.3.1 that degeneration-critical components are identified based on the MCD quantity and complexity measures of the components in a given system. For example, Figure B.4 shows the quantity of MCDs which are involved in the 14 components of the Eclipse Platform (release 3). We can easily find from this figure that there are three components which contain many more MCDs than other components. These three components are CVS (over 60 MCDs), Team (approx. 60 MCDs) and UI (near 50 MCDs), which can be thus considered degeneration-critical from the MCD quantity perspective. Note that, similar to Figure B.1 above, the quantity of MCDs in a component is related to the size of

this component. In particular, our finding shows that in release 3, 9% (289/3385), 7% (220/3385), and 37% (1257/3385) of code files are contained by components CVS, Team and UI, respectively.

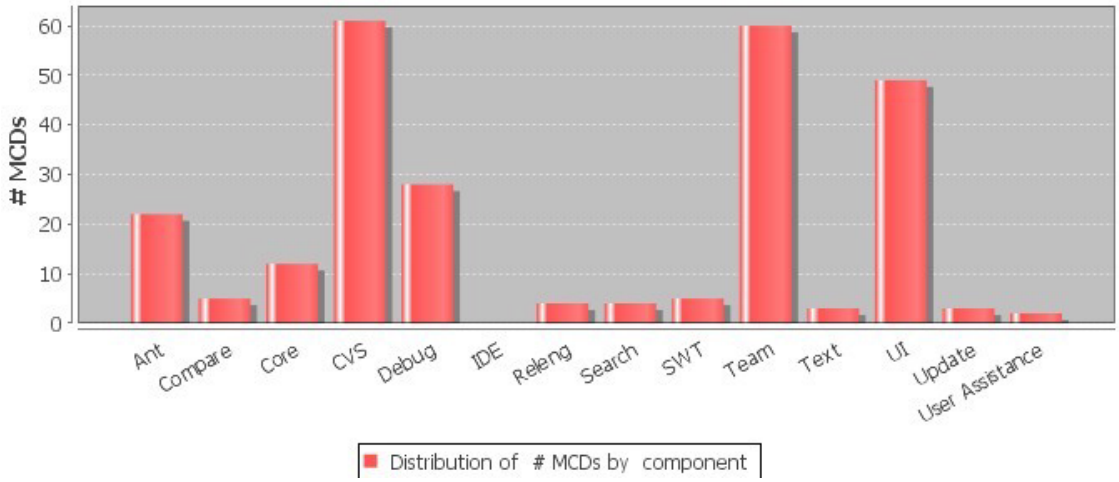


Figure B.4: Number of MCDs in components of Eclipse Platform (release 3).

Second, Figure B.5 illustrates the MCD complexity (**M3** – "#Components fixed per MCD") measures for the 10 components in the three releases (r1, r2 and r3) of the commercial system of Case Study 2. It shows that the distribution of the **M3** measures has a low degree of dispersion. This coincides with Table 6.4 (see column "#Components fixed per MCD (**M3**)").

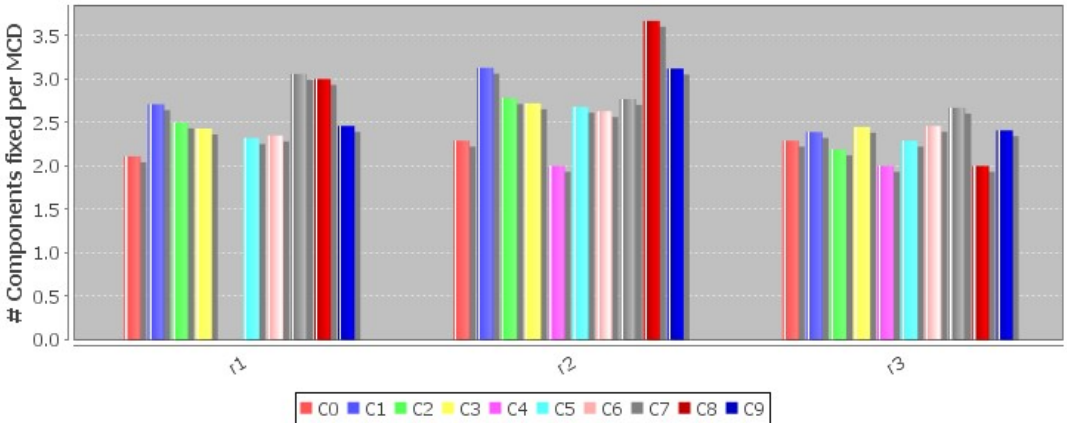


Figure B.5: Component measures with MCD complexity metric **M3** ("#Components fixed per MCD") in the commercial system.

However, from Figure B.5 we can still find that components C7 and C8 have the greatest **M3** measures in release 1, which could be identified as two degeneration-critical components in release 1. Likewise for degeneration-critical component C8 in release 2. We cannot identify degeneration-critical components for release 3 because the component measure distribution is quite even. These findings coincide with the degeneration-critical components shown in Section 6.3.1.

Figure B.5 also shows that the average **M3** ("#Components fixed per MCD") measure of the components increased from release 1 to release 2 but then decreased in release 3. This indicates that the architectural degeneration of the system increased the system evolved from release 1 to release 2 and then decreased in release 3. It thus confirms the conclusions derived in Section 6.3.5.

B.2.2 Architectural Degeneration

Recall Sections 5.2.5 and 6.3.5 that the trend in architectural degeneration is determined by evaluating the averaged MCD quantity and complexity measures of the components over development phases and releases. For example, Figure B.6 demonstrates a line chart which shows the trend in architectural degeneration of the Eclipse Platform over the seven releases, with respect to (w.r.t.) the MCD percentage metric "%MCDs" (**M1**). From this chart we can find that the architectural degeneration increased as the system evolved from release 1 to release 3.2, but then decreased in release 3.3 (from release 3.2). Mostly, this shows an "increase-only" trend. We do no have any process data or historical accounts to explain this trend. Suffice to say here, this "increase-only" trend fits the general trend in architectural degeneration as observed by the laws of software evolution (Belady and Lehman, 1976; Lehman, 1980).

Likewise, Figure B.7 illustrates the average MCD-percentage (**M1'** – "%MCDs") measures[2] for the four phases (FVT, SVT, PQA, and Field) in the three releases

[2]Here, **M1'** refers to the proportion of MCDs in all defects in a particular phase.

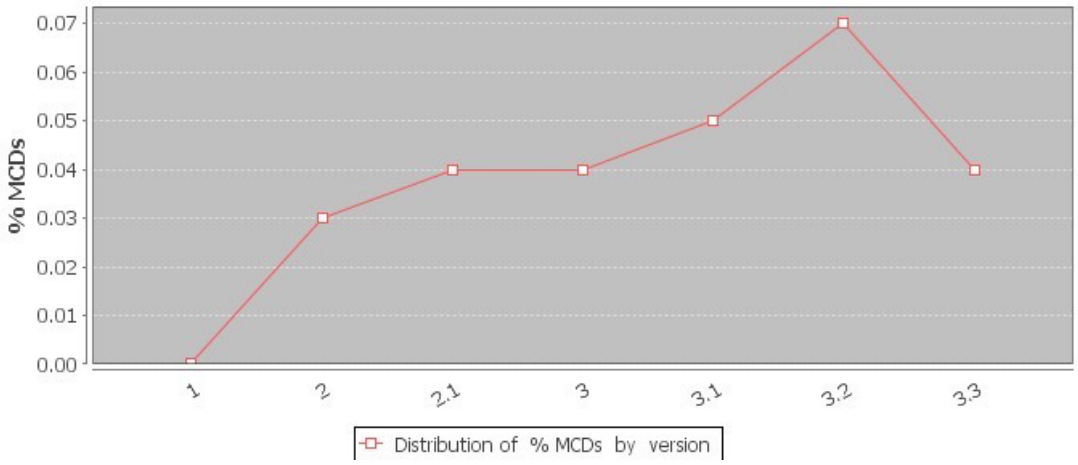


Figure B.6: Architectural degeneration trend of the Eclipse Platform across releases, w.r.t. the MCD percentage metric **M1** ("%MCDs").

(r1, r2 and r3) of the commercial system of Case Study 2. Note that here, the **M1'** measure refers to the proportion of all MCDs in the all defects in the system (a phase thereof). It is just different from the normal **M1** measures shown in Table 6.3 (see Section 6.3.1) and Figure B.6.

Figure B.7 indicates that the **M1'** measures of the system are quite different among the phases and releases. For example, the **M1'** measure for the PQA phase increased as the system evolved from r1 to r3. However, that for the field phase kept relatively stable as the system evolved from r1 to r2 but later decreased substantially in r3. From this figure, we cannot observe the "increase-then-decrease" trend for the architectural degeneration over the phases of the three releases (as concluded in Section 6.3.5). However, the average **M1** ("%MCDs") measures of the components across the three releases have indicated this trend (see row "Mean" in Table 6.3).

B.3 Defect Architectures

Recall that there are two defect architectures (see Figures 6.8 and 6.9) shown in Section 6.3.6. These are "macro" defect architectures as they are spanning

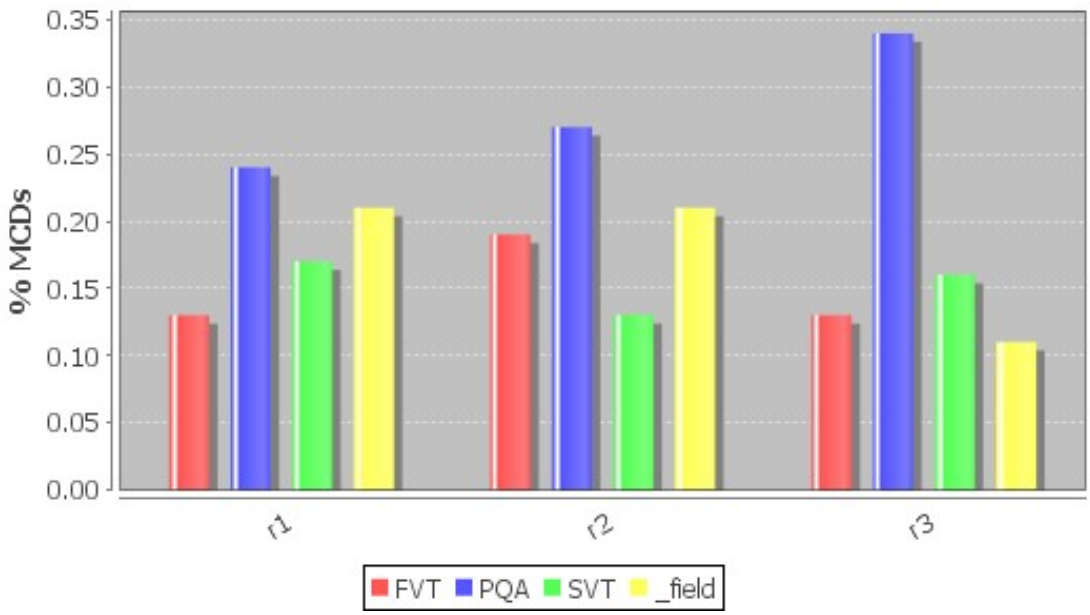


Figure B.7: Architectural degeneration trend of the commercial system across releases, w.r.t. the MCD quantity metric **M1'** ("%MCDs").

components in the system. Further, we give an example of another macro defect architecture which is derived for the Eclipse Platform; see Figure B.8.

Figure B.8 illustrates the defect architecture (segment) of the Eclipse Platform (release 3) with respect to the MCD quantity metric – "#MCDs". It describes the top 3 components (red-colored nodes) and the top 10 fix relationships which have the greatest MCD quantity values (shown in the labels) in release 3. The blue-colored nodes are shown in the figure because they are connected with these fix relationships. The numeric labeles on each node or edge in Figure B.8 indicates the "#MCDs" value of the component or fix relationship in the system.

Figure B.8 indicates that components Team, CVS and UI are degeneration-critical in release 3. This finding is consistent with the finding shown in Figure B.4 above. We also find that there are 55 MCDs pertaining to the fix relationship between components Team and CVS. Considering the 60 and 61 MCDs occurred in the two components, we can derive that most of the MCDs occurred in component Team (55/60) also occurred in component CVS; and vice versa (55/61).

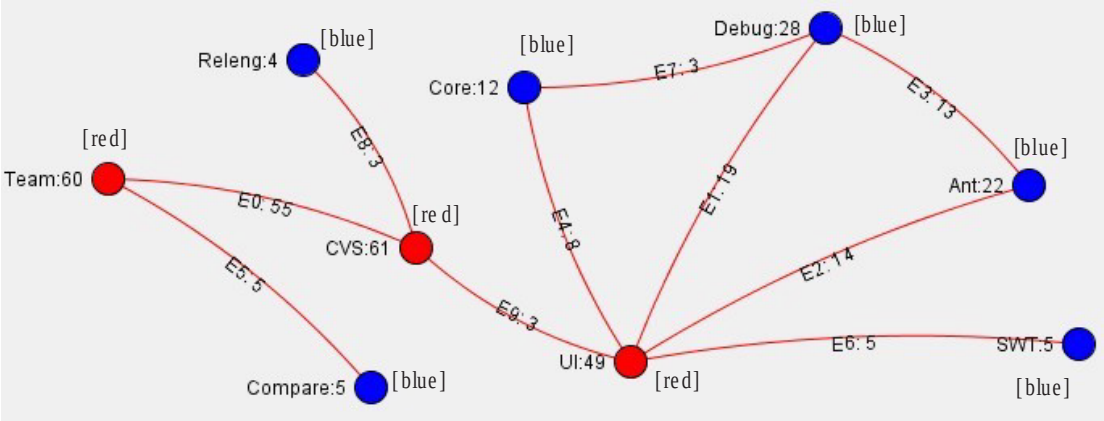


Figure B.8: "Macro" defect architecture (segment) of the Eclipse Platform (release 3) w.r.t. the MCD quantity metric "#MCDs".

Following the above demonstration of macro defect architectures, we, below, describe a "macro" defect architecture which span code files in a given system component; see Figure B.9. The feature of micro defect architecture construction is an extension to the DAD approach for the purpose of identification of degeneration-critical code files and fix relationships in a particular component.

We note from Figure 6.8 (in Chapter 6) that component C5 is a degeneration-critical component in release 1 of the subject system from the MCD-percentage perspective. Especially, C5 persists its degeneration-critical nature in later releases 2 and 3 (see Section 6.3.1). We thus want to do an in-depth investigation particularly for C5. That is to know the contribution of each code file in C5 to its degeneration and also the "degeneration-critical" code files in C5 which contribute substantially more to the degeneration than other code files.

Figure B.9 illustrates a "micro" defect architecture (segment) of component C5 in release 1. It shows the top 10 code files (red-colored nodes) that contain the most number of *multiple-file defects* (MFDs) in C5, and the top 10 most frequently occurring fix relationships between the code files (due to MFDs). Note that MFDs are defect requiring changes (fixes) in more than one code files in a component. This indicates that MFDs exclude MCDs herein.

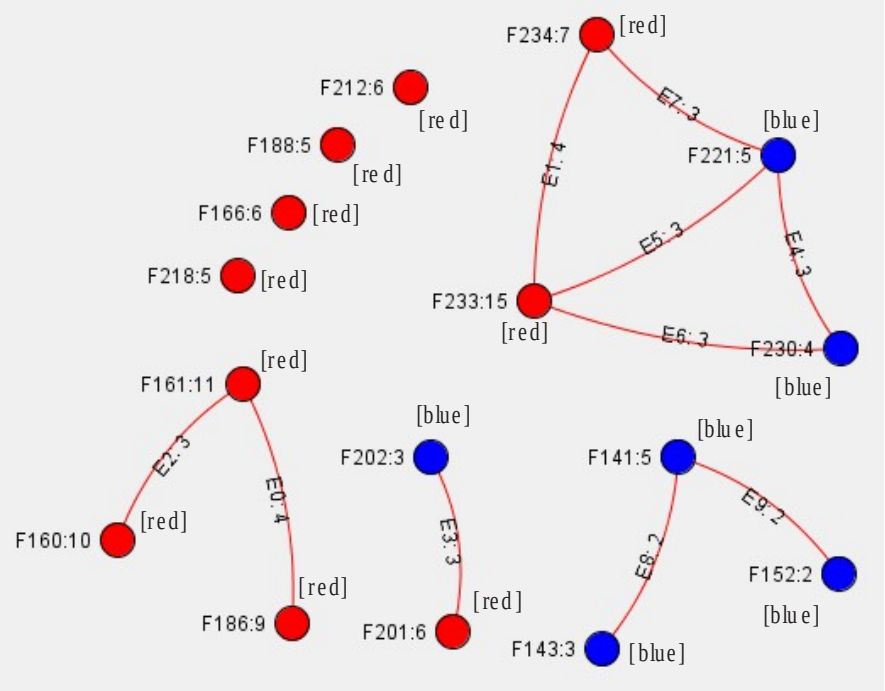


Figure B.9: "Micro" defect architecture (segment) of component C5 in the commercial system (release 1).

Figure B.9 shows that, for example, there are 15 MFDs requiring fixes in code file F233, and there are 4 MFDs that required fixes in both code files F233 and F234 (see edge E1 in the figure). Following the similar criterion used in Section 6.3.1 for identification of degeneration-critical components and fix relationships, these code files and fix relationships (red-colored) should be considered degeneration-critical in component C5 (in release 1).

Figure B.9 also shows that there are 4 of the 10 top MFD-concentred code files (red-colored) that do not have top MFD-concentred fix relationships (red-colored); see nodes F212, F188, F166, and F218. Meanwhile there are 3 fix relationships existing between files that are not in the top 10 list; see edges E4, E8 and E9. It indicates that there is a weak correlation between these degeneration-critical code files and fix relationships (between code files). This finding is in contrast to the "strong correlation" between degeneration-critical components and fix relationships (between components) shown in Figures 1.1 and 6.9.

B.4 Short Conclusion

This chapter demonstrated several typical outputs of the DAD prototype tool (based on the open-source Eclipse Platform and the commercial legacy system investigated in Case Study 2), see Figures B.1–B.9. Some of these outputs have coincided with the findings shown in Case Study 2 (see Section 6.3). Based on the demonstration, we claim that this DAD prototype tool can be used to provide information about a given system and its architectural degeneration over development phases and releases, which can help in diagnosing and treating architectural degeneration of the system.

Vitae and Thesis-Relevant Publications

NAME	Zude Li
CONTACT	zude.s.lee@gmail.com
EDUCATION	Ph.D. (Computer Science) — University of Western Ontario, November 2010 MEng. (Software Engineering) — Tsinghua University, July 2006 BSc. (Computer Science) — Hunan Normal University, July 2004
RELATED WORK EXPERIENCE	Research and Teaching Assistant (2006 - 2010) — The University of Western Ontario

THESIS-RELEVANT PUBLICATIONS

1. Zude Li, Mechelle Gittens, Syed Shariyar Murtaza, Nazim H. Madhavji, Andriy V. Miranskyy, David Godwin, and Enzo Cialini. *Analysis of Pervasive Multiple-Component Defects in a Large Software System*. In Proc. of the 25th IEEE Int'l Conference on Software Maintenance (ICSM'09), pages 265-273, Edmonton, Alberta, Canada, September 2009.

   - An enhanced version of this paper is in Number 2.

2. Zude Li, Nazim H. Madhavji, Syed Shariyar Murtaza, Mechelle Gittens, Andriy V. Miranskyy, David Godwin, and Enzo Cialini. *Characteristics of Multiple-Component Defects and Architectural Hotspots: A Large System Case Study*. Accepted by the Empirical Software Engineering (ESE) Journal, September 2010.

   - This paper describes the main results of Case Study 1 of this thesis (see **Chapter 4**).

3. Zude Li, Nazim H. Madhavji, Mechelle Gittens, Remo N. Ferrari, Syed Shariyar Murtaza, Andriy V. Miranskyy, David Godwin, and Enzo Cialini. *Characterizing Architectural Degeneration from Defect Perspective: A Case Study*. Submitted to the 33rd International Conference on Software Engineering (ICSE'11), Waikiki, Honolulu, Hawaii, May 2011.

   - This paper describes the main results of Case Study 2 of this thesis (see **Chapter 6**).