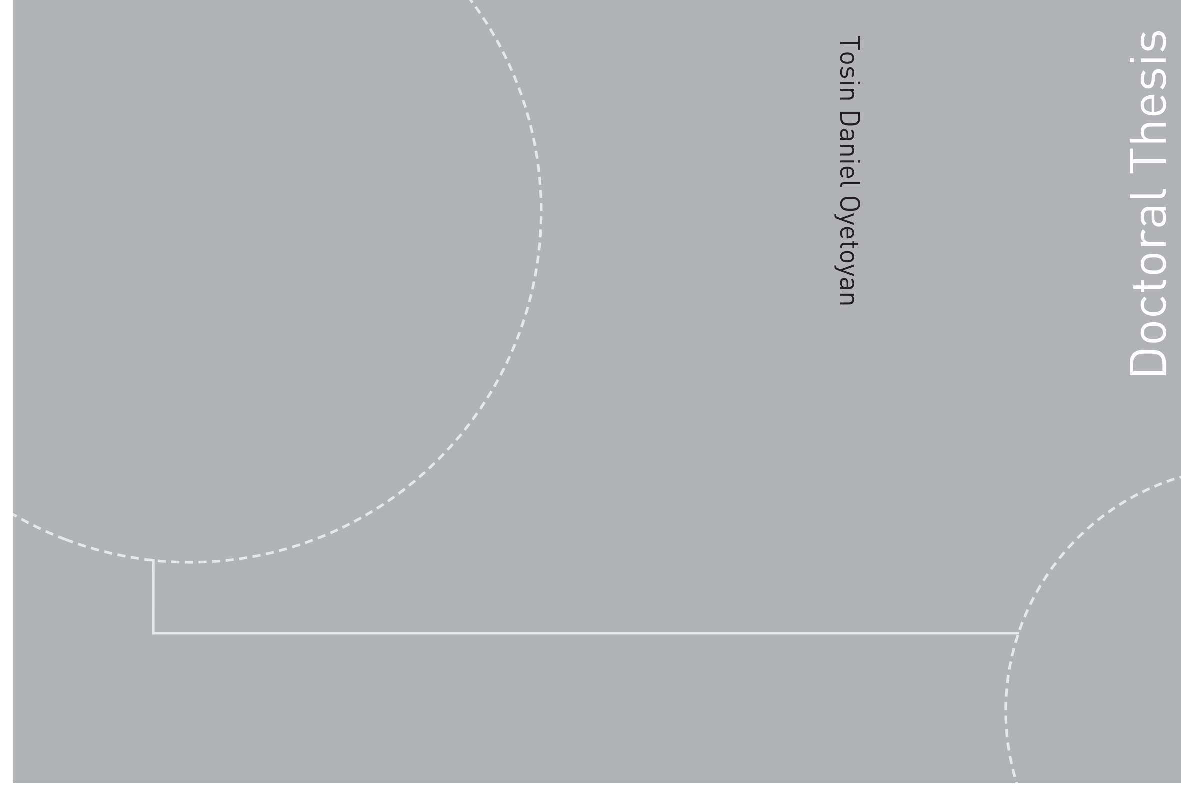
ISBN 978-82-326-0824-9 (printed version)
ISBN 978-82-326-0825-6 (electronic version)
ISSN 1503-8181



Doctoral theses at NTNU, 2015:84

Tosin Daniel Oyetoyan

Dependency Cycles in Software Systems: Quality Issues and Opportunities for Refactoring

Doctoral theses at NTNU, 2015:84

NTNU
Norwegian University of
Science and Technology
Faculty of Information Technology,
Mathematics and Electrical Engineering
Department of Computer andInformation Science

NTNU – Trondheim
Norwegian University of
Science and Technology

NTNU – Trondheim
Norwegian University of
Science and Technology

Tosin Daniel Oyetoyan

Dependency Cycles in Software Systems: Quality Issues and Opportunities for Refactoring

Thesis for the degree of Philosophiae Doctor

Trondheim, April 2015

Norwegian University of Science and Technology

NTNU – Trondheim
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the degree of Philosophiae Doctor

ISBN 978-82-326-0824-9 (printed version)
ISBN 978-82-326-0825-6 (electronic version)
ISSN 1503-8181

Doctoral theses at NTNU, 2015:84

Printed by Skipnes Kommunikasjon as

I have peace following Jesus

Abstract

Society, systems and related businesses are increasingly dependent on software applications, which are integrated and interoperate with other systems. This dependency has implications for the dependability of both the systems and the businesses. There have been reported cases over the years of both systems and business failures due to software defects. However, the work and effort needed to correct defects is not trivial. This involves changing existing applications, which may be overly complex in their structure. Software undergoes constant evolution due to changes in the business environment such as introduction of new technology or new requirements. It is therefore not surprising that the cost of software maintenance is normally estimated to be the highest in the overall software budget. As the software evolves, so does its size and complexity.

One aspect of software complexity is *dependency cycles* that are formed among software classes and packages. Many design guidelines advocate to avoid *dependency cycles* and argue that they inhibit software quality. Despite this conventional wisdom, empirical evidence shows that modern software indeed is riddled with this anti-pattern. The question remains that if cyclic property is known to be complex and is pervasive in software applications, how does it relate to defects and change in general, and what can be done to improve efforts to mitigate it?

This thesis investigates dependency cycles among software components; an aspect of software structural complexity, to find how such properties correlate with defect measures and change rates, and how this knowledge can motivate to refactor and improve these possible defects hotspots in affected systems. The two main research questions to achieve these objectives are stated as follows:

RQ1. What is the effect of dependency cycles on external quality measures of software systems?

RQ2. How to refactor dependency cycle to impact the structural quality and reduce refactoring efforts?

This work contributes mainly to improvement in software quality (**maintainability** and indirectly, **reliability**) and software metrics. The following are the three major contributions of this thesis:

C1. Better understanding of how to utilize different defect metrics to improve software quality

C2. Identification of the impact of dependency cycles on software quality

- **C2-1:** Identification of dependency cycles and neighbourhood as defect hotspots in software systems
- **C2-2:** Better understanding of the change impact of dependency cycles

C3. Tool and metrics to refactor defect- and change-prone hotspots in dependency cycles

- **C3-1:** Added metrics to understand the complexity of components and improve the refactoring of cyclically dependent components
- **C3-2:** A cycle breaking decision support system to refactor cyclically connected components

Preface

This thesis is submitted to the Norwegian University of Science and Technology (NTNU) for partial fulfilment of the requirements for the degree of philosophiae doctor.

This doctoral work has been performed at the Department of Computer and Information Science, NTNU, Trondheim, with Professor Reidar Conradi as the main supervisor and co-supervised by Dr. Daniela S. Cruzes and Professor Letizia Jaccheri. Dr. Carl-Fredrik Sørensen was co-supervisor in the last months.

This work is financed under the IME Smart Grid initiative 2011 – 2015. It also included a 25% teaching duty at the department of Computer and Information Science.

Acknowledgements

This doctoral study was made possible by many people and I would like to express my profound gratitude to them. Thanks to my main supervisor Professor Reidar Conradi for his advice and support and for the freedom he gave me to pursue the research within my area of interest. My sincere thanks to Dr. Daniela Cruzes for providing extraordinary guidance and support from the beginning to the end of my doctoral study. My thanks to Professor Letizia Jaccheri for stepping in at the right time to support my doctoral study. My warm appreciation to Dr. Carl-Fredrik Sørensen who came in the last phase and provided immense and necessary support to finish this thesis. Thanks to Associate Professor Jens Dietrich for the warm reception and supervision during my three-month stay at Massey University, New Zealand.

Many thanks to my parents and parents-in-law who have always called and prayed for the success of this work. My appreciation to my wife, Oludamilola for her support through this journey. To my sons, IfeOluwa and Eni-IbukunOluwa, I say thank you for being so understandingly and awesomely supportive.

Table of Contents

ABSTRACT I

PREFACE III

ACKNOWLEDGEMENTS V

LIST OF FIGURES IX

LIST OF TABLES IX

ABBREVIATIONS XI

1 INTRODUCTION 1
- 1.1 PROBLEM OUTLINE 1
- 1.2 RESEARCH CONTEXT 3
- 1.3 RESEARCH QUESTIONS AND DESIGN 4
- 1.4 PAPERS 6
- 1.5 CONTRIBUTIONS 9
- 1.6 THESIS STRUCTURE 12

2 STATE-OF-THE-ART 13
- 2.1 SOFTWARE ENGINEERING: DEFINITION AND CHALLENGES 13
- 2.2 SOFTWARE QUALITY 14
- 2.3 SOFTWARE TESTING 15
- 2.4 OBJECT-ORIENTED METRICS 18
- 2.5 SOFTWARE EVOLUTION AND MAINTENANCE 18
- 2.6 SOFTWARE PATTERNS AND ANTI-PATTERNS 20
- 2.7 REFACTORING 25
- 2.8 SUMMARY OF RESEARCH CHALLENGES 26

3 RESEARCH CONTEXT AND DESIGN 29
- 3.1 RESEARCH FOCUS 29
- 3.2 SUMMARY OF SOFTWARE SYSTEMS 32
- 3.3 METRICS AND MEASUREMENT 33
- 3.4 DATA COLLECTION FOR EMPIRICAL STUDIES 36
- 3.5 RESEARCH METHODS IN SOFTWARE ENGINEERING 36
- 3.6 RESEARCH DESIGN 38
- 3.7 SCOPE, CONCEPTS AND LIMITATIONS 42

4 RESULTS 45
- 4.1 EMPIRICAL INVESTIGATION OF DIFFERENT DEFECT METRICS TO CLASSIFY CRITICAL COMPONENTS 45
- 4.2 INVESTIGATION OF DEFECT AND CHANGE PRONENESS OF CYCLICALLY DEPENDENT COMPONENTS 48
- 4.3 INVESTIGATING THE EFFECT THAT REFACTORING DEPENDENCY CYCLES HAVE ON DEFECTS 51
- 4.4 IMPROVING THE STRUCTURAL QUALITY OF CYCLICALLY DEPENDENT COMPONENTS USING TOOLS AND METRICS 52

5 EVALUATION AND DISCUSSION 55
- 5.1 OVERVIEW OF THESIS CONTRIBUTIONS 56
- 5.2 EVALUATION OF THE CONTRIBUTIONS AGAINST THE RESEARCH GOAL 58
- 5.3 DISCUSSION OF CONTRIBUTIONS RELATED TO THE STATE-OF-THE-ART 59
- 5.4 RECOMMENDATIONS TO PRACTITIONERS 60

5.5 Discussion of Validity Threats ... 62
5.6 Reflections on the Research Context ... 63

6 CONCLUSION AND FUTURE WORK ... 65
6.1 Overall Summary of Findings ... 65
6.2 Directions for Future Work ... 66

GLOSSARY ... 69

REFERENCES ... 73

APPENDIX A: SELECTED PAPERS ... 85

P1: Comparison Of Different Defect Measures To Identify Defect-Prone Components ... 87

P2: A Study Of Cyclic Dependencies On Defect Profile Of Software Components ... 105

P3: Criticality Of Defects In Cyclic Dependent Components ... 141

P4: Can Refactoring Cyclic Dependent Components Reduce Defect-Proneness? ... 159

P5: Transition And Defect Patterns Of Components In Dependency Cycles During Software Evolution ... 167

P6: Circular Dependencies And Change-Proneness: An Empirical Study ... 183

P7: A Decision Support System To Refactor Class Cycles ... 199

APPENDIX B: SECONDARY PAPERS ... 215

P8: Can Reused Components Provide Lead To Future Defective Components In Smart Grid Applications? ... 217

P9: Initial Survey Of Smartgrid Activities In The Norwegian Energy Sector: Use Cases, Industrial Challenges And Implications For Research ... 219

P10: Open Source Software For The Smartgrid: Challenges For Software Safety And Evolution ... 221

List of Figures

Figure 1 Relationships between structural properties, cognitive complexity, and external quality attributes ... 2

Figure 2 Application structure in relation to internal vs. external dependencies ... 3

Figure 3 Connection between research questions, publications and contributions ... 6

Figure 4 State-of-the-art and area of contributions from PhD Study ... 12

Figure 5 A cycle in Apache Velocity v1.6.2 ... 21

Figure 6 An STK in the JRE v1.7.0 ... 22

Figure 7 Abstraction Without Decoupling ... 23

Figure 8 A case of degenerated/multiple inheritance ... 23

Figure 9 PCT of package cycles ... 24

Figure 10 The software development process with relationships to research contributions ... 30

Figure 11 Data collection from different repositories ... 37

Figure 12 Components in and near dependency cycles ... 40

Figure 13 A simple example of transitions of in-cycle components between releases ... 41

Figure 14 % of DPC with critical defects identified at the top k% of the class-files DPC over six releases ... 47

Figure 15 Class Model for the Cycle breaking decision support system ... 54

Figure 16 Overall relationships between the studies and contributions to software engineering field ... 57

List of Tables

Table 1 Research questions vs. contributions and papers ... 11

Table 2 Software quality attributes, criteria and measures (ISO/IEC 25010:2011) ... 16

Table 3 Properties of systems used in the thesis ... 33

Table 4 Defect metrics ... 34

Table 5 Cyclic dependency Metrics ... 35

Table 6 Summary of research design ... 39

Table 7 Studies and their relation to research questions, methods, and contributions ... 46

Table 8 Connection between contributions, research questions, papers, and specific software quality attributes ... 58

Abbreviations

API	Application Programming Interface
AWD	Abstraction Without Decoupling
CBO	Coupling Between Objects
CB-DSS	Cycle Breaking Decision Support System
CCD	Cumulative Component Dependency
CMS	Configuration Management System
COTS	Commercial-Off-The-Shelf
CRSS	Class Reachability Set Size
DDC	Defect-Dense Component
DIH	Degenerated Inheritance
DIT	Depth of Inheritance Tree
DPC	Defect-Prone Component
DSS	Decision Support System
DTS	Defect Tracking System
HFC	Hard-to-Fix Defective Component
ICT	Information Communication and Technology
IRCRSS	Interface-CRSS Reduction Rate
LCOM	Lack of Cohesion Of Method
LOC	Lines Of Code
NOC	Number of Children
ODC	Orthogonal Defect Classification
OO	Object-Oriented
OSS	Open Source Software
PCT	Package Containment Tree
RFC	Response For a Class
RC	Research Challenges
RQ	Research Questions
SCC	Strongly Connected Component
SDC	Severe Defective Component
SoS	System of Systems
STK	Subtype Knowledge
UML	Universal Modeling Language
WMC	Weighted Methods of a Class

1 Introduction

1.1 Problem Outline

Today, virtually all aspects of systems (critical and non-critical) and businesses are dependent on software programs to execute their functions and operate successfully. This dependency implies that a failure[1] within a software program has the likelihood to result in a system or business failure. A system or business failure may be the result of a software fault/defect[2]. As noted by Lilley (2012), software does not "wear out" after some period of proper operation as hardware components do. In addition, defects in software systems may not be apparent over time but when they are exposed, they act like a hidden bomb (Lilley, 2012). There are numerous evidence of system and business failures due to software defects (Leo, 2013; Lilley, 2012). Therefore, early knowledge of probable locations of software defects is useful to improve the dependability of these systems.

Removing a large number of defects may have a trivial effect on reliability as pointed out by Adams (Adams, 1984). The study of Adam shows that most of the latent defects lead to very few failures in practice, while the vast majority of observed failures are caused by a relatively tiny number of defects. In addition to this observation, both Ebert et al. (2005) and Boehm and Basili (2001), argue that 60-80% of the correction effort and 80% of avoidable rework are due to 20% of the defects. This shows that it is not the number of defects, but rather their severity that matters. A high severity defect usually points to a fatal error resulting in a system failure, whereas low severity defects mostly points to some cosmetic issues. Thus, there is a pressing need for more studies both to identify and remove critical defects in software systems, and to find their probable locations within the software.

The structural connections among components in a software system have been demonstrated to relate to defects (Abreu and Melo, 1996). Figure 1 shows that structural properties in software impact the human cognitive ability, which in turn affects the

[1] Failure: The inability of a system or system component to perform a required function within specified limits

[2] Defect/Fault: An anomaly in a software code unit or product that can be the cause of one or more failures

external quality attributes of a system. Conversely, quality attributes, when improved can reduce structural complexities and other software properties. Many of the current software systems are overly complex and indeed highly interconnected. The higher the complexity of a system, the more difficult it is to maintain, and the higher is the risk of accidental and unexpected failures (Fenton and Pfleeger, 1997). One area of software complexity is dependency cycles that are formed by direct or indirect decisions during software development and evolution. Dependency cycles among components are notorious for extremely increasing the coupling complexity among interconnected components (Briand et al., 1998; Briand et al., 2001b).

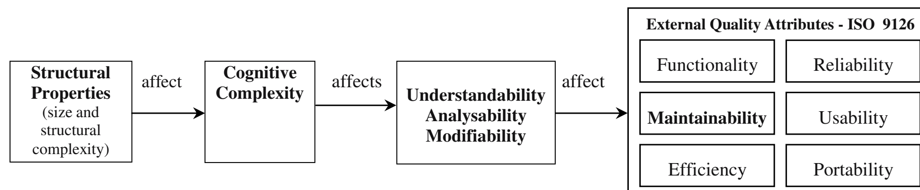


Figure 1 Relationships between structural properties, cognitive complexity, and external quality attributes (Genero et al., 2007)

There are numerous claims that cycles inhibit external software quality attributes such as extendability, understandability, testability, reusability, buildability, maintainability and reliability (Fowler, 2001; Lakos, 1996; Parnas, 1979). Evidence shows that they are widespread in real-life software systems (Briand et al., 2001a; Hanh et al., 2001; Kung et al., 1996; Melton and Tempero, 2007b; Parnas, 1979). Intuitively, since cycles increase coupling complexity, it can be expected that it should correlate with defect-proneness. However, there is no empirical evidence to support this intuition. Thus, it remains a gap in research open for thorough empirical studies.

In studying dependency cycles of object-oriented systems, it can be argued that internally (source) declared types within a software application are of particularly interest. We can exclude externally declared types (libraries) from our discussion for the following reasons[3] (see Figure 2):

1. Internally declared types usually depend on the externally declared types (e.g. standard APIs) and not vice versa. Thus, it is practically impossible for externally declared types to form cycles with internal application types.
2. Developers can easily modify types declared in available software source files and alter the dependencies they have to one another and to externally declared types. However, it is not feasible (easy nor sensible) to alter the dependencies that externally declared types (e.g., 3rd party libraries) have to one another. In many cases, the source code of externally declared types is not available.
3. Externally defined types are often more stable than internally defined types in the source files of the software applications. By making a decision to reuse externally declared types, we can assume that these types are thoroughly tested, and in general

[3] http://www.cs.auckland.ac.nz/~hayden/research.htm

of excellent quality. Therefore, it is not very likely that these data types will change or that the interfaces to the types will change.

The above reasons do not preclude dependency cycles from being formed with reused libraries in a large software organisation, or where source code to the libraries is available. An example is the Eclipse project where cycles are formed between the AWT package and SWING package. These are interesting cases for future study.

The goal of this research is twofold: Firstly, to collect empirical evidence of the effect of dependency cycles among internally declared types on defects and change rate. This can consequently motivate for refactoring of defect-prone cyclic components. Secondly, to realize a cycle-breaking decision support system that could assist developers and maintenance engineers to refactor dependency cycles and improve the structure of the software.

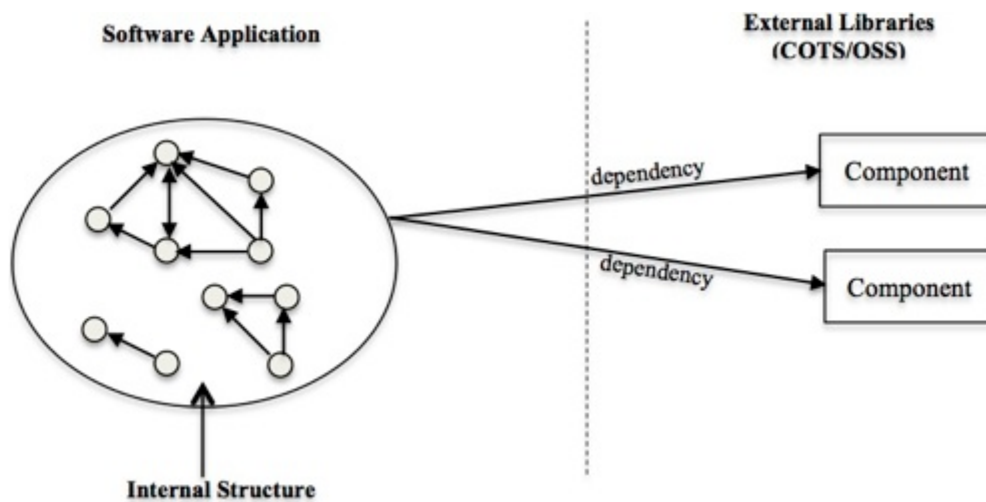


Figure 2 Application structure in relation to internal vs. external dependencies

1.2 Research Context

This PhD research has been done within the context of the Smart Grid Research Initiative of IME, NTNU. The main goal as defined within the software engineering project [4] is: ***Improved Management of Software Evolution for Smart Grid Applications***. The main case study is Smart Grid software systems. This work has been done in collaboration with Powel AS, a major Smart Grid software vendor in Norway with more than 80% market share, and with the main office in Trondheim. We have performed a longitudinal study of one of the company`s Smart Grid software for three and half years. The study explored the source code, change set, and defect repositories for the selected application. The study included participation by an MSc student in the last phase of the thesis to implement a refactoring plugin for the development environment used by the company. The justification for analysing defects in relation to the structural design of the software driving the Smart Grid, is very strong. As a system

4 http://www.ntnu.edu/ime/research/smartgrid/project-f

of systems (SoS), Smart Grid faces risks and challenges typical to SoS environment (Creel and Ellison, 2008), such as:

1. Potential for change in the system(s) from any direction: stakeholders, constituent systems as well as evolving business requirements.
2. Less predictability regarding stakeholders' needs, technology advances and component behavior typical in an environment with no central control.
3. Failures with cause or impact (cascade) beyond the individual system boundary.
4. Constraints in terms of new development and evolution because of existing collection of design choices.
5. Limited knowledge of individual system state and behaviour.

Smart Grids are still in the formation stage, and represents a shift from a relatively closed grid structure to more complex and highly interconnected systems. It thus faces practical challenges from the many requirements that are needed to accomplish its vision. It is therefore an important goal that defects and especially critical defects are located and reduced, and that the defect locations are improved within the structures of the software systems. In addition, reducing the complexity of these hot spots to reduce the effort to make changes would be important in these applications.

1.3 Research Questions and Design

The aims of this work are to (1) investigate how dependency cycles affect non-functional requirements (quality attributes) of software systems, and (2) propose tools and methods to refactor dependency cycles. In particular, the thesis investigates the effect of dependency cycles on *maintainability,* and indirectly *reliability* attributes of software systems. Direct measurement of maintainability and reliability is not the goal of this thesis. Therefore, in this thesis, defect and change rates have been used as proxy metrics to quantify the aforementioned quality attributes. Ultimately, this work is aimed to improve the structural quality and the management of the software applications during evolution.

1.3.1 Research questions

The main research questions and the sub-questions investigated in this thesis are:

RQ1. What is the effect of dependency cycles on external quality measures of software systems?

- **RQ1-1** What is the effect of using different defect metrics to identify critical software components?
- **RQ1-2** What is the effect of dependency cycles on software defect?
- **RQ1-3** What is the effect of refactoring cycles on defect-proneness?
- **RQ1-4** What is the effect of dependency cycles on change rate?

RQ2. How to refactor dependency cycles to impact the structural quality and reduce the refactoring efforts?

1.3.2 Research methodology

The research methodologies that are mostly relevant in software engineering can be (**a**) controlled experiments, (**b**) case studies, (**c**) survey, (**d**) ethnographies, and (**e**) action research (Easterbrook et al., 2008). In this study, literature review (pre-study), case studies, survey, and design science research methodologies have been used. Design science is a methodology commonly used in information systems research (Peffers et al., 2007). Case studies are empirical methods aimed to offer in-depth understanding about how and why a certain phenomenon occurs. It can use both quantitative and qualitative methods. Whereas, design science is aimed at creating artefacts that could change existing situations to a more preferred ones (Hevner et al., 2004). Design science research should address unsolved problems in a unique and innovative way, or an already solved problem in a more effective or efficient way to distinguish them from routine design, (Hevner et al., 2004). Design science research methodology (DSRM) is made up of six activities (Peffers et al., 2007), they are: *problem identification and motivation, defining the objectives of a solution, design and development, demonstration, evaluation*, and *communication*.

Studies 1, 2 and 3 in this research were empirical in nature (see Figure 3). Study 1 used case study and survey, while studies 2 and 3 used literature review and several case studies to understand the relationships between (1) dependency cycles and defects, and (2) dependency cycles and change frequency. The outcome of the previous studies motivated for refactoring of cyclically connected components. Thus, in Study 4, a cycle-breaking decision support system was implemented by using design science research methodology.

In relation to the research goals and research questions, the research methods used in this thesis are discussed below:

RQ1: Case studies (Quantitative and Qualitative methods) are used to answer the research question. Firstly, to investigate how several defect measures can be used to identify critical components in software systems. Secondly, to investigate the relationships between dependency cycles and defects, and dependency cycles and change frequency

RQ2: This RQ is addressed by using design science methodology in the construction of a cycle-breaking decision support system using an improved metric to refactor dependency cycles. We then used a case study approach and interview for evaluation.

The details of the research design are depicted in Figure 3. It shows the connections between the research questions, the methodology, and the contributions. There are challenges particularly when conducting an empirical study with industrial companies. Some of these are:

1) Unmanageable challenges and scope arising from the challenges of the Smart Grid companies.
2) Smart Grid is not yet mature; hence, it could be difficult to obtain sufficient empirical data necessary for the study
3) Lack of effort on the part of industrial partners.

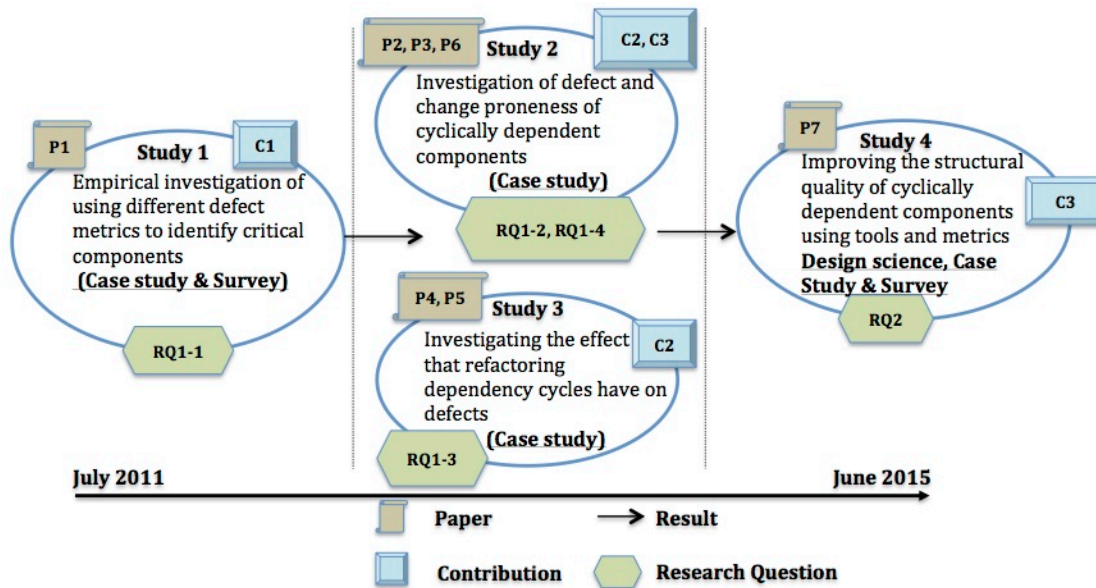


Figure 3 Connection between research questions, publications and contributions

The above risks have been minimized because of the good collaboration with Powel AS. The company has also provided all support needed at the different stages of the research work. In addition, there is a regular feedback of the research findings to the company, which have triggered further studies, and an implementation of approaches and tool to improve the system's structures.

1.4 Papers

P1 Oyetoyan, T.D., Conradi, R., Cruzes, D.S., 2013. *A Comparison of Different Defect Measures to Identify Defect-Prone Components*, Joint Conference of the 23rd International Workshop on Software Measurement and the 2013 8th International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2013, Ankara, Turkey. pp. 181-190

The main goal of this paper was to evaluate the usefulness of several defect measures such as number of defects, defect density, defect correction effort, and severity of defects, to define defect-prone components. The study aims to find out whether there are significant variations between different defect measures in identifying defect-prone components and architectural hotspots. Results demonstrated that employing several defect metrics is an important and useful approach to model construction, testing activities, and for performing defect analysis of software components. This publication contributes to **RQ1-1**

Own contribution: I was the leading author in this study. I performed the data collection, design and implementation of the experiment, and wrote the paper. The two co-authors reviewed the paper.

P2 Oyetoyan, T.D., Cruzes, D.S., Conradi, R., 2013. *A study of cyclic dependencies on defect profile of software components*. Journal of Systems and Software 86 (12), pp. 3162-3182.

This paper explores the relationships between cyclic dependent components and defects. By testing four different hypotheses on six non-trivial systems we established that:

1) Components in and near cycles have higher likelihood of defect-proneness than those not in cyclic relationships.
2) The higher number of defective components is concentrated in components in and near cycles.
3) Defective components in and near cycles account for the clear majority of defects in the systems investigated.
4) The defect density of components in and near cycles is sometimes higher than those in non-cyclic relationships.

This paper contributes significantly to addressing **RQ1-2.**

Own contribution: I was the leading author in this study. I performed the data collection, design and implementation of the experiment and wrote the paper. The remaining two authors reviewed the paper.

P3 Oyetoyan, T.D., Cruzes, D.S., Conradi, R., 2013. *Criticality of Defects in Cyclic Dependent Components*, 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), Eindhoven, Netherlands, pp. 21-30

This paper investigates the criticality of defects in cyclic dependent components. Removing a large number of defects may have trivial effect on system reliability. The most number of latent defects lead to very rare failures in practice, while the vast majority of observed failures are caused by a relatively small number of defects. This shows that it is not the number of defects, rather their severity that matters. Thus, we are compelled to find out if this majority of defects and defect-prone components in cyclic-related components are also the majority in both critical defects and severe defective components. In the two applications that are empirically investigated, results demonstrated that components in and near cycles account for almost all the critical defects. This publication addresses **RQ1-2.**

Own contribution: I was the leading author in this study. I performed the data collection, design and implementation of the experiment, and wrote the paper. The two co-authors reviewed the paper.

P4 Oyetoyan, T.D., Cruzes, D.S., Conradi, R., 2013. *Can Refactoring Cyclic Dependent Components Reduce Defect-Proneness?*, 29th IEEE International Conference on Software Maintenance (ICSM), 2013 pp. 420-423

The results from **P2** and **P3** indicate that components with cyclic relationships are responsible for the largest number and severity of defects and defect-prone components. Therefore, the goal of this paper is to investigate the variables within cyclic dependency

graphs that correlate with number of defect-prone components. By using network and statistical analysis, the results demonstrate that adding new components or creating new dependency relationships correspond strongly to an increase in the number of defect prone components. We can therefore hypothesize that refactoring dependency cycle can reduce the defect-proneness of components. This publication contributes to **RQ1-3.**

Own contribution: I was the leading author in this study. I performed the data collection, design and implementation of the experiment, and wrote the paper. The two co-authors reviewed the paper.

P5 Oyetoyan, T.D., Cruzes, D.S., Conradi, R., 2014. *Transition and Defect Patterns of Components in Dependency Cycles During Software Evolution*, IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week, Antwerp, Belgium, pp. 283-292

This study investigates the defect-proneness patterns of cyclically connected components vs. non- cyclic ones when they transition across software releases. By using empirical studies on many applications and releases, it is established that cyclically connected components remain in defective states during evolution more than non-cyclic components. In addition, the class reachability set size (CRSS) metric is found to increase more among cyclically connected components that turn defective in future releases. We conclude that (1) refactoring cyclically connected components may yield benefits in terms of reduction in defect-proneness in future releases (2) such refactoring should focus on minimizing the class reachability set size (CRSS) metric. This publication contributes to **RQ1-3**

Own contribution: I was the leading author in this study. I performed the data collection, design and implementation of the experiment, and wrote the paper. The two co-authors reviewed the paper.

P6 Oyetoyan Tosin. D, Dietrich Jens, Falleri Jean-Remy and Jezek Kamil, 2015, *Circular Dependencies and Change-Proneness: An Empirical Study*, 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, École Polytechnique de Montréal, Québec, Canada, pp. 238-247

Recent studies have proposed new heuristics and approaches to distinguish between "bad" and "harmless" cycles. In this study, we have investigated (1) whether cycles are generally change-prone more than non-cycles, (2) whether cycles that have high diameters within their package containment tree (PCT-diameter) are more change prone, and (3) whether cycles that contain subtype knowledge (STK) in their structure are more change-prone. We found that (1) dependency cycles have big change impact on their direct in-neighbours, (2) neither the PCT-diameter nor the STK properties can identify "harmless" or "critical" cycles, and (3) certain design patterns do contain cycles (e.g. Visitor pattern) and may be "harmless" in terms of their change-proneness. This paper contributes to **RQ1-4**

Own contribution: I was the leading author in this study and contributed about 60%. I wrote the code for the experiment, collected the data and performed the experiment. I also wrote about 40% of the paper.

P7 Oyetoyan, T.D., Cruzes, D., Thurmann-Nielsen, C., 2015, *A Decision Support System to Refactor Class Cycles*. Accepted at the 31st International Conference on Software Maintenance and Evolution ICSME, Bremen, Germany

In this study, we have proposed and developed a cycle breaking decision support system (CBDSS) that implements existing design approaches in combination with the class edge contextual data to refactor class dependency cycles. Furthermore, we have implemented a new metric called IRCRSS that identifies the reduction rate of class reachability set size (CRSS) from a class interface to improve the overall refactoring efforts. The results of the evaluations on multiple systems show that (1) the IRCRSS metric could identify fewer classes for cycle breaking and thus reduce the refactoring efforts reasonably, and (2) the CBDSS could assist software engineers to plan the restructuring and refactoring of large and complex dependency cycles in classes. This publication contributes to **RQ2.**

Own contribution: I was the leading author in this study. I performed the design and implementation of the Java tool and the underlying algorithm and wrote the paper. The second co-author reviewed the work and paper. The last co-author implemented the Java algorithm in a C# plugin.

1.5 Contributions

This work has contributed both to the theory and practice in software quality (**maintainability** and indirectly, **reliability**) and software metrics (See Figure 4 and Table 1). The following have been identified as the main contributions with sub-contributions related to the research objectives.

C1 Better understanding of how to utilize different defect metrics to improve software quality.

__C1-1__: Identification of the usefulness of using different defect metrics to classify critical software components.

C2 Identification of the impact of dependency cycles on software quality.

__C2-1__: Identification of dependency cycles and neighbourhood as defect hotspots in software systems.

__C2-2__: Better understanding of the change impact of dependency cycles.

C3 Tool and metrics to refactor defect- and change-prone hotspots in dependency cycles.

__C3-1__: Added metrics to improve the refactoring of cyclically dependent components.
__C3-2__: A cycle breaking decision support system to refactor cyclically connected components.

1.5.1 Contribution to software quality

C1-1 *Identification of the usefulness of different defect metrics to classify critical software components* (**RQ1-1, P1**).

A quantitative and qualitative analysis of defects and source code data of an industrial Smart Grid application, shows that several defect measures such as defect counts, defect density, defect correction effort, and defect severity have variations in their identification of defect-prone components that are critical to the application. By using the four metrics, it was possible to identify a significant number of components classified as critical by developers. The result contrasts with using the popular defect counts and defect density. The study grows the knowledge about the need to include several defect measures during component-defect analysis.

C2-1 *Identification of dependency cycles and neighbourhood as defect hotspots in software systems.*

(1). Empirical evidence of dependency cycles as defect hotspots (**RQ1-2, P2**).

A quantitative analysis of defects and source code of a commercial Smart Grid and open source applications show that components in and near dependency cycles have more defects and are more defect-prone than those not in cycles

(2). Empirical evidence of dependency cycles as hotspots for critical defects **(RQ1-2, P3).**

A quantitative analysis of defects and source code of a commercial Smart Grid and open source applications show that the majority of critical defects are concentrated in components in and near dependency cycles. This links to **C1,** that critical components are located in cycles or near cycles.

(3). Better understanding of the relationship between refactoring dependency cycles and defects **(RQ1-3, P4 & P5).**

A quantitative analysis of defects and source code of a commercial Smart Grid and open source applications show that components in cycles that turn defective in future releases distinctly and significantly have higher class reachability set size than non-cyclic ones.

C2-2 *Better understanding of the change impact of dependency cycles* **(RQ1-4, P6)**

A quantitative analysis of change proneness of cyclically connected components shows that dependency cycles could have significant change impact on its neighbourhood and less change within its structure especially for some special cycle types (e.g. cycles formed by the Visitor pattern). Furthermore, some special types of cycles (cycles with STK and cycles with high PCT-diameter) do not show higher correlation with change-proneness than cycles without these properties.

Summary: **C1 and C2** have contributed greatly to understand the location of defect hotspots in software structure. It has motivated the refactoring of cyclically connected components to create maintainable, testable, and reusable components. It has improved the understanding of the relationship between refactoring cyclically connected

components and defect-proneness. In addition, it has provided insight into the presence of high coupling and change-proneness in dependency cycles and neighbourhoods of cycles.

C3-2 *A cycle breaking decision support system for refactoring of cyclically dependent components* **(RQ2, P7)**

Currently, there is little advice on how to refactor dependency cycles at the class granularity level. The decision support system contributes to improving existing software structure by providing approaches and implementable actions to decouple classes that are in dependency cycles.

Summary: C3-2 has contributed to performing the actual restructuring task to achieve a maintainable system

1.5.2 Contribution to software metrics

C3-1 *Added metrics to understand the complexity of component and improve the refactoring of cyclically dependent components* **(RQ1-2, RQ1-4 & RQ2, P2, P3, P6 & P7).**

First, the research presents additional metrics that help to better understand cyclically connected components. It introduce a metric termed "*depend-on-cycle*" that shows that components with this property share some similarities with those directly in cycles. Second, it extends the metric (class reachability set size) proposed in (Melton and Tempero, 2007a), which is a variant of the CCD metric by Lakos (Lakos, 1996). The new metric, named the "*Interface Reduction rate for class reachability set size* (IRCRSS)", identifies the reduction rate in the reachability set size between a class and its interface. The results from empirical validation shows that the application of the metric provides better results and is a useful indicator of a reduction in software complexity and refactoring efforts on existing systems. It is able to select fewer candidates when applied as compared to refactoring without using the metric.

Table 1 Research questions vs. contributions and papers

Research Questions	Contribution	Papers	Area of contribution
RQ1-1	C1	P1	Software quality
RQ1-2	C2-1, C3-1	P2, P3	Software quality and metrics
RQ1-3	C2-1	P4, P5	Software quality
RQ1-4	C2-2, C3-1	P6	Software quality and metrics
RQ2	C3	P2, P3, P6, P7	Software quality and metrics

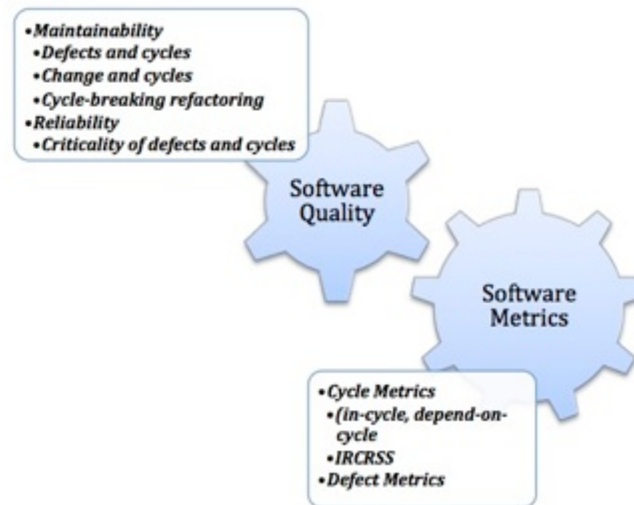


Figure 4 State-of-the-art and area of contributions from PhD Study

1.6 Thesis Structure

This thesis is structured into two parts. The first part contains six chapters to introduce the thesis, put the research into context, evaluate the research design, integrate the results and discuss the contributions of this thesis. The second part is a collection of the selected papers for the thesis.

Chapter 2: State-of-the-Art discusses the state-of-the-art in software engineering, software quality, software testing, object-oriented metrics, software patterns and anti-patterns, software evolution, maintenance and software refactoring.

Chapter 3: Context and Research Design explores the research context and design of the thesis. We discuss and describe the research focus, the software we have used for the studies, the metrics and the approach for measurement and data collection. The research designs for the studies are then described and lastly, the threats to the validity of our results are discussed.

Chapter 4: Results presents the results of the studies.

Chapter 5: Evaluation and Discussion of Results evaluates and discusses the contributions of this thesis.

Chapter 6: Conclusion and Future Work provides the conclusion and a direction for future studies.

Appendix A: (enclosed, selected papers) presents the seven primary papers selected for this thesis.

Appendix B: presents abstracts of three secondary papers, which are partly related to this thesis.

2 State-of-the-Art

This chapter presents state-of-the-art topics relevant to this thesis. Section 2.1 presents software engineering with definition, concepts, and challenges. Section 2.2 looks into the topic of software quality and discusses related quality attributes. Section 2.3 presents software testing. Section 2.4 presents object-oriented metrics. Section 2.5 presents software evolution and maintenance. In Section 2.6, software patterns and anti-patterns are presented. Section 2.7 presents related studies in software refactoring. Lastly, in Section 2.8, we summarize and discuss the challenges of this thesis.

2.1 Software Engineering: Definition and Challenges

According to IEEE Computer Science (SWEBOK) (Bourque and Fairley, 2014), Software engineering is defined as:

"(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1)"

In ISO/IEC/IEEE Systems and Software Engineering Vocabulary (ISO/IEC/IEEE 24765:2010), Software engineering is defined as:

"*(1) The systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software (2) the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software*"

The term "software engineering" was first coined at a NATO conference in 1968 (Naur and Randell, 1969) to discuss the prevalent software crisis at this time (Sommerville, 2011; van Vliet, 2000). Software problems such as late delivery, unfulfilled functionality, large post-release errors, and adaptability challenges were common at this time. These challenges stem from individual approaches and lack of a systematic or an engineering approach to the development of large and complex software systems (Sommerville, 2011). Following the 1969 conference, substantial effort in the 70s and 80s, has been invested into creating new software engineering techniques and methods

such as structured programming, information hiding, object-oriented development, tools, and standard notations (Sommerville, 2011). Software engineering has thus evolved into a discipline where development and the development process are standardized and supported with several approaches and tools (Mohagheghi, 2004).

Despite the improvements in the way software is produced, software projects and development still face many problems. There is still evidence of system failures due to software failure (Leo, 2013; Lilley, 2012). Due to the nature and role of software, there are some key challenges that affect them and they are briefly discussed below:

1. Heterogeneity/interoperability (Pfleeger, 2001; Sommerville, 2011): More and more, systems must talk to each other, they need to run on different platforms, and interfacing new systems with legacy systems are all common challenges.
2. Change (Pfleeger, 2001; Sommerville, 2011): The dynamic nature of our environment forces change in technologies. This necessitates the need for adaptable software processes and products that are fulfilled within budget and on time.
3. Security and trust (Sommerville, 2011): Software is increasingly interconnected with every aspect of people's lives. Thus issues about confidentiality and integrity are critical subjects in software.

To limit the impact of change within a software structure, several design guidelines (e.g. Acyclic Dependencies Principle) and approaches (e.g. component-based software engineering (CBSE)) have been proposed. Nevertheless, recent empirical evidence show that internal software structures that usually are the first target of change, have structural complexity problems (Dietrich et al., 2010; Melton and Tempero, 2007b).

2.2 Software Quality

According to (ISO/IEC 25000:2014; ISO/IEC 25010:2011), software quality is defined as the:

"(1) Capability of a software product to satisfy stated and implied needs when used under specified conditions (2) degree to which a software product satisfies stated and implied needs when used under specified conditions"

Software quality can be viewed from many different perspectives as it relates to the stated and implied needs. Such views can be transcendental, user, manufacturing, product, and value-added (Naik and Tripathy, 2011; Pfleeger, 2001). A transcendental view says that quality is something that can be recognized but not defined in any tractable form. The user view sees quality as the extent to which a product meets user needs and expectations. The manufacturing view is concerned with whether the product meets the stated requirements. This view suggests two characteristics to measure: the defect count and the rework cost. The product view sees quality by assessing the internal qualities with the hypothesis that products with good internal quality would have good external quality. Lastly, the value-based view looks at quality from the viewpoints of excellence and worth. The product and manufacturer views are the centre of the investigation in this thesis. Essentially, the structural characteristics of software product are investigated to find correlation with the external quality metric, which is quantified by various defect metrics.

ISO/IEC/IEEE defines internal measure of software quality as: *"the measure of the degree to which a set of static attributes of a software product satisfies stated and implied needs for the software product to be used under specified conditions"*.

As noted by (Franch, 1998), non-functional requirements need a comprehensive and formally defined languages to state the quality requirements in the software itself. This approach can improve the evaluation of a product to determine whether it falls within the stated non-functional requirements. The lack of this formality has had negative impact on many software development tasks. To measure software quality, different models have been described in earlier work. McCall et al. (1977) classified the software quality model using eleven factors (correctness, reliability, efficiency, integrity, usability, maintainability, testability, flexibility, portability, reusability, and interoperability). ISO 9126, now replaced with ISO/IEC 25010:2011, categorize the product quality model into eight main quality characteristics (functionality, suitability, reliability, performance, efficiency, usability, security, compatibility, maintainability, and portability). Bass et al. (2003) have described these characteristics as software quality attributes.

The quality attributes at the higher level can only be measured indirectly. By defining lower level criteria for each attribute and combined with the ratings it is possible to have measurement of the extent that the quality factor is satisfied (van Vliet, 2000). This thesis investigates two quality attributes (*maintainability* and indirectly *reliability*) by using indirect measures. Quality is a difficult concept to define or measure, and it is also about acceptable compromises (Gillies, 1997). Notwithstanding, to assess and improve software quality would require imposing some degree of control (Kitchenham and Pfleeger, 1996), and to assert control would require defining measurable attributes (DeMarco, 1982).

Table 2 describes the ISO/IEC 25010:2011 product quality model, the quality attributes, and their criteria (sub-characteristics). Emphasis is given to the two attributes (reliability and maintainability) most relevant to this thesis. The defect metrics are employed as the indirect measure for reliability. Measuring actual reliability by using for example, the *"Mean-Time-To-Failure (MTTF)",* is not the major focus of this thesis. It is therefore not possible to conclude on the actual impact the defect metrics used have towards measuring reliability of the systems. In terms of the maintainability attribute, (Roger, 2005) has indicated using a simple time-oriented metric, *"Mean-Time-To-Change (MTTC)"*. There are of course challenges with this metric. For example, an accurate "change time" should include the time it takes a developer to reason about the task. However, this may not be possible as such cognitive task could be accomplished anywhere and anytime. In this thesis, we have used the cycle metrics and change-probability metrics as indirect indicators. Section 3.3 describes the details of these metrics.

2.3 Software Testing

According to SWEBOK (Bourque and Fairley, 2014), software testing is defined as:

"the dynamic verification of the behaviour of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behaviour"

Table 2 Software quality attributes, criteria and measures (ISO/IEC 25010:2011)

Quality Attributes	**Criteria**	**Description**	**Indirect measure (own addition)**
Functionality suitability	Completeness Correctness Appropriateness		
Reliability	Maturity Availability **Fault tolerance** Recoverability	**Fault tolerance**: degree to which a system, product or component operates as intended despite the presence of hardware or software faults	Defect severity, Defect density, defect count and defect probability
Performance efficiency	Time behaviour Resource utilization Capacity		
Usability	Appropriateness Recognizability Learnability Operability User error protection User interface aesthetics Accessibility		
Security	Confidentiality *Integrity* Non-repudiation *Accountability* Authenticity		
Compatibility	Co-existence Interoperability		
Maintainability	**Modularity** *Reusability* Analyzability **Modifiability** *Testability*	**Modularity**: degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components **Modifiability**: degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality. ***Modifiability*** *is a combination of changeability and stability*	Cycle metrics and change-probability metric
Portability	Adaptability Installability Replaceability		

One key aim of software testing is to discover defects in software and expose failures (van Vliet, 2000). Defects can be present at various phases of software development; requirements, design, and implementation (Pfleeger, 2001) and during its operational usage, i.e., maintenance phase (van Vliet, 2000). For instance, during initial development, a requirement specification may be defective because of a missing or un-

implementable requirement. Undiscovered defects at previous phases thus have potential to spread to other stages in the software life cycle. The type of defect to be treated can be categorized using the orthogonal defect classification (ODC) (Chillarege et al., 1992). The ODC classifies defects into *function*, *interface*, *checking*, *assignment*, *timing/serialization*, *build/package/merge*, *documentation,* and *algorithm*. The orthogonal feature of ODC enables defects to belong to only one category. It is thus effective to discover the part of the development phase that requires attention (Pfleeger, 2001).

Van Vliet (2000) states that testing a **requirement** specification should be aimed at testing its completeness, consistency, feasibility, and testability. (Poston, 1987) grouped the common errors in a requirement specification as missing information, wrong information, and extra information.

In the **design** phase, a high-level conceptual model of the system is developed from the requirement specification. This model shows how the system is decomposed into subsystems, components, and modules, and the interactions among them (van Vliet, 2000). This decomposition allows the architecture to be tested against specific quality attributes (Bass et al., 2003). Designing a system for testability requires that the design is not too complex and the states of the system are controllable and observable (Bass et al., 2003)

The **implementation** phase involves translating the design to executable source code. A number of testing techniques are applied such as *code-inspection* and *code-walkthrough* (van Vliet, 2000). Different test stages are involved such as *unit testing*, *integration testing, system testing, function/acceptance testing*, and *installation testing* (Pfleeger, 2001; van Vliet, 2000).

The **maintenance** phase is the post-release stage of the system. At this stage, changes can be introduced in the system due to defects, new/changed requirements, or changes in technology. These changes are captured using different maintenance terminologies; corrective, preventive, adaptive, and perfective (van Vliet, 2000). When changes are made to the system, it would need to be retested to ensure its correctness. The testing performed at this stage is termed *regression testing* (van Vliet, 2000).

To successfully manage software development and testing activities, a configuration management system (CMS) is required. A CMS allows the management of versions and releases of software, and enables coordination among testers and developers (Pfleeger, 2001).

Bertolino (2007) argued that the term “software testing” is used for a variety of aims and scopes, thus giving rise to multiple of meanings. However, the common denominator for the different testing goals is that testing always consists of observing a *sample of executions*, and giving a verdict over them. The author provided a unifying classification using six questions underlying any test approach. The questions ***why***, ***how***, ***how much***, ***what***, ***where***, and ***when,*** can be used to distinguish the specific aspects that characterize the sample of observations. The “***why”*** deals with *test objective*: why is it that we make the observation? The “***how”*** deals with *test selection*: which sample do we observe, and how do we choose it? The “***how much***” deals with *test adequacy, or* stopping rule: how big of a sample? The “***what***” addresses *levels of testing* (unit test,

component/subsystem test, and integration test) otherwise described as testing stages by (van Vliet, 2000): what is it that we execute? The "***where***" asks: where is the observation performed (in-house, simulated environment, or the final target context)? Lastly, the "***when***" asks: when is it in the product lifecycle that we perform the observations?

This thesis has investigated defects exposed in the operational stage of the software.

2.4 Object-Oriented Metrics

Object-oriented (OO) metrics have been proposed as a quality indicator for software components. The pioneering work in software metrics: Halstead's software science metrics (Halstead, 1977), McCabe cyclomatic complexity metric (McCabe, 1976), and Henry and Kafura's information flow metric (Henry and Kafura, 1981) have concentrated on complexity measures in the procedural paradigm. However, the OO paradigm expresses certain different programming philosophies such as inheritance, class or message passing that are not expressed in the procedural paradigm (Li and Henry, 1993).

Chidamber and Kemerer (1994) proposed a suite of OO metrics to indicate the quality of a component (class). These metrics are; Weighted Methods for a Class (WMC), Depth of class in Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Object Classes (CBO), Response For a Class (RFC) and Lack of Cohesion of a Method (LCOM). Li and Henry (1993) used the six metrics in addition to others to predict maintenance effort in OO systems. They concluded that the OO metrics are able to predict maintenance effort more than what size metrics can predict. Basili et al. (1996) validated the six OO metrics and claimed that CBO and RFC significantly correlate to defects than the rest four metrics. Briand et al. (Briand et al., 1998; Briand et al., 2001b) have investigated the set of metrics by Chidamber and Kemerer with several other derived metrics. They claim that CBO and especially import and method invocation coupling are important when building an OO quality model.

Challenges with software metrics for building software quality models are addressed in (Fenton and Neil, 1999a; Fenton and Neil, 1999b). The author argues that:

1) Complexity and/or size measures alone cannot provide accurate predictions of software defects
2) Information about software defects (discovered pre-release) on its own provides no information about likely defects post-release.
3) Traditional statistical (regression-based) methods are inappropriate for defects prediction

Fenton propose that the way forward is to construct prediction models that account for explanatory factors, most notable testing effort and operational usage.

2.5 Software Evolution and Maintenance

Evolution is a natural phenomenon for software system that is used. According to Lehman (1980), software that is used undergoes continual change or it becomes progressively less useful. Arguably, the terms evolution and maintenance are used interchangeably (Sommerville, 2011). Software systems undergo changes in many

ways. They have to adapt to new environments or technologies, or undergo change because of defect fixes. The activity of changing a software system after its release is termed maintenance. Maintenance can be classified into four types; corrective, adaptive, perfective and preventive (van Vliet, 2000). However, these types of maintenance (with the exception of corrective maintenance) have no distinct difference in practice (Sommerville, 2011). According to (van Vliet, 2000), the 'real' maintenance activity is corrective maintenance and it accounts for about 21% of the total maintenance effort only. While perfective consumes 50%, adaptive 25%, and preventive 4%.

Each of the maintenance types is defined as follows (ISO/IEC/IEEE 24765:2010):

Corrective maintenance: the reactive modification of a software product performed after delivery to correct discovered problems.

Adaptive maintenance: modification of a software product, performed after delivery, to keep a software product usable in a changed or changing environment.

Perfective maintenance: software maintenance performed to improve the performance, maintainability, or other attributes of a computer program.

Preventive maintenance: the modification of a software product after delivery to detect and correct latent faults in the software product before they become operational faults.

The more a system is changed, the more it grows in complexity (Lehman and Ramil, 2001) and the more it ages (Parnas, 1994). Parnas states that the key to control software aging is to design it for change (Parnas, 1994). This is consistent with Lehman's Seventh Law – Declining Quality (Lehman and Ramil, 2001). Systems should be adapted to account for changes in the operational environment to prevent a decline in quality. It presupposes that the advice to reduce maintenance problems for systems during evolution is relevant during the initial system development (van Vliet, 2000). Some of the possible solutions to reduce maintenance problems as stated by (van Vliet, 2000) are:

- Higher-quality code, better test procedures, better documentation and adherence to standards and convention, can pay off for corrective maintenance;
- Evaluation of software architecture with respect to ease of change can make future perfective and adaptive maintenance to be realized more easily;
- Finer tuning to user needs may lead to savings in perfective maintenance;
- Code size is correlated to maintenance. Less code means less maintenance. Reusing a bulky code has a maintenance penalty.

In conclusion, we can assume that maintenance challenges may be unavoidable but they can be controlled and minimized with adequate adherence to design guidelines during initial development. In the evolution phase, it is then necessary to implement an iterative improvement program for refactoring and improving the code quality (see Figure 5).

2.6 Software Patterns and Anti-patterns

2.6.1 Software patterns and empirical investigation on software maintenance

Design pattern was first formulated by (Alexander et al., 1977). Patterns are known to be recurring solutions to recurrent design problems. They capture existing, well-proven designs (Rising, 1998). Gamma et al. (1995) catalogued twenty-three design patterns widely used in software development and classified them into three broad types; these are creational, structural, and behavioural patterns. Creational patterns include patterns such as Factory Method, Abstract Factory, Builder, Prototype and Singleton. Structural patterns include, e.g., Adapter, Bridge, Composite, Decorator, Façade, Flyweight, and Proxy. Lastly, behavioural patterns include, e.g., Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, and Visitor.

According to Prechelt et al. (2002), the following are the advantages being claimed for design patterns:

1. Using patterns improves programmer productivity and program quality.
2. Novices can increase their design skills significantly by studying and applying patterns.
3. Patterns encourage best practices, even for experienced designers.
4. Design patterns improve communication, both among developers and from developers to maintainers.

Thus, one of the claimed benefits of design patterns is that it reduces the effort and cost of software maintenance.

2.6.2 Empirical validation of design patterns on software maintenance

A plethora of studies have investigated the role of design patterns on software maintenance and change-proneness. Bieman et al. (2003) investigated the impact of design patterns on the change proneness of classes by using five systems, four small ones and one large system. They have mined the change data from a configuration management system. They concluded that classes participating in design patterns are rather more change-prone. A recent study on mining software repository (Herzig and Zeller, 2013) shows, however, that multiple tangled code changes could result into an incorrect classification of change/fault data.

Di Penta et al. (2008) investigated whether certain design pattern roles are more change-prone in general, and whether certain roles are prone to particular types of changes. Their results confirm that many design pattern roles do undergo changes within the pattern. Vokac (2004) analysed the defect rates of classes that participated in selected design patterns of a large commercial product. The study concluded that the Observer and Singleton patterns are correlated with large code structures and can thus serve as indicators for special attention. On the other hand, the Factory pattern instances tend to have lower defect counts. (Prechelt et al., 2002) reported a controlled experiment that showed Observer and Decorator patterns to result in less maintenance time while the results for the Visitor pattern were inconclusive. (Vokáč et al., 2004) replicated the

experiment by (Prechelt et al., 2002). Their results confirmed the previous results that the Observer, Decorator, and Abstract Factory patterns favour ease of maintenance. However, the Visitor and Composite patterns had strongly negative results on maintenance.

Jeanmart et al. (2009), however, reported a positive relationship between the use of Visitor pattern and maintenance efforts. Wendorff (2001) reported on a large commercial software project where the uncontrolled use of design patterns has contributed to a severe maintenance problem.

2.6.3 Software Anti-patterns

Conversely, software anti-patterns are recognized as poor design choices and can exist at the code, design, and architectural levels (Koenig, 1998; Lippert and Roock, 2006). (Lippert and Roock, 2006) have catalogued a number of anti-patterns at the architectural level termed as *"architectural smell"*. In the next sections, selected anti-patterns at both the code and architectural levels are discussed. Furthermore, the section discusses empirical studies that have related anti-patterns to software quality.

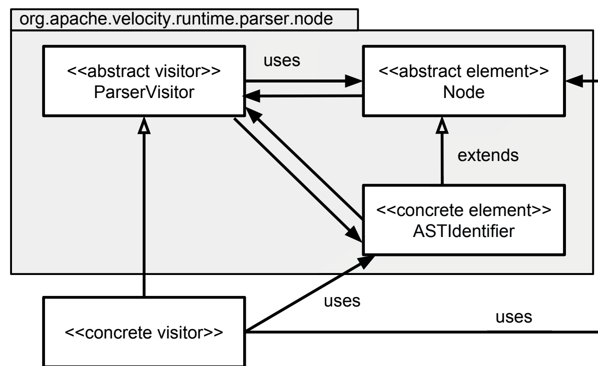


Figure 5 A cycle in Apache Velocity v1.6.2 (Oyetoyan et al., 2015b)

2.6.3.1 Dependency cycles

As illustrated in a concrete example in Figure 5, a cyclic dependency is formed when components depend on one another in a circular manner. The cycle in this figure is caused by the Visitor pattern that involves the abstract visitor, the abstract element, and the concrete element. This relationship covers both direct and indirect connection between those components. Formally, in graph theory (Cormen et al., 2001), a cyclic dependency graph, also known as strongly connected components (SCC) in a directed graph $G = (V, E)$, is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, both are reachable from each other. Cyclic relationships increase coupling complexities and thus have the potential to propagate defects in a network (Abreu and Melo, 1996).

In terms of classes, Parnas (1979) identified "Uses relation" between two components and argues that the loops in the "Uses Relation" are detrimental to extensibility of a software system. Lakos (1996) provided extensive discussion concerning cyclic dependencies among C++ classes. Lakos claimed that cyclic physical dependencies among classes in C++ programs inhibit understanding, testing, and reuse. Other authors

also claimed that cycles inhibit system understanding (Fowler, 2001), testing in isolation, integration testing (Briand et al., 2001a; Hanh et al., 2001; Hashim et al., 2005; Kung et al., 1996), and reuse (Martin, 1996). Cyclically connected components are mutually dependent, thus in terms of understanding any of the classes; it is necessary to understand all other classes in the cycle. Furthermore, to test a class in isolation is practically impossible when it is involved in a cycle with other classes (Lakos, 1996). In integration testing, cycles prevent the topological ordering of classes that can be used as a test order (Briand et al., 2001a, 2003; Hanh et al., 2001; Jungmayr, 2002; Kung et al., 1996; Melton and Tempero, 2007b), thereby inhibiting the testability of a system.

In many OO systems developed with programming languages such as Java or C++, a package represents a physical organization of software components (Knoernschild, 2012; Lakos, 1996). Packages are used to group classes that perform similar functions. They focus on manpower and they represent the granule of release (Martin, 1996). Applications are usually a network of interrelated packages and the work to manage, test, build, and release those packages is non-trivial (Martin, 1996). When cycles are formed at the package level, it seriously affects manpower since software engineers working on individual packages need to build with every other dependent package before they can release their package. Cycles among packages have thus been claimed to be detrimental to understandability (Fowler, 2001), production (Lakos, 1996; Martin, 1996), marketing (Lakos, 1996), development (Lakos, 1996; Martin, 1996), usability (Lakos, 1996; Martin, 1996), and reliability (Lakos, 1996).

It has been stated (Briand et al., 2001a; Hashim et al., 2005; Kung et al., 1996; Lakos, 1996) and implied (Jungmayr, 2002; Martin, 1996) that cycles are pervasive in real-life software systems. However, it appears that only Melton and Tempero (2007b) have performed an elaborate empirical study of cycles in many software systems at the class level. Melton and Tempero carried out an empirical study of 78 Java applications. The result shows that almost all the 78 Java applications contain large and complex cyclic structures among their classes.

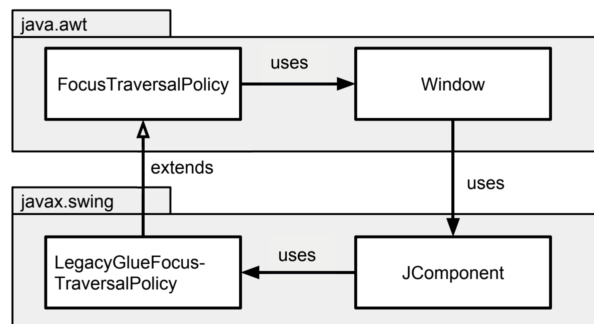


Figure 6 An STK in the JRE v1.7.0 (Oyetoyan et al., 2015b)

2.6.3.2 Subtype knowledge

Subtype knowledge (STK) is an instance of a cycle and was first studied by (Riel, 1996). It occurs in a cycle that contains at least one inheritance (extends) or realization

(implements) edge, and a back reference path connecting the target of the edge to its source. This type of anti-pattern violates the principle that base/super types should not know anything about their derived/sub types. Empirical studies have shown that the subtype knowledge anti-pattern is common in real world programs (Dietrich et al., 2010). An example of STK anti-pattern is shown in Figure 6 where *javax.swing* is in an STK type of cycle with *java.awt*.

2.6.3.3 Abstraction without decoupling

The Abstraction Without Decoupling (AWD) anti-pattern occurs when a client depends on an abstract type and at the same time uses the concrete implementation of this abstract type (Dietrich et al., 2012). The drawback is that it would be difficult to dynamically upgrade or replace the concrete implementation without touching the client code. This case is depicted in Figure 7 where class **B** depends on both the abstract class **A** and the implementation of **A** (**Impl-A**). It is possible to use dependency injection to refactor this anti-pattern (Dietrich et al., 2012; Fowler, 2004).

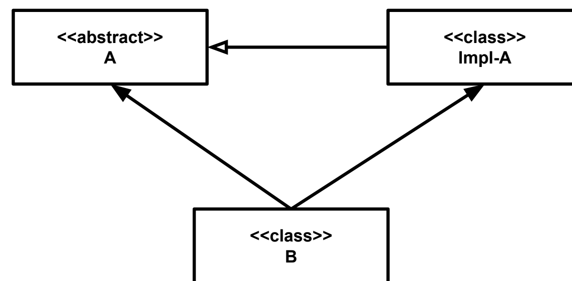


Figure 7 Abstraction Without Decoupling

2.6.3.4 Degenerated Inheritance

A Degenerated Inheritance (DIH) anti-pattern is caused by cases of multiple inheritance paths from a subtype to a supertype (Dietrich et al., 2012; Sakkinen, 1989; Singh, 1994). In object-oriented programming languages such as Java this is caused by using multiple interfaces. An example is depicted in Figure 8 where class **D** indirectly inherits from class **A** through two other classes **B** and **C**.

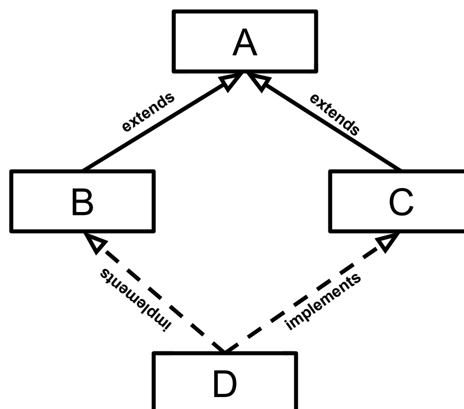


Figure 8 A case of degenerated/multiple inheritance

2.6.3.5 Cycles in package containment tree

Recent studies have also investigated new heuristics to classify “bad” and “harmless” package cycles (Falleri et al., 2011). The motivation behind this is that most cycles are formed within the package containment tree (PCT) hierarchies. By computing the diameter of a package cycle, it is possible to detect package cycles that cross module hierarchies, as a high diameter would be an indicator of “harmful” package cycles. The example in Figure 9 shows the two packages *“java.awt”* and *“javax.swing”* to have circular reference between them.

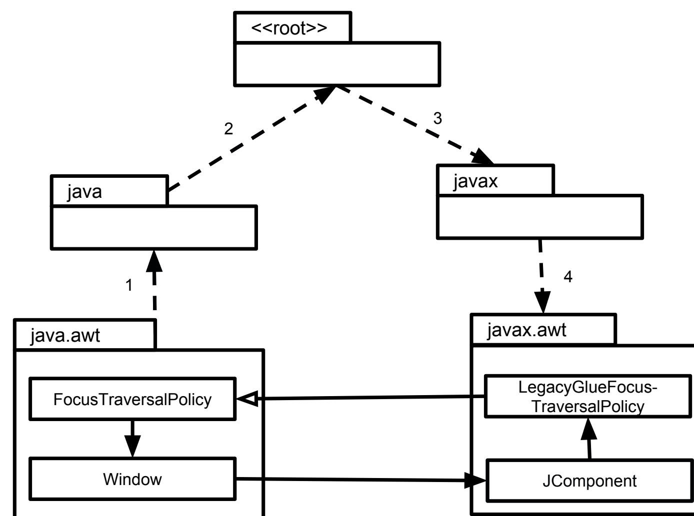


Figure 9 PCT of package cycles (Oyetoyan et al., 2015b)

2.6.3.6 Code smells

Beck and Fowler (1999) introduced the term code-smells and argued that code smells are indication of deeper problems in the source code. For instance, “Duplicated Code” is a case where similar code structures exist in different parts of the program. This type is also referred as code clone (Baxter et al., 1998) and can be refactored by extract method approach, e.g., the approach described in (Tsantalis and Chatzigeorgiou, 2011). Wake (2004) discussed Fowler’s code smells under six categories; “Duplication”, “Data”, “Interfaces”, “Responsibility”, “Unnecessary complexity”, and “Message calls”. Similarly, Mäntylä and Lassenius (2006); (Wake, 2004) provided a taxonomy for the code smells described by Beck and Fowler. The authors specified five categories for the twenty-four code smells as: “The Bloaters”, “The Object-Orientation Abusers”, The Change Preventers”, “The Dispensables”, and “The Couplers”. Wake states that not all code smells indicate a problem, but most are worthy of a look and a decision (Wake, 2004).

Several authors have investigated the relationship between code smells and the change-proneness of software artefacts. Khomh et al. (2012) examined classes involved in code smells, and their change and fault proneness. The study investigated four systems and thirteen code smells. The claims from this study are that classes participating in anti-patterns are more change- and defect-prone than others, and that structural changes

affect more classes with code smells than others. Romano et al. (2012) investigated the impact of code smells on change-proneness. The result of this study is consistent with (Khomh et al., 2012). In addition, they showed that certain code smells are prone to certain types of changes such as API changes. Olbrich et al. (2010) performed a study on two open source applications to study the impact of code smells. Their results showed that different phases during the evolution of code smells could be identified, and in particular, components infected with code smells display a higher change frequency than others.

2.7 Refactoring

Fowler refers to refactoring as a disciplined way to clean up code in such a way that the chances of defects is reduced (Fowler, 1999). Refactoring is an act of safely improving the design of existing program (Wake, 2004). It is a process that improves the internal structure of a software system without changing its external behaviour (Fowler, 1999; Mens and Tourwe, 2004). It is believed that refactoring improves software quality and increases productivity by making it easier to understand and maintain software codes (Kim et al., 2012). It is considered an important part of a software lifecycle or else the program design will decay (Fowler, 1999). Refactoring can be applied to smells at the code or at an architectural level (Lippert and Roock, 2006).

The targets of refactoring are software artefacts that could be program source code, design artefacts, and/or requirement specifications (Mens and Tourwe, 2004). Finkelstein et al. (1994) and Hunter and Nuseibeh (1998) have used classical logic and quasi-classical logic to identify inconsistencies in requirements specification, and proposed approaches to support continued action. Russo et al. (1998) reported their work on restructuring of multi-perspective requirements specification from a NASA project. They identified inconsistencies in the requirements by decomposing the specifications into “viewpoints”, and identifying the relationships and inter-dependencies among the “viewpoints”. The restructuring facilitates better requirements understanding, maintenance and evolution.

Design level refactoring concerns the restructuring of design artefacts and most notably in the form of Universal Modelling Language (UML) models (Correa and Werner, 2004; Sunyé et al., 2001). Van Gorp et al. (2003) proposed an extension to the UML metamodel to maintain consistency between a refactored design and its underlying source code. Boger et al. (2003) developed a refactoring browser integrated in a UML modelling tool. Van Der Straeten and D'Hondt (2006) implemented a rule-based approach for resolving inconsistencies in or between models.

Refactoring at the source code or program level has been largely addressed (Beck and Fowler, 1999; Fowler, 1999; Opdyke, 1992; Wake, 2004). These refactoring make use of a number of approaches e.g. ***Extract class***, ***Move method***, ***Encapsulate field***, ***Extract method, Pull-up method, Extract Interface***, (Fowler, 1999) and are implemented on development environments (e.g. Eclipse, NetBeans, Visual Studio).

Graph transformation has been extensively applied to model code-level refactoring activities (Mens, 2006; Mens et al., 2007; Van Der Straeten et al., 2004; Zhang et al., 2005). The type of graph manipulation we have employed in this thesis does not

demand detailed graph formalism since we are only interested in removing or adding single edges in a graph during refactoring.

Dietrich et al. (2012) identified high impact edges from the program dependency graph by assigning weights to each edge based on the number of anti-patterns it is involved with. Their results on the graph model demonstrated that it is possible to remove many anti-patterns (e.g., dependency cycles at the package level) by removing such high impact edges. Shah et al. (2013) implemented an automated refactoring on these edges. Their results from applying several refactoring approaches, show that certain edges are removable, while removing certain edges would introduce errors.

Laval and Ducasse (2014) implemented an enriched dependency structural matrix (eDSM) to detect dependency cycles between packages. They use contextual information such as the types of relationships between the coupled components, the proportion of referencing classes in the client package, and the proportion of referenced classes in the provider package. In addition, the matrix table uses different colour annotations to differentiate between direct cycles, indirect cycles, and other types of dependencies. Finally, the tool reports actions and propositions to be performed to remove detected dependency cycles.

In relation to refactoring and software defects there are conflicting evidence of the benefits of refactoring. Weissgerber and Diehl (2006) found no correlation between refactoring and defects opened in the subsequent days. Their results show that there are periods where high refactoring was followed by an increase in the number of defects as well as phases where refactoring led to no defects, although, the latter type were more prevalent. Ratzinger et al. (2008) demonstrate that the number of software defects decreases in the preceding time period when the number of refactoring activities increases. Bavota et al. (2012) show that some kinds of refactoring are unlikely to be harmful, but certain kinds such as refactoring involving hierarchies (e.g. pull up method) are likely to induce defects. Kim et al. (2011) found that refactoring edits have a strong temporal and spatial correlation with bug fixes. In another study, Kim et al. (2012) discovered that refactored binary modules of Windows 7 experienced significant reduction in the number of inter-module dependencies and post-release defects.

2.8 Summary of Research Challenges

This chapter has presented software quality, software testing, software evolution and maintenance, software patterns, anti-patterns, and refactoring. In this section, the challenges that are relevant to this thesis are presented. These challenges are within the context of the presented topics in this chapter. We define the research challenges (RC) as follows:

RC1: Metrics to identify critical software components: Several studies have focused on building software quality models by using software metrics as predictor variables, and defect counts or defect density as response variables: a comprehensive survey of studies can be found in (Hall et al., 2011). One challenge is that other important defect metrics such as *correction effort* and *defect severity* are not included. We can relate this to the challenges discussed in (Fenton and Neil, 1999a; Fenton and Neil, 1999b). We consider the *correction effort* and *defect severity* metrics as crucial to understand the **maintainability** and **reliability** attributes of software systems. Another challenge is that

software components differ in the degree of their criticality to the system. We have not found any study that investigates how metrics can be used to identify such critical components in software. In the context of our study, *criticality of defects* and *criticality of components* are important concepts that we want to investigate.

RC2: Empirical evidence of defect- and change-proneness of dependency cycles: Existing studies have claimed that dependency cycles can be harmful for software quality, e.g. (Lakos, 1996; Parnas, 1979). We need to understand the impact of dependency cycles on external quality metrics such as defect and change metrics. This understanding can help us to identify defect hotspots and guide refactoring efforts. To the best of our knowledge we have not found any systematic study of anti-patterns at the architectural level and their relationships to defect and/or change-proneness.

RC3: Refactoring of dependency cycles: Refactoring of dependency cycles have been largely focused at the package granularity level (Dietrich et al., 2012; Falleri et al., 2011; Laval and Ducasse, 2014) while there has been little focus for this activity at the class granularity level (Melton and Tempero, 2007c). Since class files represent maintenance units in systems developed with OO languages (e.g., Java and C#), it is important to consider refactoring at this granularity level. An approach and tool for refactoring dependency cycles would be important to improve the structural quality of the identified hotspots in software systems.

RC4: Software Testing and Quality Assurance: Challenges related to software testing can be grouped into the "*why*", "*how*", "*how much*", "*what*", "*where*", and "*when*" questions raised in (Bertolino, 2007). How to focus testing in the right sample of observation is important to reduce the possibility of software failure. In practice, it is impossible to achieve 100% test coverage (Roger, 2005), because quality assurance resources are limited and there is pressure of time-to-market. Efforts are therefore needed to identify specific code locations that should be focused for thorough testing.

RC5: Software maintenance and evolution: Software grows in complexity during evolution. This complexity leads to a declining quality (Lehman, 1980). The main challenge is the "***technical debt***" (Brown et al., 2010) that is not paid by the software organization during the software life cycle. Technical debt describes a state of software where its future is negatively affected by the past decisions. For example, early software release versus maintainability. There is a need to develop approaches and tools to help manage *technical debt* during software evolution.

RC6: Software patterns and anti-patterns: Patterns are claimed to be good solutions to design problems while anti-patterns are recurrent problems in software design. However, conflicting evidence exists as to the impact of patterns on software maintenance (see (Jeanmart et al., 2009; Prechelt et al., 2002)). There are also cases of patterns with instances of anti-patterns. A thorough empirical study of patterns with anti-patterns in relation to maintenance would be useful to guide software design decisions.

3 Research Context and Design

This chapter presents the research context and design of this thesis. In Section 3.1, the research focus and questions are presented. Section 3.2 presents the software systems that have been used to investigate the research questions in this thesis. Section 3.3 presents the metrics that are used in the studies. In Section 3.4, the approach of data collection is presented. Section 3.5 discusses research methods commonly used in software engineering. Section 3.6 presents the research design and lastly, Section 3.7 presents the scope, concepts, and limitation of this thesis.

3.1 Research Focus

The main goal of this thesis, as stated in Section 1.2, is to **Improve the Management of Software Evolution for Smart Grid Applications**. A *Smart Grid* represents the injection of Information and Communication Technology (ICT) infrastructure to the electricity grid to allow for bi-directional flow of energy and information (NIST, 2010). A Smart Grid is a system-of-systems (SoS) where heterogeneous systems must interconnect and interoperate together (NIST, 2010). Continuous changes in the open-world settings for heterogeneous systems need approaches to make them dependable (Bertolino et al., 2011). One key challenge of SoS is the ripple effect of change (Creel and Ellison, 2008). A change or failure in one system can cascade to some other systems in a SoS. Such effects can threaten the reliability of the SoS. Understanding defect/change-prone locations in individual systems of a SoS is thus a step in the right direction. Such knowledge can guide quality assurance and motivate for improvements in the different locations.

A consistent approach taken in this thesis is to investigate whether the individual systems have design structures that lend themselves to modifiability and ultimately maintainability. To achieve this, we have quantitatively analysed a structural anti-pattern called "**dependency cycle**" against ***maintainability*** and indirectly ***reliability***. Maintainability and reliability are hard concepts to measure, however, it is possible to use proxy metrics such as defect metrics and change metrics to indirectly quantify maintenance and reliability. In pursuance of the research goals, we have performed an analysis of the defect repository and source code of a distribution management system of Powel AS in Trondheim.

This thesis focuses on the maintenance and evolution of software systems. We are interested to know how dependency cycles among software components affects the non-functional requirements of software systems. In particular, we focus on maintainability and reliability quality attributes. We investigated this goal by performing empirical studies on released software systems. Figure 10 shows a summarized software development process model, the area of the investigation, and the contributions of this thesis. We show in this figure (see a - e) how an improvement program can be initiated by the findings from the empirical studies of released software systems as follows:

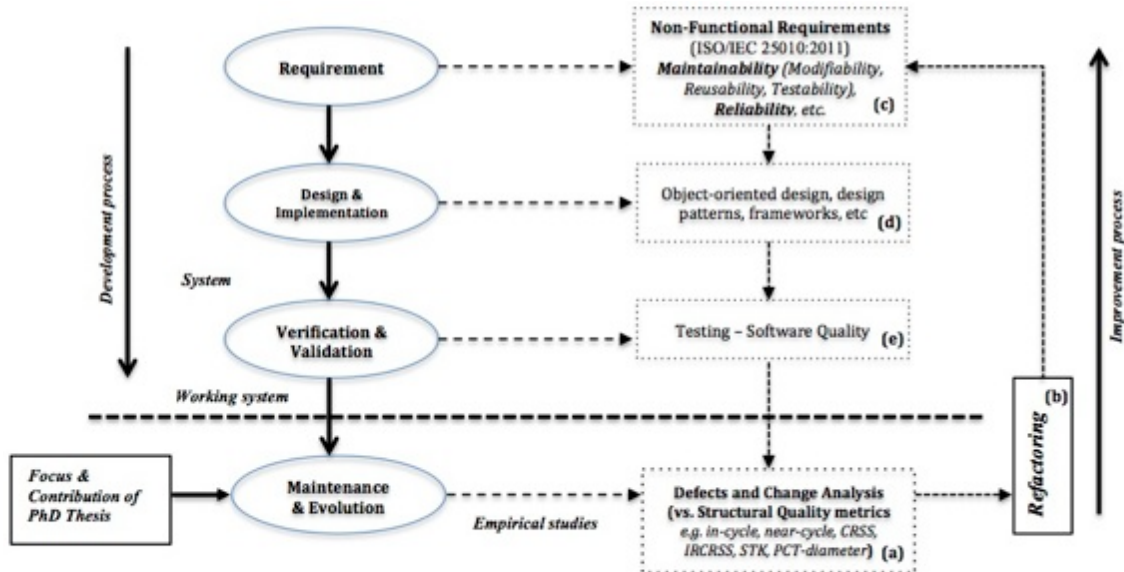


Figure 10 The software development process with relationships to research contributions

We can identify the locations for code restructuring through measurements of the structural quality of the system and analysis (**a**) against external quality metrics (defect and change frequency). The refactoring (**b**) could aim at improving the quality attributes (**c**) (e.g. maintainability and reliability that are focused in this thesis). During the refactoring to improve the software quality (non-functional requirements), it is possible to introduce new design patterns and implement best design practice (**d**). Regression testing (**e**) would have to be performed to ensure the system's behaviour is preserved (van Vliet, 2000). This iteration is possible for the entire life cycle of the software.

Analysing the internal quality metrics against the external quality metrics (defect and change frequency) provide opportunities to pinpoint specific locations (components) that should be focused for improvement. This will reduce the probability of false positives and false negatives. This is the approach we have taken in this thesis to achieve the research goal. This is rather different from using internal quality metrics as the only decision variables for performing refactoring. The shortfall of the latter approach is that many code locations have to be improved based on the results from applying the metrics. However, quality assurance resources are indeed limited and in many cases it would be impossible to improve all the suggested area. In addition, we cannot really tell whether the locations we have refactored are more defect-prone or change-prone than the untouched ones.

We have discussed the two quality attributes (i.e., maintainability and reliability) in this thesis because we are able to quantify them using external quality metrics (***defect*** and ***change rate***) as indirect metrics.

We have outlined and investigated two main research questions in this thesis. The motivation for the questions are discussed below:

RQ1. What is the effect of dependency cycles on external quality of software components?

The sub-questions to explore **RQ1** are motivated as follows:

RQ.1.1 What is the effect of using different defect metrics to identify critical software components?

Defect distribution in software systems has been shown to follow the Pareto rule of 20-80. This motivates the prioritization of components with the majority of defects for testing activities. Several studies have also suggested that smaller components have higher likelihood of defects when compared to the larger ones (Basili and Perricone, 1984; Hatton, 1997; Moller and Paulish, 1993; Ostrand and Weyuker, 2002). These studies have used defect density measured as number of defects per thousand lines of code (LOC). It has been demonstrated that most complexity metrics (e.g. McCabe complexity metrics) correlate with a component's size (Fenton and Pfleeger, 1998). It then means that more complex components and invariably larger components are given higher priority in prediction models that use defect count approach (largest-first prioritization).

However, removing a large number of defects may have little effect on the reliability of the system, since most failures are caused by a tiny number of defects (Adams, 1984). This demonstrates the significance of a defect severity metric.

The question we seek to investigate is whether there are significant variations between defective components and architectural hotspots identified by multiple defect measures. In addition, we seek to investigate whether defect metrics classify differently the defective components that developers consider critical in terms of their functionality to the system.

RQ1.1 forms a basis for the remaining empirical studies in this thesis that explore component-defect analyses.

RQ.1.2 What is the effect of dependency cycles on software defects?

Best design practice advocates to avoid dependency cycles between software artifacts (Bass et al., 2003; Lakos, 1996; Martin, 2000; Parnas, 1979). Many authors (Bass et al., 2003; Parnas, 1979) have claimed that such complex structure inhibit software quality (e.g. reliability testability, modifiability or reusability). Empirical evidence shows that dependency cycles are common among software components (Melton and Tempero, 2007b). However, the question of how dependency cycles correlate with defects remains open. Since a dependency cycle is structurally complex, we hypothesize that it would contain the majority of defects and defect-prone components. The findings from this question can assist developers, maintenance engineers, and software project managers to effectively allocate resources during software quality assurance.

RQ.1.3 What is the effect of refactoring cycles on defect-proneness?

This research question has been developed from the results of **RQ1.2,** to empirically investigate whether cycle-breaking refactoring lowers defect-proneness of components. The hypothesis is that transitioning a component from a dependency cycle to an *out-of-cycle* structure would reduce its structural complexity and improve its testability. Thus, it should result in a lowered defect-proneness. It is important to explore and understand whether any empirical evidence exists to support cycle refactoring in relation to defect proneness.

RQ.1.4 What is the effect of dependency cycles on change rate?

The research questions above have focused on corrective maintenance. Lientz et al. (1978) showed that correction efforts consume about 17.4% of the total maintenance effort. Van Vliet (2000) stated that correction efforts account for about a quarter (25%) of the total maintenance effort. We have thus dealt with a subset of normal software maintenance activity. However, how dependency cycles relate to other types of changes (perfective, preventive and adaptive) has not been explored. Our conjecture is that since dependency cycles have strong and complex structures, they would have potential for higher change propagation and ripple effects.

We also conjecture that certain types of cycles could be more change-prone because of their properties. Cycles with inheritance relationships (defined by the STK metric), cycles that spread across a large package structure (defined by the PCT-diameter metric) and cycles formed by patterns (e.g. Visitor) are of particular interest. Our hypothesis is that it is possible to use these properties to classify "critical" or "harmless" cycles.

RQ2. How to refactor dependency cycles to impact the structural quality and reduce the refactoring efforts?

This research question is formulated based on the results from the previous research questions. The goal is to improve the structural quality of components in dependency cycles. Dependency cycles are detrimental to many software quality attributes such as modifiability, reusability, testability and reliability (Lakos, 1996; Parnas, 1979). It inhibits the formation of components to have manageable size and stand-alone properties (Melton and Tempero, 2007a). It is thus crucial to refactor those locations that are particularly defect and change prone. Performing a cycle breaking refactoring on existing systems at the class granularity level is not a trivial activity. It is therefore necessary to provide a decision support tool that could assist a developer or maintenance engineer to perform such refactoring activities. We seek to provide and implement approaches that could be used for cycle-breaking refactoring and at the same time would reduce the refactoring effort.

3.2 Summary of Software Systems

We have investigated 30 systems in the various studies in this thesis. These systems are different in functionality, age, domain, programming language, usage, and context. Table 3 provides a summary of the properties of the systems, and the papers where they have been studied and reported. One of the systems is a commercial Smart Grid application (with the pseudonym "CommApp"), which has been both quantitatively and

qualitatively studied. Twelve systems from the list are selected from the Qualitas corpus (Tempero et al., 2010). Using this standard dataset facilitates the replication of studies. The rest of the systems are selected based on common criteria such as popularity, age, support base, and programming language.

3.3 Metrics and Measurement

Software processes and products need *measurements* to characterize, evaluate, predict, and identify areas of improvements (Park et al., 1996). Naik and Tripathy (2011) stated three reasons for quantitative measurements of software quality: (1) Measurements allow an establishment of baselines for quality, (2) Measurements are a key to process improvement, and (3) the needs for improvement can be investigated after performing measurements. Similarly, Fenton and Pfleeger (1998) indicated that measurements make concepts more visible and as a consequence they are more understandable and controllable. Fenton and Pfleeger (1998) defines a ***measurement*** as:

"the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules"

A *software metric* is a measurable property, which indicates the software quality criteria to be measured (Gillies, 1997). In this thesis, various metrics have been defined and employed to study software product quality with respect to reliability and maintainability. In the subsequent sections, the different definitions of metrics used, are presented and linked to the specific quality attribute they indirectly measure.

Table 3 Properties of systems used in the thesis

System	Description	Language	License	Papers
CommApp	An industrial Smart Grid system	C#	Commercial	P1, P2, P3, P5, P7
Eclipse	Integrated development environment (IDE)	Java	Open	P2, P4, P5
Apache-Camel	Routing and Mediation Engine	Java	Open	P2
Apache-ActiveMQ	Messaging and Enterprise Integration Pattern Server	Java	Open	P2, P3, P4, P5
Apache-Lucene	Search Engine	Java	Open	P2, P6
Apache-CXF	Service framework	Java	Open	P5
openPDC	Smart Grid	C#	Open	P2
Azureus (Vuze)	File Streaming tool	Java	Open	P7
JStock	Stock market application	Java	Open	P7
VidCoder	Ripping and video transcoding application for Windows	C#	Open	P7
Hibernate	Object/Relational Mapper tool	Java	Open	P6, P7
Openproj	Desktop project management application similar to Microsoft Project	Java	Open	P7
JXplorer	Mature LDAP, LDIF and DSML client with i18n support	Java	Open	P7
Megamek	A networked Java clone of BattleTech, a turn-based sci-fi boardgame for 2+ players	Java	Open	P7

System	Description	Language	License	Papers
Weka	A collection of machine learning algorithms for solving real-world data mining problems	Java	Open	P6, P7
SomToolBox	Open-source implementation in Java, that allows you to easily train self-organizing maps, and analyze them	Java	Open	P7
GanttProject	A project scheduling application	Java	Open	P7
Squirrel-sql	A graphical SQL client	Java	Open	P7
OpenRocket	An Open Source model rocket simulator	Java	Open	P7
ermaster	Eclipse plug-in to make ER diagram	Java	Open	P7
Logisim	Educational tool for designing and simulating digital logic circuits	Java	Open	P7
Ant	Parsers/generators/make	Java	Open	P6
Antlr	Parsers/generators/make	Java	Open	P6
Argouml	Diagram generator/data visualization	Java	Open	P6
Freecol	Game	Java	Open	P6
Freemind	Diagram generator/data visualization	Java	Open	P6
Jgraph	Graph components	Java	Open	P6
Jmeter	Testing tool	Java	Open	P6
Jung	Diagram generator/data visualization	Java	Open	P6
Junit	Testing	Java	Open	P6

Table 4 Defect metrics

Metrics	Definition	Measurement scale	Papers
Defect counts	Number of post release defects recorded for a component	Interval	P1, P2, P3, P4, P5
Defect severity	The severity of recorded defect ranked on a scale (critical/blocker, high, medium or low)	Ordinal	P1, P3
Defect correction effort	The absolute number of hours for fixing a post-release defect	Interval	P1
Defect density	Number of defects per lines of code	Interval	P1, P2
Defect probability	The ratio of defective component to all components	Ratio	P2

3.3.1 Defect metrics as proxy for reliability

Measuring actual reliability (e.g., *Mean-Time-To-Failure*) is not the major focus of this thesis. It is therefore not possible to conclude on the actual impact of the defect metrics we have used with respect to the reliability of the systems. As discussed in Section 1.1, a study by Adams (1984) shows that most of the latent defects lead to very few failures in practice, while the vast majority of observed failures are caused by a relatively tiny number of defects. In this sense, it is important to consider defect metrics that could be

closely related to reliability. In Table 4, five defect metrics used in this study are defined and described.

3.3.2 Change as a proxy for maintainability

According to IEEE 610.12, ***maintenance*** is *"the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment"* (Radatz et al., 1990). Following this definition, we can use change (frequency of modifications) to only approximate maintainability. At a lower granularity level, the change metric has been defined as a probability measure for change in a component. A component may refer to a class or a package in object-oriented system. The probability of change for a component is defined as (Oyetoyan et al., 2015b):

"Given a program P, let C be the set of classes in P, and V be the set of versions of P such that for each version $v \in V$, a successor version succ(v) exists. For a given set of classes $S \subseteq C$ and a version $v \in V$ we use changed(S,v) to denote the set of classes in S that have changed from v to succ(v). The change probability of a class in S is then defined as a function $p_{change} : 2^C \times V \rightarrow [0, 1]$ defined as:"

$$P_{change}(S, v) = \frac{|changed(S, v)|}{|S|}$$

Table 5 Cyclic dependency Metrics

Metric	Definition	Measurement scale	Papers
in-SCC	A component is in a strongly connected relationship to other components (circular dependency)	Nominal	P2, P3, P4, P5, P6, P7
near-SCC	A component (not in-SCC) but directly depends on in-SCC component (Oyetoyan et al., 2013b; Oyetoyan et al., 2015b)	Nominal	P2, P3, P4, P6
out-of-SCC	A component that is not in-SCC and not near-SCC	Nominal	P2, P3, P5, P6
isSTK	An SCC that have a sub-type knowledge (Dietrich et al., 2010)	Nominal	P6
isVisitor	An SCC that is formed by Visitor pattern	Nominal	P6
PCT-Diameter	The normalized diameter of package tree for an SCC (Falleri et al., 2011; Oyetoyan et al., 2015b)	Interval	P6
CRSS	The set of classes that a class can reach transitively (Melton and Tempero, 2007a)	Interval	P5, P7
IRCRSS	The difference between the CRSS of a component and CRSS of its interface normalized by the CRSS of the component (Oyetoyan et al., 2015a)	Interval	P7

3.3.3 Cyclic metrics as proxy for maintainability

Modularity is a sub-characteristic of maintainability. A system where modules are in circular dependencies violate the acyclic dependency principle (Martin, 2000) and would be difficult to maintain (Bass et al., 2003). Table 5 describes the metrics that have been used and defined in connection to the studies presented in this thesis.

3.4 Data collection for Empirical Studies

The studies in this thesis have used data from configuration management systems, defect tracking systems (DTS), program source files, and binary files. Figure 11 shows the different sources of data for this study and their mapping. We have extracted dependencies using both byte code analysis (binary files) and text analysis (for source files). From these dependencies, a simple graph model of components (classes or packages) and their relationships is built. The higher level of abstraction (e.g. packages) is computed by aggregating the dependencies at the class level. Strongly connected components (SCCs) are then computed based on Tarjan's algorithm, which is suitable for this purpose and completes in linear time (Tarjan, 1972).

Defect data is collected from DTS and mapped to the source files that are changed because of corrective maintenance action. This mapping is usually done through a defectID logged against the changed class. The defectID is used as a key to map the class files to other defect metrics (e.g. defect severity and correction effort). Indirect defect metrics such as the defect density are computed from the aggregated defect count for each class by dividing with the lines of code (LOC).

The change data comes from differing two releases of a class using byte code analysis. Essentially, the members of a class in release *i* are compared to its members in release *i+1* to detect a change (e.g., change in the return type or parameters of a method).

The members considered are those that are API elements (public, protected or package-private) and could be methods, fields or constructors. These elements are the source through which change can ripple through the system.

In the final step, the graph model and the defect or change data are mapped. The data is presented for analysis. The kind of analysis to be performed depends on the measurement scale of the dependent and independent variables (Fenton and Pfleeger, 1998). We have used both the Wilcoxon rank sum and Spearman/Pearson correlation tests in R statistical package for our analyses.

3.5 Research Methods in Software Engineering

Empirical research follows three types of research paradigms; the **qualitative**, **quantitative** and **mixed-methods** research (Robson, 2011; Wohlin et al., 2003). **Qualitative** research is about studying objects in their natural setting (Wohlin et al., 2003). The goal of using this method is to explore and understand the meaning individuals or groups ascribe to social or human problems (Creswell, 2013). **Quantitative** research is concerned with quantifying a relationship between variables or comparing groups (Creswell, 2013; Robson, 2011; Wohlin et al., 2003). The variables are measurable and data generated is statistically analysed (Creswell, 2013). **Mixed-methods** research combines both quantitative and qualitative methods (Robson, 2011).

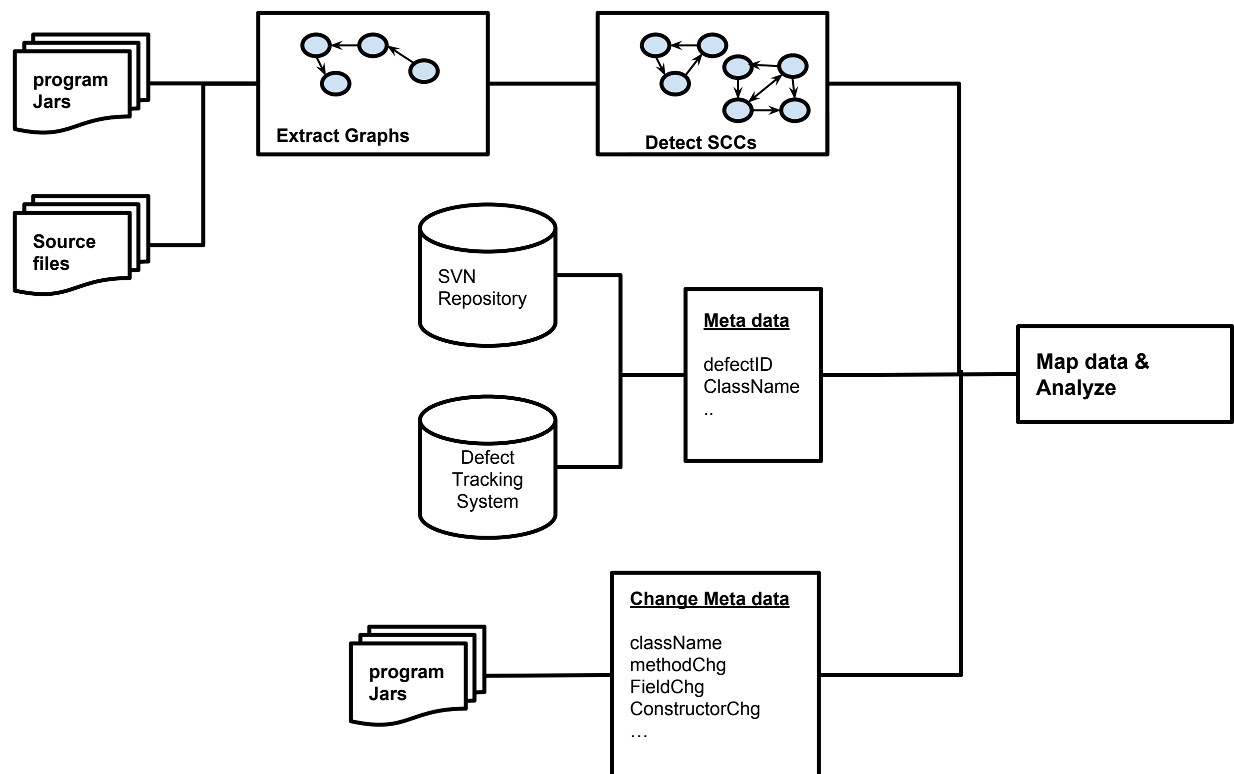


Figure 11 Data collection from different repositories

The approach integrates both qualitative and quantitative data to better understand the research problem (Creswell, 2013). It allows for triangulating results to enhance the validity of findings (Robson, 2011).

Empirical studies can be exploratory (i.e., investigating parameters), prescriptive (i.e. finding distributions of certain characteristics), or explanatory (investigating why certain phenomena occur). Different strategies can be used in an empirical study; **experiment**, **case study**, **survey**, **ethnography,** or **action research** (Easterbrook et al., 2008). We briefly discuss these strategies. In addition, we discuss another method; **design science** that is popular in information science but is as well applied in software engineering.

Experiment: Experiment is a rigorous, controlled investigation where one or more independent variables are manipulated to measure their effects on one or more dependent variables (Easterbrook et al., 2008; Fenton and Pfleeger, 1998). A controlled experiment is useful to determine cause-effect relationship between variables. Experiments are normally performed in the laboratory and are thus referred as research-in-the-small (Fenton and Pfleeger, 1998; Wohlin et al., 2003).

Case study: A case study is used to investigate *how* and *why* certain phenomena occur and it is sometimes referred to as research-in-the-typical (Easterbrook et al., 2008; Fenton and Pfleeger, 1998). In a case study research, key factors that may affect an outcome of an activity are identified; then the inputs, constraints, resources and outputs of the activity are documented (Fenton and Pfleeger, 1998). A case study is an observational study in contrast to an experiment that is a controlled study. It is usually

aimed at tracking a specific attribute or establishing relationships between different attributes (Wohlin et al., 2003).

Survey: A survey is referred as a research-in-the-large and it is often an investigation performed in retrospect (Wohlin et al., 2003). For example, it can be used to investigate the impact of a tool that has been in use for a while. A survey uses questionnaires or interviews as instruments to collect qualitative or quantitative data. In a survey, a *representative sample* must be drawn from a *well-defined population* (Easterbrook et al., 2008). The results are analysed to derive descriptive and explanatory conclusions (Wohlin et al., 2003) and can be generalized to the population (Easterbrook et al., 2008).

Ethnography: (Robson, 2011) defines ethnography as “an approach to the description and understanding of the life and customs of people living in various culture”. For software engineering, ethnography can be useful to understand how technical communities build a culture of practices and communication strategies that help them to perform technical jobs collaboratively (Easterbrook et al., 2008). Ethnographic studies are typically long term studies in their natural settings (Robson, 2011). Central to ethnographic study is a research question that is focused on the cultural tradition of the community and accessibility to the community (Easterbrook et al., 2008).

Action research: In action research, researchers are interested to solve a real-world problem while simultaneously studying the experience (Easterbrook et al., 2008). Improvement and involvement are central goals of action research (Robson, 2011). Action researchers get involved in the studied situation with the central goal of improving it (Easterbrook et al., 2008; Robson, 2011). Central to action research is the collaboration between the researchers and the problem owner who must be willing to engage in an effort to solve an identified problem (Easterbrook et al., 2008; Robson, 2011).

Design science: Design science is an engineering approach to creating and evaluating software artefacts (Peffers et al., 2007). The artefact may extend the knowledge base or apply existing knowledge in new and innovative ways (Hevner et al., 2004). Design science research methodology (DSRM) is made up of six activities (Peffers et al., 2007), these are: (**1**) *problem identification and motivation,* (**2**) *defining the objectives of a solution,* (**3**) *design and development,* (**4**) *demonstration,* (**5**) *evaluation*, and (**6**) *communication*. The research must produce an artefact to address a problem. The utility, quality and efficacy of the artefact must then be rigorously evaluated. Its contribution should be verifiable and it must be effectively communicated to the right audience (Hevner et al., 2004).

3.6 Research Design

The study was divided into four studies as shown in Figure 3 in Section 1.3. Table 6 provides the summary of the research design.

Study 1: In this study, the impact of using different defect metrics on software components is investigated. The study uses both case study and survey methodologies. It forms a background for the remaining studies that focused on defect analysis of software components.

Study 2: explores dependency cycles among software components and their relation to defects and change. The data comes from both defect tracking systems, configuration management systems, and the source/byte code. A case study methodology is used.

Study 3: investigates the effect that refactoring dependency cycles have on the defect-proneness of the components. A case study methodology is used.

Study 4: The last study has used design science as an approach to improve the refactoring of components in dependency cycles and evaluated using case study and interview.

3.6.1 S1: Empirical investigation of using different defect metrics to classify critical components

The goal for **Study 1** was to investigate identification patterns of multiple different metrics to identify critical components. **Study 1** addressed the research question **RQ1-1**. We investigated four different defect metrics using the post release defects of an

Table 6 Summary of research design

Study	Study description	Research methods	RQ 1-1	RQ 1-2	RQ 1-3	RQ 1-4	RQ 2	Paper	Contri-bution
S1	Empirical investigation of using multiple defect metrics to classify critical components	Case study & Survey (Quantitative & Qualitative)	X					P1	C1
S2	Investigation of defect and change proneness of cyclically dependent components	Case study (Quantitative)		X		X		P2, P3, P6	C2-1, C2-2, C3-1
S3	Investigating the effect that refactoring dependency cycles have on defects	Case study (Quantitative)			X			P4, P5	C2-1
S4	Improving the structural quality of cyclically dependent components using tools and metrics	Design Science, Case study & Survey (Quantitative& Qualitative)					X	P7	C3

industrial Smart Grid system. The four defect metrics are *defect counts*, *defect density*, *defect severity,* and *defect correction effort*. These metrics were quantified on software components by using four related measures. These measures are:

1) Defect-Prone Components (DPC), classified as the top 25% of components with the highest number of defects
2) Defect-Dense Components (DDC), classified as the top 25% of components with the highest defect density
3) Severe-Defective Component (SDC), classified as the top 25% of components with the highest number of critical defects
4) Hard-to-fix Defective Components (HFC), classified as the top 25% of components with the highest correction effort

We have used a case study to answer the research question since the data we sought could be gathered from the system's repositories. In addition to a case study method, we have used survey to collect the data about the criticality of the components to the system from the developers. We did this by asking the developers to rank a set of identified components by the metrics using a defined scale. We then quantitatively analysed the results.

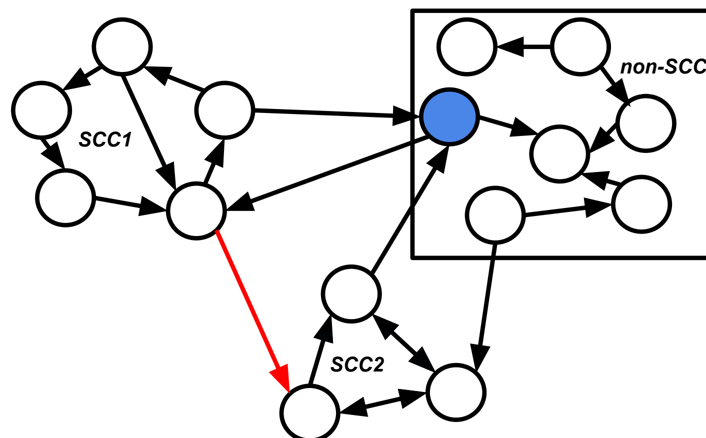


Figure 12 Components in and near dependency cycles

3.6.2 S2: Investigation of defect and change proneness of cyclically dependent components

The goal of **Study 2** was to assess the impact of dependency cycles on the defect proneness and change proneness of components. **Study 2** addressed the research questions **RQ1-2** and **RQ1-4.** The study is divided into three sub studies. Since the data needed for these studies could be mined from the various systems' repositories, we have therefore used a case study methodology**.** We selected six systems with different properties in the first sub-study and two out of the six systems for the second sub-study. Furthermore, we mined the defect data from the defect tracking system and associate them against the components that are changed in the subversion repository (see Section 3.4 for details). We defined a set of metrics to classify components into three groups, "*in-SCC*", "*near-SCC*" and "*out-of-SCC*". Where *"near-SCC"* refer to components that

are not in cycles but directly depend on components in cycles (see Section 3.3.3 and Figure 12).

In the third sub-study, we chose twelve systems from a standard curated dataset (Qualitas Corpus, see Section 3.2 for details). We then collect the change frequency data by extracting changed meta-data between two successive releases. We defined three new metrics (isSTK, isVisitor and PCT-diameter) in addition to the metrics in the first two sub-studies, to classify various SCCs.

Lastly, we apply Wilcoxon rank sum test, Spearman, and Pearson correlation test for the analyses.

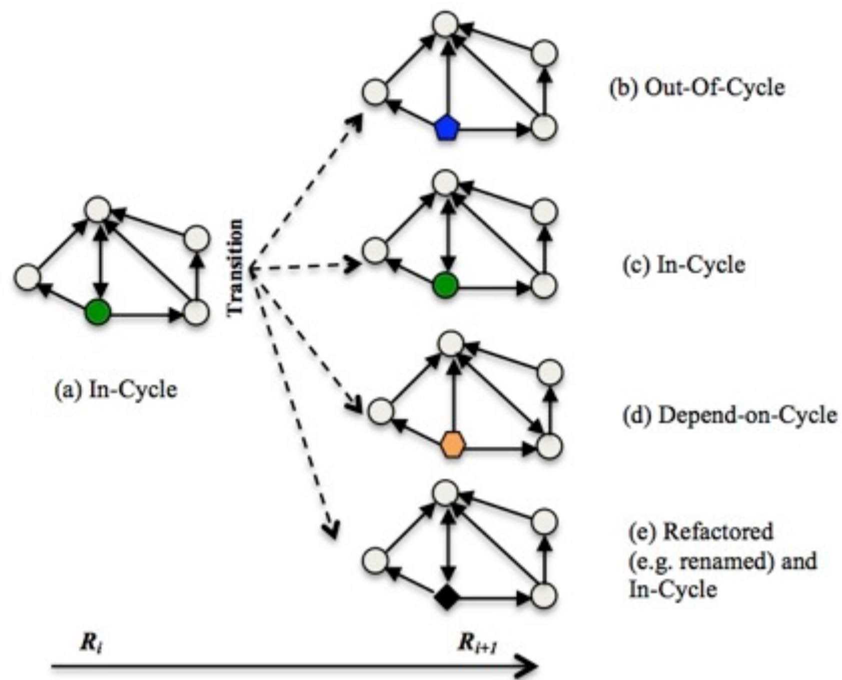


Figure 13 A simple example of transitions of in-cycle components between releases

3.6.3 S3: Investigating the effect that refactoring dependency cycles have on defects

The goal of **Study 3** was to empirically investigate whether refactoring components in dependency cycles could reduce their defect-proneness. **Study 3** addressed the research question **RQ1-3.** We divided the study into two sub studies. The first part studied the correlation between graph properties of a cycle (such as its diameter, density, vertex size and edge size) and the number of defect-prone components in each cycle graph. The second part studied the transition and evolution patterns of cyclically dependent components. We chose a case study methodology in order to answer the research questions in both studies.

In the first sub-study, we defined two-transition states as *in-cycle* and *out-of-cycle* and formed a Cartesian product between the two states as shown in Figure 13. The resulting four states made it possible to investigate whether:

1. Components in dependency cycles persist as defective in the "*in-cycle*" state more than components that persist in the *out-of-cycle* state
2. There evidence of cycle-breaking refactoring between releases
3. The transition of defective components from *in-cycle* to *out-of-cycle* reduce the defect-proneness of such components
4. The coupling or size complexity of components that transition as defective between *in-cycle* states increase at a significantly higher rate than those that transition between *out-of-cycle states*

We applied Spearman and Pearson correlation tests to determine the relationship between the cycle graph properties and defects in the first sub-study. For second sub-study, we applied proportion and t-tests.

3.6.4 S4: Improving the structural quality of cyclically dependent components using tools and metrics

The goal of **Study 4** was to improve the refactoring of classes in dependency cycles while reducing refactoring effort. **Study 4** addressed the research question **RQ2.** We implemented a metric (IRCRSS) to identify candidates for cycle breaking refactoring (see Section 3.3.3). This is computed by finding the reduction between the CRSS value of a candidate and the CRSS of its abstraction (interfaces or abstract classes). This study aimed to improve an existing metric in addition to developing a refactoring tool. Therefore, a design science methodology was selected. In addition, we used case study and interview methods to evaluate the metric and tool. We investigated the following:

1. Whether the system restructuring is better when IRCRSS reduction exists and it is used.
2. Whether tuning with IRCRSS metric would produce refactoring fitness that is better than candidates' selection without tuning with IRCRSS.
3. Whether tuning with IRCRSS metric always improve the software structure Is it a common property that tuning with IRCRSS results to better fitness in every system or not? We want to find out empirically whether many applications exhibit this opportunity.
4. Whether using IRCRSS metric would reduce the restructuring effort

We then evaluated the improved approach on fifteen applications and used statistical test to determine the significance of the improvements. Finally, we performed a qualitative evaluation of the tool and approach.

3.7 Scope, Concepts and Limitations

Type of systems: In this thesis, object-oriented systems have been used for the analyses. Specifically, the systems are written with either Java or C# programming languages as shown in Table 3. This can be a limitation regarding the application of the findings in other programming language domain.

Size measure: There are two common approaches to define size or length of a source code; (1) as the number of lines of code (LOC) or (2) as a measure of functionality in software (Fenton and Pfleeger, 1998). LOC is more commonly used and can be defined as (Fenton and Pfleeger, 1998); non-commented source statements (NCLOC) or a combination of NCLOC and comment lines (CLOC). The studies in this thesis have used non-commented lines of code (NCLOC) as the size measure. Critics of LOC, e.g. (Jones, 1985) have argued that it is technology-dependent and therefore difficult to use for comparing software across different programming languages. The systems we have investigated are written in object-oriented languages (Java and C#). Both languages share similar properties and therefore simplify the possibility to use LOC as a size measure for the systems.

Type and phase of defects: The analyses that have been performed in this thesis are based on post-release defects. These are defects reported during the operational period of the software and are different from pre-release defects that are captured during initial software development. The various types of defect, for instance, based on Orthogonal Defect Classification (ODC) (Chillarege et al., 1992), are not the focus of this study. Therefore, the defect metrics are used as described in Table 4. It will be an interesting future study to use the ODC for investigating defect-proneness in dependency cycles.

Type of analysis: The studies in this thesis are based on *static* coupling measurements and not *dynamic* coupling measurements (Arisholm et al., 2004). It is thus possible that actual coupling among classes at runtime are not completely captured. This imprecision can occur due to polymorphism, dynamic binding and dead code in the software. For instance, the use of reflection in Java cannot be detected during static analysis as the coupling occurs at runtime. However, static code analysis has been found to be practically useful and less expensive to collect (Basili et al., 1996; Briand et al., 1998; Chidamber and Kemerer, 1994; Zimmerman et al., 2011). Additionally, static coupling measures reflects to a very high degree the coupling among classes at runtime. Therefore, the imprecision cannot bias the results obtained in the studies.

4 Results

This chapter summarizes the results obtained in this thesis. The results are synthesized with the research questions, the papers containing the results and the main contribution from the research.

Overview of Results and Contributions

A summary of the studies, the research questions, its contributions and the papers for each study is given in Table 7. In **Study 2**, three sub-studies are reported. Similarly, in **Study 3**, two sub-studies are presented.

4.1 Empirical Investigation of different Defect Metrics to classify Critical Components

In **Study 1**, we evaluated the usefulness of several defect measures such as the *number of defects*, *defect density*, *defect correction effort*, and *severity of defect*, to identify defect-prone components that are critical to the system. The research question addressed in this study is:

RQ1-1: *What is the effect of using different defect metrics to identify critical software components? This is answered by paper **P1*** and led to contribution **C1-1**, i.e., "*Identification of the usefulness of multiple defect metrics to classify critical software components*".

The study aims to find out whether there are significant variations between the different defect measures to identify defect-prone components and architectural hotspots. We analysed the post-release data of an industrial Smart Grid application with a well-maintained defect tracking system. Using the Pareto principle, we identify and compare defect-prone and hotspots components based on four defect metrics. Furthermore, we validated the quantitative results against qualitative data from the developers**.** The results from the study show that at the top 25% of the measures: (1) significant variations exist between the defective components identified by the different defect metrics and that some of the components persist as defective across releases. (2) The top defective components based on number of defects could only identify about 40% of critical components in this system. (3) Other defect metrics identify about 30% additional critical components and (4) by considering the pairwise intersection of the

Table 7 Studies and their relation to research questions, methods, and contributions

Study	**S1**	**S2**	**S3**	**S4**
Research Questions	**RQ1**	**RQ1-2, RQ1-4**	**RQ1-3**	**RQ2**
Papers	**P1**	**P2, P3, and P6**	**P4, P5**	**P7**
Contributions	**C1**. Identified the usefulness of using different defect metrics to classify critical software components and the need to incorporate different defect metrics during defect analysis of software components	**C2**: (1) Observed a higher defect-proneness for components in dependency cycles and near cycles. (2) Observed a significant change impact for components in/near dependency cycles **C3**. Added new metrics to understand cycle neighbourhoods and to improve the refactoring of cyclically dependent components	**C2**: (1) Observed a strong correlation between the size of a cycle graph (nodes and edges) and defect-prone components. (2) Observed a higher defect-proneness for components that transition between dependency cycles than those that transition between out-of-cycles. Observed no systematic cycle-breaking refactoring between releases	**C3**: (1) Inclusion of new metric that improved the refactoring of cyclically dependent components and reduced refactoring efforts over previous approach. (2) Constructed and validated a cycle breaking decision support system for refactoring cyclically connected components
Area of contribution to software quality	Reliability	Reliability & Maintainability	Reliability & Maintainability	Maintainability
Research Methods	Case study & Survey (Quantitative & Qualitative)	Case study (Quantitative)	Case study (Quantitative)	Design Science, Case study & Survey

defect metrics, additional quality challenges of a component could be identified.

Critical defects spread across components in systems: Defect distribution in systems has been shown to follow the Pareto rule of 20-80 where a few components account for the majority of the defects in the systems. However, Figure 14 shows the percentage of defect-prone components with critical defects when ordered by defect counts. It shows that ***critical defects*** spread across all the components and using the 20-80 rule for prioritizing testing could be a major limitation as many defect-prone components with ***critical defects*** might be missed.

Critical components and defect-proneness: It is observed by analysing several defect metrics and grouping them against the rankings by developers that:

1. ***Critical components*** in a system are sometimes missed during testing when quality assurance (QA) is focused on a percentage of the components. The criticality of component to a system is determined by the role such component plays in the system and to other systems (e.g. a connector may be a critical component). Unfortunately, QA resources are limited; therefore, it is often impossible to thoroughly test every component in a system.
2. The two metrics; ***defect severity*** and ***defect correction effort*** identified additional components that were ranked **critical** by developers and these were neither identified by the ***defect count*** nor the ***defect density*** metrics.
3. Using multiple ***defect metrics*** can provide more insight about ***critical*** and ***defective components*** and guide the allocation of resources for testing. For example, by finding the pairwise intersections between the set of components identified by the different defect metrics, additional quality challenges of components are exposed. It will, however, require that developers exhibit a disciplined approach to recording defect data and that the development organization take advantage of the data for improvement purposes.

Largest vs. smallest first prioritization approach: Proponents of "largest-first prioritization" typically focus testing efforts to components with the highest number of defects. On the other hand, "smallest-first prioritization" proponents focus testing efforts to components with the highest defect density. Interestingly, findings showed both approaches to miss a significant number of defective components that are ranked critical by the developers.

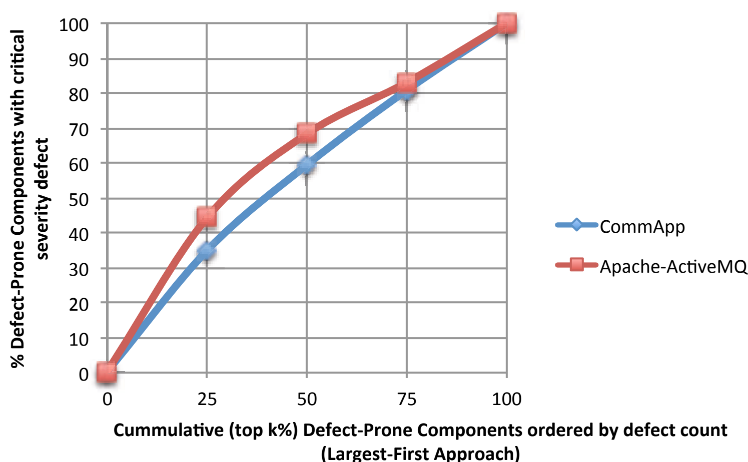


Figure 14 % of DPC with critical defects identified at the top k% of the class-files DPC over six releases (Oyetoyan et al., 2013a)

Contribution: The findings in **S1** exposed the usefulness and need to include different defect metrics when performing defect analysis of software components. Since quality assurance effort is limited, we have proposed including several ***defect metrics*** to identify significant number of ***critical components*** during defects analysis and to discover additional quality challenges of defective components.

4.2 Investigation of Defect and Change Proneness of Cyclically Dependent Components

In **Study 2**, we investigated the relationships between cyclically dependent components and (1) defects, and (2) change rates. The research questions addressed in Study 2 were:

RQ1-2: *What is the effect of dependency cycles on software defect?* The results are reported in papers ***P2*** *and* ***P3*** *and presented in sections* ***4.3.1*** *and* ***4.3.2*** *respectively.*

RQ1-4: *What is the effect of dependency cycles on change rate? The results are reported in* paper ***P6*** *and presented in Section* ***4.3.3****.*

Study 2 supports the main contributions stated in **C2** and partly **C3, i.e.,**

C2.1: *Identification of dependency cycles and neighborhood as defect hotspots in software systems*

C2.2: *Better understanding of the change impact of dependency cycles*

C3.1: *Added metrics to understand the complexity of components and improve the refactoring of cyclically dependent components*

4.2.1 Empirical evidence of dependency cycles as defect hotspots in software components

An investigation of the correlation between cycle components and defects was carried out. This is explored by using cycle metrics to mine and classify software components into two groups - the cyclic and the non-cyclic ones. Next, we have performed an empirical study of six software applications. Using standard statistical tests on four different hypotheses, we have determined the significance of the defect profiles of both groups.

The results show that:

1. Components in and near dependency cycles have higher likelihood of defect-proneness than those not in cyclic relationships.
2. The higher number of defective components is concentrated in components in and near dependency cycles.
3. Defective components in and near dependency cycles account for the clear majority of defects in the systems investigated.
4. The defect density of components in and near dependency cycles is sometimes higher than those in non-cyclic relationships.

Dependency cycles and defect counts: We have identified that a clear majority of defects and defective components are contained within the cyclically dependent components and their direct neighbourhood. This follows that the majority of the components that account for most of the defects are concentrated in dependency cycles. We found this to be interesting in the discussion of the "20-80" defect distribution rule in software systems (discussed in **Study 1**). It presupposes then that the majority of the "20%" components are in dependency cycles. Some cycles are also found to have large components and many incoming references (dependencies) in comparison to others. An example concerns utility components that are largely referenced because they provide

services to other components. In certain systems, they are found to be a major hub for large dependency cycles.

Dependency cycles and defect density: Components in dependency cycles and the immediate neighbourhood showed lower defect densities in some systems as against those not in cycles. We observed that components not in dependency cycles have smaller size and thus higher defect densities. This is interesting in relation to the previous discussions about "smallest-first prioritization". The findings in **S1** confirmed that ***defect density*** metric for "classes" could only identify "average" and "minor" components in the studied system, whereas, other metrics could identify largely, "critical" and "major" components of the system. It reinforced the fact that dependency cycles and its neighbourhood accounted for very complex components in the systems studied and should be focused for refactoring and testing.

4.2.2 Empirical evidence of the criticality of defects in cyclic dependent components

We investigate the criticality of defects in cyclically dependent components. Removing a large number of defects may have trivial effect on system reliability. The most number of latent defects lead to very rare failure in practice, while the vast majority of observed failures are caused by a relatively tiny number of defects. This shows that it is not the number of defects, rather their severity that matters. Thus, we are compelled to find out if this majority of defects and defect-prone components in cyclically related components are also the majority in both critical defects and severe defective components.

In the two applications that are empirically investigated, the main findings are that cyclically related components and their neighborhood components account for almost all of the critical defects and defect-prone components affected by these critical defects.

Dependency cycles and critical defect: The vast majority of critical defects that could result into a system failure and the components affected by the defects are concentrated in cycles and the immediate cycle neighbourhood. Most of the critical defects are associated with the system's reliability and are therefore very important. Defects mined in this category are for example; "Database *running at 100% CPU* and "Network *bridges can deadlock when memory limit exceeded"*. We found that approximately 50% of the in-cycle and neighbourhood components in the systems investigated accounted for almost all of the critical defects in the systems. It is thus more efficient to focus testing effort to about half of the components than all of the components.

Dependency cycles and neighbourhood components: We have proposed a metric termed "*depend-on-cycle*" to show the complexity of components that depend on other in-cycle components. The results we have obtained showed that these components share similar complexity as those components in cycles and are therefore of specific interest.

In summary, the case studies showed results that displayed a correlation between *in-cycle/near-cycle* components and defects. We thus proposed a refactoring of the components in dependency cycles and hypothesized that such refactoring can reduce the defect-proneness of the components "in" and "near" cycles over time.

4.2.3 Empirical study of dependency cycles and change rate

Two key issues are open with respect to the pervasiveness of cycles among software components. (1) It is probable that components in cycles incur maintenance penalty and (2) It could suggest that not all cycles are bad. Recent studies have proposed new heuristics and approaches to distinguish between "bad" and "harmless" cycles. In this study, we have investigated (1) whether classes in cycles are generally change-prone more than those not in cycles (2) whether cycles that have high diameters within their package containment tree are more change prone and (3) whether cycles that contain subtype knowledge in their structure are more change-prone.

We find that (1) the presence of cycles can have a significant impact on the change proneness of the classes near these cycles and (2) neither subtype knowledge nor the location of the cycle within the package containment tree are suitable criteria to distinguish between critical and harmless cycles.

Dependency cycles and change rate: We have found that dependency cycles have big impact on the change proneness of their direct neighbourhood. However, in the majority of the systems we investigated, components in dependency cycles undergo less change than those not in cycles. One explanation for this observation could be that certain utility components that are heavily referenced and turned back to become a major hub for big cycles. An example that fits this explanation is the case where abstraction is combined with the Singleton design pattern (Gamma et al. 1994). Components that are heavily reused have a high responsibility and therefore tend to be more stable. However, components around the cycles (i.e. *depend-on-cycle components*) are unstable. In relation to our previous studies that used only defect data, we found consistency with the impact of cycles on the neighbourhood components. However, there are differences with the results we obtained from the studies that investigated cycles and defects. Cycles tend to be more correlated to defects (corrective maintenance) in the majority of the applications than the remaining types of change (perfective, preventive and adaptive). Replicated studies would be useful to consolidate the findings in these studies.

Harmless and critical cycles: We attempted to distinguish between "critical" and "harmless" cycles by using certain cycle properties and heuristics. We found that certain cycles (e.g. cycles formed by the use of the Visitor pattern) may be stable. We call this inadvertent cycles formed as a result of limitations in technology. In the case of Visitor, this pattern is chosen to overcome the lack of support for multiple dispatch in Java (Muschevici et al., 2008). For other cycle properties investigated, such as those with subtype knowledge (STK) or high PCT-diameters, there was no difference in their change-proneness to the cycles without these properties. In particular, neither STK nor PCT-diameter is useful to classify "bad" or "harmless" cycles. We thus propose a study of a trade-offs between patterns and anti-patterns.

Contribution: One main contribution of **Study 2** was the identification of components in and near dependency cycles as hotspots for most defects and critical defects. Another contribution was the understanding of the impact of cycle neighbourhood on change rate. Lastly, there was better understanding about whether certain cycle properties could be used to classify harmless or critical cycles by considering their change rates.

4.3 Investigating the effect that Refactoring Dependency Cycles have on Defects

The results from **Study 2** indicate that components with cyclic relationships are responsible for the largest number and severity of defects and defect-prone components. Therefore, the goal in **Study 3** was to investigate the variables within cyclic dependency graphs that correlate with number of defect-prone components. Furthermore, we wanted to investigate whether there are systematic cycle-breaking refactoring between releases, and whether the transitions of components between dependency cycles to out-of-cycle states, have effect on their defect-proneness. The research question investigated in **Study 3** is:

RQ1-3: *What is the effect of refactoring cycles on defect-proneness? The results are reported in papers **P4** and **P5** and presented in sections 4.4.1 and 4.4.2 respectively. The study supports the main contribution **C2-1**: Identification of dependency cycles and neighbourhood as defect hotspots in software systems*

4.3.1 Empirical investigation of whether refactoring cyclic dependent components can reduce defect-proneness

We have examined the relationships between the size and distance measures of cyclic dependency graphs and defect-prone components. The goal was to determine variables within dependency cycle structures that could be the focus for refactoring activity.

Results demonstrated that the size of the cyclic graphs consistently correlates more with the defect-proneness of components in these systems than other measures. This implies that adding new components to and/or creating new dependencies within an existing cyclic dependency structures are stronger in increasing the likelihood of defect-proneness. Since causality is evaluated, at least initially with data that describe correlation, we could hypothesize that refactoring (breaking) cyclic dependencies can reduce defect-proneness of components.

Size of cycle and defective components: We observed that the size of a cycle correlate strongly with the number of defect-prone components. Increasing the number of components in cycles or forming new dependencies within cycles correlates with an increased number of defect-prone components.

4.3.2 Empirical investigation of defect patterns of components in dependency cycles during software evolution

We investigated the defect-proneness patterns of cyclically connected components vs. non- cyclic ones when they transition across software releases. In addition, we investigated whether cycle-breaking refactoring are performed between software releases and whether they have impact on the defect-proneness of affected components. The study also examined the coupling and size complexity of the components to determine their effect on the components defect-proneness during transition.

The study found that most movements of classes occurred in the same state. For instance, the transitions between releases are mostly from "*in-cycle*" to "*in-cycle*" or from "*out-of-cycle*" to "*out-of-cycle*". In other words, we found no evidence of any

systematic "cycle-breaking" refactoring between releases of the software systems. Furthermore, the results show that during software evolution, components that transitions between dependency cycles have higher probability to be defect-prone than those that transition outside of cycles. This case holds when the direct cycle neighbourhoods are not considered. In relation to defects, out of the three independent variables (LOC[5], CBO[6] and CRSS[7]) investigated, the study found that the CRSS metric tends to be more associated with classes that move between "*in-cycle*" states.

Evidence of cycle-breaking refactoring: Among the software systems (including the industrial Smart Grid system), we found no systematic cycle-breaking refactoring between software releases of these systems. The few components that transitioned outside of cycles in their next releases appeared to be an accidental transition. This may explain why dependency cycles are so pervasive among software components and across releases.

Transitions of components between cycles and defects: We found that components that transition between dependency cycles across releases persist as defective than those that transition outside of dependency cycles. This pattern occurred when we excluded the *depend-on-cycle* category. It can therefore be hypothesized that refactoring of components in cycles could reduce their defect-proneness.

Transitive coupling of cycles and defects: Among the three variables (LOC, fan-in and fan-out, and CRSS) that were investigated, we found that CRSS and LOC increased significantly for components that persist as defective across releases when they transition between cycles.

Refactoring of dependency cycles and defects: we could not directly answer the question of whether the refactoring of components in dependency cycles could reduce their defect proneness in future releases. This is due to the lack of evidence to support *cycle breaking refactoring* as discussed above. The evidence we have obtained is indirect and concerns the transitions of components between dependency cycles and their defect-proneness. To have a direct answer to this hypothesis, we have thus formulated an experiment and discussed it in paper **P4.**

Contribution: In **Study 3**, one of the major contributions was the identification of variables that could be focused for cycle-breaking refactoring. Another major contribution was a better understanding of defect-proneness of components when they transited from *"in-cycle"* to *"out-of-cycle"*. In addition, it provided more evidence of whether cycle-breaking activities occur between software releases.

4.4 Improving the Structural Quality of Cyclically Dependent Components using Tools and Metrics

In **Study 4**, we have implemented a cycle breaking decision support system and a new metric called IRCRSS. The IRCRSS identifies the reduction rate of class reachability set size (CRSS) from a class interface to reduce the number of components in dependency cycles. The research question investigated in Study 4 is:

[5] Lines of code
[6] Coupling between objects
[7] Class reachability set size

RQ2: *How to refactor dependency cycles to impact the structural quality and reduce the refactoring efforts? The results are presented in paper **P7** and it supports contribution **C3**: Tool and metrics to refactor defect- and change-prone hotspots in dependency cycle*

We have implemented a metric to identify the reduction ratio in the Class Reachability Set Size (CRSS) between a class and its interface. This is a new metric based on the CRSS metric by (Melton and Tempero, 2007a). The metric named "Interface-CRSS Reduction Rate (IRCRSS), in combination with an enhanced parameter selection method, aimed to reduce the number of classes in dependency cycles and the overall refactoring efforts. To evaluate the approach, we have constructed a cycle breaking decision support system that implements existing design approaches in combination with the class edge contextual data.

The evaluations on multiple systems show that (1) the improved CRSS metric could identify fewer classes as candidates for breaking large cycles, and reduce the refactoring efforts reasonably, and (2) the model could assist software engineers to plan the restructuring of classes in complex dependency cycles.

Added metric: We have identified the interface reduction ratio for the CRSS of a class and its interface (the IRCRSS metric). This metric is an improvement over the CRSS metric. By using this metric, we recorded a significant improvement in the "cycle-breaking" refactoring results and the overall refactoring efforts in some systems.

Cycle breaking decision support system: We have constructed a cycle breaking decision support system that could aid developers and software maintenance engineers to refactor class dependency cycles (see Figure 15 for the system's class diagram). The system integrates the improved metric to determine candidates for high impact refactoring. The system leverages the class contextual data (such as relationship and class type data) to provide practical and implementable actions for developers and maintenance engineers. The tool and approach have been evaluated using several applications to show that it could provide decision support for planning cycle-breaking refactoring activities at the class granularity level. The tool is useful to improve the structural quality of software systems that are riddled with defect-prone cycles.

The decision support system is publically accessible at: https://bitbucket.org/ootos/j-guirestructurer and https://bitbucket.org/ootos/c-sharprestructurer

The C# version, developed by an MSc student, has integrated the refactoring module and algorithms from the decision support system written in Java.

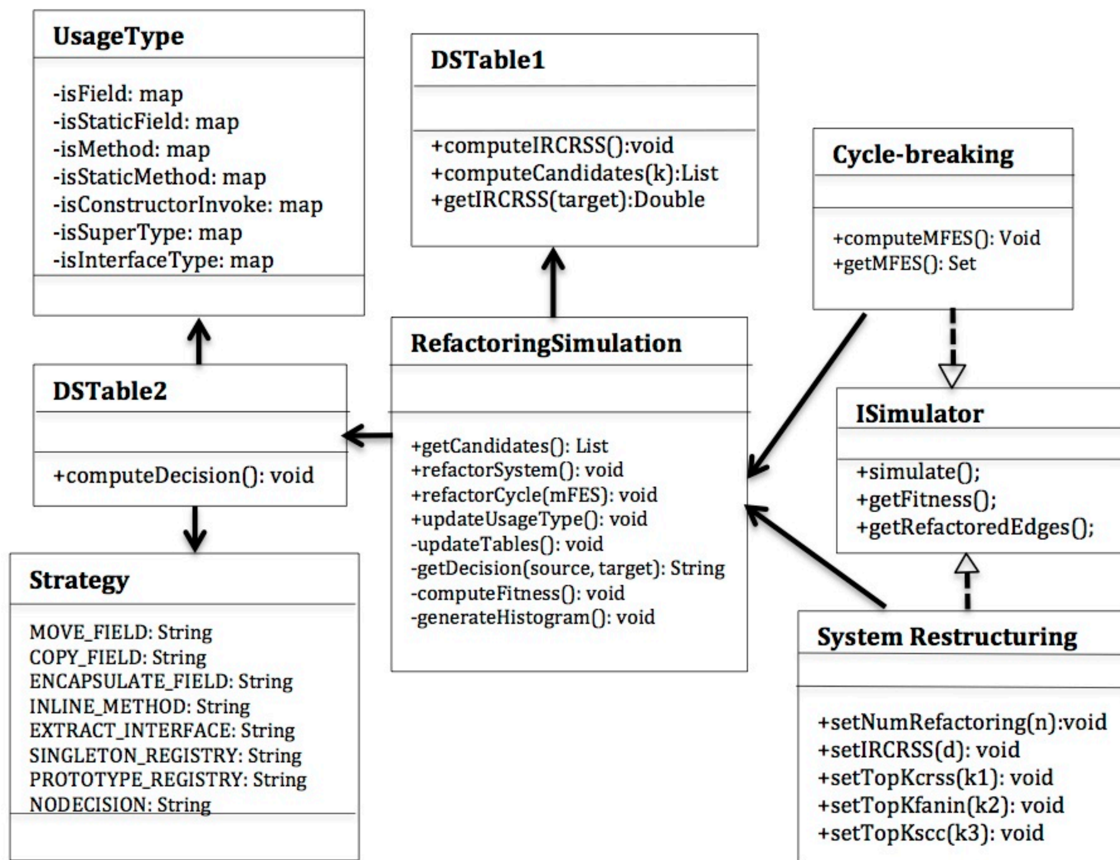


Figure 15 Class Model for the Cycle breaking decision support system (Oyetoyan et al., 2015a)

Contribution: The main contribution of this study was to (1) improve the structure of software systems with dependency cycles by constructing a cycle breaking decision support system and (2) reduce refactoring efforts during this process.

5 Evaluation and Discussion

This thesis has investigated dependency cycles among software artefacts with respect to maintainability and indirectly to reliability. For maintainability, we have used change and defect metrics and for reliability, we have used defect severity as an indirect metric. We have performed several empirical studies to investigate the relationship between (1) cycles and defects and (2) cycles and change rate in general. In response to the results from the empirical studies, a decision support system to refactor class cycles has been developed and a new metric proposed to improve the structural quality of software systems.

The thesis mainly contributes to improvement in the areas of **software quality** and **software metrics**. In the following, we discuss the contributions under each topic. The main contributions and the sub-contributions of this thesis are:

C1. Better understanding of how to utilize different defect metrics to improve software quality
- **C1-1:** Identification of the usefulness of different defect metrics to classify critical software components

C2. Identification of the impact of dependency cycles on software quality
- **C2-1:** Identification of dependency cycles and neighbourhood as defect hotspots in software systems
- **C2-2:** Better understanding of the change impact of dependency cycles

C3. Tool and metrics to refactor defect- and change-prone hotspots in dependency cycle
- **C3-1:** Added metrics to understand the complexity of components and improve the refactoring of cyclically dependent components
- **C3-2:** A cycle breaking decision support system to refactor cyclically connected components

The rest of this chapter is organized as follows: Section 5.1 provides the overall view of the thesis contributions in relation to the software engineering topic. Section 5.2 evaluates the contributions of this thesis. Section 5.3 discusses the contribution against the state of the art. In Section 5.4, we provide overall recommendations. Section 5.5

briefly evaluates the validity threats to the study while Section 5.6 provides a reflection on the research context.

5.1 Overview of Thesis Contributions

Figure 16 shows how the studies and the contributions are related to software quality attributes. The summary is listed in Table 8. The effects of the different studies on specific quality attributes are discussed.

Reliability: The contribution of this thesis affects the reliability of software systems indirectly. We have identified critical defect hotspots (Contributions **C1-1** and **C2-1**). Critical defects as discussed, impact on the reliability of software. By improving the structural quality in such hotspots through refactoring (Contribution **C3**), the overall reliability of the system can be improved.

Modifiability: Dependency cycles are known to be detrimental to modifiability (Bass et al., 2003). Refactoring them is important to facilitate ease of change during software maintenance and evolution. The extent of the impact of dependency cycles is shown by the structural complexity of components that depend on them (in-neighbours). What is particularly useful here is that software evolution data can be used to identify specific cycles that could be recommended for refactoring (Contribution **C2**). Refactoring such defect and change-prone cycles can reduce ripple effect of change (Contribution **C3**).

Reusability: During software development, code reuse is a standard practice and it can occur at different granularity levels. We are concerned with code reuse at the class granularity level and among internally declared types (see Section 1.1). A component that reuses another component in a dependency cycle has to depend on all the components contained in the cycle including the in-neighbours of the cycle. It has been shown that usage relationships among components can be used to predict their defect-proneness (Oyetoyan et al., 2012; Schroeter et al., 2006). Breaking defect-prone (Contribution **C3**) and change-prone cycles (Contribution **C2**) can reduce the probability of defect-proneness since unnecessary code would not be copied during reuse.

Testability: Dependency cycles are known to be detrimental to testability. Components in cycles are expensive to test, as it is impossible to test them in isolation. As (Lakos, 1996) states, "Testing a component in isolation is an effective way to ensure reliability". Effective testing would require components to be decomposed to become stand-alone and of manageable size. The acyclic dependency principle (Martin, 1996) is important to decompose components to become stand-alone and be of manageable size. It is advisable and necessary to refactor (Contribution **C3**) defect- and change-prone locations (Contributions **C1** and **C2**) in dependency cycles to make the different components testable. In addition, selecting test sample that is adequate and representative enough is a challenge (Bertolino, 2007). Contributions **C1** and **C2** point to aspects of the software where test samples can be drawn to make it more efficient.

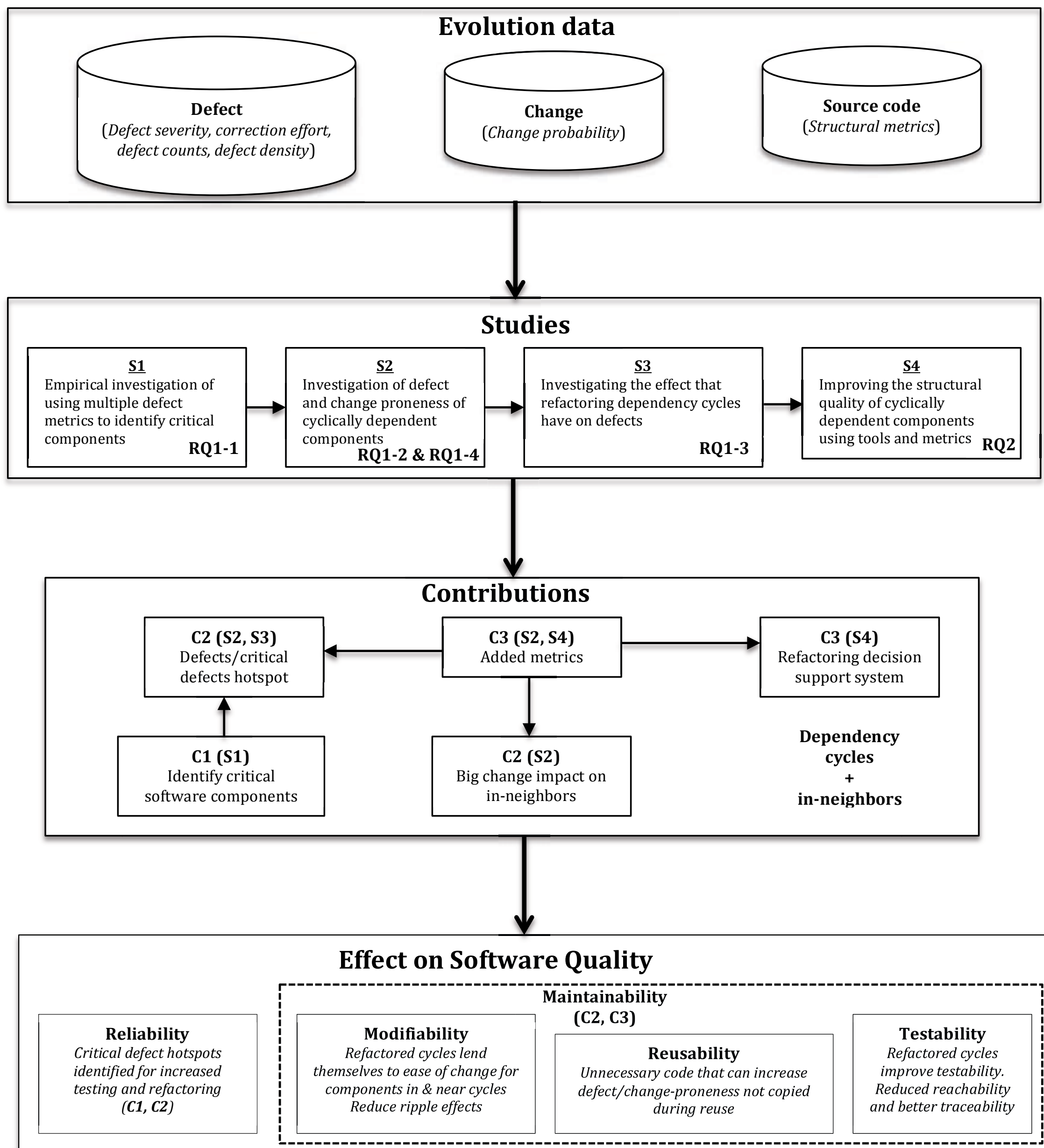


Figure 16 Overall relationships between the studies and contributions to software engineering field

Table 8 Connection between contributions, research questions, papers, and specific software quality attributes

Contributions	Description	Study	Research Questions	Papers	Effect on software quality
C1	Better understanding of how to utilize different defect metrics to improve software quality	S1	RQ1	P1	Reliability
C2	Identification of the impact of dependency cycles on software quality	S2, S3	RQ1-2, RQ1-3, RQ1-4	P2, P3, P4, P5, P6	Reliability & Maintainability
C3	Tool and metrics to refactor defect- and change-prone hotspots in dependency cycle	S2, S4	RQ1-2, RQ1-4, RQ2	P2, P3, P6, P7	Maintainability

5.2 Evaluation of the Contributions against the Research Goal

The main goal of the IME research initiative for Smart Grid within software engineering is: ***Improved Management of Software Evolution for Smart Grid Applications***. We discuss in the following how the contributions in this thesis are related to the research goal.

C1: Identification of the usefulness of different defect metrics to identify critical software components

The analysis of the defect and source code repositories of an industrial Smart Grid system resulted into contribution **C1**. Software evolution is a constant phenomenon in the software lifecycle. As software evolves, so does its complexity (Lehman, 1980). Software complexity has been correlated with defects. The contribution shows that it is possible to identify significant number of software components that are critical to a system when multiple defect metrics exist and are used. This contrast with using popular defect metrics, i.e. only defect counts and defect density. Such critical components can then be focused for improved quality assurance. The higher the test coverage that could be achieved for key software components during development and maintenance, the lower the risk for defects and system failure. The contribution can be considered to be significant for critical applications where sufficient test coverage is vital, but this remains a goal that equally competes with other goals for scarce quality assurance resources.

C2-1: Identification of dependency cycles and neighbourhood as defect hotspots

Contribution **C2** showed a subset of the software components (i.e. dependency cycle and neighbourhood components) that should be focused for increased testing and refactoring. From empirical study of Smart Grid and other open source systems, we showed that dependency cycles and their neighbourhood are the major hub for defects and even critical defects that could result into a system failure. A system failure is critical in a system of systems and could have a cascading effect. This contribution improved the understanding of defect location in software systems. This knowledge is useful and provides new opportunities to identify refactorable locations in the system.

C2-2: Better understanding of the change impact of dependency cycles

We found that dependency cycles have significant impact on the direct neighbourhoods. These are the components that depend on cycles. Additionally, we discussed the stability of Visitor pattern as interesting because it also contains anti-patterns. The usefulness of this result affects the design of software systems. Understanding “critical” and “harmless” cycles is important knowledge that could guide refactoring effort. It is more useful to refactor candidate cycles that are harmful as such refactoring would translate to improving the external quality of the system.

C3-1: Added metrics to improve the refactoring of cyclically dependent components

Metrics provide measurement that could be used for evaluating the quality of a system. In contribution **C3-1**, the two metrics that are proposed provide measurement that (1) increase the knowledge of defect location and component complexity in relation to dependency cycles (*depend-on-cycle*) and (2) improve the refactoring of components in dependency cycles in terms of refactoring effort and reduction of cycles (*IRCRSS*). Measurements of the software structural quality are important during evolution. We need to use the measurements as a feedback to assess the direction of the structural evolution of the system and the opportunity for restructuring and refactoring.

C3-2: A cycle breaking decision support system for refactoring cyclically connected components

Architectural erosion is common during software evolution. To mitigate the effect of structural decay as the software evolves, there is a need for “agile” refactoring practices to monitor and regularly improve the structure of the system. Without adequate tool support, this is a challenging and difficult task for developers and maintenance engineers. Dependency cycles are signs of structural decay in many systems. A cycle breaking decision support system has been proposed and developed that can be used to discover and thus improve the structural quality of the system. The tool monitors the structure of a system and identifies undesirable cycles in the system that can be refactored. This tool has been implemented in a commercial Smart Grid company to assist the developers and maintenance engineers in their maintenance tasks. There is currently little advice about how to refactor dependency cycles at the class granularity level. We believe this contribution is filling an important gap in the area of software evolution, maintenance, and refactoring. This is strongly connected to the research goal as the tool and approach would be useful to improve the structural quality of the Smart Grid systems (and other systems) during their evolution. Such improvement would result in a better testable and maintainable system during their lifecycle.

5.3 Discussion of Contributions related to the State-of-the-Art

This section discusses how each of the contributions is related to the state-of-the-art in software engineering research and practice.

Contribution C1: Several studies have shown that the defect distribution in software systems follows the Pareto rule of 20-80 (Andersson and Runeson, 2007; Basili and Perricone, 1984; Boehm and Basili, 2001; Ebert et al., 2005; Fenton and Ohlsson, 2000; Ostrand and Weyuker, 2002; Tihana Galinac et al., 2012). Among these studies only

Ostrand and Weyuker (2002) reported the distribution of defective components based on the severity of their defects. The contribution **C1** is novel because (1) it uses multiple defect dimensions (2) it identifies gaps and synergy between the defect metrics and (3) it investigates the effect and coverage the defect metrics have on critical software components. To the best of our knowledge, we have found no reported studies with this contribution.

Contribution C2-1: The study by Zimmermann and Nagappan (2007) has observed a similar pattern in one of several hypotheses we explored in this thesis. Precisely, hypothesis P2.H_{A1}: *The number of defective components in cyclic relationships is significantly higher than non-cyclic defective components (in Paper **P2**)*. Our contribution **C2**, however, provides an in-depth study of the topic "dependency cycles versus defects" by exploring several defect metrics and a number of case studies. For instance, we have found no study that reports on dependency cycles and the criticality of defects.

Contribution C2-2: In relation to dependency cycles versus change, we have found no studies that have explored this relationship.

Contribution C3-1: In relation to the topic of dependency cycles, it appears that only our studies have discussed "*depend-on-cycle*" metric. However, this concept and metric has been used in other contexts in previous studies e.g. (Zimmermann and Nagappan, 2008) and it is popular in network analysis (Wasserman and Faust, 1994). The second metric "*IRCRSS*" that we introduced is a novel contribution. It is a derivation of the metric CRSS by (Melton and Tempero, 2007a). The CRSS is synonymous to the cumulative component dependency[8] (CCD) and its variants by Lakos (1996). In comparison to object-oriented metrics, the CRSS metric is related to the Response For Class (RFC) metric. We can assume that there is a linear relationship between CRSS and RFC metrics. The more classes a class can reach, the more its tendency to have a high RFC coupling since RFC considers the transitive method coupling closure of a class.

Contribution C3-2: There are studies and approaches devoted to breaking dependency cycles albeit at the package granularity level (Dietrich et al., 2012; Falleri et al., 2011; Laval et al., 2009; Laval and Ducasse, 2014). Existing tools such as; JDepend (http://clarkware.com/software/JDepend.html), NDepend (http://www.ndepend.com), Dependometer (http://source.valtech.com/display/dpm/Dependometer), PASTA (Hautus, 2002), Classycle (http://classycle.sourceforge.net), CARE (Shah et al., 2013), STAN (http://stan4j.com/), LDM (Sangal et al., 2005), have also primarily focused on refactoring cycles at the package level. Apart from JooJ by (Melton and Tempero, 2007c), there is currently little focus at the class granularity level. Our contribution is focused on refactoring cycles at the class granularity level by providing implementable actions and opportunities for assessing the structure of the code.

5.4 Recommendations to Practitioners

The results we have obtained in this study are interesting and useful for improving the structural quality of object-oriented software applications. In this section, various

[8] CCD of a subsystem is the sum of the components' dependencies for each component in this subsystem

recommendations for stakeholders of software projects are suggested. These recommendations are drawn from the results that we have obtained in all the studies presented in this thesis. Some of the recommendations are conventional wisdoms that are reinforced from the empirical results obtained in this thesis; however, some are new knowledge that would require some follow-up studies.

Make measurement more reliable

In our experience with mining defects data from repositories, we found that in some cases there are missing data points. An example is reporting defects in defect tracking systems and logging the defects fixed against the actual source files that are modified or created in the configuration management system. The reliability of the analysis results depends on the quality of the data. We recommend that defect data be accurately logged and consistently associated in the configuration management system to allow for quality measurement, analysis and feedback.

Make measurement of work products and design

Measurements provide the means for assessment and improvement. Our experience shows that the structural quality of work products is not usually measured. Some of the findings from the Smart Grid application of our industrial partner were surprising. It thus indicates that the data logged have not been used optimally for quality improvement programs. It is hard to know the structural quality and the extent of erosion unless a conscious measurement is made. We recommend that structural analysis of software be performed to determine refactoring possibilities/opportunities.

Incorporate refactoring practices

Architectural refactoring practices are many times overlooked in the software development process. The results we have obtained and interactions with the industrial partner confirmed this statement. The class cycles in the systems grow during evolution and we did not detect any major cycle refactoring practices. This can be due to the unavailability of tools that are practical and usable for such level of refactoring. Some of the tools available on the development environments are not for major code restructuring. During an evaluation phase performed with the industrial partner, it was obvious that the developers only perform low-level code refactoring. This is consistent with the results of the study in (Murphy-Hill et al., 2009). Another reason is that such refactoring is not prioritized during the software development process.

A “functional” code does not mean that all is well. Erosion of code structure incurs change penalty and can drive maintenance cost to a much higher level. After all, this may not be surprising because software maintenance cost is claimed to be the highest in the software life cycle (van Vliet, 2000). We recommend a top-driven refactoring practice to be introduced. Why top-driven? Most times, managers are driven by time-to-market at the expense of code quality (e.g. maintainability). Such attitude has impact on the developers who only work to produce “functional” code as against maintainable code. Theoretically, this is called a **technical debt** (Brown et al., 2010). Managers need to realize the need to avoid architectural erosion as the system evolves.

Take advantage of tools to avoid architectural erosion

To achieve an "agile" refactoring practice, there is a need for tool support. Tools make the possibility of continuous refactoring a reality. Without tool support, it is not feasible for the work product and code structure to be easily controlled in terms of measurement, feedback, and improvement. We have in this study implemented a useful refactoring tool that could be leveraged by developers and maintenance engineers to monitor and restructure their existing code structure.

Identify "critical" and "harmless" cycles

Applications vary in many ways and generalization of results across systems may not be plausible. Organizations need to perform their own analysis to identify what roles dependency cycles play in their systems. We have found in certain systems that dependency cycles and their neighbourhood components do not incur change more than other components. Some systems have used patterns that have a cycle property, for example, the case of Visitor pattern. One of our findings showed the Visitor pattern to have less change than other types of cycles. There are also instances where cycles are formed because of references to utility components. Utility components usually have high in-degree (incoming connections). As Wake puts it, not all code smells are indicative of problems but they are certainly worthy of a look and decision (Wake, 2004). Individual evaluation would thus be necessary to identify what is critical and harmless.

5.5 Discussion of Validity Threats

This section evaluates the threats to the validity of the studies we have performed. The threats to the validity of the studies are discussed in details in each paper (see Appendix-A).

Conclusion validity: This is about "right analysis". This concerns the ability to draw correct conclusions from the relationships between the treatment and the outcome (Wohlin et al., 2003). Issues such as effect size or statistical power in data analysis are important test that should be performed to increase confidence in the conclusion.

We believe we have used the right statistical analysis during our studies. One common challenge with defect data is that it is skewed; as such it is usually not normally distributed. It would therefore be inappropriate to use standard t-test when the data has not been tested for normality. In our studies, we have performed normality test to determine whether to use parametric (e.g. standard t-test) or non-parametric test (e.g. Wilcoxon ranked sum test). We have also considered the measurement types (interval, ordinal, nominal or ratio) of the variables to determine the appropriate analysis technique. We have also performed effect-size test to judge whether the conclusions drawn from the hypotheses testing are meaningful or not.

Internal validity: This concerns "right data". If the outcome is caused by the treatment and not by other factors not measured, we can conclude that the result has internal validity (Robson, 2011). It focuses mainly on the study design and whether the results follow from the data (Easterbrook et al., 2008).

The defect data we have used in our studies have been mined from defect tracking repositories. A common threat is missing data points that occurred when not all defects

are logged in the repository. This threat is minimal in our studies as we were able to have good sample size for our analysis. Another challenge is the tagging of defect ids in the commit log of changed source files in the configuration management systems. In some cases, a source file that is changed because of defect fixes may not be associated with the defect ids in the commit messages. There are also cases of erroneous tagging. All these are known threats in these kinds of studies. In our studies, the mapped data we have mined is large enough for statistical analysis. In the case of source code data, our tool could not identify weak relationships caused by the use of reflection. This is a common challenge for static analysis tools.

Construct validity: This is about “right metrics”. The main question asked here is whether the metrics measure what we think it measures (Robson, 2011). Are concepts clearly defined including interactions of different treatments in a way that right measurements can be taken? (Wohlin et al., 2003). We have addressed the threats to construct validity by identifying balanced hypotheses and research questions. We have considered different defect dimensions and used well-established metrics from the research literature.

External validity: This concerns “right context”. Can the results be generalized outside the scope of the study? Issues such as selection (sampling), context of study, constructs that are specific to the study are threats to generalization and should be considered (Robson, 2011). This thesis has used thirty applications with different functionality, domains, age, size, context, programming language and usage. Nevertheless, we cannot generalize the results to all software systems. In addition, only one of the systems was a commercial application. We therefore cannot conclude that the results from the studies of the commercial application reflect a general trend. The diversity between commercial application and open source applications (e.g. development practices) also shows that we cannot generalize across the systems. More studies will be necessary to compare with the results we have obtained in our studies.

5.6 Reflections on the Research Context

The research context is industry oriented and focused on Smart Grid software and its evolution. As mentioned previously in Section 1.2, we established cooperation with a relevant and large software company that develops applications for the power grid. This has enabled us to pursue the studies in this thesis within the research context. We were lucky to have a good partnership with the industrial partner. The company has provided all support needed at the different stages of the research work. As such, our work did not suffer from the challenges that are common when conducting an empirical study with industrial companies (see Section 1.3).

We have conducted several presentations to provide feedback of our findings to the industrial partner. The results of the several empirical studies have triggered the implementation of a new tool. A decision support tool was developed to assist the developers and maintenance engineers to perform continuous refactoring of their code structure. We have also evaluated our proposed refactoring approach and tool using the industrial application as one of the major case studies.

A common threat to deal with is a follow-up of the results and the applications of the tool and approach that we have proposed and developed. We have had one maintenance

engineer assigned to us for the whole duration of the tool evaluation and refactoring. This lasted for about 3 weeks (40 man-hours). However, we believe the knowledge should be disseminated to other developers on the project and in the company. In this way, the company can take full advantage of the results and tool that resulted from this thesis.

In conclusion, the studies in this thesis have contributed to the improvement of the quality of Smart Grid systems during evolution. The results are not only relevant to the Smart Grid systems we have studied; they are also relevant to other open source systems that share similar context and domains as those we have used in this thesis.

6 Conclusion and Future Work

This thesis presents results of several empirical studies that have investigated defect distribution and change rate of software components in dependency cycles and near dependency cycles. A new metric has been introduced and investigated which can improve refactoring of undesirable dependency cycles and reduce refactoring efforts. Case studies have been drawn from an industrial Smart Grid software company and several open source projects. The thesis presents valuable insights into how defect metrics can be utilized to identify important and critical software components. Furthermore, the research has led to increased knowledge about defect locations in relation to components in cycles and those that reference cycles. Lastly, a cycle breaking decision support tool has been proposed, implemented, and evaluated, which can assist software developers and maintenance engineers to improve the structural quality of their software artefacts and code.

This chapter sums up the findings and contributions of this work and outlines directions for future studies.

6.1 Overall Summary of Findings

We summarize the four key findings in this thesis as follows:

Multiple defects and critical components: In the Smart Grid application that was investigated, we found that *defect counts* and *defect density* are not sufficient to identify and classify critical components in the system. Indeed, other metrics such as *defect severity* and *defect correction effort* classified a significant number of key components that developers considered critical to this application. Since quality assurance resources are limited, we propose to consider multiple metrics to identify critical components that should be focused for increased testing. As a critical infrastructure, it is essential to limit the risk of failure by focusing the testing efforts in the right direction.

Observed patterns with respect to defects and dependency cycles: The results of several empirical studies in our research confirm that defects and critical defects are concentrated within dependency cycles and in components that have reference to cycles

(we call them *in-neighbours*). We found that in some systems about 50% of the components are in cycles and near cycles and these account for a clear majority of the defects, defective components, and critical defects. The findings provide important information about the subsets of the systems that should be focused for increased testing and possible refactoring.

Observed patterns with respect to change and dependency cycles: Dependency cycles do have significant change impact on their neighbourhood (*in-neighbours*). However, the majority of the components in cycles have a lower change rate than those outside cycles. We attempt to distinguish between "*critical*" and "*harmless*" cycles by using some cycle properties. Our findings show that: (1) No significant difference exist between the change rate of cycles with subtype knowledge (STK) and those without (2) no correlation exists between change rate and high PCT-diameter and (3) the cycles with the Visitor pattern are stable within the application where they exist. Overall, the results are interesting and open up possibilities for further studies. We discuss this in Section 6.2

Opportunities for refactoring class cycles: The focus here are two areas; metrics and tools. First, we propose an improved metric named IRCRSS derived from the CRSS metric. By using this metric, it is possible to improve cycle-breaking refactoring and reduce refactoring efforts. Second, we have implemented a cycle breaking decision support system to improve the structural quality of software systems. The tool has integrated the improved metric and other enhanced selection parameters. It then makes propositions that developers or maintenance engineers can manually implement. There is little attention given to refactoring of class cycles. We therefore believe that our tool and approach are filling an important gap.

Impact on software quality: The results reported in this thesis contribute to *maintainability* and indirectly to *reliability* of software systems. We have identified that dependency cycles and their immediate neighbourhoods are hotspots for critical defect in software applications. We show the need to apply multiple defect metrics to identify critical component that should be specifically in focus for more thorough testing efforts. We have developed a cycle breaking decision support system that could assist developers to improve the structural quality of their software. These contributions affect testability, reusability, modifiability, and reliability of the systems.

6.2 Directions for Future Work

Based on the results from this thesis, we outline different proposals for future study as follows:

Distinguishing between "critical" and "harmless" cycles: More work is needed to identify cycle metrics and properties that can help to distinguish between a "critical" and a "harmless" cycle. Some cycles are stable and not defect-prone while some undergo high change and are defect-prone. Some cycles are formed by certain design patterns, for example the case of Visitor discussed earlier. In some cases, these kinds of patterns are code automatically generated by parser tools (e.g. ANTLR). We believe that context information would be necessary during cycle classification. If we are able to classify cycles into "critical" and "harmless" categories, we could save refactoring efforts by focusing on the critical ones.

A study of trade-off between patterns and anti-patterns: Cycles are pervasive in real-life software systems and certain design patterns are also known to contain this anti-pattern, which is traditionally known as an indicator of bad design (e.g. Visitor pattern). It presupposes that certain cycles may be justified because of the role that the involved artefacts play in the system. The question remains as to what extent a pattern should harbour an anti-pattern. Should a pattern be refactored? For instance the Acyclic Visitor proposed by Martin (1997). It is more complex than the Visitor pattern itself and seems not to be widely used. Are patterns with anti-patterns penalized by changes/defects more than those without anti-patterns? An extensive study of the trade-off would be an interesting topic and a step to provide better design advice for future systems.

Utilization of abstraction: Interfaces need to be better utilized during development to achieve loose coupling. Many times, cycles are formed because the Dependency Inversion Principle (DIP) is not applied and existing studies already showed that even when the abstraction exist, they are under-utilized (Gobner et al., 2004; Mayer, 2003; Steimann et al., 2003). One way to optimize interface utilization for the purpose of decoupling is to create a plugin in the development environment (e.g. Eclipse) that can provide a real-time feedback of the IRCRSS value of a dependent component. This solution can assist developers to implement dependency inversion with interface when they are reusing (importing) an exisiting class implementation. The idea is to base this feedback on the normalized CRSS values of the imported class and a real-time feedback of IRCRSS value when an interface is used instead. In this way, a developer can quickly decide for an abstraction of this class type rather than its implementation.

Automation of propositions from the cycle-breaking decision support system: One aspect that would be beneficial for practitioners and researchers is to automate the manual propositions from the decision support system on the graph model to the actual source code. This is possible as such automated implementation can be applied on the abstract syntax tree (AST) model in many development environment (e.g. Eclipse). One way to achieve this is to create a plugin that passes the class binaries from an IDE as input to the CB-DSS, while the output (proposed actions) from CB-DSS is fed back to the plugin. The actions can now be applied on the AST model with an appropriate pre- and post-condition checks.

Study of dependency cycles in Service-oriented systems: Applications based on service oriented architecture (SOA) are ubiqutous. Development *for* reuse (Sindre et al., 1995) has in particular made organizations to transform software components into services that can be discovered for reuse. It will be interesting to study dependencies between services that are composed for reuse. Whether dependency cycles occur at this layer both from static and dynamic dependency viewpoints, would be useful to understand the maintenance cost.

Glossary

Anti-Pattern: Poor design choices and can exist at the code, design and architectural levels

Anomaly: Anything observed in the documentation or operation of software that deviates from expectations based on previously verified software products or reference documents (IEEE Std 1012-1986)

Component: A component may refer to a class, a package or a jar file

Defect: Imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced (IEEE Std 1044-2009)

Defect-density: The number of defects per unit of product size (ISO/IEC/IEEE 24765:2010)

Dependability: Trustworthiness of a computer system such that reliance can be justifiably placed on the service it delivers (IEEE Std 982.1-2005)

Dependency: A relationship that defines that a component needs another component to function

Dependency Cycle: A cyclic dependency graph also known as strongly connected components (SCC) in a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, both are reachable from each other (Cormen et al. 2001)

Design Pattern: Patterns are recurring solutions to design problems. They capture existing, well-proven designs

Fault: Manifestation of an error in software (ISO/IEC/IEEE 24765:2010)

Defect Tracking System: A system for recording a reported defect and for capturing the attributes of the defect such as its severity, date reported, priority and so on

Error: The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition (ISO/IEC/IEEE 24765:2010)

External measure of Software Quality: measure of the degree to which a software product enables the behaviour of a system under specified conditions to satisfy stated and implied needs for the system (ISO/IEC 25010:2011)

Failure: The inability of a system or system component to perform a required function within specified limits

Fan-in: The number of components that reference a component (in-coming connections to a component)

Fan-out: The number of components that is referenced by a component (out-going connections from a component)

In-Neighbours: Component that is not in dependency cycle but references component(s) in dependency cycles**.**

Internal measure of Software Quality: the measure of the degree to which a set of static attributes of a software product satisfies stated and implied needs for the software product to be used under specified conditions (ISO/IEC 25010:2011)

Maintainability: The ease with which a software system or component can be modified to change or add capabilities, correct faults or defects, improve performance or other attributes, or adapt to a changed environment (ISO/IEC/IEEE 24765:2010)

Measurement: The process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules (Fenton and Pfleeger, 1998)

Reliability: The ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE Std 610.12-1990)

Smart Grid: A modernized grid that enables bidirectional flows of energy and uses two-way communication and control capabilities that will lead to an array of new functionalities and applications (NIST, 2010)

Software Engineering: (1) The systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software (2) the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software (ISO/IEC/IEEE 24765:2010)

Software Quality: (1) capability of a software product to satisfy stated and implied needs when used under specified conditions (2) degree to which a software product satisfies stated and implied needs when used under specified conditions” (ISO/IEC/IEEE)

Software Maintenance: the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment (IEEE 610.12)

Software Evolution: The stage in a software system’s life cycle where it is in operational use and is evolving as new requirements are proposed and implemented in the system

Software Metric: A *software metric* is a measurable property, which indicates the software quality criteria to be measured (Gillies, 1997)

Subversion system: A version control system for managing files and directories, and the changes made to them over time (SVNBOOK: www.svnbook.red-bean.com)

System: Combination of interacting elements organized to achieve one or more stated purposes (ISO/IEC 25000:2014)

References

Abreu, F.B.E., Melo, W., 1996. *Evaluating the impact of Object-Oriented design on software quality*. Proceedings of the 3rd International Software Metrics Symposium, 90-99.

Adams, E.N., 1984. *Optimizing Preventive Service of Software Products*. Ibm J Res Dev 28, 2-14.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S., 1977. *A Pattern Language*: Oxford University Press. New York.

Andersson, C., Runeson, P., 2007. *A replicated quantitative analysis of fault distributions in complex software systems*. IEEE T Software Eng 33, 273-286.

Arisholm, E., Briand, L.C., Foyen, A., 2004. *Dynamic coupling measurement for object-oriented software*. IEEE T Software Eng 30, 491-506.

Basili, V.R., Briand, L.C., Melo, W.L., 1996. *A validation of object-oriented design metrics as quality indicators*. IEEE T Software Eng 22, 751-761.

Basili, V.R., Perricone, B.T., 1984. *Software Errors and Complexity - an Empirical-Investigation*. Commun Acm 27, 42-52.

Bass, L., Clements, P., Kazman, R., 2003. *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc.

Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., Strollo, O., 2012. *When Does a Refactoring Induce Bugs? An Empirical Study*, IEEE 12th International Working Conference onSource Code Analysis and Manipulation (SCAM), 2012, pp. 104-113.

Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L., 1998. *Clone detection using abstract syntax trees, Software Maintenance*, 1998. Proceedings., International Conference on. IEEE, pp. 368-377.

Beck, K., Fowler, M., 1999. *Bad smells in code, Refactoring: Improving the design of existing code*. Addison-Wesley, pp. 75-88.

Bertolino, A., 2007. *Software testing research: Achievements, challenges, dreams*, 2007 Future of Software Engineering. IEEE Computer Society, pp. 85-103.

Bertolino, A., Calabró, A., Di Giandomenico, F., Nostro, N., 2011. *Dependability and Performance Assessment of Dynamic CONNECTed Systems*, in: Bernardo, M., Issarny, V. (Eds.), Formal Methods for Eternal Networked Software Systems. Springer Berlin Heidelberg, pp. 350-392.

Bieman, J.M., Straw, G., Wang, H., Munger, P.W., Alexander, R.T., 2003. *Design patterns and change proneness: An examination of five evolving systems*, Software metrics symposium, 2003. Proceedings. Ninth international. IEEE, pp. 40-49.

Boehm, B., Basili, V.R., 2001. *Software Defect Reduction Top 10 List*. Computer 34, 135-137.

Boger, M., Sturm, T., Fragemann, P., 2003. *Refactoring browser for UML, Objects, components, architectures, services, and applications for a networked world*. Springer, pp. 366-377.

Bourque, P., Fairley, R.E., 2014. *Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society.

Briand, L.C., Daly, J., Porter, V., Wust, J., 1998. *Predicting fault-prone classes with design measures in object-oriented systems*. Ninth International Symposium on Software Reliability Engineering, Proceedings, 334-343.

Briand, L.C., Labiche, Y., Yihong, W., 2001a. *Revisiting strategies for ordering class integration testing in the presence of dependency cycles*, Proc. 12th International Symposium on Software Reliability Engineering, (ISSRE 2001) pp. 287-296.

Briand, L.C., Labiche, Y., Yihong, W., 2003. *An investigation of graph-based class integration test order strategies*. IEEE Transactions on Software Engineering. 29, 594-607.

Briand, L.C., Wuest, J., Lounis, H., 2001b. *Replicated Case Studies for Investigating Quality Factorsin Object-Oriented Designs*. Empirical Softw. Engg. 6, 11-58.

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., 2010. *Managing technical debt in software-reliant systems*, Proceedings of the FSE/SDP workshop on Future of software engineering research. ACM, pp. 47-52.

Chidamber, S.R., Kemerer, C.F., 1994. *A Metrics Suite for Object-Oriented Design*. IEEE Transactions on Software Engineering 20, 476-493.

Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong, M.-Y., 1992. *Orthogonal defect classification - a concept for in-process measurements*. IEEE Transactions on Software Engineering, 18, 943-956.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2001. *Introduction to algorithms*, 2nd ed. MIT Press, Cambridge, Mass.

Correa, A., Werner, C., 2004. *Applying refactoring techniques to UML/OCL models, «UML» 2004—The Unified Modeling Language.* Modeling Languages and Applications. Springer, pp. 173-187.

Creel, R., Ellison, B., 2008. *System-of-Systems Influences on Acquisition Strategy Development*. Carnegie Mellon University, Carnegie Mellon University.

Creswell, J.W., 2013. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. SAGE Publications.

DeMarco, T., 1982. *Controlling software projects*. Prentice Hall PTR.

Di Penta, M., Cerulo, L., Guéhéneuc, Y.-G., Antoniol, G., 2008. *An empirical study of the relationships between design pattern roles and class change proneness*, IEEE International Conference on Software Maintenance, 2008. ICSM 2008. IEEE, pp. 217-226.

Dietrich, J., McCartin, C., Tempero, E., Shah, S.M.A., 2010. *Barriers to Modularity-An Empirical Study to Assess the Potential for Modularisation of Java Programs*, Research into Practice–Reality and Gaps. Springer, pp. 135-150.

Dietrich, J., McCartin, C., Tempero, E., Shah, S.M.A., 2012. *On the existence of high-impact refactoring opportunities in programs*, Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122. Australian Computer Society, Inc., Melbourne, Australia, pp. 37-48.

Easterbrook, S., Singer, J., Storey, M.-A., Damian, D., 2008. *Selecting Empirical Methods for Software Engineering Research*, in: Shull, F., Singer, J., Sjøberg, D.K. (Eds.), Guide to Advanced Empirical Software Engineering. Springer London, pp. 285-311.

Ebert, C., Dumke, R., Bundschuh, M., Schimietendorf, A., 2005. *Defect Detection and Quality Improvement, Best Practices in Software Measurement*. Springer Berlin Heidelberg, pp. 133-156.

Falleri, J.-R., Denier, S., Laval, J., Vismara, P., Ducasse, S., 2011. *Efficient retrieval and ranking of undesired package cycles in large software systems*, Proceedings of the 49th international conference on Objects, models, components, patterns. Springer-Verlag, Zurich, Switzerland, pp. 260-275.

Fenton, N.E., Neil, M., 1999a. *A critique of software defect prediction models*. IEEE Transactions on Software Engineering 25, 675-689.

Fenton, N.E., Neil, M., 1999b. *Software metrics: successes, failures and new directions*. Journal of Systems and Software 47, 149-157.

Fenton, N.E., Ohlsson, N., 2000. *Quantitative analysis of faults and failures in a complex software system*. IEEE Transactions on Software Engineering 26, 797-814.

Fenton, N.E., Pfleeger, S.L., 1997. *Software Metrics: A Rigorous & Practical Approach*, 2nd ed. PWS Publishing Press, Boston.

Fenton, N.E., Pfleeger, S.L., 1998. *Software Metrics: A Rigorous and Practical* Approach. PWS Publishing Co.

Finkelstein, A.C., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B., 1994. *Inconsistency handling in multiperspective specifications*. IEEE Transactions on Software Engineering, 20, 569-578.

Fowler, M., 1999. *Refactoring: improving the design of existing code*. Pearson Education India.

Fowler, M., 2001. *Reducing coupling*. IEEE Software, 18, 102-104.

Fowler, M., 2004. *Inversion of control containers and the dependency injection pattern*, http://martinfowler.com/articles/injection.html.

Franch, X., 1998. *Systematic formulation of non-functional characteristics of software*, Proceedings. 1998 Third IEEE International Conference on Requirements Engineering, pp. 174-181.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc.

Genero, M., Manso, E., Visaggio, A., Canfora, G., Piattini, M., 2007. *Building measure-based prediction models for UML class diagram maintainability*. Empir Softw Eng 12, 517-549.

Gillies, A., 1997. *Software Quality: Theory and Management*. International Thomson Computer Press.

Gobner, J., Mayer, P., Steimann, F., 2004. *Interface utilization in the Java Development Kit*, Proceedings of the 2004 ACM symposium on Applied computing. ACM, Nicosia, Cyprus, pp. 1310-1315.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2011. *A Systematic Review of Fault Prediction Performance in Software Engineering*. IEEE Transactions on Software Engineering 99.

Halstead, M.H., 1977. *Elements of Software Science* (Operating and programming systems series). Elsevier Science Inc.

Hanh, V.L., Akif, K., Le Traon, Y., Jezequel, J.M., 2001. *Selecting an Efficient OO Integration Testing Strategy: An Experimental Comparison of Actual Strategies*. Proc. 15th European Conf. Object-Oriented Programming (ECOOP), 381-401.

Hashim, N.L., Schmidt, H.W., Ramakrishnan, S., 2005. *Test order for class-based integration testing of Java applications*, Fifth International Conference on Quality Software, 2005. (QSIC 2005). , pp. 11-18.

Hatton, L., 1997. *Reexamining the fault density - Component size connection*. IEEE Software 14, 89-97.

Hautus, E., 2002. *Improving Java Software Through Package Structure Analysis*, The 6th IASTED International Conference Software Engineering and Applications, Cambridge, MA, USA.

Henry, S., Kafura, D., 1981. *Software structure metrics based on information flow*. IEEE Transactions on Software Engineering, 510-518.

Herzig, K., Zeller, A., 2013. *The impact of tangled code changes*, Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, San Francisco, CA, USA, pp. 121-130.

Hevner, A.R., March, S.T., Park, J., Ram, S., 2004. *Design science in information systems research*. MIS Q. 28, 75-105.

Hunter, A., Nuseibeh, B., 1998. *Managing inconsistent specifications: reasoning, analysis, and action*. ACM Transactions on Software Engineering and Methodology (TOSEM) 7, 335-367.

IEEE Std 982.1-2005. *IEEE Standard Dictionary of Measures of the Software Aspects of Dependability*. IEEE Std 982.1-2005 (Revision of IEEE Std 982.1-1988), 0_1-34.

IEEE Std 1012-1986. *IEEE Standard for Software Verification and Validation Plans*. IEEE Std 1012-1986, I.

IEEE Std 1044-2009. *IEEE Standard Classification for Software Anomalies*. IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993), 1-23.

ISO/IEC 25000:2014. *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE)* - Guide to SQuaRE.

ISO/IEC 25010:2011. *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE)* - System and software quality models.

ISO/IEC/IEEE 24765:2010. *Systems and software engineering—Vocabulary*.

Jeanmart, S., Gueheneuc, Y.-G., Sahraoui, H., Habra, N., 2009. *Impact of the visitor pattern on program comprehension and maintenance*, Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement. IEEE Computer Society, pp. 69-78.

Jones, C., 1985. *Programming productivity*. McGraw-Hill, Inc.

Jungmayr, S., 2002. *Identifying test-critical dependencies*, Software Maintenance, pp. 404-413.

Khomh, F., Di Penta, M., Guéhéneuc, Y.-G., Antoniol, G., 2012. *An exploratory study of the impact of antipatterns on class change-and fault-proneness*. Empir Softw Eng 17, 243-275.

Kim, M., Cai, D., Kim, S., 2011. *An empirical investigation into the role of API-level refactorings during software evolution*, Proceedings of the 33rd International Conference on Software Engineering. ACM, Waikiki, Honolulu, HI, USA, pp. 151-160.

Kim, M., Zimmermann, T., Nagappan, N., 2012. *A field study of refactoring challenges and benefits*, Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, Cary, North Carolina, pp. 1-11.

Kitchenham, B., Pfleeger, S.L., 1996. *Software quality: The elusive target*. IEEE Software 13, 12-21.

Knoernschild, K., 2012. *Java Application Architecture: Modularity Patterns with Examples Using OSGi*, 1st ed. Prentice Hall.

Koenig, A., 1998. *Patterns and antipatterns*, in: Linda, R. (Ed.), The patterns handbooks. Cambridge University Press, pp. 383-389.

Kung, D., Gao, J., Hsia, P., Toyoshima, Y., Chen, C., 1996. *On Regression Testing of Object-Oriented Programs*. Journal of Systems and Software 32, 21-40.

Lakos, J., 1996. *Large-scale C++ software design*. Addison-Wesley Longman, Redwood City, CA.

Laval, J., Denier, S., Ducasse, S., Bergel, A., 2009. *Identifying Cycle Causes with Enriched Dependency Structural Matrix*, Proceedings of the 2009 16th Working Conference on Reverse Engineering. IEEE Computer Society, pp. 113-122.

Laval, J., Ducasse, S., 2014. *Resolving cyclic dependencies between packages with enriched dependency structural matrix*. Software Pract Exper 44, 235-257.

Lehman, M.M., 1980. *Programs, Life-Cycles, and Laws of Software Evolution*. Proceedings of the Special Issue Software Engineering 68, 1060-1076.

Lehman, M.M., Ramil, J.F., 2001. *Rules and tools for software evolution planning and management*. Annals of software engineering 11, 15-44.

Leo, K., 2013. *Why banks are likely to face more software glitches in 2013.*, BBC. http://www.bbc.com/news/technology-21280943 (Accessed April 24, 2013)

Li, W., Henry, S., 1993. *Object-Oriented Metrics That Predict Maintainability*. Journal of Systems and Software 23, 111-122.

Lientz, B.P., Swanson, E.B., Tompkins, G.E., 1978. *Characteristics of application software maintenance*. Commun Acm 21, 466-471.

Lilley, S., 2012. *Critical Software: Good Design Built Right*. NASA System Failure Case Studies 6.

Lippert, M., Roock, S., 2006. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. John Wiley & Sons.

Mäntylä, M.V., Lassenius, C., 2006. *Subjective evaluation of software evolvability using code smells: An empirical study*. Empir Softw Eng 11, 395-431.

Martin, R.C., 1996. *Granularity*, C++ Report, pp. 57-62.

Martin, R.C., 1997. *Acyclic visitor, Pattern languages of program design 3*. Addison-Wesley Longman Publishing Co., Inc., pp. 93-103.

Martin, R.C., 2000. *Design principles and design patterns*. Object Mentor 1, 34.

Mayer, P., 2003. *Analyzing the use of interfaces in large OO projects*, Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. ACM, Anaheim, CA, USA, pp. 382-383.

McCabe, T.J., 1976. *A complexity measure*. IEEE Transactions on Software Engineering, 308-320.

McCall, J.A., Richards, P.K., Walters, G.F., 1977. *Factors in software quality*. General Electric, National Technical Information Service.

Melton, H., Tempero, E., 2007a. *The CRSS metric for package design quality*, Proceedings of the thirtieth Australasian conference on Computer science - Volume 62. Australian Computer Society, Inc., Ballarat, Victoria, Australia, pp. 201-210.

Melton, H., Tempero, E., 2007b. *An empirical study of cycles among classes in Java*. Empir Softw Eng 12, 389-415.

Melton, H., Tempero, E., 2007c. *JooJ: real-time support for avoiding cyclic dependencies*. Proceedings of the thirtieth Australasian conference on Computer science 62, 87-95.

Mens, T., 2006. *On the use of graph transformations for model refactoring*, Proceedings of the 2005 international conference on Generative and Transformational Techniques in Software Engineering. Springer-Verlag, Braga, Portugal, pp. 219-257.

Mens, T., Taentzer, G., Runge, O., 2007. *Analysing refactoring dependencies using graph transformation*. Softw Syst Model 6, 269-285.

Mens, T., Tourwe, T., 2004. *A survey of software refactoring*. IEEE Transactions on Software Engineering, 30, 126-139.

Mohagheghi, P., 2004. *The Impact of Software Reuse and Incremental Development on the Quality of Large Systems*, Computer and Information Science. NTNU, NTNU, p. 272.

Moller, K.H., Paulish, D.J., 1993. *An empirical investigation of software fault distribution,* Software Metrics Symposium, 1993. Proceedings., First International, pp. 82-90.

Murphy-Hill, E., Parnin, C., Black, A.P., 2009. *How we refactor, and how we know it*, Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, pp. 287-297.

Muschevici, R., Potanin, A., Tempero, E., Noble, J., 2008. *Multiple dispatch in practice*, Acm Sigplan Notices. ACM, pp. 563-582.

Naik, K., Tripathy, P., 2011. *Software Testing and Quality Assurance: Theory and Practice*. Wiley.

Naur, P., Randell, B., 1969. *Software Engineering: Report of a conference sponsored by the NATO Science Committee*, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO.

NIST, 2010. *NIST Framework and Roadmap for Smart Grid Interoperability Standards Release 1.0.* NIST Special Publication 1108.

Olbrich, S.M., Cruzes, D.S., Sjoberg, D.I., 2010. *Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems*, Software Maintenance (ICSM), 2010 IEEE International Conference on. IEEE, pp. 1-10.

Opdyke, W.F., 1992. *Refactoring object-oriented frameworks*. University of Illinois at Urbana-Champaign.

Ostrand, T., Weyuker, E., 2002. *The distribution of faults in a large industrial software system*. ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis 27, 55-64.

Oyetoyan, T.D., Cruzes, D., Thurmann-Nielsen, C., 2015a. *A Decision Support System for breaking Class Cycles*, 31st IEEE International Conference on Software Maintenance and Evolution, Bremen, Germany (Submitted).

Oyetoyan, T.D., Cruzes, D.S., Conradi, R., 2012. *Can Reused Components Provide Lead to Future Defective Components in Smart Grid Applications*?, Parallel and Distributed Computing and Systems : Software Engineering and Applications (PDCS 2012). ACTA Press.

Oyetoyan, T.D., Cruzes, D.S., Conradi, R., 2013a. *Criticality of Defects in Cyclic Dependent Components*, 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), Eindhoven, Netherlands, pp. 21-30.

Oyetoyan, T.D., Cruzes, D.S., Conradi, R., 2013b. *A study of cyclic dependencies on defect profile of software components*. Journal of Systems and Software 86, 3162-3182.

Oyetoyan, T.D., Jens, D., Jezek, K., Falleri, J.-R., 2015b. *Circular Dependencies and Change-proneness: An Empirical Study*, 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, École Polytechnique de Montréal, Québec, Canada, pp. 238-247.

Park, R.E., Goethert, W.B., Florac, W.A., 1996. *Goal-Driven Software Measurement. A Guidebook*. DTIC Document.

Parnas, D.L., 1979. *Designing Software for Ease of Extension and Contraction*. IEEE Transactions on Software Engineering SE-5, 128-138.

Parnas, D.L., 1994. *Software aging*, Proceedings of the 16th international conference on Software engineering. IEEE Computer Society Press, pp. 279-287.

Peffers, K., Tuunanen, T., Rothenberger, M., Chatterjee, S., 2007. *A Design Science Research Methodology for Information Systems Research*. J. Manage. Inf. Syst. 24, 45-77.

Pfleeger, S.L., 2001. *Software Engineering: Theory and Practice*. Prentice Hall.

Poston, R., 1987. *Preventing Most-Probable Errors in Requirements*. IEEE Software 4, 81-83.

Prechelt, L., Unger-Lamprecht, B., Philippsen, M., Tichy, W.F., 2002. *Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance*. IEEE Transactions on Software Engineering, 28, 595-606.

Radatz, J., Geraci, A., Katki, F., 1990. *IEEE standard glossary of software engineering terminology*. IEEE Std 610121990, 121990.

Ratzinger, J., Sigmund, T., Gall, H.C., 2008. *On the relation of refactorings and software defect prediction*, Proceedings of the 2008 international working conference on Mining software repositories. ACM, Leipzig, Germany, pp. 35-38.

Riel, A.J., 1996. *Object-oriented Design Heuristics*. Addison-Wesley Publishing Company.

Rising, L., 1998. *The Patterns Handbook: Techniques, Strategies, and Applications*. SIGS.

Robson, C., 2011. *Real world research : a resource for users of social research methods in applied settings*, 3rd ed. Wiley-Blackwell, Chichester, West Sussex ; Hoboken, N.J.

Roger, S.P., 2005. *Software engineering: a practitioner's approach*. McGrow-Hill International Edition.

Romano, D., Raila, P., Pinzger, M., Khomh, F., 2012. *Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes*, Reverse Engineering (WCRE), 2012 19th Working Conference on. IEEE, pp. 437-446.

Russo, A., Nuseibeh, B., Kramer, J., 1998. *Restructuring requirements specifications for managing inconsistency and change: A case study*, Proceedings of the Third IEEE International Conference on Requirements Engineering, pp. 51-60.

Sakkinen, M., 1989. *Disciplined Inheritance*, ECOOP, pp. 39-56.

Sangal, N., Jordan, E., Sinha, V., Jackson, D., 2005. *Using dependency models to manage complex software architecture*. SIGPLAN Not. 40, 167-176.

Schroeter, A., Zimmermann, T., Zeller, A., 2006. *Predicting component failures at design time*, Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering. ACM, Rio de Janeiro, Brazil, pp. 18-27.

Shah, S.M.A., Dietrich, J., Mccartin, C., 2013. *On the Automation of Dependency-Breaking Refactorings in Java*, 29th IEEE International Conference on Software Maintenance (ICSM), Eindhoven, Netherlands, pp. 160 - 169.

Sindre, G., Conradi, R., Karlsson, E.A., 1995. *The Reboot Approach to Software Reuse*. Journal of Systems and Software 30, 201-212.

Singh, G.B., 1994. *Single versus multiple inheritance in object oriented programming*. ACM SIGPLAN OOPS Messenger 5, 34-43.

Sommerville, I., 2011. *Software Engineering*. Pearson Education.

Steimann, F., Siberski, W., Kuhne, T., 2003. *Towards the systematic use of interfaces in JAVA programming*, Proceedings of the 2nd international conference on Principles and practice of programming in Java. Computer Science Press, Inc., Kilkenny City, Ireland, pp. 13-17.

Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J.-M., 2001. *Refactoring UML models, « UML» 2001—The Unified Modeling Language*. Modeling Languages, Concepts, and Tools. Springer, pp. 134-148.

Tarjan, R., 1972. *Depth-first search and linear graph algorithms*. SIAM journal on computing 1, 146-160.

Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J., 2010. *The Qualitas Corpus: A curated collection of Java code for empirical studies*, 17th Asia Pacific Software Engineering Conference (APSEC), 2010 IEEE, pp. 336-345.

Tihana Galinac, G., Per, R., Darko, H., 2012. *A Second Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems*. IEEE Transactions on Software Engineering 99, 1-1.

Tsantalis, N., Chatzigeorgiou, A., 2011. *Identification of extract method refactoring opportunities for the decomposition of methods*. Journal of Systems and Software. 84, 1757-1782.

Van Der Straeten, R., D'Hondt, M., 2006. *Model refactorings through rule-based inconsistency resolution*, Proceedings of the 2006 ACM symposium on Applied computing. ACM, pp. 1210-1217.

Van Der Straeten, R., Jonckers, V., Mens, T., 2004. *Supporting Model Refactorings Through Behaviour Inheritance Consistencies*, in: Baar, T., Strohmeier, A., Moreira, A., Mellor, S. (Eds.), «UML» 2004 — The Unified Modeling Language. Modeling Languages and Applications. Springer Berlin Heidelberg, pp. 305-319.

Van Gorp, P., Stenten, H., Mens, T., Demeyer, S., 2003. *Towards automating source-consistent UML refactorings*, «UML» 2003-The Unified Modeling Language. Modeling Languages and Applications. Springer, pp. 144-158.

van Vliet, H., 2000. *Software Engineering: Principles and Practice*. Wiley.

Vokac, M., 2004. *Defect frequency and design patterns: An empirical study of industrial code*. IEEE Transactions on Software Engineering 30, 904-917.

Vokáč, M., Tichy, W., Sjøberg, D.I., Arisholm, E., Aldrin, M., 2004. *A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment.* Empir Softw Eng 9, 149-195.

Wake, W.C., 2004. *Refactoring workbook*. Addison-Wesley Professional.

Wasserman, S., Faust, K., 1994. *Social network analysis : methods and applications*. Cambridge University Press, Cambridge ; New York.

Weissgerber, P., Diehl, S., 2006. *Are refactorings less error-prone than other changes?*, Proceedings of the 2006 international workshop on Mining software repositories. ACM, Shanghai, China, pp. 112-118.

Wendorff, P., 2001. *Assessment of design patterns during software reengineering: Lessons learned from a large commercial project*, Fifth European Conference on Software Maintenance and Reengineering, 2001. IEEE, pp. 77-84.

Wohlin, C., Host, M., K, H., 2003. *Empirical Research Methods in Software Engineering*, in: Conradi, R., Wang, A.I. (Eds.), In Lecture Notes in Empirical Methods and Studies in Software Engineering: Experiences from ESERNET. Springer Verlag.

Zhang, J., Lin, Y., Gray, J., 2005. *Generic and Domain-Specific Model Refactoring using a Model Transformation Engine,* in: Sami Beydeda, M.B., and Volker Gruhn, eds (Ed.), Model-driven Software Development. Springer, pp. 199-218.

Zimmerman, T., Nagappan, N., Herzig, K., Premraj, R., Williams, L., 2011. *An Empirical Study on the Relation between Dependency Neighborhoods and Failures*, IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST), pp. 347 - 356.

Zimmermann, T., Nagappan, N., 2007. *Predicting subsystem failures using dependency graph complexities*. ISSRE 2007: 18th IEEE International Symposium on Software Reliability Engineering, Proceedings, 227-236.

Zimmermann, T., Nagappan, N., 2008. *Predicting Defects using Network Analysis on Dependency Graphs.* 2008 30th International Conference on Software Engineering: (ICSE), Vols 1 and 2, 530-539.

Appendix A: Selected papers

This Appendix contains the seven papers that have contributed towards the work presented in this thesis. They are presented in a chronological order.

P1. A Comparison of Different Defect Measures to Identify Defect-Prone Components

P2. A study of cyclic dependencies on defect profile of software components

P3. Criticality of Defects in Cyclic Dependent Components

P4. Can Refactoring Cyclic Dependent Components Reduce Defect-Proneness?

P5. Transition and Defect Patterns of Components in Dependency Cycles During Software Evolution

P6. Circular Dependencies and Change-Proneness: An Empirical Study

P7. A Decision Support System to Refactor Class Cycles

P1: Comparison of Different Defect Measures to Identify Defect-Prone Components

Published In Proc. Joint Conference of the 23rd International Workshop on Software Measurement and the 2013 8th International Conference on Software Process and Product Measurement (IWSM-MENSURA), Ankara, Turkey. pp. 181-190

Comparison of Different Defect Measures to Identify Defect-Prone Components

Tosin Daniel Oyetoyan[1], Reidar Conradi[1]
[1]Department of Computer and Information Science
Norwegian University of Science and Technology, Trondheim, Norway
{tosindo, conradi}@idi.ntnu.no

Daniela Soares Cruzes[1,2]
[2]SINTEF
Trondheim, Norway
dcruzes@idi.ntnu.no

***Abstract*—(*Background*) Defect distribution in software systems has been shown to follow the Pareto rule of 20-80. This motivates the prioritization of components with the majority of defects for testing activities. (*Research goal*) Are there significant variations between defective components and architectural hotspots identified by other defect measures? (*Approach*) We have performed a study using post-release data of an industrial Smart Grid application with a well-maintained defect tracking system. Using the Pareto principle, we identify and compare defect-prone and hotspots components based on four defect metrics. Furthermore, we validated the quantitative results against qualitative data from the developers. (*Results*) Our results show that at the top 25% of the measures 1) significant variations exist between the defective components identified by the different defect metrics and that some of the components persist as defective across releases 2) the top defective components based on number of defects could only identify about 40% of critical components in this system 3) other defect metrics identify about 30% additional critical components 4) additional quality challenges of a component could be identified by considering the pairwise intersection of the defect metrics. (*Discussion and Conclusion*) Since a set of critical components in the system is missed by using largest-first or smallest-first prioritization approaches, this study, therefore, makes a case for an all-inclusive metrics during defect model construction such as number of defects, defect density, defect severity and defect correction effort to make us better understand what comprises defect-prone components and architectural hotspots, especially in critical applications.**

Keywords—defect distribution; defect measures; defect metrics; defect severity; defect correction effort; defect density; defect-prone component; Smart Grid; critical system; architectural hotspots

I. Introduction

Software testing is resource intensive in complex industrial systems [1], as such; any information that can help reduce testing effort is a step in the right direction. This is mainly the motivation for developing defect models to predict the part of the system that has the highest likelihood to be defect prone. The early knowledge of the components that may turn defective in the future release of a system is useful to focus code inspections, walkthroughs and reviews activities to catch defects in such part of the system.

However, existing defect models are largely constructed to predict based on number of defects (e.g. [2-9]). The study by Adams [10] showed that removing large number of defects may have a trivial effect on reliability. As pointed out in [10, 11], the most number of latent defects lead to very rare failure in practice while the vast majority of observed failures are caused by a relatively tiny number of defects. In addition to this observation, both Ebert et al. [9] and Boehm and Basili [8] argued that 60-80% of the correction effort and 80% of avoidable rework are due to 20% of the defects. Therefore, showing that it is not the number of defects, rather their severity that matters. A high severity defect usually points to a fatal error that results into system failure whereas low severity defects mostly point to some cosmetic issues. Few studies have reported defect prediction models that predict the defect-proneness of components using defect severity (e.g. [12-14]).

Several studies have also suggested that smaller components have higher likelihood of defects when compared to the larger ones [7, 15-17]. These studies have used defect density measured as number of defects per thousand lines of code (LOC). Since it has been demonstrated that most complexity metrics correlate with a component's size [18], it supposes that more complex

components and invariably larger components are given higher priority in prediction models that use defect count approach. Results from literature [19] show that few studies have reported models that are based on defect density (e.g. [2, 20-22]) when compared to using number of defects. Koru et al. [23], by using the proposed theory of relative defect proneness (RDP) argued that the largest-first prioritization should lead to less effective defect detection compared to the smallest-first approach.

Van Moll et al. [24] claimed that those components that are defect-prone and difficult to maintain should be the candidates for high priority inspection and testing. Thus maintenance effort may be pointing to structural complexity [25] in the components or could be a case of changing business requirements that is typical in critical system of systems (SoS) [26]. For instance, implementing a new technology (e.g. IPv6). The effort could be more than predicted as told by one of the stakeholders in the software company of the system under study. Very few studies have focused on predicting maintenance or development effort at the class level [25, 27, 28].

Thus, on one hand, a plethora of studies have used the largest-first prioritization, while on the other hand, a number of studies have suggested that the smallest-first approach would be more effective. In the light of this, it becomes obvious that in a software system, we can begin to focus on defective components from different points of views using both direct and indirect defect measures. Since there is no synergy between the various defect measures in a single defect prediction model environment, we conjecture that some significant differences might exist between what is identified by the top most defective components based on number of defects and those that are based on other measures.

We also hypothesize that defect distribution based on defect count may not identify most of the critical components to the application in spite of the number of defects that the defective components may account for. In a software system certain components are more critical than the others. For instance, a component (connector) that integrates hardware and other subsystems is obviously critical to the application. The performance and reliability of the system will depend on such critical components. It is in this context that we are interested to know how much of such critical components can be identified at the top region of these defect measures and how this knowledge can help in testing effort and improvement of the components' quality.

Therefore, the central goal of this study is to examine and investigate the differences between components identified by four defect measures, namely; number of defects, defect density, severity of defect and defect correction effort. We want to know how significant or trivial these differences are in terms of the criticality of the components to the application under study. In addition, we want to know, if those differences persist across releases in this application. To the best of our knowledge, we have not found any study that focused on investigating the gap or the synergy between the distributions of several defect measures on the affected components at the same time.

Our hypothesis is that all the data should be seen in tandem in order to not misplace critical defective components and architectural hotspots in critical systems. Architectural hotspots are defect-prone components with interface defects [3, 29]. This study complements other studies with focus on defect distribution in large and complex industrial systems. We conjecture that to be able to predict and identify subtle but critical defective components and architectural hotspots in critical systems need more than only one defect measure and one direction of prioritization.

The rest of the work is structured as follows; in section II, we discuss related work to this study. Section III describes the software application that is under study. In section IV, we detail our empirical set up, such as: the definitions, research questions to be investigated and our approach of data collection. The results and discussion are provided in section V, while in section VI, we draw out the threats to the validity of our results. Lastly, section VII

provides the conclusion to this study with a note on future work.

II. Related work

Several empirical studies have shown the distribution of defects in software components to follow a 20-80 rule [7-9, 11, 15, 30, 31], that is, the so-called Pareto distribution principle. Among these studies only Ostrand and Weyuker [7] reported the distribution of defective components based on the severity of their defects.

Defect distribution has also explored the relationship between size, measured as lines of code (LOC) and defect proneness of components. Two measures are focused in this relationship, the absolute number of defects and defect density. Studies in [7, 15-17] found trends between size and defect density. Either the defect density increases as the component size decreases [15] or that defect density increases above a size threshold and decreases below the size threshold [16, 17] or that defect density decreases up to a certain size threshold and then leveled up [7]. The findings in these studies suggest (1) that smaller modules are more defect-prone than the larger ones and (2) that there is a medium size that indicates the optimal size for a component. El Emam et al. [32] disputed these studies and reported that plotting the size vs. defect density can be misleading and that no such thresholds of optimal component's size exist in the systems they analyzed. However, a study by Koru et al. [33] that focused their analysis on size-defect relationship solely arrived at the same conclusion of previous studies that smaller components are more defect-prone than larger components. Results from Fenton and Ohlsson [11] supported neither claims, their studies found no trend between size and defect density.

Defect prediction models have taken advantage of defect distribution in a system to build models that can predict the defect proneness of software components. For example, many novel approaches have tested their models on the most defective parts of the system. Ostrand et al. [2] validated their prediction models using the top 20%. Schroeter et al. [34] obtained their best model by testing with the top 5% of the defective files in Eclipse. Zhang et al. [4] used the statistical medium to define a defect dense component for their model. Briand et al. [25] supported the largest-first prioritization of components during inspection and concluded that to use model across several projects, the number of the "largest-first" would in practice be driven by available time and budget.

Maintenance effort prediction at a fine-grained level such as class level has also been explored. Li and Henry [35] built a regression model using object-oriented (OO) metrics as independent variables and class change frequency as dependent variable to model maintenance effort. Bocco et al. [28] reported that the number of methods and the number of associations are good predictors of maintenance effort of a UML class diagram. Alshayeb and Li [27] concluded that OO metrics could fairly predict error-fix effort of a class in an XP-like process only when the system has sufficient design structure. Briand et al. [25] showed that fairly accurate prediction of class development effort could be made based on the class interface size alone.

These previous studies have mainly focused on identifying defective components that can be focused for quality assurance by using the metrics independently. Furthermore, they have not considered how much of critical defective components to a system can be identified using any or all of these metrics. Also, there is no report of how the defect metrics can complement one another when dealing with defective components that should be given priority in testing activities. In this study, we seek to empirically validate the quantity of critical defective components that can be identified using largest-first and smallest-first prioritization approaches as used in previous studies. In addition, we seek to understand if some other defect metrics such as defect correction effort and defect severity can identify different critical defective components from the other defect metrics. Lastly, we want to find out how by using several measures we can prioritize testing activities.

III. System Description

In this study, we performed an empirical study of an industrial Smart Grid application, a type of system of systems (SoS) applications. Several domains of Smart Grid exist such as the generation, transmission, distribution, markets, operations, service providers and consumers [36] with different types of software running in these domains (legacy systems and new applications). This software, in addition, performs different functions and provides different services.

Our motivation for the choice of this case study is that, as a critical infrastructure, the availability and reliability of the Smart Grid is crucial to its safety and security. Smart Grid represents the injection of Information and Communication Technology (ICT) infrastructure to the electricity grid to allow for bi-directional flow of energy and information [37]. Smart Grid is still in the formation stage, and represents a shift from relatively closed grid to a more complex and highly interconnected systems.

Although, efforts are in place to develop interoperable standard, there is still substantial gap identified as a results of evolving requirements and different implemented software and hardware products [38]. The fact that these systems have to interoperate poses a quality challenge for the entire system. For instance, if software for collecting data from field devices that are designated for monitoring the health or quality of the transmission line fails as a result of software defect, the end operator (human or automated device) is denied real-time data for taking adequate control action.

Thus, defect analysis using realistic and useful measurements is needed to support QA focus on different and identified defective parts of the various Smart Grid applications. A defect that results into failure in one part of such critical system can have cascading and serious effects on the rest of the Smart Grid systems. Since different software will drive different Smart Grid systems and many of the system will interoperate and integrate together, proper identification of critical defect-prone components in these systems is important for improving the overall quality of the grid.

The system under study is a distribution management system designed to monitor and plan the Grid operations. It provides real-time operational support by continuously receiving status data from the power grid. By using this data, automation processes can then provide improved error handling, operational statistics, reporting and automated customer notification that ensure that customers are informed of power interruptions and irregularities.

The system consists of three main parts:

- The design module that is used for designing the single line diagrams based on input from Network Information System (NIS).
- The operation center that monitors the distribution grid (through real-time data from the SCADA-system), where the operator performers switching in case of power interruptions or the execution of planned maintenance/repair. In addition the operations center simulates effects of changes to the grid and plans grid operations. It can also notify affected customers automatically.
- The call center that provides information about the current grid status for customer center representatives, and also sends customer observations back to the distribution management system.

The system has been in development for about six years and we have analyzed six post releases (field and operational) of this application. It is mostly developed with C# programming language with .NET framework. As listed in Table I, the system consists about 380KLOC and contains 1459 class files and 2484 classes as of version 4.2.4. In this system, a reported defect goes through several stages in its life cycle before it is finally closed. First, it is reviewed to ascertain if it is a defect, then it is assigned to a developer. Subsequently, the defect is closed after it has been fixed and tested. The actual fix effort (person-hours) therefore comprises the time to code, test and provide documentation. For copyright reason, we can only

provide the meta-data and the results of the study in this paper.

Initial investigation of the defects data shows an agreement with the Pareto distribution of 20-80 as noted in [8, 9]. Figure- 1a shows the distribution of defects against correction efforts. Between 20-40% of defects are responsible for an average of about 80% of the correction efforts. Figure-1b displays the distribution of defective class files against the cumulative number of defects. In five of the releases, 20% of the defective class files are responsible for approximately 80% of the total number of defects. We can therefore safely conclude that the Pareto distribution of 20-80 holds for this system. Figure 2 shows defect density between 0.14 and 0.012 defects/LOC for the smallest components (class-files) less than 90 LOC and starting to flatten at 0.0002 defects/LOC for class-files larger than 680 LOC. This trend agrees with the results reported in [7] and is similar to the rest of the releases.

IV. Empirical setup

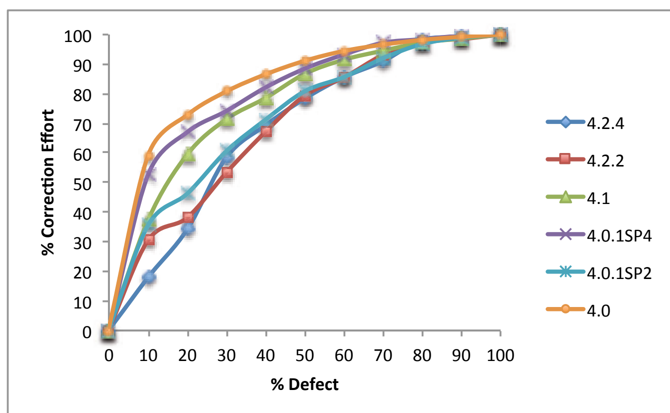
Our goal is to investigate if there are significant differences between the distributions of defect measures over components and if they persist. In order to adequately investigate our research goal, it is appropriate to define some of the terms that are relevant and used in the rest of this paper.

A. Definitions

Defect-prone component (DPC): A defect-prone component is defined as a component in the top 25% of components with the most number of defects.

TABLE I. SUMMARY OF SOFTWARE SOURCE CODE AND DEFECT DATA

Release	Date	#Pkg	#Class-File	#Class	KLOC	#Defective Class-File	#Defect
4.2.4	Nov 14 2012	261	1459	2484	380.50	29	14
4.2.2	Oct 12 2012	261	1454	2475	378.25	49	18
4.1	Aug 17 2012	237	1246	2208	353.04	60	42
4.0.1SP4	Apr 11 2012	204	1141	1953	322.81	69	29
4.0.1SP2	Mar 26 2012	205	1139	1947	321.55	46	28
4.0	Oct 14 2011	187	1041	1791	296.00	137	143



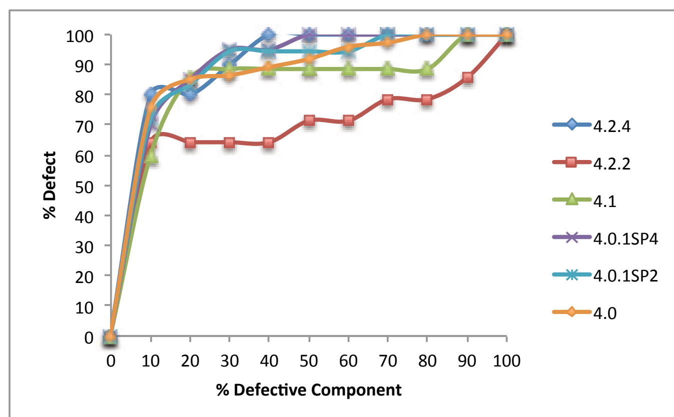


Fig. 1. (a) Distribution of defect against correction effort (b) Distribution of defective component against defect

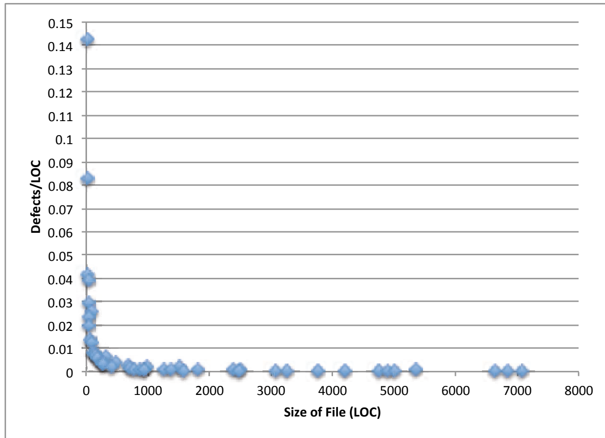


Fig. 2. Defects/LOC vs. Size (LOC)

Defect-dense component (DDC): A defect-dense component is defined as component in the top 25% of components with the most number of defect densities.

Severe defective component (SDC): A severe defective component is defined as a component within the top 25% with the most severe defects.

Hard-to-fix defective component (HFC): A hard-to-fix defective component is defined as a component within the top 25% with the highest correction effort.

Architectural hotspots: An architectural hotspot is defined as a component within the top 25% with the highest number of multiple-component defects (MCD). This definition is similar to Li et al. [3].

Multiple-component Defect (MCD): A MCD is a defect that affects more than one component. For detail discussions about MCD and interface defects see [3] and [29] respectively.

Component: A component in our study represents a class file in C#.

The decision to use the top 25% is based on the distribution of the system's defects data as shown in Figure 1b. As observed in this plot, the top 25% of defective components account for an average of 80% of the total defects. It is thus practical to use this number for our definitions. Li et al. [3] used 20% to define the various measures in the study and argued that the quality of the system has no bearing on the top 20%. Irrespective of how defective the system may be, the 20% will still identify about 80% of MCDs. We find this to be true in the system under study. Since the goal of our study is to investigate if differences occur between these measures, we find it appropriate to use this uniform figure (25%) across the various constructs.

In our system, the interface defects among the source files account for an average of 78% of all the reported defects. This seems to agree more with the reported figure by Perry and Evangelist [29] than the figures in Li et al. [3] Therefore, in our subsequent analysis, we have treated both DPC and hotspots in the same way and did not analyze hotspots separately.

B. Research questions

In this study, we investigate three research questions that can help us clarify our original research goal.

RQ1. *Are there differences between components identified by DPC, DDC, SDC and HFC?*

In order to quantify the significance of the differences, we want to know if those identified components persist in the top 25% of the measure. Secondly and more importantly, we want to know how critical these components are from the system's developers' assessments. RQ2 and RQ3 address these goals:

RQ2. *Do the identified defective components in RQ1 persist across releases in the same top 25% region?*

RQ3. *How critical to the application are the identified components in RQ1?*

To answer RQ1:

a) We use DPC as a reference measure and compute the set differences (1) DDC-DPC (2) SDC-DPC and (3) HFC-DPC. By this, we show how many defective components are identified differently by other metrics from DPC.
b) We use DDC as a reference metric and compute the set differences (1) SDC-DDC (2) HFC-DDC. By this, we show how many defective

components are identified differently by other measures from DDC.

c) We use SDC as a reference metric and compute the set difference (1) HFC-SDC. By this, we show how many defective components are identified differently by HFC metric from SDC.

In mathematical form, we compute the set difference D between two sets containing components measured by M_i and M_j as:

$$D = \begin{cases} M_i - M_j, & M_i \neq M_j \\ \emptyset, & M_i = M_j \end{cases}$$

Figure 3 shows the region of interest between M_i and M_j. The shaded portion captures components in M_i but not in M_j.

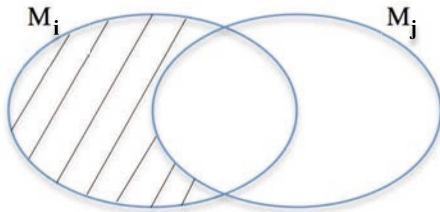


Fig. 3. Region of set difference between M_i and M_j i.e. $M_i - M_j$

To address the persistence of identified defective components, i.e. components in D (**RQ2**) across releases, we compute for each release the forward intersection of components in D in the current release with the components measured by M_i in all the future releases. In set form, we compute:

$$D^k_{persist} = \bigcup_{r=k+1}^{n} (D^k \cap M^r_i),$$
$$where\ r = release, \quad n = number\ of\ release$$
$$and\ k = current\ release, \quad k \in [1{:}n-1]$$

The number of defective components that persist across future releases is calculated as the cardinality of the set:

$$number(D^k_{persist}) = |D^k_{persist}|$$

We discuss how we address RQ3 in the result section under qualitative investigation.

C. Data collection

We have collected data for six releases of this application by using an in-house automated tool. We describe in each subsection the details of our approach: (1) to collect the data from the defect repository, (2) to map the class files to the defects, (3) to aggregate the defect counts at the package level (4) of computing the defect-fix effort for each class file and (5) of ranking components by severity of defect.

1) Defects collection from the defect tracking system (DTS)

We have collected defect data from the HP-QC DTS. A Defect repository gives typically a high level overview of a problem report. For example, typical attributes of the HP-QC defect tracking system (DTS) are the Defect ID, severity of the defect, the type of defect, date defect is detected, the module containing the defect, the version where defect is detected, and the date the defect is fixed.

Our first step is to determine the defects that affect each version of the system. In the HP-QC, we use “Detected in Version(s)”. A certain defect may affect multiple versions of a system. By this we mean persistent defects [3] that keep re-occurring and span several versions of a system. We include such defects in all the versions they affect. Next, we filtered out “Enhancement” and “Task” cases from the “Defect Type” field.

2) Method to map class files to defects

Version repository on the other hand is a configuration management system used by the developers to manage source code versions. The version system provides historical data about the actual class file that is changed and/or added as a result of corrective action (defect fixes), adaptive, preventive and perfective actions [39]. Thus, the SVN/CVS provides a detailed granularity level to know which source file(s) in the module(s) are changed to fix a reported defect. A common way to figure out what operation is performed on the source file is to look at the message field of the SVN commit. When developers provide this information with the bug number and/or useful keywords (e.g. bug or fix), it is possible to map the reported defect with the actual source file(s) [34, 40]. In some cases, not all bug commits in the version repository contain the bug number or useful

keyword in the message field. In the past, researchers have approached this situation by mapping from defect repository to the version repository [34, 41].

We have used both approaches to map defect from the HP-QC DTSs to the code changes. The defect fixed date allows us to map some of the untagged commits in the version system to the resolved bugs. Overall, we mapped an average of 71% for the six releases used for this study (see Table II). Table II lists in addition, the percentage of defect-fix efforts (correction efforts) recorded for the mapped defects.

TABLE II. % OF DEFECTS MAPPED FROM DTS TO SVN AND % OF FIX-EFFORTS RECORDED FOR THE MAPPED DATA

Version	%Defects	%Fix Efforts
4.2.4	71.4	80.0
4.2.2	83.3	100.0
4.1	85.7	97.2
4.0.1SP4	69.0	95.0
4.0.1SP2	64.0	100.0
4.0	51.7	75.7

3) Aggregating number of defects per class file

In a release, it is possible that multiple reported bugs be associated to one class file. The unique defect ID is thus appropriate to compute the number of defects fixes that affect a class file. From the mapped change data, we look up each class file and determine the total of defects per class file by counting the number of unique defect ID in this release.

4) Method of computing the defect-fix efforts of class files

In this system, each corrected defect has a recorded fix effort in person-hours in the DTS. The approach of collecting defect-fix effort for a resolved defect in this application has been discussed in section 2. Now, consider component ***x*** with **2** resolved defects, d_1 and d_2. Let us say that the recorded fix effort for d_1 is f_1 and the recorded fix effort for d_2 is f_2. If ***x*** shares d_1 with **n** number of other components and shares d_2 with **m** number of other components, what is the defect-fix effort for component ***x***? To answer this question, we have used equal-blame approach where we assume that the correction effort is equally distributed among the components affected by a defect. This is similar to the approach used by Rombach [42] to characterized corrective maintainability effort. Based on this assumption, we can compute the defect-fix effort for ***x*** as:

$$fix - effort(x) = \frac{f_1}{n} + \frac{f_2}{m}$$

We thus define the defect fix-effort for a component ***x*** that shares defects $d_1, d_2, d_3, ..., d_k$ with $n_1, n_2, n_3, ..., n_k$ number of components respectively, where $f_1, f_2, f_3, ..., f_k$ represent the fix efforts as:

$$Fix - effort(x) = \sum_{i=1}^{k} \frac{f_i}{n_i}$$

5) Ranking of components by severity

We want to be able to rank the components based on their number of the most severe defects. Unlike other studies [12-14] that have developed multiple models to predict two or three categories of defect severity, we can only devise an approach to have a single ranking of component based on its most severe defects. We describe this method [43] in this section since we believe other researchers and practitioners can find it useful.

The defect tracking system (DTS) of the system under study uses four values (***critical***, ***major***, ***average*** or ***minor***) to describe the severity of each recorded defect. The severity is determined based on the impact of the defect on the system and the business. We keep in mind that a component can have many defects and therefore contain different severity values (i.e. different severities distributed over a component). For instance, a component can have 3 defects in this order {Critical=1, Major=1, Average=0, Minor=1}. To rank according to severity requires that we make some transformation to give the highest weights to components according to their most severe defect.

We describe the transformation process we use for this purpose:

1. Given ***n*** number of components and ***m*** number of defect severity, we form an ***mxn*** matrix, where the column elements in the matrix stand for the severity values of a component in their order of severity.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

2. We form a new matrix B as follows; for each column element, starting from the first element, replace all elements below with zero if the element above is greater than zero.

$$\forall a_{i,j} \ if \ a_{i,j} > 0, a_{k,j} = 0, for \ k = i+1, \ldots, m \quad i \in [1{:}m-1]$$

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & \cdots & b_{1n} \\ b_{21} & b_{22} & b_{23} & \cdots & b_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{m1} & b_{m2} & b_{m3} & \cdots & b_{mn} \end{bmatrix}$$

3. Form a weight row vector W of ***1xm*** dimension containing the sum of the maximum element of each row below the kth row in *B*. The last column element in W is kept as 0:

$$w_k = \sum_{i=k+1}^{m} \max_{j=1,\cdots,n} (b_{i,j}), for \ k = 1, 2, \cdots, m-1$$

$$W = [w_1, w_2, \cdots, w_{m-1}, 0]$$

4. Form a new ***mxn*** matrix W_D, where *W* is the diagonal elements and all other elements are zero

$$W_D = \begin{bmatrix} w_1 & 0 & 0 & \cdots & 0 \\ 0 & w_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

5. Form matrix C by dividing each element in B by itself

$$C = \begin{bmatrix} \frac{b_{11}}{b_{11}} & \frac{b_{12}}{b_{12}} & \frac{b_{13}}{b_{13}} & \cdots & \frac{b_{1n}}{b_{1n}} \\ \frac{b_{21}}{b_{21}} & \frac{b_{22}}{b_{22}} & \frac{b_{23}}{b_{23}} & \cdots & \frac{b_{2n}}{b_{2n}} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{b_{m1}}{b_{m1}} & \frac{b_{m2}}{b_{m2}} & \frac{b_{m3}}{b_{m3}} & \cdots & \frac{b_{mn}}{b_{mn}} \end{bmatrix}$$

6. Finally, compute $D = W_D*C + B$

For example, with components; **c_1**: {Critical=2, Major=1, Average=0, Minor=1}, **c_2**: {Critical=0, Major=1, Average=3, Minor=0}, **c_3**: {Critical=0, Major=3, Average=0, Minor=0} and **c_4**: {Critical=0, Major=0, Average=0, Minor=1} gives matrix:

$$A = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 1 & 1 & 3 & 0 \\ 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

Following the transformation steps II-VI yields matrices:

$$B = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} W_D = \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$D = \begin{bmatrix} 6 & 0 & 0 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

From the matrix D's result, **c_1** has the highest weight of 6, followed by **c_3** with a weight 4, then **c_2** with a weight 2 and lastly **c_4** with a weight of 1.

Table III presents the grouping of the components based on their most severe defects. Each component in the groups is ranked according to the transformation procedure (not shown in the table). For example, release 4.2.4 contains components with no critical defect but 20 defective components contain at least 1 major severity defect, while 2 components contain at least 1 average severity defect and possibly minor defects as well but no critical or major severity defects. Lastly, 7 defective components have at least 1 minor defect but contain no critical, major or average severity defects. As we proceed from left to right, we can identify components purely on the scale of their highest severity defects. For instance, in release 4.2.2, from this table we identify 17 components with at least 1 critical defect and 20 components that contain major defects but contain no critical defects. We also know that 10 components contain average severity defects but have neither critical nor major defects. This is useful and beneficial and can complement the other approach [12-14] of classifying severe defects in software components.

TABLE III. GROUPING OF COMPONENTS BY THEIR MOST NUMBER OF SEVERE DEFECTS

Release	Critical	Major	Average	Minor
4.2.4	0	20	2	7
4.2.2	17	20	10	2
4.1	8	34	15	3
4.0.1SP4	16	35	18	0
4.0.1SP2	6	18	14	8
4.0	93	29	15	0

V. Results and Discussion

Table IV presents the results of the differences between the defect measures described in **RQ1**. Our goal is to measure the difference between the most used measure (number of defects and defect density) in defect model and other defect measures (severity of defect and correction effort). In columns 2 (DDC-DPC), 3 (SDC-DPC) and 4 (HFC-DPC), we present the percentage of defective components that are identified by DDC (**density**), SDC (**severity**) and HFC (**correction effort**) respectively and not by DPC (**number of defects**). Similarly, columns 5 – 6 lists the percentage of defective components identified by SDC and HFC and not by DDC. Lastly, column 7 presents the percentage of defective components identified by HFC and not by SDC.

The results demonstrate clearly that other defect measures such as severity and correction effort could identify other defective components that could neither be identified by number of defects nor by defect density at the top 25% cut-off point. From the results, DDC shows an average of 20.5%, SDC, 10.8% and HFC, 12.2% of defective components in all the six releases that are not identified by DPC. Furthermore, by using DDC as a reference measure, we show that SDC and HFC identified 18.7% and 19.9% defective components that are not identified by DDC respectively.

In Table V, we list the results of the persistence (**RQ2**) of the identified components in **RQ1** over releases. As shown, an average of 36.9% of the identified components by DDC persist across releases in the top 25% region. In a similar way, SDC and HFC show an average of 31.3% and 32.2% persistence respectively with DPC as a reference measure. With DDC as a reference measure, they show an average of 39.3% and 43.5% persistence respectively. Finally, using SDC as a reference measure, HFC gives an average of 29.4% persistence over releases. The results reveal the significance of the identified components by the other measures. These components remain defective in the top 25% region of the measure across releases.

TABLE IV. PERCENTAGE OF IDENTIFIED COMPONENTS BASED ON DIFFERENCES BETWEEN DEFECT MEASURES

Release	DDC-DPC	SDC-DPC	HFC-DPC	SDC-DDC	HFC-DDC	HFC-SDC
4.0	18.9	6.6	6.6	17.5	16.8	8.8
4.0.1SP2	21.7	10.9	17.4	21.7	23.9	21.7
4.0.1SP4	21.7	14.5	11.6	24.6	17.4	18.8
4.1	21.7	10.0	10.0	18.3	18.3	8.3
4.2.2	18.4	12.2	10.2	16.3	22.5	22.5
4.2.4	20.7	10.3	17.2	13.8	20.7	13.8
Average	20.5	10.8	12.2	18.7	19.9	15.7

The results in Table VI demonstrate the possibility to identify components that share other quality challenges together. For instance, 17.5% of the top most defective components also have very high correction effort. 14.6% of the defective components have high number of defects, high correction efforts and high number of severe defects (column- 4).

TABLE V. PERCENTAGE OF IDENTIFIED COMPONENTS THAT PERSIST ACROSS RELEASES BASED ON DIFFERENCES BETWEEN DEFECT MEASURES

Release	DDC-DPC	SDC-DPC	HFC-DPC	SDC-DDC	HFC-DDC	HFC-SDC
4.0	38.5	0.0	44.4	50.0	65.2	41.7
4.0.1SP2	40.0	40.0	75.0	40.0	81.8	70.0
4.0.1SP4	26.7	0.0	25.0	17.7	25.0	15.4
4.1	46.2	83.3	16.7	63.6	36.4	20.0
4.2.2	33.3	33.3	0.0	25.0	9.1	0.0
Average	36.9	31.3	32.2	39.3	43.5	29.4

TABLE VI. PERCENTAGE OF INTERSECTION OF DEFECT-PRONE COMPONENTS AMONG THE MEASURES

Release	HFC∩ DPC	SDC∩ DPC	HFC∩SDC∩ DPC	HFC∩SDC∩ DDC	All
4.0	17.5	18.2	14.6	3.6	2.9
4.0.1SP2	8.7	15.2	2.2	0.0	0.0
4.0.1SP4	11.6	10.1	4.3	0.0	0.0
4.1	15.0	15.0	10.0	6.7	3.3
4.2.2	14.3	12.2	2.0	0.0	0.0
4.2.4	6.9	13.8	6.9	3.4	0.0

A. Qualitative investigation

In order to address research question 3 (**RQ3**), we investigated the criticality of the defective components identified by the various measures from the system's developers. We devised a scale from 1 to 5, where 1 indicates the least critical and 5 indicates the most critical component. Subsequently, we asked the developers to rank each defective component in the order of their criticality to the system. We then formed four categorical values Critical (4 + 5), Major (3), Average (2) and Minor (1). Figure 4 shows the summary of the ranking of 117 defective components by the developers.

Among the defective components ranked by the developers, 15% are the most critical to the application, while 36% are of major importance, 39% are of average significance and 10% of the components are minor to the system.

In Table VII, we list the results of grouping the various components into the order of criticality as provided by the developers. In column 2, we list the average values of all identified defective components in all the six releases for each scale category. Column 3 lists the percentage of defective components identified by all the four measures for each scale category.

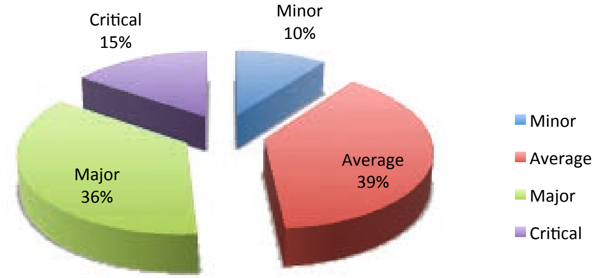


Fig. 4. Distribution of defective components ranked by developers

Columns 4-7 present the percentage of components identified by each measure for each scale category. Lastly, columns 8-13 present the percentage of the differences between the measures for each scale category.

As shown in the result, by using all the four measures at the 25% top most cut-off point, we could identify an average of 72% of components that are critical to the application. Also, we could identify 71% of major, 64% of average and all the minor components to this application. By using DPC, we could identify 43.4% of critical components followed by HFC that identifies 41.6% of critical components and then 37.7% by SDC and lastly 9.4% by DDC. We observe that, DDC identifies largely average and minor components to this system, whereas, other metrics could identify largely, critical and major components of the system.

However, the differences in the measures show that all the critical components identified by DDC are not identified by DPC (column 8). This implies that, although, measures based on defect density might identify mostly non-critical components, it is still an important measure in identifying critical components in a system. In this application, HFC gave the most result in identifying other critical components not captured by the popular metrics (DPC and DDC).

B. Findings

First, we show that the defect distribution of this application behaves like other systems in previous studies [7-9, 11, 15, 30, 31]. We discovered a 20-80 rule in the relationship between cumulative defective components and total defects that they

account for. In addition, we found a 30-67 rule in the relationship between cumulative defects and total correction efforts. Secondly, in this application, we found that other defect metrics HFC and SDC can identify significant and critical components in the application almost as much as DPC and much more than DDC. In addition, these metrics are able to identify other critical components not covered by DPC. Although, from the Pareto distribution plot (see Figure 1b), 25% of the defective components with the most number of defects in this system account for an average of about 85% of the total defects. Nevertheless, they could only identify 43.4% of critical components and 41.4% of major components in this application. The remaining 30% (approximate) of critical and major components are identified by other defect measures. In addition, the identified components by the other defect metrics persist in the top 25% region of these defect measures.

The results in this study support neither the largest-first prioritization approach nor the smallest-first[9] prioritization of components for testing activities. Since, we could only identify less than half of the critical defective components in this system by using either of the methods. However, it makes a strong case for an all-inclusive defect measures in order to discover critical components that are defect-prone in this system. For instance, focusing a defect model based on defect counts (largest-first) on the top 25% or 5% region could leave out quite many critical components to the application. Neither is it optimal to use defect density (smallest-first) only at the class-file level. As observed, many critical components of this system are not identified. Using either or both number of defects and defect density in defect prediction models is a common practice, however, this study provides a useful direction to identifying most important components.

Furthermore, our findings show that using this technique; it is possible to discover additional quality challenges of a component in a step-wise manner. For example, the intersection between DPC and HFC shows those components that share both high number of defects and high correction efforts. The intersection between HFC and DDC reveals smallest components that are defect dense with hard to fix defects. The intersection between DPC, HFC and SDC reveal components with high number of severe defects in addition to number of defects and high correction efforts. Thus, we can use this knowledge for effective decision support in assigning testing effort and improving the component's quality.

In defect prediction models, it makes sense to predict the number in addition to the severity, the density and the correction effort of defective components. Even if we have such defect prediction model that achieves 100% accuracy, it is still very useful and practical to group the predicted components in such a way that testing effort is appropriately focused. For instance, how severe is the defect in this component? How defect dense is the component? Will the correction effort to fix defects in this component be high? And does the component have many defects? A model based on only one measure leaves us with no further explanations about the predicted components. However, a possible shortfall of model that is based on multiple defect measures is the increase in the size of the components to be inspected due to the union of the models' outputs. Nevertheless, the intersections between the models' outputs present very useful outlook of the predicted components and could also minimize the number of components to be examined.

VI. Threats to validity

We have performed an analysis and evaluation of an industrial Smart Grid system using data from a single environment. Therefore, we cannot claim that this kind of pattern or related will be visible in other Smart Grid system or systems in other domains. As it is with most case studies, we cannot generalize these results across all systems. Further studies will be necessary to compare results across several systems.

[9] Note that the smallest-first approach in this study is based on defect density measure. In our study, this measure prioritized smallest components (in the top 25%. See Fig. 2).

TABLE VII. GROUPING OF IDENTIFIED COMPONENTS BY THEIR CRITICALITY TO THE SYSTEM

		% of Mean										
Scale	**Mean**	**All 4**	**DPC**	**DDC**	**SDC**	**HFC**	**DDC-DPC**	**SDC-DPC**	**HFC-DPC**	**SDC-DDC**	**HFC-DDC**	**HFC-SDC**
Critical	8.83	72	43.4	9.4	37.7	41.6	9.4	7.6	15.1	37.7	39.6	22.6
Major	21.33	71	41.4	14.9	32.8	36.7	10.2	8.6	14.9	28.1	32.0	21.9
Average	14.33	64	15.1	48.8	23.2	23.2	39.6	14.0	14.0	8.2	10.47	12.8
Minor	3.00	100	27.7	83.3	22.3	27.7	66.7	16.7	11.0	5.7	16.67	22.3

For this study, we have relied on the defects logged in the defect tracking systems of each application. Our approach of defect data extraction is similar to what other researchers have used in the past [34, 40, 41]. Nevertheless, common threats are whether defects logged in the DTS are accurately tagged in the respective code changes in the version systems. In addition, we cannot be sure if all defects are logged in the DTS. Also, there could be cases that the message log of the file that consists a change is not tagged with the bug numbers of the resolved defect. Furthermore, there could be cases of typographical error in the recording of the bug number in the version systems [41] and lastly, it is still possible that duplication will occur.

We have used an approach that equally assigns correction effort to a group of components that share a fixed defect for the reason that the system's developers did not track correction effort per component (class-file) and this is neither recorded in the DTS repository. This assumption that each component in the group has equal correction effort may lead to imprecision. For instance, the effort spent to fix a reported defect in two components may vary. Greater effort may be spent on one component than the other. We believe that the possibility of such imprecision cannot bias the result in a significant way.

The recording of defect severity in many defect-tracking systems has been argued to be subjective [19]. We cannot exclude the possibilities of subjective severity records in the DTS that we have used. However, most records point to the correct severity of defects in this application, therefore, we can rely on the quality of the data to a great degree.

VII. Conclusion

In this study, we have investigated the distribution differences of four defect measures in a non-trivial industrial Smart Grid system, namely; number of defects, defect density, defect correction effort and defect severity at the top 25% cut-off. We subsequently validated the results against developers' assessment of the criticality of the components to the under study system. At the 25% top most region, we found significant differences among the measures in identifying critical and very important components to this application. We discovered that the severity of defect and correction effort could as well identify significant number of critical components to the application. Also, the defect density, although, identified mainly non-critical components, nevertheless, could identify few distinct critical components that are not covered by the number of defects.

We found that focusing defect prediction models using the largest-first approach leave out significant number of critical and significant components to this application. Also, using the smallest-first approach is not optimal in the identification of critical defective components in this system. Lastly, we discovered that it is possible to identify additional quality challenges of a component by performing pairwise intersections of the measures.

This empirical study of a complex and industrial smart grid system brought out a useful perspective to our definition of component's defect-proneness. This study demonstrates the need for an all inclusive defect measures in the construction of defect prediction models for critical systems. It further shows that using several measures

concurrently can assist to prioritize testing activities for important and critical defective components in critical systems.

As future work, we propose to use the four measures in a single prediction model environment. We are interested to see how the output of models constructed with the four defect measures can improve the results of defect prediction models in a more useful and practical way, both in the industry and the academia.

Acknowledgment

The authors would like to thank Powel AS where this study is performed, for providing access to the software data and also the qualitative data. This study would not have been possible without the support we received.

References

[1] Myers, G.J., et al., *The art of software testing*. 2nd ed. 2004, Hoboken, N.J.: John Wiley & Sons. xv, 234 p.

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

[3] Li, Z.D., et al., *Characteristics of multiple-component defects and architectural hotspots: a large system case study.* Empirical Software Engineering, 2011. **16**(5): p. 667-702.

[4] Zhang, H., A. Nelson, and T. Menzies, *On the value of learning from defect dense components for software defect prediction*, in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering* 2010, ACM: Romania. p. 1-9.

[5] Ohlsson, M.C. and C. Wohlin, *Identification of green, yellow and red legacy components.* International Conference on Software Maintenance, Proceedings, 1998: p. 6-15.

[6] Ostrand, T.J., E.J. Weyuker, and R.M. Bell, *Where the bugs are*, in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis* 2004, ACM: Boston, Massachusetts, USA. p. 86-96.

[7] Ostrand, T. and E. Weyuker, *The distribution of faults in a large industrial software system.* ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, 2002. **27**: p. 55-64.

[8] Boehm, B. and V.R. Basili, *Software Defect Reduction Top 10 List.* Computer, 2001. **34**(1): p. 135-137.

[9] Ebert, C., et al., *Defect Detection and Quality Improvement*, in *Best Practices in Software Measurement*. 2005, Springer Berlin Heidelberg. p. 133-156.

[10] Adams, E.N., *Optimizing Preventive Service of Software Products.* IBM Journal of Research and Development, 1984. **28**(1): p. 2-14.

[11] Fenton, N.E. and N. Ohlsson, *Quantitative analysis of faults and failures in a complex software system.* IEEE Transactions on Software Engineering, 2000. **26**(8): p. 797-814.

[12] Shatnawi, R. and W. Li, *The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process.* Journal of Systems and Software, 2008. **81**(11): p. 1868-1882.

[13] Zhou, Y.M. and H.T. Leung, *Empirical analysis of object-oriented design metrics for predicting high and low severity faults.* IEEE Transactions on Software Engineering, 2006. **32**(10): p. 771-789.

[14] Singh, Y., A. Kaur, and R. Malhotra, *Empirical validation of object-oriented metrics for predicting fault proneness models.* Software Quality Journal, 2010. **18**(1): p. 3-35.

[15] Basili, V.R. and B.T. Perricone, *Software Errors and Complexity - an Empirical-Investigation.* Communications of the ACM, 1984. **27**(1): p. 42-52.

[16] Hatton, L., *Reexamining the fault density - Component size connection.* IEEE Software, 1997. **14**(2): p. 89-97.

[17] Moller, K.H. and D.J. Paulish. *An empirical investigation of software fault distribution. Proceedings., First International Software Metrics Symposium*. 1993.

[18] Fenton, N.E. and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*. 1998: PWS Publishing Co. 656.

[19] Hall, T., et al., *A Systematic Review of Fault Prediction Performance in Software Engineering.* IEEE Transactions on Software Engineering, 2011. **99**(PrePrints).

[20] Zhang, H.Y., *An Investigation of the Relationships between Lines of Code and Defects.* 2009 IEEE International Conference on Software Maintenance, Conference Proceedings, 2009: p. 274-283.

[21] Nagappan, N. and T. Ball, *Use of relative code churn measures to predict system defect density.* ICSE 05: 27th International Conference on Software Engineering, Proceedings, 2005: p. 284-292.

[22] Knab, P., M. Pinzger, and A. Bernstein, *Predicting defect densities in source code files with decision tree learners*, in *Proceedings of the 2006 international workshop on Mining software repositories*2006, ACM: Shanghai, China. p. 119-125.

[23] Koru, G., et al., *Testing the theory of relative defect proneness for closed-source software.* Empirical Software Engineering, 2010. **15**(6): p. 577-598.

[24] van Moll, J.H., et al., *The importance of life cycle modeling to defect detection and prevention.* 10th International Workshop on Software Technology and Engineering Practice, Proceedings, 2003: p. 144-155.

[25] Briand, L.C. and J. Wust, *Modeling development effort in object-oriented systems using design properties.* IEEE Transactions on Software Engineering, 2001. **27**(11): p. 963-986.

[26] Smith, J. and M. Phillips, *Interoperable Acquisition for Systems of Systems: The Challenges (CMU/SEI-2006-TN-034)*, 2006, Software Engineering Institute, Carnegie Mellon University: Pittsburgh, Pennsylvania.

[27] Alshayeb, M. and W. Li, *An empirical validation of object-oriented metrics in two different iterative software processes.* IEEE Transactions on Software Engineering, 2003. **29**(11): p. 1043-1049.

[28] Bocco, M.G., D.L. Moody, and M. Piattini, *Assessing the capability of internal metrics as early indicators of maintenance effort through experimentation.* Journal of Software Maintenance and Evolution-Research and Practice, 2005. **17**(3): p. 225-246.

[29] Perry, D.E. and W.M. Evangelist. *An Empirical Study of Software Interface Faults*. in *Proceedings of the*

Twentieth Annual Hawaii International Conference on Systems Sciences, January 1987, Volume II. 1987.

[30] Andersson, C. and P. Runeson, *A replicated quantitative analysis of fault distributions in complex software systems*. IEEE Transactions on Software Engineering, 2007. **33**(5): p. 273-286.

[31] Tihana Galinac, G., R. Per, and H. Darko, *A Second Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems*. IEEE Transactions on Software Engineering, 2012. **99**(PrePrints): p. 1-1.

[32] El Emam, K., et al., *The optimal class size for object-oriented software*. IEEE Transactions on Software Engineering, 2002. **28**(5): p. 494-509.

[33] Koru, A.G., et al., *Theory of relative defect proneness*. Empirical Software Engineering, 2008. **13**(5): p. 473-498.

[34] Schroeter, A., T. Zimmermann, and A. Zeller, *Predicting component failures at design time*, in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*2006, ACM: Rio de Janeiro, Brazil. p. 18-27.

[35] Li, W. and S. Henry, *Object-Oriented Metrics That Predict Maintainability*. Journal of Systems and Software, 1993. **23**(2): p. 111-122.

[36] NIST, *NIST Framework and Roadmap for Smart Grid Interoperability Standards Release 1.0*. NIST Special Publication, 2010. **1108**.

[37] Rahimi, F. and A. Ipakchi, *Demand Response as a Market Resource Under the Smart Grid Paradigm*. Smart Grid, IEEE Transactions on, 2010. **1**(1): p. 82-88.

[38] Oyetoyan, T.D., R. Conradi, and K. Sand. *Initial survey of Smart Grid activities in the Norwegian energy sector - use cases, industrial challenges and implications for research*. in *International Workshop on Software Engineering for the Smart Grid (SE4SG), 2012*.p. 34-37

[39] Gupta, A., et al., *An examination of change profiles in reusable and non-reusable software systems*. Journal of Software Maintenance and Evolution-Research and Practice, 2010. **22**(5): p. 359-380.

[40] S'liwerski, J., T. Zimmermann, and A. Zeller, *When do changes induce fixes?*, in *Proceedings of the 2005 international workshop on Mining software repositories* 2005, ACM: St. Louis, Missouri. p. 1-5.

[41] C'ubranic, D., *Project History as a Group Memory: Learning From the Past.* , in *PhD Thesis* 2004, University of British Columbia: Canada.

[42] Rombach, H.D., *A Controlled Experiment on the Impact of Software Structure on Maintainability*. IEEE Transactions on Software Engineering, 1987. **13**(3): p. 344-354.

[43] Oyetoyan, T.D., R. Conradi, and D.S. Cruzes. *Criticality of Defects in Cyclic Dependent Components*. in *13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM) 22-23 September 2013 (Accepted)*.

P2: A Study of Cyclic Dependencies on Defect Profile of Software Components

Published: In Journal of Systems and Software 86 (12), 2013, pp. 3162-3182.

A Study of Cyclic Dependencies on Defect Profile of Software Components

Tosin Daniel Oyetoyan[1], Daniela S. Cruzes[1,2], Reidar Conradi[1]

[1]Computer and Information Science Department, NTNU,
Trondheim, Norway
[2]SINTEF, Norway
tosindo@idi.ntnu.no, dcruzes@idi.ntnu.no, reidar.conradi@idi.ntnu.no

***Abstract* - (*Background*) Empirical evidence shows that dependency cycles among software components are pervasive in real-life software systems, although such cycles are known to be detrimental to software quality attributes such as understandability, testability, reusability, build-ability and maintainability. (*Research Goals*) Can the use of extended object-oriented metrics make us better understand the relationships among cyclic related components and their defect-proneness? (*Approach*) First, we extend such metrics to mine and classify software components into two groups - the cyclic and the non-cyclic ones. Next, we have performed an empirical study of six software applications. Using standard statistical tests on four different hypotheses, we have determined the significance of the defect profiles of both groups. (*Results*) Our results show that most defects and defective components are concentrated in cyclic-dependent components, either directly or indirectly. (*Discussion and Conclusion*) These results have important implications for software maintenance and system testing. By identifying the most defect-prone set in a software system, it is possible to effectively allocate testing resources in a cost efficient manner. Based on these results, we demonstrate how additional structural properties could be collected to understand component's defect proneness and aid decision process in refactoring defect-prone cyclic related components.**

***Keywords*-Cyclic Dependencies; Dependency cycle metrics; Software metrics; Empirical Study; Defect proneness; Software components**

1. Introduction

Dependency cycles among components have for long been regarded as symptoms of design decay that should be avoided in software systems (Briand et al., 2001a; Fowler, 2001; Lakos, 1996; Martin, 2000; Martin, 1996; Parnas, 1979). Lakos (1996) argues that cycles among components present a unique problem in terms of understandability since there is no reasonably starting point and no single part of the system can make sense on its own. For Parnas (1979), cycles, referred as "*loops in the Uses Relation*", can lead to a situation where nothing works in a system until everything works and that cycles prevent easy extension of software components. Similarly, Fowler (2001) says that cycles are problematic and that it makes system harder to understand "because you have to go around the cycle many times". In addition, Fowler argues that cycles inhibit the reusability of the class code. Martin (2000) states that cycles inhibit build-ability because to release a module, it should be tested and this implies that all dependent components must compile and build. In addition, many authors have proposed several strategies to optimize stubs in order to break cycles during integration testing, showing that cycles are detrimental to testability (Briand et al., 2001a; Hanh et al., 2001; Kung et al., 1996).

Despite numerous claims that cycles inhibit software quality attributes such as extensibility, understandability, testability, reusability, build-ability and maintainability (Fowler, 2001; Lakos, 1996; Martin, 2000; Parnas, 1979), evidence shows that cycles are widespread in real life software systems (Briand et al., 2001a; Hanh et al., 2001; Kung et al., 1996; Le Traon et al., 2000; Melton and Tempero, 2007a; Parnas, 1979; Tai and Daniels, 1997). The extent to which cycles are pervasive in software systems suggests that design advice (Fowler, 2001; Lakos, 1996; Martin, 2000; Parnas, 1979) regarding cyclic dependencies has not been followed. Melton and Tempero (2007a) argues that, either we have a lot of bad software out there, or the advice is not useful. Intuitively, we would expect that since cycles increase

coupling complexities among components (Briand et al., 1998; Briand et al., 2001b), then it should have a positive correlation with defects.

Furthermore, special analysis tools have been developed to facilitate refactoring of code modules, to detect and to warn developers about dependencies that are already cyclic or that could result into cycles. For instance, tools like, JDepend (http://clarkware.com/software/JDepend.html), NDepend (http://www.ndepend.com), JooJ (Melton and Tempero, 2007b), Dependometer (http://source.valtech.com/display/dpm/Dependometer), Classycle (http://classycle.sourceforge.net), Dependency Structural Matrix (Laval et al., 2009) are examples of existing tools and approaches that can be leveraged to dissuade developers from cyclic dependencies. However, in terms of defect-proneness and multiplicity of defects among components, are there undisclosed relationship between cycles and defect-proneness? If yes, we speculate that such evidence can reinforce the need to seek the benefits of such tools, if we want to further discourage dependency cycles among components.

The goal of the study presented in this paper is therefore to investigate the relationships between cyclic dependencies and defect profiles of cyclically dependent components. Although research efforts have focused on breaking cycles during integration testing (Briand et al., 2001a, 2003; Hanh et al., 2001; Kung et al., 1996; Le Traon et al., 2000; Tai and Daniels, 1997) collecting empirical evidence of cycles in software systems (Melton and Tempero, 2007a) and developing tools to break dependency cycles among components (Hautus, 2002; Melton and Tempero, 2007b; Sangal et al., 2005), there exist gaps regarding empirical evidence of defect proneness of cyclically related components.

We have performed an empirical study on six software systems to provide field evidence of actual cyclic dependencies in object-oriented systems and how such interconnection can be used to discover patterns of defect-prone components. We choose Apache Camel, Apache ActiveMQ, Apache Lucene, Eclipse and openPDC because they are open source and to compare between systems with different development technologies (i.e. Java and C#) and with different functionalities. Lastly, we choose a commercial application to understand if the cyclic effects are the same or different from open source domain.

Thus, the main contribution of this work is an empirical study of cycles and defect proneness of components caught in cycles. Furthermore, we propose metrics that identify cyclic dependency relations among components and use this information to understand the components defect proneness. For instance, it is not sufficient to only know components in cycle; we might be interested in the neighbors (Zimmerman et al., 2011) that depend directly on these cyclically connected components. Our findings will be useful for both practitioners and researchers in the collective effort to minimize defects in real life systems and minimize effort and resource usage in system testing. Also, this study points out additional software structural properties that can be focused for understanding components' defect proneness. Lastly, this effort is aimed to add significance to the need of collecting cycle metrics and focus on defect-prone cyclically dependent components for refactoring possibilities.

The rest of the work is organized as follows; Section 2 explores related literature to our work. Section 3 describes relationships and dependency concepts among software components and explains cyclic dependencies with examples. Furthermore, we define the terms used in the paper. In section 4, we detail our empirical design setup; In addition, we define the proposed metrics that are used in this paper. Additionally, we define our hypotheses and explain the statistical approaches that we employ. We likewise describe the case studies for this study and explain how our data is collected. In section 5, we present the results of our analysis and provide further discussions of the results and their implications. We draw out threats to the validity of the results in section 6. Finally, in section 7, we conclude the paper with additional notes on future work.

2. Related Work

This study concerns cycles among inter-related components and understanding the defect proneness of components in this cyclically related category by using extended object-oriented (OO) metrics. We thus discuss related work in two areas; the first part covers related work in cyclic dependencies among components and the second part covers related work in OO metrics and approach for determining component's defect proneness.

A. Cyclic dependencies

Over the years, researchers have conducted studies and provided design advice regarding cycles among components. In this section, we review previous work in dependency cycles among classes and packages and present empirical study of cyclic dependencies.

In terms of classes, Parnas (1979) identified "Uses relation" between two components and argues that the loops in the "Uses Relation" are detrimental to extensibility of a software system. Lakos (1996) provided extensive discussion about cyclic dependencies among C++ classes. The author claimed that cyclic physical dependencies among classes of C++ program inhibit understanding, testing and reuse. Other authors also claimed that cycles inhibit system understanding (Fowler, 2001), testing in isolation, integration testing (Briand et al., 2001a; Hanh et al., 2001; Hashim et al., 2005; Kung et al., 1996) and reuse (Martin, 1996). Cyclically connected components are mutually dependent, thus in terms of understanding any of the classes; it is necessary to understand all other classes in the cycle. Furthermore, to test a class in isolation is practically impossible when it is involved in a cycle with other classes (Lakos, 1996). In integration testing, cycles prevent the topological ordering of classes that can be used as a test order (Briand et al., 2001a, 2003; Hanh et al., 2001; Jungmayr, 2002; Kung et al., 1996; Melton and Tempero, 2007a), thereby inhibiting the testability of a system.

From package point of view, in many OO systems developed with language such as Java or C++, package represents a physical organization of software components (Knoernschild, 2012; Lakos, 1996). Packages are used to group classes that perform similar functions. As such they focus on manpower and they represent the granule of release (Martin, 1996). Applications are usually a network of interrelated packages and the work to manage, test, build and release those packages is non-trivial (Martin, 1996). When cycles are formed at the package level, it seriously affects manpower since software engineers working on individual packages need to build with every other dependent package before they can release their package. Cycles among packages have thus been claimed to be detrimental to understandability (Fowler, 2001), production (Lakos, 1996; Martin, 1996), marketing (Lakos, 1996), development (Lakos, 1996; Martin, 1996), usability (Lakos, 1996; Martin, 1996) and reliability (Lakos, 1996).

Although, it has been stated (Briand et al., 2001a; Hashim et al., 2005; Kung et al., 1996; Lakos, 1996) and implied (Jungmayr, 2002; Martin, 1996) that cycles are pervasive in real-life software systems. However, it appears that only Melton and Tempero (2007a) have performed an elaborate empirical study of cycles on many software systems at the class level. Melton and Tempero carried out an empirical study of 78 Java applications and employed three "Uses Relation" types; "USES", "USES-IN-SIZE" and "USES-IN-THE-INTERFACE" as stated in Lakos (1996) to describe cyclically connected components within the applications. The result shows that almost all the 78 Java applications contain large and complex cyclic structures among their classes. This study is conducted on open source Java applications showing that further work is still needed to investigate other domains and programming languages before any generalization can be made.

B. OO Metrics and defect proneness of components

Object Oriented metrics have been widely used to indicate defect proneness of components. Basili et al. (1996) validated a set of OO metrics proposed by Chidamber and Kemerer (1994). Among these metrics, coupling between object classes (CBO) and response for class (RFC), are shown to correlate significantly to

a component's defect. Briand et al. (1998); (2001b) have also conducted several studies that showed CBO, and especially import and method invocation coupling to be important properties when building an OO quality model.

Additionally, Marinescu (2001) identified code smells (GodClass, ShortgunSurgery, GodPackage, etc.) by defining threshold values and rules based on code metrics. The author showed that code smells violated good design principles of low coupling, high cohesion, manageable complexity, proper data abstraction and standard component reuse (Capretz and Capretz, 1996; Capretz and Lee, 1992; Coad and Yourdon, 1991). Empirical study by Olbrich et al. (2009) on two open source applications further showed that different phases could be identified during the evolution of code smells. In addition, they pointed out that code smell infected components display a different change behavior.

Actual dependencies of a component have also been employed to indicate the defect proneness. For instance, Schroeter et al. (2006) demonstrated that imported components in the eclipse software could predict the defect proneness of their dependent components. Further, we have recently validated this approach on a Smart Grid application (Oyetoyan et al., 2012).

Social network analysis has been explored for defect prediction. Zimmermann and Nagappan (2008) showed that there is significant correlation between their proposed dependency graph metrics and the number of defects in the graph related components. Their results from a study of Microsoft Windows Server 2003 demonstrated that a network-based model could predict the number of defects and could identify critical binaries missed by complexity models. In fact, in their previous study of similar system (Zimmermann and Nagappan, 2007) they made an implicit observation that binaries in dependency cycle have on average twice as many defects as those binaries not in cycle. In another related study of Microsoft Vista and Eclipse, Zimmerman et al. (2011) showed that the properties of a component's neighbor such as size, code churn; complexity, test coverage and organizational structure can influence the quality of the component. However, Weyuker et al. (2008) disputed the effect of the number of developers' impact on defect-proneness of components. An elaborate empirical study by the authors concluded that the number of developers is not a major factor that could contribute to a component's defect-proneness.

Yutao et al. (2010) have proposed a multiple-dependency metric, *m* based on network analysis. The metric measures the degree of reusability of a component (incoming dependencies) as well as its direct and indirect coupling (reachable set). In the open source systems they analyzed, the authors found that fewer classes have high *m* value and that correlations exist (though weak) with WMC and LCOM (Lack of Cohesion of Method). Indicating that *m* may be used as a statistical indicator for defect-prone classes identified by WMC or LCOM.

Related Metrics

The following metrics regarding coupling between objects in object-oriented systems are of interest for our work:

- CBO: The coupling between object classes (CBO) shows the number of other classes that are directly coupled to the class (Basili et al., 1996; Briand et al., 1998; Briand et al., 1999; Briand et al., 2001b; Chidamber and Kemerer, 1994).
- RFC: Response for class (RFC) indicates the set of all methods that can potentially be invoked in response to a message received by the object of the class (Basili et al., 1996; Briand et al., 1998; Briand et al., 1999; Briand et al., 2001b; Chidamber and Kemerer, 1994). It considers both direct and indirect connections. However, it does not show if the connection is cyclic or not.
- CyclicClassCoupling (Nagappan and Bhat, 2007; Zimmermann and Nagappan, 2008): This metric counts the number of direct cyclic connections between two classes. For instance, C_1 depends on C_2 and C_2 depends on C_1 (see Figure 3d). However, this metric only deal with direct cyclic coupling between two classes and does not consider transitive relationship where components can become cyclic indirectly (see Figure 3a).

- dwReach (Nagappan and Bhat, 2007; Zimmermann and Nagappan, 2008): This metric shows the number of components that can reach another component with the distance weighted by the number of steps.

The above metrics measure the number of connections between objects and cannot be used for our purpose. Our proposed cyclic dependency metrics in section IV contains metrics that simply flag a component when it has a cyclic relationship. Although our work extends these previous studies, it differs in focus. We provide the first empirical study of defect proneness of cyclic dependent components. From this study, we are able to point out additional structural complexities that can be focused for defect tracing and testing activities. Additionally, our work indicates coupling property that can be useful and assessed for building quality models.

3. Relationships and dependency concepts

In this section we focus on explaining various types of relationships that exist among components when modeling with UML. Furthermore, we describe how a UML diagram is translated to an Object Relation Diagram (ORD) when the analysis concerns a client to server or supplier relationships. In addition, we present the definitions of an ORD that are appropriate for our study. In addition, we explain the component relationship level at which dependency is stronger and how this influences our choice of analysis decision. Lastly, we present and explain cyclic dependencies with examples and provide definitions that are necessary for our metrics.

A. Relationships: From UML to Dependency Graph

In software designs, class interactions are modeled based on the various relationships that exist among them (Bennett et al., 1999; Souza and Wills, 1999). If we concern ourselves with UML modeling, these relationships among the various classes can be modeled in UML[10] as association (uni-directional, bi-directional or reflexive), aggregation, composition, generalization (inheritance) and realization. An association relationship indicates a structural relationship between two class objects. The reasons for this relationship and the rules that govern the relationship are specified in an association relationship. Aggregation typifies a "whole-part" (*has-a*) relationship, where a class is modeled as a part of an aggregate class (whole). The "part" can exist independently of the "whole" and is therefore not destroyed when the lifecycle of the aggregate class ends. A composition relationship (*part-of*) is a special type of aggregation where the "part" class can no longer exist once the "whole" class lifecycle ends. Generalization illustrates an inheritance (*is-a*) relationship between a child class and its parent (super or base class). Realization relationship exists when a class implements (realizes) the behavior of another class.

An Object Relation Diagram (ORD) has been widely used to describe components and their relationships (Briand et al., 2001a, 2003; Hanh et al., 2001; Kung et al., 1996; Le Traon et al., 2000; Tai and Daniels, 1997). The term component in this study is used to represents a class or a package. A component *X* is said to have dependency on another component *Y* if *X* requires *Y* to compile or function correctly (Jungmayr, 2002). Three relationship types are described in ORD, that is: inheritance, ***I***, association, ***As*** and aggregation, ***Ag***. Where ***I***, is used for both inheritance and realization relationships, and ***Ag*** is used to represent both composition and aggregation relationships[11], while ***As*** maps to other cases of dependencies and associations. When an ORD is represented as a dependency graph, the relationship labels are usually ignored. Figures 1b and 1c show two design diagrams of the implemented code in Figure 1a. In Figure 1b, the UML relationship diagram shows a reflexive association relationship between class B and itself. In other words, an instance of class B can be related to another instance of B. A generalization relationship exists between classes A and

[10] http://www.omg.org/spec/UML/2.3/Superstructure/

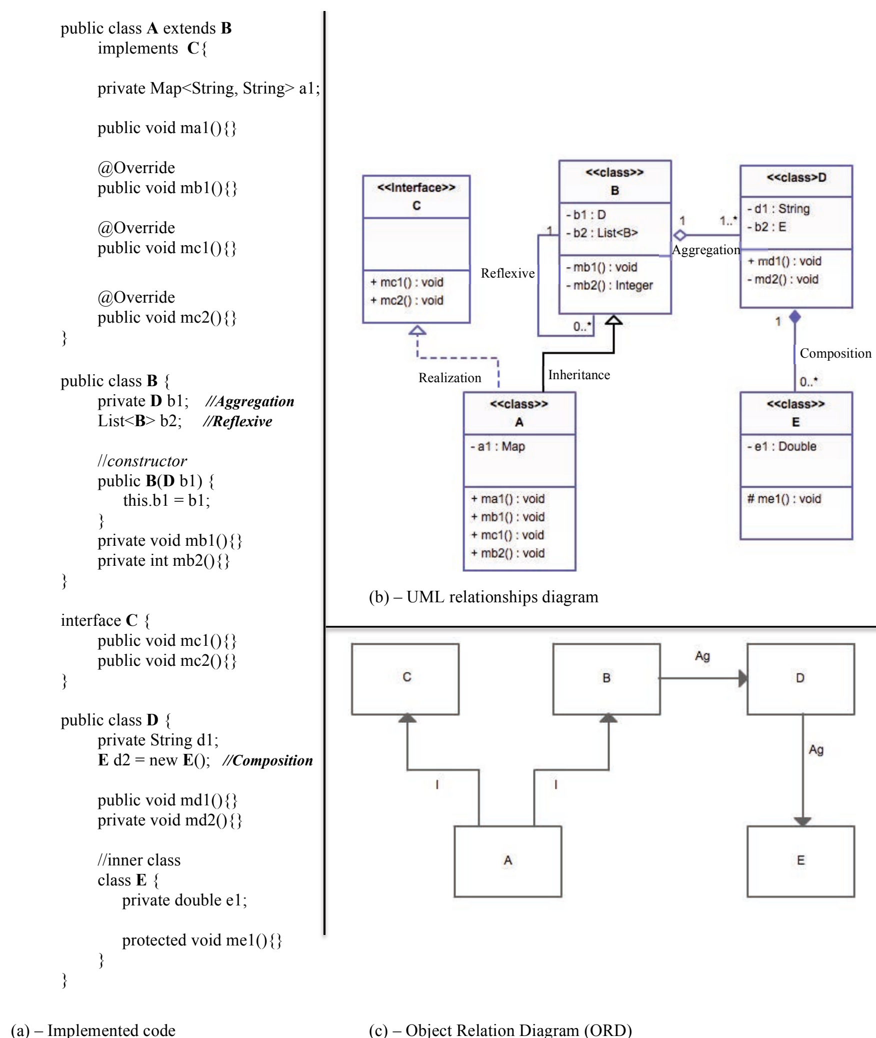
[11] Briand et al. (2001a) maps only composition relationship to *Ag* with the claim that compositions have a lifetime constraints between the whole and the parts and thus represent tight coupling. Whereas, *As* maps to simple aggregation (a type that is considered as a special type of association in UML and does not denote strong coupling), dependencies and associations.

B, since B is the super class of A. A Realization relationship exists between classes A and C since class A implements the behavior or contracts specified in class C. An aggregation relationship is shown between classes B and D and lastly, a composition relationship exist between classes D and E. Class E cannot exist when D's lifecycle ends.

For formal representation of an ORD, we borrow two definitions from Kung et al. (1996) and state these as follows:

- **Definition 1.** An edge labeled digraph G = (V, L, E) is a directed graph, where $V = \{V_1, ..., V_n\}$ is a finite set of nodes, $L = \{L_1, ..., L_k\}$ is a finite set of labels, and $E \subseteq V \times V \times L$ is the set of labeled edges.
- **Definition 2.** The ORD for an OO program P is an edge-labeled directed graph (digraph) ORD = (V, L, E), where V is the set of nodes representing the object classes in P, $L = \{I, Ag, As\}$ is the set of edge labels, and $E = E_I \cup E_{Ag} \cup E_{AS}$ is the set of edges.

Applying these definitions to the ORD presented in Figure 1c gives $V = \{A, B, C, D, E\}$, $L = \{I, Ag, As\}$ and $E = \{E_{A\text{-}B}, E_{A\text{-}C}, E_{B\text{-}D}, E_{D\text{-}E}\}$, where $E_{X\text{-}Y}$ denotes an edge that connects node X to node Y in the direction of Y. For the purpose of this paper, we ignore the edge labels L and concern ourselves with the set of nodes and the set of edges. Furthermore, in the data collection section (Section 4.4.4), we describe how the set of edges are determined for each class node and each package node.



(a) – Implemented code (c) – Object Relation Diagram (ORD)

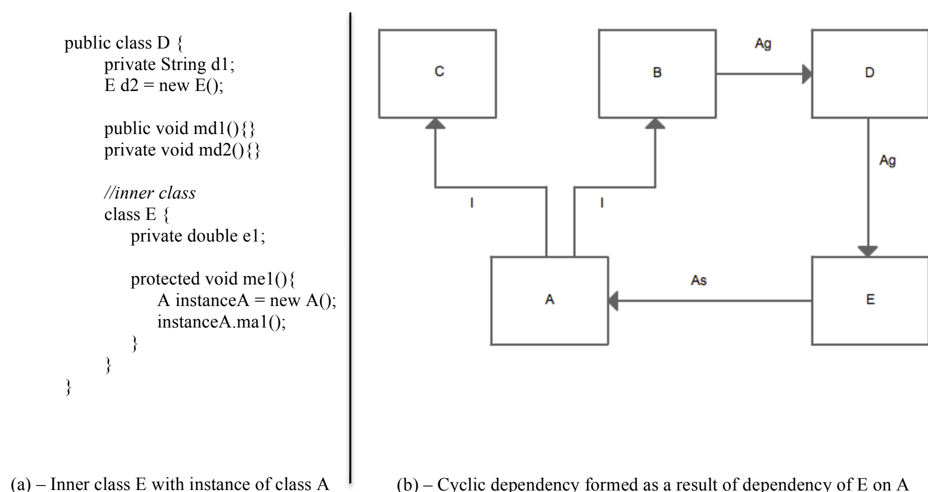
Figure 1(a – c) – Representation of component relationships with UML and ORD

B. Physical or Logical Dependency

Lakos (1996) differentiated between logical relationships and physical dependencies and states that a physical dependency concerns dependency of the physical entities of a software unit and for instance requires the use of an "#include" directive in C++ in another components and that this type is stronger. Physical design decisions impact on deploy-ability, reusability and maintainability. A physical dependency implies that the dependent component[12] requires the dependee component[13] in order to compile and link. In Java systems, we can imply that to mean relationships between a .java and another .java files. In C#, this is equivalent to a .cs files relationships. Physical dependency thus requires that a physical class file have knowledge of another physical class file. A file can enclose multiple classes, including nested classes and we can define the logical relationships among these classes, for example, the case of class D and E in Figure 1. A physical dependency relationship is formed when there is dependency with classes in another physical file. However, we can infer or imply physical dependencies from logical relationships among components (Lakos, 1996). We concern ourselves in this study with dependency at physical level, that is, both files (top-level classes) and packages as described above since we can infer strong dependencies from it.

C. Cyclic dependencies

In Figure 1a, let us say that during system evolution, class **E** for some reasons is required to pass a message to class **A**. We introduce an instance variable of class **A** implemented locally in method ***me1()*** of class **E** (see Figure 2a). Figure 2b shows the dependency relationship that now occurs as a result of the new dependency of class **E** on class **A**. What we have is a cyclic dependency between classes B, D, E and A. In graph theory, this type is referred as strongly connected component (SCC) (Jungmayr, 2002). If we assume that class D resides in another package or another file, then a strong physical cyclic dependency exist among the connected classes.



(a) – Inner class E with instance of class A (b) – Cyclic dependency formed as a result of dependency of E on A

Figure 2 – Cycles and representation with Object Relation Diagram (ORD)

Hypothetical Example

As depicted in Figure 3a, cyclic dependency is formed when components depend on one another in a circular manner. For example, B depends on A, C depends on B, D depends on C and A in turn depends on D. In this network diagram, 2 cyclical paths exist: (i) A-D-C-B-A (ii) A-D-C-F-E-A. This relationship covers both direct and indirect connection between components. Cyclic

[12] A component that depends on another component is called a dependent component

[13] A component that is dependent upon by other component is called dependee component

relationships increase coupling complexities and have the potential to propagate defects in a network (Abreu and Melo, 1996). A hypothetical case as depicted in Figure 3a-b demonstrates such effect. From Figure 3a, assume that component **I** contains some defects. We can further assume that the rest of the components **A – H** will have a certain probability to inherit the defect from **I**, since they are directly and indirectly dependent on **I**. To reduce the likelihood of defect propagation e.g. in Figure 3b, let us say that, a new component **J** is created so that components **D** and **C** depend on **J** directly thereby breaking the cyclic effect. By performing such a refactoring, the effect of possible defect propagation is reduced to only component **G**.

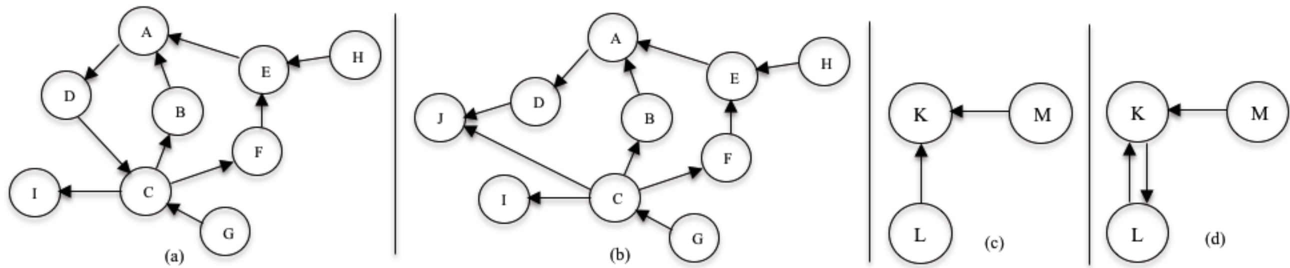


Figure 3(a - d) - Cyclic Dependencies and propagation effect on components in a software network

For the purpose of this paper, we define some of the terms used henceforth: Assume a component c ∈ System P then:

D1. **Component's Children**: Components that are directly and transitively dependent on c. E.g. in Figure 3(a), All the components except component **I** are directly or transitively dependent on **A**. Components **G**, and **H** have no children. We use **TChildren** for both direct and transitive children and **DChildren** for direct children. For example, **DChildren(A)** = {B, E}.

D2. **Component's Parent**: All components that c is both directly and transitively dependent upon. We use **TParent** for both direct and transitive parents and **DParent** for direct parents. For instance, **TParent(G)** = {A, B, C, D, E, F, I} and **DParent(G)** = {C}.

D3. **Component In-Cycle**: Component c is said to be in cycle, if it has at least one parent that is the same as one of its children. E.g. **B** is in cycle because its parent **A** is also one of its children.

D4. **Component Depend on In-Cycle Component**: Component c is said to depend on another in-cycle component if at least one of its direct parents is in cycle. **G** and **H** are examples of components that depend on **In-Cycle** components **C** and **E** respectively.

D5. **Component's Minimum Number of Cycle**: The minimum number of cycle that component c is involved with is defined as the sum of the number of its direct children that are in-Cycle and the number of its direct parents that are in-Cycle minus one. For instance, **DChildren(A)** = $\{B_{in\text{-}cycle}, E_{in\text{-}cycle}\}$, and **DParent(A)** = $\{D_{in\text{-}cycle}\}$. Therefore, **minCycle(A)** = 3-1 = 2.

D6. **Associated Defect**: Two components have associated defect if a specific defect affect both components. We use defect ID to track associated defect between components.

D7. **Cyclic Propagated Defect**: Consider component **M** that directly depend on **K** (Figure 3c). Let us say that **L** contains some defects. These defects from **L** cannot be propagated to **M**. However, if **M** forms a cycle with **L** by depending on it (Figure 3d), we can thus infer that the defect from **L** may be propagated to **M**.

4. Empirical Design

Our goal in this work is to explore the defect profiles of cyclic dependent components in a system. As explained in (Fowler, 2001; Lakos, 1996; Martin, 2000; Martin, 1996), cyclic dependencies are

better studied at physical design levels such as the source file (compilation unit) and package levels, since physical dependencies are formed at such levels. In addition, previous empirical studies (Melton and Tempero, 2007a) on cyclic dependency have performed analysis at the file levels. Furthermore, when developers resolve defects, they usually log the changes at the file level and thus have file to defect mapping. Based on the above reasons, we identify relationships and dependencies at the compilation units (top-level classes for Java) and at the package level. We perform our evaluation in three ways: First, we propose a set of metrics built around cyclic dependency relationships. Second, we use our proposed metrics to mine software components and classify them into two groups, "Cyclic" and "Non-Cyclic". Third, we statistically evaluate data from cyclic-related components and non-cyclic related components to determine their defect profiles.

A. Proposed Metrics for our study

We describe as follows the cyclic metrics and notations for our study. Consider a set of components, C in an object-oriented system. For each component $c \in C$:

1. **Component In-Cycle**:
   inCycle: boolean
   $\exists p : p \subseteq (TParent(c) \wedge TChildren(c))$
   $\{\forall c.inCycle(c) \leftrightarrow p \neq \emptyset\}$
   *Where inCycle(**c**) denotes **c** to be in a cyclic dependency*

   Example 1. (Figure 3a):
   if c = A
   TParent(A) = {D, C, I, B, F, E} and
   TChildren(A) = {B, C, D, E, F, G, H}
   p = {B, C, D, E, F}
   Since $p \neq \emptyset \therefore inCycle(A) \Rightarrow True$

2. **Depend On Cycle:**
   depOnCycle: boolean
   $\exists \mathbf{x} : \mathbf{x} \in DParent(c)$
   $\{\forall c.depOnCycle(c) \leftrightarrow (\neg inCycle(c) \wedge inCycle(\mathbf{x}))\}$
   *Where depOnCycle(c) denotes c depends on inCycle component **x** that is a direct parent of c.*

   Example 2. (Figure 3a):
   if c = H
   inCycle(H) = False,
   But DParent(H) = {E} and inCycle(E) = True
   $\therefore depOnCycle(A) = \neg inCycle(A) \wedge inCycle(E) = True$

3. **Minimum Number of Cycles:**
   minCycle: Integer
   $\exists p : p \subseteq (TParent(c) \wedge TChildren(c))$
   $minCycle(c) = (|p \wedge DParent(c)| + |p \wedge DChildren(c)|) - 1$

   Example 3. (Figure 3a):
   From Example 1, if c = A
   p = {B, C, D, E, F}
   DChildren(A) = {B, E}
   DParent(A) = {D}
   $\therefore minCycle(A) = (|\{D\}| + |\{B, E\}|) - 1 = 2$

B. Hypotheses

The main goal of this study is to investigate the impact of cyclic dependencies among components regarding their defect proneness. To verify the conjecture that the most defects are concentrated in the components with cyclic dependencies, we define our hypotheses as follows:

***H_A**: Cyclic dependent components are more defect-prone than non-cyclic dependent components.*

To evaluate this hypothesis, we further define two sub hypotheses:

- ***H_{A1}**: The number of defective components in cyclic relationships is significantly higher than non-cyclic defective components.*
  With this hypothesis, we seek to establish the group with the higher number of defective components. In addition, the number of defective components in each group allows us to measure the recall value that shows the ratio of defective components in each group to the total number of defective components in the system.

- ***H_{A2}**: The proportion (ratio) of defective components in cyclic group is significantly higher than the proportion of defective components in non-cyclic group.*
  Using this hypothesis, we aim to establish the group with higher defect propagation among their components. It is not sufficient to know the number of defective components in each group. We are also interested in knowing if defects spread in a group more than the other. The proportion data gives us idea about the concentration of defects in each group. This measures the ratio of defective components to non-defective components within each group and allows us to identify the group with relatively higher number of defective components.

***H_{B1}**: The actual number of defects in cyclic dependent components is higher than non-cyclic dependent components.*

***H_{B2}**: Defect density in cyclic dependent group is higher than non-cyclic dependent components.*

Defects can be associated in nature, that is, a defect may propagate to a number of components. Therefore, in terms of number of defects, a component may have many defects and many components may have very few defects. If H_A is true, H_{B1}, therefore, allows us to verify if the components in cyclic group are defective due to more actual defects than the non-cyclic group. If this hypothesis is not rejected, we can conjecture that cycles probably trigger more defects.

Defect density takes the size of the components into account. We compute defect density as the number of defects in each group per the source line of code in the group. We seek to know if there is an implicit relationship between size and defect in each group.

C. Statistical Analysis

For this study, we identify cyclic group and non-cyclic group from each system. In Table 1, we use **C** to represent all cyclic-related group, **inC** for group with components that are only in-cycle and **NC** for "Non-Cyclic" group. We have performed analysis both at the class and package levels. A cyclic group consists of all components (classes or packages) that are flagged to be (1) in-cycle and (2) cyclic-related, i.e. both in-cycle and also directly dependent on in-cycle components. If we use our Figure 3a, then **inC** and **C** groups for this hypothetical example consists of components {A, B, C, D, E, F} and {A, B, C, D, E, F, G, H} respectively, and non-cyclic (**NC**) group consists of only {I}. We use Table 1 to present how the data for each category and for each hypothesis is computed. For each system, we collect both cyclic dependency data and defect data for multiple versions (Table 2). For each version and each group, we determine the number of components, the number of defects, the number of defective components and the source line of code. Subsequently, we compute the proportion data and the defect density per group as shown in Figure 4.

Table 1 – Data computation for groups and hypotheses

Group	#Component	#Defect	#Defective Component	#Non-Defective Component	SLOC
inC	inC_N	D_{inC}	inC_D	$inC-inC_D$	inC_{SLOC}
C	C_N	D_C	C_D	$C-C_D$	C_{SLOC}
NC	NC_N	D_{NC}	NC_D	$NC-NC_D$	NC_{SLOC}

Hypothesis H_{A1}: Number of defective components
NumDefective (X) = X_D

Hypothesis H_{A2}: Proportion of defective components
Proportion (X) = X_D/X

Where **X** can be **inC**, **C** or **NC**

Hypothesis H_{B1}: Actual defect
ActualDefect (X) = D_X

Hypothesis H_{B2}: Defect density
Defect Density (X) = D_X/X_{SLOC}

Figure 4 – Computed data per group and for the hypotheses

The next step is to determine what statistical approach is appropriate to test our hypotheses, either a t-test or non-parametric test. We initially perform statistical test to determine if our data sample is from a normally distributed population. For this, we use the Shapiro-Wilk normality test. If the data is normally distributed, we employ a t-test; otherwise we use a non-parametric statistical approach (Fenton and Pfleeger, 1997) such as Wilcoxon signed rank test (see Appendix B).

Lastly, we test the difference in mean between both groups for significant difference that is greater than zero. Four categories are identified for both groups based on our hypotheses:

I. Number of defective components in each group
II. Proportion of defective components in each group
III. Actual defect counts produced in each group
IV. Defect density for each group measured as actual defect in each group per source lines of code in the group

For these four categories, we test the hypothesis (1-tailed significance test):

- H_0: $\mu_C \leq \mu_{NC}$ The mean of cyclic group is significantly less than or equal to the mean of non-cyclic group
- H_1: $\mu_C > \mu_{NC}$ The mean of cyclic group is significantly higher than the mean of non-cyclic group

D. Data Collection

We have performed a study on two Smart Grid systems, an open sourced (openPDC) [14] and a commercial application (commApp) developed with C#. In addition, we choose an integrated development environment (Eclipse)[15], a search engine (Apache Lucene)[16], an integration framework (Apache-Camel) [17] and a messaging and integration pattern server (Apache-ActiveMQ) [18], all developed with java. We have selected very active projects from the open source community and we also considered projects that have different functionalities and different development languages.

Apache Camel is an integration framework that can serve as a routine and mediation engine between applications. ActiveMQ is a messaging server with the capability to handle various integration patterns. OpenPDC is a medium-sized Smart Grid open source software (OSS) named openPDC, supported by the Tennessee Valley Authority (TVA). The solution is developed using the .NET

[14] http://openpdc.codeplex.com/

[15] http://archive.eclipse.org/eclipse/downloads/index.php
[16] http://lucene.apache.org/core/index.html

[17] http://camel.apache.org/index.html
[18] http://activemq.apache.org/index.html

Framework and mainly with the C# programming language. The openPDC is a phasor data concentrator software that is designed to process real time data for user-defined actions and for archival purpose.

The commercial application shares the same Smart Grid domain with openPDC. It is a distribution management system designed to allow for monitoring and planning of Grid operations. It provides real-time operational support by continuously receiving status data from the power grid.

Eclipse is a popular open source integrated development environment (IDE), while Lucene is a high-performance search engine. Table 2 details some properties of the applications we have used for this study.

Table 2 – Properties of selected applications

System	Language	#Developers[19]	Domain	License	Bug Tracker	Age	Module	Versions Analyzed
Apache-Camel	Java	34	Routing and Mediation Engine	Open	JIRA	5	CORE	2.10.2, 2.10.1, 2.10.0, 2.9.2, 2.9.1, 2.9.0
Apache-ActiveMQ	Java	24	Messaging and Enterprise Integration Pattern Server	Open	JIRA	6	CORE	5.7.0, 5.6.0, 5.5.1, 5.5.0, 5.4.2, 5.4.1
Apache-Lucene	Java	31	Search Engine	Open	JIRA	7	CORE	4.0, 3.6, 3.5, 3.4, 3.3, 3.2
Eclipse	Java		IDE	Open	Bugzilla		All	3.0, 2.1, 2.0
commApp	C#	28	Smart Grid	Commercial	HP-Quality Center	6	All	4.2.4, 4.2.2, 4.1, 4.0.1SP4, 4.0.1SP2, 4.0
openPDC	C#	13	Smart Grid	Open	CodePlex	3	All	1.5, 1.4SP2, 1.4

1) Defects collection from the defect tracking system (DTS)

We have collected defect data from three different DTSs. Some DTSs contain more details than the others and some are more difficult to filter. Defect repository gives typically a high level overview of a problem report. For example, typical attributes of the HP-QC defect tracking system (QC-DTS) are the Defect ID, severity of the defect, the type of defect, date defect is detected, the module containing the defect, the version where defect is detected, and the date the defect is fixed. These fields are similar to the Apache JIRA and CodePlex DTSs.

Our first step is to determine the bugs that affect each version of the system. In Apache JIRA DTS, we readily use the "Affects Version" field to filter all bugs that affect a particular version of the system. For CodePlex, we use the "RELEASE" field and for HP-QC, we use "Detected in Version(s)". A certain defect may affect multiple versions of a system. By this we mean "hotspot" defects (Li et al., 2011) that keep re-occurring and span several versions of a system. We include these defects in all the versions they affect. Next, we filtered out "duplicate", "Not a problem", and "Invalid" cases from the resolution field. The Eclipse dataset that we use in this paper has been mapped in previous study (Zimmermann et al., 2007).

2) Method of mapping class files to defects

Version repository on the other hand is a configuration management system used by the developers to manage source code versions. The version system provides historical data about the actual file that is changed and/or added as a result of corrective action (defect fixes), adaptive, preventive and perfective actions (Gupta et al., 2010). Thus, the SVN/CVS provides a detailed granularity level to know which source file(s) in the module(s) are changed to fix a reported bug. A common way to figure out what operation is performed on the source file is to look at the message field of the SVN

[19] #Developers as used in this study represent all committers to the SVN

commit. When developers provide this information with the bug number and/or useful keywords (e.g. bug or fix), it is possible to map the reported defect with the actual source file that is modified to fix it (S'liwerski et al., 2005; Schroeter et al., 2006). In some cases, not all bug commits in the version repository contain the bug number or useful keyword in the message field. In the past, researchers have approached this situation by mapping from defect repository to the version repository (C'ubranic, 2004; Schroeter et al., 2006).

We have used both approaches to map defect from JIRA and HP-QC DTSs to the code changes. The resolution date allows us to map some of the untagged commits in the version system to the resolved bugs. The second approach of mapping from defect repository to code repository is found suitable for CodePlex DTS. None of the bug is tagged in the commit log of the openPDC application. The observed style of developers in this community is to include the SVN revision number of the corrected bug in the comment field of the defect repository (e.g. “resolved with change set 79160”). We use the revision numbers from the comment field to identify class files that are changed because of bug fix. Overall, we mapped an average of 89.5% for Apache-Lucene, 90.1% for Apache-Camel, 75.7% for Apache-ActiveMQ, 71.3% for commApp and 81.4% for openPDC.

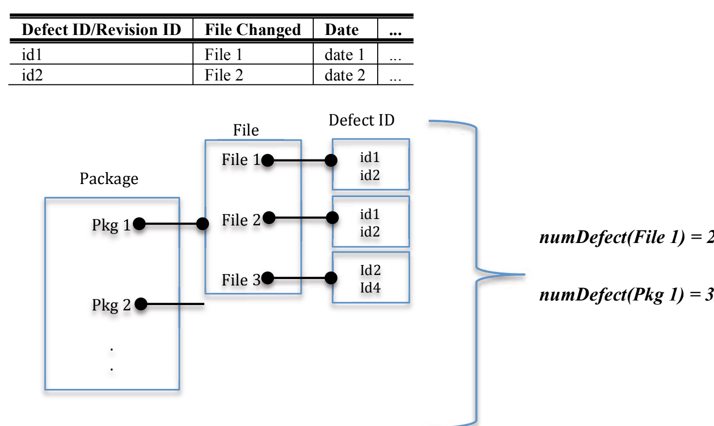


Figure 5 – Aggregating defect count at package or file level

3) Aggregating number of defects per class file and per package

In a release, it is possible that multiple reported bugs can be associated to one class file. The unique defect ID is thus appropriate to compute the number of defects fixes that affect a class file and a package. From the mapped change data, we look up each file and determine the total of defects per file by counting the number of unique defect ID in this release. At the package level, we aggregate the unique defect IDs for each class file in the package. As demonstrated in Figure 5, File1, File2 and File3 have 2 defects each, based on the defect ID and Pkg 1 has a total of 3 defects although it contains 3 files with 2 defects each. The unique defect-ID shows that for pkg1, only 3 defects are fixed.

4) Source code data collection

We have developed a small but very efficient java tool to extract source files meta-data. The source files are downloaded from the version repository. Organizational rules in java source file are substantially different from C# source file. As demonstrated in the relation diagram of a simple F1.java and F1.cs (Figure 6), a java source file has a one-one mapping from file to top-level class and it is not allowed to define another top-level class in a java file. In addition, the top-level class must have the same name as its enclosing file. Also, there is a one-zero or one-one mapping from file

to package; a maximum of one package can be defined in a java file. Finally, a java class can contain nested classes (one to many relation). In C#, multiple relations are possible. A file can contain many top-level classes and many top-level namespaces can also be defined in a file. It is also possible that a class contains nested classes and a namespace can equally contain nested namespaces. Unlike java file, the file name does not need to match any of the classes defined in it, although, good practices suggest to have filename as the same as a top-level class.

Since the compilation unit for both Java and C# is the source file and we are considering dependencies at the physical level as explained in section III, we decide for the following:

1. A dependency on any class in a source file implies a dependency on the source file.
2. The cyclic metric for a class is computed using dependencies that cross compilation units (source files). We skip cycles that are formed among classes within a source file.
3. The number of cycles for a compilation unit (source file) is the maximum cycle recorded for any of its classes.

Melton and Tempero (2007a) adapts "USES" relations from Lakos (1996) to a set of Java software to study cyclic dependency among the systems' classes. These relations have been applied on static code. Identifying coupling among classes using static code analysis has its drawback. As mentioned by Arisholm et al. (2004), because of polymorphism and the common presence of unused code in applications, coupling measures based on static code analysis loose precision, as they do not capture the actual coupling among classes at runtime. This study uses static code analysis because we consider various types of coupling that is not limited to message passing (method-method interactions) only. Also, class-level coupling data is easier to collect when using static code analysis and lastly, because ample evidence (Basili et al., 1996; Briand et al., 1998; Briand et al., 2001b; Chidamber and Kemerer, 1994; Zimmerman et al., 2011; Zimmermann and Nagappan, 2008) shows them to be useful predictors of defect-proneness of classes. We use the "USES" relations, which we have defined earlier as **DParent** and apply them also to the six software applications. We ignore all external library types (e.g., .NET and Java API) that developers have no access to their source codes since it is practically impossible for these external classes to form cycles with internal application's classes.

Figure 7 shows an example of the actual dependencies for **MyClass** and **mypackage** components. In order to collect other nodes (classes) to which **MyClass** is connected to requires that we scan the text of **MyClass**. The edge between **MyClass** and other **DParent(MyClass)** nodes is a directed path (without label, L) from **MyClass** to each node in the **DParent** set (Figure 7a-b). In the case of **mypackage** (Figure 7c-d), the **DParent(mypackage)**, is a set of unique imported packages and is processed from the collected class data.

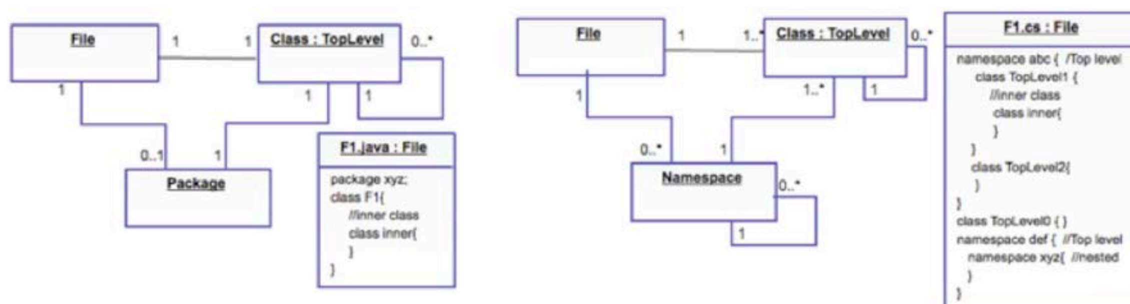


Figure 6 – (a) Java: File-Package-Class Relation (b) C#: File-Namespace-Class Relation

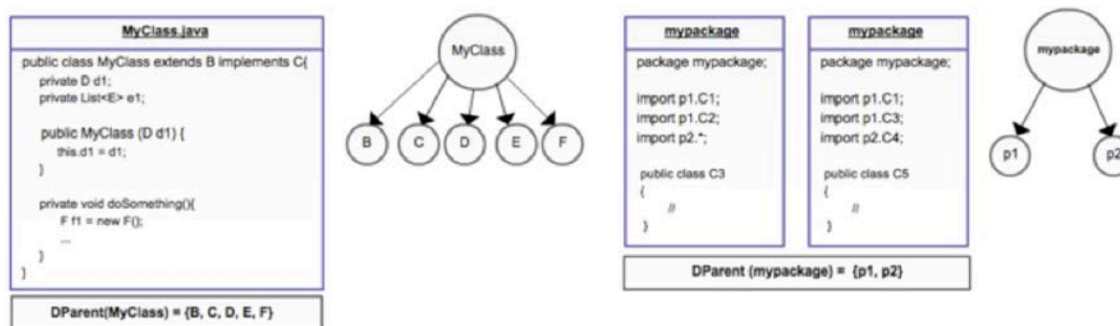


Figure 7 – (a) Class source data (b) Dependency Graph for Class (c) Package source data (d) Dependency Graph for package

5. Results and Discussions

We report the results of mining the software data with the Cycle metrics. In section 5.1, we present the results of the analysis for both package and class mined with the proposed cycle metrics. Sections 5.2 and 5.3 discuss the effect size and sanity checks on the reported results. Lastly, section 5.4 provides further discussions of the results.

We present the results of the statistical analysis of the **in-cycle** and **cyclic-related** (both in-cycle and depend-on-cycle) groups versus **non-cyclic** group. To simplify the replication of this study, we have listed the full results in Appendix A, Tables A.1 (Summary of the systems' data), A.2 and A.3. Figures 8a-b, 9a-b, 10a-b and 11a-b show the number of defective packages and classes, their proportions, the actual number of defects they produce and their defect densities for the cyclic-related and non-cyclic groups. In addition, in Appendix A, Figures A.1 and A.2, we provide the plots of the outgoing (efferent coupling) and incoming (afferent coupling) dependencies <V, E_{out}, E_{in}> and vertex vs. edge <V, E> for the cyclic dependency graphs for the last release of each system. As well, we show the diameters (Wasserman and Faust, 1994)[20] and radius vs. number of cycles for each system. We have used the Floyd-Warshall algorithm (Cormen et al., 2001) to calculate the "all-pairs shortest distance" between the nodes. Also, Table 3 lists the results of data normality tests using Shapiro and the t-tests or non-parametric Wilcoxon-test depending on the Shapiro p-value (see Appendix B, Figure B.3). A very small shapiro-wilk p-value (of less than 0.05) suggests that the data is significantly skewed (positively or negatively) or with significant kurtosis. The p-values of 1-tailed test for in-cycle vs. non-cyclic group is reported in column 2 and the p-values for cyclic-related vs. non-cyclic group are listed in column 3 of Table 3.

A. Distribution of defect and defect-prone components (DPCs) in cyclic and non-cyclic groups

We provide a break down of the results on the four categories of our data using Figures 8, 9, 10, 11 and Table 3. Statistical results at the package level show that the number, proportion and actual defect count of defective components in the cyclic-related group are consistently higher in most cases than those in the non-cyclic. We use Figures 8 and 9 to present the **package** results as follows:

- For **Camel**: defective components in *cyclic-related* group are **4.75 times higher** than those in the *non-cyclic* group (Figure 8a). **31%** of packages in the *cyclic-related* group are defective while **11%** are defective in the *non-cyclic* group (Figure 8b). Furthermore, the *cyclic-related* group has **6.1 times** the number of defect in the *non-cyclic* group (Figure 9a). Finally, the defect density in *cyclic-related* group is **0.5 times lower** than the *non-cyclic* group (Figure 9b).

[20] The diameter of a graph is the length of the shortest path between the most distanced nodes. This is calculated as the maximum of the eccentricities of the nodes or the maximum of the nodes' geodesic distances in the graph. The eccentricity of a node is the longest geodesic distance between the node and any other node in the graph. A geodesic represents the shortest path between two nodes.

- In **ActiveMQ**, the defective components in the *cyclic-related* group are about **8.11 times higher** than those in the *non-cyclic* group (Figure 8a). **45%** of the packages in *cyclic-related* group turn defective while **12%** are defective in *non-cyclic* group (Figure 8b). Defects produced by the *cyclic-related* group are **8.8 times higher** than those produced in the *non-cyclic* group (Figure 9a). Finally defects per a 1000-LOC in the *cyclic-related* group are **3.7 times higher** than those in the *non-cyclic* group (Figure 9b).
- In **Lucene,** the defective components in the *cyclic-related* group are **11 times** higher than defective components in the *non-cyclic* group (Figure 8a). Over **30%** of the packages in *cyclic-related* group turn defective while **6%** of packages in *non-cyclic* group are defective (Figure 8b). The *cyclic-related* group has **14 times** more defects than the *non-cyclic* group (Figure 9a). Finally, defects per a 1000-LOC in the *cyclic-related* group are **twice as higher** as those in the *non-cyclic* group (Figure 9b).
- In **commAPP**, defective components in *cyclic-related* group are **2.15 times higher** than those in the *non-cyclic* group. In terms of proportion of defective components in each group, **34%** of components in *in-cycle* group are defective while **9%** of components in *non-cyclic* are found with defects. In addition, the total defects produced by the *cyclic-related* group are **2.9 times higher** than those in the *non-cyclic* group. The defect density in the *in-cycle* group is **1.72 times higher** than the *non-cyclic* group.
- For **openPDC**: defective components in *cyclic-related* group are **14.2 times higher** than those in the *non-cyclic* group. In terms of proportion, **15%** of packages in *cyclic-related* are defective, whereas, **1%** of packages in non-cyclic group turn out to be defective. Also, the defects produced by the *cyclic-related* group are **20.2 times higher** than the *non-cyclic* group. The defect density in *cyclic-related* group is **7.3 times higher** than the *non-cyclic* group.
- For **Eclipse**, defective components in *cyclic-related* group are about **11 times higher** than the *non-cyclic* group. In terms of proportion, over **50%** of components in *cyclic-related* group are defective whereas **30%** in *non-cyclic* are found with defects.

At the **class-file** level, Figures 10 and 11 reveal that for:

- **Camel**: defective components in the *cyclic-related* groups are **13.7 times more** than defective components in the *non-cyclic* group (Figure 10a). **5.5%** of *in-cycle* classes are defective while the *non-cyclic* group has **1.7%** defective classes. (Figure 10b). The *in-cycle* classes have about **9.6 times more** defects than the *non-cyclic* group classes (Figure 11a). Lastly, defects per a 1000-LOC in the *in-cycle* group are **1.4 times more** than the *non-cyclic* group (Figure 11b).
- **ActiveMQ**: defective components in the *cyclic-related* group are about **3.6 times higher** than those in the *non-cyclic* group (Figure 10a). **12%** of the classes in *in-cycle* group turn defective while **2%** of the classes in *non-cyclic* group are defective (Figure 10b). Defects produced by the *cyclic-related* group are approximately **4.6 times higher** than those in the *non-cyclic* group (Figure 11a). Finally, defects per a 1000-LOC in the *in-cycle* group are about **2.74 times more** than the *non-cyclic* group (Figure 11b).
- **Lucene:** defective components in the *cyclic-related* group are **4.28 times higher** than defective components in the *non-cyclic* group (Figure 10a). **3%** of the classes in *cyclic-related* group turn defective while **1%** of classes in *non-cyclic* group are defective (Figure 10b). The *cyclic-related* group has **3.1 times more** defects compare to the *non-cyclic* group (Figure 11a). Finally, defects per a 1000-LOC in the *cyclic-related* group are **the same as** that of the *non-cyclic* group (Figure 11b).
- **commAPP**: defective components in *cyclic-related* are **5.2 times** more than those in the *non-cyclic* group. In terms of proportion of defective components in each group, **15%** of components in *in-cycle* are defective while **3%** of components in *non-cyclic* are found with defects. In addition, the total defects produced by the *cyclic-related* group are **3.2 times higher** than those in

the *non-cyclic* group. The defect density in the *in-cycle* group is **1.5 times higher** than those in the *non-cyclic* group.

- **openPDC**: defective components in *cyclic-related* group are **1.43 times higher** than the *non-cyclic* group. In terms of proportion, **2.2%** of classes in *cyclic-related* are defective, whereas, **1.4%** of classes in *non-cyclic* turn out to be defective. Also, the defects produced by *cyclic-related* group are approximately **0.86 times lower** than the *non-cyclic* group. The defect density in *cyclic-related* group is about **0.58 times lower** than the *non-cyclic* group.
- **Eclipse**: defective components in *cyclic-related* group are about **4.3 times higher** than the *non-cyclic* group. In terms of proportion, **19%** of components in the *in-cycle* group are defective whereas **11%** are found with defects in the *non-cyclic* group.

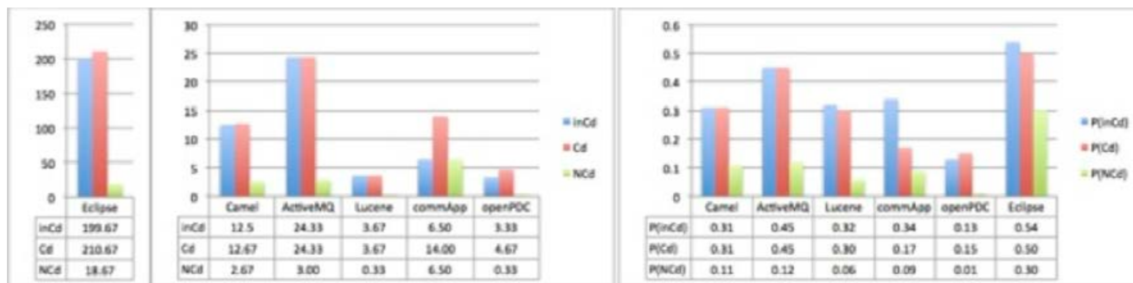


Figure 8 – (a) #Defective packages and (b) their proportions in in-cycle (inC), Cyclic (inC U DC) and non-cyclic (NC) groups

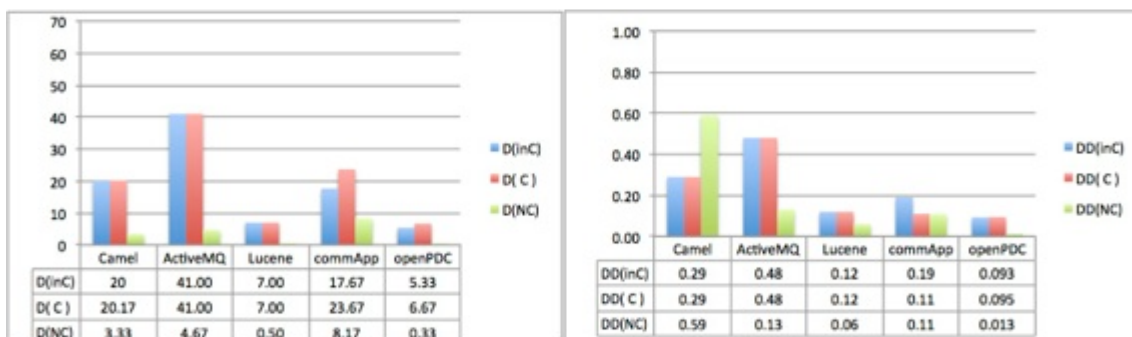


Figure 9 – (a) #Defects and (b) Defect Densities of Packages in in-cycle (inC), Cyclic (inC U DC) and non-cyclic (NC) groups

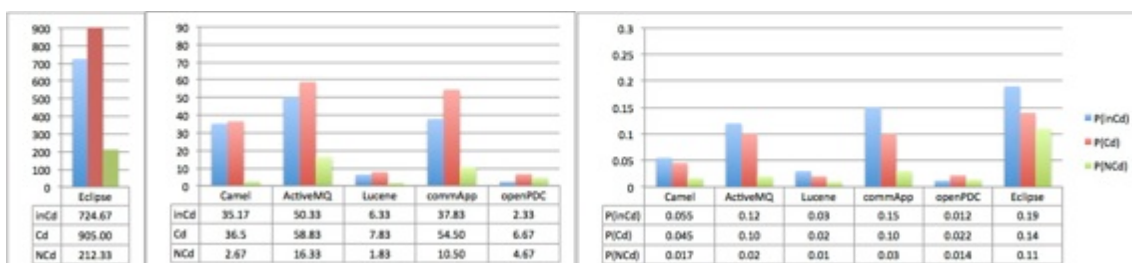


Figure 10 – (a) #Defective class-files and (b) their proportions in in-cycle (inC), Cyclic (inC U DC) and non-cyclic (NC) groups

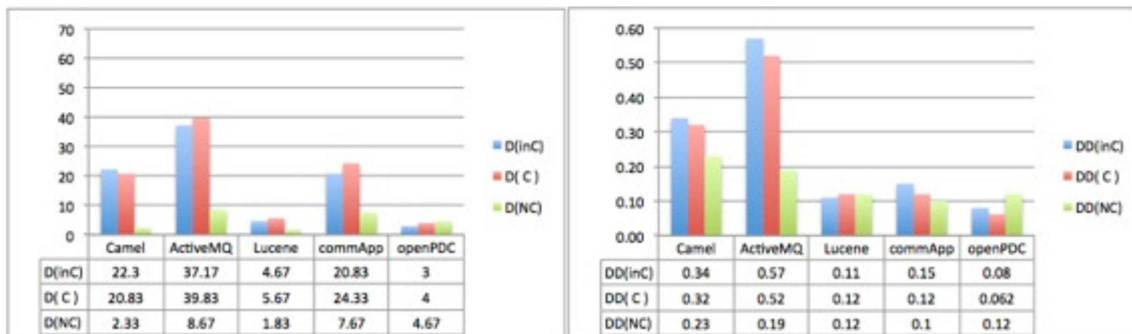


Figure 11 – (a) #Defects and (b) Defect Densities of class-files in in-cycle (inC), Cyclic (inC U DC) and non-cyclic (NC) groups

Table 3 – 1-tailed test for comparing cyclic and non-cyclic defective components

	CLASS			**PACKAGE**		
System	**H_{A1} : Test of #defective components for Cyclic and Non-Cyclic defective components**					
	Shapiro-wilk p-value	**p-value (inC)**	**p-value (C)**	**Shapiro-wilk p-value**	**p-value (inC)**	**p-value (C)**
Camel	0.8389	0.0089*	<0.0001*	0.8870	0.0008*	0.0012*
ActiveMQ	0.4773	0.0023*	<0.0001*	0.3947	<0.0001*	<0.0001*
Lucene	0.4928	0.0197*	0.0156*	0.4150	0.0005*	0.0005*
commApp	0.8429	0.0017*	<0.0001*	0.5999	0.5000	0.0002*
openPDC	0.6369	0.0007*	0.7302	0.7804	0.0117*	0.0029*
Eclipse	0.9596	0.0307*	0.0008*	0.088	0.0136*	0.0129*
	H_{A2} : Test of Proportion data for Cyclic and Non-Cyclic defective components					
	Shapiro-wilk p-value	**p-value (inC)**	**p-value (C)**	**Shapiro-wilk p-value**	**p-value (inC)**	**p-value (C)**
Camel	0.9451	0.0133*	0.0176*	0.9059	0.0016*	0.0021*
ActiveMQ	0.3805	0.0011*	0.0016*	0.3279	<0.0001*	<0.0001*
Lucene	0.7329	0.0308*	0.0402*	0.8411	0.0007*	0.0008*
commApp	0.4661	0.0023*	0.0016*	0.9505	0.0006*	0.0002*
openPDC	0.4375	0.5703	0.3613	0.4375	0.1338	0.3613
Eclipse	0.2369	0.0669	0.1338	0.6548	0.0047*	0.0093*
	H_{B1} : Test of number of defect for Cyclic and Non-Cyclic defective components					
	Shapiro-wilk p-value	**p-value (inC)**	**p-value (C)**	**Shapiro-wilk p-value**	**p-value (inC)**	**p-value (C)**
Camel	0.2265	0.0079*	0.0086*	0.063	0.0084*	0.0090*
ActiveMQ	0.1400	0.0019*	0.0022*	0.5220	0.0006*	0.0006*
Lucene	0.2151	0.1177	0.0184*	0.2543	0.0065*	0.0065*
commApp	0.0693	0.0156*	0.0078*	0.1368	0.0257*	0.0218*
openPDC	0.8998	0.7062	0.5733	0.6369	0.0530	0.0139*
	H_{B2} : Test of defect density for Cyclic and Non-Cyclic defective components					
	Shapiro-wilk p-value	**p-value (inC)**	**p-value (C)**	**Shapiro-wilk p-value**	**p-value (inC)**	**p-value (C)**
Camel	0.1589	0.0727	0.0936	0.3373	0.9737	0.9746
ActiveMQ	0.0186	0.0313*	0.0313*	0.1142	0.0009*	0.0009*
Lucene	0.5237	0.6432	0.6275	0.2543	0.0433*	0.0433*
commApp	0.2357	0.0156*	0.2578	0.0930	0.0389*	0.8437
openPDC	0.6537	0.6911	0.7483	0.2983	0.0664	0.0606

*: Significant p-value at α = 0.05

Table 4 – Summary of hypotheses test

	Summary of Hypotheses Test							
	Class				**Package**			
System	**H_{A1}**	**H_{A2}**	**H_{B1}**	**H_{B2}**	**H_{A1}**	**H_{A2}**	**H_{B1}**	**H_{B2}**
Camel	Y	Y	Y	N	Y	Y	Y	N
ActiveMQ	Y	Y	Y	Y	Y	Y	Y	Y
Lucene	Y	Y	Y	N	Y	Y	Y	Y
commApp	Y	Y	Y	Y	Y	Y	Y	Y
openPDC	Y	N	N	N	Y	Y	Y	N
Eclipse	Y	N	-	-	Y	Y	-	-

Significant = Y

Not Significant = N

B. Effect size

We discuss in this section the effect size check performed on the statistical data. As noted in Kampenes et al. (2007), effect size quantifies the size of the difference between two groups and allows us to judge whether the conclusions drawn from our hypotheses testing are meaningful or not. It is possible that the effect is small even when the statistical test is significant and vice versa. Therefore, for practical use of the results drawn from this study, we are compelled to carry out an effect size check on our results. In this study, we are concerned with two groups; the cyclic and the

non-cyclic groups. We apply the Hedges, g standardized effect size measure. Hedges, g is calculated as (Kampenes et al., 2007):

$$Hedges, g = \frac{\bar{X}_1 - \bar{X}_2}{s_p}$$

Where:

X_1 and X_2 represent the sample mean for each defect measure from cyclic and non-cyclic groups and s_p stands for the pooled standard deviation derived from the standard deviations s_1 and s_2 of cyclic and non-cyclic groups respectively as:

$$s_p = \sqrt{\frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{(n_1 - 1) + (n_2 - 1)}}$$ (Kampenes et al., 2007)

With small samples, the correction factor for the Hedges, g, when multiplied with g, adjusts for small sample bias. The correction factor (***cf***) is computed as (Kampenes et al., 2007):

$$1 - \frac{3}{4(N - 2) - 1}$$

Where:

n_1 *= Sample size for the cyclic group (equals the number of analyzed releases)*
n_2 *= Sample size for the non-cyclic group (equals the number of analyzed releases)*
$N = n_1 + n_2$
$n_1 = n_2 = 6$*, for Camel, ActiveMQ, Lucene and commApp.*
$n_1 = n_2 = 3$*, for openPDC and Eclipse*
For the effect size test, we are mostly concerned with the number of defect-prone components (DPCs) and the percentage of DPCs (proportion of DPCs*100) in both cyclic and non-cyclic groups. We therefore, in Table 5, report the Hedges, g for the two measures and for each system.

Interpretation

There are different ways to interpret effect size results as described in Kampenes et al. (2007). We choose to compare our effect size results to the reported results in Software Engineering empirical studies and categorized in Kampenes et al. (2007) under Table 9. In this Table, the size category for 284 estimated values for Hedges, g is given as: *Small: 0.00 – 0.376*, *Medium: 0.378 – 1.000* and *Large: 1.002 – 3.40*

As shown in Table 5, the effect sizes as measured by the Hedges, g for both number of DPC and percentage of DPC for the package results are in the "large" category. At the class level, the effect size for openPDC is in the "small" category while in Lucene, it ranges between "medium" (0.52) and "large" (1.12) categories. For the remaining four systems, the effect size falls in the "large" category. It can be explained that these two systems, openPDC and Lucene have very small number of DPCs (see Table A.1). We speculate that if the number of analyzed releases is increased, the results for these two might be somewhat different. Overall, the effect size test suggests that a random selection of defect-prone components in these systems has a higher probability to originate from the cyclic related group, either from in-cycle or both in-cycle and depend-on-cycle groups.

Table 5 – Hedges, g effect size measure (**cyclic vs. non-cyclic group**)

System	Package							Class-File						
	X_1	X_2	S_1	S_2	cf	S_p	g	X_1	X_2	S_1	S_2	cf	S_p	g
#DPC														
Camel	12.5	2.67	4.64	1.37	0.923	3.42	**2.65**	35.17	2.67	24.52	2.16	0.923	17.41	**1.72**
ActiveMQ	24.33	3	5.75	1.67	0.923	4.23	**4.65**	58.83	16.33	22.25	10.13	0.923	17.29	**2.27**
Lucene	3.67	0.33	1.03	0.82	0.923	0.93	**3.31**	6.67	2.67	4.37	1.63	0.923	3.29	**1.12**
commApp	14	6.5	4	3.4	0.923	3.71	**1.86**	37.83	10.5	20.37	10.13	0.923	16.09	**1.57**
openPDC	4.67	0.33	1.15	0.58	0.8	0.91	**3.81**	6.67	4.67	7.02	4.16	0.8	5.77	**0.28**
Eclipse	199.33	19	57.74	6.56	0.8	41.09	**3.51**	729	208.7	265.6	58.4	0.8	192.29	**2.16**
Percentage of DPC														
Camel	31	11	11.3	5.7	0.923	8.95	**2.06**	5.5	1.8	3.9	1.4	0.923	2.93	**1.17**
ActiveMQ	45	12	11	7	0.923	9.2	**3.30**	11.5	2.3	4.2	1.5	0.923	3.15	**2.69**
Lucene	35.7	4.8	12.8	11.7	0.923	12.3	**2.33**	2.2	1.5	1.5	0.9	0.923	1.24	**0.52**
commApp	17.3	9.1	6.6	6.2	0.923	6.4	**1.18**	15.2	2.6	9	3	0.923	6.71	**1.73**
openPDC	15.1	0.8	6.1	1.5	0.8	2.58	**2.58**	2.2	1.4	2.3	1.2	0.8	1.83	**0.35**
Eclipse	54	30.2	2.9	2.3	0.8	2.62	**7.27**	18.4	11.3	4.9	0.5	0.8	3.48	**1.63**

C. Sanity Check

We want to verify if the proportion of defect-prone components (DPCs) in the cyclic group is of interest or not. Earlier, we demonstrated that the cyclic group contains the higher number of DPCs than the non-cyclic group. However, this proportion can be a very small number since the distribution of defects and DPCs in a software system is usually skewed (Fenton and Ohlsson, 2000) and the proportion in each group (cyclic or non-cyclic) is relative to the proportion of DPCs in the entire system. As listed in Table A.1, the systems we analyzed agree with this observation because the DPCs are indeed few in number relative to the entire systems' components.

What is therefore of interest is to see if a standard classifier can find ***precisions*/*recalls*** over (100 – "*actual percentage of DPCs in cyclic group*") or ***false alarm rates*** under "*actual percentage of DPCs in cyclic group*". If either of these conditions is fulfilled, we can conclude that the proportion of DPCs in the cyclic group is important in this data set. To achieve this objective, we use Naïve Bayes (http://www.cs.waikato.ac.nz/ml/weka/) classifier because of its simplicity (Hall et al., 2011) and Random Forest because of its ability to generalize well on small dataset (Breiman, 2001). For the classification task, we employ three independent variables; the source lines of code (SLOC), the weighted method for a class (WMC) and coupling between class objects (CBO) (Efferent, ce and afferent, ca couplings) metrics because our tool already measures them and because they are shown to be good predictors of defect-proneness of components (Basili et al., 1996; Briand et al., 1998; Chidamber and Kemerer, 1994; Zhang, 2009). We trained the models by using cross validation method on the dataset for each group. In this approach, a training dataset is divided into 10-folds and the model is trained on each fold with the result cross-validated on the rest folds in each iteration. By doing this, we achieve both training of the model using each fold as training set and at the same time

testing the model's performance on the entire dataset. For the purpose of the sanity check, we consider that this approach of model training and testing suffices. Also, since we are not focused on building a reusable model, we therefore do not concentrate on thorough training of each of the models. For these reasons, we have used default classifier parameters for Naïve Bayes and have only changed the default number of trees in Random Forest from 10 to 500.

Table 6 lists the precision, recall, false alarm rates and the actual percentage of defect-prone components in the cyclic group averaged over the number of releases. As shown, in all the cases, the ***false alarm rates*** are lower than the *actual percentage of defect-prone components in the cyclic group* (i.e. Actual %C_{DPC}). In addition, the ***precisions*** for Camel, ActiveMQ, Lucene and Eclipse are over (100-Actual %C_{DPC}) at the package level. Also, the ***recalls*** for Camel, ActiveMQ, Lucene and Eclipse are over (100-Actual %C_{DPC}). At the class-file level, the false alarm rates for Camel, ActiveMQ, commApp and Eclipse are lower than the *actual percentage of defect-prone components in the cyclic group.* But, for Lucene and openPDC, the classifiers could not divide between the DPCs and non-DPCs in some of the releases in these dataset because of the few number of DPCs recorded in these two systems. We therefore decided to exclude them from the results. As listed in Table A.2, Lucene has an average of 9.3 DPCs out of 501 class-files and openPDC has an average of 11.3 DPCs out of 616 class-files. Although, the small sample sizes of these two systems and the decision to exclude them based on the above stated reason do not override/invalidate the claims in this study. We however, learn a great deal that sanity checks can guide our decisions regarding where to focus such expensive cyclic dependency analysis efforts in software systems both for research and for industrial practices.

D. Discussion

Clearly, the results show interesting trends of significant higher defect profiles for cyclic dependent components in the systems. As revealed in Table 4, at the package level the null hypotheses for $\boldsymbol{H_{A1}}$ and $\boldsymbol{H_{A2}}$ are rejected for all the systems indicating that the results are all significant. Similarly, the null hypotheses for $\boldsymbol{H_{B1}}$ are rejected for all the 5 systems that we have their actual defect dataset. We fail to reject the null hypotheses of $\boldsymbol{H_{B2}}$ for Camel and openPDC. At the class level the null hypotheses for $\boldsymbol{H_{A1}}$, $\boldsymbol{H_{A2}}$ and $\boldsymbol{H_{B1}}$ are equally rejected for all the systems except for openPDC and $\boldsymbol{H_{A2}}$ for Eclipse. However, for Eclipse, the effect size in Table 5 shows a large effect. This confirms that, even though the statistical test is not significant which largely can be due to the small sample size (number of releases). The effect size shows that the difference between the two groups for Eclipse is not negligible. The null hypotheses for $\boldsymbol{H_{B2}}$ are rejected for 3 out of the 5 systems. In all the cases where we fail to reject the null hypothesis for $\boldsymbol{H_{B2}}$, it is either there are higher number of cyclic components than non-cyclic or that the cyclic group's size (LOC) is higher than the non-cyclic group. openPDC shows a contrasting result to the rest of the systems at the class level. It is hard to imply any pattern from the C# applications' results at the class level because of the sample size (i.e. Number of systems analyzed) and also because of the number of defect-prone components available for this study. Further studies will be necessary to observe patterns in this direction.

1) Multiplicity of Defect

A clear observation from the results is that the cyclic-related group has significantly more defective components and accounts for higher number of defects than those in the non-cyclic group. In addition, the proportion of defective components in cyclic group is higher than those in the non-cyclic group except for openPDC (class-file level). It thus means that components in cyclic relationships tend to be more defect-prone and that possibly, defects propagate more among components in cyclic relationships. Cyclic dependencies increase the probability of defect propagation and the tendency to make the system fragile, thus leading to possible increase in the number of system's defects. While we cannot claim exclusively that cyclic relationship is

unequivocally responsible for this behavior, the results from this study, however, lend support to this pattern. This effect poses huge maintenance challenge as the system evolves. Defects become difficult to trace and system become more strenuous to test, thus resulting into higher maintainability cost.

Table 6 – Classification results averaged over the number of releases

System	Cyclic				
	Precision	**Recall**	**FP Rate**	**Actual %C_{DPC}**	**(100-Actual %C_{DPC})**
Package					
Camel*	69.6	51.1	8.8	30.5	69.5
ActiveMQ*	75.2	43.5	10.8	44.9	55.1
Lucene+	77.1	57.9	8.3	32.4	67.6
commApp*	67.8	55.1	5.1	19.2	80.8
openPDC+	55.6	19.4	3.8	15.1	84.9
Eclipse+	67.9	71.2	34.1	50.0	50.0
Class-File					
Camel*	25.2	26.2	3.6	4.5	95.5
ActiveMQ*	33.8	35.5	6.8	8.9	91.1
Lucene	-	-	-	2.2	97.8
commApp*	42.3	45.7	6.4	10.1	89.9
openPDC	-	-	-	2.2	97.8
Eclipse+	55.7	26.1	3.3	13.8	86.2

*** Naïve Bayes**
+ Random Forest

2) Cycle-Size Relationship

We discover a positive correlation between LOC and minimum number of component's cycle for both the packages and the classes in all the 6 systems. In many cases, there is a correlation of more than 0.5 between the size and the minimum number of component's cycle.

A look at the cyclic-related group distribution against the size (KLOC) in each group (Appendix A, Tables A.2 and A.3) reveals for example that:

- For ActiveMQ: the in-cycle group has about 32.5% (*inC*/*N*) of the total classes but accounts for 55% of the total size ($KLOC_{inC}/(KLOC_C+KLOC_{NC})$).
- For commApp: Packages in the cyclic group are 12.6% of the total packages and this number account for a total of 32% of the total size. Relatedly, the cyclic group at the class level contains 25.2% of the total classes and contributes 46.8% of the total size.

- For Lucene: Classes in the cyclic-related group are 38.8% of all the classes but account for 57.5% of the total size.

This effect accounts for the mixed results for hypothesis $\boldsymbol{H_{B2}}$. The correlation values also show that some large classes have many cycles and thus seem to promote cyclic relationships among interconnected components. (Melton and Tempero, 2007a) made similar observations about the presence of cycles in some large classes. Lines of Code (LOC) and degree of coupling have long been validated to correlate to defect-proneness (Basili et al., 1996; Briand et al., 2001b; Menzies et al., 2007). The large components we found in the cyclic group contain many cycles and a high number of incoming and outgoing couplings. A beneficial approach would then be the need to reduce the cyclic connections. Performing such refactoring would invariably reduce both the size and tight coupling in these large components.

3) Number of Defect Prone Components in cyclic vs non-cyclic group

In terms of number of defective components, cyclic relationships are convincingly important. As observed in Figure 12, the number of defective classes located in the cyclic group is very high. For instance, Apache Camel has 90% of all the defective classes (82% at the package level) in the in-cycle group and 93% (83% at the package level) when combined with depend-on-cycle, that is, the cyclic-related group. We observe however that both applications developed with C# (commApp and openPDC) give the least results in the in-cycle group. Further investigation of many C# systems will thus be necessary to study the defect patterns of components in the cyclic group. Overall, this is a very useful finding that can be employed during system testing to effectively allocate testing resources in a software development and maintenance project. Furthermore, we suggest that based on these results, it is possible to investigate the cyclic metrics for improving existing quality models. Finally, since cyclic related components account for the highest number of defects and defect-prone components in these systems, we argue that focusing on defect-prone cyclically related components for refactoring could be a positive step. Our speculation therefore is that since cycles increase structural complexities (Briand et al., 1998; Briand et al., 2001b), performing such refactoring by taking advantage of existing refactoring tools could reduce the defect-proneness of components and consequently improve the reliability of the system.

4) Package vs Class

The burden of cyclic dependency is high as it increases the cost of software testing to trace or track a defect. As noted in the results, package level results are more significant than class level. openPDC has significant results for cyclic group at the package level even though at the class level the results are mostly significant for the non-cyclic group. This reinforces the Acyclic Dependencies Principle as proposed by Martin (2000). Package to package dependency also implies for instance, in Java that an "import" directive is used. Additionally, it translates to a strong cyclic physical dependency as mentioned by Lakos (1996). Cyclic dependencies among packages will result into strong structural complexity by making the modules to be tightly coupled and thereby increasing the tendencies of defect propagation. As Martin (2000) states, "*a dependency upon a package is a dependency upon everything within the package*" The implication is increasing cost of testing and maintenance as the system evolves. Is it then necessary to focus on this property? Our empirical results in this study suggest an affirmative yes. Empirical evidence shows that cycles in real-life systems mostly grow as these systems evolve (Melton and Tempero, 2007a), our results also agree with this pattern leaving us with strong doubt that refactoring option hardly focus on breaking dependency cycles. This study has very useful implications for maintenance engineers and system testers. The more information we have about the groups or subsets within a software system with the most defect-prone components, the better we can allocate quality assurance resources and efforts to trace and test components in the system.

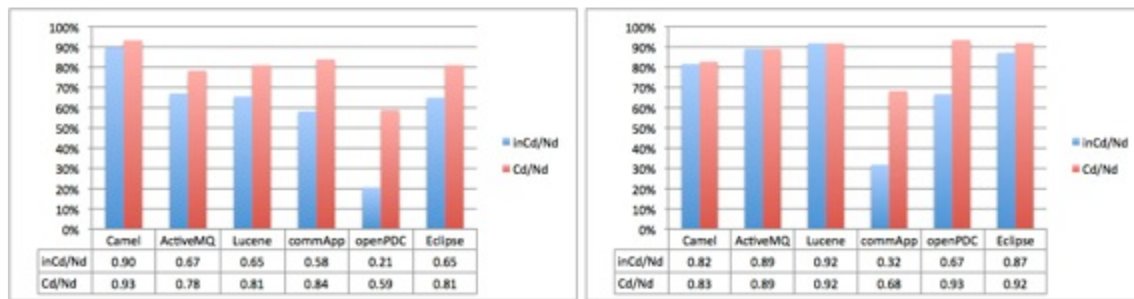


Figure 12 - Number of defective components found in the cyclic group (a) Class and (b) Package

6. Threats to Validity

We have analysed and evaluated two Smart Grid systems, an integrated development environment, a search engine application, a versatile routing and mediation engine and a messaging and integration pattern server. Although, these six systems vary in terms of properties such as domain, functionality, programming language, size, age, usage, context and study period, we cannot claim that the observed defect patterns or related will hold for other systems. As it is with most case studies, we cannot generalize these results across all systems. Further studies will be necessary to compare results across several systems and domains.

Our study is based on static coupling measurements and not dynamic coupling measurements (Arisholm et al., 2004); as such actual coupling among classes at runtime may not be completely captured. This imprecision can occur due to polymorphism, dynamic binding and dead code in the software. This as it may, static code analysis has been found to be practically useful and less expensive to collect (Basili et al., 1996; Briand et al., 1998; Briand et al., 2001b; Chidamber and Kemerer, 1994; Zimmerman et al., 2011; Zimmermann and Nagappan, 2008). Our study collects coupling types that are not only based on method invocation. In addition, static coupling measures reflects to a very high degree the coupling among classes at runtime. We do not think the data collected based on static code analysis can bias our result in any significant manner.

For this study, we have relied on the defects logged in the defect tracking systems of each application. Our approach of defect data extraction is similar to what other researchers have used in the past (C'ubranic, 2004; S'liwerski et al., 2005; Schroeter et al., 2006). Nevertheless, a common threat is whether defects logged in the DTS are accurately tagged in the respective code changes in the version systems. In addition, we cannot be sure if all defects are logged in the DTS especially in cases where the defects are discovered by the developers. Also, there could be cases that the message log of the file that consists a change is not tagged with the bug numbers of the resolved defect. Furthermore, there could be cases of typographical error in the recording of the bug number in the version systems (C'ubranic, 2004) and lastly, it is still possible that duplication will occur. For instance, in cases where the commits in the log is not tagged with the bug number from the defect tracking system, we can never be sure that a commit with a particular bug fix is not “re-commit” in the version control system with the same bug fix. All these are common threats to the internal validity of studies that use mapped data from both the DTS and the version control system. Comparably, independent defect dataset of Eclipse yield results in the same direction as the defect dataset that we collected.

We address construct validity using four different hypotheses. These hypotheses measure in detail the number, ratio and size of defect profiles of components in both groups. All dimensions to establish which particular group has higher defect profile are adequately captured with the stated hypotheses.

7. Conclusions and Future Work

We have carried out, to the best of our knowledge the first and an extensive investigation into cyclic properties of software components and their defect profiles. Using our proposed cycle metrics, we divided the mined data into two groups; cyclic group and non-cyclic group. Statistical analysis reveals that components in cyclic relationships are more defect-prone, having more number of defects and containing more defective components. In addition, it shows that defect propagation in the cyclic group is significantly higher than non-cyclic group. This study shows additional structural component property that may impact on the defect proneness of software components.

Furthermore, it reinforces the results from previous studies on coupling complexity and the impacts on system quality. A noteworthy observation is the presence of cycles in all the systems that we analyzed. Evidence from previous studies supports our result that cycles are indeed pervasive in real-life systems. This further supports our hypothesis that cyclic dependencies should be considered when collecting structural properties of software components.

These results have implications for software maintenance. By focusing on the cyclic group, it is possible to discover most defects and defective components in the system. Testing resources can therefore be effectively allocated to trace defects and test components in a cost efficient manner.

As further study, we seek to analyze a large number of versions in each system we have analyzed so as to understand the evolution behavior of dependency cycles and defect proneness. We seek to know, if defective components increase in cyclic group as the system evolves and if certain factors have some effect (such as refactoring) on the evolution of defect in the cyclic group.

In this study, we have used all types of dependency relationships that result in cycles. Some dependencies are stronger than the other in terms of their coupling characteristics (Briand et al., 1999; Kung et al., 1996). Can we identify which dependency relationship (Inheritance, Aggregation or Composition) contribute most defects in a cyclically dependent components? Lakos (1996) explained that intrinsic cyclic dependencies are those cycles that cannot be avoided giving example of a Node and Edge in a graph, with node having information about the edge and vice-versa. Are there cycles we may care less about regarding their tendencies to propagate defects among inter-related components and thus prune the cyclic group to those with higher probability of defect proneness? We would investigate these in our future work.

In addition, we plan to investigate the most common types and severity of defects involved in cyclic dependencies and compare to non-cyclic group. Also, we will investigate how these results can be used in combination with other approaches to improve defect prediction models. Based on the current results, it is positive that we can employ the cycle variables as predictors of a component's defect-proneness.

References

Abreu, F.B.E., Melo, W., 1996. Evaluating the impact of Object-Oriented design on software quality. Proceedings of the 3rd International Software Metrics Symposium, 90-99.

Arisholm, E., Briand, L.C., Foyen, A., 2004. Dynamic coupling measurement for object-oriented software. IEEE T Software Eng 30, 491-506.

Basili, V.R., Briand, L.C., Melo, W.L., 1996. A validation of object-oriented design metrics as quality indicators. IEEE T Software Eng 22, 751-761.

Bennett, S., McRobb, S., Farmer, R., 1999. Object-Oriented systems analysis and design using UML. McGraw Hill, Maidenhead, Berkshire England.

Breiman, L., 2001. Random Forests. Mach. Learn. 45, 5-32.

Briand, L.C., Daly, J., Porter, V., Wust, J., 1998. Predicting fault-prone classes with design measures in object-oriented systems. Ninth International Symposium on Software Reliability Engineering, Proceedings, 334-343.

Briand, L.C., Daly, J.W., Wust, J.K., 1999. A unified framework for coupling measurement in object-oriented systems. IEEE T Software Eng 25, 91-121.

Briand, L.C., Labiche, Y., Yihong, W., 2001a. Revisiting strategies for ordering class integration testing in the presence of dependency cycles, Proc. 12th International Symposium on Software Reliability Engineering, (ISSRE 2001) pp. 287-296.

Briand, L.C., Labiche, Y., Yihong, W., 2003. An investigation of graph-based class integration test order strategies. Software Engineering, IEEE Transactions on 29, 594-607.

Briand, L.C., Wuest, J., Lounis, H., 2001b. Replicated Case Studies for Investigating Quality Factorsin Object-Oriented Designs. Empirical Softw. Engg. 6, 11-58.

C'ubranic, D., 2004. Project History as a Group Memory: Learning From the Past. , PhD Thesis. University of British Columbia, Canada.

Capretz, L.F., Capretz, M.A.M., 1996. Object-Oriented Software: Design and Maintenance. World Scientific, Singapore, 263 pages.

Capretz, L.F., Lee, P.A., 1992. Reusability and Life Cycle Issues within an Object-Oriented Design Methodology (refereed), in: Ege, R., Singh, M., Meyer, B. (Eds.), Technology of Object-Oriented Languages and Systems. Prentice Hall, Englewood Cliffs, USA, pp. 139-150.

Chidamber, S.R., Kemerer, C.F., 1994. A Metrics Suite for Object-Oriented Design. IEEE T Software Eng 20, 476-493.

Coad, P., Yourdon, E., 1991. Object-Oriented Design, 2nd ed. Prentice Hall, London.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2001. Introduction to algorithms, 2nd ed. MIT Press, Cambridge, Mass.

Fenton, N.E., Ohlsson, N., 2000. Quantitative analysis of faults and failures in a complex software system. IEEE T Software Eng 26, 797-814.

Fenton, N.E., Pfleeger, S.L., 1997. Software Metrics: A Rigorous & Practical Approach, 2nd ed. PWS Publishing Press, Boston.

Fowler, M., 2001. Reducing coupling. Software, IEEE 18, 102-104.

Gupta, A., Li, J., Conradi, R., Ronneberg, H., Landre, E., 2010. Change profiles of a reused class framework vs. two of its applications. Information and Software Technology 52, 110-125.

Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2011. A Systematic Review of Fault Prediction Performance in Software Engineering. IEEE T Software Eng 99.

Hanh, V.L., Akif, K., Le Traon, Y., Jezequel, J.M., 2001. Selecting an Efficient OO Integration Testing Strategy: An Experimental Comparison of Actual Strategies. Proc. 15th European Conf. Object-Oriented Programming (ECOOP), 381-401.

Hashim, N.L., Schmidt, H.W., Ramakrishnan, S., 2005. Test order for class-based integration testing of Java applications, Fifth International Conference on Quality Software, 2005. (QSIC 2005). , pp. 11-18.

Hautus, E., 2002. Improving Java software through package structure analysis., In The 6th IASTED International Conference Software Engineering and Applications.

Jungmayr, S., 2002. Identifying test-critical dependencies, Software Maintenance, pp. 404-413.

Kampenes, V.B., Dyba, T., Hannay, J.E., Sjoberg, D.I.K., 2007. A systematic review of effect size in software engineering experiments. Inform Software Tech 49, 1073-1086.

Knoernschild, K., 2012. Java Application Architecture: Modularity Patterns with Examples Using OSGi, 1st ed. Prentice Hall.

Kung, D., Gao, J., Hsia, P., Toyoshima, Y., Chen, C., 1996. On Regression Testing of Object-Oriented Programs. Journal of Systems Software 32, 21-40.

Lakos, J., 1996. Large-scale C++ software design. Addison-Wesley Longman, Redwood City, CA.

Laval, J., Denier, S., Ducasse, S., Bergel, A., 2009. Identifying cycle causes with Enriched Dependency Structural Matrix. 16th Working Conference on Reverse Engineering (WCRE 2009), 113-122.

Le Traon, Y., Jeron, T., Jezequel, J.M., Morel, P., 2000. Efficient object-oriented integration and regression testing. Reliability, IEEE Transactions on 49, 12-25.

Li, Z.D., Madhavji, N.H., Murtaza, S.S., Gittens, M., Miranskyy, A.V., Godwin, D., Cialini, E., 2011. Characteristics of

multiple-component defects and architectural hotspots: a large system case study. Empir Softw Eng 16, 667-702.

Marinescu, R., 2001. Detecting design flaws via metrics in object-oriented systems. Tools 39, 173-182.

Martin, R., 2000. Design Principles and Design Patterns, in: Mentor, O. (Ed.).

Martin, R.C., 1996. Granularity, C++, pp. 57-62.

Melton, H., Tempero, E., 2007a. An empirical study of cycles among classes in Java. Empir Softw Eng 12, 389-415.

Melton, H., Tempero, E., 2007b. JooJ: real-time support for avoiding cyclic dependencies. Proceedings of the thirtieth Australasian conference on Computer science 62, 87-95.

Menzies, T., Greenwald, J., Frank, A., 2007. Data mining static code attributes to learn defect predictors. IEEE T Software Eng 33, 2-13.

Nagappan, N., Bhat, T., 2007. Technologies for Code Failure Proneness Estimation. Microsoft Corporation, USA.

Olbrich, S., Cruzes, D.S., Basili, V., Zazworka, N., 2009. The Evolution and Impact of Code Smells: A Case Study of Two Open Source Systems, Int Symp Emp Softwar, pp. 391-401.

Oyetoyan, T.D., Cruzes, D.S., Conradi, R., 2012. Can Reused Components Provide Lead to Future Defective Components in Smart Grid Applications?, in: Gonzalez, T., Hamza, M.H. (Eds.), 16th IASTED International Conference on Software Engineering And Applications, Las Vegas, USA.

Parnas, D.L., 1979. Designing Software for Ease of Extension and Contraction. Ieee T Software Eng SE-5, 128-138.

S'liwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes?, Proceedings of the 2005 international workshop on Mining software repositories. ACM, St. Louis, Missouri, pp. 1-5.

Sangal, N., Jordan, E., Sinha, V., Jackson, D., 2005. Using dependency models to manage complex software architecture. Acm Sigplan Notices 40, 167-176.

Schroeter, A., Zimmermann, T., Zeller, A., 2006. Predicting component failures at design time, Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering. ACM, Rio de Janeiro, Brazil, pp. 18-27.

Souza, D., Wills, A., 1999. Objects, components, and frameworks with UML:The catalysis approach. Addison-Wesley, Reading, Mass.

Tai, K.-C., Daniels, F.J., 1997. Test order for inter-class integration testing of object-oriented software. Computer Software and Applications Conference, 1997. COMPSAC '97. Proceedings., The Twenty-First Annual International, 602-607.

Wasserman, S., Faust, K., 1994. Social network analysis : methods and applications. Cambridge University Press, Cambridge ; New York.

Weyuker, E., Ostrand, T., Bell, R., 2008. Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models. Empir Softw Eng 13, 539-559.

Yutao, M., Keqing, H., Bing, L., Xiaoyan, Z., 2010. How multiple-dependency structure of classes affects their functions a statistical perspective, 2nd International Conference on Software Technology and Engineering (ICSTE), 2010, pp. V2-60-V62-66.

Zhang, H.Y., 2009. An Investigation of the Relationships between Lines of Code and Defects. 2009 IEEE International Conference on Software Maintenance, Conference Proceedings, 274-283.

Zimmerman, T., Nagappan, N., Herzig, K., Premraj, R., Williams, L., 2011. An Empirical Study on the Relation between Dependency Neighborhoods and Failures, IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST), pp. 347 - 356.

Zimmermann, T., Nagappan, N., 2007. Predicting subsystem failures using dependency graph complexities. ISSRE 2007: 18th IEEE International Symposium on Software Reliability Engineering, Proceedings, 227-236.

Zimmermann, T., Nagappan, N., 2008. Predicting Defects using Network Analysis on Dependency Graphs. 2008 30th International Conference on Software Engineering: (ICSE), Vols 1 and 2, 530-539.

Zimmermann, T., Premraj, R., Zeller, A., 2007. Predicting Defects for Eclipse, International Workshop on Predictor Models in Software Engineering, pp. - 9.

Appendices

Appendix A - Software data

Table A.1 – Summary of software source code and defect data

Release/Version	Date	#Pkg	#Class-File	#Class	KLOC	#Defective Class-File	#Defect
Apache-Camel							
2.10.2	Oct 15 2012	67	991	1108	78.91	16	11
2.10.1	Aug 28 2012	67	991	1108	78.55	20	9
2.10.0	Jul 01 2012	67	991	1108	78.23	59	30
2.9.2	Apr 17 2012	65	959	1074	74.97	31	12
2.9.1	Mar 05 2012	65	955	1068	74.32	23	17
2.9.0	Dec 31 2011	65	952	1063	73.43	86	49
Apache-ActiveMQ							
5.7.0	Nov 22 2012	82	1517	1665	136.22	35	68
5.6.0	Jun 15 2012	83	1505	1649	133.25	88	102
5.5.1	Oct 16 2011	78	1331	1472	118.27	54	76
5.5.0	Apr 01 2011	78	1331	1472	118.27	115	105
5.4.2	Dec 02 2010	77	1258	1393	113.01	80	66
5.4.1	Sept 21 2010	77	1256	1386	112.20	79	63
Apache-Lucene							
4.0	Oct 12 2012	20	620	1115	76.60	9	6
3.6	Apr 12 2012	18	503	810	73.78	2	2
3.5	Nov 27 2011	18	498	792	68.22	15	13
3.4	Sep 14 2011	17	478	752	65.44	3	3
3.3	Jul 01 2011	16	466	726	59.28	16	13
3.2	Jun 03 2011	15	441	683	56.04	11	13
Eclipse							
3.0	Jun 25 2004	645	10635	12671	1308.66	1566	-
2.1	Mar 27 2003	429	7909	9258	988.45	845	-
2.0	Jun 27 2002	378	6751	7704	797.93	968	-
commApp							
4.2.4	Nov 14 2012	191	1203	2142	341.83	29	14
4.2.2	Oct 12 2012	191	1199	2134	339.78	49	18
4.1	Aug 17 2012	171	1002	1884	316.22	60	42
4.0.1SP4	Apr 11 2012	141	904	1650	286.99	69	29
4.0.1SP2	Mar 26 2012	142	903	1645	285.89	46	28
4.0	Oct 14 2011	133	849	1546	266.11	137	143
openPDC							
1.5	Oct 11 2012	73	637	798	105.69	6	6
1.4SP2	Dec 28 2011	75	623	794	103.03	14	10
1.4	March 12 2011	72	575	740	85.37	14	5

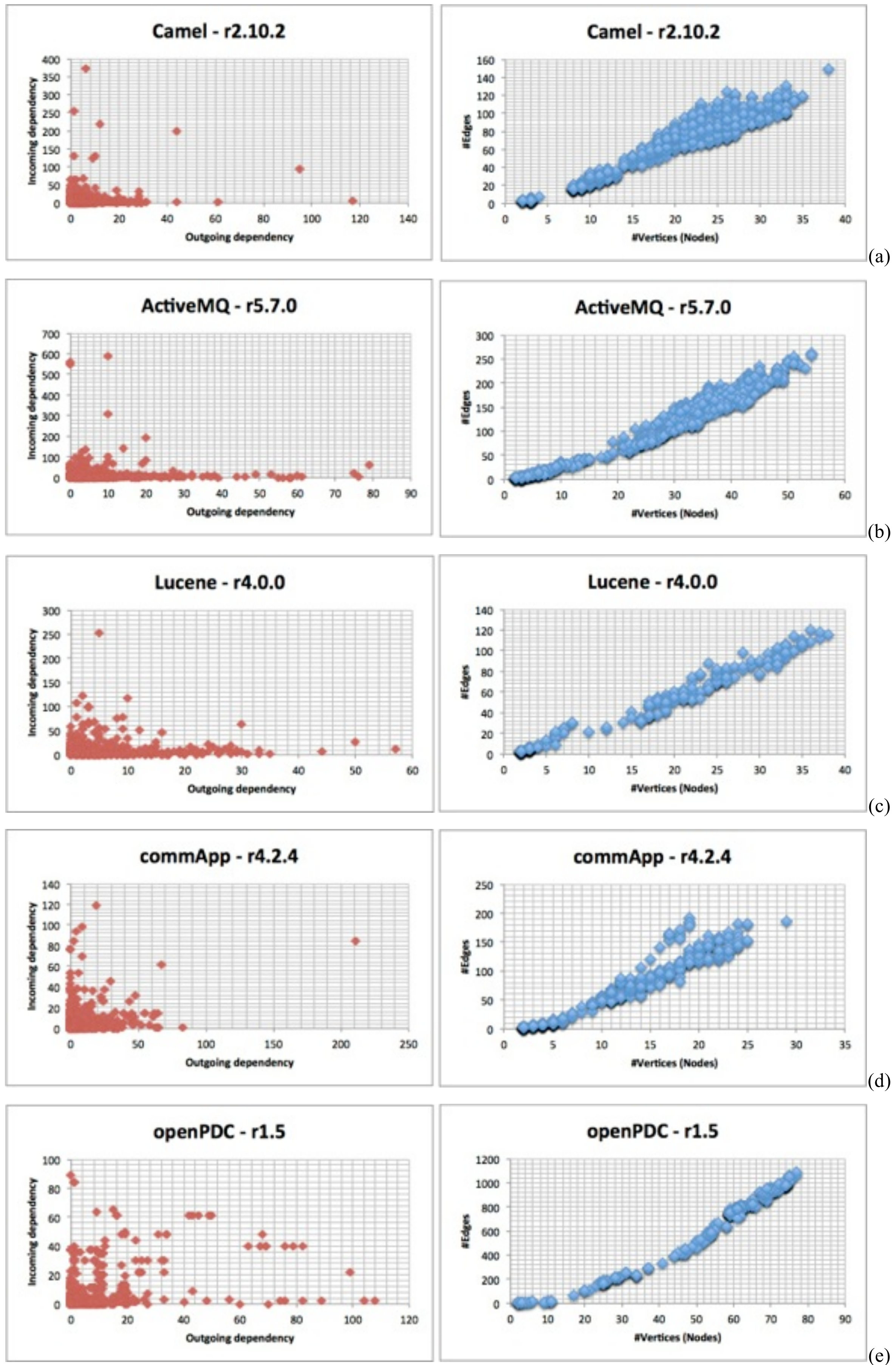
Table A.2 – Average of #defective components and their proportions for both Cycle and Not-In-Cycle groups[21]

System	N	N_d	inC	DC	C	NC	inC_d	DC_d	C_d	NC_d	$P(inC_d)$	$P(C_d)$	$P(NC_d)$
Class													
Camel	973.17	39.17	641.00	181.17	822.17	151	35.17	1.33	36.5	2.67	0.06	0.05	0.02
ActiveMQ	1366.33	75.17	444.67	215.33	660.00	706.33	50.33	8.50	58.83	16.33	0.12	0.10	0.02
Lucene	501.00	9.33	194.33	122.33	316.66	184.33	5.00	1.67	6.67	2.67	0.03	0.02	0.01
commApp	1010.00	65.00	254.50	290.33	544.83	465.17	37.83	16.67	54.50	10.50	0.15	0.10	0.03
openPDC	611.67	11.33	202.33	96.33	298.67	313.00	2.33	4.33	6.67	4.67	0.01	0.02	0.01
Eclipse	8431.33	1126.33	3971.33	2610.00	6581.33	1850	729	188.67	917.67	208.67	0.18	0.14	0.11
Package													
Camel	66.00	15.33	40.50	1.00	41.50	24.50	12.50	0.17	12.67	2.67	0.31	0.31	0.11
ActiveMQ	79.17	27.33	54.33	0.00	54.33	24.83	24.33	0.00	24.33	3.00	0.45	0.45	0.12
Lucene	17.33	4.00	10.83	1.00	11.83	5.50	3.67	0.00	3.67	0.33	0.36	0.32	0.05
commApp	161.50	20.50	20.33	63.33	83.67	77.83	6.50	7.50	14.00	6.50	0.34	0.17	0.09
openPDC	73.33	5.00	25.67	6.33	32.00	41.33	3.33	1.33	4.67	0.33	0.13	0.15	0.01
Eclipse	484.00	229.33	368.00	53.67	421.67	62.33	199.33	11.00	210.33	19.00	0.54	0.50	0.30

Table A.3 – Average of LOC, Actual defect and defect density for both Cycle and Not-In-Cycle groups

System	$KLOC_{inC}$	$KLOC_C$	$KLOC_{NC}$	D_{inC}	D_C	D_{NC}	D	DD_{inC}	DD_C	DD_{NC}
Class										
Camel	60.71	66.30	10.1	22.30	20.83	2.33	22.33	0.34	0.32	0.23
ActiveMQ	66.63	77.82	44.05	37.17	39.83	8.67	44.00	0.57	0.52	0.19
Lucene	38.29	46.08	20.48	4.33	5.50	3.00	8.33	0.13	0.13	0.14
commApp	143.26	224.94	81.20	20.83	24.33	7.67	25.67	0.15	0.12	0.10
openPDC	38.33	63.17	34.93	3.00	4.00	4.67	7.00	0.08	0.06	0.12
Eclipse	664.17	851.78	179.89	-	-	-	-	-	-	-
Package										
Camel	69.95	70.62	5.79	20.00	20.17	3.33	22.33	0.29	0.29	0.59
ActiveMQ	86.70	86.70	35.17	41.00	41.00	4.67	44.00	0.48	0.48	0.13
Lucene	58.74	58.75	7.81	7.83	7.83	0.5	8.33	0.14	0.14	0.04
commApp	98.04	224.40	81.74	17.67	23.67	8.17	25.67	0.19	0.11	0.11
openPDC	58.05	70.16	27.94	5.33	6.67	0.33	7.00	0.09	0.10	0.01
Eclipse	936.59	997.25	47.96	-	-	-	-	-	-	-

[21] inC = in-cycle; DC = depend-on-cycle; C = (inC ∪ DC); NC = non-cycle; N = Number of Components; X_d = Defective (X), where X can represent inC, C, DC, or NC, D = Total defect in the system; D_X = Total defect for X group; $DD_X = D_X/KLOC_X$, Defect density of X group; $P(X_d) = X_d/X$, proportion of defective X group



Camel - r2.10.2
Incoming dependency
Outgoing dependency
Camel - r2.10.2
#Edges
#Vertices (Nodes)
(a)
ActiveMQ - r5.7.0
Incoming dependency
Outgoing dependency
ActiveMQ - r5.7.0
#Edges
#Vertices (Nodes)
(b)
Lucene - r4.0.0
Incoming dependency
Outgoing dependency
Lucene - r4.0.0
#Edges
#Vertices (Nodes)
(c)
commApp - r4.2.4
Incoming dependency
Outgoing dependency
commApp - r4.2.4
#Edges
#Vertices (Nodes)
(d)
openPDC - r1.5
Incoming dependency
Outgoing dependency
openPDC - r1.5
#Edges
#Vertices (Nodes)
(e)

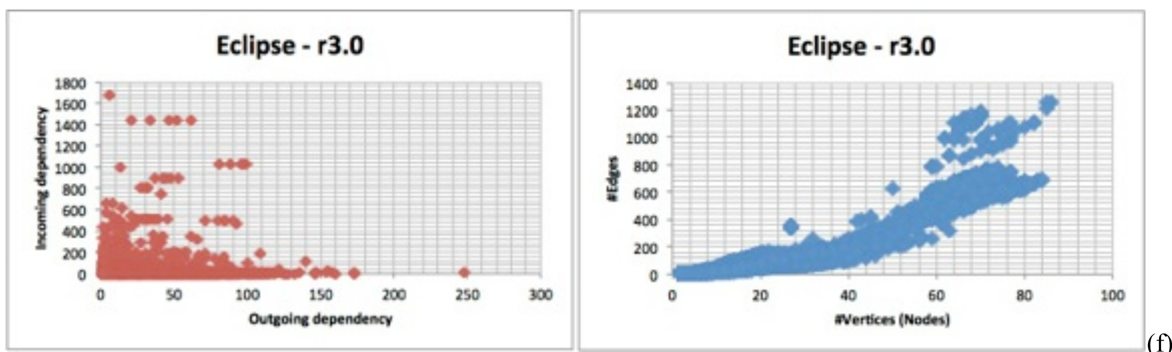
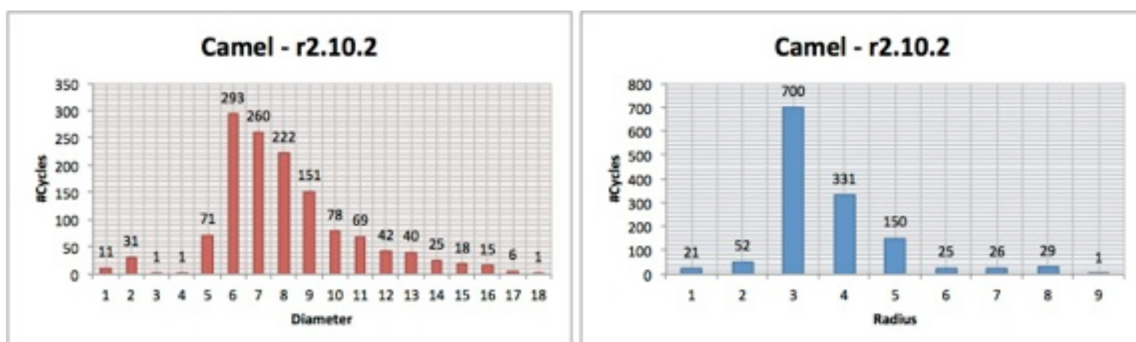
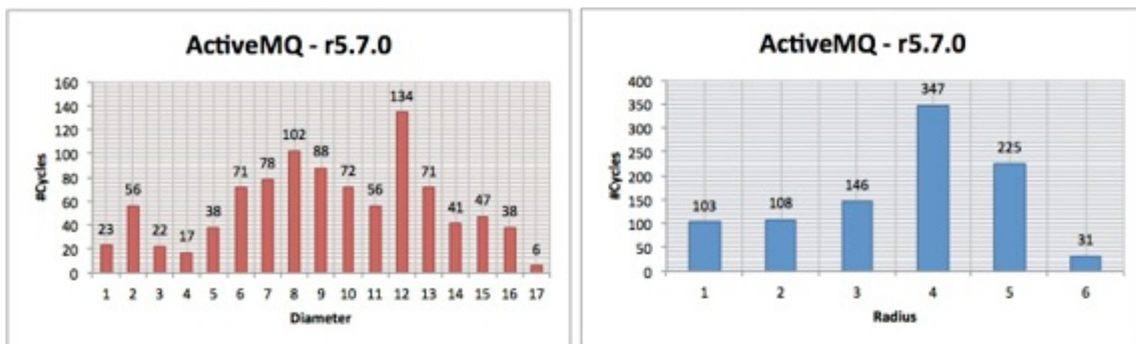
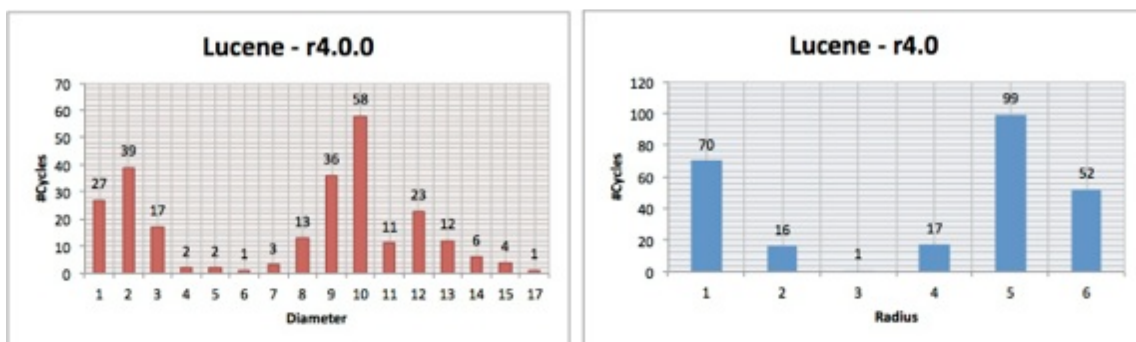


Figure A.1 - Scatter plots of <V, E_{out}, E_{in}>[22] and <V, E>[23] for cyclic dependency graphs for the last release of (a) Camel (b) ActiveMQ (c) Lucene (d) commApp (e) openPDC (f) Eclipse







---

[22] E_{out}: Outgoing edge from a component and E_{in}: Incoming edge from a component. Each dot in the chart represents a single component (class file) and shows the number of E_{out} and the number of E_{in} at the same time

[23] Each dot in the <V, E> graph represent a cyclic dependency graph with a number of nodes and edges

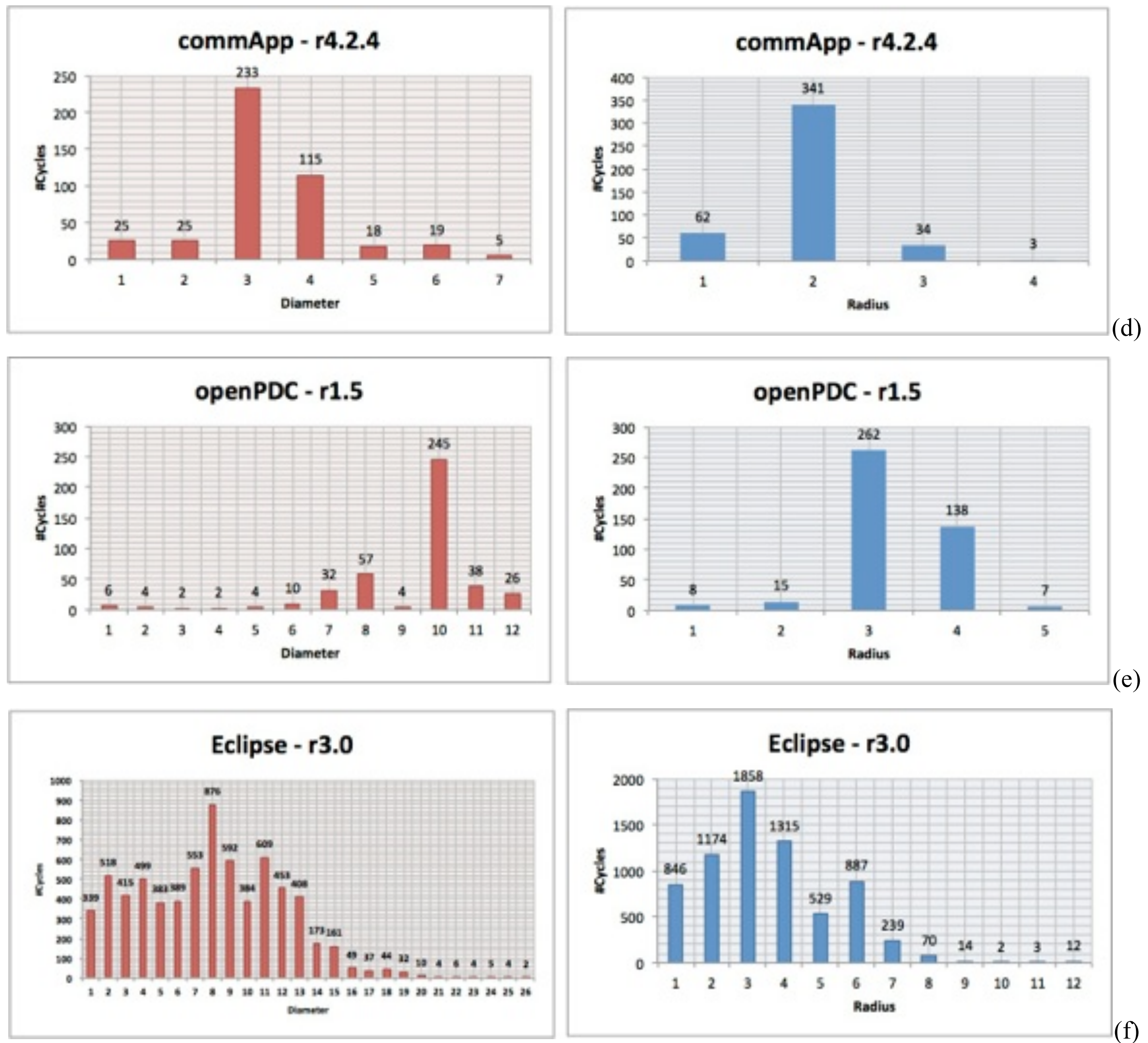


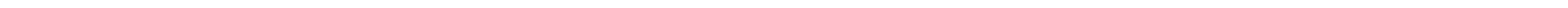
Figure A.2 – Diameter[24] and Radius[25] vs. number of Cycles for the last release of (a) Camel (b) ActiveMQ (c) Lucene (d) commApp (e) openPDC (f) Eclipse

[24] Diameter is the maximum eccentricities of the nodes in the graph

[25] Radius is the minimum eccentricities of the nodes in the graph

Appendix B – R Code

```
#Author: Tosin Daniel Oyetoyan
#**********************************************
#Normality shapiro.test
#T.Test
#Non-Parametric wilcox.test
#**********************************************
#x1 -> inCycle
#x2 -> Not-In-Cycle
#**********************************************
my.tests <- function(){

    data <- read.table("/Users/odtosin/Documents/activemq/ndata/activemq-
parent-5.7.0_FilestatCycle.csv", header=TRUE, sep=",", na.strings="NA", dec=".", strip.white=TRUE);
    #Number of defective components
    x1 <- c(data[9,2], data[9,3], data[9,4], data[9,5], data[9,6], data[9,7]);
    x2 <- c(data[7,2], data[7,3], data[7,4], data[7,5], data[7,6], data[7,7]);
    p <- x1-x2;
    numS <- shapiro.test(p);
    if(numS$p.value > 0.05){
        numT <- t.test(x1,x2,paired=TRUE, conf.level=.95, alternative="greater")
    } else {
        numT <- wilcox.test(x1,x2,paired=TRUE, alternative="greater")
    }

    #Proportion of defective components
    x1 <- c(data[12,2], data[12,3], data[12,4], data[12,5], data[12,6], data[12,7]);
    x2 <- c(data[13,2], data[13,3], data[13,4], data[13,5], data[13,6], data[13,7]);
    p <- x1-x2;
    propS <- shapiro.test(p);
    if(propS$p.value > 0.05){
        propT <- t.test(x1,x2,paired=TRUE, conf.level=.95, alternative="greater")
    } else {
        propT <- wilcox.test(x1,x2,paired=TRUE, alternative="greater")
    }

    #Actual Number of defect
    x1 <- c(data[22,2], data[22,3], data[22,4], data[22,5], data[22,6], data[22,7]);
    x2 <- c(data[23,2], data[23,3], data[23,4], data[23,5], data[23,6], data[23,7]);
    p <- x1-x2;
    defS <- shapiro.test(p);
    if(defS$p.value > 0.05){
        defT <- t.test(x1,x2,paired=TRUE, conf.level=.95, alternative="greater")
    } else {
        defT <- wilcox.test(x1,x2,paired=TRUE, alternative="greater")
    }

    #Defect density
    x1 <- c(data[26,2], data[26,3], data[26,4], data[26,5], data[26,6], data[26,7]);
    x2 <- c(data[27,2], data[27,3], data[27,4], data[27,5], data[27,6], data[27,7]);
    p <- x1-x2;
    ddS <- shapiro.test(p);
    if(ddS$p.value > 0.05){
        ddT <- t.test(x1,x2,paired=TRUE, conf.level=.95, alternative="greater")
    } else {
        ddT <- wilcox.test(x1,x2,paired=TRUE, alternative="greater")
    }

    head <- c("#DefSh","#DefT","propS","propT","defS","defT","ddS","ddT");
    res <- c(numS$p.value,numT$p.value,propS$p.value,propT$p.value,defS$p.value,defT$p.value,ddS
$p.value,ddT$p.value);
    cres <- cbind(head,res);
    write.csv(cres,file = "/Users/odtosin/Documents/activemq/ndata/TTests-Cycle.csv", quote = FALSE);
}
```



P3: Criticality of Defects in Cyclic Dependent Components

Published: In Proc. 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), Eindhoven, Netherlands, 2013, pp. 21-30

Criticality of Defects in Cyclic Dependent Components

Tosin Daniel Oyetoyan[1], Daniela Soares Cruzes[1,2] Reidar Conradi[1]
[1]Department of Computer and Information Science
Norwegian University of Science and Technology Trondheim, Norway
{tosindo, dcruzes, conradi}@idi.ntnu.no
[2]SINTEF
Trondheim, Norway
danielac@sintef.no

***Abstract*—(*Background*) Software defects that most likely will turn into system and/or business failures are termed critical by most stakeholders. Thus, having some warnings of the most probable location of such critical defects in a software system is crucial. Software complexity (e.g. coupling) has long been established to be associated with the number of defects. However, what is really challenging is not in the number but identifying the most severe defects that impact reliability. (*Research Goal*) Do cyclic related components account for a clear majority of the critical defects in software systems? (*Approach*) We have empirically evaluated two non-trivial systems. One commercial Smart Grid system developed with C# and an open source messaging and integrated pattern server developed with Java. By using cycle metrics, we mined the components into cyclic-related and non-cyclic related groups. Lastly, we evaluated the statistical significance of critical defects and severe defect-prone components (SDCs) in both groups. (*Results*) In these two systems, results demonstrated convincingly, that components in cyclic relationships account for a significant and the most critical defects and SDCs. (*Discussion and Conclusion*) We further identified a segment of a system with cyclic complexity that consist almost all of the critical defects and SDCs that impact on system's reliability. Such critical defects and the affected components should be focused for increased testing and refactoring possibilities.**

Keywords—defect severity; dependency cycles; defect distribution; defect-prone components; software reliability; empirical study

1. Introduction

According to [1], software reliability is the probability that software will not cause a system to fail (i.e. behave incorrectly) for a specified time under specified conditions. A system failure may be the result of a software fault/defect [1]. Moreover as noted by [2], software does not "wear out" after some period of proper operation as hardware components do. In addition, defects in software systems may not be apparent over time but when they are exposed, they act like a hidden bomb [2].

There are many cases of system failures due to software defects. For example [2]: The "STS-126 Shuttle Software Anomaly-2008"; The "Air Traffic Control Communication Loss – Los Angeles 2004"; The "Widespread Power Outage in the Northeast in Northern Ohio – 2003"; the "Ariane 5 Failure Forty Seconds After Lift-Off – 1996". In all these cases, the failures were caused by defects that we could classify to be of critical severity because of their impact on these systems. Critical defects are not limited to system and/or hardware failures. They may also be associated with many business failures. Many examples exist, for instance [3]; recently, a "Faster Payment System" at Lloyds bank, meant to speed up payment, was hit by critical defects and ironically delayed wage and bill payments and caused duplicate charges for PayPal users. Similarly, a trading software glitch was caused by critical defects that resulted in a $461.1million loss for Knight Capital last year [3].

Many of today's software systems are overly complex and indeed highly interconnected. The higher the complexity of a system, the more difficult it is to maintain and the higher the risk of accidental and unexpected failures [4]. One area of such software complexity is dependency cycles that are formed by direct or indirect decisions during software development and evolution. Dependency cycles among components are notorious for extremely increasing coupling complexity among interconnected components [5, 6]. Despite numerous claims that cycles inhibit software quality attributes such as extensibility, understandability, testability, reusability, build-ability, maintainability and reliability [7-9], evidence shows that they are widespread in real life software systems [9-13]. Intuitively, we expect that since cycles increase coupling

complexities among components [5, 6], then it should have a positive correlation with the most defects. In fact, by performing an empirical study on six software systems, we confirmed this conjecture of higher defect profiles for cyclic-related components in all the six systems [14].

On the other hand, the study by Adams [15] showed that removing large number of defects may have a trivial effect on reliability. As pointed out in [15, 16], the most number of latent defects lead to very rare failure in practice while the vast majority of observed failures are caused by a relatively tiny number of defects [17, 18]. Therefore showing that it is not the number of defects, rather their severity that matters. A critical severity defect usually points to a fatal error that results into system, hardware and/or business failures whereas low severity defects mostly points to some cosmetic issues.

Since cycles reduce our cognitive ability to reason about interconnected components, we conjecture that the most severe defects detected in such systems may have an "undisclosed" relationship with the cyclic complexity. Our hypothesis in this study therefore, is that, components in cyclic relationships have higher likelihood of containing significant number of highest severity defects and severe defective components than those not in any cyclic relationships.

The remainder of the work is structured as follows: In section II, we lay the background for our study. In section III, we detail our empirical setup. Section IV presents the

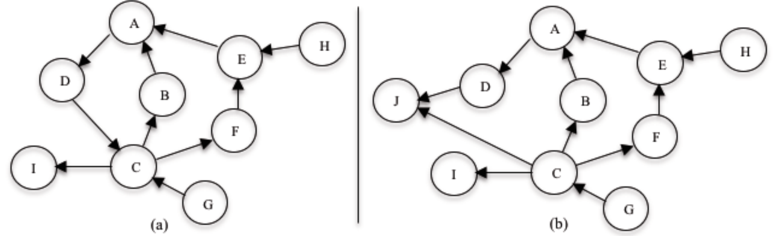


Fig. 5. (a - b) - Cyclic Dependencies and propagation effect on components in a software network [14]

results of this work while section V discusses our findings. In section VI, we draw out the threats to the validity of the results. Lastly, in section VII, we give the conclusion with a note on future studies.

2. Background

In a software system, a component *X* is said to have dependency on another component *Y* if *X* requires *Y* to compile or function correctly [19]. For formal representation of a dependency graph for an object-oriented (OO) program, we borrow two definitions from [12] and state these as follows:

Definition 1. An edge labeled digraph G = (V, L, E) is a directed graph, where $V = \{V_1, ..., V_n\}$ is a finite set of nodes, $L = \{L_1, ..., L_k\}$ is a finite set of labels, and $E \subseteq V \times V \times L$ is the set of labeled edges.

Definition 2. The object relation diagram (ORD) for an OO program P is an edge-labeled directed graph (digraph) ORD = (V, L, E), where V is the set of nodes representing the object classes in P, L = {*I*, *Ag*, *As*} is the set of edge labels representing the relationships (Inheritance, Aggregation and Association) between the classes and $E = E_I \cup E_{Ag} \cup E_{AS}$ $E = E_I \cup E_{Ag} \cup E_{As}$ is the set of edges.

We concern ourselves in this study with dependency at the physical level, that is, both files (top-level classes) and packages since we can infer strong dependencies from physical relationship [8]. As illustrated in Figure 1a, cyclic dependency is formed when components depend on one another in a circular manner. This relationship covers both direct and indirect connection between components. Cyclic relationships increase coupling complexities and thus have the potential to propagate defects in a network [20].

A hypothetical case as depicted in Fig. 1 (a-b) demonstrates such effect. From Fig. 1(a), assume that component **I** contains some defects. We can further assume that the rest of the components **A – H** will have a certain probability to inherit the defect from **I**, since they are directly and

indirectly dependent on **I**. To reduce the likelihood of defect propagation e.g. in Fig. 1(b), let us say that, a new component **J** is created so that components **D** and **C** depend on **J** directly thereby breaking the cyclic effect. By performing such a refactoring, the effect of possible defect propagation is reduced to only component **G**.

For the purpose of this paper, we use some of the terms defined in [14]: Assume a component c ∈ System P then:

Component's Children: Components that are directly and transitively dependent on c. E.g. in Fig. 1(a), All the components except component **I** are directly or transitively dependent on **A**. Components **G**, and **H** have no children. We use **TChildren** for both direct and transitive children and **DChildren** for direct children. For example, **DChildren(A)** = {B, E}.

Component's Parent: All components that c is both directly and transitively dependent upon. We use **TParent** for both direct and transitive parents and **DParent** for direct parents. For instance, **TParent(G)** = {A, B, C, D, E, F, I} and **DParent(G)** = {C}.

Component In-Cycle: Component c is said to be in cycle, if it has at least one parent that is the same as one of its children. E.g. **B** is in cycle because its parent **A** is also one of its children.

Component Depend on In-Cycle Component: Component c is said to depend on another in-cycle component if at least one of its direct parents is in cycle. **G** and **H** are examples of components that depend on **In-Cycle** components **C** and **E** respectively.

Associated Defect: Two components have associated defect if a specific defect affect both components. We use defect ID to track associated defect between components.

Some metrics such as CBO, RFC [21], CyclicClassCoupling[26] and dwReach [22, 23] are of interest but not useful for our purpose since they cannot classify whether a component is involved in cyclic components indirectly. Therefore, we describe the cyclic metrics and notations [14] relevant for our study. Consider a set of components, C in an object-oriented system. For each component c ∈ C:

- **Component In-Cycle:**

  inCycle: boolean
  $\exists p : p \subseteq (TParent(c) \wedge TChildren(c))$
  $\{\forall c.\, inCycle(c) \leftrightarrow p \neq \emptyset\}$
  $where$:
  $inCycle(c)$ *denotes* **c** *to be in a cyclic dependency*

- **Depend On Cycle:**

  depOnCycle: boolean
  $\exists x : x \in DParent(c)$
  $\forall c.\, depOnCycle(c) \leftrightarrow (\neg inCycle(c) \wedge inCycle(x))$
  $where$:
  $depOnCycle(c)$ *denotes c depends on inCycle component x that is a direct parent of c*

Cycles among components have been claimed to be detrimental to understandability [7], production [8, 24], marketing [8], development [8, 24], usability [8, 24], testability [8], integration testing [10-12, 19, 25], reusability [24], extensibility [9] and reliability [8].

Although, it has been stated and implied, to date, it appears that only one study [13] has performed an elaborate empirical study of cycles on many software systems at the class level. The result shows that almost all the 78 Java applications they analyzed contain large and complex cyclic structures among their classes.

In a recent study [14], we established that components in cyclic relationships, either directly or indirectly, have significantly more defect-prone components than those not in any cyclic relationships. The four hypotheses we tested on six different and non-trivial systems confirm that:

i. Components in cycles have higher likelihood of defect-proneness than those not in cyclic relationships.
ii. The higher number of defective components is concentrated in cyclic dependent components.
iii. Defective components in cyclic relationships account for the clear majority of defects in the systems investigated.
iv. The defect density of cyclic related components is sometimes higher than those in non-cyclic relationships.

[26] This metric counts the number of direct cyclic connections between two classes. For instance, C_1 depends on C_2 and C_2 depends on C_1

However, as found in [15, 16], this is not sufficient to focus testing resources. We are compelled to find out if this majority of defects and defect-prone components are also the majority in both critical defects and severe defective components.

Zhou and Leung [26], Shatnawi and Li [27] and Singh et al. [28] demonstrated that object-oriented design metrics could predict defect-proneness of classes based on defect severity. Bhattacharya et al. [29] on the other hand, revealed that graph based metrics are capable of predicting defect severity, maintenance effort and number of defects at both the function and module levels. In similar direction like these studies but not concerned with prediction of defect-proneness of components, Menzies and Marcus [30], Lamkanfi et al. [31], Iliev et al. [32] and Yang et al. [33] have all focused efforts on models that could assign severity levels to defect reports.

These studies focused on (a) predicting defect-prone components based on their severity of defects using both the OO design and graph metrics and (b) predicting the severity of reported defects and not the affected components. Our work differs from these efforts in the sense that: None of these studies analyzed defect severity using cyclic complexity. In our study we identified cyclic dependent components as a set within software components that consists the majority of critical defects and defect-prone components with such critical defects.

This study extends our previous study [14] and the findings are aimed to add significance to the need to collect cycle metrics and focus on defect-prone cyclic related components with critical defects for increased testing activities and refactoring possibilities.

3. Empirical Setup

Our goal in this work is to determine the severity of defects in the cyclic dependent components of the systems under study. As explained in [7, 8, 24], cyclic dependencies are better studied at physical design levels such as the source file (compilation unit) and package levels (Organizational and deployable units), since this type of dependencies is stronger than logical dependencies [8]. In addition, previous empirical studies [13] on cyclic dependency have performed analysis at the file levels. Furthermore, when developers resolve defects, they usually log the changes at the file level and thus have file to defect mapping. Based on the above reasons, we identify relationships and dependencies at the compilation units (top-level classes for Java) and at the package level.

We perform our evaluation in two ways: First, we use the set of metrics built around cyclic dependency relationships proposed in our previous study [14] to mine software components and classify them into two groups, “Cyclic” and “Non-Cyclic”. Second, we statistically evaluate the severity of defects from cyclic related components and non-cyclic related components.

A. Systems under study

We choose two systems primarily because of their criticality to the environments where they operate. First, we analyze an industrial Smart Grid application, a type of system of systems (SoS) applications. Our motivation for the choice of this case study is that, as a critical infrastructure, the availability and reliability of the Smart Grid is crucial to its safety and security. Smart Grid represents the injection of Information and Communication Technology (ICT) infrastructure to the electricity grid to allow for bi-directional flow of energy and information [34].

The system under study is a distribution management system designed to monitor and plan the Grid operations. It provides real-time operational support by continuously receiving status data from the power grid. The system has been in development for about six years and we have analyzed six post releases (field and operational) of this application. It is mostly developed with C# programming language with .NET framework. As listed in Table I, it has a size of approximately 341KLOC and contains 1203 class files and 2142 classes as of version 4.2.4.

Furthermore, we choose Apache-ActiveMQ[27], a very powerful and open source messaging and

[27] http://activemq.apache.org/

enterprise integration pattern server. Our motivation for this choice is that systems that provide integration platform for other applications are very critical and form the backbone for these applications. The security and reliability of the applications running on this platform depend mostly on it. We consider this system to be non-trivial. As listed in Table I, the CORE module we analyzed as at release 5.7.0 (penultimate release) has about 136.22KLOC, containing 1517 class files and 1665 classes.

B. Research Hypotheses

The hypotheses investigated in this study are as follows:

- *H_A: Cyclic dependent components have significantly higher number of highest severity defects than non-cyclic components.*
- *H_B: Cyclic dependent components have significantly higher severe defect-prone components than non-cyclic components.*

A ***severe defect-prone component*** (SDC) is defined as a component within the top 25% with the highest severity defects.

To test our hypotheses, either a t-test or non-parametric test [4] such as Wilcoxon signed rank test will be applicable depending on whether our data sample is normally distributed or not. Lastly, we test the difference in mean between both groups for significant difference that is greater than zero. Three categories are identified for both groups based on our hypotheses:

i. Number of critical severity defects recorded in each group

ii. Number of defect-prone components with critical severity defects.

iii. Number of severe defect-prone components in each group.

For these three categories, we test the hypothesis (1-tailed significance test):

- H_0: $\mu_C \leq \mu_{NC}$ (The mean of cyclic group is significantly less than or equal to the non-cyclic group)
- H_1: $\mu_C > \mu_{NC}$ (The mean of cyclic group is significantly higher than the non-cyclic group)

A. Data collection

We have collected data for six releases of two important applications. We describe in each subsection the details of our approach: (1) to collect the data from the defect repository, (2) to map the class files to the defects, (3) to aggregate the defect counts at the class-file and the package level, (4) of ranking components by severity of defect and (5) of dependency data collection

1) Defects collection from the defect tracking system (DTS)

We have collected defect data from both the HP-QC DTS and JIRA DTS. A Defect repository gives typically a high level overview of a problem report. For example, typical attributes of the HP-QC defect tracking system (QC-DTS) are the Defect ID, severity of the defect, the type of defect, date defect is detected, the module containing the defect, the version where defect is detected, and the date the defect is fixed. Our first step is to determine the defects that affect each version of the system. In the HP-QC, we use "Detected in Version(s)" and in Apache JIRA DTS, we use the "Affects version" field to filter all bugs that affect a particular version of the system. A certain defect may keep re-occurring and span several versions of a system (persistent defects [35]). We include such defects in all the versions they affect. Next, we filtered out "duplicate", "Not a problem", "Invalid", "Enhancement" and "Task" cases from the resolution/Defect Type field.

1) Method to map class files to defects

Version repository, on the other hand, is a configuration management system used by the developers to manage source code versions. The version system provides historical data about the actual class file that is changed and/or added as a result of corrective action (defect fixes), adaptive, preventive and perfective actions [36]. Thus, the SVN/CVS provides a detailed granularity level to know which source file(s) in the module(s) are

TABLE VIII. SOFTWARE SOURCE CODE AND DEFECT DATA

Release	Date	#Pkg	#Class-File	#Class	KLOC	#Defective Class-File	#Defects
Apache-ActiveMQ							
5.7.0	Nov 22 2012	82	1517	1665	136.22	35	68
5.6.0	Jun 15 2012	83	1505	1649	133.25	88	102
5.5.1	Oct 16 2011	78	1331	1472	118.27	54	76
5.5.0	Apr 01 2011	78	1331	1472	118.27	115	105
5.4.2	Dec 02 2010	77	1258	1393	113.01	80	66
5.4.1	Sept 21 2010	77	1256	1386	112.20	79	63
CommApp							
4.2.4	Nov 14 2012	191	1203	2142	341.83	29	14
4.2.2	Oct 12 2012	191	1199	2134	339.78	49	18
4.1	Aug 17 2012	171	1002	1884	316.22	60	42
4.0.1SP4	Apr 11 2012	141	904	1650	286.99	69	29
4.0.1SP2	Mar 26 2012	142	903	1645	285.89	46	28
4.0	Oct 14 2011	133	849	1546	266.11	137	143

TABLE IX. % OF DEFECTS MAPPED FROM DTS TO SVN

Apache-ActiveMQ		CommApp	
Release	**%Bugs**	**Release**	**%Bugs**
5.7.0	85.3	**4.2.4**	71.4
5.6.0	93.1	**4.2.2**	83.3
5.5.1	77.6	**4.1**	85.7
5.5.0	82.9	**4.0.1SP4**	69.0
5.4.2	80.3	**4.0.1SP2**	64.0
5.4.1	85.7	**4.0**	51.7

changed to fix a reported defect. A common way to figure out what operation is performed on the source file is to look at the message field of the SVN commit. When developers provide this information with the bug number and/or useful keywords (e.g. bug or fix), it is possible to map the reported defect with the actual source file(s) [37, 38]. In some cases, not all bug commits in the version repository contain the bug number or useful keyword in the message field. In the past, researchers have approached this situation by mapping from defect repository to the version repository [38, 39].

We have used both approaches to map defect from the HP-QC and JIRA DTSs to the code changes. The resolution date allows us to map some of the untagged commits in the version system to the resolved bugs. Overall, for the six releases of each system, we mapped an average of 84.2% for Apache-ActiveMQ and 71% for **CommApp** (see Table II). From these percentages, we consider only defects that are associated with source files of the analyzed modules and ignore defects for non-source files, test source files and source files of other modules not analyzed. Consequently, the reported defect figures in the results section are fractions of the mapped percentages.

2) Aggregating number of defects per class file and per package

In a release, it is possible that multiple reported bugs be associated to one class file. The unique defect ID is thus appropriate to compute the number of defects fixes that affect a class file and a package. From the mapped change data, we look up each file and determine the total of defects per file by counting the number of unique defect ID in this release. At the package level, we aggregate the unique defect IDs for each class file in the package. As demonstrated in Fig. 2, based on the defect ID, File1, File2 and File3 have 2 defects each and Pkg 1 has a total of 3 defects although it contains 3 files with 2 defects each. The unique defect-ID shows that for pkg1, only 3 defects are fixed.

3) Ranking of components by defect severity

The HP-QC DTS of CommApp uses four values to describe the severity of each recorded defect while in JIRA five values are used. The severity is determined based on the impact of the defect on the system and the business. From our observation of the message logs in both DTSs, defect severities with "blocker" and/or "critical" values relate strongly to reliability, performance and/or security issues in the system. For instance, in CommApp, an example is: "*Database running at 100% CPU*". An example in ActiveMQ is:

"*Network bridges can deadlock when memory limit exceeded*". For HP-QC, a defect can be ***critical***, ***major***, ***average*** or ***minor***. In JIRA, a defect can be ***blocker***, ***critical***, ***major***, ***minor*** or ***trivial***. Because of the few numbers of ***blockers*** and ***critical*** defects in the defect data for Apache-ActiveMQ, we decided to merge these two to form only ***critical*** category. We transform the severity scales in both defect-tracking systems to map ***critical***, ***high***, ***medium*** and ***low***.

We want to be able to rank the components based on their number of the highest severity defects. This presents the possibility to be able to evaluate ***severe defect-prone components*** (SDC). Unlike other studies [26-28] that have developed multiple models to predict two or three categories of defect severity, we can only devise an approach to have a single ranking of component based on its most severe defects. We describe this method in this section since we believe it can find practical use by other researchers and practitioners.

We keep in mind that a component can have many defects and therefore contain different severity values (i.e. different severities distributed over a component). For instance, a component can have 3 defects in this order {Critical=1, High=1, Medium=0, Low=1}. To rank according to the highest severity of defects requires that we make some transformation to give the highest weights to components according to their most severe defects.

We describe the transformation process we use for this purpose:

1. Given ***n*** number of components and ***m*** number of defect severities, we form an ***mxn*** matrix, where the column elements in the matrix stand for the severity values of a component in their order of severity.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix}$$

2. We form a new matrix B as follows; for each column element, starting from the first element, replace all elements below with zero iff the element above is greater than zero.

$$\forall a_{i,j} \; if \; a_{i,j} > 0, a_{k,j} = 0, for \; k = i+1, \dots, m$$
$$i \in [1{:}m-1]$$

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & \cdots & b_{1n} \\ b_{21} & b_{22} & b_{23} & \cdots & b_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ b_{m1} & b_{m2} & b_{m3} & \cdots & b_{mn} \end{bmatrix}$$

3. Form a weight row vector W of ***1xm*** dimension containing the sum of the maximum element of each row below the kth row in B. The last column element in W is kept as 0:

$$w_k = \sum_{i=k+1}^{m} \max_{j=1,\cdots,n}(b_{i,j}), for \; k = 1, 2, \cdots, m-1$$

$$W = [w_1, w_2, \cdots, w_{m-1}, 0]$$

4. Form a new ***mxn*** matrix W_D, where W is the diagonal elements and all other elements are zero

$$W_D = \begin{bmatrix} w_1 & 0 & 0 & \cdots & 0 \\ 0 & w_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

5. Form matrix C by dividing each element in B by itself

$$C = \begin{bmatrix} \frac{b_{11}}{b_{11}} & \frac{b_{12}}{b_{12}} & \frac{b_{13}}{b_{13}} & \cdots & \frac{b_{1n}}{b_{1n}} \\ \frac{b_{21}}{b_{21}} & \frac{b_{22}}{b_{22}} & \frac{b_{23}}{b_{23}} & \cdots & \frac{b_{2n}}{b_{2n}} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{b_{m1}}{b_{m1}} & \frac{b_{m2}}{b_{m2}} & \frac{b_{m3}}{b_{m3}} & \cdots & \frac{b_{mn}}{b_{mn}} \end{bmatrix}$$

1.

6. Lastly, compute D = W_D*C + B

For example, with components; **c_1**: {Critical=2, Major=1, Average=0, Minor=1}, **c_2**: {Critical=0, Major=1, Average=3, Minor=0}, **c_3**: {Critical=0, Major=3, Average=0, Minor=0} and **c_4**: {Critical=0, Major=0, Average=0, Minor=1} gives matrix:

$$A = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 1 & 1 & 3 & 0 \\ 0 & 3 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

Following the transformation steps II-VI yields matrices:

$$B = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad W_D = \begin{bmatrix} 4 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$D = \begin{bmatrix} 6 & 0 & 0 & 0 \\ 0 & 2 & 4 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

From the matrix D's result, **c_1** has the highest weight of 6, followed by **c_3** with a weight 4, then **c_2** with a weight 2 and lastly **c_4** with a weight of 1.

4) Dependency data collection

We have developed a small Java tool to extract source files dependency data [14]. The source files are downloaded from the version repository. Organizational rules in Java source file are substantially different from C# source file. A Java source file has a one-one mapping from file to top-level class and it is not allowed to define another top-level class in a Java file. In addition, the top-level class must have the same name as its enclosing file. Also, there is a one-zero or one-one mapping from file to package; a maximum of one package can be defined in a Java file. Finally, a Java class may contain nested classes (one to many relation). In C#, multiple relations are possible. A file can contain many top-level classes and many top-level namespaces can also be defined in a file. It is also possible that a class contains nested classes and a namespace can equally contain nested namespaces. Unlike Java file, the file name does not need to match any of the classes defined in it, although, good practices suggest to have filename as the same as a top-level class.

Since the compilation unit for both Java and C# is the source file and we are considering dependencies at the physical level [8], we decide for the following:

1. A dependency on any class in a source file implies a dependency on the source file.
2. The cyclic metric for a class is computed using dependencies that cross compilation units (source files). We skip cycles that are formed among classes within a source file.

We use the "USES" relations [8], which we have defined earlier as **DParent** and apply them to the two systems. We ignore all external library types (e.g., .NET and Java API) that developers have no access to their source codes since it is practically impossible for these external classes to form cycles with internal application's classes. Fig. 3 shows an example of the actual dependencies for **MyClass** and **mypackage** components. In order to collect other nodes (classes) to which **MyClass**

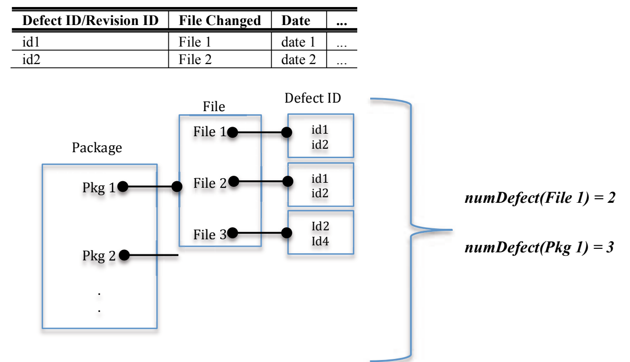


Fig. 6. Aggregating defect count at the package and file level [14]

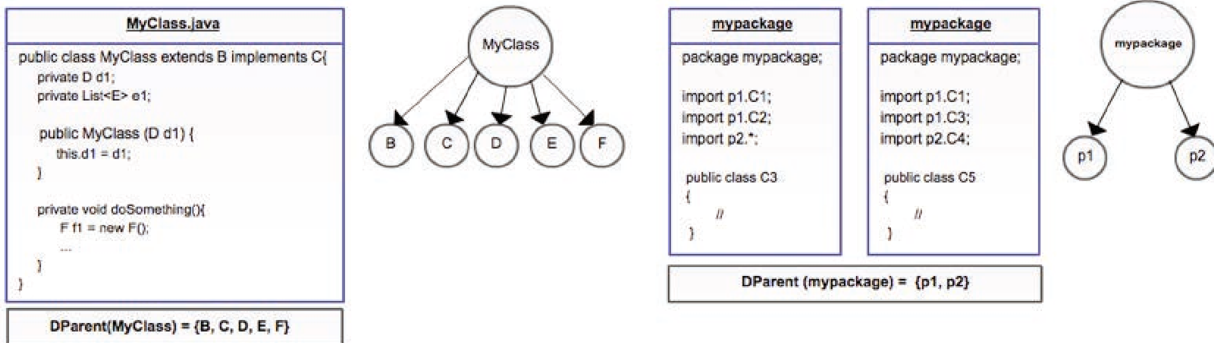


Fig. 7. (a) Class source data (b) Dependency Graph for Class (c) Package source data (d) Dependency Graph for package [14]

is connected to requires that we scan the text of **MyClass**. The edge between **MyClass** and other **DParent(MyClass)** nodes is a directed path (without label, L) from **MyClass** to each node in the **DParent** set (Fig. 3a-b). In the case of **mypackage** (Fig. 3c-d), the **DParent(mypackage)**, is a set of unique imported packages and is processed from the collected class data.

4. Results

Table III lists the distribution of defect severity for the entire systems. Table IV lists the distribution of defects in each group. The total number of defects (N_D), the number of defects in

in-cycle group (**IC_D**), the number of defects in the *depend-on-cycle* group (**DC_D**), the total number of defects in both groups (i.e. **$C_D = IC_D \cup DC_D$**) and the number of defects in the *non-cyclic* group (**NC_D**) for both class-files and packages. In Table V, we report the results of mining the components (class-files and packages) into *in-cycle* (**IC**), *depend-on-cycle* (**DC**) and *non-cyclic* (**NC**) categories. In Table VI, we list for each severity value and SDC, the percentages of *cyclic-related defect-prone components* (**C_{DPC}**), *non-cyclic defect-prone components* (**NC_{DPC}**), the number of defects in *cyclic* group accounted for by *cyclic defect-prone components* (**C_D**), the number of defects in *non-cyclic* group accounted for by *non-cyclic defect-prone components* (**NC_D**), and the difference between the defects in *cyclic* group and *non-cyclic* group (i.e. **C_D-NC_D** and **NC_D-C_D**).

Table VII reports the hypotheses tests of SDC, defect-prone components with critical severity defects and the number of defects with critical severity in *cyclic* and *non-cyclic* groups.

A. Distribution of Defect Severity

Table III lists the average distribution of defect severity in the six releases of the two systems. In Apache-ActiveMQ, 8% of defects are **Critical** (Blocker + Critical) defects and are distributed in 8% of defect-prone components[28] (DPC). 75% of defects are **High** (Major) severity defects and are spread across 78% of DPC, while 15% of total defects are **Medium** (Minor) severity defects and are distributed in 13% of DPC. Lastly, 2% of all defects are **Low** (Trivial) severity defects and are spread across 1% of DPC. In CommApp, 12% of defects are **Critical** (critical) severity defects and are distributed in 25% of DPC. 45% of defects are **High** (high) severity defects and are spread across 42% of DPC. 36% of defects are **Medium** (average) severity defects and are distributed in 27% of DPC and lastly, 7% of defects are **Low** (low) severity defects and are distributed in 6% of DPC.

Fig. 4 illustrates how much of defect-prone components affected by critical severity defects are accounted for when using the largest-first or the smallest-first[29] prioritization approaches [40]. For both systems, this distribution shows critical severity defects to spread across both DPC with large size and number of defects and those with small size and number of defects. At the top 25%, we could only account for less than 45% of DPC with critical severity defects. This number is, of course, not desirable for critical applications. Even at the top 75%, we are still unable to account for all the DPC with critical severity defects (since the percentage identified is approximately 80%).

In conclusion of this section, we can caution that models that target top k% may not uncover a significant number of defect-prone components affected by critical severity defects. At least this is confirmed in these two applications. There is a need for more studies in prediction methods that focus further on the severity of defects rather than number of defects. This is an additional motivation for us to conduct an investigation into how much of critical severity defects and defect-prone components with critical severity defects are contained in cyclic dependent components.

TABLE X. % (AVERAGE) OF DISTRIBUTION OF DEFECT SEVERITY

Severity	Apache-ActiveMQ		CommApp	
	DPC	#Defect	DPC	#Defect
Critical	8	8	25	12
High	78	75	42	45
Medium	13	15	27	36
Low	1	2	6	7

B. Distribution of defect and DPC in cyclic and non-cyclic groups

Tables IV and V list the average of cyclic data and their defect profiles for the two systems we investigated. On the average, in Apache-ActiveMQ, there are 1366.3 class-files out of which 75.2 are defective. 32.5% of class-files are in-cycle (**IC**) while 15.8% of class-files are dependent on other components in cycles (**DC**).

[28] A defect-prone component as used in this study is defined as components with one or several defects

[29] Largest-first approach assumes that larger components are more defect-prone and therefore ranks components in the order of their highest number of defects while the smallest-first approach, however, assumes that smaller components are relatively more defect-prone

51.7% are not involved in any cyclic relationships (**NC**). Of the total defective class-files, both in-cycle and depend-on-cycle components account for 78.3%, while non-cyclic components account for 21.7%. The system contains an average of 44 defects, of which the **IC** group accounts for 84.5%, the **DC**, 17% and the **NC** 19.8%.

At the package level, there are average of 79.2 packages with 27.3 defective ones. 68.6% of packages are in-cycle while none is dependent on any in-cycle components. 31.4% of packages are not in any cyclic relationships. Furthermore, packages in-cycle account for 89% of the total defective components while non-cyclic packages account for 11%. Out of the total average of 44 defects in the system, **IC** group accounts for 93.2% while **NC** group accounts for 10.7%. As can be seen from these statistics, the cyclic related components in Apache-ActiveMQ account for the clear majority of defective components and number of defects at both the class-file and package levels.

For CommApp, the average class-files totaled 1010, out of which only 65 are defective from 25.7 defects. 24.8% of these class-files are in-cycle and account for 57.4% of defect-prone components and 80.5% of total defects. The **DC** group contains 28.8% of the class-files and account for 26.5% of defect-prone components and 51.4% of the defects. Lastly, the **NC** group

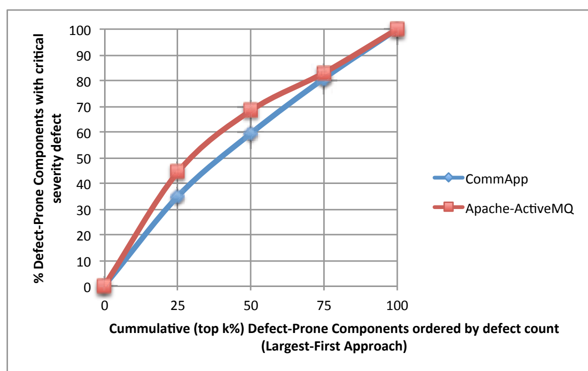


Fig. 8. % of DPC with **critical** defects identified at the top k% of the class-files DPC over six releases

has 46.4% of the total class-files and account for 16.2% of defect-prone components and 30% of total defects.

At the package level, the CommApp contains an average of 161.5 packages of which 20.5 turned defective. 12.6% of these packages are in-cycle and account for 31.7% of defect-prone components and 68.9% of total defects. The **DC** group contains 40.7% of the packages and account for 37.6% of defect-prone components and 57.6% of the defects. Lastly, the **NC** group has 46.6% of the total packages and account for 30.7% of defect-prone components and 31.9% of total defects. As observed from these statistics, the cyclic related components in CommApp also account for the clear majority of both defect-prone components and the number of defects.

C. Distribution of critical defects and SDC in cyclic and non-cyclic groups

We now investigate if this majority in both defect-prone components and number of defects are also the clear majority in the number of critical defects and severe defect-prone components. As listed in Table VI, in Apache-ActiveMQ, the cyclic group of class-files contains 90.4% of SDC[30], that is, defect-prone components in the top 25% based on their number of critical defects while the non-cyclic group has 9.6%. Furthermore, the total percentage of the SDC defects[31] in cyclic group is 96.1% while that of NC group is 10.2%. Also, the cyclic group accounts for 90.3% of the defect-prone components with critical severity defects while the non-cyclic group accounts for 9.7%. At the package level, 97.7% of SDC are in cyclic relationships while 2.3% of SDC are not in cyclic relationships. SDC defects in the cyclic group account for 95% while 5.2% are recorded for non-cyclic group. In addition, all the defect-prone packages with critical severity defects are in cyclic relationships and they account for all the critical severity defects in this system.

In the case of CommApp, the cyclic group of class-files consist 88.6% of SDC and this number accounts for 94.7% of SDC defects. Furthermore, the cyclic group accounts for all (100%) the critical severity defects and contains 82.2% of defect-prone components affected by the critical severity defects. At the package level, cyclic group comprises 65.6% of SDC and accounts for

[30] According to the ranking algorithm and the percentile figure used (Top 25% = 75th percentile), SDC might contain **high** severity defects in addition to **critical** severity defects.

[31] Note that defects in SDC are ranked as the highest severity defects.

80.8% of SDC defects. Also, the cyclic group accounts for all (100%) of critical severity defects and contains 75% of defect-prone packages affected by critical severity defects.

Table VII lists the results of the hypotheses tests. Regarding the first hypothesis H_A, the p-values of 1-tailed test for the two systems and for both types of components (class-files and packages) are less than 0.05. Therefore, we reject the null hypothesis and confirm that the number of critical severity defects in the cyclic defect-prone components is significantly higher than those in the non-cyclic defect-prone components. Regarding H_B, we reject the null hypothesis for H_{B1} at both the class-file and package levels and affirm that defect-prone components with critical defects are significantly higher in the cyclic group than non-cyclic group. The null hypothesis H_{B2} is rejected for all cases except for package-level result for CommApp.

5. Findings and Discussion

First, as demonstrated in the distribution data of Fig. 4, defect-prone components affected by critical severity defects are spread across DPCs. In other words, prioritizing testing activities using the "largest-first" or "smallest-first" [40] approach is not optimal to discover such "first class" candidates that should be prioritized in critical systems. Furthermore, we revealed that all critical severity defects (100%) are located in the packages that are in cyclic relationships. Likewise, between 95% and 100% of critical severity defects can be discovered in the class-files that are in cyclic relationships.

When we look at the defect-prone components affected by those critical defects, we discovered that for cyclic related components, between 82.2% and 90.3% are class-files in cyclic relationships and between 75% and 100% are packages that have cyclic relationships. In addition to these, when we rank components according to the number of their highest severity defects, we found that between 88.6% and 90.4% of defect-prone components (class-files) with the highest severity defects are in cyclic relationships. Also, between 65.6% and 97.7% of packages that are defect-prone and with the highest severity defects are in cyclic relationships.

The set differences $\mathbf{C_D}$-$\mathbf{NC_D}$ and $\mathbf{NC_D}$-$\mathbf{C_D}$ present useful perception into identifying defects that are unique to each group. The findings in this study show that with certainty, we can confirm that 86% (class-files) and 100% (packages) of critical severity defects originate from the cyclic group in Apache-ActiveMQ while 4.8% (class-files) of critical severity defects originate from the non-cyclic group. For CommApp, we are sure that 44.4% (class-file) and 50% (packages) of critical severity defects originate from the cyclic group whereas no critical severity defect can be said with certainty to originate from the non-cyclic group (The set difference NC_D-C_D = null).

One major contribution of this work is that we are able to partition a software dataset into sub sets that allows a maintenance engineer and software testers to look for defect and most especially critical severity defects in the right places. For instance, it is far more efficient to look for highest severity defects in 50% or less of a system´s components than the whole 100%. The cyclic related components in our study range between 48.3% and 53.6% of the class-files and within this range, we can discover between 95.1% and 100% of critical severity defects. Several empirical results already revealed that defect distribution in software systems is skewed and followed the Pareto rule (20-80) [16, 17]. The challenge is higher when dealing with large and complex systems with thousands of components but extremely few defect-prone components. In such situations, prediction models have lower chance of good performance. It is even worse when fewer of those components are associated with critical severity defects, which is the case in many software systems. Finding them can be analogous to looking for a needle in a haystack (see Fig. 4). To reinforce this point, the study in [26] confirmed that prediction based on low severity defect performed better than prediction on high severity defects.

The results in this study are useful to employ for focusing testing resources and refactoring possibilities in both industry and the academia. Many studies [9-13] of dependency cycle in

TABLE XI. SUMMARY OF COMPONENTS #DEFECTS[32] (AVERAGED OVER SIX RELEASES)

System		Class-File				Package			
	N_D	IC_D	DC_D	C_D	NC_D	IC_D	DC_D	C_D	NC_D
ActiveMQ	44	37.2	7.5	39.8	8.7	41	0	41	4.7
CommApp	25.7	20.7	13.2	24.3	7.7	17.7	14.8	23.7	8.2

TABLE XII. SUMMARY OF CYCLIC DATA (AVERAGED OVER SIX RELEASES)

	Class-File								Package							
System	N	N_{DPC}	IC	IC_{DPC}	DC	DC_{DPC}	NC	NC_{DPC}	N	N_{DPC}	IC	IC_{DPC}	DC	DC_{DPC}	NC	NC_{DPC}
ActiveMQ	1366	75.2	444.7	50.4	215.3	8.5	706.3	16.3	79.2	27.3	54.3	24.3	0	0	24.8	3
CommApp	1010	65	250.5	37.3	290.6	17.2	468.8	10.5	162	20.5	20.3	6.5	65.8	7.7	75.3	6.3

TABLE XIII. AVERAGE % OF COMPONENTS IN CYCLIC[33] AND NON-CYCLIC GROUPS AND GROUPED BY DEFECT SEVERITY

	Apache-ActiveMQ						CommApp					
Metric	C_{DPC}	NC_{DPC}	C_D	NC_D	C_D-NC_D	NC_D-C_D	C_{DPC}	NC_{DPC}	C_D	NC_D	C_D-NC_D	NC_D-C_D
Class-Files												
SDC (25%)*	**90.4**	**9.6**	**96.1**	**10.2**	**90**	**4**	**88.6**	**11.4**	**94.7**	**34**	**66**	**5.3**
Critical*	**90.3**	**9.7**	**95.1**	**14.3**	**86**	**4.8**	**82.2**	**17.8**	**100**	**55.6**	**44.4**	**Ø**
High	78.6	21.4	89.4	21.1	79	10.6	84.9	15.1	91.4	35.7	64.3	8.6
Average	75.8	24.2	92.3	18	82.1	7.7	84.5	15.5	96.3	18.2	81.8	3.6
Minor	100	0	100	0	100	Ø	90.6	9.4	100	9.1	91	Ø
Package												
SDC (25%)*	**97.7**	**2.3**	**95**	**5.2**	**94.8**	**5.2**	**65.6**	**34.4**	**80.8**	**61.5**	**38.5**	**19.2**
Critical*	**100**	**0**	**100**	**0**	**100**	**Ø**	**75**	**25**	**100**	**50**	**50**	**Ø**
High	88.4	11.6	94	11	89	6	62.6	37.4	87	40	60	12.9
Average	87.9	12.1	84	15.4	84.6	15.4	82	18	96.3	18.2	81.8	3.6
Minor	100	0	100	0	100	Ø	89.5	10.5	91.1	18.2	81.8	9.1

* Both categories that are focused in this study

TABLE XIV. 1-TAILED TEST FOR COMPARING HIGHEST SEVERITY DEFECTS AND DEFECT-PRONE COMPONENTS IN CYCLIC AND NON-CYCLIC GROUPS

System	CLASS	PACKAGE
	H_A : Test of Number of critical defects in Cyclic vs. Non-Cyclic groups	
	p-value (C)	**p-value (C)**
ActiveMQ	0.0113*	0.0012*
CommApp	0.0214*	0.0299*
	H_{B1} : Test of Number of Defect-prone components with critical defects in Cyclic vs. Non-Cyclic groups	
	p-value (C)	**p-value (C)**
ActiveMQ	0.0337*	0.0052*
CommApp	0.0295*	0.0125*
	H_{B2} : Test of SDC (Top 25%) in Cyclic vs. Non-Cyclic groups	
	p-value (C)	**p-value (C)**
ActiveMQ	0.0051*	0.0001*
CommApp	0.0003*	0.0586

* Significant at $\propto = 0.05$

[32] It is important to note that defects can overlap in both categories since a defect can spread to many components

[33] Note that $C_{DPC} = (IC_{DPC} \cup DC_{DPC})$ and $C_D = (IC_D \cup DC_D)$

software systems suggest that cycles are inherent in real-life systems and appear like a menace we have to live with. We confirm our conjecture in these two systems that dependency cycles contain the highest severity defects. We rush to say that we make no claim to this pattern in all systems, however, we believe this is a step forward to encourage further studies and to understanding dependency cycles, defect-prone components and defect severity.

6. Threats to Validity

We have performed analysis and evaluation of an industrial Smart Grid system and a messaging and enterprise integration pattern server. Therefore, we cannot claim that this kind of pattern or related will be visible in other systems and other domains. As it is with most case studies, we cannot generalize these results across all systems. Further studies will be necessary to compare results across several systems.

Our study is based on static coupling measurements and not dynamic coupling measurements [41]; as such actual coupling among classes at runtime may not be completely captured. This imprecision can occur due to polymorphism, dynamic binding and dead code in the software. This as it may, static code analysis has been found to be practically useful and less expensive to collect [5, 6, 21, 23, 42-44]. In addition, we collect coupling types that are not only based on method invocation. We do not think the data collected based on static code analysis can bias our result in any significant manner.

For this study, we have relied on the defects logged in the defect tracking systems of each application. Our approach of defect data extraction is similar to what other researchers have used in the past [37-39]. Nevertheless, common threats are whether defects logged in the DTS are accurately tagged in the respective code changes in the version systems. In addition, we cannot be sure if all defects are logged in the DTS. Also, there could be cases that the message log of the file that consists a change is not tagged with the bug numbers of the resolved defect. Furthermore, there could be cases of typographical error in the recording of the bug number in the version systems [39] and lastly, it is still possible that duplication will occur.

The recording of defect severity in many defect-tracking systems has been argued to be subjective [45]. We cannot exclude the possibilities of subjective severity records in the DTSs that we have used. However being critical applications and from our investigation of the repositories, most records that impact on reliability, performance and/or security point to the highest severity values (blocker/critical). These are, essentially, the focus in our analysis and therefore, we can rely on the quality of the data to a great degree.

7. Conclusions

We have empirically investigated if defects with the highest criticality and the components impacted by such defects are mostly concentrated in cyclic dependent components. Our findings based on the two non-trivial systems we investigated revealed that DPC with critical defects are spread across the systems. Furthermore, we confirmed our conjecture that cyclic dependent components account for almost all of the critical severity defects and most severe defect-prone components.

Empirical analysis shows that in three out of four cases, all the highest severity defects are found in components that are involved directly or indirectly in cyclic relationships and in the 4th case, over 95% of the highest severity defects are discovered in the cyclic related group. The results in this study have practical use in allocating testing resources to a subset of systems with the highest likelihood of containing the most critical defects. Furthermore, it provides reasoning for refactoring and/or reengineering of especially defect-prone cyclic dependent components with critical defects.

Lastly, it shows a subset of software systems that can be further explored for improved prediction models based on defect severity. As future studies, we aim to conduct a large empirical study of critical systems with well-maintained

repositories to understand if the findings in this study relate to a general pattern in systems with cyclic relationships. Furthermore, dataset imbalance as discussed in Hall et al. [45] is a threat to prediction models' performance. We speculate that we can explore the results of dividing the datasets into these categories to build better models that can predict defect-proneness of components based on defect severity.

References

[1] *IEEE Recommended Practice on Software Reliability.* IEEE STD 1633-2008, 2008: p. c1-72.

[2] Lilley, S., *Critical Software: Good Design Built Right.* NASA System Failure Case Studies, 2012. **6**(2).

[3] Leo, K. *Why banks are likely to face more software glitches in 2013*. [Web] 2013 April 24, 2013]; Available from: http://www.bbc.co.uk/news/technology-21280943.

[4] Fenton, N.E. and S.L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*. 2nd ed. 1997, Boston: PWS Publishing Press.

[5] Briand, L.C., et al., *Predicting fault-prone classes with design measures in object-oriented systems.* Ninth International Symposium on Software Reliability Engineering, Proceedings, 1998: p. 334-343.

[6] Briand, L.C., J. Wuest, and H. Lounis, *Replicated Case Studies for Investigating Quality Factorsin Object-Oriented Designs.* Empirical Softw. Engg., 2001. **6**(1): p. 11-58.

[7] Fowler, M., *Reducing coupling.* Software, IEEE, 2001. **18**(4): p. 102-104.

[8] Lakos, J., *Large-scale C++ software design*. 1996, Redwood City, CA: Addison-Wesley Longman.

[9] Parnas, D.L., *Designing Software for Ease of Extension and Contraction.* IEEE Transactions on Software Engineering, 1979. **SE-5**(2): p. 128-138.

[10] Briand, L.C., Y. Labiche, and W. Yihong. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. in Proc. 12th International Symposium on Software Reliability Engineering, (ISSRE) 2001.

[11] Hanh, V.L., et al., *Selecting an Efficient OO Integration Testing Strategy: An Experimental Comparison of Actual Strategies.* Proc. 15th European Conf. Object-Oriented Programming (ECOOP), 2001: p. 381-401.

[12] Kung, D., et al., *On Regression Testing of Object-Oriented Programs.* Journal of Systems Software, 1996. **32**(1): p. 21-40.

[13] Melton, H. and E. Tempero, *An empirical study of cycles among classes in Java.* Empirical Software Engineering, 2007. **12**(4): p. 389-415.

[14] Oyetoyan, T.D., D.S. Cruzes, and R. Conradi, *A Study of Cyclic Dependencies on Defect Profile of Software Components.* Journal of Systems and Software, 2013. (in press)

[15] Adams, E.N., *Optimizing Preventive Service of Software Products.* IBM Journal of Research and Development, 1984. **28**(1): p. 2-14.

[16] Fenton, N.E. and N. Ohlsson, *Quantitative analysis of faults and failures in a complex software system.* IEEE Transactions on Software Engineering, 2000. **26**(8): p. 797-814.

[17] Boehm, B. and V.R. Basili, *Software Defect Reduction Top 10 List.* Computer, 2001. **34**(1): p. 135-137.

[18] Ebert, C., et al., Defect Detection and Quality Improvement, in Best Practices in Software Measurement. 2005, Springer Berlin Heidelberg. p. 133-156.

[19] Jungmayr, S. Identifying test-critical dependencies. in Software Maintenance. 2002.

[20] Abreu, F.B.E. and W. Melo, *Evaluating the impact of Object-Oriented design on software quality.* Proceedings of the 3rd International Software Metrics Symposium, 1996: p. 90-99.

[21] Chidamber, S.R. and C.F. Kemerer, *A Metrics Suite for Object-Oriented Design.* IEEE Transactions on Software Engineering, 1994. **20**(6): p. 476-493.

[22] Nagappan, N. and T. Bhat, *Technologies for Code Failure Proneness Estimation*, 2007, Microsoft Corporation: USA.

[23] Zimmermann, T. and N. Nagappan, *Predicting Defects using Network Analysis on Dependency Graphs.* 2008 30th International Conference on Software Engineering: (ICSE), Vols 1 and 2, 2008: p. 530-539.

[24] Martin, R.C., *Granularity, C++ Report*, 1996. p. 57-62.

[25] Briand, L.C., Y. Labiche, and W. Yihong, *An investigation of graph-based class integration test order strategies.* Software Engineering, IEEE Transactions on, 2003. **29**(7): p. 594-607.

[26] Zhou, Y.M. and H.T. Leung, *Empirical analysis of object-oriented design metrics for predicting high and low severity faults.* IEEE Transactions on Software Engineering, 2006. **32**(10): p. 771-789.

[27] Shatnawi, R. and W. Li, The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process. Journal of Systems and Software, 2008. **81**(11): p. 1868-1882.

[28] Singh, Y., A. Kaur, and R. Malhotra, *Empirical validation of object-oriented metrics for predicting fault proneness models.* Software Quality Journal, 2010. **18**(1): p. 3-35.

[29] Bhattacharya, P., et al., *Graph-Based Analysis and Prediction for Software Evolution.* 2012 34th International Conference on Software Engineering (ICSE), 2012: p. 419-429.

[30] Menzies, T. and A. Marcus, *Automated Severity Assessment of Software Defect Reports.* 2008 IEEE International Conference on Software Maintenance, 2008: p. 346-355.

[31] Lamkanfi, A., et al., *Comparing Mining Algorithms for Predicting the Severity of a Reported Bug.* 2011 15th European Conference on Software Maintenance and Reengineering (CSMR), 2011: p. 249-258.

[32] Iliev, M., et al. Automated prediction of defect severity based on codifying design knowledge using ontologies. in First International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2012.

[33] Yang, C.-Z., et al. An Empirical Study on Improving Severity Prediction of Defect Reports Using Feature Selection. in Software Engineering Conference (APSEC), 19th Asia-Pacific. 2012.

[34] Rahimi, F. and A. Ipakchi, *Demand Response as a Market Resource Under the Smart Grid Paradigm.* Smart Grid, IEEE Transactions on, 2010. **1**(1): p. 82-88.

[35] Li, Z.D., et al., Characteristics of multiple-component defects and architectural hotspots: a large system case study. Empirical Software Engineering, 2011. **16**(5): p. 667-702.

[36] Gupta, A., et al., *An examination of change profiles in reusable and non-reusable software systems.* Journal of Software Maintenance and Evolution-Research and Practice, 2010. **22**(5): p. 359-380.

[37] S'liwerski, J., T. Zimmermann, and A. Zeller, When do changes induce fixes?, in Proceedings of the 2005 International Workshop on Mining Software Repositories 2005, ACM: St. Louis, Missouri. p.1-5.

[38] Schroeter, A., T. Zimmermann, and A. Zeller, Predicting component failures at design time, in Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering 2006, ACM: Rio de Janeiro, Brazil. p. 18-27.

[39] C'ubranic, D., *Project History as a Group Memory: Learning From the Past.* , in *PhD Thesis* 2004, University of British Columbia: Canada.

[40] Koru, A.G., et al., *Theory of relative defect proneness.* Empirical Software Engineering, 2008. **13**(5): p. 473-498.

[41] Arisholm, E., L.C. Briand, and A. Foyen, *Dynamic coupling measurement for object-oriented software.* IEEE Transactions on Software Engineering, 2004. **30**(8): p. 491-506.

[42] Basili, V.R., L.C. Briand, and W.L. Melo, *A validation of object-oriented design metrics as quality indicators.* IEEE Transactions on Software Engineering, 1996. **22**(10): p. 751-761.

[43] Zimmerman, T., et al. An Empirical Study on the Relation between Dependency Neighborhoods and Failures. in IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST). 2011.

[44] Xu, J., D. Ho, and L.F. Capretz, *An Empirical Validation of Object-Oriented Design Metrics for Fault Prediction.* Journal of Computer Science, 2008. **4**(7): p. 571- 577.

[45] Hall, T., et al., *A Systematic Review of Fault Prediction Performance in Software Engineering.* IEEE Transactions on Software Engineering, 2011. **99**(PrePrints).

P4: Can Refactoring Cyclic Dependent Components Reduce Defect-Proneness?

Published: In Proc. 29th IEEE International Conference on Software Maintenance (ICSM), 2013 pp. 420-423

Can Refactoring Cyclic Dependent Components Reduce Defect-Proneness?

Tosin Daniel Oyetoyan[1], Daniela Soares Cruzes[1,2] Reidar Conradi

[1]Department of Computer and Information Science
Norwegian University of Science and Technology Trondheim, Norway
{tosindo, dcruzes, conradi}@idi.ntnu.no

[2]SINTEF
Trondheim, Norway
danielac@sintef.no

Abstract—**Previous studies have shown that dependency cycles contain significant number of defects, defect-prone components and account for the most critical defects. Thereby, demonstrating the impacts of cycles on software reliability. This preliminary study investigates the variables in a cyclic dependency graph that relate most with the number of defect-prone components in such graphs so as to motivate and guide decisions for possible system refactoring. By using network analysis and statistical methods on cyclic graphs of Eclipse and Apache-ActiveMQ, we have examined the relationships between the size and distance measures of cyclic dependency graphs. The size of the cyclic graphs consistently correlates more with the defect-proneness of components in these systems than other measures. Showing that adding new components to and/or creating new dependencies within an existing cyclic dependency structures are stronger in increasing the likelihood of defect-proneness. Our next study will investigate whether there is a cause and effect between refactoring (breaking) cyclic dependencies and defect-proneness of affected components.**

Keywords— *cyclic dependency graphs; defect-proneness; graph complexities; refactoring*

1. Introduction

Maintaining software systems is a non-trivial task since these systems grow both in size and complexity over time [1]. Thus, it is not surprising that the cost of software maintenance is estimated to be the highest in the overall software budget [2]. Despite high maintenance costs and continuous research efforts to improve software quality, there are still evidence of system and business failures due to software defects [3, 4].

The structural complexity of software systems has been associated with defects [5]. The more complex a system is, the higher the risk of defects and failures. One area of such complexity is cyclic dependencies among software components, yet evidence confirms that they are wide spread in software systems [6]. Recently, we have demonstrated that components in dependency cycles account for most number and severity of defects [7, 8] both at the class file and package levels. Our findings show that, about 65% of defect-prone[34] class files are in cyclic graphs. Where cyclic class files are approximately 44%[35] of the total number of class files. Furthermore, the cyclic components account for an additional 11-17% of defect-prone components from the "depend-on-cycle [36] " group. In total, an approximate 80% of defect-prone class files are cyclic-related. Similarly, an approximate 90% of defect-prone packages are cyclic-related with cyclic packages accounting for about 58% of the total number of packages. We do not consider these figures to be trivial and based on the significance of the results; we are motivated to believe that further understanding of cyclic dependent components will be useful to guide decisions and provide reasoning for refactoring activities on cycles.

In relation to refactoring and software defects, Weissgerber and Diehl [9] found no correlation in particular between refactoring and defects opened in the subsequent days. Their results showed that there are periods where high refactoring was followed by increase in the number of defects as well as phases where refactoring led to no defects, although, the latter type were more prevalent. Ratzinger et al. [10] demonstrated that the number of software defects decreases when the number of refactoring increases in the preceding time period. Bavota et al. [11] showed that some kinds of refactoring are unlikely to be harmful but certain

[34] A defect-prone component as used in this study is defined as components with one or several defects
[35] This figure is an average of cyclic data from six different applications
[36] We define a "depend-on-cycle" component as component that is not in cycle but depends on component that is in cycle

kinds such as refactoring involving hierarchies (e.g. pull up method) are likely to induce defects. These studies have not considered refactoring cyclic dependent components in relation to defect-proneness. Thus they differ from our work.

There have been previous studies on network analysis of dependency graphs in relation to defect proneness of components [12-14]. Zimmermann and Nagappan [15] analyzed three different types of dependency sub-graphs (INTRA, OUT and DEP) at three separate levels of granularity. Using graph complexity measures, they obtained correlation between these measures and defects in each of the three sub-graphs and built regression models across the sub-graphs. One significant difference between our work and theirs is in the type of sub-graphs. We have focused on cyclic dependency graphs, which are missed in their study. Furthermore, our focus is to obtain variables to guide decision making during refactoring activities.

This paper extends our previous studies [7, 8] by investigating the correlation between the number of defect-prone components and cyclic graphs complexities. Consequently, we want to use these findings to motivate the refactoring (breaking) of defect-prone cyclic dependent components. This paper reports the preliminary results of this investigation and presents the direction for future work.

We are aware that correlation does not necessarily imply causality [16] because of the possibilities of hidden variables that may explain this higher number and severity of defects in the cyclic groups of these systems. Hence, our approach is to perform an experiment based on these preliminary findings in an industrial setup whereby we control for as many factors that could explain these effects and thus allow us to draw a reasonable conclusion on the effect of refactoring cyclic dependent components in relation to defect-proneness. We provide the details in Section IVa.

The rest of the work is structured as follows; in Section II, we lay the background to this study. Section III details our preliminary study. We report the results in Section IV and discuss the case study for our next study. Lastly, we conclude this study in Section V.

2. Background

In a software system, a component *X* is said to have dependency on another component *Y* if *X* requires *Y* to compile or function correctly [17]. Formally, a dependency graph of an object-oriented (OO) program, is defined as follows [18]:

Definition 1. An edge labeled digraph G = (V, L, E) is a directed graph, where V = $\{V_1, ..., V_n\}$ is a finite set of nodes, L = $\{L_1, ..., L_k\}$ is a finite set of labels, and $E \subseteq V \times V \times L$ is the set of labeled edges.

Definition 2. The object relation diagram (ORD) for an OO program P is an edge-labeled directed graph (digraph) ORD = (V, L, E), where V is the set of nodes representing the object classes in P, L = $\{I, Ag, As\}$ is the set of edge labels representing the relationships (Inheritance, Aggregation, Association) between the classes and $E = E_I \cup E_{Ag} \cup E_{AS}$ $E = E_I \cup E_{Ag} \cup E_{As}$is the set of edges.

Furthermore, we define the various measures of a graph that are important for our study [19].

i. ***Geodesic*** is the shortest path between two nodes in a graph
ii. The ***eccentricity*** of a node is its longest geodesic.
iii. The ***diameter*** of a graph is the maximum eccentricity of the nodes.
iv. The ***radius*** of a graph is the minimum eccentricity of the nodes.
v. The ***density*** of a graph is the ratio of the number of edges in the graph to the maximum possible edges in the graph.

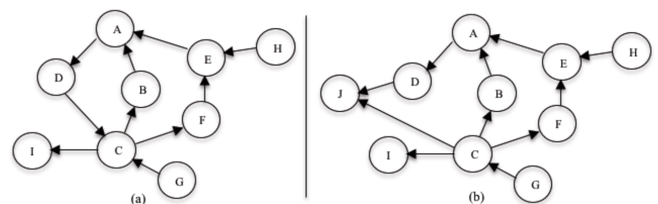


Fig. 9. Cyclic dependencies and defect propagation effect in a software network [8]

Cyclic dependencies and the hypothesis of defect propagation

In graph theory [20], a cyclic dependency graph also known as strongly connected components (SCC) in a directed graph G = (V, E) is a maximal set of vertices C ⊆ V such that for every pair of vertices u and v in C, both are reachable from each other. An example is depicted in Figure 1a of a hypothetical cyclic graph. In this graph, all the six components in two cycles (**A**, **D**, **C**, **B**, **F**, **E**) are mutually reachable from one another.

A related concept is the notion of dependency on components that are in cyclic relationships. We termed this as "depend-on-cycle" components [8]. Such "depend-on-cycle" components (e.g. **G** and **H** in Fig. 1a) obviously share the same complexity as the "in-Cycle" components that they depend on since they can reach all other components that are in these cyclic paths.

Cyclic dependency increases coupling complexities and thus has the potential to propagate defects in a software network [21]. Consider the hypothetical example in Fig. 1a, a defect in component **I** has the potential to propagate to components **C, B, A, D, G, F, E** and **H**. Let us say these cyclic components are refactored such that a new component **J** is introduced as depicted in Fig. 1b. The possible propagation of defects from **I** is significantly reduced to only **C** and **G**.

3. Preliminary Study

In this initial study, our goal is to find variables in cyclic dependency graphs that correlate with defect and thus motivate for refactoring possibilities. The research questions we want to investigate in this study and subsequent study are:

RQ1. What variables within a cyclic dependency graph correlate most with the number of defects and defect-prone components?

RQ2. Can the refactoring of factors in RQ1 correspond to a decrease in the number of defects and defect-prone components?

Data Computation

***Dependency and defect data*:** We have used the dependency data of two (Eclipse and ActiveMQ) of the systems collected in one of our previous papers [8] for this analysis. We subsequently compute the SCCs from this dependency data. Previous studies (e.g. [6]) already show that most systems have very long cycles such that hundreds of components are tangled in one large dependency cycle. In our analysis, the largest SCC in release 5.7.0 of ActiveMQ contains 414 class files while the largest SCC in Eclipse r3.0 has 690 classes. Our approach here is to break the long cycles into several smaller cyclic sub-graphs in such a way that all cyclic components are covered (Example of such sub-graphs is shown in Fig. 2). Using this approach allows us to verify whether an increase or decrease in size-based and distance-based cyclic graph measures correlate with defects. Similarly, we used the popular Eclipse defect data reported in [22] and the defect data collected for ActiveMQ in our previous study [8].

***Network Measures*:** As summarized in Table I, we have computed size and distance based measures. For size measures, we aggregate the number of nodes and number of edges in each graph. To compute the distance measures, we used the Floyd-Warshall algorithm [20] to calculate the geodesics i.e. "all-pairs shortest distance" between the nodes of each of the generated sub-graphs. We then compute the eccentricity (i.e. the maximum of the shortest distance) for each node. Subsequently, we calculate the diameter and the radius for each graph and sub-graphs from the values of the nodes' eccentricities.

TABLE XV. NETWORK METRICS USED IN THE STUDY

Metric	Formula
Size-based	
#Nodes	$\|V\|$
#Edges	$\|E\|$
Distance-based	
Diameter	$Max(Eccentricities)$
Radius	$Min(Eccentricities)$
Other complexity	
Density	$\frac{\|E\|}{\|V\|.\|V\|}$

4. Preliminary results and future study

Concerning RQ1, the results (Table II) show that the size-based measures (number of nodes and edges) correlate strongly and better than distance based measures with the number of defects and defect-prone components in these systems and at both levels (class and packages) of granularity. In other words, the higher the number of components and dependency relationships in cycle, the higher the probability of defects and defect-proneness of components. Furthermore, as displayed in Fig. 3, the R-squared value shows a good linear fit that suggests that we can predict the number of defect-prone components in a cyclic dependency graph using the graph size measures.

We can further interpret these results to mean that adding a new class/package (node) or dependency relationships (edge) to an existing cycle strongly increases the number of defect-prone components and defects in the cycle. In addition, size measures (i.e. node/edge) increase defect-proneness more than the strength of connection (diameter). These results agree with previous studies [12-14] that graph complexities correlate with defect-proneness of components and in comparison to [15] at the same granularity levels, the correlation is higher and stronger for cyclic graphs as against other types of graph.

Until now, we have not seen a systematic measurement of the impact of refactoring cyclic dependent components in relationship to defect-proneness. This has therefore motivated us to verify this in our next study. Our speculation is that if this hypothesis is not rejected such knowledge can strengthen the refactoring of highly defect-prone cyclic related components at both class and package hierarchies for existing systems and dissuade developers from writing cyclically connected programs.

Fig. 10. A cyclic dependency graph with many inner cycles from Eclipse 3.0 (Generated with Gephi: http://gephi.org/)

TABLE XVI. CORRELATION BETWEEN NETWORK METRICS AND NUMBER OF DEFECT-PRONE COMPONENTS IN CYCLIC GRAPHS

	Eclipse				**ActiveMQ**			
	Class		**Package**		**Class**		**Package**	
Metric	**S**	**P**	**S**	**P**	**S**	**P**	**S**	**P**
Nodes	0.79	0.89	0.89	0.98	0.74	0.71	0.81	0.91
Edges	0.80	0.85	0.89	0.97	0.75	0.73	0.80	0.92
Diameter	0.41	0.19*	0.72	0.64	0.50	0.52	0.13*	0.26*
Radius	0.38	0.13*	0.61	0.61	0.38	0.45	0.51	0.59
Density	-0.52	-0.52	-0.78	-0.78	-0.38	-0.63	-0.77	-0.77

S - Spearman; P – Pearson *(All non-asterisked results are significant at α = 0.01)*

Proposed Experiment

Regarding RQ2, we want to investigate whether breaking (refactoring) defect-prone cyclic dependent components would have effect on such components. There are several tools [37] and approaches that we can take advantage of to dissuade developers from writing cyclic codes or to refactor existing cyclic codes. Since a high correlation does not necessarily imply causality and there could be other hidden variables [16] (e.g. other design measures and complexities) that account for the defect-proneness of those components in the cyclic graphs, we have taken a practical approach to verify this conjecture.

[37] JDepend, NDepend, Dependometer, Dependency Structural Matrix

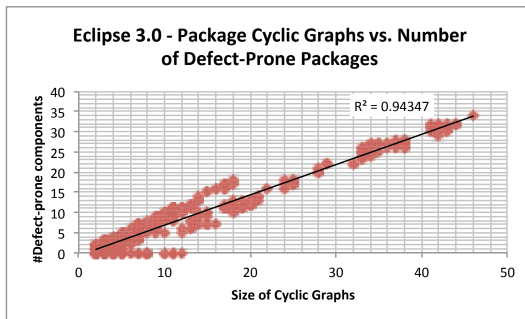


Fig. 11. Scatter plot between size of cyclic graph vs. #DPCs in Eclipse 3.0

The case study for this experiment is an industrial Smart Grid system. The application is a distribution management system that provides real-time operational support by continuously receiving status data from the power grid. The system has been in development for about six years and we have analyzed six post releases data of this application. It is mostly developed with C# programming language Furthermore, it has a size of about 380KLOC and contains 1459 class files and 2484 classes as of release 4.2.4 (Release date: Nov. 14, 2012). Our analysis of this application gave similar results with those presented in this paper.

Since the software undergo frequent releases by the company, we consider a six to nine months post refactoring data (after release) to be appropriate for our analysis. In order to eliminate unwanted factors that could possibly explain the difference between refactored cyclic groups and the rest of the groups, we decide to take measurements of well known object-oriented (OO) metrics and complexities in addition to the defect measures for three groups: (1) Refactored cyclic groups (2) Non-refactored cyclic groups and (3) Non-cyclic groups. We are particularly interested in post release defect data and therefore, our approach for the case study is as follows:

1. Select ***N*** sample of most defect-prone cyclic dependent graphs (class and package)
2. For each graph, record all measurement for the various metrics (number of defects, type of defects, severity of defects, correction efforts, lines of code, defect densities, OO metrics and complexities) for all components in these cyclic graphs (i.e. six/nine months pre-refactoring data)
3. Similarly, take measurements for the remaining cyclic groups and non-cyclic groups
4. Perform refactoring
5. From six/nine months after the next release, take new measurements detailed in steps 2 and 3 (post-refactoring)
6. Compare the new measures with the previous measures for each of the components in the groups

We admit that it is difficult to control and randomize all relevant factors in this experiment. For instance, the system may evolve and increase further in complexity as a result of dynamic changes in business requirements within the experiment phase. However, by going through these steps, whereby the measurements for the fixed sets of pre-refactoring data of components are compared against their post-refactoring data, we can, at least, move a step further to understanding the effect of refactoring defect-prone cyclic components and verify whether such activity can reduce the defect-proneness of affected components.

5. Conclusions

We have considered dependency cycles as an important area of dependency graph with very high complexities. We show that cyclic graphs complexities certainly have very strong correlation to defect-proneness of components. An increase in the size and strength of connections in a software cyclic dependency graphs correspond to an increase in the number of defect-prone components. Our next study is therefore focused on verifying the hypothesis that refactoring highly defect-prone cyclic dependency graphs will reduce the defect-proneness of the affected components in these graph structures.

References

[1] Lehman, M.M., *Programs, Life-Cycles, and Laws of Software Evolution*. Proceedings of the Special Issue Software Engineering, 1980. **68**(9): p. 1060-1076.

[2] Erlikh, L., *Leveraging Legacy System Dollars for E-Business*. IT Professional, 2000. **2**(3): p. 17-23.

[3] Leo, K. *Why banks are likely to face more software glitches in 2013*. [Web] 2013 April 24, 2013]; Available from: http://www.bbc.co.uk/news/technology-21280943.

[4] Lilley, S., *Critical Software: Good Design Built Right*. NASA System Failure Case Studies, 2012. **6**(2).

[5] Basili, V.R., L.C. Briand, and W.L. Melo, *A validation of object-oriented design metrics as quality indicators.*

IEEE Transactions on Software Engineering, 1996. **22**(10): p. 751-761.

[6] Melton, H. and E. Tempero, *An empirical study of cycles among classes in Java.* Empirical Software Engineering, 2007. **12**(4): p. 389-415.

[7] Oyetoyan, T.D., D.S. Cruzes and R. Conradi. *Criticality of Defects in Cyclic Dependent Components*. in *13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM) 22-23 September 2013 (Accepted)*.

[8] Oyetoyan, T.D., D.S. Cruzes, and R. Conradi, *A Study of Cyclic Dependencies on Defect Profile of Software Components.* Journal of Systems and Software, 2013. (*in press*).

[9] Weissgerber, P. and S. Diehl, *Are refactorings less error-prone than other changes?*, in *Proceedings of the 2006 International Workshop on Mining Software Repositories* 2006, ACM: Shanghai, China. p. 112-118.

[10] Ratzinger, J., T. Sigmund, and H.C. Gall, *On the relation of refactorings and software defect prediction*, in *Proceedings of the 2008 International Working Conference on Mining Software Repositories* 2008, ACM: Leipzig, Germany. p. 35-38.

[11] Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., & Strollo, O., *When Does a Refactoring Induce Bugs? An Empirical Study*. in *IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2012*.

[12] Bhattacharya, P., Iliofotou, M., Neamtiu, I., & Faloutsos, M., *Graph-Based Analysis and Prediction for Software Evolution.* 2012 34th International Conference on Software Engineering (ICSE), 2012: p. 419-429.

[13] Tosun, A., B. Turhan, and A. Bener, *Validation of network measures as indicators of defective modules in software systems*, in *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*2009, ACM: Vancouver, British Columbia, Canada. p. 1-9.

[14] Zimmermann, T. and N. Nagappan, *Predicting Defects using Network Analysis on Dependency Graphs.* 2008 30th International Conference on Software Engineering: (ICSE), Vols 1 and 2, 2008: p. 530-539.

[15] Zimmermann, T. and N. Nagappan, *Predicting subsystem failures using dependency graph complexities.* ISSRE 2007: 18th IEEE International Symposium on Software Reliability Engineering, Proceedings, 2007: p. 227-236.

[16] Pease, C.M. and J.J. Bull, *Scientific Desicion-Making*, 2000, Retrieved from http://www.utexas.edu/courses/bio301d/index.html.

[17] Jungmayr, S. *Identifying test-critical dependencies*. in *Software Maintenance*. 2002.

[18] Kung, D., Gao, J, Hsia, P, Toyoshima, Y, & Chen, C., *On Regression Testing of Object-Oriented Programs.* Journal of Systems Software, 1996. **32**(1): p. 21-40.

[19] Wasserman, S. and K. Faust, *Social network analysis : methods and applications*. Structural analysis in the social sciences. 1994, Cambridge ; New York: Cambridge University Press. xxxi, 825 p.

[20] Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C., *Introduction to algorithms*. 2nd ed. 2001, Cambridge, Mass.: MIT Press. xxi, 1180.

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

[22] Zimmermann, T., R. Premraj, and A. Zeller. *Predicting Defects for Eclipse*. in *International Workshop on Predictor Models in Software Engineering 2007*

P5: Transition and Defect Patterns of Components in Dependency Cycles during Software Evolution

Published: In Proc. IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week, Antwerp, Belgium, pp. 283-292

Transition and Defect Patterns of Components in Dependency Cycles during Software Evolution

Tosin Daniel Oyetoyan[1] Daniela Soares Cruzes[1,2], Reidar Conradi[1]
[1]Department of Computer and Information Science
Norwegian University of Science and Technology
Trondheim, Norway
[2]SINTEF, Trondheim, Norway
[1]{tosindo,conradi}@idi.ntnu.no [2]danielac@sintef.no

***Abstract*—The challenge to break existing cyclically connected components of running software is not trivial. Since it involves planning and human resources to ensure that the software behavior is preserved after refactoring activity. Therefore, to motivate refactoring it is essential to obtain evidence of the benefits to the product quality. This study investigates the defect-proneness patterns of cyclically connected components vs. non-cyclic ones when they transition across software releases. We have mined and classified software components into two groups and two transition states– the cyclic and the non-cyclic ones. Next, we have performed an empirical study of four software systems from evolutionary perspective. Using standard statistical tests on formulated hypotheses, we have determined the significance of the defect profiles and complexities of each group. The results show that during software evolution, components that transition between dependency cycles have higher probability to be defect-prone than those that transition outside of cycles. Furthermore, out of the three complexity variables investigated, we found that an increase in the class reachability set size tends to be more associated with components that turn defective when they transition between dependency cycles. Lastly, we found no evidence of any systematic "cycle-breaking" refactoring between releases of the software systems. Thus, these findings motivate for refactoring of components in dependency cycle taking into account the minimization of metrics such as the class reachability set size.**

***Index Terms*—dependency cycle, defect-proneness, refactoring**

1. Introduction

Today, virtually all aspects of systems (critical and non-critical) and businesses depend on software programs in order to perform their functions. This dependence implies that a failure within a software program is likely to result into a system or business failure[38]. Therefore, locating and improving potential locations of defects[39] will continue to be important for software systems. Despite continuous research efforts to improve software quality there are still evidence of system and business failures due to defects [1, 2].

Software engineers apply refactoring as a way to improve problematic locations in software systems. Refactoring is a process that improves the internal structure of a software system without changing its external behavior [3]. It is believed that refactoring improves software quality and increase productivity by making it easier to understand and maintain software codes [4].

However, in relation to refactoring and software defects there are conflicting evidence of the benefits of refactoring. Weissgerber and Diehl [5] found no correlation in particular between refactoring and defects opened in the subsequent days. Their results showed that there are periods where high refactoring was followed by increase in the number of defects as well as phases where refactoring led to no defects, although, the latter type were more prevalent. Ratzinger et al. [6] demonstrated that the number of software defects decreases when the number of refactoring increases in the preceding time period. Bavota et al. [7] showed that some kinds of refactoring are unlikely to be harmful but certain kinds such as refactoring involving hierarchies (e.g. pull up method) are likely to induce defects. Kim et al. [8]

[38] Failure: The inability of a system or system component to perform a required function within specified limits
[39] Defect/Fault: An anomaly in a software code unit or product that can be the cause of one or more failures

found that refactoring edits have a strong temporal and spatial correlation with bug fixes. In another study, Kim et al. [4] discovered that refactored binary modules of Windows 7 experienced significant reduction in the number of inter-module dependencies and post-release defects.

In recent studies [9, 10], we established that components in dependency cycles account for both the majority of defects and the most number of critical defects in the systems investigated. Although dependency cycle is known to be a sign of design decay [11, 12], evidence shows that it pervades software at different granularity levels. The results of these studies [9, 10] prompt further investigation of components in dependency cycles.

Specifically, we want to investigate if there is a pattern of increasing or decreasing defect-proneness for defective components that transition between dependency cycles across releases. Also, we want to know whether there is a systematic cycle-breaking refactoring between software releases. We distinguish between the word defective and the word defect-prone. A component is defective if it contains one or more defects in a release. While we define a component to be defect-prone if it persists as being defective in one or more future releases. Keeping existing programs acylic or breaking cyclic programs is not a trivial task since it involves behavior preservation of the original state of the software. Thus the fundamental question is whether companies would want to invest resources to refactor cyclically connected programs without empirical evidence of its benefits to the product quality.

Therefore, this study aims to investigate the evolution patterns of components in dependency cycle in order to understand:

i. Whether there is a pattern of increasing or decreasing defect-proneness of components that transition between dependency cycles. Is the probability of defect higher for components that move between dependency cycles than for those that move between out of cycle structure?
ii. Whether components in dependency cycle undergo any systematic “cycle-breaking” refactoring between releases.
iii. Whether factors such as coupling and size complexities provide further explanation to understanding the defect-proneness of components that transition between cycles.

The rest of this work is structured as follows; Section II provides the background to this study and reports on previous work. In Section III, we detail our empirical design for this study. Section IV presents the results and the discussion. In Section V, the threats to the validity of our results are discussed. Lastly, Section VI provides the conclusion of this study and the future work.

2. Background

In a software system, a component *X* is said to have dependency on another component *Y* if *X* requires *Y* to compile or function correctly [13]. Formally, a dependency graph of an object-oriented (OO) program, is defined as follows [14]:

Definition 1. An edge labeled digraph G = (V, L, E) is a directed graph, where V = $\{V_1, ..., V_n\}$ is a finite set of nodes, L = $\{L_1, ..., L_k\}$ is a finite set of labels, and E ⊆ VXVXL is the set of labeled edges.

Definition 2. The object relation diagram (ORD) for an OO program P is an edge-labeled directed graph (digraph) ORD = (V, L, E), where V is the set of nodes representing the object classes in P, L = $\{I, Ag, As\}$ is the set of edge labels representing the relationships (Inheritance, Aggregation, Association) between the classes and E = $E_I \cup E_{Ag} \cup E_{AS}$ $E = E_I \cup E_{Ag} \cup E_{As}$is the set of edges.

Cyclic Dependencies and the Hypothesis of Defect Propagation [15]

In graph theory [16], a cyclic dependency graph also known as strongly connected components (SCC) in a directed graph G = (V, E) is a maximal set of vertices C ⊆ V such that for every pair of vertices u and v in C, both are reachable from each other. An example is depicted in Figure 1a of a hypothetical cyclic graph. In this graph, all the six components in two cycles (**A**, **D**, **C**, **B**, **F**, **E**) are mutually reachable from one another. A related concept is the notion of dependency on

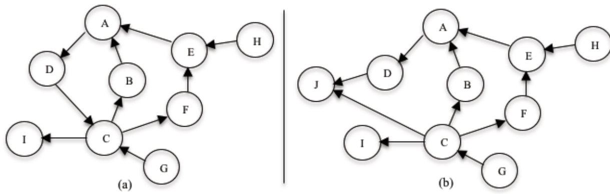


Fig. 12. Cyclic dependencies and defect propagation effect in a software network [10]

components that are in cyclic relationships. We termed this as "depend-on-cycle" components [10]. Such "depend-on-cycle" components (e.g. **G** and **H** in Fig. 1a) obviously share the same complexity as the "*in-cycle*" components that they depend on since they can reach all other components that are in these cyclic paths.

Cyclic dependency increases coupling complexities and thus has the potential to propagate defects in a software network [17]. Consider the hypothetical example in Fig. 1a, a defect in component **I** has the potential to propagate to components **C, B, A, D, G, F, E** and **H**. Let us say these cyclic components are refactored such that a new component **J** is introduced as depicted in Fig. 1b. The possible propagation of defects from **I** is significantly reduced to only **C** and **G**.

Cycles and software quality: Cycles among components have been claimed to be detrimental to understandability [18], production [11, 19], marketing [11], development [11, 19], usability [11, 19], testability [11], integration testing [13, 14, 20-22], reusability [19], extensibility [12] and reliability [11].

In relation to empirical findings of dependency cycle, it appears that only one study [23] has performed an elaborate empirical study of cycles on many software systems. The result shows that almost all the 78 Java applications they analyzed contain large and complex cyclic structures among their classes.

Dependence clusters: Existing studies [24, 25] have also shown that dependency cycles are not limited to classes or packages. They are also pervasive among program statements in software systems. This type is termed "dependence clusters" and they are formed when a set of program statements are mutually inter-dependent. Dependence clusters have been demonstrated to be detrimental to software maintenance activities [24, 25].

Cycles and defects**:** Zimmermann and Nagappan [26] performed a study on Windows Server 2003 to build a defect prediction model by using graph complexities. In this system, they found that binaries in dependency cycles have on average twice as many defects as those binaries not in cycles. In a recent study [10], we established that components in cyclic relationships, either directly or indirectly, have significantly more defect-prone components than those not in any cyclic relationships. The four hypotheses we tested on multiple systems confirm that:

1) Components in cycles have higher likelihood of defect-proneness than those not in cyclic relationships.
2) The higher number of defective components is concentrated in cyclic dependent components.
3) Defective components in cyclic relationships account for the clear majority of defects in the systems investigated.
4) The defect density of cyclic related components is sometimes higher than those in non-cyclic relationships.

Similarly, we established that components in dependency cycles account for the most number of highest severity defects in the software systems we investigated [9].

Refactoring, coupling and defects: In relation to refactoring, the studies in [4-8] have investigated the connections between refactoring and defect-proneness of components after refactoring. In general, refactoring focuses on rewriting code to make it easier to maintain and are mostly performed while adding features or fixing bugs [4]. In some cases, refactoring is aimed at reducing inter-component dependencies [4]. We distinguish between refactoring that reduce inter-component dependencies and a "cycle-breaking" refactoring. We show using Fig. 1b and Fig. 2 that breaking cycle does not necessarily reduce inter-component dependencies. Hence, the refactoring context we consider in this study relates only to breaking dependency cycles.

3. Empirical Setup

This study investigates the transition and evolution patterns of cyclic dependent class-files (we refer to these as components in the rest of the paper) over releases. We want to know if there is evidence of cycle-breaking refactoring and if there is evidence of increase or decrease in defect-proneness of components that transition in and out of dependency cycles. As shown in Fig. 1(a-e), we identify four transition states of a cyclic dependent component between releases. Thus, such component can:

1) Transition from *in-cycle* to *out-of-cycle* between releases
2) Transition from *in-cycle* to *depend-on-cycle* between releases
3) Remain *in-cycle* between releases
4) Be refactored (e.g. renamed) or deleted as it transition from one state to another between releases

We focus on states 1 and 3 and ignore state 4 for the reason that we could not accurately associate a renamed/deleted component between releases and we ignore state 2 because we assume similar complexity as state 3.

A. Research Questions

We formulate our research questions to address this goal as follows:

RQ1. Do components in dependency cycles persist as defective in the "*in-cycle*" state more than components that persist in the *out-of-cycle* state?

This research question allows us to compare the rate of defect-proneness of components that transition between *in-cycle* states across releases and components that transition between *out-of-cycle* states across releases. We formulate two null hypotheses to investigate this question as follows:

H_{01a}: *There is no significant difference between the proportions of defective components that remain defective as they transition between in-cycle states and the proportion of defective components that remain defective as they transition between out-of-cycle states.*

H_{01b}: *There is no significant difference between the proportions of non-defective components that become defective as they transition between in-cycle states and the proportion of non-defective components that become defective as they transition between out-of-cycle states.*

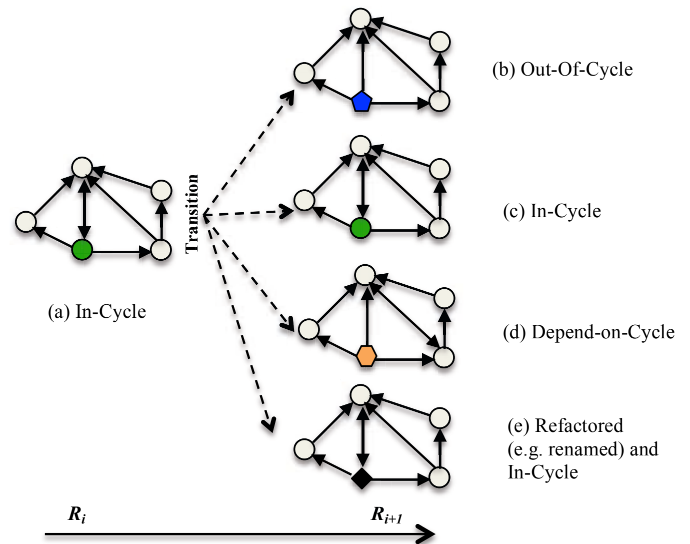


Fig. 13. (a-e). A simplistic example of transitions of in-cycle components between releases

RQ2. Is there evidence of cycle-breaking refactoring between releases?

Our assumption is that while other refactoring activities may be taking place between releases it does not typically include a systematic "cycle-breaking" refactoring.

RQ3. Does the transition of defective components from *in-cycle* to *out-of-cycle* reduce the defect-proneness of such components?

This research question corroborates **RQ1**. By investigating this question, we can verify whether there is a significant difference (increased/decreased) in defect-proneness of cyclic dependent components when they move out of dependency cycle in subsequent release(s). The null hypothesis for this question is:

H_{03}: *There is no significant difference between the proportions of defective components that transition as defective from in-cycle to out-of-cycle state and the proportion of defective "in-cycle" components in the previous release.*

RQ4. Does coupling or size complexity of components that transition as defective between *in-cycle* states increase at a significantly higher rate than those that transition between *out-of-cycle states*?

This question allows investigating whether the coupling/size variables are factors that may well explain the differences between the two groups as it relates to their defect-proneness. Our null hypothesis for this question is:

H_{04}: *There is no significant difference between the coupling/size density of components that become defective as they transition between in-cycle states and the coupling/size density of components that become defective as they transition between out-of-cycle states.*

B. Analysis Method

To answer the research questions, we categorize the transitions of components across releases as a Cartesian product between the current state (s_c) and the future state (s_f). That is $s_c \times s_f$ where s_c and s_f can take on values *in-cycle* or *out-of-cycle*. By using the resulting four transition categories it should then be possible to understand the movement patterns of components both *in-cycle* and *out-of-cycle* across releases.

Furthermore, we identify the transition of a component between the current and the future states regardless of its defect status in the future release by computing the set:

$$C^k_{s_c \to s_f} = C^k_{s_c} \cap C^r_{s_f}$$

$where:$

$C = Components$

$s_c = \ current\ state,\ s_f = future\ state, n = number\ of\ release$

$k = current\ release, \quad k \in [1{:}\,n-1]$

$r = future\ release, \quad r \in [k+1{:}\,n]$

To identify components that turn defective in the next release, we compute for each release the forward intersection of components in the current release at the current state with defective components in the future release at the future state. In set form, we compute:

$$C^k_{s_c \to s_{f(defect)}} = C^k_{s_c} \cap DC^r_{s_f}\ , where{:}\ DC = Defective\ Components$$

The number of components that persist as defective or become defective in the next release and between the two states is calculated as the cardinality of the set:

$$number\left(C^k_{s_c \to s_{f(defect)}}\right) = \left|C^k_{s_c \to s_{f(defect)}}\right|$$

In summary;

1) $C^k_{s_c \to s_f}$: Is a set of components in release k at current state (s_c) that transition to future state (s_f) as defective or not defective in the future release.
2) $C^k_{s_c \to s_{f(defect)}}$: Is a set of components in release k at current state (s_c) that persist to be defective or become defective in future state (s_f) in the future release.
3) $C^k_{s_c \to s_f} - C^k_{s_c \to s_{f(defect)}}$: Is a set of components in release k at current state (s_c) that transition to future state (s_f) in the future release as not defective.

Finally, we compute the percentage of defect-prone and non defect-prone components between two transition states and across releases as:

1) $$\%\ Defect-prone_{s_c \to s_f} = \frac{\left|C^k_{s_c \to s_{f(defect)}}\right|}{\left|C^k_{s_c \to s_f}\right|} \times 100$$
2) $$\%\ Non-Defect-prone_{s_c \to s_f} = \frac{C^k_{s_c \to s_f} - C^k_{s_c \to s_{f(defect)}}}{\left|C^k_{s_c \to s_f}\right|} \times 100$$

1) Computing Coupling and Size Complexity

To answer **RQ4**, we first calculate our coupling complexity in the form of the degree of coupling for each node (component) in the graph [27]. Such that:

$$Degree\ (C) = Fan-in\ (C) + Fan-out\ (C)$$

Where Fan-in represents incoming connections to C and Fan-out represents outgoing connections from C. Then, we compute the coupling density (i.e. the difference in the degree between the two

releases divided by the number of components involved in the transition) as:

$$Density(Degree) = \frac{\sum_r Degree\left(C^k_{s_c \to s_{f(defect)}}\right) - \sum_k Degree\left(C^k_{s_c \to s_{f(defect)}}\right)}{\left|C^k_{s_c \to s_{f(defect)}}\right|}$$

Furthermore, we obtain the class reachability set size[40] (crss) metric [28] and compute crss density for the whole set as:

$$Density(CRSS) = \frac{\sum_r CRSS\left(C^k_{s_c \to s_{f(defect)}}\right) - \sum_k CRSS\left(C^k_{s_c \to s_{f(defect)}}\right)}{\left|C^k_{s_c \to s_{f(defect)}}\right|}$$

Similarly, we compute the size density using the lines of code (LOC) metric (i.e. the difference in LOC between the two releases divided by the number of components involved in the transition)

$$Density(LOC) = \frac{\sum_r LOC\left(C^k_{s_c \to s_{f(defect)}}\right) - \sum_k LOC\left(C^k_{s_c \to s_{f(defect)}}\right)}{\left|C^k_{s_c \to s_{f(defect)}}\right|}$$

By comparing the coupling difference per component and the size difference per component, we can understand whether the complexities increase significantly more in one group than the other.

2) Testing the Hypotheses

For testing the hypotheses, we use the t-test when our data is normally distributed and a non-parametric test (e.g. Wilcoxon signed rank) when it is not [29]. In addition, we employ a proportion test for system with fewer numbers of releases. In all cases the data is unpaired and so the "paired" variable is set to FALSE in the r-statistical package[41] that is used. Table I summarizes the design and the hypotheses we test for each of the research questions.

For **RQ3**:

$$prop1(Ri) = \frac{\#\text{inc (defect)}}{\#inc}$$

$$prop2(Ri+1) = \frac{\#\text{inc} \to \text{oinc (defect)}}{\#inc}$$

C. Method of Data Collection

We have performed a study on a commercial Smart Grid application (commApp) developed with C#. In addition, we choose an integrated development environment (Eclipse)[42], a messaging and integration pattern server (Apache-ActiveMQ)[43],

TABLE XVII. SUMMARY OF RESEARCH DESIGN AND HYPOTHESES

Research Question	Data		H_0	H_1
	Prop1	*Prop2*		
RQ1, RQ4	inc→inc	oinc→oinc	Prop1≤Prop2	Prop1>Prop2
RQ3	inc	inc→oinc	Prop1≤Prop2	Prop1>Prop2

a. inc: in-cycle, oinc: out-of-cycle

and a service framework (Apache-CXF)[44] all developed with Java. We have purposefully selected very active projects from the open source community and we also considered projects that have different functionalities with different development languages and variations in release dates (see Table II and Table III). The variation in the release dates especially allows us to understand whether observed patterns in the data are similar irrespective of the time-span between the releases. The commercial application commApp, is a distribution management system designed to allow for monitoring and planning of Grid operations. It provides real-time operational support by continuously receiving status data from the power grid. Eclipse is a popular open source integrated development environment (IDE) while ActiveMQ is a messaging server with the capability to handle various integration patterns. Lastly, Apache-CXF is a service framework that helps to build and develop services using frontend-programming APIs, like JAX-WS and JAX-RS.

[40] This metric counts, for a given class, all the other classes in the system's source code that it transitively depends-on for its compilation (Melton and Tempero 2006)

[41] http://www.R-project.org

[42] http://archive.eclipse.org/eclipse/downloads/index.php

[43] http://activemq.apache.org/index.html

[44] http://cxf.apache.org/

TABLE XVIII. PROPERTIES OF SELECTED APPLICATIONS

System	Language	Domain	License	Bug Tracker	Age	Versions Analyzed
Apache-ActiveMQ	Java	Messaging and Enterprise Integration Pattern Server	Open	JIRA	6	5.7.0, 5.6.0, 5.5.1, 5.5.0, 5.4.2, 5.4.1
Eclipse	Java	IDE	Open	Bugzilla		3.0, 2.1, 2.0
commApp	C#	Smart Grid	Commercial	HP-Quality Center	6	4.2.4, 4.2.2, 4.1, 4.0.1SP4, 4.0.1SP2, 4.0
Apache-CXF	Java	Service framework	Open	JIRA	6	2.6.0, 2.5.0, 2.4.0, 2.3.0, 2.2.0, 2.1.0

TABLE XIX. SUMMARY OF SOFTWARE SOURCE CODE AND DEFECT DATA

Release/Version	Date	#Class-Files	KLOC	#Defective Class-Files	#Defects
Apache-ActiveMQ					
5.7.0	Nov 22 2012	1517	136.22	35	68
5.6.0	Jun 15 2012	1505	133.25	88	102
5.5.1	Oct 16 2011	1331	118.27	54	76
5.5.0	Apr 01 2011	1331	118.27	115	105
5.4.2	Dec 02 2010	1258	113.01	80	66
5.4.1	Sept 21 2010	1256	112.20	79	63
Eclipse					
3.0	Jun 25 2004	10635	1308.66	1566	-
2.1	Mar 27 2003	7909	988.45	845	-
2.0	Jun 27 2002	6751	797.93	968	-
commApp					
4.2.4	Nov 14 2012	1203	341.83	29	14
4.2.2	Oct 12 2012	1199	339.78	49	18
4.1	Aug 17 2012	1002	316.22	60	42
4.0.1SP4	Apr 11 2012	904	286.99	69	29
4.0.1SP2	Mar 26 2012	903	285.89	46	28
4.0	Oct 14 2011	849	266.11	137	143
Apache-CXF					
2.6.0	Apr 17 2012	2874	268.1	60	45
2.5.0	Nov 11 2011	2726	252.8	50	41
2.4.0	Apr 11 2011	2542	233.1	84	74
2.3.0	Oct 11 2010	2335	219	86	91
2.2.0	Mar 18 2009	2096	185.3	96	74
2.1.0	Jul 03 2007	1797	153.3	96	88

TABLE XX. AVERAGE OF MIN AND MAX VALUES PER CLASS-FILE OF COLLECTED METRICS

System	CRSS		LOC		Fan-out		Fan-in		SCC	
	Min	*Max*	*Min*	*Max*	*Min*	*Max*	*Min*	*Max*	*Min*	*Max*
Eclipse	1	5914	3	5102	0	220	0	1497	0	687
CommApp	1	618	7	6873	0	157	0	182	0	130
Active-MQ	1	632	3	1995	0	76	0	482	0	358
Apache-CXF	1	601	3	6663	0	67	0	378	0	58

1) Defect Data Collection

We have collected defect data from two different defect-tracking systems (DTSs). Defect repository gives typically a high level overview of a problem report. For example, typical attributes of the HP-QC defect tracking system (QC-DTS) are the Defect ID, severity of the defect, the type of defect, date defect is detected, the modules containing the defect, the version where defect is detected, and the date the defect is fixed. These fields are similar to the Apache JIRA DTS. Our first step is to determine the bugs that affect each version of the system. In Apache JIRA DTS, we use the “Affects Version” field to filter all bugs that affect a particular version of the system. For HP-QC, we use “Detected in Version(s)”. A certain defect may affect multiple versions of a system. By this we mean “hotspot” defects [30] that keep re-occurring and span several versions of a system. We keep only to defects that are marked “fixed” in the “resolution” field for JIRA and those that are marked “closed”, “fixed” or “tested-ok” in the “status” field for HP-QC. In HP-QC, the status field is used as the resolution field. Thus the status of a resolved defect can change from tested-ok to fixed and finally to

closed. The Eclipse dataset that we use in this paper has been mapped in previous study [31].

Mapping defects to class-files: A common way to figure out what operation is performed on the source file is to look at the message field of the SVN commit. When developers provide this information with the bug number and/or useful keywords (e.g. bug or fix), it is possible to map the reported defect with the actual source file that is modified to fix it. In our case, we have used the bug number in the commit message to map the defects from the DTS to the actual class-files that are changed. It is important to state that the defects that affect each version as previously collected from the DTS provide the boundaries for the class-files that are mapped for each version of the systems. Table III reports the defect data for each of the systems.

1) Dependency Data Collection

We have used the same tool as in our previous work [10] and reused some algorithms implemented in the tool by Melton and Tempero [23] to collect dependency data and measurement values for the number of strongly connected components (SCC), CRSS, LOC, Fan-in and Fan-out metrics for each class-file. Table IV lists the average values over the releases for each of the software systems analyzed.

4. Results and Discussion

RQ1: *Do components in dependency cycles persist as defective in cyclic state in the future release more than non-cyclic components that persist in non-cyclic state?*

TABLE XXI. TRANSITION OF DEFECTIVE COMPONENTS FROM R_I TO R_{I+1}

Systems ($R_i \rightarrow R_{i+1}$)	# of Defective components involved in transition from $R_i \rightarrow R_{i+1}$				% (Defective→Defective) components from $R_i \rightarrow R_{i+1}$			
	inc→inc	*oinc→oinc*	*inc→oinc*	*oinc→inc*	*inc→inc*	*oinc→oinc*	*inc→oinc*	*oinc→inc*
Eclipse								
2.0 → 2.1	619	25	7	5	27.6	16	14.3	0
2.1 → 3.0	387	24	0	8	41.6	33.3	0	50
CommApp								
4.0 → 4.0.1SP2	78	20	0	1	27	0	0	0
4.0.1SP2 → 4.0.1.SP4	29	2	0	0	55.2	0	0	0
4.0.1SP4 → 4.1	39	12	0	0	66.7	8.3	0	0
4.1 → 4.2.2	40	4	0	2	35	50	0	100
4.2.2 → 4.2.4	21	11	0	0	52.4	9.1	0	0
Active-MQ								
5.4.1 → 5.4.2	62	10	0	0	42	50	0	0
5.4.2 → 5.5.0	51	14	0	0	45.1	28.6	0	0
5.5.0 → 5.5.1	67	26	0	0	47.8	15.4	0	0
5.5.1 → 5.6.0	42	6	0	0	59.3	50	0	0
5.6.0 → 5.7.0	61	7	0	1	19.7	0	0	0
Apache-CXF								
2.1.0 → 2.2.0	27	30	0	0	26	20	0	0
2.2.0 → 2.3.0	43	18	0	2	28	11	0	0
2.3.0 → 2.4.0	30	17	0	0	26.7	23.5	0	0
2.4.0 → 2.5.0	37	12	0	3	16	0	0	0
2.5.0 → 2.6.0	19	3	0	0	15.8	0	0	0

TABLE XXII. TRANSITION OF NON-DEFECTIVE COMPONENTS FROM R_i TO R_{i+1}

Systems (R_i→R_{i+1})	# of Non-Defective components involved in transition from R_i→R_{i+1}				% (Non-Defective→Defective) components from R_i→R_{i+1}			
	inc→inc	oinc→oinc	inc→oinc	oinc→inc	inc→inc	oinc→oinc	inc→oinc	oinc→inc
Eclipse								
2.0 → 2.1	1934	1115	4	72	8.1	1.9	0	0
2.1 → 3.0	2508	1128	8	38	16.9	4.5	12.5	13.2
CommApp								
4.0 → 4.0.1SP2	174	604	0	0	2.9	0.3	0	0
4.0.1SP2 → 4.0.1.SP4	234	688	0	0	9.8	1.7	0	0
4.0.1SP4 → 4.1	225	672	0	1	5.8	0.9	0	0
4.1 → 4.2.2	236	737	8	1	2.1	1.2	0	0
4.2.2 → 4.2.4	277	912	0	0	4.3	0	0	0
Active-MQ								
5.4.1 → 5.4.2	365	263	0	0	6.8	3.8	0	0
5.4.2 → 5.5.0	377	259	0	0	11.4	8.5	0	0
5.5.0 → 5.5.1	362	255	0	0	2.8	0.8	0	0
5.5.1 → 5.6.0	387	265	0	4	8	1.9	0	25
5.6.0 → 5.7.0	410	271	0	2	0.7	0.7	0	0
Apache-CXF								
2.1.0 → 2.2.0	271	869	2	8	6.6	1.7	0	0
2.2.0 → 2.3.0	321	980	1	8	4.4	1.0	0	12.5
2.3.0 → 2.4.0	377	1112	3	2	6.1	0.9	0	0
2.4.0 → 2.5.0	419	942	0	5	3.1	0.4	0	0
2.5.0 → 2.6.0	428	987	0	4	1.9	0.2	0	0

TABLE XXIII. TEST OF DIFFERENCE IN MEAN VALUES BETWEEN THE GROUP PROPORTIONS FOR **RQ1**

System	Defective → Defective (H1a)			Non-Defective → Defective (H1b)		
	inc→inc	*oinc→oinc*	*p-value*	*inc→inc*	*oinc→oinc*	*p-value*
Eclipse †	34.6	24.6	0.26	12.5	3.2	0.001*
CommApp◊	47.3	13.5	0.01*	4.98	0.82	0.018*
Active-MQ◊	42.8	29	0.14	5.94	3.14	0.138
Apache-CXF◊	22.5	10.9	0.04*	4.42	0.84	0.006*

b. †:proportion test ◊:t-test *:significant at $\alpha = 0.05$

On the average, we found that components that transition in the *in-cycle* state are more defect-prone at a higher rate than components that transition in *out-of-cycle* state (see Table VII). For the first hypothesis that focuses on defective → defective transitions between the two groups; we found that two (commApp and Apache-CXF) out of the four systems we analyzed have significantly higher rate of defect-proneness for components that transition between *in-cycle* state than those that transition between *out-of-cycle* state (Table VII). While for the hypothesis that investigates non-defective → defective transitions between the groups, we found that all the systems except ActiveMQ have significantly higher rate of defect-proneness for components that transition between *in-cycle* states than between *out-of-cycle* states. We can therefore infer that a non-defective/defective component that is *in-cycle* and remains *in-cycle* in the next release has a higher probability to become defective than a non-defective/defective component that is *out-of-cycle* and remains in *out-of-cycle* state in the next release.

RQ2: *Is there evidence of systematic breaking of dependency cycles between releases?*

We observe for all the systems that *in-cycle* components transition mostly in the *in-cycle* state (see Table VIII inc→inc vs. inc→oinc). Results show that for all the systems we analyzed over 99% of both defective and not-defective components that are *in-cycle* persist in the *in-cycle* state in the next release. Therefore suggesting that there is no intentional cycle-breaking refactoring. By using Ref-Finder[45], we further investigated the refactoring between the Eclipse versions. Our findings show that many micro refactoring were carried out such as "Move method", "Remove control flag", "Inline method", "Extract method" and so on between those versions. In addition, we investigated one out of the few class-files (*org.eclipse.debug.internal.ui.views.console. ConsoleDocument.java*), that transitioned between in-cycle (in version 2.0) to out-of-cycle (in version 2.1). The class-file, *ConsoleDocument.java* was originally contained in a strongly connected components (SCC) with thirty-six other class-files in version 2.0. We found that, refactoring such as move field and move method were performed on one of the neighbors that involved moving out a method with a reference to the class "*org.eclipse.debug.internal.ui.launchConfigurations. LaunchConfigurationHistoryElement.java*" that was in the same cycle. Resulting into the *in-cycle* to *out-of-cycle* transition of *ConsoleDocument.java*. However, the number of SCC for other components in the same cycle increased from thirty-seven in version 2.0 to fifty-eight in version 2.1 and to fifty-nine in version 3.0.

It therefore indicates that while there are possible many other refactoring activities prior to release, those refactoring do not automatically translate to cycle-breaking refactoring. The few components that transition from *in-cycle* to *out-of-cycle* appear to be accidental movements. Hence corroborating previous results on the pervasiveness of dependency cycles across releases of software components [23]. This seems so because cycle breaking is more of an architectural refactoring [32] which is not trivial because such activity needs planning and would need to take advantage of tools and methods [32-35] for detecting and breaking those dependency cycles.

Empirical evidence of cycles [9, 10, 23] shows that object-oriented concepts such as abstraction and design guidelines are violated due to unguided design decisions as the system evolves. We submit therefore that software engineers need to purposely take advantage of existing cycle detecting tools (e.g. [33, 35]) and approaches to prevent dependency cycles in their software systems.

RQ3: *Does the transition of defective components from in-cycle to out-of-cycle reduce the defect-proneness of such components?*

As listed in Tables V and VI, there is not sufficient data to answer this research question. We show previously (in **RQ2**), that the most components in cycle transition to the same state they were in the previous release.

TABLE XXIV. % OF COMPONENTS THAT MOVE FROM IN-CYCLE TO OUT-OF-CYCLE

System	# inc→inc	% inc→oinc
Eclipse	2733.5	0.35
CommApp	272.2	0.59
Active-MQ	436.8	0
Apache-CXF	395.6	0

Showing that **RQ1** provides so far the only evidence of possible benefits that breaking cycles and moving the affected components to *out-of-cycle* state may result into lower defect-proneness of these components. We would focus on **RQ3** in our future work.

RQ4: *Does coupling or size complexity of components that transition between in-cycle state increase at a significantly higher rate than those that transition between out-of-cycle state?*

Table IX shows the mean data and the p-values of the statistical test for both coupling density and size density. On the average, the size density of components that transition between *in-cycle* state increased at a higher rate than those components that transition between *out-of-cycle* state. In all the systems, the code size measured by lines of code (LOC) increased more in the *in-cycle* group but not significant in about half of the cases. The coupling density of *in-cycle* components inreased

[45] https://webspace.utexas.edu/kp9746/www/reffinder/

at a different rate as those in the *out-of-cycle* group. For instance, consider the "defective → defective" transition in Eclipse; result shows that the average coupling density for *in-cycle* group (i.e. 17.8 dependencies per class-file) increased significantly more than that of the *out-of-cycle* group (9.25 dependencies per class-file). While for "non-defective → defective" transition in Apache-CXF and CommApp; results show that the average coupling density increased more in the *out-of-cycle* group than the *in-cycle* group.

While we observe a pattern of higher code increase (LOC) for *in-cycle* group than the *out-of-cycle* group. Which may be a pointer to the complexity of the components in cycle. There is no general pattern of increase/decrease in the degree of coupling (Fan-out + Fan-in) between *in-cycle* and *out-of-cycle* groups. We can thus infer that the degree of coupling may or may not increase as a result of transition between *in-cycle* state. In addition, our observations regarding the complexity of *in-cycle* components that turn defective is the increase in the reachability set density of these components. As listed in

Table X, the reachability density of the *in-cycle* group for all the systems increase significantly at a higher rate than the *out-of-cycle* group. To substantiate this finding, we inspect the reachability set of *in-cycle* → *out-of-cycle* and *out-of-cycle* → *in-cycle* groups of Eclipse (v2.1 → v3.0). We found that the reachability set of class-files in the "*in-cycle* → *out-of-cycle*" transition reduced by 1137 class-files while the *out-of-cycle* → *in-cycle* increased by 1351 class-files.

In conclusion, reachability set and code size appear to be two variables that associate more with the complexities of *in-cycle* components and their defect-proneness. Melton and Tempero [28], presented this metric named *class reachability set size* (CRSS) and demonstrated that a refactoring that reduces the crss of software components can potentially improve the quality of the software. Our results extends their findings to show the association between the defect-proneness of *in-cycle* components during evolution and their complexities as demonstrated by their crss values.

The results of this study support pivotal metrics such as CRSS as a metric that can be focused for optimization during cycle-breaking refactoring of defect-prone components. By minimizing the CRSS values of problematic (defect-prone) components that are in cycles, it might be possible to effectively reduce the probability of defect propagation to other components.

Implications: Does breaking dependency cycles imply reduction in defect-proneness of components? The results of our study do not have a direct connection to this question but do have an indirect one. The observed pattern in the data we analyzed show that the rate of defect-proneness is higher for components that move between dependency cycles. Moreso, there is a pattern of increased class reachability set size and code size for those defect-prone components involved in transition between dependency cycles.

TABLE XXV. COUPLING AND SIZE DENSITIES FOR **RQ4**

Systems	**Coupling density (Mean)**						**Size density (Mean)**					
	Defective → Defective			***Non-Defective → Defective***			***Defective → Defective***			***Non-Defective → Defective***		
	inc→inc	*oinc→oinc*	*p*	*inc→inc*	*oinc→oinc*	*p*	*inc→inc*	*oinc→oinc*	*p*	*inc→inc*	*oinc→oinc*	*p*
Eclipse	17.8	9.25	**0.009***	9.18	5.91	**0.055**	86.92	80.83	**0.418**	30.44	7.2	**0.004***
CommApp	1.47	0.67	**0.062**	-0.03	0.35	**0.287**	97.8	13.2	**0.009***	39.13	1.61	**0.028***
Active-MQ	0.58	0.42	**0.500**	1.51	0.32	**0.166**	18.75	2.53	**0.086**	7.6	5.75	**0.393**
Apache-CXF	2.49	6.52	**0.164**	2.06	2.81	**0.576**	45.37	13.5	**0.121**	38.26	15.2	**0.123**

c. *: Significant at $\alpha = 0.05$

TABLE XXVI. REACHABILITY DENSITY FOR **RQ4**

Systems	Reachability density (Mean)					
	Defective → Defective			*Non-Defective → Defective*		
	inc→inc	*oinc→oinc*	*p-value*	*inc→inc*	*oinc→oinc*	*p-value*
Eclipse	349	77	**0.043***	348	61.3	**0.018***
CommApp	8.5	0.7	**0.039***	11.2	0.42	**0.016***
Active-MQ	30	0.4	**0.037***	33.4	12.7	**0.377**
Apache-CXF	40.3	0.54	**0.015***	31.5	1.53	**0.036***

d. *: Significant at $\alpha = 0.05$

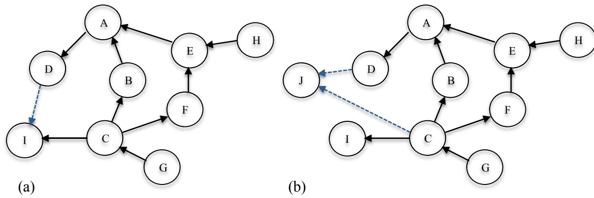


Fig. 14. Optimal vs. non-optimal cycle-breaking refactoring

We are quick to state that other factors may indeed be explanatory factors in the defect-proneness of *in-cycle* components. For instance, requirement changes, developers experience, change proneness of the components and so on. However, the results in this study uncovers a pattern that makes us better understand the association between defect-proneness of *in-cycle* components and their complexities during system evolution. Thus, software engineers can employ these results as a motivation and use the investigated variables as decision variables when performing a cycle-breaking refactoring. Such decision should therefore consider an optimal solution from the point of view of problematic components. Take for instance the cycles in Figure 1a., we present two solutions to create an acyclic graph in Figure 3. If we assume that component **I** is highly defect-prone (or reduces other quality factors), then the solution in Figure 3a is not optimal since all the other components can still reach **I** transitively. However, the solution in Figure 3b is optimal because only two components (C and G) can reach **I** transitively.

5. Threats to Validity

We have analyzed and evaluated a Smart Grid system, an integrated development environment, a service framework application and a messaging and integration pattern server. Although, these four systems vary in terms of properties such as domain, functionality, programming language, size, usage, context and study period, we cannot claim that the observed defect patterns or related will hold for other systems. As it is with most case studies, we cannot generalize these results across all systems. Replicated and/or further studies will be necessary to compare results across several systems and domains.

We have used density measure in our analysis that penalizes all components equally. In some cases, size, coupling and reachability measures may be skewed in the dataset resulting in few components having high number or low number of these measures. However, we do not think this approach can affect our result in a significant way or its interpretation since we have limited the analysis to the defective subset of the whole dataset. Furthermore, we have ignored the transitions between the "*depend-on-cycle*" states. This group may also contain some measurement data in some cases. However, we consider the results to be valid since our focus is the transition that concerns *in-cycle* structure vs. *out-of-cycle* structure.

For this study, we have relied on the defects logged in the defect tracking systems of each application. Our approach of extracting defect data is similar to what other researchers have used in the past [36-38]. Nevertheless, common threats are whether defects logged in the DTS are accurately tagged in the respective code changes in the version systems. In addition, we cannot be sure if all defects are logged in the DTS. Also, there could be cases that the message log of the file that consists a change is not tagged with the bug numbers of the resolved defect. Furthermore, there could be cases of typographical error in the recording of the bug number in the version systems [36] and it is still possible that duplication will occur. Lastly, since we have not eliminated the effect of tangled class files as discussed in [39] when analyzing the repository, it is thus possible that some class-files are incorrectly associated with bug reports.

6. Conclusions and Future Work

In this study, we have examined whether components (defective/non-defective) undergo any systematic cycle-breaking refactoring before the next release. Our findings show that even though other types of refactoring might be taking place before the next releases of these software systems. They do not automatically translate to cycle-breaking refactoring. Furthermore, we investigated whether components that move between dependency cycles have higher tendencies of being defective in the future releases of these software systems than components that move outside of dependency cycles. The results of our analysis demonstrate that averagely, movement of components between dependency cycles across releases increase their defect-proneness more than movement of components outside of dependency cycles.

Lastly, we found that between releases, inter-component dependencies do not increase differently in components that transition as defective between dependency cycles to those that move outside of dependency cycles. However, we found a consistent pattern of increased class reachability set size (CRSS) and increased lines of code for the *in-cycle* group. Thus suggesting that a "cycle-breaking" refactoring that minimizes the CRSS value of defect-prone components in dependency cycles has tendency to reduce the defect-proneness of these components in the next release.

In conclusion, this study shows a pattern that suggests a possibility that we can gain the benefit of reduced defect-proneness by performing a cycle-breaking refactoring. Most especially when such cycle-breaking activity considers the minimization of important metrics such as the CRSS value.

As future work, we aim to investigate whether the movements of components from dependency cycle to outside of cycle could reduce their defect-proneness. We found that there is currently not sufficient empirical data to answer this question. Thus, to answer **RQ3**, we have proposed an experiment in [15] whereby a set of defective cyclically connected components are purposely moved to *out-of-cycle* state. This experiment takes a number of factors into consideration in order to understand their effects on the refactoring activity of the components. Details of the experiment are contained in the paper [15]. Currently, we are at the execution stage of this experiment in an industrial setup.

In addition, our work would focus on developing optimization methods to effectively reduce transitive dependencies (CRSS values) of identified (user/automated) problematic components during cycle-breaking refactoring.

References

[1] Leo, K. *Why banks are likely to face more software glitches in 2013.* [Web] 2013 April 24, 2013]; Available from: http://www.bbc.co.uk/news/technology-21280943.

[2] Lilley, S., *Critical Software: Good Design Built Right.* NASA System Failure Case Studies, 2012. **6**(2).

[3] Mens, T. and T. Tourwe, *A survey of software refactoring*. Software Engineering, IEEE Transactions on, 2004. **30**(2): p. 126-139.

[4] Kim, M., T. Zimmermann, and N. Nagappan, *A field study of refactoring challenges and benefits*, in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*2012, ACM: Cary, North Carolina. p. 1-11.

[5] Weissgerber, P. and S. Diehl, *Are refactorings less error-prone than other changes?*, in *Proceedings of the 2006 International workshop on Mining software repositories* 2006, ACM: Shanghai, China. p. 112-118.

[6] Ratzinger, J., T. Sigmund, and H.C. Gall, *On the relation of refactorings and software defect prediction*, in *Proceedings of the 2008 International working conference on Mining software repositories* 2008, ACM: Leipzig, Germany. p. 35-38.

[7] Bavota, G., et al. *When Does a Refactoring Induce Bugs? An Empirical Study*. in *IEEE 12th International Working Conference onSource Code Analysis and Manipulation (SCAM),* 2012.

[8] Kim, M., D. Cai, and S. Kim, *An empirical investigation into the role of API-level refactorings during software evolution*, in *Proceedings of the 33rd International Conference on Software Engineering* 2011, ACM: Waikiki, Honolulu, HI, USA. p. 151-160.

[9] Oyetoyan, T.D., D.S. Cruzes, and R. Conradi. *Criticality of Defects in Cyclic Dependent Components*. in *13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2013. Eindhoven, Netherlands. p. 21-30.

[10] Oyetoyan, T.D., D.S. Cruzes, and R. Conradi, *A Study of Cyclic Dependencies on Defect Profile of Software Components.* Journal of Systems and Software, 2013. **86**(12): p. 3162-3182.

[11] Lakos, J., *Large-scale C++ software design*. 1996, Redwood City, CA: Addison-Wesley Longman.

[12] Parnas, D.L., *Designing Software for Ease of Extension and Contraction.* IEEE Transactions on Software Engineering, 1979. **SE-5**(2): p. 128-138.

[13] Jungmayr, S. *Identifying test-critical dependencies*. in *Software Maintenance*. 2002.

[14] Kung, D., Gao, J, Hsia, P, Toyoshima, Y, & Chen, C., *On Regression Testing of Object-Oriented Programs.* Journal of Systems Software, 1996. **32**(1): p. 21-40.

[15] Oyetoyan, T.D., D.S. Cruzes, and R. Conradi, *Can Refactoring Cyclic Dependent Components Reduce*

Defect-Proneness? 29th IEEE International Conference on Software Maintenance 22 - 28 September 2013 - Eindhoven, The Netherlands, 2013. in press.

[16] Cormen, T.H., et al., *Introduction to algorithms*. 2nd ed. 2001, Cambridge, Mass.: MIT Press. xxi, 1180.

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

[18] Fowler, M., *Reducing coupling.* Software, IEEE, 2001. **18**(4): p. 102-104.

[19] Martin, R.C., *Granularity, C++ Report*, 1996. p. 57-62.

[20] Briand, L.C., Y. Labiche, and W. Yihong. *Revisiting strategies for ordering class integration testing in the presence of dependency cycles*. in *Proc. 12th International Symposium on Software Reliability Engineering, (ISSRE 2001)* 2001.

[21] Hanh, V.L., et al., *Selecting an Efficient OO Integration Testing Strategy: An Experimental Comparison of Actual Strategies.* Proc. 15th European Conf. Object-Oriented Programming (ECOOP), 2001: p. 381-401.

[22] Briand, L.C., Y. Labiche, and W. Yihong, *An investigation of graph-based class integration test order strategies.* Software Engineering, IEEE Transactions on, 2003. **29**(7): p. 594-607.

[23] Melton, H. and E. Tempero, *An empirical study of cycles among classes in Java.* Empirical Software Engineering, 2007. **12**(4): p. 389-415.

[24] Binkley, D. and M. Harman. *Locating dependence clusters and dependence pollution*. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005 (ICSM'05)* 2005.

[25] Binkley, D. and M. Harman. *Identifying 'Linchpin Vertices' That Cause Large Dependence Clusters*. *9th IEEE International Working Conference on Source Code Analysis and Manipulation, 2009. SCAM '09*. 2009.

[26] Zimmermann, T. and N. Nagappan, *Predicting subsystem failures using dependency graph complexities.* ISSRE 2007: 18th IEEE International Symposium on Software Reliability Engineering, Proceedings, 2007: p. 227-236.

[27] Wasserman, S. and K. Faust, *Social network analysis : methods and applications*. Structural analysis in the social sciences. 1994, Cambridge ; New York: Cambridge University Press. xxxi, 825 p.

[28] Melton, H. and E. Tempero, *The CRSS metric for package design quality*, in *Proceedings of the thirtieth Australasian conference on Computer science - Volume 62*2007, Australian Computer Society, Inc.: Ballarat, Victoria, Australia. p. 201-210.

[29] Fenton, N.E. and S.L. Pfleeger, *Software Metrics: A Rigorous & Practical Approach*. 2nd ed. 1997, Boston: PWS Publishing Press.

[30] Li, Z.D., et al., *Characteristics of multiple-component defects and architectural hotspots: a large system case study.* Empirical Software Engineering, 2011. **16**(5): p. 667-702.

[31] Zimmermann, T., R. Premraj, and A. Zeller. *Predicting Defects for Eclipse*. in *International Workshop on Predictor Models in Software Engineering*. 2007.

[32] Dietrich, J., et al., *On the existence of high-impact refactoring opportunities in programs*, in *Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122*2012, Australian Computer Society, Inc.: Melbourne, Australia. p. 37-48.

[33] Melton, H. and E. Tempero, *JooJ: real-time support for avoiding cyclic dependencies.* Proceedings of the thirtieth Australasian conference on Computer science, 2007. **62**: p. 87-95.

[34] Shah, S.M.A., J. Dietrich, and C. McCartin, *Making Smart Moves to Untangle Programs*, in *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*2012, IEEE Computer Society. p. 359-364.

[35] Shah, S.M.A., J. Dietrich, and C. Mccartin. *On the Automation of Dependency-Breaking Refactorings in Java*. in *29th IEEE International Conference on Software Maintenance (ICSM)*. 2013. Eindhoven, Netherlands.

[36] C'ubranic, D., *Project History as a Group Memory: Learning From the Past.* , in *PhD Thesis* 2004, University of British Columbia: Canada.

[37] S'liwerski, J., T. Zimmermann, and A. Zeller, *When do changes induce fixes?*, in *Proceedings of the 2005 international workshop on Mining software repositories* 2005, ACM: St. Louis, Missouri. p. 1-5.

[38] Schroeter, A., T. Zimmermann, and A. Zeller, *Predicting component failures at design time*, in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering* 2006, ACM: Rio de Janeiro, Brazil. p. 18-27.

[39] Herzig, K. and A. Zeller, *The impact of tangled code changes*, in *Proceedings of the 10th Working Conference on Mining Software Repositories* 2013, IEEE Press: San Francisco, CA, USA. p. 121-130.

P6: Circular Dependencies and Change-Proneness: An Empirical Study

Published: In Proc. 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering 2015, École Polytechnique de Montréal, Québec, Canada, pp. 238-247

Circular Dependencies and Change-Proneness: An Empirical Study

Tosin Daniel Oyetoyan
Department of Computer and Information Systems
Norwegian University of Science and Technology
Trondheim, Norway
tosindo@idi.ntnu.no

Jens Dietrich
School of Engineering and Advanced Technology
Massey University Palmerston
North, New Zealand
J.B.Dietrich@massey.ac.nz

Jean-Rémy Falleri LaBRI,
University of Bordeaux
Bordeaux, France
jr.falleri@gmail.com

Kamil Jezek
Department of Computer Science and Engineering
University of West Bohemia Pilsen, Czech Republic
kjezek@kiv.zcu.cz

Abstract—Advice **that circular dependencies between programming artefacts should be avoided goes back to the earliest work on software design, and is well-established and rarely ques- tioned. However, empirical studies have shown that real-world (Java) programs are riddled with circular dependencies between artefacts on different levels of abstraction and aggregation. It has been suggested that additional heuristics could be used to distinguish between bad and harmless cycles, for instances by relating them to the hierarchical structure of the packages within a program, or to violations of additional design principles.**

In this study, we try to explore this question further by analysing the relationship between different kinds of circular dependencies between Java classes, and their change frequency.

We find that (1) the presence of cycles can have a significant impact on the change proneness of the classes near these cycles and (2) neither subtype knowledge nor the location of the cycle within the package containment tree are suitable criteria to distinguish between critical and harmless cycles.

Keywords—Circular dependency, maintainability, patterns

I. INTRODUCTION

Avoiding circular dependencies between software artefacts is a classic software design principle that can be traced back to Parnas' advise that modules should be organised in a hierarchy with respect to dependency relationships, thereby keeping dependencies "loop free" [31]. In the context of modern object- oriented languages, this is known as the Acyclic Dependencies Principle (ADP): The dependencies between packages must not form cycles [24].

The justification for this principle has often been related to maintenance. For instance, Parnas pointed out that it is unde- sirable to have systems where "nothing runs unless everything runs" [31]. Later work has related this to testing, where the presence of cycles prevents unit testing and requires the use of expensive methods such as the use of stubs [29].

Empirical studies on a large set of real-world Java programs have shown that these programs are riddled with circular dependencies [25], [8]. This applies to both simple circular dependencies [25] as well as to more sophisticated antipatterns like subtype knowledge [36], [8].

This seems to indicate that not all cycles are as critical for the quality of software as previously thought, and that the notion of cyclic dependencies in software must be re- evaluated. One possible approach taken by Falleri et al [11] is to distinguish between "bad" and "harmless" cycles based on the topology of dependency graph. In a nutshell, the authors argue that cycles forming in branches of the package containment tree evolve when packages grow, and are harmless, while cycles that span across the entire package containment tree are undesirable. Mutawa et al [1] studied the topology of cycles on a large set of real-world Java programs and found that (1) most cycles do form in branches of the package containment tree (and are therefore not critical according to [11]), and (2) that the parent packages are the "hubs" within these circular structures – indicating that cycles grow around these parent packages. This offers an explanation of why circular dependencies are common, and do not necessarily compromise the quality of programs.

However, the question how cycles in general and certain types of cycles in particular relate to the maintainability of programs remains open. In this paper, we present a study that investigates this issue for Java programs. We use the qualitas corpus [40] data set in our study. Maintainability is difficult to measure directly. According to IEEE 610.12,

maintenance is "the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment" [35]. Following this definition, we use change (frequency of modifications) to approximate maintainability, and therefore set out to answer the following question: *Is there a co-relation between the fact that a Java class is in a (certain kind of) cycle, and the change frequency of this class.* In other terms, do cycles incur a maintenance penalty that can be measured? We will investigate both general circular dependencies between classes and special kinds of circular dependencies that have been portrayed as particularly undesirable in previous research.

This study extends our previous work on dependency cycles where we have investigated the relationship between cycles and defects [30]. The result of this study revealed that classes within and near cycles account for the most defects in programs. This study did not investigate particular types of cycles and their relationship with change proneness. It used a smaller data set, and did not study the classes directly, but mined the comments in the issue tracking and subversion systems instead.

The rest of this paper is organised as follows: we first present the core concepts used in this paper in Section II. We then discuss related work in Section III. We describe our methodology in Section IV. We present our results in Section V and discuss them in Section VI. Finally we conclude and present the future work in Section VII.

II. BACKGROUND

A. Cycles and Dependency Graphs

The notion of cyclic dependency corresponds to strongly connected components (SCCs) in dependency graphs. SCCs can be effectively computed with Tarjan's algorithm in linear time [38].

A dependency graph is a simple model representing soft- ware artefacts and their relationships. Such a graph can be built on several levels of abstraction and aggregation. For instance, in the case of Java programs, we can consider methods and fields and their invoke and access relationships, classes and interfaces and their uses, extends and implements relationships, packages and their dependencies, and containers (jar files) and their dependencies. Low-level cycles have been associated with potential problems for comprehension, testing, and maintenance [3], [4]. However, to the best of our knowledge no empirical studies on larger sets of real-world programs exist to support this claim, and at least some of the cycles are created by widely-used programming techniques like recursion.

Higher-level dependency graphs are typically obtained from lower-level graphs by means of aggregation. For instance, a package-level dependency graph is built from the dependency graph of the classes contained in these packages. Cyclic dependencies between classes in different packages induce cyclic dependencies in the package graph. Therefore, we focus our attention on SCCs in the class graph. The vertices in this graph represent the classes of a Java program, while the edges represent the relationships between these vertices. Classes here refers to compiled classes, and also include other Java types like annotations, interfaces and enums. Edges are labelled with either *uses, extends* or *implements.* The *extends* and *implements* labels are used according to the meaning of the respective keywords defined in the Java Language Specification [15], *uses* covers all other dependencies. We also use the label *inherits* defined as the union of *extends* and *implements.*

Several empirical studies on real-world programs suggest that the number of SCCs found in both the class-level and package-level dependency graphs is large [25], [8]. The fact that many of these systems are regarded as functional and widely used suggests that not all cycles are as detrimental to the quality of systems as previously thought. This seems to indicate that it is not sufficient to only study general cycles. Instead, certain types of cycles must be studied as well in order to distinguish between critical and harmless cycles.

B. Subtype Knowledge

Subtype knowledge (STK) is an "antipattern" first studied by Riel [36]. An instance of STK is basically a cycle that has at least one *extends* or *implements* edge, and a back- reference path connecting the target of this edge with its source. Because the Java compiler (as well as most other compilers) enforces that there are no cycles in the

supertype (inherits) graph, this path must contain at least one *uses* edge. Situations producing inheritance cycles still exist when classes are compiled separately, but they are rare and can be caught by the Java Virtual Machine by means of static analysis during linking.

The intention behind this pattern is that in a well designed program, abstraction and implementation artefacts are separated, and implementation artefacts depend on abstractions, but not vice versa. This is also known as the dependency inversion principle (DIP) [22]. STK cycles directly violate this principle. Surprisingly, STK cycles are still common in real- world programs [8].

Figure 1 depicts a STK cycle found in the Java Run- time Environment, version 1.7.0. This is a class-level cy- cle, but it also induces a package level cycle between java.awt and javax.swing. The documentation of LegacyGlueFocusTraversalPolicy indicates that this is a FocusTraversalPolicy implementation that pro- vides support for legacy applications. Yet, every other imple- mentation of FocusTraversalPolicy depends on it as there is a dependency from the abstract type to this particular implementation. This is clearly an undesirable constraint for a modular design.

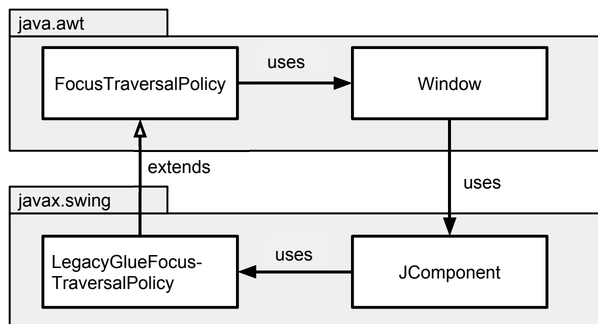


Fig. 1. A STK cycle in the Java Runtime Environment, version 1.7.0

Note that not all STK instances are equally critical. An example is discussed below in section II-D where a STK is a side-effect of using the visitor design pattern. This might still have negative consequences, however, they are outweighed by the benefits of using the design pattern.

C. Cycles and the Package Containment Tree

One possibility to distinguish between critical and harmless cycles is to consider their location within the package containment tree (PCT) [11]. The PCT of a Java program is formed by the hierarchical structure of package names. The Java language specification stipulates that “The hierarchical naming structure for packages is intended to be convenient for organizing related packages in a conventional manner, but has no significance in itself ... ” [15, ch 7.1]. However, developers seem to use some sub-package semantics when organising code. For instance, the package javax.swing has circular de- pendencies with its “child packages” javax.swing.tree and javax.swing.table. It appears that these cycles forming in branches of the PCT are the result of splitting large packages to facilitate maintainability, but the respec- tive packages retain a high level of cohesion. AWT features a similar structure. However, the core Java interface libraries also provide an example of a critical cyclic de- pendency spanning *across branches* of the PCT: AWT and Swing mutually depend on each other. Figure 1 also shows this. The critical dependency is caused by references to javax.swing.JComponent in several AWT classes, in- cluding java.awt.Window and java.awt.Component. On the other hand, javax.swing.JComponent is a sub- class of java.awt.Component. This design flaw had a significant impact on early versions of the Java platform, and there is evidence that it can be removed without impacting on the functionality of the respective libraries. This is discussed in more detail in [9].

D. Inadvertent Cycles

There are situations where cycles are a direct result of the features and limitations of technologies and methods used in projects. The most simple example in this category are the cycles formed between non-static nested classes and their outer classes in Java byte code. In particular, the compiler generates access fields to reach inner class from outer one and vice-versa.

A more complex case that is common originates from the use of certain design patterns that induce cycles. An example is the use of Visitor, one of the classic gang of four patterns [14]. The pattern consists of abstract and concrete visitors, and abstract and concrete visited “elements”. The visitors reference all concrete element types as parameters in the (overloaded) visit methods, while the element types (both abstract and concrete) use

the abstract visitor type as parameter type in the accept methods. Visitor is a very popular pattern, in particular in programs that use hierarchical data structures such as parsers for domain specific languages (DSLs). Such an example is depicted in figure 2. The cycle is even an instance of STK, caused by the inherits relationship between the concrete elements (such as ASTIdentifier) and the abstract element (Node). Note that the number of concrete elements is typically large, in this example, there are 33 such classes each representing a particular AST node type. This can result in large SCCs.

These cycles can hardly be interpreted as signs of bad design, on the contrary, the use of Visitor is widely seen as good design as it allows developers to "plug-in" functionality into complex object structures. This is also a case of choosing a particular design to overcome limitations of the programming language, in this case the lack of support for multiple dispatch in Java [26]. Acyclic versions of Visitor have been proposed [23]. However, acyclic visitors are even more complex than visitors as additional abstract visitor types are required, and it appears that they are not widely used.

In the velocity example used in figure 2, the Visitor has been manually implemented. However, in many cases parser code is generated by parser generators from abstract grammar specifications. This is becoming more and more common with the availability of good tools (such as ANTLR), and the popularity of DSLs. Code with generated cycles can have interesting change characteristics, for instance, if the code is regenerated during each iteration as part of automated builds.

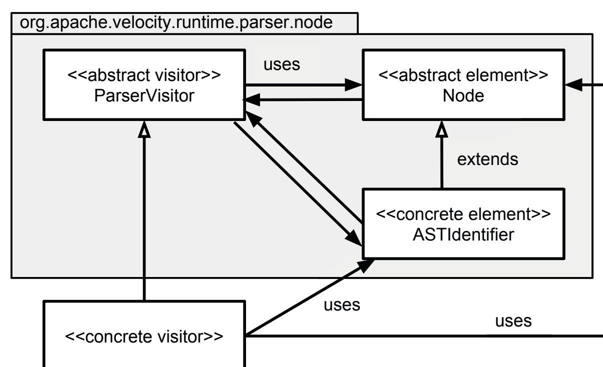


Fig. 2. A cycle caused by the use of the Visitor pattern in Apache Velocity version 1.6.2

III. RELATED WORK

Several authors have investigated the relationship between anti-patterns and the change-proneness of software artefacts. Khomh et al. [20] examined classes involved in anti- patterns and code smells and their change and fault proneness. The study investigated four systems and thirteen anti-patterns. The claims from this study are that classes participating in anti-patterns are more change- and fault-prone than others and that structural changes affect more classes with anti-patterns than others. Romano et al. [37] investigated the impact of anti-patterns on change-proneness using change data from source code analysis. The results of this study is consistent with [20]. In addition, they showed that certain anti-patterns are prone to certain types of changes such as API changes. Olbrich et al. [28] performed a study on two open source applications to study the impact of code smells. Their results show that different phases could be identified during the evolution of code smells and in particular, components infected with code smells display a higher change frequency than others. Fontana et al. [13] investigated the correlations between different smells and antipatterns.

In our study, we have investigated one particular antipattern on the structural/architectural level, and this is different from these studies.

On the other hand, while anti-patterns are claimed to be poor design choices, design patterns are recurring solutions to design problems. A plethora of studies have also investigated the relationships between design patterns and class change-proneness. Bieman et al. [2] investigated the impact of design patterns on the change proneness of classes by using five systems, four small ones and one large system. They have mined the change data from a configuration management system. They concluded that classes participating in design patterns are rather more change-prone. A recent study on mining repository [16], however showed that multiple tangled code changes could result into an incorrect classification of change/fault data.

Di Penta et al. [6] investigated whether certain design pattern roles are more change-prone in general, and whether certain roles are prone to particular types of changes. Their results confirmed that many design pattern roles do undergo changes

within the pattern. Vokac [41] analyzed the defect rates of classes that participated in selected design patterns of a large commercial product. The study concluded that Observer and Singleton patterns are correlated with large code structures and can thus serve as indicators for special attention. On the other hand, Factory pattern instances tend to have lower defect counts. Prechelt et al. [33] reported a controlled experiments that showed Observer and Decorator patterns to result in less maintenance time while the results for Visitor pattern were inconclusive. Vokac et al. [42] replicated the experiment by [33]. Their results confirmed the previous results that Observer, Decorator and Abstract Factory patterns favour ease of maintenance. However, the Visitor and Composite patterns had strongly negative results on maintenance. On the contrary, Jeanmart et al. [19] reported a positive relationship between the use of Visitor pattern and maintenance efforts.

In our study we investigate the impact of one particular anti-pattern on maintenance using change data as a proxy. We do not focus on the impact of design patterns in general, however, we discuss the impact of one particular pattern, Visitor, as it results in dependency cycles. To the best of our knowledge, there is no study that has systematically explored the relationship between change proneness and cycles. The key papers of research on cycles in dependency graphs are discussed in the previous section.

IV. METHODOLOGY

A. Data Set

We have conducted the study using the Qualitas Corpus dataset [40]. This is a curated dataset of open source real world systems that has been widely used in empirical studies on software quality issues. Using a standard dataset facilitates the replication of our study. The Qualitas Corpus version 20120401 contains 111 programs. The full release (20120401f) combines the standard release (20120401r) with the evolution release (20120401e) which contains multiple versions of pro- grams, a total of 661 versions. We chose programs that had at least 10 versions in the corpus in order to observe evolution over a longer period of time. This means that the following programs were included in this study: ant (21 versions), antlr (20), argouml (16), freecol (28), freemind (16), hibernate (100), jgraph (39), jmeter (20), jung (23), junit (23), lucene (28) and weka (55).

The scripts we have used and developed for this study can be found here: https://bitbucket.org/ootos/scc-project. Table I provides some statistics of the dataset used. A total of twelve (12) systems are analyzed consisting of 389 versions.

B. Experiment Setup

The experiments consist of the following steps to extract, process and analyse data:

1) Graph Extraction: Dependency data is extracted from Java byte code with scripts using the Apache BCEL library [5]. Since the units of maintenance are compilation units, we merge nested classes with their outer, top-level classes. The dependencies of nested classes are aggregated to their top- level classes. These aggregated classes form the vertices of the dependency graph. Extends and inherits edges are created when the respective constructs are encountered in byte code, all other occurrences of a class in the byte code of another class result in the creation of a uses edge.

2) Graph Pre-processing: We sanitise the dependency graphs by removing test classes and generated code. Test cases are removed as tests (1) tend to be more stable[46] due to the fact that in many projects they are used as specification artefacts as suggested by the test-driven development (TDD) methodology, (2) it is unusual to have cross-references between tests, and references from core functional code to tests, making it very unlikely to encounter tests that participate in cycles. We therefore believe that including tests would have skewed the results. We have also tried to remove generated code. In particular, parser APIs generated by ANTLR and similar parser generators are removed. Even minor changes in grammar definitions can produce a large amount of changes as many generated artefacts are regenerated and renamed. But this has nothing to do with whether these artefacts are in cycles or not, this is only

[46] In the context of this study, stability relates to whether a class is frequently changed or not

caused by the fact that they are generated together. On the other hand, the process of regenerating these classes often does not incur any maintenance effort, as code generation is completely automated. Note that generated parser APIs often use the Visitor pattern and therefore often contain SCCs, as discussed in section II-D.

We use simple naming pattern filters to remove tests (looking for the "Test" token in class names). To remove generated code, we have manually inspected the (ANT, Maven and Gradle) build scripts of the projects for references to code generators and the target packages names used by them. We found two projects where parser generators are used: (1) hibernate uses ANTLR and JAXB, and we excluded the following packages: org.hibernate.hql.internal.antlr.*,org.hibernate.sql.ordering.antlr and org.hibernate.internal.jaxb.*. (2) Weka uses JFlex and CUP, and we excluded the following packages: weka.core.mathematicalexpression, weka.filters.unsupervised.instance.subset-byexpression and weka.core.json.

3) SCC Detection and Classification: Once the dependency graph is built, we use an implementation of Tarjan's algorithm [38] to detect the strongly connected components (SCCs). The detected SCCs are classified in categories (STK vs non-STK, Visitor vs non-Visitor), and associated with their PCT diameter relative to the diameter of the entire dependency graph. STK is approximated by the presence of *inherits* edges in a SCC as discussed in section II-B. Visitor instances are detected based on naming patterns.

4) SCC Membership: Finally, we establish the association of a class with a cycle. The most obvious option is to look for whether the vertex representing the class is an element of the respective SCC . However, we are also interested in assessing the impact SCCs have on their direct neighbourhood, i.e., classes that are not in a cycle, but depend directly on a class within the cycle (in-neighbours), or a class in a cycle that directly depends on such a class (out-neighbours). A *neighbour* is either an in-neighbour or an out-neighbour.

5) Extracting Change Data: We use the change data set also used in [7]. This data contains fine-grained, per-class information of change classified by a change category. Details on how this is done can be found in this paper.

C. Research Questions

The general problem we are interested in is the correlation between the presence of certain types of cycles in programs, and the maintainability of these program measured in terms of change frequency, as discussed above. We break this down into the following research questions:

Firstly, we want to investigate whether a class within or near a cycle is more prone to change than a class outside a cycle. Our hypothesis is that the structural complexity associated with cycles could make it easier for change to spread to other classes within the cycle, and classes either directly referencing classes in the cycle, or being directly referenced by classes from within the cycle.

RQ1. Are classes within or near cycles more prone to change than other classes?

Secondly, we want to investigate whether classes that are in or near STK cycles are more prone to change than classes in non-STK cycles as these cycles violate a second principle of object-oriented design (the dependency inversion principle (DIP) [22]). This leads to the following question:

RQ2. Are classes in or near cycles with STK more change prone than classes in cycles without STK?

Finally, we want to investigate whether the PCT − diameter of a cycle is correlated with the change proneness of the classes within this cycle, following the argument made by Falleri et al that PCT-local cycles are less critical than cycles that span across different branches of the PCT [11]. We thus hypothesize that cycles with a large PCT-diameter would be more change-prone than those with a smaller PCT-diameter.

RQ3. Is there a correlation between the P C T − diameter of a cycle and the change frequency of the classes in or near this cycle?

D. Metrics and Measurement

For statistical analysis, we compute data series with data points for each version. The values are change probabilities, and each data series corresponds to a

set of classes resulting from a classification, such as whether a class is in or near a particular type of SCC.

1) Computing the change probability of a set of classes:

Given a program P, let C be the set of classes in P, and V be the set of versions of P such that for each version v ∈ V a successor version succ(v) exists.

For a given set of classes S ⊆ C and a version v ∈ V we use changed(S,v) to denote the set of classes in S that have changed from v to succ(v). We then define the change probability of a class in S as a function pchange : 2C × V → [0, 1] defined as:

$$p_{change}(S, v) = \frac{|changed(S,v)|}{|S|}.$$

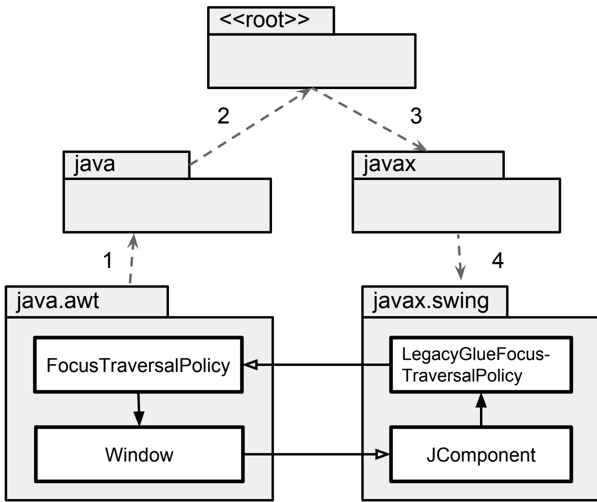


Fig. 3. The PCT-diameter of an SCC

2) Measuring the PCT diameter of an SCC: Given a set of packages P and the package containment tree (PCT) they form (see Section II-C), we compute the PCT-diameter of a set of classes as the diameter of the packages of these classes in the PCT. The PCT-diameter is computed by first computing the shortest distance between each pair of packages in the PCT and then finding the longest of the computed shortest distances. This is referred as the longest shortest path in network analysis [43]. We can normalise this value to [0,1] by dividing this number by the diameter of the set of all packages within the program.

For instance, consider the example depicted in figure 3. We have discussed the same cycle earlier. The longest shortest path between the respective packages has a length of 4 (java.awt → java → <root> → javax → javax.swing). Note that the PCT shown in this figure is incomplete, there are several core Java packages with 5 tokens, such as javax.swing.text.html.parser. Therefore, the diameter of the entire program is 10, and the PCT-diameter of the SCC in figure 3 is 0.4 (4/10). The normalised PCT computation just described defines a PCT function pct : $2^C \times V \rightarrow [0,1]$.

3) Detecting SCCs with STK: Finding instances of STK is computationally expensive as the NP-complete subgraph isomorphism problem must be solved. However, STK can be easily approximated by computing SCCs that contain at least one *inherits* edge. The drawback of this approach is that these SCCs may contain both STK and non-STK sub-cycles.

This defines a STK membership function stk : $2^C \times V \rightarrow$ {false, true}, where stk(SCC_i, v) = true iff SCC_i is a STK in version *v*.

4) Measuring Nearness of a Cycle: We also want to find out whether classes that are in the neighbourhood of a cycle of a certain type are penalized by increased change-proneness. We differentiate between outward nearness (fan-outs of the classes in cycles) and inward nearness (fan-ins of the classes in cycles). In many cases, multiple cycles can have the same

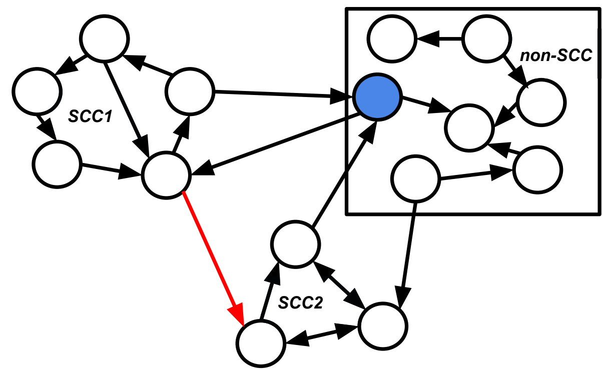


Fig. 4. Neighborhood to an SCC

neighbours. For instance, figure 4 shows an example where two cycles scc1 and scc2 share the same outward neighbour. In order to avoid assigning a class to multiple cycles, we use the following set of rules when a class is near multiple

cycles:

1) If the class changes, prioritize cycles with change. If there are multiple cycles that change, pick one randomly.
2) If the class does not change, prioritize cycles without change. If there are multiple cycles that change, pick one randomly.
3) Otherwise randomly select a cycle.

E. Statistical analysis

1) Analysis Method: The input data for the statistical analysis are provided by the three functions pchange, stk and pct that associate SCCs version pairs with information representing change probability, STK classifications and PCT values.

We want to investigate (1) the change proneness of SCCs against non-SCCs, (2) the change proneness of SCCs with STK over SCCs without STK and (3) whether the PCT diameters of SCCs are correlated with change proneness.

a) Analyzing Change Proneness of SCCs vs. Non SCCs:

We analyse two data series for the two sets of classes: the classes in SCCs, and the classes not in SCCs. The hypothesis here is that classes in SCC are more change- prone and they propagate change more to their neighbourhoods because of their structural complexity. It is easy to expand this investigation to include neighbourhoods of an SCC, by also considering neighbours (in-neighbours out-neighbours) as elements of SCCs as described above.

b) Analyzing Change Proneness of STK vs Non-STK : Here we analyse two data series: the classes within STKs, and the classes in non-STK SCCs. Note that we do not directly compare STK instances with non-SCCs, however, this relationship can be inferred by combining the results of this and the previous experiment.

c) Analysing the Correlation between PCT Diameter and Change Proneness: To answer this question, we use a slightly different method. The input data are not just two data series, but consist of two matrices where we map pairs consist- ing of versions and individual SCCs to a change probability using the formula defined above, and to the PCT diameter value, respectively.

2) Testing of the Hypotheses: We have employed two different statistical analysis methods to test our hypotheses. The choice of either one depends on the measurement type of the variables under investigation. To analyse the correlation between two data series (RQ1 and RQ2), we used a non-parametric test. To test the hypotheses in this category, the data is first tested for normality using the Shapiro test. It turned out that each dataset deviates strongly from normality. Subsequently, we use a non-parametric test (Wilcoxon rank- sum)[12] for analysis.

For interval variables used in the experiment for RQ3, we have used Pearson and Spearman correlation.

3) Measuring interactions among experimental factors: It is the goal to also understand if there are interactions among the two factors being investigated in this study. We suspect that classes with high PCT-diameter could also be prone to STK anti-pattern. It is thus appropriate to treat the two factors as a competing treatments and use one factor as a blocking factor in the experiment [12]. A nested design is chosen where the factor STK is selected as a blocking factor, since it is nominal in its scale whereas PCT-Diameter is interval. Next, the sccs are grouped into hasSTK -*True* or *False* groups and a statistical analysis is performed between PCT-Diameter and change-probability (dependent variable) in each group.

V. RESULTS

A. System Properties

Table I shows the average values for several system properties while Table II reports the (average) percentage of classes in and near cycles. Averages are computed over all versions of the respective program in the data set. The distribution of classes within SCC range from 10.3% to 80.7%. For some of the systems, a surprisingly high number of classes is within cycles, including freecol (80.7%), jgraph (77%), hibernate (62.8%) and freemind (55.3%). Two systems, jgraph and freecol, have relatively large PCT-diameter values. Freemind has the largest percentage of changed classes (53.6%) as shown in pchange column, while the rest of the

systems have change probabilities between 10.8% (jung) to 35.3% (freecol).

B. RQ1 Are classes within or near cycles more prone to change than other classes?

The results for RQ1 are presented in Table III. In column 2, the significance test results for classes within SCC against those outside SCC are listed.

While columns 3 and 4 show the results when we investigated the neighborhood of the SCCs. Only two systems (freecol and jgraph) have significant change proneness for the SCC group. However, when we considered the SCC direct neighbourhood, 75% of the systems showed significant change proneness. As shown in the results, the change frequencies of the classes increase as the size of the neighbourhood expands. This is not surprising giving that the size of the class set increases as shown in Table II. However, what is surprising is the big impact of SCCs on their neighbourhood. Investigation of the actual changes revealed that in many cases, SCCs and their direct (in-) neighbours account for more than 90 % of the total change. For instance, Ant has the average of 76.3% classes in SCCs and its direct in-neighbours, but these classes account for 94% of the total change volume. We can therefore confirm the hypothesis that the presence of SCCs could have a significant impact on the stability of the classes near those SCCs (Table VII column 3).

This may indicate a significant increase in maintenance costs, in particular as many test cases would be required to achieve sufficient coverage of the many unstable classes in the neighbourhood of cycles.

C. RQ2:Are classes in or near cycles with STK more change prone than classes in cycles without STK?

Table IV presents the results of testing this hypothesis. Column 2 of the table presents the p-values of testing SCCs with STK against SCCs without STK. The 3rd column presents the results when the *in-neighbours* are included in the S C C graph and the 4th column presents the results when both *in-neighbours* and *out-neighbours* are included in the S C C graph.

Out of the 12 systems we have studied, only 3 systems have SCCs with STK that show significant change proneness over SCCs without STK (see Table VII for summary of the results of the hypothesis).

Hibernate presents an interesting case because we detected instances of the Visitor pattern in many of its cycles. The Visitor cycles all have the STK property and the results show that in hibernate the STK cycles are more change prone than non STK cycles. To understand the role of cycles with Visitor pattern in this category, we removed the Visitor SCCs and ob- served that the mean values of the

change probability increased from 17.9% to 19.6%. That means that the Visitor SCCs are relatively stable and as a result, removing them produces an increased change ratio. For us, this is an interesting result in the sense that, although using the Visitor pattern produces instances of an ”anti-pattern” in the sense that it violates certain object-oriented design principle, nevertheless, it is stable.

A study of trade-offs between design patterns and the anti- patterns is an interesting topic for future studies.

D. RQ3:Is there a correlation between the PCT −diameter of a cycle and the change frequency of the classes in or near this cycle?

The results of testing this hypothesis is presented in table V. All values in asterisks have a correlation of 0.5 or greater and are significant at $\alpha = 0.05$. We report both the Pearson and Spearman correlation results. Only one (freecol) of the systems has a fair correlation between the PCT-diameter and the change probability. As earlier reported in Table I, freecol has a very large relative PCT-diameter. We have no result for jgraph because it only contains one SCC and as a result, one data point. We detect no consistent pattern in the relationship between the PCT-diameter of class cycles and their change proneness (see Table VII). This result is also surprising as we expected that cycles spanning across branches of the PCT would be more prone to change.

TABLE I. SUMMARY OF SYSTEM PROPERTIES, AGGREGATED VALUES ARE OBTAINED BY AGGREGATING OVER THE VALUES FOR EACH SCC AND EACH VERSION v

Systems	Versions	Num of classes	PCT-diameter		Size of STK-SCCs		Size of Non-STK SCCs		Size of SCCs		Size of Non-SCCs		$avg(p_{change}(C, v))$
		Mean	Max	Mean	Max	Mean	Max	Mean	Max	Mean	Max	Mean	
ant	21	162.7	1	0.17	5	3.79	6	3.21	205	113.95	357	211.47	0.285
antlr	20	120.9	1	0.05	6	3.22	10	5.06	166	81.89	328	159.83	0.184
argouml	16	891.6	1	0.09	4	2.92	17	11.46	855	568.92	1705	1214.31	0.325
freecol	28	189.1	1	0.67	2	1.04	2	0.46	473	305.12	90	73.12	0.353
freemind	16	45.1	1	0.33	3	1.14	9	1.79	162	49.86	333	40.29	0.536
hibernate	100	513.2	1	0.19	21	5.06	11	4.12	1406	653.65	1191	372.66	0.160
jgraph	39	25.3	1	1	1	1.00	0	0.00	39	39.00	14	11.67	0.107
jmeter	20	286.5	0.71	0.21	4	2.79	8	6.21	188	132.16	576	440.89	0.283
jung	23	152.7	0.6	0.07	4	1.91	8	6.95	48	31.82	415	273.55	0.108
junit	23	39.6	1	0.09	3	0.76	8	4.62	27	15.52	152	63.62	0.233
lucene	28	163.4	0.75	0.07	6	2.96	9	6.00	143	95.85	339	230.92	0.211
weka	55	325.3	0.875	0.05	13	5.23	26	14.53	263	86.83	969	563.87	0.139

TABLE III. WILCOXON TEST: P-VALUES OF SCCS VS. NON-SCCS ($\alpha = 0.05$)

Systems	SCCs	+ In-Neighbor	+ In/Out-Neighbor
ant	0.5	**0.035***	**0.037***
antlr	0.665	0.233	0.238
argouml	0.147	0.075	0.178
freecol	**0.004***	**0.001***	**5.27E-05***
freemind	0.198	**0.009***	**8.81E-04***
hibernate	0.052	**4.70E-05***	**0.021***
jgraph	**3.39E-10***	**9.18E-10***	**2.24E-11***
jung	0.742	**0.038***	**0.041***
junit	0.435	**0.003***	**0.010***
lucene	0.142	0.108	0.078
weka	0.511	**0.005***	**0.005***
jmeter	0.420	**0.007***	**0.022***

E. Interaction between STK and PCT-diameter

The results in table VI shows the correlation between PCT-diameter and change when grouped in the STK category and non STK category. The STK category is represented in columns 2 and 3, while the non-STK category is represented in columns 4 and 5. The results indicate that there are just two systems (freemind and hibernate) with fair correlation (see table VI). This result is different from the correlation results in Table V that reports only freecol with a relatively high and significant correlation. We therefore conclude that there is no relationship between the STK property of a cycle and the PCT- diameter of the cycle in this dataset.

TABLE IV. WILCOXON TEST: P-VALUES OF CHANGE PRONENESS OF STK-SCCS VS. NON-STK SCCS ($\alpha = 0.05$)

Systems	SCC	+ in-neighbor	+ in/out-neighbor
ant	**0.009***	**0.013***	**0.008***
antlr	0.550	0.210	0.196
argouml	0.171	0.185	0.229
freecol	**9.08E-11***	**5.45E-11***	**4.80E-11***
freemind	0.224	0.111	0.080
hibernate	**8.68E-08***	**1.38E-08***	**2.44E-09***
jgraph	-	-	-
jung	0.627	0.837	0.843
junit	0.994	0.996	0.992
lucene	0.374	0.354	0.371
weka	0.733	0.304	0.247
jmeter	0.648	0.453	0.121

TABLE V. CORRELATION TEST BETWEEN PCT-DIAMETER AND CHANGE-PROBABILITY

	SCC groups		+ in-neighbor		+ in/out-neighbor	
Systems	Pearson	Spearman	Pearson	Spearman	Pearson	Spearman
ant	0.01	0.28	0.04	0.20	0.03	0.20
antlr	-0.02	0.16	-0.16	-0.08	-0.09	-0.01
argouml	0.20	0.22	0.13	0.11	0.24	0.28
freecol	0.46	**0.64***	**0.54***	**0.78***	**0.50***	**0.69***
freemind	-0.08	-0.10	0.04	-0.08	0.10	0.04
hibernate	0.21	0.48	0.19	0.44	0.26	0.49
jgraph	-	-	-	-	-	-
jung	-0.04	-0.01	0.00	0.29	0.00	0.32
junit	0.07	0.00	0.24	0.29	0.22	0.30
lucene	-0.02	0.18	0.08	0.21	0.07	0.17
weka	0.08	0.19	0.08	0.21	0.11	0.22
jmeter	-0.02	0.08	0.06	0.17	0.25	0.32

TABLE VI. CORRELATION TEST BETWEEN PCT-DIAMETER AND CHANGE-PROBABILITY BLOCKED BY STK/NON-STK

Systems	STK		Non-STK	
	Pearson	Spearman	Pearson	Spearman
ant	-0.05	0.24	-	-
antlr	-0.09	0.17	0.04	0.09
argouml	0.22	0.31	0.12	0.07
freecol	-0.23	-0.23	-	-
freemind	**0.60***	**0.56***	-	-
hibernate	0.26	**0.61***	0.12	0.02
jgraph	-	-	0.00	0.00
jung	0.10	0.18	-0.16	-0.12
junit	0.33	0.40	-0.05	-0.04
lucene	-0.09	0.18	-	-
weka	0.07	0.17	0.13	0.20
jmeter	0.08	0.18	0.05	0.05

TABLE VII. SUMMARY OF HYPOTHESES TEST: Y DENOTES H_0 IS REJECTED

	RQ1		**RQ2**		**RQ3**	
Systems	in-SCC	in/near SCC	in-SCC	in/near SCC	in-SCC	in/near SCC
ant	N	**Y**	**Y**	**Y**	N	N
antlr	N	N	N	N	N	N
argouml	N	N	N	N	N	N
freecol	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**
freemind	N	**Y**	N	N	N	N
hibernate	N	**Y**	**Y**	**Y**	N	N
jgraph	**Y**	**Y**	-	-	N	N
jung	N	**Y**	N	N	N	N
junit	N	**Y**	N	N	N	N
lucene	N	N	N	N	N	N
weka	N	**Y**	N	N	N	N
jmeter	N	**Y**	N	N	N	N

VI. DISCUSSION

A. Cycles and the Shape of Java Programs

Overall, the results are somehow surprising, and we do not have an ultimate explanation for all the findings. However, the results seem to be consistent with some other recent research on the shape of software. Several authors have studied the networks formed by software artefacts and their relationships and found that they are scale-free, and have a heavy tail distribution with a very few nodes with high connectivity [44], [17], [18], [32].

A commonly used model to explain how scale-free net- works come to exist is preferential attachment [34] – in a nutshell, this model stipulates that nodes that are added to the network have a higher probability to link to nodes with an already high degree. In particular, in the case of software that would mean that there are classes with a high in-degree based on their popularity (because they provide useful utilities, or because they are widely known by developers), and the in- degree of these classes increases further as new classes are added to the program that use these utilities. On the other hand, classes with a lot of incoming dependencies have a high responsibility, and therefore tend to be more stable. It has been demonstrated that such a model can explain the network topology found in Java programs [39]. Conversely, this model suggests that high coupling is unavoidable [39]. This is in a way similar to the finding we made here: we found evidence that software evolution follows a pattern that leads to properties that are traditionally regarded as indicators of a bad design.

The results we obtained could therefore be explained by a model where cycles form in the heavy tail of the distribution. In particular, this would explain the results for RQ1: classes in cycles are relatively stable, but not the classes that reference the cycles (we called them "in-neighbours"). This could also offer an explanation for RQ2: developers may abstract from classes providing useful utilities, but eventually these abstractions themselves reference these utilities as they are useful, for instance, in order to provide defaults for certain services. An example where this happens is the combination of abstraction and the Singleton design pattern [14], where an abstract service class references a single instance of one of its subclasses. There are several case of this kind in the Java Runtime Environment, all with a high in-degree, including java.lang.Runtime and java.awt.Toolkit.

Note that this model is supported by the results of earlier research that many cycles form around hubs (nodes with betweenness centrality, usually corresponding to a high degree) [1], and that there are a few dependencies that support a large percentage of cycles and other antipattern instances, and therefore present high-impact refactoring opportunities [9].

However, this model does not offer an explanation for the results for RQ3. But we notice that package naming is sometimes influenced by considerations not related to the semantics of the actual code. Examples are the use of different package branches in the Java Developer Kit (such as java.*, javax.*, sun.*, org.w3c.*, ..) based on intellectual property rights, and the use of org.junit and junit branches in junit to provide older versions for backward

compatibility.

But at this stage, this is only one model that could be used to explain the observations we have made. Further research is needed to assess the validity of this explanation.

B. Threats to Validity

Graph extraction: Our tools cannot recognise weak uses relationships created by reflection. This is a common limitation for tools based on static analysis.

Graph pre-processing: Our method to recognise and remove tests is prone to both false positives and false negatives. We expect that it may make non-SCCs to appear slightly more stable for the reasons discussed in section IV-B. Our method to detect generated code may be incomplete as other scripts and tools could have been used in some projects. We think that this is unlikely as most successful open source projects automate routine tasks using build scripts.

SCC Membership: The mechanism to assign vertices to the neighbourhood of cycles is not deterministic, and this could influence the outcome of the respective experiments. However, we executed these experiments at least 10 times, and found that the impact of this on the outcome of the experiments is negligible. In addition, we did not detect any siginificant difference by using a different mechanism (e.g. random assignment of neighborhood).

Detecting STK: As described above in section IV-D3, we use an approximation to detect STK mainly for performance reasons. The result of this is that we may classify some larger STKs cycles as STK even though they are *predominantly* non- STK.

Detecting Visitors: Instances of the visitor design pattern are detected using naming patterns. This might yield both false positives and false negatives. However, in our experience the accuracy of this method is very high.

Controlling for size and dependencies: We have not con- trolled for the size of classes and the size of their dependencies within each group. Both metrics have been shown to correlate with the change/fault-proneness of components [45], [27], [21]. By investigating the size/dependencies of classes in cycle and their neighborhood, we can further understand the association between the fact that classes in and near cycles are more change-prone as reflected in the results (Table VII, column 3) and whether those classes account for the significant size and dependencies in the systems.

VII. CONCLUSION

We have investigated whether classes in and near dependency cycles are more likely to change than other classes. We did this in order to investigate whether cycles are related to poorer maintainability as change ripple effects propagate easier through cycles. We used change frequency as an indicator for maintainability. We found no evidence that classes in cycles are more change prone. However, classes in and near cycles have an increased change probability.

We also investigated two heuristics that had been proposed to distinguish between critical and harmless cycles: subtype knowledge and location of the cycle within the package containment tree (PCT). We found no strong correlation between these criteria and change proneness.

We believe that our findings indicate the need for more research to describe and detect cycles as well as other types of anti-patterns that are truly detrimental to the maintainability of a program. A particularly interesting open problem is the relationship between cycles and the scale-free property of class dependency graphs.

In addition, it would be interesting to control for the size of classes and their dependencies as it has been shown to have a confounding effect on the validity of metrics [10]. We plan to investigate this in future work.

REFERENCES

[1] Hussain A Al-Mutawa, Jens Dietrich, Stephen Marsland, and Catherine McCartin. On the shape of circular dependencies in java programs. In *Software Engineering Conference (ASWEC), 2014 23rd Australian,* pages 48–57. IEEE, 2014.

[2] James M Bieman, Greg Straw, Huxia Wang, P Willard Munger, and Roger T Alexander. Design patterns and change proneness: An examination of five evolving systems. In *Software metrics symposium, 2003. Proceedings. Ninth international,* pages 40–49. IEEE, 2003.

[3] David Binkley and Mark Harman. Locating dependence clusters and dependence pollution. In *Software*

Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on, pages 177– 186. IEEE, 2005.

[4] David Binkley and Mark Harman. Identifying'linchpin vertices' that cause large dependence clusters. In *Source Code Analysis and Manipu- lation, 2009. SCAM'09. Ninth IEEE International Working Conference on,* pages 89–98. IEEE, 2009.

[5] [5] Markus Dahm. The Apache bytecode engineering library (BCEL). *URL: http://jakarta.apache.org/bcel,* 2010.

[6] Massimiliano Di Penta, Luigi Cerulo, Yann-Gae ̈l Gue ́he ́neuc, and Giu- liano Antoniol. An empirical study of the relationships between design pattern roles and class change proneness. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on,* pages 217–226. IEEE, 2008.

[7] Jens Dietrich, Kamil Jezek, and Premek Brada. Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week- IEEE Conference on,* pages 64–73. IEEE, 2014.

[8] Jens Dietrich, Catherine McCartin, Ewan Tempero, and Syed M Ali Shah. Barriers to modularity-an empirical study to assess the potential for modularisation of java programs. In *Research into Practice–Reality and Gaps,* pages 135–150. Springer, 2010.

[9] Jens Dietrich, Catherine McCartin, Ewan Tempero, and Syed M Ali Shah. On the existence of high-impact refactoring opportunities in programs. In *Proceedings of the Thirty-fifth Australasian Computer Science Conference-Volume 122,* pages 37–48. Australian Computer Society, Inc., 2012.

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

[11] Jean-Re ́my Falleri, Simon Denier, Jannik Laval, Philippe Vismara, and Ste ́phane Ducasse. Efficient retrieval and ranking of undesired package cycles in large software systems. In *Objects, Models, Components, Patterns,* pages 260–275. Springer, 2011.

[12] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998.

[13] Francesca Arcelli Fontana, Vincenzo Ferme, Alessandro Marino, Bar- tosz Walter, and Pawel Martenka. Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on,* pages 260–269. IEEE, 2013.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Pearson Education, 1994.

[15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java TMLanguage Specification 7th Edition.* Oracle, Inc., California, USA, 2012.

[16] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on,* pages 121–130. IEEE, 2013.

[17] David Hyland-Wood, David Carrington, and Simon Kaplan. Scale- free nature of java software package, class and method collaboration graphs. In *Proceedings of the 5th International Symposium on Empirical Software Engineering, Rio de Janeiro, Brasil.* Citeseer, 2006.

[18] Makoto Ichii, Makoto Matsushita, and Katsuro Inoue. An exploration of power-law in use-relation of java software systems. In *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on,* pages 422–431. IEEE, 2008.

[19] Sebastien Jeanmart, Yann-Gael Gueheneuc, Houari Sahraoui, and Naji Habra. Impact of the visitor pattern on program comprehension and maintenance. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement,* ESEM '09, pages 69–78, Washington, DC, USA, 2009. IEEE Computer Society.

[20] Foutse Khomh, MassimilianoDi Penta, Yann-Gal Guhneuc, and Giu- liano Antoniol. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering,* 17(3):243–275, 2012.

[21] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering,* page 50. ACM, 2012.

[22] Robert C Martin. The dependency inversion principle. *C++ Report,* 8(6):61–66, 1996.

[23] Robert C Martin. Acyclic visitor. In *Pattern languages of program design 3,* pages 93–103. Addison-Wesley Longman Publishing Co., Inc., 1997.

[24] Robert C Martin. Design principles and design patterns. *Object Mentor,* 1:34, 2000.

[25] Hayden Melton and Ewan Tempero. An empirical study of cycles among classes in java. *Empirical Software Engineering,* 12(4):389– 415, 2007.

[26] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. In *Acm sigplan notices,* volume 43, pages 563–582. ACM, 2008.

[27] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on,* pages 284–292. IEEE, 2005.

[28] Steffen M Olbrich, Daniela S Cruzes, and Dag IK Sjoberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *Software Maintenance (ICSM), 2010 IEEE International Conference on,* pages 1–10. IEEE, 2010.

[29] Jan Overbeck. *Integration testing for object-oriented software.* PhD thesis, Vienna University of Technology, Vienna, Austria, 1994.

[30] Tosin Daniel Oyetoyan, Daniela S Cruzes, and Reidar

Conradi. A study of cyclic dependencies on defect profile of software components. *Journal of Systems and Software,* 86(12):3162–3182, 2013.

[31] David Parnas. Designing software for ease of extension and contraction. *Software Engineering, IEEE Transactions on,* (2):128–138, 1979.

[32] Alex Potanin, James Noble, Marcus Frean, and Robert Biddle. Scale- free geometry in oo programs. *Communications of the ACM,* 48(5):99– 103, 2005.

[33] Lutz Prechelt, Barbara Unger, Walter F. Tichy, Peter Brossler, and Lawrence G. Votta. A controlled experiment in maintenance: comparing design patterns to simpler solutions. *Software Engineering, IEEE Transactions on,* 27(12):1134–1144, 2001.

[34] Derek de Solla Price. A general theory of bibliometric and other cumulative advantage processes. *Journal of the American Society for Information Science,* 27(5):292–306, 1976.

[35] Jane Radatz, Anne Geraci, and Freny Katki. IEEE standard glossary of software engineering terminology. *IEEE Std,* 610121990:121990, 1990.

[36] Arthur J Riel. *Object-oriented design heuristics*. Addison-Wesley Publishing Company, 1996.

[37] Daniele Romano, Paulius Raila, Martin Pinzger, and Foutse Khomh. Analyzing the impact of antipatterns on change-proneness using fine- grained source code changes. In *Reverse Engineering (WCRE), 2012 19th Working Conference on,* pages 437–446. IEEE, 2012.

[38] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing,* 1(2):146–160, 1972.

[39] Craig Taube-Schock, Robert J Walker, and Ian H Witten. Can we avoid high coupling? In *ECOOP 2011–Object-Oriented Programming,* pages 204–228. Springer, 2011.

[40] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. Qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Soft- ware Engineering Conference (APSEC2010),* pages 336–345, December 2010.

[41] Marek Vokac. Defect frequency and design patterns: An empirical study of industrial code. *Software Engineering, IEEE Transactions on,* 30(12):904–917, 2004.

[42] Marek Voka ́cˇ, Walter Tichy, Dag IK Sjøberg, Erik Arisholm, and Magne Aldrin. A controlled experiment comparing the maintainability of programs designed with and without design patternsa replication in a real programming environment. *Empirical Software Engineering,* 9(3):149–195, 2004.

[43] S. Wasserman and K Faust. *Social network analysis : methods and applications. Structural analysis in the social sciences*. Cambridge University Press, 1994.

[44] Richard Wheeldon and Steve Counsell. Power law distributions in class relationships. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on,* pages 45–54. IEEE, 2003.

[45] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering,* pages 531–540. ACM, 2008.

P7: A Decision Support System to Refactor Class Cycles

Accepted at the: 31st International Conference on Software Maintenance and Evolution ICSME2015, Bremen, Germany

A Decision Support System to Refactor Class Cycles

Tosin Daniel Oyetoyan
Computer and Information Science
Norwegian University of Science and Technology, Trondheim, Norway
tosindo@idi.ntnu.no

Daniela Soares Cruzes
Software Engineering, Safety and Security
SINTEF, Trondheim, Norway
daniela.s.cruzes@sintef.no

Christian Thurmann-Nielsen
EVRY ASA,
Oslo, Norway
christian.thurmann-nielsen@evry.com

***Abstract*—Many studies show that real-world systems are riddled with large dependency cycles among software classes. Dependency cycles are claimed to affect quality factors such as testability, extensibility, modifiability, and reusability. Recent studies reveal that most defects are concentrated in classes that are in and near cycles. In this paper, we (1) propose a new metric: IRCRSS based on the Class Reachability Set Size (CRSS) to identify the reduction ratio between the CRSS of a class and its interfaces, and (2) presents a cycle-breaking decision support system (CB-DSS) that implements existing design approaches in combination with class edge contextual data. Evaluations of multiple systems show that (1) the IRCRSS metric can be used to identify fewer classes as candidates for breaking large cycles, thus reducing refactoring effort, and (2) the CB-DSS can assist software engineers to plan restructuring of classes involved in complex dependency cycles.**

Index Terms—*Dependency cycle, CRSS, refactoring, software quality, decision support system.*

1. Introduction

Best design practice advocates to avoid dependency cycles between software artifacts [1-4]. Dependency cycles are claimed to increase structural complexity among software artifacts such as classes or packages, and to inhibit software qualities like understandability, modifiability, testability, reusability and extensibility [1, 2, 4-6]. Testing a class in isolation is practically impossible when in a cycle with other classes [2]. A class that is tied to a large chunk of unnecessary classes cannot be reused effectively [2]. In integration testing, cycles prevent the topological ordering of classes that can be used as a test order [7-9], thereby inhibiting the testability. Recent studies have investigated the relationship between dependency cycles and defects [10-12], and found that most of the defects are concentrated within components in and near cycles.

Application development frameworks have considered binding dependencies at runtime to better manage dependencies and provide loose coupling among modules, e.g., dependency injection frameworks (e.g. Spring framework [13]) and dynamic component models (e.g. OSGi framework [14])

Despite these advances, empirical evidence shows that dependency cycles are pervasive in modern software systems [15, 16], at different granularity levels. Time-to-market often forces developers to accumulate technical debt, e.g., by focusing more on "functional code" rather than "maintainable code" [17]. This suggests a need for approaches and tools to deal with accumulated technical debt through refactoring of large and complex cycles.

A major motivation for developing a cycle breaking decision support system is based on dialog with an industrial partner seeking to refactor class cycles, but who found no support in the C# development environment (Visual Studio). The developers do not envision an automated approach or tool where they loose control of the code structure and organization after refactoring. One respondent says: *"When you have a complex part of code, it seems more like you are losing control when you just press a button and it does everything for you: which is not ideal. Especially when there is complex code and you want to know what's going on when you are debugging*".

Against this background, we have implemented a decision support system (DSS) for refactoring class cycles. It is called DSS because it proposes architectural refactoring actions to maintenance engineers, and indicates code locations where actions can be manually implemented. The problem of breaking dependency cycles at the class granularity level is not trivial. Class cycles are large and much more complex than cycles at higher abstraction layers.

Breaking large cycles requires heuristics to suggest the minimum edges that should be treated (e.g. *greedy cycle removal*) [18]. Such heuristics have been applied to dependency cycle problems among software artifacts (e.g. in [19]). However, there are challenges with *cycle removal* heuristics

when applied to software artifacts, e.g., there are edges suggested that are impractical to refactor [19]. They do not take into account the effect each edge removal/reversal has on the current structure of the system. For large and complex cycles, the minimum number of edges to break the cycles is usually large, and translates to creating a large number of new components. Lastly, breaking the suggested edges does not guarantee that the cycles would be removed. Approaches and tools are therefore needed to simulate refactoring in an adaptive and dynamic way.

In this context, we have investigated metrics to support cycle removal heuristics. This paper proposes a new metric named **interface-CRSS reduction rate (IRCRSS)**, based on the class reachability set size (CRSS) metric proposed in [20]. The CRSS metric counts for a given class, all other classes in the system's source code it requires for its compilation. The CRSS metric was chosen because it provides possibility to limit the number of components to be introduced during cycle breaking refactoring. This discussion is elaborated in Section II. The proposed IRCRSS metric and approach are evaluated using the cycle-breaking decision support system (CB-DSS).

Three research questions are stated to determine the performance of the new metric and the usefulness of the CB-DSS.

RQ1 Is the system restructuring better with IRCRSS metric? Will tuning with IRCRSS produce a refactoring fitness that is better than refactoring without?

RQ2 Will tuning with IRCRSS always improve software structure? Is it a common property that tuning with IRCRSS finds better fitness in every system? How many applications exhibit this opportunity?

RQ3 Can the use of IRCRSS metric reduce the restructuring effort? We want to find out whether tuning with IRCRSS reduces the number of refactoring edges.

Lastly, we performed a qualitative evaluation of the CB-DSS in an industrial setup.

This paper is structured to partly follow the design science research methodology [21]. The problem identification is discussed in this section. Section II provides the background of this work. Section III presents the implementation of CB-DSS. Section IV presents the results of validating the approach. Section V provides the evaluation of the metric and the system on different cases. Section VI draws out the threats and limitations of the system. Lastly, the conclusion is in Section VII.

2. Background

A. Class Reachability Set Size (CRSS) metric

Melton and Tempero [20] present a metric named "class reachability set size" (CRSS) to detect package partitioning problems in software systems, and propose a refactoring strategy that uses CRSS to improve the package design quality. By investigating the distribution of CRSS values for all classes in a system, it is possible to identify whether the relationships among the classes preclude them from a "good partitioning". The notion of "good partitioning" is measured by how package design affects software quality attributes, like deployability, understandability, reusability, and testability. A good package design can be quantified by the **manageable size**, **stand-alone**, **cohesion,** and **encapsulation** principles. Two of these properties (**manageable size** and **stand-alone**) are focused in [20].

Package dependencies are aggregated at the class (compilation) abstraction level. Thus, the distribution of the *class reachability set size* values of the classes in the whole system can be effectively used to understand its package formation problems. The CRSS metric is computed from the *Class Dependency Graph* (CDG). The shape of the CRSS distribution provides information about the underlying Package Dependency Graph (PDG). A situation where there are many classes with large CRSS values shows a symptom of tall or cyclic PDGs and cannot be easily separated to *stand-alone* and of *manageable sizes* unless the **class relationships** are refactored. An example is the case of Azureus application (Vuze in later versions) in Fig. 1. It has approximately 1900 top-level class files. About 1000 of these class files transitively depend on 1300-1500 other classes, while approximately 900 of the classes transitively depend on 1-100 classes.

A refactoring strategy based on the dependency inversion principle [22] and a registry of singletons [23], is proposed to decouple classes and reduce CRSS values for systems with large CRSS values. This strategy is applied by extracting interfaces from 10 identified candidates. The candidates are classes widely referenced and have high CRSS values. The result after the 10th refactoring showed only 400 classes to have CRSS value of 1300+, and nearly 1300 classes transitively depended on less than 100 other classes.

B. Minimum Feedback Edge Set (mFES) and CRSS metrics

In graph theory [24], strongly connected components (SCC) also known as a cyclic dependency graph in a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, both are reachable from each other. An SCC can consist of several directed cyclic graphs as shown in Fig. 2 with one SCC containing two different cyclic sub graphs (**A, D, C, B** and **A, D, C, F, E**). The problem to solve is to eliminate undesirable SCCs among system classes and obtain a directed acyclic graph (DAG). Finding the smallest number of edges (minimum feedback arc/edge set) whose reversal or removal can turn a SCC into a DAG is an NP-complete problem [18]. It is therefore common to employ heuristics (e.g. *greedy cycle removal*) [18].

Sometimes, the minimum feedback edge set (mFES) is not 'small' in many software systems. To implement mFES for cycle breaking, would involve creating several new classes. For instance, to turn the SCC in Fig. 2a would require creating a new component **J** (as in Fig. 2b) to break the edge between **D** and **C** (mFES). Arguably, this edge (**D → C**) can be reversed. In reality, however, edges (relationships) between classes cannot just be reversed as they involve much more complex interactions. The mFES for Azureus 2.3.0.2 using the "greedy cycle removal" algorithm [18], gave **211** edges that should be treated (removed/reversed) to turn the SCC with **804** classes and **4275** edges to a DAG. A challenge is the need to create many new classes or interfaces to break the SCC. More challenging is the fact that not all the suggested edges in the mFES could be treated, as they represent relationships considered as strong coupling (e.g., an edge between a class and its abstract type).

In the example of Azureus 2.3.0.2 above, by utilizing the interfaces of 10 identified classes as candidates (with high CRSS and incoming dependencies), the SCC with 804 classes could be reduced to 253 (nearly 68% reduction).

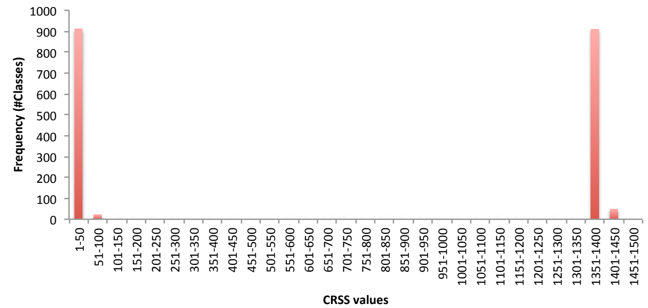


Fig. 1. CRSS distribution of Azureus (Vuze)

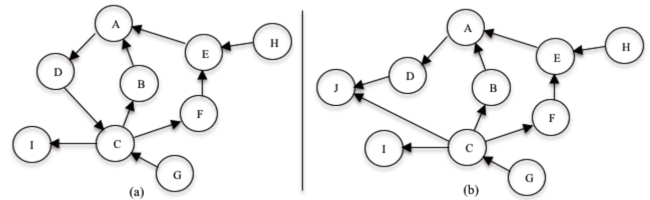


Fig. 2. (a) an SCC with 2 inner cycles and (b) The refactored SCC [11]

This motivated us to consider the CRSS metric before the mFES metric, when seeking to perform cycle breaking. Classes with high CRSS are typically in a large SCC or in the neighborhood of SCC (see an example of components **H** and **G** that depend on the SCC in Fig. 2a). By using the CRSS metric as an objective function, we do not only refactor classes in complex SCCs and their neighborhoods, but we also create a decoupled system that fits the discussions of **manageable sizes** and **standalone** properties of package design.

C. A new metric based on CRSS

Following the discussion above, when an interface is introduced for decoupling, the extent that the CRSS values of the clients can be reduced, may be based on the CRSS values of the utilized interfaces of the candidates. The reason is that the interface would only depend on the types declared in the published method's signatures of its implementation. We establish that one candidate might be better than the other because their methods' signatures are not tightly coupled to different concrete classes. This can be done by

inspecting the CRSS values of both the extracted interface and the implementation of the candidate. If the CRSS values of the implementation and its interface are pretty much the same, we say that this may be a non-optimal refactoring point. Essentially, we may not have any reduction immediately in the transitive coupling but rather an increase in coupling because of this refactoring. We demonstrate this concept with the following examples: Consider classes ***A*** and ***B*** (Listing 1): Class ***A*** depends on class ***C*** as the parameter type of method **ma1**, on class ***D*** as a return type of method **ma2**, and on class ***G*** as an exception type. Similarly, class ***B*** depends

```
public class A {
    public void ma1(C c) {/*…*/}
    public List<D> ma2() { return new ArrayList<D>();}
    public void ma3() throws G{/*…*/}
}
public class B {
        private double fb;
        public void mb1(String s) { D.md1(s)}
        public double mb2(int n) { return n*fb;}
        public int mb3(){return -1*C.fc }
}
```

Listing 1. Opportunities in relation to the CRSS metric

```
public interface IA {
    public abstract void ma1(C c);
    public abstract List<D> ma2();
    public abstract void ma3() throws G;
}
public interface IB {
        public abstract void mb1(String s);
        public abstract double mb2(int n);
        public abstract int mb3();
}
```

Listing 2. Default extracted interfaces of A and B

on ***C*** in the method body **mb3** and on ***D*** in the method body **mb1**. If we assume that class ***C*** can transitively reach 100 other classes, ***D*** can reach 200 other classes, and ***G*** can reach 2 other classes to compile. An ***Extract interface*** performed on ***A*** and ***B*** would produce interfaces, ***IA*** and ***IB*** as shown in Listing 2. A close observation of the two interfaces shows that ***IA*** still has dependencies on ***C*** and ***D*** and therefore would have at least a transitive dependency of 200. This value is the same as the maximum CRSS (200) of its actual implementation class ***A***. Whereas, the interface ***IB*** contains no dependencies on the concrete implementations in class ***B*** and therefore has a maximum CRSS of 0 while the implementation class ***B*** has at least a CRSS of 200.

Using this background, we determine a new metric named interface-CRSS reduction rate that is based on the difference between the CRSS value of a class and its interface. Formally, we define the interface-CRSS reduction rate for a class X as:

$$IRCRSS(X) = \frac{CRSS(X) - CRSS(IX)}{CRSS(X)}$$

Where:

IRCRSS (***X***) is the class reachability set size (CRSS) reduction rate for the interface of ***X***
CRSS (***X***) is the class reachability set size of ***X***
CRSS (***IX***) is the class reachability set size of the interface of ***X*** (***IX***)

The IRCRSS of **X** gives the likely rate at which the CRSS value of a client **Y** that depends on **X** would be reduced if it depends on the interface of **X** (i.e. **IX**). The value of IRCRSS ranges from 0 to 1. A value of zero implies no reduction in the CRSS value when the dependency of **Y** is changed from **X** to **IX**, while a value of 1 implies a possible 100% reduction.

D. Strategies for edge breaking between a source and a target type

The dependency between two program classes can be represented as: *source* depends on *target* (source → target), where the *target* class is used within the *source* class. We have used these notations "*source*" and "*target*" in the following presentation. In addition, we have used standard refactoring notations [25] such as ***Extract interface***, ***Move method***, ***Move field***, ***Encapsulate field*** and so on in our presentation.

1) Type generalization

Type generalization involves declaring a variable with its abstraction (interface or abstract class). This is considered a good programming practice [25]. In general, when an interface of an implementation type is introduced, it should be utilized by all of its clients wherever possible [26]. However, studies show that interface types are sparingly used in software development despite its several potentials [26, 27]. Type generalization can be used to break dependency between a source type and a target type when it is a case of aggregation (*has-a*) and not composition (*part-of*).

2) Registry (Service Locator)

Two cases are considered here. First, the target's constructor is explicitly invoked through a "**new**" keyword in the source class (*part-of*). This type requires that the source use a new object of the target class. To break this dependency, the target class needs to be cloned each time a new object is requested by the source classes. The pattern is referred as the Registry of Prototypes [23].

Second, when a utility class (that contains only static members) has high incoming dependencies and it is a hub for big and complex SCCs. It might be needful to refactor the utility class into a singleton and its static methods to instance-side methods to break such complex SCCs. This case is dealt with in [20] using the Registry of Singletons/Service Locator pattern [23] [28].

The instantiation of the target classes for Java applications can be done with the ServiceLoader[47] or in the entry class of the application [20]. For C# applications, the lightweight injection container called Unity[48] can be used to configure target classes. The refactoring may sometimes require some extra modifications to the target's class. An example is when the target class uses parameters in its constructor(s). It is not possible to pre assign these parameters when configuring the instance of the target class. One solution is to modify the target as shown in Listing 3. A new empty constructor and a new public setter method are created in Class **A**. The public setter method takes the parameters of the first constructor. The body of the constructor with parameters is moved to the setter method and replaced with a reference to the method. This way, the code is not broken and refactored clients can request the instance of the target and pass the parameters through the setter method.

3) Static final (read-only) field (Copy field)

High coupling between the source and a target class could occur because of static final field invocation or static field that is used as read-only (final). An example is a case in Azureus v2.3.0.2, where "***BackGroundGraphic***" class depends on "***MainWindow***" class because it uses a ***static Color white***. The bizarre decision here is that while other color types were defined and used in the ***BackGroundGraphic*** class, the developer simply referenced the color field "***white***" from the ***MainWindow*** class rather than defining it in the ***BackGroundGraphic*** class. This has been refactored in the latest version by moving "***Color white***" to the ***BackGroundGraphic*** class. The refactoring approach here is to ***Copy field*** from the target to the source class. This makes sense because the value of such final field would not change or become updated.

```
/* Before refactoring */
public class A {
    public A (B b) { run(b);}
}
/* After refactoring */
public class A {
    public A () {/*…*/}                    /*new empty constructor*/
    public A (B b) { setB(b);}      /*keep to not break the code*/
    public void setB(B b) { run(b);}
}
```

Listing 3. Refactoring target with parameters in its constructor

4) Encapsulate static field

A source class can use a target class through static field invocation. Listing 4 presents an example where class **A** is coupled to class **B** through a static field. ***Encapsulate field*** [25] and ***Extract interface*** with ***Registry*** refactoring can be applied to break this dependency. The static field (***fb***) is declared private in **B** and assigned to an auxiliary instance field[49] **id**. A getter method is

```
/* Before refactoring */
public class A {
    public void m() { B.fb; }
}
public class B {
    public static int fb;
}
/* After refactoring */
public class A {
    public void m() { IB b = Registry.getBImpl();
    b.getFbFromID();}
}
public class B implements IB{
   private static int fb;
   private int id;
   public B () { id = fb;}
   public int getFbFromID() { return id;}
}
public interface IB {
    public abstract int getFbFromID();
}
```

Listing 4. Encapsulation and Extract Interface + Registry refactoring for static field dependency

[47] https://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html
[48] http://msdn.microsoft.com/en-us/library/ff649614.aspx
[49] https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html

declared for **id** and declared in interface **IB**. Class **A** can then access ***fb*** through the instance field **id**.

5) Inline Static Method

A source can depend on a target through the invocation of the target's static method. To inline a static method would imply moving the method from the target to the source and creating a delegate in the target's method to the moved method in the source [29]. Essentially, the dependency is reversed. This is similar to a situation where ***Move method*** [25] is applied to break a dependency, however, this does not involve reversing the dependency. Moving method body can create some recursive actions and higher reachability size. We therefore propose an ***Extract Interface*** with ***Registry of singletons*** refactoring when a target class has incoming dependencies that is more than one and static method inline when it has only one.

```
public class A extends B implements C {
    private D d;
    private static E e = new E();
    public F meth(G g, D d) throws H {
        this.d = d;
        P.log(Q.Status, R.ID);    /*Assume ID is a final variable in class R*/
        return (F) g.typeOfF();
    }
}
```

Listing 5. Example of dependency types

E. Dependency types and refactoring strategy

Dependency can be formed in different ways between a source class and a target class [30]. We illustrate this with the following example code snippet in Listing 5. In this snippet, class **A** depends on classes **B, C, D, E, F, G, H, P, Q,** and **R**. Table 1 lists the default strategy for the dependency types.

TABLE I. DEPENDENCY TYPES AND DEFAULT REFACTORING STRATEGY

Dependency type	**Example**	**Default Refactoring Strategy**
Variable declaration	A uses D, E	Extract interface
Variable declaration with initialization	A *uses* E	Extract interface + Registry of prototype
Method return type	A uses F	Extract interface
Method parameter type	A uses G, D	Extract interface
Method exception type	A uses H	Extract interface (abstract class)
Static method invocation	A uses P	Inline Method or Extract interface + Registry of singleton
Static field invocation	A uses Q	Encapsulate + Extract interface + Registry
Static final field invocation	A uses R	Copy field or Move field to Interface (Extract interface)
Constructor invocation	A uses E	Extract interface + Registry of prototype
Super type	A uses B	None
Interface type	A uses C	None
Others (e.g. casting)	A uses F	Extract interface

F. Related cycle-breaking studies and tools

Graph transformation has been extensively applied in software engineering and notably in code-level refactoring activities [31, 32]. The type of graph manipulation we have employed in this study does not demand detailed graph formalism since we are only interested in removing or adding single edges in a graph. We therefore limit our discussion to other studies devoted in this manner to cycle-breaking refactoring.

Dietrich and McCartin [30] identified high impact edges from the program dependency graph by assigning weights to edges based on the number of anti-patterns they are involved with. Their results on the graph model demonstrated that many anti-patterns (e.g., dependency cycles at the package level) could be removed by removing such high impact edges. [29] implemented an automated refactoring on these edges using various refactoring techniques. Their results show that certain edges are removable, while removing certain edges would introduce errors.

Laval and Ducasse [33] implemented an enriched dependency structural matrix (eDSM) to detect dependency cycles between packages. They use contextual information, e.g. types of relationships between the coupled components and the proportion of referencing classes in the client package. The tool reports actions to be performed to remove detected dependency cycles.

Several other tools have been proposed to detect cycles. For instance, JDepend [50], NDepend [51], JooJ [19], Dependometer [52], Classycle[53], STAN[54], Jepends [34], PASTA [35], Lattix [55], and Structure101 [56]. Of the aforementioned approaches and tools, only the work of [33] has close similarity to ours in the sense that they used context data to propose refactoring actions. However, it differs in focus because we have considered breaking dependency cycles at the class granularity level.

3. Implementation

We have built a CB-DSS in Java (publicly accessible at: https://bitbucket.org/ootos/j-guirestructurer and used the Jepends-bcel by Melton [57] to collect dependency data. The dependency data is collected from the bytecode of Java classes using the Apache Byte Code Engineering Library[58] (BCEL). We are interested in top-level classes (compilation units), since they represent maintenance units. Therefore, the dependencies of nested classes are aggregated to their top-level classes. An MSc student has integrated the CB-DSS into a Visual Studio plugin (Accessible at https://bitbucket.org/ootos/c-sharprestructurer). The simple model diagram for the CB-DSS is shown in Fig 3. There are seven major components of the model: (1) decision support table 1 - DSTable1, (2) decision support table 2 – DSTable2, (3) the dependency types – UsageType, (4) refactoring strategy - Strategy, and (5) RefactoringSimulation, (6) System Restructuring, and (7) Cycle breaking.

A. DSTable 1

This table implements the IRCRSS metric for each class in the application. IRCRSS value ranges between 0 and 1. The table is used as a look up table to decide the choice of the best class as candidate for refactoring. The selection mechanism from DSTable-1 is driven by IRCRSS, high incoming dependencies (FAN-IN), high CRSS and high SCC values. The list of candidates is selected using the following rules:

1. The candidate must fall within the specified topKs positions for all the three measures (FAN-IN, CRSS, SCC)
2. The IRCRSS value of the candidate must be equal or greater than the specified value by the user
3. The candidate must not be an interface or an abstract class (since these types cannot be instantiated)

Next, the selected topK classes are sorted by using four attributes FAN-IN, CRSS, SCC and STATIC. The STATIC variable implies that all the class members are static. In some applications, static members are usually widely referenced and are potential hub for large SCCs. The sorting is implemented by selecting the principal attribute. The algorithm then sorts on the principal attribute and two other attributes. The possible combination of sorting is as follows (the bold and underlined attribute denotes the principal sort attribute):

1) **STATIC**, FAN-IN, CRSS
2) **CRSS**, FAN-IN, SCC
3) **FAN-IN**, CRSS, SCC
4) **SCC**, FAN-IN, CRSS

We have decided on these sorting combinations based on the results of several experiments. The sorting combination orders produced the best refactoring results on different systems.

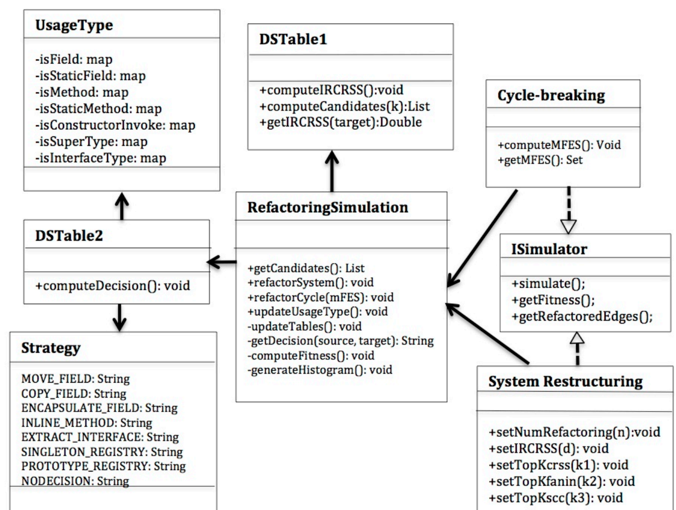


Fig. 3. Class model for the CB-DSS

B. DSTable 2

This table stores context data and computed refactoring decisions for each edge (source →

[50] http://clarkware.com/software/JDepend.html
[51] http://www.ndepend.com
[52] http://source.valtech.com/display/dpm/Dependometer
[53] http://classycle.sourceforge.net
[54] http://stan4j.com
[55] http://lattix.com
[56] http://structure101.com/products
[57] https://www.cs.auckland.ac.nz/~hayden/software.htm
[58] http://commons.apache.org/proper/commons-bcel/

target). The DSTable-2 is computed by using the ***UsageType*** and the default refactoring ***Strategy*** as described in Table I. The DSTable-2 serves as a look up directory to select the refactoring decision for each suggested edge during refactoring.

C. System restructuring

The approach is to begin every refactoring with ***System restructuring*** with fitness function as CRSS and SCC. This refactoring focuses on decoupling the entire software structure and it uses the DSTable-1 to determine the classes that are candidates for refactoring. A pre-selected number (N) of refactoring iteration and a combination of tuning and sorting parameters are presented to the ***RefactoringSimulation*** module. For each refactoring, the system selects the best class as candidate from DSTable-1 and then simulates the refactoring of all classes that depend on the candidate. The refactoring strategy to break each edge (class → candidate) is selected from DSTable-2. At the end of each refactoring, the fitness values are computed and a list of refactoring actions are generated (e.g. see Fig 4). The general refactoring is performed as follows:

a Create the interface (or abstract class) for the selected candidate (Class)
b Create an edge between the candidate and its interface
c Move all published dependencies of the candidate to its newly created interface
d Create the registry class and the respective edges from the candidate and the main class
e Update all relevant relationships and edges
f Determine the refactoring strategy
g Compute the SCC and fitness values
h Update decision tables

D. Cycle-breaking refactoring

This is used to further resolve SCCs that are not refactored during the "***System restructuring***". It is driven by selecting the SCC of interest and then activating a "greedy cycle removal" algorithm [18] to determine the minimum feedback edge set (mFES). The mFES is passed to the ***RefactoringSimulation*** module. The refactoring strategies for each edge are looked up from DSTable-2. The refactoring for this edge is then simulated based on the returned refactoring strategy.

4. Validation

We report on four case studies to evaluate the accuracy of the CB-DSS. In the first case study, we performed refactoring on Azurues 2.3.0.2 using ten candidates. Next, we refactored JStock using five candidates. The refactoring for the above two case studies were performed by one of the authors. The third case study is VidCoder, an open source application, developed in C#. An MSc student has performed the refactoring of six candidates for this case study. The fourth case study (commApp) is an industrial Smart Grid application developed with C#. The company's software maintenance engineer has performed the actual refactoring of three candidates. For space reason, the properties of the selected applications can be found here: http://www.idi.ntnu.no/~tosindo/resources/systems.pdf

We summarized the results of the validations, performed on the different case studies in Table II. As shown in the Table, the fitness values, mean (CRSS), and max (SCC) for manual refactoring are close to the fitness values of the CB-DSS. For both Azureus and JStock, the results of the CB-DSS and the actual refactoring are nearly the same. For both VidCoder and commApp, the results of manual refactoring are modest and are reasonably comparable to the result of the CB-DSS. In the case of VidCoder, the differences could be due to the fact that 5 edges out of the 9 proposed were not refactored. The reason is that the developer used the lightweight injection container (Unity) in Visual Studio instead of defining a custom registry class. This is a positive contribution as the developers have control during the refactoring activities. For commApp, some of the changes made by the maintenance engineer involved additional refactoring such as splitting a class into two and thereby increasing both the number of nodes (classes) and edges (relationships). These would affect the fitness values. Overall, the validation's results show that it is possible to use the CB-DSS as a decision support tool for planning refactoring activities.

Source	Action/Relationship	Target	Strategy	Location in Code	
ted.TedMainDialog_Interface	CREATE		NEW_INTERFACE		
ted.TedMainDialog_Interface	USES	ted.TedTable	DECLARE_IN_MEMBER	getSerieTable()	actionPerformed()
ted.TedMainDialog_Interface	USES	ted.TedSerie	DECLARE_IN_MEMBER	actionPerformed()	resetStatusOfAllShows()
ted.TedMainDialog_Interface	USES	ted.ui.logdialog.TedLogDialog	DECLARE_IN_MEMBER	windowClosing()	tLog
ted.TedMainDialog_Interface	USES	ted.TrayIcon	DECLARE_IN_MEMBER	setIcon()	updateGUI()
ted.TedMainDialog_Interface	USES	ted.TedTablePopupMenu	DECLARE_IN_MEMBER	initGUI()	updateGUI()
ted.TedMainDialog	IMPLEMENTS	ted.TedMainDialog_Interface	IMPLEMENTS		
refactor.Registry	CREATE		NEW_CLASS		
ted.TedMain	USES	refactor.Registry	SINGLETON_REGISTRY		

Fig. 4. Example of proposed refactoring actions for manual execution

TABLE II. RESULTS OF VALIDATION

Fitness	Azureus (N=10)			JStock (N=5)			VidCoder (N=6)			commApp (N=3)		
	BR	AR (CB-DSS)	AR (Actual)	BR	AR (CB-DSS)	AR (Actual)	BR	AR (CB-DSS)	AR (Actual)	BR	AR (CB-DSS)	AR (Actual)
Mean (CRSS)	703.67	295.46	295.66	155.01	108.57	108.57	17.22	10.87	11.32	115.06	115.88	115.08
Std.Dev (CRSS)	684.88	525.28	525.53	133.47	128.84	128.84	37.63	21.07	24.91	285.82	286.36	284.61
Mean (SCC)	22.82	7.36	7.30	41.0	21.0	21.0	6	4	3.86	10.03	9.63	9.57
Max (SCC)	804	253	253	153	110	110	14	8	8	115	111	109

N = number of selected classes; BR = Before Refactoring; AR = After Refactoring

5. Evaluation and Discussion

We have used 15 software applications to answer the research questions. These are: commApp, Azureus (Vuze), Jstock, VidCoder, Hibernate, Openproj, Jxplorer, Megamek, Weka, SomToolBox, GanttProject, Squirrel-sql, OpenRocket, ermaster, and Logisim. Apart from commApp (industrial application) and VidCoder developed in C#, the criteria for selecting the remaining applications on SourceForge is that the application must be driven through a user interface, it must be popular (four to five stars rating), must have at least 500 downloads per week and must be developed in Java. For properties of selected applications, see the link in Section IV.

A. RQ1: Is the system restructuring better when IRCRSS exists?

Case Study: Azureus 2.3.0.2

Our goal here is to find out whether using the CB-DSS with the IRCRSS metric would improve the result when compared to the manually selected candidates used for refactoring in [20]. We found that Azureus 2.3.0.2 fits the version analyzed in [20] because it has approximately the same value of CRSS (just a difference of 4 which could be due to exclusion or inclusion of test classes). Both the versions before and after this version have a wide CRSS range gap to the reported value.

1) Approach:

We have simulated the refactoring with the manually selected candidates by [20]. In this simulation, we turned off the adaptive selection algorithm and allow the CB-DSS to iterate through the selected candidates as presented by the authors. In the second simulation, we turned on the adaptive selection algorithm and allow the CB-DSS to automatically select candidates for refactoring. This is determined by a combination of tuning and sorting parameters. We performed different simulations by varying the percentiles (topKs) of the sorting parameters (SCC, CRSS and FAN-IN) and the value of the IRCRSS metric.

2) Results and discussion

Table III lists the candidates selected by our approach vs. the ones reported in [20]. The last candidate (*org.gudy.azureus2.pluginsimpl.local.torrent.TorrentImpl*) is a multi-ton. Fig. 5 shows that at the 7th refactoring, the selection made by the CB-DSS has better results than the 10th refactoring with the manual mode. It indicates that using a CB-DSS with IRCRSS metric can significantly improve the refactoring results. The result from Table IV shows there is more reduction in the CRSS and SCC values when IRCRSS is used and optimal values for CRSS, FANIN, and SCC are chosen. As listed in Table IV, the max SCC after refactoring drops from 804 to 253 while for the selection by [20], it drops to 333. Furthermore, The reported number from [20] for frequency of

classes with CRSS of 1000 or more is 400, which is modestly comparable to the simulated number of 427. Using automatic selection produced a better result of 348 classes (18.5% reduction).

In all cases, the results from using the selection parameters from the CB-DSS produced better results but notably, with the IRCRSS metric.

We performed a statistical test using Wilcoxon rank sum test [36] to determine whether the fitness values (CRSS) by using the IRCRSS metric is statistically and significantly lower than the fitness values without(i.e. H_0 = *The fitness of refactoring with IRCRSS is significantly higher than the fitness without IRCRSS*).

TABLE III. CB-DSS VS MANUAL CANDIDATES SELECTION

Order	Candidates by [20]	Candidates (CB-DSS)
1	org.gudy.azureus2.core3.logging.LGLogger	org.gudy.azureus2.core3.util.Debug
2	org.gudy.azureus2.core3.config.COConfigurationManager	org.gudy.azureus2.core3.config.COConfigurationManager
3	org.gudy.azureus2.core3.util.Debug	org.gudy.azureus2.core3.internat.MessageText
4	org.gudy.azureus2.core3.util.FileUtil	org.gudy.azureus2.ui.swt.Messages
5	org.gudy.azureus2.platform.PlatformManager	org.gudy.azureus2.core3.util.FileUtil
6	org.gudy.azureus2.core3.internat.MessageText	org.gudy.azureus2.ui.swt.Utils
7	org.gudy.azureus2.core3.util.TorrentUtils	org.gudy.azureus2.core3.util.TorrentUtils
8	org.gudy.azureus2.core3.internat.LocaleUtil	org.gudy.azureus2.ui.swt.components.shell.ShellFactory
9	org.gudy.azureus2.core3.util.DisplayFormatters	org.gudy.azureus2.ui.swt.mainwindow.Colors
10	org.gudy.azureus2.core3.util.DirectByteBufferPool	org.gudy.azureus2.pluginsimpl.local.torrent.TorrentImpl

TABLE IV. FITNESS VALUES FOR AZAREUS 2.3.0.2 USING THE CB-DSS

Fitness	Before refactoring	After Refactoring (N=10)			p-value (α=0.05)
		Selection by [20]	IRCRSS=False	IRCRSS=True	N=15 (IRCRSS=True vs. IRCRSS=False)
Mean (CRSS)	703.67	341.11	326.51	295.46	0.011
Std. Dev (CRSS)	684.88	566.55	554.58	525.28	
Max (SCC)	804	333	306	253	

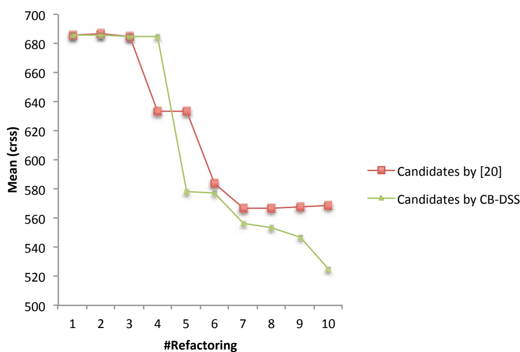


Fig. 5. Simulation steps and corresponding fitness values

We performed 15 refactoring (see column 6 of Table IV). The mean CRSS from both options are recorded separately and are then tested for significant difference. The result of the test is statistically significant at alpha = 0.05 (with **p-value=0.011**). We thus reject the null hypothesis and conclude that applying the IRCRSS metric gives a significantly lower (better) result for this application.

3) Manual refactoring:

The CB-DSS proposed nine singleton classes and one multi-ton class as candidates. We then manually refactored the code by using the actions reported from the system. The entire refactoring was completed in approximately 24 hours and was done by one of the authors. There are instances where an interface already exists for the candidate. This case occurs for the non-singleton class, ***TorrentImpl*** class that implements ***Torrent*** (an interface). To refactor the proposed edges, some new methods must be declared in the old interface or in the proposed interface. A standard maintenance practice is to introduce a new interface that extends ***Torrent*** and add those methods in the new interface. This is a kind of interface upgrade. By doing so, we maintain a downward compatibility of the old interface (***Torrent***) and do not break the code. Otherwise, declaring new methods in the interface, would

force other children of ***Torrent*** to implement them.

The new Java SDK[59] version 8, however makes it possible to declare such new methods with empty or default implementations in **Torrent**. This produces the same results, as the old classes are not forced to implement the new methods. This is significant because it simplifies maintenance and refactoring activities. Rather than defining a new interface because of additional functions, it is now possible to define new contracts as default methods and without the burden of forceful implementation of new methods by all children. Arguably, this feature can also be a shortfall. First, it would be hard for children to be aware of declared methods in the interface because it is not required anymore. Second, in terms of maintenance and upgrade, it would be hard to keep track of changes (extensions, etc.) that have been made as the system evolves.

B. RQ2: Does tuning with IRCRSS always improve the system's structure?

To answer this question, we simulate refactoring on the fifteen applications. The results in Table V demonstrate that it is possible to take advantage of the IRCRSS metric in several applications. There are cases such as Logisim, JXplorer, Azureus, OpenRocket, and Hibernate, where relatively high SCC reductions exist when the measurement from IRCRSS metric is applied. However, in a few applications (e.g. Megamek, VidCoder and Squirrel-sql), there is no difference in the results with the IRCRSS metric. In total, there are improvements in the fitness values of 12 out of the 15 applications. We can conclude that the IRCRSS metric can improve the code structure in the majority of the cases. The metric (IRCRSS) provides tangible and useful information to clients/services that are being coupled. The IRCRSS value is zero or nearly zero when the published types of a class are tightly coupled with other classes (in most cases, concrete classes and not interfaces). This has implications for maintenance and testing. A class/service that is heavily reused and is tightly coupled in its published members would be difficult to reuse, maintain and test.

C. RQ3: Can the use of IRCRSS metric reduce restructuring and refactoring effort?

During code restructuring, new edges/relationships are created and some edges/relationships are removed. To answer this question, we have categorized the restructuring effort by the number of edges (source → target) that the CB-DSS proposes for refactoring. In addition, we complement it by inspecting the reported number of edges created by the system after refactoring. We compare the fitness when IRCRSS metric is applied to the fitness when it is not. As shown in the column "%Reduction-Refactoring Edges" in Table V, when IRCRSS metric is used, the CB-DSS is able to reduce the refactoring efforts. In six cases, there are significant reductions in the refactoring edges. For instance in Logisim and Hibernate; the refactoring edges are reduced by 63.2% and 66% respectively when IRCRSS metric is used. This is noteworthy in the sense that refactoring fewer edges would translate to a reduction in refactoring efforts.

We complement this result by reporting the rate of reduction (%Edge Reduction) in the class edges created in the applications when IRCRSS is applied. As shown in this table, the total number of edges after refactoring with IRCRSS reduced reasonably in some applications (e.g. Logisim, OpenRocket, Hibernate and JStock). In other words, fewer relationships are created in the code when IRCRSS is applied. This is related to the explanation given above (**RQ2**). When the signatures of published methods of a class are tightly coupled to other concrete classes, it would result in more relationships/edges being created during decoupling/restructuring.

It is positive to have fewer classes and relationships/edges during restructuring. The fewer the number of edges that exist in an application, the better it would be to reason about the coupling situation in the application.

A. Qualitative Evaluation

We have carried out an interview with the software maintenance engineer of our industrial

[59] http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

partner to determine the usefulness and usability of the CB-DSS. We drafted questions that covered four areas namely; user experience, compatibility, impact and functionalities. For user experience, we asked whether the CB-DSS is easy to use, whether it is easy to learn quickly, whether the system's functionalities are clear and understandable and whether it will be used in the future. Under compatibility we asked whether the CB-DSS fits well with the work practices. In terms of impact, we asked whether the approach is useful for refactoring complex structural part of the code, whether the approach is able to identify good candidates for refactoring and whether the actual code structure improved after refactoring.

Summary of Respondent's views

User experience: The respondent views are that the tool is easy to use and can be learned quickly and individually with a proper help file. It does not take time to learn how to use it. The functionalities are clear and understandable and can motivate maintenance practices in the company. The only challenging part is how to choose the parameters for the algorithm.

TABLE V. REFACTORING BY TUNNING WITH AND WITHOUT IRCRSS METRIC

System	Before Refactoring			After Refactoring							
				IRCRSS=False, N=10			*IRCRSS =True, N=10*				
	Mean (CRSS)	*Mean (SCC)*	*Max (SCC)*	*Mean (CRSS)*	*Mean (SCC)*	*Max (SCC)*	*Mean (CRSS)*	*Mean (SCC)*	*Max (SCC)*	*%Reduction-Refactoring Edges*	*%Edge (Reduction)*
Logisim	363.31	16.15	437	341.92	14.22	**406**	245.89	9.6	**272**	**63.2**	**40.1**
JXplorer	48.69	11	35	51.14	11.4	**37**	43.24	9.2	**29**	**18.45**	**4.0**
Azureus	703.67	22.82	804	326.51	8.51	**306**	295.46	7.37	**253**	**3.12**	**4.1**
OpenRocket	206.35	10.64	213	203.64	10.03	**190**	200.96	9.65	**165**	**36.2**	**32.7**
Hibernate	756.87	19.64	1442	756.9	19.62	**1439**	701.35	18.39	**1254**	**66.0**	**57.7**
Somtoolbox	252.1	21.23	229	206.26	14.71	**185**	201.85	15.06	**178**	**48.1**	**2.9**
ermaster	577.78	580	580	424.63	89.8	**411**	405.3	85.2	**388**	-1.7	-3.4
JStock	154.64	41	153	108.7	21	**110**	106.26	20.5	**107**	**22.1**	**13.9**
commApp	115.06	10.03	115	112.78	8.48	**83**	112.07	8.45	**82**	**5.2**	**4.6**
GanttProject	171.82	12.85	247	169.56	12.28	**210**	160.76	12.24	**208**	-10.9	**5.2**
Weka	255.31	11.76	232	54.85	3.63	**13**	54.76	3.61	**12**	**1.63**	**0.4**
Openproj	274.49	12.57	269	247.71	11.62	**177**	247.32	11.61	**176**	**1.53**	**0.9**
Megamek	1450.96	246.5	1464	1146.66	163.88	1205	1146.66	163.88	1205	0.0	0.0
VidCoder	17.22	6	14	5.60	2.17	3	5.51	2.2	3	**5.2**	**5.1**
Squirrel-sql	402.53	29.14	457	359.06	22.83	366	359.06	22.83	366	**0.3**	**0.4**

Compatibility: The tool will fit maintenance work practices and using the tool regularly can help developers to have a picture of the code's structure and keep an eye on maintainability.

Impact: Respondent states that at present, large parts of their code are not that maintainable, looking at the code you can spot some areas that should be changed (e.g. excessively large and coupled classes), some bad coding practices and so on. The tool will stimulate actions to correct some of these problems. In addition, code reuse would be easier.

Functionalities: The approach is able to identify good targets for refactoring and the code structure improved after refactoring. The respondent prefers a simulation tool rather than an automated tool for this large scale restructuring. This agrees with the feedback we got from three other developers in the same company during our presentation sessions.

6. Threats and Limitations

The CB-DSS is implemented on top of the Jepend tool that uses BCEL to collect class dependency data. Java's specification uses type erasure, therefore, information about type parameters of generic types are not available in the Java bytecode. In addition, we cannot identify dependencies created by the use of reflection. This is a common limitation of static analysis. The accuracy of the CB-DSS depends on the accuracy of the parser tool that generate the dependency data.

The refactoring result by CB-DSS is sometimes an approximation due to the use of a new and generic interface. A candidate may have an existing interface that only needs to be upgraded during refactoring. The CB-DSS does not take this into consideration during its computation and simulation.

7. Conclusion

We have implemented a new metric, IRCRSS and a cycle breaking decision support system (CB-DSS) to resolve class dependency cycles and improve the overall code structure. The evaluation of the CB-DSS proved that it is useful and implementable in many cases in real life systems.

Our contributions in this work are therefore as follows:

1. Significant improvement on the strategy employed in [20] by introducing a new metric IRCRSS, to identify CRSS reduction between an interface and its implementation. In this way, it is possible to improve the structural quality of the code and reduce the refactoring efforts
2. A cycle breaking system that proposes executable refactoring actions. These actions are fine-tuned for each proposed edge (source → target), with details such as the strategy and action to break the edge, and the actual code location (method or field) where the strategy should be applied in the source class
3. We demonstrate the validity of the CB-DSS by the manual refactoring on industrial and open source systems.

References

[1] Bass, L., P. Clements, and R. Kazman, Software Architecture in Practice. 2003: Addison-Wesley Longman Publishing Co., Inc. 528.

[2] Lakos, J., Large-scale C++ software design. 1996, Redwood City, CA: Addison-Wesley Longman.

[3] Martin, R.C., Design principles and design patterns. Object Mentor, 2000. 1: p. 34.

[4] Parnas, D.L., Designing Software for Ease of Extension and Contraction. IEEE Transactions on Software Engineering, 1979. SE-5(2): p. 128-138.

[5] Fowler, M., Reducing Coupling. IEEE Softw., 2001. 18(4): p. 102-104.

[6] Martin, R. Design Principles and Design Patterns. 2000; Available from: http://www.objectmentor.com.

[7] Briand, L.C., Y. Labiche, and W. Yihong. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. in Proc. 12th International Symposium on Software Reliability Engineering, (ISSRE 2001) 2001.

[8] Jungmayr, S. Identifying test-critical dependencies. in Software Maintenance. 2002.

[9] Kung, D., et al., On Regression Testing of Object-Oriented Programs. Journal of Systems Software, 1996. 32(1): p. 21-40.

[10] Oyetoyan, T.D., D.S. Cruzes, and R. Conradi. Criticality of Defects in Cyclic Dependent Components. in 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM). 2013. Eindhoven, Netherlands.

[11] Oyetoyan, T.D., D.S. Cruzes, and R. Conradi, A study of cyclic dependencies on defect profile of software components. Journal of Systems and Software, 2013. 86(12): p. 3162-3182.

[12] Zimmermann, T. and N. Nagappan, Predicting subsystem failures using dependency graph complexities. ISSRE 2007: 18th IEEE International Symposium on Software Reliability Engineering, Proceedings, 2007: p. 227-236.

[13] Johnson, R., et al., Professional Java Development with the Spring Framework. 2009: John Wiley & Sons.

[14] Walls, C., Modular java: Creating flexible applications with OSGi and spring. 2009: pragmatic bookshelf.

[15] Dietrich, J., et al., Barriers to Modularity-An Empirical Study to Assess the Potential for Modularisation of Java Programs, in Research into Practice–Reality and Gaps. 2010, Springer. p. 135-150.

[16] Melton, H. and E. Tempero, An empirical study of cycles among classes in Java. Empirical Software Engineering, 2007. 12(4): p. 389-415.

[17] Brown, N., et al. Managing technical debt in software-reliant systems. in Proceedings of the FSE/SDP workshop on Future of software engineering research. 2010. ACM.

[18] Eades, P., X. Lin, and W.F. Smyth, A fast and effective heuristic for the feedback arc set problem. Inf. Process. Lett., 1993. 47(6): p. 319-323.

[19] Melton, H. and E. Tempero, JooJ: real-time support for avoiding cyclic dependencies. Proceedings of the thirtieth Australasian conference on Computer Science, 2007. 62: p. 87-95.

[20] Melton, H. and E. Tempero, The CRSS metric for package design quality, in Proceedings of the thirtieth Australasian conference on Computer science - Volume 62 2007, Australian Computer Society, Inc.: Ballarat, Victoria, Australia. p. 201-210.

[21] Peffers, K., et al., A Design Science Research Methodology for Information Systems Research. J. Manage. Inf. Syst., 2007. 24(3): p. 45-77.

[22] Martin, R.C., Granularity, C++ Report, 1996. p. 57-62.

[23] Gamma, E., et al., Design patterns: elements of reusable object-oriented software. 1995: Addison-Wesley Longman Publishing Co., Inc. 395.

[24] Cormen, T.H., et al., Introduction to algorithms. 2nd ed. 2001, Cambridge, Mass.: MIT Press. xxi, 1180.

[25] Fowler, M., Refactoring: improving the design of existing code. 1999: Pearson Education India.

[26] Gobner, J., P. Mayer, and F. Steimann, Interface utilization in the Java Development Kit, in Proceedings of the 2004 ACM symposium on Applied computing2004, ACM: Nicosia, Cyprus. p. 1310-1315.

[27] Steimann, F., W. Siberski, and T. Kuhne, Towards the systematic use of interfaces in JAVA programming, in Proceedings of the 2nd international conference on Principles and practice of programming in Java2003, Computer Science Press, Inc.: Kilkenny City, Ireland. p. 13-17.

[28] Fowler, M. Inversion of control containers and the dependency injection pattern. 2004.

[29] Shah, S.M.A., J. Dietrich, and C. Mccartin. On the Automation of Dependency-Breaking Refactorings in Java. in 29th IEEE International Conference on Software Maintenance (ICSM). 2013. Eindhoven, Netherlands.

[30] Dietrich, J., et al., On the existence of high-impact refactoring opportunities in programs, in Proceedings of the Thirty-fifth Australasian Computer Science Conference - Volume 122 2012, Australian Computer Society, Inc.: Melbourne, Australia. p. 37-48.

[31] Mens, T., G. Taentzer, and O. Runge, Analysing refactoring dependencies using graph transformation. Software & Systems Modeling, 2007. 6(3): p. 269-285.

[32] Van Der Straeten, R., V. Jonckers, and T. Mens, Supporting Model Refactorings Through Behaviour Inheritance Consistencies, in «UML» 2004 — The Unified Modeling Language. Modeling Languages and Applications, T. Baar, et al., Editors. 2004, Springer Berlin Heidelberg. p. 305-319.

[33] Laval, J. and S. Ducasse, Resolving cyclic dependencies between packages with enriched dependency structural matrix. Software-Practice & Experience, 2014. 44(2): p. 235-257.

[34] Melton, H. and E. Tempero. Identifying refactoring opportunities by identifying dependency cycles. in Proceedings of the 29th Australasian Computer Science Conference-Volume 48. 2006. Australian Computer Society, Inc.

[35] Hautus, E. Improving Java Software Through Package Structure Analysis. in The 6th IASTED International Conference Software Engineering and Applications. 2002. Cambridge, MA, USA.

[36] Fenton, N.E. and S.L. Pfleeger, Software Metrics: A Rigorous and Practical Approach. 1998: PWS Publishing Co. 656

Appendix B: Secondary papers

Appendix B contains three other papers produced during this PhD research. P8 is contained in the discussion part of the study. P9 and P10 contains discussions that are related to Smart Grid systems

P8. Oyetoyan, T.D., Cruzes, D.S., Conradi, R., 2012. Can Reused Components Provide Lead to Future Defective Components in Smart Grid Applications?, Parallel and Distributed Computing and Systems : Software Engineering and Applications (PDCS 2012). ACTA Press

P9. Oyetoyan, T.D., Conradi, R., Sand, K., 2012. Initial survey of Smart Grid activities in the Norwegian energy sector - use cases, industrial challenges and implications for research, ICSE 2012 International Workshop on Software Engineering for the Smart Grid (SE4SG), Zurich, Switzerland, pp. 34-37.

P10. Oyetoyan, T.D., Conradi, R., Cruzes, D.S., 2011. Open Source Software for the Smartgrid: Challenges for Software Safety and Evolution. NIK: Norsk Informatikkonferanse 2011, Tromsø¸, Norway, pp. 239-243

P8: Can Reused Components Provide Lead To Future Defective Components In Smart Grid Applications?

Published: In Proc. Parallel and Distributed Computing and Systems: Software Engineering and Applications (PDCS 2012). ACTA Press

Tosin D. Oyetoyan
Computer and Information Science Department, NTNU, Trondheim, Norway
tosindo@idi.ntnu.no

Daniela S. Cruzes
Computer and Information Science Department, NTNU, Trondheim, Norway
dcruzes@idi.ntnu.no

Reidar Conradi
Computer and Information Science Department, NTNU, Trondheim, Norway
reidar.conradi@idi.ntnu.no

ABSTRACT

Smart Grid systems are kind of System of Systems with distributed and highly heterogeneous software connected to provide various services. Early knowledge of defect prone parts is useful for improving the safety and maintenance of these kinds of systems. We report the results of an empirical study of using reused components to predict future defective components in a type of open source Smart Grid application (transmission and operation domain of Smart Grid). Our results showed that reused components of this Smart Grid application are strong predictors of future defective components. The model's best predictors gave an average recall of 0.92 (average precision of 0.406) when tested across three future releases. Which implied that 92% of predicted defective components in the next release turned out to be defective. This model can be employed to tailor quality assurance (QA) efforts in a way that blind spots are avoided in such critical system and QA effectiveness significantly improved.

KEY WORDS

Component; Smart Grid; prediction model; empirical study; Component defect-proneness; Import types; System of Systems.

P9: Initial Survey of Smartgrid Activities in the Norwegian Energy Sector: Use Cases, Industrial Challenges and Implications for Research

(Position paper)

Published: ICSE 2012 International Workshop on Software Engineering for the Smart Grid (SE4SG), Zurich, Switzerland, pp. 34-37

Tosin Daniel Oyetoyan[1], Reidar Conradi[1]
Norwegian University of Science and Technology (NTNU), Trondheim, Norway
{tosindo,reidar.conradi}@idi.ntnu.no

Kjell Sand[1,2]
SINTEF Trondheim, Norway
kjell.sand@sintef.no

Abstract—Motivation: Understanding user requirements and technological challenges for smartgrid is important to deliver competitive and visionary products and services, and thus to shape the direction of research and development. Since smartgrid is still in the formation stage with many stakeholders, we should quickly develop consensual and pragmatic international standards and strategies. **Goals:** To assess the feasibility of proposed smartgrid requirements, formulated as 16 generic use-cases by an EU working group, and to identify attitudes, products, services and future technologies. Subsequently, we want to provide information on identified gaps between technologies, functionalities and stakeholders` views, and future direction. **Approach:** We have designed and carried out an initial industrial survey in Norway on how generic use-cases for smartgrid activities are interpreted by 6 representative stakeholders in the Norwegian energy sector. To achieve this goal, we designed a survey with metrics built on and around these use-cases. **Results:** The users' work experience and views on the functionality expressed in the use-cases revealed a gap in focus and culture. Also, there was no agreement on what the term "smartgrid" stood for. In addition, the relevance of smartgrid functionalities is shown to vary over time and with different stakeholders. **Discussion:** The pre-study results indicated that there is potential for using information from future data collected from over 270 actors to bridge gaps and focus on smartgrid research and development.

Keywords: smartgrid usage; stakeholders, requirements; use- case; pre-study; survey; Norwegian energy industry.

P10: Open Source Software for the Smartgrid: Challenges for Software Safety and Evolution

Published: Norsk Informatikkonferanse 2011, Tromsø¸, Norway, pp. 239-243

Tosin Daniel Oyetoyan, Reidar Conradi, Daniela Soares Cruzes

Department of Computer and Information Science, NTNU, Trondheim, Norway.

Abstract

The growing Smartgrid behind today's electricity supply introduces many challenges. One aspect is the management of various software that drive these new systems at different domains (generation, transmission, distribution and consumption) and nodes of the Smartgrid network. Managing such concerted, distributed, evolving and heterogenous System of Systems requires a methodical approach to support more standardized processes and products to reach the Smartgrid vision. This paper presents a recent research project focusing on assessing the adoption of OSS for the Smartgrid by investigating its safety and evolution criteria.

NTNU

GODKJENNING
FRA
OPPNEVNT SENSURKOMITE
av avhandling for graden

PERSONALOPPLYSNINGER

Kandidat: **Tosin Daniel Oyetoyan**

Institutt: **Department of Computer andInformation Science**

AVHANDLING

Tittel på avhandlingen:

Dependency Cycles in Software Systems: Quality Issues and Opportunities for Refactoring

SENSURKOMITEENS VURDERING

Avhandlingen er bedømt og godkjent for graden philosophiae doctor ☐
Avhandlingen er bedømt og godkjent for graden doctor philosophiae ☐

Trondheim, den

....	....
navn	underskrift
....	....
navn	underskrift
....	....
navn	underskrift
....	....
navn	underskrift