2015 IEEE International Conference on Software Quality, Reliability and Security

Are Anti-patterns Coupled? An Empirical Study

Wanwangying Ma, Lin Chen*, Yuming Zhou, Baowen Xu
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
xiaobena@live.cn, {lchen, zhouyuming, bwxu}@nju.edu.cn

Xiaoyu Zhou
School of Computer Science and Engineering
Southeast University
Nanjing, China
zhouxy@seu.edu.cn

***Abstract*—The interactions between anti-patterns are claimed to affect maintenance. However, little work has been conducted to examine how anti-patterns interact. In this paper, we aim to investigate which pairs of anti-patterns tend to be coupled, i.e., interact with each other. We employ Fisher's exact test and Wilcoxon rank-sum test to identify coupled anti-patterns in the same class and coupled classes. Analyzing the relationships amongst 10 kinds of anti-patterns in five open-source projects, our results show that 1) several kinds of anti-patterns tend to be coupled, but some are conflicting; 2) the effect of anti-patterns on their dependent or co-changed ones are significant but small; 3) in ArgoUML, Xalan and Xerces-J, the classes infected with dependent anti-patterns are mostly (69.9% ~ 100%) modified in maintenance activities. Our findings offer empirical evidences for the existence of anti-pattern interactions, which provides valuable implications for practitioners and researchers.**

Keywords-anti-pattern; interaction; coupled; inter-pattern; maintenance

I. INTRODUCTION

Anti-patterns are common responses to design problems which are claimed in the literature [23] to be ineffective and hinder software evolution. Previous studies have investigated their effects on different aspects related with maintainability, such as code comprehension [1-3], change-proneness [4-8] and fault-proneness [8-12]. Surprisingly, some analyses [1, 2, 3, 9] show that individual anti-patterns do not always cause notably detrimental effects as supposed. Hall et al. [9] pointed out that some anti-patterns have a significant but small impact on faults. Abbes et al. [1] concluded that *Blob* or *Spaghetti Code* alone does not significantly decrease developers' performance.

Recently, researchers have paid attention to the impact of multiple anti-patterns in the same artifact. They concluded that some inter-pattern relations are associated with problems during software maintenance [13], and intensify the negative impact of individual anti-patterns [1]. In addition, anti-pattern interactions are not manifested only within one artifact. Correlated anti-patterns distributed across coupled artifacts have comparable negative effects as when they are co-located [13]. These studies imply that further analyses on interacted anti-patterns will lead to a more complete understanding of the relationship between anti-patterns and software maintainability. However, little work provides empirical evidences for anti-patterns interaction, into which we are willing to make a preliminary investigation.

Unlike Yamashita et al. [13] who concentrate on the impact of inter-smell relations on software maintenance, we aim to quantitatively provide actual examples for anti-pattern interactions. The proved inter-pattern relations will carry complementary information which is valuable for anti-pattern detection, as well as offer the basic premise for further investigation into the effects of coupled anti-patterns.

Specifically, the goal of this study is to explore which pairs of anti-patterns interact with each other, that is, tend to be coupled. We seek to answer the following research questions:

- RQ1 (co-located anti-patterns): which pairs of anti-patterns tend to be coupled in the same class?
- RQ2 (dependent anti-patterns): which pairs of anti-patterns tend to be coupled in inter-dependent classes?
- RQ3 (co-changed anti-patterns): which pairs of anti-patterns tend to be coupled in co-changed classes?

In RQ1, we aim to find out the pairs of anti-patterns which are more likely to be present within the same class. In RQ2 and RQ3, we extend the notion of coupled artifacts [13, 18, 31]: 1) the artifacts with static dependency relationships, including data dependency and function call dependency (inter-dependent classes); and 2) the artifacts with temporal associations, i.e., those co-changed in a maintenance task by programmers (co-changed classes). We intend to investigate whether the occurrence of one kind of anti-pattern will positively affect the occurrence of another kind through static dependencies or temporal associations.

In order to answer the three research questions, we conduct our study on five open-source Java projects and perform a quantitative analysis on the coupling relations of 10 kinds of anti-patterns commonly inspected in literature. Firstly, we identify anti-patterns for each project, as well as collect the static and temporal relationships between classes. Then, we apply Fisher's exact test and odds ratio on each pair of anti-patterns to evaluate whether they are more likely to appear together in a class. Finally, we employ Wilcoxon rank-sum test and Cliff's δ effect size to examine the impact of an anti-pattern on the presence of other anti-patterns in its dependent and co-changed classes. Our results show that 1) several pairs of anti-patterns do interact with each other in the same class or coupled classes; 2) most interactions through static or temporal dependencies are weak, but a large proportion of the classes containing dependent anti-patterns

*corresponding author

978-1-4673-7989-2/15 $31.00 © 2015 IEEE
DOI 10.1109/QRS.2015.43

are related to maintenance activities; 3) some anti-patterns tend to be mutual exclusive in a class.

The rest of the paper is organized as follows: Section II provides the related work. Our study design is described in Section III. In Section IV, we analyze and discuss the results of case study and give answers to the research questions. Section V discusses the threats to validity of our work. Section VI contains the conclusions and future work.

II. Related Work

This section briefly introduces the previous studies related to the coupled anti-patterns analysis.

Abbes et al. [1] investigated the impact of *Blob* and *Spaghetti Code* on the system understandability during maintenance tasks. They conducted their study on six open-source projects and collected data on the performance of 24 students and professionals on basic tasks related to program comprehension. They then used Wilcoxon rank-sum test and *t*-test to check whether there were statistically significant differences between subject's efforts, times, and percentages of correct answers on systems with and without anti-patterns. Their results showed that the occurrence of one kind of anti-pattern did not significantly reduce the system understandability while the combination of the two significantly made the system harder to comprehend. Our work is partially motivated by their study which emphasized the impact of related anti-patterns on maintainability. We aim to find out which anti-patterns are interacted with each other, thus offer the basic premise for further investigation of their effects.

Yamashita and Moonen [13] empirically explored the interactions amongst 12 code smells and analyzed how they were related to maintenance problems. Their study was based on four functionally equivalent Java systems originally implemented by different companies. Six professional developers were hired for a period of four weeks to perform three maintenance tasks. The authors recorded the specific problems that the developers encountered and the associated artifacts, as well as detected the code smells in pre-maintenance versions of the systems. They then employed Principal Component Analysis (PCA) to identify patterns of co-located code smells and revealed how they interacted to affect maintainability. Additionally, they found that code smells occurred across coupled artifacts had comparable negative effects as in the same artifact. Our work is different from theirs. We concentrate on the interactions between two kinds of anti-patterns and intend to identify coupled anti-patterns by statistically investigating whether the existence of one kind of anti-pattern significantly affects the other.

Pietrzak and Walter [14] defined six distinct relations between code smells: plain support, mutual support, rejection, aggregate support, transitive support, and inclusion. They were inspired from the observation that smells are not independent, separated phenomena, as well as got the idea from a superficial analysis of Fowler's bad smells descriptions which revealed that most of them are related to each other. Thus, they argued that the presence or absence of a code smell often carried knowledge about other smells and could be exploited to leverage detection. Our work is different from theirs, since we intend to find actually related anti-patterns using statistical analysis, which can provide concrete empirical cases for some of the inter-smell relations conjectured by them.

III. Study Design

A. Studied Projects

We conduct our study on five Java projects: ArgoUML[1], Cocoon[2], Log4j[3], Xalan[4], and Xerces-J[5]. They are open-source software systems of different application domains. ArgoUML is the leading UML modeling tool and includes support for all standard UML diagrams. The other four projects are all developed under Apache Software Foundation. Cocoon is a framework built around the concepts of separation of concerns and component-based development. Log4j is a logging library for Java. Xalan is an Extensible Stylesheet Language Transformation (XSLT) processor for transforming XML documents into HTML, text, or other XML document types. Xerces-J is a library for parsing, validating, and manipulating XML documents. These projects have hundreds to thousands of classes, span several years and versions, and have been used in previous work. TABLE. I lists the investigated versions of these projects and their sizes in terms of the number of classes and KLOC.

TABLE I. Some Properties of The Investigated Projects

Project	Version	#Classes	KLOC
ArgoUML	0.24	1430	155.5
Cocoon	3.0.0	304	19.5
Log4j	2.1	894	69.2
Xalan	2.6	822	155.1
Xerces-J	2.10	557	112.1

B. Data Collection

1) Anti-patterns

We use the DEtection for CORrection approach DECOR [15], which is widely used in the literature, to detect anti-patterns. DECOR is a metric-based detection tool which has a flexible rule engine and a unique semantic rule detector. TABLE II shows the 10 kinds of investigated anti-patterns and their descriptions.

TABLE III summarizes the numbers of each kind of anti-pattern detected in the studied projects. It can be seen that *Complex Class*, *Long Parameter List,* and *Lazy Class* appear in all these projects, and the former two anti-patterns account for the largest percentage of the detected anti-patterns for all except Xerces-J. *Blob*, *Message Chains*, and *Swiss Army Knife* are found only in Xerces-J.

[1] http://argouml.tigris.org/
[2] http://cocoon.apache.org
[3] http://logging.apache.org/log4j/2.x/
[4] http://xalan.apache.org/
[5] http://xerces.apache.org/#xerces2-j

TABLE II. INVESTIGATED ANTI-PATTERNS AND THEIR DESCRIPTIONS

Anti-pattern	Description
Anti-singleton (AS)	A class that provides mutable class variables, which consequently could be used as global variables.
Blob(Bb)	A class that takes too many responsibilities relative to the classes with which it is coupled.
Class Data Should Be Private (CP)	A class that exposes its fields, thus violating the principle of encapsulation.
Complex Class (CC)	A class that has (at least) one large and complex method, in terms of cyclomatic complexity and LOCs.
Lazy Class (YC)	A class that does too little.
Long Parameter List (LP)	A class that has (at least) one method with a too long list of parameters in comparison to the average number of parameters per methods in the system.
Message Chains (MC)	A class that uses a long chain of method invocations to realize (at least) one of its functionality.
Refused Parent Bequest (RB)	A class inheriting functionalities that it never uses.
Speculative Generality (SG)	A class that is defined as abstract but that has very few children, which do not make use of its methods
Swiss Army Knife (SK)	A class that has a large number of interfaces to provide for all possible uses of it.

TABLE III. THE NUMBERS OF ANTI-PATTERNS IN STUDIED PROJECTS

	ArgoUML	Cocoon	Log4j	Xalan	Xerces-J
AS	1	0	50	29	8
BB	0	0	0	0	71
CP	0	1	58	38	31
CC	170	19	47	111	47
YC	46	10	10	77	24
LP	284	49	41	105	130
MC	0	0	0	0	29
RB	4	0	0	11	222
SG	0	0	1	3	4
SK	0	0	0	0	11
Sum	411	69	163	301	380

2) Static Dependencies

The static dependency in this study is an undirected relationship between two pieces of code [29]. We consider two kinds of the most commonly investigated static dependencies [29, 30]: the data dependency between the use of a variable and its definition, and the call dependency between the sites where a function is called and its declaration. *Class A* and *Class B* are inter-dependent by data, if *Class A* uses the data defined in *Class B*, or vice versa. *Class A* and *Class B* are inter-dependent by function call, if *Class A* calls a function or method declared in *Class B*, or vice versa.

We use the program-understanding tool *Understand*[6] to collect the data and call dependencies between classes. For each of the investigated projects, we first generate an *Understand* database which stores information about entities (such as methods and classes) and references (such as function call and variable references). Then a Perl script is developed to track the dependency information between classes.

3) Temporal Associations

Two classes have a temporal association if they have been modified during the same maintenance task. Software engineers commit the modifications of a project's files to the version control systems (GIT[7] or SVN[8]), and a maintenance task may correspond to several change logs. Therefore, to identify co-changed classes, we first associate the change logs with different maintenance tasks based on the time that the logs were committed.

Hatton [16] presented an empirical study to estimate the time for a particular maintenance request (or change request, CR). The author showed that the average duration of a CR is 5.17 hours and the largest duration is less than 40 hours. Thus, we assume that a set of subsequent change logs are correlated with a maintenance task, if their largest duration is less than 40 hours, and any duration of two subsequent logs is less than 5.17 hours. For each studied project, we parse the logs committed between the release dates of the investigated version and its next version to assign them into several maintenance tasks as above. The classes modified during the same maintenance task are co-changed and have temporal associations.

C. Analysis Methods

The goal of this study is to find which two kinds of anti-patterns are coupled, i.e. interact with each other. For simplicity, we distinguish the two anti-patterns of each compared pair as AP1 and AP2. In addition, we call the classes infected with an AP1 (AP2) *AP1-classes (AP2-classes)*.

1) Fisher's exact test and odds ratio for RQ1

RQ1 is to identify which two anti-patterns are co-located in the same class. For each pair of the 10 investigated anti-patterns, our experiment is pulled out in two steps: 1) using Fisher's exact test to examine whether the occurrence of AP1 and AP2 significantly relates to each other; and 2) using odds ratio to check whether they are positively related.

Fisher's exact test [19] is used to determine whether there is a significant association between two kinds of classifications, in our study: *AP1-classes* and *AP2-classes*. The null hypothesis for Fisher's exact test is that the proportions of *AP1-classes* are independent of *AP2-classes*.

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

[7] http://git-scm.com/

[8] http://subversion.apache.org/

We set the significance level α=0.05 to assess the p-values. A refusal to the null hypothesis means that the occurrences of AP1 and AP2 are correlated with each other.

The odds ratio (OR) [20] quantifies to what extent an event (*AP1-classes* in this study) is associated to another event (*AP2-classes*). Specifically, the OR represents the ratio of the odds that an *AP1-class* is also an *AP2-class*, to the odds that a *non- AP1-class* is an *AP2-class*. Thus, if the probability of an *AP2-class* with or without AP1 is p and q respectively, the odds ratio is calculated as follows:

$$OR = \frac{p/(1-p)}{q/(1-q)}$$

An OR greater than 1 indicates that the two events are more likely to co-occur, that is, the *AP1-classes* are more likely to be the *AP2-classes* and vice versa.

Noticeably, this co-located relation is symmetric. The presence of either of the two anti-patterns participating in the relation carries the information about the other. Then Fisher's exact test is performed on $10 \times (10\text{-}1)/2 = 45$ pairs of anti-patterns. We decide that a pair of anti-patterns are coupled in the same class, if and only if 1) the p-value obtained from the Fisher's exact test is lower than 0.05; and 2) the odds ratio is greater than 1.

2) Wilcoxon rank-sum test and Cliff's δ for RQ2/3

In RQ2 and RQ3, we aim to determine whether the presence of AP1 in a class will positively affect the presence of AP2 in its dependent or co-changed classes.

For simplicity, let x_c represent the presence of AP1 in *Class C*. *C* has n_c dependent (co-changed) classes, of which m_c classes contain AP2. Thus, the proportion of *AP2-classes* in *C*'s dependent (co-changed) classes is $p_c = m_c/n_c$.When x_c takes the value of 1 or 0, p_c falls into two categories: the proportion of *AP2-classes* in the coupled classes with *AP1-classes* (Cat.1) and the proportion of *AP2-classes* in the coupled classes with *non-AP1-classes* (Cat.2).

We then use one-sided Wilcoxon rank-sum test [21] to compare whether the values in Cat.1 are significatly higher than those in Cat.2, in order to decide whether AP1 increases the occurence of *AP2-classes* in its coupled classes. Wilcoxon rank-sum test is a non-parametric test which is used to determine whether two samples are drawn from the same or identical distributions. It does not require normal distributions of the populations. The null hypothesis in RQ2 (RQ3) is that the proportion of *AP2-classes* in the dependent (co-changed) classes with *AP1-classes* are not higher than the proportion of *AP2-classes* in the dependent (co-changed) classes with *non-AP1-classes*. The p-value lower than the significance level (α=0.05) means the refusal to the null hypothesis, thus indicating that AP2 is dependent on (co-changed with) AP1.

In addition, the dependent and co-changed relations, unlike co-located relation, are asymmetric. AP2 is dependent on (co-changed with) AP1, which does not mean that the opposite is true. Then Wilcoxon rank-sum test is carried out twice for each of the 45 pairs of anti-patterns. And, we decide that a pair of anti-patterns are coupled in inter-dependent (co-changed) classes, if either one of the compared pair is dependent on (co-changed with) the other.

Furthermore, we use Cliff's δ to quantify the magnitude of the impact of one anti-pattern on its coupled one. By convention, the magnitude of the impact is considered either trivial ($|\delta| < 0.147$), small (0.147-0.33), moderate (0.33-0.474), or large (> 0.474) [22].

IV. Results and Discussion

This section presents the results of our empirical study with respect to each research question and make some discussions.

A. Co-located Anti-patterns (RQ1)

TABLE IV shows the odds ratios of co-located anti-patterns for which the p-values obtained from the Fisher's exact tests are smaller than 0.05. Our results indicate that *Complex Class* and *Long Parameter List* are coupled in the same class for all investigated projects. *Anti-singleton* and *Class Data Should Be Private* interact with each other in Xalan and Log4j who have relatively large numbers of *AS*- and *CP-classes*. In Xalan, *Anti-singleton* tends to be present with *Complex Class* and *Long Parameter List*. In Xerces-J, the only project who participates *Blob*, *Message Chain* and *Swiss Army Knife*, the three kinds of anti-patterns are all likely to be co-located with *Complex Class* and *Long Parameter List*. In addition, *Lazy Class* and *Refused Parent Bequest* are also correlated in one class.

Moreover, the odds ratios calculated for *Anti-singleton* and *Class Data Should Be Private* are obviously larger than for others, which indicates that the interactions between the two anti-patterns are much stronger than others.

B. Dependent Anti-patterns (RQ2)

We distinguish two types of dependent anti-patterns: 1) data-dependent anti-patterns which are coupled through data dependencies; and 2) call-dependent anti-patterns which are coupled through function call dependencies. TABLE V and TABLE VI list the results of Wilcoxon rank-sum tests which compare between the proportions of the *AP2-classes* in the classes that are data/call-dependent with the *AP1-classes* and *non-AP1-classes*. Each row corresponds to AP1, while each column corresponds to AP2. *A*, *C*, *X*, *J* and *L* are short for ArgoUML, Cocoon, Xalan, Xerces-J and Log4j respectively. The presence of *A/C/X/J/L* in TABLE V and TABLE VI

TABLE IV. Odds Ratios of Significantly Related Anti-Patterns

	CC&LP	AS&CP	AS&CC	AS&LP	BB&CC	BB&LP	CC&MC	YC&RB	LP&MC	MC&SK
ArgoUML	5.99									
Cocoon	6.94									
Log4j	4.16	62.37								
Xalan	2.77	24.63	3.04	5.30						
Xerces-J	7.20				1.56	1.99	4.75	3.62	3.31	7.43

TABLE V. DATA-DEPENDENT ANTI-PATTERNS

	AS	BB	CP	CC	YC	LP	MC	RB	SG	SK
AS	---		X		X	X		X	L	
BB		---	J	J	J	J	J		J	J
CP	X	J	---	J	X	XJ	J			J
CC	JL	J	JL	---	AXJ	AJL	J	AJ	JL	J
YC			L		---					
LP	XL	J	XJL	ACXJL	ACJ	---	J	A	X	
MC		J	J	J	J	J	---			J
RB	X							---		
SG	L	J			J	X			---	
SK		J	J		J	J		J		---

TABLE VI. CALL-DEPENDENT ANTI-PATTERNS

	AS	BB	CP	CC	YC	LP	MC	RB	SG	SK
AS	---		XJ		X	X			L	
BB	J	---	J	J		J	J		J	J
CP			---		X	J				J
CC	AJL	J	JL	---	ACX	CJL	J		J	J
YC				X	---			A		
LP	AXL		XJL	CX	AC	---	J			
MC		J	J			J	---			J
RB					AJ			---		
SG	L					X			---	
SK			J						J	---

means the p-value of Wilcoxon rank-sum test calculated for the corresponding project is lower than 0.05, thus indicating that AP1 significantly and positively affects the occurrence of AP2 through data or call dependencies in this project.

From the results, we know that *Long Parameter List* and *Complex Class* are very influential for the presence of other anti-patterns in coupled classes both through data and call dependencies. For two to five projects, *Complex Class* significantly increases the possibilities that its dependent classes are affected with *Class Data Should Be Private*, *Speculative Generality*, *Lazy Class*, *Long Parameter List* and *Anti-singleton*. The coupled classes with *LP-classes* are more likely to contain *Anti-singleton*, *Class Data Should Be Private*, *Lazy Class* or *Complex Class*. In particular, *Complex Class* is dependent on *Long Parameter List i*n all five projects. Additionally, in Xerces-J, we find that *Blob* has a positive effect on all other anti-patterns by data or call dependencies.

C. Co-changed Anti-patterns (RQ3)

TABLE VII. CO-CHANGED ANTI-PATTERNS

	AS	BB	CP	CC	YC	LP	MC	RB	SG	SK
AS			LX	X		LX			X	
BB	J									
CP	L			L		L				
CC	AXJ	J	XJ		A	AXJ	J	J	XJ	J
YC				A				X		
LP	ALJ		LX	ALX	AX			X	XJ	
MC	J	J	J	J		J		J	J	J
RB										
SG	X			X		X				
SK	J	J		J		J	J		J	

TABLE VII shows the co-changed anti-patterns identified in five studied projects. Much similar to the results of RQ2, both *Long Parameter List* and *Complex Class* significantly affect the occurrence of *Class Data Should Be Private*, *Speculative Generality*, *Lazy Class*, and *Anti-singleton* in co-changed classes for at least two projects. At the same time *Long Parameter List* and *Complex Class* are positively correlated with each other in ArgoUML and Xerces-J. However, *Blob* does not show comparable influence on other anti-patterns in co-changed classes as in dependent classes for Xerces-J, because only the proportion of *Anti-singleton* in the co-changed classes of *Blob* increases. *Message Chains* takes its place and is coupled with all but *Lazy Class*. Another observation is that no coupled anti-patterns appear in Cocoon. Very few temporal associations lie between the classes in Cocoon, which makes the test data set too small to draw a statistical conclusion.

D. Discussion

1) The rationale behind co-located anti-patterns

This study reveals that several kinds of anti-patterns interact with each other in the same class. The studied software systems are developed for different applications in different environments by different developers using different coding styles, which lead to different kinds and numbers of anti-patterns across them. However, some co-located anti-patterns are consistent across projects, which indicates that the associations between them are independent of the software features, but determined by their definitions and manifestations.

Complex Class (CC) and Long Parameter List (LP). We find that 17.1% to 31.3% of the *LP-classes* participate *CC*. It is not surprising that a method with many parameters is likely to grow large and complex. On the other hand, for ArgoUML, Cocoon, and Xerces-J, over half of the *CC-classes* contain *LP*. It indicates that a method which realizes complex treatments, using many if and switch instructions usually has many parameters.

Anti-singleton (*AS*) and *Class Data Should Be Private* (*CP*). For Log4j and Xalan which have a number of *AS*- and *CP-classes*, our results indicate that *AS* and *CP* are coupled in the same class. 44.8% to 66.0% of the *AS-classes* contain *CP*, while 34.2% to 56.9% of the *CP-classes* have *AS*. An *AS-class* declares a public class variable that acts as the global variable, thus possibly exposing its fields. *CP* occurs when the data encapsulated by a class is public, thus allowing client classes to change this data without the knowledge of the declaring class. The two kinds of anti-patterns tend to be present in the same class since they share common roots and originate from the similar code flaws, which violate the principle of encapsulation.

Blob (BB) and *Complex Class (CC)/Long Parameter List (LP)*. A *BB-class* monopolizes most of the processing done by a system, takes most of the decisions, and closely directs the processing of other classes. Out of 71 *BB-classes* in Xerces-J, we find that 30 classes are infected with *CC*, 28 classes with *LP* and 18 classes with both. It indicates that 56.3% of *BB-classes* have at least one method with a high complexity or a great number of parameters. In addition,

63.8% of the *CC-classes contains BB*. This finding accords with the intuition that a complex class may implement many functionalities.

2) Further analysis on dependent and co-changed anti-patterns

Our results show that several pairs of anti-patterns are coupled across dependent or co-changed classes. For the sake of a better understanding of the impact of an anti-pattern on its dependent or co-changed ones, we calculated Cliff's δ for each compared pair. Due to the limited space, TABLE VIII only shows the results of call-dependent anti-patterns in Xerces-J, which contains the most number and kinds of anti-patterns. The rows and columns indicate the anti-patterns that a class and its dependent classes are infected with respectively. Each entry is the value of Cliff's *δ*. It is clear that the effect sizes are trivial or small in most cases. It indicates that although the occurrence of a certain kind of anti-pattern in a class will increase the probability of another kind in its dependent classes, the promotion is very limited. This finding implies that it is difficult for most anti-patterns to affect other classes through static dependencies or temporal associations. However, there exist some exceptions. The impacts of *Spaghetti Code* to their coupled anti-patterns are always large in Xalan and Log4j. The odds ratios calculated for other projects can be seen in Appendix.

Furthermore, since existing work has stated that interacted anti-patterns which are distributed across coupled classes can lead to maintenance problems [13], we also observed the relationship between dependent anti-patterns and maintenance activities. Significantly, in this study, we consider a maintenance activity is completed during a change period (mentioned in section III), and the classes modified in this period are all related to the activity. We care about whether the classes with dependent anti-patterns were revised in any of the studied change periods. TABLE IX and TABLE X summarize the relevant statistical information about data- and call-dependent anti-patterns respectively. The first row lists the number of pairs of inter-dependent classes which are infected with two coupled anti-patterns (called AD pairs). The second and fourth rows show the numbers of AD pairs, of whom at least one class or both two classes were modified in maintenance tasks (called AAD or SAD pairs respectively). Note that the two classes were not necessarily changed in the same change period. The third and fifth rows indicate the proportions of AAD and SAD pairs in AD pairs separately. We find that for ArgoUML, Xalan and Xerces-J, at least one class of most AD pairs (from 69.9% to 100%) was related to maintenance issues. For over 80% of AD pairs in ArgoUML and Xalan, both two classes were modified. However, the proportions of AAD and SAD pairs are relatively small in Cocoon and Log4j. These findings, on one hand, indicate that dependent anti-patterns are tightly bound to software maintenance in some projects, but are not in other projects. We guess that it is possibly due to the unique software features, as well as the different numbers and kinds of coupled anti-patterns in different projects. On the other hand, the large proportions of AAD or SAD pairs in ArgoUML Xalan and Xerces-J imply that if a software practitioner finds two dependent anti-patterns distributed in the coupled classes, these classes are very likely to associate with maintenance issues. Thus, dependent anti-patterns may be an indicator for maintainability, such as change-proneness and fault-proneness. The impacts of each pair of dependent anti-patterns on various maintenance issues deserve further analyses. It can be used to build prediction models, in order to help developers find and resolve problems earlier.

Additionally, co-changed anti-patterns, which tend to be located in two modified classes during a maintenance task, can also provide useful information for software refactoring and evolution. Our results indicate that the classes co-

TABLE VIII. CLIFF'S δ FOR CALL-DEPENDENT ANTI-PATTERNS IN XERCES-J

	AS	**BB**	**CP**	**CC**	**YC**	**LP**	**MC**	**RB**	**SG**	**SK**
AS			0.248							
BB	0.051		0.110	0.206		0.301	0.306		0.163	0.293
CP						0.391				0.288
CC	0.087	0.172	0.315			0.154	0.170		0.148	0.288
YC										
LP			0.253				0.130			
MC		0.206	0.409		0.101	0.248				0.157
RB				0.006	0.063					
SG										
SK			0.612						0.131	

TABLE IX. THE RELATIONSHIP BETWEEN DATA-DEPENDENT ANTI-PATTERNS AND MAINTENANCE ACTIVITIES

	ArgoUML	**Cocoon**	**Log4j**	**Xalan**	**Xerces-J**
# of AD pairs	190	12	60	203	509
# of AAD pairs	189	0	24	203	370
% of AAD pairs	99.5%	0.0%	40.0%	100.0%	72.7%
# of SAD pairs	162	0	5	169	130
% of SAD pairs	85.3%	0.0%	8.3%	83.3%	25.5%

TABLE X. THE RELATIONSHIP BETWEEN CALL-DEPENDENT ANTI-PATTERNS AND MAINTENANCE ACTIVITIES

	ArgoUML	Cocoon	Log4j	Xalan	Xerces-J
# of AD pairs	87	36	154	417	621
# of AAD pairs	87	0	53	417	434
% of AAD pairs	100.0%	0.0%	34.4%	100.0%	69.9%
# of SAD pairs	80	0	12	373	173
% of SAD pairs	92.0%	0.0%	7.8%	89.4%	27.9%

TABLE XI. ODDS RATIOS OF CONFLICTING ANTI-PATTERNS

	CC&YC	YC&LP	CB&YC	BB&MC	BB&RB	CC&RB	CP&RB	MC&RB	BB&SK
ArgoUML	0	0.27							
Xalan	0		0						
Xerces-J				0	0.50		0.40	0.16	0

changed with some certain kinds of anti-patterns, such as C*omplex Class* and *Long Parameter List*, are more likely to participate in their coupled anti-patterns. Therefore, if a developer finds an anti-pattern when he is revising a class, he has the reason to suspect that its coupled anti-patterns may infect the co-changed classes. Then, he could further examine their existence and inspect the infected classes, thus to improve the class structures and avoid potential bad impacts.

3) Conflicting anti-patterns

When analyzing the relationships between anti-patterns, we find that some anti-patterns rarely appear in the same class. In ArgoUML and Xalan, Fisher's exact tests show that the occurrences of *Complex Class*/*Long Parameter List* and *Lazy Class* are significantly correlated with each other, but the odds ratios are lower than 1 (ranging from 0 to 0.27). It indicates that the presence of *Complex Class* or *Long Parameter List* rejects the existence of *Lazy Class*, and vice versa. It is natural since these anti-patterns manifest themselves in exclusive code flaws. A *CC*- or *LP-class*, which has methods with a high complexity or a large number of parameters, is usually designed to implement some functionalities. This class is unlikely an *YC-class*, which does too little. Additionally, there are other conflicting anti-patterns found in Xerces-J. *Refused Parent Bequest* is incompatible with *Message Chains*, *Complex Class*, *Class Data Should Be Private,* and *Blob*. *Blob* and *Swiss Army Knife* are hardly co-located in the same class. TABLE XI presents the odds ratios of conflicting anti-patterns.

It is of great importance to further investigate the mutually exclusive relationship of anti-patterns and the underlying causes. It will not only help researchers and practitioners to better understand anti-patterns, but also provides useful information for the detection of them, especially for those whose presence are difficult to decide. The knowledge of the occurrence of one anti-pattern allows giving up further exploration towards its conflicting ones, thus easing the detection process.

4) Implications for research and practice

This study investigates the interactions between anti-patterns and reveals several pairs of them are likely to be present in the same class or coupled classes. Our findings provide concrete empirical evidences for some of the inter-pattern relations conjectured earlier by Pietrzak and Walter [14]. We also confirm the feasibility of leveraging anti-pattern detection with their interactions. On one hand, since some anti-patterns share common code flaws, the existence of already discovered anti-patterns becomes a valuable indicator for others. On the other hand, the coupled relationships together with the above-mentioned conflicting relationships can be used to check the detection results, thus reducing false positives. However, because the impact of an anti-pattern on another through data or call dependency is very limited, their interactions across different classes should be carefully used.

In addition, a deep analysis on the effect of multiple anti-patterns on maintenance issues is needed. Previous studies concentrate on individual anti-patterns. However, we find that many classes contains more than one kind of anti-patterns. For example, in ArgoUML, 33.3% of the infected classes have two or more kinds. Since some anti-patterns tend to be present together, the existence of its coupled ones may confound the effects of a given kind of anti-pattern. One cannot tell whether the negative effects are caused by a single anti-pattern or the interactions between them, which is left for the researchers to further investigate.

Moreover, our results indicate that the presence of some certain anti-patterns in a class does affect the occurrence of others in its coupled classes. This phenomenon implies that the impact of anti-patterns can transmit to other code elements through static dependencies or temporal associations. We argue that the study on the anti-pattern propagation is an interesting topic that deserves more attention. The exploration into the propagation path and range will lead to a more complete understanding of anti-patterns.

V. THREATS TO VALIDITY

We now discuss the threats to validity of our study following the guidelines provided for case study research [24].

Construct validity is the degree to which the variables used in a study accurately measure the concept they purport to measure. Firstly, the anti-patterns detection strategies implemented in DECOR includes some subjective understanding of the definitions of anti-patterns, which could be a potential threat to validity. However, since DECOR is most frequently used in literature with a 100% recall and a precision averagely greater than 60%, we believe that the detection results are reliable and acceptable. The second

threat concerns the degree of accuracy of static dependency relationships, which depends on the tool we used. We choose *Understand* because it is a mature commercial tool and has been used to identify dependencies in many previous studies [25-28]. In future work, other static tools will be adopted to compare and validate the collected data and call dependencies. In addition, other kinds of static dependencies will be included in further analysis on the dependent anti-patterns. The last threat lies in the identification of temporal associations between co-changed classes during a maintenance task. Our method of allocating change logs into several change periods is based on Hatton L's empirical study [16], which is also adopted by Jaafar et al. [17] to achieve a higher accuracy of identifying co-changed files.

Conclusion validity concerns the relation between the treatment and the outcome. To mitigate these threats, our conclusions have been supported by proper statistical tests. We choose Wilcoxon rank-sum test, because it is a non-parametric test which does not require any assumption on the underlying data distribution. In addition, though in practice Fisher' exact test is employed when sample sizes are small, it is valid for all sample sizes.

External validity concerns the possibility to generalize our findings to other systems. We have studied five long-lived and popular open-source projects varying in size and application domain. The data sets collected from these systems are large enough to draw statistically meaningful conclusions. We believe that our study makes a significant contribution to the software engineering body of empirical knowledge about the interaction between anti-patterns. Nevertheless, we cannot assume that our results generalize beyond the specific environment where they were conducted. Further validation on a larger set of software systems is desirable.

VI. Conclusion

Previous studies have raised the awareness of the impact of anti-pattern interactions on software development and maintenance activities. Inspired by them, we proposed experiments to investigate which two kinds of anti-patterns interact with each other in order to gather quantitative evidences on the existence of coupled anti-patterns.

Conducted on five open-source projects, this study first explored which of 10 kinds of anti-patterns are likely to appear in the same class using Fisher's exact test and odds ratio. Then, Wilcoxon rank-sum test was employed to determine whether the occurrence of one kind of anti-pattern in a class will positively affect another in its dependent and co-changed classes. Based on these results, we have the following findings: 1) several pairs of anti-patterns tend to be co-located in a class, such as *Complex Class* and *Long Parameter List*; 2) some anti-patterns will significantly increase the possibility that its coupled classes participate in certain anti-patterns, but the impact is small in most cases; 3) in ArgoUML, Xalan and Xerces-J, a large proportion of the classes infected with dependent anti-patterns have been changed during maintenance tasks; and 4) there exists some conflicting anti-patterns which are not likely to occur together, e.g. *Complex Class* and *Lazy Class*.

This study quantitatively shows the existence of anti-pattern interactions, thus providing valuable data in an important area for which there is limited experimental data available. Moreover, our findings offers important implications for research and practice, which also guide us to extend the future work in the following directions. First, the confirmed coupled and conflicting anti-patterns carry the information of each other, which can be used to ease the detection process and promote detection accuracy. Thus, we intend to develop anti-pattern detection tools with the aid of anti-pattern interactions. Secondly, since previous studies concluded that some inter-pattern relations may cause maintenance problems, a quantitative analysis of the impact of identified coupled anti-patterns on software maintenance and evolution is then needed. Thirdly, anti-patterns are able to propagate through static dependencies and temporal associations between source codes. A further study into anti-pattern propagation is of great importance for a deep understanding of them. We also plan to replicate this study on more software systems with other programming languages and application domains to examine whether our conclusions drawn in this study can be generalized to other systems.

Acknowledgment

This work is supported by the National Key Basic Research and Development Program of China (2014CB340702), the National Natural Science Foundation of China (91418202, 61472178, 61472175, 61170071), and the National Natural Science Foundation of Jiangsu Province (BK2011190). All support is gratefully acknowledged.

Appendix

TABLE XII to TABLE XX show the Cliff's δ calculated for the comparisons of the data- or call- dependent anti-patterns in ArgoUML, Cocoon, Log4j, Xalan and Xerces-J. Note that the results for call-dependent anti-patterns in Xerces-J are listed in Section IV. TABLE XXI to TABLE XXIV show the Cliff's δ calculated for the comparisons of the co-changed anti-patterns in ArgoUML, Log4j, Xalan and Xerces-J.

TABLE XII. Cliff's δ for Data-dependent APs in ArgoUML

	AS	CC	YC	LP	RB
AS					
CC			0.021	0.341	0.012
YC					
LP		0.150	0.016		0.007
RB					

TABLE XIII. Cliff's δ for Call-dependent APs in ArgoUML

	AS	CC	YC	LP	RB
AS					
CC	0.029		0.069		
YC					0.019
LP	0.016		0.048		
RB			0.219		

TABLE XIV. CLIFF'S δ FOR DATA-DEPENDENT APS IN COCOON

	CP	CC	YC	LP
CP				
CC				
YC				
LP		0.148	0.020	

TABLE XV. CLIFF'S δ FOR CALL-DEPENDENT APS IN COCOON

	CP	CC	YC	LP
CP			0.010	0.330
CC				
YC				
LP		0.266	0.037	

TABLE XVI. CLIFF'S δ FOR DATA-DEPENDENT APS IN LOG4J

	AS	CP	CC	YC	LP	RB	SG
AS							0.198
CP							
CC	0.052	0.048		0.018	0.187		0.030
YC		0.082					
LP	0.087	0.058	0.165				
RB							
SG	1.000						

TABLE XVII. CLIFF'S δ FOR CALL-DEPENDENT APS IN LOG4J

	AS	CP	CC	YC	LP	RB	SG
AS							0.182
CP							
CC	0.199	0.180			0.192		
YC							
LP	0.117	0.072					
RB							
SG	0.944						

TABLE XVIII. CLIFF'S δ FOR DATA-DEPENDENT APS IN XALAN

	AS	CP	CC	YC	LP	RB	SG
AS		0.132		0.105	0.186	0.031	
CP	0.149			0.182	0.192		
CC				0.072			
YC							
LP	0.223	0.368	0.197				0.010
RB	0.548						
SG					0.960		

TABLE XIX. CLIFF'S δ FOR CALL-DEPENDENT APS IN XALAN

	AS	CP	CC	YC	LP	RB	SG
AS		0.159		0.089	0.247		
CP				0.100			
CC				0.076			
YC			0.165				
LP	0.320	0.219	0.188				
RB							
SG					0.902		

TABLE XX. CLIFF'S δ FOR DATA-DEPENDENT APS IN XERCES-J

	AS	BB	CP	CC	YC	LP	MC	RB	SG	SK
AS										
BB			0.154	0.465	0.117	0.409	0.375		0.172	0.305
CP		0.278		0.333		0.412	0.199			0.357
CC	0.054	0.422	0.343		0.190	0.302	0.174	0.210	0.173	0.171
YC										
LP		0.185	0.347	0.162	0.092		0.119			
MC		0.475	0.436	0.442	0.252	0.308				0.171
RB										
SG		0.678			0.221					
SK		0.503	0.822		0.328	0.331		0.306		

TABLE XXI. CLIFF'S δ FOR CO-CHANGED APS IN LOG4J

	AS	CP	CC	YC	LP	SG
AS		0.073			0.115	
CP	0.052		0.096		0.054	
CC						
YC						
LP	0.126	0.082	0.108			
SG						

TABLE XXII. CLIFF'S δ FOR DATA-DEPENDENT APS IN XERCES-J

	AS	BB	CP	CC	YC	LP	MC	RB	SG	SK
AS										
BB	0.102									
CP										
CC	0.183	0.363	0.241			0.386	0.362	0.273	0.201	0.389
YC										
LP	0.029								0.103	
MC	0.156	0.429	0.143	0.434		0.502		0.392	0.289	0.433
RB										
SG										
SK	0.159	0.299		0.321		0.299	0.289		0.292	

TABLE XXIII. CLIFF'S δ FOR CO-CHANGED APS IN ARGOUML

	AS	CC	YC	LP	RB
AS					
CC	0.062		0.247	0.433	
YC		0.381		0.360	
LP	0.055	0.360	0.167		
RB					

TABLE XXIV. CLIFF'S δ FOR CO-CHANGED APS IN XALAN

	AS	CP	CC	YC	LP	RB	SG
AS		0.281	0.477		0.234		0.577
CP							
CC	0.535	0.492			0.460		0.508
YC						0.411	
LP		0.101	0.142	0.108		0.138	0.122
RB							
SG	0.674		0.649		0.820		

REFERENCES

[1] Abbes M, Khomh F, Guéhéneuc Y G, and Antoniol G, "An empirical study of the impact of two antipatterns, *Blob* and *Spaghetti Code*, on program comprehension", 15th European Conference on IEEE Software Maintenance and Reengineering, 2011: 181-190.

[2] Du Bois B, Demeyer S, Verelst J, Mens T., and Temmerman M, "Does god class decomposition affect comprehensibility?", IASTED Conf on Software Engineering, 2006: 346-355.

[3] Deligiannis I, Stamelos I, Angelis L, Roumeliotis M., and Shepperd M, "A controlled experiment investigation of an object-oriented design heuristic for maintainability", Journal of Systems and Software, 2004, 72(2): 129-143.

[4] Kim M, Sazawal V, Notkin D, and Murphy G, "An empirical study of code clone genealogies", ACM SIGSOFT Software Engineering Notes, ACM, 2005, 30(5): 187-196.

[5] Khomh F, Di Penta M, and Gueheneuc Y. "An exploratory study of the impact of code smells on software change-proneness", 16th Working Conference on IEEE Reverse Engineering, 2009: 75-84.

[6] Olbrich S M, Cruzes D S, and Sjoberg D I K. "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems", 2010 IEEE International Conference on Software Maintenance, IEEE, 2010: 1-10.

[7] Olbrich S, Cruzes D S, Basili V, and Zazworka N, "The evolution and impact of code smells: A case study of two open source systems", Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement, IEEE Computer Society, 2009: 390-400.

[8] Khomh F, Di Penta M, Guéhéneuc Y G, and Antoniol G, "An exploratory study of the impact of antipatterns on class change-and fault-proneness", Empirical Software Engineering, 2012, 17(3): 243-275.

[9] Hall T, Zhang M, Bowes D, and Sun Y, "Some code smells have a significant but small effect on faults", ACM Transactions on Software Engineering and Methodology, 2014, 23(4): 33.

[10] Li W, and Shatnawi R, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution", Journal of systems and software, 2007, 80(7): 1120-1128.

[11] D'Ambros M, Bacchelli A, and Lanza M. "On the impact of design flaws on software defects", (QSIC), 10th International Conference on Quality Software, 2010: 23-31.

[12] Rahman F, Bird C, and Devanbu P, "Clones: What is that smell?", Empirical Software Engineering, 2012, 17(4-5): 503-530.

[13] Yamashita A, and Moonen L, "Exploring the impact of inter-smell relations on software maintainability: an empirical study?", Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, 2013: 682-691.

[14] Pietrzak B, and Walter B. "Leveraging code smell detection with inter-smell relations", Extreme Programming and Agile Processes in Software Engineering, Springer Berlin Heidelberg, 2006: 75-84.

[15] Moha N, Gueheneuc Y G, Duchien L, and Le Meur A, "DECOR: A method for the specification and detection of code and design smells", IEEE Transactions on Software Engineering, 2010, 36(1): 20-36.

[16] Hatton L, "How accurately do engineers predict software maintenance tasks?", IEEE Computer, 2007, 40(2): 64-69.

[17] Jaafar F, Guéhéneuc Y G, Hamel S, and Antoniol G, "An exploratory study of macro co-changes", 18th Working Conference on Reverse Engineering , IEEE, 2011: 325-334.

[18] Jaafar F, Gueheneuc Y G, Hamel S, and Khomh F, "Mining the relationship between anti-patterns dependencies and fault-proneness", 20th Working Conference on Reverse Engineering (WCRE), IEEE, 2013: 351-360.

[19] Sheskin D J, Handbook of parametric and nonparametric statistical procedures, crc Press, 2003.

[20] Mosteller F, "Association and estimation in contingency tables", Journal of the American Statistical Association, 1968: 1-28.

[21] Fay M P, and Proschan M A, "Wilcoxon-Mann-Whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules", Statistics surveys, 2010, 4: 1

[22] Romano J, Kromrey J, Coraggio J, and Skowronek J, "Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys?", Annual meeting of the Florida Association of Institutional Research, 2006: 1-3.

[23] McCormick H W, Mowbray T J, and Malveau R C, AntiPatterns: refactoring software, architectures, and projects in crisis, John Wiley & Sons, Hoboken, NJ, 1998.

[24] Yin R K, Case Study Research: Design and Methods - Third Edition,3rd ed, SAGE Publications, 2002.

[25] Zhou Y, Xu B, and Leung H, "On the ability of complexity metrics to predict fault-prone classes in object-oriented systems", Journal of Systems and Software, 83(4), 2010: 660-674.

[26] Zhou Y, Leung H, Xu B, "Examining the potentially confounding effect of class size on the associations between object-oriented Metrics and change-proneness", IEEE Transactions on Software Engineering, 35(5), 2009: 607-623.

[27] Pan K, Kim S, and Whitehead E J, "Bug classification using program slicing metrics", Sixth IEEE International Workshop on Source Code Analysis and Manipulation, IEEE, 2006: 31-42.

[28] Koru A, and Tian J, "Comparing high-change modules and modules with the highest measurement values in two large-scale open-source products", IEEE Transactions on Software Engineering, 31(8), 2005: 625-642.

[29] Zimmermann T, and Nagappan N, "Predicting defects with program dependencies", Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, IEEE Computer Society, 2009: 435-438.

[30] Zimmermann T, and Nagappan N, "Predicting defects using network analysis on dependency graphs", Proceedings of the 30th international conference on Software engineering, ACM, 2008: 531-540.

[31] Bavota G, Dit B, Oliveto R, Di Penta M, Poshyvanyk D, and De Lucia A, "An empirical study on the developers' perception of software coupling", Proceedings of the 2013 International Conference on Software Engineering, IEEE Press, 2013: 692-701.