Empir Software Eng (2016) 21:896–931
DOI 10.1007/s10664-015-9361-0

Evaluating the impact of design pattern and anti-pattern dependencies on changes and faults

Fehmi Jaafar · Yann-Gaël Guéhéneuc · Sylvie Hamel · Foutse Khomh · Mohammad Zulkernine

Published online: 10 March 2015

Abstract On the one hand, design patterns are solutions to recurring design problems, aimed at increasing reuse, flexibility, and maintainability. However, much prior work found that some patterns, such as the Observer and Singleton, are correlated with large code structures and argued that they are more likely to be fault prone. On the other hand, anti-patterns describe poor solutions to design and implementation problems that highlight weaknesses in the design of software systems and that may slow down maintenance and increase the risk of faults. They have been found to negatively impact change and fault-proneness. Classes participating in design patterns and anti-patterns have dependencies with other classes, e.g., static and co-change dependencies, that may propagate problems to other classes. We investigate the impact of such dependencies in object-oriented systems by studying the relations between the presence of static and co-change dependencies and (1) the fault-proneness, (2) the types of changes, and (3) the types of faults that these classes exhibit. We analyze six design patterns and 10 anti-patterns in 39 releases of ArgoUML, JFreeChart, and XercesJ, and investigate to what extent classes having dependencies with design patterns or anti-patterns have higher odds of faults than other classes. We show that in almost all releases of

Communicated by: Romain Robbes, Massimiliano Di Penta and Rocco Oliveto

F. Jaafar (✉) · M. Zulkernine
School of Computing, Queen's University, Ontario, Canada
e-mail: jaafar@cs.queensu.ca

M. Zulkernine
e-mail: mzulker@cs.queensu.ca

S. Hamel
DIRO, Université de Montréal, Québec, Canada
e-mail: hamelsyl@iro.umontreal.ca

Y.-G. Guéhéneuc · F. Khomh
DGIGL, École Polytechnique de Montréal, Québec, Canada

Y.-G. Guéhéneuc
e-mail: yann-gael.gueheneuc@polymtl.ca

F. Khomh
e-mail: foutse.khomh@polymtl.ca

the three systems, classes having dependencies with anti-patterns are more fault-prone than others while this is not always true for classes with dependencies with design patterns. We also observe that structural changes are the most common changes impacting classes having dependencies with anti-patterns. Software developers could use this knowledge about the impact of design pattern and anti-pattern dependencies to better focus their testing and reviewing activities towards the most risky classes and to propagate changes adequately.

Keywords Anti-patterns · Design patterns · Faults proneness · Change proneness · Static relationships · Co-change

1 Introduction

Developers use design patterns as recurring design solutions for object-oriented design problems that can improve several quality attributes of their classes, such as reusability, flexibility, and ultimately maintainability. Yet, recent work, e.g., (Aversano et al. 2009; Gatrell and Counsell 2011; Vokac 2004), has shown that some design pattern classes, such as the ones participating in the Composite design pattern, were more fault-prone than non design-pattern classes in some systems. Similarly, developers may introduce poor solutions when solving recurring design problems in their object-oriented systems. These poor solutions are documented in the form of anti-patterns (Webster 1995). Having instances of anti-patterns in a design negatively impacts code quality (Moha et al. 2010) because anti-pattern classes are more change and fault-prone than others (Khomh et al. 2012). However, no previous work (Aversano et al. 2009; Gatrell and Counsell 2011; Khomh et al. 2012; Moha et al. 2010) analyzed the impact on faults of the presence of instances of design patterns and anti-patterns on the rest of the classes of a system through their static and co-change dependencies. Static dependencies between classes in software systems are typically use, association, aggregation, and composition relationships (Guéhéneuc and Albin-Amiot 2004). Co-change dependencies (or temporal dependencies) exist when developers, while changing a class, also must change other classes. It is not clear how classes having static or co-change dependencies with instances of anti-patterns and design-patterns are linked with faults.

Conjecture Therefore, our conjecture is that classes having dependencies with anti-pattern and design-pattern classes could be involved in fault fixing changes more often than other classes in a system and that faults could propagate from anti-pattern and design-pattern classes to other classes through their dependencies.

Context We showed in our previous work (Jaafar et al. 2013a, b) that more attention should be given to static and co-change dependencies between classes participating in instances of anti-patterns and other classes. The finding by Marinescu and Marinescu (2011), that clients of classes with Identity Disharmonies are more fault-prone than others, also supported our work. In this paper, we replicate our study on 10 more releases of ArgoUML, JFreeChart, and XercesJ to assess the generalizability of our results. We also present new results about the impacts of static and co-change dependencies between classes participating in instances of design patterns and other classes. Another contribution with respect to previous work is a qualitative analysis of these impacts by reporting the variety of changes and faults occurring in affected classes.

Thus, we extend our previous work (Jaafar et al. 2013b) with the following three contributions. First, we extend our approach to analyze the fault proneness of classes having dependencies with design patterns. Second, we provide more details about our findings by presenting a fine-grain analysis and some actionable recommendations. Third, we investigate the kind of changes and faults impacting classes having dependencies with anti-pattern or design-pattern classes. We study structural changes (Gerlec and Hericko 2012), which are defined as changes that transform an object-oriented constituent (e.g., class, method, field) as types of changes. We study data fault, interface fault, logic fault, description fault, and syntax fault as types of faults.[1]

Research Questions We analyze the static and co-change dependencies between anti-pattern and design-pattern classes and other classes in two ways. Quantitatively, we investigate whether classes having static relationships (use, association, aggregation, and composition relationships) with instances of anti-patterns or design patterns are more fault-prone than others. Then, we investigate whether classes co-changing with anti-pattern or design pattern classes are more fault-prone than others. Qualitatively, we analyze the types of changes and faults impacting these classes. Consequently, we formulate the following research questions:

- **RQ1:** *Are classes that have static relationships with anti-pattern classes more fault-prone than other classes?*
- **RQ2:** *Are classes that co-change with anti-pattern classes more fault-prone than other classes?*
- **RQ3:** *Are classes that have static relationships with design pattern classes more fault-prone than other classes?*
- **RQ4:** *Are classes that co-change with design pattern classes more fault-prone than other classes?*
- **RQ5:** *What types of changes and faults are propagated by static and co-change dependencies?*

We perform the study on 11 releases of ArgoUML, 9 of JFreeChart, and 19 of XercesJ, and across the changes and fault-fixing changes occurring between these releases. We detect the instances of six design patterns and 10 anti-patterns in these systems to investigate the relations between anti-pattern and design pattern classes and other classes through their static and co-change dependencies while considering changes and faults.

The major results of this paper are that (1) static dependencies with anti-pattern classes do have a negative impact on changes and faults and (2) co-change dependencies with anti-pattern and design pattern classes have similar negative impacts on changes and faults. We also discuss the types of changes and faults that affect classes with dependencies on anti-pattern and design pattern classes. Classes with dependencies on anti-pattern classes are more subject to logic faults and structural changes. Classes with dependencies on design pattern classes are more subject to code addition changes and syntax faults.

Finally, we show the benefits of considering static and co-change dependencies with anti-pattern and design pattern classes to improve the precision of the prediction of class fault-proneness, along with other discussions on the usefulness and limitations of our study and of its results. We show that considering anti-pattern dependencies improve the prediction of fault-prone classes.

[1] http://standards.ieee.org/findstds/standard/1044-2009.html

Organization Section 2 presents our methodology. Section 3 describes the empirical study. Section 4 presents the study results while Section 5 analyzes the results along with threats to their validity. Then, Section 6 relates our study with previous work. Finally, Section 7 concludes the study and outlines future work.

2 Methodology

This section describes our methodology as well as the steps necessary to extract and analyze the data required to perform our empirical study. Given an object-oriented program, we use DECOR (Moha et al. 2010) to extract instances of anti-patterns and DeMIMA (Guéhéneuc and Antoniol 2008) to extract instances of design patterns. As we focused on the dependencies of these patterns, we use PADL[2] to extract static relationships and Macocha (Jaafar et al. 2011) to detect co-changes. Finally, we analyze the impact of anti-pattern and design-pattern dependencies on fault-proneness and we investigate the types of changes and faults propagated by such dependencies.

2.1 Identifying Anti-pattern Classes

We use DECOR (Moha et al. 2010) to specify and detect instances of anti-patterns. DECOR is based on a thorough domain analysis of anti-patterns defined in the literature and provides a domain-specific language to specify code smells and anti-patterns and methods to detect their occurrences automatically. DECOR uses rules to describes anti-patterns, with different types of properties: lexical (i.e., class names), structural (i.e., classes declaring public static variables), internal (i.e., number of methods), and the relation among properties (i.e., use, association, aggregation, and composition relationships among classes). Using this language, DECOR describes several anti-patterns.

In the following, we consider the instances of 10 anti-patterns from Brown et al. (1998) described in Table 1. We choose these anti-patterns because they are representative of problems with data, complexity, size, and the features provided by classes (Khomh et al. 2012). We also use these anti-patterns because they have been used and analyzed in previous work (Khomh et al. 2012; Moha et al. 2010). Definitions and specifications are beyond the scope of this paper and are available elsewhere (Romano et al. 2012; Brown et al. 1998).

We preprocess inconsistent anti-patterns (due to eventual errors and imprecisions of the detection tools) to eliminate false positives. We manually validate each occurrence of the anti-patterns detected by DECOR to verify the correctness of the analyzed set. This preprocessing reduces the risks that we would answer our research questions incorrectly. Yet, our results could still be affected by the presence of false negatives, i.e., by a low recall exhibited by the detection approach. However, the sample of detected instances of anti-patterns is large enough to support our conclusions.

2.2 Identifying Design Pattern Classes

We use DeMIMA (Guéhéneuc and Antoniol 2008) to specify and detect instances of design patterns. DeMIMA ensures the traceability between design patterns and source code by first identifying idioms related to binary class relationships to obtain an idiomatic model of the

[2] http://www.ptidej.net/tool/

Table 1 List of anti-patterns considered in this study

Anti-patterns	Specifications
AntiSingleton	A class that provides mutable class variables, which consequently could be used as global variables
Blob	A class that is too large and not cohesive enough, that monopolizes most of the processing, makes most of the decisions, and is associated with data classes
ClassDataShouldBePrivate	A class that exposes its fields, thus violating the principle of encapsulation
ComplexClass	A class that has (at least) one large and complex method, in terms of cyclomatic complexity and LOCs
LongMethod	A class that has a method that is overly long in term of LOCs
LongParameterList	A class that has (at least) one method with an excessively long list of parameters with respect to the average number of parameters per method in the system
MessageChain	A class that uses a long chain of method invocations to realize (at least) one of its features
RefusedParentBequest	A class that redefines one or more inherited methods using empty bodies thus breaking polymorphism
SpeculativeGenerality	A class that is defined as abstract but that has very few children, which do not make use of its methods
SwissArmyKnife	A class whose methods can be divided in disjunct sets of many methods, thus providing many different unrelated functionalities

source code. Then, using this model, it can identify solutions to design patterns and generate a design model of the system. Thus, DeMIMA makes it possible to recover two types of design choices from source code: idioms pertaining to the relationships among classes and design motifs characterizing the organization of the classes.

Design patterns were originally grouped into the categories: creational patterns, structural patterns, and behavioral patterns (Gamma et al. 1994). Creational patterns describe creation mechanisms to create objects in a suitable manner. They make code functionalities independents of how objects are created, composed, and represented. Structural patterns ease the design by offering simple ways to relate classes. They are concerned with how classes and objects are composed to form larger structures. Finally, behavioral patterns describe common communication models among objects. They are concerned with algorithms and the assignment of responsibilities among objects to increase software system flexibility.

In the following, we focus on the instances of six design patterns belonging to these three categories: two creational patterns (Factory method and Prototype), two structural patterns (Composite and Decorator), and two behavioral patterns (Command and Observer), which are defined briefly in Table 2 while complete definitions are available elsewhere (Guéhéneuc and Antoniol 2008; Gamma et al. 1994). In addition to belonging to different categories, we chose these six design patterns because a preliminary study indicated that we could identify enough of their instances to carry out our study. Indeed, some other design patterns do not exist in the releases of the analyzed systems. Future work could replicate our study on other design patterns if enough of their instances can be identified.

Table 2 List of design patterns considered in this study

Design patterns	Specifications
Command	An object encapsulates all the information needed to call a method This information includes the method name, the object and values for the method parameters
Composite	Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly
Decorator	Extend the functionality of a certain object statically, or in some cases at run-time, independently of other instances of the same class
Factory method	Define an interface for creating an object, but let the classes that implement the interface decide which class to instantiate
Observer	An object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods
Prototype	Allows an object to create customized objects without knowing their class or any details of how to create them

2.3 Identifying Anti-patterns and Design-Pattern Static Relationships

DeMIMA depends on a set of formalizations for unidirectional binary class relationships (Guéhéneuc and Albin-Amiot 2004). The formalizations define the relationships in terms of four language-independent properties that are derivable from static and dynamic analyses of systems: exclusivity, type of message receiver, lifetime, and multiplicity. Thus, we also use DeMIMA to detect the relationships among classes (including anti-pattern classes).

We use the open-source Ptidej tool suite[3] to identify anti-pattern and design pattern classes as well as static relationships among classes. The Ptidej tool suite implements DECOR and DeMIMA and uses the PADL (Guéhéneuc and Antoniol 2008) meta-model. It parses the source code of systems to build models that include all of the constituents found in any object-oriented system: class, interface, member class and interface, method, field, inheritance and binary-class relationships.

First of all, we identify all the instances of the anti-patterns and design patterns of interest in the different analyzed systems. Then, we detect the dependencies of the classes in these instances with the rest of the classes. Thus, a class *A* having dependencies with two classes *B* and *C*, which belong to an instance of an anti-pattern or design pattern, is considered as having a dependency with the anti-pattern or design pattern.

2.4 Identifying Anti-pattern and Design-Pattern Co-changes

We use Macocha (Jaafar et al. 2011) to mine software repositories and identify classes that are co-changing with instances of the anti-patterns or design patterns of interest. In Macocha, a change contains several attributes: the changed class names, the dates of changes, the developers having committed the changes. Macocha takes as input a CVS/SVN change log. First, it calculates the duration of different change periods using the *k*-nearest

[3]https://bitbucket.org/yann-gael/ptidej-5/overview

neighbor algorithm (Dasarathy 1991). Second, it groups changes in change periods. Third, it creates a profile that describes the evolution of each class in each change period. Fourth, it uses these profiles to compute the stability of the classes and, then, to identify co-changed classes.

Similar to Aversano et al. (2007), we assume that a class C co-changes with an instance of a design pattern or an anti-pattern X, if C co-changes with at least one class participating in X. We also assume that a class S has static relationships with a design pattern or an anti-pattern instance A, if S has use, association, aggregation, or composition relationships with at least one class participating in A.

2.5 Identifying Faults in Classes

Fault-proneness refers to whether a class underwent at least one fault fix in the system life cycle. Fault fixes are documented in bug reports that describe different types of problems in a system. They are usually posted in issue-tracking systems, i.e., Bugzilla for the three studied systems, by users and developers to warn their community about pending issues with their functionalities; issues in these systems deal with different types of change requests: fixing faults, restructuring, and so on.

We parse the CVS/SVN change logs of our subject systems and apply the heuristics by Sliwerski et al. (2005) to identify fault fix locations and changes: we parse commit log messages using a Perl script and extract bug IDs and specific keywords, such as “fixed” or “bug” to identify fault-related commits from which we extract the list of files that were changed to fix the faults. Then, we use the commit log messages of the identified faults to categorize them as data fault, interface fault, logic fault, description fault, and syntax fault.

3 Study Definition Design

This section describes the design of our empirical study. We detail the approach and the analyzed object systems in the study. Then, we discuss our research questions and the analysis method.

The *goal* of our study is to assess whether classes having static or co-change dependencies with anti-pattern or design pattern classes have a higher likelihood than other classes to be involved in changes and faults. The quality focus is the improvement of maintainability and the reduction of maintenance effort by detecting and analyzing the impacts of anti-patterns and design patterns on faults and changes. The context of our study is both the comprehension and the maintenance of systems.

3.1 Object Systems

We apply our study on three Java systems: ArgoUML,[4] JFreeChart,[5] and XercesJ.[6] We use these systems because they are open source, are of different domains, span several years and versions, and have between hundreds and thousands of classes. Table 3 summarizes some statistics about these systems.

[4]http://ArgoUML.tigris.org

[5]http://www.jfree.org

[6]http://xerces.apache.org/xerces-j

Table 3 Descriptive statistics of the object systems

	ArgoUML	JFreeChart	XercesJ
# of classes	3,325	1,615	1,191
# of snapshots	4,480	2,010	159,196
# of releases	11	9	19
SLOC average	240,762	181,895	206,291
# of AntiSingleton	3	38	24
# of Blob	100	49	12
# of ClassDataShouldBePrivate	51	3	6
# of ComplexClass	158	52	7
# of LongMethod	336	75	7
# of LongParameterList	281	76	4
# of MessageChains	162	59	8
# of RefusedParentBequest	123	5	7
# of SpeculativeGenerality	22	3	29
# of SwissArmyKnife	13	26	29
# of Command	126	118	72
# of Composite	154	202	46
# of Decorator	48	36	29
# of Factory method	39	43	19
# of Observer	24	26	7
# of Prototype	38	29	16

ArgoUML is a UML diagramming system in Java and released under the open-source BSD License. For our study, we extracted a total number of 4,480 snapshots in the time interval between September 27^{th}, 2008 and December 15^{th}, 2011.

JFreeChart is a Java open-source framework to create charts. For our study, we considered an interval of observation ranging from June 15^{th}, 2007 (release 1.0.6) to November 20^{th}, 2009 (release 1.0.13α) in which we extracted 2,010 snapshots.

XercesJ is a collection of software libraries to manipulate XML documents. It is developed in Java and managed by the Apache Foundation. For our analysis, we extracted a total number of 159,196 snapshots from release 1.0.4 to release 2.9.0 in the time interval between October 14^{th}, 2003 and November 23^{th}, 2006.

Tables 4 and 5 show the proportions of co-changing classes and static relationships among classes participating in the anti-patterns and design patterns considered in this study. They show that the chosen systems are relevant for the study because some classes of these systems *do* co-change and have static relationships with anti-pattern and design-pattern classes.

3.2 Research Questions

We break down our study into three steps: first, we investigate whether classes co-changing or having static relationships (use, association, aggregation, and composition relationships)

Table 4 Proportion of the anti-pattern dependencies (CC: co-change dependencies of anti-pattern classes with other classes; SR: anti-pattern static relationships)

Anti-patterns	Systems	# of CC	# of SR
AntiSingleton	ArgoUML	13	152
	JFreeChart	20	201
	XercesJ	18	188
Blob	ArgoUML	51	304
	JFreeChart	36	164
	XercesJ	24	93
ClassDataShouldBePrivate	ArgoUML	4	167
	JFreeChart	0	82
	XercesJ	0	113
ComplexClass	ArgoUML	2	192
	JFreeChart	0	146
	XercesJ	0	96
LongMethod	ArgoUML	42	282
	JFreeChart	51	314
	XercesJ	0	266
LongParameterList	ArgoUML	12	344
	JFreeChart	0	276
	XercesJ	0	309
MessageChains	ArgoUML	48	244
	JFreeChart	8	196
	XercesJ	16	183
RefusedParentBequest	ArgoUML	47	326
	JFreeChart	6	183
	XercesJ	25	93
SpeculativeGenerality	ArgoUML	13	128
	JFreeChart	4	139
	XercesJ	8	201
SwissArmyKnife	ArgoUML	20	69
	JFreeChart	9	142
	XercesJ	18	108

with anti-pattern classes are more fault-prone than other classes in the three analyzed systems:

– For **RQ1:** Are classes that have static relationships with anti-pattern classes more fault-prone than other classes? We test the following null hypothesis:
  – H_{RQ1_0}: The proportions of faults carried by classes having static relationships with instances of anti-patterns and other classes are the same. If we reject the null hypothesis H_{RQ1_0}, it means that the proportions of faults carried by

Table 5 Proportion of the design-pattern dependencies (CC: co-change dependencies of design-pattern classes with other classes; SR: design-pattern static relationships)

Design patterns	Systems	# of CC	# of SR
Command	ArgoUML	63	269
	JFreeChart	25	226
	XercesJ	22	165
Composite	ArgoUML	81	254
	JFreeChart	66	144
	XercesJ	34	89
Decorator	ArgoUML	16	123
	JFreeChart	14	184
	XercesJ	10	86
Factory method	ArgoUML	13	58
	JFreeChart	15	37
	XercesJ	9	26
Observer	ArgoUML	42	69
	JFreeChart	48	64
	XercesJ	29	35
Prototype	ArgoUML	12	39
	JFreeChart	14	27
	XercesJ	8	28

classes having static relationships with anti-patterns and faults carried by other classes in the analyzed systems are not the same.

- For **RQ2:** Are classes that co-change with anti-pattern classes more fault-prone than other classes? We test the following null hypotheses:
  - H_{RQ2_0}: The proportions of faults involving classes having co-change dependencies with instances of anti-pattern and other classes are the same. If we reject the null hypothesis H_{RQ2_0}, the proportion of faults carried by classes co-changing with anti-patterns is not the same as the proportion of faults carried by classes not co-changing with anti-patterns.

Second, we investigate whether classes co-changing or having static relationships with instances of design patterns are more fault-prone than other classes.

- For **RQ3:** Are classes that have static relationships with design pattern classes more fault-prone than other classes? We test the following null hypothesis:
  - H_{RQ3_0}: The proportions of faults carried by classes having static relationships with instances of design patterns and other classes are the same. If we reject the null hypothesis H_{RQ3_0}, the proportions of faults carried by classes having static relationships with design patterns and faults carried by other classes are not the same.

- For **RQ4:** Are classes that co-change with design pattern classes more fault-prone than other classes? We test the following null hypothesis:
  - H_{RQ4_0}: The proportions of faults involving classes having co-change dependencies with instances of design patterns and other classes are the same. If we reject the null hypothesis H_{RQ4_0}, the proportion of faults carried by classes co-changing with design patterns is not the same as that of other classes.

Third, we conduct a qualitative study to explore the types of changes and faults carried by classes having dependencies with instances of anti-patterns or design patterns. Thus, we answer the following research question:

- For **RQ5:** What types of changes and faults are propagated by static and co-change dependencies? We study whether classes having dependencies with instances of anti-patterns or design patterns undergo specific structural changes (Gerlec and Hericko 2012) (addition/removal/change of/to attributes, addition/removal of methods, or changes to the methods' signatures) than non-structural changes, i.e., adding/deleting/changing lines without changing the structure of the classes. We also study whether such classes have specific types of faults (data faults, interface faults, logic faults, description faults, and syntax faults).

3.3 Analysis Method

We perform the analyses reported in Section 4 using the R statistical environment.[7] We use Fisher's exact test (Sheskin 2007) because it is a significance test that is considered to be more appropriate for sparse and skewed samples of data than other statistical tests, such as the log likelihood ratio or Pearson's Chi-Squared test (Pedersen 1996). Indeed, this test is useful for categorical data that results from classifying objects in two different groups. In our study, we examine the significance of the relation between the occurrence of a static or co-change dependency on anti-pattern or design pattern class(es) and the risk of fault.

We also compute the odds ratio (Sheskin 2007) that indicates the likelihood for an event to occur. The odds ratio shows the strength of association between a predictor and the response of interest. Indeed, its advantage is that it is invariable across case control, follow-up, and cross-sectional studies and, thus, can be used to directly compare findings of different study designs. Also, in the case of our study, the odds ratio can be computed directly from the regression coefficients of logistic regressions. Finally, it is a good estimator of risk ratio if the analyzed phenomena is rare and the cases and controls are randomly selected from the population (Rothman et al. 2004). The odds ratio is defined as the ratio of the odds p of an event occurring in one sample, i.e., the odds that classes having static relationships with anti-patterns are identified as fault-prone to the odds q of the same event occurring in the other sample, i.e., the odds that the rest of classes are identified as fault-prone. Thus, if the probabilities of the event in each of the groups are p (faulty classes for example) and q (not faulty classes), then the odds ratio is: $OR = \frac{p/(1-p)}{q/(1-q)}$. An odds ratio greater than 1 indicates that the event is more likely in the first sample, while an odds ratio less than 1 indicates that it is more likely in the second sample.

[7] http://www.r-project.org

4 Study Results

In this section, we answer each research question by reporting the empirical results and some typical examples and other observations. Tables 6, 7, 8, and 9 summarize our findings in term of fault-proneness.

4.1 RQ1: Are Classes that have Static Relationships with Anti-Pattern Classes more Fault-prone than other Classes?

Table 6 reports for ArgoUML, JFreeChart, and XercesJ the numbers of (1) classes having static relationships with anti-pattern classes and identified as faulty; (2) classes having static relationships with anti-pattern classes and identified as clean (i.e., not faulty); (3) classes without static relationships with anti-pattern classes and identified as faulty; and (4) classes without static relationships with anti-pattern classes and identified as clean. Considering all classes in the different analyzed systems, we also separate the results in Table 6 between

Table 6 Contingency table and Fisher test results for ArgoUML, JFreeChart, and XercesJ for classes having static relationships with anti-patterns (SR: static relationship, AP: anti-pattern)

	System	Faulty	Clean
Classes having SR with AP	ArgoUML	1,062	1,003
Classes having SR with AP and not being part of anti-pattern	ArgoUML	402	600
Class not having SR with AP	ArgoUML	681	579
Class not having SR with AP and not being part of anti-pattern	ArgoUML	205	456
Fisher's test in ArgoUML	0.9336		
Odd-ratio in ArgoUML	0.9002625		
Fisher's test for classes not being part of anti-patterns in ArgoUML	$9.241e-05$		
Odd-ratio for classes not being part of anti-patterns in ArgoUML	1.489969		
Classes having SR with AP	JFreeChart	432	226
Classes having SR with AP and not being part of anti-pattern	JFreeChart	281	103
Class not having SR with AP	JFreeChart	310	647
Class not having SR with AP and not being part of anti-pattern	JFreeChart	140	342
Fisher's test in JFreeChart	$2.2e-16$		
Odd-ratio in JFreeChart	3.98613		
Fisher's test for classes not being part of anti-patterns in JFreeChart	$2.2e-16$		
Odd-ratio for classes not being part of anti-patterns in JFreeChart	6.647686		
Classes having SR with AP	XercesJ	445	121
Classes having SR with AP and not being part of anti-patterns	XercesJ	262	75
Class not having SR with AP	XercesJ	126	499
Class not having SR with AP and not being part of anti-patterns	XercesJ	81	279
Fisher's test in XercesJ	$2.2e-16$		
Odd-ratio in XercesJ	14.5249		
Fisher's test for classes not being part of anti-patterns in XercesJ	$2.2e-16$		
Odd-ratio for classes not being part of anti-patterns in XercesJ	11.97391		

Table 7 Contingency table and Fisher test results for ArgoUML, JFreeChart and XercesJ for classes that co-changed with anti-patterns (AP: anti-pattern)

	System	Faulty	Clean
Classes co-changing with AP	ArgoUML	241	102
Classes co-changing with AP and and not being part of anti-patterns	ArgoUML	120	59
Class not co-changing with AP	ArgoUML	1,502	1,480
Classes not co-changing with AP and and not being part of anti-patterns	ArgoUML	1,023	852
Fisher's test in ArgoUML	$1.005e-12$		
Odd-ratio in ArgoUML	2.32758		
Fisher's test for classes not being part of anti-patterns in ArgoUML	0.0007733		
Odd-ratio for classes not being part of anti-patterns in ArgoUML	1.693495		
Classes co-changing with AP	JFreeChart	68	26
Classes co-changing with AP and and not being part of anti-patterns	JFreeChart	33	10
Class not co-changing with AP	JFreeChart	674	847
Classes not co-changing with AP and and not being part of anti-patterns	JFreeChart	357	482
Fisher's test in JFreeChart	$8.556e-08$		
Odd-ratio in JFreeChart	3.284357		
Fisher's test for classes not being part of anti-patterns in JFreeChart	$9.158e-06$		
Odd-ratio for classes not being part of anti-patterns in JFreeChart	4.4483		
Classes co-changing with AP	XercesJ	37	21
Classes co-changing with AP and and not being part of anti-patterns	XercesJ	20	12
Class not co-changing with AP	XercesJ	534	599
Classes not co-changing with AP and and not being part of anti-patterns	XercesJ	343	401
Fisher's test in XercesJ	0.009414		
Odd-ratio in XercesJ	1.975234		
Fisher's test for classes not being part of anti-patterns in XercesJ	0.02696		
Odd-ratio for classes not being part of anti-patterns in XercesJ	2.160657		

anti-pattern classes and other classes. We perform this separation because it is difficult to tell whether the observations made are caused by the fact that the classes belong to anti-patterns or by the fact that they depend upon classes belonging to anti-patterns.

The results of the Fisher's exact tests and odds ratios, when testing H_{RQ1_0}, are significant for all three systems. For the three systems, the p-values are less then 0.05 and the likelihood that a class with static relationship(s) with some anti-pattern class(es) experiences a fault (i.e., odds ratio) is about two times higher than the likelihood that other classes have faults.

We can thus positively answer **RQ1** as follows: classes having static relationships with anti-pattern classes are significantly more fault-prone than other classes.

Other Observations When we take as input the list of code and change metrics described in Section 2 and study whether there is a statistically significant difference on fault proneness between a model based only on these metrics and a model based on these metrics plus anti-pattern static relationships, we found that it is impossible to definitely exclude the possibility

Table 8 Contingency table and Fisher test results for ArgoUML, JFreeChart and XercesJ for classes having static relationships with design patterns (SR: static relationship, DP: design pattern)

	System	Faulty	Clean
Classes having SR with DP	ArgoUML	1,174	1,217
Classes having SR with DP and not being part of design patterns	ArgoUML	165	198
Classes not having SR with DP	ArgoUML	569	365
Classes not having SR with DP and not being part of design patterns	ArgoUML	326	263
Fisher's test in ArgoUML		1	
Odd-ratio in ArgoUML		0.618911	
Fisher's test for classes not being part of design patterns in ArgoUML		0.9988	
Odd-ratio for classes not being part of design patterns in ArgoUML		0.6725882	
Classes having SR with DP	JFreeChart	539	440
Classes having SR with DP and not being part of design patterns	JFreeChart	145	132
Classes not having SR with DP	JFreeChart	203	433
Classes not having SR with DP and not being part of design patterns	JFreeChart	258	228
Fisher's test in JFreeChart		$2.2e-16$	
Odd-ratio in JFreeChart		2.611343	
Fisher's test for classes not being part of design patterns in JFreeChart		0.6074	
Odd-ratio for classes not being part of design patterns in JFreeChart		0.9707799	
Classes having SR with DP	XercesJ	341	407
Classes having SR with DP and not being part of design patterns	XercesJ	16	18
Classes not having SR with DP	XercesJ	230	213
Classes not having SR with DP and not being part of design patterns	XercesJ	156	148
Fisher's test in XercesJ		0.9851	
Odd-ratio in XercesJ		0.7760848	
Fisher's for classes not being part of design patterns in XercesJ		0.7426	
Odd-ratio for classes not being part of design patterns in XercesJ		0.8437318	

that there is no statistically significant differences in fault prediction between classes having static relationships with anti-patterns and other classes with similar complexity, change history, and code size. However, if we group the results according to distinct anti-patterns (see Table 14 in Section 5), we observe that classes having static relationships with classes belonging to the Blob, ComplexClass, and SwissArmyKnife anti-patterns are significantly more fault prone than other classes with similar complexity, change history, and code size. We provide more details and discussions in Section 5.

4.2 RQ2: Are Classes that Co-change with Anti-pattern Classes more Fault-prone than other Classes?

In the three systems, we detect co-change situations for the majority of anti-pattern classes. In ArgoUML, we observe that classes participating in the Blob, LongMethod, and RefusedParentBequest anti-patterns co-change with other classes more than the rest of the

Table 9 Contingency table and Fisher test results for ArgoUML, JFreeChart and XercesJ for classes co-changing with design patterns (DP: design pattern)

	System	Faulty	Clean
Classes co-changing with DP	ArgoUML	139	93
Classes co-changing with DP and not being part of design patterns	ArgoUML	62	24
Classes not co-changing with DP	ArgoUML	1,604	1,489
Classes not co-changing with DP and not being part of design patterns	ArgoUML	1,308	1,056
Fisher's test in ArgoUML	0.01049		
Odd-ratio in ArgoUML	1.387325		
Fisher's test for classes not being part of design patterns in ArgoUML	0.001258		
Odd-ratio for classes not being part of design patterns in ArgoUML	2.085003		
Classes co-changing with DP	JFreeChart	61	72
Classes co-changing with DP and not being part of design patterns	JFreeChart	49	52
Classes not co-changing with DP	JFreeChart	681	801
Classes not co-changing with DP and not being part of design patterns	JFreeChart	410	662
Fisher's test in JFreeChart	0.543		
Odd-ratio in JFreeChart	0.9965146		
Fisher's test for classes not being part of design patterns in JFreeChart	0.02861		
Odd-ratio for classes not being part of design patterns in JFreeChart	1.520868		
Classes co-changing with DP	XercesJ	34	34
Classes co-changing with DP and not being part of design patterns	XercesJ	19	16
Classes not co-changing with DP	XercesJ	537	586
Classes not co-changing with DP and not being part of design patterns	XercesJ	301	486
Fisher's test in XercesJ	0.4106		
Odd-ratio in XercesJ	1.091165		
Fisher's test for classes not being part of design patterns in XercesJ	0.0435		
Odd-ratio for classes not being part of design patterns in XercesJ	1.915825		

anti-pattern classes. During the evolution of JFreeChart and XercesJ, classes participating in the Blob anti-pattern co-change the most with other classes.

Table 7 presents a contingency table for ArgoUML, JFreeChart, and XercesJ that reports the number of (1) classes co-changing with anti-pattern classes and identified as faulty; (2) classes co-changing with anti-pattern classes and identified as clean; (3) other classes identified as faulty; and (4) other classes identified as clean.

The results of the Fisher's exact tests and odds ratios when testing H_{RQ2_0} are significant. For all three systems, the p-values are less than 0.05 and the likelihood that a class co-changing with some anti-pattern class(es) experiences a fault is about two and half times higher than the likelihood that other classes experience faults.

We can also positively answer **RQ2** as follows: classes co-changing with anti-pattern classes are significantly more fault-prone than other classes.

Other Observations In ArgoUML, we detect some instances of the ClassDataShouldBePrivate, ComplexClass, and LongParameterList anti-patterns whose classes co-change with other classes. However, we do not detect any class belonging in these anti-patterns which are co-changing with other classes in JFreeChart and XercesJ. We do not detect, also, classes that are co-changing with LongMethod classes in XercesJ.

Finally, we observe that classes that are co-changing with anti-pattern classes are significantly more fault-prone than other classes with similar complexity, change history, and code size (see Table 14 in Section 5).

4.3 RQ3: Are Classes that have Static Relationships with Design Pattern Classes more Fault-prone than other Classes?

Table 8 reports, for ArgoUML, JFreeChart, and XercesJ, the numbers of (1) classes having static relationships with design patterns and identified as faulty; (2) classes having static relationships with design patterns and identified as clean; (3) classes without static relationships with design patterns and identified as faulty; and (4) classes without static relationships with design patterns and identified as clean. Considering all classes in the different analyzed systems, we also also separate the results in Table 6 between classes in design pattern instances and the rest of classes. We perform this distinction because it is difficult to tell whether our observations are caused by the fact that the classes themselves belong to design pattern instances or by the fact that they depend upon design pattern instances.

The results of the Fisher's exact tests and odds ratios when testing H_{RQ3_0} are not significant for all three systems for classes having static relationships with design patterns and not being part of design patterns. For the three systems, the p-values are more then 0.05: the likelihood that a class with static relationship(s) with design pattern class(es) experiences a fault is almost equal to the likelihood of other classes to experience faults.

We therefore answer **RQ3** as follows: classes having static relationships with design-pattern classes are not significantly more fault-prone than other classes.

Other Observations Design pattern classes can have static relationships with anti-pattern classes. We believe that developers attempt to overcome the negative impact of anti-patterns by relating their classes with specific design pattern classes. This method can reduce the impacts of anti-patterns on software systems so that, in the long term, developers could eliminate these anti-patterns. Future studies should analyze the benefits and disadvantages of such design choices on changes and faults.

For example, in XercesJ v1.0.4, class `org.apache.xerces.validators.common.XMLValidator.java` is an excessively complex class. The developers attempt to provide services for all possible uses of this class. In their attempt, they added a large number of interface signatures to meet all possible needs. The developers may not have a clear abstraction or purpose for `XMLValidator.java`, which is represented by the lack of focus in its interface. Thus, this class is a SwissArmyKnife. This anti-pattern is problematic because the complicated interface is difficult for other developers to understand and it obscures the class intent and use. It also makes it difficult to debug, document, and maintain the class.

We observe that this class has a use-relationship with class `org.apache.xerces.validators.dtd.DTDImporter.java`, which belongs to the Command design pattern. Using Command classes makes it easier to construct general components that delegate or execute method calls at a time of their choosing without the need to know the owner of the method or the method parameters. Thus, a developer may use

`XMLValidator.java` through the related Command, which encapsulates all the needed information, to overcome the SwissArmyKnife.

4.4 RQ4: Are Classes that Co-change with Design Pattern Classes more Fault-prone than other Classes?

Table 9 presents a contingency table for ArgoUML, JFreeChart, and XercesJ that reports the number of (1) classes co-changing with design patterns and identified as faulty; (2) classes co-changing with design patterns and identified as clean; (3) other classes identified as faulty; and (4) other classes identified as clean.

The results of the Fisher's exact tests and odds ratios when testing H_{RQ4_0} are significant for the set of classes co-changing with design patterns and not being part of design patterns. For all three systems, the p-values are less than 0.05 and the likelihood that a class co-changing with some design-pattern class(es) experiences a fault is about one and half times higher than that for other classes.

We consequently positively answer **RQ4** as follows: classes co-changing with design-pattern class(es) are more fault-prone than other classes.

Other Observations In the three systems, we observe that if co-change dependencies with design patterns are not properly maintained, they can lead to faults in the systems. For example, class `chart.Title.java` belongs to an Observer design pattern in JFreeChart. This class is co-changing with class `chart.event.ChartChangeEvent.java`. In the Bugzilla database of JFreeChart, the bug ID772[8] reported that "a problem occurs when JFreeChart is created with constructor parameter *title* set to null [...] As a result, no *ChartChangeEvent* is sent when title is modified". These results could help providing developers with lists of classes that they should carefully consider to ensure proper change propagation and to increase maintainability.

The findings of our analysis also indicate a relation between classes having dependencies with design patterns and their fault-proneness during the evolution of systems. This connection was expected from previous work, such as (Vokac 2004; Gatrell and Counsell 2011). However, design-pattern static relationships did not improve the precision of a fault prediction model based on complexity metrics and change metrics. We will discuss this point with more explanations in Section 5.

We detect co-change situations for all studied design patterns. Particularly, we observe that Composite and Observer classes co-changed with other classes more than classes in the other instances of the analyzed design patterns. The specifications and uses of these two design patterns promote co-change dependencies between subjects and observers and composite and components, respectively. The Observer design pattern defines an object, called the subject, that maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. Class `chart-.Title.java` belongs to an Observer design pattern in JFreeChart and co-changes with the class `chart.event.ChartChangeEvent.java`. The composite class maintains a collection of components. Typically, composite methods are implemented by iterating over that collection and invoking the appropriate method for each Component in the collection,

[8]http://sourceforge.net/p/jfreechart/bugs/772/

as in ArgoUML for `CrNodeInsideElement.java` (a composite class) that co-changes with `WizAssocComposite.java`

4.5 RQ5: What Types of Changes and Faults are Propagated by Static and Co-change Dependencies?

To determine the types of changes and faults that are propagated by dependencies on anti-patterns and design patterns, we mine software repositories, such as change logs and Bugzilla. We query the SVN of each studied system using `diff`, which is a tool to compare files and generate a list of differences. We record the lines of code that have been added, deleted, or changed, as reported by `diff`. We identified the type of change by analyzing this list of differences and the log-messages in the different commits manually. We also investigate the types of faults by analyzing the messages logs and the corresponding comments in Bugzilla. This method is simple to implement because it is easy to extract the deltas from a version control system, such SVN. Then, we report the types of changes and faults propagating through dependencies on anti-patterns and design patterns.

Table 10 reports for ArgoUML, JFreeChart, and XercesJ, the most frequent (more than 50 % of the cases) types of changes for classes having dependencies with anti-patterns. We observe that the majority of classes having dependencies with anti-patterns share structural changes. In fact, classes having dependencies with anti-patterns are usually subject to changes impacting their interfaces. For example, classes having static relationships with the LongMethod, ComplexClass, and LongParameterList anti-patterns undergo more refactoring operations than any other change operations, possibly because such classes are too large and automated tools must be applied to divide them and make them manageable. We also observe that classes having static relationships with the MessageChains and SwissArmyKnife anti-patterns, which are complex and provide or call too many services, are more likely to undergo structural changes, possibly to break down their complexity.

Table 11 reports for ArgoUML, JFreeChart, and XercesJ the more frequent (more than 50 % of classes) types of faults for classes having static and co-change dependencies with anti-patterns. The majority of classes having dependencies with anti-pattern classes and having faults share specific types of faults. For example: Blob, ComplexClass, and LongParameterList anti-patterns characterize classes with a higher number of complex methods

Table 10 Most frequent types of changes for classes having static and co-change dependencies with anti-patterns

	Types of changes	Descriptions	Examples
Blob, AntiSingleton	Refactoring (92 %)	Extract class	Moving part of code into a new class
ClassDataShouldBePrivate	Code addition (63 %)	Encapsulation	Forcing access fields with getter/setter
LongMethod, ComplexClass, LongParameterList	Refactoring (86 %)	Extract method	Breaking down code in smaller pieces
MessageChains, SwissArmyKnife	Refactoring (100 %)	Move method/field	Moving method to appropriate classes
RefusedParentBequest, SpeculativeGenerality	Refactoring (72 %)	Generalize type	Creating more general types

Table 11 Most frequent types of faults for classes having static and co-change dependencies with anti-patterns

	Types of faults	Descriptions	Examples
AntiSingleton	Interface fault (61 %)	Concurrency errors in critical sections	Race condition[a]
Blob, ComplexClass, LongParameterList	Logic fault (73 %)	High computational complexity	Infinite loops[b]
ClassDataShouldBePrivate, MessageChains	Data faults (52 %)	Uninitialized variable or a wrong type	Null pointer[c]
SwissArmyKnife	Interface fault (64 %)	Incorrect implementation	Incorrect API usage[d]
RefusedParentBequest, SpeculativeGenerality	Description fault (60 %)	Incorrect Design	Incorrect assumption[e]

[a]http://sourceforge.net/p/jfreechart/bugs/1049/

[b]http://sourceforge.net/p/jfreechart/bugs/1122/

[c]http://sourceforge.net/p/jfreechart/bugs/1094/

[d]http://sourceforge.net/p/jfreechart/bugs/950/

[e]http://sourceforge.net/p/jfreechart/bugs/332/

than the average classes. Thus, developers adding new features or fixing issues are more likely to touch these classes and their dependencies. Such maintenance activities increases the risk of these classes to be involved on logic faults, such as infinite loops and infinite recursion (more than 50 % of faults). This observation confirms Fowler and Brown's warnings about such classes (Brown et al. 1998).

Table 12 reports for ArgoUML, JFreeChart, and XercesJ, that the more frequent types of changes for classes having static relationships and co-change dependencies with the six analyzed design patterns was method addition/removal. Design patterns are solutions to recurring problems in software design that help in improving reusability, maintainability, comprehensibility, and robustness (Gamma et al. 1994). Thus, classes having dependencies to design patterns are less subject to structural changes, such as refactoring. For example, the Command design pattern interface defines usually only one method, typically `execute()`, and therefore can be expected to not be subject to structural changes.

Design patterns do not always improve the quality of systems, thus they should be used with caution. In **RQ3** we observe that static relationships with design patterns do not impact fault-proneness. In **RQ4** we observe that co-change dependencies have an impact on fault proneness. We report also that these co-change dependencies concern mainly the Composite

Table 12 Most frequent types of changes for classes having static and co-change dependencies with design patterns

	Changes types	Descriptions	Examples
Command, Composite, Factory method, Observer, Prototype, Decorator	Code addition (93 %)	Adding code to add features or implement requirements	Method addition

Table 13 Most frequent types of faults for classes having static and co-change dependencies with design patterns

	Fault types	Descriptions	Examples
Composite and Observer	Logic fault (54 %)	Programmer changes a class but forgets another class which uses the same code	Late propagation, missing co-change
Decorator, Factory method Prototype and Command	Syntax fault (96 %)	Using an invalid instruction or a wrong data type	Using and displaying null values as zero.

and Observer design patterns. Table 13 reports for ArgoUML, JFreeChart, and XercesJ, that the more frequent types of faults for classes having dependencies with the Composite and Observer design patterns are logic faults. We observe also that the more frequent types of faults for classes having dependencies with Decorator, Factory method, Prototype, and Command are syntax errors. For the rest of the design patterns, we found that faults on dependent classes are mostly related to the internal quality of these classes, such as invalid instructions or wrong data types.

Thus, we answer **RQ5** as follows: classes with dependencies on anti-pattern classes are more subject to logic faults and structural changes. Classes with dependencies on design pattern classes are more subject to code addition and syntax faults.

Other Observations Based on the finding of this study, it is impossible to categorize the types of faults and changes for classes having static relationships or co-change dependencies with LongMethod. Two observations could explain this fact. First, we observe the low number of classes participating in the instances of this anti-pattern. Second, this anti-pattern characterizes classes with long methods and using global variables for processing. Thus, classes having dependencies with such anti-pattern instances are often too few and have the structure of Lazy classes, i.e., with little reasons to change.

4.6 Improvement in Fault Prediction Models

Previous studies (Zimmermann and Nagappan 2008; Hassan 2009) showed that size, complexity, and historical metrics are good predictors of faults in systems. Thus, we decide to study whether static and/or co-change dependencies could provide additional useful information, complementary to these traditional fault-prediction metrics. Our investigation compares two models for predicting the presence or absence of faults in classes: (1) one using only static code metrics and (2) one using static code metrics and dependencies with anti-patterns or design patterns.

First, we use seven metrics to build a fault-prediction model using Support Vector Machines (SVM). There are various machine learning techniques available to build such models. We use Support Vector Machines because this technique has been widely used in literature and has shown good results (Ostrand et al. 2005; Moser et al. 2008).

The metrics are: (1) the number of lines of code per class; (2) the number of method calls of a class; (3) the depth of nested blocks in the methods in a class; (4) the number of parameters of the methods in a class; (5) the McCabe cyclomatic complexity of the methods in a class; (6) the number of fields of a class; and (7) the number of methods of a class. We chose these seven metrics because they have been successfully used in the past (Nagappan and Ball 2005) to predict faults.

Second, we added to the previous model metrics based on the static and co-change dependencies of classes with anti-pattern and design-pattern classes to investigate whether such dependencies could improve fault prediction. The metrics consist of four variables that must take an integer value *(0, 1, 2, ...)*. The first variable indicates the number of static relationships between a class and design-pattern classes. The second variable indicates the number of co-change dependencies between a class and design-pattern classes. The third variable indicates the number of static relationships between a class and anti-pattern classes. The fourth variable indicates the number of co-change dependencies between a class and anti-pattern classes. The models output the likelihood of a class to having one or more post-release faults.

We use the significance test based on a likelihood ratio test (Hosmer and Lemeshow 2000), commonly used for logistic regressions, to examine (the significance of) the difference between the performance of the two models when predicting faults. Table 14 reports the cases in which fault prediction was improved using anti-pattern or design pattern dependencies. It shows that dependencies with different anti-patterns and design patterns do not consistently bring improvements. If all anti-patterns and design patterns are considered, it is impossible to definitely exclude the possibility that there is no statistically significant differences in fault prediction. However, if we consider specific anti-patterns, we observe that using static and co-change dependencies improve the precision of the model. In fact, we observe that for anti-patterns that have a high complexity in term of LOC, such as Blob and ComplexClass, adding the analysis of static relationships improves the precision of the model. Similarly, when considering anti-patterns that have a high number of method calls, such as MessageChains and SwissArmyKnife, adding the analysis of co-change relationships improves the precision of the model.

Table 14 Difference in fault prediction between a model based on only static code metrics and a model based on static code metrics plus anti-pattern and design-pattern dependencies

	Static relationships	Co-change
# of AntiSingleton	No	No
# of Blob	Yes (+3 %)	Yes (+8 %)
# of ClassDataShouldBePrivate	No	No
# of ComplexClass	Yes (+4 %)	No
# of LongMethod	No	Yes (+6 %)
# of LongParameterList	No	No
# of MessageChains	No	Yes (+5 %)
# of RefusedParentBequest	No	Yes (+2 %)
# of SpeculativeGenerality	No	Yes (+2 %)
# of SwissArmyKnife	Yes (+2 %)	Yes (+3 %)
# of Command	No	No
# of Composite	No	No
# of Decorator	No	No
# of Factory method	No	No
# of Observer	No	No
# of Prototype	No	No

These results are encouraging because they show that even with a naive model, built using only static code metrics, considering anti-pattern dependencies improves the prediction of fault-prone classes. Thus, we conclude with this conservative and simplistic investigation that the knowledge about the propagation of faults through static and co-change dependencies is useful to improve the prediction of faults in classes and should be studied further in future work.

5 Discussion

We now discuss the results of our empirical study. For each research question, we report explanations about our findings and we compare them with related work. Finally, we discuss threats to validity of our study.

First of all, we observe with Tables 4 and 5 that different anti-patterns and design patterns have different proportions of static relationships with other classes in the analyzed systems. These differences are not surprising because these systems have been developed in three unrelated contexts, under different processes, and by different developers. Tables 4 and 5 highlight the importance of analyzing and reporting anti-pattern and design-pattern dependencies when assessing the quality of systems.

The fact that we cannot find a correlation between having static relationships with the six design patterns analyzed in this study and fault-proneness is not surprising. It confirms that the key benefit of using design patterns is their positive impact on software quality, such as reusability, flexibility, and maintainability (Gamma et al. 1994).

5.1 Dependencies Between Anti-patterns and Design Patterns

We observed that many dependencies from anti-pattern classes are with design-pattern classes. We believe that developers use design patterns, possibly unintentionally, as proven solutions to recurring design problems (Iacob 2011), i.e., when there is a proliferation of similar methods and/or the user-interface code becomes difficult to maintain.

In our previous study (Jaafar et al. 2013a), we observe that anti-patterns do have static relationships with design patterns, but that these relationships are temporary. Yet, anti-pattern classes participating in such relationships are more change-prone but less fault-prone than other anti-pattern classes. Therefore, it seems that developers sometimes use design patterns as a temporary fix for anti-patterns because both the anti-patterns and their dependencies on design pattern classes are removed later from the systems. The fix seems to be working because the fault-proneness of anti-patterns is reduced. Detecting and analyzing static relationships of anti-pattern classes is thus important from the points of view of both researchers and practitioners.

We bring evidence in our previous work (Jaafar et al. 2013a) that (1) anti-patterns *do* statically relate to some design patterns, and (2) some anti-patterns have more relationships with design pattern classes than (with) other classes. We confirm with this study that (3) these relationships indicate specific trends for the evolution of classes in term of fault-proneness and change-proneness. For example, we observe in our previous work (Jaafar et al. 2013a) that 50 % of static relationships among SpeculativeGenerality and design patterns in ArgoUML are with the Command design pattern. In XercesJ, we observe that 41 % of relationships among ClassDataShouldBePrivate was with the Command design pattern. In all releases, except ArgoUML 28.1, we observe a significant difference of proportions of changes among classes participating and not participating in a relationship between anti-patterns and design patterns. Indeed, the change-proneness of classes participating in static relationships between anti-patterns and design patterns are, in most cases, higher than those of other classes with anti-patterns and design patterns.

5.2 Impact of the Dependencies

Observing that classes having static relationships with anti-patterns undergo specific kind of changes or faults support claims in previous work (Khomh et al. 2012; Yamashita and Moonen 2013). Indeed, Khomh et al. (2012) show that classes participating in anti-patterns undergo more structural changes than others changes (e.g., changes in the method implementations). Thus, it is no surprising that classes sharing dependencies with anti-pattern classes also undergo structural changes. Similarly, Yamashita and Moonen (2013) show that specific maintenance problems, such as undesired behavior or unavailability of functionalities, are correlated with bad smells. We complement this previous work by also showing that classes having co-change dependencies with anti-pattern and design-pattern classes are more fault-prone than other classes.

We also confirm that class co-changes imply the existence of (hidden) dependencies between these two classes. If these dependencies are not properly maintained, they can introduce faults in a system (Zimmermann et al. 2007). We find that classes that co-change with anti-patterns and design patterns can be more fault-prone than other co-changing classes in ArgoUML, JFreechart, and XercesJ. Thus, by knowing the sets of classes that co-changed with anti-pattern and design-pattern classes, we could explain and possibly prevent faults. If co-change dependencies with anti-patterns and design patterns are not properly followed, they can lead to faults in the system. As an example, the class `GoClassToNavigableClass.java` belongs to a Blob anti-pattern in ArgoUML v0.26. Concretely, this class is co-changing with class `GoClassToAssociatedClass.java`. However, these two classes are not always maintained together. Indeed, when the developer makes some changes to `GoClassToNavigableClass.java`, she should also assess `GoClassToAssociatedClass.java` for change. In the Bugzilla database of ArgoUML, the bug ID5505[9] confirms that the two classes are related but were not maintained together, leading to a fault. Our approach could thus warn developers based on the system history and point out risky classes that are co-changing with anti-patterns and design patterns. In addition, with the availability of such information, a tester could decide to focus on classes having dependencies with anti-pattern and design-pattern classes, because she knows that such classes are likely to contain more faults.

5.3 Threats to Validity

We now discuss threats to the validity of our results (Yin 2002).

Construct validity threats concern the relation between theory and observation. In our context, they are mainly due to errors introduced in measurements. We are aware that the detection approaches used in our study include some subjective understanding of the definitions of anti-patterns and design patterns. However, we are interested in analyzing anti-patterns "as they are defined in DECOR" (Moha et al. 2010) and design patterns "as they are defined in DeMIMA" (Guéhéneuc and Antoniol 2008). Similarly, our approach to identify macro co-changes may have missed real and report wrong co-changes.

[9] http://ArgoUML.tigris.org/issues/show_bug.cgi?id=5505

Yet, the precision and recall values of the anti-pattern, design-pattern, and co-changes detection approaches are concerns that we agree to accept. Moha et al. (2010) reported that the current DECOR detection algorithms for anti-patterns ensure 100 % recall and have a precision greater than 31 % in the worst case, with an average precision greater than 60 %. Guéhéneuc and Antoniol (2008) reported that DeMIMA can detect design patterns with a recall of 100 % and a precision greater than 34 %. While, for the detection of relationships among classes, DeMIMA guarantees 100 % recall and precision by definitions of the relationships (Guéhéneuc and Albin-Amiot 2004). Macocha approach for macro co-change ensures 96 % recall and has a precision greater than 85 % (Jaafar et al. 2011). Future investigations aimed at assessing the extent to which the choice of the detection approaches impact our results are needed.

Classes participating in an anti-pattern instance (respectively, a design pattern instance) can have dependencies (static relationships and/or co-change dependencies) with classes participating in other anti-pattern instances (or design pattern instances). Thus, the tests reported in this paper cover classes that have a dependency with an anti-pattern instance (or design patterns instance), regardless of the fact that these classes could belong to other patterns. Nevertheless, we present in Section 4 the result of our analysis of the impact of dependencies, for classes participating in anti-patterns or design patterns, and other classes separately. If the analyzed class has dependencies with more than one pattern/anti-pattern, it will be avoided by the analysis to minimize the noise of the interaction effects. In this context, Yamashita and Moonen (2013) reported that such interactions affect the maintenance and the code quality. For example, they revealed how smells that were co-located in the same artifact interacted with each other, and affected maintainability.

We computed the fault-proneness of a class by relating fault reports and commits to the class. Fault-fixing changes are documented in fault reports that describe different types of problems in a system. We match faults/issues to changes using their IDs and their dates in the change-log files and in the fault reports. We check independent changes that were accidentally combined in the same commit. We also manually investigated the code to be sure that the fault fixes documented in the commit message were related to the class changes reported in the SVN/CVS log files.

Internal validity threats concern the extent to which a causal conclusion based on a study is warranted. First of all, the study presented in this paper is an exploratory study. Thus, we do not claim causation (Yinn 2002), but relate the presence of anti-pattern and design pattern dependencies with the occurrences of changes and faults. Nevertheless, we tried to explain by looking at specific changes, commit notes, and change histories why some of these dependencies could have been the cause of changes/issues/faults. We are also aware that, on the one hand, anti-patterns can be dependent on each other and relied on the logistic regression model-building procedure to select the subset of non-correlated anti-patterns. On the other hand, having a fault is a temporary property, whereas being involved in an anti-pattern or design pattern is a rather somewhat long-term, persistent property. In previous work (Jaafar et al. 2013a), we analyzed the volatility of anti-pattern and design pattern dependencies in the same software systems (ArgoUML, JFreeChar, and XercesJ) and we concluded that they continued to exist in all versions of these systems. Thus, there will be times when an anti-pattern having dependencies with a class will have no fault and times when it will have faults. In this study, and as in previous work (Khomh et al. 2012) analyzing change- and fault-proneness, we declared that a class is a faulty class if it was involved in at least one fault-fixing change. Indeed, we considered changes and faults that occurred after the structural/co-change dependencies are detected. For instance, we cannot definitively claim that the co-changes are the result of a defect fix or are causing the defect. But we reported the presence of these dependencies with the next occurrences of changes and faults.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate the assumptions of the statistical test that we used, i.e., the Fisher's exact test, which is a non-parametric test.

Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details to replicate our study. Moreover, the source code repositories of ArgoUML, JFreeChart, and XercesJ are publicly available as well as the anti-pattern and design pattern detection approaches used in this study. The analysis process is described in detail in Section 2. All the data used in this study are available on-line.[10]

External validity threats concern the possibility of generalizing our observations. First, although we performed our study on three different, real systems belonging to different domains and with different sizes and histories, we cannot assert that our results and observations are generalizable to any other systems. All the analyzed systems were in Java and open-source, which may also reduce the generalizability of our findings. In the future, we plan to analyze more systems, written in different programming languages, to draw more general conclusions.

Second, we used particular, yet representative, sets of anti-patterns and design patterns. Different anti-patterns and design patterns could have lead to different results, which are part of our future work. In addition, the list of metrics used in our study is by no means complete. Therefore, using other metrics may yield different results. However, we believe that the same approach can be applied to any list of metrics. The odds ratio and p-value thresholds used in our study were chosen because they proved to be successful in previous work (Khomh et al. 2012).

6 Related Work

During the past years, different approaches have been developed to address the problem of detecting design patterns, specifying anti-patterns, and studying their impact on change- and fault-proneness.

6.1 Anti-patterns Definition and Detection

Code smells and anti-patterns both describe occurring software problems. There are large overlaps between many definitions of anti-patterns and code smells and they both span abstraction levels that go from attributes associated to inner workings of the class to more design/micro-architectural-related attributes (Webster 1995; Brown et al. 1998). Concretely, code smells give warnings to software developers that the source code has some problems, while anti-patterns provide software managers, architects, designers, and developers a common vocabulary for recognizing possible sources of higher-level problems in advance. We noted that some of the definitions of code smells are equivalent to many of the anti-patterns. For example: Blob can be interpreted as God Class, Long method can be interpreted as God Method, etc. In this paper, we consider the specification of 10 anti-patterns described in Brown et al. (1998) and follow several previous work (Moha et al. 2010; Khomh et al. 2012), etc.

[10]http://www.ptidej.net/download/experiments/emse14a/

Vokac (2004) analyzed the corrective maintenance of a large commercial program, comparing the defect rates of classes participating in design motifs against those that did not. He found that the Observer and Singleton motifs are correlated with larger classes; classes playing roles in Factory Method were more compact, less coupled, and less defect prone than other classes; and, no clear tendency existed for Template Method. Their approach showed correlation between some design patterns and smells like LargeClass but did not report an exhaustive investigation of possible correlations between these patterns and anti-patterns. Pietrzak and Walter (2006) defined and analyzed the different relationships that exist among smells and provided tips about how they could be exploited to alleviate the detection of anti-patterns. The authors performed an experiment to show that the use of knowledge about identified smells in Jakarta Tomcat code supports the detection process. They found examples of several smell dependencies, including aggregate relationships. The certainty factor for those relations in that code suggested the existence of correlation among the dependent smells and applicability of this approach to anti-patterns detection.

The first book on “anti-patterns” in object-oriented development was written in 1995 by Webster (1995). In this book, the author reported that an anti-pattern describes a frequently used solution to a problem that generates ineffective or decidedly negative consequences. Brown et al. (1998) presented 40 anti-patterns, which are often described in terms of lower-level code smells. These books provide in-depth views on heuristics, code smells, and anti-patterns aimed at a wide academic audience. They are the basis of all the approaches to detect anti-patterns.

The study presented in this paper relies on the anti-pattern detection approach DECOR (Moha et al. 2010). However several other approaches have been proposed in the past. For example, Van Emden and Moonen (2002) developed the JCosmo tool, which parsed source code into an abstract model (similar to the Famix meta-model). JCosmo used rules to detect the presence of smells and anti-patterns. It could visualize the code layout and display anti-pattern locations to help developers assess code quality and perform refactorings.

Marinescu et al. developed a set of detection strategies to detect anti-patterns based on metrics (Ratiu et al. 2004). They defined history measurements which summarize the evolution of the suspect parts of code. Then, they showed that the detection of God Classes and Data Classes can become more accurate by using historical information of the suspected flawed structures.

Settas et al. explored the ways in which an anti-pattern ontology, a representation of anti-pattern specification in the form of a set of concepts, can be enhanced using Bayesian networks (Settas et al. 2012). Their approach allowed developers to quantify the existence of an anti-pattern using Bayesian networks, based on probabilistic knowledge contained in an anti-pattern ontology.

The Integrated Platform for Software Modeling and Analysis (iPlasma) described in previous work (Lanza and Marinescu 2006) can be used for anti-pattern detection. This platform calculates metrics from C++ or Java source code and applies rules to detect anti-patterns. The rules combine the metrics and are used to find code fragments that exceed thresholds.

We share with all the above authors the idea that anti-pattern detection is an interesting approach to assess code quality, in particular to study whether the existence of anti-patterns and their relationships make the source code more difficult to maintain. This previous work significantly contributed to the specification and detection of anti-patterns. The approach used in this study, DECOR (Moha et al. 2010), builds on this previous work to offer a method to specify and automatically detect anti-patterns. It has appropriate performance, precision, and recall for our study. In addition, DECOR can be applied on any object-oriented system through the use of the PADL (Guéhéneuc and Antoniol 2008) meta-model and POM framework (Guéhéneuc et al. 2004). PADL describes the structure of systems and a subset of their behavior, i.e., classes and their relationships. POM is a PADL-based framework that implements more than 60 structural metrics.

6.2 Design Pattern Definition and Detection

The first book on “design patterns” in object-oriented development was written in 1996 by Gamma et al. (1994). Since this book, several workshops and conferences have emerged to propose new patterns. Many papers have been published studying the use and impact of design patterns.

The study presented in this paper relies on the design-pattern detection approach DeMIMA (Guéhéneuc and Antoniol 2008). However, several other approaches have been proposed in the past. For example, one of the first papers about detecting design patterns was written by Krämer et al. (1996) in 1996. It introduced an approach for detecting design information directly from C++ header files. This information was stored in a repository. The design patterns were expressed as Prolog rules to query the repository and detect five structural design patterns: Adapter, Bridge, Composite, Decorator, and Proxy.

Several other approaches have been proposed, using different representations of both the design patterns and the systems in which to detect their occurrences and various algorithms with different trade-offs between simplicity, performance, precision, and recall. Algorithms used for the first time to detect design pattern include logic programming (Wuyts 1998), constraint programming (Quilici et al. 1997), queries (Kullbach and Winter 1999), fuzzy networks (Jahnke et al. 1997), graph transformations (Antoniol et al. 1998). Some dedicated algorithms have also been introduced, e.g., (Alencar et al. 1995; Brown 1996; Lethbridge 1998; Tatsubori and Chiba 1998).

Recently, an approach based on similarity scoring has also been proposed by Tsantalis et al. (2006), which provides an efficient means to compute the similarity between the graph of a design pattern and the graph of a system to identify classes potentially participating in the design pattern. The authors proposed an approach to exploit the fact that each design pattern resides in one or more inheritance hierarchies because most design patterns involve at least one abstract class/interface and its descendants. Consequently, the system is partitioned into clusters of hierarchies (pairs of communicating hierarchies), so that the similarity algorithm is applied to smaller subsystems rather than to the entire system.

We concur with this previous work on the importance of identifying design patterns in systems as a means to aid program comprehension, to assess code quality as well as to study their impact. The approach used in this study, DeMIMA (Guéhéneuc and Antoniol 2008), follows previous work on using constraint programming to automatically identify design patterns and some of their variants with reasonable performance, precision, and recall. Indeed, DeMIMA makes it possible to recover two kinds of design choices from source code: idioms pertaining to the relationships among classes and design motifs characterizing the organization of the classes. DeMIMA depends on a set of definitions for unidirectional binary class relationships. The formalizations define the relationships in terms of four language-independent properties that are derivable from static and dynamic analyses of systems: exclusivity, type of message receiver, lifetime, and multiplicity. DeMIMA keeps track of data and links to identify and ensure the traceability of these relationships. DeMIMA also uses explanation-based constraint programming to identify microarchitectures similar to design motifs. This technique makes it possible to identify microarchitectures similar to a model of a design motif without having to describe all possible variants explicitly. We also use DeMIMA to detect motifs's relationships. In fact, DeMIMA distinguishes use, association, aggregation, and composition relationships because such relationships exist in most notations used to model systems, for example, in UML.

6.3 Anti-patterns and Design Patterns Static Relationships

Binkley et al. (2008) defined the Dependence anti-patterns which indicated potential problems for ongoing software maintenance and evolution. They showed how these anti-patterns can be identified using techniques for dependence analysis and visualization. While it is hard to define what a "bad" dependence structure looks like, we believe that it is comparatively easy to identify dependence between anti-patterns and others classes. In this paper, we investigate the impact of such dependencies in terms of fault proneness.

Oliveto et al. (2011) reported that analyzing smell relationships could help to refactor the code and, in particular, to remove the Feature Envy smell. Indeed, this bad smell can be considered as the most common symptom related to problems with class coupling and cohesion. Our work differs in the studied objects (the anti-patterns and design patterns), the analyzed parameters (static relationships and co-changes), and its goals (detecting the impact on fault-proneness and the types of changes).

Marinescu and Marinescu (2011) reported that if a class makes use of a class that reveals design flaws, that class is more likely to exhibit faults. Thus, when a developer is aware of a class revealing design flaws within a system, she should also monitor the clients of this class because they are likely to also exhibit faults. They considered four design flaws called Identity Disharmonies and that a class use an Identity Disharmonies if that class calls at least one method from a flawed class.

Our work differs from this previous work in that we analyze different types of dependencies with both anti-patterns and design patterns. Thus, we claim that our study is the first detailed analysis of the impact of different relationships (including calls, i.e., use relationship), co-change dependencies, and fault-proneness. We also analyzed six design patterns and 10 anti-patterns in comparison to four Identity disharmonies.

Vokac (2004) analyzed the corrective maintenance of a large commercial system, comparing the fault rates of design pattern classes with others. Their approach showed a correlation between some design patterns and smells such as LargeClass but did not report an exhaustive investigation of possible correlations between these design patterns and anti-patterns.

Yamashita and Moonen (2012) reported maintainability factors that are important from the software maintainer's perspective, and provided an overview of the capability of code smell definitions to evaluate the overall maintainability of a system. They also found that code smell interactions occurred across coupled artifacts, with comparable negative effects such as same artifact co-location. Pietrzak and Walter (2006) defined and analyzed the different relationships that exist among smells and provided suggestions to exploit these relationships to improve the detection of anti-patterns. They proposed six coarse relations: plain support, mutual support, rejection aggregate support, transitive support, and inclusion. Indeed, the authors found examples of several smell dependencies, including simple, aggregate and transitive support and rejection relation. The authors noticed that the certainty factor for those relations suggested the existence of correlation among the dependent smells.

Rather than focusing on the relationships among code smells and anti-patterns to increase the accuracy of anti-pattern detection, our study focused on analyzing anti-pattern and design pattern dependencies to understand their impact on fault-proneness.

6.4 Co-change Dependencies

Bouktif et al. (2006) defined the general concept of change patterns and described one such pattern, Synchrony, that highlights co-changing groups of classes. Their approach used a sliding window algorithm (Zimmermann et al. 2004) to identify occurrences of the Synchrony change pattern. This pattern models the fact that a change to one artifact may imply a large number of changes to various other artifacts. Macro co-changes are another kind of change pattern of interest to developers (Jaafar et al. 2013a, b).

Ying et al. (2004) and Zimmermann et al. (2004) applied association rules to identify co-changing files, i.e., occurrences of the Asynchrony change pattern. They used past co-changed files to recommend source code files potentially relevant to a change request. An association-rule algorithm extracts frequently co-changing files of a transaction into sets that are regarded as change patterns to guide future changes. Such an algorithm uses co-change history in CVS and avoids the source code dependency parsing process. Other approaches to detect co-changing files exist (Gall et al. 1998; Zimmermann et al. 2004; Aversano et al. 2007; Gîrba et al. 2007). They all are intrinsically limited in their definition of co-change. They cannot express change patterns in long time intervals and–or performed by different developers.

We introduced the novel change patterns of *macro co-changes* (MCCs) and *dephase macro co-changes* (DMCCs) (Jaafar et al. 2011), inspired from co-changes and using the concept of change periods. A MCC describes a set of classes that always change together in the same periods of time (of duration much greater than 200 ms). A DMCC describes a set of classes that always change together with some shift in time in their periods of change. We proposed an approach, Macocha (Jaafar et al. 2013a, b), to mine software repositories (CVS and SVN) and identify (dephase) macro co-changing classes. We showed that Macocha has a better precision and recall for co-changes detection than an approach based on association rules. We used external information provided by bugs reports, mailing lists, and requirement descriptions to show that detected MCCs and DMCCs explain real, important evolution phenomena. In this paper, we use Macocha to mine software repositories and identify classes that are co-changing with anti-patterns or design patterns.

Aversano et al. (2007) presented results from an empirical study analyzing the evolution of design patterns in three open-source programs (JHotDraw, ArgoUML, and Eclipse-JDT). The study analyzed the frequency of the modification of design pattern classes, the type of changes that they undergo, and co-changing classes. Results suggested that developers should carefully consider design pattern usage to support crucial features because the design pattern classes will likely undergo frequent changes and be involved in large maintenance activities. While Aversano et al. focused on design patterns, our study analyzed classes that co-changed with anti-patterns and design patterns.

Khomh et al. (2012) investigated the impact of anti-patterns on classes in object-oriented systems by studying the relation between the presence of anti-patterns and the change- and fault-proneness of the classes. The authors showed that in 50 out of 54 releases of the four analysed systems, classes participating in anti-patterns are more change and fault-prone than others. In this paper, we showed that detecting classes that are co-changing with anti-pattern classes help to identify which classes are more likely to be fault prone. Similarly, Aversano et al. (2009) investigated the impact of design patterns on crosscutting code spread across classes using the patterns as well as its fault proneness.

Similar to this previous work, we investigate fault proneness of classes to analyze the impact of anti-pattern and design pattern dependencies. We aim to understand if the negative effects of anti-patterns can propagate to other classes through static and co-change dependencies.

6.5 Fault-proneness

The most studied approach for fault prediction is to relate software faults with size and complexity metrics (McCabe 1976; Halstead 1977). Chidamber and Kemerer (1994) proposed a suite of object-oriented design metrics that has been supported by several theoretical and empirical studies (Basili et al. 1996; Subramanyam and Krishnan 2003). Results show that (1) the more complex the code is, the more faults exist in it and (2) size is one of the best indicators for fault proneness. However, other measures can provide interesting, complementary information to further improve our understanding of fault-proneness and its propagation. Thus, Hassan and Holt (2005) proposed heuristics to analyze fault proneness. They found that recently modified and fixed classes were the most fault-prone.

D'Ambros et al. (2009) reported that there was a correlation between change coupling and defects that is higher than the one observed with complexity metrics. Further, defects with a high severity seem to exhibit a correlation with change coupling which, in some classes, is higher than the change rate of the classes. They also used change coupling information in fault prediction models based on complexity metrics.

Marcus et al. (2008) used a cohesion metric based on Latent semantic indexing (LSI), an indexing and retrieval method, to relate the terms and concepts contained in an unstructured collection of source code. LSI is based on the principle that words that are used in the same contexts tend to have similar meanings. The authors used LSI for fault prediction and reported that structural and semantic cohesion impacts the understandability and readability of the source code and, hence, its fault-proneness.

Ostrand et al. (2005), Bernstein et al. (2007), and Neuhaus et al. (2007) predict faults in systems using historical change and fault data. Moser et al. (2008) used code and historical metrics (e.g., code churn, past faults, and refactorings, etc.) to predict the presence/absence of faults in files of Eclipse.

We share with all this previous work its interest in the fault-proneness of classes and we reuse the traditional algorithms to assign faults to classes in the literature by mining software repositories and issues tracking systems and relating the issues-ID in the tracking systems with the changes in the repositories.

Previous Work Capitulation These previous works raised the awareness of the community towards the impact of anti-patterns and design patterns on software development and maintenance activities. In this paper, we build on these previous works and analyze the existence and the impact of anti-pattern and design-pattern dependencies. We studied the negative effects of anti-patterns and design patterns and their propagation through co-change and static dependencies. We also performed a qualitative study to investigate the types of changes and faults impacting classes having dependencies with anti-pattern and design pattern classes.

7 Conclusion

Our conjecture was that classes having static and co-change dependencies with anti-pattern and design-pattern classes could be involved in changes and faults more often than other classes because the changes and faults could propagate from anti-pattern and design-pattern classes to other classes through their dependencies.

We conducted a quantitative study to answer four research questions and we reported that:

- Classes having static or co-change dependencies with anti-pattern classes have significantly more faults than other classes.
- Classes having co-change dependencies with design-pattern classes also have significantly more faults than other classes, but not than classes having static relationships with design pattern classes.
- Structural changes are more likely to occur in classes having dependencies with anti-pattern classes than other classes.
- Code addition changes are more likely to occur in classes having dependencies with design pattern classes than in classes.
- Specific types of faults are more prevalent with certain anti-patterns and design patterns: for examples, Blob and Complex Class propagate mostly logic faults.

We confirmed that our conjecture is valid within the context of this study, and thus, researchers can use these results (1) to improve fault prediction models, as we briefly illustrated in Section 5, (2) to understand the evolution of software systems, and (3) to detect the types of faults and changes that possibly affect classes having such dependencies. Developers could use these observations to assign maintenance tasks and focus testing efforts on classes having dependencies with anti-pattern and design pattern classes.

As limitation of this study, we used a particular, yet representative, set of design patterns and anti-patterns. Different design patterns and anti-patterns could have lead to different results. In addition, the list of metrics used in our study is not complete. Therefore, using other metrics may yield different results. However, we believe that the same approach can be tested on any list of patterns and metrics. Thus, future work includes (1) replicating our study on other systems to assess the generalizability of its results, (2) studying change log files, mailing lists, and issue reports to seek evidence of cause-effect relationships between the presence of anti-patterns or design patterns and issues, and (3) analyzing the evolution of such dependencies among different releases of a system.

Acknowledgments This research was partially supported by FQRNT, NSERC, the Canada Research Chair in Software Patterns and Patterns of Software, and the Canada Research Chair in Software Dependability.

References

Alencar PSC, Cowan DD, Morales-Germán D, Lichtner KJ, Pereira de Lucena CJ, Nova LC (1995) A formal approach to design pattern definition and application. Tech Rep CS-95-29, Computer Systems Group, University of Waterloo

Antoniol G, Fiutem R, Cristoforetti L (1998) Design pattern recovery in object-oriented software. In: Tilley S, Visaggio G (eds) Proceedings of the 6^{th} International Workshop on Program Comprehension, pp 153–160. IEEE Computer Society Press

Aversano L, Canfora G, Cerulo L, Del Grosso C, Di Penta M (2007) An empirical study on the evolution of design patterns. In: Foundations of Software Engineering. ACM Press, New York, NY, pp 385–394

Aversano L, Cerulo L, Di Penta M (2009) Relationship between design patterns defects and crosscutting concern scattering degree: an empirical study. Institu Eng Technol 3(5):395–409

Basili VR, Briand LC, Melo WL (1996) A validation of object-oriented design metrics as quality indicators. IEEE Trans Soft Eng 22(10):751–761

Bernstein A, Ekanayake J, Pinzger M (2007) Improving defect prediction using temporal features and non linear models, In: Ninth International Workshop on Principles of Software Evolution, pp 11–18. ACM

Binkley D, Gold N, Harman M, Li Z, Mahdavi K, Wegener J (2008) Dependence anti patterns. In: 4^{th} International ERCIM Workshop on Software Evolution and Evolvability, pp 25–34

Bouktif S, Antoniol G, Merlo E, Neteler M (2006) A plugin based architecture for software maintenance. Tech. Rep. EPM-RT-2006-03, Department of Computer Science École Polytechnique de Montréal

Brown K (1996) Design reverse-engineering and automated design pattern detection in Smalltalk. Tech. Rep. TR-96-07, Department of Computer Science, University of Illinois at Urbana-Champaign

Brown WJ, Malveau RC, Brown WH, McCormick III H. W., Mowbray TJ (1998) Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, 1^{st} edn. Wiley

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493

D'Ambros M, Lanza M, Robbes R (2009) On the relationship between change coupling and software defects In: Proceedings of the 16th Working Conference on Reverse Engineering. IEEE Computer Society, Washington, DC, pp 135–144

Dasarathy B (1991) Nearest Neighbor ({NN}) Norms:{NN} Pattern Classification Techniques. IEEE Computer Society Press, Washington, DC

Gall H, Hajek K, Jazayeri M (1998) Detection of logical coupling based on product release history In: The International Conference on Software Maintenance. IEEE Computer Society, Washington, DC, pp 190–200

Gamma E, Helm R, Johnson R, Vlissides J (1994) Design Patterns – Elements of Reusable Object-Oriented Software, 1^{st} edn. Addison-Wesley

Gatrell M, Counsell S (2011) Design patterns and fault-proneness a study of commercial C# software. In: Research Challenges in Information Science (RCIS) pp 1–8. IEEE

Gerlec C, Hericko M (2012) Analyzing structural software changes: A case study. In: The 5th Balkan Conference in Informatics, pp 117–120

Gîrba T, Ducasse S, Kuhn A, Marinescu R, Daniel R (2007) Using concept analysis to detect co-change patterns In: International Workshop on Principles of Software Evolution. ACM, New York, NY, pp 83–89

Guéhéneuc YG, Albin-Amiot H (2004) Recovering binary class relationships: Putting icing on the UML cake. In: Schmidt DC (ed) Proceedings of the 19^{th} Conference on Object-Oriented Programming, Systems, Languages, and Applications pp 301–314. ACM Press

Guéhéneuc YG, Antoniol G (2008) DeMIMA: A multi-layered framework for design pattern identification. IEEE Trans Softw Eng 34(5):667–684

Guéhéneuc YG, Sahraoui H, Zaidi F (2004) Fingerprinting design patterns In: Proceedings of the 11th Working Conference on Reverse Engineering. IEEE Computer Society, Washington, DC, pp 172–181

Halstead MH (1977) Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc., New York, NY

Hassan AE (2009) Predicting faults using the complexity of code changes In: Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, Washington, DC, pp 78–88

Hassan AE, Holt RC (2005) The top ten list: Dynamic fault prediction In: Proceedings of the 21st IEEE International Conference on Software Maintenance, pp 263–272. IEEE Computer Society

Hosmer DW, Lemeshow S (2000) Applied logistic regression (Wiley Series in probability and statistics) Wiley-Interscience Publication

Iacob C. (2011) A design pattern mining method for interaction design. In: The 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pp 217–222. ACM

Jaafar F, Guéhéneuc YG, Hamel S, Antoniol G (2011) An exploratory study of macro co-changes. In: Proceedings of the 18^{th} Working Conference on Reverse Engineering, pp 325–334

Jaafar F, Guéhéneuc YG, Hamel S, Khomh F (2013) Analysing anti-patterns static relationships with design patterns. Electronic Communications of the European Association of Software Science and Technology 59:1–26

Jaafar F., Guéhéneuc YG, Hamel S, Khomh F (2013) Mining the relationship between anti-patterns dependencies and fault-proneness. In: Proceedings of the 20^{th} Working Conference on Reverse Engineering:351–360

Jahnke JH, Schäfer W, Zündorf A (1997) Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. In: Jazayeri M (ed) Proceedings of the 6^{th} European Software Engineering Conference pp 193–210. ACM Press

Khomh F, Penta MD, Guéhéneuc YG, Antoniol G (2012) An exploratory study of the impact of antipatterns on class change- and fault-proneness. Empirical Software Engineering, pp 243–275

Krämer C, Prechelt L (1996) Design recovery by automated search for structural design patterns in object-oriented software. In: Wills LM, Baxter I (eds) Proceedings of the 3^{rd} Working Conference on Reverse Engineering. IEEE Computer Society Press, pp 208–215

Kullbach B, Winter A (1999) Querying as an enabling technology in software reengineering. In: Nesi P, Verhoef C (eds) Proceedings of the 3^{rd} Conference on Software Maintenance and Reengineering. IEEE Computer Society Press, pp 42–50

Lanza M, Marinescu R (2006) Object-Oriented Metrics in Practice. Springer-Verlag

Lethbridge NAT (1998) Extracting concepts from file names; a new file clustering criterion In: Proceedings of the International Conference on Software Engineering, pp 84–93

Marcus A, Poshyvanyk D, Ferenc R (2008) Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE Transactions On Software Engineering, pp 287–300

Marinescu R, Marinescu C (2011) Are the clients of flawed classes (also) defect prone? In: Proceedings of the IEEE 11th International Working Conference on Source Code Analysis and Manipulation. IEEE Computer Society, Washington, DC, pp 65–74

McCabe TJ (1976) A complexity measure In: Proceedings of the 2Nd International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, pp 407–417

Moha N, Guéhéneuc YG, Duchien L, Le Meur AF (2010) DECOR: A method for the specification and detection of code and design smells. Trans Softw Eng:20–36

Moser R, Pedrycz W, Succi G (2008) A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction In: The 30th International Conference on Software Engineering. ACM, New York, NY, pp 181–190

Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density In: The 27th International Conference on Software Engineering, ACM

Neuhaus S, Zimmermann T, Holler C, Zeller A (2007) Predicting vulnerable software components. In: The 14th Conference on Computer and Communications Security. ACM, pp 529–540

Oliveto R, Gethers M, Bavota G, Poshyvanyk D, De Lucia A (2011) Identifying method friendships to remove the feature envy bad smell (nier track) In: Proceedings of the 33rd International Conference on Software Engineering. ACM, New York, NY, pp 820–823

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

Pedersen T (1996) Fishing for exactness In: Proceedings of the South-Central SAS Users Group Conference cmp-lg/9608010, pp 188–200

Pietrzak B, Walter B (2006) Leveraging code smell detection with inter-smell relations. Extreme Programming and Agile Processes in Software Engineering, pp 75–84

Quilici A, Yang Q, Woods S (1997) Applying plan recognition algorithms to program understanding. J Autom Softw Eng 5(3):347–372

Ratiu D, Ducasse S, Gîrba T, Marinescu R (2004) Using history information to improve design flaws detection. In: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering. IEEE Computer Society, pp 223–233

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

Rothman KJ, Lanes S, Sacks ST (2004) The reporting odds ratio and its advantages over the proportional reporting ratio. Pharmacoepidemiol Drug Safety 13(8):519–523

Settas D, Cerone A, Fenz S (2012) Enhancing ontology-based antipattern detection using bayesian networks. Expert Syst Appl 39(10):9041–9053

Sheskin DJ (2007) Handbook of Parametric and Nonparametric Statistical Procedures. Chapman & Hall/CRC

Sliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? SIGSOFT Softw Eng Notes 30(4):1–5

Subramanyam R, Krishnan MS (2003) Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. IEEE Trans Softw Eng 29(4):297–310

Tatsubori M, Chiba S (1998) Programming support of design patterns with compile-time reflection. In: Fabre JC, Chiba S (eds) Proceedings of the 1^{st} OOPSLA workshop on Reflective Programming in C++ and Java. Center for Computational Physics, University of Tsukuba, pp 56–60. UTCCP Report 98-4

Tsantalis N, Chatzigeorgiou A, Stephanides G, Halkidis S (2006) Design pattern detection using similarity scoring. IEEE Trans Softw Eng 32(11):896–909

Van Emden E, Moonen L (2002) Java quality assurance by detecting code smells In: The 9th Working Conference on Reverse Engineering IEEE Computer, pp 97–107. Society Press

Vokac M (2004) Defect frequency and design patterns: An empirical study of industrial code. IEEE Transaction on Software Engineering, pp 904–917

Webster BF (1995) Pitfalls of Object Oriented Development, 1^{st} edn. M & T Books

Wuyts R (1998) Declarative reasoning about the structure of object-oriented systems. Proceedings of Technology of Object-Oriented Systems (TOOLS'98), pp 112–124

Yamashita A, Moonen L (2012) Do code smells reflect important maintainability aspects? In: Software Maintenance (ICSM), 2012 28th IEEE International Conference on. IEEE, pp 306–315

Yamashita A, Moonen L (2013) To what extent can maintenance problems be predicted by code smell detection? - an empirical study. Info Softw Technol 55(12):2223–2242

Yin RK (2002) Case Study Research: Design and Methods - Third Edition. SAGE Publications, London

Ying ATT, Murphy GC, Ng R, Chu-Carroll MC (2004) Predicting source code changes by mining change history, vol 30, pp 574–586

Yinn RK (2002) Case study research: design and methods. SAGE, London, England

Zimmermann T, Nagappan N (2008) Predicting defects using network analysis on dependency graphs. In: Proceedings of the 30th International Conference on Software Engineering, pp 531–540. ACM

Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse In: Proceedings of the Third International Workshop on Predictor Models in Software Engineering. IEEE Computer Society, Washington, DC, pp 9–16

Zimmermann T, Weisgerber P, Diehl S, Zeller A (2004) Mining version histories to guide software changes. In: Proceedings of the 26th International Conference on Software Engineering, pp 563–572. IEEE Computer Society

Fehmi Jaafar is a postdoctoral fellow at Queen's Reliable Software Technology GROUP in Queen's University. He received his Ph.D. in 2013 from the Department of Computers Sciences and Operations Research of the Université de Montréal in Quebec, Canada. He is a member of IEEE. His current research projects focus on software maintenance and evolution, software quality, and software reliability. These projects are sponsored by a number of research funding agencies and industry. He has published several papers in international conferences and journals, including WCRE, CSMR, WCRE, JSP, and QSIC. He had several years of industry experience in Canada and Africa in the areas of software engineering, web 2.0, and business process management.

Yann-Gaël Guéhéneuc is full professor at the Department of computer and software engineering of École Polytechnique de Montréal where he leads the Ptidej team on evaluating and enhancing the quality of object-oriented programs by promoting the use of patterns, at the language-, design-, or architectural-levels. He is IEEE Senior Member since 2010. In 2009, he was awarded the NSERC Research Chair Tier II on Software Patterns and Patterns of Software. He holds a Ph.D. in software engineering from University of Nantes, France (under Professor Pierre Cointe's supervision) since 2003 and an Engineering Diploma from École des Mines of Nantes since 1998. His Ph.D. thesis was funded by Object Technology International, Inc. (now IBM OTI Labs.), where he worked in 1999 and 2000. His research interests are program understanding and program quality during development and maintenance, in particular through the use and the identification of recurring patterns. He was the first to use explanationbased constraint programming in the context of software engineering to identify occurrences of patterns. He is interested also in empirical software engineering; he uses eye-trackers to understand and to develop theories about program comprehension. He has published many papers in international conferences and journals, including IEEE TSE, Springer EMSE, ACM/IEEE ICSE, and IEEE ICSM. He was the program co-chair and general chair of several events, including IEEE ICSM'13, APSEC'14, and IEEE SANER'14. He visited KAIST, Seoul National University, and Yonsei University, Korea, during his sabbatical year in 2013–2014.

Sylvie Hamel received her PhD degree in mathematics at Université du Québec à Montréal in 2002. After a postdoc in complexity theory at McGill University she obtained a position in the Computer Science Department of the Université de Montréal in 2003, where she is now an associate professor. Her research interests lies at the intersection of mathematics, bioinformatics, and computer science. She worked on several projects in algebraic and word combinatorics, and bioinformatics.

Foutse Khomh is an assistant professor at the École Polytechnique de Montréal, where he heads the SWAT Lab on software analytics and cloud engineering research (http://swat.polymtl.ca/). He received a Ph.D in Software Engineering from the University of Montreal in 2010. His research interests include software maintenance and evolution, cloud engineering, service-centric software engineering, empirical software engineering, and software analytic. He has published several papers in international conferences and journals, including ICSM, MSR, WCRE, ICWS, JSS, JSP, and EMSE. He has served on the program committees of several international conferences including ICSM, WCRE, MSR, ICPC, SCAM, and has reviewed for top international journals such as SQJ, EMSE, TSE and TOSEM. He is program co-chair of the Workshops track at WCRE 2013, program chair of the Tool track at SCAM 2013, program chair for Satellite Events at SANER 2015, and program co-chair for SCAM 2015. He is one of the organizers of the RELENG workshop series (http://releng.polymtl.ca) and guest editor for a special issue on Release Engineering in the IEEE Software magazine.

Dr. Mohammad Zulkernine is a Canada Research Chair in Software Dependability and an Associate Professor at the School of Computing, Queen's University, Canada. He leads the Queen's Reliable Software Technology (QRST) research group. His current research projects focus on software reliability and security that are sponsored by a number of provincial and federal research funding agencies and industry. Dr. Zulkernine was one of the program co-chairs of SSIRI 11, COMPSAC 12, and HASE 14. He is a senior member of the IEEE and the ACM, and a licensed professional engineer in the province of Ontario, Canada.