Anti-pattern Mutations and Fault-proneness

Fehmi Jaafar[1], Foutse Khomh[2], Yann-Gaël Guéhéneuc[2], and Mohammad Zulkernine[1]
[1] QRST Group, School of Computing, Queen's University, Ontario, Canada
[2] SWAT, PTIDEJ, École Polytechnique de Montréal, QC, Canada
E-Mails: {jaafarfe, mzulker}@cs.queensu.ca, {foutse.khomh, yann-gael.gueheneuc}@polymtl.ca

***Abstract*— Software evolution and development are continuous activities that have a never-ending cycle. While developers commit changes on a software system to fix bugs or to implement new requirements, they sometimes introduce anti-patterns, which are bad solutions to recurring design problems in the system. Many previous studies have shown that these anti-patterns have negative effects on code quality, in particular fault-proneness. However, it is not clear if and how anti-patterns evolve and which evolutionary behaviors are more fault-prone. This paper presents results from an empirical study aimed at understanding the evolution of anti-patterns in 27 releases of three open-source software systems: ArgoUML, Mylyn, and Rhino. Specifically, the study analyzes the mutations of anti-patterns, the changes that they undergo, and the relation between anti-pattern evolution behaviors and fault-proneness. Results show that (1) anti-patterns mutate from one type of anti-patterns to another, (2) structural changes are behind these mutations, and (3) some mutations are more risky in terms of fault-proneness.**

***Index Terms*—Anti-patterns; Faults proneness; Markov Chain**

I. Introduction

Code quality analysis aims to identify the most important or the more risky parts of a software system, using for example static code analysis [1] [2]. Such analysis can specify and detect structural patterns in software systems that are signs of poor design decisions, like code smells and anti-patterns. Indeed, code smells [3] are low-level or local problems that are usually symptoms of more global design smells such as anti-patterns. An anti-pattern [4] is a literary form that describes a bad solution to a recurring design problem that has negative effects on code quality [5]. In object-oriented systems, an anti-pattern can pertain to one class or a set of classes.

Previous work [6], [7] reported that anti-patterns render the maintenance of systems more difficult, *e.g.*, classes participating in anti-patterns are more change and fault-prone than others. However, there has been little effort to identify and understand the evolution of anti-patterns and their potential impacts on code quality, especially on fault proneness. We argue that it is important to obtain and disseminate the information about the risks in relation with anti-pattern evolutions. Concretely, assuming that all anti-pattern classes are considered to have the same likelihood for fault-proneness is not realistic. Many previous work showed that some anti-patterns are more fault prone than others [7]. For example, we observed in a previous work that different anti-patterns have different probabilities of fault-proneness and that faults occurred to some entities just after their evolutions, *e.g.*, the class `GoClassToNavigableClass.java` in ArgoUML [8]. Indeed, knowledge about anti-pattern mutations and their impacts on fault-proneness should facilitate the development of strategies to mitigate these risks. We define an anti-pattern mutation as the set of changes occurred into an anti-pattern and that result could be the appearance of a different anti-pattern, the preservation of the previous anti-pattern, or the deletion of the anti-pattern symptoms.

Unlike previous work [9], [10] that analyzed the evolution of design problems or our preceding paper [8] that reported the impact of anti-pattern dependencies on fault proneness, the purpose of this paper is not only to study how anti-patterns have changed over time but also (1) to model the behavior of these changes, (2) to detect the different states through which the anti-patterns are mutated during their evolutions, (3) and the impact of such mutations on fault-proneness.

Our main motivation is to obtain insight into the actual risks of anti-pattern mutation and to use the results from this exploratory study as the basis for more in-depth studies. Concretely, we explore the use of Markov chains [11] to capture evolving states of anti-patterns and the transitions that model anti-pattern mutations during the evolution of a system as well as their fault-proneness at each stage of their evolution. Such modeling of anti-pattern mutations provides developers with the knowledge about the risk of system code decay and describe anti-pattern genealogies.

By examining anti-pattern mutations and tracing their evolution, we will gain the ability to not only identify existing anti-patterns and their possible impact on fault-proneness, but also even predict future anti-pattern states and, therefore, allow developers to decide if they want to reach these states. Using anti-pattern information extracted from three open-source systems, we answer the following research questions:

- **RQ1:** *Are anti-patterns persistent in software systems?* Our motivation is to analyze the persistence/mutation of anti-patterns among different releases. We investigate the mutation of anti-patterns using Markov chains [11] and examine their evolution behaviors.
- **RQ2:** *What kind of changes lead to anti-patterns mutation?* In this research question, we examine how anti-patterns are mutated and whether there are some specific changes that lead their mutations.
- **RQ3:** *Are anti-patterns equally fault-prone at every mutation?* A common knowledge is that some anti-patterns are more fault-prone than others. Our motivation in this research question is to verify if some anti-patterns become more fault-prone after mutations.

The results of **RQ1** show the high probabilities for each anti-pattern to mutate to another anti-pattern. We extract changes behind anti-patterns mutations in **RQ2** and found that, in the majority of anti-patterns, structural changes, such as adding a huge number of attributes and long methods, dominate the change types. Results of **RQ3** show that the mutation of some anti-patterns, such as RefusedParentBequests into MessageChains, are the most risky evolution phenomena in terms of fault-proneness.

The remainder of this paper is organized as follows. Section II describes the empirical study. Section III presents our methodology. Section IV presents the study results, while Section V analyzes the results along with threats to their validity. Then, Section VI relates our study with previous work. Finally, Section VII concludes the study and outlines some avenues for future work.

II. Study Design

This section describes the design of our empirical study. In particular, we present more details about the approach and the analyzed subject systems. Then, we discuss our research questions and the analysis method.

The design of our study follows the Goal-Question-Metric (GQM) approach [12]: the *goal* of our study is to investigate the fault proneness of anti-patterns through all the stages of their evolution. The *quality focus* is the identification of risky anti-pattern mutations in terms of fault-proneness. The *perspective* is that of researchers, interested in studying anti-pattern mutations and their impacts. The results may also be of interest to developers, who perform development or maintenance activities, in deciding which classes are most at risk for faults and in prioritizing the code for testing. The *context* of this study is three software systems, ArgoUML, Mylyn, and Rhino. In this paper, we report the results of analysis of nine different releases of each system.

A. Analyzed Systems

ArgoUML[1] is a UML-modeling application that provides a set of views and tools to model systems using UML diagrams. The project started in January 1998 and is still active. It has over 3.1M LOC and 18k commits in its software repository. We consider an interval of observation ranging from January 1998 to February 2006. Mylyn[2] is an open-source system written in Java to implement a Task-Focused Interface. We analyze nine releases of this system by considering an interval of observation ranging from June 2007 to September 2008. Rhino[3] is an open source JavaScript engine developed entirely in Java to convert JavaScript scripts into classes. In this paper, we study the evolution of anti-patterns in Rhino's code by analyzing nine releases of this system in an interval of observation ranging from May 1999 to July 2006.

[1]http://argouml.tigris.org/

[2]http://www.eclipse.org/mylyn/

[3]https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino

B. Analyzed Anti-patterns

We focus on 12 anti-patterns from Brown *et al.* [13]. We choose these anti-patterns because they are representative of problems with data, complexity, size, and the features provided by classes [7]. We also use these anti-patterns because they have been used and analyzed in previous work [7] to validate our results by comparing them with previous data [14]. The definitions and specifications of the anti-patterns are beyond the scope of this paper and are available in [15]. The list of anti-patterns considered in this study is as follows:

- AntiSingleton (AS): a class that provides mutable variables, which could be used as global variables.
- God Class (GC): a class that is too large and not cohesive enough, that monopolizes most of the processing, takes most of the decisions, and is associated to data classes. It is also known as a Blob.
- ClassDataShouldBePrivate (CDSBP): a class that exposes its fields, thus violating the principle of encapsulation.
- ComplexClass (CC): a class that has (at least) one large method, in terms of cyclomatic complexity and LOCs.
- LargeClass (LGC): a class that has (at least) one large method, in terms of LOCs.
- LazyClass (LZC): a class that has few fields and methods.
- LongMethod (LM): a class that has a method that is overly long, in terms of LOCs.
- LongParameterList (LPL): a class that has (at least) one method with a long list of parameters with respect to the average number of parameters per methods.
- MessageChain (MC): a class that uses a long chain of method invocations to realize one of its functionality.
- RefusedParentBequest (RPB): a class that redefines inherited method using empty bodies.
- SpeculativeGenerality (SG): a class that is defined as abstract but that has very few children, which do not make use of its methods.
- SwissArmyKnife (SAKF): a class whose methods can be divided in disjoint sets of many methods, thus providing many different unrelated functionalities.

III. Methodology

This section describes the steps of our data analysis process. As shown in Figure 1, we perform the identification of anti-patterns using the detection model presented in [14]. Then, we use Markov chains [11] to built a probabilistic model of anti-pattern mutations. Finally, we apply the heuristics by Sliwersky *et al.* [16] to identify fault fix locations and investigate the fault proneness of anti-pattern mutations.

A. Detecting Anti-patterns

We use the Defect DEtection for CORrection Approach DECOR [14] to specify and detect anti-patterns in each release of the three systems. DECOR proposes a domain-specific language to specify and generate design defect detection algorithms automatically. A domain-specific language offers greater flexibility than ad hoc algorithms [14] because the domain experts, *i.e.*, the software engineers, can specify and

TABLE I
DESCRIPTIVE STATISTICS OF THE OBJECT SYSTEMS

System	Release	Date of publication	Number of files	Number of classes	Number of line of codes
ArgoUML	0.10.1	October 2002	777	881	146,041
	0.12	August 2003	850	984	163,423
	0.14	December 2003	1,133	1,294	197,942
	0.15.6	May 2004	1,122	1,271	199,195
	0.16	June 2004	1,122	1,271	200,051
	0.17.5	February 2005	1,161	2,276	217,190
	0.18.1	April 2005	1,226	1,360	240,762
	0.19.8	November 2005	1,364	1,538	280,996
	0.20	February 2006	1,386	1,559	285,284
Mylyn	2.0.0	June 2007	1589	1668	207,436
	2.1.0	September 2007	1695	1737	209,214
	2.2.0	December 2007	1736	1809	211,052
	2.3.0	February 2008	1805	1939	214,869
	2.3.1	March 2008	1836	1935	230,144
	2.3.2	April 2008	1839	1937	236,219
	3.0.0	June 2008	2005	2216	244,053
	3.0.1	July 2008	2002	2217	256,553
	3.0.2	September 2008	2319	2233	263,759
Rhino	1.4R3	May 1999	82	97	30,748
	1.5R1	September 2000	112	132	45,699
	1.5R2	July 2001	136	177	56,452
	1.5R3	January 2002	134	176	56,685
	1.5R4	February 2003	141	202	60,353
	1.5R5	February 2004	139	207	63,166
	1.6R1	November 2004	154	226	74,529
	1.6R2	September 2005	155	229	74,755
	1.6R3	July 2006	156	223	76,592

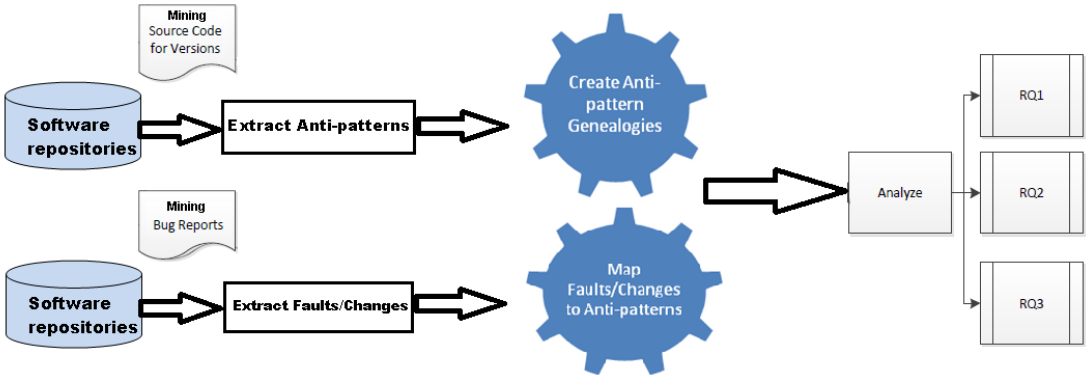

Fig. 1. Overview of our approach to study the evolutionary behavior of anti-patterns

modify the detection rules manually. Indeed, software engineers used high-level abstractions, taking into account the context, environment, and characteristics of the analyzed systems. DECOR can be applied on any object-oriented system through the use of the Pattern and Abstract-level Description Language, PADL meta-model [17], and POM (Primitives, Operators, Metrics) framework [18]. PADL is a meta-model to describe object-oriented systems [17] at different levels of abstractions[4]. POM is a PADL-based framework that implements more than 60 metrics.

Moha *et al.* [14] reported that DECOR's current anti-patterns' detection algorithms achieve 100% recall and have an average precision greater than 60%.

B. Analyzing Anti-pattern Mutation

We determine the different mutation of anti-patterns in order to illustrate the evolution of anti-patterns over time. During their evolution, anti-patterns are sometimes mutated from one type into another (*e.g.*, a LongParameterList can become a Blob). To the best of our knowledge, we present in this paper the first exploration of the use of Markov chains [11] to model anti-pattern mutations. A Markov chain refers to the sequence of random variables that a process moves through, with the Markov property that defines serial dependencies between periods in sequence. Markov chains are suitable for describing sequences of linked events: what happens next release depends on the previous releases of the system. In our study, we use Markov chains to capture the sequence of anti-pattern mutations during the evolutionary histories of software systems. Specifically, for each release, we detect occurrences of anti-patterns. Then, we map the anti-patterns across all the releases of the system and identify anti-pattern mutations. Using information about mutated and non-mutated anti-patterns, we compute the likelihood for each type of anti-pattern to become another type and the risk of such mutation

[4]http://wiki.ptidej.net/doku.php?id=padl

on term of fault-proneness. We extract and categorize the source code changes that caused anti-patterns mutations by mining version-control systems (Concurrent Versions System named CVS[5] and Apache Subversion System named SVN[6]), as described in the next step.

C. Analyzing the Fault Proneness

We compute the fault-proneness of a class by relating fault reports (extracted from Bugzilla) and commits to the class. We mine version-control systems (CVS and SVN), to detect changes committed for each class. A commit contains several attributes such as the changed files names that contain the class, the dates of the changes, and the names of the developers who committed the changes. In fact, fault fixing changes are documented in textual reports that describe different kinds of problems in a program. These textual reports are stored in issue tracking systems like Bugzilla or Jira. We use a Perl script to parse the SVN/CVS change logs of the systems to identify all fault fixing commits. This Perl script, which was used successfully in previous work [7], implements the heuristics by Sliwersky *et al.* [16][7]. From each fault fixing commit, we extract the list of files that were changed to fix the fault. This list of files contain classes that were changed to fix the faults.

IV. Study Results

Table I summarizes the data obtained by analyzing ArgoUML, Mylyn and Rhino. We validated anti-pattern occurrences manually and checked external sources of information provided by bugs reports and textual descriptions of change commits to confirm and to discuss some typical examples.

A. Are anti-patterns persistent in software systems?

1) Motivation: A good understanding of the evolutionary behavior of anti-patterns is important if we want to maintain them efficiently. Many tools exist to automatically assess entities and check if they present anti-patterns (*e.g.*, [19], [20], [21], and [10]). However, to the best of our knowledge, developers are still unable to predict/anticipate the evolution of an anti-pattern. All anti-patterns are not equally risky as shown by previous work [7]; different anti-patterns have different "bad" properties that make them prone to faults or hard to understand/maintain. Knowing the evolutionary behavior of an anti-pattern could help prevent its mutation into a most fault-prone anti-pattern. Studying the evolutionary behavior of anti-patterns can also help identify instabilities in software systems. Indeed, frequent mutations of anti-patterns could indicate an instability in the system. Consequently, a good knowledge of the evolutionary behavior of anti-patterns will not only help development teams track the most risky anti-patterns, but also prevent some occurrences of anti-patterns. It could also help them spot areas of instability in the code.

2) Method: For each system, we identify anti-pattern occurrences in the analyzed releases using DECOR. To build the anti-patterns genealogies, we map anti-patterns from the detected anti-patterns across releases. We define anti-patterns genealogies as the set of states that identify all possible anti-pattern mutations among the releases. These genealogies are structures comprised of singular nodes and modeled by Markov chain. A Markov chain is a sequence of random variables $X1, X2, X3, ..., Xn$ with the Markov property:

$$\Pr(X_{n+1} = x | X_1 = x_1, X_2 = x_2, \ldots, X_n = x_n) = \Pr(X_{n+1} = x | X_n = x_n)$$

The possible values of Xi form a countable set S called the state space of the chain. In our study, the state space is the 12 anti-patterns. Markov chains are described by a directed graph, as showed in Figure 2, where the edges are labeled by the probabilities of mutation from one state to the other. The arcs of the chain define transitions between anti-pattern states.

3) Results: Figure 2 shows a Markov chain that models the anti-pattern mutations of the AntiSingleton to other forms of anti-patterns in the different analyzed release of ArgoUML. The sum of all the mutation from AntiSingleton state to other states in the Markov chain is equal to one.

Actually, we observed that not all anti-pattern occurrences undergo mutations, some of them remain stable during the system evolution or could be maintained without being mutated to another anti-pattern. Indeed, the classes of these anti-patterns are refactored to become a part of non-anti-pattern codes or remain in the same state without any change.

In general, anti-patterns do not remain persistent in ArgoUML, Mylyn, and Rhino. The majority of anti-patterns are mutated during their evolution as shown by Table II. In this table, we add an additional state (NoAP) to model the set of classes that do not belong to the anti-pattern set after their mutations. Indeed, more than the half of anti-pattern occurrences mutated among the different releases of ArgoUML. For example, in ArgoUML, 41.5% of AntiSingleton occurrences retain their states from a release to another. 57.3% are mutated to another form of anti-patterns and around 1% are corrected and do not belong to anti-pattern set. We found similar results in Mylyn and Rhino (the complete results of these systems are available on the Web[8])

> *More than the half of anti-pattern occurrences mutated among the different releases of the three analyzed systems.*

B. What kind of changes lead to anti-patterns mutation?

1) Motivation: Because all anti-patterns are not equally fault-prone [7] and anti-pattern mutations occur frequently, understanding the causes of anti-pattern mutations could allow us to predict when an anti-pattern class becomes more fault-prone. In fact, change is an essential feature of the evolutionary history of anti-patterns. Therefore, recognizing the changes

[5]http://cvs.nongnu.org/

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

[7] Rhino and Mylyn bugs were validated manually by Marc Eaddy. We thank him for making his data on faults freely available.

[8]http://www.ptidej.net/downloads/replications/qsic14/

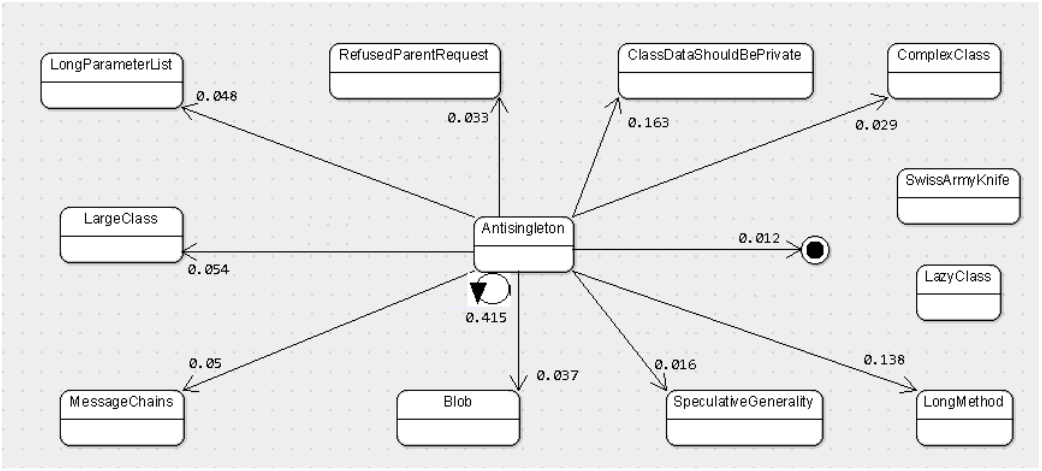

Fig. 2. Anti-singleton mutation among the different releases of ArgoUML

TABLE II
MUTATION PROBABILITIES OF ANTI-PATTERNS IN ARGOUML

	AS	GC	CDSBP	CC	LGC	LZC	LM	LPL	MC	RPB	SG	SAKF	NoAP
AS	**0.415**	0.037	0.163	0.029	0.054	0.000	0.138	0.048	0.050	0.033	0.016	0	0.012
GC	0.010	**0.269**	0.055	0.149	0.193	0.000	0.146	0.053	0.086	0.008	0.011	0	0.015
CDSBP	0.059	0.071	**0.378**	0.054	0.071	0.001	0.144	0.018	0.1	0.069	0.008	0	0.022
CC	0.010	0.192	0.054	**0.235**	0.237	0.000	0.124	0.041	0.077	0.001	0.008	0	0.014
LGC	0.014	0.193	0.055	0.182	**0.265**	0.000	0.131	0.063	0.067	0.002	0.010	0	0.013
LZC	0.000	0.000	0.009	0.000	0.000	**0.705**	0.009	0.000	0.073	0.170	0.018	0	0.013
LM	0.021	0.078	0.058	0.051	0.069	0.002	**0.473**	0.061	0.118	0.033	0.008	0	0.023
LPL	0.022	0.091	0.023	0.055	0.106	0.000	0.182	**0.394**	0.082	0.020	0.002	0	0.016
MC	0.012	0.068	0.062	0.046	0.053	0.007	0.177	0.039	**0.428**	0.053	0.008	0	0.041
RPB	0.011	0.007	0.060	0.000	0.000	0.038	0.073	0.014	0.076	**0.610**	0.064	0	0.043
SG	0.019	0.049	0.027	0.029	0.044	0.014	0.061	0.004	0.039	0.233	**0.445**	0	0.029
SAKF	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0	0.000

that the anti-pattern classes have gone through their life time is essential to understand how and why a system has reached its current state. In the same context, we believe that spotting the causes of anti-patterns mutation is a key knowledge for maintenance activities, because it allows us to detect the critical parts of the system that represent the starting point for a maintenance process.

2) Method: To determine the cause of anti-pattern mutations detected in **RQ1**, we query the SVN/CVS of each system, on the date of publication of the release considered in this study, using *diff*, a tool that compares files and generates a list of differences between them. This method is simple to implement, because it is easy to extract the deltas from a versioning control systems, such as CVS or SVN. Next, we record lines of code that have been added, deleted, or changed, as reported by diff. Then, we verify if changed lines of code concern attributes and methods. Finally, we report the kind of changes responsible for anti-pattern mutations. In this study, we are interested in structural changes [22], which are changes that transforms an object-oriented element.

3) Results: Table III summarizes the most frequent kind of changes detected in anti-pattern mutations among the three analyzed systems. For lack of space we cannot present all the result in this paper, we are planning to extend our work in another paper to discuss more results but the complete results could be deduced from our data available on the Web[9] and the code source of the analyzed open source system publicly available. We study whether classes participating in anti-patterns mutation undergo more (or less) structural changes (addition/removal/change of/to attributes, addition/removal of methods, or changes to the methods signatures) than non-structural changes, *i.e.,* addind/deleting/changing lines without changing the structure of the class. We found in the the three analyzed systems that anti-patterns classes are usually subject to structural changes impacting their interface and, thus, possibly their states. For example, we observe that MessageChains, which are complex anti-patterns and provide or call too many classes, are more likely to undergo structural changes as shown in Table III. In Rhino, the developers of the version *1.5.R2* added methods and attributes to `Importer-TopLevel.java` (a MessageChains anti-pattern's class) but they forgot to verify that all the data used in this class are private. Thus, this class mutated to a new anti-pattern's state: ClassDataShouldBePrivate. The result of such changes is often the creation of new anti-pattern occurrences or the refactoring of an anti-pattern, by breaking code apart into more logical pieces. This observation confirms previous results by Khomh *et al.* [7] who found that structural changes are more frequent on classes participating to anti-patterns.

[9]http://www.ptidej.net/downloads/replications/qsic14/

TABLE III
THE MOST FREQUENT ANTI-PATTERNS MUTATIONS IN ARGOUML, MYLYN AND RHINO

Transition	Proportion	The kind of changes
All anti-patterns $->$ AntiSingleton	72%	adding global variable or adding a long list of variables
All anti-patterns $->$ LargeClass	68%	adding long methods
All anti-patterns $->$ ClassDataShouldBePrivate	66%	adding attributes and methods violating encapsulation

We conclude that structural changes occur more often on mutated anti-patterns than other changes.

C. Are anti-patterns equally fault-prone at every mutation?

1) Motivation: In **RQ1**, we observed that anti-patterns frequently mutate in other types of anti-patterns during the evolution of the system. In this research question, we aim to determine whether these transitions are risky in terms of fault-proneness. On the one hand, we will determine if mutated anti-pattern are more fault-prone than non-mutated anti-pattern. On the second hand, we will detect the most risky mutation among anti-patterns. Knowing the location of anti-patterns and the fault-proneness of anti-patterns mutations could help development teams better target classes that should be reviewed carefully in order to ensure a good maintainability of the software system and reduce the risk for faults.

2) Method: For each system, we identify if anti-patterns undergo faults after their mutation. We use Fisher's exact test [23] to check whether the difference in fault proneness between mutated anti-patterns and non-mutated anti-patterns is significant. We also compute the odds ratio [23] that indicates the likelihood for a fault to occur. The odds ratio is defined as the ratio of the odds p of an event occurring in one sample, *i.e.*, the odds that mutated anti-pattern classes experience a fault in the future (experimental group), to the odds q of the same event occurring in the other sample, *i.e.*, the odds that other anti-pattern classes experience faults (control group): $OR = \frac{p/(1-p)}{q/(1-q)}$. An odds ratio of 1 indicates that the event (*i.e.*, a fault) is equally likely in both samples. 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.

We verify the null hypothesis that we state as follows:

- H_{RQ3_0}*: There is no statistically significant difference between proportions of faults carried by mutated anti-pattern classes and non-mutated anti-pattern classes in ArgoUML, Mylyn, and Rhino.*

If we reject the null hypothesis H_{RQ3_0}, then we explain the rejection as follows:

- H_{RQ3_1}*: There is a statistically significant difference between proportions of faults carried by mutated anti-pattern classes and non-mutated anti-pattern classes.*

Then, we create Markov chains that describe the fault-proneness of all specific mutations (*e.g.*, from Blob to ComplexClass).

3) Results: Table IV presents a contingency table for ArgoUML, Mylyn and Rhino that reports the number of (1) mutated anti-pattern cases that are identified as fault-prone; (2) mutated anti-pattern cases that are identified as clean; (3) non-mutated anti-pattern cases that are identified as fault-prone; and, (4) mutated non anti-pattern cases that are identified as non anti-patterns. The result of Fisher's exact test and odds ratios when testing H_{RQ3_0} are significant. The p-value is less than 0.05 and the odds ratio for fault-prone non-mutated anti-patterns is six times higher than for fault-prone mutated classes.

TABLE IV
CONTINGENCY TABLE AND FISHER TEST RESULTS IN ARGOUML, MYLYN, AND RHINO FOR ANTI-PATTERN MUTATION AND NON MUTATION CORRELATED WITH FAULT BETWEEN TWO SUCCESSIVE RELEASES

	Faulty	Clean
Anti-pattern mutations cases in ArgoUML	2211	25186
Anti-pattern non mutation cases in ArgoUML	514	1578
Fisher's test for ArgoUML	2.2e-16	
Odds-ratio for ArgoUML	3.710	
Anti-pattern mutations cases in Mylyn	70	14332
Anti-pattern non mutation cases in Mylyn	31	353
Fisher's test for Mylyn	2.2e-16	
Odds-ratio for Mylyn	17.967	
Anti-pattern mutation cases in Rhino	346	2881
Anti-pattern non mutation cases in Rhino	197	65
Fisher's test for Rhino	2.2e-16	
Odds-ratio for Rhino	25.190	
The Sum of anti-pattern mutation cases	2627	42399
The Sum of anti-pattern non-mutation cases	742	1996
Fisher's test for the three systems	2.2e-16	
Odds-ratio for the three systems	6	

Thus, we can answer to **RQ3** as follows: we showed that mutated anti-patterns are significantly less fault-prone than non-mutated anti-patterns. We can explain this finding by relating it with previous observations in **RQ2** and [7]. In fact, according to [7], anti-pattern classes are more change prone. The non-mutated anti-patterns were not mutated because they evolved without showing a new anti-pattern, and not because they didn't evolve at all, *i.e.*, new changes happened to anti-patterns but without changing their structures. Indeed, structural changes that may cause anti-pattern mutations are less risky than the fact of not maintaining anti-patterns or performing other simples change on them. Thus, the result imply that anti-patterns with structural changes tend to be less buggy and that anti-patterns evolved without structural changes tend to be more buggy.

In addition to this statistical analysis of the relation between mutated anti-pattern and fault-proneness, we compute the fault-proneness of all specific mutations. For example, Figure 3 presents the fault-proneness of anti-pattern in the different analyzed releases of Mylyn modeled by a Markov chain. We notice that the Markov chain shows that non-mutated anti-patterns (anti-patterns that preserve the same symptoms among different releases and are not subject to structural

changes) are more fault-prone than mutated anti-patterns. We observe this fact in all analyzed anti-patterns and especially for RefusedParentBequest and LongMethod.

Non-mutated anti-patterns are more fault-prone than mutated anti-patterns.

V. Discussions

We now discuss some observation from our results. Then, we discuss the threats to validity of our study.

First of all, we notice that different software systems can have different anti-pattern evolution behaviors. This observation is not surprising because these systems have been developed in three unrelated contexts, under different processes. Thus, their entities undergo different kind of changes and by consequence, their anti-patterns follow different mutations.

Second, some of the studied anti-patterns are never mutated, they remain in the same state all through their evolution history. This was the case for `org.mozilla.javascript.Node`. This class was detected as a LargeClass in all the analyzed releases of Rhino. Indeed, this class declare long methods with no parameters, and using global variables for processing. Thus, `org.mozilla.javascript.Node` is difficult to reuse and to maintain while it does not take advantage of object-orientation mechanisms such as polymorphism and inheritance. This exploratory study provides, within the limits of its validity, evidence that classes participating in anti-pattern mutations are more less fault-prone than anti-pattern classes not participating in such mutation. Further investigation, devoted to mine change logs, mailing lists, and issue reports, is desirable to seek evidence of causeeffect relationships between the mutation of anti-patternsor the need to remove themand the class change- and fault-proneness. For example, a fault may be in a part of the code that didn't really play a role in the original or the new anti-pattern or both. It just happened to be in a class that belongs to both anti-patterns.

We found also that LargeClass anti-patterns, were the most fault-prone entities. In fact, this anti-pattern defines a class that "knows too much or does too much" and centralizes many functionalities. A LargeClass precisely corresponds to a large controller class that depends on data stored in surrounding data classes. Thus, LargeClass entities, such as `org.mozilla.javascript.Context` and `org.mozilla.javascript.Parser` in Rhino, are hard to change and to modify, and when they are modified, they could present a huge risk of fault.

One of the most risky anti-pattern mutation was the modification of some anti-patterns to the state of MessageChain. This anti-pattern is a bad structure from a dependencies point of view. Ideally, objects should only interact with a small number of direct collaborators (a design principle known as the Law of Demeter). The MessageChain anti-pattern arises when a particular class is highly coupled to other classes in chain-like delegations. We think that many anti-patterns could mutate to MessageChain while developers try to fix them. It was the case in Mylyn for classes such as `TaskEditorAttachmentPart` and `TaskEditorCommentPart`.

We also observed cases of anti-pattern mutations into design patterns. For example, we found that some Blobs are mutated into Abstract Factory, Adapter, Observer, and Prototype. Theoretically, anti-patterns and design patterns are unrelated by their contradiction definition. However, developers often use design patterns to refractor or wrap anti-patterns [10], [24]. After such refactoring activities, previous anti-pattern entities become a part of design patterns. In future work, we plan to investigate anti-patterns to design patterns and design patterns to anti-patterns mutations more extensively.

Markov chains prove to be a good model for several reasons. First of all, a Markov model is a discrete-time stochastic process that describes the state of a system at successive points in time. Thus, Markov chains allow us to provide both researchers and developers a simplified but accurate picture of the anti-pattern mutations, precisely because they are simple and we believed suitable for the purpose of visualization. Second, it is a tractable stochastic process and a good basis for statistical testing. There is a rich body of theory, analytical results, and mutation models. In this paper, we present an exploratory study on anti-pattern mutations and we use Markov chain to model them. Our future work also include performing an exhaustive stochastic study based on our actual results to predict anti-pattern mutations and fault-proneness in future releases.

A. Threats to Validity

We now discuss in details the threats to the validity of our results, following the guidelines provided in [25].

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 technique used includes some subjective understanding of the definitions of the anti-patterns. For this reason, the precision of the anti-patterns detection is a concern that we agree to accept. We preprocessed the inconsistent anti-patterns to eliminate false positives. This preprocessing reduces the chances that we could answer our research questions wrongly. In case that anti-pattern specifications are variants of the specification used by DECOR, some anti-patterns may be missed during the detection phase. Although the sample of detected anti-patterns can be considered large enough to claim our conclusions, further investigations are desirable to further verify our findings. Another concern that can impact the result is the renaming of files among different releases of a system. Indeed, during the evolution of the system, several classes can be renamed or divided into two files with different names. To mitigate such cases, we identified class renamings and class structural changes using a previous approach ADvISE [26] [27]. ADvISE identifies class renamings using the structure-based and the text-based metrics, *e.g.*, their common methods, attribute types, and relationships, which assess the similarities between original and renamed classes. Last but not least, the fact that anti-pattern is a multiclass problem, we can find a

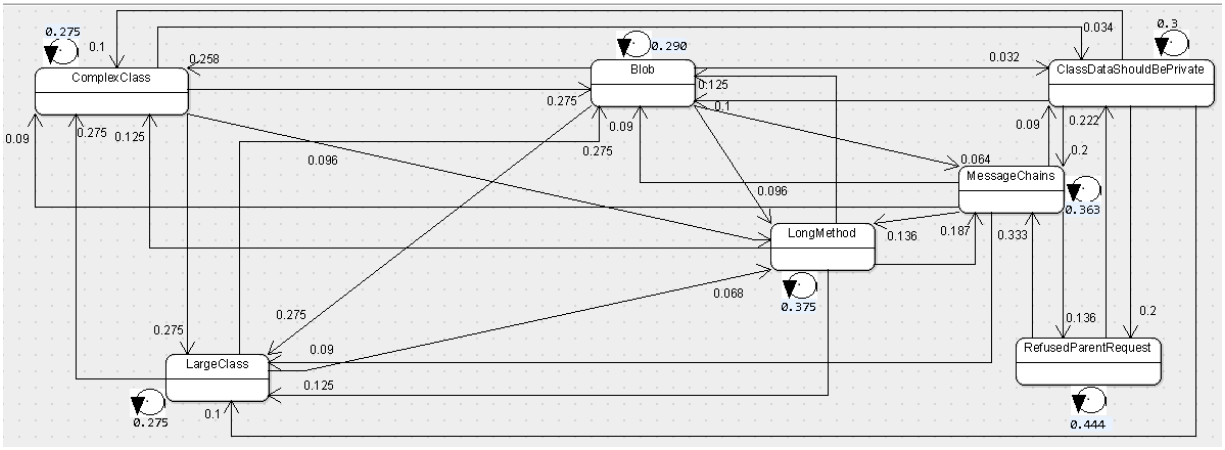

Fig. 3. Anti-pattern mutations and fault-proneness in Mylyn

class that has at the same time the specification of two different anti-patterns. In our study, we manually validate such cases to verify if the class represents many anti-patterns and we keep it in our analysis by calculating and adding all its mutations.

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. Moreover, both ArgoUML, Mylyn, and Rhino source code repositories are publicly available, as well as the anti-pattern detection tool used in this study. We also make all the data used in this study available on the Web[10].

Threats to *external validity* concern the possibility to generalize our observations. First, we performed our study on three systems belonging to different domains and with different sizes and histories. However, we cannot assert that our results and observations are generalizable to any other systems. Second, the facts that all the analyzed systems are in Java and open-source may also reduce the generalizability of our findings. Future work includes replicating our study with other systems.

VI. Related Work

During the past years, different approaches have been developed to address the problem of detecting design patterns, specifying anti-patterns, and spotting their impacts on fault proneness. This section discusses the literature that aims at investigating these problems.

A. Anti-patterns Definition and Detection

The first book on "anti-patterns" in object-oriented development was written in 1995 by Webster [4]. 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.* [13] 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. They are the basis of all the approaches to detect anti-patterns.

[10]http://www.ptidej.net/downloads/replications/qsic14/

The study presented in this paper relies on the anti-pattern detection approach DECOR [14]. However several other approaches have been proposed in the past. For example, Van Emden *et al.* [19] developed the JCosmo tool, which parses source code into an abstract model (similar to the Famix meta-model). JCosmo used primitives and rules to detect the presence of smells and anti-patterns. It could visualize the code layout and display anti-patterns 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 [20]. They defined history measurements which summarize the evolution of the suspects parts of code. Then, they showed that the detection of God Classes and Data Classes can become more accurate 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 [28]. Their approach allowed developers to detect anti-patterns using Bayesian networks, based on probabilistic knowledge in the anti-pattern ontology. The Integrated Platform for Software Modeling and Analysis (iPlasma) described in [21] can be used for anti-patterns 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 some thresholds.

We share with all the above authors the idea that anti-patterns detection is a powerful mechanism to asses code quality, in particular to study whether the existence of anti-patterns and their evolutions make the code more difficult to maintain. Indeed, previous work significantly contributed to the specification and detection of anti-patterns. The approach used in this study, DECOR, builds on these work to offer a method to specify and automatically detect anti-patterns.

B. Software Evolution

Object-oriented programs evolve continuously, requiring constant maintenance and development [29]. Thus, they undergo changes throughout their lifetimes as features are added and faults are fixed. When evolution occurs in an uncontrolled

manner, the programs become more complex over time and thus, harder to maintain [30][31].

Chatzigeorgiou and Manakos [9] presented the results of a case study that investigated the evolution of three bad smells throughout successive releases of two open-source systems. The results indicate that in most cases, bad smells persist up to the latest examined release accumulating as the project matures. Moreover, a significant percentage of bad smells were introduced at the time when the method in which they reside was added to the system. Our work differs from this previous work in that we analyze twelve anti-patterns and we define their mutations. Given this differences between our two studies, we claim that our study is the first detailed analysis of anti-patten mutations, its causes, and its impacts. Vaucher *et al.* [10] used a Bayesian approach to detect the presence of God Classes in software systems and determine how they are introduced, removed, and how they evolved. The authors noted that God Classes remain relatively untouched from release to release and that the correction of a God Class may also move the problem to a different class. In this paper, we propose a more general analysis of the evolution of well known anti-patterns and we provide a simplified but accurate picture of anti-pattern genealogies by observing the causes of their mutations and their impacts on fault-proneness. Aversano *et al.* [32] presented results from an empirical study aimed at understanding the evolution of design patterns in three open-source programs, namely JHotDraw, ArgoUML, and Eclipse-JDT. The study analyzed the frequency of the modification of patterns, the type of changes that patterns undergo and classes that co-change with patterns. Results suggested that developers should carefully consider pattern usage when this supports crucial features of the systems. Such patterns will likely undergo frequent changes and be involved in large maintenance activities, that would be highly affected by wrong pattern choices. While Aversano *et al.* focused on design patterns, our study analyzes the evolution of anti-patterns to determine the impact of mutating from one type of anti-patterns to another. In our previous work published on [8], we conducted an empirical study, performed on three object-oriented systems, which provided empirical evidence of the negative impact of dependencies with anti-patterns on fault-proneness. In fact, we found that having static relationships with anti-patterns can significantly increase fault-proneness. In addition, classes having co-change dependencies with anti-patterns are more fault prone than other classes in the analyzed systems. We 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 almost all releases of the four systems, classes participating in anti-patterns are more change and fault-prone than others.

C. Fault-proneness

The most studied and traditional approach for fault prediction is to relate software faults to the size and complexity [33], [34]. Chidamber and Kemerer [35] proposed a suite of object-oriented design metrics which has been substantiated by several theoretical and empirical studies [36], [37]. 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. Hassan and Holt [38] proposed heuristics to analyze fault proneness. They found that recently modified and fixed classes were the most fault-prone. D'Ambros *et al.* [39] reported that there was a correlation between change coupling and defects which 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 instances, is higher than the change rate of the components. They also enriched bug prediction models based on complexity metrics with change coupling information. Marcus *et al.* [40] used a cohesion measurement based on Latent semantic indexing (LSI), an indexing and retrieval method to identify patterns in the relationships between 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 author used LSI for fault prediction and they stated that structural and semantic cohesion directly impacts the understandability and readability of the source code. Ostrand *et al.* [41], Bernstein *et al.* [42] , and Neuhaus *et al.* [43] predict faults in systems, using change and fault data. While Moser *et al.* [44] used metrics (*e.g.* code churn, past faults and refactorings, etc.) to predict the presence/absence of faults in files of Eclipse.

D. Summary

These previous works raised the awareness of the community towards the impact of anti-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-patterns mutations. We aim to understand if the negative effects of anti-patterns on fault-proneness can propagate among the different releases of object-oriented systems.

VII. Conclusion

The paper presents an empirical study to discover new knowledge about evolution anti-patterns: the mutation of anti-patterns and their impacts. Our motivation is to help software developers in non-trivial tasks related to code evolution analysis by modeling anti-pattern mutations.

This paper reports a preliminary results of predicting the mutations of anti-patterns in three open-source systems. We showed that anti-patterns mutate to represent others form of more complicated anti-patterns. Results showed also that mutated anti-pattern classes are significantly less fault-prone than non-mutated classes. We calculated also the risks that an anti-pattern undergoes a fault fixing change after the mutation and we reported these risks by using a Markov Chain.

Further improvements are foreseen. We plan to apply this approach to analyze the genealogies of a more complete anti-pattern list. We want also to develop a more sophisticated analysis of fault-proneness, so that it contains more information about the developers and their activities that could be

the cause of faults after anti-pattern mutations. Among other things, we want to enrich our approach with information about the duration of evolution period as well as the duration and frequency of appearances of anti-pattern mutations.

REFERENCES

[1] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, "Code quality analysis in open source software development," *Information Systems Journal*, vol. 12, no. 1, pp. 43–60, 2002.

[2] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *Software, IEEE*, vol. 25, no. 5, pp. 22–29, 2008.

[3] M. Mantyla, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. IEEE, 2003, pp. 381–384.

[4] B. F. Webster, *Pitfalls of Object Oriented Development*, 1^{st} ed. M & T Books, February 1995.

[5] W. Brown, H. McCormick, T. Mowbray, and R. Malveau, *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley, 1998, vol. 20.

[6] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2011, pp. 181–190.

[7] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, pp. 243–275, 2012.

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

[9] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Proceedings of the Seventh International Conference on the Quality of Information and Communications Technology*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 106–115.

[10] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes," in *16th Working Conference on Reverse Engineering*. IEEE, 2009, pp. 145–154.

[11] S. Meyn and R. L. Tweedie, *Markov Chains and Stochastic Stability*, 2nd ed. New York, NY, USA: Cambridge University Press, 2009.

[12] BasiliV. and PerriconeB.T., "Software errors and complexity: An empirical investigation," *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, 1984.

[13] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1^{st} ed. John Wiley and Sons, 1998.

[14] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *Transactions on Software Engineering*, pp. 20–36, 2010.

[15] D. Romano, P. Raila, M. Pinzger, and F. Khomh, "Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes," *Working Conference on Reverse Engineering*, pp. 437–446, 2012.

[16] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[17] Y.-G. Gueheneuc and G. Antoniol, "Demima: A multilayered approach for design pattern identification," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 667–684, 2008.

[18] Y.-G. Gueheneuc, H. Sahraoui, and F. Zaidi, "Fingerprinting design patterns," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. IEEE, 2004, pp. 172–181.

[19] E. V. Emden and L. Moonen, "Java quality assurance by detecting code smells," in *The 9th Working Conference on Reverse Engineering. IEEE Computer*. Society Press, 2002, pp. 97–107.

[20] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2004, pp. 223–233.

[21] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[22] C. Gerlec and M. Hericko, "Analyzing structural software changes: A case study." in *The 5th Balkan Conference in Informatics*, 2012, pp. 117–120.

[23] D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 2007.

[24] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, and F. Khomh, "Analysing anti-patterns static relationships with design patterns," *Journal of Electronic Communications of the European Association of Software Science and Technology (accepted)*, 2014.

[25] R. K. Yin, *Case Study Research: Design and Methods - Third Edition*. London: SAGE Publications, 2002.

[26] S. Hassaine, Yann-Gaël, S. Hamel, and A. Giuliano, "Advise: Architectural decay in software evolution," in *Proc. 16th European Conference on Software Maintenance and Reengineering*. ACM, 2012, pp. 267–276.

[27] F. Jaafar, S. Hassaine, Y.-G. Guéhéneuc, S. Hamel, and B. Adams, "On the relationship between program evolution and fault-proneness: An empirical study," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, 2013, pp. 15–24.

[28] D. Settas, A. Cerone, and S. Fenz, "Enhancing ontology-based antipattern detection using bayesian networks," *Expert Systems with Applications*, vol. 39, no. 10, pp. 9041–9053, 2012.

[29] M. Lehman, J. F. R. an P.D. Wernick, D. Perry, and W. Turski, "Metrics and laws of software evolution-the nineties view," in *Fourth International Software Metrics Symposium*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 20–.

[30] D. Bell, *Software Engineering, A Programming Approach*. Addison-Wesley, 2000.

[31] D. Hamlet and J. Maybee, *The Engineering of Software*. Addison-Wesley, 2001.

[32] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta, "An empirical study on the evolution of design patterns," in *Foundations of Software Engineering*. New York, NY, USA: ACM Press, 2007, pp. 385–394.

[33] T. J. McCabe, "A complexity measure," in *Proceedings of the 2Nd International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 407–417.

[34] M. H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*. New York, NY, USA: Elsevier Science Inc., 1977.

[35] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transaction Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[36] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transaction Software Engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[37] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *IEEE Transaction Software Engineering*, vol. 29, no. 4, pp. 297–310, 2003.

[38] A. E. Hassan and R. C. Holt, "The top ten list: Dynamic fault prediction," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*. IEEE Computer Society, 2005, pp. 263–272.

[39] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *Proceedings of the 16th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 135–144.

[40] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, pp. 287–300, 2008.

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

[42] A. Bernstein, J. Ekanayake, and M. Pinzger, "Improving defect prediction using temporal features and non linear models," in *Ninth International Workshop on Principles of Software Evolution*. ACM, 2007, pp. 11–18.

[43] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *The 14th Conference on Computer and Communications Security*. ACM, 2007, pp. 529–540.

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