Mining the Relationship Between Anti-patterns Dependencies and Fault-proneness

Fehmi Jaafar[1,2], Yann-Gaël Guéhéneuc[1], Sylvie Hamel[2] and Foutse Khomh [3]
[1] PTIDEJ Team, École Polytechnique de Montréal, QC, Canada
[2] LBIT Team, DIRO, Université de Montréal, QC, Canada
[3] SWAT, École Polytechnique de Montréal, QC, Canada
E-Mails: {jaafarfe, hamelsyl}@iro.umontreal.ca, {yann-gael.gueheneuc, foutse.khomh}@polymtl.ca

***Abstract*—Anti-patterns describe poor solutions to design and implementation problems which are claimed to make object oriented systems hard to maintain. Anti-patterns indicate weaknesses in design that may slow down development or increase the risk of faults or failures in the future. Classes in anti-patterns have some dependencies, such as static relationships, that may propagate potential problems to other classes. To the best of our knowledge, the relationship between anti-patterns dependencies (with non anti-patterns classes) and faults has yet to be studied in details. This paper presents the results of an empirical study aimed at analysing anti-patterns dependencies in three open source software systems, namely ArgoUML, JFreeChart, and XerecesJ. We show that, in almost all releases of the three systems, classes having dependencies with anti-patterns are more fault-prone than others. We also report other observations about these dependencies such as their impact on fault prediction. Software organizations could make use of these knowledge about anti-patterns dependencies to better focus their testing and reviews activities toward the most risky classes, *e.g.*, classes with fault-prone dependencies with anti-patterns.**

***Keywords*—Anti-patterns; co-change; static relationships; mining software repositories; fault-proneness; empirical software engineering.**

I. Context and Problem

Software systems are never complete and evolve continuously [1]. As they evolve, their complexity grows. Prior work has shown that software complexity is an obstacle to introducing changes and that complex modules tend to be fault-prone [2], [3]. Developers often introduce bad solutions, anti-patterns [4], to recurring design problems in their systems and these anti-patterns lead to negative effects on code quality.

While existing work has shown that anti-patterns are problematic ([5], [6], and [7]), we believe that more attention should be focus on static and co-change relationships between anti-patterns classes and other classes without anti-patterns. We conjecture that, static and co-change relationships with anti-patterns can impact the fault-proneness classes without anti-patterns. A recent finding by Radu and Cristina Marinescu [8] that clients of classes with Identity Disharmonies are more fault-prone than other classes, supports this conjecture. The static relationships between anti-patterns classes and other classes (and vice versa) are typically use, association, aggregation, and composition relationships [9]. Also, classes participating in anti-patterns may have "hidden", temporal dependencies. These dependencies occur when developers know that, when changing a class, they must also change another. The literature describes many approaches to extract and analyse such hidden dependencies and to infer the patterns that describe these changes to help developers to maintain their systems. For example, some previous work [10], [11] detected motifs that highlight co-changing groups of classes and that describe the (often implicit) dependencies or logical couplings among classes that have been observed to frequently change together [12]. Two classes are co-changing if they were changed by the same author and with the same log message in a time-window between some milliseconds and some minutes at the most [12], [13]. Recently, we introduced the novel concept of macro co-change[1] and proposed detection algorithms to identify various co-change situations among the classes of a software system[14].

In this paper, we analyze static and temporal relationships (*i.e.*, co-changes) between anti-pattern and (non)anti-pattern classes from three Java open source software systems: ArgoUML, JFreeChart, and XercesJ.

Research Problem. On the one hand, previous work agree that anti-patterns are commonly introduced by developers but they are more fault prone and counterproductive in program development and maintenance [6]. On the other hand, static relationships and co-change dependencies can be "channels" propagating faults among classes in software systems. However, there is no much information available in the literature about the fault proneness of classes having static or co-change dependencies with classes infected by anti-patterns. In this study, we are looking for evidence that practitioners should pay attention to systems with a high number of classes related to classes infected by anti-patterns, because these classes are likely to be the subject of their change efforts.

As in previous work [15], we assume that a class *C* co-changes with the anti-pattern *A* if *C* co-changes at least with one class belonging to *A*. We also assume that a class *S* has a static relationships with the anti-pattern *A* if *S* has a use, association, aggregation, or composition relationships with at

[1] two or more changed files that exactly change together with long time intervals between their changes and–or performed by different developers and with different log messages

least one class belonging to *A* in one of the versions of the analysed systems.

We analyse dependencies with anti-patterns in two ways: first, we investigate whether classes having static relationships (use, association, aggregation, and composition relationships) with anti-patterns classes are more fault-prone than others. Second, we investigate whether classes co-changing with anti-patterns classes are more fault-prone than others. We formulate the following research questions:

- **RQ1:** *Are classes that have static relationships with anti-patterns more fault-prone than other classes?*
- **RQ2:** *Are classes that co-change with anti-patterns more fault-prone than other classes?*

We found that, in ArgoUML, JFreeChart, and XercesJ, classes having static or co-change dependencies with anti-patterns are more fault prone. We also found that such dependencies can be used to predict faults and–or improve fault prediction models.

Organisation. Section II presents our approach. Section III describes our empirical study. Section IV presents the study results while Section V discusses them along with threats to their validity. Then, section VI relates our study with previous work. Finally, Section VII concludes the study and outlines future work.

II. Approach

This section describes the steps necessary to extract and analyse the data required to perform this study.

A. Step 1: Extracting Anti-patterns From the Source Code

We use the DEtection for CORrection approach DECOR [5], to specify and detect 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. It can be applied on any object-oriented system through the use of the PADL [16] meta-model and POM framework [17]. 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.

We use seven of these metrics to verify if we find differences in fault-proneness between classes having dependencies with anti-patterns and other classes with similar complexity or size. These metrics measure : (1) the total lines of code per class; (2) the number of method calls of a class; (3) the nested block depth of the methods in a class; (4) the number of parameters of the methods in class; (5) the McCabe cyclomatic complexity of the methods in a class; (6) the number of fields of a classes; and (7) the number of methods of a classes. We choose these seven metrics because they have been successfully used in the past [18] to predict post-release faults.

We parse the CVS change logs of our subject systems and apply the heuristics by Sliwersky *et al.* [19] to identify fault fix locations. Precisely, we parse commit log messages using a Perl script and extract bug IDs and specific keywords, such as "fixed" or "bug" to identify fault fixing commits. For each fault fixing commit, we extract the list of files that were changed to fix the fault.

B. Step 2: Detecting Anti-patterns Static Relationships

We use the Ptidej tool suite [16] to detect anti-patterns static relationships. Ptidej characterizes the constituents of class diagrams and proposes algorithms to identify these constituents in source code. Ptidej distinguishes use, creation, association, aggregation, and composition relationships because such relationships exist in most notations used to model systems. This approach uses the PADL [16] meta-model and parses the source code of systems to detect models that include all of the constituents found in any object-oriented system: class, interface, member class and interface, method, field, inheritance and implementation relationships, and rules controlling their interactions. Ptidej depends on a set of definitions for unidirectional binary class relationships that we proposed and formalized in a previous work [16].

C. Step 3: Detecting Anti-pattern Temporal Dependencies

We use Macocha [14] to mine software repositories and identify classes that are co-changing with anti-patterns. Macocha mines version-control systems (CVS or SVN) to identify the change periods in a program, to group classes according to their stability through the change periods, and to identify, among changed classes, those that are co-changing with anti-patterns.

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 neighbor algorithm. Second, it groups changes in adequate 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 changed classes. Finally, Macocha detects classes that are co-changed with anti-patterns.

Macocha also calculates the following process metrics, defined and successfully used in previous work [20] to predict software faults. These metrics are used to verify if we find a difference in fault-proneness between classes having dependencies with anti-patterns and other classes. Thus, process metrics are used to check if classes having similar change histories are more or less fault-prone than classes having dependencies with anti-patterns. Here are the process metrics calculated with the Macocha approach as defined in [20]:

1) Total Prior Changes: measures the total number of changes to a class in the 6 months period before the release.
2) Prior Fault Fixing Changes: the number of fault fixing changes done to a class in the 6 months period before the release.
3) Pre-release faults: the number of pre-release faults in a class in the 6 months period before the release (these

are faults observed during development and testing of a program).

4) Post-release faults: the number of post-release faults in a class in the 6 months period after the release (these faults are observed after the program has been deployed to the users).

D. Step 4: Analysing Anti-patterns Dependencies

Table I provides some statistics about the anti-patterns found in the subject systems considered in this paper. To perform the empirical study, we choose to analyse the relationships of well known anti-patterns. 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 analysed in previous work [7], [5]. Definitions and specifications are beyond the scope of this paper and are available in [6] and [21].

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

In **RQ1**, we test whether the proportion of classes in ArgoUML, JFreeChart, and XercesJ that have static relationships with anti-patterns classes have (or do not have) significantly more faults than those that do not have static relationships with anti-patterns classes.

In **RQ2**, we test whether the proportion of co-changed classes with anti-patterns in ArgoUML, JFreeChart, and XercesJ have (or do not have) significantly more faults than the other classes.

Because previous studies [20], [22] have found size, complexity and process metrics to be good predictors of faults in software systems. We perform an experiment to verify if static relationships and–or co-change relations can provide additional information over these traditional fault prediction metrics. Precisely, our experiment consists in building two models for predicting the presence or absence of faults in classes: (1) one using only change and code metrics and (2) one using change metrics, code metrics, and anti-pattern dependencies information. The goal is to investigate the impact of using anti-patterns dependencies to build an effective fault prediction model. In our experiment, the independent variables are the collection of code and process metrics and the dependent variable is a two value variable that represents whether or not a class has one or more post-release fault. There are various machine learning methods available to build such models. We use Support Vector Machines to build the prediction models because this machine learning method has been widely used in literature and has shown good results [23], [24]. The models output the likelihood of a class to have one or more post release faults. We use statistical tests to examine (the significance of) the difference between the performance of the two models when predicting faults. More specifically, we use off-the-shelf methods from the R[2] statical package to analyze the statistical significance and collinearity attributes of the independent variables used in our experiment.

TABLE I
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 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 SpaghettiCode	1	2	6
# of SpeculativeGenerality	22	3	29
# of SwissArmyKnife	13	26	29

Classes belonging to an anti-pattern can have dependencies (static relationships and–or co-change dependencies) with classes belonging to other anti-patterns. Thus, the tests reported in this paper cover classes that have a dependency with an anti-pattern, regardless of the fact that these classes could belong to other anti-patterns. Nevertheless, we present in Section IV the result of our analysis of the impact of anti–patterns dependencies, for classes belonging to anti-patterns and other classes separately.

III. Study Definition and Design

The *goal* of our study is to assess whether classes having dependencies with anti-patterns have a higher likelihood than other classes to be involved in issues documenting faults. The *quality focus* is the improving of program comprehension and the reducing of maintenance effort by detecting and using anti-patterns static or co-change dependencies. The *context* of our study is both the comprehension and the maintenance of systems.

A. Objects

We apply our approach on three Java systems: ArgoUML[3], JFreeChart[4], and XercesJ[5]. We use these systems because they are open source, have been used in previous work, are of different domains, span several years and versions, and have between hundreds and thousands of classes. Table I summarises some statistics about these systems.

ArgoUML is UML diagramming system written in Java and released under the open-source BSD License. For anti-patterns dependencies analysis, we extracted a total number of 4,480 snapshots in the time interval between September 27th, 2008 and December 15th, 2011.

[2]http://www.r-project.org/
[3]http://argouml.tigris.org/
[4]http://www.jfree.org/
[5]http://xerces.apache.org/xerces-j/

JFreeChart is a Java open-source framework to create charts. For co-change analysis, we considered an interval of observation ranging from June 15th, 2007 (release 1.0.6) to November 20th, 2009 (release 1.0.13 ALPHA). In such interval we extracted 2,010 snapshots.

XercesJ is a collection of software libraries for and manipulating XML. It is developed in Java and managed by the Apache Foundation. For anti-patterns dependencies 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 14th, 2003 and November 23th, 2006.

B. Research Questions

We break down our study into two steps:

- **RQ1:** *Are classes that have static relationships with anti-patterns more fault-prone than other classes?*

First, we check if classes having static relationships (use, association, aggregation, and composition relationships) with anti-patterns classes are more fault-prone than other classes in the three analysed systems.

- **RQ2:** *Are classes that co-change with anti-patterns more fault-prone than other classes?*

Second, we investigate whether classes that are co-changing with anti-patterns classes are more fault-prone than other classes.

TABLE II
PROPORTION OF THE ANTI-PATTERNS DEPENDENCIES (CC: CO-CHANGING SITUATIONS OF ANTI-PATTERNS WITH OTHER CLASSES; S.R.: ANTI-PATTERNS STATIC RELATIONSHIPS)

Anti-patterns	Systems	# of CC	# of S.R.
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
SpaghettiCode	ArgoUml	0	0
	JFreeChart	0	0
	XercesJ	0	0
SpeculativeGenerality	ArgoUml	13	128
	JFreeChart	4	139
	XercesJ	8	201
SwissArmyKnife	ArgoUml	20	69
	JFreeChart	9	142
	XercesJ	18	108

We test the following null hypotheses:

- H_{RQ1_0}: The proportions of faults carried by classes having static relationships with anti-patterns and other classes are the same.
- H_{RQ2_0}: The proportions of faults involving classes having co-change dependencies with anti-patterns and other classes are the same.

If we reject the null hypothesis H_{RQ1_0}, it could mean that the proportions of faults carried by classes having static relationships with anti-patterns and faults carried by other classes in the analysed systems are not the same.

If we reject the null hypothesis H_{RQ2_0}, we explain the rejection as that 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.

C. Analysis Method

The analysis reported in Section IV have been performed using the R statistical environment[6]. We use the contingency tables to assess the direction of the difference, if any. In statistics, a contingency table is a table in a matrix format that displays the frequency distribution of the variables. Fisher's exact test [25] is a statistical significance test used in the analysis of contingency tables. Although in practice it is employed when sample sizes are small, it is valid for all sample sizes. The test is useful for categorical data that result from classifying objects in two different ways. It is used to examine the significance of the association (contingency) between the two kinds of classification, in our study: Faulty classes and clean classes. To compute the p-value of the test, the contingency tables must then be ordered by some criterion that measures dependence and those tables that represent equal or greater deviation from independence than the observed table are the ones whose probabilities are added together. The contingency tables tested in this study contain the total numbers of faulty and clean classes identified in ArgoUML, JFreeChart, and XercesJ.

We also compute the odds ratio [25] that indicates the likelihood for an event 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 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.

IV. STUDY RESULTS

We now present the results of our empirical study. Tables II, III and IV summarise our findings.

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

A. RQ1: Are classes that have static relationships with anti-patterns more fault-prone than other classes?

Table III reports for ArgoUML, JFreeChart, and XercesJ the numbers of (1) classes having static relationships with anti-patterns and identified as faulty; (2) classes having static relationships with anti-patterns and identified as clean (*i.e.*, not faulty); (3) classes without static relationships with anti-patterns and identified as faulty; and, (4) classes without static relationships with anti-patterns and identified as clean. For each case, we present separately the result of the set of classes that do not belonged to other anti-patterns. The result of Fisher's exact test and odds ratios when testing $HRQ1_0$ are significant for all three systems. For the three systems, the p-value is less then 0.05 and the likelihood that a class with static relationship(s) with anti-patterns experiences a fault (*i.e.*, odds ratio) is about two times higher than the likelihood that other classes experience faults.

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

But... Two observations limit the results of **RQ1**: First, in the three systems, as shown in Table II, we do not detect any class having static dependencies (use, association, aggregation, and composition relationships) with SpaghettiCode. In this case, we cannot relate the impact of using this anti-pattern and the fault-proneness of other classes in the systems. Second, based on complexity metrics and change metrics analysis, it is neither possible to conclude that other classes having similar complexity, change history, and code size are less fault-prone than classes having static relationships with anti-patterns. In fact, we take as input the list of code and change metrics described in Section II and check if there is a significant statistical difference on fault proneness between a model based on only these metrics and a model based on these metrics plus anti-patterns static relationships. If all anti-patterns are considered in this comparison, it is impossible to definitely exclude the possibility that there is no statistically differences in fault-proneness between classes related to anti-patterns and other classes with similar complexity, change history, and code size. However, if we group the results according to distinct anti-patterns, we observe that classes having static relationships with Blob, ComplexClass, and SwissArmyKnife are significantly more fault prone than other classes with similar complexity, change history, and code size. Future work include the categorisation of anti-patterns according to the impact of their dependencies on fault proneness.

Other observations. Many anti-patterns' relationships were with classes playing roles in design patterns. Opposite to anti-patterns, design patterns [21] are "good" solutions to recurring design problems, conceived to increase reuse, code quality, code readability and, above all, maintainability and resilience to changes. As a consequence, these classes, playing roles in design patterns and having static relationships with anti-patterns, can bias the results. Indeed, we observe cases where developers amended anti-patterns using design patterns to facilitate maintenance tasks and reduce comprehension effort. For example, in XercesJ v1.0.4, the class `org.apache.xerces.validators.common.XMLValidator.java` is an excessively complex class interface. The developer attempted to provide services for all possible uses of this class. In her attempt, she added a large number of interface signatures to meet all possible needs. The developer may not have a clear abstraction or purpose for `org.apache.xerces.validators.common.XMLValidator.java`, which is represented by the lack of focus in its interface. Thus, we claim that this class belongs to a SwissArmyKnife anti-pattern. This anti-pattern is problematic because the complicated interface is difficult for other developers to understand and obscures how the class is intended to be used, even in simple cases. Other consequences of this complexity include the difficulties of debugging, documentation, and maintenance. We detect that this class has a use-relationship with the 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 sequence 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, developer can correct `org.apache.xerces.validators.common.XMLValidator.java`, by using the related Command pattern, to represent and encapsulate all the information needed to call a method at a later time. This information includes the method name, the object that owns the method, and values for the method parameters. Thus, by using the relationships of an anti-pattern with a specific design pattern, we could help developers to maintain the anti-pattern classes while reducing its influence on the system by benefiting from its relationships with other design pattern so that, in the long term, developers could eliminate this anti-pattern while propagating changes adequately. We plan to study in future work the effect of knowing and using the relationships of anti-patterns and design patterns in maintenance tasks and comprehension effort.

B. RQ2: Are classes that co-change with anti-patterns more fault-prone than other classes?

In the three systems, we detected co-change situations for the majority of anti-patterns classes. In ArgoUML, we observe that Blob, LongMethod, and RefusedParentBequest co-change with other classes more than the rest of anti-patterns. Whilst during the evolution of JFreeChart and XercesJ, Blob is the anti-pattern that co-change the most with other classes.

Table IV presents a contingency table for ArgoUML, JFreeChart, and XercesJ that reports the number of (1) classes co-changing with anti-patterns and identified as faulty; (2) classes co-changing with anti-patterns and identified as clean (*i.e.*, not faulty); (3) other classes identified as faulty; and, (4) other classes identified as clean. For each case, we present separately the result of the set of classes that do not belonged

TABLE III
CONTINGENCY TABLE AND FISHER TEST RESULTS IN ARGOUML, JFREECHART AND XERCESJ FOR CLASSES WITH AT LEAST ONE FAULT (S.R.: STATIC RELATIONSHIPS, AP: ANTI-PATTERN, NBPA: NOT BELONGED TO OTHER ANTI-PATTERNS

	Faulty	Clean
Total of classes having S.R. with AP in ArgoUML	1062	1003
Classes having S.R. with AP and NBAP	402	600
Other classes in ArgoUML	681	579
Total of classes having S.R. with AP in JFreeChart	432	226
Classes having S.R. with AP and NBAP	281	103
Other classes in JFreeChart	310	647
Total of classes having S.R. with AP in XercesJ	445	121
Classes having S.R. with AP and NBAP	262	75
Other classes in XercesJ	126	499
Total of classes related to AP	1939	1350
Classes having S.R. with AP and NBAP	945	778
Total of other classes	1117	1725
Fisher's test	$2.2e-16$	
Odd-ratio	2.21802	
Fisher's test for NBAP	$2.2e-16$	
Odd-ratio for NBAP	1.875567	

TABLE IV
CONTINGENCY TABLE AND FISHER TEST RESULTS IN ARGOUML, JFREECHART AND XERCESJ FOR CLASSES WITH AT LEAST ONE FAULT (AP: ANTI-PATTERNS, NBPA: NOT BELONGED TO OTHER ANTI-PATTERNS)

	Faulty	Clean
Classes co-changing with AP in ArgoUML	241	102
Classes co-changing with AP and NBPA	120	59
Other classes in ArgoUML	1502	1480
Classes co-changing with AP in JFreeChart	68	26
Classes co-changing with AP and NBPA	33	10
Other classes in JFreeChart	674	847
Classes co-changing with AP in XercesJ	37	21
Classes co-changing with AP and NBPA	20	12
Other classes in in XercesJ	534	599
Total of classes co-changing with AP	346	149
Classes co-changing with AP and NBPA	173	81
Total of other classes	2710	2926
Fisher's test	$2.2e-16$	
Odd-ratio	2.50723	
Fisher's test for NBAP	$2.2e-16$	
Odd-ratio for NBAP	2.305731	

to other anti-patterns. The result of Fisher's exact test and odds ratios when testing $HRQ2_0$ are significant. For all the three systems, the p-value is less than 0.05 and the likelihood that a class co-changing with anti-patterns experiences a fault (*i.e.*, odds ratios) is about two and half times higher than the likelihood that other classes experience faults.

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

But... We observe in Table II, in the three analysed systems, that if a class belongs to the SpaghettiCode anti-pattern, it does not co-change with any other class in the system. In ArgoUML, we detect some occurrences of ClassDataShouldBePrivate, ComplexClass, and LongParameterList that co-changed with other classes. However, we do not detect any class playing role in these anti-patterns and which is 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 found that classes that are co-changing with anti-patterns classes are significantly more fault prone than other classes with similar complexity, change history, and code size. However, it is impossible to exclude the possibility that there is no impact on fault-proneness for classes that co-changed with SpaghettiCode, ClassDataShouldBePrivate, ComplexClass, and LongParameterList.

Other observations. If co-change dependencies of anti-patterns are not properly maintained, they can lead to faults in the system. For example, the class `GoClassToNavigableClass.java` belongs to a Blob anti-pattern in ArgoUML0.26. Concurrently, this class is co-changed with the class `GoClassToAssociatedClass.java`. However, these two classes are not always maintained together although the developer changing `GoClassToNavigableClass.java` should, also, assess `GoClassToAssociatedClass.java` for change. Yet, in the Bugzilla database of ArgoUML, the bug ID5505[7] confirms that the two classes are related but were not maintained together, leading to a fault.

V. DISCUSSION

This section discusses the results reported in Section IV as well as the threats to their validity.

A. Exploratory Findings

From Table II, we note that many anti-patterns in ArgoUML, JFreechart, and XercesJ have static relationships and–or have been co-changed with other classes. To the best of our knowledge, we are the first to analyze these dependencies in details; especially their impact on fault proneness.

We do not consider that an anti-pattern is necessarily the result of a "bad" implementation or design choice; only the concerned developers can make such a judgement. We do not exclude that, in a particular context, an anti-pattern can be the best way to actually implement and–or design a (part of a) class. For example, automatically-generated parsers are often very large and complex classes. Only developers can evaluate their impact according to the context: it can be perfectly sensible to have these large and complex classes if they come from a well-defined grammar. Such classes are excluded from our analysis because generated code is likely to be of a very different nature than hand-written code and possibly more reliable because the domain must be well understood before one can develop a code generator. On the other hand, the interface of such code may have the same issues as hand-written code and could affect maintenance in a similar way.

From Table II, we report that different anti-patterns have different proportion of static relationships with other classes in systems. This difference is not surprising because these systems have been developed in three unrelated contexts, under different processes. It highlights the interest of analysing and

[7]http://argouml.tigris.org/issues/show_bug.cgi?id=5505

reporting the anti-patterns dependencies when assessing finely the quality of systems.

SpaghettiCodes do not co-change and have no static relationships (use, association, aggregation, and composition) with other classes in the three analysed systems. This observation is not surprising because a SpaghettiCode is revealed by classes with no structure, declaring long methods with no parameters, and utilising global variables for processing. A SpaghettiCode does not exploit and prevent the use of object-orientation mechanisms: polymorphism and inheritance. With a SpaghettiCode, minimal relationships exist between objects. Many object methods have no parameters, and utilise classes or global variables for processing. Thus, a SpaghettiCode is difficult to reuse and to maintain, and when it is, it is often through cloning. In many cases, however, code is never considered for reuse.

We found that classes that have dependencies with numerous anti-patterns (such as Blob and ComplexClass) are significantly more fault prone than other classes with similar complexity, change history, and code size. However, for the three analysed systems, it is impossible to get significant statistical difference on fault proneness for some anti-patterns such as SpaghettiCode.

We also observe that many anti-patterns dependencies were with other motifs in systems such as design patterns. We noted that developers use design patterns, possibly unintentionally, as proven solutions to recurring design problems [26], *e.g.*, when there is a proliferation of similar methods and–or the user-interface code becomes difficult to maintain.

Last but not least, we confirmed that knowing that two classes are co-changing implies the existence of (hidden) dependencies between these two classes. If these dependencies are not properly maintained, they can introduce faults in a program [27]. We found that classes that co-changed with anti-patterns are more fault-prone than other co-changed classes in ArgoUML, JFreechart, and XercesJ. Thus, by knowing the sets of classes that co-changed with anti-patterns, we could explain and possibly prevent faults, thus lessening the anti-patterns negative impact. Indeed, team managers can guide programmers based on the program history and point out risky item coupling such as classes that are co-changing with anti-patterns classes. In addition, with the availability of such information, a tester could decide to focus on classes having dependencies with anti-patterns, because she knows that such classes are likely to contain faults.

B. Threats to Validity

We now discuss in details the threats to the validity of our results, following the guidelines provided in [28].

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. However, as discussed, we are interested to relate anti-patterns *as they are defined in DECOR* [5] with other classes by static relationships *as they are defined in PADL* [16]. For this reason, the precision of the anti-patterns detection is a concern that we agree to accept. Moha *et al.* [5] 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%. Macocha's approach detection for macro co-change ensures 96% recall and has a precision greater than 85% [14]. We preprocessed the inconsistent anti-patterns to eliminate false positives. This preprocessing reduces the chances that we could answer our research questions wrongly. In addition, our results can still be affected by the presence of false negatives, *i.e.*, by a low recall exhibited by the anti-pattern detection tool. In case 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 aimed at assessing to what extent the detection tool performance assess our results are needed.

We compute the fault-proneness of a class by relating fault reports and commits to the class. In fact, fault fixing changes are documented in text reports that describe different kinds of problems in a program. Thus, we match faults/issues to changes using their IDs and their dates in the ChangeLog files and in the fault reports. On the one hand, we care about independent changes that were accidentally combined in the same commit. Thus, we manually investigate the code to be sure that the fault fixes documented in the commit message is really related to the class committed in the SVN/CVS log file. On the other hand, having a fault is a temporary property, whereas being involved in an anti-pattern is a rather even somewhat long-term persistent property. Thus, these two different types of properties can be related to each other: there will be times when an anti-pattern related to a class will have no fault and times when it will have faults. In this study, and as in previous work [7] analyzing fault proneness, we declared that a class is a faulty class if it was involved in at least one fault fixing change.

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. A possible threat to the conclusion validity is our particular choice of complexity and change metrics. Although these metrics are widely used and accepted by other researchers, there is no consensus on their universality. We do not yet fully understand the complex mechanism of why and how faults are introduced in software systems. Thus, in theory there could exist some better fault prediction metrics, that are yet to be discovered.

Reliability validity threats concern the possibility of replicating this study. We attempted here to provide all the necessary details to replicate our study. Moreover, both ArgoUML, JFreeChart, and XercesJ source code repositories are publicly available, as well as the anti-pattern detection tool used in this study. Our analysis process is described in detail in Section II. Finally, all the data used in this study are available on the

Web[8].

Threats to *external validity* concern the possibility to generalise 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 generalisable to any other systems and the facts that all the analysed systems are in Java and open-source may also reduce the generalizability of our findings. In the future, we plan to analyze further systems, written in different programming languages, to draw more general conclusions. Second, we used particular, yet representative, sets of anti-patterns. Different anti-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 on 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 studies [7].

VI. Related Work

Several works have studied the detection and the analysis of anti-patterns. Other work studied co-changes. Because of lack of space, we only cite some relevant work, the interested readers can find more references in our previous work [5].

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.* [29] 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 anti-patterns detection approach proposed in [5]. However several other approaches have been proposed in the past. For example, Van Emden *et al.* [30] developed the JCosmo tool. This tool parses source code into an abstract model (similar to the Famix meta-model). It used primitive and rules to detect the presence of smells and anti-patterns. Marinescu *et al.* developed a set of detection strategies to detect anti-patterns based on metrics [31]. Settas *et al.* explored the ways in which an anti-pattern ontology can be enhanced using Bayesian networks [32]. 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 [33] 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 indicating whether the existence of anti-patterns and the growth of their relationships makes the source code more difficult to maintain.

Anti-patterns Static Relationships. There are few papers analyzing empirically the anti-patterns relationships.

Binkley [34] *et al.* defined the dependence anti-pattern as a dependence structure that may indicate potential problems for ongoing software maintenance and evolution. Dependence anti-patterns are not structures that must always be avoided. Rather, they denote warnings that should be investigated. Typically these problems will take the form of difficulties in comprehension, testing, reverse engineering, re-use, and maintenance. The authors 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 should look like, we believe that it is comparatively easy to identify dependence between anti-patens and others classes in the systems that denote potential problems.

Radu and Cristina Marinescu [8] 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, he should also monitor the clients of this class since they are likely to also exhibit faults. Our work differs from Radu and Cristina's study in that we analyze different types of dependencies and anti-patterns. Indeed, Radu and Cristina considered only four design flaws called Identity Disharmonies. Identity Disharmonies are design flaws that affect single entities such as classes and method. They are characterized by a lack of harmony between the size of operations and classes; a lack of harmony in the collaboration of data and operations; these entities exhibit more than one responsibility. Example of Identity Disharmonies are: Data Class, God Class, Brain Class, and Feature Envy. In their study, they considered that a class (client) makes use of a God Class, Brain Class or Feature Envy if that class calls at least one method from a flawed class (*i.e.*, God Class, Brain Class or Feature Envy). They considered that a client makes use of a Data Class if the client accesses at least one attribute of the Data class or calls at least one of the exposed methods from the class. Given this differences between our two studies, we can claim that our study is the first detailed analysis of the impact of different anti-patterns relationships (including use relationship), co-change dependencies and fault-proneness. In total, we have analyzed 11 anti-patterns.

Vokac [35] analyzed the corrective maintenance of a large commercial program, comparing the fault rates of classes participating in design motifs against those that did not. Their approach showed correlation between some design patterns and smells like LargeClass but do not report an exhaustive investigation of possible correlations between these patterns and anti-patterns. Pietrzak and Walter [36] defined and analysed the different relationships that exist among smells and provide

[8]http://www.ptidej.net/download/experiments/wcre13/

tips on how they could be exploited to alleviate the detection of anti-patterns. They proposed six coarse relations that describe dependencies between smells: plain support, mutual support, rejection aggregate support, transitive support, and inclusion. Rather than focusing on the relationships among code smells and anti-patterns, our study focuses on analysing anti-patterns dependencies and their impact on fault-proneness.

Co-change Dependencies. Aversano *et al.* [15] 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. Specifically, the study analysed 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 application. 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 analyses classes that co-changed with anti-patterns. Bouktif *et al.* defined the general concept of change patterns and described one such pattern, synchrony, that highlights co-changing groups of classes. Other approaches to detect co-changes exist [12], [13], [15], and [37]. These approaches are intrinsically limited in their definition of co-change. They cannot express patterns of changes between long time intervals and-or performed by different developers. Thus, we introduced the novel patterns of *macro co-changes* (MCCs) and *dephase macro co-changes* (DMCCs) [14], 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 [14], to mine software repositories (CVS and SVN) and identify (dephase) macro co-changing classes. We showed that Macocha have a better precision and recall for co-changes detection than the 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 [7], Khomh *et al.* showed that anti-patterns *do* have a negative impact on class change-proneness and fault-proneness and that certain kinds of anti-patterns do have a higher impact than other. In this paper, we showed, in **RQ2**, that detecting classes that are co-changing with anti-patterns classes help to identify which entities are more likely to be fault prone.

A. Fault-proneness

Nagappan and Ball [18] performed a study on the influence of code churn [22] on the fault density. They found that relative code churn was a better predictor than absolute churn. Moser *et al.* [24] used metrics (*e.g.* code churn, past faults and refactorings, etc.) to predict the presence/absence of faults in files of Eclipse. Hassan and Holt [38] proposed heuristics to analyse fault proneness. They found that recently modified and fixed classes were the most fault-prone. Ostrand *et al.* [23] predict faults in two industrial systems, using change and fault data. Bernstein *et al.* [39] used fault and change information in non-linear prediction models. Zimmermann and Nagappan [20] used dependencies between binaries in Windows server 2003 to predict faults. Marcus *et al.* [40] used a cohesion measurement based on LSI for fault prediction. Neuhaus *et al.* [41] used a variety of features of Mozilla, such as past faults, package imports, call structure, to determine fault vulnerabilities. Previous approach on fault-proneness out there did not link class evolution behaviors to faults. In this paper, we spotted the links between software evolution and fault-proneness.

VII. Conclusions and Future Work

A large amount of effort has been put into analysis models to explain and forecast faults in software systems. As this area of research grows, a greater number of metrics is being used to predict faults. In this paper, we reported the results of an empirical study, performed on three object-oriented systems, which provides empirical evidence of the negative impact of dependencies with anti-patterns on fault-proneness. Through our two research questions:

- **RQ1:** *Are classes that have static relationships with anti-patterns more fault-prone than other classes?*
- **RQ2:** *Are classes that co-change with anti-patterns more fault-prone than other classes?*

We found that:

- Having static relationships with anti-patterns can significantly increase fault-proneness.
- Classes having co-change dependencies with anti-patterns are more fault prone than others.

This empirical study confirms, within the limits of the threats to its validity, the conjecture in the literature that anti-patterns have a negative impact on system architecture. It also suggests to use the knowledge about the anti-patterns dependencies to maintain a system correctly, to eliminate design defects, and to propagate changes adequately.

Future work includes (i) replicating our study on other systems to assess the generalizability of our results, (ii) studying the effect of the anti-patterns dependencies on change-proneness, and (iii) analysing relation between some of the considered size and complexity measures and classes dependent on other classes that are involved in anti-patterns.

Acknowledgment

This work has been partly funded a FQRNT team grant and the Canada Research Chair in Software Patterns and Patterns of Software.

References

[1] M. M. Lehman and L. A. Belady, *Software Evolution - Processes of Software Change*. Academic Press London, 1985.

[2] M. M. Lehman, "Laws of software evolution revisited," in *Proceedings of the 5th European Workshop on Software Process Technology*, 1996, pp. 108–124.

[3] D. L. Lanning and T. M. Khoshgoftar, "Canonical modelling of software complexity and fault correction activity," in *Proceedings of IEEE International Conference on Software Maintenance*, Victoria, 1994, pp. 374–381.

[4] B. F. Webster, *Pitfalls of Object Oriented Development*, 1^{st} ed. M & T Books, February 1995. [Online]. Available: www.amazon.com/exec/obidos/ASIN/1558513973

[5] Naouel 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 (TSE)*, 2010.

[6] 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.

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

[9] Y.-G. Guéhéneuc and H. Albin-Amiot, "Recovering binary class relationships: Putting icing on the UML cake," in *Proceedings of the* 19^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, D. C. Schmidt, Ed. ACM Press, 2004, pp. 301–314.

[10] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 2004, pp. 563–572.

[11] S. Bouktif, Y.-G. Guéhéneuc, and G. Antoniol, "Extracting change-patterns from cvs repositories," in *Proceedings of the 13th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 221–230.

[12] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings of the International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 190–200.

[13] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 563–572.

[14] Fehmi Jaafar, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol, "An exploratory study of macro co-changes," in *Proceedings of the* 18^{th} *Working Conference on Reverse Engineering*, M. Pinzger and D. Poshyvanyk, Eds., 2011, pp. 325–334.

[15] 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.

[16] Y.-G. Guéhéneuc and G. Antoniol, "DeMIMA: A multi-layered framework for design pattern identification," *Transactions on Software Engineering (TSE)*, vol. 34, no. 5, pp. 667–684, September 2008.

[17] Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi, "Fingerprinting design patterns," in *Proceedings of the 11th Working Conference on Reverse Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 172–181.

[18] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 284–292.

[19] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083147

[20] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th International Conference on Software Engineering*. ACM, 2008, pp. 531–540.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, 1^{st} ed. Addison-Wesley, 1994.

[22] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 78–88.

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

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

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

[26] C. Iacob, "A design pattern mining method for interaction design," in *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, ser. EICS '11. ACM, 2011, pp. 217–222.

[27] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 9–16.

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

[29] 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.

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

[31] 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.

[32] D. Settas, A. Cerone, and S. Fenz, "Enhancing ontology-based antipattern detection using bayesian networks," *Expert Systems with Applications*, 2012.

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

[34] D. Binkley, N. Gold, M. Harman, Z. Li, K. Mahdavi, and J. Wegener, "Dependence anti patterns," in 4^{th} *International ERCIM Workshop on Software Evolution and Evolvability*, 2008, pp. 25–34.

[35] M. Vokac, "Defect frequency and design patterns: An empirical study of industrial code," *IEEE Transaction on Software Engineering*, pp. 904–917, 2004.

[36] B. Pietrzak and B. Walter, "Leveraging code smell detection with inter-smell relations," *Extreme Programming and Agile Processes in Software Engineering*, pp. 75–84, 2006.

[37] T. Gîrba, S. Ducasse, A. Kuhn, R. Marinescu, and R. Daniel, "Using concept analysis to detect co-change patterns," in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. New York, NY, USA: ACM, 2007, pp. 83–89.

[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] 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.

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

[41] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 529–540.