A Study on the Relation Between Antipatterns and the Cost of Class Unit Testing

Aminata Sabané[1,2], Massimiliano Di Penta[3], Giuliano Antoniol[2], Yann-Gaël Guéhéneuc[1]

[1] *Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada*

[2] *Soccer Lab., DGIGL, École Polytechnique de Montréal, Canada*

[3] *Department of Engineering, University of Sannio, Benevento, Italy*

E-mails: aminata.sabane@polymtl.ca, dipenta@unisannio.it, yann-gael.gueheneuc@polymtl.ca,antoniol@ieee.org

***Abstract*—Antipatterns are known as recurring, poor design choices; recent and past studies indicated that they negatively affect software systems in terms of understandability and maintainability, also increasing change-and defect-proneness. For this reason, refactoring actions are often suggested. In this paper, we investigate a different side-effect of antipatterns, which is their effect on testability and on testing cost in particular. We consider as (upper bound) indicator of testing cost the number of test cases that satisfy the minimal data member usage matrix (MaDUM) criterion proposed by Bashir and Goel. A study—carried out on four Java programs, Ant 1.8.3, ArgoUML 0.20, CheckStyle 4.0, and JFreeChart 1.0.13—supports the evidence that, on the one hand, antipatterns unit testing requires, on average, a number of test cases substantially higher than unit testing for non-antipattern classes. On the other hand, antipattern classes must be carefully tested because they are more defect-prone than other classes. Finally, we illustrate how specific refactoring actions—applied to classes participating in antipatterns—could reduce testing cost.**

***Keywords*-Antipatterns, Object oriented testing, Testing cost, Refactoring**

I. Introduction

Object-oriented programs consist of hundreds or thousands of classes each contributing to different functionality, component implementation, organization and behavior. Object-oriented (OO) development promotes encapsulation and information hiding to improve software maintainability and comprehensibility. In past and recent years, several metric profiles have been proposed to characterize the quality of an object-oriented design or implementation, for example the Chidamber & Kemerer metrics suite [1]. Furthermore, some authors—*e.g.*, Brown [2]—have tried to provide a systematic classification of poor design choices, referred to as *antipatterns* (APs) [2]. APs are opposite to design patterns [3], *i.e.*, they identify "poor" solutions to recurring design problems, for example Brown's 40 APs describe the most common pitfalls in the software industry [2]. They are generally introduced by developers not having sufficient knowledge and–or experience in solving a particular problem or having misapplied some design patterns.

Despite the rich literature on the undesired side effects of APs and code smells, *e.g.*, [4], [5] on software change-proneness, defect-proneness, and comprehensibility, to the best of our knowledge, there is no study aimed at answering the following question:

What is the impact of APs on software testing effort?

We conjecture that, on the one hand, APs are more difficult to test as they require a substantially higher class unit testing effort, measured in terms of number of test cases in this paper. On the other hand, as previous studies have shown [4], APs are arguably more difficult to understand and maintain and thus it is also reasonable to expect that they have a higher defect-proneness than other classes [5]. Consequently, APs have to be tested more.

Based on such a conjecture, in this paper we investigate (i) whether the number of test cases required to perform class unit testing is higher for classes participating in APs—and specifically in particular kinds of APs—than in other classes, and (ii) what is the cost-benefit tradeoff achieved when prioritizing testing of classes participating in APs compared to other classes. Then, the paper shows how specific source code refactoring actions can be used as a means to reduce the high effort required to test AP classes.

In this paper we estimate the cost of class unit testing based on the number of test cases required by the minimal data members usage matrix, MaDUM [6] technique. We choose such a technique because, differently from others, it does not require specific design documentation—e.g., state machines or invariants—to be applied. This technique is also known to suggest a larger number of test cases and hence can be considered as an upper-bound for the testing cost. MaDUM's core idea is that to test a class a developer must test, in isolation, methods interacting with any given attribute, referred to as *data slice*. For each data slice, MaDUM starts with testing all accessors of the attribute, all constructors, and then all setters. After that, it tests all possible permutations of transformers, *i.e.*, methods that modify the attribute. When a class has many data slices and, in turn, each data slice has a high number of transformers, the number of test cases increases exponentially.

In this paper, we computed the number of MaDUM test cases for classes of four Java open-source programs, namely Ant 1.8.3, ArgoUML 0.20, CheckStyle 4.0, and JFreeChart 1.0.13. Then, we detected, using an existing

tool named DECOR [7], the presence of APs in programs, and related the number of test cases with the presence of APs. Also, we related the occurrences of APs to class defect-proneness. Results of our study support the evidence that classes participating in APs—in particular Blobs, Anti-Singleton and Complex Classes—require a higher number of test cases than other classes. In most cases, APs are also more defect-prone than other classes. Hence, even if testing classes that participate in APs is expensive, it is crucial to detect a high proportion of defects. Finally, with the aim of reducing such a cost, we explain how specific refactoring actions—generally different from "traditional" refactoring aimed at removing APs and increasing understandability and maintainability—can be applied to reduce the size of the test suite and therefore the testing cost.

Structure of the paper. Section II provides background information about the MaDUM class testing strategy. Section III and Section IV describes the empirical study design and the obtained results, respectively. Section V illustrates how specific refactoring actions can be applied to classes participating in APs to reduce the testing cost. Section VI discusses the threats to study validity. Section VII overviews the related literature. Finally, Section VIII concludes the paper and outlines future work.

II. A Primer on the MaDUM Testing Strategy

Many authors pointed out the insufficiency of traditional methods of functional and structural unit testing concerning OO unit (class) testing [8]. The main reason is that many defects depend on the behavior of particular methods when the class is in a certain state.

Indeed, black-box and white box testing strategies conceived for procedural programs test OO methods as standalone units, therefore they could miss errors that are due to the interaction between methods (inter-methods). To address this insufficiency, Bashir and Goel [6] proposed an approach to test the interaction of OO methods. The approach, widely known as the MaDUM (Minimal Data members Usage Matrix) testing strategy is based on *data slices*, defined as the set of methods that access or modify a data field (*i.e.*, attribute). The correctness of a class is tested in terms of correctness of all its slices, that are tested separately.

The identification of each slice is based on two key elements: the enhanced call-graph (ECG) and the MaDUM. An ECG represents the accesses (usage or invocation) of members of a class by the other members. The ECG of a class C can be defined as: $ECG(C) = (M(C), F(C), Emf, Emm)$, where $M(C)$ represents the set of methods of C, $F(C)$ is the set of fields of C, $Emf = (m_i, f_j)$ indicates that there is an edge between the method m_i and the field f_j, *i.e.*, m_i accesses the field f_j, and $Emm = (m_i, m_j)$ indicates that there is an edge between the methods m_i and m_j, *i.e.*, m_i invokes m_j. The MaDUM strategy classifies methods into four categories:

- Constructors (c): class constructors;
- Transformers (t): methods that alter the state of one or more fields;
- Reporters (r): methods that return the value of a field;
- Others (o): methods that do not fit in the categories above, *e.g.*, methods that handle special conditions/exceptional behavior.

A MaDUM is an $nf \cdot nm$ matrix, where nf and nm are respectively the number of fields and the number of methods in the class. Built based on the ECG of the class, each cell (i, j) of the MaDUM is marked as follows:

- t if a method m_j transforms a field f_i;
- r if m_j reports the state of f_i;
- o if m_j accesses the field f_i without being a transformer or a reporter.

The MaDUM testing strategy works as follows. First, reporters are tested, followed by setters (if present) and by constructors. Then, interactions among transformers are tested generating, for each given slice, the permutation of slice transformers for each constructor context. In other words, let c be the set of constructors and t be the set of transformers in the given slice, to test the slice the tester must produce $|c| \cdot |t|!$ test cases, where $|c|$ is the number of constructors (number of elements of the constructor set) and $|t|$ the number of transformers. Finally, the others (o) are tested using traditional black or white-box testing strategies.

A method m_j can access a field f_i directly or indirectly through another method m_k (transitively) invoked by m_j. However, if m_j accesses f_i only through m_k, and m_k has been tested already in the f_i slice, then according to the strategy [6], there is no need to also test m_j in the f_i slice.

Further details about the approach and the algorithm used to generate the MaDUM can be found in [6].

III. Study Design

The *goal* of this study is to investigate the cost of unit testing for classes participating in APs, as opposed to other classes, and the potential benefits obtained with such a testing activity. The *quality focus* is the effort needed to produce test cases, and the extent to which testing particular classes would help to reveal defects. The *perspective* is of researchers, interested to understand the influence of APs on software quality from the point of view of testing and to conduct more research in this direction.

The *context* of this study consists of a class unit testing technique—the MaDUM technique—and one release of four Java open-source projects. We have chosen the MaDUM strategy because it requires information usually available from source code and a high number of test cases—that combinatorially increases with the number of transformers. In summary, such a number of test cases would represent an upper bound when testing the class, while other OO testing techniques—such as those based on pre- and post-conditions

Table I
NUMBER OF CLASSES PARTICIPATING IN DIFFERENT KINDS OF APS.

Name (Abbr)	Ant	ArgoUML	CheckStyle	JFreeChart
AntiSingleton (AS)	2	257	15	20
BaseClassShouldBeAbstract (BCSBA)	20	20	3	14
Blob (B)	50	123	11	32
ClassDataShouldBePrivate (CDSBP)	84	44	4	20
ComplexClass (CC)	103	214	34	55
LazyClass (LzC(	46	60	4	21
LongMethod (LM)	178	267	69	102
LongParameterList (LPL)	34	237	8	50
MessageChains (MC)	186	145	7	56
RefusedParentBequest (RPB)	67	497	80	62
SpaghettiCode (SC)	1	45	1	0
SpeculativeGenerality (SG)	4	23	1	1
SwissArmyKnife (SAK)	1	4	0	14
Antipattern classes	452	901	161	245
No Antipattern (None)	297	376	99	233

[9] as well as state-based techniques [10], [8]—would often require a smaller number of test cases, but unfortunately rely on representations rarely available in practice.

We chose the four open-source programs according to different criteria (i): systems belonging to different application domains, (ii) availability of bug-fixing data from versioning and issue-tracking system, and (iii) use in previous studies concerning APs and-or testability [11], [12]. Table I reports the number of classes that participate in APs or not for the four systems[1]. *Apache Ant*[2] is a built tool for Java. Its release 1.8.3 has 209 KLOC for 767 classes. *ArgoUML*[3] is an open-source tool for UML diagrams. We used its version 0.20, which consists of 1,277 classes for 196 KLOC. *CheckStyle*[4] is a development tool for Java programs. It checks whether Java code adheres to a specific coding standard chosen by the developers. Its release 4.0 has 261 classes for 56 KLOC. *JFreeChart*[5] is a Java class library to embed/generate charts in Java programs. Its release 1.0.13 consists of 484 classes for 183 KLOC.

A. Research Questions

This study aims at addressing three research questions:

RQ1: *How large is the MaDUM test suite for classes participating in APs compared to that of other classes?* This research question investigates whether classes participating in APs have larger MaDUM test suites than other classes. The conjecture is that poor design and coding practices have also consequences on testing. Because APs make possibly difficult to partition methods into data slices, *i.e.*, most of the methods belong to all slices, they require a high number of test cases to fulfill the MaDUM testing strategy.

RQ2: *How does the size of the MaDUM test suite vary among classes participating in different kinds of APs?* This research question refines the question previously investigated in **RQ1**. The conjecture is that some APs can have a higher impact on the testing cost than others.

RQ3: *What is the potential cost-benefit achieved when focusing testing on APs, as opposed to other classes?* This research question adds a further dimension, *i.e.*, given the cost needed to test a class, we investigate what would be the benefit we gain—in terms of discovered defects—assuming that the testing strategy we employ is able to detect all defects of the class under test.

B. Analysis Method

In the following, we describe the dependent and independent variables of this study, and the statistical procedures used to address each research question. All statistics have been performed using the *R* statistical environment[6]. For all statistical tests, we assume a significance level of 5%.

For **RQ1**, the *dependent variable* we measure is the number of the needed test cases—using the MaDUM testing strategy—to test each class. The *independent variable* is the participation of classes in APs. Basically, such a Boolean variable is true if a class participates to at least one AP. It is false otherwise.

We statistically compare the number of test cases between AP and non-AP classes. Specifically, we test the following hypotheses H_{01} : *There is no significant difference between the number of test cases of classes participating and not in APs.* We test the hypothesis using a non-parametric test, the Mann-Whitney U test. Because we do not know a priori whether the number of test cases will be higher in one direction or in the other, we perform a two-tailed test. Besides testing the hypothesis, we also estimate the magnitude of the differences of means between classes participating and not in APs. We use non-parametric effect size measure Cliff's d [13], which indicates the magnitude of the effect size of the treatment on the dependent variable. The effect size is small for $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$ [13].

For **RQ2**, the *dependent variable* is also the number of MaDUM test cases. The *independent variable* is a Boolean variable for each kind of AP, indicating whether a class participates in that kind of AP or not. We test the following null hypothesis: H_{02} : *There is no significant difference between the number of test cases of classes participating in different kinds of APs.* First, we test the hypothesis using the Kruskal-Wallis test, which is a non-parametric test for comparing multiple medians. Then, we pairwise compare the number of test cases for different kinds of APs using the Mann-Whitney U test. Finally, we correct the obtained p-values using the Holm's correction [14]. This procedure

[1]The sum of classes participating in different kinds of APs can be greater than the number of classes participating in at least one AP, because some classes participate in more than one AP.

[2]http://ant.apache.org/

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

[4]http://checkstyle.sourceforge.net/

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

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

sorts the p-values resulting from n tests in ascending order of values, multiplying the smallest by n, the next by $n-1$, and so on.

For **RQ3** we identify—for the analyzed releases of the four projects—the number of post-release defects affecting each class. Then, we put ourselves in the perspective of a "lazy" tester, who starts to test the classes in an increasing order of number of test cases regardless of other more sound criteria, for example the number of needed stubs or the different kinds of interactions with other classes. We thus assume a simple and blind testing strategy as our goal is to verify the potential increase in detected defects rather than using more sophisticated approaches such as the firewall strategy [8], [15] or other strategies to determine the integration testing order in OO systems [16].

We analyze what the potential benefit achieved is in terms of defects that can be discovered when test cases are increased. We perform such an analysis—by plotting cumulative curves of number of test cases (*independent variable)* vs. number of defects that can be discovered if properly testing these classes (*dependent variable*)—for classes that participate in APs or not.

C. Data Extraction

In this section we describe how we extract data needed for our study, *i.e.*, how we measure the dependent and independent variables.

As for the number of MADUM test cases, we implement the MaDUM strategy and count the number of test cases as detailed in Section II.

For what concerns the detection of APs, as in previous works [5], [11], we use DECOR, the approach with highest precision in the literature [7], to detect APs. DECOR uses a set of rules (metrics, relations between classes) that describe the characteristics of each AP. The input of DECOR is a Patterns and Abstract level Description Language (PADL) model [17]. A PADL model is an abstract representation of the structure and part of the behavior of object-oriented systems, including classes, interfaces, methods, attributes, inheritance relations, etc. PADL models are generated by the Ptidej tool suite [18] based on the source or bytecode of programs. Further details can be found in [7].

Finally, the number of post-release defects, has been identified as follows:

1) we identify, from the commit notes of the versioning system, changes related to bug-fixing, by matching issue tracking system IDs and keywords such as "bug" and "fixed". We limit our attention to the time frame between the release date and the next release;
2) we check, by analyzing the information in the issue tracking system, whether the fix concerns a corrective maintenance (defect fixing) or whether it is the implementation feature-request/enhancement. . Then, we discard the latter. Also, we restrict our attention to issues marked as "CLOSED" and "FIXED" in the issue tracking system;
3) finally, for the remaining issue after step (2), we map the change to the affected classes by analyzing the change that occurred. After that, we count the number of defect-fixing changes that occurred to each class.

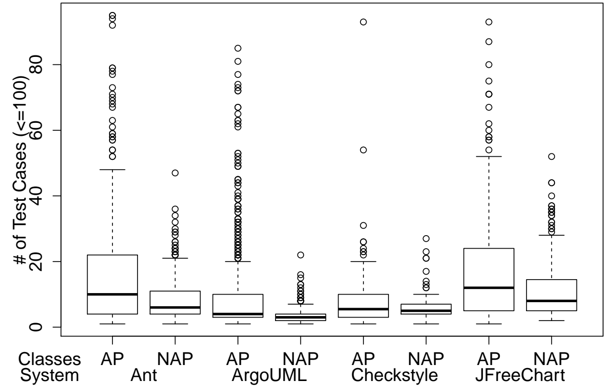


Figure 1. Number of test cases in the MaDUM test suite for classes participating (AP) and not (NAP) in antipatterns.

Table II
MADUM TEST SUITE SIZE FOR CLASSES PARTICIPATING AND NOT TO APS: MANN-WHITNEY TESTS AND CLIFF'S d RESULTS.

System	Mean TCs AP	Mean TCs NAP	p-Value	Cliff's d
Ant	18	9	$<$ **0.01**	0.23 (Small)
ArgoUML	10	3	$<$ **0.01**	0.35 (Medium)
CheckStyle	9	6	$=$ **0.01**	NA
JFreeChart	26	13	$<$ **0.01**	0.22 (Small)

The number of defect-prone classes is, for Ant 131 out of 767 (17%), for ArgoUML 190 out of 1284 (15%), for CheckStyle 3 out of 261 (1%), and JFreeChart 16 out of 484 (3%).

IV. EMPIRICAL STUDY RESULTS

This section reports the results of our empirical study. Working data sets are available for replication purposes[7].

A. RQ1: How large is the MaDUM test suite for classes participating in APs compared to that of other classes?

Figure 1 shows—for the four analyzed systems—the distribution of MaDUM test suite size for classes participating and not in APs. The figure clearly highlights how, with some exceptions (CheckStyle in particular), the test suite size is larger for classes participating in APs than for other classes.

Table II reports the Mann-Whitney test results and Cliff's d effect size obtained when comparing the number of MaDUM Test cases between classes that participate in APs or not . Except for CheckStyle, results show statistically-significant differences. The Cliff's d effect size is small for Ant and JFreeChart, and medium for ArgoUML.

[7]http://ser.soccerlab.polymtl.ca/ser-repos/public/tr-data/MaDUM-AP-Replication-Package.tar.gz

RQ1 Summary: We can reject the null hypothesis H_{01} for Ant, ArgoUML, and JFreeChart. In these three systems, the number of test cases required for MaDUM testing strategy of AP classes are significantly higher than those of other classes. For CheckStyle, the difference is not statistically significant, therefore we cannot reject H_{01} for this system. We can conclude stating that AP classes are less testable than non-AP classes. If developers want to test AP classes thoroughly using the MaDUM testing strategy, they need to write more test cases and this may increase the testing cost and effort.

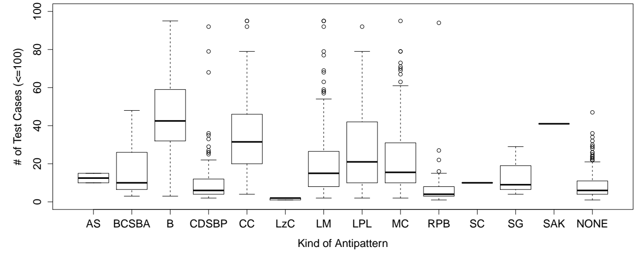
B. RQ2: How does the size of the MaDUM test suite vary among classes participating in different kinds of APs?

For Ant, classes participating in Blob (B) APs require a significantly higher number of test cases than classes participating in other APs (BCSBA, CSBP, LzC, LM, MC, RPB) and than classes not participating in APs. A high number of test cases is also required for Complex Class (CC), and, specifically, significantly higher than CDSBP(ClassDataShouldBePrivate), LzC (LazyClass), LM (LongMethod), MC (MessageChains), RPB (RefusedParentBequest), and classes not participating in APs. In all cases, the Cliff's delta effect size is high. Some APs do not imply a high number of test cases: it is the case of LzC or RefusedParentBequest (RPB).

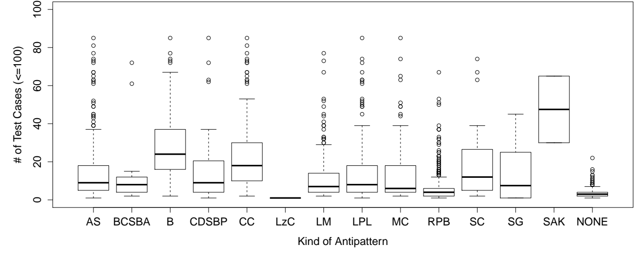
We obtain similar results for ArgoUML, where Blob (B) APs require a significantly higher number of test cases than LzC, LM, LPL, MC, RPB, BCSBA (high effect size), than CDSBP, SG, SC (medium effect size), and classes not participating in APs (high effect size). ComplexClass (CC) APs require a significantly higher number of test cases than LzC, RPB, BCSBA (BaseClassShouldBeAbstract)—with high effect size—than CSBLP, LM, LPL, MC (medium effect size), and than classes not participating in APs (high effect size). Also, AntiSingleton (AS) APs require more test cases than LzC, RPB and no AP classes (high effect size in all cases), and CDSBP APs require more test cases than LzC, RPB and no AP classes (high effect size). We found no significant difference for SwissArmyKnife (SAK)—despite what the boxplot shows—because of the limited number of instances of this AP found.

For CheckStyle, due to the limited number of AP instances, the only significant difference found is for AntiSingleton (AS) classes, that require a significantly higher number of test cases than LzC and RPB, in both cases with a high effect size.

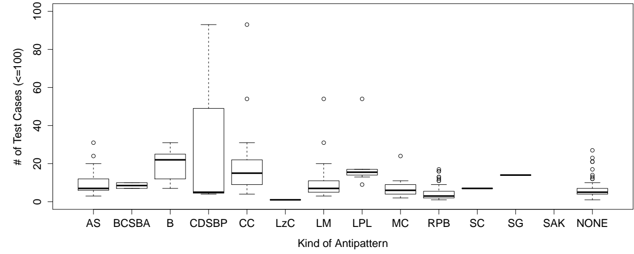
A similar situation occurs for JFreeChart, here again, AS classes require a significantly higher number of test cases than LzC and RPB (with a high effect size).



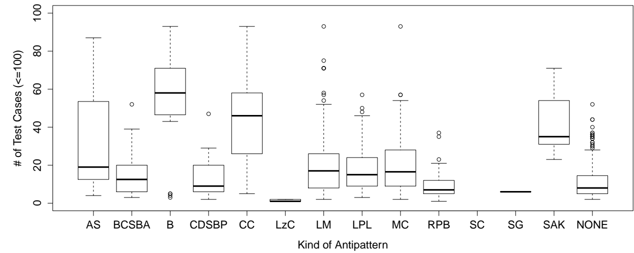
(a) Ant



(b) ArgoUML



(c) CheckStyle



(d) JFreeChart

Figure 2. Number of test cases in the MaDUM test suite for classes participating in different kinds of APs.

Overall, the obtained results tell that, despite **RQ1** indicated low or no significant difference between the number of test cases required for classes that participate in APs in general and those that do not, different kinds of APs exhibit different results. Intrinsically, some APs are classes with more responsibility—see for instance Blob, ComplexClass, or AntiSingleton. In these cases the presence of APs is

Table III
RESULTS OF FISHER'S EXACT TEST FOR DEFECT-PRONENESS OF CLASSES PARTICIPATING AND NOT IN APS. (DP: DEFECT PRONE).

System	AP classes		NAP classes		p-value	OR
	DP	Not DP	DP	Not DP		
Ant	91	354	28	268	$<$ **0.001**	2.45
ArgoUML	167	726	19	357	$<$ **0.001**	4.31
CheckStyle	2	158	0	99	0.5259	Inf
JFreeChart	12	221	0	232	$<$ **0.001**	Inf

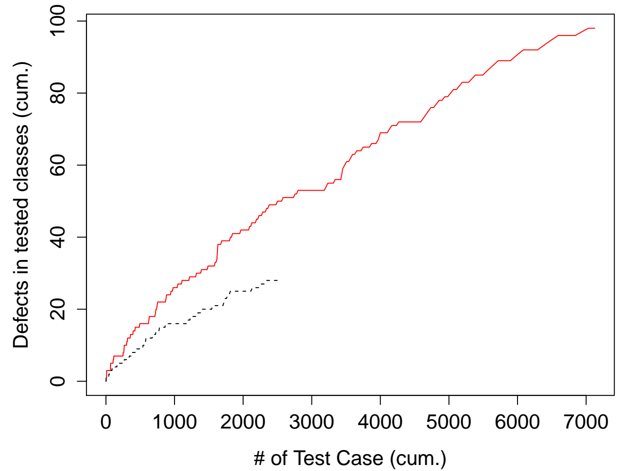
also related with a higher number of test cases. Other kinds of APs, such as LazyClass, RefusedParentBequest or MethodChains are not really related to classes having too much responsibility, hence the presence of APs does not imply having more test cases.

RQ2 Summary: Classes participating in APs related to "excess of responsibility" such as Blob, ComplexClass, AntiSingleton, or SwissArmyKnife, require a significantly higher number of test cases than other classes. Instead, APs such as LazyClass, MethodChains, or RefusedParentBequest require a relatively small number of test cases.

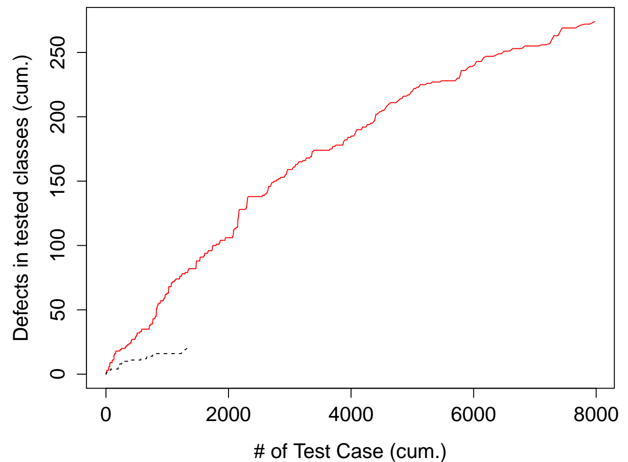
C. RQ3: What is the potential cost-benefit achieved when focusing testing on APs, as opposed to other classes?

Table III reports results of the Fisher's exact test, which statistically compares the proportion of defect-prone classes between classes participating and not in APs (replicating what has been done by Khomh *et al.* [5]). For Ant and ArgoUML, results of the test are statistically significant, and classes participating in APs have 2.45 and 4.31 more time the chance of exhibiting at least a post-release defect than other classes. As for CheckStyle and JFreeChart, only AP classes exhibited post release defects, therefore the Fisher's exact test OR is infinite. In summary—and in agreement with the findings of Khomh *et al.*—classes participating in APs have higher chances of exhibiting post-release defects than other classes.

Figure 3 shows what would be the cumulative number of defects found when testing only classes participating in APs (red line) or only classes not participating in APs (black line)" We adopt a "lazy" ordering, testing classes in ascending order according to their number of test cases, *i.e.*, testing those with a lower number of test cases first. When interpreting such results, it is important to point out that: (i) we are considering class unit testing therefore we are not following any strategy for integration ordering, and (ii) we consider that, when we test a class, all defects will be discovered. We are aware this is not always true; however what we show is basically an upper bound of the number of defects one can discover if testing such classes.



(a) Ant



(b) ArgoUML

Figure 3. Testing cost-effectiveness for classes participating (red) and not (black/dashed) in APs.

Due to lack of space, we only report results for Ant and ArgoUML. As both figures show, it is clear that, even when starting to test the first few classes, the number of defects that can be found for classes participating in AP is substantially higher than for other classes. In the graphs of results for CheckStyle and JFreechart, the non-AP line is flat because there is no defects in these classes in these two systems.

RQ3 Summary: Classes participating in APs exhibit a significantly higher defect-proneness than other classes. Despite the fact that their testing cost is higher, it is cost-effective to analyze/test them with a higher priority than other classes.

In summary, this study has highlighted that APs increase the class unit test cost, although at the same time such classes have to be properly tested because they contain a high proportion of defects. In the following, we will qualitatively show how specific refactoring actions can be applied to classes participating in APs. Our aim is to pursue refactoring objectives that are different from the usual ones, *i.e.*, decreasing the testing cost rather than improving maintainability.

V. Refactoring for Reducing the Testing Cost

According to the MaDUM testing strategy—see Section II—the number of transformers and of constructors in each data slice can dramatically increase the number of test cases. Table IV reports data about cases in which specific refactoring activities, discussed below, can be used to reduce the number of transformers in the data slices of such classes. It is important to point out that such refactoring activities do not remove APs, *i.e.*, they are complementary to "traditional" refactorings that one often performs with the aim of increasing comprehensibility and maintainability.

A typical situation that we found in most of the classes reported in Table IV is related to source code fragments—cloned in multiple methods—that transform the same fields. These cloned statements contribute to increase the number of transformers per slice and consequently the number of test cases. We reduce the testing cost by performing an extract method refactoring.

For example, in the class *PropPanel* of ArgoUML, we found a sequence of statements that transforms the field *listenerList*, and that is repeated in four methods with a slight variation. These repeated sequences increase the number of transformers for the slice of *listenerList* field and, consequently, the number of test cases required to test that class according to the MaDUM strategy. We extract those statements and create a new method that is then called by the old ones. With this refactoring action, the number of transformers for the field *listenerList* becomes 3 instead of 5, and the number of test cases of the refactored class is 43 instead of 271. This refactoring reduces the number of the transformers of this class and then the number of test cases required to test it according to the MaDUM testing strategy.

Another case concerns multiple methods having a very similar code structure and behavior while having a different name. This is possibly meant to make the source code easier to understand. Those methods transform the same field(s). An example of this case occurs in the class *AxisState* of JFreeChart. In this class, we have four methods, namely *cursorUp*, *cursorDown*, *cursorLeft*, and *cursorRight* that increment (*cursorDown* and *cursorRight*) or decrement (*cursorUp* and *cursorLeft*) the field *cursor* by a given value passed as argument. We can refactor to reduce the number of test cases by replacing the four methods by a new one, namely *moveCursor*. Then, we replace the call of the old methods by the new one and adjust the argument: a positive argument is passed instead of calling *cursorDown* or *cursorRight* and a negative argument is passed instead of calling *cursorUp* and *cursorLeft*. This refactoring helps to reduce the number of transformers from 5 to 1 and, consequently, the number of test cases. However, this refactoring could negatively affect code understandability. Indeed, the old methods had more appropriate and straightforward names than the new one. As an alternative, it is possible to use the old methods as simple wrappers directly calling the new method *moveCursor*. Thus the old methods are still used. The code is actually refactored into *moveCursor* and only *moveCursor* must be tested achieving thereby the double goal of not affecting understandability while reducing the number of test cases. This example shows that refactoring performed for the sake of reducing the testing cost must be carefully chosen to avoid decreasing other quality attributes such as understandability.

In summary, the examples reported in Table IV show how there can be opportunities for refactoring that can reduce the testing cost. Noticeably, all classes reported in our examples except the class AxisState participate in APs and this means—as discussed in Section IV—that a high number of test cases is required to test them. We suggest that APs refactoring should not only consider actions to improve cohesion, reduce coupling, and in general address all maintainability issues. It should also consider specific refactoring activities, as those described above, aimed at reducing the number of transformers per data slice and thus the number of test cases. It is also important to notice that such refactorings can be worthwhile also for some classes that do not participate in APs and that however have a high number of transformers, *e.g.*, class *AxisState* of JFreeChart. The examples above show also that, when applying refactoring actions to classes with the purpose of reducing testing

Table IV
Impact of the reduction of the number of transformers (TRS) on the number of test cases (TC).

Class (system)	Before refactoring Type	TRS	TCs	After refactoring TRS	TCs
TokenFilter (Ant)	CDSBP	5	263	2	27
PropPanel (ArgoUML)	Blob	5	271	3	43
BooleanExpressionComplexityCheck (Checkstyle)	LPL	6	732	5	132
AxisState (JFreeChart)	NAP	5	248	1	11
DynamicTimeSeriesCollection (JFreeChart)	Blob	4	208	2	122

costs, we must be aware of the possible impact on other quality attributes and find the best tradeoff, *e.g.*, between testability and comprehensibility/maintainability.

VI. Threats to Validity

In this section, we discuss the main threats to validity that could affect our study.

Threats to *construct validity* concern the relation between theory and observation. In this paper, this is mainly due to possible mistakes/imprecisions in the APs detection, in the classification of defects used to address **RQ3**, and in the "lazy" testing ordering considered in **RQ3**. Concerning APs detection, the study was based on DECOR APs identification [7]. Although DECOR is known to be accurate [7], there is no guarantee that we detect all APs or that what we classified as APs are indeed true APs.

A second threat to construct validity derives from the definition of defect, the content of bug tracking systems and the way in which defects are assigned to classes. It is well-known that issue tracking systems contain all sort of change requests [19]. Thus, in general, we cannot guarantee that all issue tracking entries are indeed related to fixing defects. For two systems (Ant and ArgoUML), the issue tracking system uses a specific category to classify corrective maintenance changes ("DEFECT"). For the two others (CheckStyle and JFreeChart), we relied on information about fixed bugs available in the release notes. Last but not least, the approach we used to link issue reports to commits can miss some fixes not explicitly mentioning the issue ID in the commit note [20].

Finally, the simplistic "lazy" integration strategy, has to be considered as a way of gauging the maximum theoretical difference, if any, between defects detection obtained by prioritizing APs classes testing over classes not participating in APs. We cannot claim that MaDUM test cases will eventually detect all defects or what percentage of undetected defects exists. Also, such a simplistic testing approach does not account for class interactions, nor it considers the actual complexity of writing and running test cases. In essence, we can only claim that, no matter what the testing strategy or the testing order, developers should focus their quality assurance efforts on classes participating in APs, and possibly remove APs as a first step toward an improved design.

Threats to *internal validity* concern any confounding factors that could have influenced the results of our study. This mainly concerns the subjectivity in performing manual class refactoring to reduce the number of test cases. To some extent, this threat is also related to the perception the authors (who performed the refactoring) have of what could be considered a good design.

Threats to *conclusion validity* concern the relation between the treatment and the outcome. We used proper tests and effect size measures to address our research questions. In particular, we used non-parametric tests (Wilcoxon and Fisher's exact test) and effect size measures (Cliff's delta and Odds Ratio) that do not make any assumption about the underlying distributions of data. Finally, we dealt with problems related to performing multiple Wilcoxon tests using the Holm's correction procedure.

Threats to *external validity* concern the possibility of generalizing our results. Although we analysed programs belong to different application domains and developed by different teams, the study needs to be replicated on further programs to confirm or contradict our results.

VII. Related Work

This section discusses related literature concerning APs and their detection, class testability, and OO unit testing strategies.

A. Antipatterns

Many researchers and practitioners described APs and proposed solutions to refactor and–or remove them. Webster first discussed about APs in the context of OO programming in 1995 [21]. He described conceptual, implementation, and quality-assurance problems. Brown *et al.* [22] introduced 40 APs, including Blob and Spaghetti Code.

Researchers have proposed a number of approaches to detect code smells and APs based on different techniques, ranging from manual inspection [23] to rule-based systems [7], [24]. More recently, researchers have used machine learning techniques — see for instance Khomh *et al.* [25] and Maiga *et al.* [26] — to locate code smells and APs.

Furthermore, various studies suggested that code smells and APs negatively impact software quality. Deligiannis *et al.* conducted an empirical study to analyze the influence of God classes on software understandability and maintainability [27]; their findings support the claim that God classes have a negative impact on the evolution of design structures. Abbes *et al.* [11] investigated the influence of Blob and Spaghetti Code on software understandability. They discovered that the presence of Blob or Spaghetti Code does not have significantly negative impact on software understandability, but the combination of the two APs made programs significantly more difficult to understand. Olbrich *et al.* [28] and Khomh *et al.* [5] studied the relation between the presence of smells and change/defect-proneness. The two studies agree on the negative impact of APs on change- and defect-proneness.

Finally, some authors have discussed refactoring opportunities to remove APs. For example, both Brown [22] and Fowler [29] provide APs definitions as well as rules to refactor them. Tsantalis *et al.* [30] proposed a semi-automatic approach for move-method refactoring with the aim of removing Feature Envy smells, while Fokaefs *et al.* [31] proposed JDeodorant, a tool for extract-class refactoring in presence of God Classes.

B. Testability and Unit Testing Strategies

The software testing literature reports several definitions of testability, all related to the testing effort and cost. The IEEE standard glossary [32] defines testability as *"the degree to which a system or component facilitates the establishment of test criteria and performance of tests to determine whether those criteria have been met."* Bache and Mullerburg [33] defined testability in terms of the effort required for testing. They measured this effort using the number of test cases required to satisfy a given coverage criterion.

Software testability is an important software quality attribute that can significantly contribute to facilitate testing activities and to reduce testing effort and costs [34], [35]. Many works investigated factors that influence testability and proposed techniques to improve this attribute [36], [37].

Testability is defined and measured at different levels of testing: unit testing [12], integration testing, system testing [36]; and also for different artifacts, *i.e.*, design [38] and source code [12]. Bruntink *et al.* [12] proposed the usage of a pool of OO class metrics—namely LOC, Fan-Out, Number of Fields (NOF), Number of Methods (NOM), Response For Classes (RFC), and Weighted Method Complexity (WMC)—to predict the testability of a class.

Following the definition provided by Bache and Mullerburg [33], we used the number of test cases required to test a class according to the MaDUM testing strategy as our testability measure. Thus, this measure has been used to evaluate the testability of classes participating in APs, compared to that of other classes. However, it is important to notice that there are numerous OO unit testing criteria focusing on detecting different kinds of defects or on detection efficiency [8]. Criteria such design by contract, pre and post conditions first proposed for procedural code have been adapted to OO [9]. Other criteria such as state-based techniques [8], [10] are quite powerful [39], although they require the availability of specific design artifacts, namely state machine diagrams.

Overall, to the best of our knowledge, no previous work investigated the impact of APs on class testability and testing effort, nor conjectured the possibility of a testability-driven refactoring, or provided evidence that APs negatively impact testability.

VIII. Conclusion and Future Work

This paper investigated the effects of antipatterns (APs) on the cost of class unit testing in object-oriented systems and on how refactoring activities could possibly reduce testing cost. We detected APs using the DECOR tool [7] in four Java programs, namely Ant 1.8.3, ArgoUML 0.20, Checkstyle 4.0, and JFreeChart 1.0.13. Then, we estimated the number of test cases required to test each class using the MaDUM class unit testing strategy [6]. Finally, we compared the number of test cases required for classes that participate in APs or not and related such a number of test cases with the number of post-release fixed defects of such classes.

Findings of the study strongly support the evidence that:

1) classes participating in APs are, in general, more expensive to test than other classes;
2) specifically, classes participating in some kinds of APs, such as Blobs, Anti-Singleton or Complex Classes have a significantly higher testing cost than other classes, whereas some APs, such as Method Chains or Lazy Classes, do not strongly contribute to the testing cost. Interestingly, this contradicts findings related to AP defect-proneness, indicating Method Chains among the APs with the highest defect-proneness [5];
3) giving a high priority to classes participating in APs makes testing activities more cost-effective because they contain a higher proportion of defects than other classes.

In addition, we showed how appropriate refactoring activities can be adopted for classes participating in APs—but also for some other classes—with the aim of reducing the testing cost. These kinds of refactoring are different, complementary, and in some cases can pursue conflictual objectives with respect to traditional refactoring actions aimed at improving comprehensibility and maintainability.

Future work aims at (i) extending our evaluation to class integration by accounting for integration ordering strategies [16], [15] and related test integration; (ii) investigating the feasibility of automatic or semi-automatic approaches to promote refactoring and refactoring aimed at reducing the testing cost and in general testability-driven refactoring; and (iii) extending the empirical study to more programs.

References

[1] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[2] 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*, 1st ed. John Wiley and Sons, March 1998.

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

[4] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc, "An exploratory study of the impact of code smells on software change-proneness," in *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE)*. IEEE CS Press, October 2009.

[5] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Softw. Eng.*, vol. 17, no. 3, pp. 243–275, 2012.

[6] I. Bashir and A. L. Goel, *Testing Object-Oriented Software: Life-Cycle Solutions*, 1st ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2000.

[7] 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," *IEEE Transactions on Software Engineering*, vol. 36, pp. 20–36, 2010.

[8] R. V. Binder, *Testing object-oriented systems: models, patterns, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[9] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: automated testing based on java predicates," in *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '02. New York, NY, USA: ACM, 2002, pp. 123–133.

[10] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Trans. Softw. Eng.*, vol. 4, no. 3, pp. 178–187, May 1978.

[11] 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*, ser. CSMR '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 181–190.

[12] M. Bruntink and A. van Deursen, "An empirical study into class testability," *J. Syst. Softw.*, vol. 79, no. 9, pp. 1219–1232, 2006.

[13] R. Grissom and J. Kim, *Effect sizes for research: a broad practical approach*. Lawrence Erlbaum Associates, 2005.

[14] S. Holm, "A simple sequentially rejective Bonferroni test procedure," *Scandinavian Journal of Statistics*, vol. 6, pp. 65–70, 1979.

[15] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima, "Class firewall, test order, and regression testing of object-oriented programs," *JOOP*, vol. 8, no. 2, pp. 51–65, 1995.

[16] L. Briand, J. Feng, and Y. Labiche, "Experimenting with genetic algorithms and coupling measures to devise optimal integration test orders," *Software Engineering with Computational Intelligence Kluwer*, 2003.

[17] Y.-G. Gueheneuc and G. Antoniol, "DeMIMA: A multi-layered approach for design pattern identification," *IEEE Transactions on Software Engineering*, vol. 34, pp. 667–684, 2008.

[18] Y.-G. Guéhéneuc, "Ptidej: Promoting patterns with patterns," in *Proceedings of the* 1st *ECOOP workshop on Building a System using Patterns*, M. E. Fayad, Ed. Springer-Verlag, July 2005.

[19] G. Antoniol, Kamel Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement? a text-based approach to classify change requests," in *Proceedings of the* 18th *IBM Centers for Advanced Studies Conference (CASCON)*, M. Vigder and M. Chechik, Eds. ACM Press, October 2008.

[20] A. Bachmann, C. Bird, F. Rahman, P. T. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. ACM, 2010, pp. 97–106.

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

[22] W. J. Brown, R. C. Malveau, H. W. M. III, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, T. Hudson, Ed. John Wiley & Sons, Inc., 1998.

[23] G. Travassos, F. Shull, M. Fredericks, and V. R. Basili, "Detecting defects in object-oriented designs: using reading techniques to increase software quality," in *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '99. New York, NY, USA: ACM, 1999, pp. 47–56.

[24] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ser. ICSM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 350–359.

[25] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "BDTEX: A GQM-based bayesian approach for the detection of antipatterns," *J. Syst. Softw.*, vol. 84, no. 4, pp. 559–572, 2011.

[26] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aimeur, "SMURF: a SVM-based incremental anti-pattern detection approach," in *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press, October 2012.

[27] I. S. Deligiannis, I. Stamelos, L. Angelis, , M. Roumeliotis, and M. Shepperd, "A controlled experiment investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 72, no. 2, pp. 129 – 143, 2004.

[28] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 390–400.

[29] M. Fowler, *Refactoring: improving the design of existing code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[30] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Trans. Software Eng.*, vol. 35, no. 3, pp. 347–367, 2009.

[31] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "JDeodorant: identification and application of extract class refactorings," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011, pp. 1037–1039.

[32] IEEE, "IEEE standard glossary of software engineering terminology," 1990.

[33] R. Bache and M. Mullerburg, "Measures of testability as a basis for quality assurance," *Softw. Eng. J.*, vol. 5, no. 2, pp. 86–92, 1990.

[34] R. V. Binder, "Design for testability in object-oriented systems," *Commun. ACM*, vol. 37, no. 9, pp. 87–101, 1994.

[35] L. C. Briand, Y. Labiche, and H. Sun, "Investigating the use of analysis contracts to improve the testability of object-oriented code," *Softw. Pract. Exper.*, vol. 33, no. 7, pp. 637–672, 2003.

[36] S. Jungmayr, "Testability measurement and software dependencies," 2002.

[37] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Trans. Softw. Eng.*, vol. 30, no. 1, pp. 3–16, 2004.

[38] B. Baudry and Y. L. Traon, "Measuring design testability of a UML class diagram," *Inf. Softw. Technol.*, vol. 47, no. 13, pp. 859–879, 2005.

[39] L. C. Briand, M. Di Penta, and Y. Labiche, "Assessing and improving state-based class testing: A series of experiments," *IEEE Trans. Software Eng.*, vol. 30, no. 11, pp. 770–793, 2004.