Noname manuscript No.
(will be inserted by the editor)

An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness

Foutse Khomh, Massimiliano Di Penta,
Yann-Gaël Guéhéneuc, and Giuliano Antoniol

Received: date / Accepted: date

Abstract
Context: Antipatterns are poor design choices that are conjectured to make object-oriented systems harder to maintain.
Aim: We investigate the impact of antipatterns on classes in object-oriented systems by studying the relation between the presence of antipatterns and the change- and fault-proneness of the classes.
Method: We detect 13 antipatterns in 54 releases of ArgoUML, Eclipse, Mylyn, and Rhino, and analyse (1) to what extent classes participating in antipatterns have higher odds to change or to be subject to fault-fixing than other classes, (2) to what extent these odds (if higher) are due to the sizes of the classes or to the presence of antipatterns, and (3) what kinds of changes affect classes participating in antipatterns.
Results: We show that, in almost all releases of the four systems, classes participating in antipatterns are more change- and fault-prone than others. We also show that size alone cannot explain the higher odds of classes with antipatterns to underwent a (fault-fixing) change than other classes. Finally, we show that structural changes affect more classes with antipatterns than others. We provide qualitative explanations of the

We thank Marc Eaddy for making his data on faults freely available. This work has been partly funded by the NSERC Research Chairs in Software Change and Evolution and in Software Patterns and Patterns of Software.

Foutse Khomh
Ptidej Team
Département d'informatique et de recherche opérationnelle
Université de Montréal, QC, Canada
E-mail: foutsekh@iro.umontreal.ca

Massimiliano Di Penta
Department of Engineering
University of Sannio, Benevento, Italy
E-mail: dipenta@unisannio.it

Giuliano Antoniol and Yann-Gaël Guéhéneuc
SOCCER Lab. and Ptidej Team
Département de génie informatique et génie logiciel
École Polytechnique de Montréal, QC, Canada
E-mail: {giuliano.antoniol,yann-gael.gueheneuc}@polymtl.ca

increase of change- and fault-proneness in classes participating in antipatterns using release notes and bug reports.
Conclusions: The obtained results justify *a posteriori* previous work on the specification and detection of antipatterns and could help to better focus quality assurance and testing activities.

Keywords Antipatterns, Mining Software Repositories, Empirical Software Engineering.

1 Context and Problem

Antipatterns—such as those presented in [6]—have been proposed to embody poor design choices. These antipatterns stem from experienced software developers' expertise and are conjectured in the literature to negatively impact systems by making classes more change-prone and–or fault-prone. They are opposite to design patterns [18], *i.e.*, they identify "poor" solutions to recurring design problems, for example Brown's 40 antipatterns describe the most common pitfalls in the software industry [6]. 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 many studies on antipatterns summarised in Section 6, only a few studies empirically analysed the impact of antipatterns on source code-related phenomena [5, 36], in particular class change- and fault-proneness, even though such phenomena directly impact the developers' work.

Examples of Antipatterns. In practice, antipatterns are in-between design and implementation: they concern the design of one or more classes, but they concretely manifest themselves in the source code as classes through specific code smells [17]. Often, antipatterns are defined in terms of thresholds imposed on metric values [30, 41]. An example of antipattern is the LazyClass, which occurs when a class does too little, *i.e.*, has few responsibilities in a system. A LazyClass is a class with few methods and fields; its methods have little complexity. It often stems from speculative generality during a system design and–or implementation.

Another example of antipattern is the MessageChain, which occurs when the realisation of a functionality of a class requires a long chain of method invocations between objects of different classes. A MessageChain is conjectured to impact change- and fault-proneness because of the high number of indirections. Classes using message chains are detected by computing the number of transitive invocations of a class to other classes.

The two previous antipatterns are quite simple. A more complex example of antipattern is the Blob. A Blob, also called God Class, is a large and complex class that centralises the behaviour of a portion of a system and only uses other classes as data holders, *i.e.*, data classes. A Blob prevents the use of polymorphism through inheritance, making changes more complex and risk-prone. A class is a Blob if it has a low cohesion, it is large, some of its method names recall procedural programming, and it is associated to data classes, which only provide fields and–or accessors to their fields.

Goal and Process. In this paper, using data mined from version control systems, we study whether classes participating in an antipattern have an increased likelihood to change than other classes between any two given releases. Also, by combining data from version control and issue-tracking systems, we assess whether classes participating in

Table 1 Summary of the characteristics of the analysed systems.

Systems	Releases (#)		Classes	LOCs	Changes	Fault-fixing Changes
ArgoUML	0.10.1–0.26.2	(10)	792–1,841	128,585–316,971	40,409	2,064
Eclipse	1.0–3.3.1	(13)	4,647–17,167	781,480–3,756,164	196,193	24,335
Mylyn	1.0.1–3.1.1	(18)	1,625–2,762	207,436–276,401	36,328	118
Rhino	1.4R3–1.6R6	(13)	89–270	30,748–79,406	6,925	1,068

antipatterns have a higher likelihood than others to be involved in issues documenting faults; we use a set of faults that, in some cases, have been manually validated by a third-party, as explained in Section 2. We also study the possible effect of class sizes on the results of our study by comparing the sizes of classes participating in antipatterns with those of other classes. Finally, we study the kinds of changes affecting classes participating in antipatterns.

Study. We perform the study on 10 releases of ArgoUML, 13 of Eclipse, 18 of Mylyn, and 13 of Rhino, and across the changes and fault-fixing changes occurring between the releases. We detect 13 antipatterns in the classes of these systems (see Section 2.2) to investigate their relations with change- and fault-proneness. We show that antipatterns *do* have a negative impact on class change- and fault-proneness and that certain kinds of antipatterns *do* have a higher impact than others. We also show that size alone cannot explain the higher change- and fault-proneness of classes participating in antipatterns. We finally discuss the kinds of changes that affect classes participating or not in antipatterns, *i.e.*, addition/deletion of methods/attributes, changes of method signatures and method implementation.

Relevance. Understanding if antipatterns increase the likelihood of classes to change or to be subject to fault-fixing is important from both researchers' and practitioners' points of view. We show that the presence of antipatterns is related to an increase of class change- and fault-proneness. We also bring evidence that, like design patterns [2,4,12,45], particular kinds of antipatterns are more correlated to change- and fault-proneness than others. Therefore, within the limits of its threats to validity, this study provides quantitative evidence that antipatterns indeed may affect the developers' work negatively and, thus possibly, software evolution. Thus, we justify *a posteriori* previous work on antipatterns and prove to be true the conjecture from the literature on the negative impact of antipatterns.

We also provide evidence to practitioners—developers, quality assurance personnel, and managers—of the importance and usefulness of antipattern detection techniques to assess class change- and fault-proneness. With the availability of such information, a tester could decide to focus on classes participating in antipatterns, because she knows that such classes are likely to contain faults. Similarly, a manager could use such techniques to assess the volume of classes participating in antipatterns in a to-be-acquired system and, thus, adjust her offer and forecast the system cost-of-ownership and–or plan for refactorings.

Organisation. Section 2 describes the empirical study definition and design. Section 3 presents the study results. Section 4 and 5 discusses the results and the threats to their validity. Section 6 relates our study with previous work. Finally, Section 7 concludes the paper and outlines future work.

2 Study Definition and Design

The *goal* of our study is to investigate the relation between classes participating in antipatterns and their change- and fault-proneness as well as the kinds of changes impacting antipatterns. The *quality focus* is the source code change- and fault-proneness, which, if high, can have a concrete effect on developers' effort and on the overall project development and maintenance cost and time.

The *perspective* is that of researchers, interested in the relation between antipatterns and evolution phenomena in software systems. Also, results can be of interest to developers, who perform development or maintenance activities and need to take into account and forecast their effort, and to testers, who need to know which classes are important to test. Finally, they can be of interest to managers and–or quality assurance personnel, who could use antipattern detection techniques to assess the future changes and faults of in-house or to-be-acquired source code to better quantify its cost-of-ownership.

The *context* of this study consists in the change history and issue-tracking systems of four Java systems[1]. ArgoUML is an open source UML-based system design tool. Eclipse is an open-source integrated development environment. It is a platform used both in open-source communities and in industry. Mylyn is a plug-in for Eclipse, which aims at reducing information overload and making developers' multi-tasking easier. Rhino is an open-source implementation of a JavaScript interpreter.

The four systems have different sizes and belong to different domains. Eclipse is a large system (release 3.3.1 is larger than 3.5 MLOCs) and, therefore, close to the size of many real industrial systems. It is also developed partly by a commercial company, IBM, and thus is likely to embody industrial practices. ArgoUML, Mylyn, and Rhino have wide ranges of sizes, are open-source, and also have different architectures. Specifically, ArgoUML is a monolithic system, Eclipse has a plugin-based architecture, Mylyn is an Eclipse plugin, and Rhino a component of a larger system, *i.e.*, the Mozilla/Firefox Web browser. Previous studies—performed also on ArgoUML and Eclipse—suggested that systems exhibiting different architectures exhibited different change-proneness and underwent different kinds of changes [2, 12].

Table 1 summarises the main characteristics of the systems: the first and last analysed releases, the numbers of releases considered, the system sizes ranges in LOCs, and the overall numbers of considered changes and fault-fixing changes. (Detailed figures are available in a technical report [23]; fault classification for Mylyn is only available for the first three releases [14].)

We do not include release 2.1 of Eclipse in our study because we observed that the number of committed changes and fixed faults between release 2.1 and 2.1.1 is about one order of magnitude smaller than those numbers between any other two subsequent releases. Also, the number of classes did not substantially change between 2.0, 2.1, and 2.1.1. Finally, the period of time between 2.1 and 2.1.1 is also shorter (three months) than those between other pairs of releases. Thus, we preferred to consider the period between releases 2.0 and 2.1.1 as one "release" for consistency.

For the four systems, it is relevant to study the relation between antipatterns, change- and fault-proneness, and class sizes, because the percentages of classes participating in antipatterns are not negligible. Figure 1 shows that these percentages vary

[1] `http://argouml.tigris.org/`, `http://www.eclipse.org`, `http://www.eclipse.org/mylyn/`, and `http://www.mozilla.org/rhino/`

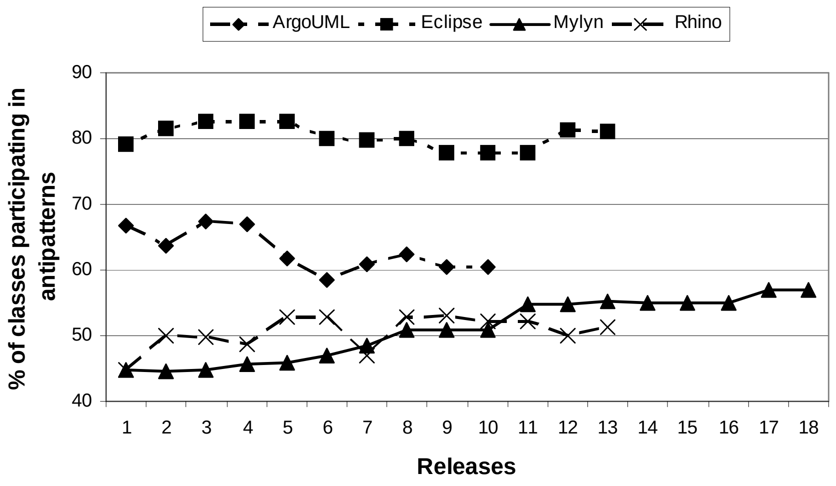

Fig. 1 Percentages of classes participating in antipatterns in the releases of the four systems.

Table 2 Distribution of antipatterns in the analysed releases.

Antipatterns	Number of Antipatterns in First and Last Releases (in parentheses, the percentages of participating classes)			
	ArgoUML	Eclipse	Mylyn	Rhino
AntiSingleton	352 (44.44)–3 (0.16)	330 (7.10)–1784 (10.39)	4 (0.25)–127 (4.60)	16 (17.98)–1 (0.37)
Blob	26 (3.28)–116 (6.30)	600 (12.91)–2,194 (12.78)	40 (2.46)–93 (3.37)	0 (0)–0 (0)
CDSBP	136 (17.17)–51 (2.77)	382 (8.22)–2,285 (13.31)	61 (3.75)–183 (6.63)	4 (4.49)–17 (6.30)
ComplexClass	42 (5.30)–103 (5.59)	511 (11.00)–2,125 (12.38)	29 (1.78)–72 (2.61)	6 (6.74)–14 (5.56)
LargeClass	56 (7.07)–166 (9.02)	1 (0.02)–8 (0.05)	43 (2.65)–99 (3.58)	9 (10.11)–19 (7.04)
LazyClass	16 (2.02)–44 (2.39)	2,403 (51.71)–8,561 (49.87)	2 (0.12)–18 (0.65)	4 (4.49)–9 (3.33)
LongMethod	172 (21.72)–348 (18.90)	2,372 (51.04)–7,956 (46.34)	134 (8.25)–349 (12.64)	14 (15.73)–35 (12.96)
LPL	195 (24.62)–300 (16.30)	1,087 (23.39)–3,233 (18.83)	43 (2.65)–95 (3.44)	9 (10.11)–8 (2.96)
MessageChain	79 (9.97)–166 (9.02)	1,043 (22.44)–3,041 (17.71)	70 (4.31)–181 (6.55)	20 (22.47)–66 (24.44)
RPB	105 (13.26)–574 (31.18)	397 (8.54)–2,582 (15.04)	45 (2.77)–290 (10.50)	5 (5.62)–11 (4.07)
SpaghettiCode	9 (1.14)–22 (1.20)	2 (0.04)–1 (0.01)	12 (0.74)–39 (1.41)	0 (0.00)–2 (0.74)
SG	0 (0.00)–0 (0.00)	54 (1.16)–228 (1.33)	0 (0.00)–0 (0.00)	0 (0.00)–0 (0.00)
SwissArmyKnife	0 (0.00)–0 (0.00)	67 (1.44)–96 (0.56)	1 (0.06)–0 (0.00)	0 (0.00)–0 (0.00)

across releases in the four systems and that it is always higher than 45%, with peaks as high as 80%. We further report that classes participating in antipatterns participate, in average, to 2 antipatterns in ArgoUML, 3 in Eclipse, 2 in Mylyn, and 2 in Rhino and to, in maximum, between 7 and 9 antipatterns in ArgoUML, 13 and 24 antipatterns in Eclipse, 6 and 7 antipatterns in Mylyn, and 5 and 7 antipatterns in Rhino. (Detailed data is available elsewhere [23].)

Table 2 shows the distribution of the antipatterns of interest, detailed in Section 2.2. A cell in the table reports on the left side of the dash (respectively, on its right), the number of classes in the first release of a given system (respectively, its last), which participates in a given antipattern, followed by their percentages with respect to the total numbers of classes. For example, the cell at the intersection of the ArgoUML column and the AntiSingleton row reports that in its first release, 352 classes were AntiSingleton, representing 44.44% of the total number of its classes, while in the last release, only 3 classes were AntiSingleton, representing 0.16% of the total number of its classes. Percentages go as high as 51.71% of classes participating in LazyClass in the first release of Eclipse.

2.1 Research Questions

Our study aims at addressing six null hypotheses, specifically concerning the relations between classes participating in antipatterns and their: change-proneness (RQ1 and RQ2), fault-proneness (RQ3 and RQ4), size (RQ5), and kinds of changes (RQ6).

RQ1. *What is the relation between antipatterns and change-proneness?* We investigate whether classes participating in at least one antipattern are more change-prone than others, by testing the null hypothesis: H_{01}*: the proportion of classes undergoing at least one change between two releases is not different between classes in antipatterns or not.*

RQ2. *What is the relation between kinds of antipatterns and change-proneness?* We analyse whether certain antipatterns imply more changes than others, by testing the null hypothesis: H_{02}*: classes participating in certain antipatterns are not more change-prone than others.*

RQ3. *What is the relation between antipatterns and fault-proneness?* This research question focuses on the relation between antipatterns and fault-fixing issues. The null hypothesis is: H_{03}*: the proportion of classes undergoing at least one fault-fixing change between two releases does not differ between classes participating or not in at least one antipattern.*

RQ4. *What is the relation between particular kinds of antipatterns and fault-proneness?* We also analyse the influence of kinds of antipatterns on fault-proneness, by testing the null hypothesis: H_{04}*: classes participating in certain kinds of antipatterns are not more prone to fault-fixing than other classes.*

RQ5. *Does the presence of antipatterns in classes relate to the sizes of these classes?* This research question stems from El Emam *et al.* [15] findings showing that many metrics correlate to size. Specifically, we study whether the higher change- and–or fault-proneness of classes participating in antipatterns is due to their sizes (in terms of LOCs) or to the presence of the antipatterns, by testing the hypothesis: H_{05}*: classes participating in antipatterns are not larger than other classes.*

RQ6. *What kind of changes are performed on classes participating or not in antipatterns?* We study whether classes participating in antipatterns undergo more (or less) structural changes (addition/removal/change of/to attributes, addition/removal of methods, or changes to the methods signatures) than other kinds of changes by testing the hypothesis: H_{06}*: classes participating in antipatterns do not undergo a number of structural changes different than other kinds of changes.*

Hypotheses H_{01} to H_{05} are one-tailed because we are interested in investigating only whether antipatterns relate to an *increase* of change-proneness, fault-proneness, and size. Hypothesis H_{06} is two-tailed because we investigate whether the presence of antipatterns is related to a higher or a lower number of structural changes.

2.2 Independent Variables

We use our previous approach, DECOR (Defect dEtection for CORrection) [30, 40, 41], to specify and detect antipatterns. DECOR is based on a thorough domain analysis of code smells and antipatterns in the literature, from which is built a domain-specific language. This language uses rules to describes antipatterns, with different types of properties: lexical (*e.g.*, class names), structural (*e.g.*, classes declaring public static variables), internal (*e.g.*, number of methods), and the relation among properties (*e.g.*, association, aggregation, and composition relations among classes). Using this language, DECOR proposes the descriptions of several antipatterns. It also provides algorithms and a framework, DeTeX, to convert antipattern descriptions automatically into detection algorithms. DeTeX allows detecting occurrences of antipatterns in systems written in various object-oriented programming languages, such as Java.

Moha *et al.* [40] showed that the current detection algorithms obtained from DECOR ensure 100% recall and have precisions between 41.1% and 87% for three antipatterns: Blob, SpaghettiCode, and SwissArmyKnife [40]. The detection algorithms for these three antipatterns have an average accuracy of 99% for the Blob, of 89% for the SpaghettiCode, and of 95% for the SwissArmyKnife; and a total average of 94%. In the following, we focus on 13 antipatterns from [6, 17]:

- AntiSingleton: A class that provides mutable class variables, which consequently could be used as global variables.
- Blob: A class that is too large and not cohesive enough, that monopolises most of the processing, takes most of the decisions, and is associated to data classes.
- ClassDataShouldBePrivate (CDSBP): A class that exposes its fields, thus violating the principle of encapsulation.
- ComplexClass: A class that has (at least) one large and complex method, in terms of cyclomatic complexity and LOCs.
- LargeClass: A class that has (at least) one large method, in term of LOCs.
- LazyClass: A class that has few fields and methods (with little complexity).
- LongMethod: A class that has a method that is overly long, in term of LOCs.
- LongParameterList (LPL): A class that has (at least) one method with a too long list of parameters with respect to the average number of parameters per methods in the system.
- MessageChain: A class that uses a long chain of method invocations to realise (at least) one of its functionality.
- RefusedParentBequest (RPB): A class that redefines inherited method using empty bodies, thus breaking polymorphism.
- SpaghettiCode: A class declaring long methods with no parameters and using global variables. These methods interact too much using complex decision algorithms. This class does not exploit and prevents the use of polymorphism and inheritance.
- SpeculativeGenerality (SG): A class that is defined as abstract but that has very few children, which do not make use of its methods.
- SwissArmyKnife: A class whose methods can be divided in disjunct set of many methods, thus providing many different unrelated functionalities.

We choose only these antipatterns because (1) they are well-described by Brown [6], (2) we could find enough of their occurrences in several releases of several of the studied systems, and (3) they are representative of design and implementation problems with data, complexity, size, and the features provided by classes. The specifications of these antipatterns are outside of the scope of this paper and available in [29].

Our independent variables are the number of classes participating in the 13 antipatterns. In our computations, we use variables $AP_{i,j,k}$, which indicate the number of times that a class i participates in an antipattern j in a release k. For RQ1 and RQ3, we aggregate these variables into a Boolean variable $AP_{i,k}$ indicating if a class i participates or not in any antipattern.

2.3 Dependent Variables

Dependent variables measure the phenomena related to classes participating in antipatterns.

RQ1 and RQ2. Change-proneness refers to whether a class underwent *at least a change* between release k (in which it was participating in some antipatterns) and the subsequent release $k+1$. Changes are identified, for each class in a system, by looking at commits in their control-version systems (CVS or SVN). For the sake of simplicity, we assumed to have one class per file. This assumption could introduce an error in case of non-public top-level classes and inner classes. We did not find any inner class participating in any antipattern in the analysed releases of the systems. Non-public top-level classes are rare and did not participate in any antipattern.

RQ3 and RQ4. Fault-proneness refer to whether a class underwent *at least a fault-fixing change* between releases k and $k+1$. Fault fixing changes are documented in text reports that describe different kinds of problems in a system. They are usually posted in issue-tracking systems—*e.g.*, Bugzilla for the four 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, adding features, restructuring, and so on. We trace faults/issues to changes by matching their IDs in the commits [16].

For Mylyn and Rhino, we consider a set of manually-validated and publicly-available faults [14]. For ArgoUML, issues dealing with fixing faults are marked as "DEFECT" in the issue tracking system[2]. For Eclipse, such a "DEFECT" tag was not used and, given the high number of issues (34,634 between releases 1.0 and 3.4), a manual classification is not practical. Thus, we consider issues posted on the Eclipse Bugzilla that (1) are referred to as *"Bug <issueID>"* in the CVS commits, (2) have the *Resolution* field set to "FIXED" or the *Status* field set to "CLOSED", *i.e.*, they indeed required some changes, and (3) are not tagged as "Enhancement" in the *Severity* field. Our choice, however, does not guarantee that *all* the considered issues are fault-fixing issues.

RQ5. We measure the sizes of classes participating or not in antipatterns using their LOCs, excluding comments and blank lines. Each classes is associated with its size, the total number of antipatterns and the kinds of antipatterns in which it participates. Abstract and native methods and methods declared in interfaces count for zero LOC as they do not have a body (or have a body not implemented in Java).

RQ6. We count the number of *structural* and *non-structural* changes occurring in antipattern classes vs. other classes between two releases k and $k+1$. As in previous

[2] `http://argouml.tigris.org/issues`

work [2,12], we consider as *structural* changes those changes that would alter the class interface, *i.e.*, addition/removal/change of/to attributes, addition/removal of methods, or changes to the method signatures, *i.e.*, change of return type, exception(s) being thrown, parameter type, addition/removal/change of parameters. We consider as *non-structual* changes those related to method bodies. Changes were identified using an analyzer developed with JavaCC[3] (and used in our previous work), which extracts class diagram models from source code, and a Perl script, which identifies differences between two models. Further kinds of changes—*e.g.*, those modifying exception-handling code—could also be considered, although we opt for a lightweight analysis because (1) we perform it on all revisions of all classes and (2) our focus is to identify the main changes affecting classes participating in antipatterns.

2.4 Analysis Method

RQ1 and RQ3. We study whether changes to and faults in a class are related to the class participating in antipatterns, regardless of the kinds of antipatterns. Therefore, we test whether the proportions of classes exhibiting (or not) at least one change/fault significantly vary between classes participating in antipatterns and other classes. We use Fisher's exact test [13] for H_{01} and H_{03}. We did not consider releases where either only antipattern or non-antipattern classes changed because of a very small number of changes (*e.g.*, less than 10).

We also compute the *odds ratio* (OR) [13] indicating the likelihood of an event to occur, *e.g.*, change. OR is defined as the ratio of the odds p of an event occurring in one sample, *i.e.*, the set of classes participating in some antipatterns (experimental group), to the odds q of it occurring in the other sample, *i.e.*, the set of classes participating in no antipattern (control group): OR $= \frac{p/(1-p)}{q/(1-q)}$. An odds ratio of 1 indicates that the event (*e.g.*, change) is equally likely in both samples. OR > 1 indicates that the event is more likely in the first sample (experimental group of classes participating in some antipatterns) while an OR < 1 indicates the opposite (control group of classes not participating in any antipatterns).

RQ2 and RQ4. We want to understand the relation of specific kinds of antipatterns with changes and faults. Let us focus on RQ2 and changes. We use a logistic regression model [20] to correlate the presence of antipatterns with changes. While in other contexts, *e.g.*, [19], such a model was used for prediction purposes; as in [36,45], we use it as an alternative to the Analysis Of Variance (ANOVA) for dichotomous dependent variables to test H_{02} and H_{04}.

In a logistic regression model, the dependent variable is commonly a dichotomous variable and, thus, it assumes only two values $\{0,1\}$, *e.g.*, changed or not. The multivariate logistic regression model is based on the formula:

$$\pi(X_1, X_2, \ldots, X_n) = \frac{e^{C_0 + C_1 \cdot X_1 + \ldots + C_n \cdot X_n}}{1 + e^{C_0 + C_1 \cdot X_1 + \ldots + C_n \cdot X_n}} \tag{1}$$

where:

– X_i are characteristics describing the modelled phenomenon, in our case the number of classes participating in an antipattern of kind i.

[3] `http://javacc.dev.java.net/`

– $0 \leq \pi \leq 1$ is a value on the logistic regression curve. The closer the value is to 1, the higher is the probability that a class participating in this kind of antipattern underwent a change.

Then, we count, for each antipattern, the number of times that, across the analysed releases, the p-values obtained by the logistic regression are significant. We use $t = 75\%$ (as in current state of the art literature [10,44]) to assess whether classes participating in a specific kind of antipattern have significantly greater odds to change than others: If these classes are more likely to change in more than t releases, then we say that this antipattern has a significant impact on increasing the change-proneness.

RQ5. We perform the analysis related to RQ5 in three steps. First, we compare, for each release, the average size of (1) classes participating in at least one antipattern and (2) classes participating in no antipattern. We use the Mann-Whitney test and compute Cohen d effect size [9]. For independent samples and unpaired analyses, the Cohen d effect size is the difference between the means M_1 and M_2 divided by the pooled standard deviation $\sigma = \sqrt{(\sigma_1^2 + \sigma_2^2)/2}$ of both groups: $d = (M_1 - M_2)/\sigma$. The effect size is small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$, and large for $d \geq 0.8$ [9]. We expect the test results to be statistically significant and the odds ratios to be greater or equal to 1 because many antipatterns are, according to their definitions, related to size, *e.g.*, Blob, ComplexClass, and LargeClass.

Second, we perform the same test and compute the same odds ratios between the set of classes participating in each antipattern and those not participating in any antipattern. We expect that, for some antipatterns, the test would not be significant and–or the odds ratios would be lower than 1. Indeed, while the definitions of some antipatterns directly relate to their size, others specifically target small classes, *e.g.*, LazyClass, or are orthogonal to size, *e.g.*, ClassDataShouldBePrivate.

Third, we again perform Fisher's exact test and compute the odds ratios between large classes participating or not to size-related antipatterns, *i.e.*, Blob, ComplexClass, and LargeClass. We single out the classes whose sizes are greater than the 75% percentile and divide them in two sets: those participating in the considered antipatterns and those that do not participate in these antipatterns. We expect that classes participating in Blob, ComplexClass, and LargeClass antipatterns are not significantly larger than the largest classes.

RQ6. We again use Fisher's exact test to compare the proportions of structural changes in classes participating in antipatterns with those of non-structural changes, also in classes not participating in any antipattern.

3 Study Results

This section reports the results of our empirical study, which are further discussed in Section 4. Detailed results can be found in a technical report [23] while raw data is available on-line[4].

[4] `http://www.ptidej.net/downloads/experiments/emse10`

Table 3 Change-proneness ORs. Releases where Fisher's exact test did not show significant differences are highlighted in gray; ORs< 1 are also highlighted in gray.

Change Proneness							
ArgoUML		Eclipse		Mylyn		Rhino	
Releases	Odds Ratios	Releases	Odds Ratios	Releases	Odds Ratios	Releases	Odds Ratios
0.10.1	4.17	**1.0**	1.13	1.0.1	10.51	1.4R3	10.41
0.12	7.16	2.0	**0.75**	2.0M1	10.37	1.5R1	17.98
0.14	6.22	2.1.1	2.59	2.0M2	7.38	1.5R2	17.37
0.16	15.84	2.1.2	1.42	2.0M3	206.60	1.5R3	15.71
0.18.1	10.00	2.1.3	1.15	2.0	14.17	1.5R4	16.19
0.20	26.54	3.0	**0.88**	2.1	10.89	1.5R41	30.71
0.22	8.83	3.0.1	**0.86**	2.2.0	11.10	1.5R5	15.51
0.24	15.40	3.0.2	**0.89**	2.3.0	9.83	1.6R1	24.73
0.26	3.98	3.2	2.19	2.3.1	7.66	1.6R2	12.69
0.26.2	6.75	3.2.1	1.94	2.3.2	24.38	1.6R3	19.95
		3.2.2	1.47	3.0.0	9.45	1.6R4	33.05
		3.3	2.43	3.0.1	9.85	1.6R5	19.97
		3.3.1	1.42	3.0.2	5.31	1.6R6	20.56
				3.0.3	8.18		
				3.0.4	3.77		
				3.0.5	4.96		
				3.1.0	10.53		
				3.1.1	5.59		

3.1 RQ1: What is the relation between antipatterns and change-proneness?

Table 3 summarises the odds ratios when testing H_{01}. Each row shows, for each system, a release number and the ORs of classes participating in at least one antipattern in that release to exhibit at least one change before the next release. In all releases, except Eclipse 1.0, Fisher's exact test indicates a significant difference of proportions between change-prone classes among those participating and not in antipatterns.

Odds ratios vary across systems and, within each system, across releases. While in few cases, ORs are close to 1, *i.e.*, the odds is even that a class participating in an antipattern changes or not, in some pairs of systems/releases, such as ArgoUML 0.20, Mylyn 2.0M3, or Rhino 1.5R41, ORs are greater than 25. Overall, ORs for Eclipse are lower than those of other systems, by one or two orders of magnitude. The odd ratios of classes participating in some antipatterns to change are, in most cases, higher than that of other classes.

We therefore conclude that, in most cases, *there is a relation between antipatterns and change-proneness*: a greater proportion of classes participating in antipatterns change with respect to other classes. The rejection of H_{01} and the ORs provide *a posteriori* concrete evidence of the impact of antipatterns on change-proneness.

3.2 RQ2: What is the relation between kinds of antipatterns and change-proneness?

Table 4 summarises the results of the logistic regression for the relations between change-proneness and the different kinds of antipatterns. A cell in the table reports

Table 4 Number (percentage) of releases where each antipattern significantly correlates with change-proneness.

Antipatterns	Change Proneness			
	ArgoUML	Eclipse	Mylyn	Rhino
AntiSingleton	**8 (80%)**	5 (38%)	7 (39%)	–
Blob	2 (20%)	8 (62%)	9 (50%)	–
CDSBP	3 (30%)	7 (54%)	9 (50%)	6 (46%)
ComplexClass	2 (20%)	**12 (92%)**	2 (11%)	–
LargeClass	2 (20%)	–	4 (22%)	4 (31%)
LazyClass	5 (50%)	**12 (92%)**	3 (17%)	1 (8%)
LongMethod	**10 (100%)**	**12 (92%)**	**17 (94%)**	5 (38%)
LPL	**9 (90%)**	**10 (77%)**	7 (39%)	3 (23%)
MessageChain	**10 (100%)**	**12 (92%)**	**18 (100%)**	**13 (100%)**
RPB	**9 (90%)**	6 (46%)	10 (56%)	5 (38%)
SpaghettiCode	–	–	–	–
SG	–	3 (23%)	6 (33%)	1 (8%)
SwissArmyKnife	–	6 (46%)	–	–

the number (and percentage) of releases, for a given system, in which the participation of classes in a given antipattern significantly correlate with change-proneness. For example, the cell at the intersection of the ArgoUML column and the AntiSingleton row indicates that, in 8 releases of ArgoUML out of 10 (80%), classes participating in the AntiSingleton antipattern were significantly more change-prone than other classes.

From Table 4, we can reject H_{02} for some antipatterns, *i.e.*, for antipatterns that are significantly correlated to change-proneness in at least 75% of the releases, highlighted in gray. Following our analysis method, only MessageChain has a significant impact on change-proneness in all systems: classes participating in this antipattern are more likely to change than classes participating in other or no antipattern in more than 75% of the releases. Other antipatterns have significant impact on a subset of the systems: LongMethod in ArgoUML, Eclipse, and Mylyn; LongParameterList in ArgoUML and Eclipse; AntiSingleton and RefusedParentBequest in ArgoUML; Complexclass and LazyClass in Eclipse.

We conclude that *there is a relation between kinds of antipatterns and change-proneness* but not for all antipatterns and not consistently across systems and releases.

3.3 RQ3: What is the relation between antipatterns and fault-proneness?

Table 5 summarises Fisher's exact test results and ORs for H_{03}. Similarly to Table 3, each row shows, for each system, a release number and the ORs of classes participating to at least one antipattern in that release to exhibit at least one fault-fixing change before the next release. The differences in proportions are significant and thus we can reject H_{03} in all cases. The proportion of classes participating in antipatterns and reported in faults is between 1.32 and 31.29 times larger than that of other classes.

Odds ratios for faults are not always higher than those for changes: although classes participating in antipatterns are more likely to exhibit fault-fixing changes than other classes, they seem to be even more likely to undergo restructuring changes in addition to fault-fixing changes than other classes.

Therefore, we conclude that *there is a relation between antipatterns and fault-proneness*; although this relation is not as strong as the relation with change-proneness.

Table 5 Fault-proneness ORs. Releases where Fisher's exact test did not show significant differences are highlighted in gray.

Fault Proneness							
ArgoUML		Eclipse		Mylyn		Rhino	
Releases	Odds Ratios	Releases	Odds Ratios	Releases	Odds Ratios	Releases	Odds Ratios
0.10.1	4.43	**1.0**	1.14	1.0.1	10.45	1.4R3	6.44
0.12	4.87	2.0	2.06	2.0M1	17.70	1.5R1	31.29
0.14	17.53	2.1.1	2.19	2.0M2	>>300	1.5R2	–
0.16	6.58	2.1.2	2.27	2.0M3	–	1.5R3	13.93
0.18.1	5.33	2.1.3	2.75	2.0	–	1.5R4	9.06
0.20	4.95	3.0	3.30	2.1	–	1.5R41	30.05
0.22	9.42	3.0.1	2.12	2.2.0	–	1.5R5	10.57
0.24	2.25	3.0.2	1.75	2.3.0	–	1.6R1	29.26
0.26	8.08	3.2	3.55	2.3.1	–	1.6R2	–
0.26.2	9.73	3.2.1	2.54	2.3.2	–	1.6R3	–
		3.2.2	2.41	3.0.0	–	1.6R4	23.00
		3.3	2.90	3.0.1	–	1.6R5	13.29
		3.3.1	1.17	3.0.2	–	1.6R6	–
				3.0.3	–		
				3.0.4	–		
				3.0.5	–		
				3.1.0	–		
				3.1.1	–		

Table 6 Number (percentage) of releases where each antipattern significantly correlates with fault-proneness.

Antipatterns	Fault Proneness			
	ArgoUML	Eclipse	Mylyn	Rhino
AntiSingleton	5 (50%)	**11 (84%)**	–	–
Blob	1 (10%)	6 (46%)	–	–
CDSBP	2 (20%)	7 (54%)	2 (66%)	3 (33%)
ComplexClass	–	**13 (100%)**	1 (33%)	–
LargeClass	3 (30%)	–	–	3 (33%)
LazyClass	–	**12 (92%)**	–	2 (22%)
LongMethod	1 (10%)	**13 (100%)**	–	3 (33%)
LPL	5 (50%)	7 (54%)	2 (66%)	3 (33%)
MessageChain	7 (70%)	**10 (77%)**	1 (33%)	**7 (78%)**
RPB	4 (40%)	3 (23%)	1 (33%)	–
SpaghettiCode	–	–	–	–
SG	–	3 (23%)	–	1 (11%)
SwissArmyKnife	–	3 (23%)	–	–

3.4 RQ4: What is the relation between particular kinds of antipatterns and fault-proneness?

Table 6 reports the results of the logistic regression for the relations between fault-proneness and kinds of antipatterns. Similarly to Table 4, a cell in the table reports the number (and percentage) of releases, for a given system, in which the participation of classes in a given antipattern significantly correlates with fault-proneness. For Mylyn, we could analyse only 3 releases for fault-proneness and for Rhino, only 9 releases, because of the limited number of faults occurring in some releases (< 10). We can reject H_{04} for MessageChain in Eclipse and Rhino; AntiSingleton, ComplexClass, LazyClass, and LongMethod in Eclipse.

We conclude that *there is a relation between kinds of antipatterns and fault-proneness* but not for all antipatterns and not consistently across systems and releases.

3.5 RQ5: Do the presence of antipatterns in classes relate to the sizes of these classes?

We found that, as expected, classes participating in some specific kinds of antipatterns are significantly larger than classes not participating in antipatterns (with a *medium* to *large* effect size). (Detailed results are reported in the technical report [23].) Yet, we observe the following exceptions:

– Classes participating in AntiSingleton are not significantly larger than classes not participating in any antipattern in 10 out of 18 Mylyn releases;
– Classes participating in LazyClass are significantly smaller than other classes in all the analysed releases of ArgoUML, Mylyn, and Rhino. This observation was expected because, by definition, LazyClasses are small;
– Classes participating in RefusedParentBequest are not significantly larger than classes not participating in any antipattern in 1 out of 10 ArgoUML releases, 15 out of 18 Mylyn releases, and 9 out of 13 Rhino releases;
– Classes participating in SpeculativeGenerality are not significantly larger than classes not participating in any antipattern in 5 out of 10 ArgoUML releases, all 18 Mylyn releases, and all 13 Rhino releases.

In Eclipse, all kinds of antipatterns have a significantly larger size than classes not participating in any antipattern, although for the above-mentioned antipatterns the effect size was generally *small* while for the others it was *medium* to *small*.

Table 7 ORs of change- and fault-proneness for large classes participating in the Blob, LargeClass, ComplexClass antipatterns with respect to large classes not participating in any antipattern (highlighted values indicate statistical significance of the Fisher's exact test).

ArgoUML			Eclipse			Mylyn			Rhino		
Rel.	ORs (Changes)	ORs (Faults)	Rel.	ORs (Changes)	ORs (Faults)	Rel.	ORs (Changes)	ORs (Faults)	Rel.	ORs (Changes)	ORs (Faults)
0.10.1	–	–	1.0	0.98	3.07	1.0.1	0.92	1.09	1.4R3	–	–
0.12	–	0.79	2.0	0.73	0.86	2.0M1	**5.60**	–	1.5R1	–	–
0.14	**7.81**	–	2.1.1	1.79	0.91	2.0M2	1.46	–	1.5R2	5.76	–
0.16	–	3.41	2.1.2	1.78	2.20	2.0M3	–		1.5R3	5.51	17.72
0.18.1	2.32	3.05	2.1.3	1.58	1.04	2.0.0	**2.56**		1.5R4	–	–
0.20	–	0.45	3.0	1.22	1.86	2.1	1.17		1.5R41	–	–
0.22	2.64	0.57	3.0.1	0.96	2.01	2.2.0	2.85		1.5R5	3.10	**13.52**
0.24	5.26	–	3.0.2	0.93	1.08	2.3.0	0.77		1.6R1	1.67	**13.52**
0.26.2	**3.46**	1.85	3.2	**2.18**	**3.46**	2.3.1	**3.89**		1.6R2	8.53	–
0.26	**3.10**	1.32	3.2.1	**2.56**	**2.24**	2.3.2	3.95		1.6R3	2.30	–
			3.2.2	**1.52**	**2.14**	3.0.0	**5.39**		1.6R4	–	–
			3.3	**8.08**	–	3.0.1	1.16		1.6R5	3.63	4.19
			3.3.1	**2.48**	**1.35**	3.0.2	**2.89**		1.6R6	–	–
						3.0.3	1.04				
						3.0.4	1.66				
						3.0.5	2.38				
						3.1.0	–				
						3.1.1	**3.27**				

Finally, Table 7 reports results of the Fisher's exact test (highlighted ORs are statistically significant), comparing change- and fault-proneness of classes having a size greater than the 75% percentile of the overall size distribution and participating or not in the Blob, ComplexClass, and LargeClass antipatterns. For example, for Eclipse release 1.0, the ORs of large classes participating (or not) to Blob, ComplexClass, and LargeClass are 0.98 for change-proneness and 2.12 for fault-proneness. ORs are greater to 1 in:

1. ArgoUML: 6 out of 10 releases for change-proneness and 4 out of 7 releases for fault-proneness;
2. Eclipse: 9 out of 13 releases for change-proneness and 8 out of 13 releases for fault-proneness (plus 2 other cases where the ORs are just above one);
3. Mylyn: 13 out of 15 releases for change-proneness (plus another case where the ORs is just above one), while it was not possible to get statistically-significant results for fault-proneness, due to the limited number of detected occurrences;
4. Rhino: 7 out of 8 releases for change-proneness and 4 out of 5 releases for fault-proneness.

However, the Fisher's exact test only reported statistical significance in a small number of cases—due to the limited number of classes having a size above the 75% percentile of the distribution.

ORs are always above one (in most cases above two) and reach 13.66 every time the Fisher's exact test found a statistically-significant difference in the proportions of changes and fault-fixing changes between classes participating or not in size-related antipatterns. Therefore, *large classes participating in antipatterns change more and are more fault-prone than large classes not participating in any antipatterns.*

We conclude that classes participating in antipatterns are generally larger than other classes. This conclusion was expected because, many antipatterns, such as Blob, ComplexClass, LargeClass, or LongMethod, stem from an excessively large size *and* of other negative characteristics of the classes. We also conclude that, except for some releases of the analysed systems, some kinds of antipatterns (AntiSingleton, LazyClass, RefusedParentBequest, and SpeculativeGenerality) describe symptoms of poor design that are unrelated to size.

We thus generally conclude that *some kinds of antipatterns are related to size as expected by their definitions but size only does not explain the classes greater change- and fault-proneness.*

3.6 RQ6: What kind of changes are performed on classes participating or not in antipatterns?

While studying the relation between kinds of antipatterns and change- and fault-proneness, we also studied the kinds of changes impacting classes participating in antipatterns. Table 8 reports Fisher's exact test results and odds ratios for the proportions of *structural* changes to classes participating in antipatterns with respect to those of other classes. For simplicity's sake, and because there are no substantial variations across releases, we report results obtained by aggregating data from the whole history of each system, rather than for each release separately.

Table 8 shows that classes participating in antipatterns in Rhino do not undergo more structural changes than other changes: it is a small system and, therefore, the different kinds of changes may occur to any class. Although, we can reject H_{06} for Eclipse, the OR ≈ 1 downplays this rejection, which we explain by the extensive use of inheritance in Eclipse [2], leading to few structural changes to classes.

Table 8 Proportion of changes to antipattern classes vs. other classes: Fisher's exact test result and ORs.

Systems	p-values	ORs
ArgoUML	< 0.01	1.22
Eclipse	< 0.01	1.03
Mylyn	< 0.01	1.19
Rhino	0.08	1.04

Table 9 Summary of our findings.

	ArgoUML	Eclipse	Mylyn	Rhino	Explanations
RQ1	✓	∼	✓	✓	Future changes could impact several classes
RQ2 – LongMethod	✓	✓	✓	×	Complex code likely requires more changes
RQ2 – MessageChains	✓	✓	✓	✓	Chains of messages possibly increase the change impact
RQ3	✓	✓	✓	✓	Antipatterns may increase the risk that developers introduce faults
RQ4 – MessageChains	×	✓	×	✓	Long chains of messages reduce the developers' view of the context, which may lead to more faults
RQ5	✓	✓	✓	✓	Size, per se, does not explain the change-/fault-proneness of classes participating in antipatterns
RQ6	✓	✓	✓	✓	Classes participating in antipatterns are more subject to changes impacting their interface and, thus, possibly their change- and fault-proneness

Detailed analysis for different kinds of antipatterns reveal that, for all antipatterns except LazyClass in ArgoUML, Mylyn, and Rhino, and RefusedParentBequest in Eclipse, classes participating in antipatterns undergo more structural changes than others changes (*e.g.*, changes in the method implementations). The methods implementations of LazyClasses, as reported in Section 3.2, change to increase their behaviour. Changes in the method organisations and implementations of RefusedParentBequest are generally performed to correct them.

We conclude that *structural changes occur more often on classes participating to antipatterns than other changes.*

4 Discussion

We now discuss the results using the releases' histories. We also discuss the relation between antipattern and developers' intent and refactoring activities. Table 9 summarises our findings.

4.1 Correlations among Antipatterns

We analysed whether a correlation exists between the presence of different antipatterns and, hence, between the rules and metrics used in their definitions (*i.e.*, the metrics thresholds and values in each revisions of each system). We used the non-parametric Spearman correlation, which results indicate that Blob, ComplexClass, and LargeClass are lowly correlated ($0.5 < \rho < 0.7$ [9]) in all releases of ArgoUML, Mylyn, and Rhino, but in none of Eclipse. For all other antipatterns and releases, we obtained no correlation among antipatterns ($\rho \ll 0.5$). We expected that we would not find correlations among antipatterns because their definitions are different and capture different types of design pitfalls. We also analysed whether we could find a correlation between the presence of different antipatterns and traditional object-oriented metrics, such as Chidamber and Kemerer's metric suite. As expected, we could not find a correlation that was consistent across the systems and their releases because antipatterns are higher level than metrics, thus showing that antipatterns, albeit detected using metrics, bring different information to developers than metrics.

4.2 Statistical Significance/Unexpected ratios

Tables 3 and 5 show that, in most cases, classes belonging to antipatterns are more change- and fault-prone than others. However, there is a case were H_{01} could not be rejected for lack of statistical significance and four cases with unexpected ORs, which indicate that classes participating in antipatterns changed less than others (highlighted in the tables).

We explain the lack of statistical significance for Eclipse 1.0 by the major changes between releases 1.0 and 2.0, which imply that many classes were added/changed (Eclipse size increased from 781 to 1,250 KLOCs and 4,647 to 6,742 classes), irrespective of their participation in antipatterns.

The first case with an unexpected OR concerns classes having changed between Eclipse 2.0 and 2.1.1, with $OR = 0.75$. Eclipse 2.1 introduced several new features with respect to 2.0, including navigation history, sticky hovers, prominent status indication, and so on.

The second, third, and fourth cases concern classes having changed between releases 3.0 and 3.2. Eclipse 3.0 was a major improvement over the 2.x series, with a new runtime platform implementing the OSGi R3.0 specifications[5] to become a Rich Client Platform and support any type of tooling (not necessarily an IDE). Eclipse 3 had many problems at first, corrected in the subsequent 3.0.1, 3.0.2, and 3.2 releases. No less than $15,153 - 11,166 = 3,987$ classes were added between 3.0 and 3.2, which did not only participate to antipatterns. Eclipse size increased by $3,271 - 2,260 = 1,011$ KLOCs.

4.3 Changes/Faults Odds Ratios

For ArgoUML, change-proneness ORs are never smaller than 3.98. The highest OR occurs between releases 0.20 and 0.22, period during which a major restructuring[6]

[5] `http://www.eclipse.org/osgi/`

[6] `http://argouml.tigris.org/servlets/NewsItemView?newsItemID=1675`

took place with many faults fixed and 293 issues resolved. ORs for fault-fixing are high but often lower than those for change-proneness, which suggests that antipatterns are potential symptoms of change-proneness, but not necessarily of fault-proneness: they make a system harder to maintain because future changes will likely impact several classes, but only indirectly impact fault-proneness. The highest fault-related OR occurs between releases 0.14 and 0.16, period during which many fault-fixing activities took place. Release 0.16 is the release with the highest number of fault-fixing changes [23]: 851, the second-highest is release 0.26 with 591.

For Eclipse, we found lower ORs than those of other systems for both change- and fault-proneness. We explain such a difference by the fact that $\sim 80\%$ of Eclipse classes participate in at least one antipattern, with a higher proportion of these classes to be LazyClasses (*e.g.*, 51.71% in the first release). Therefore, we expected to find lower ORs because Eclipse includes many more classes participating in antipatterns than not. The high proportion of LazyClasses conforms to the results of previous studies [2], which observed that Eclipse is designed to evolve through sub-classing, which, in turn, leads to a lower class change-proneness.

Eclipse is the only system with greater ORs for fault/issue- than change-proneness. We recall that we considered issues, as discussed in Section 2.3, and that as discussed in our previous study [1], a majority of Eclipse issues are likely *not* related to faults but to other maintenance activities, such as restructuring. Thus, it is consistent to find more classes impacted by issues with respect to faults only.

Verifying H_{01} for Mylyn between releases 2.0M3 and 2.0 results in an extreme OR = 206.60, which we explain by the amount of issues fixed between the releases: 304[7]. Table 5 shows that antipatterns are correlated with fault-fixing changes. The OR reflects this relation plus that with other changes unrelated to faults, such as restructuring, which impacted classes in antipatterns.

For Rhino, ORs for change-proneness range between 10.41 in release 1.4R3 and 33.05 in 1.6R4, two numbers which we explain by (1) the number of new features added in release 1.5R1: many classes not participating in antipatterns were added/changed and (2) the number of issues between releases 1.6R3, 1.6R4, and 1.6R5: respectively 4, 7, and 24[8]. More faults have been filled against 1.6R4 than other releases, thus explaining the change of ORs.

4.4 Kinds of Antipatterns and Changes/Faults

Tables 4 and 6 show that antipatterns impact change- and fault-proneness but that we could not reject H_{02} or H_{04} for all of them, in particular LargeClass, Blob, ClassDataShouldBePrivate, SpaghettiCode, SpeculativeGenerality, and SwissArmyKnife. We explain this fact by the low number of classes participating in these antipatterns; for examples, on average, in Eclipse, there are 479 LargeClasses for 11,618 classes per release and in ArgoUML, 80 for 960 classes per release [23]. The number of occurrences of the SpaghettiCode is even lower, with, on average, 2 per Eclipse release. No SpaghettiCode was found in ArgoUML, Mylyn, and Rhino.

Eclipse, ArgoUML, Mylyn, and Rhino use extensively object orientation. They “divide to conquer”, which helps to avoid: Blob, which is a class that knows/does

[7] `http://eclipse.org/mylyn/new/new-2.0.html`

[8] `https://bugzilla.mozilla.org/buglist.cgi?query_format=specific&order=relevance+desc&bug_status=__all__&product=Rhino&content={1.6R3|1.6R4|1.6R5}`

too much; LargeClass and SwissArmyKnife, which are complex classes that provides too many services; and SpeculativeGenerality, which is an abstract class with very few children. The use of polymorphism and encapsulation explains the few number of ClassDataShouldBePrivate, which occurs when the data encapsulated by a class is public, as well as the SpaghettiCode, which is a class with too many long methods with too many branches.

Among the remaining antipatterns, we rejected, for MessageChain, H_{02} for all systems and H_{04} for Eclipse and Rhino. The MessageChain antipattern characterises classes that use long chains of calls to perform their functionality, which makes them dependent on classes "far" from each other. Finding many occurrences of the MessageChain is not surprising in Eclipse and Rhino. Eclipse has thousands of classes; developers fixing issues are likely to touch many classes because of their dependencies with one another and the likelihood of faults related to these dependencies is high. Rhino is small but the classes forming its parse tree and interpreter are tightly coupled.

The other antipatterns satisfy the conditions to reject H_{02} or H_{04} for at least one system. By tracking their occurrences through releases, we found that antipatterns are generally removed from the systems while new ones are introduced. Thus, some antipatterns are in small number or are absent in some releases; thus, the logistic regression analysis indicated that some antipatterns are statistically significant only in some releases.

Classes participating in the antipatterns ComplexClass and LazyClass are more change- and fault-prone than others in Eclipse. ComplexClass characterises classes with a higher number of complex methods than the average class, thus developers adding new features or fixing issues are more likely to touch these classes, which consequently increases their likelihood to have faults. This observation confirms Fowler and Brown's warnings about complex classes. Lazy classes tend to be removed or changed to increase their behaviour, while others are introduced: there were 2,765 lazy classes in Eclipse 1.0 (59% of the system), 8,967 in 3.3.1 (52%). Class `org.eclipse.search.internal.core.SearchScope`, for example, was a lazy class in 1.0 but, in 3.0, 2 methods and 2 constructors were added and the inner class `WorkbenchScope` was removed. New lazy classes, *e.g.*, `org.eclipse.team.internal.ccvs.ui.actions.ShowEditorsAction`, were introduced.

Classes participating in AntiSingleton are more fault/issue-prone in Eclipse and more change-prone in ArgoUML than other classes. They are generally removed from the system or changed. In Eclipse, 16% of the AntiSingleton classes were removed between releases 1.0 and 3.0 and only 53% of the classes were still AntiSingleton in that release; the other classes were changed. For example, all methods of `org.eclipse.compare.internal.CompareWithEditionAction`, an AntiSingleton, were removed between releases 1.0 and 3.0 and the class became a LazyClass with no behaviour.

We can reject H_{02} for LongMethod for ArgoUML, Eclipse, and Mylyn, and H_{04} for Eclipse. LongMethod classes are more change-prone than any other class, and more fault-prone than other Eclipse classes, possibly because such classes are complex and thus more likely to change to fix issues. Faults are also more likely to be introduced when changing these classes due to their complexity. Moreover, we observe that LongMethod classes keep on participating in this antipattern during their evolution and are, in general, central to the system core features. Previous studies, *e.g.*, [2], confirm that central classes are more change-prone.

Classes participating in RefusedParentBequest are more change-prone than others in ArgoUML, possibly due to the need for re-organising badly organised hierarchies:

this antipattern occurs when a subclass does not use attributes and–or public/protected methods inherited from its parent. We expected this results because ArgoUML implements deep hierarchies of models, diagram elements, and tools.

Although classes participating in antipatterns are more change- and fault-prone than other classes; in some situations, an antipattern may be the best and possibly only way to implement some requirements and–or functionalities. For example, one of the LargeClass in Eclipse is class `org.eclipse.swt.internal.win32.OS`, which is the unique access point to the underlying Windows platform for the Standard Widget Toolkit. While the class is large, it provides a unique access point to non-object-oriented, platform-dependent resources, thus increasing portability and possibly efficiency.

5 Threats to Validity

With our study, we show that antipatterns do impact the change- and fault-proneness of classes and that certain kinds of antipatterns have a greater impact than others. However, we do not claim that antipatterns *cause* changes and faults. Indeed, our study cannot say anything about the *reasons* for classes to have antipatterns and, consequently, the reasons for changes/faults to occur/appear in these classes. We only empirically verified that classes with antipatterns are more change- and fault-prone that others, thus confirming the conjecture in the literature.

In addition, some parts of the source code of a system will always change as new functionalities are added and as faults are fixed. Thus, we cannot use only the correlation between antipatterns and change proneness to *predict* which classes will change in the future. Indeed, the fact that some classes are more likely to change in a release has complex reasons that are beyond the scope of this paper, as already noted by Zimmermann *et al.* [50].

We now discuss the threats to validity of our study following the guidelines for case study research [48]. *Construct validity* threats concern the relation between theory and observation; in this study, they are mainly due to measurement errors. The identification of changes is reliable because based on the CVS/SVN change logs. Yet, it may not reflect exactly the commits related to a (fault-fixing) change and developers' efforts accurately because developers follow different patterns for committing their changes, *e.g.*, from committing changes as faults are fixed to committing all changes once a week. However, these varying patterns do not affect our measure of change-proneness because we just observed whether a class underwent at least one change during a given period of time.

We were able to identify fault-fixing changes for ArgoUML, Mylyn, and Rhino using an existing classification [14]. The only cases where some fixed issues might not be related to faults is Eclipse, as we pointed out in our previous work [1]. We mitigated the use of possibly erroneous faults by discarding issues explicitly labeled as "Enhancements" and focusing on issues marked as "FIXED" or "CLOSED" because they required some changes. It is unlikely, in Eclipse, that hard-to-fix issues would stay longer "OPENED" than others, because Eclipse is being backed up by IBM, which strives to offer a stable product.

We did not include release 2.1 of Eclipse but its inclusion is unlikely to change our results: between releases 1.0 and 2.0, we observed 11,632 class change commits, and 1,541 fault fixings; between 2.0 and 2.1, 21,211 class change commits and 1,756 fault fixing; while between 2.1 and 2.1.1 only 3,664 change commits and 240 fault fixings. We thus observed that the number of committed changes and fixed faults between release 2.1 and 2.1.1 is about one order of magnitude smaller than those numbers between any other two subsequent releases. Moreover, we analyzed the odds ratios of classes participating in antipatterns to exhibit changes/faults, with respect to classes not participating in antipatterns, between releases 2.1 and 2.1.1. The Fischer's exact test indicated a significant difference, with odds ratios of 2.59 for change proneness and of 1.59 for fault proneness, thus consistent with the results obtained when analyzing other pairs of subsequent releases.

Finally, we observe that DECOR includes its authors' subjective understanding of the antipatterns and that the accuracy of its detection algorithms is not perfect [40]. DECOR accuracy impacts our results because we may have classified a class not participating in an antipattern as participating in it and vice-versa. Other techniques and tools should be used to confirm our findings.

Threats to *internal validity* do not affect this study, being an exploratory study [48]. We do not claim causation, but relate the presence of antipatterns with the occurrences of changes, faults, and issues. Nevertheless, we tried to explain—by looking at specific changes, commit notes, and change histories—why some antipatterns could have been the cause of changes/issues/faults. We are aware that antipatterns can be dependent to each other and relied on the logistic regression model-building procedure to select the subset of non-correlated antipatterns. When studying antipatterns, we did not exclude that, in a particular context, an antipattern can be the best way to implement or design a (part of a) system. For example, automatically-generated parsers are often Spaghetti Code, *i.e.*, very large classes with a high number of very long methods with many branches. However, this observation does not impact our study because we only consider antipatterns as warnings.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the performed statistical tests. Also, we mainly used non-parametric tests that do not require to make assumption about the data set distribution. For Mylyn, we are aware that fault-proneness is analysed on only 3 releases for which the manual fault classification is available [14], thus it would be difficult to draw strong conclusions for this system about the relation between antipatterns and fault-proneness.

Reliability validity threats concern the possibility of replicating this study. We attempted to provide all the necessary details to replicate our study. Moreover, the source code repositories and issue-tracking systems of the studied systems are available to obtain the same data. The raw data used to compute the statistics is available on-line[4].

Threats to *external validity* concern the possibility to generalise our results. First, we studied four systems having different sizes and belonging to different domains. Nevertheless, further validation on a larger set of systems is desirable, considering systems from different domains, as well as several systems from the same domain, to better analyze the cross-domain and inter-domain influence of antipatterns on change- and fault-proneness. Second, we used a particular yet representative subset of antipatterns. Different antipatterns could lead to different results in future work.

6 Related Work

We now discuss work on antipatterns, design patterns, and metrics, in relation to software evolution.

Code Smells/Antipatterns Definition and Detection. The first book on "antipatterns" in object-oriented development was written in 1995 by Webster [47]; his contribution includes conceptual, political, coding, and quality-assurance problems. Riel [34] defined 61 heuristics characterising good object-oriented programming to assess a system quality manually and improve its design and implementation. Beck [17] defined 22 code smells, suggesting where developers should apply refactorings. Mäntylä [27] and Wake [46] proposed classifications for code smells. Brown *et al.* [6] described 40 antipatterns, including the well-known Blob and Spaghetti Code. These books provide in-depth views on heuristics, code smells, and antipatterns aimed at industrial and academic audiences. They are the basis of all approaches to detect (semi-)automatically code smells and antipatterns, such as DECOR [40] used in this study.

Several approaches to specify and detect code smells and antipatterns exist in the literature. They range from manual approaches, based on inspection techniques [42], to metric-based heuristics [28,31,33], using rules and thresholds on various metrics or Bayesian belief networks [39].

Some approaches for complex software analysis use visualisation [11,35]. Visualisation is an interesting compromise between fully automatic detection techniques, which are efficient but loose track of the context, and manual inspections, which are slow and subjective [24]. However, visualisation requires human expertise and is thus time-consuming. Other approaches perform fully automatic detection and use visualisation techniques to present their results [25,43].

This previous work significantly contributed to the specification and detection of antipatterns. The approach used in this study, DECOR, builds on this previous work to offer a method to specify and automatically detect antipatterns.

Code Smells/Antipatterns and Software Evolution. Deligiannis *et al.* [21,22] proposed the first quantitative study of the relation between antipatterns and software quality. They performed a controlled experiments with 20 students on two systems to understand the impact of God Classes on the understandability and maintainability of systems. The results of their study suggested that God Classes affect the evolution of design structures and considerably affects the subjects' use of inheritance.

Du Bois *et al.* [5] showed that the decomposition of God Classes into a number of collaborating classes using well-known refactorings can improve comprehension. They did not consider source code evolution phenomena.

Li *et al.* [26] investigated the relationship between the probability of a class to be faulty and some antipatterns based on three versions of Eclipse and showed that classes with the antipatterns God Class, Shotgun Surgery, and Long Method have a higher probability to be faulty than other classes. They concluded on the need for broader studies to validate their results. We provide such a broader study by studying the impact of 13 antipatterns on change- and fault-proneness in four systems.

Olbrich *et al.* [32], analysed the historical data of Lucene and Xerces over several years and concluded that God Classes and Shotgun Surgery have a higher change frequency than other classes; with God Classes featuring more changes. They neither performed an analysis to control the effect of the size on their results nor studied the kinds of changes affecting these antipatterns.

Using Azureus and Eclipse, Khomh *et al.* [36] studied the impact of classes with code smells on change-proneness and the particular impact of certain code smells. They showed that the likelihood for classes with code smells to change is very high, except in a few explainable cases. Some of the studied code smells are similar to some of the antipatterns studied in this paper, although antipatterns identify poor design solutions at a higher level of abstraction than code smells. Considering that code smells make classes more change-prone, it is natural that classes having antipatterns are also more change-prone than classes without antipattern. However, the study reported in this paper was necessary to confirm this relation because previous work did not test the different impact between having one, two, or more particular code smells. It could have been possible that having some combinations of code smells could have rendered classes more difficult to change and, consequently, less change-prone than others. Also, in this paper, we add three additional systems (ArgoUML, Mylyn, and Rhino), we study fault-proneness in addition to change-proneness, and we analyse the particular kinds of changes occurring on classes participating in certain antipatterns.

This previous work raised the awareness of the community towards the impact of code smells and antipatterns on software development. We build on this previous work and propose a more detailed and extensive empirical study of the impact of antipatterns on code evolution phenomena.

Design Patterns and Software Evolution. While antipatterns are poor design choices, design patterns are recurring solutions to design problems, increasing reusability, expandability, and understandability [18]. Several authors studied the impact of design patterns on systems. Vokac [45] analysed the corrective maintenance of a large commercial system over three years and compared the fault rates of classes that participated in design patterns with those of other classes. He noticed that participating classes were less fault-prone than others with differences in fault rates ranging from 63 percent to 154 percent on average. He also noticed that the Observer and Singleton patterns are correlated with larger classes; that classes playing roles in Factory Method were more compact, less coupled, and less fault-prone than others classes; and that no clear tendency existed for Template Method. His work provided the first quantitative evidence of a relationship between design patterns and the fault-proneness of systems.

Bieman *et al.* [4] analysed four small and one large systems to evaluate the impact of design patterns on change-proneness and concluded that participating classes are rather more change-prone. Khomh and Guéhéneuc [37] performed an empirical study of the impact of the 23 design patterns from [18] on ten different quality characteristics and concluded that patterns do not necessarily promote reusability, expandability, and understandability. Other studies focused on the change-proneness and resilience to change of design patterns [2] and of classes playing a specific role in design patterns [12]. They concluded that design patterns and change-proneness are related.

In our previous work [38], we studied the impact on classes of playing zero, one, and two or more roles in design motifs. We used several metrics, including the number of past and future changes and the number of faults, to study the impact of playing roles. We showed, using a representative population of classes, that classes playing one, two or more roles are more change-prone than classes playing zero roles. We could not find any statistically significant impact of playing one role vs. two roles on change- and fault-proneness but classes playing two roles changed 1.52 times more than classes playing one role. We can relate the impact on class change- and fault-proneness of participating in two or more antipatterns with the impact of playing two or more roles in design motifs. Indeed, in both cases, the impacted classes are bigger than others and play either a central role in the functioning of the system (design motifs) or in the maintenance of the system (antipatterns). Future work should study the statistical relations between design motifs and antipatterns.

While this previous work investigated the impact of design patterns, we study the impact of antipatterns on code evolution phenomena. Vokac's work inspired this study in the use of logistic regression to analyse the correlations between antipatterns and change- and fault-proneness.

Metrics and Software Evolution. Several studies, such as Basili *et al.*'s seminal work [3], used metrics as quality indicators. Cartwright and Shepperd [7] conducted on an industrial C++ system (over 133 KLOCs) an empirical study that supported the hypothesis that classes in inheritance relations are more fault-prone than others. Consequently, Chidamber and Kemerer (C&K) DIT and NOC metrics [8] could be used to find classes likely to have higher fault rates. Gyimothy *et al.* [19] compared the capability of sets of C&K metrics to predict fault-prone classes in Mozilla, using logistic regression and other machine learning techniques. They concluded that CBO is the most discriminating metric. They also found LOC to discriminate fault-prone classes well. Zimmermann *et al.* [49] conducted an empirical study on Eclipse showing that a combination of complexity metrics can predict faults and that the more complex the code, the more faults. El Emam *et al.* [15] showed that, after controlling for the confounding effect of size, the correlation between metrics and faults disappeared: many metrics are correlated with size and, therefore, do not bring more information to predict fault-proneness than size.

In our study, we relate change- and fault-proneness to antipatterns rather than to metrics and control for the size effect. We do not claim that antipatterns are better predictor of change- and fault-proneness than metrics: we use them as abstractions of metrics, thus likely to be better indicators than metrics for developers because they refer to specific design and implementation styles. Antipatterns can tell the developers whether a design choice is "poor" or not, by means of thresholds defined over metrics and of lexical information.

For example, the Blob is defined by imposing some empirical thresholds upon values of cohesion metrics and other metrics, as shown in Listing 1, suggesting to the developers whether the class lack cohesiveness, is too large, and is associated to data classes. If we were to provide developers only with the values of LCOM5, NAD, and NMD, then they would have to judged by themselves whether such values are excessive or not and warrant the classes has being tagged as a Blob.

```
RULE_CARD : Blob {
    RULE : Blob { ASSOC: associated FROM: mainClass ONE TO: DataClass MANY };
    RULE : mainClass { UNION LargeClassLowCohesion ControllerClass };
    RULE : LargeClassLowCohesion { UNION LargeClass LowCohesion };
    RULE : LargeClass { (METRIC: NMD + NAD, VERY_HIGH, 0) };
    RULE : LowCohesion { (METRIC: LCOM5, VERY_HIGH, 20) };
    RULE : ControllerClass { UNION
           (SEMANTIC: METHODNAME, {Process, Control, Ctrl, Command, Cmd,
                                   Proc, UI, Manage, Drive})
           (SEMANTIC: CLASSNAME,  {Process, Control, Ctrl, Command, Cmd,
                                   Proc, UI, Manage, Drive, System, Subsystem}) };
    RULE : DataClass { (STRUCT: METHOD_ACCESSOR, 90) };
};
```

Listing 1 Specification of the Blob Antipattern

Another important issue about the potential usefulness of antipatterns is whether they provide more information than size. El Emam *et al.* [15] found that many metrics are correlated to size, thus antipatterns could also be correlated to size because they use metrics. However, as discussed in RQ5 (Section 3.5), we found that this is not the case for many antipatterns, *i.e.*, classes participating in some kinds of antipatterns are not significantly larger than other classes. Moreover, large classes participating in antipatterns are generally more change- and fault-prone than other large classes.

7 Conclusions and Future Work

In this paper, we provided empirical evidence of the negative impact of antipatterns on classes change- and fault-proneness in four systems: ArgoUML, Eclipse, Mylyn, and Rhino. We studied the odds ratios of changes, faults, and issues on classes participating (or not) in 13 antipatterns in (overall) 54 releases of the four systems. We showed, through the five research questions **RQ1–4**, that classes participating in antipatterns are significantly more likely to be subject to changes and to be involved in fault-fixing changes (issue-fixing changes for Eclipse) than other classes. We also showed that size alone cannot explain the participation of classes to antipatterns with **RQ5** and, thus, that antipatterns bring additional, complementary information to developers to analyse their systems. We also studied with **RQ6** the kinds of changes that impacted classes participating in antipatterns and other classes and found that, in ArgoUML and Mylyn, structural changes are more likely to occur in classes participating in antipatterns (although odds ratios are not high (≈ 1.2)) than in other classes, while it is not the case for Eclipse and Rhino. Furthermore, we analysed the correlations among antipatterns and found that the Blob, ComplexClass, and LargeClass are correlated with one another in all releases of ArgoUML, Mylyn, and Rhino, but in none of Eclipse. As expected, other antipatterns are unrelated.

This exploratory study provides, within the limits of its validity, evidence that classes participating in antipatterns are more change- and fault/issue-prone than classes not participating in antipatterns. The study also provides evidence to practitioners that they should pay attention to systems with a high number of classes participating in antipatterns, because these classes are more likely to contain faults and to be the subject of their change efforts. More specifically, managers and developers can use these results to guide maintenance activities: for example, they can recommend their developers to avoid MessageChain as this antipattern is consistently related with high fault and change rates.

Future work includes replicating this study on industrial systems, other than Eclipse, on systems developed using different languages, and with different antipatterns. We are also interested in studying to what extent similar systems—*e.g.*, development environments, parser generators, productivity tools—exhibit similar relationships between the presence of antipatterns and code change-/fault-proneness. We also plan to study the categorisation of classes as change-prone, error-prone, or none, and compute Type I and II errors to assess whether antipatterns perform better than metrics.

Future work also includes studying further the relations between antipatterns and other characteristics of systems, such as their architecture and the use of design patterns. Moreover, it would be desirable to use antipatterns—other than metrics—to build more accurate/informative change- and fault-prediction methods. Last, but not least, further investigation, devoted to mine change logs, mailing lists and issue reports, is desirable to seek evidence of cause–effect relationships between the presence of antipatterns—or the need to remove them—and the class change- and fault-proneness.

References

1. G. Antoniol, Kamel Ayari, M. Di Penta, Foutse Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement? a text-based approach to classify change requests. In M. Vigder and M. Chechik, editors, *Proceedings of the* 18^{th} *IBM Centers for Advanced Studies Conference (CASCON)*. ACM Press, October 2008. 15 pages.
2. L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta. An empirical study on the evolution of design patterns. In *Proc. of the the* 6^{th} *European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*, pages 385–394. ACM Press, 2007.
3. V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. on Software Engineering*, 22(10):751–761, 1996.
4. J. M. Bieman, G. Straw, H. Wang, P. W. Munger, and R. T. Alexander. Design patterns and change proneness: An examination of five evolving systems. In *9th International Software Metrics Symposium (METRICS'03)*, pages 40–49. IEEE CS Press, 2003.
5. B. D. Bois, S. Demeyer, J. Verelst, T. Mens, and M. Temmerman. Does god class decomposition affect comprehensibility? In *Proceedings of the IASTED International Conference on Software Engineering*, pages 346–355. IASTED/ACTA Press, 2006.
6. 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*. John Wiley and Sons, 1^{st} edition, March 1998.
7. M. Cartwright and M. Shepperd. An empirical investigation of an object-oriented software system. *IEEE Trans. Software Eng.*, 26(8):786–796, August 2000.
8. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans Software Eng.*, 20(6):476–493, June 1994.
9. J. Cohen. *Statistical power analysis for the behavioral sciences*. L. Earlbaum Associates, 1988.
10. S. D. Conte and R. L. Campbell. A methodology for early software size estimation. Technical Report SERC-TR-33-P, Purdue University, jan 1989.
11. K. Dhambri, H. Sahraoui, and P. Poulin. Visual detection of design anomalies. In *Proceedings of the* 12^{th} *European Conference on Software Maintenance and Reengineering, Tampere, Finland*, pages 279–283. IEEE CS Press, April 2008.
12. M. Di Penta, Luigi Cerulo, Y.-G. Guéhéneuc, and G. Antoniol. An empirical study of the relationships between design pattern roles and class change proneness. In *Proceedings of the* 24^{th} *International Conference on Software Maintenance (ICSM)*. IEEE CS Press, September–October 2008.
13. S. D.J. *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.
14. M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho. Do crosscutting concerns cause defects? *IEEE Trans. Software Eng.*, 34(4):497–515, July-August 2008.
15. K. E. Emam, S. Benlarbi, N. Goel, and S. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Software Eng.*, 27(7):630–650, July 2001.
16. M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32, Amsterdam Netherlands, September 2003. IEEE CS Press.
17. M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1^{st} edition, June 1999.
18. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994.
19. T. Gyimóthy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Software Eng.*, 31(10):897–910, 2005.
20. D. Hosmer and S. Lemeshow. *Applied Logistic Regression (2nd Edition)*. Wiley, 2000.

21. D. Ignatios, S. Ioannis, A. Lefteris, R. Manos, and S. Martin. A controlled experiment investigation of an object oriented design heuristic for maintainability. *Journal of Systems and Software*, 65(2), February 2003.
22. D. Ignatios, S. Martin, R. Manos, and S. Ioannis. An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 72(2), 2004.
23. 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. Technical report, Ecole Polytechnique de Montreal, September 2009.
24. G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *proceedings of the* 20^th *international conference on Automated Software Engineering*. ACM Press, Nov 2005.
25. M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.
26. W. Li and R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software*, 80(7), 2007.
27. M. Mantyla. *Bad Smells in Software - a Taxonomy and an Empirical Study*. PhD thesis, Helsinki University of Technology, 2003.
28. R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the* 20th *International Conference on Software Maintenance*, pages 350–359. IEEE CS Press, 2004.
29. N. Moha. *DECOR : détection et correction des défauts dans les systèmes orientés objet*. PhD thesis, Université de Montréal et Université de Lille, August 2008.
30. N. Moha, Y.-G. Guéhéneuc, A.-F. L. Meur, and L. Duchien. A domain analysis to specify design defects and generate detection algorithms. In *Proceedings of the* 11th *International Conference on Fundamental Approaches to Software Engineering*, pages 276–291. Springer-Verlag, 2008.
31. M. J. Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In *Proceedings of the* 11th *International Software Metrics Symposium*. IEEE Computer Society Press, September 2005.
32. 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 *Third International Symposium on Empirical Software Engineering and Measurement*, 2009.
33. R. Oliveto, F. Khomh, G. Antoniol, and Y.-G. Guéhéneuc. Numerical signatures of antipatterns: An approach based on b-splines. In R. Capilla, R. Ferenc, and J. C. Dueas, editors, *Proceedings of the* 14th *Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, March 2010.
34. A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.
35. F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR'01)*, page 30. IEEE CS Press, 2001.
36. Foutse 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.
37. Foutse Khomh and Y.-G. Guéhéneuc. Do design patterns impact software quality positively? In *Proceedings of the* 12th *Conference on Software Maintenance and Reengineering (CSMR)*. IEEE CS Press, April 2008.
38. Foutse Khomh, Y.-G. Guéhéneuc, and G. Antoniol. Playing roles in design patterns: An empirical descriptive and analytic study. In K. Kontogiannis and T. Xie, editors, *Proceedings of the* 25th *International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, September 2009. 10 pages.
39. Foutse Khomh, Stéphane Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui. A bayesian approach for the detection of code and design smells. In *Proceedings of the* 9th *International Conference on Quality Software (QSIC)*. IEEE CS Press, August 2009. 10 pages.
40. Naouel Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur. DECOR: A method for the specification and detection of code and design smells. *Transactions on Software Engineering (TSE)*, 2009.
41. Naouel Moha, Amine Mohamed Rouane Hacene, P. Valtchev, and Y.-G. Guéhéneuc. Refactorings of design defects using relational concept analysis. In R. Medina and S. Obiedkov, editors, *Proceedings of the* 4th *International Conference on Formal Concept Analysis (ICFCA)*. Springer-Verlag, February 2008.

42. 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* 14^{th} *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–56. ACM Press, 1999.
43. E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*. IEEE CS Press, Oct. 2002.
44. S. Vicinanza, T. Mukhopadhyay, and M. Prietula. Software-effort estimation: an exploratory study of expert performance. *Information Systems Research*, 2(4):243–262, dec 1991.
45. M. Vokac. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Transactions on Software Engineering*, pages 904–917, Dec. 2004.
46. W. C. Wake. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
47. B. F. Webster. *Pitfalls of Object Oriented Development*. M & T Books, 1^{st} edition, February 1995.
48. R. K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, London, 2002.
49. T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for Eclipse. In *Proceedings of the* 3^{rd} *ICSE International Workshop on Predictor Models in Software Engineering*. IEEE CS press, 2007.
50. 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*, pages 563–572. IEEE Computer Society, 2004.