An Exploratory Study of the Impact of Antipatterns on Software Changeability

Foutse Khomh[1], Massimiliano Di Penta[3],
Yann-Gaël Guéhéneuc[1], and Guiliano Antoniol[2]

[1]Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
[2] SOCCER Lab., DGIGL, École Polytechnique de Montréal, Canada
[3] University of Sannio, Dept. of Engineering, Benevento, Italy

E-mails: {foutsekh,guehene}@iro.umontreal.ca,
dipenta@unisannio.it, giuliano.antoniol@polymtl.ca

ABSTRACT

Antipatterns are poor design choices that make object-oriented systems hard to maintain by developers. In this study, we investigate if classes that participate in antipatterns are more change-prone than classes that do not. Specifically, we test the general hypothesis: classes belonging to antipatterns are not more likely than other classes to undergo changes, to be impacted when fixing issues posted in issue-tracking systems, and in particular to unhandled exceptions-related issues—a crucial problem for any software system. We detect 11 antipatterns in 13 releases of Eclipse and study the relations between classes involved in these antipatterns and classes change-, issue-, and unhandled exception-proneness. We show that, in almost all releases of Eclipse, classes with antipatterns are more change-, issue-, and unhandled-exception-prone than others. These results justify previous work on the specification and detection of antipatterns and could help focusing quality assurance and testing activities.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools And Techniques—*Object-oriented design methods*

General Terms

Design, Experimentation, Measurement

Keywords

Antipatterns, Mining Software Repositories, Empirical Software Engineering

1. CONTEXT AND PROBLEM

Antipatterns [7] are poor design choices that are conjectured in the literature to hinder object-oriented software evolution. They are opposite to design patterns [16]: they are "poor" solutions to recurring design problems. Antipatterns are composed of code smells, which are poor implementation choices. Code smells are to antipatterns what roles are to design patterns: classes participating in an antipattern may possess one ore more of the smells defining it. Fowler's 22 code smells [15] are symptoms of antipatterns, such as Brown's 40 antipatterns [7].

Consequently and 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 with specific code smells.

One example of an antipattern is the LazyClass, which occurs in classes with few responsibilities in a system. A LazyClass is revealed by classes with methods with little complexity and few declared methods and fields. A LazyClass is often the result of some speculative generality [17] during the design or implementation of the system.

Goal. Despite the many works on code smells and antipatterns, no previous work has contrasted the changeability of classes participating in antipatterns with other classes to study empirically the impact of antipatterns on software evolution. Because antipatterns concretely manifest themselves in the source code as code smells, we identify antipatterns from the source code to investigate the relations between antipatterns and three types of code evolution phenomena.

First, we study whether classes participating in an antipattern have more risk, *i.e.*, an increased probability, to change than other classes. Second, given that issue-tracking systems contain issues related to faults—plus other issues, *e.g.*, issues related to enhancement or restructuring, we want to understand if classes belonging to some antipatterns have more risk to be involved in issues. Third, we want to assess whether classes participating in antipatterns have more risk than others to be involved in the subset of issues documenting unhandled exceptions (UHE in the following).

Study. We present an exploratory study investigating the relations between antipatterns and classes changes, classes involved in issue-fixing, and classes involved in UHE issue

fixing, in 13 releases of Eclipse. We show that antipatterns *do* have a negative impact on classes, that certain kinds of antipatterns *do* impact classes more than others, and that classes involved in antipatterns present higher risk of UHE.

To deepen our understanding of the impact of antipatterns on changeability, we also show that some code smells defining the antipatterns are more symptomatic of changes, issues, and UHE, for example LongMethod-LongMethodClass.

Relevance. Understanding if antipatterns increases the risk of a class to be changed, to be modified for fixing issues, or to be modified for preventing UHE is important from the points of view of both researchers and practitioners.

We bring evidence to researchers that antipatterns do increase the numbers of (1) changes that classes undergo, (2) issues in which classes are discussed, and (3) UHE issues in which classes are present. We also bring evidence that, like design patterns [3, 6, 11, 34], particular symptoms of antipatterns—their code smells—are more important than others. Therefore, this study justifies *a posteriori* the previous work on code smells and antipatterns: within the limits of the threats to its validity, classes belonging to antipatterns are more change-prone than others and therefore antipatterns may indeed hinder software evolution; we prove the conjecture in the literature—premise of this study—true.

We also provide evidence to practitioners—developers, quality assurance personnel, and managers—of the importance and usefulness of antipattern detection techniques to assess the quality of their systems by showing that classes participating to antipatterns are more likely to change often and–or to be prone to issues and unhandled exceptions than other classes. Consequently, a tester could decide to focus on classes belonging to antipatterns, because she knows that such classes have more risks to generate UHE. Similarly, a manager could use antipattern detection techniques to assess the volume of classes belonging to antipatterns in a to-be-acquired system and, thus, narrow down her price offer and forecast the system cost-of-ownership.

Organisation. Section 2 relates our study with previous works. Section 3 provides definitions and a description of our specification and detection approach for antipatterns. Section 4 describes the exploratory study definition and design. Section 5 presents the study results, while Section 6 discusses them, along with threats to their validity. Finally, Section 8 concludes the paper and outlines future work.

2. RELATED WORK

This section discusses related work on antipatterns definition and detection and on the impact of design patterns and object-oriented metrics on software evolution.

Antipatterns Definitions and Detection. The first book on "antipatterns" in object-oriented development was written in 1995 by Webster [36]; his contribution includes conceptual, political, coding, and quality-assurance problems. Riel [28] defined 61 heuristics characterising good object-oriented programming to assess a system quality manually and improve its design and implementation. Beck [15] defined 22 code smells, suggesting where developers should apply refactorings. Mäntylä [23] and Wake [35] proposed classifications for code smells. Brown *et al.* [7] 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 a wide academic audience. They are the basis of all the approaches to specify and detect (semi-)automatically code smells and antipatterns, such as DECOR [26], presented in Section 3.

Several approaches to specify and detect code smells and antipatterns have been proposed. They range from manual approaches, based on inspection techniques [32], to metric-based heuristics [24, 27], where antipatterns are identified according to sets of rules and thresholds defined on various metrics. Rules may also be defined using fuzzy logic and executed by means of a rule-inference engine [1].

Some approaches for complex software analysis use visualisation techniques [10, 29]. Such semi-automatic approaches are an interesting compromise between fully automatic detection techniques that can be efficient but loose track of the context and manual inspections that are slow and subjective [21]. However, they require human expertise and are thus time-consuming. Other approaches perform fully automatic detection and use visualisation techniques to present the detection results [22, 33].

This previous work has contributed significantly to the specification and automatic detection of code smells and antipatterns. The approach used in this study, DECOR, builds on this previous work and offers a complete method to specify antipatterns and automatically detect them.

Design Patterns and Software Evolution. Several authors have studied the impact of design patterns on systems. Vokac [34] analysed the corrective maintenance of a large commercial system over three years and compared the fault rates of classes that participated in design patterns against those of classes that did not. He noticed that participating classes were less fault prone than others. He also concluded that the Observer and Singleton patterns are correlated with larger classes that could require special attention and that classes designed with the Factory Method pattern are more compact and less coupled than others and, consequently, have lower fault rates. Vokac could not find a clear tendency for the Template Method pattern. Vokac's work inspired this study, in particular the use of logistic regression to analyse the correlations between antipatterns and change-, issue-, and unhandled exception-proneness.

Bieman *et al.* [6] analysed four small and one large systems to identify the impact of design patterns, such as pattern change proneness. Khomh and Guéhéneuc [30] performed an empirical study of the impact of the 23 design patterns from [16] on ten different quality characteristics and concluded that patterns do not necessarily promote reusability, expandability, and understandability, as advocated by Gamma *et al.* Other studies deal with the changeability and resilience to change of design patterns [3] and of classes playing a specific role in design patterns [11], or their impact on the maintainability of a large commercial system [37].

While this previous work investigates the positive (and negative) impact of good design principles, *i.e.*, design patterns, on software systems, we study the impact of poor design choices, *i.e.*, antipatterns, on software evolution and focus specifically on three kinds of software changes.

Metrics and Software Evolution. Several studies, such as Basili *et al.*'s seminal work [5], used metrics as quality indicators. Cartwright and Shepperd [8] conducted an empirical study on an industrial C++ system (over 133 KLOC), which supported the hypothesis that classes in inheritance

relations are more fault prone. It followed that Chidamber and Kemerer DIT and NOC metrics [9] could be used to find classes that are likely to have higher fault rates. Gyimothy *et al.* [19] compared the capability of sets of Chidamber and Kemerer metrics to predict fault-prone classes within Mozilla, using logistic regression and other machine learning techniques (*e.g.*, artificial neural networks). They concluded that the CBO metric is the most discriminating metric. They also found LOC to discriminate fault-prone classes well. Zimmermann *et al.* [39] conducted an empirical study on Eclipse showing that a combination of complexity metrics can predict faults and suggesting that the more complex the code, the more faults. El Emam *et al.* [13] showed that after controlling for the confounding effect of size, the correlation between metrics and fault-proneness disappeared: many metrics are correlated with size and, therefore, do not bring more information to predict fault proneness.

In our study, we relate changeability and issue proneness to antipatterns rather than to metrics as antipatterns are a higher-level of abstraction than metrics, thus likely to be better indicators for developers than metrics. We also relate antipatterns with issues. However, differently from Gyimothy *et al.* [19], we talk about issues rather than "faults" because we are aware that only about 50% of posted issues are actually related to corrective maintenance, others relate to enhancement or restructuring [4].

3. DEFINITIONS AND DETECTION

We now define the studied phenomena and the detection technique used to relate antipatterns to these phenomena.

3.1 Changes, Issues, Unhandled Exceptions

In this study, we relate antipatterns with changes. We also relate antipatterns with issues posted in issue-tracking systems and issues related to unhandled exceptions (UHE).

Changes. Changes are counted for each class in a system as the number of commits related to that class in its versioning system (CVS in the case of Eclipse). Changes provide an indication of the development and maintenance efforts put into the class: the more changes, likely the more efforts.

Issues. Issues are reports that describe different kinds of problems related to a system [4]. They are usually posted in issue-tracking systems by users and developers to warn the system community of impending issues with its functionalities. An issue often includes a description, steps to reproduce, and possibly some patches to resolve it. We only consider issues marked as "FIXED" or "CLOSED", because they required some changes, and traced into changes by matching the issue ID in the commits [14].

UHE Issues. We distinguish issues and *UHE issues*, which are issues reporting null-pointer exceptions and other unhandled exceptions thrown by the system at the users. UHE issues are critical because they often prevent the use of the system and thus must be avoided at all costs.

3.2 Code Smells and Antipatterns

When studying antipatterns, we do not excluded that, in a particular context, an antipattern can be the best way to actually implement or design a (part of a) system. For example, automatically-generated parsers are often Spaghetti Code, *i.e.*, very large classes with very long methods. Only developers can evaluate their impact according to the context: it may be perfectly sensible to have a Spaghetti Code if it comes from a well-defined and well-managed grammar.

We use our previous approach, DECOR (Defect dEtection for CORrection) [25, 31], to specify and detect antipatterns. DECOR is based on a thorough domain analysis of existing code smells and antipatterns on which is based a domain-specific language to specify antipatterns. It also provides methods to specify antipatterns and detect them automatically. It can be applied on any object-oriented system because it is based on the PADL meta-model and POM framework. PADL is a meta-model to describe object-oriented systems [18]; parsers for C++ and Java are available. POM is a PADL-based framework that offers implementations of more than 60 metrics, including McCabe cyclomatic complexity and Chidamber and Kemerer metric suite, and statistical features, *e.g.*, computing and accessing metrics boxplots, to compensate for the effect of size. Moha *et al.* [26] showed that the current detection algorithms obtained from DECOR ensure 100% recall and have precisions between 31% and 70% (average 60%).

Listing 1 shows the specification of the LazyClass antipattern. LazyClass is an *antipattern* because it combines two symptoms: the code smells NotComplexClass and FewMethods. A class is a LazyClass if it is a class with low complexity (code smell NotComplexClass) and with only few methods (code smell FewMethods). A class is a NotComplexClass if the sum of its number of declared methods weighted by their complexity—measured as their number of method invocations in terms of the Chidamber and Kemerer's WMC metric—is very low, *i.e.*, under the lower quartile when considering all classes. A class has FewMethods if its number of declared methods and fields (measured using the NMD and NAD metrics) is also very low. The values 20 and 5 indicates that, in these two code smells, a deviation from the lower quartile is possible. For example, classes with a WMC value at most 20% greater than the lower quartile are also tagged as complex classes.

In the following, we focus on 11 antipatterns from [7, 15]: AntiSingleton, Blob, ClassDataShouldBePrivate, ComplexClass, LargeClass, LazyClass, LongMethod, LongParameterList, SpaghettiCode, SpeculativeGenerality, SwissArmyKnife, because these antipatterns are representative of problems with data, complexity, size, and the features provided by a class. They divide in 20 code smells, listed in Table 6. Their specifications are outside of the scope of this paper and can be obtained upon request.

4. STUDY DEFINITION AND DESIGN

The *goal* of our study is to investigate the relations between classes participating in antipatterns and their proneness to changes, issues, and unhandled exceptions.

The *quality focus* is to provide developers, quality assurance personnel, and managers with recommendations on antipatterns, to understand and forecast their system evolution, and researchers and practitioners with evidence on the conjecture of the impact of antipatterns on changeability.

The *perspective* is that of developers, who perform development or maintenance activities on systems, and of testers who perform testing activities and who need to know which classes are important to test. It is also that of managers and–or quality assurance personnel, who could use detection techniques to assess the change-, issue-, and unhandled

```
RULE_CARD : LazyClass {
        RULE : LazyClass { INTER NotComplexClass FewMethods };
        RULE : NotComplexClass { (METRIC: WMC, VERY_LOW, 20) };
        RULE : FewMethods { (METRIC: NMD + NAD, VERY_LOW ,5) };
};
```

Listing 1: Specification of the LazyClass Antipattern

Dates	Releases	Number of				
		LOC	Classes	Changes	Issues	UHE
2001-11-07	1.0	781,480	4,647	21,553	2,208	166
2002-06-27	2.0	1,249,840	6,742	26,378	3,367	294
2003-06-27	2.1.1	1,797,917	8,730	10,397	2,211	165
2003-11-03	2.1.2	1,799,037	8,732	11,534	1,923	133
2004-03-10	2.1.3	1,799,702	8,736	15,560	3,137	309
2004-06-25	3.0	2,260,165	11,166	11,582	798	113
2004-09-16	3.0.1	2,268,058	11,192	24,150	4,225	248
2005-03-11	3.0.2	2,272,852	11,252	49,758	10,000	1,061
2006-06-29	3.2	3,271,516	15,153	2,745	550	111
2006-09-21	3.2.1	3,284,732	15,176	11,854	4,078	229
2007-02-12	3.2.2	3,286,300	15,184	10,682	2,137	153
2007-06-25	3.3	3,752,212	17,162	7,386	1,822	244
2007-09-21	3.3.1	3,756,164	17,167	40,314	14,915	792
Total	13	31,579,975	151,039	243,903	51,371	4,018

Table 1: Summary of the 13 analysed releases of Eclipse. Changes, issues, and UHE are counted from one release to the next, Eclipse 3.4 excluded.

exception-proneness 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 of Eclipse. Eclipse is famous as an open-source integrated development environment. It is a platform used both in open-source communities and in industry. Eclipse is mostly written in Java, with C/C++ code used mainly for widget toolkits. We chose Eclipse because it is a large system (more than 3.5 MLOC in it 3.3.1 release) that, therefore, come close to the size of real industrial systems. It is also a system that has been developed partly by a commercial company (IBM), which makes it more likely to embody industrial practices. Also, it has been used by other researchers in related studies, *e.g.*, to predict faults [39]. We analysed 13 releases of Eclipse available on the Internet between 2001 and 2008. Table 1 summarises the analysed releases and their key figures. Changes, issues, and UHE are counted from one release to the next (values for 3.3.1 refer to the period between releases 3.3.1 and 3.4).

4.1 Research Questions

We divide our study in three sets of null hypotheses to be tested against the data collected from Eclipse to answer research questions concerned with the relations between antipatterns, their code smells, and (1) change proneness, (2) issue proneness, and (3) UHE proneness.

Change Proneness.

- **RQ1:** *What is the relation between antipatterns and change proneness?* We investigate whether classes participating in some antipatterns 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 does not significantly differ between classes participating in some antipatterns and other classes.*
- **RQ2:** *What is the relation between particular kinds of antipatterns and change proneness?* Also, we analyse whether particular kinds of antipatterns contribute more than others to changes by testing the null hypothesis: H_{02}*: classes participating in particular kinds of antipatterns are not significantly more change prone than other classes.*
- **RQ3:** *What is the relation between the code smells composing the studied antipatterns and change proneness?* We also analyse the influence of the 20 code smells composing the 11 studied antipatterns on changes by testing the null hypothesis: H_{03}*: classes with particular kinds of code smells are not significantly more change-prone than other classes.*

Issue Proneness.

- **RQ4:** *What is the relation between antipatterns and issue proneness?* This research questions relates antipatterns with issues posted in Eclipse issue-tracking system (Bugzilla) and fixed. The null hypothesis tested is: H_{04}*: the proportion of classes undergoing at least one change to fix one issue between two releases does not significantly differ between classes participating in some antipatterns and other classes.*
- **RQ5:** *What is the relation between particular kinds of antipatterns and issue proneness?* Also, we analyse whether particular kinds of antipatterns tend to be more related to issues than other kinds: H_{05}*: classes participating in particular kinds of antipatterns are not significantly more prone to issue-fixing than other classes.*
- **RQ6:** *What is the relation between the code smells composing the studied antipatterns and issue proneness?* We also analyse the influence of the code smells on classes appearing in issues by testing the null hypothesis: H_{06}*: classes with particular kinds of code smells are not significantly more prone to issue-fixing than other classes.*

Antipatterns and UHE Proneness.

- **RQ7:** *What is the relation between antipatterns and UHE proneness?* This research question focuses on the relation between antipatterns and severe issues that caused a null pointer exceptions or other unhandled exceptions. The null hypothesis is: H_{07}*: the proportion of classes undergoing at least one UHE issue fixing between two releases does not significantly differ between classes participating in some antipatterns and other classes.*

- **RQ8:** *What is the relation between particular kinds of antipatterns and UHE proneness?* We also analyse the influence of particular kinds of antipatterns on UHE issues by testing the null hypothesis: H_{08}*: classes participating in particular kinds of antipatterns are not significantly more prone to UHE issue-fixing than other classes.*

- **RQ9:** *What is the relation between the code smells composing the studied antipatterns and UHE proneness?* We also analyse the likelihood for classes with code smells to appear in UHE issues by testing the null hypothesis: H_{09}*: classes with particular kinds of code smells are not significantly more prone to UHE issue-fixing than other classes.*

4.2 Variable Selection

We relate the following dependent and independent variables to test the previous null hypotheses and, thus, answer the associated research questions.

Independent variables. Independent variables divide in two sets: the numbers of (1) classes participating in the 11 kinds of antipatterns and (2) classes belonging to the 20 code smells defining the 11 antipatterns. Therefore, we have as many independent variable as kinds of code smells and antipatterns; each variable indicates the number of times that a class participates in a kind of antipattern or code smell in a release r_k.

Dependent Variables. Dependent variables measure the phenomena to relate to classes belonging to antipatterns:

- *Change proneness* refers to the number of changes that a class underwent between release r_k (in which it was participating in some antipatterns) and the subsequent release r_{k+1}. This number is measured by counting the number of changes occurring in classes as the number of commits in Eclipse CVS versioning system.

- *Issue proneness* counts the number of fixed and resolved issues posted in Eclipse Bugzilla that refer to classes participating in some antipatterns between releases r_k and r_{k+1}. We link changes in the CVS to issues in Bugzilla by matching Bugzilla issue ID with CVS commit notes, as in previous work [14].

- *UHE proneness* counts to the number of the previous issues that caused unhandled exceptions by looking at their content and searching for the presence of specific keywords, such as "Null Pointer Exception" or "NPE", and–or of stack traces.

Given $C_{i,k}$, $I_{i,k}$, and $U_{i,k}$ the sets of changes, issues, and UHE issues that a class c_i undergo in release r_k, the sets are inclusive: $U_{i,k} \subseteq I_{i,k} \subseteq C_{i,k}$, because they all refer to changes in the CVS, either counting all changes ($C_{i,k}$), changes that have been traced back into some issues posted in Bugzilla ($I_{i,k}$), or changes resulting from an issue posted in Bugzilla and that includes a stack trace ($U_{i,k}$).

4.3 Analysis Method

In RQ1, RQ4, and RQ7, we are interested in understanding whether the occurrences of changes, issues, and–or UHE issues 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, one issue, and–or one UHE significantly vary between classes participating in antipatterns and other classes. We use Fisher's exact test [12] for H_{01}, H_{04}, and H_{07}. Fisher's exact test checks whether proportions vary between two samples and is an equivalent of the χ^2 test but is deemed more accurate for small samples, such as the samples of the classes exhibiting issues or UHE issues, see in Table 1. We also performed the χ^2 test (results not shown here) and obtained consistent results in term of significance.

We also compute the *odds ratio* (OR) [12] that indicates how much likely is an event to occur or not. The odds ratio 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 belonging to no antipattern (control group): $OR = \frac{p/(1-p)}{q/(1-q)}$. An odds ratio of 1 indicates that the event is equally likely in both samples. An OR greater than 1 indicates that the event is more likely in the first sample (antipatterns) while an OR less than 1 that it is more likely in the second sample.

In RQ2, RQ5, and RQ8, we want to understand how specific kinds of antipatterns are correlated with change-, issue-, and UHE-proneness. We use a logistic regression model [20], similarly to Vokac's previous work [34], to correlate the presence of antipatterns with, for example, change-proneness. 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:

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

where:

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

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

While in other contexts (*e.g.*, [2] and [19]), logistic regression models were used for prediction purposes; as in [34], we use such models as an alternative to the Analysis Of Variance (ANOVA) for dichotomous dependent variables. This is to say that we use logistic regression to reject H_{02}, H_{03}, H_{05} H_{06}, H_{08} and H_{09}.

Then, we count, for each antipattern, the number of times, across the 13 analysed Eclipse releases, that the p-values obtained by the logistic regression were significant. We define a threshold $t = 10$ (75%) to assess whether classes participating to a specific kind of antipattern have a significantly greater odds to change than classes not participating to this kind: if in more than t releases of Eclipse the classes with a kind of antipattern are more likely *e.g.*, to change, then we conclude that this antipattern has a significant negative impact on *e.g.*, change-proneness.

With RQ3, RQ6, and RQ9, similarly to our previous work [11], we want to understand if particular *roles* in the an-

Releases	AP-Changes	AP-No Changes	No AP-Changes	AP-No Changes	p-values	OR
1.0	1921	1669	538	510	0.22	1.09
2.0	3493	1350	947	259	< **0.01**	0.71
2.1.1	2148	3714	269	1088	< **0.01**	2.34
2.1.2	2320	3544	439	918	< **0.01**	1.37
2.1.3	2833	3037	625	730	0.16	1.09
3.0	3090	4703	986	1232	< **0.01**	0.82
3.0.1	5780	2020	1730	490	< **0.01**	0.81
3.0.2	5363	2437	1615	607	< **0.01**	0.83
3.2	1593	8490	341	3341	< **0.01**	1.84
3.2.1	2504	7588	565	3130	< **0.01**	1.83
3.2.2	2877	7221	853	2844	< **0.01**	1.33
3.3	1682	10018	241	3190	< **0.01**	2.22
3.3.1	4056	7660	963	2460	< **0.01**	1.35

Table 2: Contingency table and Fisher's exact test results for classes participating in at least one antipattern and undergoing at least one change. (AP means antipatterns).

Releases	AP-Issues	AP-No Issues	No AP-Issues	No AP-No Issues	p-values	OR
1.0	489	3101	109	939	**0.01**	1.36
2.0	740	4103	134	1072	< **0.01**	1.44
2.1.1	418	5444	64	1293	< **0.01**	1.55
2.1.2	377	5487	52	1305	< **0.01**	1.72
2.1.3	574	5296	77	1278	< **0.01**	1.80
3.0	512	7281	68	2150	< **0.01**	2.22
3.0.1	927	6873	153	2067	< **0.01**	1.82
3.0.2	2336	5464	451	1771	< **0.01**	1.68
3.2	743	9340	125	3557	< **0.01**	2.26
3.2.1	1181	8911	234	3461	< **0.01**	1.96
3.2.2	954	9144	229	3468	< **0.01**	1.58
3.3	999	10701	118	3313	< **0.01**	2.62
3.3.1	1659	10057	457	2966	0.24	1.07

Table 3: Contingency table and Fisher's exact test results for classes participating in at least one antipattern reported in at least one fixed issue.

tipatterns, *i.e.*, if particular code smells defining the studied antipatterns, are more symptomatic of their antipattern change-, issue-, or UHE-proneness than others. Therefore, we again use a logistic regression model [20] to correlate the presence of code smells with the three phenomena.

5. STUDY RESULTS

We now report the results of our study to address the research questions. We discuss these results in Section 6.

5.1 Change Proneness

Table 2 reports the results of Fisher's exact test and ORs when testing H_{01}. Besides two exceptions, releases 1.0 and 2.1.3, proportions are significantly different, thus allowing to reject H_{01}. We therefore conclude that there is a greater proportion of classes participating in antipatterns that change wrt. other classes. While in many cases ORs are close to 1, *i.e.*, the odds is even that a class belonging to an antipattern changes or not, it is worth highlighting some releases, such as 2.1.1, 3.2, 3.2.1, and 3.3, for which the odds of changing are about two times in favour of classes belonging to antipatterns. We conclude that the odds for a class belonging to some antipatterns to change are in general higher, in some cases twice, than other classes. H_{01} rejection and the ORs provide *a posteriori* concrete evidence of the negative impact of antipatterns on change-proneness. Developers should be wary of classes participating in antipatterns, because they are more likely to be the subject of their efforts.

Releases	AP-UHE	AP-No UHE	No AP-UHE	No AP-No UHE	p-values	OR
1.0	94	3496	2	1046	< **0.01**	14.06
2.0	151	4692	13	1193	< **0.01**	2.95
2.1.1	99	5763	9	1348	< **0.01**	2.57
2.1.2	80	5784	8	1349	**0.02**	2.33
2.1.3	161	5709	5	1350	< **0.01**	7.61
3.0	140	7653	7	2211	< **0.01**	5.78
3.0.1	196	7604	10	2210	< **0.01**	5.70
3.0.2	608	7192	70	2152	< **0.01**	2.60
3.2	130	9953	8	3674	< **0.01**	6.00
3.2.1	313	9779	28	3667	< **0.01**	4.19
3.2.2	139	9959	25	3672	< **0.01**	2.05
3.3	150	11550	21	3410	< **0.01**	2.11
3.3.1	252	11464	27	3396	< **0.01**	2.76

Table 4: Contingency table and Fisher's exact test results for classes participating in at least one antipattern reported with at least one fixed UHE.

Antipatterns	Proneness to		
	Changes	Issues	UHE
AntiSingleton	6	**13**	7
Blob	9	4	2
ClassDataShouldBePrivate	7	5	6
ComplexClass	**11**	**13**	**11**
LargeClass	0	0	0
LazyClass	**11**	**12**	**11**
LongMethod	**10**	**13**	**12**
LongParameterList	7	6	4
SpaghettiCode	0	0	0
SpeculativeGenerality	3	4	2
SwissArmyKnife	8	3	3

Table 5: Number of significant p-values across the 13 analysed releases obtained by logistic regression for the correlations between changes, issues, and UHE and kinds of antipatterns. Boldface indicates significant p-values for at least 75% of the releases.

Table 5 (column 2) and Table 9 report the results of the logistic regression for the correlations between change-proneness and the different kinds of antipatterns. We can reject H_{02} for all antipatterns but LargeClass and SpaghettiCode. Following our analysis method, we remark that ComplexClass, LazyClass, and LongMethod have a significant negative impact on change proneness: classes involved in these antipatterns are more likely to change than classes participating to other or no antipattern in more than 75% (10) of Eclipse releases.

Table 6 (column 2) shows the results of the logistic regression for the correlations between changes and the different kinds of code smells composing the studied antipatterns. We can reject H_{03} for 13 out of the 20 code smells, *i.e.*, 13 out of the 14 detected code smells. We analyse that only four code smells are symptomatic of changes: BlobDataClass, ComplexClass-LargeClassOnly, LazyClass-Few-

Code Smells	Proneness to		
	Changes	Issues	UHE
AntiSingleton-NotClassGlobalVariable	7	**13**	6
Blob-ControllerClass	5	9	3
Blob-DataClass	**13**	2	2
Blob-BigClass	1	2	1
Blob-LowCohesion	3	2	0
ClassDataShouldBePrivate-FieldPublic	8	5	6
ComplexClass-LargeClassOnly	**10**	**11**	8
LargeClass-VeryLargeClassOnly	–	–	–
LargeClass-LowCohesionOnly	–	–	–
LazyClass-FewMethods	**11**	**12**	**11**
LazyClass-NotComplexClass	–	–	–
LongMethod-LongMethodClass	**11**	**13**	**12**
LongParameterList-LongParameterListClass	7	6	4
SpaghettiCode-ClassGlobalVariable	0	0	0
SpaghettiCode-ComplexMethod	–	–	–
SpaghettiCode-MethodNoParameter	–	–	–
SpaghettiCode-NoInheritance	–	–	–
SpeculativeGenerality-AbstractClass	3	4	2
SpeculativeGenerality-OneChildClass	–	–	–
SwissArmyKnife-MultipleInterface	8	2	3

Table 6: Number of significant p-values across the 13 analysed Eclipse releases obtained by logistic regression for the correlations between change-, issue-, and UHE-proneness and kinds of code smells. Boldface indicates significant p-values for at least 75% of the analysed releases, a – indicates cases where no class was detected with that specific code smell.

Methods, and LongMethod-LongMethodClass.

5.2 Issue Proneness

Table 3 reports Fisher's exact test results and ORs for H_{04}. The differences in proportions are always significant, but for release 3.3.1; thus, in general, we can reject H_{04}. ORs are higher than for changes: they are close to 2 or higher, as for releases 3.0 ($OR = 2.22$), 3.2 ($OR = 2.26$), and 3.3 ($OR = 2.62$). The proportion of classes participating to antipatterns and reported in issues is twice as large as the proportion of other classes. Therefore, we conclude that antipatterns may have a negative impact on the issue-proneness of classes.

Table 5 (column 3) and Table 10 report the results of the logistic regression for the correlations between issue-proneness and the different kinds of antipatterns. We can reject H_{05} for all antipatterns but LargeClass and SpaghettiCode. Four antipatterns characterise, in at least 75% of the releases, classes that are more issue-prone than other classes: AntiSingleton, ComplexClass, LazyClass and LongMethod.

Table 6 (column 3) shows the results of the logistic regression for the correlations between issues and the code smells. We can reject H_{06} again for the 13 detected code smells. Following our analysis method, we conclude that four code smells are symptomatic of issues: AntiSingleton-NotClassGlobalVariable, ComplexClass-LargeClassOnly, LazyClass-FewMethods, and LongMethod-LongMethodClass.

5.3 UHE Proneness

Table 4 reports Fisher's exact test results and ORs for H_{07}. Differences in proportions are always significant, thus we can reject H_{07}. ORs are always high, ranging from 2.11 in release 3.3 to 14 in release 1.0: classes belonging to antipatterns are, at least, two times more likely to be involved in an UHE-related issue fixing than other classes. Therefore, we again conclude that antipatterns may have a negative impact on evolution wrt. UHE issues than other classes.

Table 5 (column 4) and Table 11 report the results of the logistic regression for the correlations between UHE-proneness and the different kinds of antipatterns. We reject H_{08} for all antipatterns but LargeClass and SpaghettiCode. Three antipatterns characterise UHE-prone classes in 75% of the releases: ComplexClass, LazyClass and LongMethod.

Table 6 (column 4) shows the results of the logistic regression for the correlations between issues and the code smells. We can reject H_{09} again for the 13 detected code smells. Following our analysis method, we conclude that only two code smells are symptomatic of UHE: LazyClass-FewMethods and LongMethod-LongMethodClass.

6. DISCUSSION

6.1 Relations with Antipatterns

Table 2 shows the ORs for classes participating in antipatterns to undergo at least one change between one release and the next. In general, classes belonging to antipatterns are more change-prone than others. However, there are four cases where the ratios are unexpected, *i.e.*, classes *not* belonging to antipatterns changed more than classes that do. (Greek-letter note-marks refer to Table 7.)

The first of these cases concerns classes having changed between 2.0 and 2.1.1, with an OR of 0.71. We explain this ratio using the release notes of Eclipse 2.1, 2.1.1, and 2.1.2. Eclipse 2.1$^{\alpha}$ introduced several new features wrt. the 2.0, including navigation history, sticky hovers, prominent status indication, and many more. In addition to these features, 283 issues$^{\beta}$ were fixed between 2.0 and 2.1 and 126 more$^{\gamma}$ between 2.1 and 2.1.1, including issues related to the new features, for example issue ID 1694 *"FEATURE: Contributed inspection formatter"* or 17872 *"Hover help for static final fields is inconsistent"*. Moreover, $8,730 - 6,742 = 1,988$ classes were added for a total increase in size of $1,797,17 - 1,249,840 = 548,077$ LOC. Therefore, between 2.0 and 2.1.1, Eclipse underwent such dramatic changes to its behaviour and API that many classes (not belonging to antipatterns) changed or were added to provide the new features, thus explaining the ORs.

The second, third, and fourth cases concern classes having changed between releases 3.0 and 3.2. We also explain these ratios using release notes. Eclipse 3.0 was a major improvement over the 2.0 series, with a new runtime platform implementing the OSGi R3.0 specifications$^{\delta}$ to become a Rich Client Platform, which can be used by organisations to develop any tools (not necessarily an IDE). It had many problems at first$^{\epsilon}$, corrected in the subsequent 3.0.1, 3.0.2, and 3.2 releases, with respectively 266$^{\zeta}$, 70$^{\eta}$, and 285 issues$^{\theta}$ fixed. As between 2.0 and 2.1.1, $15,153 - 11,166 = 3,987$ classes were added between 3.0 and 3.2, increasing Eclipse size by $3,271,516 - 2,260,165 = 1,011,351$ LOC. Thus, we argue that the changed classes were not only classes belonging to antipatterns but also classes related to the new features, thus explaining the ORs of 0.82, 0.81, and 0.83.

Tables 3 and 4 show results that confirm our expectations and the conjecture in the literature: the ORs of classes participating in antipatterns to be in issues or UHE issues range from 1.07 (issue-proneness, Eclipse 3.3.1) to 14.06 (UHE-proneness, Eclipse 1.0).

6.2 Relations with Specific Antipatterns

Link IDs	URLs
α	http://archive.eclipse.org/eclipse/downloads/drops/R-2.1-200303272130/whats-new-all.html
β	https://bugs.eclipse.org/bugs/buglist.cgi?product=JDT&product=PDE&product=Platform&target_milestone=2.1&resolution=FIXED&order=bugs.bug_id
γ	https://bugs.eclipse.org/bugs/buglist.cgi?product=JDT&product=PDE&product=Platform&target_milestone=2.1.1&resolution=FIXED&order=bugs.bug_id
δ	http://www.eclipse.org/osgi/
ϵ	For example, a search for "Eclipse 3.0 crash" returns 224 messages on http://www.eclipsezone.com/
ζ	https://bugs.eclipse.org/bugs/buglist.cgi?product=JDT&product=PDE&product=Platform&target_milestone=3.0.1&resolution=FIXED&order=bugs.bug_id
η	https://bugs.eclipse.org/bugs/buglist.cgi?product=JDT&product=PDE&product=Platform&target_milestone=3.0.2&resolution=FIXED&order=bugs.bug_id
θ	https://bugs.eclipse.org/bugs/buglist.cgi?product=JDT&product=PDE&product=Platform&target_milestone=3.2&resolution=FIXED&order=bugs.bug_id

Table 7: URLs of the discussed release notes and issues listings.

Table 5 shows the numbers of times, across the 13 releases, that each kind of antipatterns led to an increase in change-, issue-, and UHE-proneness. LargeClass and SpaghettiCode do not impact change-proneness at all. We explain these results by the low number of classes participating in these two antipatterns: an average of 479 and 2 classes respectively per release, for an average of 11,618 classes per release. Eclipse uses extensively object-oriented concepts, in particular polymorphism, and thus avoid large classes and spaghetti code. Moreover, detected antipatterns concerned stable classes. For example, in release 2.0, the only LargeClass was class `org.eclipse.core.internal.indexing.IndexedStoreException`, which is also an AntiSingleton and a DataClass. It changes once between 2.0 and 2.1.1 and never changed again.

Classes participating to the antipatterns ComplexClass, LazyClass, and LongMethod are more change-, issue-, and UHE-prone than others. ComplexClass classes are complex and central to the system, as discussed in the next Section 6.3. Classes participating in the LongMethod and LazyClass antipatterns contain long methods, which are also very complex, as also discussed below. This result confirms Fowler and Brown's warnings about these kinds of antipatterns. Classes participating in the LazyClass antipattern tend to be removed from the system or changed to increase their behaviour while other are introduced: there were 2,765 lazy classes in Eclipse 1.0, 59% of the system, while there were 8,967 lazy classes in Eclipse 3.3.1, 52% of the system. The class `org.eclipse.search.internal.core.SearchScope`, for example, was a lazy class in 1.0 but, in 3.0, methods `public boolean encloses(IResourceProxy proxy)` and `public void setDescription(String description)` were added with two more constructors and the inner class `WorkbenchScope` was removed. Some new lazy classes, like `org.eclipse.team.internal.ccvs.ui.actions.ShowEditorsAction`, were then introduced.

Classes belonging to the AntiSingleton are only more issue-prone than other classes. As mentioned in the next Section 6.3, AntiSingleton classes are generally removed from the system or changed. However, with their public attributes, as it happens for procedural "global variables", the risk of introducing errors in the system while changing these classes is likely to be higher.

Other antipatterns are not consistently related to the studied phenomena across releases. We conclude that they may impede the evolution of a particular release but do not impact the overall evolution of Eclipse.

6.3 Relations with Code Smells

Table 6 shows that classes with particular antipattern-related code smells behave differently from the others.

For the AntiSingleton-NotClassGlobalVariable code smell, we observed that they are removed from Eclipse or changed during its evolution: 16% of the classes with this smell present in release 1.0 had been removed in release 3.0 and only 53% of the classes were still AntiSingleton in that release. The other classes were changed. For example, `org.eclipse.compare.internal.CompareWithEditionAction`, an AntiSingleton in release 1.0, was changed during Eclipse evolution: all its methods were removed between 1.0 and 3.0 and it became a LazyClass with no behaviour.

We explain the change-proneness of Blob-DataClass (and of LazyClass-FewMethods) because they are representative of classes that are only used to store data, without providing much behaviour. These classes are likely to disappear or to change when new behaviour is added to the system. We found that only 8% of Blob-DataClasses from release 1.0 are still Blob-DataClasses in release 3.0: 23% of the classes have been removed from the system and behaviour was added to the remaining classes. For example, the class `org.eclipse.jdt.internal.compiler.problem.AbortCompilation`, had no behaviour in release 1.0, while two methods were added to it in release 3.0; `public void updateContext(InvocationSite invocationSite,CompilationResult unitResult)` and `public void updateContext(ASTNode astNode, CompilationResult unitResult)`.

ComplexClass-LargeClassOnly characterises classes that have a number of methods higher than the average. Therefore, developers adding new features or fixing issues are more likely to touch these classes because of the higher number of features in which they must participate.

Two particular code smells, LazyClass-FewMethods and LongMethod-LongMethodClass, are more change-, issue-, and UHE-prone than other classes. This result is expected for LongMethodClass, because classes with long methods are complex and are thus more likely to be modified for corrections and improvements. For the LazyClass-FewMethods, 57% of its classes across the 13 releases also are LongMethod-LongMethodClass. Thus, they are quite complex classes involved in higher numbers of issues and UHE issues.

We also found that the presence of the LazyClass-FewMethods code smell in a class is generally related to the LongMethod-LongMethodClass and LongParameterList-LongParameterListClass code smells. For example, in release 1.0, 55% of LazyClass-FewMethods are also LongMethod-LongMethodClass while 46% are LongParameterList-LongParameterListClass. Thus, their likelihood to throw unhandled exceptions is in part due to the complexity of their long methods with long parameter lists.

In addition, we observed that 82% of classes with the code smell ComplexClass-LargeClassOnly in release 1.0 are also LongMethod-LongMethodClass, which confirms the Fowler and Brown's suggestion: classes with long methods are complex. These classes tend to remain with their code smells during the evolution of the system and are, in general, central to the system core features. Previous studies [3] showed that classes and patterns central to the system core features are more change-prone than others. Similarly, one could expect that they are more issue prone too.

Classes with the SpaghettiCode-ClassGlobalVariable are not interesting. We expected this result because of the little number of such classes: an average of 2 for each release. For example, in release 3.0.1, the only SpaghettiCode was `org.eclipse.core.runtime.adaptor.EnvironmentInfo`, a distorted implementation of the Singleton design pattern using many `if`–`else` statements, but a quite stable class that underwent no changes in the following releases, as shown by its source code. The other code smells do not impact consistently Eclipse changeability.

Some code smells are indeed symptomatic of their antipatterns, for example LazyClass-FewMethods for LazyClass, but they do not explain *alone* that some classes participating in some antipatterns are more likely to change, to be mentioned in an issue, or to be involved in unhandled exception-related issue than others. Therefore, we conclude that antipatterns and code smells provide related yet different information and should be considered together to better understand the evolution of a software system. Similarly to roles in design patterns, some code smells are more subject to change than others [11].

Some code smells could not be detected, for example SpaghettiCode-MethodNoParameter, because of the good use of object-oriented concepts in Eclipse.

7. THREATS TO VALIDITY

We now discuss the threats to validity of our study following the guidelines provided for case study research [38].

Construct validity threats concern the relation between theory and observation; in our context, they are mainly due to errors introduced in measurements. The count of changes occurred to classes is reliable as it is based on the CVS change log. However, we are aware that we only distinguish classes undergoing changes between two releases from classes that do not, neither quantifying the amount nor the frequency of change. For issues, as in other previous works [19, 14], we focus on those that can be linked to changes; thus, we are aware that we are only considering a subset of all issues posted on Eclipse Bugzilla and impacting classes. For the same reason, we consider a limited set of UHE issues. Also, we are aware that the detection technique used include our subjective understanding of the antipattern definitions, although we followed previous works and they were validated in our previous work.

Threats to *internal validity* do not affect this particular study, being an exploratory study [38]. Thus, we cannot claim causation, but just relate the presence of antipatterns with the occurrences of changes, issues, and UHE. Nevertheless, we tried to explain why some antipatterns could have been the cause of changes/issues/UHE.

Conclusion validity threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the statistical tests that we used (we mainly used non-parametric tests).

Reliability validity threats concern the possibility of replicating this study. We attempted here to provide all the necessary details to replicate our study. Moreover, both Eclipse source code repository and issue-tracking system are available to obtain the same data. Finally, the raw data on which our statistics have been computed are available on the Web[1].

Threats to *external validity* concern the possibility to generalise our findings. First, we are aware that our study has been performed on Eclipse only and that, thus, generalisation will require further case studies. Yet, Eclipse is somewhat representative of large industrial systems. Second, we used a particular yet representative subset of antipatterns. Different antipatterns could have lead to different results and should be studied in future work. However, within its limits, our results confirm the conjecture in the literature.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we provided empirical evidence of the negative impact of antipatterns on classes change-proneness, on their likelihood to be subject of some issues in an issue-tracking system, and on their risk to throw unhandled exceptions (UHE). We studied the odds ratios of changes, issues, and UHE on classes participating or not in certain kinds of antipatterns. We showed that classes participating in antipatterns are significantly more likely to be the subject of changes, to be involved in issue-fixing changes, and to issue-fixing changes related to unhandled exceptions, than others classes. We also showed that some code smells defining the antipatterns, similarly to roles in design patterns, are also more likely to be of concern during evolution.

This exploratory study provides, within the limits of the threats to its validity, convincing evidence that classes belonging to antipatterns are more change-, issue-, and UHE-prone than classes not participating in antipatterns. Therefore, we provide evidence for the conjecture in the literature that antipatterns may have a negative impact on software evolution. We justify *a posteriori* previous work on antipatterns and provide a basis for future research to understand precisely the root causes of their negative impact.

The study also provides evidence to practitioners that they should pay more attention to systems with a high prevalence of antipatterns during development and maintenance. Indeed, systems containing a high proportion of antipatterns are likely to be more change prone, and their classes are more likely to be the root causes of issues and unhandled exceptions. Therefore, the cost-of-ownership of such systems will be higher than for other systems, because developers will have to spend more time fixing issues.

Future work includes replicating this study on other systems than Eclipse to assess the generality of our results. It also includes studying more antipatterns as well as studying in detail the roles code smells in antipatterns and their individual impact on software evolution notwithstanding their composition in antipatterns.

Data. All data as well as a technical report with more detailled results are available on the Web[1].

Acknowledgements. This work has been partly funded by the NSERC Research Chair in Software Change and Evolu-

[1] `http://www.ptidej.net/downloads/experiments/prop-FSE09`

tion and by NSERC Discovery Grant #293213.

APPENDIX

A. DETAILLED DEFINITIONS OF THE DESIGN SMELLS

In this study we focused on the following code smells:

Anti-Singleton: It is a class that declares public class variable that are used as "global variable" in procedural programming. It reveals procedural thinking in object-oriented programming and may increase the difficulty to maintain the program.

Blob: (called also God class [28]) corresponds to a large controller class that depends on data stored in surrounded data classes. A large class declares many fields and methods with a low cohesion. A controller class monopolises most of the processing done by a system, takes most of the decisions, and closely directs the processing of other classes [**?**].

Class Data Should Be Private: It occurs when the data encapsulated by a class is public, thus allowing client classes to change this data without the knowledge of the declaring class.

Complex Class: It is a class that both declares many fields and methods and which methods realise complex treatments, using many if and switch instructions. Such a class is probably providing lots of services while being difficult to maintain and fragile due to its complexity.

Large Class: It is a class with too many responsibilities. This kind of class declares a high number of usually unrelated methods and attributes.

Lazy Class: It is a class that does not do enough. The few methods declared by this class have a low complexity.

Long Method: It is a method with a high number of lines of code. A lot of variables and parameters are used.Generally, this kind of method does more than its name suggests it.

Long Parameter List: It corresponds to a method with high number of parameters. This smell occurs when the method has more than four parameters.

The Spaghetti Code: It is an antipattern that is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes with no structure, declaring long methods with no parameters, and utilising global variables for processing. Names of classes and methods may suggest procedural programming. Spaghetti Code does not exploit and prevents the use of object-orientation mechanisms, polymorphism and inheritance.

Speculative Generality: It is an abstract class without child classes. It was added in the system for future uses and this entity pollutes the system unnecessarily.

Swiss Army Knife: It refers to a tool fulfilling a wide range of needs. The Swiss Army Knife design smell is a complex class that offers a high number of services, for example, a complex class implementing a high number of interfaces. A Swiss Army Knife is different from a Blob, because it exposes a high complexity to address all foreseeable needs of a part of a system, whereas the Blob is a singleton monopolising all processing and data of a system. Thus, several Swiss Army Knives may exist in a system, for example utility classes.

B. DETAILLED NUMBERS OF CODE SMELLS PER RELEASES

C. DETAILLED LOGISTIC REGRESSION RESULTS

This appendix provides Tables 9, 10, and 11 with more details on the results of applying logistic regression for the correlations between changes, issues, and UHE issues and kinds of smells.

D. REFERENCES

[1] E. H. Alikacem and H. Sahraoui. Generic metric extraction framework. In *Proceedings of the* 16^{th} *International Workshop on Software Measurement and Metrik Kongress (IWSM/MetriKon)*, pages 383–390, 2006.

[2] G. Antoniol, K. 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 2008 conference of the Centre for Advanced Studies on Collaborative Research, October 27-30, 2008, Richmond Hill, Ontario, Canada*, page 23, 2008.

[3] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta. An empirical study on the evolution of design patterns. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 385–394, New York, NY, USA, 2007. ACM Press.

[4] K. Ayari, P. Meshkinfam, G. Antoniol, and M. D. Penta. Threats on building models from cvs and bugzilla repositories: the mozilla case study. In *IBM Centers for Advanced Studies Conference*, pages 215–228, Toronto CA, Oct 23-25 2007. ACM.

[5] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Eng.*, 22(10):751–761, 1996.

[6] 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 Computer Society, 2003.

[7] 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.

[8] M. Cartwright and M. Shepperd. An empirical investigation of an object-oriented software system. *IEEE Trans. on Software Engineering*, 26(8):786–796, August 2000.

[9] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. on Software Engineering*, 20(6):476–493, June 1994.

[10] 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 Computer Society, April 2008.

[11] 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 H. Mei and K. Wong, editors, *Proceedings of the* 24^{th} *International Conference on Software Maintenance*. IEEE Computer Society Press, September–October 2008.

Design and Associated Code Smells	Numbers per Eclipse Versions												
	1.0	2.0	2.1.1	2.1.2	2.1.3	3.0	3.0.1	3.0.2	3.2	3.2.1	3.2.2	3.3	3.3.1
AntiSingleton	392	600	767	769	770	1370	1223	1224	1560	1568	1568	1985	1986
NotClassGlobalVariable	392	600	767	769	770	1370	1223	1224	1560	1568	1568	1985	1986
Blob	311	450	556	558	559	1009	801	801	1055	1066	1067	1264	1265
ControllerClass	106	121	164	164	164	285	251	251	374	375	375	451	453
DataClass	399	591	664	665	665	1256	910	911	1011	1015	1021	1183	1184
BigClass	190	298	352	354	355	653	497	497	624	633	632	725	724
LowCohesion	15	31	40	40	40	71	53	53	57	58	60	88	88
ClassDataShouldBePrivate	411	477	567	569	573	1050	1723	1723	2083	2100	2085	2398	2402
FieldPublic	411	477	567	569	573	1050	1723	1723	2083	2100	2085	2398	2402
ComplexClass	274	408	486	487	488	896	702	701	879	888	888	1001	1002
LargeClassOnly	274	408	486	487	488	896	702	701	879	888	888	1001	1002
LargeClass	1	1	1	1	1	2	5	4	4	4	4	8	8
VeryLargeClassOnly	1	1	1	1	1	2	5	4	4	4	4	8	8
LowCohesionOnly	1	1	1	1	1	2	5	4	4	4	4	8	8
LazyClass	2765	3530	4118	4118	4120	7650	5413	5414	7160	7171	7180	8966	8967
FewMethods	2765	3530	4118	4118	4120	7650	5413	5414	7160	7171	7180	8966	8967
NotComplexClass	2765	3530	4118	4118	4120	7650	5413	5414	7160	7171	7180	8966	8967
LongMethod	2840	4207	5131	5133	5168	9375	6344	6369	7033	7049	7046	8709	8737
LongMethodClass	2840	4207	5131	5133	5168	9375	6344	6369	7033	7049	7046	8709	8737
LongParameterList	1224	2229	2283	2283	2283	4512	2914	2918	2393	2445	2449	3342	3505
LongParameterListClass	1224	2229	2283	2283	2283	4512	2914	2918	2393	2445	2449	3342	3505
SpaghettiCode	2	1	1	1	1	2	1	1	0	0	0	1	1
ClassGlobalVariable	2	1	1	1	1	2	1	1	0	0	0	1	1
ComplexMethod	2	1	1	1	1	2	1	1	0	0	0	1	1
MethodNoParameter	2	1	1	1	1	2	1	1	0	0	0	1	1
NoInheritance	2	1	1	1	1	2	1	1	0	0	0	1	1
SpeculativeGenerality	61	88	109	109	109	197	174	174	211	211	211	249	249
AbstractClass	61	88	109	109	109	197	174	174	211	211	211	249	249
OneChildClass	61	88	109	109	109	197	174	174	211	211	211	249	249
SwissArmyKnife	75	56	52	52	52	108	73	73	78	79	80	96	97
MultipleInterface	75	56	52	52	52	108	73	73	78	79	80	96	97

Table 8: Summary of the numbers of smells in the analysed releases of Eclipse.

[12] S. D.J. *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

[13] 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. on Software Engineering*, 27(7):630–650, July 2001.

[14] 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 Computer Society Press.

[15] M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1^{st} edition, June 1999.

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

[17] J. Garzás and M. Piattini. *Object-oriented Design Knowledge: Principles, Heuristics, and Best Practices*. IGI Publishing, 2007.

[18] Y.-G. Guéhéneuc and G. Antoniol. DeMIMA: A multi-layered framework for design pattern identification. *Transactions on Software Engineering*, 34(5):667–684, September 2008.

[19] T. Gyimóthy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transaction on Software Engineering*, 31(10):897–910, 2005.

[20] D. Hosmer and S. Lemeshow. *Applied Logistic Regression (2nd Edition)*. Wiley, 2000.

[21] 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.

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

[23] M. Mantyla. *Bad Smells in Software - a Taxonomy and an Empirical Study*. PhD thesis, Helsinki University of Technology, 2003.

[24] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the* 20^{th} *International Conference on Software Maintenance*, pages 350–359. IEEE Computer Society Press, 2004.

[25] N. Moha, Y.-G. Guéhéneuc, A.-F. L. Meur, and L. Duchien. A domain analysis to specify design defects and generate detection algorithms. In J. Fiadeiro and P. Inverardi, editors, *Proceedings of the* 11^{th} *International Conference on Fundamental Approaches to Software Engineering*, pages 276–291. Springer-Verlag, 2008.

[26] N. Moha, Y.-G. Guéhéneuc, A.-F. L. Meur, L. Duchien, and A. Tiberghien. From a domain analysis to the specification and detection of code and design smells. *Formal Aspects of Computing*, 2009. Accepted for publication.

[27] M. J. Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In F. Lanubile and C. Seaman, editors, *Proceedings of the* 11^{th} *International Software Metrics Symposium*. IEEE Computer Society Press, September 2005.

[28] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[29] 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, Washington, DC, USA, 2001. IEEE Computer Society.

[30] Foutse Khomh and Y.-G. Guéhéneuc. Do design patterns impact software quality positively? In C. Tjortjis and A. Winter, editors, *Proceedings of the* 12^{th} *Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, April 2008. Short Paper.

[31] 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* 4^{th} *International Conference on Formal Concept Analysis*. Springer-Verlag, February 2008.

[32] 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.

[33] 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 Computer Society Press, Oct. 2002.

Smells	1.0	2.0	2.1.1	2.1.2	2.1.3	3.0	3.0.1	3.0.2	3.2	3.2.1	3.2.2	3.3	3.3.1
AntiSingleton-NotClassGlobalVariable	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
Blob-ControllerClass	0.10	**< 0.01**	**0.03**	**< 0.01**	**< 0.01**	0.21	**< 0.01**	**< 0.01**	0.59	0.23	**0.01**	**< 0.01**	**0.01**
Blob-DataClass	0.79	0.18	0.08	0.25	**0.01**	0.07	0.15	0.09	0.11	**< 0.01**	0.57	0.79	0.16
Blob-BigClass	**0.02**	0.81	0.26	0.36	**0.05**	0.15	0.18	0.07	0.29	0.41	0.20	0.42	0.69
Blob-LowCohesion	0.41	0.08	0.18	0.81	0.19	0.80	**< 0.01**	0.15	0.99	0.08	0.94	0.15	**< 0.01**
ClassDataShouldBePrivate-FieldPublic	**< 0.01**	**< 0.01**	0.08	**< 0.01**	0.52	0.12	**< 0.01**	**< 0.01**	0.13	0.30	0.16	0.59	0.45
ComplexClass-LargeClassOnly	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	0.08	0.93	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LargeClass-VeryLargeClassOnly	0.95	0.97	0.98	0.98	0.97	0.97	0.62	0.47	0.89	0.67	0.94	0.41	0.55
LargeClass-LowCohesionOnly	–	–	–	–	–	–	–	–	–	–	–	–	–
LazyClass-FewMethods	0.33	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LazyClass-NotComplexClass	–	–	–	–	–	–	–	–	–	–	–	–	–
LongMethod-LongMethodClass	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LongParameterList-LongParameterListClass	**< 0.01**	**< 0.01**	**< 0.01**	0.32	**< 0.01**	0.32	**< 0.01**	**< 0.01**	0.19	0.48	0.07	0.90	0.66
SpaghettiCode-ClassGlobalVariable	0.25	0.97	0.97	0.98	0.97	0.98	0.96	0.93	–	–	–	0.94	0.92
SpaghettiCode-ComplexMethod	–	–	–	–	–	–	–	–	–	–	–	–	–
SpaghettiCode-MethodNoParameter	–	–	–	–	–	–	–	–	–	–	–	–	–
SpaghettiCode-NoInheritance	–	–	–	–	–	–	–	–	–	–	–	–	–
SpeculativeGenerality-AbstractClass	0.82	0.19	0.14	0.97	0.98	0.12	**0.02**	**0.01**	**< 0.01**	**< 0.01**	0.36	0.15	0.36
SpeculativeGenerality-OneChildClass	–	–	–	–	–	–	–	–	–	–	–	–	–
SwissArmyKnife-MultipleInterface	0.30	0.76	0.82	0.34	0.35	0.63	0.42	**0.02**	0.39	0.08	0.43	0.98	**< 0.01**

Table 10: Logistic regression results for the correlations between issue-proneness and kinds of smells

Smells	1.0	2.0	2.1.1	2.1.2	2.1.3	3.0	3.0.1	3.0.2	3.2	3.2.1	3.2.2	3.3	3.3.1
AntiSingleton-NotClassGlobalVariable	0.55	0.21	**< 0.01**	0.59	**< 0.01**	0.59	**0.05**	0.23	**< 0.01**	**< 0.01**	0.32	**< 0.01**	**0.01**
Blob-ControllerClass	**0.01**	0.88	**< 0.01**	0.30	**0.03**	0.11	0.06	0.36	0.49	0.52	**0.03**	**0.02**	0.86
Blob-DataClass	**0.01**	**0.01**	**< 0.01**	**< 0.01**	**0.02**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**0.02**	**< 0.01**
Blob-BigClass	**< 0.01**	0.09	0.45	0.40	0.64	0.32	0.06	0.14	0.44	0.64	0.57	0.38	0.46
Blob-LowCohesion	0.16	0.39	0.33	**0.03**	0.05	0.24	**< 0.01**	0.61	**0.01**	0.82	0.29	0.92	0.47
ClassDataShouldBePrivate-FieldPublic	0.06	**< 0.01**	0.11	**< 0.01**	**< 0.01**	**< 0.01**	0.57	0.89	0.68	**< 0.01**	**0.02**	**0.03**	**< 0.01**
ComplexClass-LargeClassOnly	**< 0.01**	**0.01**	**0.03**	**0.01**	0.50	**< 0.01**	0.32	0.44	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LargeClass-VeryLargeClassOnly	0.96	0.96	0.95	0.95	0.95	0.79	0.99	0.86	0.44	0.29	0.90	0.80	0.49
LargeClass-LowCohesionOnly	–	–	–	–	–	–	–	–	–	–	–	–	–
LazyClass-FewMethods	0.16	0.10	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**0.02**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LazyClass-NotComplexClass	–	–	–	–	–	–	–	–	–	–	–	–	–
LongMethod-LongMethodClass	**0.02**	0.69	**< 0.01**	**< 0.01**	**< 0.01**	0.16	**< 0.01**	**0.04**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LongParameterList-LongParameterListClass	**< 0.01**	**< 0.01**	0.12	0.32	**< 0.01**	**< 0.01**	0.06	0.50	0.10	**< 0.01**	0.91	**< 0.01**	**< 0.01**
SpaghettiCode-ClassGlobalVariable	0.96	0.95	0.95	0.96	0.95	0.93	0.92	0.92	–	–	–	0.92	0.92
SpaghettiCode-ComplexMethod	–	–	–	–	–	–	–	–	–	–	–	–	–
SpaghettiCode-MethodNoParameter	–	–	–	–	–	–	–	–	–	–	–	–	–
SpaghettiCode-NoInheritance	–	–	–	–	–	–	–	–	–	–	–	–	–
SpeculativeGenerality-AbstractClass	**0.03**	0.69	**0.03**	0.15	0.85	0.31	0.18	0.52	0.16	0.05	0.19	0.27	**0.03**
SpeculativeGenerality-OneChildClass	–	–	–	–	–	–	–	–	–	–	–	–	–
SwissArmyKnife-MultipleInterface	**0.05**	0.79	**0.01**	**0.01**	0.90	**< 0.01**	**0.01**	**0.01**	**0.02**	0.19	0.85	0.42	**< 0.01**

Table 9: Logistic regression results for the correlations between change-proneness and kinds of smells

Smells	1.0	2.0	2.1.1	2.1.2	2.1.3	3.0	3.0.1	3.0.2	3.2	3.2.1	3.2.2	3.3	3.3.1
AntiSingleton-NotClassGlobalVariable	**< 0.01**	0.12	0.17	0.83	0.06	**< 0.01**	**< 0.01**	**< 0.01**	**0.03**	0.13	0.33	**< 0.01**	0.05
Blob-ControllerClass	**0.01**	**< 0.01**	0.57	**0.03**	0.26	0.31	0.93	0.36	0.87	0.82	0.71	0.96	**0.01**
Blob-DataClass	0.89	0.86	0.12	0.91	**0.04**	0.30	0.11	0.75	0.48	**0.03**	0.23	0.35	0.83
Blob-BigClass	0.29	0.05	0.34	0.23	**0.03**	0.33	0.06	0.07	0.85	0.07	0.11	0.14	0.07
Blob-LowCohesion	0.99	0.98	0.98	0.99	0.97	0.99	0.97	0.12	0.70	0.97	0.98	0.97	0.97
ClassDataShouldBePrivate-FieldPublic	0.22	**0.02**	0.21	**0.04**	0.25	**< 0.01**	**0.03**	**0.01**	**< 0.01**	0.93	0.31	0.16	0.08
ComplexClass-LargeClassOnly	**< 0.01**	**0.05**	**< 0.01**	**0.03**	0.92	0.10	0.06	**< 0.01**	**< 0.01**	0.96	**< 0.01**	0.70	**0.01**
LargeClass-VeryLargeClassOnly	1.00	1.00	1.00	1.00	1.00	1.00	0.99	0.96	0.98	0.99	0.99	0.13	0.97
LargeClass-LowCohesionOnly	–	–	–	–	–	–	–	–	–	–	–	–	–
LazyClass-FewMethods	0.11	**0.03**	**< 0.01**	**< 0.01**	0.24	**< 0.01**	**< 0.01**	**< 0.01**	**0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LazyClass-NotComplexClass	–	–	–	–	–	–	–	–	–	–	–	–	–
LongMethod-LongMethodClass	0.25	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**0.02**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**0.01**	**0.02**	**< 0.01**
LongParameterList-LongParameterListClass	0.06	0.54	0.40	0.77	**0.04**	**< 0.01**	0.33	**< 0.01**	0.59	**0.04**	0.20	0.08	0.91
SpaghettiCode-ClassGlobalVariable	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98	–	–	–	1.00	0.99
SpaghettiCode-ComplexMethod	–	–	–	–	–	–	–	–	–	–	–	–	–
SpaghettiCode-MethodNoParameter	–	–	–	–	–	–	–	–	–	–	–	–	–
SpaghettiCode-NoInheritance	–	–	–	–	–	–	–	–	–	–	–	–	–
SpeculativeGenerality-AbstractClass	0.98	0.12	0.71	0.98	0.59	0.94	0.95	**0.03**	0.96	0.83	**0.01**	0.60	0.95
SpeculativeGenerality-OneChildClass	–	–	–	–	–	–	–	–	–	–	–	–	–
SwissArmyKnife-MultipleInterface	0.56	0.30	0.88	0.12	0.93	0.98	0.35	**0.01**	0.18	**0.03**	0.76	0.97	**0.02**

Table 11: Logistic regression results for the correlations between unhandled exception-proneness and kinds of smells

[34] M. Vokac. Defect frequency and design patterns: An empirical study of industrial code. pages 904 – 917, December 2004.

[35] W. C. Wake. *Refactoring Workbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

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

[37] P. Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In P. Sousa and J. Ebert, editors, *Proceedings of 5^{th} Conference on Software Maintenance and Reengineering*, pages 77–84. IEEE Computer Society Press, March 2001.

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

[39] 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 Computer Society, 2007.

Antipatterns	1.0	2.0	2.1.1	2.1.2	2.1.3	3.0	3.0.1	3.0.2	3.2	3.2.1	3.2.2	3.3	3.3.1
AntiSingleton	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
Blob	0.39	0.15	0.24	**0.02**	0.30	0.77	**< 0.01**	**< 0.01**	0.85	0.27	0.06	**0.03**	0.20
ClassDataShouldBePrivate	**< 0.01**	**< 0.01**	0.06	**< 0.01**	0.35	0.15	**< 0.01**	**< 0.01**	0.26	0.68	0.06	0.39	0.26
ComplexClass	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LargeClass	0.95	0.97	0.97	0.98	0.97	0.97	0.80	0.39	0.95	0.71	0.94	0.50	0.67
LazyClass	0.35	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LongMethod	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LongParameterList	**< 0.01**	**< 0.01**	**< 0.01**	0.23	**< 0.01**	0.29	**< 0.01**	**< 0.01**	0.27	0.25	0.10	0.98	0.86
SpaghettiCode	0.26	0.97	0.97	0.98	0.97	0.98	0.96	0.93	–	–	–	0.94	0.92
SpeculativeGenerality	0.78	0.15	0.14	0.94	0.94	0.10	**0.02**	**0.01**	**< 0.01**	**< 0.01**	0.38	0.16	0.36
SwissArmyKnife	0.49	0.75	0.79	0.30	0.39	0.64	0.42	**0.02**	0.37	**0.05**	0.28	0.81	**< 0.01**

Table 13: Logistic regression results for the correlations between issue-proneness and kinds of antipatterns

Antipatterns	1.0	2.0	2.1.1	2.1.2	2.1.3	3.0	3.0.1	3.0.2	3.2	3.2.1	3.2.2	3.3	3.3.1
AntiSingleton	0.62	0.20	**< 0.01**	0.53	**< 0.01**	0.57	0.06	0.24	**< 0.01**	**< 0.01**	0.23	**< 0.01**	**0.03**
Blob	**< 0.01**	**< 0.01**	0.16	**< 0.01**	0.71	**0.05**	**< 0.01**	**< 0.01**	0.23	**0.01**	0.56	**< 0.01**	**< 0.01**
ClassDataShouldBePrivate	0.09	**< 0.01**	0.23	**< 0.01**	**< 0.01**	**< 0.01**	0.70	0.89	0.49	**< 0.01**	0.05	**0.03**	**< 0.01**
ComplexClass	**0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**0.03**	**< 0.01**	0.12	0.24	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LargeClass	0.96	0.95	0.95	0.95	0.95	0.76	0.97	0.87	0.54	0.28	0.91	0.72	0.71
LazyClass	0.09	0.12	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**0.02**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LongMethod	**0.02**	0.82	**< 0.01**	**< 0.01**	**< 0.01**	0.30	**< 0.01**	0.07	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.01**
LongParameterList	**< 0.01**	**< 0.01**	0.12	0.35	**< 0.01**	**< 0.01**	0.09	0.54	0.06	**< 0.01**	0.87	**< 0.01**	**< 0.01**
SpaghettiCode	0.96	0.95	0.95	0.96	0.95	0.93	0.92	0.92	–	–	–	0.92	0.92
SpeculativeGenerality	**0.03**	0.68	**0.03**	0.14	0.83	0.31	0.18	0.51	0.16	0.05	0.19	0.27	**0.03**
SwissArmyKnife	**0.03**	0.79	**0.01**	**0.01**	0.89	**< 0.01**	**0.01**	**0.01**	**0.01**	0.15	0.68	0.38	**< 0.01**

Table 12: Logistic regression results for the correlations between change-proneness and kinds of antipatterns

Antipatterns	1.0	2.0	2.1.1	2.1.2	2.1.3	3.0	3.0.1	3.0.2	3.2	3.2.1	3.2.2	3.3	3.3.1
AntiSingleton	< **0.01**	0.10	0.14	0.78	**0.04**	< **0.01**	< **0.01**	< **0.01**	**0.03**	0.14	0.45	< **0.01**	0.07
Blob	0.75	**0.01**	0.76	0.07	0.37	0.94	0.39	0.10	0.69	0.62	0.44	0.66	**0.01**
ClassDataShouldBePrivate	0.24	**0.02**	0.26	**0.04**	0.32	< **0.01**	**0.03**	**0.01**	< **0.01**	0.62	0.66	0.19	0.18
ComplexClass	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	0.13	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	0.21	< **0.01**
LargeClass	1.00	0.98	0.98	1.00	0.98	0.99	0.98	0.96	0.98	0.97	0.98	0.32	0.97
LazyClass	0.19	**0.01**	< **0.01**	< **0.01**	0.11	< **0.01**	< **0.01**	< **0.01**	**0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**
LongMethod	0.22	< **0.01**	< **0.01**	< **0.01**	< **0.01**	**0.02**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	**0.02**	< **0.01**
LongParameterList	0.10	0.51	0.43	0.65	**0.05**	< **0.01**	0.29	< **0.01**	0.55	**0.02**	0.34	0.07	0.95
SpaghettiCode	1.00	0.98	0.99	1.00	0.98	1.00	0.99	0.98	–	–	–	1.00	0.99
SpeculativeGenerality	0.98	0.09	0.72	0.97	0.63	0.95	0.87	**0.02**	0.97	0.82	**0.01**	0.60	0.98
SwissArmyKnife	0.91	0.33	0.88	0.08	0.90	0.97	0.39	**0.01**	0.18	**0.01**	0.65	0.97	**0.01**

Table 14: Logistic regression results for the correlations between unhandled exception-proneness and kinds of antipatterns