An Exploratory Study of the Impact of Code Smells on Software Change-proneness

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

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

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

Abstract

Code smells are poor implementation choices, thought to make object-oriented systems hard to maintain. In this study, we investigate if classes with code smells are more change-prone than classes without smells. Specifically, we test the general hypothesis: classes with code smells are not more change prone than other classes. We detect 29 code smells in 9 releases of Azureus and in 13 releases of Eclipse, and study the relation between classes with these code smells and class change-proneness. We show that, in almost all releases of Azureus and Eclipse, classes with code smells are more change-prone than others, and that specific smells are more correlated than others to change-proneness. These results justify a posteriori *previous work on the specification and detection of code smells and could help focusing quality assurance and testing activities.*

1 Context and Problem

In theory, code smells [12] are poor implementation choices, opposite to idioms [8] and, to some extent, to design patterns [13]. They are "poor" solutions to recurring implementation problems. In practice, code smells are in-between design and implementation: they may concern the design of a class, but they concretely manifest themselves in the source code as classes with specific implementation. They are usually revealed through particular metric values [22].

One example of a code smell is the ComplexClassOnly smell, which occurs in classes with a very high McCabe complexity when compared to other class in a system. At a higher level of abstraction, the presence of some specific code smells can, in turn, manifest in antipatterns [5], of which code smells are parts of. Studying the effects of antipatterns is, however, out of scope of this study and will be treated in other works.

Premise. Code smells are conjectured in the literature to hinder object-oriented software evolution. Yet, despite the existence of many works on code smells and antipatterns, no previous work has contrasted the change-proneness of classes with code smells with this of other classes to study empirically the impact of code smells on this aspect of software evolution.

Goal. We want to investigate the relations between these code smells and three types of code evolution phenomena. First, we study whether classes with code smells have an increased likelihood of changing than other classes. Second, we study whether classes with more smells than others are more change-prone. Third, we study the relation between particular smells and change-proneness.

Contribution. We present an exploratory study investigating the relations between 29 code smells and changes occurring to classes in 9 releases of Azureus and 13 releases of Eclipse. We show that code smells *do* have a negative impact on classes, that certain kinds of smells *do* impact classes more than others, and that classes with more smells exhibit higher change-proneness.

Relevance. Understanding if code smells increase the risk of classes to change is important from the points of view of both researchers and practitioners.

We bring evidence to researchers that (1) code smells *do* increase the number of changes that classes undergo, (2) the more smells a class has, the more change-prone it is, and (3) certain smells lead to more change-proneness than others. Therefore, this study justifies *a posteriori* previous work on code smells: within the limits of the threats to its validity, classes with code smells are more change-prone than others and therefore smells may indeed hinder software evolution;

we empirically support such a conjecture reported in the literature [12, 21, 32], which is the premise of this study.

We also provide evidence to practitioners—developers, quality assurance personnel, and managers—of the importance and usefulness of code smells detection techniques to assess the quality of their systems by showing that classes with smells are more likely to change often, thus impacting on the maintenance effort.

Organisation. Section 2 relates our study with previous works. Section 3 provides definitions and a description of our specification and detection approach for code smells. 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 7 concludes the study and outlines future work.

2 Related Work

Several works studied code smells, often in relation to antipatterns. We summarise these works as well as works aimed at relating metrics with software change-proneness.

Code Smell Definition and Detection. The first book on "antipatterns" in object-oriented development was written in 1995 by Webster [33]; his contribution includes conceptual, political, coding, and quality-assurance problems. Riel [25] defined 61 heuristics characterising good object-oriented programming to assess a system quality manually and improve its design and implementation. These heuristics are similar and–or precursor to code smells. Beck [12] defined 22 code smells, suggesting where developers should apply refactorings. Mäntylä [21] and Wake [32] proposed classifications for code smells. Brown *et al.* [5] described 40 antipatterns, which are often described in terms of lower-level code smells. 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 (semi-)automatically detect code smells (and antipatterns).

Several works proposed approaches to specify and detect code smells and antipatterns. They range from manual approaches, based on inspection techniques [29], to metric-based heuristics [22, 24], where code smells and–or 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] or using visualisation techniques [9, 27].

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 [19]. 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 [20, 30].

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 code smells and antipatterns and automatically detect them.

Design Patterns and Software Evolution. While code smells and antipatterns represent "poor" implementation and–or design choices, design patterns are considered to be "good" solutions to recurring design problems. Nevertheless, they may not always have positive effects on a system. Vokac [31] 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. Vokac's work inspired us in the use of logistic regression to analyse the correlations between code smells and change-proneness.

Bieman *et al.* [4] analysed four small and one large systems to study pattern change proneness. Other studies dealt with the changeability and resilience to change of design patterns and of specific pattern roles [2, 10, 18], and with their impact on the maintainability of a large commercial system [34].

While previous works investigated the impact of good design principles, *i.e.*, design patterns, on systems, we study the impact of poor implementation choices, *i.e.*, code smells, on software evolution.

Metrics and Software Evolution. Several studies, such as Basili *et al.*'s seminal work [3], used metrics as quality indicators. Cartwright and Shepperd [6] 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 [7] could be used to find classes that are likely to have higher fault rates. Gyimothy *et al.* [15] 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 CBO is the most discriminating metric. They also found LOC to discriminate fault-prone classes well. Zimmermann *et al.* [36] 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.* [11] 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.

Table 1. List of code smells considered in this study (definitions can be found [17]).

AbstractClass	ChildClass
ClassGlobalVariable	ClassOneMethod
ComplexClassOnly	ControllerClass
DataClass	FewMethods
FieldPrivate	FieldPublic
FunctionClass	HasChildren
LargeClass	LargeClassOnly
LongMethod	LongParameterListClass
LowCohesionOnly	ManyAttributes
MessageChainsClass	MethodNoParameter
MultipleInterface	NoInheritance
NoPolymorphism	NotAbstract
NotComplex	OneChildClass
ParentClassProvidesProtected	RareOverriding
TwoInheritance	

We do not claim that smells are better predictor of change-proneness than metrics, which instead provide more fine-grained and precise information to prediction models. On the other hand smells refer to specific programming styles and are therefore a better tool than metrics for developers. They are able to tell the developer whether a code artefact is bad or not, by means of thresholds defined over metrics. A ComplexClassOnly smells warns against excessive complexity, while McCabe cyclomatic complexity of WMC [7] leave such a judgement to the developer.

3 Code Smells

We use our previously proposed approach, DECOR (Defect dEtection for CORrection) [23], to specify and detect code smells. DECOR is based on a thorough domain analysis of code smells and antipatterns defined the literature, and provides a domain-specific language to specify code smells and antipatterns and methods to detect their occurrences automatically. It can be applied on any object-oriented system through the use of the PADL meta-model and POM framework. PADL is a meta-model to describe object-oriented systems [14]; parsers for AOL, C++, and Java are available. POM is a PADL-based framework that implements more than 60 metrics, including McCabe cyclomatic complexity, Brian Henderson-Sellers' cohesion metric, Chidamber and Kemerer metric suite, and statistical features, *e.g.*, computing and accessing metrics box-plots, to compensate for the effect of size.

Moha *et al.* [23] reported that the DECOR current detection algorithms for antipatterns ensure 100% recall and have a precision greater than 31% in the worst case, with an average greater than 60%. Although such a precision could be an issue in general, in this paper we use only the code smells detection algorithms of DECOR (antipatterns are defined in terms of code smells), which have a higher precision (80% on average), because the definition of a code smell is always more constraining than that of an antipattern, and includes less variability, such as fuzzy threshold or union between many rules.

The definition of a code smell includes several metrics with specific thresholds. In the current algorithms, the thresholds have been defined based on the literature and empirical studies.

Listing 1 shows the specifications of the ComplexClassOnly and LowCohesionOnly code smells. A class has the ComplexClassOnly smell if its McCabe complexity, computed as the sum of the McCabe complexities of all its methods, is very high with respect to the complexity of all the other class in the system. A class is with the LowCohesionOnly smell if it lacks cohesion, measured using Brian Henderson-Sellers' cohesion metric LCOM5 and evaluated as very high, *i.e.*, over the upper quartile when considering all classes. The values 20 indicates that, in these two code smells, a deviation from the upper quartile is possible, *e.g.*, classes with McCabe values that are up to 20% below the upper quartile are also complex classes.

In the following, we study 29 code smells [5, 12], as shown in Table 1. We choose these smells because they are representative of problems with data, complexity, size, and the features provided by classes. Their definitions and specifications are outside of the scope of this paper and are available in a longer technical report [17].

4 Study Definition and Design

The *goal* of our study is to investigate the relation between the presence of smells in classes and class change-proneness. The *quality focus* is the increase of maintenance effort and cost due to the presence of code smells.

The *perspective* is that of researchers, wanting to get evidence on the conjecture of the impact of smells on change proneness—to further our understanding of the impact of implementation and design choices on systems. Also, recommendations on code smells can be useful from the perspective of developers: the presence of change-prone classes likely increases the maintenance effort and cost. Finally, they can be viewed from the perspective of managers and–or quality assurance personnel, who could use code smell detection techniques to assess the change-proneness of in-house or to-be-acquired systems to better quantify their cost-of-ownerships.

The *context* of this study consists of the change history of two systems, Azureus and Eclipse, having a different size and belonging to different domains. Azureus[1], now known as "Vuze", is an open source BitTorrent client written in Java. BitTorrent is a protocol that allows to exchange files over the Internet. Eclipse[2] is an open-source integrated development environment used both in open-source communities and in industry. It is mostly written in Java, with

[1] http://azureus.sourceforge.net/

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

```
RULE_CARD : ComplexClassOnly {
   RULE : ComplexClassOnly { (METRIC: McCabe, VERY_HIGH, 20) };
};
RULE_CARD : LowCohesionOnly {
   RULE : LowCohesionOnly { (METRIC: LCOM5, VERY_HIGH, 20) } ;
};
```

Listing 1. Specification of the ComplexClassOnly and LowCohesionOnly code smells.

Table 2. Summary of the 9 releases of Azureus (changes are counted from one release to the next, Azureus 4.2.0.2 is thus excluded).

Dates	Releases	Number of		
		LOC	Classes	Changes
2008-06-16	3.1.0.0	589,049	2,954	669
2008-07-01	3.1.1.0	604,527	3,026	7,035
2008-10-15	4.0.0.0	690,116	3,045	383
2008-10-24	4.0.0.2	648,942	3,099	387
2008-11-20	4.0.0.4	651,642	3,111	1,589
2009-01-26	4.1.0.0	664,163	3,149	238
2009-02-05	4.1.0.2	664,554	3,149	478
2009-02-25	4.1.0.4	664,810	3,150	1,341
2009-03-23	4.2.0.0	680,238	3,210	106
Total	9	5,858,041	27,893	12,226

Table 3. Summary of the 13 analysed releases of Eclipse (changes are counted from one release to the next, Eclipse 3.4 is thus excluded).

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

C/C++ code used mainly for widget toolkits. Eclipse 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 [36].

We analysed 9 releases of Azureus, from release 3.1.0.0 to 4.2.0.0, in the years 2008-2009. We tracked the change history between releases using its Concurrent Versions System (CVS). Characteristics of the analysed releases are shown in Table 2. We analysed 13 releases of Eclipse available on the Internet between 2001 and 2008. Table 3 summarises the analysed releases and their key figures. On each considered release, we apply the 29 current code smell detection algorithms provided by DECOR to obtain the sets of classes with smells.

4.1 Research Questions

Based on the data collected from Azureus and Eclipse, our study aims at answering three research questions on the relationship between code smells and classes change-proneness,

- **RQ1:** *What is the relation between smells and change proneness?* We investigate whether classes with smells 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 with code smells and other classes*.
- **RQ2:** *What is the relation between the number of smells in a class and its change-proneness?* We are also interested to evaluate whether classes with a higher number of smells are more change-prone than others by testing the null hypothesis: H_{02}*: the number of smells in change-prone classes is not significantly higher than the number of smells in classes that do not change*.
- **RQ3:** *What is the relation between particular kinds of smells and change proneness?* Also, we analyse whether particular kinds of smells contribute more than others to changes by testing the null hypothesis: H_{03}*: classes with particular kinds of code smells are not significantly more change-prone 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. We have as many independent variables as kinds of code smells: we investigate the presence of 29 different kinds of smells. Each variable $s_{i,j,k}$

indicates the number of times a class i has a smell j in a release r_k. For RQ1, we aggregate these variables into a Boolean variable $S_{i,k}$ indicating whether a class i has at least one smell of any kind. For RQ2, we consider the number of changes $c_{i,k}$ a class i to underwent between r_k and r_{k+1}, and convert $c_{i,k}$ into a Boolean variable $C_{i,k}$ (*true* if the class underwent at least one change, *false* otherwise).

Dependent variables. The dependent variables measure the phenomena related to our independent variables. Our dependent variable for RQ1 and RQ3 is the class *change proneness*, which is measured, as above described, as the number of changes $c_{i,k}$ that a class i underwent between release r_k (in which it has some smells) and the subsequent release r_{k+1}. This number of changes is counted as the number of commits in the CVS. For RQ1 and RQ3, we are interested to distinguish classes that underwent, between two releases, at least one change. In RQ2, we compare the number of smells in change-prone classes with that in non-change-prone classes, using as dependent variable the total number of smells $st_{i,k}$ a class i has in a release r_k.

4.3 Analysis Method

In RQ1, to attempt rejecting H_{01}, we test whether the proportion of classes exhibiting (or not) at least one change, significantly varies between classes with (some) smells and other classes. We use Fisher's exact test [26], which checks whether a proportion vary between two samples. We also compute the *odds ratio* (OR) [26] that indicates the likelihood for an event to occur. The odds ratio is defined as the ratio of the odds p of an event occurring in one sample, *i.e.*, the odds that classes with some smells underwent a change (experimental group), to the odds q of the same event occurring in the other sample, *i.e.*, the odds that classes with no smell underwent a change (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 (smells), while an OR less than 1 that it is more likely in the second sample.

In RQ2, we use a (non-parametric) Mann-Whitney test to compare the number of smells in change-prone classes with the number of smells in non-change-prone classes. Non-parametric tests do not require any assumption on the underlying distributions. We also test the hypothesis with the (parametric) t-test. Other than testing the hypothesis, it is of practical interest to estimate the magnitude of the difference of the number of smells in classes with and without changes: we use the Cohen d effect size [26], which indicates the magnitude of the effect of a treatment on the dependent variables. The effect size is considered small for $0.2 \leq d < 0.5$, medium for $0.5 \leq d < 0.8$ and large for $d \geq 0.8$. For independent samples (to be used in the context of unpaired analyses, as in our case), it is defined as 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: $d = (M_1 - M_2)/\sigma$.

In RQ3, we use a logistic regression model [16], similarly to Vokac's study [31] to relate change-proneness with the presence of particular kinds of smells. In a logistic regression model, the dependent variable is commonly a dichotomous variable and, thus, assumes only two values $\{0, 1\}$, *e.g.*, changed or not. The multivariate logistic regression model is based on the formula:

$$\pi(X_1, X_2, \ldots, X_n) = \frac{e^{\beta_0+\beta_1 \cdot X_1+\ldots+\beta_n \cdot X_n}}{1 + e^{\beta_0+\beta_1 \cdot X_1+\ldots+\beta_n \cdot X_n}}$$

where (i) X_j are characteristics describing the modelled phenomenon, in our case the number of smells of kind j a class contains, *i.e.*, $s_{i,j,k}$ when the model is applied to the class i of release r_k[3]; (ii) β_j are the model coefficients; and (iii) $0 \leq \pi \leq 1$ is a value on the logistic regression curve. The closer the value is to 1, the higher is the likelihood that the class undergoes a change.

While in other contexts (*e.g.*, [15]), logistic regression models were used for prediction purposes; as in [31], 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_{03}. Then, for each smell and for the 9 analysed Azureus releases and for the 13 Eclipse releases, we count the number of times that the p-values obtained by the logistic regression were significant. It is also important to highlight that the procedure for building the logistic regression model discards variables that are highly correlated to others (*i.e.*, it only selects one variable)—that can happen between some smells—thus only selects a non-redundant set of features (smells) useful to warn against classes change-proneness.

5 Study Results

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

5.1 RQ1: Smells and Change Proneness

Tables 4 and 5 report, for each analysed release of Azureus and Eclipse, the number of classes (1) with smells and that changed; (2) with smells but that did not change; (3) without smells but with changes; and, (4) without smells nor changes. The tables also report the result of Fisher's exact test and ORs when testing H_{01}.

Results for Azureus in Table 4 show that the ORs are very high (always greater than 3); in most cases the odds

[3] for simplicity we omit i and k from the formula.

Table 4. Azureus: contingency table and Fisher test results for classes with at least one smell that underwent at least one change.

Releases	Smells-Changes	Smells-No Changes	No Smells-Changes	No Smells-No Changes	p-values	OR
3.1.0.0	220	1967	20	1433	$<$ **0.01**	8.01
3.1.1.0	564	1686	101	1381	$<$ **0.01**	4.57
4.0.0.0	83	2238	7	1519	$<$ **0.01**	8.05
4.0.0.2	106	2206	12	1510	$<$ **0.01**	6.04
4.0.0.4	435	1886	39	1484	$<$ **0.01**	8.77
4.1.0.0	50	2297	11	1533	$<$ **0.01**	3.03
4.1.0.2	112	2235	11	1533	$<$ **0.01**	6.98
4.1.0.4	112	2236	12	1532	$<$ **0.01**	6.39
4.2.0.0	37	2353	3	1580	$<$ **0.01**	8.28

Table 5. Eclipse: contingency table and Fisher test results for classes with at least one smell that underwent at least one change.

Releases	Smells-Changes	Smells-No Changes	No Smells-Changes	No Smells-No Changes	p-values	OR
1.0	2042	1731	417	448	$<$ **0.01**	1.27
2.0	3673	1373	767	236	**0.02**	0.82
2.1.1	2224	3838	193	964	$<$ **0.01**	2.89
2.1.2	2400	3664	359	798	$<$ **0.01**	1.46
2.1.3	2942	3125	516	642	**0.01**	1.17
3.0	3415	4880	684	1032	0.32	1.06
3.0.1	6216	2087	1294	423	0.69	0.97
3.0.2	5784	2520	1194	524	0.91	1.01
3.2	1819	9621	115	2210	$<$ **0.01**	3.63
3.2.1	2778	8680	291	2038	$<$ **0.01**	2.24
3.2.2	3321	8144	409	1921	$<$ **0.01**	1.92
3.3	1778	10844	145	2364	$<$ **0.01**	2.67
3.3.1	4337	8290	682	1830	$<$ **0.01**	1.40

for classes with smells to change is six times higher or more than for classes without smells. H_{01} rejection and the ORs provide *a posteriori* concrete evidence of the negative impact of smells on change-proneness. Developers should be wary of classes with smells, because they are more likely to be the subject of their maintenance effort. For Eclipse, except for the 3.0 release series, proportions are significantly different, thus allowing to reject H_{01}. There is a greater proportion of classes with smells that change with respect to other classes. In some cases (*e.g.*, releases 1.0, 2.0, 2.1.2, 2.1.3, and the 3.0 release series), ORs are close to 1, *i.e.*, the odds is even that a class with a smell changes or not. In the other releases, the odds of changing are 2 to 3.6 times in favour of classes with smells. We conclude that the odds to change are in general higher for classes with smells.

Table 6. Azureus: Mann-Whitney and t-test results for number of smells in classes that are change-prone or not.

Releases	M-W p	t-test p	Cohen d
3.1.0.0	$<$ **0.01**	$<$ **0.01**	0.72
3.1.1.0	$<$ **0.01**	$<$ **0.01**	0.71
4.0.0.0	$<$ **0.01**	$<$ **0.01**	1.01
4.0.0.2	$<$ **0.01**	$<$ **0.01**	0.86
4.0.0.4	$<$ **0.01**	$<$ **0.01**	0.83
4.1.0.0	$<$ **0.01**	$<$ **0.01**	0.59
4.1.0.2	$<$ **0.01**	$<$ **0.01**	0.93
4.1.0.4	$<$ **0.01**	$<$ **0.01**	0.85
4.2.0.0	$<$ **0.01**	$<$ **0.01**	1.02

Table 7. Eclipse: Mann-Whitney and t-test results for number of smells in classes that are change-prone or not.

Releases	M-W p	t-test p	Cohen d
1.0	0.79	**0.03**	0.06
2.0	$<$ **0.01**	$<$ **0.01**	-0.08
2.1.1	$<$ **0.01**	$<$ **0.01**	0.31
2.1.2	$<$ **0.01**	$<$ **0.01**	0.13
2.1.3	**0.04**	$<$ **0.01**	0.07
3.0	0.07	0.10	0.03
3.0.1	0.11	0.26	-0.03
3.0.2	0.12	0.28	-0.02
3.2	$<$ **0.01**	$<$ **0.01**	0.41
3.2.1	$<$ **0.01**	$<$ **0.01**	0.29
3.2.2	$<$ **0.01**	$<$ **0.01**	0.25
3.3	$<$ **0.01**	$<$ **0.01**	0.41
3.3.1	$<$ **0.01**	$<$ **0.01**	0.18

5.2 RQ2: Number of Smells and Change Proneness

Tables 6 and 7 report, for Azureus and Eclipse respectively, results of the Mann-Whitney two-tailed test, t-test, and Cohen d effect size, aimed at comparing the number of code smells in classes that changed or not. For Azureus, the p-values are always significant with a high effect size, indicating that for all the analysed releases change-prone classes are those with a higher number of smells. For Eclipse, results are significant (although with a small effect size), except for the 3.0 release series, where differences are not significant, thus confirming the findings from RQ1 regarding the limited relation of smells with change-proneness for this release series. In summary we can reject H_{02}.

5.3 RQ3: Kinds of Smells and Change Proneness

Tables 8 and 9 show the results of the logistic regression for the correlations between changes and the different kinds of code smells. In particular, the tables summarise the num-

Table 8. Azureus: number of significant p-values in the 9 analysed releases obtained by logistic regression for the correlations between change-proneness and kinds of smells. Boldface indicates significant p-values for at least 75% of the releases.

Smells	Proneness to Changes
AbstractClass	5
ChildClass	3
ClassGlobalVariable	2
ClassOneMethod	1
ComplexClassOnly	2
ControllerClass	2
DataClass	4
FewMethods	2
FieldPrivate	1
FieldPublic	2
FunctionClass	2
HasChildren	1
LargeClass	5
LargeClassOnly	–
LongMethod	–
LongParameterListClass	1
LowCohesionOnly	2
ManyAttributes	–
MessageChainsClass	4
MethodNoParameter	2
MultipleInterface	4
NoInheritance	3
NoPolymorphism	3
NotAbstract	**7**
NotComplex	2
OneChildClass	1
ParentClassProvidesProtected	–
RareOverriding	1
TwoInheritance	–

Table 9. Eclipse: number of significant p-values in the 13 analysed releases obtained by logistic regression for the correlations between change-proneness and kinds of smells. Boldface indicates significant p-values for at least 75% of the releases.

Smells	Proneness to Changes
AbstractClass	1
ChildClass	6
ClassGlobalVariable	2
ClassOneMethod	4
ComplexClassOnly	8
ControllerClass	4
DataClass	4
FewMethods	2
FieldPrivate	6
FieldPublic	8
FunctionClass	1
HasChildren	**11**
LargeClass	8
LargeClassOnly	–
LongMethod	9
LongParameterListClass	6
LowCohesionOnly	5
ManyAttributes	9
MessageChainsClass	**10**
MethodNoParameter	8
MultipleInterface	5
NoInheritance	–
NoPolymorphism	3
NotAbstract	1
NotComplex	**10**
OneChildClass	2
ParentClassProvidesProtected	–
RareOverriding	4
TwoInheritance	–

ber of analysed releases for which each kind of smells was significant in the logistic regression model. Smells that are significant for at least 75% of the releases (7 for Azureus, 10 for Eclipse) are highlighted in boldface. Detailed results of the logistic regression are in a longer technical report [17]. In Azureus, only the smell NotAbstract has a significant impact on change proneness in more than 75% of releases. AbstractClass and LargeClass resulted to be significant in more than 50% of the releases (5 out of 9). In Eclipse, the smells that have a significant effect on change-proneness for 75% of the releases or more are HasChildren, MessageChainsClass, and NotComplex. In summary, although results sometimes depend on the particular context—*e.g.*, system analysed and particular release—we can reject H_{03}, *i.e.*, there are smells that are more related to others to change-proneness.

As discussed in Section 4, the logistic regression procedure has pruned out from the model smells that are significantly correlated to others, initially inserted in the model as their definition in terms of metrics was different. We also performed a Spearman rank correlation analysis and identified pairs of smells that had a significant and high ($>$0.8) correlation. Such correlations were consistent in all the analysed releases of Azureus and Eclipse (see [17]). It is the case, for both Azureus and Eclipse, of LargeClass and LargeClassOnly, and, for Azureus, of NotAbstract and ParentClassProvidesProtected, and of RareOverriding and ParentClassProvidesProtected. In all these cases, the logistic regression discarded the second smell in the pair.

6 Discussion

This section discusses results reported in Section 5, along with threats to validity.

From Tables 4 and 5, it can be noticed that large proportions of classes in each release of both Azureus and Eclipse are with smells. This fact is not surprising because we used 29 code smell detection algorithms, which cover almost all aspects of the implementation and–or design of classes. Moreover, we do not consider that a class with a smell is necessarily the result of a "bad" implementation or design choice; only the concerned developers could make such a judgement. We do not exclude that, in a particular context, a code smell can be the best way to actually implement and–or design a (part of a) class. For example, automatically-generated parsers are often very large and complex classes. Only developers can evaluate their impact according to the context: it may be perfectly sensible to have these large and

complex classes if they come from a well-defined grammar.

In the following we discuss in details results for the two systems, Azureus and Eclipse.

6.1 Azureus

Classes with smells are more change-prone than those without smells in all the 9 releases of Azureus, and this with high odds ratios (3 to 8 times in favour of classes with smells). Moreover, the likelihood of change increases with the increase of the number of code smells in a class, underscoring the fact that code smells are costly and therefore should be detected and removed as early as possible during the development of a system. Across the 9 releases of Azureus, three particular kinds of code smells lead almost consistently to change-prone classes: the result for NotAbstract is statistically significant for 7 out of 9 releases, while AbstractClass and LargeClass results are statistically significant for 5 releases. By observing the presence of smells across releases, we found that, in each release, existing smells are generally removed from the system while some new are introduced in the context of new features addition. This explains why some smells are not visible in some releases, and that the logistic regression indicated some smells statistically significant only for some releases of Azureus. Finally, we found that smells often related to immature design and implementation (lack of use of abstraction, of polymorphism, etc.) often occur in the first releases, when developers might not have an idea of the future system size yet. This is the case for example of the smells NoInheritance, NoPolymorphism, and NotComplex.

Going to smells that are significantly correlated to changes in most of the releases, the NotAbstract smell generally occurs when a developer does not properly use abstraction to simplify her code. Given the extensive use of inheritance in Azureus, it is not surprising that parts of its design could be improved by abstracting some classes, because they may be the root of some important hierarchies. The second frequent code smell (AbstractClass) occurs when a class contains generic or abstract code not used at the time when it is introduced. Such code often exists in the system to support future pieces of functionality. It is not surprising that such a code smell is found in Azureus, since it is a common mistake developers make when using object-orientation [28]. Finally, the third frequent code smell (LargeClass), is a class that "is trying to do too much". Thus, it does not follow the good practice of divide-and-conquer, *i.e.*, decomposing a complex problem into smaller problems. Yet, some problems are not easily decomposable or, because of strong requirements imposed on the efficiency, decomposition might just constitute an overhead. Again, this is the case of Azureus, where complex algorithms are implemented, and where the efficiency (being it a network system) is a crucial issue.

6.2 Eclipse

Classes with smells (and, in particular, those with a higher number of smells) are more change-prone than others except in Eclipse 2.0 and in the Eclipse 3.0 series (including 3.0.1 and 3.0.2). We explain this by studying the release notes of Eclipse 2.0 and 3.0. For example, in the "New and Noteworthy" file coming along Eclipse 3.0[4] are described the many changes made to the system, including a new Rich Client Platform, new OSGi implementation, new look-and-feel, and so on. Similarly, but to a smaller extent, Eclipse 2.0, was a major advancement with respect to the Eclipse 1.0 series. Consequently, it is not surprising that many classes changed or were added, thus explaining the discrepancies in results for different releases. In summary, in releases such as the 3.0 series when a radical enhancement of the system was made in terms of new features, changes were not really related to smells.

Across the 13 Eclipse releases, three particular kinds of code smells lead to change-prone classes: HasChildren, MessageChainsClass, and NotComplex. The first, HasChildren, describes classes with many children. Given the extensive use of inheritance, and the frequent changes of class hierarchies in Eclipse (as it was previously found for Eclipse-JDT in particular [2]), it is not surprising that many classes have subclasses. The second, MessageChainsClass, characterises classes that use long message chains to perform their functionality. This makes the code dependent on relationships between potentially unrelated objects. Again, finding many classes with this smell is not surprising in a system with thousands of collaborating classes, known for its rich API. Finally, the third code smell, NotComplex, can also be explained by the extensive object-orientation, leading to many classes performing "atomic" functionality, with little complexity *per se*.

6.3 Threats to Validity

We now discuss the threats to validity of our study following the guidelines provided for case study research [35].

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 based on the CVS change log. In this context, we are just interested to check whether a class changes or not, rather than quantifying the amount of change, which is however possible and could be investigate in future work. Also, we are aware that the detection technique used includes our subjective understanding of the

[4] `http://archive.eclipse.org/eclipse/downloads/drops/R-3.0-200406251208/eclipse-news-R3.html`

smell definitions, as discussed in Section 3. However, as discussed, we are interested to relate smells "as they are defined in DECOR" [23] with change-proneness. For this reason, smell detection imprecision does not affect our study. Finally, we are aware that smells can be dependent each other. However, we relied on the logistic regression model building procedure to select the subset of non-correlated smells. In addition, we also performed a Spearman rank correlation analysis to identify highly-correlated smells—actually discarded by the logistic regression— as discussed in Section 5.

Threats to *internal validity* do not affect this particular study, being an exploratory study [35]. Thus, we cannot claim causation, but just relate the presence of smells with the occurrences of changes, although our discussion tries to explain why some smells could have been the cause of changes.

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 and Azureus source code repositories are available to obtain the same data. Finally, the data set on which our statistics have been computed is available on the Web[5].

Threats to *external validity* concern the possibility to generalise our findings. First, we are aware that our study has been performed on two systems, Eclipse and Azureus, thus generalisation will require further case studies. However, we limited such a threat by choosing two different systems, belonging to different domains, and studied a reasonably long history of both—spanning 9 releases for Azureus and 13 releases for Eclipse. Second, we used a particular yet representative set of smells. Different smells 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.

7 Conclusions and Future Work

In this paper, we reported an exploratory study, performed on 9 releases of Azureus and 13 releases of Eclipse, which provides empirical evidence of the negative impact of code smells on classes change-proneness. We showed that classes with smells are significantly more likely to be the subject of changes, than other classes. We also showed that some specific code smells, are more likely to be of concern during evolution.

[5] `http://www.ptidej.net/downloads/experiments/prop-WCRE09`

This exploratory study supports, within the limits of the threats to its validity, the conjecture in the literature that smells may have a negative impact on software evolution. We justify *a posteriori* previous work on smells, 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 smells during development and maintenance. Indeed, systems containing a high number of smells are likely to be more change prone: therefore, the cost-of-ownership of such systems will be higher than for other systems, because developers will have to put more effort.

Although previous studies correlated source code metrics with change-proneness, we believe that smells can provide to developers recommendations easier to understand than what metric profiles can do. In fact, smells are defined in terms of thresholds on metrics, thus they can tell whether some metric values are becoming critical or not, while in the absence of thresholds, such a decision is left to the developer, who might lack of skills and experiences to do judge. On the other hand, it must be clear that smells are not replacement to metrics in the ability of building change-proneness or fault-proneness prediction models.

Future work includes (i) replicating this study on other systems to assess the generality of our results; (ii) studying the effect of antipatterns, *i.e.*, problems at a higher level of abstraction than smells, and (iii), relating smells and antipatterns not only to change-proneness, but also to other phenomena such as the fault-proneness.

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

Acknowledgements. This work has been partly funded by the Canada Research Chair on Software Patterns and Patterns of Software.

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] L. Aversano, G. Canfora, L. Cerulo, C. Del Grosso, and M. Di Penta. An empirical study on the evolution of design patterns. In *proceedings of ESEC-FSE '07*, pages 385–394, New York, NY, USA, 2007. ACM Press.

[3] 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.

[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 Soft-*

ware Metrics Symposium (METRICS'03), pages 40–49. IEEE Computer Society, 2003.

[5] 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.

[6] 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.

[7] 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.

[8] J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1^{st} edition, August 1991.

[9] 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, April 2008.

[10] 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 (ICSM)*. IEEE Computer Society Press, September–October 2008.

[11] 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.

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

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

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

[15] 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.

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

[17] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc. An exploratory study of the impactof code smells on software change-proneness. Technical report, http://www.ptidej.net/downloads/experiments/prop-WCRE09/, June 2009.

[18] F. 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* 25^{th} *International Conference on Software Maintenance (ICSM)*. IEEE Computer Society Press, September 2009.

[19] 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.

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

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

[22] 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.

[23] N. Moha, Y.-G. Guéhéneuc, L. Duchien, , and A.-F. Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, To appear.

[24] 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.

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

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

[27] 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.

[28] N. A. Solter and S. J. Kleper. *Professional C++*. Wiley Publishing, Inc, 10475 Crosspoint Boulevard, Indianapolis, IN 46256, 2005.

[29] 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.

[30] 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.

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

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

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

[34] 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.

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

[36] 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.

A Detailled Definitions of the Design Smells

In this study we focused on the following code smells:

AbstractClass: This code smell is characteristic of the Speculative Generality Antipattern. This odor exists when we have generic or abstract code that isn't actually needed today. Such code often exists to support future behavior, which may or may not be necessary in the future.

ChildClass: This code smell occurs when the number of methods declared in a class and the number of it's declared attributes is very high. It is a symptom of poor object decomposition. The public interface of the class differing greatly from the one of its super-class. This code smell characterises the Tradition Breaker antippatern.

ClassGlobalVariable: This code smell occurs when a class declares public class variable that are used as "global variable" in procedural programming.

ClassOneMethod: This code smell occurs when a class has only one method.

ComplexClassOnly: This code smell is present when a class 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.

ControllerClass: This odor is present when a class monopolises most of the processing done by a system, takes most of the decisions, and closely directs the processing of other classes.

DataClass: This code smell is present when a class contains only data and performs no processing on these data. It is composed of highly cohesive fields and accessors.

FewMethod: This code smell characterise Lazy classes that declare few methods.

FieldPrivate: This code smell is present when many private fields are declared. It's generally symptomatic of the Functional Decomposition antipattern.

FieldPublic: This code smell is symptomatic of the Class Data Should Be Private antippatern. 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.

FunctionClass: This code smell occurs when we have a main class, i.e., a class with a procedural name, such as Compute or Display. It is symptomatic of the Functional Decomposition antipattern.

HasChildren: This code smell describes classes with many children.

LargeClass: This odor concerns classes that are trying to do too much. These classes do not follow the good practice of divide-and-conquer which consists of decomposing a complex problem into smaller problems. These classes also have low cohesion.

LargeClassOnly: This code smell concerns classes with a very high number of attributes and/or methods defined.

LongMethod: This odor 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.

LongParameterListClass: This odor corresponds to a method with high number of parameters. This smell occurs when the method has more than four parameters. Long lists of parameters in a method, though common in procedural code, are difficult to understand and likely to be volatile.

LowCohesionOnly: This code smell characterises the lack of cohesion in a class.

ManyAttributes: This code smell occurs when the number of attributes declared in a class is too high.

MessageChainsClass: This code smell is present when you see a long sequence of method calls or temporary variables to get some data. This chain makes the code dependent on the relationships between many potentially unrelated objects.

MethodNoParameter: This code smell occurs when a class declares methods with no parameter.

MultipleInterface: This code smell occurs when a class implements a high number of interfaces. It is generally symptomatic of the Swiss Army Knife antipattern.

NoInheritance: This odor is present when inheritance is scarcely used.

NoPolymorphism: This odor is present when polymorphism is scarcely used.

NotAbstract: This odor occurs when a developer haven't yet seen how a higher-level abstraction can clarify or simplify his code.

NotClassGlobalVariable: This odor manifest itself in the anipattern Anti-Singleton when a class 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.

NotComplex: This code smell characterises classes performing "atomic" functionality, with little complexity.

OneChildClass: This code smell occurs when a class does not have child class.

ParentClassProvidesProtected: This code smell occurs when a subclass does not use attributes and/or methods protected inherited by a parent.

RareOverriding: This code smell occurs when a class rarely overrides inherited attributes and/or methods.

TwoInheritance: This odor characterises a hierarchy with a depth greater than two.

B Correlation between smells

Table 10. Correlation between smells (the table indicates the number of releases for which the Spearman rank correlation is higher than 0.8)

Smell 1	Smell 2	Releases
	AZUREUS	
ParentClassProvidesProtectedDetection_ParentClassProvidesProtected	NotAbstract_NotAbstract	9
NotClassGlobalVariable_NotClassGlobalVariable	ClassGlobalVariable_ClassGlobalVariable	9
RareOverridingDetection_RareOverriding	NotAbstract_NotAbstract	9
RareOverridingDetection_RareOverriding	ParentClassProvidesProtectedDetection_ParentClassProvidesProtected	9
	ECLIPSE	
NotClassGlobalVariable_NotClassGlobalVariable	ClassGlobalVariable_ClassGlobalVariable	13

C Detailled Number of Code Smells per Releases

Smells	Number of Smells per Azureus Release								
	3.1.0.0	3.1.1.0	4.0.0.0	4.0.0.2	4.0.0.4	4.1.0.0	4.1.0.2	4.1.0.4	4.2.0.0
AbstractClass_AbstractClass	111	116	116	115	116	119	119	119	125
ChildClassDetection_ChildClass	581	601	589	586	588	575	575	576	581
ClassGlobalVariable_ClassGlobalVariable	247	250	158	159	158	157	156	157	159
ClassOneMethod_ClassOneMethod	378	389	418	413	410	330	330	330	353
ComplexClassOnlyDetection_ComplexClassOnly	308	310	303	303	303	310	310	310	318
ControllerClassDetection_ControllerClass	262	262	273	273	273	278	278	278	292
DataClass_DataClass	489	480	488	488	492	493	493	493	497
FewMethodsDetection_FewMethods	303	304	284	284	289	229	229	229	233
FieldPrivate_FieldPrivate	769	811	769	766	772	788	788	788	801
FieldPublic_FieldPublic	249	261	283	282	282	283	282	283	285
FunctionClassDetection_FunctionClass	13	13	14	14	14	14	14	14	14
HasChildrenDetection_HasChildren	203	211	221	220	221	223	223	223	223
LargeClassDetection_LargeClass	167	170	178	177	177	187	187	187	191
LargeClassOnlyDetection_LargeClassOnly	167	170	178	177	177	187	187	187	191
LongMethodClassDetection_LongMethodClass	449	456	481	481	479	482	481	482	491
LongMethodDetection_LongMethod	322	335	344	344	350	352	352	352	358
LongParameterListClassDetection_LongParameterListClass	271	276	285	285	285	289	289	289	294
LowCohesionDetection_LowCohesion	18	19	25	21	21	69	69	69	69
LowCohesionOnlyDetection_LowCohesionOnly	18	19	25	21	21	69	69	69	69
ManyAttributesDetection_ManyAttributes	206	211	224	223	223	229	229	229	230
MessageChainsClassDetection_MessageChainsClass	663	752	791	787	780	828	828	829	843
MethodNoParameterDetection_MethodNoParameter	83	79	81	82	82	76	76	76	79
MultipleInterface_MultipleInterface	57	55	64	64	64	65	65	65	67
NoInheritanceDetection_NoInheritance	1380	1379	1407	1404	1413	1434	1434	1435	1475
NoPolymorphism_NoPolymorphism	1719	1756	1811	1798	1805	1828	1828	1828	1866
NotAbstract_NotAbstract	2096	2158	2242	2229	2236	2259	2259	2260	2300
NotClassGlobalVariable_NotClassGlobalVariable	247	250	158	159	158	157	156	157	159
NotComplexClassDetection_NotComplexClass	1260	1277	1352	1338	1341	1354	1354	1354	1390
NotComplexDetection_NotComplex	891	889	940	930	932	941	941	942	972
OneChildClassDetection_OneChildClass	87	89	98	98	98	101	101	101	99
ParentClassProvidesProtectedDetection_ParentClassProvidesProtected	2207	2274	2358	2344	2352	2378	2378	2379	2425
RareOverridingDetection_RareOverriding	2034	2100	2180	2166	2170	2205	2205	2206	2247
TwoInheritanceDetection_TwoInheritance	827	895	951	940	939	944	944	944	950
TOTAL	19082	19617	20089	19971	20021	20233	20229	20240	20646

Table 11. Summary of the numbers of smells in the analysed releases of Azureus.

D Detailled Logistic Regression Results

This appendix provides tables with more details on the results of applying logistic regression for the correlations between changes and kinds of smells.

Smells	Number of Smells per Eclipse Release												
	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
AbstractClass_AbstractClass	370	487	539	539	539	772	772	772	1141	1141	1143	1284	1284
ChildClassDetection_ChildClass	949	1233	1468	1467	1468	2055	2056	2057	2955	2957	2959	3352	3358
ClassGlobalVariable_ClassGlobalVariable	330	478	615	617	618	975	980	980	1605	1613	1613	1842	1843
ClassOneMethod_ClassOneMethod	301	453	557	557	557	769	768	768	1031	1031	1030	1161	1162
ComplexClassOnlyDetection_ComplexClassOnly	460	614	716	719	719	1007	1004	1001	1517	1516	1514	1854	1856
ComplexClass_ComplexClassOnly	56	0	0	0	0	0	0	0	0	0	0	0	0
ComplexClass_LargeClassOnly	28	0	0	0	0	0	0	0	0	0	0	0	0
ControllerClassDetection_ControllerClass	234	214	282	282	282	491	493	493	683	683	684	816	816
DataClass_DataClass	733	945	1113	1114	1115	1652	1651	1650	2276	2280	2285	2485	2486
FewMethodsDetection_FewMethods	238	299	416	416	416	493	483	483	614	616	614	665	666
FieldPrivate_FieldPrivate	1162	1934	2414	2414	2414	3133	3142	3144	4315	4326	4331	4702	4709
FieldPublic_FieldPublic	382	412	448	450	454	1466	1470	1470	2196	2197	2198	2396	2400
FunctionClassDetection_FunctionClass	51	68	73	73	73	87	87	87	170	170	170	196	196
HasChildrenDetection_HasChildren	631	857	936	936	936	1232	1233	1232	1768	1768	1770	1964	1964
LargeClassDetection_LargeClass	218	326	391	392	393	571	569	569	915	918	918	1041	1041
LargeClassOnlyDetection_LargeClassOnly	218	326	391	392	393	571	569	569	915	918	918	1041	1041
LongMethodClassDetection_LongMethodClass	958	1336	1611	1610	1612	2154	2157	2159	3026	3029	3029	3568	3577
LongMethodDetection_LongMethod	680	920	1122	1121	1116	1495	1495	1497	2147	2143	2144	2497	2506
LongParameterListClassDetection_LongParameterListClass	324	598	693	694	695	979	980	981	1130	1134	1136	1339	1341
LowCohesionDetection_LowCohesion	59	95	114	114	114	148	152	152	151	151	154	155	155
LowCohesionOnlyDetection_LowCohesionOnly	59	95	114	114	114	148	152	152	151	151	154	155	155
ManyAttributesDetection_ManyAttributes	369	420	518	519	519	664	665	665	1003	1009	1010	1127	1128
MessageChainsClassDetection_MessageChainsClass	1043	1438	1558	1558	1559	1803	1816	1815	2673	2693	2694	3038	3041
MethodNoParameterDetection_MethodNoParameter	180	209	236	237	237	361	362	362	667	670	673	758	758
MultipleInterface_MultipleInterface	67	43	38	38	38	68	68	68	93	94	95	111	112
NoInheritanceDetection_NoInheritance	1884	2443	3213	3215	3217	4725	4732	4733	6305	6319	6321	6876	6881
NoPolymorphism_NoPolymorphism	3027	4022	4786	4787	4787	6575	6581	6582	9096	9108	9113	10052	10053
NotAbstract_NotAbstract	3411	4567	5533	5535	5538	7534	7542	7543	10311	10329	10334	11359	11364
NotClassGlobalVariable_NotClassGlobalVariable	330	478	615	617	618	975	980	980	1605	1613	1613	1842	1843
NotComplexClassDetection_NotComplexClass	2048	2699	3231	3232	3232	4337	4321	4320	5783	5791	5802	6216	6211
NotComplexDetection_NotComplex	1236	1592	1907	1908	1909	2605	2597	2597	3449	3452	3459	3734	3731
OneChildClassDetection_OneChildClass	237	333	371	371	371	493	494	493	735	735	735	834	834
ParentClassProvidesProtectedDetection_ParentClassProvidesProtected	3781	5054	6072	6074	6077	8306	8314	8315	11452	11470	11477	12642	12647
RareOverridingDetection_RareOverriding	2114	2974	3716	3714	3718	5815	5821	5825	8302	8314	8320	9272	9275
TwoInheritanceDetection_TwoInheritance	1899	2616	2865	2865	2866	3587	3588	3588	5166	5170	5175	5801	5802
TOTAL	30067	40578	48672	48691	48714	68046	68094	68102	95346	95509	95585	106175	106236

Table 12. Summary of the numbers of smells in the analysed releases of Eclipse.

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
AbstractClass_AbstractClass	0.68	0.53	0.13	0.27	0.17	0.78	0.81	0.93	0.10	**0.03**	0.09	0.95	0.93
ChildClassDetection_ChildClass	0.08	0.07	0.49	0.57	0.09	**0.01**	0.39	0.94	< **0.01**	**0.04**	< **0.01**	**0.01**	< **0.01**
ClassGlobalVariable_ClassGlobalVariable	**0.04**	0.63	0.46	0.15	0.06	0.06	0.98	0.98	0.12	0.12	0.10	**0.01**	0.40
ClassOneMethod_ClassOneMethod	0.07	0.92	0.19	**0.01**	0.38	< **0.01**	0.76	0.41	0.76	0.32	0.46	**0.03**	**0.02**
ComplexClassOnlyDetection_ComplexClassOnly	0.69	0.10	< **0.01**	< **0.01**	0.07	< **0.01**	0.37	0.99	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**
ControllerClassDetection_ControllerClass	< **0.01**	0.88	**0.01**	0.84	0.44	0.39	< **0.01**	< **0.01**	0.53	0.82	0.78	0.08	0.05
DataClass_DataClass	0.45	0.88	**0.01**	0.64	0.09	0.06	0.70	0.76	< **0.01**	< **0.01**	0.83	0.37	< **0.01**
FewMethodsDetection_FewMethods	0.99	< **0.01**	0.10	0.30	0.62	0.16	0.19	0.28	0.30	0.31	0.63	0.06	**0.04**
FieldPrivate_FieldPrivate	< **0.01**	< **0.01**	< **0.01**	< **0.01**	0.21	0.82	0.10	0.37	0.09	0.18	**0.02**	0.08	< **0.01**
FieldPublic_FieldPublic	**0.04**	< **0.01**	0.69	< **0.01**	< **0.01**	< **0.01**	< **0.01**	**0.01**	0.46	0.30	0.91	0.18	< **0.01**
FunctionClassDetection_FunctionClass	0.74	0.40	0.42	0.31	0.57	< **0.01**	0.73	0.22	0.79	0.30	0.92	0.67	0.90
HasChildrenDetection_HasChildren	< **0.01**	< **0.01**	**0.02**	< **0.01**	**0.02**	**0.02**	< **0.01**	< **0.01**	0.07	**0.01**	< **0.01**	0.09	< **0.01**
LargeClassDetection_LargeClass	**0.04**	0.51	0.88	0.76	0.58	0.62	< **0.01**	< **0.01**	< **0.01**	**0.03**	**0.01**	**0.02**	**0.01**
LargeClassOnlyDetection_LargeClassOnly	–	–	–	–	–	–	–	–	–	–	–	–	–
LongMethodClassDetection_LongMethodClass	**0.05**	**0.01**	**0.02**	**0.04**	0.58	0.29	0.13	0.67	**0.03**	0.11	0.36	**0.04**	0.82
LongMethodDetection_LongMethod	**0.03**	< **0.01**	0.57	**0.01**	0.13	0.60	0.60	0.16	0.18	0.19	0.14	0.13	0.09
LongParameterListClassDetection_LongParameterListClass	0.06	< **0.01**	0.06	**0.01**	0.12	< **0.01**	< **0.01**	< **0.01**	0.60	**0.01**	0.47	0.09	0.33
LowCohesionDetection_LowCohesion	0.09	0.29	**0.02**	0.51	< **0.01**	**0.01**	< **0.01**	0.23	0.76	**0.05**	0.35	0.81	0.06
LowCohesionOnlyDetection_LowCohesionOnly	–	–	–	–	–	–	–	–	–	–	–	–	–
ManyAttributesDetection_ManyAttributes	< **0.01**	0.22	0.24	0.10	**0.02**	**0.01**	< **0.01**	< **0.01**	0.10	< **0.01**	< **0.01**	< **0.01**	**0.01**
MessageChainsClassDetection_MessageChainsClass	**0.03**	0.23	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	**0.02**	0.32	< **0.01**	< **0.01**	0.05	< **0.01**
MethodNoParameterDetection_MethodNoParameter	< **0.01**	< **0.01**	< **0.01**	< **0.01**	0.08	< **0.01**	< **0.01**	< **0.01**	0.10	0.87	0.30	< **0.01**	0.86
MultipleInterface_MultipleInterface	0.21	0.78	0.08	**0.04**	0.87	**0.03**	**0.04**	**0.01**	0.25	0.31	0.49	0.78	< **0.01**
NoInheritanceDetection_NoInheritance	0.84	0.68	0.65	0.58	0.11	0.62	0.98	0.93	0.55	0.33	0.22	0.57	0.99
NoPolymorphism_NoPolymorphism	0.32	0.38	0.57	0.51	**0.02**	0.27	0.30	**0.03**	0.77	0.60	0.65	**0.02**	0.20
NotAbstract_NotAbstract	0.47	0.80	0.05	0.18	0.31	0.84	0.97	0.93	0.13	**0.04**	0.05	0.95	0.94
NotClassGlobalVariable_NotClassGlobalVariable	–	–	–	–	–	–	–	–	–	–	–	–	–
NotComplexClassDetection_NotComplexClass	0.83	0.87	< **0.01**	< **0.01**	< **0.01**	< **0.01**	0.73	0.07	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**
NotComplexDetection_NotComplex	0.59	< **0.01**	< **0.01**	**0.02**	< **0.01**	< **0.01**	0.33	0.07	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**
OneChildClassDetection_OneChildClass	**0.04**	0.83	0.10	0.71	0.31	0.57	0.99	0.40	0.30	0.44	0.45	0.66	**0.01**
ParentClassProvidesProtectedDetection_ParentClassProvidesProtected	–	–	–	–	–	–	–	–	–	–	–	0.95	0.93
RareOverridingDetection_RareOverriding	**0.01**	0.88	0.09	< **0.01**	< **0.01**	0.06	0.46	0.88	0.95	0.13	0.13	**0.03**	0.41
TwoInheritanceDetection_TwoInheritance	0.80	0.76	0.78	0.63	0.17	0.59	0.62	0.93	0.85	0.43	0.22	0.60	0.74

Table 13. Eclipse: summary of logistic regression

Smells	3.1.0.0	3.1.1.0	4.0.0.0	4.0.0.2	4.0.0.4	4.1.0.0	4.1.0.2	4.1.0.4	4.2.0.0
AbstractClass_AbstractClass	< **0.01**	< **0.01**	< **0.01**	**0.03**	**0.01**	0.86	0.71	0.99	0.12
ChildClassDetection_ChildClass	0.55	**0.01**	0.92	0.47	0.51	0.39	**0.01**	< **0.01**	0.81
ClassGlobalVariable_ClassGlobalVariable	0.53	< **0.01**	0.40	0.50	< **0.01**	0.76	0.26	0.90	0.41
ClassOneMethod_ClassOneMethod	0.70	0.11	0.99	0.28	< **0.01**	0.67	0.16	0.49	0.60
ComplexClassOnlyDetection_ComplexClassOnly	**0.03**	< **0.01**	0.40	0.06	0.40	0.54	0.58	0.44	0.56
ControllerClassDetection_ControllerClass	< **0.01**	**0.04**	0.66	0.43	0.16	0.77	0.77	0.56	0.98
DataClass_DataClass	< **0.01**	< **0.01**	0.97	0.06	< **0.01**	0.96	**0.01**	0.15	0.98
FewMethodsDetection_FewMethods	< **0.01**	0.46	0.55	0.58	< **0.01**	0.31	0.47	0.59	0.13
FieldPrivate_FieldPrivate	< **0.01**	0.07	0.74	0.43	0.23	0.32	0.57	0.16	0.68
FieldPublic_FieldPublic	< **0.01**	0.22	0.06	0.98	0.19	0.68	< **0.01**	0.12	0.41
FunctionClassDetection_FunctionClass	0.74	0.67	< **0.01**	< **0.01**	0.40	0.99	0.09	0.16	1.00
HasChildrenDetection_HasChildren	0.41	0.65	0.08	**0.04**	0.91	0.40	0.49	0.96	0.66
LargeClassDetection_LargeClass	0.32	< **0.01**	0.06	**0.01**	< **0.01**	**0.03**	0.05	**0.02**	0.05
LargeClassOnlyDetection_LargeClassOnly	–	–	–	–	–	–	–	–	–
LongMethodClassDetection_LongMethodClass	0.56	0.28	0.28	0.92	0.46	0.22	0.08	0.94	0.59
LongMethodDetection_LongMethod	0.99	0.45	0.45	0.95	0.44	0.30	0.38	0.56	0.74
LongParameterListClassDetection_LongParameterListClass	0.20	< **0.01**	0.16	0.24	0.17	0.79	0.39	0.70	0.31
LowCohesionDetection_LowCohesion	0.97	0.98	**0.05**	0.61	**0.05**	0.99	0.98	0.98	0.99
LowCohesionOnlyDetection_LowCohesionOnly	–	–	–	–	–	–	–	–	–
ManyAttributesDetection_ManyAttributes	0.62	0.24	0.30	0.30	0.31	0.46	0.16	0.91	0.58
MessageChainsClassDetection_MessageChainsClass	< **0.01**	0.49	< **0.01**	0.08	0.21	0.33	< **0.01**	< **0.01**	0.61
MethodNoParameterDetection_MethodNoParameter	**0.04**	0.50	< **0.01**	0.86	0.45	0.98	0.16	0.51	0.99
MultipleInterface_MultipleInterface	0.81	< **0.01**	< **0.01**	0.06	< **0.01**	0.98	0.29	**0.01**	0.35
NoInheritanceDetection_NoInheritance	< **0.01**	< **0.01**	0.65	0.09	< **0.01**	0.73	0.05	0.79	0.76
NoPolymorphism_NoPolymorphism	< **0.01**	0.54	0.88	0.63	0.89	0.20	**0.01**	0.87	**0.04**
NotAbstract_NotAbstract	< **0.01**	< **0.01**	< **0.01**	< **0.01**	< **0.01**	0.68	0.09	**0.01**	**0.01**
NotClassGlobalVariable_NotClassGlobalVariable	–	–	–	–	–	–	–	–	–
NotComplexClassDetection_NotComplexClass	**0.02**	0.76	0.85	0.85	0.12	0.81	0.94	0.36	0.38
NotComplexDetection_NotComplex	0.15	< **0.01**	< **0.01**	0.75	0.14	0.14	0.22	0.76	0.59
OneChildClassDetection_OneChildClass	0.18	0.34	0.80	**0.01**	0.39	0.84	0.23	0.26	0.44
ParentClassProvidesProtectedDetection_ParentClassProvidesProtected	–	–	–	–	–	–	–	–	–
RareOverridingDetection_RareOverriding	0.06	0.20	0.17	0.17	**0.02**	0.20	0.53	0.30	0.69
TwoInheritanceDetection_TwoInheritance	–	–	–	–	–	–	–	–	–

Table 14. Azureus: summary of logistic regression