On the Analysis of Co-occurrence of Anti-patterns and Clones

Fehmi Jaafar[1], Angela Lozano[2], Yann-Gaël Guéhéneuc[3], and Kim Mens[4]
[1] Concordia University of Edmonton, Alberta. Canada.
[2] The Software Languages Lab. Vrije Universiteit Brussel, Belgium
[3] Ptidej Team, École Polytechnique de Montréal, QC, Canada
[4] The RELEASeD group, Université catholique de Louvain, Belgium
E-Mails: fehmi.jaafar@concordia.ab.ca, alozanor@vub.ac.be, yann-gael.gueheneuc@polymtl.ca, Kim.Mens@uclouvain.be

***Abstract*—**
In software engineering, a smell is a part of a software system's source code with a poor quality and that may indicate a deeper problem. Although many kinds of smells have been studied to analyze their causes, their behavior, and their impact on software quality, those smells typically are studied independently from each other. However, if two smells coincide inside a class, this could increases their negative effect (*e.g.,* spaghetti code that is being cloned across the system). In this paper we report results from an empirical study conducted to examine the relationship between two specific kind of smells: code clones and anti-patterns. We conducted our study on three open-source software systems: Azureus, Eclipse, and JHotDraw. Results show that between 32% and 63% of classes in the analysed systems present co-occurrence of smells, and that such classes could be five times more risky in term of fault-proneness.

***Index Terms*—Code smells, Anti-patterns; Clones; Faults proneness; Software Quality;**

I. Introduction

To help software developers and analysts to maintain a system and evaluate its quality, Beck and Fowler [**?**] introduced the concept of smells in software system as "bad structures in the code that suggest the possibility of refactoring". Indeed, smells are indicators of a poor quality of a source code with respect to reusability, maintainability, efficiency, and portability [**?**]. Beck and Fowler presented a list of 22 smells discovered in object-oriented source code [**?**]. One of these smells indicated code clones, others smells are considered as indicators of the presence of design smells, *i.e.,* anti-patterns.

On the one hand, clones are often considered as a mark of poor or lazy programming style because they deteriorate the changeability of a source code: if a clone needs to be maintained, we shall propagate the maintenance changes to the rest of duplicated instances [**?**], [**?**]. Other negative consequences attributed to clones are unintentionally incomplete changes and bugs [**?**]. However, in the software engineering literature, claims and counterclaims about whether clones are harmful or not remains an open and interesting research question. For example, Kasper and Godfrey [**?**] have discussed the advantages and disadvantages associated with using specific patterns of clones. In addition, they have showed how code clones are usually used as a principled engineering tool.

On the other hand, another important subset of smells specified anti-patterns, design defects that describe bad solutions to recurring design problems [**?**]. In contrast to code-level smells like clones, anti-patterns are coarse-grained and can be represented at the design level. Previous work [**?**], [**?**] have reported that the existence of anti-patterns in a software source code co-occurred with a high probability of change and fault-proneness, and thus, the maintenance activities of a the software system became more complicated and difficult.

As it was not clear from previous work [**?**], [**?**] if the interference of these smells could increase or decrease the risk of bugs, we report in this paper an empirical study to analyze the resulting impact of the interference of these code smells on fault-proneness. In addition, we provide empirical evidences of co-occurrences of two different smells indicating two different problems: duplicating code and poor design.

Indeed, using smell occurrences extracted from three open-source systems, we answer the following research questions:

- **RQ1:** *What is the percentage of classes participating in anti-patterns and clones?* This question provides quantitative data on the number of cases with which anti-patterns and clones occur and co-occur in the releases of the studied systems. We find that, between 32% and 63% of classes in the five analysed systems present co-occurrence of smells.
- **RQ2:** *What is the impact on fault-proneness for a class that participates in anti-patterns and clones?* Our motivation is to investigate if the co-occurrence of anti-patterns and clones is related to higher or lower fault-proneness. We find that classes presenting co-occurrence of anti-patterns and clones are at least 3 times more risky to occur faults than the rest of classes.

The rest of the paper will contain the following sections: Section **??** relates our study with previous work. Section **??** describes the steps of our data analysis process to examine the co-occurrence of anti-patterns and clones in software systems. Section **??** reports the design of our empirical study. Section

?? presents the answers of the research questions based on the study results, while Section **??** reports several observations from our results and discuss the threats to validity of our study. Finally, Section **??** concludes our paper and outlines future work.

II. Related Work

In this section we discussed some previous work related to code smells specification and analysis. In particular, we focus on papers that presented anti-patterns specification, code clones detection, and fault proneness analysis.

A. Smells Co-occurrence

There has been some efforts to verify if a class can present the symptoms of different kind of smells like clones and anti-patterns. For example, in a previous work [**?**], we showed evidence found of five possible relations (Plain Support, Mutual Support, Rejection, Common Refactoring, and Inclusion) among four bad smells (God Class, Long Method, Feature Envy, and Type Checking). Our work in this paper is different as we focus on the co-occurrence between two kind of smells: clones and anti-patterns.

Palomba *et al.* [**?**] conducted an empirical investigation involving 13 code smell types and 395 releases of 30 software systems. The results of the study highlighted six pairs of code smells that frequently co-occur together. For example, the authors reported that they found an unexpected relationship between smells such as between Message Chains and Refused Bequest.

Garg *et al.* [**?**] studied the co-occurrences of 7 code smell types in Chromium and Mozilla. The authors reported the percentage of smells co-occurring over the change history of such projects, and showed that the co-occurrence of some bad smells are more commun such as the co-existance of Data Clumps and Code Duplication. In the same context, Fontana *et al.* [**?**] reported an emprical analysis of code smells co-occurrence in a set of 111 open source systems. The authors found that Classes in the analysed software systems attract many smells. In particular, God Classes can be related to many other smells such as Data Class and Brain Method.

In another previous work [**?**], we detected evidences of smell mutations. Indeed, we modeled the evolution of code smells using Markov chains to show the phenomena of anti-patterns mutation. Then, we evaluated the fault-proneness of classes that participated in mutated and non-mutated anti-patterns. In fact, we reported that the principal cause of this phenomena is the structural changes performed by developers during the maintenance phase, and that specific mutations (such as from LargeClass to MessageChain) are very unsafe regarding the fault-proneness of the participated classes.

Our study is complementary to these previous works since it considers a quantitative and qualitative analysis of the co-occurrence of clones and anti-patterns and to investigate the relation of such co-occurrences with the quality of software systems.

B. Smells impact on Code Quality

One of the first published study that pointed out the possible impact of smells on cede quality was published by Chidamber and Kemerer [**?**]. Indeed, the authors proposed a set of object-oriented design metrics that has been used by various subsequent studies [**?**], [**?**]. The main lesson learned from the results of these studies is the following: the more complex the code is, the more risk of faults is notified.

Khomh *et al.* [**?**] analyzed the participation of classes in anti-patterns in object-oriented software systems with their stability (the state of remaining unchanged) and correctness (the state of being free from fault). Indeed, the authors showed evidences that such classes are more change and fault-prone than the rest of classes in the analyzed software systems.

In our previous work [**?**], we reported the results of an empirical study that evaluate the impact of dependencies between anti-patterns and the rest of classes on fault-proneness. That study presented evidences that classes with static relationships among anti-patterns are more fault-prone. More ever, we noticed that classes co-changed with anti-patterns during the maintenance of software systems had a higher risk of faults.

Several papers explored the relation between clones and faults. In particular, Aversano et al. found that several late propagations were linked to fault-fixes [**?**]. This specific pattern of clone evolution may be the result of inconsistent changes of a clone pair [**?**]. Juergens et al. showed that from 3 to 23% of the unintentionally inconsistent changes on clones were related to faults [**?**]. Indeed, if a code fragment contains a fault and that fragment is cloned in different places, the same fault appear in all the duplicated instances. Thus, code cloning can increase the probability of fault propagation [**?**] [**?**].

However, other previous work did not exclude that, in some contexts, clones can be a practical way to design and implement software features. For example, Kapser and Godfrey [**?**] introduced eight cloning patterns that they noticed their presence in software systems. These patterns present good motivations for cloning, like Templating, which occurs when the design of the new software feature is already known and a previous solution could be used to implement it [**?**].

Indeed, previous studies [**?**], [**?**] investigated independently the impact of clones and of anti-patterns on software quality. The purpose of this paper is to analyze the impact of co-occurrence of clones and anti-patterns on fault-proneness and to investigate their impacts on software quality. Concretely, we study in this paper whether the existence of anti-pattern and clones in the same classes makes the code more difficult to maintain.

III. Methodology

In this section we present the components of our study. As shown in Figure **??**, we detect a set of anti-patterns in the source code of the analysed software systems using DECOR [**?**]. Then, we use CCFinder [**?**] to detect duplicated code describing clones in the software systems. Finally, we evaluate the fault-proneness of different set of classes using a heuristics

presented in previous work by Sliwersky *et al.*. [**?**] Thus, we can address the following research questions:

- **RQ1:** *What is the percentage of classes participating in anti-patterns and clones?*
- **RQ2:** *What is the impact on fault-proneness for a class that participates in anti-patterns and clones?*

A. Detecting Anti-patterns

To detect the set of classes involved in anti-patterns automatically, we use DECOR tool based on PADL [**?**] (the Pattern and Abstract-level Description Language meta-model) and POM framework[**?**] (Primitives, Operators, Metrics) developed and used in several previous empirical studies to analyse anti-pattern occurrences and impacts [**?**][**?**]. PADL is a meta-model to describe the elements of object-oriented software systems and the relationships between them [**?**]. POM is a PADL-based framework used to spot anti-patterns by implementing and analysing more than 60 metrics.

We focus on the 15 anti-patterns documented by Brown *et al.* [**?**]. These anti-patterns show recurring problem in the implementation and the maintenance of software code. These anti-patterns were evaluated in a previous work [**?**] and hence we could validate our results using this previous data.

In this study, we include the following list of anti-patterns:

- AntiSingleton: a class participates in this anti-pattern if it involves a mutable variables that may be considered as global variables. This anti-pattern makes the code inflexible and very difficult to test code as it introduces a dependency through a side-channel that is not explicitly given as a parameter to constructors or other functions that use it.
- BaseClassShouldBeAbstract: a class that has many subclasses without being abstract. This anti-pattern makes the design more complicated because it does not promote the creation of a hierarchical relationship via inheritance. Thus, it increase the maintenance cost of software systems.
- Blob: this anti-pattern is detected in a source code of a software system where one class monopolizes the processing, and a set of related classes save the data. Thus, features are implemented in just one class. If class is involved in a Blob, it could include a lot of unnecessary code. This makes the maintenance activities performed by developers harder as they cannot discriminate the useful code from the not useful code.
- ClassDataShouldBePrivate: a class participates in this anti-pattern if its fields are public and by consequence violates the principle of encapsulation in object-oriented paradigm. This anti-patterns is the cause of confusion and lack of understanding of the source code during the maintenance phase. This increases the maintenance cost and the chance of introducing errors.
- ComplexClass: if a class includes at least one complex method, it is involved in this anti-pattern. We evaluate the complexity of a method using the Cyclomatic complexity. The Cyclomatic complexity is a software metric of the number of linearly independent paths inside the method. This anti-pattern make the testing activities harder as it implies a high number of test cases to achieve adequately the test coverage of the complex method.
- LargeClass: if a class includes at least one large method, it is involved in a LargeClass anti-pattern. The large method is evaluated using the number of line of code (the LOC metric). Indeed, this metric gives developers an idea about the size of a method and thus they can predict the amount of effort during the maintenance phase to understand and update the code. This anti-pattern increase the effort to develop, maintain and test a software system as previous studies have showed that effort is highly correlated with LOC. Indeed, a class with larger LOC values take more time to develop and to maintain.
- LazyClass: if a class contains a limited number of attributes and methods, then it is considered as a LazyClass. This anti-pattern increase the Cyclomatic complexity of the software system as it only delegates its requests to other more complicated classes.
- LongMethod: this anti-pattern specifies the case of a class when it contains just one large method regarding the number of line of code. This anti-pattern make the maintenance and the testing phase of software system more difficult as a set of shorter methods will help the developers in understanding and updating the class.
- LongParameterList: this anti-pattern specifies the case of a class when it includes at least one method with a high number of parameters regarding the average number of parameters per methods in the rest of classes in the software system. This makes the maintenance of the affected class more complicated, as more parameters a method has, the more complex it is.
- ManyFieldAttributesButNotComplex: if a class contains many attributes but a very limited number of line of code, it is considered as ManyFieldAttributesButNotComplex anti-pattern as it do not contain functionalities to manage independently its data.
- MessageChain: this anti-pattern specifies the case when class needs a long chain of method invocations to implement one functionality. This anti-pattern makes the software system more memory and resource consuming as it implies watching out for high number of sequences of method calls to use a data or implement a simple feature.
- RefusedParentRequest: this anti-pattern specifies a scenario of polymorphism breaking when a class redefines an inherited method using empty bodies. This anti-pattern reduce code clarity and organization as the related superclass and subclass are completely different.
- SpaghettiCode: if a class contains or implies a procedural structure (instead of an object-oriented structure), it is

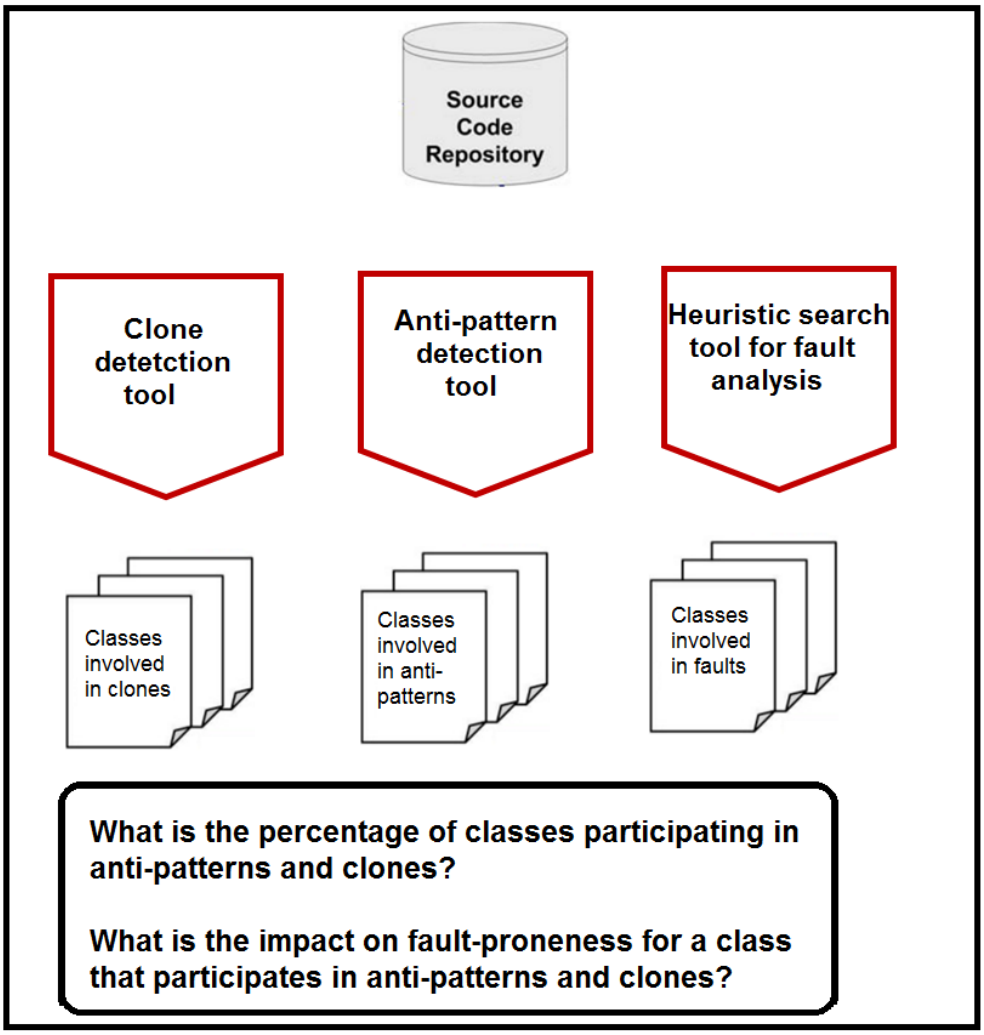

Fig. 1. Overview of our study to analyze the co-occurrence of clones and anti-patterns

involved in a SpaghettiCode anti-pattern. The existence of such class is the cause of the existence of limited number of classes with large methods and that icludes a single, multistage process flow. SpaghettiCode makes the software system very difficult to update or to reuse, and thus, increases the maintenance cost.

- SpeculativeGenerality: if a class is defined as an abstract class and contains a very few children, some of its methods may not be used, and thus, it is considered as an anti-pattern. This anti-pattern adds complexity, extra code to maintain and test, and make the code less readable.
- SwissArmyKnife: this anti-pattern specifies the scenario of a class that contains some methods divided into disjunct sets of many methods. Indeed, this scenario includes the use of a high number of unrelated functionalities. A SwissArmyKnife is problematic as it provides a complicated interface difficult to understand and to maintain.

B. Clone identification

We use CCFinder [?] to detect the clones in the source code of software systems. CCFinder is a token based clone detection tool which used suffix tree matching algorithm to detect clones. This tool extracts tokens from the source code using lexical analysis. CCFinder has been widely used for clone detection in numerous previous work [?] [?]. In Addition, CCFinder presented the best trade-off between precision and recall in accordance with other tools and techniques [?].

We choose CCFinder as it has a better performance, scalability, and recall at the expense of a lower precision [?], [?]. Although this means that more false positives are identified, CCFinder does not require the code to be compilable and does not depend on the syntax of the language [?].

The study reported in this paper is done by analusing the Type 1 clones specified in the literature [?] [?]. In fact, if two or more entities in a software system are exactly the same, these entities are considered as exact clones or Type 1 clones. Previous work [?], [?], [?], and [?], pointed out that Type 1

clones cloud increase maintenance effort and bug introduction risks in case of change inconsistency among clones.

C. Analyzing the Fault Proneness

We evaluate the fault-proneness by analysing the change report and fault report included in software repertories. We mine the project's version-control systems (CVS[1] and SVN[2]) to identify changes committed for each class to fix faults. Indeed, we analyse the commit message co-occured with a maintenance activities to detect bug-fixing commits. For this, we use a Perl script used successfully in previous work [**?**], implements the heuristics by Sliwersky *et al.* [**?**]. Those heuristic search for commit message that contain some words such as "bug" and "fault" and performed during the maintenance phase of the studied release of a system (in the period between the publication date of the studied release and the publication date of the next release).

The bug identifier found in the commit log message is then compared with the project's list of known bugs to determine the list of files (and classes) that were changed to fix a bug. Finally, our script check the locations of changes performed on this list of files with CCFinder and DECOR results to keep files in which a code smell and a fault occurred in the same location of the file.

We perform the fault proneness analysis for the period during the maintenance phase of each studied release, starting from the publication date of the studied release and the publication date of the next release.

IV. Study Design

In this section we explain the approach used in our study and the analyzed software systems.

The design of our study follows the Goal-Question-Metric (GQM) approach [**?**]: the *goal* of our study is to investigate the co-occurrence of anti-patterns and clones. The *quality focus* is the identification of the risk specific smells in term of fault-proneness. The *perspective* is that of researchers and practitioners who should be aware of the hidden co-occurrence of clones and anti-patterns to make informed maintenance activities. The *context* of this study is three java software systems, Azureus, Eclipse, and JHotDraw.

A. Analyzed Systems

We use several criteria to select five datasets. We selected open-source systems because the availability of data repositories and that other researchers can replicate our study. Second, we choose different sizes of systems from different domains. Third, we selected datasets with change log history.

Azureus (renamed Vuze)[3] is is a BitTorrent client written in Java to transfer files via the BitTorrent protocol[4]. Azureus (version 2.5.0.0) dataset contains 2,296 and 4,004 Java files and classes respectively.

[1]http://cvs.nongnu.org/
[2]http://subversion.apache.org/
[3]http://www.vuze.com/
[4]http://www.bittorrent.org/

Eclipse[5] is an integrated development environment written mostly in Java and used to develop applications. Eclipse (version 2.0) dataset contains 6,751 and 10,156 Java files and classes respectively

JHotDraw[6] is a two-dimensional graphics framework for structured drawing editors that is written in Java. This software system is used to create several editors and it provides functionalities for saving, loading, and drawing. JHotDraw (version 5.4b1) dataset contains 727 and 826 Java files and classes respectively.

V. Study Results

Table **??** summarizes the data obtained by analyzing Azureus, Eclipse, and JHotDraw. We validated a randomly selected set of anti-pattern occurrences and clones manually. In the following we will answer our research questions and discuss some examples from the results of the empirical study.

A. What is the percentage of classes participating in anti-patterns and clones?

1) Motivation: A better evaluation of the impact of smells is essentiel to better focus on maintenance activities. Since anti-patterns can be considered as design-level smells while code clones are code-level smells, being aware of the co-occurrence of these smells can help software developers to identify the most important files for maintenance activities such as refactoring and tracking. Furthermore, not all smells are equally risky [**?**] as different smells have different "bad" properties and thus have different probabilities to lead faults in a software system.

In this research question, we are detection the cases of co-occurrences of anti-patterns and clones. Our purpose is to help developers to identify the entities that share numerous kind of smells in order to prioritize reviews and testings activities towards the most most risky parts of code. We report here the first study that report the distribution of cloned files among anti-pattern. We will discuss in the next section some lessons learned from this distribution.

2) Approach: First, we detect the set of anti-pattern occurrences in the analyzed systems using DECOR. Second, we identify clones in the source code using CCFinder. Then, we measure the frequency with which anti-patterns and clones occur and co-occur in the studied systems. Specifically, we analyse for each system the overall number of classes, the number of classes participating in anti-patterns the number of classes having clones, and the number of classes participating both in anti-patterns and clones. Then, we report the percentages of the different kind of anti-patterns that share clones.

[5]http://www.eclipse.org/
[6]http://www.jhotdraw.org/

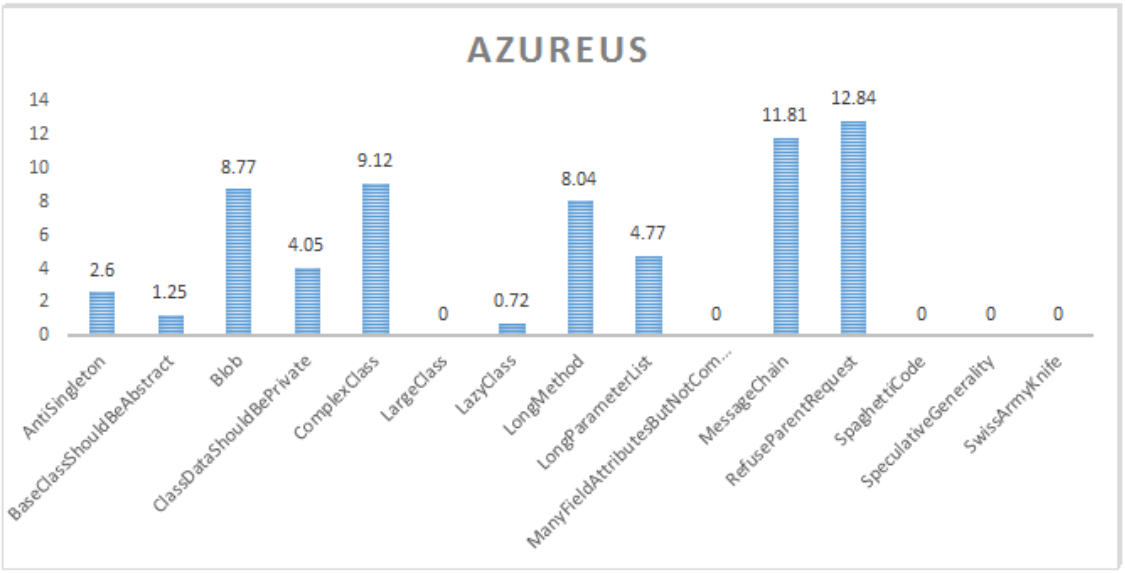

Fig. 2. Distribution of the percentage of co-occurrence of anti-pattern and clones in Azureus

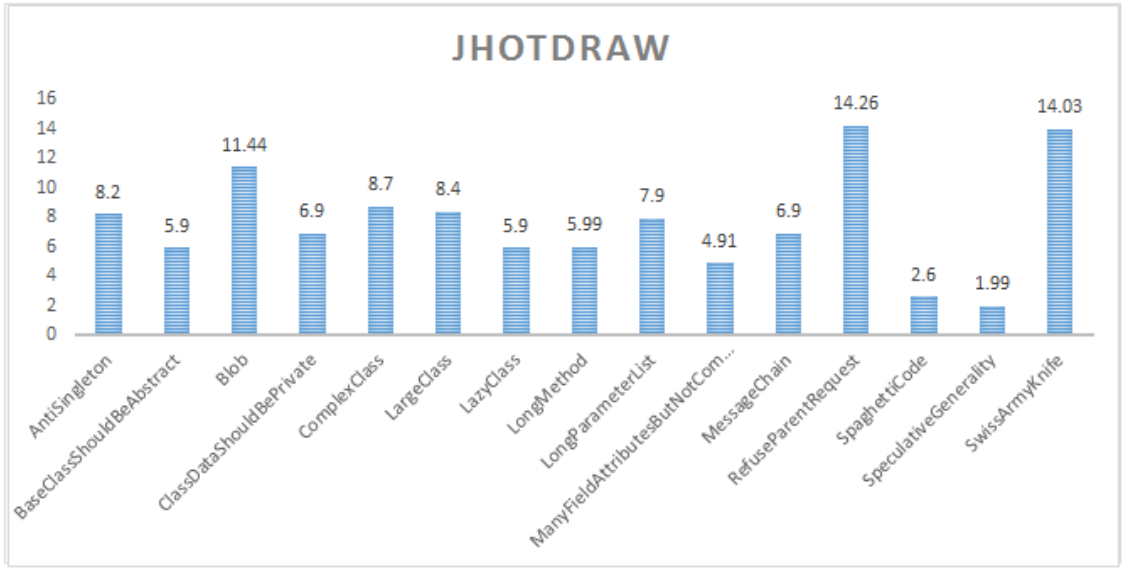

Fig. 3. Distribution of the percentage of co-occurrence of anti-pattern and clones in JHotDraw

3) Results: Table **??**, Figure **??**, Figure **??**, and Figure **??**, summarize the **RQ1** results. We observed in the three analyzed systems that between 52% and 99% of anti-pattern classes contain clones, while between 59% and 78% of clones are participating in anti-pattern. For example, in Azureus, we detect 4,004 different classes, 1306 of them are participating in one or more of the 15 anti-patterns considered in this study. For clone analysis, we found that 2194 classes in Azureus are involved in clones type 1, and 1304 of them are ivolved in anti-patterns and clones. Indeed, having clones and anti-patterns is a frequent observation in the analyzed systems.

Figures **??**, Figure **??**, Figure **??**, and Table **??** report for each anti-pattern and studied system the percentage of the anti-pattern's occurrences that share clones. For example, more than 12% of anti-patterns had clones in Azureus are RefusedParentBequest. We notice that the anti-patterns that are large by default, *i.e.,* Blob, LongMethod, LongParameterList, and LargeClass do not have a much larger percentage of internal clones. Contrary to intuition, anti-patterns resulting in large classes are not the most affected by clones. Future work will include the examination of a potential statistical relationship between code clone and anti-pattern's sizes.

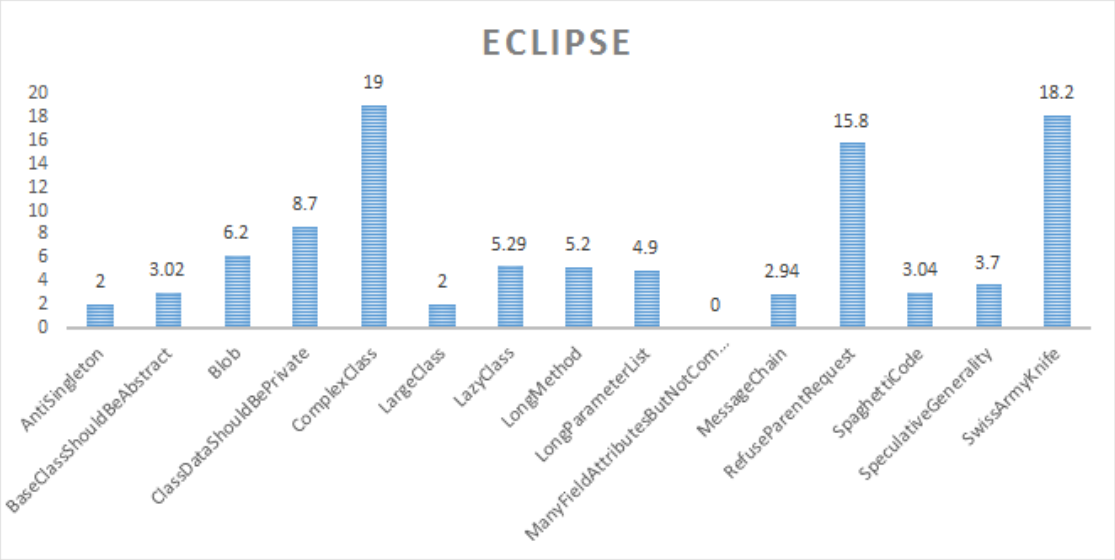

Fig. 4. Distribution of the percentage of co-occurrence of anti-pattern and clones in Eclipse

TABLE I
DESCRIPTIVE STATISTICS OF THE OBJECT SYSTEMS (CAC: PRESENTING CO-OCCURRENCE OF ANTI-PATTERN AND CLONE)

System	Release	# files	# line of codes	# classes	# anti-pattern classes	# clone classes	# classes CAC
Azureus	2.5.0.0	2,296	432,799	4,004	1306	2194	1304
Eclipse	2.0	6,751	1,355,899	10,156	6050	6449	4370
JHotDraw	5.4b1	727	70,398	826	691	466	364

Having clones and anti-patterns is a very frequent observation: at least, more than 52% of anti-pattern classes have clones, while 59% to 78% of classes with clones are participating in anti-patterns.

B. What is the impact on fault-proneness for a class that participates in anti-patterns and clones?

1) Motivation: To the best of our knowledge, we present in this paper the first study to investigate whether co-occurrence of anti-patterns and clones correlates with fault proneness. Our purpose is to evaluate the fault proneness of a class if it belongs to an anti-pattern, it contains a cloned code, or it presents co-occurrence of anti-patterns and cloned code, and to compare these fault-proneness of the rest of files in the three analysed software systems.

We report in the following a quantitative analysis to investigate whether the co-occurrence of bad code smells elevates the chance that the corresponding segment of code is faulty. While a lot of attention has been paid to the study of code clones [**?**][**?**], and also, independently, anti-patterns [**?**], this study goes a step further by investigating the likelihood and implications of co-occurrence of these smells for fault-proneness. Indeed, while an individual codes smell might have negative impact on the quality and the correctness of a software system, the co-occurrence of two different kinds of smell, such as anti-patterns on the one hand, and code cloning on the other hand, could be a sign that there is a greater risk of problems.

2) Approach: For each system, we identify whether a class undergoes faults during the developing phase of the studied systems as described in Section **??**.

Then, we use Chi-squared's test [**?**] to check whether the difference in fault-proneness between classes sharing the considered anti-patterns and clones and other classes is significant. Chi-squared's test is a nonparametric statistical test to evaluate the significance of the association (contingency) between two set of data. Indeed, an inferential test such the Chi-squared's test is used in order to understand whether the overall population medians are significantly different, based on a sample median. In this case, the considered population is the set of classes of the considered systems. The considered sample used in the Chi-squared's test is the set of classes of the specific analyzed version. We test the following null hypothesis:

- H_{RQ2_0}: The proportions of faults involving classes having a co-occurrence of anti-pattern and code clones, and the rest of classes are the same. If we reject the null hypothesis H_{RQ2_0}, the proportion of faults carried by

TABLE II
FAULT PRONENESS AND CHI-SQUARED'S TEST RESULTS IN AZUREUS, JHOTDRAW, ECLIPSE, LUCENE AND XALANJ (CAC: PRESENTING CO-OCCURRENCE OF ANTI-PATTERN AND CLONE)

	Faulty	Clean
Number of classes CAC in Azureus	1042	262
Number of the rest of classes in Azureus	356	636
Number of anti-patterns in Azureus	604	702
Number of clones classes in Azureus	512	1682
Chi-squared's test for Azureus	2.2e-16	
Odds-ratio for Azureus	7.1052	
Number of classes CAC in JHotDraw	282	82
Number of the rest of classes in JHotDraw	120	143
Number of anti-patterns in JHotDraw	185	506
Number of clones in JHotDraw	166	300
Chi-squared's test for JHotDraw	2.2e-16	
Odds-ratio for JHotDraw	4.0982	
Number of classes CAC in Eclipse	1876	2494
Number of the rest of classes in Eclipse	650	6136
Number of anti-patterns in Eclipse	929	5121
Number of clones in Eclipse	726	5723
Chi-squared's test for Eclipse	2.2e-16	
Odds-ratio for Eclipse	7.1008	

classes having a co-occurrence of anti-pattern and code clones is not the same as that of other classes.

Then, we calculate the odds ratio [**?**] to examine the likelihood for a fault to occur for classes having a co-occurrence of anti-pattern and code clones. The odds ratio is defined as the ratio of the odds p that classes having anti-pattern and code clones experience a fault in the future (experimental group), to the odds q of the same event occurring in the other sample, *i.e.*, the odds that other classes classes experience faults (control group): $OR = \frac{p/(1-p)}{q/(1-q)}$.

The analysis done to answer this research question was accomplished with the R statistical tool, and we chose a confidence interval of 95 percent.

3) Results: Table **??** presents fault proneness analysis for Azureus, JHotDraw, and Eclipse. Indeed, the result of Chi-squared's test and odds ratios when testing H_{RQ2_0} are significant for the three analyzed systems. The p-value is less than 0.05 and the odds ratio for fault-prone CAC classes (classes presenting co-occurrence of anti-pattern and clone) is higher than for fault-prone other classes. In addition, we observe that this risk is higher than the cases when classes is involved just an anti-pattern or in a cloned code.

For example, in Eclipse, we observe that the risk for a class to be fault-prone if it has a co-occurrence of clone and anti-pattern is at least 7 times higher than a class that hasn't neither a co-occurrence of clone and anti-pattern. Moreover, this risk is higher than a class that has just an occurrence of an anti-pattern and higher than a class that has just an occurrence of a clone. Indeed, we can answer to **RQ2** as follows: we showed that classes having clones and anti-patterns are significantly more fault-prone than other classes.

Then, we examine the fault-proneness of all kind of classes in the analyzed systems. For example, in Azureus, we observe that 46% of anti-pattern classes are faulty, 23% of clone classes are faulty, 80% of classes having anti-patterns and clones are faulty, while just 36% of the rest of classes are faulty. In JHotDraw and Eclipse, we observe similar facts: classes having anti-patterns or clones are more faulty than other classes, and the risk of fault proneness increases if the class has clones and participates in anti-patterns in the same time. In the next section, we will discuss in more detail the relation of these results with the reliability of software systems.

Classes having clones and anti-patterns are significantly more fault-prone than other classes.

VI. DISCUSSION

In this section we are discussing some observations from our empirical study and its threats to validity.

A. Differences in Smells Distribution

We notice that different software systems can have different proportions of smell co-occurrences. This finding can be the consequence of the fact that these systems were created in different contexts to resolve different problems and to implement divergent requirements. Results show that the proportion of classes participating to anti-patterns, having clones, and both, has not relationship with the systems sizes. For example, in Eclipse, the largest considered software system in this study, we detected the smallest proportion of smell co-occurrence (about 23% of the totality of classes as shown is Table **??**). Several of the studied anti-patterns are not detected This was the case for ManyFieldAttributesButNotComplex anti-pattern. Clones are detected in all considered systems with different proportion. Indeed, clones represent more than 50% of Azureus, Eclipse, and JHotDraw. This confirms the observation of Baker [**?**] and Ducasse *et al.* [**?**] which observed that clones exist at rates of over 50% of the effective lines of code (ELOC) in *COBOL* and *C* systems.

B. Smell Co-occurrence and the Reliability of Software Systems

Software Reliability is an important attribute of software quality. It is defined as the probability of failure-free operation for a specified period of time in a specified environment [**?**]. A failure can occur if the observable behavior of a software system is different from its expected behavior, while a fault is a static software characteristic which causes a failure to occur. Thus, in order to evaluate the reliability of systems, many previous work used the number of faults per line as a reliability measure [**?**], [**?**]. We showed in this paper that, in the three considered software systems,classes with co-occurrence of cloned code and anti-pattern are more fault-proneness, and thus, less reliable, than other classes.

By revising Figure **??**, Figure **??**, and Figure **??**, and analysing the set of faulty classes, we noticed the high

proportion of co-occurrence of cloned code with one specific anti-patterns: RefusedParentRequest. Indeed, in the case of the RefusedParentRequest anti-pattern, developers were motivated to create inheritance between classes only by the desire to reuse the code in a superclass. But the superclass and subclass are completely different. That incompatibility between the cloned code of the superclass using for reuse and the semantic of the subclass increase the fault-proneness of the two classes and thus decrease their reliability.

C. Smell Co-occurrence and the Quality of Software Systems

The most involved anti-patterns in cloned code include the following anti-patterns:

- RefusedParentRequest.
- Blob.
- ComplexClass.

Two common points between these anti-patterns: first, they are the result of a misunderstanding scenario, as the example of duplicating code when developers do not fully understand the problem (incompatible inheritance for the RefusedParentRequest anti-pattern, unchecked unused or dead code for Blob and ComplexClass anti-pattern). Second, they describe the inconsideration of the major design principles of Object-Oriented Programming, such as the high cohesion and low coupling. For example, the RefusedParentRequest anti-pattern occurrences show a low cohesion as they implement a great variety of inherited actions and are not focused on what they should do. As another example, the Blob occurrences introduce a high coupling as Blob contains the majority of the implemented features, and the other related classes contain only the data, which would make the code difficult to make changes as well as to maintain it.

In comparison with Kapser and Godfrey study [**?**], our study examine the co-occurrence of cloned code with bad smells detected in three software systems which may increase the the maintainability cost of the software system, and thus, reduce their qualities. Our previous studies analyzed independently the impact of anti-patterns [**?**] and clones [**?**] on change propagation on fault-proneness. We thus reported that the detection and the analysis of anti-patterns and clones allow us to detect the critical elements of software systems since they may represent the set of more risky files in terms of fault-proneness. Other previous work showed that clones require consistent changes [**?**] [**?**]. A consistent change is a change performed at the same time to update cloned code with the same instructions. We also observed in [**?**] the existence of change dependencies between anti-patterns and other artefacts in software systems. A misunderstanding of such dependencies may increase the risk of faults. Thus, by spotting the sets of co-occurrences of anti-pattern and clones, we detect the more risky part of code in term of consistent change.

D. Threats to Validity

To discuss the threats to the validity of our empirical study, we follow the guidelines provided in [**?**].

Construct validity threats concern the relation between theory and observation. Indeed, the detection tools used to identfy clones and anti-patterns involve a set of subjective specification of smells. For this reason, the precision of DECOR and CCfinder is a concern that we agree to accept. In case of other smell specifications that are differ with the specification used by CCFinder and DECOR, some clone and anti-pattern occurrences may be missed during the detection phase. The same problem could be the result of choosing the default parameters of CCFinder [**?**]. We try to address the confounding configuration choice problem for clone detection by choosing a configuration that has been widely used for clone detection in previous work [**?**] [**?**]. Although the set of data reported in this study is large enough to claim our conclusions, more analysis using other detection tools could be considered in future work. *Internal Validity* Internal validity is the validity of causal inferences in studies based on experiments. In this paper, we are not claiming causation but we are reporting a raise of the number of detected fault in the analysed software systems that co-occurs with the involvement of classes in anti-pattern and clone. As fault occurrence is a temporary property, and anti-pattern or clone symptoms could be a long-term property, we analysed changes and faults that occurred during the development phase of the studied systems and we limit our analysis on the co-occurrence of faults and smell co-occurrences without claiming causation.

Reliability validity threats concern the possibility of replicating this study. We presented in this paper all the details to replicate our study. In addition, Azureus, Eclipse, and JHotDraw source code repositories are publicly published, as well as the two used tools DECOR and CCfinder. The specification and the implementation of the fault analysis heuristic iare also publicly published.

Conclusion validity threats concern the relation between the treatment and the outcome. In this empirical study we used the Chi-squared's test and we paid attention to follow and respect the assumptions of this non-parametric test.

Threats to *external validity* concern the possibility to generalize our observations. We accept that we could have different results if we analysed different object-oriented software systems. In our empirical study, we chose a set of systems with different features, size, and areas in order to reduce the threat to the external validity. However, we agree that more studies, preferably on industrial datasets and others programming languages, are required to generalize the results reported in this paper.

VII. Conclusion

We reported an empirical study investigating the co-occurrence(in Azureus, Eclipse, and JHotDraw) of anti-patterns and clones and the relations of co-occurrences with class fault proneness. Study results show that the percentages of classes involved in co-occurrences of anti-patterns and clones range between 63% and 32%. We found also that the number of classes sharing clones, anti-patterns, and both increases/decreases consistently with the systems sizes. In

all systems, we found that class fault proneness odds ratios significantly increase for classes that had co-occurrence of anti-patterns and clones in comparison with the rest of classes in the studied systems.

Our research schedule includes several extensions of this work. Among others: (1) the analysis of the co-evolution and co-change of classes participating in anti-patterns and sharing clones, and (2) the replication of the study with other specifications of anti-patterns and cloned code.

ACKNOWLEDGMENT

We thank the Wallonie-Bruxelles International (WBI) in Belgium and the international internships of the Fonds québécois de la recherche sur la nature et les technologies (FRQNT) for supporting this research work.