Do they Really Smell Bad? A Study on Developers' Perception of Code Bad Smells

Fabio Palomba[1], Gabriele Bavota[2], Massimiliano Di Penta[2], Rocco Oliveto[3], Andrea De Lucia[1]
[1]University of Salerno, Italy [2]University of Sannio, Italy [3]University of Molise, Italy

***Abstract*—In the last decade several catalogues have been defined to characterize code bad smells, i.e., symptoms of poor design and implementation choices. On top of such catalogues, researchers have defined methods and tools to automatically detect and/or remove bad smells. Nevertheless, there is an ongoing debate regarding the extent to which developers perceive bad smells as serious design problems. Indeed, there seems to be a gap between theory and practice, i.e., what is believed to be a problem (theory) and what is actually a problem (practice).**

This paper presents a study aimed at providing empirical evidence on how developers perceive bad smells. In this study, we showed to developers code entities—belonging to three systems—affected and not by bad smells, and we asked them to indicate whether the code contains a potential design problem, and if any, the nature and severity of the problem. The study involved both original developers from the three projects and outsiders, namely industrial developers and Master students. The results provide insights on characteristics of bad smells not yet explored sufficiently. Also, our findings could guide future research on approaches for the detection and removal of bad smells.

I. Introduction

Code bad smells represent symptoms of poor design and implementation choices [1]. Bad smells are usually introduced in software systems because developers poorly conceived the design of the code component or because they did not care about properly designing the solution due to strict deadlines. *Complex Class*, i.e., a class that contain complex methods and it is very large in terms of LOC; or *God Class*, i.e., a class that does too much/knows too much about other classes, are only some examples of a paramount of bad smells identified and characterized in well-known catalogues [2], [1].

Recent empirical studies showed that code smells hinder comprehensibility [3], and possibly increase change- and fault-proneness [4]. Also, the interaction between different, co-existing code smells can negatively affect maintainability [5]. Hence, there is empirical evidence that code smells have a negative effect on software evolution, and therefore should be carefully monitored and possibly removed through refactoring operations. Thus, a lot of effort has been devoted for the definition of approaches aiming at detecting and removing code bad smells [6], [7], [8], [9].

Despite the existing evidence about the negative effects of code smells [3], [4], [5] and the effort devoted to the definition of approaches for detecting and removing them, it is still unclear whether developers would actually consider all bad smells as actual symptoms of wrong design/implementation choices, or whether some of them are simply a manifestation of the intrinsic complexity of the designed solution. In other words, there seem to be a gap between the theory and the practice. For example, a recent study found that some source code files of the Linux Kernel intrinsically have high cyclomatic complexity. However, this is not considered a design or implementation problem by developers [10]. Also, empirical studies indicated that (i) God Classes sporadically changing are not felt as a problem by developers [11]; and (ii) some developers, in particular junior programmers, work better on a version of a system having some classes that centralized the control, i.e., God classes [12]. These results suggest that the presence of bad smells in source code is sometimes tolerable, and part of developers' design choices.

Recently, Yamashita and Moonen [13] performed an exploratory survey aimed at investigating developers knowledge about code smells, by asking questions like "How familiar are you with code bad smells?". Results showed that a large proportion of respondents did not know about code bad smells. While the study of Yamashita and Moonen aimed at investigating to what extent developers had a theoretical knowledge of code smells (i.e., knowing them from their name and definition), no study so far investigated whether, given a problem instance—that can be brought back to the presence of a bad smell in the code—developers actually perceive the problem as such and whether they associate the problem to the same symptoms explained in the smell definition.

To bridge this gap, we conducted a study aimed at investigating the developers' perception of code smells. First, we identified and manually validated instances of 12 different bad smells in three open source projects. Then, we provided a questionnaire to the participants where we showed code snippets affected and not affected by bad smells, and asked whether, in the respondents' opinion, the code component has any problem. In case of a positive answer, we asked them to explain what kind of problem they perceived and how severe they judged it. We asked different categories of subjects to participate in the study, namely (i) Master's students, representing a population of subjects pretty knowledgeable about the theoretical concepts of code smells, (ii) industrial developers, i.e., people with experience on real development projects, but not knowing the code being shown; and (iii) developers from the open-source projects in which the bad smells have been collected. In total, we received responses from 34 subjects, and specifically 15 Master's students, 9 industrial developers, and 10 original developers of the studied projects. The data used in our study are publicly available as replication package[1].

[1]http://tinyurl.com/o6lk584

TABLE I
BAD SMELLS ANALYZED IN OUR STUDY [2] [1].

Name	Description
Class Data Should Be Private (CDSBP)	A class exposing its attributes
Complex Class (CC)	Classes having high complexity
Feature Envy (FE)	A method making too many calls to methods of another class to obtain data and/or functionality
God Class (GC)	A class having huge dimension and implementing different responsibilities
Inappropriate Intimacy (II)	Two classes exhibiting high coupling between them
Lazy Class (LC)	A very small class that does not do too much in the system
Long Method (LM)	A method having huge size
Long Parameter List (LPL)	A method having a long list of parameters
Middle Man (MM)	A class delegating all its work to other classes
Refused Bequest (RB)	A class inheriting functionalities that it never uses
Spaghetti Code (SC)	A class without structure that declare long methods without parameters
Speculative Generality (SG)	An abstract class that is not actually needed, as it is not specialized by any other class

TABLE II
CHARACTERISTICS OF THE OBJECT SYSTEMS.

Project	KLOC	#Classes	#Methods
ArgoUML 0.34	280	1,889	10,450
Eclipse 3.6.1	440	2,181	18,234
jEdit 4.5.1	165	520	5411

II. DESIGN OF THE EMPIRICAL STUDY

The *goal* of the study is to investigate to what extent bad smells reflect developers' perception of poor design and implementation choices and, in this case, what is their perceived severity of the problem. The *quality focus* is source code comprehensibility and maintainability that can be hindered by the presence of bad smells. The *context* of the study consists of: (i) *objects*, i.e., bad smells identified in three software projects; and (ii) *subjects* (hereby referred to as "participants"), i.e., Master's students and professional developers providing their opinions about bad smells.

A. Research Questions

Our study aims at addressing the following two research questions:

- **RQ$_1$:** *To what extent do bad smells reflect developers' perception of design problems?*
- **RQ$_2$:** *What are the bad smells that developers feel as the most harmful?*

In the context of our study, we considered the twelve code smells briefly described in Table I. Our choice of these smells is not random, but guided by the will of considering a mix of bad smells related to complex/large code components (e.g., *Complex Class*, *God Class*) as well as smells related to the non-adoption of good Object-Oriented coding practices (e.g., *Inappropriate Intimacy*, *Refused Bequest*). However, we did not consider smells such as *Divergent Change* or *Parallel Inheritance*, because their full understanding would require a deep knowledge and/or exploration of the system history.

B. Context Selection

The *objects* considered in the study are bad smells identified in three open-source projects, namely ArgoUML, Eclipse, and JEdit. ArgoUMLis an open-source UML modeling tool while Eclipseis a popular Integrated Development Environment supporting different programming languages. Finally, JEditis a text editor for programmers.Table II reports the characteristics of the analyzed projects, namely the size in terms of KLOC, number (#) of classes and number of methods.

TABLE IV
STUDY PARTICIPANTS: INVITED AND ACTUAL RESPONDENTS.

Category	Invited	Answered	Return rate (%)
Original Developers	45	10	22%
Industrial Developers	28	9	32%
Master's students	15	15	100%
OVERALL	88	34	39%

To answer our research questions we needed to identify instances of the twelve considered bad smells in the object systems. Unfortunately, since there are no annotated sets of such smells available in literature, we had to manually identify them. A Master's student from the University of Salerno manually identified instances of the considered bad smells in each of the object systems by relying on the definition of the smells reported in the literature. In such a process, the student also relied on metric extractors and on metric-based definitions of code smells, such as the one of *DECOR* [7]. For example, *God Classes* were identified as large classes implementing several responsibilities and controlling many other objects in the system, while *Long Methods* were simply identified by analyzing the lines of code composing them. The resulting set of smells has been then validated by a second Master's student to verify that all affected code components identified by the first student were correct. Finally, two of the authors reviewed the identified instances to double check the results of the students' analysis. Note that, while this does not ensure completeness in the identification of smells, having multiple manual evaluations ensure enough confidence about the absence of false positives, that could instead occur if relying on automatic detection tools. Also, such a multiple evaluation limited the bias in our dataset. For the aim of our study this was exactly what we needed: a set of reliable code bad smells on the object systems. Note that, we did not find instances of all considered smells in each object system. Table III reports, for each code smell, the number of its instances identified in the object systems.

As summarized in Table IV, *participants* involved in the study belong to the following three categories:

1) *Developers working on the three open-source systems*. We sent invitations to active developers of the three object systems, identified by analyzing the systems' commit history[2]. In total, we invited 19 developers from Ar-

[2]We considered developers that performed at least one commit in the last two years.

TABLE III
BAD SMELLS INSTANCES IDENTIFIED IN EACH SYSTEM.

Project	CDSBP	CC	FE	GC	II	LC	LM	LPL	MM	RB	SC	SG
ArgoUML	5	4	1	3	4	0	28	0	2	4	15	28
Eclipse	32	35	6	15	7	15	180	0	2	31	24	12
jEdit	7	21	0	6	4	0	33	9	0	3	18	14

goUML, 11 from Eclipse, and 15 from jEdit. We received responses from 4, 3, and 3 developers, respectively. In the following, we will refer to them as *original developers*. Note that each of the original developers was asked to work on tasks related to the code belonging to the system she had worked on only.

2) *Industrial developers*. We invited 28 industrial developers to take part in our study. We obtained an answer from 9 of them; each one performed tasks related to all three systems.
3) *Master's students*. We recruited 15 Master's students attending the Advanced Software Engineering course at the University of Salerno (Italy). Students had good knowledge of Object Oriented programming and they attended a seminar of three hours about code bad smells and design problems. All students performed tasks related to all three systems.

The reason for having these different categories of participants is to get the opinion of developers who know the code very well, as well as of outsiders (industrial developers and Master's students) that, while being less knowledgeable about the code, might provide a less biased indication.

C. Study Procedure

The experimental tasks consisted of questionnaires that participants had to answer through a Web application tool named *eSurveyPro*. In these questionnaires we showed to the participants source code snippets (that may or may not contain code smells) and asked questions about whether the code contained possible design/implementation problems, as well as the perceived severity of the problem, if any.

Specifically, given the object system S_i, the following process was performed:

1) For each code smell c_j having at least one instance in S_i, we randomly selected one instance or took the only one available. Note that with "instance" we refer to the code component(s) affected by the smell. For example, it could be a single method affected by the *Long Parameter List* smell as well as a pair of classes affected by the *Inappropriate Intimacy* smell. Note that we did not select code components affected by more than one bad smell, since we want to isolate each smell involved in our study.
2) For each selected smell instance, we created a task composed of the following questions:
   - In your opinion, does this code component[3] exhibit any design and/or implementation problem?
   - If YES, please explain what are, in your opinion, the problems affecting the code component.
   - If YES, please rate the severity of the design and/or implementation problem by assigning a score on the following five-points Likert scale: 1 (very low), 2 (low), 3 (medium), 4 (high), 5 (very high).
3) For each task related to a code component affected by a bad smell, we also instantiated a task—requiring to participants the same answers seen above—concerning randomly selected code components not affected by any of the code smells considered in our study. This was done to limit the bias in the study, i.e., avoid that participants always indicated that the code contained a problem and the problem was a serious one.

The final questionnaires included 20 tasks for ArgoUML (of which 10 related to components affected by bad smells), 22 for Eclipse (11 affected by bad smells), and 18 for JEdit (9 affected by bad smells). As explained before, the difference in the number of tasks for the three systems is because as shown in Table III, we identified instances of 10 kinds of smells in ArgoUML , 11 in Eclipse and 9 in JEdit.

All participants invited in our study received an email with instructions on how to answer the survey and a link to the website where each participant could log in to visualize and answer the questions. Participants had up to four weeks to complete this survey.

D. Data Analysis

To answer $\mathbf{RQ}_1$ we compute, for each type of code smell:

1) The percentage of cases the bad smell has been *perceived* by the participants. With *perceived*, we mean cases where participants answered *yes* to the question: "In your opinion, does this code component exhibit any design and/or implementation problem?"
2) The percentage of times the bad smell has been *identified* by the participants. With *identified*, we mean cases where besides perceiving the smell, participants were also able to identify the exact smell affecting the analyzed code components, by describing it when answering to the question "If yes, please explain what are, in your opinion, the problems affecting the code component". Note that we consider a bad smell as *identified* only if the design problems described by the participant are clearly traceable onto the definition of the bad smell affecting the code component. For example, given the following bad smell description for the *Feature Envy* bad smell: *"a method making too many calls to methods of another class to obtain data and/or functionality"*, examples of "correct" descriptions of the problem are *"the method is too coupled with the C_i class"*, or *"the method invokes too many methods of the C_i class"* where, C_i is the class *envied* by the method. On the other side, an answer like

[3]Depending on the code smell object of the question, a code component could be a method, a class, or a pair of classes.

"the method performs too many calls" is not considered enough to mark the bad smell as *identified*.

Performing this analysis for each code bad smell and for each category of participants in our study we should be able to verify (i) what are the most perceived and identified code smells, and (ii) if the participants' experience and system knowledge play a role in the ability of perceiving and identifying code smells.

As for $\mathbf{RQ}_2$, we exploited the answers to the question "please rate the severity of the coding problem" provided by participants. Answers have been mainly analyzed through descriptive statistics.

III. Analysis of the Results

Before answering the two research questions formulated in Section II-A, we analyze to what extent participants perceived a design problem in classes not containing any of the bad smells considered in our study. As previously explained, this is a sanity check aimed at verifying whether respondents were negatively biased. In total, we showed to participants 30 code components containing no smell (i.e., 10 on ArgoUML, 11 on Eclipse, and 9 on JEdit). Master's students, industrial developers, and original developers marked as affected by design problems 10%, 5%, and 1%, respectively, of these code components. The low percentage indicates the absence of a negative bias in the respondents, and that this is particularly true for those with more experience (industrial developers) and with a better knowledge of the code (original developers). Moreover, when manually analyzing these cases of false positives, we found that most of the design problems observed by participants in classes not affected by any smell were related to problems in comments (e.g., *comments are missing*) or method/class naming (e.g., *class name does not reflect the class purpose*). In other words, in some sense the respondents correctly identified some kinds of problems, although these are not really structural code smells, but more similar to lexical smells [14], out of scope for this study.

Turning to the core of our study, Figures 1(a) (ArgoUML), 1(b) (Eclipse), and 1(c) (JEdit), report report the percentage of participants (of the different categories) that (i) perceived a problem in the analyzed code, and (ii) correctly identified the bad smell present in the code component. Note that a code component that is correctly identified is also perceived (the opposite is not true). Columns labeled with "M", "I", and "O" report results for Master's students, industrial developers, and original developers, respectively. Also, Table V reports the median severity assigned by developers to the identified design/implementation problems[4].

A. Smells Generally not Perceived as Design or Implementation Problems

When looking at Figures 1(a), 1(b), and 1(c), one can immediately realize that some smells are, generally, not perceived as actual problems. This is particularly true for *Class Data Should Be Private*, *Middle Man*, *Long Parameter List*, *Lazy Class*, and *Inappropriate Intimacy*. In the following, we provide a qualitative analysis for them, based on the collected feedbacks and on the analysis of the code itself.

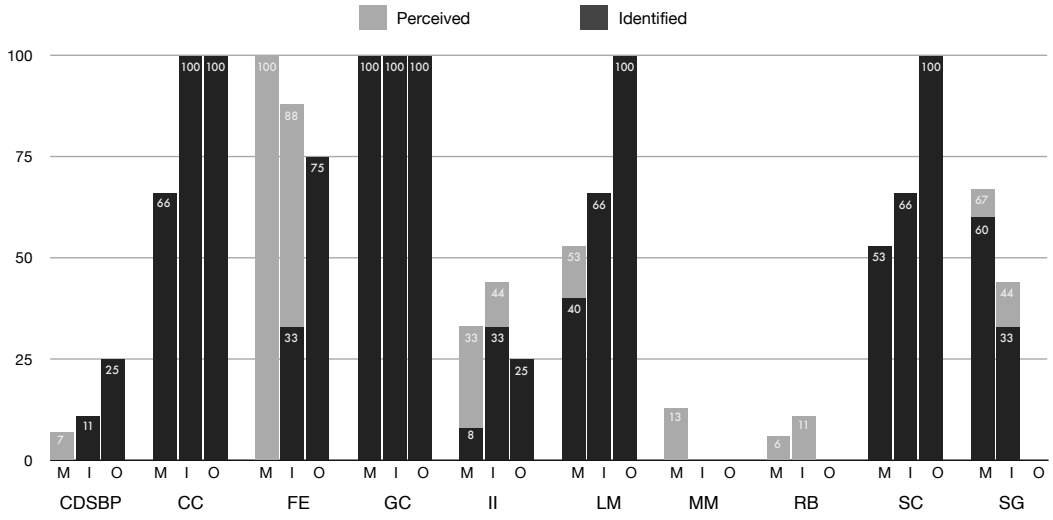
Class Data Should Be Private (CDSBP). This smell arises when a class exposes its attributes. Respondents did not perceive this as an issue for the analyzed code components. Only a small percentage of Master's students perceived a problem in such components; however, they were never able to associate the problem to the characteristics of the *CDSBP* smell, mainly claiming issues related to poor commenting and methods complexity. Few (18% on average on the three systems) industrial developers recognized *CDSBP* as an issue, while one original developer for each system recognized the problem in the code. However, by looking at the severity values (Table V), it emerges that respondents did not feel *CDSBP* as a real problem in the code. For instance, the severity assigned by original developers is *very low* (1) on ArgoUML, *medium* (3) on Eclipse, and *low* (2) on JEdit. It is interesting to report an observation made by the ArgoUML developer who recognized the *CDSBP* instance, while assigning it a *very low* severity: *"this class exposes all its fields, and this could look like bad coding. However, at the end of the day this is an utility class with public static fields*[5] *that can be used from everywhere in the system"*.

Middle Man (MM). *Middle Man* instances arise when a class is delegating all its work to other classes. Classes affected by this smell were perceived by developers as classes without any design problem. The only exceptions are 13% and 6% of Master's students perceiving (but not identifying) a design problem on ArgoUML and Eclipse, respectively (note that this bad smell is not present in JEdit). Thus, high levels of delegation between classes do not seem to bother developers. In our understanding, developers could better perceive such a smell when doing performance analysis—e.g., because the *Middle Man* could introduce overhead.

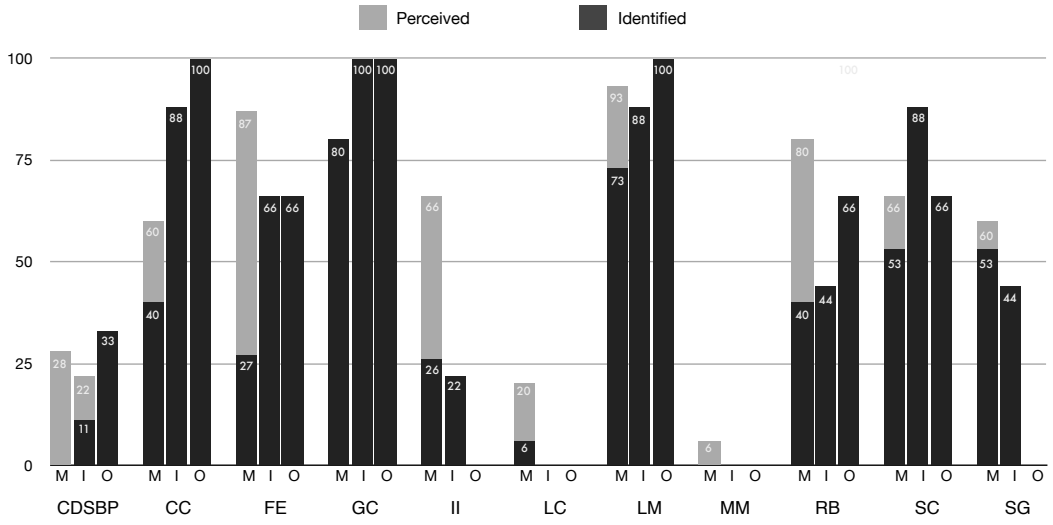
Long Parameter List (LPL). We found this smell in JEdit only (see Figure 1(c)). The method affected by this smell was `adjustDockingAreasToFit`, taking 11 parameters as input. While 53% of Master's students perceived a problem in the method, just 20% of them indicated the number of parameters as the issue. In most of the other cases, the problem felt by Master's students was the method complexity, the same perceived by the only original developer reporting a problem in the method. Finally, among the industrial developers, only one of them (11%) identified the problem in the method, however assigning it a severity of 3 (*medium*). The explanation provided justifies the low severity score: *"the method has several parameters; however, the feature implemented in it requires all of them"*. For this bad smell, the differences observed between the perception of students and professional developers can be explained as follows. Students are not used to large and complex projects. Consequently, they are more concerned by a method with several parameters as compared

[4]Complete data about the answers provided by participants are available in our replication package

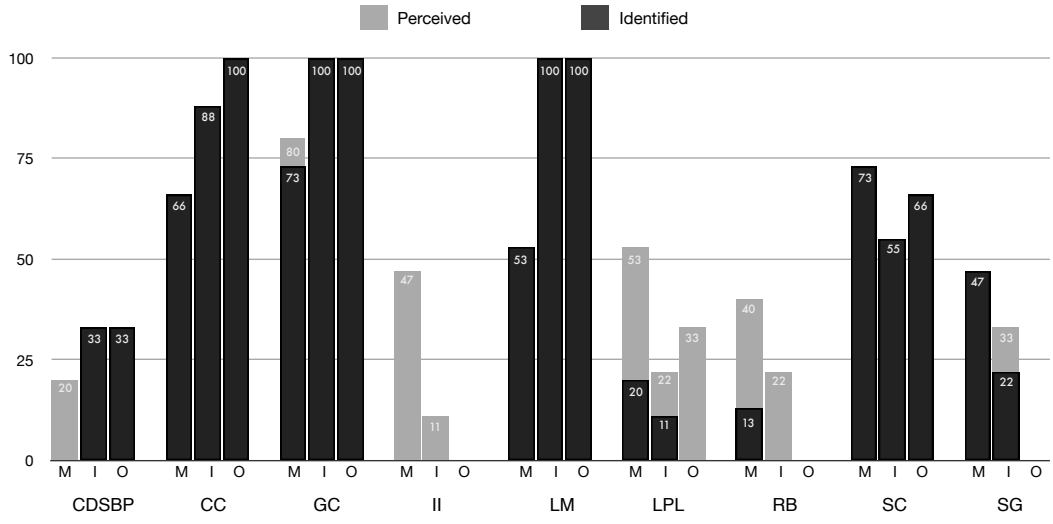
[5]Note that the fields were not *final*.



(a) ArgoUML



(b) Eclipse



(c) JEdit

Fig. 1. Percentage of (M)aster's students, (I)ndustrial developers, and (O)riginal developers that perceived and identified the bad smell examples.

TABLE V
MEDIAN OF THE SEVERITY ASSIGNED BY PARTICIPANTS TO THE IDENTIFIED DESIGN PROBLEMS.

System	Participants	CDSBP	CC	FE	GC	II	LC	LM	LPL	MM	RB	SC	SG
ArgoUML	Master's students	-	4	-	3	2	-	3	-	-	-	2	3
	Industrial	2	5	4	5	3	-	4	-	-	-	5	3
	Original	1	5	4	5	3	-	4	-	-	-	3	-
Eclipse	Master's students	-	4	4	5	4	1	4	-	-	4	3	3
	Industrial	3	4	5	5	4	-	5	-	-	-	5	3
	Original	3	5	5	5	-	-	5	-	-	4	5	-
JEdit	Master's students	-	4	-	3	-	-	3	3	-	4	3	3
	Industrial	2	5	4	5	3	-	4	3	-	-	5	3
	Original	2	5	-	5	-	-	5	-	-	-	4	-

to more experienced developers. Also, original developers are aware of the reasons why such methods have a high number of parameters and are therefore not particularly concerned.

Lazy Class (LC). This smell represents a very small class that does little in the system. It is considered a bad smell since *"each class costs money to maintain and understand"* [1]. This smell affects only one of the investigated projects (Eclipse) and it has been identified by only one Master's student (6%), that however ranked the problem as a *very low* severity one. In summary, respondents were not concerned about the class `SelectionOnNameOfMemberValuePair` of Eclipse that just contains two methods, one of which is a simple `print` method. On the one hand, it is not surprising to see a lower severity perception here. On the other hand, in order to recognize a *Lazy Class* as possible problem, one should have clearly in mind whether having such an additional class could, in perspective, have benefits (e.g., because the class itself is likely to evolve or to be extended by others), or negative effects (because it means scattering maintenance activities). Unfortunately, in this case we did not get opinions from original developers, the only ones that could have expressed an informed opinion.

Inappropriate Intimacy (II). This smell describes high levels of coupling between two classes. Our results show that respondents did not consider high coupling as a problem. Master's students perceived a problem in the relationship between the two highly coupled classes in 48% of the cases, although they only identified the problem in 11% of the cases. Industrial developers were able to identify the problem in 18% of the cases while, among the original developers, only one of them recognized the existence of a coupling issue in ArgoUML, by assigning it a *medium* (3) severity. The two involved classes (i.e., `ShortcutMgr` and `ActionWrapper`) have 27 dependencies among them. Despite that, the ArgoUML developer explained why this is not a big issue: *"*`ActionWrapper` *represents an action in the system that can be associated to a specific keyboard shortcut, while* `ShortcutMgr` *is in charge of managing all ArgoUML's shortcuts. Thus high coupling between these two classes is justified from my point of view"*.

Summarizing, the bad smells described above are not perceived as problems by respondents that, consequently, are not able to identify them in source code. This is true for all the three categories of developers involved in our study, highlighting how developer's experience and system's knowledge do not play any important role in these cases. Also, it is interesting to observe that all these five smells (i.e., *CDSBP*, *MM*, *LPL*, *LC*, and *II*) are related to the lack of applying good Object-Oriented (OO) design practices, rather than to something one can easily perceive by looking at the code, (e.g., as it would be for *God Class*). Indeed, by carefully looking at the definition of such smells, we notice that: *CDSBP* violates the information hiding principle; *MM* is a symptom of something wrong in the distribution of responsibilities between classes; *LPL* should be avoided in OO programming, since a method can ask other objects for the information it needs without the necessity of receiving all of them through parameters; *LC* often represents a class without a precise responsibility; and *II* identifies high-levels of coupling between classes.

B. Smells Generally Perceived and Identified by Respondents

There are some categories of smells that: (i) are highly perceived and identified by developers, (ii) create more concerns for developers having more experience and system knowledge, and (iii) are rated with high severity values. These smells are *Complex Class*, *God Class*, *Long Method*, and *Spaghetti Code*. As one can immediately notice, differently from the previous group, such smells are the ones for which the problem can be immediately perceived by looking at the code (which may be long and/or complex). In the following, we provide a detailed discussion for each of them.

Complex Class (CC). Original developers always identify this bad smell in the affected code components, assigning to it the maximum severity (i.e., 5, *very high*). The provided explanations highlight the problems derived by classes having a high code complexity: *"the class is too complex, intricate, and very difficult to comprehend"*, *"several methods in this class are very complex, negatively affecting its maintainability"*. Also, industrial developers generally identify the problem (92% of cases, on average) while the less experienced participants (i.e., Master's students) were able to describe the problem in 57% of case, on average. Thus, higher experience seem to alert developers about problems caused by working on complex code. Note that the median of severity assigned by all participants to *CC* is *high* or *very high* (see Table V).

God Class (GC). *GC* is the smell for which the respondents assigned the highest severity. Specifically, industrial and original developers always identified the problem in the analyzed code components, explaining how classes affected by *GC* are *"too large"*, *"a mixture of different responsibilities"*, resulting in *"difficulties in creating a mental model of how the class works"*. Industrial and original developers ranked *GC* with a median severity of 5 (*very high*). An example of GC instance evaluated in this study is the `GeneratorJava` class of

ArgoUML composed by 66 methods and explicitly defined by one of the original developers as a *"big class in need of refactoring"*. Master's students were able to identify the design problem in 84% of cases, on average, by however assigning lower severity values than industrial and original developers (see Table V). Thus, also on this bad smell higher developers experience seems to increase the threats perceived by *GC* instances.

Long Method (LM). Also this smell has been always identified by the original developers. The assigned median severity was 5 (*very high*) on Eclipse and JEdit, and 4 *high* on ArgoUML. An interesting comment made by one of the ArgoUML developers was *"this method is way too long, it could be split into three different methods"*. This comment confirms the *LM* bad smells as indicator of *Extract Method* refactoring [1] opportunities. Industrial developers identified the *LM* instances in 85% of cases, on average, assigning them a median severity of 4 (*high*) on ArgoUML and JEdit, and of 5 (*very high*) on Eclipse. On average, Master's students perceived the problem in 67% of the cases, correctly identifying it in 55% of the cases. Also in this case, the proportion of respondents with more experience that identified the problem is greater than the proportion of students.

Spaghetti Code (SC). On average, original developers identified *SC* instances in 77% of cases, followed by industrial developers (70%), and Master's students (61%). It is interesting to report the comment left by an ArgoUML developer: *"this class looks like procedural programming"*. This is exactly what the *SC* smell is: the abuse of procedural programming in OO code. The severity assigned by original and industrial developers is generally *high* or *very high*, compared to that perceived by students bounded between *low* and *medium*.

C. Smells whose Perception may Vary

Finally, there is a group of smells for which the perception varies case by case. Such smells are *Feature Envy*, *Refused Bequest*, and *Speculative Generality*.

Feature Envy (FE). This smell arises when a method is more interested in a class other than the one it is implemented in [1]. We found instances of *FE* in ArgoUML and Eclipse. Master's students almost always perceived some problems in methods affected by *FE* (100% in ArgoUML and 87% in Eclipse), however they always failed in correctly identifying the *FE* symptom in ArgoUML, and they only identify it in 27% of the cases in Eclipse. Instead, industrial developers were able to identify the *FE* instances in 50% of the cases, while for original developers this percentage goes up to 70%. One of the industrial developers explained that *"method* `parseMessage` *is placed in the wrong class. It should be moved to* `MyTokenizer` *since it is likely to change with that class"*. When identifying the *FE* smell, developers assigned to it high levels of severity, ranging from a median value of 4 (*high*) assigned by Master's students and industrial developers, to a median value of 5 (*very high*) assigned by original developers. As for other smells, it can be noticed that highly experienced developers are the ones that perceive this bad smell the most. We conjecture that *FE* smells are perceived mainly by original developers, because the "interest" of the *FE* method to other classes often grows over time and/or can be identified by the need for co-changing such a method together with other classes. Indeed, *FE* can be effectively identified by using historical data [8].

Refused Bequest (RB). This smell arises between two classes when one inherits pieces of functionality from the other and never uses them. This is the only smell for which we have strong contradicting results across the three object systems. In particular, on ArgoUML (see Figure 1(a)) and JEdit (see Figure 1(c)) all respondents almost never perceived classes affected by *RB* as problematic ones. The situation is quite different on Eclipse, where 40% of Master's students, 44% of industrial developers, and 66% of original developers identified instances of *RB* in the analyzed pair of classes. We analyzed the instances of *RB* evaluated by participants on the three object systems. What we found was that:

- in ArgoUML, the *RB* arises due to classes `TabSpawnable` and `TablePanel`. The latter overrides four out of the five methods inherited by `TabSpawnable`.
- in JEdit, the *RB* is between classes `CompletitionPopup` and `CompleteWord`. The latter overrides five out of the eight methods inherited by `CompletitionPopup`.
- in Eclipse, the *RB* is quite more extreme. In particular, class `DefaultBindingResolver` inherits 53 methods from class `BindingResolver` overriding 52 of them.

Note that in none of the three above cases the overriding methods invoke the `super` method of the superclass. The *RB* instance present in Eclipse was quite more visible than those present in ArgoUML and JEdit, concerning developers about its presence. The median severity assigned to this issue by original developers was *high* (4)–see Table V.

Speculative Generality (SG). This smell represents the only one that was mainly perceived and identified by developers with low experience than by experienced ones. Master's students identified this smell in 53% of cases, on average, followed by industrial developers (33%), and original developers, never perceiving any design problem in classes affected by this smell. By looking back at the definition of *SG*, instances of this smell arise when a class is declared `abstract` but it is not specialized by any other class in the system. From the perspective of a Master's student, that mainly learned OO in courses and textbooks, this looks like a wrong usage of the OO paradigm. The median severity assigned by them is 3 (*medium*). The industrial developers identifying the problem (33% on average) also provided a *medium* severity to the *SG* instances, and one of them left a comment likely explaining the reason why those classes do not represent a problem for original developers: *"this class is* `abstract` *but not inherited by any class of the system. However, it could be that it is a partial implementation of something that will be integrated in*

the system in future system releases". In other words, without having a deep knowledge of the system, of the rationale behind the implementation choices, and of the project schedule, it could be difficult in some cases to assess design and/or implementation problems. Results obtained for this smell also warns against the abuse of too aggressive bad smell detectors that, in cases like this one, would report potential problems based on symptoms like the ones describing the *SG*. Only by observing the class evolution—i.e., an `abstract` class would never be inherited during a long period of observation—one can say this is, indeed, a problem. Again, this reinforces the conjecture that historical data are extremely useful when identifying code bad smells [8].

IV. Threats To Validity

Threats to *construct validity* are mainly related to how the sample of smells used in the study was identified, and to how we measured the developers' perception of code smells. Concerning the identification of the smell sample, a big threat can be due to the fact that, despite the presence of multiple evaluators minimized the possible effect of false positives, the identified smell instances may depend on the perception of students and authors who inspected the code to identify such smells. Hence, it could be the case that participants evaluated what the two students and ourselves perceived as smells. However, the identification of smells was performed having all the possible support available, including smell definitions [1], [2], tools to compute metrics, and the source code change history. Certainly, in any case we had to limit to one (randomly selected) smell of each type per system, and this could have excluded instances of smells where the "magnitude" of the problem was more or less evident. However, such a kind of study involving industrial and original developers had quite strict constraints, i.e., we could not afford to involve them in long inspection tasks on a huge number of smells.

Concerning the measure of perception (Section II), we asked developers to tell us whether they perceived a problem in the code shown to them. In addition, we asked them to explain what kind of problem they perceived in order to understand whether or not they were able to correctly identify the design and/or implementation problem. Finally, for the severity we use a Likert scale that allows to compare responses of multiple respondents. We are aware that questionnaires could only reflect a subjective perception of the problem, and might not fully capture the extent to which the bad smell could affect software development activities. To this aim, studies such as the one done by Yamashita and Moonen [5] are more suited.

Threats to *internal validity* may be related to factors that have influenced our results. One factor is the response rate: while appearing not very high (39%), it is higher than what it is normally expected in survey studies—i.e., below 20% [15]—even for the part of study done with the original developers, for which we obtained 22% return rate. Note also that we ensured a participation of at least three original developers for each system. We have limited a possible bias effect—i.e., the fact that developers could have told us that they perceived the presence of smells even in code not containing any smell—by also showing source code elements without smells.

Threats to *external validity* concern the generalization of our findings. Such threats can be related to (i) the set of chosen objects, (ii) the kinds of smells investigated in the study, and (iii) the pool of the participants of the study. Concerning the chosen objects, we are aware that our study is based on smells detected in three systems only, and that further studies are needed to confirm our conjecture. In this study we had to constrain our analysis to a limited set of smell instances, because the task to be performed by each respondent had to be reasonably small. In this study we covered a pretty large variety of smells, i.e., the 12 described in Table I. However, there are some smells we did not consider. Finally, for what concerns the participants, they represent different categories of developers, ranging from Master's students, representative of junior developers, to senior industrial programmers, and original developers of the investigated systems, having a deep knowledge of the code used in the study.

V. Related Work

This section analyzes the literature related to (i) the identification of code bad smells in source code; and (ii) the analysis of the evolution of code smells in existing software systems.

A. Code Bad Smell Detection

In the last decade, several approaches have been proposed for the automatic detection of code bad smells. All the proposed approaches are based on top of catalogues and heuristics to identify code design defects reported in well-known books [16], [17], [1], [2]. Such approaches detect smells in different ways. A common practice is the use of constraints based on source code metric values. For example, Lanza and Marinescu [18] describe how to exploit quality metrics to identify "disharmony patterns" in code by defining a set of thresholds based on the measurement of the exploited metrics in real software systems. Marinescu [6] formulate metric-based rules that capture deviations from good design principles and heuristics. Munro [19] presents a metric-based detection technique able to identify instances of two smells, namely *Lazy Class* and *Temporary Field*, in source code. A set of thresholds is applied to the measurement of some structural metrics to identify those smells.

Khomh *et al.* [20] use quality metrics to train a Bayesian belief networks aiming at detecting bad smells. The main novelty of that approach is that it provides a likelihood that a code component is affected by a smell, instead of just providing a Boolean value like the previous techniques. Our study highlights the importance of such feature, because participants pointed out that only some smell instances—i.e., those where the symptoms are particularly evident—indeed represent serious problems.

Moha *et al.* [7] introduce DECOR, a method for specifying and detecting code and design smells. DECOR uses a Domain-Specific Language (DSL) for specifying smells using high-level abstractions. Four design smells are identified

by DECOR, namely *Blob*, *Swiss Army Knife*, *Functional Decomposition*, and *Spaghetti Code*. Code bad smells have also been identified by means of code structural analysis. For instance, Tsantalis *et al.* [9] presents JDeodorant, a tool for detecting *Feature Envy* smells with the aim of suggesting move method refactoring opportunities.

Besides structural information, historical data can be exploited for the detection of bad smells. Ratiu *et al.* [11] propose to use the historical information of the suspected flawed structure to increase the accuracy of the automatic problem detection. Palomba *et al.* [8] provide evidence that historical data can be successfully exploited to identify not only smells that are intrinsically characterized by their evolution across the program history—such as Divergent Change, Parallel Inheritance, and Shotgun Surgery—but also smells such as Blob and Feature Envy [8].

Finally, there are other approaches that use visualization technique to help developers in the detection of bad smells [21] [22] or design change propagation probability matrix to detect, in particular, two bad smells, called Divergent Change and Shotgun Surgery [23].

The study presented in our paper complements all these approaches. Specifically, our study helps to identify smells that are more relevant for developers, and hence to be recommended with higher priority. Also, our study permits the identification of symptoms that are simply insufficient to characterize problematic smells, as well as other symptoms that are, instead, necessary to consider. For example, in the context of our study we observed that in several cases symptoms derived from the evolution history of the system could help to properly identify a bad smell.

B. Empirical Studies on Code Bad Smells

Code bad smells have also been the object of empirical studies aimed at investigating their evolution and their effect on source code comprehension and maintenance.

Regarding the evolution of bad smells, Chatzigeorgiou and Manakos [24] observe how the number of bad smells in software systems increases over time and that the developers almost never invest effort in removing them from the system. Similar results are achieved in the study conducted by Peters and Zaidman [25], who observe that even if often developers are aware of code smells, they are not very interested in activities aimed to remove them. This result is also confirmed by Arcoverde *et al.* [26], that report the preliminary results of a survey aimed at understanding the longevity of code smells and the reasons why developers do not care of removing them from the code; Arcoverde *et al.* show that often code smells remain in source code for a long time and the main reason to postpone their removal through refactoring activities is to avoid API modifications [26].

Bad smells are also studied to investigate their impact on maintenance properties, such us code comprehensibility and defect-or change-proneness. Abbes *et al.* [3] investigate the impact of two types bad smells—Blob and Spaghetti Code—on program comprehension. The results show that the presence of an antipattern in the source code does not decrease the developers' performance, while a combination of bad smells makes developers to significantly decrease their performance. The interaction between code smells has been extensively studied by Yamashita and Moonen [5]. They show that the maintenance problems are not deriving from the presence of a single code smell in a class, but they are strongly related to the co-occurrence of bad smells in the same file.

As for the defect- or change-proneness, Khomh *et al.* [4] provide evidence that code containing code smells or participating in antipatterns is significantly more change prone than other code; they also found that code participating in antipatterns has a higher fault-proneness than the rest of the system code.

Li and Shatnawi [27] present an empirical study that investigate the correlation between the presence of bad smells and class error probability. The study is conducted in the context of the post-release system evolution process, and the results show that there are bad smells positively associated with the class error.

All these studies provide evidence that code bad smells have negative effects on some maintenance properties. However, it is still unclear the extent to which bad smells are actually perceived as serious design problems by developers. Our paper aims at bridging the gap between the theory and the practice, providing empirical evidence on the extent to which bad smells are perceived as actual design and implementation problems by developers.

VI. Conclusion and Future Work

In this paper we conducted an empirical study aimed at analyzing to what extent code bad smells are perceived by developers as actual design and/or implementation problems. The study concerned examples of 12 kinds of smells detected in three Java open source projects—ArgoUML, Eclipse, and JEdit—and involved 10 original developers from the three projects and 24 outsiders, of which 9 are industrial developers and 15 are Master's students. The study results allowed us to distill the following lessons learned:

Lesson I. *There are some smells that are generally not perceived by developers as design problems*. Those smells are *Class Data Should Be Private*, *Middle Man*, *Long Parameter List*, *Lazy Class*, and *Inappropriate Intimacy*. As explained, these smells are all related to object-oriented (OO) good programming practice more than to complex/long code. Some of the explanations provided by developers highlighted as apparent violations of OO design principles, such as high levels of coupling or the absence of information hiding, do not necessarily reflect problematic situations. Sometimes they are simply the result of conscious choices made by developers. This underlines how approaches to (semi)automatically improve source code quality (e.g., refactoring recommendation systems) cannot simply be evaluated through quality metrics, but should always be assessed with developers, in order to verify if the refactoring recommendations really reflect design problems from a developer's point-of-view.

Lesson II. *Instances of a bad smell may or may not represent a problem based on the "intensity" of the problem*. This, for example, happens for *Refused Bequest*, for which only the instance detected in Eclipse was recognized as a serious problem. This is pretty much consistent with results of the previous study made by Ratiu *et al.* for God classes [11]. This result highlights the usefulness of smell detectors providing a measure of severity for each identified smell, as in the approach by Khomh *et al.* [20]. In this way, developers can focus their attention on smells that are more likely to represent a threat from their point of view.

Lesson III. *Smells related to complex/long source code are generally perceived as an important threat by developers*. This happens for *Complex Class*, *God Class*, *Long Method*, and *Spaghetti Code*. Not only these smells were consistently identified by a very high proportion of respondents, but also they were rated with the highest level of severity. Intuitively, it could simply be the case that such smells are the easiest to be identified by developers, but it can also be that these are the problems for which developers require the most effective solutions, i.e., precise recommenders that identify the smells and propose working solutions aimed at factoring them out.

Lesson IV. *Developer's experience and system's knowledge play an important role in the identification of some smells*. This is particularly true not only for the smells related to complex/long code, but also for smells related to possible misuses of OO principles, e.g., the *Feature Envy*. This confirms that code quality assessment is a crucial task and team managers should allocate senior developers on them rather than junior programmers; despite a good academic background, the latter might not be able to properly identify and judge the problems in the code. In addition to that, as discussed above, an appropriate judgement of the severity of smells often require a good knowledge of the overall system design, of the rationale of decisions taken in the past, and of possible evolution trajectories the system would have in the future. Only experienced developers would know all these details.

As always happens with empirical studies, extending the study using a larger set of smells, other software, and different participants is the only way to corroborate our findings. Such replications are part of the agenda of our future work.

REFERENCES

[1] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.

[2] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray, *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1^{st} ed. John Wiley and Sons, March 1998.

[3] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *15th European Conference on Software Maintenance and Reengineering, CSMR 2011*. IEEE Computer Society, 2011, pp. 181–190.

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

[5] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 682–691.

[6] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*. IEEE Computer Society, 2004, pp. 350–359.

[7] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur, "Decor: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, pp. 20–36, 2010.

[8] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *11th ACM/IEEE International Conference on Automated Software Engineering (ASE 2013)*. ACM, pp. 268–278.

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

[10] A. Jbara, A. Matan, and D. G. Feitelson, "High-MCC functions in the linux kernel," in *IEEE 20th International Conference on Program Comprehension, ICPC 2012, Passau, Germany, June 11-13, 2012*, 2012, pp. 83–92.

[11] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu, "Using history information to improve design flaws detection," in *8th European Conference on Software Maintenance and Reengineering (CSMR 2004), 24-26 March 2004, Tampere, Finland, Proceeding*. IEEE Computer Society, 2004, pp. 223–232.

[12] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjøberg, "Are all code smells harmful? a study of God Classes and Brain Classes in the evolution of three open source systems," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE, 2010, pp. 1–10.

[13] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *20th Working Conference on Reverse Engineering (WCRE 2013), October 14-17 2013, Koblenz, Germany*. IEEE, 2013 (to appear).

[14] S. Lemma Abebe, S. Haiduc, P. Tonella, and A. Marcus, "The effect of lexicon bad smells on concept location in source code," in *11th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM 2011*. IEEE, 2011, pp. 125–134.

[15] Y. Baruch, "Response rate in academic studies a comparative analysis," *Human Relations*, pp. 52(4):421–438, 1999.

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

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

[18] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[19] M. J. Munro, "Product metrics for automatic identification of "bad smell" design problems in java source-code," in *Proceedings of the* 11^{th} *International Software Metrics Symposium*, 2005.

[20] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Proceedings of the 9th International Conference on Quality Software*. Hong Kong, China: IEEE CS Press, 2009, pp. 305–314.

[21] F. Simon, F. Steinbr, and C. Lewerentz, "Metrics based refactoring," in *Proceedings of 5th European Conference on Software Maintenance and Reengineering*. Lisbon, Portugal: IEEE CS Press, 2001, pp. 30–38.

[22] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*. IEEE CS Press, Oct. 2002.

[23] A. A. Rao and K. N. Reddy, "Detecting bad smells in object oriented design using design change propagation probability matrix," in *Proceedings of the International MultiConference of Engineers and Computer Scientist*, Hong Kong, China, 2008.

[24] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 2010, pp. 106–115.

[25] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *European Conference on Software Maintenance and ReEngineering*. IEEE, 2012, pp. 411–416.

[26] R. Arcoverde, A. Garcia, and E. Figueiredo, "Understanding the longevity of code smells: preliminary results of an explanatory survey," in *Proceedings of the International Workshop on Refactoring Tools*. ACM, 2011, pp. 33–36.

[27] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, pp. 1120–1128, 2007.