Information and Software Technology 92 (2017) 223–235

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

How developers perceive smells in source code: A replicated study

Davide Taibi[*], Andrea Janes, Valentina Lenarduzzi

Free University of Bozen-Bolzano, Piazza Università 1, 39100 Bolzano, Italy

ARTICLE INFO

Article history:
Received 12 December 2016
Revised 16 August 2017
Accepted 17 August 2017
Available online 6 September 2017

Keywords:
Software maintenance
Code smells
Bad smells
Antipatterns
Refactoring

ABSTRACT

Context. In recent years, smells, also referred to as bad smells, have gained popularity among developers. However, it is still not clear how harmful they are perceived from the developers' point of view. Many developers talk about them, but only few know what they really are, and even fewer really take care of them in their source code.

Objective. The goal of this work is to understand the perceived criticality of code smells both in theory, when reading their description, and in practice.

Method. We executed an empirical study as a differentiated external replication of two previous studies. The studies were conducted as surveys involving only highly experienced developers (63 in the first study and 41 in the second one). First the perceived criticality was analyzed by proposing the description of the smells, then different pieces of code infected by the smells were proposed, and finally their ability to identify the smells in the analyzed code was tested.

Results. According to our knowledge, this is the largest study so far investigating the perception of code smells with professional software developers. The results show that developers are very concerned about code smells in theory, nearly always considering them as harmful or very harmful (17 out of 23 smells). However, when they were asked to analyze an infected piece of code, only few infected classes were considered harmful and even fewer were considered harmful because of the smell.

Conclusions. The results confirm our initial hypotheses that code smells are perceived as more critical in theory but not as critical in practice.

1. Introduction

Software Quality Assurance (SQA), i.e., assuring that software fulfills the posed quality standards, remains a task that requires effort and expertise.

Software is invisible, which "means that it is very easy for the project to proceed for a considerable time before problems become apparent, and without it being possible to verify that the passing of time and expenditure of money correlate with progression of the project in the desired direction [1]." To cope with the invisibility of software is costly: quality-related information is difficult to collect [2,3], requires time and effort; as a consequence, often other activities, e.g., adding new features to a product are given a higher priority then investing in improving the internal quality of the software [3]. Quality tools aim to reduce the costs of quality, however, while such tools alleviate the task of collecting quality-related information, they often require substantial effort to understand the analysis they provide.

One type of analysis that quality tools provide, is the detection of code smells [4] and antipatterns [5]. Code smells are structural characteristics of software, which may indicate code or design problems that can make software hard to evolve and maintain [4]. In this work we adopt the term code smells for both code smells and antipatterns.

Several studies consider code smells harmful from a maintenance point of view [6–11], while others suggest that smells are not terribly problematic [12]. Code Smells are also considered a cause of potential faults by several studies [13–17], while other studies report a significant but small effect on them [18]. Moreover, Code Smells present in the source code are also considered causes of higher change-proneness [14,15,19–21] and low code understandability [22].

Developers often do not know about the code smells they introduce into their source code (including the decreasing maintainability). For this reason, the identification of code smells (and the investment of time to remove them) is gaining acceptance in industry [23].

* Corresponding author.
E-mail addresses: davide.taibi@unibz.it (D. Taibi), andrea.janes@unibz.it (A. Janes), valentina.lenarduzzi@unibz.it (V. Lenarduzzi).

http://dx.doi.org/10.1016/j.infsof.2017.08.008
0950-5849/© 2017 Elsevier B.V. All rights reserved.

Some examples of SQA tools that are proposing techniques for detecting and reducing code smells are JDeodorant,[1] DECOR [24], or SonarQube.[2] However, the term “code smell” is not always clearly understood. As an example, SonarQube, one of the most frequently used tools for SQA, renamed the concept of “Code Issues”, which originally related to the adherence to coding standards, into the term “code smells”, increasing the misunderstanding of code smells from the developers’ point of view. In this context, in addition to the question of whether SQA practices are worth the effort or not, researchers are also discussing how developers perceive code smells:

- if they are perceiving them as a serious problem that deserves extra effort to be solved; and
- if they see them as some sort of hint that they will take into consideration next time they will edit that code.

The first question that arises is whether developers have a common understanding what a code smell is; the second how harmful they consider them.

Several researchers have investigated code smells in the past; we focus on two particular studies that studied the two questions just mentioned: the first study by Yamashita and Moonen [8] entitled “Do Developers Care about Code Smells? An Exploratory Survey” investigates if developers consider code smells as something harmful; the second study by Palomba et al. [25] entitled “Do they Really Smell Bad? A Study on Developers’ Perception of Bad Code Smells” investigates if developers have a common understanding of code smells, i.e., if when confronted with them in the code they recognize it as the same problem, and how harmful they see the problem. At the best of our knowledge, these two studies were the only ones assessing the perceived harmfulness of code smells.

We designed this study as a differentiated external replication of two studies [26]. The differences of this replication to the previous two studies are:

- The respondents in the study by Yamashita and Moonen [8] were mostly originating from India, USA, Pakistan and Romania (38 from 73 respondents). Since we conducted our study during a conference that took place in Europe (see Section 3), we mainly had European and American participants. Moreover, the origin of the respondends in [25] was not reported so we cannot compare this aspect to our study.
- Palomba et al. [25]. interviewed 34 participants in their study. Only 9 were industrial developers, 10 were developers involved in the original projects from which the authors took code examples, and 15 were master students. The selected interviewees expose this study to two threats to validity:
  - Developers who are familiar with the code being evaluated might have a different perception of the harmfulness[3] of a code smell than somebody who has never seen the code. Some code smells may be intentionally left in the code because they are a side effect of another design decision. Such background knowledge can bias the elicitation of the attitude of developers towards code smells.
  - The question of whether students can be used as subjects in software engineering experiments is widely debated (e.g., in [27]). We find that in the study on how code smells are perceived, experience plays a crucial role. If the question is how developers perceive code smells, students might assess problematic Java classes differently from professional software developers.
- Palomba et al. consider only 12 code smells, while we extended our study to all 23 code smells proposed by Fowler et al. [4].

According to our knowledge, this is the largest study so far investigating the perception of code smells with professional software developers: we collected the perception about the harmfulness of code smells of 63 participants and assessed the ability to identify and categorize code smells in source code examples of 41 participants. Moreover, we compared the perceived harmfulness of 32 participants first just based on the definition and then again when they were confronted with infected[4] source code.

The main findings of our work are:

1. Some smells are considered important in theory but are not perceived as a design problem in practice.
2. Some smells are considered as not harmful in theory but are perceived as a design problem in practice.
3. Smells related to size and complexity are considered harmful by a higher percentage of participants than others.

The remainder of this paper is structured as follows. In Section 2, we discuss the background and related work: in Section 3, we describe the empirical study focusing on the study process, the study execution, and the data analysis. In Section 4, we present results obtained, while in Section 5 we discuss them. Section 6 discuss on threats to validity. Finally, in Section 7 we draw conclusions and outline future work.

2. Background and related work

In this section, we first introduce code smells and then report on empirical studies on them.

2.1. Code smells

Code smells, also referred to as bad smells, were first introduced by Kent Beck and Martin Fowler in 1999 [4], extending the code “pitfalls” defined in 1995 by Webster [28]. They can be considered as “poor” implementation and design decisions that may make it difficult for programmers to carry out changes and that hinder the evolution of systems. They are not considered defects but may increase the probability that flaws exist in a piece of software that may affect both the design stages and the implementation.

Table 1 presents the list of code smells proposed by Martin Fowler [4] and used in this study.

2.2. Empirical studies on code smells

Several studies found that code smells indeed are good indicators for code parts with a *low maintainability*, e.g., Deligiannis et al. [9] found that considering a specific design heuristic (to avoid a bad smell) in an experiment had a positive impact on creating more maintainable design structures; Malhotra et al. [10] found that bad smells could be used as an important source of information to *quantify flaws* in classes; and Fenske and Schulze [29] found that code smells that also consider variability, are good indicators of *low program comprehension, maintenance*, and *evolution* in software product lines.

Unfortunately, other studies found that considering code smells can have a negative impact on *maintainability* or that the impact is not always clear: e.g., Kim et al. [6] found that refactoring code clones did not always improve software quality; Yamashita and Moonen [7] found that only some code smells reflected important maintainability aspects, others required combining different

[1] https://marketplace.eclipse.org/content/jdeodorant.

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

[3] Some authors instead of “harmfulness” speak of “criticality”. Both terms express a potential risk attached to a problem but have slightly different connotations. For the sake of clarity, in this paper, we only use the term “harmfulness”.

[4] We call source code “infected” if it contains code smells.

Table 1
Code smells description (adapted from [4]).

Name	Description
Application-level smells	
Duplicated code	The same code reused in different locations.
Class-level smells	
Blob	A big class (usually a singleton) that has dependencies with data contained in other data classes. It could monopolize several system operations.
Class data should be private	Class publicly exposing variables.
Cyclomatic complexity	Also referred as McCabe's Cyclomatic Complexity. It refers to methods that should be simplified since they have too many independent paths of execution.
Downcasting	A cast to a derived class.
Excessive use of literals	Too many literal variables embedded into the code instead of being declared as constants.
Feature envy	One object gets at the fields of another object to perform some sort of computation or make a decision, rather than asking the object to do the computation itself.
Functional decomposition	A class with too many functionalities, that needs to be broken down into smaller and simpler classes.
God Class	A huge class implementing different responsibilities.
Inappropriate intimacy	A method that has too much intimate knowledge of another class or method's inner workings, inner data...
Large class	A class that is too big.
Lazy class/Freeloader	A class with very limited functionalities.
Orphan variable or constant class	A class containing variables used in other classes.
Refused bequest	A children class that never uses the inherited methods.
Spaghetti code	A class with a very complex control flow.
Speculative generality	An abstract class with a very limited number of children who are not using its methods.
Swiss army knife	A class who is providing many services, for different purposes, such as a swiss army knife.
Tradition breaker	A class that, despite from inheriting from another class, does not fully extend the parent class and has no subclasses.
Method-level smells	
Excessively long identifiers	Too long variable names that no not respect the naming conventions
Excessively short identifiers	Too short variable names that do not allow to understand the purpose of the variable.
Excessive return of data	A method that returns more that what is needed from the calling methods.
Long method	A method that is too long.
Too many parameters	Methods with too many parameters, which usually become hard to read and to test.

approaches to achieve more complete and accurate evaluations of overall maintainability of a system; Sjoberg et al. [12] found that refactoring code smells did not reduce maintenance effort in their experiment; and Yamashita and Moonen [22] found that code smell interactions occurred across coupled artifacts, with comparable negative effects as same-artifact co-location (e.g., *low code understandability*).

Some studies report that the presence of code smells is not necessarily an indicator for *more defects* or *defect proneness*: Olbrich et al. [15] found that the presence of God and Brain Classes is not necessarily harmful, in fact, such classes may be an efficient way of organizing code; and Danphitsanuphan and Suwantada [16] found that not all analyzed code smells could be linked to the structural defects they analyzed.

Other studies attribute a *higher change-proneness* to infected sources: Khomh et al. [19] found that in almost all releases of the code they studied, classes with code smells are more change-prone than others, and that specific smells are more correlated than others to change-proneness. Olbrich et al. [21] found that they could identify different phases in the evolution of code smells during the system development and that code smell infected components exhibited a different change behavior over time. Zazworka et al. [14] found that god classes are changed more often and contain more defects than non-god classes.

We conclude from the existing empirical studies that code smells should not always lead to refactoring, that the effects of specific code smells need to be further investigated, and that their presence might have repercussions beyond the infected code fragment.

2.3. The replicated studies

This paper is based on two studies: one study published in 2013 by Yamashita and Moonen [8] and another study published in 2014 by Palomba et al. [25].

Yamashita and Moonen [8] investigated the importance of code smells from the developers' point of view, to understand if code smells are not important because developers are not aware of them or because they do not know them. They ran a exploratory, descriptive survey with 85 professional software developers. As a result, they found that nearly one third of the participants never heard about code smells, while the remaining two thirds got in touch with code smells on blogs, seminars and forums. Most of the participants were slightly concerned about code smells. However, the not concerned ones were not experienced with code smells. Finally, the top three most popular code smells they identified from the survey were: Duplicated Code, Long Method, and Accidental Complexity.

Palomba et al. [25] built on the study performed by Yamashita and Moonen described above: while Yamashita and Moonen investigated to which extent developers **knew** about code smells, Palomba et al. studied whether developers *perceive* a code smell in the source code as a problem and if they can describe the issue in a way that is attributable to the code smell definition. The goal of their work was to understand whether code smells were believed to be a theoretical problem or if they actually are a practical problem. In their study, they showed 34 developers code fragments containing code smells from three open source projects. They also showed code fragments (Java classes) without code smells, so as to understand if results were biased by the expectation of always find an issue into the code. They asked them if there was a design problem in the code, to describe the problem and to evaluate the severity of the problem. The participating developers included 15 master students, 9 industrial developers, and 10 developers involved in the projects from which the code fragments were taken. As a result, Palomba et al. found that:

- There are smells that developers do not perceive as design problems; in their study, this included Class Data Should Be Private, Middle Man, Long Parameter List, Lazy Class, and Inappropriate Intimacy;

- the perceived severity of a problem varied among the participants;
- Long Class, Cyclomatic Complexity, and Long Method were mainly considered as harmful by developers; and
- to identify the smell, the experience of developers and their knowledge of the system played a crucial role.

The next section presents the empirical study based on the two studies presented here listing the research questions, and describing the adopted study process, the study design, and the study execution.

3. Empirical study

As stated in the introduction, the objective of this research is to understand how developers perceive code smells and how well they are able to identify them in code. We based our research on the two papers of Yamashita and Moonen [8] and Palomba et al. [25] and designed it as an differentiated external replication of the two studies. Therefore, our replication also consists of two steps, representing the replication of the two initial studies.

In the first step, we aimed to understand whether developers perceive code smells as harmful, based on the description of the code smell, while in the second step we aimed to understand whether developers are able to recognize code injected with code smells as problematic code. In addition to the previous studies, we then aimed to understand whether developers perceive a code smell from the definition as harmful as when they find it in the source code.

In this section, we present the study plan, including the goal, the research questions, the study execution, and the data analysis.

Formulated as a Goal–Question–Metric (GQM) goal [30,31], the goal of our study is:

Compare the theoretical and practical perceived harmfulness of code smells
from the point of view of software developers
in the context of agile software development processes

We formulated the following five research questions:

(RQ1) How harmful are the different code smells perceived by developers *just reading their definitions*? In this RQ, given a description and an example of each code smell, we aim at understanding the perceived harmfulness of a code smell from the practitioners' point of view. Note that in this RQ we do not evaluate whether practitioners know a specific code smell but only whether they consider it as a threat in the code [8].

(RQ2) Do developers perceive a piece of *code that is infected* with a specific code smell as harmful and worth to invest effort to remove it? With this research question we aim to understand whether developers consider a piece of code infected with code smells as problematic. Note that here we are not aiming at understanding their knowledge of a specific smell or asking them to recall which smell is injected into the code. Out hypothesis is that practitioners are able to identify a harmful piece of code even if they do not recall the formal definition of a specific code smell [25].

(RQ3) Can developers correctly *identify and describe* code smells identified in the source code? In this RQ, we extend RQ2 by analyzing whether practitioners are able to correctly describe the problem of a piece of code infected by a code smell [25].

(RQ4) Can developers correctly *name* code smells in the source code? In this RQ, we extend RQ2 by analyzing whether practitioners are able to correctly associate the name of an existing code smell with a piece of code infected by the same code smell [25].

(RQ5) Do developers consider a piece of code infected with a specific code smell *as harmful* as they perceive the same smell based on the description? With this research question we aim at understanding whether developers consider a piece of code infected with code smells as harmful. Note that here we are not aiming at understanding their knowledge of a specific smell or asking them to recall which smell is injected into the code. Our hypothesis is that practitioners are able to identify a harmful piece of code but do not know about the existence of a code smell, or that they perceive the description as harmful but the infected code as not harmful.

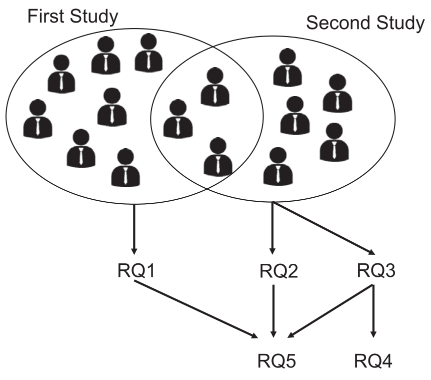

Fig. 1. The study process.

3.1. Study process

In this section, we describe the study process used to carry out both studies. Fig. 1 depicts the study process and the research questions mapped to the two studies.

The first study investigates RQ1, the second study RQ2, RQ3, and RQ4. Only the answers of participants that participated in both studies could be then used to compare the perceived harmfulness before seeing infected code and the perceived harmfulness when identifying a problem within the code, therefore answering RQ5.

3.2. First study: developers' perceived harmfulness of code smells

With this first empirical study, we aimed to answer RQ1, i.e., to understand the developers' perceived harmfulness of code smells based on their definitions. For this purpose, and in order to build on the results based on the previous body of knowledge and to compare our results with them, we externally replicated the study [8], extending it by:

- Considering the comprehensive list of code smells (see Table 1); and by
- analyzing the answers based on team size, personal experience, and application domain.

3.2.1. First study: study design

The study was designed as a survey carried out through a questionnaire and conducted with developers and professionals using Agile development processes with different experience levels. Since we aim at characterize and study the perceived harmfulness of code smells, we replicated the same explorative and descriptive survey used in [8]. However, to extend the study, as suggested in the future work section of Yamashita and Moonen [8], we considered the complete list of code smells as reported in Table 1.

The questionnaire was composed of four sections:

1. *Background information*: Here we collected the profile of the respondents. We considered age, country, gender, predominant roles, development language experience, familiarity with programming languages, and working experience (in months and in code size). Moreover, to collect information on the company where the respondents work, we collected the organization's size via the number of employees and the common application domain. This section was added as suggested in [8] to understand possible differences among developers working in different domains or at companies with different sizes.
2. *Knowledge of code smells*: In this section, we wanted to understand whether the participants are familiar with code smells and whether they already consider the removal of code smells in their development process. This section of the questionnaire is useful for understanding whether the answers provided are based on personal experience and previous knowledge of code smells or only on the reading of the description we provided in the next question. For this reason, we asked how developers evaluate their code quality, whether they are familiar with code smells, and how they get in touch with code smells. Finally, we wanted to know whether they use any tool for detecting code smells and if so, in which phase of their development process, and how often they remove them.
3. *Perceived harmfulness of code smells*: To capture the general perception of the awareness of code smells of our respondents, in this section we asked each participant to rate how concerned they are about smells in the code. Finally, to understand how harmful different code smells are perceived from their definitions, we asked the participants to rate how concerned they are with all the code smells listed in Table 1. Unlike [8], we proposed the complete list of code smells [4] instead of asking them to list them in an open question. The answers were proposed in the form of an ordinal 4-points Likert scale, where 1 corresponded to not concerned at all and 4 to very concerned.

Moreover, at the end of the questionnaire, we left space for further comments.

3.2.2. First study: study execution

The study was executed over the course of four days, during the 17th International Conference on Agile Processes, in Software Engineering, and Extreme Programming (XP 2016). We distributed a total of 250 questionnaires.

3.3. Second study: perceived harmfulness of code injected with code smells

With this study, we aimed to answer RQ2, RQ3 and RQ4, i.e., to understand whether developers are able to recognize code injected with code smells as problematic code and whether they are able to associate the correct code smell with the infected piece of code. For this purpose, we externally replicated the study [25]. However, since the material used in the study was not completely available on-line, we contacted the authors so as to use exactly the same material and tasks. The code smells used in [25] were chosen from the original list of Fowler et al. [4] in a ”subjective manner [25]”. The selection was driven by those code smells considered ”bad smells” by the authors, related either to complex/large code components or to the nonadoption of good object-oriented coding practices. Therefore, to be coherent with the results of the first study, we extended the replication by:

- Considering all the 23 code smells reported in Table 1, instead of the 12 previously considered in [25];
- comparing the concernedness of each respondent towards code smells who participated in both studies.

As in [25], the main focus of the study was on code comprehensibility and maintainability.

3.3.1. Second study: study design

For the population of our study, we selected developers with at least three years of experience. We invited all the participants of the first questionnaire to participate in the second study. Moreover, we extended the invitation to other developers with the same requirements. We analyzed the perceived harmfulness of the 23 code smells described in Table 1. Prior to the start of the study, we identified instances of the 23 considered code smells in the same three opensource projects analyzed in [25], namely ArgoUML,[5] Eclipse,[6] and JEdit.[7] Moreover, we identified additional code samples infected with the additional code smells not considered in [25] but considered in this study.

Using DECOR (Defect dEtection for CORrection) [24], we identified source code fragments in the three open-source projects that contain only single code smell. We consider fragment pieces of code composed by one or more classes. We adopted DECOR [24], supported by a manual identification of the Code Smells, to closely replicate the results obtained in [25]. Moreover, as also reported by Moha et al. [24], the code smells detection algorithms of DECOR ensures 100% recall and a precision higher than 80%.

From the list of candidate fragments we selected those already chosen by Palomba et al. [25]; therefore, for the 12 types of code smells of Palomba et al. [25], the selected source code fragments are the same. We randomly chose additional source code fragments that contained the missing types of code smells until we obtained the complete set of source code fragments for each of the 23 code smells considered.

We manually validated that the source code fragments really contained the code smells reported by DECOR [24], based on the definition of the respective code smells by Fowler et al. [4]. No disagreement was identified in this process. Therefore, we did not calculate the pairwise inter-rater reliability.

We implemented the questionnaire using Opinio,[8] asking the participants to identify whether a set of code fragments had design/implementation issues and reporting their perceived harmfulness of such problems. Since we wanted to test the perceived harmfulness of code injected with code smells, as in [25], we did not present to the participants a list of code smells beforehand, to not influence the participants in their ability to recognize a potential issue only because they see the list of code smells we consider.

In the following, we summarize the tasks contained in the online questionnaire. We presented a random set of 30 source code fragments including the 23 infected with only one code smell and 7 randomly selected instances without problems. The six not infected instances were added to allow us to evaluate whether the participants tend to always find problems, only because they are expecting to find them.

Note that we use the term source code fragment to refer to the code class(es) containing the code smell. For example, a code fragment could contain a too long method or a method of another class can have too many parameters. to isolate each smell, we selected fragments affected by only one smell, so as to avoid to confuse participants. to closely replicate the results obtained in [25], we defined a task for each smell reported in Table 1, composed by the same questions defined in [25]:

- In your opinion, does this code component exhibit any design and/or implementation problem?

[5] http://argouml.tigris.org/.
[6] http://www.eclipse.org/.
[7] http://jedit.org/.
[8] http://www.objectplanet.com/opinio/.

  - 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-point Likert scale: 1 (very low), 2 (low), 3 (medium), 4 (high), 5 (very high).
- We also added six tasks where we asked participants to answer the same questions on different code fragments not affected by code smells.

Instructions to participants were sent via email, describing how to answer the survey. As in [7], the participants were allowed to answer in four weeks.

3.3.2. Second study: study execution

We conducted this empirical study from June 2016 to September 2016. Thanks to the XP2016 organizers, we were able to invite all the developers who registered for the conference to participate in the survey. Moreover, we sent the invitation to participate to several developers' mailing lists for a total of more than 300 sent invitations. The participants were allowed to fill out the questionnaire in multiple rounds (e.g., answering a set of questions one day and the remaining ones on different days). The website used to conduct the survey automatically proposed the first unanswered question in each round, hiding those to which the participant had already replied. However, to allow this procedure, the participants were required to use the same PC and the same browser each time they connected to the survey.

3.4. Data analysis

In this section, we describe the procedure used for analyzing the data collected in both studies.

We partitioned our responses into more homogeneous subgroups prior to the analysis based on demographic information to compare the responses obtained from all the participants with the different subgroups separately.

The first study asked the participants to respond to multiple-choice questions on nominal data and ordinal data such as five-point Likert scales. It also includes open questions. Nominal data were analyzed by determining the proportion of responses in each category. A Chi-square test was carried out to measure associations among variables.

Ordinal data, such as 5-point Likert scales, were not converted into numerical equivalents, since using a conversion from ordinal to numerical data entails the risk that subsequent analysis will give misleading results if the equidistance between the values cannot be guaranteed. Moreover, analyzing each value of the scale allows us to better identify the possible distribution of the answers.

Open questions were analyzed via open and selective coding [32]. We extracted codes from the answers provided by the participants and answers were grouped into different code smells. For other open questions, answers were interpreted extracting concrete sets of similar answers and grouping them based on their perceived similarity.

The qualitative data analysis has been conducted individually by each author. Participants reported very similar answers, easily attributable to a specific smells. As example, participants reported the smell as “Long method” or “Method too long” or “The method is too big” or even “This method is bloody huge” and therefore it was pretty easy to attribute these description to the related smell. However, in few cases, some description were interpreted differently by some authors. Therefore, we measured pairwise inter-rater reliability across the three sets of decisions and we clarified possible discrepancies and different classifications together, so as to have a 100% agreement among all the authors.

Considering the second study, for each code smell (CS_i), we computed:

1. The percentage of cases perceived as containing a design and/or implementation problem:

$$\frac{\#\text{cases with perceived design and/or implementation problems}}{\#\text{classes infected by } CS_i} \times 100$$

2. The percentage of correctly identified cases:

$$\frac{\#\text{CS}_i \text{ identified}}{\#\text{classes infected by CS}_i} \times 100$$

We consider as “identified”, answers where participants described the code smell affecting the code fragment when the problems description is clearly attributable to the code smell definition affecting the code fragment.

We assessed the reliability by means of Cronbach's α for both studies, to in ensure that the statements on the given scale measure the same underlying concepts [33]. Cronbach's α is commonly used as internal consistency measure. It varies between 0 and 1, where values closer to 1 indicate high internal consistency and values closer to 0 indicate low internal consistency. For all the research questions of the first study, Cronbach's α was calculated for the results related to the first four research questions, since RQ5 is a comparison among the previous ones. In the case of RQ3 and RQ4, which are computed for binary (e.g., true/false) items, Cronbach's α is identical to the Kuder–Richardson [34] formula of reliability for sum scales. For all RQs, we obtained good internal consistency, with a Cronbach's α ranging from 0.87 to 0.93:

- RQ1: 0.90
- RQ2: 0.93
- RQ3: 0.87
- RQ4: 0.89

4. Results and discussion

This section presents the results of the studies. The replication package, containing all raw data of this study is publicly available [35].

4.1. First study

For the first study, we distributed more than 250 paper-based questionnaires, collecting 71 completed ones and 63 containing valid answers. Moreover, 39 participants also indicated their availability to fill in the second questionnaire by providing their email address.

4.1.1. Background information

The respondents were working mainly developers (55.56%), project managers (11.11%), and software architects (11.11%) . 58.73% of them had more than 8 years of experience, while the remaining 19.05% had more than 4 years. This is also confirmed by their age, ranging from 23 to 63, with an average of 39.64 years and a median of 36.5, and their very good knowledge of object-oriented development languages, as reported in Table 2. The respondents worked mostly in large enterprises (41.27%), followed by SMEs (33.30%) and micro-enterprises (25.40%), while the organization domain was mostly software development (79.37%). We would like to note that the participants from academia were senior developers working in the IT departments and not researchers.

4.1.2. Knowledge of smells

At this point we investigated whether the participants were familiar with code smells and whether they already considered the removal of code smells in their development process. This was an

Table 2
Programming language skills.

Experience level	Percentage (%)					
	Java	C/C++	C#	VB	Python	Javascript
Novice	3.17	15.87	19.05	25.40	25.40	6.35
Beginner	9.52	11.11	7.94	11.11	28.57	11.11
Competent	19.05	25.40	14.29	15.87	14.29	26.98
Proficient	22.22	11.11	12.70	4.76	1.59	30.16
Expert	25.40	4.76	4.76	4.76	3.17	0

Table 3
Knowledge of code smells.

Knowledge of code smells	Percentage (%)
Never heard of them	1.69
Heard about them but not sure what they are	5.08
General understanding but do not apply the concepts	15.25
Good understanding and apply the concepts	37.29
Strong understanding and apply concepts frequently	42.37

Table 4
Information sources on smells.

Information sources	Percentage (%)
Blog	71.43
Books	57.14
Websites of gurus	44.44
Internet forum	39.68
Research papers	26.98
Tool vendors' websites	14.29

important step needed to understand if they had experience with smells. We first asked how the participants evaluated their own code quality, offering three possibilities (and one option "Other"): based on their own experience, on code inspection, on automated code analysis. We allowed multiple answers to this question. The results show that 66.67% of the respondents mainly evaluate the code quality based on their experience, followed by code inspection (55.56%) and automated analysis (44.44%).

The frequent use of such practices by the respondents confirmed our assumption that the participating developers were highly experienced and also trained in software quality assessment.

The majority of the respondents already knew the concept of code smells (Table 3), with 42.37% having a strong understanding and applying their concept frequently, 37.29% having a good understanding and applying them, 15.25% generally understanding but not applying the concepts, and the remaining ones only having fairly low knowledge or not knowing about smells at all.

Participants learn about code smells from different sources of information. Table 4 shows the sources of information reported by our respondents. Please note that multiple answers were allowed for this question. Blogs and books were the most frequent answers among the respondents (71.43% and 57.14%, respectively). Also, websites of gurus and Internet forums were frequently chosen by the respondents (44.44% and 39.68%, respectively). The least frequently mentioned sources were tool vendors' websites (14.29%) and research papers (26.98%).

Then we asked whether the participants use smells detection tools integrated into a continuous building environment or standalone tools. We allowed multiple answers to this question. Therefore, the sum of the percentages can be higher than 100%. 45 respondents out 63 used tools to detect smells. The most frequently used tool integrated into a continuous building environment was Sonarqube (62%), while Findbugs[9] (42%) was the most frequently used standalone tool. The results are showed in Table 5.

[9] http://findbugs.sourceforge.net/.

Table 5
Tools used to evaluate the code quality.

Tools used	Percentage %	
	Stand alone	Integrated
Checkstyle [10]	33	49
Pmd[11]	31	38
Findbugs	42	27
Sonarqube	24	62

Table 6
Usefulness of smells for various activities.

Code smell analysis/tools	Usefulness (Percentage %)				
	None	Low	Average	High	Essential
Refactoring	4.55	13.64	15.91	22.73	34.09
Quality assessment	15.91	18.18	13.64	20.45	25.00
Bug prediction	2.27	15.91	34.09	29.55	13.64
Effort prediction	9.09	20.45	27.27	20.45	15.91
Code inspection	9.09	6.82	22.73	29.55	22.73

Table 7
Code smell removal.

Code smell removal	Percentage (%)
Combination (refactoring sessions and immediately)	47.62
During refactoring sessions	7.94
Immediately, when I discover them	36.51

Table 8
Frequency of refactoring meetings reported by the participants.

Frequency of refactoring	Percentage (%)
Never	0
Almost never	0
Sometimes	21.82
On regular basis	49.09
Refactoring included as activity	29.09

Considering participants who already used code smells related tools, next we analyzed the usefulness of the code smell detection tools and examined in which phase of the development process these tools are used (Table 6). More than 50% of the respondents considered tools as highly important or essential during refactoring activities.

A similar behavior is also reported for the usefulness of code smells. More than 60% considered smells of average usefulness or highly useful for error prediction, while nearly 50% considered it helpful for effort prediction. More than 75% of the respondents considered code smells more useful than average for code inspection support.

Nearly half of the respondents claimed that they remove code smells: they remove the most harmful ones during development and during dedicated and planned refactoring sessions. 36.51% remove code smells only during development and the remaining 7.94% only during planned refactoring sessions. The results are shown in Table 7.

In Table 8, we report on how often the participants plan refactoring sessions to remove smells. We got 55 answers out 63. The majority of the respondents reported refactoring sessions on a regular basis (49.09%), while more than one third (29.09%) include the refactoring activity in their daily process. Refactoring is mainly assisted by tools (Table 9), and as the complexity of the refactor-

[10] http://checkstyle.sourceforge.net/.

[11] http://pmd.github.io/.

Table 9
Code smell refactoring types.

Refactoring type	Level of occurrence (Percentage)		
	Seldom	Regularly	Often
Manual refactoring	12.70	28.57	34.92
Tool-assisted	33.33	23.81	28.57
Combined	22.22	30.16	31.75

Table 10
Frequency of code smell refactoring.

Refactoring complexity	Level of occurrence (Percentage)		
	Seldom	Regularly	Often
Low complexity	7.94	23.81	57.14
Medium complexity	12.70	55.56	19.05
High complexity	46.03	28.57	14.29

Table 11
Level of concernedness of smells.

Level of concernedness	Percentage (%)
Not concerned at all	0
Not really concerned	19.05
Concerned	50.79
Very concerned	30.16

Table 12
Perceived harmfulness of smells based on their description.

Code smell	Valid answers	Perceived harmfulness Mean
Refused Bequest	36	4.63
Orphan Variable of Constant Class	40	4.38
God Class	42	4.20
Blob	42	4.17
Duplicated Code	42	4.17
Class Data Should Be Private	42	4.00
Large Class	42	3.98
Speculative Generality	42	3.96
Tradition Breaker	40	3.96
Long Method	40	3.85
Inappropriate Intimacy	40	3.75
Swiss Army Knife	42	3.75
Excessive Use of Literals	41	3.59
Excessively Short Identifiers	40	3.54
Excessive Return of Data	39	3.53
Downcasting	41	3.48
Feature Envy	41	3.39
Functional Decomposition	41	3.25
Too Many Parameters	40	3.19
Cyclomatic Complexity	41	3.18
Spaghetti Code	41	3.13
Excessively Long Identifiers	40	2.81
Lazy Class/Free Loader	41	2.39

ing activity increases, the frequency of the refactoring decreases (Table 10).

4.1.3. Concernedness of smells

In this section, we report the results of the study about the general attitudes of our participants towards smells. 50.79% of the participants were concerned about the harmfulness of smells while the remaining respondents were divided between very concerned (30.16%) and not really concerned (19.05%). No participants were not concerned at all (Table 11). This confirms the level of expertise of the participants involved in the questionnaire, and supports the validity of the results of the results of the next question

The results for the question "Please motivate why you are/are not concerned" were extracted by two researchers through open and selective coding [32]. We collected 36 valid answers. 27 respondents proposed one motivation, 8 proposed two motivations, and one respondent proposed three motivations. We extracted 12 similar motivations that were grouped into four high-level abstraction categories. Moreover, all the participants were concerned about product maintainability, and the vast majority proposed some maintainability motivations that can be perfectly matched to the maintenance characteristics of ISO/IEC 25010:2011 [36].

These four categories can be considered as motivations for the perceived harmfulness of smells and can be characterized as follows:

- *Code quality*: This category is related to the internal product quality. 13.75% of the respondents perceived code smells as harmful from the quality point of view, without addressing a specific quality focus.
- *Maintainability*: All the participants related smells with long-term maintenance issues, while 95% of the participants, besides reporting maintainability issues, also specified the following maintainability sub-characteristics:
  - *Potential cause of bugs*: 5% of the participants were aware that the presence of code smells could increase the likelihood of a future bug.
  - *Modifiability*: 7.85% of the respondents were concerned about poor code modifiability if the code is infected with code smells. This includes codes such as "code unmodifiable", "unstable codebase", or "code hard to modify".
  - *Analyzability*: 9.16% of the respondents related the presence of code smells to analyzability issues. This also includes participants who related them to issues of "code understandability".
  - *Modularity*: 18.33% of the respondent were concerned about reduced modularity in the case of code smells.
  - *Reusability*: 55% of the participants reported that infected code is less reusable, especially in the case of duplicated code.
- *Effort/Cost*: This category includes cost and effort drivers, such as increased effort needed in the case of refactoring activities that involve the refactoring of infected code (7.85% of the participants). However, all the respondents who proposed these factors also reported the effort needed to remove code smells as worthwhile from a long-term perspective.
- *Self-improvement/Personal motivations*: This category includes participants (18.33%) who differentiate themselves from non-skilled developers or who improve their personal satisfaction when they are able to decrease the number of code smells and reduce the technical debt.

4.1.4. Perceived harmfulness of smells

The participants rated how concerned they were with respect to the smells listed in Table 1, based on a 5-point Likert scale, where 1 means "not harmful" and 5 "very harmful". The results in Table 12 report the average weight for the answers (column "Perceived harmfulness"), the frequency of occurrence of each value of the Likert scale, the percentage of respondents who selected the same value of the Likert scale (column %), and the number of respondents (column #).

Only four code smells (Class Data Should Be Private, Orphan Variable of Constant Class, Excessively Long Identifiers, Lazy Class/Free Loader) were not considered harmful by the vast majority of the respondents. The remaining ones were considered harmful or very harmful, based on the description. In detail, the most harmful smells are God Class, Spaghetti Code, and Duplicated Code, which were considered harmful by more than 90% of the respondents, and Refused Bequest and Swiss Army Knife, considered harmful by more than 80%.

Table 13
Programming language skills.

Experience level	Percentage (%)					
	Java	C/C++	C#	VB	Python	Javascript
Novice	4.88	41.46	17.07	56.10	46.34	14.63
Beginner	4.88	12.20	43.90	121.95	34.15	21.95
Competent	12.20	39.02	19.51	7.32	12.20	39.02
Proficient	39.02	4.88	14.63	7.32	2.44	24.39
Expert	39.02	2.44	4.88	7.32	4.88	0

4.2. Second study

For the second study, we collected 41 valid answers. 29 respondents filled out the previous questionnaire and another three respondents declared having participated in the first study, but did not provide their email address in the first or in the second study; therefore, we were not able to map them automatically in both studies.

4.2.1. Background information

The participants had similar and comparable profiles as in the first study. The respondents were highly experienced. The vast majority had an experience level between 4 and 8 years (48.78%) or more than 8 years (14.63%). They were working as developers (63.41%), agile coaches (17.07%), and project managers (12.19%).

Moreover, as reported in Table 13, the respondents had very good knowledge of object-oriented development languages. The respondents worked mostly in large enterprises (73.17%), followed by SMEs (14.63%) and micro-enterprises (12.19%), while the organization domain was mainly software development (97.56%).

4.2.2. Perceived harmfulness of code infected with smells

In this section, we aim at answering RQ2. Out of 41 valid questionnaires, participants provided their opinions on 30 code fragments containing 23 pieces of code infected with smells. The respondents provided their opinion on an average of 28.6 code fragments per questionnaire. 67% of code fragments containing code smells were perceived as harmful. However, 8% of the not infected code fragments presented were also perceived as harmful.

Nine classes infected with code smells were nearly always perceived as harmful, namely God Class, Large Class, Lazy Class / Free Loader, Long Method, Spaghetti Code, Too Many Parameters, Cyclomatic Complexity, Class Data, Should Be Private, and Refused Bequest. Duplicated code and Blob were considered harmful in more than 75% of the code fragments. Moreover, nine other code smells were not considered harmful so often (from 64% to 26%). It is important to notice that Downcasting, Excessive Return of Data and Tradition Breaker were never perceived as harmful.

The results are presented in Table 14 where column “Valid answers” reports the number of respondent for each class infected with a specific code smell. Column “Perceived as harmful” presents the results of the question “In your opinion, does this code component exhibit any design and/or implementation problem?” reporting both the number of respondents and the percentages, while column “Perceived harmfulness” reports the rating of the severity of the problem identified by participants, by means of a 5-points Likert scale.

4.2.3. Ability to identify and describe the correct problem in a piece of code

In this section, we report the results related to RQ3. Considering the code fragments perceived as harmful, 76% were identified and described correctly. Class Data Should Be Private, Duplicated Code, Large Class, Long Method, and Too Many Parameters were always identified and described correctly. Another five code smells were identified and described correctly by more than 75% (Cyclomatic Complexity, Excessively Short Identifiers, Excessively Long Identifiers, Excessive Use of Literals, Feature Envy, and God Class. The remaining ones were described correctly by more than 41%.

The results are presented in Table 18 where column “Problem identified” reports the number of respondent that correctly identified the code smell present in the code fragment.

4.2.4. Ability to name the code smell identified in a piece of code with the term used in the literature

In this section, we report the results regarding RQ4. Out of the code fragments correctly described as the related code smells, 72% were named based on the terms used in the literature (Table 1, column “CS named”).

4.2.5. Comparison of perceived harmfulness of code infected with code smells and perceived harmfulness from the description

In this section, we analyze the answers provided by the participants who answered both questionnaires, to answer RQ5. We analyzed the results of the 32 participants who participated in both studies.

When analyzing the perceived harmfulness of code infected with code smells, the harmfulness perceived by the participants decreased dramatically compared to what they perceived when reading the description of the smells. Only five smells were similarly perceived as harmful in both the description and in the source code, with a perceived difference lower than 10%. In only three cases did they perceive the code as more harmful than in the description, while in 15 cases they perceived it as more harmful. It is important to note that three of them were completely ignored in the code, whereas the description triggered the alertness of the participants (Downcasting, Excessive Return of Data, and Tradition Breaker).

Table 15, reports the perceived harmfulness of each code smell obtained in the first and in the second study. The column “Perceived from text” reports the perceived harmfulness (from 1 to 5) reported by reading the description (first study), the column “Perceived from code” reports the perceived harmfulness obtained after the analysis of the source code (second study), the column Δ% reports the difference in the perceived harmfulness. The results are ordered by Δ%. As an example, Downcasting was considered harmful (”Perceived from code” = 3.48 out of 5) when reading the description, while it was never detected or never considered as harmful from the text (”Perceived from code”=0). Therefore, the Δ% is calculated as a decrease of 100%. Please note that the results may slightly vary from those reported in Table 12 and Table 14 since the comparison was carried out only among those participants who participated in both studies. smells were generally perceived as less harmful when the infected source code was analyzed (17 cases out of 23).

5. Discussion

In this section we discuss how the results of the two studies differ from the studies they replicate. Then, we discuss the overall results and we report implications for tool providers, developers, and researchers.

5.1. Discussion of the results of the first study

Our study was conducted three years later than the study by Yamashita and Moonen [8], and while the respondents (their origin, role, and experience) between the two studies are comparable, their answers differ: while in [8] 23% have never heard of code smells and 19% have heard about them but are not sure what they are, in our sample these numbers are much lower, respectively 2%

Table 14
Results of the research questions RQ2, RQ3, and RQ4.

Code smell	Valid answers	Perceived as harmful		Perceived harmfulness	Problem identified		CS named		Others	
	#	#	%	(1–5)	#	%	#	%	#	%
Blob	34	25	74	2.00	23	68	7	21	2	6
Class Data Should Be Private	28	26	93	2.50	26	93	26	93	0	0
Cyclomatic Complexity	36	34	94	4.00	33	92	20	56	1	3
Downcasting	41	0	0	0	0	0	0	0		0
Duplicated Code	24	18	75	3.00	18	75	18	75	0	0
Excessive Return of Data	24	0	0	0	0	0	0	0	0	0
Eccessive Use of Literals	28	18	64	1.00	15	53	2	7	3	11
Excessively Long Identifiers	32	19	59	1.00	17	53	11	34	2	6
Excessively Short Identifiers	32	19	59	2.00	14	43	11	34	5	16
Feature Envy	32	17	53	3.00	15	47	2	6	2	6
Functional Decomposition	28	17	61	3.00	14	50	7	25	3	11
God Class	20	20	100	4.00	16	80	16	80	4	20
Inappropriate Intimacy	20	8	40	3.00	6	30	2	10	2	10
Large Class	34	34	100	4.50	34	100	34	100	0	0
Lazy Class/Free Loader	24	24	100	2.50	21	87	7	29	3	13
Long Method	28	28	100	2.50	28	100	28	100	0	0
Orphan Variable of Constant Class	23	6	26	1.50	6	26	3	13	0	0
Refused Bequest	28	26	93	3.50	24	86	8	28	2	7
Spaghetti Code	20	20	100	3.00	17	85	6	30	3	15
Speculative Generality	36	17	47	3.00	12	33	3	8	5	14
Swiss Army Knife	36	23	64	4.00	14	39	2	6	9	25
Too Many Parameters	28	28	100	4.00	28	100	28	100	0	0
Tradition Breaker	16	0	0	0	0	0	0	0	0	0
Class without CS/AP	28	3	11	1.50	0	0	0	0	3	11

Table 15
Comparison between perceived harmfulness of smells based on their description and source code analysis.

Code smell	Valid	Perceived harmfulness		Δ%
	answers	from text	from code	
Blob	9	4.17	1.56	−62.67
Class Data Should Be Private	15	4.00	2.53	−36.67
Cyclomatic Complexity	24	3.18	3.96	24.59
Downcasting	0	3.48	0	−100.00
Duplicated Code	12	4.17	3.08	−26.00
Excessive Return of Data	0	3.53	0	−100.00
Excessive Use of Literals	0	3.59	1.13	−68.70
Excessively Long Identifiers	12	2.81	1.17	−58.52
Excessively Short Identifiers	12	3.54	1.50	−57.65
Feature Envy	7	3.39	2.71	−20.00
Functional Decomposition	5	3.25	2.80	−13.85
God Class	11	4.20	4.09	−2.70
Inappropriate Intimacy	3	3.75	3.67	−2.22
Large Class	22	3.98	4.86	22.29
Lazy Class/Free Loader	11	2.39	2.55	6.67
Long Method	17	3.82	2.53	−33.85
Orphan Variable of Constant Class	2	4.38	1.50	−65.71
Refused Bequest	10	4.63	3.40	−26.49
Spaghetti Code	6	3.13	3.17	1.33
Speculative Generality	6	3.96	2.83	−28.42
Swiss Army Knife	8	3.75	4.13	10.00
Too Many Parameters	20	3.19	3.75	17.65
Tradition Breaker	10	3.96	0	−100.00

and 5%. This means that in the time between the two papers the community of software developers became much more aware of what code smells are and that they apply the concepts frequently in their work (3% in [8], around 42% in our study).

Comparing how concerned developers are about code smells, we note that the levels increased (in [8] only 26% where moderately or extremely concerned, in our study around 81%).

The sources of information on code smells are comparable in the two studies, we note while blogs are still the most used source of information, books are now ranked second instead of Internet forums (as in [8]). This might be another indicator that the discussion about code smells is now more stable and settled.

Comparing the motivations for the perceived harmfulness of smells, the mentioned motivations are comparable, we note that in our study developers did not mention that they are afraid to lack the necessary skills or to lack organizational support to remove or avoid code smells. Again, this seems to indicate that developers gained more experience on code smells over the last three years.

5.2. Discussion of the results of the second study

Comparing our results to the results of Palomba et al. [25], we observe large differences: Besides involving more participants, in

Table 16
Second study comparison with Palomba et al.[25].

Code smell	Perceived as harmful				Identified			
	[25]		Our study		[25]		Our study	
	#	%	#	%	#	%	#	%
Class Data Should Be Private	1	11%	26	93%	1	11%	26	93%
Cyclomatic Complexity	9	100%	34	94%	9	100%	33	92%
Feature Envy	6	68%	17	53%	3	33%	13	41%
God Class	9	100%	20	100%	9	100%	16	80%
Inappropriate Intimacy	4	44%	8	40%	3	33%	4	20%
Long Method	6	66%	28	100%	6	66%	28	100%
Refused Bequest	1	11%	26	93%	0	0%	16	57%
Spaghetti Code	6	66%	20	100%	6	66%	11	55%
Speculative Generality	4	44%	17	47%	3	33%	9	25%

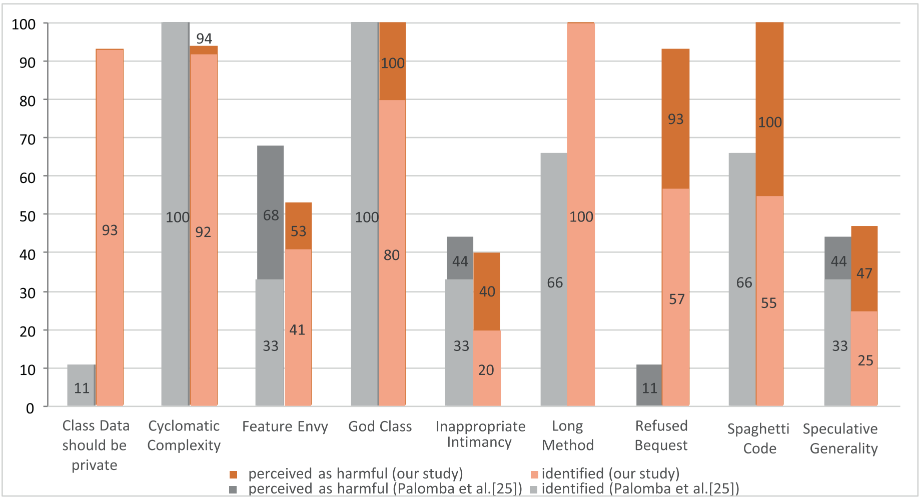

Fig. 2. Second study comparison with Palomba et al. [25]. (Values are in percent).

our study, on average, code smells were rated as harmful by a higher percentage of participants than in their study. We compare the results between the perceived harmfulness and the ability to identify code smells in the source code in Table 16 and Fig. 2.

For some smells the results are comparable, for others, like Class Data Should Be Private, Long Method, Refused Bequest, and Spaghetti Code, there are large differences, which we attribute to the aspect that we involved more professional developers in our study, which can be also seen by the number of responses listed in the table (column #).

5.3. Discussion of the overall results

Based on the overall results, we can summarize the following lessons learned:

1. *Developers are well aware of the definitions and names of code smells present in the literature.* 72% of the code smells were correctly identified and named by the developers.
2. *Some smells are considered important in theory but not practically perceived as a design problem.* 17 out of 23 smells were considered harmful in theory while code containing these issues was not considered harmful at all. This happened especially for Downcasting , Excessive Return of Data, and Tradition Breaker where, besides considering them slightly harmful in theory, developers never perceived code infected with them as harmful. This can be also explained in part because in our study we only looked at code fragments and therefore some anti-patterns would have required to analyze the entire system to provide the real perception of their harmfulness.
3. *Some smells are considered as not harmful in theory but practically perceived as a design problem.* Lazy Class/Free Loader, Swiss Army Knife, Too Many Parameters, Large Class, and Cyclomatic Complexity were considered from 10% to 25% more important in practice. This could be due to an underestimation of the issue in theory. Based on our first study, developers reported that a large class is not a big issue since they can read it all, while facing a large class they were not able to clearly understand it.
4. *Smells related to size and complexity are considered harmful by a higher percentage of participants than others.* Too Many Parameters, Large Class, Long Method, Large Class, and Cyclomatic Complexity were always perceived and identified as harmful or very harmful by a higher percentage of participants than for other smells. This could be confirmed simply by how easy it was to identify these issues or by how complex it was to read a very long piece of code, long or complex methods. This result is consistent with [25].

While we notice that code smells became more known in the past three years, it seems that the perception of their harmfulness is still based on anecdotal evidence and not on empirical research on the practitioners' experience. Moreover, the existing code smell detection tools do not differentiate the various smells according to their impact, based on scientific literature. Also, it is confusing that tools like SonarQube adopt the term “code smell” also for coding style violations, e.g., when a class name begins with a lower case letter. Therefore, we recommend tool providers to refer to the code smell definitions present in the literature and to balance their recommendations based on the expected harmfulness.

In this context, educators should know the various definitions used in the field and increase the awareness of students on misleading code smell definitions adopted by various tool vendors.

In this study we focus on the *perceived* harmfulness of code smells based on short code snippets. A perceived harmfulness is not always really harmful. As we summarized in the related literature, code smells are sometimes good indicators of a problem, sometimes not. Sometimes they indicate that the programmers should refactor the code, sometimes it would be better to leave it as it is. Programmers have to become more aware about the specific context in which it is advised to refactor or not. Therefore, in the future, researchers should study in more detail which consequences code smells really have (and in which specific context) so that developers can base their assumptions on harmfulness on empirical evidence.

According to our results, based on the description, all smells were perceived as harmful; the vast majority of them were perceived as very harmful. However, when a piece of code infected with the previous code smell was presented without the related issue being reported, only 67% of the participants perceived the code as containing harmful issues. Moreover, only 41% were able to describe the related code smells and only 29% were able to correctly name them. Therefore, we can see a gap between the theoretical knowledge about code smells and the practical application of their identification in code and the classification of their harmfulness. This is an additional aspect that should encourage researchers to study the real harmfulness of code smells in industrial contexts.

6. Threats to validity

In this section, we discuss the threats to validity, including internal, external, construct, and conclusion validity, and the different tactics [30,37] we adopted to mitigate them.

Internal validity refers to the factors that may have influenced our study. Considering the respondents, they were senior developers with experience in code quality. However, we are aware that the results could be biased by the selection of participants belonging to a set of developers trained in agile technologies and more inclined to use quality assessment tools.

Construct validity is about the correct identification of measures adopted in the measurement procedure (e.g., the questionnaires we used in the studies). To confirm that questions correspond to the actual cause we are interested in analyzing, we first analyzed the mapping of our research questions and the questions in the questionnaires used in previous studies. Then we checked each question to avoid potential misunderstandings, negative questions, and threats due to previously answered question bias. The perceived harmfulness was obtained by asking the participants to describe possible problems they perceived to understand if their perception is related to the identified code smells. Concerning the harmfulness of each code smell, we asked the participants to rate the severity of the problem by means of a Likert scale, to allow us to compare the responses based on an homogeneous scale. Concerning the set of tasks including code smells, we adopted the set of code identified initially by Palomba et al. [25]. To reduce this threat, we checked the correctness of the identification both manually and by means of automated tools. We are aware that we considered code smells as binary attributes, considering their presence in code or not while, in some cases their presence should be considered relative to the size of the entire code base and to other aspects of the system. As example, a class infected by a “Swiss-Army Knife” smell, that couples 75% of the classes of a small project (i.e. a 1000 lines of code) could be very different than another that couples 25% of the classes of a huge project (i.e. 10 millions of lines of code).

The identification of the remaining issues not considered by Palomba et al. [25] was carried out by the same researchers and using the same tools. Since we limited the analysis to only one fragment per code smell as in [25], might have led us to exclude classes where the problem could have been more or less evident. Moreover, this kind of study including highly experienced practitioners has strict time constraints since it is not possible to involve practitioners in long tasks.

External validity are related to the generalization of our findings. It can be concerned to the subjects of the study and the selected objects. To mitigate this threat, we adopted a treatment set of classes from Palomba et al. [25], extending them to cover the smells missing in their study. Moreover, all the proposed code to be analyzed belonged to well-known real open-source projects. Moreover, we are aware that further studies with different analyzed projects are needed to confirm our results. In this study, we covered the most comprehensive set of smells proposed in the literature.

Conclusion validity focuses on how sure we can be that the tasks we adopted are related to the actual outcome we observed [38]. To mitigate this threat, the questionnaires were checked by three experts on empirical studies. Moreover, it was ensured that the subjects of both groups had similar backgrounds and knowledge regarding software development.

However, we are aware that the perceived harmfulness of the code smells can be biased to the development environment since some smells are much less likely in some environments than others and some smells are more problematic in some environments.

7. Conclusions and future work

In this work, we presented an empirical study aimed at analyzing developers’ perceived harmfulness of code smells. For this purpose, we extended two previous studies ([8] and [25]) by considering set of code smells proposed in [4] and comparing the developers’ perception based on the description of the smells, their ability to identify and name them, and the perceived harmfulness of the infected source code. The study regarded the analysis of 23 smells involving two sub-studies with 63 and 41 senior developers as participants.

The results confirm that smells are generally perceived differently between theory and practice and that there is still a lot of misunderstanding surrounding code smells. This study confirms and extends the results obtained in [8] and [25] and allows comparing and extending them with the most recent body of knowledge.

The study performed for this paper is a momentary snapshot of the perceptions of developers towards smells. Such perceptions are changing over time and therefore we recommend that researchers should regularly investigate the current perceptions of developers since developers will act based on their perceptions and not based on the real consequences of injecting or removing code smells.

Further replication studies, with different settings, could extend the collected results. A more thorough investigation of the usage of code smells in practice and an analysis of the actual usefulness of their adoption are needed to provide input for researchers and practitioners regarding the development of new detection approaches for the identification or extension of existing code smells to more useful ones.

References

[1] Royal Academy of Engineering and British Computer Society, The Challenges of Complex IT Projects: The Report of a Working Group from the Royal Academy of Engineering and the British Computer Society, The Royal Academy of Engineering, 2004. Online: http://www.bcs.org/upload/pdf/complexity.pdf, accessed December 1st, 2016

[2] T. Hampp, A cost-benefit model for software quality assurance activities, in: Proceedings of the 8th International Conference on Predictive Models in Software Engineering, in: PROMISE '12, ACM, New York, NY, USA, 2012.
[3] M. Diaz-Ley, F. Garcia, M. Piattini, Implementing a software measurement program in small and medium enterprises: a suitable framework, IET Software 2 (5) (2008).
[4] J.B. Martin Fowler Kent Beck, Kent Beck, Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
[5] W.H. Brown, R.C. Malveau, H.W.S. McCormick, T.J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, 1st, John Wiley & Sons, Inc., New York, NY, USA, 1998.
[6] M. Kim, V. Sazawal, D. Notkin, G. Murphy, An empirical study of code clone genealogies, SIGSOFT Softw. Eng. Notes 30 (5) (2005).
[7] A. Yamashita, L. Moonen, Do code smells reflect important maintainability aspects? in: International Conference on Software Maintenance (ICSM), IEEE, 2012.
[8] A.F. Yamashita, L. Moonen, Do developers care about code smells? An exploratory survey., in: Proceedings of the 20th Working Conference on Reverse Engineering, IEEE Computer Society, 2013.
[9] I.S. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, M.J. Shepperd, A controlled experiment investigation of an object-oriented design heuristic for maintainability., J. Syst. Software 72 (2) (2004).
[10] R. Malhotra, A. Chug, P. Khosla, Prioritization of classes for refactoring: a step towards improvement in software quality, in: Proceedings of the Third International Symposium on Women in Computing and Informatics, in: WCI '15, ACM, New York, NY, USA, 2015.
[11] F.A. Fontana, V. Ferme, A. Marino, B. Walter, P. Martenka, Investigating the impact of code smells on system's quality: an empirical study on systems of different application domains, in: 2013 IEEE International Conference on Software Maintenance, 2014.
[12] D.I.K. Sjoberg, A. Yamashita, B. Anda, A. Mockus, T. Dyba, Quantifying the effect of code smells on maintenance effort, IEEE Trans. Softw. Eng. 39 (8) (2013) 1144–1156, doi:10.1109/TSE.2012.89.
[13] S.E.S. Taba, F. Khomh, Y. Zou, A.E. Hassan, M. Nagappan, Predicting bugs using antipatterns, in: Proceedings of the 2013 IEEE International Conference on Software Maintenance, in: ICSM '13, IEEE Computer Society, Washington, DC, USA, 2013.
[14] N. Zazworka, M.A. Shaw, F. Shull, C. Seaman, Investigating the impact of design debt on software quality, in: Proceedings of the 2Nd Workshop on Managing Technical Debt, in: MTD '11, ACM, New York, NY, USA, 2011.
[15] S.M. Olbrich, D.S. Cruzes, D.I.K. Sjoberg, 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 2010 IEEE International Conference on Software Maintenance, in: ICSM '10, IEEE Computer Society, Washington, DC, USA, 2010.
[16] P. Danphitsanuphan, T. Suwantada, Code smell detecting tool and code smell-structure bug relationship, 2012 Spring Congress on Engineering and Technology, 2012.
[17] W. Li, R. Shatnawi, An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution, J. Syst. Softw. 80 (7) (2007).
[18] T. Hall, M. Zhang, D. Bowes, Y. Sun, Some code smells have a significant but small effect on faults, ACM Trans. Softw. Eng. Methodol. 23 (4) (2014) 33:1–33:39, doi:10.1145/2629648.
[19] F. Khomh, M. Di Penta, Y.-G. Gueheneuc, An exploratory study of the impact of code smells on software change-proneness, in: Proceedings of the 2009 16th Working Conference on Reverse Engineering, in: WCRE '09, IEEE Computer Society, Washington, DC, USA, 2009.
[20] F. Khomh, Squad: Software quality understanding through the analysis of design, in: 2009 16th Working Conference on Reverse Engineering, 2009.
[21] S. Olbrich, D.S. Cruzes, V. Basili, N. Zazworka, The evolution and impact of code smells: A case study of two open source systems, in: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, in: ESEM '09, IEEE Computer Society, Washington, DC, USA, 2009.
[22] A. Yamashita, L. Moonen, Exploring the impact of inter-smell relations on software maintainability: an empirical study, in: Proceedings of the 2013 International Conference on Software Engineering, in: ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013.
[23] F.A. Fontana, P. Braione, M. Zanoni, Automatic detection of bad smells in code: an experimental assessment., J. Object Technol. 11 (2) (2012).
[24] N. Moha, Y.G. Gueheneuc, L. Duchien, A.F.L. Meur, Decor: a method for the specification and detection of code and design smells, IEEE Trans. Software Eng. 36 (1) (2010).
[25] F. Palomba, G. Bavota, M.D. Penta, R. Oliveto, A.D. Lucia, Do they really smell bad? a study on developers' perception of bad code smells, in: Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, in: ICSME '14, IEEE Computer Society, Washington, DC, USA, 2014.
[26] O.S. Gómez, N. Juristo, S. Vegas, Replications types in experimental disciplines, in: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, in: ESEM '10, ACM, New York, NY, USA, 2010.
[27] P. Runeson, Using students as experiment subjects—an analysis on graduate and freshmen student data, in: Proceedings 7th International Conference on Empirical Assessment & Evaluation in Software Engineering, 2003.
[28] B.F. Webster, Pitfalls of Object-Oriented Development., M & T, 1995.
[29] W. Fenske, S. Schulze, Code smells revisited: a variability perspective, in: Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems, in: VaMoS '15, ACM, New York, NY, USA, 2015.
[30] V.R. Basili, G. Caldiera, H.D. Rombach, The goal question metric approach, Encyclopedia of Software Engineering, Wiley, 1994.
[31] V. Basili, A. Trendowicz, M. Kowalczyk, J. Heidrich, C. Seaman, J. Münch, D. Rombach, Aligning Organizations Through Measurement: The GQM+Strategies Approach, Springer Publishing Company, Incorporated, 2014.
[32] A.L. Strauss, J. Corbin, Basics of Qualitative Research : Techniques and Procedures for Developing Grounded Theory, SAGE Publications, Los Angeles, London, New Delhi, 2008.
[33] J.M. Bland, D.G. Altman, Statistics notes: Cronbach's alpha, BMJ 314 (7080) (1997).
[34] L.J. Cronbach, Test 'reliability': its meaning and determination, Psychometrika 12 (1) (1947).
[35] V. Lenarduzzi, D. Taibi, A. Janes, How developers perceive code smells and antipatterns in source code: a replicated study - raw data, 2017, (https://data.mendeley.com/datasets/8n6k8dfw2f/1). doi: 10.17632/8n6k8dfw2f.1.
[36] ISO/IEC, ISO/IEC 25010 – Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and software quality models, Technical Report, ISO/IEC, 2010.
[37] R.K. Yin, Case Study research, Design and Methods, Applied Social research Methods Series, third ed., Sage Publications, 2009.
[38] C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: An Introduction, Kluver Academic Publishers, 2000.