Software Quality Journal
https://doi.org/10.1007/s11219-021-09567-w

A longitudinal exploratory study on code smells in server side web applications

Narjes Bessghaier[1] · Ali Ouni[1] · Mohamed Wiem Mkaouer[2]

Accepted: 6 July 2021

Abstract
Modern web applications have become one of the largest parts of the current software market over years, bringing cross-platform compatibility and data integration advantages that encouraged businesses to shift toward their adoption. Like any software application, code smells can be manifested as violations of implementation and design standards which could impact the maintainability, comprehensibility and performance of web applications. While there have been extensive studies on traditional code smells recently, little knowledge is available on code smells in web-based applications (web apps). As web applications are split into their client and server sides, we present in this study a first step in exploring the code smells diffuseness and effect on the server side of web applications. To this end, we conduct an exploratory study on a total of 430 releases from 10 long-lived open-source web-based applications on 12 common code smell types. We aim to better understand and gain insights into the diffuseness of code smells, their co-occurrences and effects on the change- and fault-proneness in server side code. Our study delivers several important findings. First, code smells are not equally diffused in web apps server side, among which smells related to complex, and large code components display high diffuseness and frequency rates. Second, the co-occurrence phenomenon is highly common, but the association degree between code smell pairs is weak. Code smells related to large size and high complexity exhibit a higher degree of co-occurrences. Third, smelly files are more likely to change than smell-free files, whereas not all smell types are likely to cause equal change sizes in the code base. Fourth, smelly files are more vulnerable to faults than smell-free files, and 86% of smelly files are more likely to manifest more faults than other files. Hence, developers should be aware of the existence of code smells in their web applications and consider detecting and refactoring them from their code bases, using appropriate tools.

✉ Ali Ouni
ali.ouni@etsmtl.ca

Narjes Bessghaier
narjes.bessghaier.1@ens.etsmtl.ca

Mohamed Wiem Mkaouer
mwmvse@rit.edu

[1] ETS Montreal, University of Quebec, QC, Montreal, Canada

[2] Rochester Institute of Technology, NY, Rochester, USA

Published online: 27 August 2021

1 Introduction

The growing importance and popularity of web applications (web apps) have increased in recent years. Such popularity led to an extremely congested web development market and strong competition that allured many developers to build quickly web apps while evolving continuously to meet the needs of users. As a consequence, such popularity has raised more concerns about the quality of the services provided. Web apps are characterized by their inherent heterogeneous nature in (1) target platforms as web apps are usually split into their client and server sides, and (2) formalisms as web apps are typically built with a mixture of programming and formatting languages. Such heterogeneity makes the evolution of web apps unique and different from traditional software Rossi et al. (2007); Kienle and Distante (2014).

As web technology demand rises, strong emphasis is put on identifying and enhancing portions of the code that may suffer from bad programming or design practices. Like any software, web applications can contain design flaws known as code smells Fowler (1999). Such code smells often lead to errors in execution, increased use of resources, and additional maintenance. Smelly code also increases the developer's inability to grasp the evolving design of the system quickly Fowler (1999); Ouni et al. (2016, 2015a, 2015b, 2017); Saidani et al. (2021); Hamdi et al. (2021). Usually, this pressure would affect the project being developed and lead to technical debts.

To investigate the effect of code smells, researchers have studied the size of code change that developers need to perform in smelly code than non-smelly code. A generally accepted assumption in many different programming languages entails that smelly files are more subject to change than others Khomh et al. (2009, 2012); Palomba et al. (2017, 2018b, 2019). Code smells are also considered to have a significant effect on code maintainability, vulnerabilities and potential failures. Bugs or faults typically lead to unpredictable actions of the program and are commonly introduced by a change in the source code or by external artifacts (*e.g.*, API) Rodríguez-Pérez et al. (2020). Tassey Planning (2002) estimated that fault fixing takes up to 80% of the program's overall cost. Researchers are, therefore, keen to find and reduce those deficiencies as much as possible. Other studies examined the lifespan of faults Saboury et al. (2017), the association between code smells and faults Muse et al. (2020) and the impact of code smells on the fault-proneness Palomba et al. (2018b); Khomh et al. (2012) in traditional object-oriented (OO) applications. Some of the results highlighted that 1) smelly files are more vulnerable to faults than other files, 2) traditional code smells tend to co-occur with faults, and 3) faults are more persistent in code experiencing smells.

Understanding and investigating code smells, in web apps, is challenging given the differences from traditional OO applications. Indeed, in web applications, code smells and faults could impact any of the tiers in the server side, leading to server crashes and eventual performance issues in the client side. To the best of our knowledge, little is known about the impact of code smells in web applications. Therefore, as the first step toward a thorough and comprehensive analysis of web applications' quality, we focus on the server side vulnerability to code smells by analyzing the code of one of the most popular web programming languages, PHP. C3.1: Server-side applications share several technological aspects with other traditional applications, while they differ in other aspects. On the one hand, server-side apps typically rely on web services available to a broader range of users compared to traditional applications that are single-users. On the other hand, server-side applications intensively consume networking, security, and authentication computing resources to establish server connections. Besides, server-side

applications are in permanent interaction with remote storage resources (databases, files), unlike traditional applications which often reside on the client machine. The intensive data processing in server-side applications might require more coding and maintenance efforts, motivating us to dig into these application's quality evaluation.

While our previous study Bessghaier et al. (2020) explored the diffuseness and impact of code smells in the server side, it is conducted on a limited set of projects, which hinders the scalability of our previous findings. Moreover, the co-occurrence of smells, and their corresponding impact on the code fault-proneness was not investigated. In this paper, we extend and build on top of our previous work Bessghaier et al. (2020). More specifically, we conduct an empirical study on 430 releases from 10 open source web applications, written in PHP, being the pioneering web programming language Statistics (2020). We aim at investigating the diffuseness and effect of code smells on server side source code maintainability in a qualitative and quantitative way. In particular, this extension consists of the following ways :

1. We extend our study by collecting a larger dataset that consists of 430 releases from 10 open source long lived web applications to gain better generalizability of our results.
2. We investigate the phenomenon of code smells co-occurrences. The identification of smell co-occurrence patterns can be useful to build practical co-occurrence-aware code smell detection tools and refactoring recommendation systems.
3. In addition to the analysis of the impact of code smells on the change-proneness performed in our previous research Bessghaier et al. (2020), we analyze the effect of code smells on the code change frequency. The aim is to measure how frequently smelly files are subject to updates compared to non-smelly files.
4. We investigate the impact of code smells on the fault-proneness to analyze whether smelly files are more prone to faults than smell-free files and how longitudinally, these files are prone to faults.
5. We provide a comprehensive replication package for future extensions and replications and foster research on code smells in web-based software systems Dataset (2020).

Our empirical study delivers several findings, (1) the majority of code smells are highly diffused throughout the studied web applications evolution. We have 10 code smells impacting more than 50% of the studied releases; (2) code smells frequently co-occur, *i.e.*, a file is likely to contain more than one instance of a smell. For example, the *High Method Complexity* smell often co-occurs with nine other code smells such as *Excessive Method length* and *Excessive Parameter List*; (3) consistent with prior research, we found that smelly files are likely to undergo more change size than other files. We also examined the impact of each type of code smells on the change-proneness, and found that each code smell has its impact depending on the application context; (4) faults are shown to persevere in smelly code rather than non-smelly code. After code smells introduction, six faults are estimated to be introduced in smelly files against only one fault before smells appear. A high rate of change- and fault-proneness may result in significant maintenance activities that could affect the system's efficiency.

The rest of the paper is organized as follows. The related literature is discussed in Sect. 2. Section 3 describes our empirical study design. Section 4 presents the analysis of our research questions, while we discuss the implications of our study in Sect. 5. We discuss the threats to validity of our study in Sect. 6. Finally, we conclude and discuss our future research directions in Sect. 7.

2 Related work

2.1 Code smells diffuseness and evolution

Unlike other OO software systems, there is little research in the field of code smells in web-based applications. There is, however, little knowledge of code smells in web applications. Rio and Abreu (2019) recently examined the propagation of six types of code smells in 4 web applications using an existing PHP code detection tool PHPMD[1]. The results show different rates of survival of each type of code smell across the systems being studied. Code smells that affect coupled software components are the most persistent, and their rate of removal is low. Palomba et al. (2018b) investigated the relationship between the diffuseness of code smells with the class size (LOC) in traditional Java OO software systems. Code smells belonging to components of large and complex code are known to be the most persistent. Each of these code smells represents poor coding practices that developers tend to perpetrate frequently. The authors also evaluated the effect of the characteristics of a program on the frequency of smells (*e.g.*, number of classes, number of methods, and lines of code LOC). The findings illustrate that code smells reflecting the complexity of code components such as *Long method* and *complex class* are often diffused in large systems.

Similar results are witnessed by Olbrich et al. (2009) when analyzing the evolution of *God Class*, and *Shotgun Surgery* code smells in two open-source projects. The findings underlined how code smells diffuseness rate is not constant, suggesting multiple introductions and removals. Tufano et al. (2015) have pointed out that poor coding practices are often performed without the developers' knowledge. Chatzigeorgiou and Manakos (2010) investigated the diffusion of *Long Method*, *Feature Envy*, and *State Checking* code smells in 24 releases of two Java projects (JFlex, and JFreeChart). Findings indicate that the number of *Long Method* code smells instances increases as the system evolves. Unlike the *Feature Envy* and *State Checking*, which had the same occurrence rate during the period under study.

2.2 Code smells co-occurrence

Other studies have investigated the co-occurrence of code smells in OO software systems. Palomba et al. (2017) adopted association rule mining technique on 13 smells types to identify common co-occurring pairs. They found six (6) pairs of code smells that tend to co-occur within the same code component. Recently, Palomba et al. (2018a) complemented their previous research by providing insights into a large-scale analysis on how co-occurring pairs reside in the system, and how developers manage to eliminate co-existing pairs. They tracked pairs of smells that lived in at least 10% of classes along with the system evolution. For example, *Long Method* also affects 38% of classes affected by Spaghetti code. Similarly, Garg et al. (2016) measured the percentages of instances of smells that occur in a class, finding that traditional smells such as *Feature Envy* and *Data clumps* coexist. Moreover, Fontana et al. (2015) relied on the percentages of the same class smell types to identify the most common pairs. More recently, Muse et al. (2020) also investigated the coexisting phenomenon between SQL and code smells in Java data-intensive programs. The authors applied the Apriori rule-mining algorithm to produce the pairs of frequent

[1] https://phpmd.org/

smells that portray an SQL and a traditional code smell. The results stressed that SQL and traditional smells can co-occur within the system but are weakly related, which decreases the probability of a causality relationship.

2.3 Impact of code smells on change-proneness

Olbrich et al. (2010) performed an exploratory investigation into the effect of the *God* and *Brain* smells, on the change-proneness and on the code churns of smelly and non-smelly files. A change frequency metric was developed to measure the number of commits in which the code was updated by developers. The findings showed that smelly files are more susceptible to regular code changes and are more likely to experience more change size. An additional study standardizing the size of *God* and *Brain* classes demonstrated that those two classes are less subject to changes. An empirical analysis was conducted by Khomh et al. (2009) on the impact of 9 code smells on the change-proneness of 13 Azureus and Eclipse releases. The results showed that smelly classes are subject to regular code changes (averaging 8 times for smelly files). The authors also investigated the effect of different types of smells on the change-proneness, and found that the more instances of smell a class has, the more it is exposed to changes. Different forms of code smells may also lead to more changes than others. Later, Khomh et al. (2012) examined the effect of code smells on the change-proneness of 54 releases of four projects. The findings support previous findings that classes encountering code smells are more susceptible to code changes Khomh et al. (2012). These findings were also confirmed recently by Spadini et al. (2018), who found that further code changes result from the existence of test smells, which may lead to faults in the production code.

2.4 Impact of code smells on fault-proneness

The fault-proneness reflects the extent to which a class is susceptible to become faulty over time. The ongoing changes in a system pertaining to different maintenance activities may increase the software complexity, which could create difficulties for developers and practitioners to understand the evolving design. As a result, the system may become more vulnerable to faults. A fault, bug, defect, or error represents a code deficiency leading to an erroneous behavior. According to IEEE Group et al. (2010), a defect represents *"an imperfection or deficiency in a work product where that work product does not meet its requirements or specifications and needs to be either repaired or replaced"*. As suggested by Rodríguez-Pérez et al. (2020), some of these faults may be the reason for miscommunication between team members, incorrect requirements, or unexplained new changes.

There have been many studies focusing on analyzing how faults are introduced into the system. Most works assume that a fault is typically introduced due to non-appropriate code changes that are generally recorded in the version control systems (VCS). Saboury et al. (2017); Spadini et al. (2018); Palomba et al. (2018b, 2019, 2018a). Specifically, Śliwerski et al. (2005) built their fault-tracking algorithm, SZZ, on the basis of an approach claiming that the last change (*i.e.*, the first change per order of introduction in the code) that touched the fixed code is responsible for the introduction of the fault. In this case, the last change applies to the *Fault-Inducing Commit* (FIC), in which the fault was first added. If a developer fixes a fault, the *Fault-Fixing Commit* (FFC) will trigger a change in the faulty code called the fixing change. Any fixing change is attached to a descriptive commit message expressing *"a change is made to fix the fault#ID"*. All the commits between FIC and FFC

are holders of a particular fault and are known as the *Commits Exhibiting a Fault* (CEF) Rodríguez-Pérez et al. (2020).

Since SZZ is regarded as the de-facto standard for faults localization, many revisited algorithms have been proposed since the SZZ was established Kim et al. (2006); Williams and Spacco (2008); Borg et al. (2019); Lenarduzzi et al. (2020); and several studies have used these SZZ revisions to detect faults and analyze the effect of code smells on fault occurrences Zimmermann et al. (2007); Da Costa et al. (2016); Rodríguez-Pérez et al. (2020); Saboury et al. (2017); Muse et al. (2020); Palomba et al. (2018b). Saboury et al. (2017) examined the effect of 12 JavaScript smells on the fault-proneness. The aim is to use the survival analysis method to capture a longitudinal pattern of faults survivability in the studied systems. The obtained results showed that non-smelly files are less likely to spread errors by 65%. The study also captured a quantitative effect on the faults of some system's characteristics like the *LOC, code churn, and the number of Previous Bugs*. The number of previous bugs that reflect the number of fault-fixing change trials (before the FFC) could endanger the system to more faults and increase their rate of growth. Muse et al. (2020), examined the correlation between traditional code and SQL smells with faults in Java programs. The study indicates that faults are not statistically proven to be correlated with SQL smells, unlike traditional code smells. Given the fact that some of the SQL smells like *Implicit Columns* contribute to software faults, there was no correlation in the fault-inducing commits. Code smells as *Complex Class*, on the contrary, have shown a stronger correlation. A broad analytical analysis was performed on over 395 releases of 30 open source projects by Palomba et al. (2018b). The investigation into the effect on the fault-proneness of 13 kinds of code smells reported that smelly files are three times more prone to faults than non-smelly files. Further research indicated that only 21% of faults are introduced before smells are introduced in a class. Besides, classes affected with two or three code smells are likely to witness more fault-fixing activities.

In our study, we build on top of these studies to investigate fault-proneness and extend them in the following ways.

- We improve prior approaches by normalizing the number of faults with respect to the history length before and after introducing smells which could impact the number of faults.
- We studied the propagation of faults in smelly and non-smelly files considering the commits-exhibiting-fault. This technique serves to measure the number of commits that indicate a fault until it is fixed. No prior study investigated this aspect, which we believe is important to get an idea on how long it takes to fix or identify the existence of a fault in the code.
- We also explore the code churn needed to fix a fault in smelly and non-smelly files to understand how code smells could affect the number of code changes needed to fix a fault, which has not been addressed in previous studies.

Previous studies have shown a potential negative impact of code smells existence in a variety of languages, including Java, JavaScript, and SQL. Moreover, these studies showed the correlation between higher levels of proneness to fault with the frequency and co-occurrence of smells. However, there is a clear lack of studies targeting the impact of smells in web applications, being one of the largest parts of the current software industry, which are characterized by their inherent heterogeneity in the target platform, formalisms, architectures and programming languages. We design our empirical investigation to cover this gap in the literature, as a

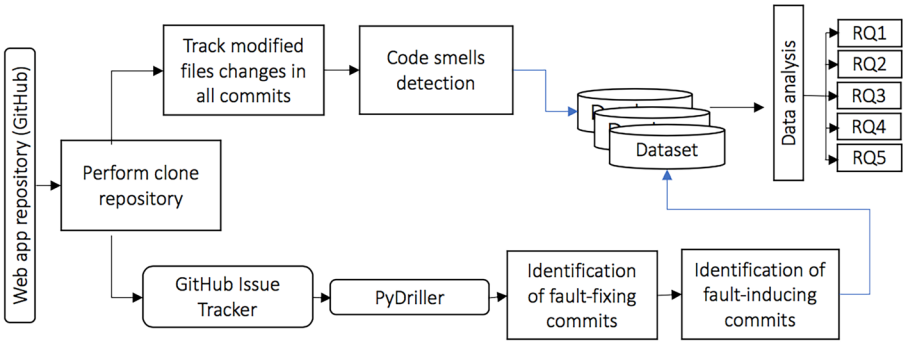


Fig. 1 Overview of the empirical study

first study specialized in the diffuseness and impact of code smells in web applications server side code. The following section details our study design.

3 Empirical study design

The goal of our study is to investigate the (i) *diffuseness*, (ii) *co-occurrence*, and (iii) *impact* of code smells in the server side. Figure 1 shows an overview of our research methodology. More specifically, our study aims at addressing the following research questions:

- **RQ1: To what extent are code smells diffused in web-apps server side?** We examine the extent to which code components are affected by code smells throughout the application's development history. This research question would provide insight into the types of smells that persist in the system, and how often the smells occur.
- **RQ2: What types of code smells tend to co-occur together?** We analyzed 107,228 affected classes to investigate the frequency at which code components are susceptible to be affected by particular pairs of code smells. We examine the co-occurrences of code smells at the class level using the association rules-mining algorithm "Apriori" Agrawal et al. (1994).
- **RQ3: Are smelly files more prone to code changes than non-smelly files?** We aim to determine the weight of code churn that smelly files undergo with the existence of code smells as compared to non-smelly files. The following null hypothesis is under test: $H3_0$: *Smelly files are not more prone to change during the software evolution as compared to non-smelly files.*
- **RQ4: Do different types of code smells exhibit different impact on the level of change-proneness?** We are particularly interested in investigating which types of code smells are more prone to changes than others. By replying to the following null hypothesis, we examine how code smells contribute differently to the change size: $H4_0$: Smelly *files undergo the same churn when exhibiting different types of code smells*.
- **RQ5: Are smelly files more prone to faults than non-smelly files?** We aim to investigate the extent to which code smells lead to more faults in the code by verifying this null hypothesis: $H5_0$: *Smelly and non-smelly files undergo the same number of fault-fixing changes.* Previous studies have demonstrated the tight relation between

the existence of code smells in traditional OO applications and the fault-proneness Palomba et al. (2018b); Khomh et al. (2012). We are interested in investigating the potential effects of smells on the fault-proneness of server side web applications.

3.1 Replication package

Our comprehensive replication package is publicly available for future replications and extensions Dataset (2020).

3.2 Project selection

To answer our research questions, we selected the subject projects for our study following a number of selection criteria.

Step 1: We use the GitHub advanced search engine to find projects written in PHP and collect the repositories with over 1,000 stars. This step resulted in 642 repositories.
Step 2: We removed all non-web-based software projects, and retained those with over 30 releases and five years of history. A collection of 54 repositories met those criteria. We aim at studying a representative number of releases per project to draw statistical evidence from the studied samples.
Step 3: We cloned the identified repositories in phase 2 and used the GitHub API[2] to return the projects with over 10,000 commits. The intent is to select the most active projects in terms of the highest number of commits (more than 10k commits) and the significant history length (more than 5 years)to allow statistical analysis of the obtained results. We ended up with 47 projects.
Step 4: Among the 47 projects found in step 3, we randomly selected a sample of 10 projects for our current study. This sample is twice as large as our initial study Bessghaier et al. (2020).

Overall, we mined a total of 430 releases from 10 popular open-source PHP web-based applications from different domains. The list of considered projects is the following:

- **PhpMyAdmin**[3] is a well-known web application mostly written in PHP to administer MySQL and MariaDB.
- **Joomla**[4] is a Content Management System (CMS) that allows us to create and publish powerful applications.
- **WordPress**[5] is a popular open-source CMS to build websites powered by thousands of plugins.
- **Piwik**[6] is a powerful web analytic application devoted to track users' visits to other websites for analysis.

[2] https://developer.github.com/v3/
[3] https://github.com/phpmyAdmin/phpmyadmin
[4] https://github.com/joomla/joomla-cms
[5] https://github.com/WordPress/WordPress
[6] https://github.com/matomo-org/matomo

Table 1 The studied systems statistics

Name	Releases	Period	Stars	# Classes	KLOCs	# Smells	# Commits
phpMyAdmin	55	2014-2020	4.7k	30-645	228-328	60,695	118,402
Joomla	34	2011-2019	3.4k	1,102-2,631	271-662	75,616	41,287
WordPress	74	2005-2019	13.4k	24-496	37-391	106,962	46,418
Piwik	38	2010-2020	12.6k	1,017-2,095	242-374	39,896	27,156
Laravel	22	2012-2020	57.3k	95-248	12-40	1,647	16,214
CakePHP	26	2011-2020	8.1k	902-3,488	201-341	23,368	42,873
Moodle	38	2011-2020	3k	3,725-11,688	295-703	71,743	153,943
Symfony	58	2011-2020	23.3k	1,466-2,159	119-183	53,622	48,759
CodeIgniter	20	2011-2019	17.9k	88-160	37-113	7,231	10,105
phpBB	65	2001-2020	1.3k	171-1,383	39-282	76,936	34,571

- **Laravel**[7] is a server side PHP framework used to alleviate the cost of web development with powerful syntax and tools required for large applications.
- **CakePHP**[8] is a robust development framework for PHP applications.
- **Moodle**[9] is a free, open-source e-learning management system (LMS) written in PHP. It is used basically for distance education, allowing trainers to create private websites.
- **Symfony**[10] is a PHP web-based application development offering a set of reusable PHP components. Symfony is used by popular PHP projects such as Drupal and Magento.
- **CodeIgniter**[11] is a PHP application development framework, destined to write applications faster by providing a rich set of commonly used libraries.
- **PhpBB**[12] is a free bulletin board software full of unique user-created extensions to easily create forums.

We chose applications with different sizes ranging from 12 to 703 KLOCs. As presented in Table 1, the studied projects belong to different application domains and are actively engineered for 9 to 19 years. Table 1 reports the number of studied releases and the number of stars on GitHub, and we count the size of the application in terms of the number of PHP classes and KLOCs. We also provided the number of total commits and examined code smells in our study.

3.3 Analysis method

In this subsection, we describe the analysis method we used to address each of our defined research questions.

[7] https://github.com/laravel/laravel
[8] https://github.com/cakephp/cakephp
[9] https://github.com/moodle/moodle
[10] https://github.com/symfony/symfony
[11] https://github.com/bcit-ci/CodeIgniter
[12] https://github.com/phpbb/phpbb

Table 2 List of code smells considered in our study

Code smell	Description
Excessive Number Of Children (ENOC)	A class with an excessive number of sub-classes generally leads to a very tightly coupled software hierarchy that is hard to maintain Rio and Abreu (2019); PHPMD (2021).
Excessive Depth Of Inheritance (EDOI)	A class with a profound inheritance tree can prompt an unmaintainable code as the coupling would increment Rio and Abreu (2019); PHPMD (2021).
High Coupling (HC)	Too many dependencies make a class harder to keep up and develop Rio and Abreu (2019); Fowler (1999); Mannan et al. (2016); PHPMD (2021).
Empty Catch Block(ECB)	Fixing an execution error of an obscure special case type will require more endeavors to comprehend the blunder condition PHPMD (2021).
Goto Statement (GTS)	Goto makes the logic of an application difficult to comprehend PHPMD (2021).
High Method Complexity (HMC)	The method-level cyclomatic complexity represents the number of potential executions paths (*e.g.*, if, for, while). The higher the number of paths, the higher the number of experiments expected to test all the distinctive execution ways PHPMD (2021); Hecht et al. (2015).
High NPath Complexity (HNPC)	The NPath intricacy is the number of settled if/else joints, which would diminish the clarity of the code and cause testing issues PHPMD (2021).
Excessive Method Length (EML)	When a method surpasses 100 NCLOC, it is viewed as a large method that does too much. These methods will probably wind up processing information uniquely in contrast to what their setting recommends until they become hard to comprehend and keep up Rio and Abreu (2019); PHPMD (2021); Hecht et al. (2015); Mannan et al. (2016); Delchev and Harun (2015).
Excessive Class Length (ECL)	Enormous classes are a decent suspect for refactoring, as their size depicts a hurdle to oversee effectively Rio and Abreu (2019); PHPMD (2021); Mannan et al. (2016); Liu and Zhang (2017).
Excessive Parameter List (EPL)	A long parameter list can show that a method is doing an excessive number of various things, which makes it harder to comprehend its conduct Rio and Abreu (2019); PHPMD (2021); Martin (2009); Delchev and Harun (2015).
Too Many Public Methods (TMPM)	An enormous number of public methods demonstrates that the class does not protect its information embodied. Thus, changing the inward conduct of the class requires new endeavors not to risk harming some dependencies. In practice, we cannot restrain the number of public methods. Just what could be uncovered ought to be open. On the off chance that external classes are broadly getting to these methods, they should be moved to decrease the coupling PHPMD (2021).
Too Many Methods (TMM)	The Too Many Methods code practice is the side effect of a class that contains countless methods that generally do not have a place among its responsibilities and thus, diminishes the cohesion level PHPMD (2021).

Code smells diffuseness (RQ1) We considered 12 common code smell instances that exist in server side web application's code to prepare our benchmark. The considered 12 code smells types were widely studied in prior studies Delchev and Harun (2015); Mannan et al. (2016); Rio and Abreu (2019); Liu and Zhang (2017); Hecht et al. (2015). To detect code smells, we employed PHPMD (2021), a widely used code smell detection tool dedicated to PHP-based web applications Rio and Abreu (2019); Mon et al. (2019); Mon and Myo (2015); Yulianto and Liem (2014). It is worth noting that we focused on common class-level and method-level code smells that affect the overall software quality. Although PHPMD supports the identification of 36 smell types, we discarded smells related to low-level violation which are likely to have negligible or no impact such as naming conventions (*e.g.*, short names), *Unused Local Variable* smell, calling the function `var_dump()` in the production code, and other simple documentation-related smells. Table 2 presents and provides the definitions of each code smell type.

For each class C_i in a release R_j, we compute the number of each code smell type instances it contains. Then, we analyze the number of affected classes and releases by each code smell type. We also assess code smells diffuseness per KLOC to better position the number of smells with respect to size.

Code smells co-occurrences (RQ2) We conduct a preliminary analysis following Palomba et al. (2018a) approach that assesses the co-occurrence score of a pair of smell types, St_i and St_j as follows:

$$co\text{-}occurrence(St_i, St_j) = \frac{(St_i \cap St_j) * 100}{St_i}, \text{where} i \neq j \tag{1}$$

where $St_i \cap St_j$ represents the number of co-occurrences of two distinct types of smells. It is worth mentioning that the $co\text{-}occurrence(St_i, St_j)$ differs from $co\text{-}occurrence(St_j, St_i)$ as the denominator varies from St_i to St_j, *i.e.*, the co-occurrence of $St_i \rightarrow St_j$ is not the same as $St_j \rightarrow St_i$.

Then, we exploit the Apriori algorithm Agrawal et al. (1994), a widely-used algorithm for association rule mining to discover frequent item-sets. The output of the algorithm is a set of association rules representing strongly associated item pairs, *i.e.*, code smells in our study. The degree of association of each rule (*i.e.*, a pair of code smells) is evaluated using five measures, *Support* Agrawal et al. (1993), *Confidence* Agrawal et al. (1993), *Lift* Brin et al. (1997), *Leverage* Piatetsky-Shapiro (1991), and *Conviction* Brin et al. (1997).

– *Support* Agrawal et al. (1993): The support metric represents how much a pair of smells (ST1 and ST2) change together in the same commit.

$$Sup(ST1 \Rightarrow ST2) = \frac{ST1 \bigcup ST2}{Transactions} \in \{0 - 1\} \tag{2}$$

– *Confidence* Agrawal et al. (1993): The confidence implies the probability that a smell is related to the presence of another smell with values varying from 0 to 1.

$$Conf(ST1 \Rightarrow ST2) = \frac{Sup(ST1 \Rightarrow ST2)}{Sup(ST1)} \in \{0 - 1\} \tag{3}$$

– *Lift* Brin et al. (1997): The lift is used to measure the smells-dependence ratio. The range of values for the lift is between 0 and $+\infty$. When the lift value is greater than 1, it implies that the pair of smells is highly correlated (*i.e.*, a higher likelihood of a causality relationship).

$$Lift(ST1 \Rightarrow ST2) : \frac{Sup(ST1 \Rightarrow ST2)}{Sup(ST1) * Sup(ST2)} in\{0-1\} \tag{4}$$

– *Leverage* Piatetsky-Shapiro (1991): The leverage tests the difference between two smells support score with values ranging from -1 to 1. A leverage of 0 indicates total independence between both smells.

$$Leverage(ST1 \Rightarrow ST2) = Sup(ST1 \Rightarrow ST2) - (Sup(ST1)Sup(ST2)) in\{0-1\} \tag{5}$$

– *Conviction* Brin et al. (1997): measures the probability of a smell occurring without another smell, returning a value within a range of 0 to $+\infty$. When the conviction score is equal to 1, this implies that the smells are independent.

$$Conviction(ST1 \Rightarrow ST2) = (1 - Sup(ST2))/(1 - Conf(ST1 \Rightarrow ST2)) in\{0-1\} \tag{6}$$

Moreover, we use the *Chi-squared* and the *Cramer's V* to assess the degree of association between the code smells. The *Chi-squared* test is a statistic used to evaluate the importance of the size of variation in the same population between two categorical variables (in our case, the combined data set). The test aims to assess the following null hypothesis: $-H2_0$: *Code smells occur independently of each other.* The *Cramer's V* is an effect size test carried out on a Chi-squared Pearson test to assess the strength of the relation. The test formula is described in Eq. 7 and includes the sample size (n), the chi-squared value denoted by X^2, the number of rows (r) and columns (c) representing the contingency table's distinct values.

$$Cramer'sV = \sqrt{\frac{X^2}{n * (r-1)(c-1)}} \tag{7}$$

Furthermore, since some code smells are by definition related to size, we conduct an additional experiment to examine the correlation between the class size (LOC) and the number of code smells in small, medium, and large classes. We consider the lower and upper quartiles of the box plot representing the distribution of class sizes as thresholds (60 and 300, respectively). This test indicates how likely is the number of code smells occurrences is influenced by the class size by measuring the correlation using the Kendall rank correlation coefficient test Kendall (1938). A coefficient in the interval [0, 0.199] is considered very weak, 0.2– >0.399 is *weak*, [0.4, 0.599] is *moderate*, [0.6, 0.799] is *strong*, and [0.8, 1] is *very strong*.

Impact of code smells on the change-proneness (RQ3) To study the impact of code smells on the change-proneness, we mine the impact history of each project using the `git` versioning system. We track all changed PHP files in all commits between two consecutive releases. Then, we use the following git command to extract the number of modifications each modified file has undergone:

```
$ git show –stat –no-commit-id –oneline -r SHA "*.php"
```

where the command `git show` shows several details related to many objects like tags and commits; `-stat` shows the amount of deleted and inserted changes; `-no-commit-id` suppresses the commit ID output. `-oneline` gives just one line per commit; *-r* is used to recurse into sub-trees; *SHA* is the commit hash, and "`*.php`" returns only changes applied on php files. Then, we identify whether the returned files are *smelly* or *non-smelly*. Afterward, we calculate using Eq. 8, the change-proneness of a changed class C_i as the amount of the changes made in all commits between two consecutive releases R_{j-1} and R_j.

$$Change\text{-}proneness(c, r_j) = \sum_{i=1}^{i=n} churn(c, com_i) \tag{8}$$

where n is the number of commits between R_{j-1} and R_j. The function $churn(c, com_i)$ returns the code churn as the number of added, removed and modified lines of code in the class C_i in the commit com_i using the GitHub API[13].

After extracting all data, we use statistical analysis and the beanplot representation Kampstra (2008) to compare the change-proneness of smelly and non-smelly classes. A beanplot expands the representation of the boxplot by displaying the density of the distribution of data and individual observations. To assess $H3_0$, we check if there is a significant difference between both groups, *i.e.*, smelly and non-smelly. Specifically, we apply the nonparametric Mann–Whitney U-test Conover (1998) to check the differences between our two groups with a confidence level of 95% (*p-value* <0.05). The Mann–Whitney is used to evaluate the alternative hypothesis stating how likely one sample exhibits dominance over the other sample. We consolidate the test by measuring the size of the Cliff-delta non-parametric effect test. The effect size is considered negligible for d <0.147, small for d <0.33, medium for d <474, and large for d >=0.47.

It is worth noting that a class is considered smelly if it has at least one code smell instance. We narrowed the distance between smelly and non-smelly groups to investigate the change-proneness phenomenon considering the studied smells. Furthermore, we measure the *Change-frequency* metric to assess how many times smelly files are prone to updates compared to non-smelly files. To this end, we count the number of commits where an update has been observed by a single modified file f_i between two consecutive releases.

Furthermore, we measure the correlation between the class change size (Churn) and its LOC to examine the effect of the LOC on the class change size using the Kendall correlation test. To this end, we cluster the sizes of classes participating in code smells based on the aforementioned defined quartiles, and for each class, we count its churn. Consequently, we have size-related code smells such as the *Excessive Class Length* in large classes.

Impact of code smells types on the change-proneness (RQ4) We measure the effect of code smells on the change-proneness in terms of their frequency count. In each release, we quantify the number of occurrences of each smell in the smelly groups. For each project, we measure the impact of each smell type in terms of the correlation score between (1) the frequency count of each smell type ST_i, and (2) the class state {0 or 1} that represents whether a class has changed between two releases $R_{j-1} \rightarrow R_j$ or not. In order to evaluate the effect of each smell type on the change-proneness statistically, we opted for a *logistic regression* test Hosmer et al. (2000), similar to Khomh et al. (2009), to assess the

[13] https://developer.github.com/v3/

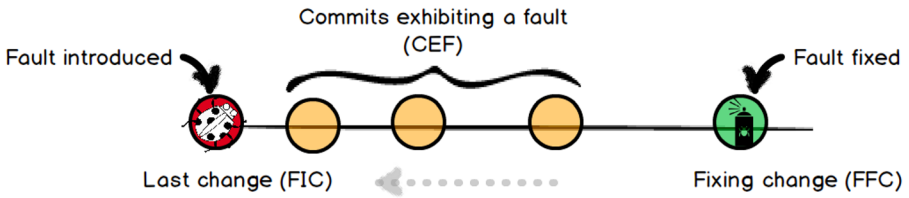


Fig. 2 A fault life cycle in a system

null hypothesis $H4_0$ claiming that classes experience the same change size for all types of smells. The logistic regression should predict if the class affected by particular smells would change. To assess a class change based on a set of smells, a class represents the C_i dependent variable that would change if one of the ST_j (independent variable) smell types also changes. For a logistic model, only two values may be taken from the dependent variable (*i.e.*, changed=1, not changed=0). The formula of the logistic model is defined as follows:

$$P(C_i) = \frac{e^{(CP+\sum_1^n b_j \times ST_j)}}{1 + e^{(CP+\sum_1^n b_j \times ST_j)}} \in [0, 1] \tag{9}$$

where P is the probability that a class will change, CP is the class change proneness which could be {0,1}, n is the number of observations, and b_j is the code smell number of type ST_j. For each smell type detected in our benchmark, we apply our logistic regression model. Then, we count the number of times that the *p-value* of the smell is significant (the likelihood is nearer to 1)

Impact of code smells on the fault-proneness (RQ5) In this experiment, we want to investigate whether faults are related to code smells in a class or not. Regardless of the type of smells, we want to study whether code smells could lead to faults. As reported in Fig. 2, our investigation is based on the logic of the SZZ algorithm Śliwerski et al. (2005) since the SZZ is regarded as the de-facto standard for faults localization, and many studies have employed it in their investigations Saboury et al. (2017); Palomba et al. (2018b); Muse et al. (2020). The SZZ consists of linking the historical changes in the VCS with the Issue Tracking System (ITS).

It is important to note that faults could appear before a code smell exists in a source file, which may increase the fault-proneness rate of smelly files. Therefore, we quantify our variables of interest (fault-fixing commits, commits exhibiting a fault, and code churn) after the introduction of code smells into the code base. Besides, we observed different results of our studied phenomenon with respect to the class size (LOC). Therefore, our classes were divided into small, medium, and large considering the lower and upper quartiles of the box plot representing the distribution of sizes as thresholds (60 and 300). In particular, we assess various fault-related behaviors as follows:

1. We check whether faults in fault-inducing commits (*i.e.*, where a fault is firstborn in the codebase) always occur before or after the smells are added. Specifically, for each class C_i, we detect and evaluate the number of fault-inducing commits before and after introducing smells between two consecutive releases R_j and $R_j + 1$. For example, assume that between $R_j - 1$ and R_j, class C was affected by at least one smell type. The smell was

then removed between R_j and $R_j + 1$ releases. Lastly, between releases $R_j + 1$ and $R_j + 2$, the smell was reintroduced. We measured class C's fault-proneness when it was smelly by summing up the number of fault-inducing commits in the periods between $R_j - 1$ and R_j and between $R_j + 1$ and $R_j + 2$. Similarly, when it was not smelly, we computed the fault-proneness of class C by measuring the number of fault-inducing commits in the period between R_j and $R_j + 1$. We used a beanplot representation to visualize the size of the difference between our two compared populations (before smells, and after smells are introduced) and complemented our investigation with the Mann–Whitney and Cliff's delta statistical tests to assess how significant is the difference.

2. We normalize the number of fault-inducing commits by the number of effective commits in a release R_j to reflect the possible impact of the history length on the number of faults, *i.e.*, the history length before a code smell is introduced in a class could be significantly smaller than the history after the code smell is introduced. We considered the number of effective commits between two releases instead of the number of days to capture the lifespan of a code smell better as adopted by Habchi et al. (2019). Indeed, the number of days can over-or under-estimate the lifespan of a code smell in a file if such a file experiences very few or too many commits. This fine-grained analysis allows capturing better the lifespan of a code smell Habchi et al. (2019). For each release R_j tag we extract its SHA using the PyDriller. Then, we use the Commits API to extract the total number of commits for R_j.
3. We assess the proportions of smelly and non-smelly classes experiencing at least one fault-fixing activity. Specifically, we compute the fault-proneness of class C_i in terms of the number of fault-fixing commits between two consecutive releases R_j and R_{j+1}. We compare the difference of the fault-proneness between our two sets of smelly and non-smelly classes using the nonparametric Mann–Whitney U-test Conover (1998) to investigate the validity of this null hypothesis $-H5_0$: *smelly and non-smelly files undergo the same number of fault-fixing commits*. We test the statistical significance at a confidence level of 95%. We also compute the effect size test Cliff's Delta (*d*) Grissom and Kim (2005) to assess the magnitude of the difference between the two distributions.
4. We assess the propagation of faults in smelly and non-smelly files to test the extent to which faults persist in a class participating in code smells. To this end, we compute for each class C_i, the number of commits exhibiting a particular fault[14] .
5. We examine code churn needed for fixing faults in classes participating in code smells. We measure the code churn (added, removed, and modified lines of code) used to fix the fault in the fault-fixing commit for each class C_i. It is necessary to note, however, that the measured code churn of the defective code fragment may not possibly be a smelly code fragment. Thus, we did not fine-grain our granularity level and distinguished between faults occurring at the class or method-level.

To collect our data for RQ5, we first used the GitHub API issue tracker[15] to acquire the *issue type* (*i.e.*, fault, enhancement, etc.) and the *issue state* (*i.e.*, closed, open).

The issue state "closed" indicates that an issue has been satisfied in the current commit, whereas the issue type "bug" allows us to distinguish between a bug and

[14] The commits exhibiting faults are the commits that touched the faulty code excluding the fault-inducing commit, which is considered the last change (cf. Sect. 2.4).
[15] https://developer.github.com/v3/

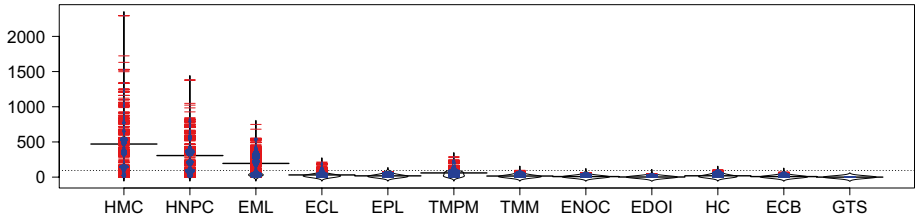
Table 3 Used keywords designating fault-fixing commits

Keywords	% fault-fixing commits
fix*	76.79%
error*	7.03%
bug*	5.33%
issue*	4.37%
close*	2.96%
solve*	2.51%
fail*	0.81%
fault*	0.09%
crash*	0.05%
defect*	0.03%
Total fault-fixing commits	105,109

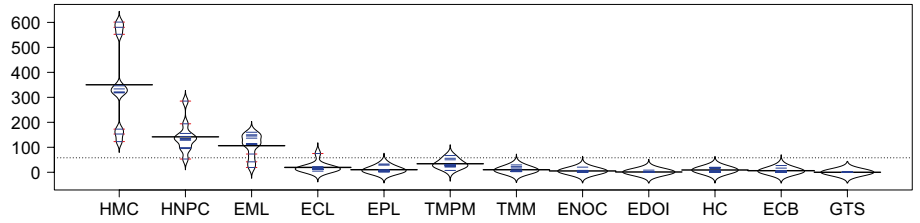
non-bug-related issues such as "enhancements". We select the commits of all closed bug-labeled issues. Then, we use PyDriller Spadini et al. (2018), a python framework to interact with GitHub repositories using git commands, to identify the fault-inducing commits, specific to a given issue. Several recent studies Asmare Muse et al. (2020); Lenarduzzi et al. (2019); Pecorelli et al. (2020) have relied on PyDriller to mine the project's control version system. First, to identify fault-fixing commits, we used 29 keywords variations (*e.g.*, fix, fixed, fixes, bug, bugs, error, errors, issue, fault, solve, close) indicating possible fixing of code faults. Our set of keywords were used in prior studies Muse et al. (2020); Palomba et al. (2019); Saboury et al. (2017); Spadini et al. (2018) to identify fault-fixing commits. Antoniol et al. (2008) indicated that those keywords are highly associated with fault-fixing commits. Table 3 presents the percentages of the used keywords variations by developers when performing a fault-fixing change.

Our tool traverses all fault-fixing commits between two releases R_{j-1} and R_j and searches for the keywords in the commits messages. If a keyword is identified, the tool examines if the returned commit has the issue state "closed" and the label "bug" and separates it as a fault-fixing commit. Furthermore, PyDriller implements the logic of the SZZ algorithm Śliwerski et al. (2005), which consists of blaming (*i.e.*, applies the git `blame` command) a modified file f_i to get the list of commits that have previously involved f_i. Then, for all the modified files, we separate the smelly from the non-smelly ones. Hence, we use PyDriller to detect the fault-inducing commit(s) from a given fault-fixing commit.

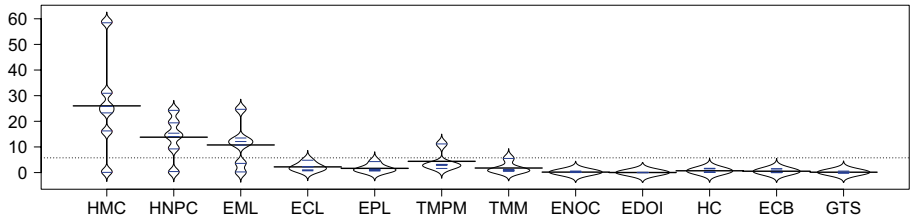
Note that the original version of the SZZ algorithm has its own limitations as pointed out by Rodríguez-Pérez et al. (2018) since it could deliver false positives related to changes in whitespaces and comments. To avoid this issue, we used the recent PyDriller implementation of SZZ which does not blame whitespace and comments. Besides, we only extract fault-inducing commits of fault-fixing commits that were closed and classified as a fault to restrict the threat of considering commits related to other types of issues. Getting fault-inducing commits enables us to investigate the number of commits aimed at fixing the faults in each class.



(a) Avg number of classes affected by each code smell type.



(b) Avg number of classes affected by each code smell type.



(c) Code smells density per KLOCs.

Fig. 3 Illustrative graphs depicting the diffuseness of code smells in the studied projects

4 Study results and analysis

In this section, we report and discuss the obtained results of our research questions.

4.1 RQ1: Code smells diffuseness and frequency

Figure 3 reports (*i*) the absolute number of code smells distributed in the studied projects, (*ii*) the number of classes affected by each code smell, and (*iii*) the density of code smells per KLOC using the beanplot visualization. For the sake of clarity, we aggregate the occurrences of each code smell into one single dataset. From the beanplots and Table 4, we observe the presence of three major types of distributions of code smell (1) high diffuseness and high frequency, (2) high diffuseness and low frequency, and (3) low diffuseness and low frequency.

High diffuseness and high frequency The list of code smells included in this category affect the majority of releases and frequently occur in classes. For instance, we found the *High Method Complexity* smell being highly diffused and frequent, with 99% of affected

Table 4 Diffuseness of code smells in the analyzed projects in terms of percentage of affected releases, classes and code smells

Code smell	% releases	% classes	% of smells
High Diffuseness and High Frequency			
High Method Complexity	99%	67.2%	41.6%
High NPath Complexity	99%	50.6%	27%
Excessive Method Length	97%	38.6%	17.2%
High Diffuseness and Low Frequency			
Excessive Class Length	97%	12.1%	2.7%
Too Many Public Methods	90%	26%	5.2%
Excessive Parameter List	86%	3.7%	1.6%
Too Many Methods	80%	6.7%	1.3%
High Coupling	75%	7.3%	1.7%
Excessive Number Of Children	74%	3.1%	~1%
Empty Catch Block	53%	2.6%	~1%
Low Diffuseness and Low Frequency			
Excessive Depth Of Inheritance	11%	0.8%	~1%
Goto Statement	1%	0.05%	< 0.01

releases and accounts for 41% of the total code smells instances. In particular, this smell affects an average of 338 classes, which indicates that methods in the studied web applications tend to have a high cyclomatic complexity exceeding the threshold of 10 PHPMD (2021). Hence, we found an average of 60 instances of the *High Method Complexity* smell per KLOC. The highest number of occurrences of the *High Method Complexity* accounts for 2,296 instances and is found in the project `Moodle` (v2.1.2). For example, the classes `lib.tcpdf.tcpdf.php` and `lib.moodlelib.php` have the highest complexity score of 98 in their methods `closeHTMLTagHandler()` and `clean_param()` with an excessive length of 336 and 358 LOC, respectively. The average number of LOC for methods with high complexity in the release 2.1.2 of `Moodle` is 347.

In addition, we found that the *High NPath Complexity* also occurs in 99% of releases, which represents 27% of the total number of smells detected. In `Moodle` (v2.1.2), the *High NPath Complexity* has a total of 1,383 occurrences affecting 184 classes. The two methods `closeHTMLTagHandler()` and `clean_param()` of `Moodle` (v2.1.2) have an NPath Complexity score exceeding 400. We assume that a potential causality relationship could be elaborated between both code smells *High Method Complexity* and *High NPath Complexity*.

Similarly, the *Excessive Method Length* smell affects 97% of releases, representing 17% of the detected code smells instances with the highest number of occurrences of 749 found in `Moodle` (v2.1.2). The diffuseness of smell instances per KLOC is depicted in Fig. 3c, which confirms that *High Method Complexity*, *High NPath Complexity*, and *Excessive Method Length* are the most diffused code smells with an average of 26, 13 and 11 instances per KLOC, respectively.

High diffuseness and low frequency The majority (58%) of the smells examined in our studied releases are highly diffused with a low frequency rate. These smells are diffused in

most of the studied releases, but with a limited frequency rate (number of instances). These smells affect more than 50% of the releases. For instance, 97% of the releases are affected by the *Excessive Class Length* smell, with an average of 20 affected classes having a peak of 162 in `Moodle` (v3.8.1). The *Too Many Public Methods* smell, for example, constitutes 5% of the total number of smells and is spread over 90% of the releases, as seen in Fig. 3 and Table 4.

Additionally, the code smells *Empty Catch Block* and *Excessive Parameter List* affect 53% and 86% of the releases, respectively, with a maximum number of occurrences of 69 in `Piwik` (v2.17.1 and v2.18.0) and 80 in `phpMyAdmin` project (v4.3.9). In the method `setProperties()` of the class `DisplayResults.class.php`, the *Excessive Parameter List* has the highest occurrence number (19) of all low frequent smells. The *Excessive Parameter List* is a one-metric violation smell which detects the smell straight away. In addition, note that `Wordpress`, `CodeIgniter`, and `phpMyAdmin` projects do not have any instance of the *Empty Catch Block* which, in total, is limited to one instance per KLOC.

Moreover, we find that the *Excessive Class Length*, *Too Many Public Methods*, *Excessive Children's Number*, and the *Too Many Methods* smells are not commonly occurring, up to a maximum of 7 instances per class. We also noticed that most of the smells in this category of diffuseness are at the class level.

Low diffuseness and low frequency As shown in Fig. 3 and Table 4, smells related to class inheritance, *e.g.*, *Excessive Depth Of Inheritance*, have low diffuseness and low frequency. Overall, the *Excessive Depth Of Inheritance* smell occurs in 469 classes, affecting only 11% of the analyzed releases, and it accounts for almost 1% of the total number of code smells found. Among the affected classes by this smell, 92% are in the project `Moodle`. The highest number of occurrences is 49 in `Moodle` v3.0.8. Likewise, the code smell of *Goto Statement* accounts for almost 1% of the total number of code smells and affects 1% of the studied releases. Table 4 presents the diffuseness of application smells, depending on the number of releases gathered. The column "% of releases affected" reflects the number of releases affected by a specific smell. For instance, the smell of *Too Many Methods* affects 80% of the releases, and accounts for only 1.3% of the total number of detected smells (482019).

To sum up, the set of code smells in the releases being examined are highly diffused. In particular, smells related to long and complex fragments of code (*i.e.*, *High Method Complexity*, *High NPath Complexity*, and *Excessive Method Length*) have the highest number of instances per KLOC and affect the highest number of classes. Our results are in line with those of Palomba et al. (2018b) on Java desktop applications, where *Long method* and *Complex Class* code smells are the most diffused. Besides, 75% of the code smells analyzed are not frequent (*i.e.*, a small number of occurrences per release), but affect 63% of the studied releases. For instance, the project `Moodle` is the most affected project with a highest number of instances of 9 code smells *HMC, HNPC, EML, ECL, TMPM, TMM, ENOC, HC, EDOI*. By observing the diffuseness of code smells, we strive to determine the interplay between the magnitude of the diffuseness for each smell and the maintainability of code. Smells of highly diffuseness and frequency represent potential candidates with a high impact on the maintainability of source code. Although it is expected that the extremely low diffused smells *Excessive Depth Of Inheritance* and *Goto Statement* would have no major effect on the change-proneness as their diffuseness rate is limited.

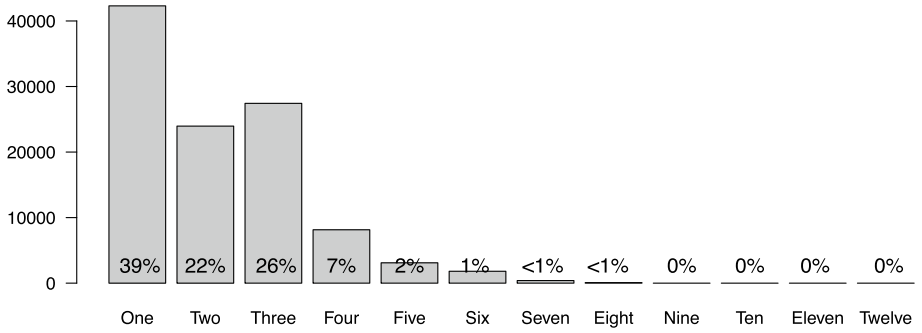


Fig. 4 Code smells co-occurrence frequency in the studied releases

Summary for RQ1 Most of the code smells are diffused in the studied web applications server sides. Mainly, the smells related to long and complex code fragments (*High Method Complexity, High NPath Complexity* and *Excessive Method Length*) are highly diffused and frequent, affecting more than 95% of the studied releases, and account for more than 17% of the number of detected smells. While other types of smells such as the *Excessive Depth of Inheritance* and the *GoTo Statements* are not diffused as they affect 11% and 1% of the releases, respectively.

4.2 RQ2: Code smells co-occurrence

Our initial analysis showed that multiple code smell types could affect many code elements (classes, methods) at once. Prior studies have shown how the co-occurrence of smells could intensify the degree of the change-proneness Abbes et al. (2011); Palomba et al. (2018a). Indeed, an increased number of code smells may negatively impact source code maintainability. As shown in Fig. 4, 39% of smelly code fragments contain only one type of smell, while 61% of the smelly code fragments contain two or more types of code smells. It is worth noting that our findings are consistent with prior studies on Android mobile applications et al. Palomba et al. (2019, 2018a) which found that ~60% of Java-based Android apps' source code was affected by over one smell. These findings reflect the importance of understanding the degree to which our analyzed code smells consolidate the co-occurrence phenomenon and its effect on source code.

The goal is to investigate the extent to which the existence of a code smell in a code fragment implies the existence of another form of code smells. Table 5 reports the absolute number and the percentages of co-occurrences across each pair of the 12 studied code smell types. We show the absolute number of code smells in the horizontal header, and the percentages in the vertical header. In each intersection cell, we report the co-occurrence score between the pair of smell types. We highlight in bold the smell type with which the smell of interest (in the vertical header) mostly co-occurs. For example, we find that the *High Method Complexity* (HNPC) mostly co-occurs with 74.73% of the *High NPath Complexity* (HMC) instances. Interestingly, we observe from Table 5 that 9 code smells (ENOC, HC, ECB, GTS, HNPC, EML, ECL, EPL, and TMPM) also co-occur with the *High Method Complexity* (HMC) smell. Indeed, this finding could be justified by our results in RQ1 since the *High Method Complexity* is the highest diffused smell (99% of releases) and represents 42% of the total number of code smells in the studied projects.

Table 5 The absolute number and percentages of code smells co-occurrences

$ST_{i/j}$	ENOC	EDOI	HC	ECB	GTS	HMC	HNPC	EML	ECL	EPL	TMPM	TMM
	(3362)	(840)	(7876)	(2829)	(63)	(72145)	(54299)	(41368)	(13026)	(3972)	(27950)	(7179)
ENOC		4	346	32	0	**535**	390	312	159	11	212	63
(3.13%)		(0.11%)	(10.29%)	(0.95%)	(0%)	**(15.91%)**	(11.60%)	(9.28%)	(4.72%)	(0.32%)	(6.30%)	(1.87%)
EDOI	4		21	0	0	0	0	0	0	0	0	0
(0.78%)	(0.47%)		**(2.5%)**	(0%)	(0%)	(0%)	(0%)	(0%)	(0%)	(0%)	(0%)	(0%)
HC	346	21		460	0	**3115**	2395	1661	823	618	1665	726
(7.34%)	(4.39%)	(0.26%)		(5.84%)	(0%)	**(39.55%)**	(30.40%)	(21.08%)	(10.44%)	(7.84%)	(21.14%)	(9.21%)
ECB	32	0	460		2	**613**	423	287	109	24	519	211
(2.63%)	(1.13%)	(0%)	(16.26%)		(0.07%)	**(21.66%)**	(14.95%)	(10.14%)	(3.85%)	(0.84%)	(18.34%)	(7.45%)
GTS	0	0	0	2		**63**	**63**	60	0	0	0	0
(0.05%)	(0%)	(0%)	(0%)	(3.17%)		**(100%)**	**(100%)**	(95.23%)	(0%)	(0%)	(0%)	(0%)
HMC	535	0	3115	613	63		**53990**	34249	7168	3081	11343	3064
(67.28%)	(0.74%)	(0%)	(4.31%)	(0.84%)	(0.08%)		**(74.73%)**	(47.47%)	(9.93%)	(4.27%)	(15.72%)	(4.24%)
HNPC	390	0	2395	423	63	**53990**		32520	6944	2893	9584	2798
(50.63%)	(0%)	(0%)	(4.41%)	(0.77%)	(0.11%)	**(99.43%)**		(59.89%)	(12.78%)	(5.32%)	(17.65%)	(5.15%)
EML	312	0	1661	287	60	**34249**	32520		7087	2555	8386	2909
(38.57%)	(0.75%)	(0%)	(4.01%)	(0.69%)	(0.14%)	**(82.79%)**	(78.61%)		(17.13%)	(6.17%)	(20.27%)	(7.03%)
ECL	159	0	823	109	0	**7168**	6944	7087		753	6634	4131
(12.14%)	(1.22%)	(0%)	(6.31%)	(0.83%)	(0%)	**(55.02%)**	(53.30%)	(54.40%)		(5.78%)	(50.92%)	(31.71%)
EPL	11	0	618	24	0	**3081**	2893	2555	753		847	287
(3.70%)	(0.27%)	(0%)	(15.55%)	(0.60%)	(0%)	**(77.56%)**	(72.83%)	(64.32%)	(18.95%)		(21.32%)	(7.22%)
TMPM	212	0	1665	519	0	**11343**	9584	8386	6634	847		6883
(26.06%)	(0.75%)	(0%)	(5.95%)	(1.85%)	(0%)	**(40.58%)**	(34.28%)	(30%)	(23.73%)	(3.03%)		(24.62%)
TMM	63	0	726	211	0	3064	2798	2909	4131	287	**6883**	
(6.69%)	(0.87%)	(0%)	(10.11%)	(2.93%)	(0%)	(42.68%)	(38.97%)	(40.52%)	(57.54%)	(3.99%)	**(95.87%)**	

Table 6 Top-3 pair of smells based on the lift value for each individual application and for all applications combined

Project	Smells pair	Support	Confidence	Lift	Leverage	Conviction
Moodle	Too Many Methods ⇔ Excessive Class Length	0.01	0.51	30.93	<.01	1.52
	Excessive Method Length ⇔ Too Many Public Methods	0.01	0.48	29.59	-0.01	1.8
	Excessive Method Length ⇔ Excessive Class Length	0.01	0.50	19.20	-0.01	1.48
Joomla	Too Many Methods ⇔ Excessive Class Length	0.02	0.20	9.22	0.01	1.12
	Too Many Methods ⇔ Excessive Method Length	0.01	0.26	7.55	<.01	0.82
	Excessive Method Length ⇔ High NPath Complexity	0.03	0.35	7.45	-0.17	0.72
phpMyAdmin	Excessive Method Length ⇔ Excessive Class Length	0.007	0.24	14.71	-0.05	1.18
	Excessive Method Length ⇔ High Method Complexity	0.007	0.24	14.64	-0.36	0.45
	Excessive Class Length ⇔ High Coupling	0.007	0.23	13.97	0.004	1.26
CakePHP	High NPath Complexity ⇔ Excessive Method Length	0.01	0.29	13.18	-0.05	1.12
	High NPath Complexity ⇔ Too Many Public Methods	0.02	0.44	10.55	-0.19	0.69
	Too Many Methods ⇔ Excessive Class Length	0.04	0.72	9.97	<.01	2.92
CodeIgniter	Excessive Method Length ⇔Leftrightarrow Too Many Public Methods	0.07	0.64	4.99	-0.04	1.88
	Excessive Method Length ⇔ High NPath Complexity	0.77	0.64	4.99	-0.17	0.82
	High NPath Complexity ⇔ High Method Complexity	0.08	0.81	4.28	-0.53	0.61
Piwik	Too Many Methods ⇔ Excessive Method Length	0.01	0.24	19.83	<.01	1.10
	High Method Complexity ⇔ Too Many Methods	0.01	0.24	19.83	-0.03	1.23
	High NPath Complexity ⇔ Too Many Methods	0.01	0.28	18.80	-0.01	1.3
Wordpress	Excessive Method Length ⇔ High NPath Complexity	0.03	0.24	7.39	-0.45	0.24
	Too Many Public Methods ⇔ Excessive Method Length	0.03	0.24	7.39	-0.12	0.52
	Excessive Class Length ⇔Leftrightarrow Excessive Method Length	0.03	0.27	7.31	-0.04	0.54
Symfony	Excessive Class Length ⇔ Too Many Public Methods	<.01	0.22	16.11	.01	0.77
	High Coupling ⇔ Too Many Public Methods	0.01	0.26	7.85	-0.04	0.80
	High Coupling ⇔ High NPath Complexity	0.01	0.24	7.52	-0.01	0.90
Laravel	High NPath Complexity ⇔ High Method Complexity	0.14	0.65	4.39	0.11	2.2

Table 6 (continued)

Project	Smells pair	Support	Confidence	Lift	Leverage	Conviction
phpBB	Too Many Public Methods ⇔ Too Many Methods	<.01	0.51	9.28	<.01	2.04
	Excessive Method Length ⇔Leftrightarrow-Leftrightarrow Too Many Public Methods	<.01	0.90	7.97	-0.16	7.99
	Excessive Class Length ⇔ Too Many Public Methods	<.01	0.41	7.78	-0.02	1.24
Projects combined	Excessive Parameter List ⇔ Too Many Methods	<.01	0.26	53.57	<.01	1.27
	High Coupling ⇔ Excessive Parameter List	<.01	0.26	34.95	<.01	1.31
	Excessive Parameter List ⇔ High NPath Complexity	<.01	0.20	32.89	-0.01	0.61
	Too Many Methods ⇔ Excessive Class Length	<.01	0.35	19.34	<.01	1.35

Furthermore, taking advantage of the Apriori association rule mining algorithm, we investigate the phenomenon of code smells co-occurrences within our benchmark at the class and method levels. We use Apriori to assess the relation based on a set of metrics between a pair of code smells. We set our metrics thresholds as follows: we selected a minimum support of 80% of a code smells pair. Based on Fig. 4, more than 60% of classes have more than one smell type. Thus, we increased the minimum support threshold to restrict the number of associated smells. The lift is set to 1 as we are looking for strongly associated smells. We set the minimum length of a rule to 2 since we want to assess the association degree between 2 code smells. Consequently, the strongest candidate rules are those with higher metrics values than the defined metrics thresholds.

Table 6 reports the top-3 associated pairs of smells based on the lift value, for each of our studied projects, and for all the projects (combined dataset). The findings indicate that only 7 out of the 12 code smells tend to co-occur frequently and have associations with each other (*i.e.*, lift>1). We note that the method-level smell *Excessive Method Length* frequently co-occurs with the method-level smell *High NPath Complexity* in the projects `WordPress`, `CodeIgniter`, `Joomla`, and `CakePHP`, which is presumed to be the result of the *Excessive Parameter List* as indicated when combining all projects in a single dataset. Moreover, the *Excessive Method Length* often co-occurs with *Too Many Public Methods*, in `phpBB`, `WordPress`, and `Moodle`. We also notice that the *Too Many Methods* co-occurs with the *Excessive Class Length* smell in 3 out of the 10 studied projects, `Moodle`, `Joomla`, and `CakePHP`. However, each pair of code smells has a leverage value close to zero, which means the considered associations are weak.

To further assess the association between smell pairs, we calculate the Chi-square coefficient McHugh (2013) and Cramir (1946) tests to examine whether these relationships are statistically significant. In particular, we apply the chi-squared and the Cramer's V tests on the code smell pairs having lift values higher or equals to 15. In total, 10 pairs of smells are identified with lift>15. Table 7 presents the results of our Chi-squared test to the related null hypothesis, $H2_0$, assessing the absolute independence of the two variables, *i.e.*, the causality relationship between the co-occurring smells. Out of the 10 examined pairs in

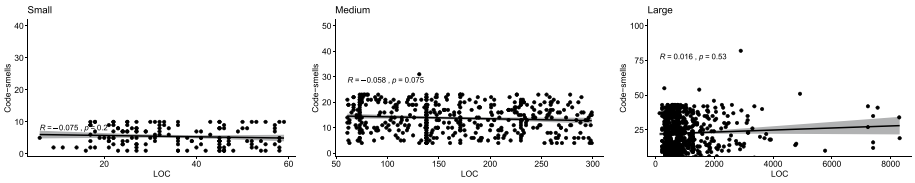
Table 7 The results obtained from the Chi-square and Cramer's V tests

Smells pair	Chi-square p-value	Cramer's V
Excessive Class Length ⇔ Too Many Public Methods	**<0.0001**	**0.2105**
Too Many Methods ⇔ Excessive Class Length	**<0.0001**	**0.1483**
High Method Complexity ⇔ Too Many Methods	**<0.0001**	**0.1402**
Excessive Method Length ⇔ Excessive Class Length	**<0.0001**	**0.1208**
Excessive Method Length ⇔ Too Many Public Methods	**<0.0001**	**0.1045**
Excessive Parameter List ⇔ High NPath Complexity	**<0.0001**	**0.0870**
High NPath Complexity ⇔ Too Many Methods	**<0.0001**	**0.0624**
High Coupling ⇔ Excessive Parameter List	**<0.0001**	**0.0616**
Too Many Methods ⇔ Excessive Method Length	**0.0004**	0.0106
Excessive Parameter List ⇔ Too Many Methods	0.1833	0.0041

Table 7, and given the p-values <0.05, we reject the null hypothesis for the first 9 smell pairs (p-values highlighted in bold). The class-level smell *Excessive Class Length* is significantly associated with three method-level smells (1) *Too Many Public Methods*, (2) *Too Many Methods*, and (3) *Excessive Method Length*. Hence, These findings suggest that there is a unidirectional cause-effect relationship in these three associations (*Method-level* ⇔ *Class-level*).

Further investigation of the Cramer's V strength shows that the *Excessive Class Length* and the *Too Many Public Methods* smells have a higher degree of association (V=0.2105) than the other smell pairs, which is still considered a weak association. Based on the results in Table 5, the *Too Many Public Methods* occurs with 50.92% of the total number of instances of the *Excessive Class Length*. The first 9 smell pairs, for which we rejected $H2_0$, have a weak degree of association based on the Cramer's V test. We can conclude that a high co-occurrence rate does not necessarily imply a high association degree. Although the *High Method Complexity* is highly co-occurring with *High NPath Complexity*, they do not have a strong degree of association.

Our reported analysis aims to depict any possible relationship between the different pairs of code smells. However, as by definition some code smells are in the class level, the co-occurrence is assumed to be highly witnessed in large classes. Thus, we have conducted a Kendall correlation test to examine the relation between the LOC and the number of code smells in the small, medium, and large classes. Figure 5 reports the correlation coefficients for the three groups. As shown, a weak correlation of <0.1 is found for the three groups. However, we see more code smells occurring in large classes, which could



(a) Small **(b)** Medium **(c)** Large

Fig. 5 Kendall correlation coefficients between LOC and number of code smells of small, medium, and large classes

somewhat introduce the class size as a play-factor in the co-occurrence phenomenon. To deeply decipher the reasons behind smells co-occurrence, in our future work, we will compute the odds that a non-class size-related smell occurs in large classes due to another smell and not to the class size.

Summary for RQ2 The co-occurrence phenomenon is highly diffused as an average of 60% of smelly classes have more than one smell. Some of the code smells could frequently co-occur in the server side and result into a causality effect (*e.g.*, *Excessive Class Length*, and *Too Many Methods*). However, their degree of association is generally weak. Code smells related to complexity and size of code components such as *Excessive Class Length* and *High Method Complexity* tend to have a higher degree of association. As well, our results depict the fact that method-level code smells could be the cause of class-level code smells.

4.3 RQ3: The impact of code smells on the change-proneness

Figure 6 displays the spectrum of code change sizes, *i.e.*, code churn, in both smelly and non-smelly groups using beanplots. We observe from Fig. 6 that smelly classes in the studied web applications have a clearly higher code change size than non-smelly classes. These findings indicate that code fragments affected with code smells may require increased maintenance efforts since they have higher code change-proneness. Specifically, the median of code change size that a smelly class could experience along the application evolution is 70, whereas non-smelly files experience an average code change size of only 30. To statistically assess the significance of the difference, we supported this result by computing the Mann–Whitney and the Cliff's delta effect size tests. The Mann–Whitney test has shown a significant difference between the two populations with a p-value$<$ 0.001 and a small effect size of 0.235, giving us statistical evidence to reject the null hypothesis $H3_0$. On average, smelly files are almost 2.3 times more subject to code changes than smells-free files. `WordPress`, `Piwik`, and `Symfony` projects have scored the highest difference in median of code changes in favor of smelly files with 294, 249, and 214, respectively. Similar to previous studies in Java object-oriented systems Khomh et al. (2012); Palomba et al. (2018b), we witnessed similar findings reported in Fig. 6, where smelly classes exhibit a clearly higher level of change-proneness compared to non-smelly classes. The smelly files have their maintenance requirements, which tends to be related to the coexistence of various types of code smells such as *High Method Complexity* and *Excessive Method Length*.

To further analyze the effect on the change-proneness of code smells, we investigated the change-frequency that reflects how likely are smelly files to change, *i.e.*, involved in commit changes. In particular, we would like to investigate how many times

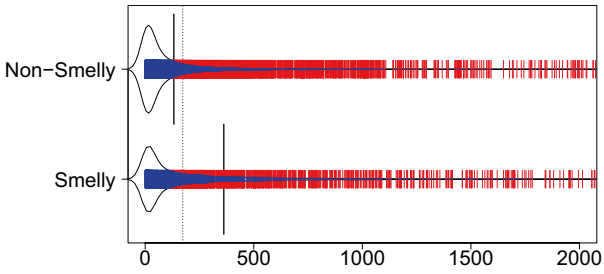


Fig. 6 Average of code changes of smelly and non-smelly classes (with a log-scale of 1-2000)

will a smelly file be changed as compared to non-smelly files between two consecutive releases. The beanplot in Fig. 7 indicates the difference between the number of commits, where a smelly file has been modified as compared to non-smelly files. Overall, we observe that smelly files are almost 1.4 more subject to get updated (with a median of 8) than non-smelly files (with a median of 6) along the project's evolution. Statistically, the Mann–Whitney proves the significant difference with a p-value< 0.001 and a large effect size of 0.674.

We observe from Fig. 8 the correlation coefficients of LOC and Churn for small, medium, and large smelly classes. In our samples data distribution, small classes have a median of 34 and 24 for LOC and Churn, respectively. Whereas medium classes have a median LOC of 155 and median Churn of 63, and large classes have 759 and 134 medians of LOC and Churn, respectively. To better understand the relationship between the class size and churn, we report in Fig. 8 the correlation results. We observe a very weak correlation between LOC and Churn for the three samples (0.159 for large, -0.08 for medium, and 0.068 for small). This result indicates there is no correlation between class size and churn. Thus, the change-proneness is not the result of class size, but rather this further supports our results as code churn is related to smells not to other factors (such as class size).

Summary for RQ3 Source code files containing code smells experience 2.3 times more code changes (median=70) than non-smelly files (median=30). Moreover, smelly files are 1.4 times more likely to get changed in commits than smell-free files. These findings indicate that code fragments affected with code smells may require increased maintenance efforts since they have higher code change-proneness.

4.4 RQ4: The impact of code smell types on the change-proneness

We used logistic regression to assess whether the existence of a smell is associated with the change-proneness of a class, *i.e.*, the state of a class {changed, unchanged}. We report the results of the logistic regression model in Table 8, where the column "% releases" indicates the percentage of releases with a $p\text{-}value < 0,05$ for which the corresponding smell type is statistically significant in the logistic model. Overall, we find that most projects are affected by smells, which increase their proneness to change, and this effect differs from a project context to another.

More specifically, we find that the smells *High Method Complexity (HMC)*, *Excessive Method Length (EML)* and *Too Many Methods (TMM)* do impact the change proneness of classes in 39%, 31% and 28% of releases, respectively. In particular, the HMC has shown an impact on 5 projects out of the 10 studied projects, namely `Piwik`, `WordPress`, `Moodle`, `phpBB` and `Symfony`. The EML has an impact on 3 projects, namely

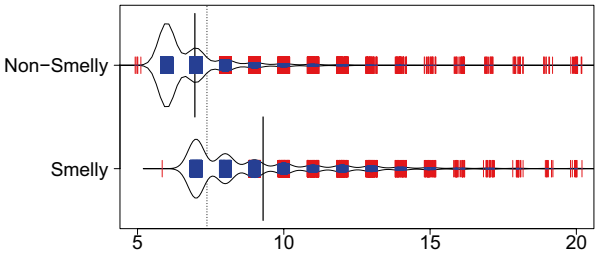
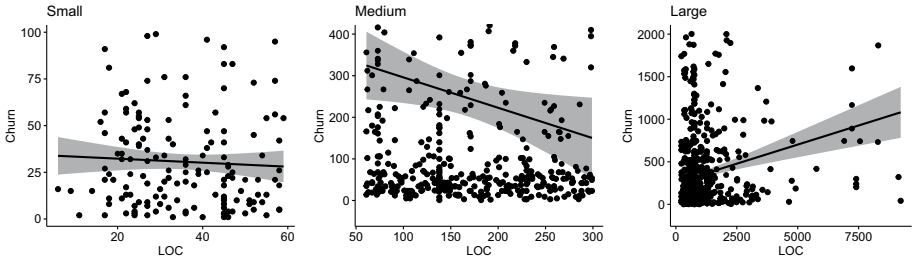


Fig. 7 Average of code change-frequency of smelly and non-smelly classes (with a log-scale of 5-20)



(a) Small **(b)** Medium **(c)** Large

Fig. 8 Kendall correlation coefficients between LOC and Churn of small, medium, and large classes

`phpMyAdmin`, `Moodle`, and `Cakephp`, while TMM impacted 4 projects, (`Joomla`, `Moodle`, `Symfony` and `phpMyAdmin`).

To better understand how the smells correspond most with the change-proneness of classes, we analyzed their probabilities of occurrence in the `Moodle` application, since its classes are the most susceptible to change according to 8 different kinds of code smells, as shown in Table 8. We found that the *High Method Complexity (HMC)*, the *Excessive Method Length (EML)*, the *High Coupling (HC)*, the *Too Many Public Methods (TMPM)*, and the *Too Many method (TMM)* are the most occurring five code smells with a probability of 0.34, 0.23, 0.11, 0.10 and 0.06 to occur in a class, respectively. The remaining code smells have a probability < 0.03. As well, we noticed that the *High Coupling (HC)* scored a higher probability to occur than the *Too Many method (TMM)* which, in general, impacts more releases (28%). Means, we have more instances of the *High Coupling (HC)* than the *Too Many method (TMM)*. It is important to note that high diffuseness does not necessarily depict high frequency in the affected projects. For instance, the *Excessive Class Length (ECL)* has shown a higher diffuseness rate (97% of the releases) and accounts for 2.7% of the total number of smells more than the *High Coupling (HC)* and the *Too Many method (TMM)*. However, it only has a 0.03 probability to occur in the classes of the `Moodle` application.

Table 8 The results of the logistic regression model reporting the number of releases for which each smell type is statistically significant (*p-value* < 0.05) over the studied releases and projects

Code smell	% releases	Projects
High Method Complexity	39%	Piwik, WordPress, Moodle, phpBB, Symfony
Excessive Method Length	31%	phpMyAdmin, Moodle, CakePHP
Too Many Methods	28%	Joomla, phpMyAdmin, Moodle, Symfony
High Coupling	23%	Piwik, WordPress, phpMyAdmin, Moodle
Too Many Public Methods	18%	WordPress, Moodle
Excessive Class Length	15%	phpMyAdmin, Moodle
High NPath Complexity	13%	Piwik, Cakephp
Excessive Parameter List	5.3%	Moodle, Symfony, phpBB
Empty Catch Block	3.1%	Moodle
Excessive Depth Of Inheritance	1.4%	Symfony
Excessive Number Of Children	~1%	–
Goto Statement	0%	–

On the other hand, we observed that other code smells such as the *Excessive Number of Children* and *Goto Statement* have a negligible impact on any release or project's change-proneness. This could be due to the fact that these smells are slightly diffused and slightly frequent, as observed in RQ1. We underline that (1) code smells with low diffuseness and frequency rate are not statistically proven to impact the change-proneness. Therefore, not all code smells should be given equal priority during source code refactoring and cleaning. The *GoTo Statement* and *Excessive Number Of Children*, are not seen as troublesome as they do not cause an increase in the number of code changes; (2) the code changes of the largest project `Moodle` is impacted by eight code smells. Long code components could increase the class complexity and, thus, makes it more vulnerable to code changes and eventually to code smells; (3) the diffuseness and frequency rate of smells does not necessarily reflect its ability of impact on the change-proneness. For example, The *High Method Complexity* smell is diffused in 99% of the releases (cf. Table 4), yet, its impact on the change-proneness is statistically significant, with only 5 out of the 10 studied projects (*i.e.*, 50% of the projects).

We can conclude that the effect of a code smell on the change-proneness differs based on the smell type and the project context. The existence of code smells is alarming as they increase the likelihood of experiencing a higher rate of code changes, particularly with *High Method Complexity*, *Excessive Method length*, *Too Many Methods* and *High Coupling*, which achieved the highest impact on the change-proneness in our experiments. For each studied project, at least one single smell is showing a significant impact. Thus, we reject the null hypothesis $H4_0$.

Summary for RQ4 The impact of code smells on the change-proneness varies from a smell type to another and from a project context to another. Notably, the *High Method Complexity* and *Excessive Method Length* have shown an increase on the change proneness of classes on more than 30% of the releases and 50% of the studied projects.

4.5 RQ5: The impact of code smells on the fault-proneness

As a preliminary study, we investigated the number of faults before and after the introduction of smells between every two consecutive releases. Figure 9 shows the beanplots of the number of fault-inducing commits before and after the smells are introduced. For each release R_j, we catch the fault-inducing commits (FIC) that take place after the introduction of the smells (*i.e.*, in the period between R_j and R_{j+1}), and before the introduction of the smells (*i.e.*, in the period between R_{j-1} and R_j). For each fault-fixing commit $FFC_{i=1}i \rightarrow n$, we count the number of FIC before and after the release R_j date.

We witnessed consistent behavior across our studied subjects, where faults are more likely to exist after code smells are introduced. We also found that the number of faults introduced in large and medium files, after the implementation of the smells, could be multiple compared to small files. As can be seen in Fig. 9, after the introduction of code smells, the number of faults in small classes is likely to increase 3 times (median of 6 faults after the introduction smells, compared to only 2 faults before the smells are introduced). Medium files are 5 times more prone to faults after the smells introduction (median of 1 fault before the smells introduction and 5 faults after the smells introduction. Finally, large files encounter a median of 7 faults after code smells introduction while a median of only 1 fault per file is observed before smells are introduced. Overall, the number of faults identified in our dataset before the introduction of code smells accounts for 24.7% as compared to 75.3% after the introduction of smells. To get more statistical evidence, we

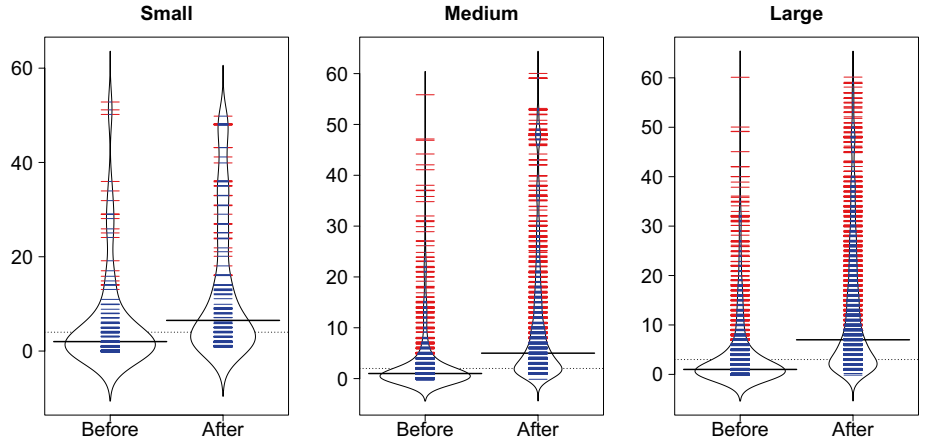


Fig. 9 The number of faults before and after the introduction of smells by class size

calculated the Mann–Whitney and the Cliff's delta tests to evaluate the differences between the samples. The small, medium and large smelly files demonstrated a statistically significant difference as compared to smell-free files with a p-value< 0.001. Both, small and large size smelly files exhibit a large effect-size (0.87 and 0.56, respectively), while medium size smelly files exhibit a negligible effect-size (0.17). Since we obtained a negligible effect size in the medium samples, we need to perform a statistical power analysis to avoid a type I (*i.e.*, rejecting a true null hypothesis) or type II (*i.e.*, Failing to reject a false null hypothesis) errors as suggested by Macbeth et al. (2011) and Cousineau and Domar (2007). We used the `wmwpow` R package to compute the probability of rejecting the hypothesis that both smelly and non-smelly medium files experience the same number of faults before and after the introduction of smells. Based on the distributions of both medium size smelly and non-smelly samples and the effect size, we obtained a power score of 0.673 to correctly assume that both samples are significantly different.

To examine the extent to which the trend of the file's smelliness and faultiness holds valid compared to non-smelly faulty files, we normalize the number of faults in both samples by the number of effective commits in a release. The first aspect to notice from the plots in Fig. 10 is that the assumption is still valid with the normalization of the number of faults. Second, we can see that we have a less dispersion of data (*i.e.*, less number of outliers) as compared to the distribution of non-normalized faults (Fig. 9), with an average difference of 0.002 for small and medium files and 0.003 for large files.

To assess the relationship between code smells and faults, we computed the number of fault-fixing commits in smelly and non-smelly classes. We removed all censored data covering groups that have zero fixing-commits. In Fig. 11, we witnessed a difference in the number of fixing-commits in favor of smelly classes. On average, small, medium and large smelly files are 1.6, 1.5, and 1.4 times subject to more fault-fixing commits, respectively, compared to non-smelly files. Overall, smelly files are likely to undergo 2 more fixing-commits than non-smelly files, regardless of their size. The same results are witnessed with the normalization of the number of fixing commits in favor of smelly classes as shown in Fig. 12. We can as well clearly see the difference in the number of fixing-commits between the smelly and non-smelly for the medium groups. Compared with the results observed with the change-proneness (RQ3), smelly files are more susceptible to code changes (a

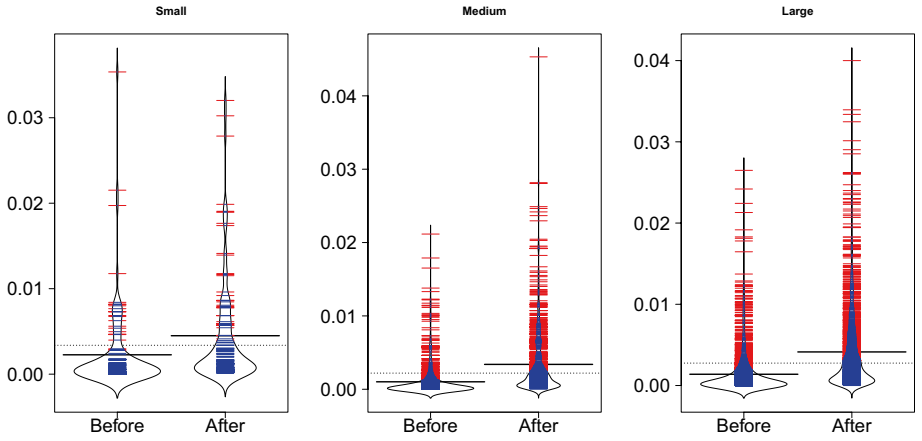


Fig. 10 Normalized number of faults before and after the introduction of smells by class

median code churn score of 70 for smelly files and 30 for non-smelly files) than faults. Our results confirm the findings by Khomh et al. (2012) and Palomba et al. (2018b) in Java traditional software applications, reporting that smelly files are more fault-proneness than other files and that the degree of the fault-proneness is less observed than with the change-proneness.

To get more statistical evidence on the fault-proneness phenomenon, we applied the Mann–Whitney, and Cliff's delta tests on the three groups of files (small, medium, large) of each application. To fix the ties observed in some of the data, we followed the methodology suggested by Conover (1973) proposing the ranking of the tied values. The results are reported in Table 9 for each individual project and for the combined dataset. We observe a statistically significant difference in the combined dataset for the three groups of smelly and non-smelly files (p-value<0.001) with a negligible effect size for small files, and a small

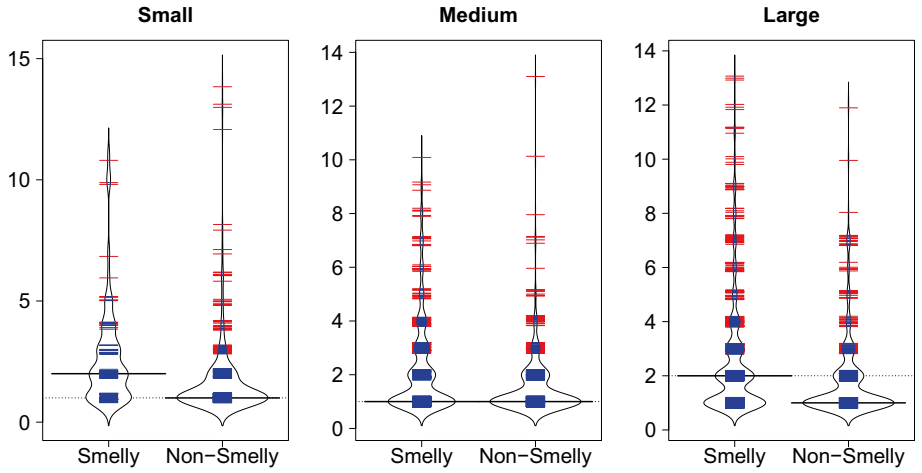


Fig. 11 The number of fixing-commits in smelly and non-smelly files by class size

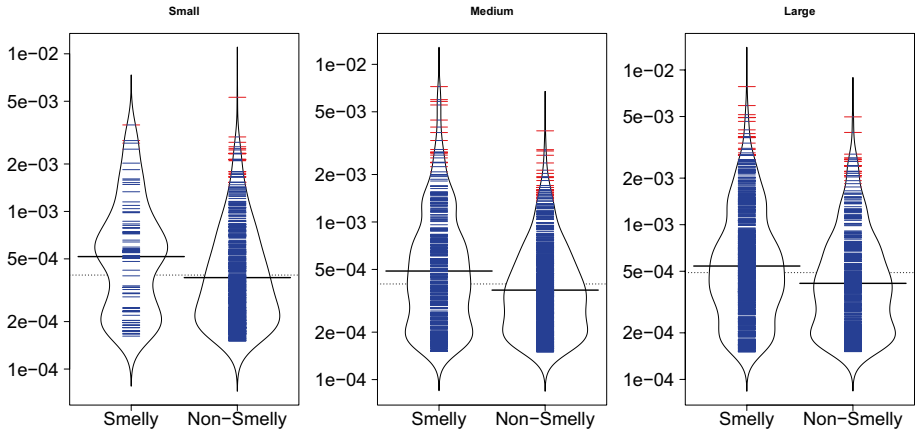


Fig. 12 Normalized number of fixing-commits in smelly and non-smelly files by class

effect size for both medium and large files. Hence, the fault-proneness phenomenon was statistically more significant in medium and large files, than small files. For example, the projects `Piwik` and `Symfony` display a statistically significant difference (p-value<0.001) in the three groups, with medium and large effect sizes (d=0.441 and 0.895, respectively). For other projects, such as `Wordpress` and `phpBB`, small size non-smelly files experienced more faults than smelly files. Similar findings are observed in the large size files in `phpMyAdmin, Moodle`, and `CodeIgniter`. However, there is no statistical significance in the `Wordpress` project. It is worth noting that, overall, in our combined dataset, 86% of the smelly files have a higher median of faults than smell-free files.

Although this analysis illustrates how smelly files encounter more faults than other classes, we did not see a substantial difference in the number of fault-fixing commits concerning file sizes. For this reason, we carried out another series of experiments to explore how the class size could impact (1) the fault perseverance in the system before it is fixed, and (2) the code churn deployed in the *fault-fixing commits*.

To investigate the propagation of faults in our studied projects, we count all commits that hold a particular fault #ID (we refer to as *commits-exhibiting a fault*) between the fault-inducing commit and the fault-fixing commit. As reported in Fig. 13, we observe that the highest number of CEF in the three groups exist in the smelly files (median of 2 in smelly files and a median of 1 in non-smelly files). Small size smelly files experience more CEF (median of 3 in smelly files and a median of 1 in non-smelly files) as opposed to large and medium smelly classes (median of 2 in smelly files and a median of 1 in non-smelly files). However, for some large files, faults could persist up to 60 commits without being fixed against 30 and 40 commits in small and medium size files, respectively. Thus, these findings indicate that faults in large code files tend to survive for a longer time. Overall, we found a significant difference of smelly files in the three groups (p-value<0.001) with a negligible effect size for small smelly files (d=0.043), and small effect sizes for medium and large smelly files (d=0.215, and d=283, respectively). The same results are witnessed when normalizing the number of CEF as shown in Fig. 14 These results provide insights that smelly and faulty code could be a tough combination to be managed efficiently during software maintenance tasks.

Table 9 The Mann–Whitney and Cliff delta tests results between smelly and non-smelly classes. The "-" describes certain situations where the small, medium or large samples contained only smelly or non-smelly files, for which tests cannot be conducted

		Mann–Whitney Test									**Cliff's delta**		
		Small			Medium			Large			Small	Medium	Large
		M	sd	P	M	sd	P	M	sd	P	d		
phpMyAdmin	S	–	–	–	1.625	1.746	**0.0242**	2.074	1.363	0.808	–	**0.822(L)**	-0.813(L)
	NS				1.6	0.736		2.230	1.535				
Joomla	S	1.5	0.940	**0.024**	1.433	0.747	0.641	1.709	1.216	**0.035**	**0.779(L)**	-0.041(n)	**0.703(L)**
	NS	1.1	0.316		1.309	0.539		1.214	0.425				
Wordpress	S	3.068	3.584	0.432	3	2.449	**<0.001**	3.795	3.346	**<0.001**	-0.816(L)	**0.601(L)**	**0.013(n)**
	NS	3.712	3.643		2.018	2.112		2.220	1.953				
Piwik	S	3.347	1.991	**<0.001**	3.416	2.2	**<0.001**	2.466	1.846	**<0001**	**0.549(L)**	**0.464(M)**	**0.832(M)**
	NS	1.896	2.559		1.843	1.517		1.725	1.456				
Laravel	S	1.5	0.707	**0.018**	1	0	0.838	1.555	0.726	**0.01**	**0.5(L)**	-0.153(S)	**0.307(S)**
	NS	1	0		1.083	0.288		1.333	0.577				
CakePHP	S	–	–	–	1.468	0.761	**0.041**	1.891	1.219	**<0.001**	–	**0.551(L)**	**0.368(M)**
	NS				1.185	0.427		1.358	0.603				
Moodle	S	3	3.091	**0.01**	1.480	1.255	**<0.001**	1.964	1.710	0.4	**0.503(L)**	**0.3(S)**	-0.684(L)
	NS	1.628	2.087		1.232	0.470		1.5	0.745				
Symfony	S	3	1.414	**<0.001**	1.536	1.142	**<0.001**	2.697	2.144	**0.01**	**0.735(L)**	**0.441(M)**	**0.895(L)**
	NS	1.547	0.971		1.409	0.721		2.033	1.376				
CodeIgniter	S	–	–	–	1.714	0.951	0.508	1.583	0.792	0.051	–	-0.583(L)	-0.973(L)
	NS				1.25	0.5		1.333	0.577				
phpBB	S	2.285	2.138	0.4	1.714	0.487	0.280	2.68	1.928	**0.042**	-0.228(S)	-0.142(n)	**0.249(S)**
	NS	2.187	2.286		1.470	0.624		1.928	1.591				
Combined	S	2.666	2.116	**<0.001**	2.232	1.8	**<0.001**	2.546	2.229	**<0.001**	**<0.001(n)**	**0.226(S)**	**0.266(S)**
	NS	1.639	1.493		1.522	1.077		1.812	1.484				

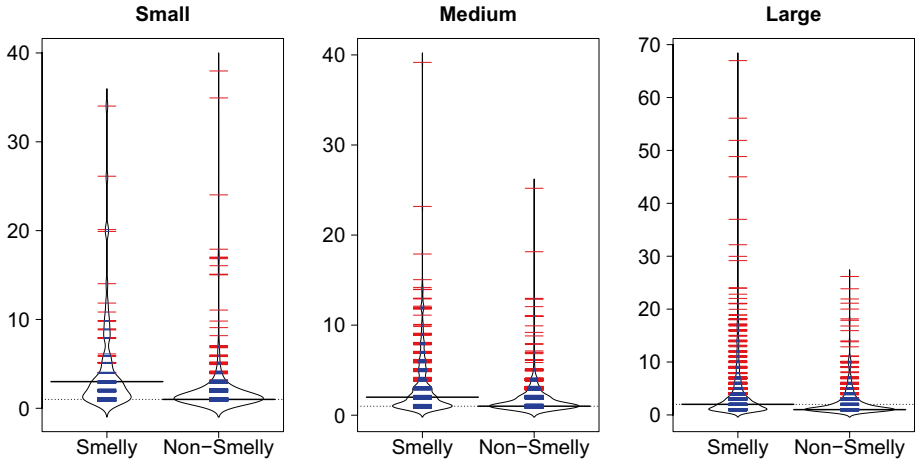


Fig. 13 Average of faults propagation in smelly and non-smelly files

Finally, to get a deeper understanding of the fault-proneness phenomenon, we investigate the required code churn in fault-fixing commits for both smelly and non-smelly files, as reported in Fig. 15. We observe a significant difference (p-value< 0.001) between smelly and non-smelly files in terms of code churn to fix faults in small, medium, and large files. A median code churn score of 152 (*i.e.*, added, removed, and modified lines of code) is witnessed in large size smelly files against 84 in smell-free large size files. The median code churn score for medium size files are 118 in smelly files and 59 in non-smelly files. Both medium and large files have a small effect size of d=0.259 and d=0.248, respectively. We also observe that small size smelly files undergo a significant magnitude of the phenomenon as smelly files have a median of 203.5 against 48 for non-smelly files. However, small size smelly files witness a negligible effect size ($d = 0.106$) as opposed to smell-free small size files. Overall, regardless of their size, smelly files witness higher code churn (with a median of 144) than non-smelly files (with a median of 62).

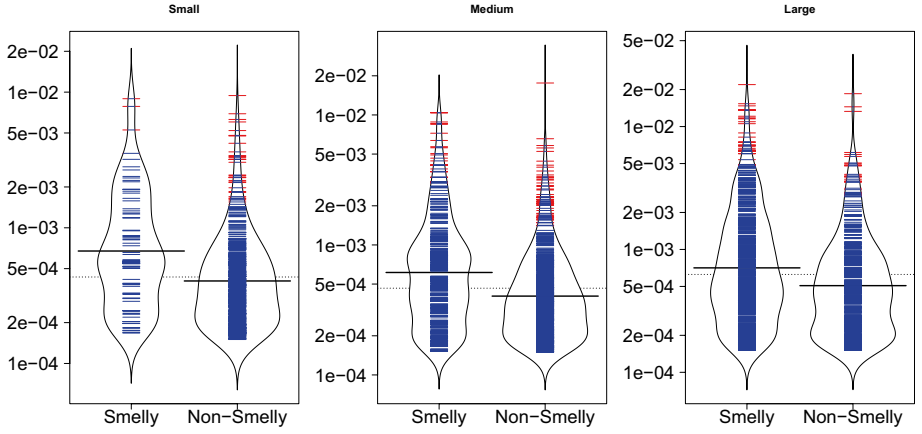


Fig. 14 Normalized average of faults propagation in smelly and non-smelly files

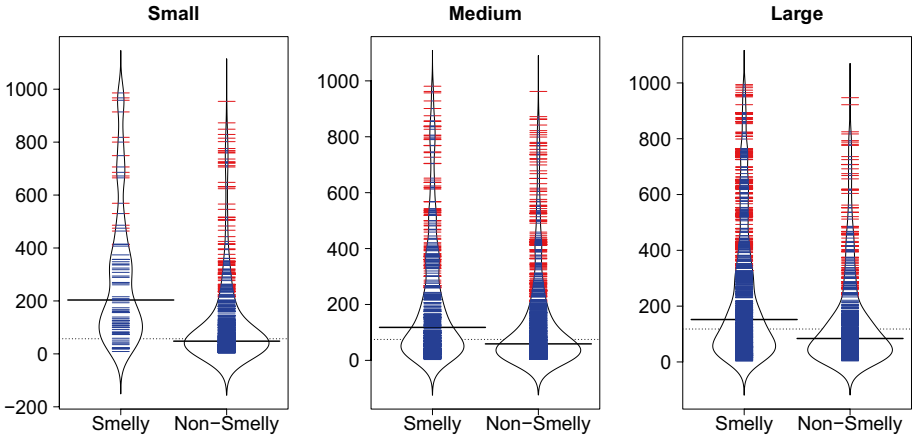


Fig. 15 Churn in fault-fixing commits in smelly and non-smelly files

Summary for RQ5 Smelly files are more fault-prone than non-smelly files. Overall, we observe that smelly files are (1) 6 times more prone to faults after the introduction of smells; (2) 2 times more likely to undergo fixing-commits; (3) 2 times more vulnerable to faulty commits; and(4) require 2.4 times more code churn in the fault-fixing commits. Same results were found when normalizing the history length of the releases.

5 Implications

Our study delivers several important messages and can have various implications for researchers, practitioners and educators.

5.1 Implications for practitioners

Prioritize code smells Since web applications are of a heterogeneous nature, fixing code smells is of paramount importance as a single failed class on the server side could crash the entire system and hinder the user experience and could be as costly for the company. Different practitioners, including developers, testers and tool builders involved in maintaining the code, can use our findings in the diffuseness, co-occurrences and impacts of code smells to focus their efforts on code smells with high severity and those related to the functional side of the system. For instance, diffused code smells (RQ1) co-occurring code smells (RQ2), as well as smells having high impact on the change-proneness (RQ3 and RQ4) and the fault-proneness (RQ5) should be given higher priority during the different maintenance tasks.

Avoid long and complex code fragments Long and complex coding practices (*e.g.*, *Excessive Method Length* and *High Method Complexity* code smells) are the most commonly diffused code smells in our studied systems, with more than 90% of releases affected, as shown in RQ1. Such particular smells have shown a significant effect on the

change-proneness. Developers are often overwhelmed with additional changes to maintain classes containing these particular types of smells. Therefore, it is recommended to refactor, *i.e.*, split, long code elements into sub-classes or sub-methods to facilitate future maintenance tasks. In particular, in the context of web applications, classes affected by *High Coupling (HC)* code smells may need special care, since web applications architectures already interconnect many different components.

Prioritize testing of smelly and frequently modified files The results of our study show that code smells do increase the change- and fault-proneness of source files. We stressed that smelly files are subject to further code changes and that different types of smells do not generally lead to the same effect on the change- and fault-proneness. Therefore, software testers are encouraged to improve and prioritize the testing of smelly files, as an attempt to avoid their impacts and risks.

Build automated code smell-aware fault fixing tools The findings of RQ4 and RQ5 showed that code smells have a significant effect on the number of faults, which may lead to higher production costs, increased maintenance activities, and potential failures. Tool builders are, therefore, recommended to establish specialized code smell-aware fault-fixing tools that are specific to a web app architecture that helps and reduces the effort, time and cost of fixing different types of faults in smelly files.

5.2 Implications for researchers

Investigate the relationship between code smells The co-occurrence of code smells (RQ2) may represent a first step toward examining potential relationship between code smells, which involves a sharp distinction between code smells in the class and method levels. Our findings suggest a high co-occurrence rate between some code elements having a high complexity and large size. Researchers could use and extend our publicly available dataset Dataset (2020) to perform an in-depth analysis of code smell co-occurrences that would provide more insights into the relationships and degree of association between code smells based on code changes and faults dependencies. Such analysis may provide efficient and customized refactoring recommendation and/or change impact analysis tools for web applications.

Plan refactoring actions The findings of our study clearly emphasize that web developers need better control of code smells. In particular, as shown in RQ4 and RQ5, at least 4 code smell types including *High Coupling*, *Excessive Method Length*, *Too Many Methods*, and *High Method Complexity* have shown a high effect on the change- and fault-proneness. The maintenance efforts and costs could benefit from the removal of such code smells. Researchers are therefore encouraged to develop efficient refactoring strategies to help eliminate such code smells, particularly, in the multi-layer architecture of the web-app which requires efforts to localize the source of a propagated fault. These refactoring solutions will support improving the overall quality and maintainability of the software system.

Explore the impact of individual code smells on maintainability aspects Code smells have a negative impact on the change- and fault-proneness. However, files could be affected by one or several code smells at the same time, which may bring different levels of change- and fault-proneness. Hence, a first step toward evaluating the severity of code smell types

is to perform further analysis to investigate how the change- and fault-proneness of affected files could vary when considering different code smells types and instances in the same files or (sub-)modules.

5.3 Implications for educators

Teach a clean code "culture" The results of our study can be valuable for educators teaching design and implementation principles. *Educators* need to implant a clean code "culture" by providing students with examples of how bad coding practices can lead to significant effects. Typical courses about software design, programming and/or software quality would introduce the topic of code smells and illustrate the main impacts on change- and fault-proneness to increase the awareness of students on the harmfulness of code smells. Students need to also learn how to solve code smells by example and know how important it is to eradicate them through following good design and programming practices. In the context of web applications, in particular, we expect the impact of code smells on the sustainability of the entire application to be severe and more burdening. As web applications, by nature, interconnect multiple components, code smells or faults may seriously jeopardize or harden the development and maintenance and cause major failures.

6 Threats to validity

Construct threats to validity concerns errors in measurements. To collect our dataset, we relied on the detection accuracy of PHPMD tool PHPMD (2021), which is an open-source tool specialized for PHP web-based software applications providing a command line option that is adequate to our analysis, and widely-used in previous studies on web applications Rio and Abreu (2019); Mon et al. (2019); Mon and Myo (2015); Yulianto and Liem (2014). However, there could still be some errors in the smells detection. Another potential threat to validity could be related to the analysis of fault-fixing commits using keywords. While this technique has been widely-used in prior studies Saboury et al. (2017); Spadini et al. (2018); Palomba et al. (2018b, 2019, 2018a), it could not be free of false positives when finding fault-inducing commits. To mitigate this issue, we randomly selected and inspected a set of 60 fault-fixing commits and their related fault-inducing commits and found only five false positives.

Conclusion threats to validity could be related to the data analysis to draw our conclusions. To analyze the diffuseness of code smells, we followed a simple method counting the absolute number of code smells instances in all the smelly classes in our examined 430 releases. To provide statistical evidence, we used the Mann–Whitney test and Cliff's delta effect size along with a beanplot representation that displays the density of code smells instances. To analyze the co-occurrence phenomenon of code smells, we computed the total number of co-occurring pairs and used other descriptive statistical tests to indicate the discrepancy between the degree of association and co-occurrence. Furthermore, in our investigation of fault-proneness, we analyzed the impact of code smells on three variables related to fixing-commits, code churn, and the number of faulty-commits where faults manifest. The Mann–Whitney and Cliff's delta statistical tests quantified the difference observed between smelly and non-smelly class change- and fault-proneness. However, still there could be errors that we did not notice. The emphasis of our research on the server

side of web applications can represent another potential threat. Our analysis could therefore provide insights that do not reflect all quality aspects of a web application. As part of our future work, we plan to consider other programming languages, formatting languages (*e.g.*, Cascading Style Sheets (CSS) Mazinanian et al. (2014)), and multi-language web applications, and also investigate the impact of code smells in the client-side.

Internal threats to validity concern the factors that could restrict the applicability of our observations or affect our conclusions. In RQ2, we do not explicitly assume that strongly co-occurring code smells result from a relationship of causality. We cannot tell if a given smell is the reason behind another smell introduction. To further explore the phenomenon of co-occurrence, we combined our research with more statistical analyses. Our observations regarding the impact of code smells on the change- and fault-proneness could not involve a direct cause-effect relationship as this latter could be affected by other factors such as refactoring or improvement activities. As part of our future work, we plan to explore the potential existence of direct cause-effect relationships between code smells in web applications. Our obtained results regarding the co-occurrence depict the possible impact of class size on the co-occurring code smells mostly related to size and complexity. Hence, as part of our future work, we plan to investigate code smells co-occurrence considering the class size, as well as different levels of co-occurring code smells (3 pairs, 4 pairs, etc.).

External threats to validity concern the generalization of our findings. To the best of our knowledge, this is the first empirical analysis conducted on code smells diffuseness, co-occurrence, and impact in web-based applications. While we considered 430 releases from 10 web-based projects, and 12 common types of code smells, we cannot generalize our results to other projects and other contexts. As part of our future work, we plan to reduce this threat further by analyzing more projects from more industrial and open-source web based software projects. We also plan to conduct a survey with developers to learn more about their awareness of the different impacts of code smells in their projects and investigate potential impacts on other software development aspects.

7 Conclusion

Code smells are symptoms of poor design choices, which typically lead to higher resource consumption and systems that are harder to maintain. Given the growth and importance of web applications, in this paper, we presented a longitudinal exploratory study on the server side of 430 releases from 10 long-lived web applications to investigate (1) the diffuseness and co-occurrence of 12 common types of code smells, (2) the impact of code smells on the maintainability aspects of server side code in terms of change- and fault-proneness. Our study delivers several important findings. The results indicate that the phenomenon of code smells diffuseness and co-occurrence is highly observed in web applications server side. Some specific code smells are diffused in nearly 99% of releases, such as the *High Complexity method*. We also found that nearly 60% of smelly classes have more than one smell instance. For instance, the *High Method Complexity* often co-occurs with five smell types, including *Excessive Method Length* and *Excessive Class Length*. Our results also indicate that code smells increase the probability that a smelly class will undergo more code changes than non-smelly classes. However, not all types of smells have the same effect on the change-proneness and the impact of a smell has been shown to depend on the

context of the project. As for the fault-proneness, our results highlight an increased vulnerability of smelly files to faults as compared to smells-free files. In particular, smelly classes are subject to an average of 6 more faults after code smells introduction leading to 2 times higher number of fault-fixing activities and 2.4 times more code-churn to fix a fault as compared to smell-free files.

As future work, we plan to further investigate the impact of code smells on software maintainability by (1) studying the severity of code smells and examine how their impact could propagate from a layer/component to another in web architectures; (2) deciphering the relationship between code smells and faults to develop smell-aware fault localization techniques; (3) studying the survivability of code smells and faults in web applications server side; and (4) further investigating the number and type of code smells on the change- and fault-proneness. C3.2:We also plan to cover a broader set of code smells affecting both the front- and back-ends. Another interesting direction is to compare our set of analyzed web-applications with others that incorporate "Micro-Frontends". This latter consists of dividing the front-end application into smaller pieces to ease its development. The aim is to examine the extent to which adopting a micro-frontend approach could improve code quality in terms of code smells and bugs.

References

Abbes, M., Khomh, F., Gueheneuc, Y.- G., & Antoniol, G. (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. *In European Conference on Software Maintenance and Reengineering*, 181–190.

Agrawal, R., Imieliński, T., & Swami, A. (1993). Mining association rules between sets of items in large databases. *In ACM SIGMOD international conference on Management of data*, 207–216.

Agrawal, R., Srikant, R., et al. (1994). Fast algorithms for mining association rules. *In 20th international conference on very large data bases, VLDB*, 1215, 487–499.

Antoniol, G., Ayari, K., Di Penta, M., Khomh, F., & Guéhéneuc, Y. -G. (2008). Is it a bug or an enhancement? a text-based approach to classify change requests. *In Conference of the center for advanced studies on collaborative research: meeting of minds*, 304–318.

Muse, B. A., Rahman, M. M., Nagy, C., Cleve, A., Khomh, F., & Antoniol, G. (2020). On the prevalence, impact, and evolution of sql code smells in data-intensive systems. *In International Conference on Mining Software Repositories (MSR)*.

Bessghaier, N., Ouni, A., & Mkaouer, M. W. (2020). On the Diffusion and Impact of Code Smells in Web Applications. *In International Conference on Services Computing (SCC)*, 67–84.

Borg, M., Svensson, O., Berg, K., & Hansson, D. (2019). Szz unleashed: an open implementation of the szz algorithm-featuring example usage in a study of just-in-time bug prediction for the jenkins project. *In 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, 7–12.

Brin, S., Motwani, R., Ullman, J. D., & Tsur, S. (1997). Dynamic itemset counting and implication rules for market basket data. *In ACM SIGMOD international conference on Management of data*, 255–264.

Chatzigeorgiou, A., & Manakos, A. (2010). Investigating the evolution of bad smells in object-oriented code. *In Seventh International Conference on the Quality of Information and Communications Technology*, 106–115.

Conover, W. J. (1973). On methods of handling ties in the wilcoxon signed-rank test. *Journal of the American Statistical Association, 68*(344), 985–988.

Conover, W. J. (1998). *Practical nonparametric statistics, 350*. John Wiley & Sons.

Cousineau, T. M., & Domar, A. D. (2007). Psychological impact of infertility. *Best Practice & Research. Clinical Obstetrics & Gynaecology, 21*(2), 293–308.

Cramir, H. (1946). *Mathematical methods of statistics* (p. 500). Princeton U. Press: Princeton.

Da Costa, D. A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., & Hassan, A. E. (2016). A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering, 43*(7), 641–657.

Dataset. (2020). https://www.github.com/stilab-ets/CodeSmells_WebApps.
Delchev, M., & Harun, M. F. (2015). Investigation of code smells in different software domains. *Full-scale Software Engineering, 31*.
Fontana, F. A., Ferme, V., & Zanoni, M. (2015). Towards assessing software architecture quality by exploiting code smell relations. *In Second International Workshop on Software Architecture and Metrics*, 1–7.
Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.
Garg, A., Gupta, M., Bansal, G., Mishra, B., & Bajpai, V. (2016). Do bad smells follow some pattern? *In International Congress on Information and Communication Technology*, 39–46.
Grissom, R. J., & Kim, J. J. (2005). *Effect sizes for research: A broad practical approach*. Lawrence Erlbaum Associates Publishers.
Group, I., et al. (2010). Ieee standard classification for software anomalies. IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993) *104*(2), 1–23.
Habchi, S., Rouvoy, R., & Moha, N. (2019). On the survival of android code smells in the wild. In *IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 87–98.
Hamdi, O., Ouni, A., AlOmar, E. A., Ó Cinnéide, M., & Mkaouer, M. W. (2021). An empirical study on the impact of refactoring on quality metrics in android applications. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 1–12.
Hecht, G., Benomar, O., Rouvoy, R., Moha, N., & Duchien, L. (2015). Tracking the software quality of android applications along their evolution (t). In *International Conference on Automated Software Engineering (ASE)*, pp. 236–247.
Hosmer, D. W., Lemeshow, S., & Cook, E. (2000). *Applied logistic regression* (2nd ed.). NY: John Wiley & Sons.
Kampstra, P., et al. (2008). Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, 28*(1), 1–9.
Kendall, M. G. (1938). A new measure of rank correlation. *Biometrika, 30*(1/2), 81–93.
Khomh, F., Di Penta, M., & Gueheneuc, Y. G. (2009). An exploratory study of the impact of code smells on software change-proneness. In *Working Conference on Reverse Engineering*, pp. 75–84.
Khomh, F., Di Penta, M., Guéhéneuc, Y. G., & Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering, 17*(3), 243–275.
Kienle, H., & Distante, D. (2014). Evolution of web systems. In *Evolving Software Systems*.
Kim, S., Zimmermann, T., Pan, K., James Jr, E., et al. (2006). Automatic identification of bug-introducing changes. In *IEEE/ACM International Conference on Automated Software Engineering*, pp. 81–90.
Lenarduzzi, V., Palomba, F., Taibi, D., & Tamburri, D. A. (2020). Openszz: A free, open-source, web-accessible implementation of the szz algorithm. In *International Conference on Program Comprehension (ICPC)*.
Lenarduzzi, V., Saarimäki, N., & Taibi, D. (2019). The technical debt dataset. In *International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 2–11.
Liu, X., & Zhang, C. (2017). The detection of code smell on software development: a mapping study. In *International Conference on Machinery, Materials and Computing Technology (ICMMCT 2017)*, Atlantis Press.
Macbeth, G., Razumiejczyk, E., & Ledesma, R. D. (2011). Cliff's delta calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica, 10*(2), 545–555.
Mannan, U. A., Ahmed, I., Almurshed, R. A. M., Dig, D., & Jensen, C. (2016). Understanding code smells in android applications. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 225–236.
Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.
Mazinanian, D., Tsantalis, N., & Mesbah, A. (2014). Discovering refactoring opportunities in cascading style sheets. In *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 496–506.
McHugh, M. L. (2013). The chi-square test of independence. *Biochemia medica: Biochemia medica, 23*(2), 143–149.
Mon, C. T., Hlaing, S., Tin, M., Khin, M., Lwin, T. M., & Myo, K. M. (2019). Code readability metric for php. In *International Conference on Consumer Electronics*, pp. 929–930.
Mon, C. T., & Myo, K. M. (2015). A practical model for measuring code complexity of php. In *Thirteenth International Conferences on Computer Applications (ICCA 2015)*.
Olbrich, S., Cruzes, D. S., Basili, V., & Zazworka, N. (2009). The evolution and impact of code smells: A case study of two open source systems. In *International symposium on empirical software engineering and measurement*, pp. 390–400.
Olbrich, S. M., Cruzes, D. S., & Sjøberg, D. I. (2010). Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *IEEE International Conference on Software Maintenance*, pp. 1–10.

Ouni, A., Kessentini, M., Bechikh, S., & Sahraoui, H. (2015a). Prioritizing code-smells correction tasks using chemical reaction optimization. *Software Quality Journal, 23*(2), 323–361.

Ouni, A., Kessentini, M., Inoue, K., & Cinnéide, M. O. (2017). Search-based web service antipatterns detection. *IEEE Transactions on Services Computing, 10*(4), 603–617.

Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., & Deb, K. (2016). Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology, 25*(3), 1–53.

Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., & Hamdi, M. S. (2015b). Improving multi-objective code-smells correction using development history. *Journal of Systems and Software, 105,* 18–39.

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., & De Lucia, A. (2018a). A large-scale empirical study on the lifecycle of code smell co-occurrences. *Information and Software Technology, 99,* 1–10.

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., & De Lucia, A. (2018b). On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering, 23*(3), 1188–1221.

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A., & De Lucia, A. (2019). On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology, 105,* 43–55.

Palomba, F., Oliveto, R., & De Lucia, A. (2017). Investigating code smell co-occurrences using association rule learning: A replicated study. In *IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pp. 8–13.

Pecorelli, F., Palomba, F., Khomh, F., & De Lucia, A. (2020). Developer-driven code smell prioritization. In *International Conference on Mining Software Repositories*.

PHPMD. (2021).PHP Mess Detector, available at https://phpmd.org

Piatetsky-Shapiro, G. (1991). Discovery, analysis, and presentation of strong rules. *Knowledge discovery in databases*, 229–238.

Planning, S. (2002). *The economic impacts of inadequate infrastructure for software testing*. National Institute of Standards and Technology.

Rio, A., & e Abreu, F. B. (2019). Code smells survival analysis in web apps. In *International Conference on the Quality of Information and Communications Technology*, Springer, pp. 263–271.

Rodríguez-Pérez, G., Robles, G., & González-Barahona, J. M. (2018). Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. *Information and Software Technology, 99,* 164–176.

Rodríguez-Pérez, G., Robles, G., Serebrenik, A., Zaidman, A., Germán, D. M., & Gonzalez-Barahona, J. M. (2020). How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering*, 1–47.

Rossi, G., Pastor, O., Schwabe, D., & Olsina, L. (2007). *Web engineering: modelling and implementing web applications*. Springer Science & Business Media.

Saboury, A., Musavi, P., Khomh, F., & Antoniol, G. (2017). An empirical study of code smells in javascript projects. In *International conference on software analysis, evolution and reengineering (SANER)*, pp. 294–305.

Saidani, I., Ouni, A., Mkaouer, M. W., & Palomba, F. (2021). On the impact of continuous integration on refactoring practice: An exploratory study on travistorrent. *Information and Software Technology*, 106618.

Śliwerski, J., Zimmermann, T., & Zeller, A. (2005). When do changes induce fixes? *ACM sigsoft software engineering notes, 30*(4), 1–5.

Spadini, D., Aniche, M., & Bacchelli, A. (2018). Pydriller: Python framework for mining software repositories. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 908–911.

Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., & Bacchelli, A. (2018). On the relation of test smells to software code quality. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 1–12.

Statistics, P. https://w3techs.com/technologies/overview/programming_language. Accessed Jul 7 2020.

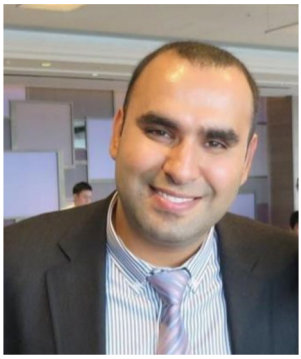
Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., & Poshyvanyk, D. (2015). When and why your code starts to smell bad. *International Conference on Software Engineering, 1,* 403–414.

Williams, C., & Spacco, J. (2008). Szz revisited: verifying when changes induce fixes. In *Workshop on Defects in large software systems*, pp. 32–36.

Yulianto, S. V., & Liem, I. (2014). Automatic grader for programming assignment using source code analyzer. In *International Conference on Data and Software Engineering (ICODSE)*, pp. 1–4.

Zimmermann, T., Premraj, R., & Zeller, A. (2007). Predicting defects for eclipse. In *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07: ICSE Workshops 2007), IEEE*, pp. 9–9.

Narjes Bessghaier is a PhD student at Ecole de technologie superieure (ETS), Montreal, University of Quebec. She obtained her MSc and BSc degrees from the University of Sfax, in 2017 and 2015, respectively. Her research interests include software engineering, software quality, mining software repositories, empirical software engineering, and web-based software systems.



Ali Ouni is an Associate Professor in the Department of Software Engineering and IT at Ecole de technologie superieure (ETS), University of Quebec, Canada. He received his Ph.D. degree in computer science from University of Montreal in 2014. Before joining ETS Montreal, he has been an assistant professor at Osaka University, Japan, and UAE University. For his exceptional Ph.D. research productivity, he was awarded the Excellence Award from the University of Montreal. He has served as a visiting researcher at Missouri University of Science and Technology, and University of Michigan, in 2013 and 2014 respectively. His research interests are in software engineering including software maintenance and evolution, refactoring of software systems, software quality, service-oriented computing, and the application of artificial intelligence techniques to software engineering. He served in organization and program committee, and reviewer in several journals and conferences.

Mohamed Wiem Mkaouer is currently an Assistant Professor in the Software Engineering Department, in the B. Thomas Golisano College of Computing and Information Sciences at the Rochester Institute of Technology. He received his PhD in 2016 from the University of Michigan-Dearborn. His research interests include software quality, systems refactoring, model-driven engineering and software testing. His current research focuses on the use computational search and evolutionary algorithms to address several software engineering problems such as software quality, software remodularization, software evolution and bug management. He served in various conferences including the International Conference on Program Comprehension (ICPC), the International Conference on Software Analysis, Evolution and Reengineering (SANER), the Association for Computing Machinery's (ACM) Special Interest Group on Computer science education (SIGCSE). he is also reviewers for various journals including the IEEE Transactions on Software Engineering (TSE), the ACM Transactions on Software Engineering andMethodology (TOSEM), the Empirical Software Enineering Journal (EMSE).