On the Diffusion and Impact of Code Smells in Web Applications

Chapter · September 2020

DOI: 10.1007/978-3-030-59592-0_5

3 authors:

Narjes Bessghaier
École de Technologie Supérieure
7 PUBLICATIONS **38** CITATIONS
SEE PROFILE

Ali Ouni
Osaka University
155 PUBLICATIONS **3,685** CITATIONS
SEE PROFILE

Mohamed Wiem Mkaouer
Rochester Institute of Technology
168 PUBLICATIONS **2,256** CITATIONS
SEE PROFILE

On the Diffusion and Impact of Code Smells in Web Applications

Narjes Bessghaier[1](✉), Ali Ouni[1], and Mohamed Wiem Mkaouer[2]

[1] Ecole de Technologie Superieure (ETS), University of Quebec, Montreal, QC, Canada
narjes.bessghaier.1@ens.etsmtl.ca, ali.ouni@etsmtl.ca
[2] Rochester Institute of Technology (RIT), Rochester, NY, USA
mwmvse@rit.edu

Abstract. Web applications (web apps) have become one of the largest parts of the current software market over years. Modern web apps offer several business benefits over other traditional and standalone applications. Mainly, cross-platform compatibility and data integration are some of the critical features that encouraged businesses to shift towards the adoption of Web apps. Web apps are evolving rapidly to acquire new features, correct errors or adapt to new environment changes especially with the volatile context of the web development. These ongoing amends often affect software quality due to poor coding and bad design practices, known as *code smells* or *anti-patterns*. The presence of code smells in a software project is widely considered as form of technical debt and makes the software harder to understand, maintain and evolve, besides leading to failures and unforeseen costs. Therefore, it is critical for web apps to monitor the existence and spread of such anti-patterns. In this paper, we specifically target web apps built with PHP being the most used server-side programming language. We conduct the first empirical study to investigate the diffuseness of code smells in Web apps and their relationship with the change proneness of affected code. We detect 12 types of common code smells across a total of 223 releases of 5 popular and long-lived open-source web apps. The key findings of our study include: 1) complex and large classes and methods are frequently committed in PHP files, 2) smelly files are more prone to change than non-smelly files, and 3) *Too Many Methods* and *High Coupling* are the most associated smells with files change-proneness.

Keywords: Code smells · Web applications · PHP · Diffuseness · Change proneness

1 Introduction

Web applications as defined by Google[1] are: *"...modern web capabilities to deliver an app-like user experience..."*. Web apps are characterized by their inherent

[1] https://developers.google.com/web/updates/2015/12/getting-started-pwa.

Q. Wang et al. (Eds.): SCC 2020, LNCS 12409, pp. 67–84, 2020.
https://doi.org/10.1007/978-3-030-59592-0_5

heterogeneous nature in (1) target platforms as web apps are usually split in 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 applications unique and different than traditional software systems.

Like any software application, web apps evolve rapidly to add new users functionalities, fix bugs, and adapt to new environmental changes. Such frequent and unavoidable changes, in the volatile context of the web development, can alter the quality of these applications. Indeed, acknowledged software design principles and practices are needed to be in place and empowered to support web apps development life-cycle. However, as software decays, some bad design and implementation practices may appear, which are known as *code smells* or *anti-patterns* [4,8]. Code smells are symptoms of poor design and implementation choices applied by developers that may hinder the comprehensibility and maintainability of software systems. Common code smells include, large classes, long methods, long parameter list, high coupling, complex class, etc. [4,8].

Several research efforts have been dedicated to studying code smells in traditional desktop software systems. It refers to bad coding practices that are committed mostly without the developers' knowledge [35]. Some studies focused on analyzing how code smells are introduced in the codebase [2,34,35], and how long they persist in the system [28,34]. Other studies focused on the impact of code smells on systems change and fault-proneness [21,32], and whether developers perceive these smells as problematic [2]. However, little is known about code smells diffuseness and impact for web applications. We cannot assume without empirical evidence the applicability of the prior findings on web apps as they widely differ. A dynamic web application package may encompass different technologies as JS, HTML, CSS, and PHP combining both programming and formatting aspects, which unlock another dimension of complexity, in comparison with traditional desktop applications. For example, web apps support combining code fragments, allowing to code PHP or JS inside HTML pages and vice versa. This coding practice emerges new types of code smells violating the *separation of concerns* design principle [30]. On a technical level, heterogeneous and dynamic web apps are more complex. Intensive computing tasks are performed to deal with databases and the HTTP client side's requests, which require more coding and maintenance efforts jeopardizing their quality and performance.

In this paper, we conduct the first empirical study on the diffuseness of code smells in web applications and investigate the impact of code smells on the source code change-proneness, *i.e.*, to investigate whether smelly files tend to require a higher frequency of changes when updating the files, as a side effect of their infection with bad programming practices. Moreover, we individually investigate how each smell type can contribute differently to the change-proneness. In particular, we focus our study on PHP-based web applications, being the top programming language used in server-side applications development. Indeed, nearly 79% of web apps are using PHP[2]. However, despite the popularity of the language in

[2] https://w3techs.com/technologies/overview/programming_language.

web development, no previous studies have empirically examined the behavior of code smells and how they impact the system's maintainability. To conduct our empirical study, we mined the historical changes of 223 releases from 5 popular web projects, phpMyAdmin, Joomla, WordPress, Piwik, and Laravel, to detect the existence of 12 common code smell types. The study provides empirical evidence that files containing code smells are more susceptible to change than non-smelly files, which negatively hinders the development of web apps, when containing code smells, as developers spend a larger amount of time and effort to update them. Results show that, at least, smelly-files are 2 times more prone to changes. Specifically, developers tend to write long and complex methods which make the code more hard to understand and modify. The obtained results indicate that the *High Method Complexity* and the *Excessive Method Length* code smells are frequently committed in PHP files. On the other hand, other smells such as the *Too Many Methods* and *High Coupling* are not frequently occurring, but they are the most smells leading to higher change-proneness. Findings from this study provide empirical evidence for practitioners that detecting and assessing code smells impact is of paramount importance to effectively reduce maintenance costs, as well as for the research community to concentrate their refactoring efforts on most harmful code smells. Further, this study serves as a first step to assess the magnitude of the severity of change-proneness related smells compared to other factors such as the number of occurrences of the smell and a class fault-proneness. We encourage the community to further harvest the data we collected by publishing our dataset for replication and extension purposes [1].

The rest of the paper is structured as follows. Section 2 presents the related literature on the diffuseness and impact of code smells. Section 3 describes the design of our empirical study. Section 4 presents and discusses the main findings. Threats to validity are discussed in Sect. 5. We conclude and highlight our main future research directions in Sect. 6.

2 Related Work

A number of studies exist on code smells in traditional software systems. We divide the existing works on 2 main categories (*i*) studies on the diffuseness and evolution of code smells, and (*ii*) studies on the impact of code smells.

2.1 Diffuseness and Evolution of Code Smells

There exists little knowledge of code smells in web applications. Recently, Rio et al. [31] targeted the survival probability of six code smell types using PHPMD[3], a PHP-based code smells detection tool, in 4 web applications. They considered three *scattered* smells (concern coupled entities) and three *localised* smells (concern an entity in itself). The findings did not show consistent behavior across the four systems. The survivability rate varies with the type of smells

[3] https://phpmd.org.

for each application. However, the introduction and removal events are higher in favor of *localised* smells. This can be explained by how code practices coupling several system components are harder to maintain. Nguyen et al. [19] proposed a detection tool *WebScent* of six kinds of the so-called embedded code smells that violate three design principles (*separation of concerns, software modularity, and compliance with coding standards*). The approach consisted of detecting code smells in the portions of PHP scripts responsible for generating the client-side code. The analysis highlighted that up to 81% of server files suffer from embedded code smells. Consequently, these files have a lower quality than smell-free files. Ouni et al. [22,24] introduced an automated approach to detect Web service anti-patterns in WSDL-based Web services.

However, to the best of our knowledge, no study has investigated the diffuseness of code smells in Web server-side projects and their relationship with development activities. Thus, we present the related literature on code smells in other programming languages. Palomba et al. [28] have investigated the diffuseness of code smells in desktop software systems and found that code smells related to large and complex code are most persistent in the system. They also investigated the correlation between the smell types and systems characteristics (*e.g.*, number of classes, number of methods, and lines of code LOC). Code smells are indeed diffused in large systems. Interestingly, this correlation does hold with smell types representing the more functional and sophisticated side of the system as like *Long method* and *complex class*. Olbrich et al. [20] analyzed the evolution of *God Class*, and *Shotgun Surgery* code smells in two open-source projects. The study first concluded that the evolution of smells is not steady along with the evolution of the systems. Concerning the change-proneness, they highlighted that smelly files exhibit more change. Same results are witnessed when considering the *God Class*, and *Brain Class* in the evolution of three projects [21]. The study consisted of analyzing the impact of smells on the change frequency (number of commits in which a file has changed) and change size (code churn) of files. An important conclusion highlighted that classes containing the examined smells are more prone to change frequency and change size. However, when the *God* and *Brain* classes are normalized with respect to size (per LOC), they are less subject to change. In the same line, Chatzigeorgiou et al. [5] studied the evolution of *Long method*, *Feature Envy*, and *State Checking* code smells in 24 releases of two Java projects (JFlex, and JFreeChart). In all examined releases, the *Long Method*, which signifies a large-sized piece of code, had exponential growth as a system evolves. Contrary to the *Feature Envy* and *State Checking*, which have shown a steady low rate of evolution.

2.2 Relationship Between Code Smells and Development Activities

Khomh et al. [12] conducted an empirical analysis on 13 different releases of Azureus and Eclipse, considering 9 code smells, to investigate three relationships. (*i*) smelly classes are more exposed to frequent updates than others (3 to 8 times in favor of smelly classes). (*ii*) the more a class has instances of smells, the more it is change-prone. (*iii*) particular but not common kinds of smells lead to more change-proneness than others. An extended study examined code smells impact in 54 releases of four projects [13] confirmed that smelly classes are more subject to change and faults. These results were confirmed by Spadini et al. [33], who found that the presence of test smells yields to more code changes, which might produce bugs in the production code. Saboury et al. [32] have carried out an empirical investigation on the impact of 12 JavaScript code smells on the fault-proneness of modified classes of five projects. They compared the fault-proneness between smelly and non-smelly files using the survival analysis test to capture a longitudinal behavior. The results show that non-smelly files have a 65% chance less than files with smells. As well, they opted for a Cox Hazard test to asses the impact of three factors on the survivability of faults (LOC, Code churn, and the number of Previous Bugs). Their results indicate that the number of Previous-Bugs, which means the number of fault-fixing changes, is fault-inducing. Aniche et al. [2] examined when code smells are introduced and how long they survive using the survival analysis test. Their study highlighted some important points (*i*) code smells are more introduced from the first release of the system, and (*ii*) code smells that are present with the first commits tend to survive more.

3 Empirical Study Design

As presented in Fig. 1, our empirical study aims at analyzing the *diffuseness* and *impact* of code smells in web applications. The diffuseness refers to the rate of code smells in code components (classes, methods), *i.e.*, how many parts of the application are affected by at least one instance of code smells. The analysis of smells distribution helps to better assess (1) the impact of code smells on the change-proneness of smelly files, and (2) how specific code smell types could result into different change sizes. It is worth noting that some smell types could lead to more change-proneness but are poorly diffused and vise versa.

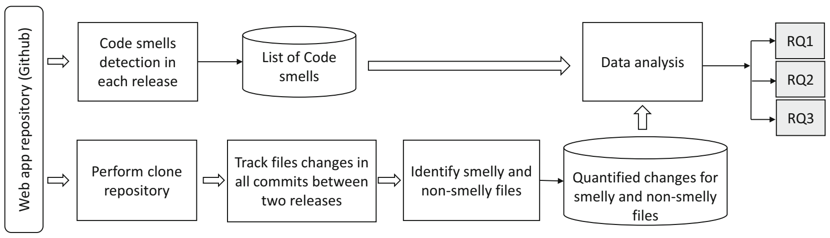
Web app repository (Github)
Code smells detection in each release
List of Code smells
Data analysis
RQ1
RQ2
RQ3
Perform clone repository
Track files changes in all commits between two releases
Identify smelly and non-smelly files
Quantified changes for smelly and non-smelly files

Fig. 1. Overview of our empirical study.

To collect our dataset for our empirical study, we considered a set of common code smell types in Web software applications. To detect instance of code smells in our benchmark, we use PHPMD [29], a widely used tool for quality assurance and code smells detection specialized for PHP software applications. We considered a list of 12 smell types as they are most known, and widely being discussed in recent studies [7,9,15,16,31]. It is worth noting that we selected class-level and method-level code smells that affect the source code's understandability or might aggravate the performance to capture a broad analysis of our studied phenomena. Although, PHPMD supports the detection of 36 types of design flaws, we basically considered common code smell types [3,18,23,25–27,31] and excluded smells related to low level violations such as calling the `var_dump()` function in the production code. Table 1 provides the list of the considered code smells along with their definitions.

3.1 Research Questions

We formulate the following research questions.

- **RQ1:** *What is the diffuseness of code smells in web apps?* We aim to know the most diffused and frequently occurring code smells to recognize which bad coding practices are more common and thus prioritize their refactoring.
- **RQ2:** *To what extent files affected by code smells exhibit a different level of change-proneness as compared to non-smelly files?* We aim to assess whether smelly files undergo more maintenance activities compared to non-smelly files by testing the following null hypotheses:
  $H2_0$: Smelly files are not more prone to change during the software evolution as compared to non-smelly files.
- **RQ3:** *What is the relationship between specific types of code smells and the level of change-proneness?* We investigate whether some smell types contribute more to the change-proneness of smelly classes. To answer our research question, we test the validity of the following null hypothesis.
  $H3_0$: Smelly files undergo the same level of change-proneness for all smell types.

To answer our research questions, we mined 223 releases of 5 popular open-source PHP web-based applications. PhpMyAdmin[4] is a framework to handle the administration of MySQL over the Web and supports MariaDB. Joomla[5] is a Content Management System (CMS) which enables you to build websites and powerful online applications. WordPress[6] is a CMS system written in PHP and paired with MySQL or MariaDB database. Piwik[7] is an analytics and full featured PHP MySQL software to download and install on webserver. Laravel[8] is a web application framework with expressive, powerful syntax and provides tools required for large, robust applications. We chose applications with different sizes ranging from 12 to 662 KLOCs. As presented in Table 2, the studied

[4] https://github.com/phpmyadmin/phpmyadmin.
[5] https://github.com/joomla/joomla-cms.
[6] https://github.com/WordPress/WordPress.
[7] https://github.com/matomo-org/matomo.
[8] https://github.com/laravel/laravel.

Table 1. List of code smells considered in our study

Code smell	Description
Excessive Number Of Children (ENOC)	A class with too many descendants usually indicates an unbalanced class hierarchy [29,31]
Excessive Depth Of Inheritance (EDOI)	A class with a deep inheritance tree can lead to an unmaintainable code as the coupling would increase [29,31]
High Coupling (HC)	A class with too many dependencies makes it harder to maintain and evolve [8,16,29,31]
Empty Catch Block (ECB)	Fixing an execution failure of an unknown exception type will require more efforts to understand the error condition [29]
Goto Statement (GTS)	Goto makes the logic of an application hard to understand [29]
High Method Complexity (HMC)	The cyclomatic complexity at the method level represents the number of decision points (*e.g.*, if, for, while). The higher the number of decision points, the higher the number of test cases needed to test all the different execution paths [9,29]
High NPath Complexity (HNPC)	The NPath complexity is the number of ways (nested if/else statements) the code can get executed, which would decrease the readability of the code and cause testing issues [29]
Excessive Method Length (EML)	When a method exceeds 100 NCLOC, it is considered a broad method that does too much. These methods are likely to end up processing data differently than what their context suggests until they become hard to understand and maintain [7,9,16,29,31]
Excessive Class Length (ECL)	Large classes are a good suspect for refactoring, as their size represents a challenge to manage efficiently, and maintain them [15,16,29,31]
Excessive Parameter List (EPL)	A long parameter set can indicate that a method is doing too many different things, which makes it harder to understand its behavior [7,17,29,31]
Too Many Public Methods (TMPM)	A large number of public methods indicate that the class does not preserve its data encapsulated. Consequently, changing the internal behavior of the class requires additional efforts not to risk damaging some dependencies. In practice, we cannot restrict the number of public methods. Only what could be exposed should be public. If external classes are extensively accessing these methods, they should be moved to reduce the coupling [29]
Too Many Methods (TMM)	The Too Many Methods is the symptom of a class that contains a large number of methods that typically do not belong to its responsibilities and consequently decreases the cohesion level [29]

projects belong to different application domains and actively engineered during 9 to 15 years. Table 2 reports the number of considered releases, the number of stars on Github, and we count the applications size in terms of the number of classes, methods, and KLOCs for each project using the PHPLOC tool[9].

Table 2. The studied systems statistics.

Name	Releases	Period	Stars	# classes	KLOCs	# smells
phpMyAdmin	55	2014–2020	4.7k	30–645	228–328	60,695
Joomla	34	2011–2019	3.4k	1,102–2,631	271–662	75,616
WordPress	74	2005–2019	13.4k	24–496	37–391	106,962
Piwik	38	2010–2020	12.6k	1,017–2,095	242–374	39,896
Laravel	22	2012–2020	57.3k	95–248	12–40	1,647

3.2 Analysis Method

To answer **RQ1**, we first compute the absolute number of code smells present in each application (aggregation of releases). Then, we assess the number of affected classes for each smell type. To better position the number of smells with respect to the size of the application, we assess the diffuseness of smells per KLOC.

To answer **RQ2**, we use the git versioning system to mine the change history of the five applications. We identify all modified PHP files in each commit between the releases r_{j-1} and r_j. Then, we extract the number of modification each modified file has undergone using the following git command:

```
$ git show --stat --no-commit-id --oneline -r SHA1..SHA2"*.php"
```

Then, we identify the nature of the returned modified files whether it is a *smelly* or a *non-smelly* class. Thereafter, we compute the change-proneness of a modified class c as the sum of the changes performed in all commits between the releases r_{j-1} and r_j.

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

where n is the number of commits between releases r_{j-1} and r_j, the function $\textit{churn}(c, com_i)$ returns the code churn in terms of number of added, removed and modified lines of code in the class c in commit com_i using the GitHub API[10].

After the extraction of all data, we compare the change-proneness of smelly and non-smelly classes using the beanplot representation [11]. A beanplot extends the boxplot's visualization by representing the density of data distribution along with the individual observations. To assess $H2_0$, we verify whether

[9] https://github.com/sebastianbergmann/phploc.
[10] https://developer.github.com/v3/.

there is a significant difference between the two tested populations (smelly, non-smelly). Therefore, after the data normality check (*p-value* $= 0.8$ and *p-value* $= 0.6$), we apply the parametric independent t-test [14] to check the magnitude of difference between our two groups. The t-test serves to evaluate the alternative hypothesis stating how likely one sample exhibits dominance compared with the other sample. We consolidate the test by measuring the parametric Cohen-d effect size. As stated by [6], the effect size tells how important the difference between the two samples is. An effect size is considered small if $0.2 \leq d < 0.5$, medium if $0.5 \leq d < 0.8$ and large if $d \geq 0.8$. It is worth noting that we consider a class as smelly only if it has at least one instance of code smell. We narrowed the gap between smelly and non-smelly classes to deeply analyze the phenomenon of change-proneness considering most harmful smells. Moreover, if a class changes from smelly to non-smelly in some releases and vise versa, it contributes to both sets of smelly and non-smelly classes.

To answer **RQ3**, we assess the impact of smells types on the change-proneness. We compute the number of occurrences of each smell type in the smelly classes of each release. We quantify the impact of each smell type for each project as the correlation score between the sum of the frequency count of each smell type ST_i and the class state $\{0 \text{ or } 1\}$ representing whether a class has changed or not between two releases $r_{j-1} \rightarrow r_j$. To statistically analyze the effect of each smell type on a class change-proneness, we opted for a *logistic regression* test [10] similar to khomh et al. [12] to reject the null hypothesis $H3_0$ stating that classes undergo the same change size for all types of smells. The logistic regression should decide whether the class would change for each smell type. To asses the change of a class based on a set of smells, a class would represent the dependent variable C_i that would change if one of the smell types ST_j (independent variable) changes as well. In a logistic model, the dependent variable could take only two values (changed = 1, not changed = 0). Thus, our multivariate model equation applied to a class C_i in a release r_t is defined as follows:

$$P(C_i) = \frac{e(CP + \sum_1^{12} b_j * ST_j)}{1 + e(CP + \sum_1^{12} b_j * ST_j)} \quad \in [0, 1] \tag{2}$$

where P is the likelihood that a class changes; CP is the change proneness of a class {0,1}; and b_j is the number of occurrences of a smell type ST_j.

We apply our logistic regression model for each smell type detected in the 223 releases in our benchmark. Then, we count the number of times the *p-value* of a smell is significant (the probability is closer to 1).

4 Study Results and Analysis

4.1 RQ1: Code Smells Diffuseness

Figure 2 reports (*i*) the absolute number of code smells distribution in the analyzed projects, (*ii*) the number of affected classes by each code smell type, and (*iii*) the density of code smells per KLOCs using the beanplot visualization. For the sake of clarity, we aggregate the occurrences of each code smell in our studied projects into one single dataset. From the beanplots and Table 3, we observe the existence of three main categories of code smell distributions (1) highly diffused and highly frequent, (2) highly diffused and slightly frequent, and (3) slightly diffused and slightly frequent.

Highly Diffused and Highly Frequent Smells: As shown in Fig. 2 and Table 3, the *High Method Complexity* smell is the most diffused (99%) and frequent (42%) code smell. It typically manifests in the form of a high cyclomatic complexity level within the methods. We found that this smell has a high number of occurrences with 1,250 instances in the two last studied releases of `Joomla` (3.9.13 and 3.9.14). For instance, the class `Joomla.CMS.Form.Form` in release 3.9.13 has a cyclomatic complexity of 64 in its method `filterField()` responsible for applying an input filter to a value based on field data. These methods are in general very long (on average, 261 LOC found in *Joomla* studied releases). Moreover, we found that the *High NPath Complexity* occurs also in 99% of the releases, representing 27% of the total number of detected smells. It has a total of 820 occurrences in `Joomla` 3.9.13. Alike, *Excessive Method Length* impacts 95% of the releases, representing 16% of the smells with a peak reaching 556 in `WordPress` 5.3.2. Indeed, we found 12 long methods using AJAX with an average of 143 LOC. Moreover, from a qualitative sense, the diffuseness of smell instances per KLOC is reported in Fig. 2c which confirms that the *High Method Complexity*, the *High NPath Complexity*, and the *Excessive Method Length* are the most diffused smells with an average of 24, 17, and 13 instances respectively.

Highly Diffused and Slightly Frequent Smells: This category of smells occur in the majority of the studied releases but with a limited number of instances. As shown in Fig. 2 and Table 3, we observe that the *High Coupling* smell exists in 98% of the releases, but representing only 2% of the accumulated number of smells. For instance, we found that the *High Coupling* smell reaches the bar of 99 instances in both releases of `Piwik` 3.13.0 and 3.13.1. On average, each of the infected releases has 25 instances of this smell. As compared to other studies in Android apps, the *High Coupling* is found to have weak diffuseness and frequency as pointed by Mannan et al. [16]. To better understand this disparity in terms of diffuseness, we conducted a closer analysis on the 3.13.0 release of `Piwik`. Most of the instances are located in the `Archive`, and `ArchiveProcessor` packages. In particular, the class `CronArchive` in `Archive` package has a coupling between objects (CBO) score of 33 surpassing the established threshold of 13 which is considered as normal [29]. Hence, this disparity in the diffuseness rate of the *High Coupling* between Android and web apps could be related to the small size of Android apps along with their different structure and workflow which typically come with a low coupling between code components.

Moreover, we found that the *Excessive Class Length*, the *Too Many Public Methods*, the *Excessive Number Of Children*, and the *Too Many Methods* are not frequent as they have a maximum number of occurrences per class that do not exceed 5. For example, the *Too Many Public Methods* smell represents 5% of the total number of smells, and is distributed across 86% of the releases as shown in Fig. 2 and Table 3. Most diffused instances are in `Joomla` with a high number of occurrences of 234 in the last five releases (from v3.9.7 to v3.9.14). Likewise, among the highly diffused, but slightly frequent code smells, we found the *Too Many Public Methods* smell which have more instances per KLOC (6). In addition, the *Empty Catch Block* and *Excessive Parameter List* code smells are impacting 65% and 89% of releases with the highest number of occurrences of 69 in `Piwik` (2.17.1 and 2.18.0) and 40 in `Piwik` 2.12.1 respectively. The *Excessive Parameter List* has the highest occurrence number (19) of all slightly frequent smells in the method `image()` in the package `com.tecnick.tcpdf`. The *Excessive Parameter List* is a one single-metric violation that straightforwardly detects the smell. Besides, it is also worth noting that, `Wordpress` and `phpMyAdmin` applications have no instance of the *Empty Catch Block*, which is limited to one instance per KLOC.

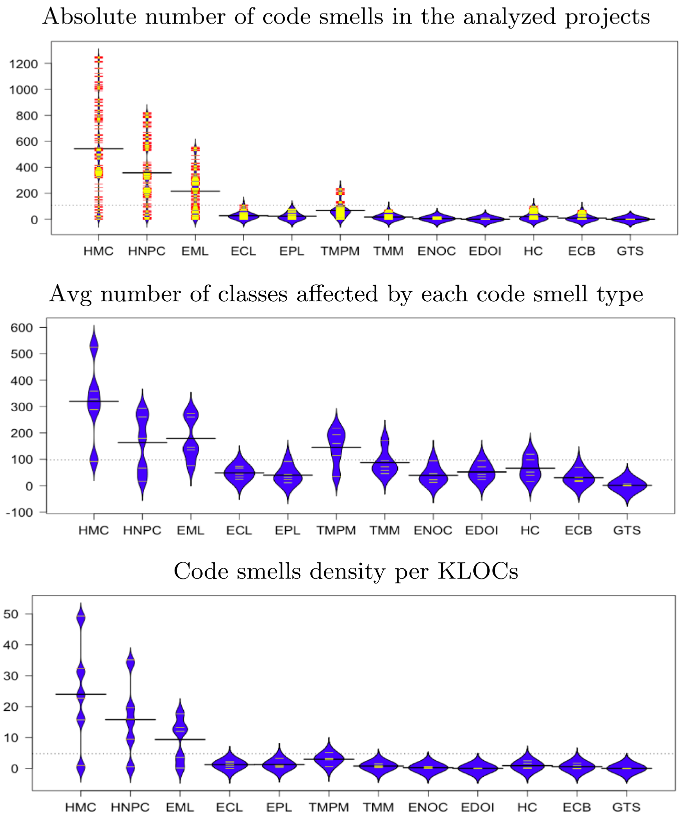

Fig. 2. The absolute number, % affected classes, and density per KLOC of smells.

Slightly Diffused and Slightly Frequent Smells: As shown in Fig. 2 and Table 3, the *Excessive Depth Of Inheritance* and the *Go to Statement* code smells are slightly diffused and not frequent. Overall, the *Excessive Depth Of Inheritance* smell exists only in 20 classes, impacting only 4% of the studied releases, and it represents nearly 1% of the total number of detected code smells. For instance, we found that the highest number of occurrences of the *Excessive Depth Of Inheritance* smell is 10 in `Piwik` 1.8.0. Similarly, the *Goto Statement* smell represents nearly 1% of the total number of code smells and affects 2% of the studied releases. This particular smell occurs only in `phpMyAdmin` (10% of the releases of phpMyAdmin), with a negligible percentage of ∼1% of the total number of code smells detected in `phpMyAdmin`. Since its spread is limited to a few classes, the correction of *Goto Statement* becomes easier for developers.

Table 3 reports the diffuseness of code smells according to the accumulated number of releases. The "% of affected releases" column represents the percentage of affected releases by a particular smell. For example, the *Too Many Methods* smell impacts 77% of the releases. The "max instances" column reports the highest number of occurrences of a given smell in a class. For instance, the *Too Many Methods* smell has the highest number of occurrences in the `libraries.simplepi.e.simplepi.e.php` file in `Joomla 2.5.3` which has 5 classes having respectively 102, 40, 40, 35, and 26 methods exceeding the basic threshold of 25 [29].

To sum up, most of the smells are quite diffused in the studied subjects. Particularly, smells related to long and complex code fragments (*i.e.*, *High Method Complexity*, *High NPath Complexity*, and *Excessive Method Length*) have the highest number of instances per KLOC, and impact the highest number of classes. Our findings align with those of Palomba et al. [28] on Java traditional code smells, where *Long method* and *Complex Class* code smells are the most diffused. Moreover, 3/4 of the analyzed code smells are not frequent (*i.e.*, limited number of occurrences per release), but affecting 68% of the studied projects. Besides, `Joomla` is the most affected project having the maximum number of occurrences of four smells *High Method Complexity*, *High NPath Complexity*, *Excessive Method Length*, and *Too Many Public method*. By studying code smells diffuseness, we aim to assess the interplay between the magnitude of the diffuseness for each smell type and code maintainability.

4.2 RQ2: The Impact of Code Smells on the Change-Proneness

The beanplots in Fig. 3 illustrate the change size range of both smelly and non-smelly classes. As previously found in studies targeting Java object-oriented systems [13,28], code smells lead to more code changes in a class and thus require higher maintenance efforts. As reported in Fig. 3, we witnessed similar findings, as we found that smelly classes clearly exhibit a higher level of change-proneness as compared to non-smelly classes. The median of change-proneness (CP) of smelly classes is 12.8 which is almost three times higher than the non-smelly classes (4.1). For example, the median change in `Laravel` is 2 against a median of 4 in the smelly classes. The intense density shape in the non-smelly class set demonstrates how the majority of non-smelly classes experience similar change

Table 3. Diffuseness of code smells in the analyzed projects

Code smell	% affected releases	% of smells	Max instances
High Method Complexity	99%	42%	96
High NPath Complexity	99%	27%	76
Excessive Method Length	95%	16%	42
High Coupling	98%	2%	2
Excessive Class Length	98%	2%	3
Excessive Parameter List	89%	1%	19
Too Many Public Methods	86%	5%	5
Excessive Number of Children	78%	∼1%	1
Too Many Methods	77%	1%	5
Goto Statement	2%	∼1%	1
Empty Catch Block	65%	1%	4
Excessive Depth of Inheritance	4%	∼1%	1

rate. For instance, 41 non-smelly classes in `Laravel` that are responsible for languages setting underwent almost the same modifications size. Unlike the smelly-classes, each has its maintenance requirements, which seems to be related to the co-existence of different types of code smells such as *High Method Complexity* and *Excessive Method Length*. Referring to statistical evidence, the t-test shows a statistically significant difference with a *p-value* $= 0.03$, while Cohen d shows a large effect size of 1.8, allowing us to reject the null hypothesis $H2_0$. To sum up, the majority of releases are affected by smells. However, the large portion of the modified classes in each project are smells-free. Still, smelly classes undergo more changes, and thus, exhibiting a higher level of change-proneness than non smelly classes.

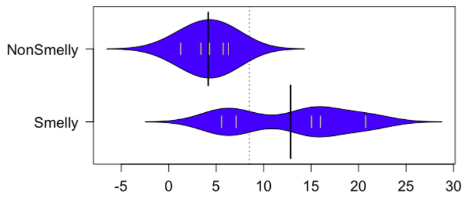

Fig. 3. Change-proneness of smelly and non-smelly classes.

4.3 RQ3: The Impact of Code Smells Types on the Change-Proneness

Table 4 reports the results of the logistic regression model for RQ3. The reported values refer to the percentage of releases for which the correspondent smell type is statistically significant in the logistic model with a $p - value < 0.05$. We observe that the existence of smells does impact the majority of projects in terms of increasing the proneness of their infected files, and this impact varies from one project to another. More precisely, we highlight the *High Method Complexity (HMC)*, *Excessive Method Length (EML)* and *Too Many Methods (TMM)* smell types, as they exhibit the highest impact on the change-proneness on 32%, 28% and 25% of the releases, respectively. In particular, HMC has shown an impact on two out of the five projects, namely *Piwik* and *WordPress*, and EML has an impact on *phpMyAdmin* and *Laravel*, while TMM impacted 3 projects (*Joomla*, *Laravel* and *phpMyAdmin*). On the other side, we observe that other smells such as the *Excessive Parameter List*, the *Excessive Number of Children*, and *Goto Statement* do not have statistically significant impact on the change-proneness on any release or project. This can be due to the fact that the latter smells are found to be slightly diffused and slightly frequent, as observed in RQ1.

Table 4. The results of the logistic regression model reporting the number of releases and projects for which each smell type is statistically significant.

Code smell	% sig. releases	Projects
High method complexity	32%	Piwik, WordPress
Excessive method length	28%	phpMyAdmin, Laravel
Too many methods	25%	Joomla, Laravel, phpMyAdmin
High coupling	21%	Piwik, WordPress, phpMyAdmin
Excessive class length	13%	phpMyadmin, Laravel
High NPath complexity	12%	Piwik
Too many public methods	4%	WordPress
Empty catch block	1.2%	–
Excessive depth of inheritance	0.9%	–
Excessive parameter list	0%	–
Excessive number of children	0%	–
Goto statement	0%	–

The main insights that we can draw from these findings could be summarized as follows (1) the slightly diffused and slightly frequent code smells have no statistically significant impact on change-proneness of files across the five projects. Hence, not all smells should be given equal removal priority. For instance, the *GoTo Statement* and *Excessive Depth Of Inheritance* are not seen as problematic, as they do not cause an increase in the number of code changes; (2) Classes in *Joomla* tend to experience an increase on change percentage whenever there is a variation in the number of *Too Many Methods* instances. (3) diffuseness and frequency of smells do not necessarily correlate with their ability to impact change-proneness of files. For example, The *High method Complexity* smell has shown the highest diffuseness in 99% of the releases (cf. Table 3), yet, its statistically significant impact on files change-proneness is limited to only two projects.

We can conclude that the impact of smells varies by type and by project. Existence of smells is alarming since they increase the chance of experiencing higher change rate, especially with *Too Many Methods* and *High Coupling* that scored the highest impact in our experiment. Since for each project, at least one single smell is showing an effect on the change-proneness. Thus, we reject the null hypothesis $H3_0$. knowing the types of code smells leading to more change-proneness will aid in preparing specific refactoring plans and focus on fixing the most harmful design and implementation practices.

5 Threats to Validity

The *Construct validity* concerns errors in measurements. In our context, we relied on the git versioning systems of each project to count the number of changes. For each release, we were interested in quantifying the changes in modified files. Moreover, while we considered 12 common code smells based on recent studies [18,31], there could be other code smell types to be considered. Moreover, similar to Khomh et al. [12], we used the logistic regression test to determine which smells are significant with the change-proneness.

The *Internal validity* concerns the factors that can limit the applicability of our observations. We assessed the cause-effect relation between the presence of code smells and the change-proneness of a file as the probability of smell to exert an impact on the state of a class. Still, we cannot assume that the changes made on a file are the result of code smells refactoring activities. Other improvement activities (exp. adding new functionalities) could yield to these changes. However, we expect that classes with high change-proneness represent the business logic of the system that does too much and gets frequently modified. Thus, these classes are more prone to having code smells and possibly exhibit more refactoring operations.

The *External validity* concerns the generalizability of our findings. We have analyzed a total of 5 PHP Web projects with different communities, sizes, and application domains and with a minimum of 9 years of history. We are aware that we cannot generalize our finding to other projects. In the future, we plan to reduce this threat further by analyzing more projects from more industrial and open-source software projects and other web programming languages.

6 Conclusion

This paper reported a large study conducted on 223 releases of five popular web-based applications. The empirical study aimed at understanding the diffuseness of code smells in web open source apps and their relation with source code change-proneness. The statistical analysis of the obtained results show that most diffused and frequent code smells are related to the size and complexity of code fragments. Moreover, our findings indicate that classes with such smells are more prone to change than other classes which may require more maintenance efforts. To provide better insights, we individually investigated the relationship between each smell type and the change-proneness using a logistic regression model. Results showed that specific smells do have an impact on the change-proneness of a class. However, the type of these change-inducing smells tend to be context related. Our findings indicate that *code smells should be carefully monitored by web programmers*, since they are diffused in web applications and related to maintainability aspects such as change-proneness. As future work, we first plan to replicate our study on other open source and industrial web applications. We plan also to analyze the impact of the co-occurences of code smells on the change-proneness. Moreover, we plan to investigate the impact of smelly-files on the fault-proneness. More interestingly, we will develop automated code smells refactoring recommendation and prioritization techniques in the context of web apps to better monitor code smells.

References

1. Replication package. https://github.com/Narjes-b/SmellsAnalysis-WebApps
2. Aniche, M., Bavota, G., Treude, C., Gerosa, M.A., van Deursen, A.: Code smells for model-view-controller architectures. Empirical Soft. Eng. **23**(4), 2121–2157 (2018)
3. Boukharata, S., Ouni, A., Kessentini, M., Bouktif, S., Wang, H.: Improving web service interfaces modularity using multi-objective optimization. Autom. Sofw. Eng. **26**(2), 275–312 (2019)
4. Brown, W.H., Malveau, R.C., McCormick, H.W., Mowbray, T.J.: AntiPatterns: Refactoring Software, Architectures, and Projects In Crisis. Wiley, New York (1998)
5. Chatzigeorgiou, A., Manakos, A.: Investigating the evolution of bad smells in object-oriented code. In: Seventh International Conference on the Quality of Information and Communications Technology, pp. 106–115. IEEE (2010)
6. Cohen, J.: Statistical Power Analysis for The Behavioral Sciences. Erihaum, Hillsdale (1988)
7. Delchev, M., Harun, M.F.: Investigation of code smells in different software domains. Full-scale Softw. Eng. **31**, 31–36 (2015)
8. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Boston (2018)
9. Hecht, G., Benomar, O., Rouvoy, R., Moha, N., Duchien, L.: Tracking the software quality of android applications along their evolution (t). In: International Conference on Automated Software Engineering (ASE), pp. 236–247 (2015)
10. Hosmer, D.W., Lemeshow, S., Cook, E.: Applied Logistic Regression, 2nd edn. Wiley, New York (2000)
11. Kampstra, P., et al.: Beanplot: a boxplot alternative for visual comparison of distributions. J. Stat. Softw. **28**(1), 1–9 (2008)
12. Khomh, F., Di Penta, M., Gueheneuc, Y.G.: An exploratory study of the impact of code smells on software change-proneness. In: WCRE, pp. 75–84 (2009)

13. Khomh, F., Di Penta, M., Guéhéneuc, Y.G., Antoniol, G.: An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empirical Softw. Eng. **17**(3), 243–275 (2012). https://doi.org/10.1007/s10664-011-9171-y
14. Kim, T.K.: T test as a parametric statistic. Korean J. Anesthesiol. **68**(6), 540 (2015)
15. Liu, X., Zhang, C.: The detection of code smell on software development: a mapping study. In: 5th International Conference on Machinery, Materials and Computing Technology (ICMMCT 2017). Atlantis Press (2017)
16. Mannan, U.A., Ahmed, I., Almurshed, R.A.M., Dig, D., Jensen, C.: Understanding code smells in android applications. In: IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 225–236 (2016)
17. Martin, R.C.: Clean Code: A Handbook of Agile Software Craftsmanship. Pearson Education, London (2009)
18. Mon, C.T., Hlaing, S., Tin, M., Khin, M., Lwin, T.M., Myo, K.M.: Code readability metric for PHP. In: IEEE 8th Global Conference on Consumer Electronics (GCCE), pp. 929–930 (2019)
19. Nguyen, H.V., Nguyen, H.A., Nguyen, T.T., Nguyen, A.T., Nguyen, T.N.: Detection of embedded code smells in dynamic web applications. In: IEEE/ACM International Conference on Automated Software Engineering, pp. 282–285 (2012)
20. Olbrich, S., Cruzes, D.S., Basili, V., Zazworka, N.: 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 (2009)
21. Olbrich, S.M., Cruzes, D.S., Sjøberg, D.I.: Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems. In: International Conference on Software Maintenance, pp. 1–10 (2010)
22. Ouni, A., Gaikovina Kula, R., Kessentini, M., Inoue, K.: Web service antipatterns detection using genetic programming. In: Annual Conference on Genetic and Evolutionary Computation (GECCO), pp. 1351–1358 (2015)
23. Ouni, A., Kessentini, M., Bechikh, S., Sahraoui, H.: Prioritizing code-smells correction tasks using chemical reaction optimization. Softw. Qual. J. **23**(2), 323–361 (2015)
24. Ouni, A., Kessentini, M., Inoue, K., Cinnéide, M.O.: Search-based web service antipatterns detection. IEEE Trans. Serv. Comput. **10**(4), 603–617 (2017)
25. Ouni, A., Kessentini, M., Ó cinnéide, M., Sahraoui, H., Deb, K., Inoue, K.: MORE: a multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. Softw. Evol. Process **29**(5), e1843 (2017)
26. Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., Deb, K.: Multi-criteria code refactoring using search-based software engineering: an industrial case study. ACM Trans. Softw. Eng. Methodol. **25**(3), 1–53 (2016)
27. Ouni, A., Kessentini, M., Sahraoui, H., Inoue, K., Hamdi, M.S.: Improving multi-objective code-smells correction using development history. J. Syst. Softw. **105**, 18–39 (2015)
28. Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., De Lucia, A.: On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empirical Softw. Eng. **23**(3), 1188–1221 (2018). https://doi.org/10.1007/s10664-017-9535-z
29. PHPMD (2020). https://phpmd.org
30. Pressman, R.S.: Software engineering: a practitioner's approach. Palgrave Macmillan, London (2005)

31. Rio, A., Brito e Abreu, F.: Code smells survival analysis in web apps. In: Piattini, M., Rupino da Cunha, P., García Rodríguez de Guzmán, I., Pérez-Castillo, R. (eds.) QUATIC 2019. CCIS, vol. 1010, pp. 263–271. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29238-6_19
32. Saboury, A., Musavi, P., Khomh, F., Antoniol, G.: An empirical study of code smells in Javascript projects. In: International Conference on Software Analysis, Evolution and Reengineering, pp. 294–305 (2017)
33. Spadini, D., Palomba, F., Zaidman, A., Bruntink, M., Bacchelli, A.: On the relation of test smells to software code quality. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 1–12. IEEE (2018)
34. Tufano, M., et al.: An empirical investigation into the nature of test smells. In: International Conference on Automated Software Engineering, pp. 4–15 (2016)
35. Tufano, M., et al.: When and why your code starts to smell bad. In: IEEE International Conference on Software Engineering, vol. 1, pp. 403–414 (2015)