Detecting Sudden Variations in Web Apps Code Smells' Density: A Longitudinal Study

Américo Rio[1,2(✉)] and Fernando Brito e Abreu[1]

[1] Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR, Lisboa, Portugal
{jaasr,fba}@iscte-iul.pt
[2] NOVAIMS, Universidade Nova de Lisboa, Lisboa, Portugal
americo.rio@novaims.unl.pt
https://www.iscte-iul.pt, https://www.novaims.unl.pt/

Abstract. Code smells are considered potentially harmful to software maintenance. Their introduction is dependent on the production of new code or the addition of smelly code produced by another team. Code smells survive until being refactored or the code where they stand is removed. Under normal conditions, we expect code smells density to be relatively stable throughout time. Anomalous (sudden) increases in this density are expected to hurt maintenance costs and the other way round. In the case of sudden increases, especially in pre-release tests in an automation server pipeline, detecting those outlier situations can trigger refactoring actions before releasing the new version.

This paper presents a longitudinal study on the sudden variations in the introduction and removal of 18 server code smells on 8 PHP web apps, across several years. The study regards web applications but can be generalized to other domains, using other CS and tools. We propose a standardized detection criterion for this kind of code smell anomalies. Besides providing a retrospective view of the code smell evolution phenomenon, our detection approach, which is particularly amenable to graphical monitoring, can make software project managers aware of the need for enforcing refactoring actions.

Keywords: PHP · Code smells · Web apps · Sudden variations · Anomaly detection · Outliers

1 Introduction

1.1 Motivation

A major manifestation of maintenance issues is the existence of code smells [10], since they are seen as potential catalysts of software evolution costs, due to increased defect incidence, poorer code comprehension, and longer times to release. A code smell (CS) may be something like a long method, or many parameters in a method. Java desktop applications have been particularly analyzed

A. C. R. Paiva et al. (Eds.): QUATIC 2021, CCIS 1439, pp. 82–96, 2021.
https://doi.org/10.1007/978-3-030-85347-1_7

regarding this aspect [21,25]. The Software Engineering community has proposed several techniques and tools, both for CS detection and refactoring, but several problems remain such as detection subjectivity [5] and, low coverage of existing CS catalogs and programming languages [22,28].

While looking for the CS phenomenon on a quantitative basis, we should not analyze the raw number of existing (or removed) CS, because that number will largely depend on system size. Some measure of CS density (the number of CS divided by a size measure) should be used instead, like for instance in [8,17]. For a reasonably large project, maintained by a large team, developing code as usual, we expect to observe an inertia effect, i.e. that CS density is relatively stable throughout time. However, there are moments in the history of a project where that density may have sudden variations.

CS sudden variations are relevant to understand the story of a project, can be used as an explanatory factor (e.g. for consequent variations in reported issues and maintenance effort) and, justify the relevance of refactoring actions. Software managers should: (i) be aware if CS are under control (i.e. if CS infection is not going wild) and, if not, (ii) prioritize refactoring of detected CS. A solution to prevent the first problem is proposed in this paper for PHP web apps. A solution to the second can be found in [9] in the context of Java systems.

Web apps are different from desktop and mobile apps. The latter run on the OS, while the former run both on a browser and a server, and thus have server-side programming and client/browser-side programming, that run in two separate environments. Thus, they encompass a heterogeneity of target platforms, programming, and content formatting languages. Due to this difference and diversity, it is necessary to perform similar and different studies (from the applications that run directly in the OS), regarding their specificity. In the study, we focus on web applications using the PHP programming language, currently reported to hold 79% of the market share in that sector [27]. We considered as many years as possible for each web app, summing up a total of 441 versions.

1.2 Research Questions

During data collection for another study [23], we noticed that sudden variations in CS density, in both directions (steep increase or steep decrease), occur in some versions (also called releases) of the target web apps. These anomalous situations deserve our attention, either for recovering the story of a project or, if used just-in-time (e.g., integrated into a pre-release tests battery), to provide awareness to decision-makers that something unusual is taking place for good or bad. In this paper we aim at providing an answer to the following research questions in the context of web apps using PHP as the server language:

RQ1 – How to detect sudden variations in the evolution of code smells?
RQ2 – When are the sudden variations in changes of code smells in a new version considered too high? or When are there too many code smells?

To answer these research questions, we perform a longitudinal study with 8 web applications and 18 server-side CS. This paper is structured as follows:

Sect. 2 introduces the study design; Sect. 3 describes the results of data analysis, while Sect. 4 discusses the findings and identifies validity threats; Sect. 5 overviews the related work on longitudinal studies on CS and in web apps; finally, Sect. 6 outlines the major conclusions and outlines future work.

2 Study Design

2.1 Applications Sample

The criteria for selecting the sample of PHP web apps were the following:

Inclusion criteria: code availability (open source); complete or self-contained applications, taken from the *GitHub* top listings; programmed with an object-oriented programming (OOP) style; covering a long period (at least 5 years)
Exclusion criteria: libraries; frameworks or applications used to build other applications; web apps built using a framework.

We excluded frameworks and libraries because we want to study typical web apps. We excluded web apps built with frameworks because we want to analyze app code and not the framework code itself, thus aiming for comparability among apps. Probably, apps made with frameworks would deserve a separate study. Another reason is that we have apps with a long history, and many started when modern frameworks were not available. Some apps that we had as candidates failed the requisite of being OOP. PHP allows for a procedural development paradigm, but the detected CS are OOP in nature. Table 1 shows the sample of apps used, including some metrics.

Table 1. Characterization of the target web apps (* on last version)

Name	Purpose	#Versions(period)	Last version	LOC*	#Classes*
PhpMyAdmin	Database administration tool	181 (09/2008-09/2019)	4.9.1	301748	1174
DokuWiki	Wiki solution	40 (07/2005-01/2019)	04-22b	271514	402
OpenCart	Shopping cart solution	28 (04/2013-04/2019)	3.0.3.2	206253	955
PhpBB	Forum/bulletin board solution	50 (04/2012-01/2018)	3.2.2	341159	1330
PhpPgAdmin	Database administration tool	29 (02/2002-09/2019)	7.12.0	71210	54
MediaWiki	Wiki solution	145 (12/2003-10/2019)	1.33.1	754941	2479
PrestaShop	Shopping cart solution	74 (06/2011–08/2019)	1.7.6.1	516737	2597
Vanilla	Forum/bulletin board solution	75 (06/2010–10/2019	3.3	193435	533

For each app we collected as many versions as possible. Sometimes we could not get the whole lifecycle either because not all versions were available online or did not match the OOP criterion in the earlier versions. The LOC (Lines Of Code) and "number of Classes" are size metrics from the last version and were measured by the *PHPLOC* tool.

2.2 CS Sample

We used *PHPMD*, an open-source tool that detects CS in PHP. *PHPMD* was the base for many other PHP CS detection tools existing today, and we can automatize it via the command line. Although the tool supports more CS, we chose the maximum number of highly cited ones in the literature for other languages, leaving us with the 18 CS which are briefly characterized in Table 2. For comparability among apps, the thresholds are the same among apps, and the default ones used in *PHPMD*, witch in turn came from *PMD*, and are generally accepted from the references in the literature [4,16]. For individual app evaluation these could be optimized [13].

Table 2. Characterization of the target code smells

Code smell name	Code smell description	Threshold
CyclomaticComplexity	Determined by the number of decision points in a method plus one for the method entry	10
NPathComplexity	Number of acyclic execution paths through that method	200
ExcessiveMethodLength	(Long method) the method is doing too much	100
ExcessiveClassLength	(Long Class) class does too much	1000
ExcessiveParameterList	Method with too long parameter list	10
ExcessivePublicCount	A large number of public methods and attributes declared in a class	45
TooManyFields	Class with too many fields	15
TooManyMethods	Class with too many methods	25
TooManyPublicMethods	Class with too many public methods	10
ExcessiveClassComplexity	Excessive Sum of complexities of all methods in a class	50
NumberOfChildren	Class with an excessive number of children	15
DepthOfInheritance	Class with many parents	6
CouplingBetweenObjects	Class with too many dependencies	13
DevelopmentCodeFragment	Development Code ex: var_dump(), print_r() etc.	1
UnusedPrivateField	A private field is declared and/or assigned a value, but not used	1
UnusedLocalVariable	A local variable is declared and/or assigned, but not used	1
UnusedPrivateMethod	A private method is declared but is unused	1
UnusedFormalParameter	Unused parameters in methods/constructors that are not used	1

2.3 Data Collection and Preparation

In the data collection and preparation phase, we downloaded the source code of all versions of the selected web applications, in ZIP format, from *GitHub*, *SourceForge*, and application site (when not available), except the alpha, beta, release candidates, and corrections for old versions. We created a database table with the application versions, later exported to a CVS file, containing the timestamps for each downloaded version. Using *PHPMD*, we extract the CS, file and line locations, dates, and other CS indicators from every version, and store them in a database. For the applications, we excluded some directories that were not part of the applications (vendor, libraries, images, etc.). The data at this point was stored by version/smell. We then exported the results to CVS format, in preparation for the data analysis phase. We used the *PHPLOC* tool to extract several code metrics from the source code of each version of each app. The collected dataset is available *here*, in csv format, for replication purposes.

3 Results and Data Analysis

3.1 Evolution of the Number of CS per Version

Figure 1 presents the evolution of CS by version. Software delivery occurs at unequally spaced moments in time, that we usually call "versions", or "releases". The visualization in Fig. 1 allows identifying the versions where a major refactoring occurred. For example, in *PhpMyAdmin*, from a graph of survival of code smells not shown here (space constrictions), versions 4.6, 4.7 and 4.8 seem to have a lot of refactoring. We drilled down this behavior in the code of version 4.8 because a lot of smells were introduced in this version, and a lot were removed. By code inspection we found that during that refactoring some files changed their names, so the smell appears in another file. In other words, a rename in a file or class causes the fake conclusion that some existing smells were removed while new ones were created. To block this fake effect, we should observe the evolution of the total number of occurrences for each CS, by version. We observed these phenomena in the record of each CS, in our data. If we observe Fig. 2, we do not understand if the smells are all new or came from renaming operations but observing the total number of CS in Fig. 3 we have a different perspective. We also pinpoint that for 2 of the applications, *Vanilla* and *MediaWiki*, the number of CS increases steadily during the life of the application. The "ExcessiveMethodLength" (aka *Long Method*) CS is one of the more recurrent. However, the "complexity" smells play an important part in the total computation, and also in some applications, the "unused" group of smells.

3.2 Anomalies in CS Evolution

During data analysis, we found versions where refactoring on file names and location in folders occurred, but CS prevailed in a different file/folder. So, is it

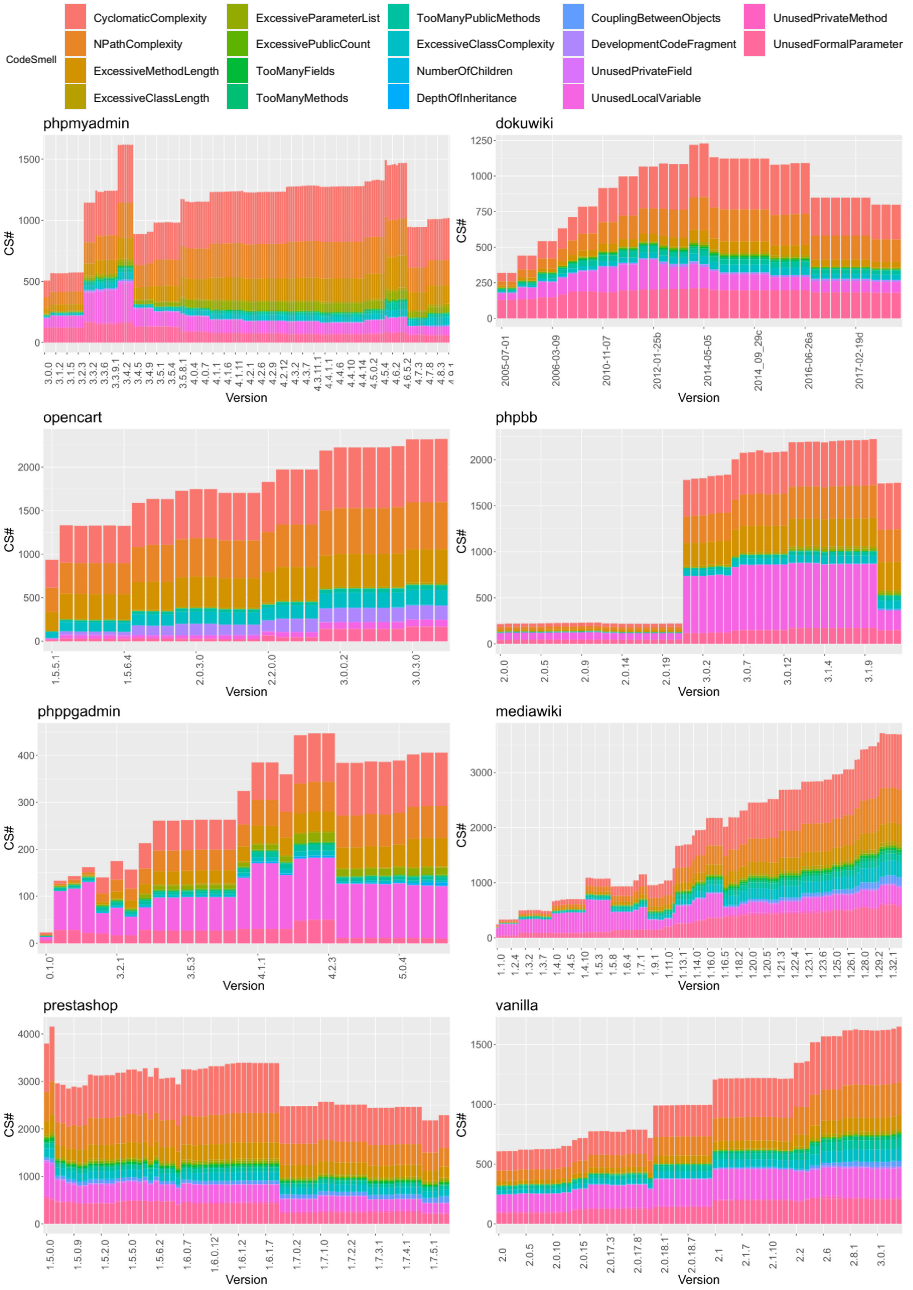

Fig. 1. Evolution of the total number of each code smell by app and version

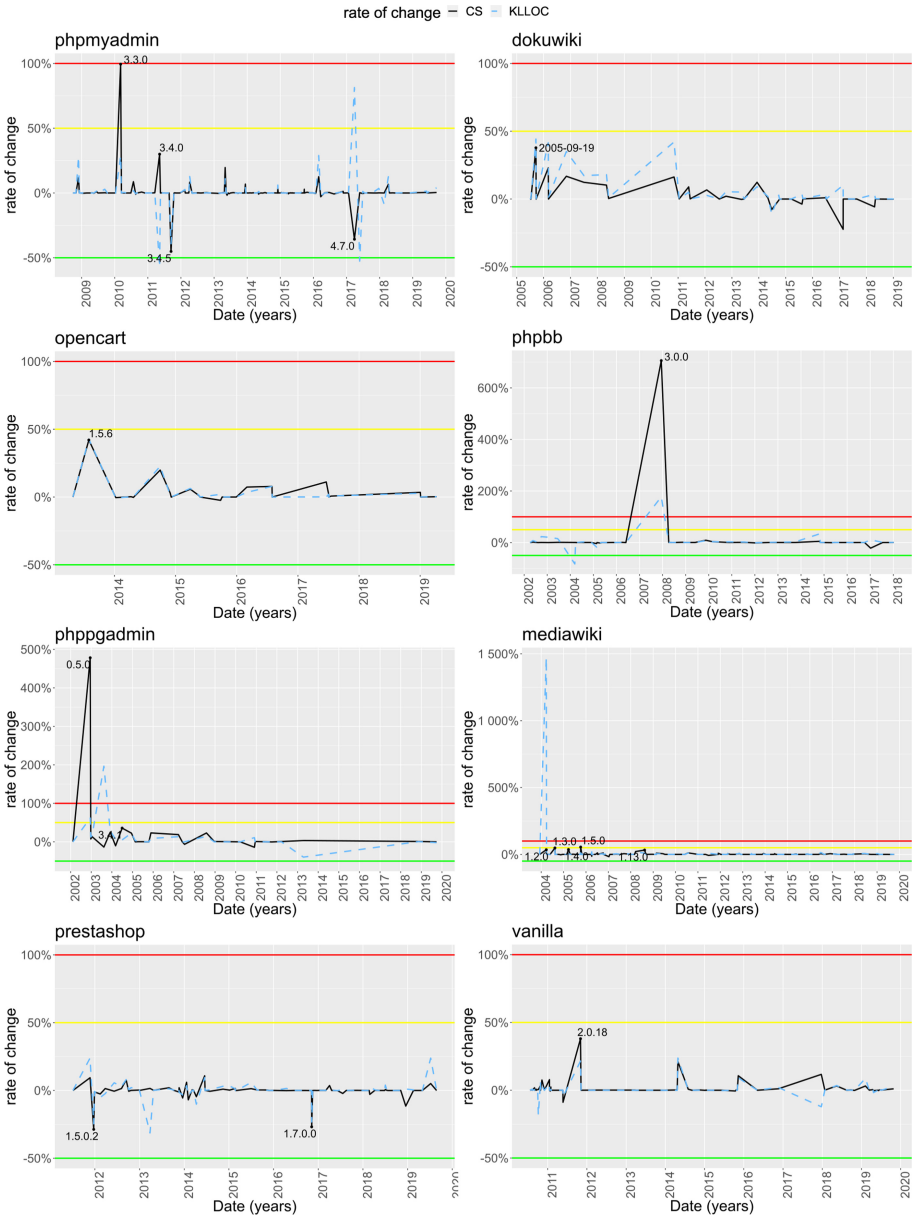

Fig. 2. CS and *KLLOC* rate of change evolution

possible to check those versions for anomalies (sudden variations) in the number/intensity of CS?

In Fig. 2, we can see the relative change of CS from the previous version (black), and the relative change in *KLLOC* (thousands of logical lines of code)

from the previous version (blue/dashed). *KLOC* (thousands of lines of code) is a well know measure, although here we used the logical lines of code. The relative change in the number of CS is given by:

$$\Delta cs = \frac{cs_i - cs_{i-1}}{cs_{i-1}} = \frac{cs_i}{cs_{i-1}} - 1 \quad (1)$$

where Δ is the rate of change, cs_i is the number of CS in the current version and cs_{i-1} is the number of CS in the previous version. The same is calculated for the size, i.e. the Logical Lines of Code (LLOC).

The sudden variations occur when there is a large increase in the number of CS and the size does not grow accordingly. It is also possible to get the version in which a lot of CS were removed by refactoring. For comparability sake, we use CS density or ρcs = number of CS/LLOC. We can now calculate the rate of change of the CS density, which we calculate in the same way as referred before for the CS number:

$$\Delta \rho cs = \frac{\rho cs_i - \rho cs_{i-1}}{\rho cs_{i-1}} = \frac{\rho cs_i}{\rho cs_{i-1}} - 1 \quad (2)$$

where $\Delta\rho cs$ is the rate of change of density of CS, ρcs_i is the density of CS in the current version and ρcs_{i-1} is the density of CS in the previous version. Figure 3 presents the evolution of CS density, making it easy to pinpoint the peaks, labeled according to the corresponding version.

In the graphs per application, we use lines representing thresholds, signaling the increase of 50% and 100% and the reduction of 50% in the rate of change in the density of CS. The thresholds can be changed according to application, team, quality, and company, if applicable.

In Table 3 we can observe the variance of *CS by KLLOC*, as well as the *Cyclomatic Complexity by LLOC* (aka Cyclomatic Complexity Density) from the current and previous versions, a long used objective metric for maintainability prediction [11].

Table 3. Metrics for the outliers

App	Version	Date	CS	LLOC	CS/kLLOC	var(CS/kLLOC)	CC/LLOC	CC/LLOC previous
phpmyadmin	3.3.0	2010-03-07	1145	130863	8.75	0.55	0.129	0.095
phpmyadmin	3.4.0	2011-05-11	1617	57338	28.20	2.02	0.425	0.137
phpmyadmin	4.7.1	2017-05-26	948	56192	16.87	1.11	0.391	0.321
phpbb	2.0.7	2004-03-13	226	12511	18.06	4.88	0.436	0.073
phpbb	3.0.0	2007-12-12	1781	32291	55.15	1.90	0.547	0.462
phppgadmin	5.1.0	2013-04-14	402	37098	10.84	0.71	0.152	0.095

4 Discussion

4.1 Introduction

We could find CS sudden variations in 4 web apps (*PhpMyAdmin*, *PhpBB*, *PhpPgAdmin* and *MediaWiki*). For example, *PhpMyAdmin* has 3 of these anomalies,

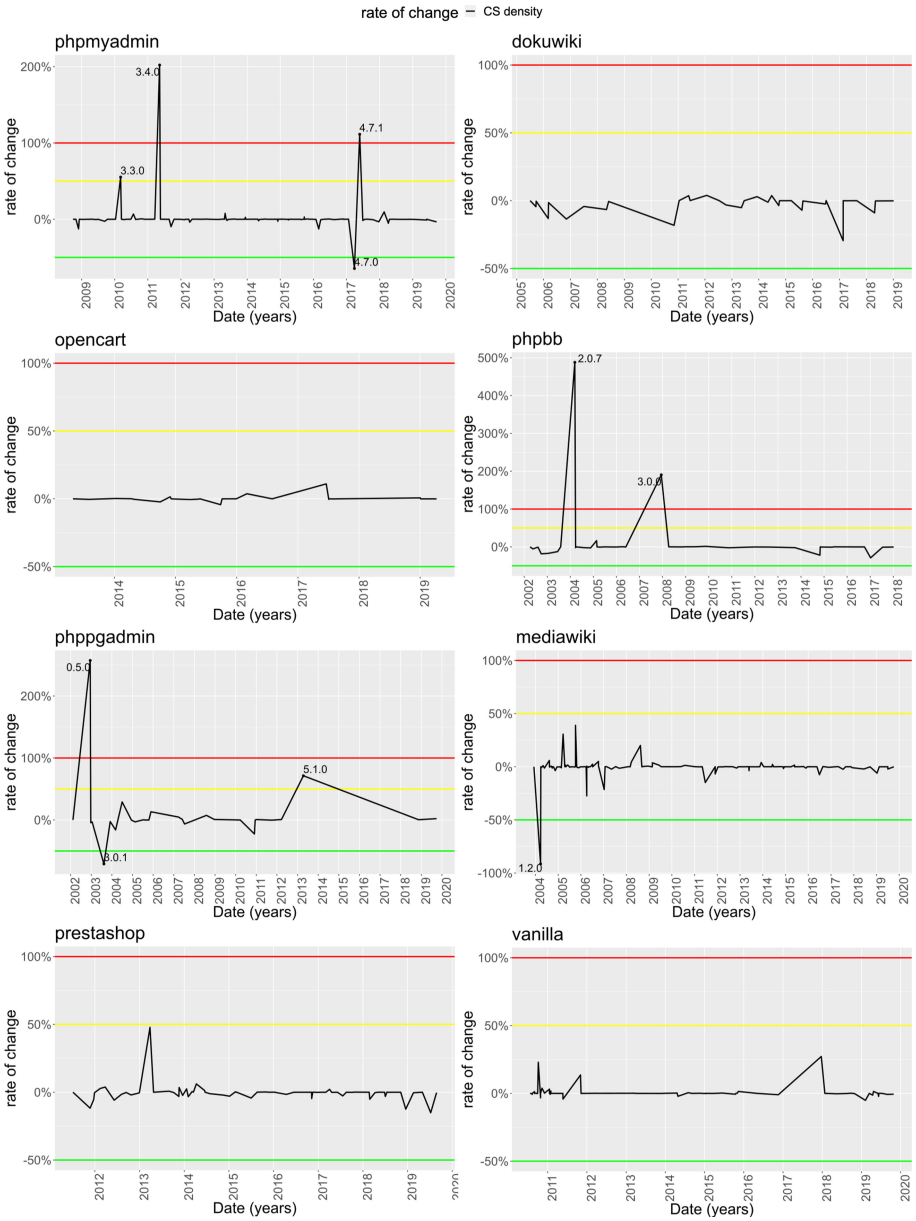

Fig. 3. CS density rate of change evolution

where CS rose abnormally without a correspondent rise in the app size. This can point to problems in those versions. We also spot a decrease anomaly, in version 4.7. By reading the code we confirmed that refactoring was then applied

considerably, which is in line with the changes observed in Fig. 2. The other 4 apps (*DokuWiki*, *OpenCart*, *PrestaShop* and *Vanilla*) are not exactly stable but do not have anomalies crossing the warning threshold. This somewhat simple method to implement - measuring $\Delta\rho cs$ - is able to detect the anomalies/sudden variations in the CS density.

4.2 Answers to Research Questions

RQ1 – How to detect sudden variations in the evolution of code smells? - As shown in Figs. 2 and 3 it is possible to detect these CS anomalies or peaks. We can analyze the rate of change (current version vs previous version) on the CS intensity alone (Eq. 1) or their relative rate of change by size (density), given by Eq. 2, which will be a more comparable metric across versions of the same application and even among applications.

RQ2 – When are the sudden variations of code smells in a new version considered too high? - or *When are there too many code smells?* Similar to limits in control charts, where you have limits equal to 3 times the variance, in this case we have to define thresholds, that can be chosen among development teams. We believe that a threshold of 50% will be sufficient to rise maintainability alerts. Knowing that we can never remove all the CS from an application, a 50% increase would raise a yellow flag, and a 100% increase would raise a red flag (stop immediately). Looking at Table 3 we can see that peaks also affect the *cyclomatic complexity per LLOC*, which it turn affects maintainability [11].

4.3 Applicability

Ideally, the removal of CS can be done in a "Total Quality" manner, where the developer is responsible to avoid the introduction of CS in the code, but often this is not possible. CS density thresholds detection can be integrated into an automation server tool such as *Jenkins*, that runs a battery of tests before a release - comparing it with the previous released version. If a threshold is reached, the release could be put on hold for some refactoring to be performed. This would act as a safeguard with the other tests in the test battery. Since *Jenkins* already has support for *PHPMD* in PHP projects, it is feasible to add our approach to the pipeline. The value of the threshold should be decided by each development team, depending on the development circumstances and requirements.

4.4 Comparison to Other Techniques

We also tried other methods, and among them, to apply SPC (Statistical Process Control) techniques with 2 or 3 standard deviations as limits, but we could not get limits due to the nature of the evolution (for long periods the value of number of CS was the same, then this value sudden increases). The main problem was that the standard deviation is 0 or close to 0. Another problem that arises with methods that use the average, for example [7], is that you have to know all

the history of the project, while in the method shown here, the computation at each point in time is just based on data collected from the previous and current versions.

4.5 Threats to Validity

Threats to construct validity concern the statistical relation between the theory and the observation, in our case the measurements and treatment to the data. We detected the CS using *PHPMD*, where we detected 18 smells. We could expand this study to consider even more CS, and compare the detection with other tools for PHP. However, some of them are based in *PHPMD*.

Threats to internal validity concern external factors we did not consider that could affect the variables and the relations being investigated. We can say that *PHPMD* allows to change the thresholds of the of the CS detection, but we worked with the default values for comparing between applications. These values can, however, be questioned for different applications.

Threats to conclusion validity concern the relation between the treatment and the outcome. One can argue that CS are often considered by absolute number or normalized by *LOC* or *LLOC*. However, our experiments have shown that the normalization by *LLOC* describes the peaks better.

Threats to external validity concern the generalization of results. We recognize that having just 8 web applications may not be enough for generalization sake.

5 Related Work

Much literature in software evolution has been published in the last decades, but few on web apps.

Longitudinal on CS: In [17] are described different phases in the evolution of CS and reported that components infected with CS have a higher change frequency. Later, [19] results indicate: CS lifespan is close to 50% of the lifespan of the systems. In [6] it is reported that a large percentage of CS was introduced in the creation of class/methods, but very few CS are removed. Later, [26] sustains that most CS are introduced when artifacts are created and not because of their evolution. In [20] the authors claim that the latest versions of the observed application have more CS/design issues than the oldest ones. They also note that the first version of the software is cleaner. In [8] the authors found that TD (Technical Debt) increases for most observed systems. However, TD normalized to the size of the system decreases over time in most systems. In [12], the authors conclude that CS can remain in the application code for years before removal, and CS detected and prioritized by linters, disappear from code before other CS. Recently [7] find that the number of TD items introduced through new code is a stable metric, although it presents some spikes; and also that the number of commits is not strongly correlated to the number of introduced TD items.

Non-longitudinal in Web Apps: These studies include [24], which found that for JS applications, and for the time before a fault occurrence, files without CS have hazard rates 65% lower than files with CS. As an extension to the previous paper, [14] show the results: files without CS have hazard rates of at least 33% lower than files with CS. In [2] study with PHP TD, which includes CS, they find that, on average, the number of times that a file with high TD is modified is 1.9 times more than the number of times a file with low TD is changed. In terms of the number of lines, the same ratio is 2.4. In [3] the authors find: complex and large classes and methods are frequently committed in PHP files; smelly files are more prone to change than non-smelly files. Studies in Java [18] report similar findings to the last two studies.

Longitudinal with PHP, without CS: Studies of this type include [15], where authors study 5 PHP web apps, and some aspects of their history, like unused code, removal of functions, use of libraries, stability of interfaces, migration to OOP, and complexity evolution. They found these systems undergo systematic maintenance. Later in [1], they expanded the study to analyze 30 PHP projects extracting their metrics, and found that not all of Lehman's laws of software evolution were confirmed in web applications.

Longitudinal with Fluctuations in CS: The only study we found regarding this aspect, which indeed is the most related to our work, is an already referred recent study [7], about the fluctuation in the evolution of technical dept (which includes CS). Their authors propose to divide applications into *stable* and *sensitive* (if they have spikes). To perform this classification, they use SMF (Software Metrics Fluctuation), which is defined as the average deviation from successive version pairs.

Related Work Discussion: The method described in [7] should prove effective for detecting anomalies or outliers in a continuous growing metric, or a metric that varies around a average value (fluctuates), with variations different from 0, witch was not the case with CS evolution with these apps. Another difference is that you have to known the history of the project.

6 Conclusions and Future Work

We studied the evolution and sudden variations of 18 CS in 8 widely used PHP web apps, across many years. It is important for PHP project managers to have an evolutionary perspective on the CS in an application, to decide on the allocation of resources to mitigate their maintainability effects. We observed sudden variations in CS occurrence in specific versions, whose root causes deserve investigation and are important for project managers to understand, especially for long-lived projects where managers' turnover inevitably happens.

In this paper we proposed a normalized technique that is simple to implement, for detecting those sudden variations in specific versions during the evolution of web apps, allowing us to unveil the CS story of a development project and make managers aware of the need for enforcing regular refactoring practices.

This technique can also be useful in an automation server pipeline, to add in the quality certification before the release. Our main goal was to achieve a simple technique that prevents a given version of software to be released with an extraordinary increase in CS, which later will be costly to maintainability. We used web applications in our study, but we think this method can be generalized to other domains or types of apps.

Regarding future work, we would like to increase the number of applications and CS studied, with more computing power. The obvious way forward here, is comparing PHP to Java since many more longitudinal studies on CS exist for the latter.

Acknowledgments. This work was partially supported by the Portuguese Foundation for Science and Technology (FCT) projects UIDB/04466/2020 e UIDP/04466/2020.

References

1. Amanatidis, T., Chatzigeorgiou, A.: Studying the evolution of PHP web applications. Inf. Softw. Technol. **72**(April), 48–67 (2016). https://doi.org/10.1016/j.infsof.2015.11.009
2. Amanatidis, T., Chatzigeorgiou, A., Ampatzoglou, A.: The relation between technical debt and corrective maintenance in PHP web applications. Inf. Softw. Technol. **90**, 70–74 (2017). https://doi.org/10.1016/j.infsof.2017.05.004
3. Bessghaier, N., Ouni, A., Mkaouer, M.W.: On the diffusion and impact of code smells in web applications. In: Wang, Q., Xia, Y., Seshadri, S., Zhang, L.-J. (eds.) SCC 2020. LNCS, vol. 12409, pp. 67–84. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59592-0_5
4. Bieman, J.M., Kang, B.K.: Cohesion and reuse in an object-oriented system. SIGSOFT Softw. Eng. Notes **20**(SI), 259–262 (1995). https://doi.org/10.1145/223427.211856
5. Bryton, S., Brito e Abreu, F., Monteiro, M.: Reducing subjectivity in code smells detection: experimenting with the long method. In: 7th International Conference on the Quality of Information and Communications Technology (QUATIC 2010), pp. 337–342. IEEE (2010)
6. Chatzigeorgiou, A., Manakos, A.: Investigating the evolution of code smells in object-oriented systems. Innov. Syst. Softw. Eng. **10**(1), 3–18 (2013). https://doi.org/10.1007/s11334-013-0205-z
7. Digkas, G., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P.: On the temporality of introducing code technical debt. In: Shepperd, M., Brito e Abreu, F., Rodrigues da Silva, A., Pérez-Castillo, R. (eds.) QUATIC 2020. CCIS, vol. 1266, pp. 68–82. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58793-2_6
8. Digkas, G., Lungu, M., Chatzigeorgiou, A., Avgeriou, P.: The evolution of technical debt in the apache ecosystem. In: Lopes, A., de Lemos, R. (eds.) ECSA 2017. LNCS, vol. 10475, pp. 51–66. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65831-5_4
9. Fontana, F.A., Ferme, V., Zanoni, M., Roveda, R.: Towards a prioritization of code debt: a code smell intensity index. In: 7th International Workshop on Managing Technical Debt (MTD 2015), pp. 16–24. IEEE (2015)

10. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (1999)
11. Gill, G.K., Kemerer, C.F.: Cyclomatic complexity density and software maintenance productivity. Trans. Softw. Eng. **17**(12), 1284 (1991)
12. Habchi, S., Rouvoy, R., Moha, N.: On the survival of android code smells in the wild. In: 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft 2019), pp. 87–98. IEEE, May 2019. https://doi.org/10.1109/MOBILESoft.2019.00022
13. Herbold, S., Grabowski, J., Waack, S.: Calculation and optimization of thresholds for sets of software metrics. Empir. Softw. Eng. **16**(6), 812–841 (2011). https://doi.org/10.1007/s10664-011-9162-z
14. Johannes, D., Khomh, F., Antoniol, G.: A large-scale empirical study of code smells in JavaScript projects. Softw. Qual. J. **27**(3), 1271–1314 (2019). https://doi.org/10.1007/s11219-019-09442-9
15. Kyriakakis, P., Chatzigeorgiou, A.: Maintenance patterns of large-scale PHP web applications. In: 30th International Conference on Software Maintenance and Evolution (ICSME 2014), pp. 381–390 (2014). https://doi.org/10.1109/ICSME.2014.60
16. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice. Springer, Heidelberg (2007)
17. 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: 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM 2009), pp. 390–400. IEEE (2009). https://doi.org/10.1109/ESEM.2009.5314231
18. Palomba, F., Bavota, G., Penta, M.D., Fasano, F., Oliveto, R., Lucia, A.D.: On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empir. Softw. Eng. **23**(3), 1188–1221 (2017). https://doi.org/10.1007/s10664-017-9535-z
19. Peters, R., Zaidman, A.: Evaluating the lifespan of code smells using software repository mining. In: European Conference on Software Maintenance and Reengineering (CSMR 2012), pp. 411–416. IEEE (2012). https://doi.org/10.1109/CSMR.2012.79
20. Rani, A., Chhabra, J.K.: Evolution of code smells over multiple versions of softwares: an empirical investigation. In: 2nd International Conference for Convergence in Technology (I2CT 2017), vol. 2017-January, pp. 1093–1098. IEEE, December 2017. https://doi.org/10.1109/I2CT.2017.8226297
21. Rasool, G., Arshad, Z.: A review of code smell mining techniques. J. Softw. Evol. Process **27**(11), 867–895 (2015). https://doi.org/10.1002/smr.1737
22. Pereira dos Reis, J., Brito e Abreu, F., de Figueiredo Carneiro, G., Anslow, C.: Code smells detection and visualization: a systematic literature review. Arch. Comput. Methods Eng. (2021). https://doi.org/10.1007/s11831-021-09566-x
23. 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
24. Saboury, A., Musavi, P., Khomh, F., Antoniol, G.: An empirical study of code smells in JavaScript projects. In: 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER 2017), pp. 294–305. IEEE, March 2017. https://doi.org/10.1109/SANER.2017.7884630

25. Singh, S., Kaur, S.: A systematic literature review: refactoring for disclosing code smells in object oriented software. Ain Shams Eng. J. **9**(4), 2129–2151 (2018). https://doi.org/10.1016/j.asej.2017.03.002
26. Tufano, M., et al.: When and why your code starts to smell bad (and whether the smells go away). Trans. Softw. Eng. **43**(11), 1063–1088 (2017). https://doi.org/10.1109/TSE.2017.2653105
27. W3techs.com: Usage Statistics and Market Share of Server-side Programming Languages for Websites, January 2021. https://w3techs.com/technologies/overview/programming_language
28. Zhang, M., Hall, T., Baddoo, N.: Code bad smells: a review of current knowledge. J. Softw. Maintenance Evol. **23**(3), 179–202 (2011). https://doi.org/10.1002/smr.521