The Journal of Systems & Software 200 (2023) 111644

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

PHP code smells in web apps: Evolution, survival and anomalies☆

Américo Rio [a,b,*], Fernando Brito e Abreu [a]

[a] Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR, Lisboa, Portugal
[b] NOVA Information Management School (NOVA IMS), Universidade Nova de Lisboa, Lisboa, Portugal

ARTICLE INFO

Article history:
Received 7 September 2022
Received in revised form 6 February 2023
Accepted 8 February 2023
Available online 11 February 2023

Dataset link: https://github.com/studydatacs/servercs, https://doi.org/10.5281/zenodo.7626150

Keywords:
Code smells
PHP
Software evolution
Survival
Web apps

ABSTRACT

Context: Code smells are symptoms of poor design, leading to future problems, such as reduced maintainability. Therefore, it becomes necessary to understand their evolution and how long they stay in code. This paper presents a longitudinal study on the evolution and survival of code smells (CS) for web apps built with PHP, the most widely used server-side programming language in web development and seldom studied.

Objectives: We aimed to discover how CS evolve and what is their survival/lifespan in typical PHP web apps. Does CS survival depend on their scope or app life period? Are there sudden variations (anomalies) in the density of CS through the evolution of web apps?

Method: We analyzed the evolution of 18 CS in 12 PHP web applications and compared it with changes in app and team size. We characterized the distribution of CS and used survival analysis techniques to study CS' lifespan. We specialized the survival studies into localized (specific location) and scattered CS (spanning multiple classes/methods) categories. We further split the observations for each web app into two consecutive time frames. As for the CS evolution anomalies, we standardized their detection criteria.

Results: The CS density trend along the evolution of PHP web apps is mostly stable, with variations, and correlates with the developer's numbers. We identified the smells that survived the most. CS live an average of about 37% of the life of the applications, almost 4 years on average in our study; around 61% of CS introduced are removed. Most applications have different survival times for localized and scattered CS, and localized CS have a shorter life. The CS survival time is shorter and more CS are introduced and removed in the first half of the life of the applications. We found anomalies in the evolution of 5 apps and show how a graphical representation of sudden variations found in the evolution of CS unveils the story of a development project.

Conclusion: CS stay a long time in code. The removal rate is low and did not change substantially in recent years. An effort should be made to avoid this bad behavior and change the CS density trend to decrease.

1. Introduction and motivation

"Code smell" is a term introduced by Kent Beck in a chapter of the famous Martin Fowler's book (Fowler, 1999) to describe a surface indication in source code that usually corresponds to a deeper problem. Code smells (CS) are symptoms of poor design and implementation choices that may lead to increased defect incidence, decreased code comprehension, and longer times to release. CS come in different shapes, such as a method with too many parameters or a complex body. Their detection may be subject to some subjectivity (Bryton et al., 2010; Pereira dos Reis et al., 2017), but that issue is beyond the scope of this paper. Keeping CS in the code may be considered a manifestation of technical debt, a term coined by Cunningham (1992). Usually, to remove a CS, a refactoring operation is performed (Fowler, 1999).

The Software Engineering community has been proposing new techniques and tools, both for CS detection and refactoring (Fernandes et al., 2016; Pereira dos Reis et al., 2021; Zhang et al., 2011), in the expectation that developers become aware of their presence and get rid of them. However, a good indicator of the success of that quest is the reduction of CS survival time (lifespan), the elapsed time since one CS appears until it disappears due to refactoring or code dropout. Therefore, software evolution (longitudinal) studies are required to assess CS survival time while revealing other aspects of *CS evolution*, such as possible causes, trends, and evolution anomalies.

☆ Editor: Dr. Alexander Chatzigeorgiou.

* Corresponding author at: Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR, Lisboa, Portugal.
E-mail addresses: jaasr@iscte-iul.pt, americo.rio@novaims.unl.pt (A. Rio), fba@iscte-iul.pt (F. Brito e Abreu).

https://doi.org/10.1016/j.jss.2023.111644

The most frequent target in CS studies are Java desktop apps (Rasool and Arshad, 2015; Singh and Kaur, 2018; Pereira dos Reis et al., 2021). However, few CS studies exist in other domains (web, mobile) and other languages. Web applications differ from desktop and mobile applications since some components or code segments run on a web server and others on a browser. The code that runs on a server is called the server-side code (using PHP, C#, Java, Python, Node.js' JavaScript, or other languages). The latter can communicate with other servers, such as a database or email servers, besides the host file system (e.g., storing files). The client-side code is the code that renders or runs inside a browser (HTML, CSS, JavaScript). Typical web apps have both server and client code. However, depending on the application's architecture, this code can be together as a monolithic app, separated as a distributed app (Frontend/Backend), or as a microservices architecture.[1]

In particular, evolution studies of CS in web apps using PHP as server language are still scarce, as reported in the related work section. We aim to mitigate this gap through this study on the evolution and survival of CS in the server-side PHP code of typical web apps. The PHP programming language is the most used server-side programming language in web apps.[2] In this longitudinal study, we considered for each web app as many years as possible, summing up to 811 releases. We intend to discover how CS evolve in web apps, characterize CS survival/timespan, and use a method to reveal sudden changes in CS density throughout time.

This paper extends and updates our previous research work reported in Rio and Brito e Abreu (2019, 2021). Since then, we collected more metrics from a larger sample of PHP web apps, which allowed us to address more research questions by considering more factors in data analysis. We also added novel discussions and conclusions on the evolution of CS and its relation to the evolution of app and team sizes.

We structured this paper as follows: After the introduction with the motivation in Section 1, Section 2 overviews the related work on longitudinal studies on CS and web apps. Next, Section 3 introduces the study design, and Section 4 describes the results of our data analysis. After Section 5 deals with identifying validity threats, and in Section 6 we discuss the findings and what they mean to developers and scholars. Finally, Section 7 outlines the significant conclusions and required future work.

2. Related work

Much literature on software evolution has been published in recent decades, but few on web apps or CS evolution. We reviewed the literature on the evolution of CS, with and without survival techniques, studies with web apps or web languages (not evolution), and evolution studies with PHP, but not with CS. We did not find any evolution study on CS in PHP web apps.

2.1. Evolution on CS

The evolution of 2 CS in 2 open-source systems is analyzed in Olbrich et al. (2009). The authors compare the increase and decrease of classes infected with CS and total classes in windows of 50 commits in a *SVN (Subversion)* repository. Their results show different phases in the evolution of CS and that CS-infected components exhibit a higher change frequency. Later, in Olbrich et al. (2010), they investigate if the higher change frequency is true for God Classes and Brain Classes in 7–10 years of 3 systems. Without normalization, God and Brain Classes were changed more frequently and contained more defects than other kinds of classes, but when normalized by size, they were less subject to change and had fewer defects than other classes.

The lifespan of CS and developers' refactoring behavior in 7 systems mined from a *SVN* repository is discussed in Peters and Zaidman (2012). The authors' conclusions are: (a) CS lifespan is close to 50% of the lifespan of the systems; (b) engineers are aware of CS but are not very concerned with their impact, given the low refactoring activity; (c) smells at the beginning of the systems life are prone to be corrected quickly.

Rani and Chhabra (2017) perform an empirical study on the distribution of CS in 4 versions of 3 software systems. The study concludes that: (a) the latest version of the software has more design issues than older ones; (b) the "God" smell has more contribution to the overall status of CS, and "Type Checking" less. However, they also note that the first version of the software is cleaner.

Digkas et al. (2017) analyze 66 Java open-source software projects on the evolution of technical debt over five years with weekly snapshots. They calculate the technical debt time-series trends and investigate the components of this technical debt. Their findings are: (a) technical debt together with source code metrics increases for most of the systems; (b) technical debt normalized to the size of the system decreases over time in most systems; (c) some of the most frequent and time-consuming types of technical debt are related to improper exception handling and code duplication. Later in Digkas et al. (2020), the authors investigate the reasons for introducing technical debt, within 27 systems, in 6-month sliding temporal windows. Their findings are: (a) the number of Technical Debt Items introduced through new code is a stable metric, although it presents some spikes; (b) the number of commits performed is not strongly correlated to the number of introduced Technical Debt Items. They propose to divide applications into *stable* and *sensitive* (if they have spikes). They use SMF (Software Metrics Fluctuation) to perform this classification, defined as the average deviation from successive version pairs.

2.2. Evolution on CS, with survival analysis

Chatzigeorgiou and Manakos (2010) study the evolution of 3 CS in a window of successive versions of 2 Java systems. They extend the work in Chatzigeorgiou and Manakos (2014) to use four smells and survival analysis with survival curves. Their conclusions are: (a) in most cases, CS persist up to the latest examined version, thus accumulating; (b) survival analysis shows that smells "live" for many versions; (c) a significant percentage of the CS was introduced in the creation of a class/method; (d) very few CS are removed from the projects, and their removal was not due to refactoring activities but a side effect of adaptive maintenance.

The change history of 200 projects is reported by Tufano et al. (2015) and later extended to include survival analysis in Tufano et al. (2017). Reported findings are: (a) most CS instances are introduced when an artifact is created and not because of its evolution, (b) 80 percent of CS survive in the system, and (c) among the 20 percent of removed instances, only 9 percent are removed as a direct consequence of refactoring operations. In Tufano et al. (2016), the authors analyze when test smells (TS) occur in Java source code, their survivability, and if their presence is associated with CS. They found relationships between TS and CS. They include survival analysis to study the lifespan of TS.

The survival of (Java) Android CS with 8 Android CS is analyzed by Habchi et al. (2019). The authors conclude that: (a) CS can remain in the codebase for years before being removed; (b) in terms of commits, it takes 34 effective commits to remove 75% of them; (c) Android CS disappear faster in bigger projects with higher releasing trends; (d) CS that are detected and prioritized by linters tend to disappear before other CS.

[1] https://www.martinfowler.com/articles/microservices.html

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

2.3. Cross-sectional or mixed studies in web apps or web languages

These studies include Saboury et al. (2017), whose authors find that, for *JavaScript* applications, files without CS have hazard rates 65% lower than files with CS. As an extension to the previous paper, Johannes et al. (2019) conclude that files without CS have hazard rates of at least 33% lower than files with CS. Amanatidis et al. (2017), a study with PHP TD (Technical Debt) which includes CS, the authors 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. Bessghaier et al. (2020), the authors find: (a) complex and large classes and methods are frequently committed in PHP files; (b) smelly files are more prone to change than non-smelly files. Studies in Java (Palomba et al., 2018) report similar findings to the previous two studies.

2.4. Evolution with PHP, without CS

Studies of this type include Kyriakakis and Chatzigeorgiou (2014), 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. In addition, they found that these systems undergo systematic maintenance. Later, in Amanatidis and Chatzigeorgiou (2016), they expanded the study to analyze 30 PHP projects and found that not all of Lehman's laws of software evolution (Lehman, 1996) were confirmed in web applications.

2.5. Studies comparing types of CS

The following studies compare CS at different abstraction levels or different scopes. In Fontana et al. (2019), the authors try to understand if architectural smells are independent of CS or can be derived from a CS or one category of them. The method used was to analyze correlations among 19 CS, 6 categories, and 4 architectural smells. After finding no correlation, they conclude that they are independent of each other. The paper by Sharma et al. (2020) aims to study architecture smell characteristics and investigate correlation, collocation, and causation relationships between architecture and design smells. They used 19 smells, mining C# repositories. Their findings are: (a) smell density does not depend on repository size; (b) architecture smells are highly correlated with design smells; (c) most of the design and architecture CS pairs do not exhibit collocation; (d) causality analysis reveals that design smells cause architecture smells (with statistical means).

3. Methods and study design

3.1. Research questions

We intend to study CS's evolution in typical web apps, characterizing it quantitatively and qualitatively and uncovering its trends and probable causes. Furthermore, comparing the survival time or removal rate with desktop counterparts will be interesting. Some studies of code smells use various releases of software but independently. However, the evolution of CS is typically performed with a longitudinal study, with time series. Therefore, we want to characterize, find trends, and correlate the CS evolution with team and app size metrics. Also, we want to assess how long CS stay in the code before removal, i.e., how long they survive, and their distributions, insertion, and removal rate. This study is done per app and CS, providing insight into the applications with longer CS lifespans and what CS typically stay in code longer. The lifespan of individual CS could reveal the relative importance developers give to each and also aid in assessing the difficulty of diagnosing some of them.

Survival studies can be further specialized. For instance, CS's effect can vary widely in breadth (localized or scattered). In localized ones, the scope is a method or a class (e.g., in Java Long Method, Long Parameter List, God Class), while the influence of others may be scattered across large portions of a software system (e.g., in Java Shotgun Surgery, Deep Inheritance Hierarchies, or Coupling Between Classes). The other factor we are concerned with regards a superordinate temporal analysis: we want to investigate whether the CS survival time, introduction, and removal changed over time, possibly caused by CS awareness by the developers or help in detecting CS from recent IDE's or tools.

During a previous analysis study, we noticed what seemed to be sudden changes in CS density of web applications. These anomalous situations may occur in both directions (steep increase or steep decrease) deserved our attention, either for recovering the history of a project or, in a quality pipeline, to provide awareness or raise alerts to decision-makers that something unusual is taking place for good or bad. We already know that CS hinder the quality of the code and its app maintainability, and if we have a mechanism to avoid the sudden increases in the density of CS, we can prevent deploying a release with less code quality before it gets released.

In sum, we aim to provide answers to the following research questions in the context of web apps using PHP as server-side language:

RQ1 — How to characterize the evolution of CS? - In this question, we study the evolution of CS, both in absolute number and density (divided by logical/effective lines of code - statement lines in the code), to unveil the trends and patterns of CS evolution. Then, we try to find possible causes for the evolution patterns, looking at team and app metrics.

RQ2 — What is the distribution and survival/lifespan of CS? - We intend to study the absolute and relative distribution (by CS and app), the survival time of CS (the time from CS introduction in the code until their removal) and the removal percentage of CS. The answer to this question will help us understand the life of the 18 different CS in the web apps and compare it between web apps.

RQ3 — Is the survival of localized CS the same as scattered CS? - We compare the lifespan and survival curves for localized CS (CS that are solely in a specific location) and scattered CS (CS that span over multiple classes or files). The former are easier to refactor, and the latter more difficult (to refactor them, we have to edit several files) and supposedly more harmful. We also compare the removal rate of the two categories of CS. This will help us understand which CS scope live longer in the systems.

RQ4 — Does the survival of CS vary over time? - We divide each application into two consecutive equal time frames and make the same methods as the previous question to assess if survival time (lifespan) is the same across these time frames. The answer to this question will help us understand if the survival time of CS is different in both halves and, if so, which is longer. We also compute and compare the number of introduced and removed CS in both halves. This will reveal if special care is being taken with CS (by lower introduction rate or lifespan, or higher removal rate in the second half of the history of the application).

RQ5 — How to detect anomalous situations in CS evolution? - We intend to find a way to detect anomalies in the evolution of CS, giving us the means to avoid a release to the public with less quality software code before it happens. This detection is also helpful in unveiling the history of the project.

We performed a longitudinal study encompassing 12 web apps and 18 CS to answer the research questions. For RQ3 only,

we used a subset of 6 code smells as surrogates of more scattered or localized scopes. The study will help researchers, practitioners and software managers to increase their knowledge of the evolution behavior of PHP web apps CS and help project managers keep CS's density under control. Next, we describe the applications and the CS used in the study.

3.2. Applications sample

This work aims to study the evolution and survival of CS in web apps built with PHP as the server-side language. PHP was built especially for the web, one of the few programming languages that can make a web app with or without a framework. A distributed app (Frontend/Backend) or an app/system with microservices architecture means separated apps, and the server-side code is no longer a web application but a set of web services or similar; therefore, the PHP web applications studied here are monolithic.

The inclusion and exclusion criteria used for selecting the sample of PHP web apps were the following: Inclusion criteria: (i) the code should be available (i.e., should be open source); (ii) complete applications/self-contained applications (monolithic), with both client- and server-side code, taken from the GitHub top forked apps; (iii) programmed with an object-oriented style (OOP)[3]; (iv) covering a long period (minimum five years, more if possible — some open source applications have long intervals between releases); Exclusion criteria: (i) libraries; (ii) frameworks or applications used to build other applications; (iii) web apps built using a framework;

We selected the 50 most forked apps in *GitHub* at the beginning of 2019. We tested by description and code inspection the adherence to the criteria from that list. The OOP criterion is because most of the CS detected by PHPMD (and used in the study) are CS for OOP applications. Due to the OOP criterion, we had to exclude several well-known apps. We excluded frameworks and libraries because we wanted to study typical web apps (apps with both client- and server-side code). Some frameworks will have almost exclusively server-side and console code to run in the command line, so although they use web languages, they are not typical web apps. In the same line of thought, PHP libraries only have server-side code, so we removed them. We also excluded web apps built with frameworks because we want to analyze the applications themselves and not the frameworks upon which they were built. For example, when PHP frameworks first appeared, having the framework code and external library code mixed with the application was common. For the same reason, later, in the pre-processing phase, we excluded folders with external code, such as libraries or other applications.

With this criterion, we considered four applications for the first survival study in 2019, doubling the number of apps when started to extend. Later, we added four more applications (but we had to go over the 50 most forked because of criterion). Most of the GitHub projects that came on top of the list are either frameworks, libraries, or applications made with frameworks, excluded by criterion.

We collected as many releases for each app as possible. Because of the survival transformation, we had to consider all the consecutive releases, a total of 811 releases. However, sometimes we could not get the beginning of the app lifecycle either because not all releases were available online or did not match the OOP criterion in the earlier releases (for example, *phpMyAdmin* only from release 3.0.0 upwards). The OOP criterion exists because the CS used are for OOP code. A brief characterization of each selected web app follows:

- *phpMyAdmin* is an administration tool for MySQL and MariaDB. The initial release was in September 1998, but we only considered release 3.0.0 upwards due to a lack of OOP support and missing release files.
- *DokuWiki* is a wiki engine that works on plain text files and does not need a database.
- *OpenCart* is an online store management system, or e-commerce or shopping cart solution. It uses a MySQL database and HTML components. The first release was on May 99.
- *phpBB* is an internet forum/bulletin board solution, supports multiple database (PostgreSQL, SQLite, MySQL, Oracle, Microsoft SQL Server) and started in December 2000.
- *phpPgAdmin* is an administration tool for *PostgreSQL*. It started as a fork of *phpMyAdmin* but now has a completely different code base.
- *MediaWiki* is a wiki engine developed for Wikipedia. It was first released in January 2002 and dubbed *MediaWiki* in 2003.
- *PrestaShop* is an e-commerce solution. It uses a MySQL database. It started in 2005 as phpOpenStore and was renamed in 2007 to *PrestaShop*.
- *Vanilla* is a lightweight Internet forum package/solution. It was first released in July 2006.
- *Dolibarr* is an enterprise resource planning (ERP) and customer relationship management (CRM), including other features. The first release came out in 2003.
- *Roundcube* is a web-based IMAP email client. The first stable release of *Roundcube* was in 2008.
- *OpenEMR* is a medical practice management software that supports Electronic Medical Records (EMR) and migrated to open software in 2002.
- *Kanboard* is a project management application. Uses a Kanban board to implement the Kanban process management system. Initial release 2014.

Table 1 shows the complete list of applications. The LOC and Classes numbers are from the last release and were measured by the PHPLOC[4] tool. As with detecting code smells, we excluded each app's folders from other vendors in the LOC measures. As an example, the excluded folders for two of the apps were: *phpMyAdmin: doc, examples, locale, sql, vendor, contrib, pmd* and for *Vanilla: cache, confs, vendors, uploads, bin, build, locales, resources*. The excluded folders for the other applications are on the replication package (file *excluded folders.txt*).

3.3. Code smells sample

This section describes the code smells used in the studies. To collect them we used *PHPMD*,[5] an open-source tool that detects CS in PHP. *PHPMD* is used as a plugin in some IDE (example: *PHPStorm*) and other tools that act as a front end to code analyzers. From the supported CS in *PHPMD*, we ask a specialist in Java CS to help choose the most similar to the ones used in the Java world. The "unused code" list of CS is commonly used in Java as a group (unused code CS).

A brief characterization of all CS used is presented in Table 2. The CS names are the ones used by *PHPMD*, but we added (Exc.) to mean excessive, denoting that it is a CS and not a metric. The thresholds used are the default ones used in *PHPMD*, which in turn came from PMD,[6] and are generally accepted[7] from the

[3] PHP can be used with a pure procedural style; the object-oriented style became available from release 4 onwards.

[4] https://phpqa.io/projects/phploc.html

[5] https://phpmd.org/

[6] https://pmd.github.io/

[7] https://pmd.github.io/latest/pmd_java_metrics_index.html

Table 1
Characterization of the target web apps.

Name	Purpose	#Releases(period)	Releases	LOC[a]	#Classes[a]
phpMyAdmin	Database administration	179 (09/2008–09/2019)	3.0.0–4.9.1	163057	375
DokuWiki	Wiki	40 (07/2005–01/2019)	2005-07-01–2018-04-22b	118000	294
OpenCart	Shopping cart	26 (04/2013–04/2019)	1.5.5.1–3.0.3.2	200698	945
phpBB	Forum/bulletin board	50 (04/2012–01/2018)	2.0.0–3.2.2	283872	864
phpPgAdmin	Database administration	29 (02/2002–09/2019)	0.1.0–7.12.0	34661	31
MediaWiki	Wiki	138 (12/2003–10/2019)	1.1.0–1.33.1	495554	1597
PrestaShop	Shopping cart	74 (06/2011–08/2019)	1.5.0.0–1.7.6.1	428513	2074
Vanilla	Forum/bulletin board	63 (06/2010–10/2019)	2.0–3.3	193422	533
Dolibarr	ERP/CRM	83 (02/2006–12/2019)	2.0.1–10.0.5	766533	625
Roundcube	Email Client	31 (04/2014–11/2019)	1.0.0–1.4.1	77918	184
OpenEMR	Medical Records	33 (06/2005–10/2019)	2.7.2–5.0.2.1	792412	1725
Kanboard	Project management	65 (02/2014–12/2019)	1.0.0–1.2.13	88731	450

[a]On last release.

Table 2
Characterization of the target code smells — original CS names used by *PHPMD*. The added (Exc.) means excessive, denoting that is a CS and not a metric.

Code smell	Description	Threshold
(Exc.)CyclomaticComplexity	Excess-method number decision points plus one	10
(Exc.)NPathComplexity	Excess-method number acyclic execution paths	200
ExcessiveMethodLength	(Long method) method is doing too much	100
ExcessiveClassLength	(Long Class) class does too much	1000
ExcessiveParameterList	Method with too many parameters	10
ExcessivePublicCount	Excess public methods/attributes 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	Excess-sum complexities all methods in class	50
(Exc.)NumberOfChildren	Class with an excessive number of children	15
(Exc.)DepthOfInheritance	Class with too many parents	6
(Exc.)CouplingBetweenObjects	Class with too many dependencies	13
DevelopmentCodeFragment	Development Code:var_dump(),print_r()	1
UnusedPrivateField	Unused private field	1
UnusedLocalVariable	Unused local variable	1
UnusedPrivateMethod	Unused private method	1
UnusedFormalParameter	Unused parameters in methods	1

references in the literature (Bieman and Kang, 1995; Lanza and Marinescu, 2007; Mccabe, 1976). These thresholds should be considered baselines and could be optimized using an approach like the one proposed in Herbold et al. (2011). The latter concludes that metric thresholds often depend on the properties of the project environment, so the best results are achieved with thresholds tailored to the specific environment. In our case, where we have 12 web apps, each developed by a different team, such optimization would lead to specific thresholds for each app, adding confounding effects to the comparability between apps we want to carry out in this study.

3.4. Data collection and preparation workflow

The workflow of our study (see Fig. 1) included a data collection and preparation phase before the data analysis phase and was fully automated using several tools. We performed the following steps in the data collection and preparation phase:

1. We downloaded the source code of all releases of the selected web apps, in *ZIP* format, from *GitHub*, *SourceForge*, or private repositories, **except** the alpha, beta, release candidates, and corrections for old releases, i.e., everything out of the main branch. We considered only the stable branch when we had two branches in parallel. We used only the higher one when we had two releases on the same day. During this step, we created a database table with the application releases, which were later exported to a *CSV* file containing the timestamps for each downloaded release. Next, we extracted the *ZIP* files (one per each release of each app) to flat files on the file system of our local computer (one folder per each release of each app).
2. Using *phpMyAdmin*, we imported the *CSV* file with the releases' timestamps created in the first step to the CS database, a *MySQL* database.
3. Using *PHPMD*, we obtained the CS and respective attributes from all releases and stored them in XML file format (one file per application release). We *excluded some directories* not part of the applications (vendor libraries, third-party code). The zips from GitHub and other locations had these folders to make the application run without additional downloads.
4. We used the *CodeSmells2DB* PHP script, developed by the first author, to read the previous XML files and, after some format manipulation, store the corresponding information in the CS *MySQL* database. The data records at this point are divided by release/smell.
5. With the script *CSLong2CSSurv*, developed by the first author, we transformed the code smells information stored in the *MySQL* database into a format suitable for survival analysis by statistics tools. That includes the date when each unique code smell was first detected and when it disappeared if that was the case. The script stores the results of this transformation back in the database in other tables. Then, the script performed a *data completion* step, where it also calculates the censoring (removal=1 or not removal=0) and survival periods. Later, we exported the results to *CSV* format (one file per app) in preparation for the data analysis phase.

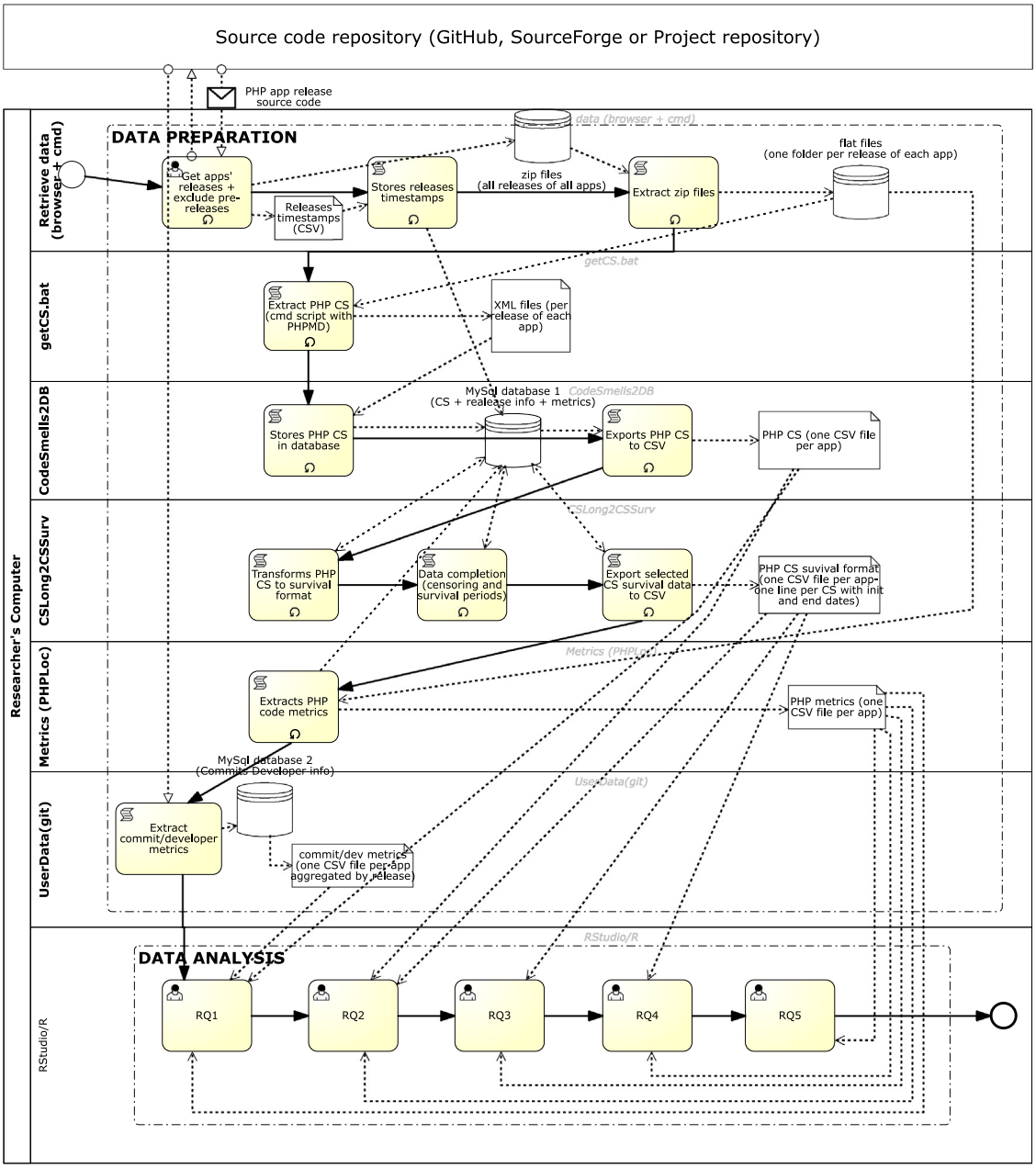


Fig. 1. Workflow of the data preparation and analysis phases.

6. We used *PHPLOC* to extract several code metrics from the source code of each release of each app, storing them in the same database and later exporting them to a CSV file per app. We excluded the folders from third-party code and libraries (different folders for each application).
7. Finally, we cloned all the git repositories of the applications and retrieved developers and commits data that we aggregated by app release, using a conjunction of *command line*, *git*, and *SQL* commands. This data was stored in a second database and later exported to *CSV* files (one per app), concluding this data preparation phase.

The *censoring* activity mentioned earlier encompassed transforming the collection of detected CS instances for each release of each web app to a table with the *life* of each instance, including the date of its first appearance, removal date (if occurred), and a censoring value meaning the following:

- Censored=1, the smell disappeared, usually due to a refactoring event;
- Censored=0, the code smell is still present at the end of the observation period.

For the anomalies study (in RQ5), we already had all the necessary data, the CS by release, and the logical (Effective) lines of code.

For replication purposes, the collected dataset is made available to the community in CSV format.[8]

3.5. Statistics used

We used the R statistics tool with several packages for all the analysis studies to perform the evaluation, correlations, and

8 https://github.com/studydatacs/servercs

output of the graphics. For the correlations, we used the standard R *cor* function (p-values given by *cor.test*).

We used survival analysis in RQ2 to RQ4. *Survival analysis* encompasses a set of statistical approaches that investigate the time for an event to occur (Clark et al., 2003). The questions of interest can be the average survival time (the one that interests us the most) and the probability of survival at a certain point, usually estimated by the Kaplan–Meier method. In addition, the log-rank test can be used to test for differences between survival curves for groups. Also, we can calculate the hazard function, i.e., the likelihood of the event occurring (not used in the present study).

The survival probability $S(t_i)$ at time t_i is given by:

$$S(t_i) = S(t_{i-1})(1 - \frac{d_i}{n_i}) \quad (1)$$

where, $S(t_{i-1})$ is the probability of being alive at t_{i-1}, n_i is the number of cases alive just before t_i, d_i is the number of events at t_i and the initial conditions are $t_0 = 0$ and S(0) = 1.

We used the R packages **survival**[9] and **survminer**.[10] Regarding survival time, the two average measures are the median (50% probability of end of the life of the subject to occur) or the restricted mean ("rmean"), .i.e., the mean taking into account the end of the life of the subject (in our case the removal of the code smell). Because the median survival time is insensitive to outliers, it better describes the mean lifespan of the CS. However, in some cases, the median cannot be calculated (it requires that the survival curve goes under 50%), and the restricted mean ("rmean") can typically be calculated. Kaplan–Meier survival curve is a plot of Kaplan and Meier (1958) survival probability against time and provides a valuable summary of the data that can be used to estimate measures such as median survival time.[11]

In question 3 and 4 we used the **log-rank test** (Schuette, 2021) and the two different co-variables (type/scope and time). The log-rank test is used to compare survival curves of two groups, in our case, two types of CS. It tests the null hypothesis that survival curves of two populations do not differ by computing a *p*-value. If the *p*-value is less than 0.05, the survival curves are different.

3.6. Methodology for each RQ

3.6.1. RQ1 — How to characterize the evolution of CS?

We study the evolution of CS in all 12 apps, both in CS absolute number and in CS density (code smells by kLLOC[12]) both qualitatively and qualitatively. For every consecutive public release, we have detected the absolute values of each CS and the corresponding CS density (absolute number by the size of the application). The CS density is obtained by dividing the CS absolute number by the size of the app. The size of the app (in Logical lines of code, or LLOC that measures only the PHP statements) is measured with *phpLOC*, excluding the same folders excluded in CS detection. We used LLOC (only PHP statement lines) because PHP files can contain HTML, CSS, and JavaScript, and the usual tools do not remove these extra lines when counting. LLOC is the Logical (or effective) Lines of Code, and it is the unbiased method to compare application sizes among projects, avoiding different programming styles.[13] But the main reason for using LLOC is that it avoids counting non-PHP code in PHP files because *phpLOC* counts the lines outside PHP tags for the LOC (for example, HTML, CSS or JavaScript).

$$CSdensity = \frac{CSabsolutenumber}{kLLOC} \quad (2)$$

where kLLOC is the *logical or effective lines of code or statements only*, not counting third-party folders code.

We present this evolution quantitatively (graphically, with bar charts) and qualitatively, in a table, within three columns, the evolution of the absolute number of CS, the evolution of effective (logical) lines of code, and the evolution of code smell density. All quantitative information is available on the replication package.

Probable causes: Instead of just presenting the CS evolution, we try to find probable causes for this evolution. We suspect that the main common reasons for the behavior in the evolution of CS smells are the evolution of the size of the application and the evolution of the team size. Therefore, we will investigate these probable causes. To quantify this association, we employ the *standard R correlation*. We also graphically show the highest relations, with each application's correlation number.

We measure the team size using the following method: using *git*, we clone all the apps from *GitHub*, and we count the number of different users between 2 consecutive releases with the following operation:

```
git shortlog –sne HEAD
  —after=<date_release_n>
  —before=<date_release_n_plus_1>
```

This command makes the data aggregated by app public release to be able to compare/correlate it with the CS evolution, which has the same date intervals.

Apart from the app and team size metrics, we want to assess if other metrics of the team and commits would affect the density of the smells. Therefore, apart from the *team size*, we also measured the *number of commits*, and calculated the *commits per dev*, *commits per day*, and *new devs* that make commits in a given release as the first commit. The reason to measure the value "*new devs*" is the possibility that when you have a peak in the number of new developers, without knowing the rules of development in the specific app and of CS in general, the number of CS can increase. We also want to understand if a rise in the number of *"commits per dev"* (if one dev is working too much and thus making code with excess CS) and "commits per day" (if one application is going through a peak in development that could lead to bad code) influence the CS evolution.

We took the commits and stored them in a database to count the other variables, for example, the *"new devs"* (we used the dev email first appearance in the git issues). Later we export them in time series (to CSV) to further analysis, with all of the values aggregated by public release, to make their time series comparable with the CS time series.

We perform the correlations between the CS density time series and each of the referred metrics, both numerically (with the *standard R correlation*) and graphically with the correlation value (in this case, after smoothing the line 10 times). In the standard R correlation *cor* we used the parametric test "Pearson" correlation and to get the p-values we used *cor.test*.

3.6.2. RQ2 — What is the distribution and survival/lifespan of CS?

The variables of interest in the study are *CS survival time* (timespan) - in days, and CS absolute and relative *distribution* (both by code smell and app). We also want to know the *percentage of removal of CS*. To get the *CS survival time*, we calculate the *median of the survival time*. However, if the CS removal does not reach 50%, we cannot calculate the median, and we show the **restricted mean** only for comparison. The median is a more

9 https://cran.r-project.org/web/packages/survival/

10 https://cran.r-project.org/web/packages/survminer/

11 http://www.sthda.com/english/wiki/survival-analysis-basics

12 Thousand's logical lines of code.

13 https://mattstauffer.com/blog/how-to-count-the-number-of-lines-of-code-in-a-php-project

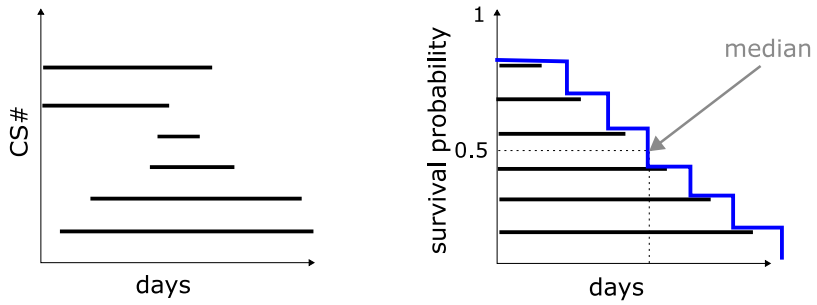


Fig. 2. Survival curves example. Left: 6 CS in survival format; Right: the same 6 smells in survival curves/Kaplan–Meier plots, with ordered CS. The median divides the ordered CS in half and the value represents also the probability of 50% survival (Y axis). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

descriptive value because it is not affected by outliers and is used in other areas, like medical and financial.

First, we used the application developed by the first author to transform the CS in unique instances with an initial and final date, as described in the data preparation phase. This application transforms the CS by release into a CS evolution format suitable for survival analysis, where each line represents a CS, with the date and release number that is introduced and the date and release that is removed (if removed), along with other info (for example, file, type). We show these lines graphically (two selected applications in the study, all apps in the replication package), where each line represents a CS. As described before, we censored data values at the end of the study, "1" if the CS was removed (the death of the CS) and "0" if the code smell continues to exist.[14] For this RQ, we combined the CS in a CSV file with all the CS to use survival analysis per attribute (i.e., per CS or app).

Next, we performed survival analysis by code smell. We use the non-parametric method Kaplan–Meier survival time estimate (Kaplan and Meier, 1958) to achieve this. Next, we calculate and present various averages per code smell (distribution, intensity by kLLOC), median, restricted mean (not used, just shown to compare with other values when there is no median), and percentage of CS removal. Finally, we show the survival curves plot, the values per CS in a table, and averages in violin plots. Continuing, we calculate each app's CS survival time (lifespan) values and *all apps*. We also show the plots of survival curves for the apps and the tables with values per app. Finally, we calculated the median with all code smells and the average lifespan in all apps, in which we calculated weighted averages (because of the different size and longevity of apps) and compared it with the simple average (to avoid larger applications skewing).

To perform the survival analysis, we use the *Surv* function to return a *surv_object* (with computed CS lifespan and censoring), then we used the *survfit* function on the *surv_object*, by app, by CS, and including all CS. To get the values, we extracted the summary table for each fit *(summary(fit)$table)*. This summary gives the *CS found and removed* and the *median and restricted mean survival life*, among other values. To make the plots, we used the function *gsurvplot* with the option *pval=true*. To extract the p-values numerically, we used the function *surv_pvalue*.

Fig. 2 shows how to calculate the probabilities and median using survival curves (also called Kaplan–Meier plots). On the left are 6 example lines representing CS with various timespans and initial dates (in days). On the right are the same lines ordered by timespan. This way, the *Y*-axis is the survival probability, and the *X*-axis remains the days, beginning with 0. The blue line is the survival curve. The median divides the smells into halves and is the value when the probability of survival is 0.5 (or 50%). Getting the value of the curve in the *X*-axis when the probability is 0.5 gives the median survival of the CS in days.

14 https://www.rdocumentation.org/packages/survival/topics/Surv

3.6.3. RQ3 — Is the survival of localized CS the same as scattered CS?

We wanted to analyze if the survival of localized smells (in the same file, class, or method) is the same as scattered smells (also called design smells, i.e., smells comprehending multiple classes or files). To answer this question, we formulate the following null hypothesis:

$H_0$1: Survival does not depend on the code smells scope

Earlier, we defined the two scopes of CS that we want to investigate, localized and scattered CS. PHPMD[15] collects three scattered CS, so we chose the same number of localized ones, using a subset of 6 code smells. The first 3 types[16] are localized ones, i.e., they lie inside a class or file, and the last three[17] are scattered ones, because they spread across several classes.

We fit and plotted the Kaplan–Meier survival curves and performed the log-rank test to compute the *p*-value (statistical significance). A *p*-value less than 0.05 means that the survival curves differ between CS types with a 95% confidence level.

Next, we compute the CS survival time to get the higher of the two scopes, using the same methods and functions as the question before (Kaplan–Meier analysis), but making the *survfit* by the scope of CS *(survfit(surv_object scope, data = dat)*. The variables of interest for the survival time (time in days a CS survives) are the median (and the restricted mean — not used). Also to consider are the number of CS found, CS removed number, and the percentage of CS removal for the two scopes (all but the percentage are given by the summary statistics of the function).

3.6.4. RQ4 — Does the survival of CS vary over time?

We divide each application into two consecutive equal time frames to assess if CS survival time is the same across these time frames. We now pose the following null hypothesis:

$H_0$2: Survival of a given code smell does not change over time

If the survival curves are not different, their survival should be the same around the application's life (no change); if they are different, we will measure the survival variables of interest (survival time median, restricted mean, and the number of introduced and removed CS in both periods).

To test the hypothesis, we also used the log-rank test and created a co-variate *timeframe*, with two values '1' and '2', '1' for the first half of the collection period, and '2' for the second half. In other words, the variable time frame will have the value '1' in the CS introduced in the first half and the value '2' in the CS introduced in the second half. For the first period, we truncated the study's variables as if they were in a sub-study ending in this period. Therefore, at the end of the first period, we filled the value

15 https://phpmd.org

16 *ExcessiveMethodLength* (aka *Long method*), *ExcessiveClassLength* (aka *God Class*), *ExcessiveParameterList* (aka *Long Parameter List*).

17 *DepthOfInheritance*, *CouplingBetweenObjects*, *NumberOfChildren*.

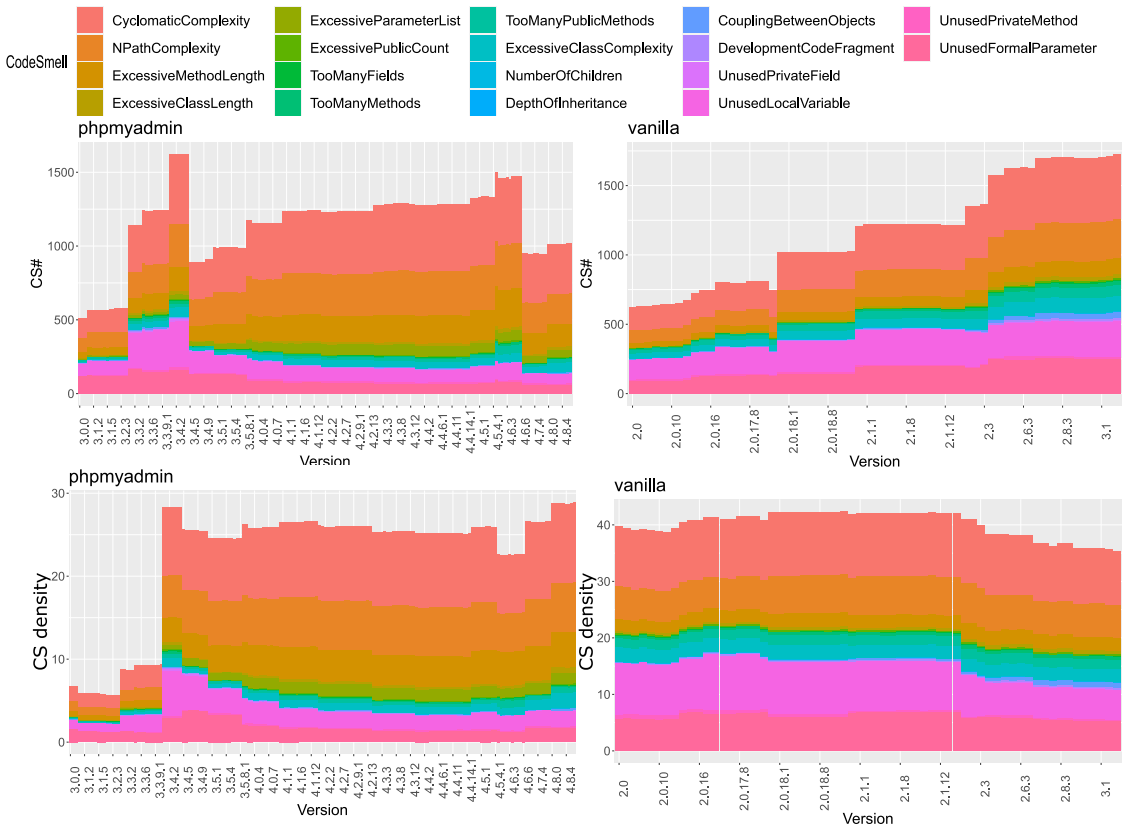


Fig. 3. Evolution of the absolute number and density of 18 code smells, for 2 applications.

of the censored column (with values described in the "statistics used" section). By doing this, we created two independent time frames for each application to analyze the code smells survival. If the *p*-value is less than 0.05, the CS survival differs between the two time frames.

After, we perform the Kaplan–Meier analyses to measure the median and calculate the CS introduced and removed during the periods. We used the *survfit* by time (the two timeframes) of CS *(survfit(surv_object time, data = dat))*. The values are given by the summary statistics table of this function *survfit* (CS found and removed, median and restricted mean survival life).

3.6.5. RQ5 — How to detect anomalous situations in CS evolution?

When we performed the first part of the study (the CS evolution), we observed releases in which there was a refactoring on file names and location in folders (see Fig. 6), but the smells prevailed, as shown by the histograms (see Fig. 3). Our algorithm considered them new because they are in a different file/folder. Our investigation question is: how do we check those releases for anomalies in the number of CS and quantify them? In other words, how to check for peaks in the evolution of CS?

The *anomalies* occur when there is a significant increase (or decrease) in the number of CS, and the size does not grow (or reduce) accordingly. We can divide the #CS by the size for a direct way to spot anomalies. For the size measure, we can use the "Lines of Code" or "Number of Classes" (Henderson-Sellers, 1995). However, if we use the number of classes, we could misrepresent the size of the programs with big classes (a CS itself). So a more accurate measure of the size is the number of lines. However, in a PHP file, it is possible to have other code than PHP (HTML. CSS, JavaScript), and the PHP code is enclosed in tags (<?php and ?>), so the PHP interpreter processes it on the web server. However, programs like PHPLOC or similar count all the lines on those files, even outside the PHP tags. Consequently, a better indicator of lines of code would be the LLOC, logical (effective) lines of code (the statement lines).

Therefore, we use CS density or ρcs= number of CS/LLOC (logical lines of code). We can calculate the rate of change of the CS density:

$$\Delta\rho cs = \frac{\rho cs_i - \rho cs_{i-1}}{\rho cs_{i-1}} = \frac{\rho cs_i}{\rho cs_{i-1}} - 1 \tag{3}$$

where $\Delta\rho$cs is the rate of change of density of CS, ρ cs_i is the density of CS in the current release and ρ cs_{i-1} is the density of CS in the previous release.

The anomalies in the evolution of CS can also be defined as sudden variations in CS density. We made automatisms via scripts to detect these outliers or anomalies in the CS evolution numerically. However, we also present the graphical evolution of CS density for each application, which makes it easy to pinpoint the anomalies or outliers that we show with a label with the release number for easier visualization. By inspecting each anomaly (peak) in the evolution of the application code and CS, we explain what happened in that release.

4. Results and data analysis

4.1. RQ1 — Evolution of code smells

We analyzed the *evolution* of CS in all 12 apps, both in absolute number and in CS density (code smells by kLLOC).

Fig. 3 shows the stack bar charts for 2 of the applications studied, with the 18 CS stacked in the bars, each release represented by a vertical bar. The complete charts for all the applications

Table 3
Qualitative evolution trends of absolute CS (CS number), size (LLOC), CS density (CS/LLOC).

App	Absolute CS	Size (LLOC)	CS density (CS/LLOC)
phpMyAdmin	inc/dec/inc/dec	inc/sharp dec/inc/dec	inc/stable
DokuWiki	increases/decreases	increases/decreases	decreases
OpenCart	increases	increases	stable
phpBB	stable(short dur)/inc	dec(short duration)/inc	inc/stable/dec
phpPgAdmin	increases	increases	dec/stable
MediaWiki	increases	increases	mainly stable
PrestaShop	dec/inc(alm. stable)	dec/inc(alm. stable)	almost stable(inc/dec)
Vanilla	increases	increases	almost stable/dec in the end
Dolibarr	increases	increases	almost stable(small inc)
Roundcube	stable(low inc)	stable	stable
OpenEMR	inc(jump in the end)	dec(alm. stable)/increases	small inc/small dec/small inc
Kanboard	increases	increases	decr/inc - U shape

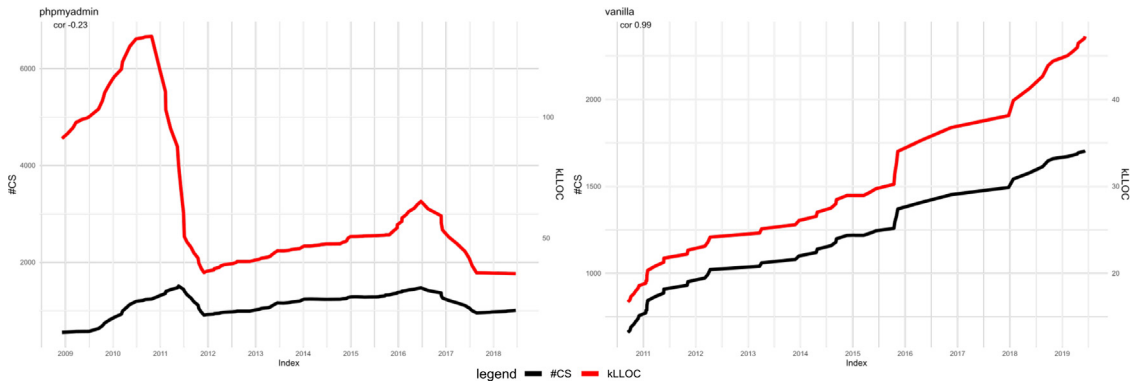


Fig. 4. Evolution and correlation of CS and app size (LLOC), for 2 applications.

are in the replication package.[18] In addition, we analyzed the absolute evolution of CS and the density for all the applications.

For the first application in Fig. 3, *phpMyAdmin*, CS density increases in a first period up to a peak, and after is more or less stable, showing some reductions along the way. As we see later, this peak, in the beginning, is probably related to an increase in the team size. The absolute number of CS for this application ranges from 500 to 1500 (remembering we excluded third-party folders). The density varies from 5 to almost 30 CS per kLLOC.

The second example is an application in which the evolution of the density of CS is stable (related to application size), and in the end period, this density even decreases. The CS absolute value ranges from 500 to around 1700 CS, and the CS density (CS per kLLOC) ranges from 40 to 35 (decreases).

Table 3 shows the qualitative evolution of CS, for all the applications. *inc* and *dec* are abbreviations to *increases* and *decreases*, while *alm. stable* is an abbreviation for *almost stable*. *Short dur* is an abbreviation to *short duration*. The most common trend in the evolution of the total absolute number of CS is the steady increase of the code smells (denoted by the word "increases/inc" in the Table 3). This trend is very similar to the evolution of the application size (second column).

A significant exception in the evolution of lines of Code is the *phpMyAdmin* application because, at some point (release 3.4.0), the developers stopped using PHP files for the different translations and moved to a mechanism similar to UNIX (LC_messages). After this release, the *Lines of Code* decreases.

To understand the absolute CS numbers evolution, we must observe the increase or decrease of the application size (shown in column LLOC). Thus, the most significant evolution is the density of the CS by size (the number of smells divided by size/LLOC), shown in column CS density.

The most common trend behavior in CS density is the stability (with some oscillations) through the evolution of the application, having some applications as exceptions.

> The absolute number of code smells increases according to the application size. In web apps, the evolution trend of server-side code smell density is mainly stable (with oscillations).

4.1.1. Probable causes for the CS evolution

This section shows the correlation with metrics time series that can cause CS evolution trends. Fig. 4 shows the evolution in a graphical way for the two example apps allowing us to inspect the correlation between CS number and app size. We have the remaining graphics in the replication package for the rest of the applications.

We made the correlations after smoothing the lines ten times to avoid oscillations in the lines. As we have seen before, the left application has a jump in the application size evolution (release 3.4). As a result, the correlation between the CS number and size is weak (0.23) but exists. On the other hand, the typical behavior is shown by an application on the right that exhibits a very strong correlation (0.99) between CS and size (kLLOC).

Table 4 shows the average values of the developers' team metrics and code metrics for each app: *CS density* is the average density of code smells (by kLLOC); *releases* is the number of releases in total per app; *release age* is the average time in days between releases (also called the *release frequency*); *num commits* is the average number of commits between releases; *num devs* is the average number of users between releases; *commits per dev* is the average commits per developer between releases and *commits per day* is the average commits by day between release — we used the number of measured days between releases which varies a lot; finally, *new devs* is the average number of new devs between release, i.e., the number of devs that did not commit to the app before.

18 https://github.com/studydatacs/servercs

Table 4
Average metrics by app.

App	CS density	Releases	Release age	Num commits	Num devs	Commits per dev	Commits per day	New devs
phpMyAdmin	22.4	179	22.8	597.0	28.0	20.5	31.7	9.0
DokuWiki	21.1	40	135.5	250.3	40.6	5.1	2.1	20.3
OpenCart	29.0	26	98.5	321.2	23.5	10.2	5.5	12.2
phpBB	33.7	50	120.1	586.2	16.4	35.2	3.8	5.5
phpPgAdmin	35.0	29	224.5	78.0	5.7	16.8	0.7	2.0
MediaWiki	42.2	138	42.4	673.3	39.6	16.6	15.1	5.1
PrestaShop	37.3	74	41.4	778.1	36.8	22.0	19.1	11.2
Vanilla	40.1	63	54.6	433.8	12.0	27.2	9.0	2.6
Dolibarr	23.6	83	61.0	866.9	24.8	59.5	17.6	5.0
Roundcube	51.0	31	67.6	138.9	9.5	15.0	2.4	4.7
OpenEMR	31.3	33	165.0	233.6	18.5	18.6	1.6	6.8
Kanboard	7.6	65	34.6	61.9	10.8	6.3	2.4	5.4

Table 5
Correlations between code smells density and the column metric — in bold if greater than 0.3.

App	Numdevs	Numcommits	Commitsper dev	Commitsper day	Newdevs
phpMyAdmin	**0.59**	−0.10	−0.42	−0.077	0.29
DokuWiki	**0.60**	**0.59**	**0.64**	0.022	**0.44**
OpenCart	**0.50**	**0.67**	−0.36	**0.83**	**0.59**
phpBB	**0.79**	**0.78**	0.20	**0.77**	**0.77**
phpPgAdmin	−0.34	**0.75**	**0.75**	**0.84**	−0.38
MediaWiki	−0.35	−0.17	0.26	−0.36	−0.40
PrestaShop	**0.48**	0.06	−0.28	0.0027	**0.30**
Vanilla	−0.51	−0.38	−0.15	−0.85	**0.36**
Dolibarr	**0.94**	−0.56	−0.95	**0.81**	**0.87**
Roundcube	**0.68**	**0.89**	**0.72**	**0.89**	0.14
OpenEMR	**0.74**	−0.85	−0.92	−0.11	0.24
Kanboard	−0.22	**0.33**	**0.62**	−0.06	0.037

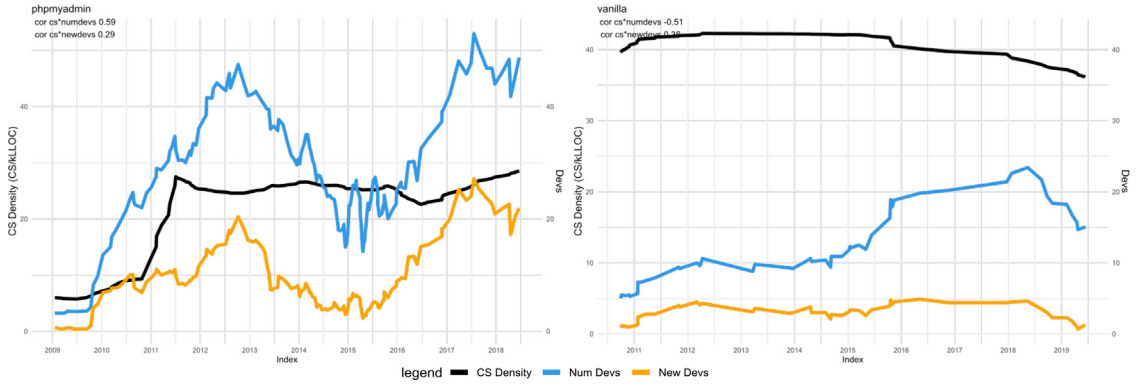


Fig. 5. Correlation and evolution of CS density (CS/kLLOC) and team size, for 2 applications.

The table with average metrics is shown to compare and check if there are outliers in the average values. The complete time series of the metrics in the Table 4 for the 12 web apps are in the replication package. The CS density varies from 20 to 50 CS/kLLOC, except for *Kanboard*, which has a minimal CS density. The two outliers in the average number of commits between releases are *Kanboard* and *phpPgAdmin* with very low commits per release. The same two applications have the lowest team size and the new dev numbers. The *number of devs* ranges from 10 to 40, being *phpPgAdmin* clearly an outlier with just 5.7 in average. The average number of *new devs* ranges from 5 to 20, if not counting the outlier. An important metric is the *average commits per dev between releases*, which ranges from 5 to 60.

We measure the correlations between the code smells density time series and the time series for each of the variables referred before for each application. Table 5 shows the correlation values. Each column represents the correlation with the metric of the column's name. The positive time series correlations greater than 0.3 and with a p-value of less than or equal to 0.05 are in bold. We can observe that the density of code smells correlates with the developers (column *num devs*) in a given release, except for four apps — ranging from 0.47 to 0.94). This correlation is also strong with the "*new devs*", except for two apps with negative correlation and one with almost no correlation. We observe that two of the apps (*phpPgAdmin* and *Kanboard*) are much smaller than the others, and also small is the number of developers in those apps (Table 4).

Fig. 5 presents the evolution and correlation between CS density, team size, and CS density in a graphical way for the two selected apps. We can find the graphics for the rest of the applications in the replication package. For the application on the left, *phpMyAdmin*, the correlation between CS density and *"num devs"* is 0.59, while the correlation between CS density and *"new devs"* is 0.29. For the application on the right, *Vanilla*, the correlation between CS density and *"num devs"* is 0.51, while the correlation

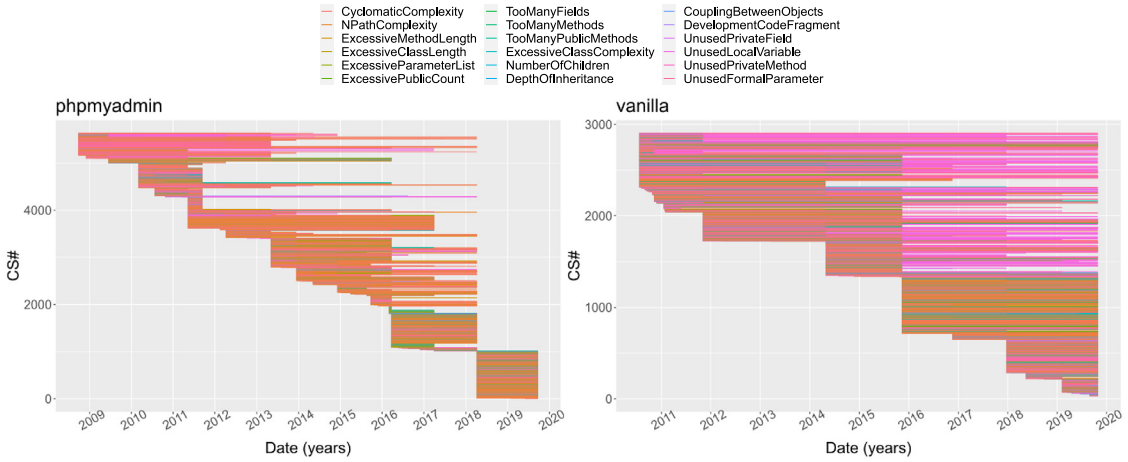


Fig. 6. Life of unique CS in 2 of the apps.

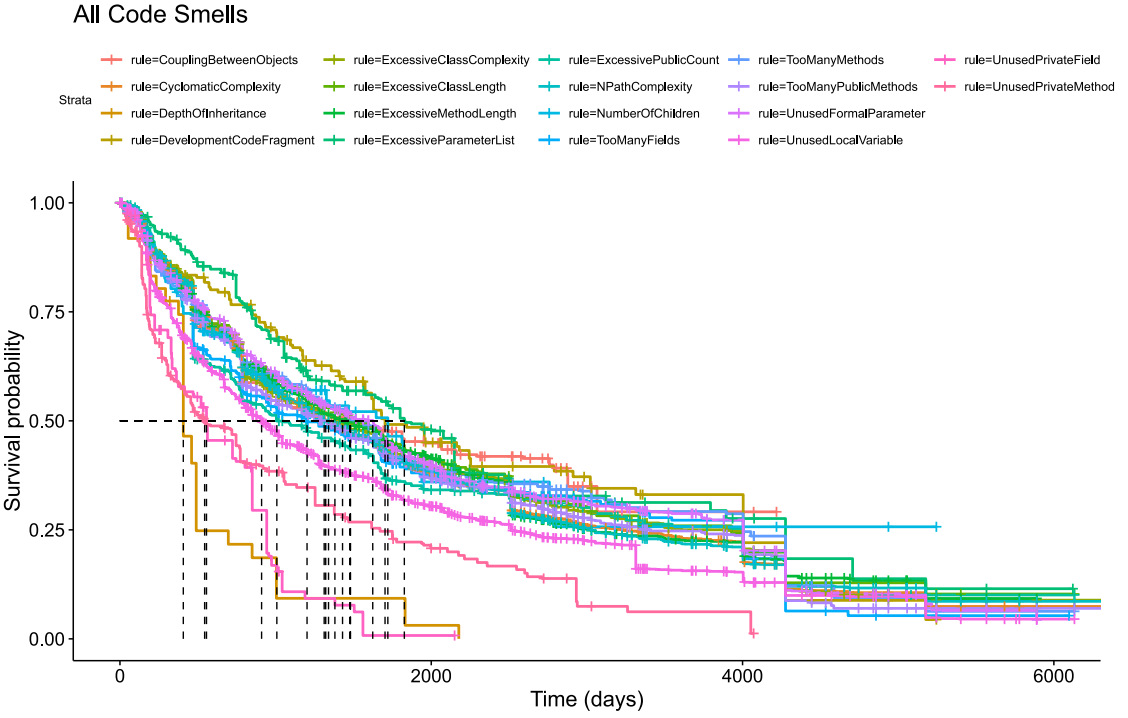


Fig. 7. Survival curves for 18 code smells. Dashed lines denote the median.

between CS density and *"newdevs"* is 0.36. Those are not the highest correlations, as we can see in the Table 5.

> The evolution of the absolute number of code smells correlated to the *LLOC* or *LOC*. Likewise, the evolution of the density of code smells correlated to *number of devs* and *number of new devs* in the release.

4.2. RQ2 — PHP code smells distribution and lifespan

This section shows the results of the code smells' survival time (lifespan) graphically and numerically. We also show the results of other variables of interest, like CS distribution and intensity.

4.2.1. CS lifespan

Fig. 6 shows the unique CS for two selected applications. The horizontal lines represent the lifespan of the code smells. Each line represents one CS from its appearance on the application code until its removal. The colors represent the different CS (keyed in the legend), but this graph is more useful to observe if we look for the overall status of the CS evolution. In the application on the left, we can see a large removal of code smells, denoted when a large number of lines end at the same time (2011, 2016, 2017, 2018), while in the application on the right, this removal happens to a much lesser degree (but still happens in 2015 and 2018).

4.2.2. Values by CS

Fig. 7 shows the survival curves or Kaplan–Meier plots for the 18 CS. With the curves, it is possible to calculate the survival probability at various times by crossing the X-axis with the Y-axis in the line. The medians (probability of survival = 0.5) for each code smell are shown in dashed lines. Next, we show the values in a table.

Table 6 represents several indicators: *CS Distribution* is the distribution of code smells averaged by CS averaged by app; *CS*

Table 6
Average distribution, density, survival time (median and restricted mean) and % removal of Code Smells (by CS).

Code smell	CS distribution	CS density (by kLLOC)	CS survival time median (days)	CS survival time rmean (days)	%CS removal
(Exc.)CyclomaticComplexity	26.14%	8.17	1292	1967	53.52%
(Exc.)NPathComplexity	16.90%	5.30	1359	1961	51.11%
UnusedLocalVariable	16.88%	6.43	1269	1564	70.57%
UnusedFormalParameter	12.46%	3.54	1735	1941	57.55%
ExcessiveMethodLength	9.60%	2.96	1513	2053	51.08%
ExcessiveClassComplexity	5.22%	1.47	1418	1793	55.66%
TooManyPublicMethods	4.45%	1.09	1431	1823	57.62%
(Exc.)CouplingBetweenObjects	1.22%	0.24	1143	2473	30.99%
ExcessiveClassLength	1.10%	0.34	1300	1873	56.06%
ExcessiveParameterList	1.08%	0.29	1433	2087	52.48%
DevelopmentCodeFragment	0.95%	0.26	1746	1999	59.65%
(Exc.)NumberOfChildren	0.93%	0.09	1128	2317	40.52%
TooManyFields	0.88%	0.27	1178	1978	53.97%
TooManyMethods	0.87%	0.27	1361	1757	59.15%
ExcessivePublicCount	0.72%	0.22	1400	1860	50.30%
UnusedPrivateMethod	0.44%	0.12	500	1185	69.82%
UnusedPrivateField	0.16%	0.05	599	1670	62.40%
(Exc.)DepthOfInheritance	0.01%	0.02	918	882	85.00%

Density is the CS intensity (by kLLOC) by CS averaged by app; *Survival Time Median (days)* is the *median* averaged by application; we also show the *restricted mean by CS*/averaged by application — only for comparison. For the three last columns, we measured the values for all applications separately and then calculated the average to account for the problem of bigger applications with more code smells, which could skew the distribution. Next, we plot the table's top values and present the analysis for both the table and the graphics.

Fig. 8(a) shows the distribution mean, quartiles, and variance of the most seen code smells. Fig. 8(b) represents the same statistics for the density of the most seen code smells. Fig. 8(c) shows the mean, quartiles, and variance for the CS survival time, and Fig. 8(d) shows the same statistics for the removal percentage by app for the most seen smells.

CS distribution: The most prevalent server-side CS in the studied web apps are by *(Excessive)CyclomaticComplexity* with 26.14%, followed by *(Excessive)NPathComplexity* with 16.90%. The other CS that appear more in the code are: *UnusedLocalVariable*: 16.88%; *UnusedFormalParameter*: 12.46%; *ExcessiveMethodLength:* 9.60%; *ExcessiveClassComplexity*: 5.22% and *TooManyPublicMethods* with 4.45%. The others are less than 2%, as shown in the Table 6. The variance is not the same in the those CS, being the *UnusedLocalVariable* the one with most variance (Fig. 8(a)). The *ExcessiveClassComplexity* and TooManyPublicMethods have a small variance, but the last one has a huge outlier in one app that goes to 20% - remembering that these values are averaged per app.

CS density (CS/kLLOC) - We also calculate the average density of the smells per kLLOC. This value is calculated first per app and then averaged. We can have these values as a reference when analyzing a new app. The CS that are more intense by average are the same as before, but the order is slightly different. The values are for the same top 7 in Fig. 8(b): 8.17; 5.30; 6.43; 3.54; 2.96; 1.47 and 1.09. The first tree CS have long tails (greater variance), and the following four are more concentrated in the medium value. It does not make statistical sense to calculate the average for these values, but if we sum all the smells together, the average value in all apps for the density of the code smells is around 31 CS per 1000 logical lines of code (considering the 18 code smells and the 12 web apps studied).

Survival time median: We want to know the *median* of the survival time of the CS. This value ranges from 500 days (*Unused Private Method*) to 1609 (*Unused Formal Parameter*). The code smells with longevity greater than 1400 days on average are *Unused Formal Parameter*, *Excessive Method Length*, *Excessive Class Complexity*, *Too Many Public Methods*. The code smells with the longevity of fewer than 1000 days on average are *Unused Private Method*, *Unused Private Field*, *(Excessive)Depth Of Inheritance*. The tails are long for both sides in the same top 7 CS (Fig. 8(c)), except for *Excessive Class Complexity*, which has a strange distribution on the top caused by outlier apps.

We also calculated the restricted mean (as described earlier). As expected, the restricted mean typically has values higher than the median, but the opposite happens for the CS *DepthofInheritance*.

Percentage of CS removal: The values range from 40% to 70% with two outliers, one in the left (*CouplingBetweenObjects*) at 30% and other on the right (*DepthOfInheritance*) at 85%. The variances for the removal of the CS between apps are quite normal (Fig. 8(d)), as the tails are not too short nor too long.

4.2.3. Values by application

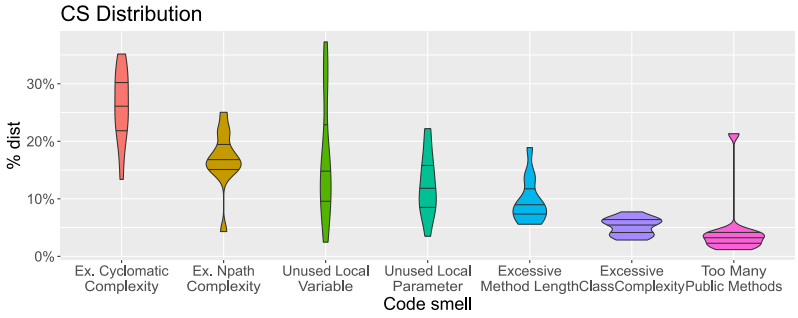
Fig. 9 shows the survival curves or Kaplan–Meier plots for the 12 apps. The medians (probability of survival = 0.5) for each code smell are shown in dashed lines.

Table 7 shows the survival variables of interest by applications and their average. *#CS* is the number of unique CS in the file of the application, *#removed* is the number of removed unique CS, *%removed* is the percentage of unique CS removed. The *CS survival time median (days)* is the same as for the other Table 6 by code smell, but here by application. We show the restricted mean column for comparing when there is no median (column *CS survival time *rmean (days)*), i.e., the average survival counting both the removed smells and the not removed ones, from when the CS first appeared until the end of the study. Because of the high rate of not removed smells, this value is much higher. We show this column because, for one app, it is not possible to calculate the median with the function *survfit* (not sufficient removed smells).

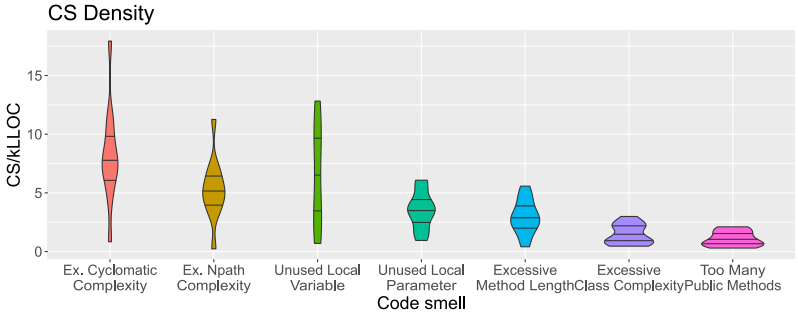
We analyzed in total 67635 unique CS in all the applications. In Table 7, the line "all apps" represents the averages for the apps studied, where only 61% of smells are removed.

Percentage of CS removed: the columns *#CS* and *CS removed* are represented in the table to understand the quantities by absolute and density value. The removal density *percentage of CS removed* explains best what happens. The values of the removal of CS vary to a great extent, from 33% in *Roundcube* to 80% in *phpMyAdmin*.

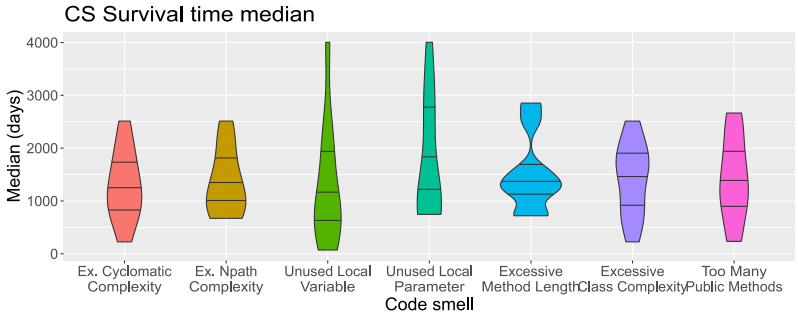
Survival life median: the median is the most useful measure for the CS survival time because it does not count the outliers.



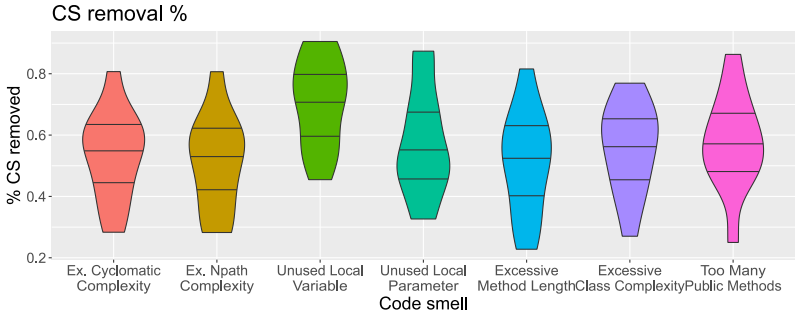
(a) Average CS distribution percentage



(b) Average CS density (CS/kLLOC)



(c) Average CS survival time (days)



(d) Average CS removal %

Fig. 8. Average plot for top 7 CS: Distribution, Density, Survival Time and removal percentage.

The median varies between 231 days (*Kanboard*) and 2699 days (*phpPgAdmin*). All the applications exhibit different CS survival values. We could consider the younger applications outliers (we just remove one application, the other has no median and it gives a median of 4 years) or the average of the median divided by 2 sizes: bigger apps 3.7 years, smaller apps 4 years — not a significant difference by size or age.

We calculated the median of all 67635 CS, which is 1266 days (around 3.5 years); however, if an application contains a high CS density, this will skew this "all apps median" value entirely.

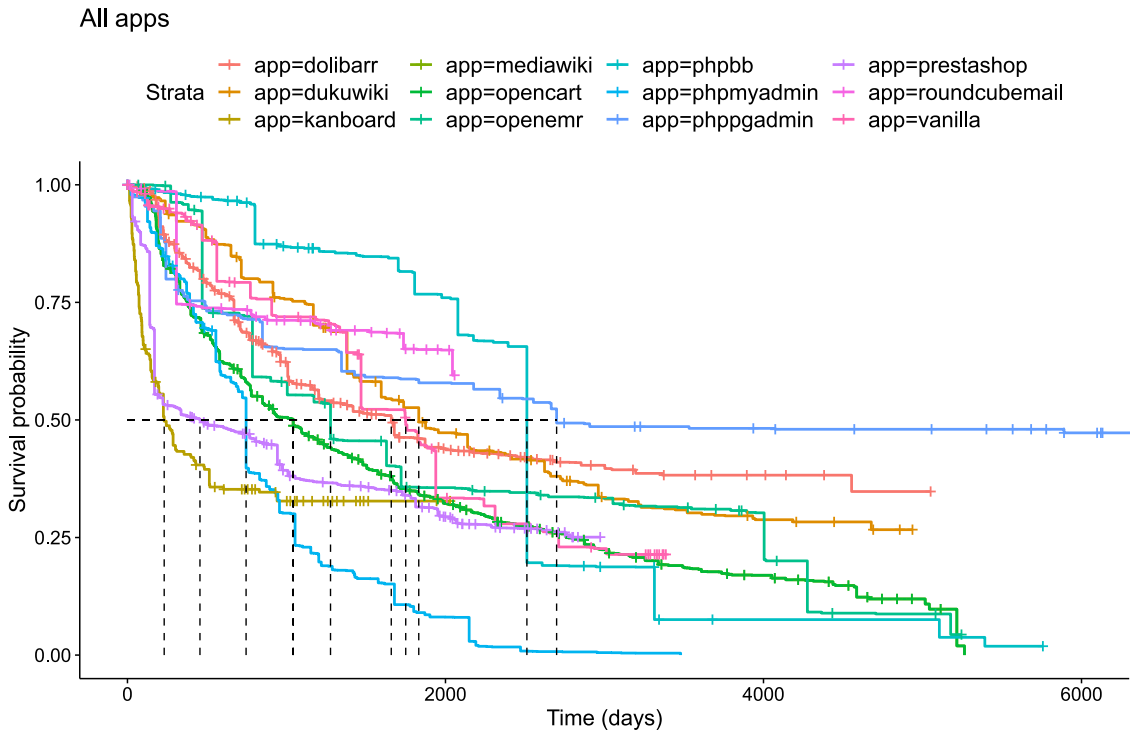


Fig. 9. Survival curves for 12 apps. Dashed lines denote the median.

Table 7
Values of survivability of code smells, by app and all applications.

App	#CS	CS removed	% CS removed	CS survival time median (days)	CS survival time *rmean (days)
phpMyAdmin	5630	4622	82.10%	746	850
DokuWiki	1811	1138	62.84%	1832	2874
OpenCart	9968	6672	66.93%	1042	1748
phpBB	4079	2390	58.59%	2512	2458
phpPgAdmin	740	378	51.08%	2699	3577
MediaWiki	9968	6672	66.93%	1042	1748
PrestaShop	6648	4529	68.13%	456	2031
Vanilla	2902	1437	49.52%	1750	2458
Dolibarr	11939	5710	47.83%	1659	2987
Roundcube	1387	470	33.89%	NA	4138
OpenEMR	12231	7057	57.70%	1277	2113
Kanboard	332	208	62.65%	231	2235
all apps	67635	41283	61.04%	1386	2435

We calculated average values for the apps *median*; the average survival time is 1386 days/3.8 years (simple average) or 1323 days/3.65 years (weighted average with size and app total age). Depending on how we calculate, the survival value for all apps and code smells in our study is between 3.5 and 3.8 years.

We calculated the median weighted by the size and age by normalizing the weights for size and age separately, multiplying the weights and then normalizing to 100%, and then multiplying the final weight to the medians of the apps. We concluded that app age does not influence the weighted average; the size of the app does but not by much; however, the value is close enough to the simple average of the median of all apps, allowing to use the latter as approximation.

Restricted mean: the last column represents the restricted mean, explained before, and considers the CS not removed. We do not use the value, but the column is shown to serve as comparison, when it is not possible to calculate the median.

We also compared the CS lifespan median to the application's studied lifespan, and this value is around 37%. The percentage value gives more information for generalization than the absolute value.

> CS live in average about 37% of the life of the applications. Depending how we calculate the survival time of all CS in all apps is between 3.5 years and 3.8 years in our study. The CS that live more days in the apps studied are: *UnusedFormalParameter, ExcessiveMethodLength, ExcessiveClassComplexity, TooManyPublicMethods*. On average, only 61% of the server code smells in web apps are removed.

4.3. RQ3 — Survival curves for different scopes of CS: localized vs. scattered

In this section, we present the CS survival study results for two scopes of CS: *Localized* vs. *Scattered*, with 3 code smells in each scope (group), as defined in the study design section.

Table 8 presents the Log-Rank test significance for all the apps. A *p*-value less than 0.05 means that the survival curves differ between CS types with a 95% confidence level. For half of the web

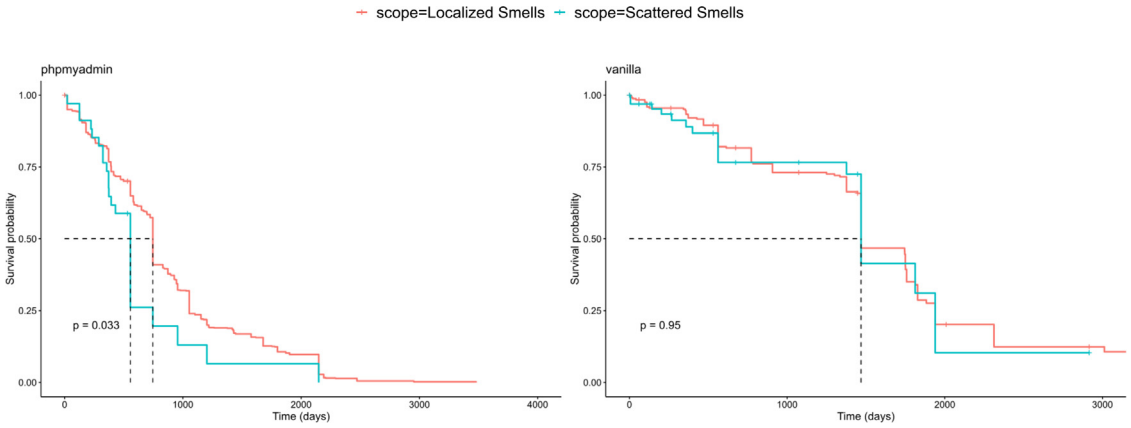


Fig. 10. Survival curves for localized and scattered code smells, for 2 applications. Dashed lines denote the median.

Table 8
Log-rank significance test — comparison of scattered and localized smells : a *p*-value less than 0.05 means different survival curves.

App	p(significance)	App	p(significance)
phpMyAdmin	**0.033**	*PrestaShop*	**0.043**
DokuWiki	0.169	*Vanilla*	0.947
OpenCart	**0.023**	*Dolibarr*	0.589
phpBB	0.098	*Roundcube*	0.89
phpPgAdmin	**0.0002**	*OpenEMR*	**<0.0001**
MediaWiki	**<0.0001**	*Kanboard*	0.73

apps(*phpMyadmin, OpenCart, phpPgAdmin, MediaWiki, PrestaShop, OpenEMR*), the significance is less than 0.05 (in bold), meaning the survival life of the scattered smells differs from the survival life of the local smells. However, this analysis will not suffice because of the low removal rate of code smells, as shown next.

Regarding the non-significant comparisons: in *DokuWiki*, there are four scattered smells, and none is removed (no comparison possible); in *phpBB*, from 20 Scattered CS, only two are removed(10%); in *Dolibarr* and *Roundcube*, also not enough CS are removed; in *Vanilla*, the removal rate of the scattered CS is half of the localized, but the median of survival is the same, probably by coincidence. Finally, in *Kanboard*, the removal rates and survival median are similar, but it is a small application (like *Roundcube*). Summing up, this test is probably inconclusive for the applications with a low removal rate of the CS, both local and scattered.

For the applications with significant values ($p < 0.05$), meaning the survival of CS is different, there is just one inconclusive or false positive, which is *OpenCart*: very few scattered smells (4), but none is removed.

Table 9 presents the numbers and percentages for the CS survival time and CS introduced and removed, separated by scope (localized vs. scattered), given by the column *CS scope*. The column *CS found* and *CS removed* are the code smells found and removed, while the *% removed* is the percentage of the CS removed. The *CS survival time median* is the value in days at which the survival probability is 0.5, and when the removal is small, it is impossible to calculate. Therefore, we also show the *CS survival time restricted mean*, that is, the mean including smells that are not removed at the end of the study and censored with 1 or 0 (for those smells, the ending time is the end of the study).

We can observe that, during the life of the applications, all the applications remove localized code smells, while some do not remove scattered CS (or remove them to a lesser degree). The removal percentage of the localized smells individually per app is normally higher than the removal percentage for the scattered smells. However, for *phpPgAdmin*, and *Kanboard*, small apps compared to the others, the opposite happens. We cannot draw the same conclusion in the application *OpenEMR* because the removal percentages are very close.

The statistics tool R cannot calculate all the medians for the applications (NA in the table) due to some of them having low removal rates, but the average median for this subset of CS (six CS) is around 3.5 years for the three local smells of the question and about 3.2 years for the scattered smells. However, the removal rate of the localized smells is higher 52% against 37% for the scattered ones, as noticed in the previous paragraph. Therefore, survival time values are slightly less for the six CS collections used in this question than for the 18 CS collections of RQ2.

Analyzing the Table 9 numerically for the survival time median and for the apps that have the survival time median for both scopes, we find: *MediaWiki* and *PrestaShop* have shorter survival times for the localized CS, and on the contrary, *phpMyadmin* and *OpemEMR* have shorter survival times for the scattered CS. Tables 8 and 9 need a complementary graphical analysis, because of the CS's low removal rate, especially for the scattered CS.

Fig. 10 presents the curves of the probability of survival of the two scopes of code smells, localized and scattered, for two selected apps. The graph shows the probability of survival(y-axis) vs. time (x-axis - in days). For the left app (*phpMyAdmin*), the median of the scattered CS (556 days) differs clearly from one of the localized CS (746 days), as the *p*-value of 0.033 also shows. However, for the second application (*Vanilla*), the median is coincident (1469 days in both cases), as also observed by the *p*-value of 0.95.

Analyzing the extended plot of graphical curves by scope, on the replication kit, folder RQ3, witch has 12 applications instead of 2 applications (Fig. 10) we find: *DokuWiki*, *OpenCart*, *phpBB*, *phpPgAdmin*, *MediaWiki*, *PrestaShop* all have shorter survival times for the localized CS; *phpMyAdmin*, *phpPgAdmin* and *Openemr* have shorter survival times for the scattered CS. For *Vanilla*, *Kanboard*, *Dolibarr* and *Roundcube* the study remains inconclusive because even graphically, the survival curves are very similar.

Combining the numerical with graphical analyses, we find: that for five applications(*DokuWiki*, *OpenCart*, *phpBB*, *MediaWiki* and *PrestaShop*), localized CS live less than the scattered; for three applications (*phpMyAdmin*, *phpPgAdmin* and *OpenRMR*, the contrary happens, and for four applications (*Vanilla*, *Dolibarr*, *Roundcube* and *Kanboard*), there is no difference between the two

Table 9
Code smells found, removed and survival time in days, by scope.

App	CS scope	CS found	CS removed	% removed	CS survival time median (days)	CS survival time *rmean (days)
phpMyAdmin	Localized	1095	874	80%	746	869
	Scattered	34	23	68%	556	629
DokuWiki	Localized	156	107	69%	1596	2273
	Scattered	4	0	0%	NA	4937
OpenCart	Localized	798	395	49%	1189	1275
	Scattered	12	0	0%	NA	2172
phpBB	Localized	747	395	53%	2512	2257
	Scattered	20	2	10%	NA	2681
phpPgAdmin	Localized	110	32	29%	NA	4634
	Scattered	10	7	70%	1589	1905
MediaWiki	Localized	1004	595	59%	1407	2004
	Scattered	228	69	30%	2877	2877
PrestaShop	Localized	761	470	62%	803	1302
	Scattered	262	124	47%	943	1453
Vanilla	Localized	249	123	49%	1469	1601
	Scattered	67	18	27%	1469	1539
Dolibarr	Localized	1381	539	39%	NA	3000
	Scattered	22	9	41%	NA	2839
Roundcube	Localized	158	52	33%	NA	1511
	Scattered	10	3	30%	NA	1474
OpenEMR	Localized	1210	677	56%	1277	1977
	Scattered	251	150	60%	514	766
Kanboard	Localized	20	10	50%	584	1004
	Scattered	30	18	60%	256	907

Table 10
Log-rank test significance — comparison of code smells in two timeframes.

App	p(significance)	App	p(significance)
phpMyAdmin	**<0.0001**	*PrestaShop*	**<0.0001**
DokuWiki	0.165053	*Vanilla*	**<0.0001**
OpenCart	**<0.0001**	*Dolibarr*	**<0.0001**
phpBB	**<0.0001**	*Roundcube*	0.852129
phpPgAdmin	**<0.0001**	*OpenEMR*	**0**
MediaWiki	**<0.0001**	*Kanboard*	**<0.0001**

scopes. Half of the applications remove 30% or less scattered CS (Table 9).

> For 2/3 of the applications, localized and scattered CS survival is different. For five applications, localized CS live less than the scattered CS; for three applications, the contrary happens. Four applications have no difference in CS survival by scope. All applications remove localized CS, while half of the applications remove 30% or less scattered CS.

4.4. RQ4 — Survival curves for different time frames

In this section, we present the CS survival study comparison for two time-frames representing two halves of each app's evolution, as defined in the study design section. The analysis is made numerically with the median and after graphically.

Table 10 shows the Log-rank test significance for all the apps. For almost all the applications, except *DokuWiki* and *Roundcube*, the CS survival is different in the first half and second half (p-value less than 0.05).

Table 11 contains the values found (CS found and removed including percentage, and survival life median and restricted mean) from the function *survfit grouped by timeframe*. The median of the survival life in days is shorter in the second half for the apps *Dukuwiki* and *phpBB* (CS with a shorter lifespan on timeframe 2); however, for the former, we cannot conclude that survival life is different (p-value on Table 10). For *MediaWiki*, and *phpMyAdmin*, the median in the first half is lower than in the second half (CS in code live fewer days in the first half), although for the second apps, just by a small margin (23 days). For the other applications, we cannot calculate the **median** in the second timeframe because of the low removal rate of the CS, which we show and analyze later. We could use the "restricted mean", but this value includes the censored and the outlier values and gives different values from the survival time (usually higher). Therefore, we will perform the graphical analysis to complete this numerical analysis.

All the applications except one (*OpenEMR*) introduce more smells in the first half of their life. This result is coherent with other studies (Chatzigeorgiou and Manakos, 2014) that found that more smells are introduced in creating the files/classes. Likewise, all the applications except one (the same) remove more smells in absolute number and in percentage (the percentage gives a better measure of density) in the first half of their life. We remember that we treat the time frames as independent (the censoring is also made at the end of the first timeframe). From the values on Table 11, in the first half of the life of the applications 64% CS are removed on average, and in the second half, only 26%.

Fig. 11 shows the survival curves for the two timeframes of two selected applications. The graphical representation of the survival curves for all the web apps is in the replication package (in folder RQ4). Analyzing the complete set of the graphs, for 10 of the 12 apps, the survival curves of the first timeframe differ significantly from the second one, being the exception **RoundCube** and *DokuWiki*, as seen in the Table 10.

We performed the graphical analyses for the ten applications with different survival curves. For *OpenEMR*, the survival curve of timeframe 2 is different and descends much quicker than the one from timeframe 1. The graphical analysis of *phpBB* agrees with the numerical analysis from the Table 11, so for these two apps, the CS survival time is shorter in the second timeframe (we observe a reduction in the CS survival life in the long term). However, for the remaining 8 of the ten apps, the survival curves are different,

Table 11
Code smells found, removed and survival time in days by timeframe.

Web App	Timeframe	CS found	CS removed	% removed	CS survival time median (days)	CS survival time *rmean (days)
phpMyAdmin	1 (<2014-03-26)	3132	2795	89%	723	877
	2 (>= 2014-03-26)	2498	1490	60%	746	646
DokuWiki	1 (<2012-04-03)	1205	667	55%	2139	1670
	2 (>= 2012-04-03)	606	315	52%	1596	1754
OpenCart	1 (<2016-04-18)	2504	1464	58%	882	765
	2 (>= 2016-04-18)	1632	136	8%	NA	1027
phpBB	1 (<2010-02-19)	2096	1651	79%	2512	2362
	2 (>= 2010-02-19)	1983	478	24%	1703	1474
phpPgAdmin	1 (<2010-12-05)	689	371	54%	2665	1965
	2 (>= 2010-12-05)	51	3	6%	NA	3070
MediaWiki	1 (<2011-11-07)	5305	4442	84%	694	1127
	2 (>= 2011-11-07)	4663	1993	43%	1674	1993
PrestaShop	1 (<2015-07-31)	5683	3868	68%	170	666
	2 (>= 2015-07-31)	965	200	21%	NA	1178
Vanilla	1 (<2015-03-10)	1563	851	54%	1469	1223
	2 (>= 2015-03-10)	1339	209	16%	NA	1452
Dolibarr	1 (<2013-01-24)	6179	4376	71%	1008	1287
	2 (>= 2013-01-24)	5760	1251	22%	NA	1972
Roundcube	1 (<2017-01-27)	1176	338	29%	NA	827
	2 (>= 2017-01-27)	211	16	8%	NA	927
OpenEMR	1 (<2012-08-05)	2577	587	23%	NA	2338
	2 (>= 2012-08-05)	9654	4917	51%	786	1048
Kanboard	1 (<2017-01-16)	254	206	81%	146	333
	2 (>= 2017-01-16)	78	2	3%	NA	1034

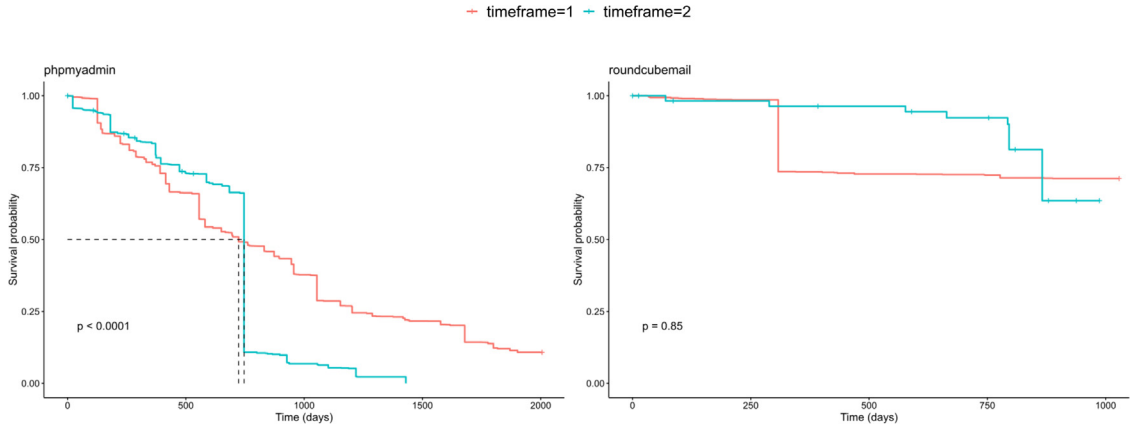


Fig. 11. Survival curves for code smells in the two timeframes, for 2 applications. Dashed lines denote the median.

and the timeframe 1 survival curve descends much quicker to the lower probability values, indicating that CS's survival time in the first timeframe is shorter.

> For almost all the applications, except two, the CS survival is different in the first and second half. For 8 of the ten apps with different CS survival times, the survival time of CS is shorter in the first half of the apps' life, while for two apps, the survival of CS is shorter in the second half. All the applications, excluding one, introduce more CS and remove more CS in absolute number and percentage in the first half of their lives.

4.5. RQ5 — Anomalies in code smells evolution

In this section, we present the anomalies in the code smells evolution study, which occur when there are sudden variations in the CS density.

Fig. 12 represents the relative change of CS number from the previous release and the relative change, in kLLOC, from the previous release. However, as shown in the study design, these absolute values are not enough to spot these anomalies because there can be sudden variations in the number of CS accompanied by the same variation in the size of the app, making no change in density.

Fig. 13 shows the CS density evolution for two select apps, the first with anomalies and the second without anomalies. In the figure, we use lines representing thresholds, signaling an increase of 50% and 100% and a reduction of 50% in the CS density change rate. However, the values of the proposed thresholds can be changed according to application, team, quality, and company, if applicable. The managers or leading developers should choose this.

Table 12 shows the CS density anomalies found, the variance from the previous release to the current release for *CS by kLLOC*, and *Cyclomatic Complexity by LLOC* (aka Cyclomatic Complexity

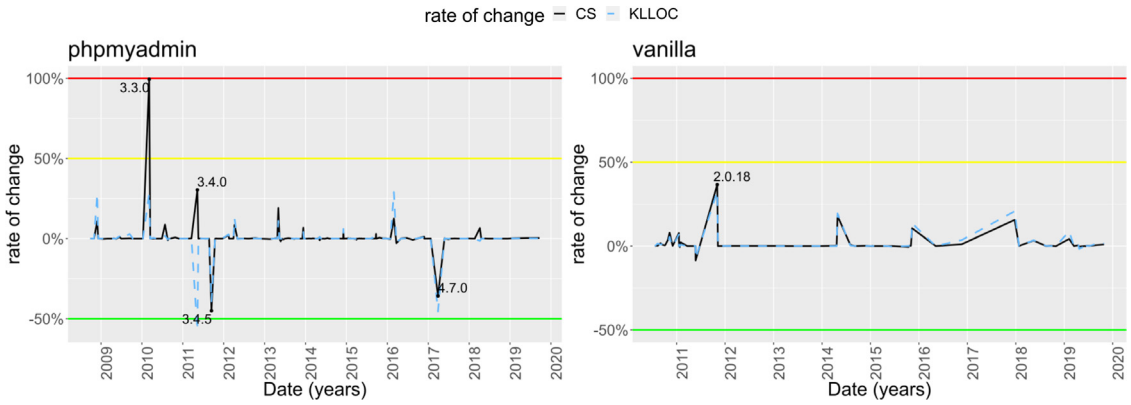


Fig. 12. Changes of cs and kLLOC.

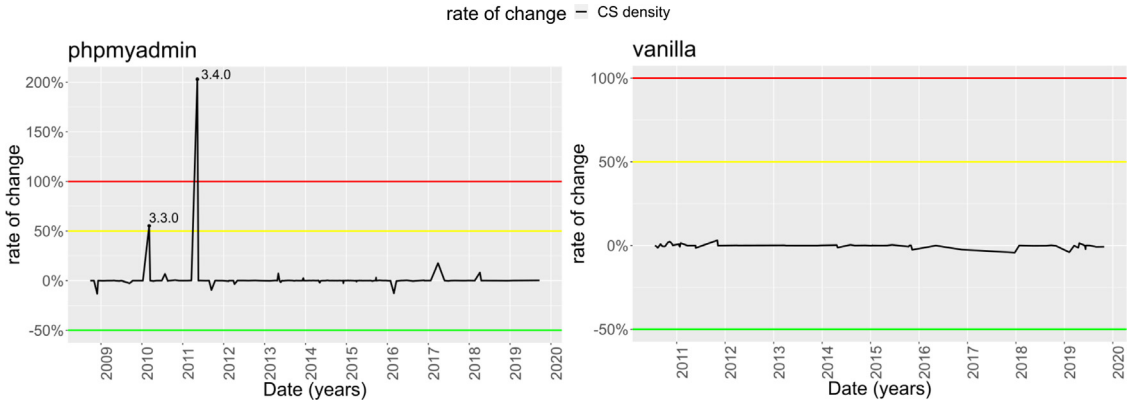


Fig. 13. Anomaly detection in number of code smells: changes in CS density (CS per LLOC.

Table 12
Code smells sudden increases.

App	Release	Date	var CS/LLOC	var CC/LLOC
phpMyAdmin	3.3.0	07/03/2010	55%	36%
phpMyAdmin	3.4.0	11/05/2011	203%	208%
phpBB	2.0.7	13/03/2004	488%	495%
phpBB	3.0.0	12/12/2007	191%	27%
MediaWiki	1.10.0	09/05/2007	95%	1%
OpenEMR	3.2.0	16/02/2010	113%	110%
Kanboard	1.0.4	04/05/2014	78%	0%

Density), a long-used objective metric for maintainability prediction (Gill and Kemerer, 1991). We find those anomalies in 5 applications, but here we represent only the *sudden increases*. For example, *phpMyAdmin* has two anomalies, where the CS rise without the corresponding app size rise. This sudden change can indicate that the code has some problems or big changes in those releases.

We investigated the table's most prominent CS density peaks: In *phpMyAdmin*, release 3.3.0, code increased about 30%, and the number of total code smells doubled, which explains the increase. In this release, there was a refactoring to more classes, the number of classes tripled, but the number of CS follows the LLOC more closely. The peak in release 3.4.0 in *phpMyAdmin* is due to 2 factors: the addition of much new code (49 new classes) with the respective increase of the new smells, and the removal of the "*.php*" files that had the translations (these files had no CS, they only count for the Logical lines of code).

The peak in release 2.0.7 of *phpBB* is due to the removal of the "*.php*" translation language files from the release. This removal makes the denominator (LLOC) decrease and thus provokes an increase in the CS density. The number of classes and number of smells stayed constant. In release 3.0.0 of *phpBB* the peak is because a lot of new code (new functionalities) was introduced. The code size almost tripled, and this release has a lot of new classes (around 160). All these modifications increased the number of code smells eightfold, resulting in that peak that appears in the table.

In the peak in *MediaWiki* 1.10.0, the classes went up 7% and LLOC 5%, but CS doubled in number. This peak was due to refactoring and significant updates, the kind of peak the developers and managers should investigate.

In *OpenEMR* 3.2.0, LLOC went down almost 50%, number of classes went up 10%, and CS all most the same — decreased 1%. This peak was due to refactoring and removing code (for example, in the "interface" folder). In release 5.0.0, the removal of code happens again, but it is not enough to make a peak (a more common behavior, as it is a completely new release, "5.0.0").

Lastly, in *Kanboard* 1.0.4, we also detect a peak. The LLOC went up 35%; classes went up 16%; CS rose 141% because of new code with more CS, which makes up for the peak. This peak resulted from adding some code, classes, and translations.

> We present a method to detect anomalies in CS evolution. Before publishing the release live, this detection method can be put to work in a test automation server. We detected 7 anomalies/sudden changes in the density of CS, in five of the studied applications.

5. Threats to validity

There are four types of threats that can affect the validity of our experiments, and we will see them in detail.

Threats to construct validity concern the statistical relation between the theory and the observation, in our case, the measurements and treatment of the data. We detected the CS using *PHPMD*, and one can question this tool for its accuracy. However, *PHPMD* is used in most of the other tools now (*PHPStorm*, *Codacy* and others), or when they first started (*sonarqube*). Also, because *PHPMD* runs in the command line, our workflow could use it easily. One way to evaluate accuracy is to use 2 or 3 different threshold sets, but we used the default ones because we wanted to compare different applications.

We used only CS related to OOP and only apps with OOP in the app's core (OOP was introduced in PHP4 around 2000). This requirement made our sample construction very difficult because we had to analyze several applications and remove them from the sample. The exclusion of code by external developers both in CS detection and in Lines of code count (we used LLOC to avoid the problem of counting HTML in PHP files – outside the tags – *PHPLOC* does this) was also a difficult task and prone to errors. We had to analyze several app releases to account for errors in folders left out.

The passage of data format from time-series to survival format, with an initial and removal date, is done by a program developed by us. We tested this program for random CS and releases with many CS removed. While we found no errors in the tests, it is worth mentioning.

Threats to internal validity concern external factors we did not consider that could affect the variables and the relations under investigation. Under this concern, *PHPMD* allows changing the CS detection thresholds, but we worked with the default values. We can question these values for different applications. The correlations between CS evolution and the team prove they are related but do not prove causality.

Threats to conclusion validity concern the relation between the treatment and the outcome. In RQ3 and RQ4, we pose two hypotheses in the survival studies, checking for differences between the two groups and answering them with statistical support and significance. In RQ3 not all CS are represented, however because we had 3 scattered CS, we used the same number of localized CS. In RQ4, we divided all the applications into two halves regarding the factor time, giving us different time frames for each application. However, this is the intended design because we wanted to measure survival at the beginning of development history and in the recent history half. Therefore, the tests used would not allow for conclusions with different-sized time-frames.

Threats to external validity concern the generalization of the results found. We chose PHP applications with support for classes not used to build other applications, like frameworks or libraries. Having 12 typical web applications makes the need to have more studies. In the average CS lifespan for all applications, we have apps of different sizes and ages. To tackle this, we also calculated a weighted average lifespan regarding size and age. It would be better to use even more applications for the best generalization. However, because we had to study every release and transform them into unique smells, CS processing and collection require even more computational power. Most evolution studies in the literature (see related work) use a reduced sample because of the high number of releases/versions in each app.

6. Discussion

6.1. RQ1 — CS evolution

For most applications, the absolute number of code smells shows the same tendency as the app size (lines of PHP code, or LLOC as we measured), and the common trend is to increase. The trend for the density of code smells in most applications is stable. For applications that do not have this tendency across part or all story of the application, the CS density correlates highly with the number of *devs* and the number of *new devs* (even more). Therefore, when the size of the PHP web application cannot explain the evolution of CS, we can look into the size of the team and the number of new developers (developers that never contributed in past releases). Possible reasons for this behavior: bigger teams will make more mistakes as it is more challenging to manage them. As for the "new devs", they are probably not aware of the programming practices of the app and do not enforce code smell avoidance.

If we compare this study to evolution studies in Java, Digkas et al. (2017) studied TD (Technical Debt), which is not the same but includes CS, increases for most observed systems, while TD normalized to the size of the system decreases over time in most systems. However, this is not the behavior in the CS in web apps, as most of the CS density is stable over time (with fluctuations).

6.2. RQ2 — CS survivability and distribution

Almost all the code smells are present in all the applications, except for *DepthOfInheritance* in only three apps. The CS that appear the most are related to *complexity*, *unused code*, *long methods*, and *too many public* methods.

On average, the median CS survival life is almost four years (3.8 years, or if weighed by size and age, 3.65 years. We also calculate the median of survival of all CS (3.5 years), but this value can be biased by apps with high density of CS (longer or shorter than the rest). The average of the apps median gives a more informative value.

The PHP web apps in the study remove around 61% of the CS inserted. However, in Java, according to Tufano et al. (2017), the removal rate is 80%. This value is probably because *Java* is the most studied language regarding CS and has more tools to detect and automatically refactor some CS. Another result, in Java applications, from Peters and Zaidman (2012) is that CS lifespan is close to 50% of the lifespan of the systems. Nevertheless, in our findings, PHP web apps' CS survival time median is around 37% of the life of the systems.

The smells that survive more days in the code are *UnusedFormalParameter*, *ExcessiveMethodLength*, *ExcessiveClassComplexity*, *TooManyPublicMethods*. The long life of the "UnusedFormalParameter" can be related to the way that PHP handles default values in methods (*parameter = "default value"*) and can be omitted in the call to the method. The CS that lives the least are the *Unused Private method* and *Unused Private Field*, which is understandable because the removal is relatively straightforward.

The values of removal per app vary to a great extent. For example, *phpMyAdmin* has a removal rate higher than 80% while *Roundcube* only shows 33%. Looking at the results in question 1, we can relate these values to the release average, which in the case of *phpMyAdmin*, is 23 days, the shortest in all the apps studied.

6.3. RQ3 — CS survivability by scope

CS occurrences may be removed by explicit refactoring actions or, much less frequently, due to code dropout. Therefore, PHP project managers need to have an evolutionary perspective on the survival of CS to decide on the allocation of resources to mitigate their technical debt effects. Furthermore, since CS are of different types regarding their scope, project managers must be aware of their evolution, with a particular concern for scattered CS since their spreads may cause more harm and may be harder to refactor without appropriate tooling.

For most applications, localized and scattered CS survival time is different, and scattered CS tend to live longer in the code. All applications remove localized CS, but the rate of removal for scattered CS is much lower.

It is worth noticing that localized CS are much more frequent targets for change than scattered CS for most apps. This behavior may be due to the lack of refactoring tools for PHP scattered code smells, and manually removing scattered CS is much more complex than removing localized ones.

6.4. RQ4 — CS survivability by timeframe

Since the topic of CS has been addressed by researchers, taught at universities, and discussed by practitioners over the last two decades, we want to investigate whether this impacted CS evolution and survival time in web apps. We expected that increased CS awareness had caused a more proactive attitude towards CS detection and removal (through refactoring actions), thus leading to shorter survival rates through the app life. However, there are more factors to consider, for example, the team's knowledge of the code and smells in the first years of the app's life (of which we do not have metrics).

For most apps, the survival curves of the two timeframes differ; for most, the CS survival time is shorter in the first timeframe. Studies in Java language (Peters and Zaidman, 2012), found that smells at the beginning of the systems life are prone to be corrected quickly. For the majority of the web apps, our study agrees with these findings.

The findings mean more work is to be done in PHP web apps to increase knowledge and awareness of code smells and the necessity of refactoring towards its reduction. However, *PHP* web applications will always be more challenging to control regarding the existence of CS due to the heterogeneity of the languages and platforms than Java desktop apps, which are the most studied until now.

6.5. RQ5 — Anomalies in code smells evolution

As we demonstrated, detecting the anomalies in code smell evolution is possible. This relatively simple method to implement – measuring $\Delta\rho$ cs – can detect the anomalies/sudden variations in the CS density.

Almost half (five) of the applications exhibit sudden increases in the density of the CS. These sudden increases indicate various problems in the maintenance capability of the code and can even lead to issues or bugs in the code. It can also show variations in the codebase and the addition or removal of external libraries. If we check instead for sudden decreases, this can usually indicate refactoring or, for example, adding code with no code smells. Therefore, it is essential to remove the external libraries from the analysis.

When is the CS number/density change too high? In factories using control charts, to control a measurement attribute that goes around a value, there are limits equal to 3 times the variance. In this case, we do not have variances around a value/metric, so we have to define thresholds that development teams or managers can tailor. We believe that a threshold of 50% will be sufficient to raise 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 12 we can see that peaks also affect the *cyclomatic complexity per LLOC*, which in turn affects maintainability (Gill and Kemerer, 1991).

We also tried other methods to measure the CS variations, 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 number of CS was the same, then this value sudden increases). The main problem was that the standard deviation was 0 or close to 0. Moreover, with methods that use the average, for example, a metrics study (Digkas et al., 2020), one must know all the project history, while in the method shown here, the computation at each point in time is based on data collected from the previous and current release.

Ideally, the removal of CS can be done in a "Total Quality" manner, where the developer is responsible for avoiding 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 release. If a threshold is reached, the release could be held for some refactoring to be performed. This mechanism 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. Each development team should decide the value of the threshold, depending on the development circumstances and requirements.

6.6. Implications for researchers

There is a need for more studies on the evolution of web apps with code smells to approach the level of knowledge in desktop apps, particularly in the Java language. While some findings in this study are similar to the desktop world (differing in the numbers), others reveal that there is more work to be done on the web area (for example, the density of CS has a stable trend — indicating few CS removal, while in desktop apps the trend is "decrease").

A substantial part of the evolution of CS can be explained by the size of the application, team size, and the number of "new devs". However, in this study, the number of commits does not seem to relate to the evolution of CS. Additionally, there can be sudden changes in the evolution of CS that can be caused by a peak in the referred variables (code size and team) or even other factors that are worth investigating.

When making evolutionary studies, investigators must inspect the software being analyzed to avoid the folders from other vendors when collecting the sample. For example, in *phpMyAdmin*, if we remove the folders from different vendors from the analysis, we have only 50% of the original code in the zip release. If this step is missed, part of the analyzed code comes from other programs. We perform this step since (Rio and Brito e Abreu, 2019).

When measuring size/lines in PHP, not all programs will count only PHP code lines inside PHP files. For example, *phpLOC* will count HTML lines in PHP files as PHP code. One way to avoid this problem is to use LLOC (logical lines of code).

We provide the data used in this study (replication package) that complies with the last two paragraphs and can be used in further studies.

PHP web apps should have more tools to detect and refactor code smells. While there are some tools to refactor localized code smells, there is a need to build tools to refactor scattered (design) code smells in this language.

Lastly, more code smells and quality topics should be taught in the disciplines of *Software Engineering* in the *Academia*.

6.7. Implications for practitioners

Removing code smells in a web app without spending time is impossible. Therefore, there will always be a trade-off between releasing quickly and more quality in the code. However, this

and other studies suggest several effective ways to mitigate the problem.

Reduce CS density: (From RQ1) The density of code smells is not going down in the evolution of the projects (primarily stable trend), as with languages with more tools for detecting and refactoring CS, like Java. Therefore, team leaders should alert the other developers of the existence of CS in the applications built with PHP.

Divide big development teams into groups that are easier to manage. CS density correlates with team size, and bigger teams will make more mistakes as it is more challenging to manage them.

As for the "new devs", they are probably not aware of the programming practices of the app and do not enforce code smell avoidance. Ensure that there is a style manual referring to avoiding CS.

Prioritize CS to remove: (From RQ2) Detect and remove unused variables and parameters. Try to reduce the complexity of the methods and classes. Try to shorten long methods and long classes by dividing them into specialized methods and specialized classes.

Scattered/design CS are less removed than localized CS and should be addressed in the code as soon as possible, as they affect more classes/files and cause more damage. They will affect maintainability later.

From RQ4, we can report that there was no significant reduction in CS survival time in the second half of the application life for all applications. Therefore, team leaders and managers must increase developers' awareness of CS and maintainability problems.

Avoid peaks in code smells, especially sudden increases, before a release. These peaks in the density of CS usually increase the *Cyclomatic Complexity* density, which is a proxy for maintainability. We provided a simple way to perform this sudden increase detection. However, we detected some peaks that were legit removal of code (a different implementation of translations, for example), which is sometimes unavoidable.

Finally, as a general development recommendation, in projects with legacy code, whenever possible, move the code from includes (from other teams) to a vendor folder (the standard for the tool *composer*) or a different folder of choice. Even if the app is not using *composer*, the external code will be more up-gradable (and ease taking metrics).

7. Conclusions and future work

The literature is still scarce on what concerns PHP CS evolution studies, the main topic of this paper. PHP, a language that fully supports object-oriented paradigm (among others), is by far the most used on the server-side for web apps, and a vast codebase exists (e.g., almost a million and a half PHP repositories on GitHub.[19]) Other researchers have confirmed the existence of CS in PHP, and the Software Engineering community has long agreed that, since they are symptoms of poor design, leading to future problems such as reduced maintainability, we should aim at avoiding them. While the best option would be not to insert them during development using detection mechanisms embedded in IDEs, we found evidence of 18 CS in 12 widely used PHP web apps over many years. Therefore, we studied their evolution, survival, and anomalies in the CS evolution.

The trends in most applications are the increase of the total number of CS, similar to LLOC trends, and stability in CS density, with some exceptions. There is a strong correlation between the density of CS with the developers metrics. The CS that appear the most are related to *complexity*, *unused code*, *long methods*, and *too many public* methods. The average CS survival life in the applications is between 3.5 and 3.8 years depending how we calculate. A more important value is that CS tend to stay a long time in code, close to 37% of the app life. Around 61% of the CS are removed from the code. We found that most of the survival of localized code smells differs from the localized ones and from application to application. All applications remove localized CS, but the rate of removal for scattered CS is much lower. For most apps, the survival curves of the app history first half and second half differ, and for most of those, the CS survival time is shorter in the first timeframe.

Last but not least, we described a normalized technique for detecting anomalies in specific releases during the evolution of web apps, allowing us to unveil the CS history of a development project and make managers aware of the need for enforcing regular refactoring practices. Furthermore, this technique can also be helpful in an automation test server, quality test, or release certification, to avoid the excessive CS density before a public release.

In summary, CS stay a long time in code. The removal rate is low and did not change substantially in recent years. An effort should be made to avoid this bad behavior and change the CS density trend to decrease.

Getting rid of all CS is probably an unjustifiable quest since some occurrences may make sense, depending on the technical context. Therefore, if this number is controlled, the web apps and developers will live with a CS number different from zero. We also hope the number of studies on code smells in web apps increases, augmenting the developers' awareness of this bad behavior in the code.

Regarding future work, we would like to increase the number of applications and the number of CS studied, provided more computing power is available since collecting CS across many continuous releases is a computationally heavy task. We collected CS on a million units for each app during our long observation periods. It is also worth researching if longitudinal studies on CS depend on the programming language.

CRediT authorship contribution statement

Américo Rio: Conceptualization, Methodology, Software, Data curation, Statistical analysis, Results, Writing – original draft, Writing – review & editing. **Fernando Brito e Abreu:** Supervision, Conceptualization, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

For replication purposes, we provide the collected dataset, available at https://github.com/studydatacs/servercs and https://doi.org/10.5281/zenodo.7626150.

Acknowledgments

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

19 https://github.com/search?q=language%3Aphp&type=repositories

References

Amanatidis, T., Chatzigeorgiou, A., 2016. Studying the evolution of PHP web applications. Inf. Softw. Technol. 72 (April), 48–67. http://dx.doi.org/10.1016/j.infsof.2015.11.009.

Amanatidis, T., Chatzigeorgiou, A., Ampatzoglou, A., 2017. The relation between technical debt and corrective maintenance in PHP web applications. Inf. Softw. Technol. 90, 70–74. http://dx.doi.org/10.1016/j.infsof.2017.05.004.

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'2020). Vol. 12409 LNCS, Springer, pp. 67–84. http://dx.doi.org/10.1007/978-3-030-59592-0_5.

Bieman, J.M., Kang, B.-K., 1995. Cohesion and reuse in an object-oriented system. SIGSOFT Softw. Eng. Notes 20 (SI), 259–262. http://dx.doi.org/10.1145/223427.211856.

Bryton, S., Brito e Abreu, F., Monteiro, M., 2010. 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). IEEE, pp. 337–342. http://dx.doi.org/10.1109/QUATIC.2010.60.

Chatzigeorgiou, A., Manakos, A., 2010. Investigating the evolution of bad smells in object-oriented code. In: 7th International Conference on the Quality of Information and Communications Technology (QUATIC'2010). pp. 106–115. http://dx.doi.org/10.1109/QUATIC.2010.16.

Chatzigeorgiou, A., Manakos, A., 2014. Investigating the evolution of code smells in object-oriented systems. Innov. Syst. Softw. Eng. 10 (1), 3–18. http://dx.doi.org/10.1007/s11334-013-0205-z.

Clark, T.G., Bradburn, M.J., Love, S.B., Altman, D.G., 2003. Survival analysis part I: basic concepts and first analyses. Br. J. Cancer 89 (2), 232. http://dx.doi.org/10.1038/sj.bjc.6601118.

Cunningham, W., 1992. The WyCash portfolio management system. SIGPLAN OOPS Messenger 4 (2), 29–30. http://dx.doi.org/10.1145/157709.157715.

Digkas, G., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., 2020. On the temporality of introducing code technical debt. In: 13th International Conference on the Quality of Information and Communications Technology (QUATIC'2020). Vol. 1266 CCIS, Springer, pp. 68–82. http://dx.doi.org/10.1007/978-3-030-58793-2_6.

Digkas, G., Lungu, M., Chatzigeorgiou, A., Avgeriou, P., 2017. The evolution of technical debt in the apache ecosystem. In: European Conference on Software Architecture (ECSA'2017). Vol. 10475 LNCS, Springer, pp. 51–66. http://dx.doi.org/10.1007/978-3-319-65831-5_4.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., Figueiredo, E., 2016. A review-based comparative study of bad smell detection tools. In: 20th International Conference on Evaluation and Assessment in Software Engineering (EASE'2016). Vol. 01-03-June, ACM, pp. 1–12. http://dx.doi.org/10.1145/2915970.2915984.

Fontana, F.A., Lenarduzzi, V., Roveda, R., Taibi, D., 2019. Are architectural smells independent from code smells? An empirical study. J. Syst. Softw. 154, 139–156. http://dx.doi.org/10.1016/j.jss.2019.04.066.

Fowler, M., 1999. Refactoring: Improving the Design of Existing Code. Addison-Wesley, p. 464.

Gill, G.K., Kemerer, C.F., 1991. Cyclomatic complexity density and software maintenance productivity. Trans. Softw. Eng. 17 (12), 1284. http://dx.doi.org/10.1109/32.106988.

Habchi, S., Rouvoy, R., Moha, N., 2019. On the survival of android code smells in the wild. In: 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft'2019). IEEE, pp. 87–98. http://dx.doi.org/10.1109/MOBILESoft.2019.00022.

Henderson-Sellers, B., 1995. Object-Oriented Metrics: Measures of Complexity. Prentice-Hall.

Herbold, S., Grabowski, J., Waack, S., 2011. Calculation and optimization of thresholds for sets of software metrics. Empir. Softw. Eng. 16 (6), 812–841. http://dx.doi.org/10.1007/s10664-011-9162-z.

Johannes, D., Khomh, F., Antoniol, G., 2019. A large-scale empirical study of code smells in JavaScript projects. Softw. Qual. J. 27 (3), 1271–1314. http://dx.doi.org/10.1007/s11219-019-09442-9.

Kaplan, E.L., Meier, P., 1958. Nonparametric estimation from incomplete observations. J. Amer. Statist. Assoc. 53 (282), 457–481. http://dx.doi.org/10.1080/01621459.1958.10501452.

Kyriakakis, P., Chatzigeorgiou, A., 2014. Maintenance patterns of large-scale PHP web applications. In: 30th International Conference on Software Maintenance and Evolution (ICSME'2014). IEEE, pp. 381–390. http://dx.doi.org/10.1109/ICSME.2014.60.

Lanza, M., Marinescu, R., 2007. Object-Oriented Metrics in Practice. Springer.

Lehman, M.M., 1996. Laws of software evolution revisited. In: European Workshop on Software Process Technology (EWSPT1996). Vol. LNCS 1149, Springer, pp. 108–124. http://dx.doi.org/10.1007/BFb0017737.

Mccabe, T.J., 1976. A complexity measure. Trans. Softw. Eng. SE-2 (4), 308–320. http://dx.doi.org/10.1109/TSE.1976.233837.

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: 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM'2009). IEEE, pp. 390–400. http://dx.doi.org/10.1109/ESEM.2009.5314231.

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: International Conference on Software Maintenance (ICSM'2010). IEEE, http://dx.doi.org/10.1109/ICSM.2010.5609564.

Palomba, F., Bavota, G., Penta, M.D., Fasano, F., Oliveto, R., Lucia, A.D., 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empir. Softw. Eng. 23 (3), 1188–1221. http://dx.doi.org/10.1007/s10664-017-9535-z.

Pereira dos Reis, J., Brito e Abreu, F., de Figueiredo Carneiro, G., 2017. Code smells detection 2.0: Crowdsmelling and visualization. In: 2017 12th Iberian Conference on Information Systems and Technologies. CISTI, IEEE, pp. 1–4. http://dx.doi.org/10.23919/CISTI.2017.7975961.

Peters, R., Zaidman, A., 2012. Evaluating the lifespan of code smells using software repository mining. In: European Conference on Software Maintenance and Reengineering (CSMR'2012). IEEE, pp. 411–416. http://dx.doi.org/10.1109/CSMR.2012.79.

Rani, A., Chhabra, J.K., 2017. 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, IEEE, pp. 1093–1098. http://dx.doi.org/10.1109/I2CT.2017.8226297.

Rasool, G., Arshad, Z., 2015. A review of code smell mining techniques. J. Softw.: Evol. Process 27 (11), 867–895. http://dx.doi.org/10.1002/smr.1737.

Pereira dos Reis, J., Brito e Abreu, F., de Figueiredo Carneiro, G., Anslow, C., 2021. Code smells detection and visualization: A systematic literature review. Arch. Comput. Methods Eng. http://dx.doi.org/10.1007/s11831-021-09566-x.

Rio, A., Brito e Abreu, F., 2019. Code smells survival analysis in web apps. In: 12th International Conference on the Quality of Information and Communications Technology (QUATIC'2019). Springer, pp. 263–271. http://dx.doi.org/10.1007/978-3-030-29238-6_19.

Rio, A., Brito e Abreu, F., 2021. Detecting sudden variations in web apps code smells' density: A longitudinal study. In: Paiva, A.C.R., Cavalli, A.R., Ventura Martins, P., Pérez-Castillo, R. (Eds.), Proceedings of the International Conference on the Quality of Information and Communications Technology (QUATIC'2021). Springer, pp. 82–96. http://dx.doi.org/10.1007/978-3-030-85347-1_7.

Saboury, A., Musavi, P., Khomh, F., Antoniol, G., 2017. An empirical study of code smells in JavaScript projects. In: 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER'2017). IEEE, pp. 294–305. http://dx.doi.org/10.1109/SANER.2017.7884630.

Schuette, D., 2021. Survival analysis in R for beginners.

Sharma, T., Singh, P., Spinellis, D., 2020. An empirical investigation on the relationship between design and architecture smells. Empir. Softw. Eng. 25 (5), 4020–4068. http://dx.doi.org/10.1007/s10664-020-09847-2.

Singh, S., Kaur, S., 2018. A systematic literature review: Refactoring for disclosing code smells in object oriented software. Ain Shams Eng. J. 9 (4), 2129–2151. http://dx.doi.org/10.1016/j.asej.2017.03.002.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., Poshyvanyk, D., 2016. An empirical investigation into the nature of test smells. In: 31st International Conference on Automated Software Engineering (ASE'2016). IEEE, pp. 4–15. http://dx.doi.org/10.1145/2970276.2970340.

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. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. Vol. 1, IEEE, pp. 403–414. http://dx.doi.org/10.1109/ICSE.2015.59.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Penta, M.D., De Lucia, A., Poshyvanyk, D., 2017. When and why your code starts to smell bad (and whether the smells go away). Trans. Softw. Eng. 43 (11), 1063–1088. http://dx.doi.org/10.1109/TSE.2017.2653105.

Zhang, M., Hall, T., Baddoo, N., 2011. Code bad smells: A review of current knowledge. J. Softw. Maintenance Evol. 23 (3), 179–202. http://dx.doi.org/10.1002/smr.521.