Empirical Software Engineering (2024) 29:133
https://doi.org/10.1007/s10664-024-10478-0

Causal inference of server- and client-side code smells in web apps evolution

Américo Rio[1,2] · **Fernando Brito e Abreu**[1] · **Diana Mendes**[3]

Accepted: 19 March 2024

Abstract

Context 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. Web applications and systems are seldom studied, probably due to the heterogeneity of platforms (server and client-side) and languages, and to study web code smells, we need to consider CS covering that diversity. Furthermore, the literature provides little evidence for the claim that CS are a symptom of poor design, leading to future problems in web apps.
Objective To study the quantitative evolution and inner relationship of CS in web apps on the server- and client-sides, and their impact on maintainability and app time-to-release (TTR).
Method We collected and analyzed 18 server-side, and 12 client-side code smells, aka web smells, from consecutive official releases of 12 PHP typical web apps, i.e., with server- and client-code in the same code base, summing 811 releases. Additionally, we collected metrics, maintenance issues, reported bugs, and release dates. We used several methodologies to devise causality relationships among the considered irregular time series, such as Granger-causality and Information Transfer Entropy(TE) with CS from previous one to four releases (lag 1 to 4).
Results The CS typically evolve the same way inside their group and its possible to analyze them as groups. The CS group trends are: Server, slowly decreasing; Client-side embed, decreasing and JavaScript,increasing. Studying the relationship between CS groups we found that the "lack of code quality", measured with CS density proxies, propagates from client code to server code and JavaScript in half of the applications. We found causality relationships between CS and issues. We also found causality from CS groups to bugs in Lag 1, decreasing in the subsequent lags. The values are 15% (lag1), 10% (lag2), and then decrease. The group of client-side embed CS still impacts up to 3 releases before. In group analysis, server-side

Communicated by: Fabio Palomba

✉ Américo Rio
jaasr@iscte-iul.pt; americo.rio@novaims.unl.pt

Fernando Brito e Abreu
fba@iscte-iul.pt

Diana Mendes
diana.mendes@iscte-iul.pt

[1] Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR, Lisboa, Portugal

[2] NOVAIMS, UNL, Campus de Campolide, Lisboa, Portugal

[3] Instituto Universitário de Lisboa (ISCTE-IUL), BRU, Lisboa, Portugal

Published online: 05 August 2024

CS and JavaScript contribute more to bugs. There are causality relationships from individual CS to TTR on lag 1, decreasing on lag 2, and from all CS groups to TTR in lag1, decreasing in the other lags, except for client CS.

Conclusions There is statistical inference between CS groups. There is also evidence of statistical inference from the CS to web applications' issues, bugs, and TTR. Client and server-side CS contribute globally to the quality of web applications, this contribution is low, but significant. Depending on the outcome variable (issues, bugs, time-to-release), the contribution quantity from CS is between 10% and 20%.

Keywords Web apps · Code smells · Software evolution · PHP · Granger causality · Transfer entropy

1 Introduction and Motivation

In the last three decades, web applications (web apps for short) have evolved from simple and almost static apps to fully-fledged ones (Dwivedi et al. 2011), almost rivaling their desktop counterparts, with the most notable advantages for the users, the absence of installation or need to update. However, with this "always-on" and "connected" perspective comes the imperious need for quality and rapid maintenance capability, primarily for corrective actions (Vern and Dubey 2014; Ricca and Tonella 2003).

One of the prominent areas of study in software quality improvement is code smells (CS). CS are symptoms of poor design and implementation choices, therefore fostering problems like increased defect incidence, insufficient code comprehension, and longer times to release, as reported in studies involving desktop apps (Palomba et al. 2017). Most of these studies are cross-sectional, but there are also some longitudinal/evolution ones. Software Evolution is an active research thread in Software Engineering, where longitudinal studies have been conducted on software products or processes, focusing on aspects such as software metrics, teams' activity, defects identification and correction, or time to release (Herraiz et al. 2013; Radjenović et al. 2013). However, most of the studies target desktop apps.

PHP is the most used server-side programming language in web development [1], making almost 80% of the web apps built. The research results on CS within the server-side code only (e.g., in Rio and e Abreu 2019; Bessghaier et al. 2020) are very similar to the ones reported for desktop apps in the literature (Zhang 2010; dos Reis et al. 2021). However, web apps are not built with only the server-side languages since another part runs in the browser. While a single language is usually used on the server-side (e.g., PHP, C#, Ruby, Python, or Java), several languages are used to build the client-side (e.g., JavaScript for the programmatic part, HTML for content, and CSS for formatting).

We study PHP web applications because they are the most frequently used - as previous referred. Secondly, because typical PHP web applications have the server-side and client-side code in the same code-base (monolithic web apps), makes it possible to study the client and server-side code relationship. Lastly, because PHP web applications have been around for many years, they offer more data for long-term studies like ours.

Figure 1 shows an example file that contains client-side and server-side code. The server-side code is processed in the web server to client code, merges with the untouched client-side code into a temp file (simplification), and then processed by the browser(web client). Client-side code runs in the browser and is HTML, CSS and JavaScript. JavaScript is a part of

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

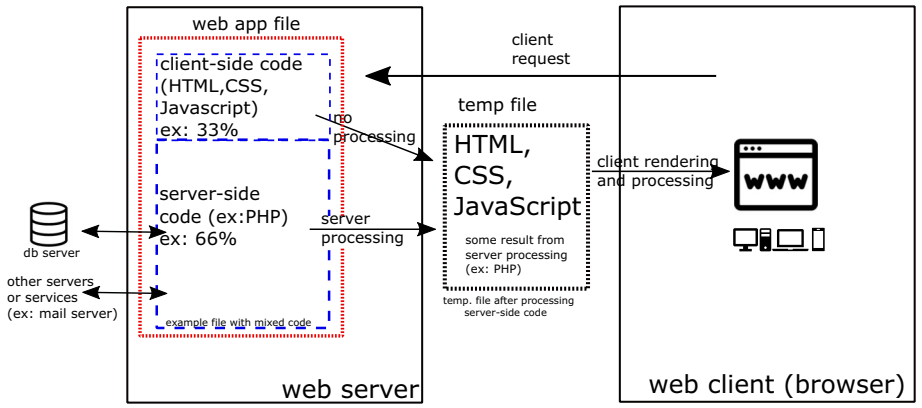


Fig. 1 Anatomy of a PHP monolithic web app, containing server-side and client-side code - very simplified. Percentages are examples. Exact percentages are in Table 1

the client code, often more than half of this code. Server code can be PHP, C#, Ruby, Java, Python, server JavaScript(node.js) or others. While the server code can separated from the client code, even in monolithic applications, for typical PHP web apps the code is entangled. An example of a file with mixed client- and server-side code is shown on Listing 1.

```
<?php $db= new $db(<db credentials>);   //php code
$list=$db->get("table1");
?>
<!doctype html>
<html>
    <head>
        <style>h1{font:somefont;}</style>
    </head>
    <body>
        <script>alert("hello");</script>
        <h1>Display List</h1>
        <?php foreach($list as $line){ ?>
        <h2 style="font-size:30px"><?php print $line['title']?></h2>
        <?php } ?>
        <div onclick="dosomething()">do something</div>
        <script src="lib.js"></script>
    </body>
</html>
```

Listing 1: PHP file code example, with mixed client- and server-side code, PHP code inside tags '<?php ?>'. PHP code runs in web server and after the file is processed in the web browser(the web client)

The code in Listing 1 is a simple example of mixing client-side and server-side code in PHP. The server-side code gets data from the database and transforms it into client-side code. The client-side code, both parsed by the PHP parser and untouched, goes to the browser. The

browser renders the HTML and CSS and runs JavaScript code. Therefore, the file contains client and server code mixed, and it gets parsed twice, once in the web server (the PHP code inside `<?php ?>` tags) and a second time in the web client, i.e., the web browser (by different parsers/compilers).

Therefore, web app CS can be found on the server and the client-side code, but more is needed to know about their evolution, as shown in the literature review. We want to study client and server CS evolution and their inner relationship due to the entangled code. We also aim to discover if CS evolution impacts maintainability metrics such as issues, bugs, and time-to-release, using time-series techniques to assess causality in the same release or previous releases (with lags, i.e., the delays between the time series). We need to use time series to infer statistical causality. The time series are irregular because open-source apps' releases are not regular - they do not release at regular intervals in time.

The main novelties in the study are:

- An evolution study with both server- and client-side CS in web apps;
- The use of irregular time series and special correlation techniques for time series with different observation granularity (release date vs days);
- The use novel statistical techniques to infer statistical causality (e.g., Transfer Entropy) and compare it with other causality methods(e.g., Granger-Causality).
- Use typical web apps with server-side and client-side code and a web smells catalog of both client and server code smells.
- We also developed a tool to detect client CS (link provided in the study design section).

The causality inferred from data, if found, does not mean that CS exclusively determines the cause of the outcome but instead that it contributes to the outcome variability and predictability.

This paper is structured as follows: Section 2 overviews the related work on longitudinal studies on CS and in web apps; Section 3 introduces the study design and methodology; Section 4 describes the results of data analysis, while Section 5 discusses the findings and next section identifies validity threats; finally, the last section outlines significant conclusions and future work.

2 Related Work

Extensive literature on software evolution and CS impact have been published during the last decade. We will refer to CS evolution studies, CS impact studies, and studies with CS in web apps or web languages. To complement the literature review, we will also refer to other evolution studies in PHP and SE using Granger-causality and Entropy.

2.1 Evolution of CS

Olbrich et al. (2009) described different phases in the evolution of CS and reported that components infected with CS have a higher change frequency. Later, Peters and Zaidman (2012) results indicate that CS lifespan is close to 50% of the lifespan of the systems. Chatzigeorgiou and Manakos (2013) reported that a large percentage of CS was introduced in the creation of classes/methods, but very few CS are removed. Later, Tufano et al. (2017) sustain that most CS are introduced when artifacts are created and not because of their evolution. Rani and Chhabra (2017) claim that the latest versions of the observed application have more CS/design issues than the oldest ones. They also note that the first version of the software is

cleaner. Next, the authors Digkas et al. (2017) found that TD (Technical Debt, that includes CS) increases for most observed systems. However, TD normalized to the size of the system decreases over time in most systems. Habchi et al. (2019) conclude that CS can remain in the application code for years before removal, and CS detected and prioritized by linters disappear from code before other CS. Recently, Digkas et al. (2020) found that the number of TD items introduced through new code is a stable metric, although it presents some spikes; and also that the number of commits is not strongly correlated to the number of introduced TD items.

2.2 CS Impact in Issues or Defects/bugs

Li and Shatnawi (2007) analyzed six code smells in three versions of an open-source system, confirming a correlation between three code smells (God Class, God Method, and Shotgun Surgery) and class error probability. D'Ambros et al. (2010) explored the relationship between software defects and the number of design flaws in six open-source software tools across multiple versions. They found a correlation between code smells and software defects but noted no single design flaw consistently correlates more than others. Olbrich et al. (2009) investigated the correlation between the code smells God Class and Brain Class and the frequency of defects detected post-release. They found a higher defect rate in God and Brain classes, but this rate decreases when adjusted for class size.

Marinescu and Marinescu (2011) studied three versions of Eclipse focusing on four class-based code smells and defects. They did not confirm a direct correlation between specific code smell types and defect rates. However, classes affected by code smells increased the likelihood of defects in their clients, especially in post-release defects. Zazworka et al. (2013) examined four approaches to structural flaw detection in thirteen releases of a system, focusing on ten types of code smells. They found a correlation between two code smells (Dispersed Coupling and God Classes) and higher defect-proneness. Bán and Ferenc (2014) explored the relationship between antipatterns (code smells), bugs, and maintainability across different systems, finding a significant positive correlation between the number of bugs and antipatterns.

Khomh et al. (2011) investigated the impact of antipatterns (CS) on the change- and fault-proneness of classes in object-oriented systems. They concluded that classes with antipatterns are more prone to changes and faults than others, and that size alone doesn't explain this difference. Palomba et al. (2017) extended Khomh et al.'s work with a large-scale empirical investigation on the prevalence of code smells and their impact on code change- and fault-proneness. They found that code smells characterized by long and/or complex code are highly diffused and that smelly classes are more prone to changes and faults than smell-free classes.

2.3 CS in Web Apps or Web Languages

Detection Studies Nguyen et al. (2012) presented a list of 6 client-side CS mainly concerning JavaScript and CSS: JS in HTML, CSS in JS; CSS in HTML; Scattered Sources;Duplicate JS; HTML Syntax Error. They claim that WebScent is a tool for detecting embedded CS in server code, but detected CS lie only on the client-side. A year later, Fard and Mesbah (2013) proposed another tool, JNose, to automate the process of detecting JavaScript CS. They also present some JavaScript CS and the embedding (mixing) of JavaScript with HTML. They propose the detection of the following JavaScript CS: Closure smell, Coupling JS/HTML/CSS, Empty catch - Lines of code, Excessive global variables, Large object, Lazy object,

Long message chain, Long method/function, Long parameter list, Nested callback, Refused bequest, Switch statement, Unused/dead code. In Mesbah and Mirshokraie (2012), the authors propose an automated technique to support styling code maintenance, which analyzes the runtime relationship between the CSS rules and DOM elements of a given web application and detects unmatched selectors, overridden declaration properties, and undefined class values. They implement the technique in a tool called Cilla. The results show an average of 60% unused CSS selectors in the applications studied. Gharachorlu (2014) describes a set of 26 CSS smells and errors and proposes an automated technique to detect them, conducting a large empirical study on 500 websites. The author proposes a model to predict the total number of CSS CS, and also shows a study of unused CSS code on 187 websites.

Evolution Rio and e Abreu (2019) studies the survival of CS in web apps, and later (Rio and e Abreu 2021) studies the sudden variations of CS evolution in the life of 8 web applications. This study was extended in Rio and Brito e Abreu (2021) for 8 apps, and the final version was Rio and e Abreu (2023) with novel investigations and 12 web apps. Conclusions: In the evolution of server-side CS of PHP web apps, the CS number increases, like app size. CS density is mostly stable with variations, correlated with the number of developers. CS lifespan median is 4 years, and 61% of CS introduced are removed. Scattered CS (CS that are scattered in the classes) survival is different from localized CS (cs in one class or method). More CS are introduced and removed in the first half of app life. From the 12 apps, sudden increases were found in 5 apps.

Impact These studies include (Saboury et al. 2017), which found that for JS applications and for the time before a fault occurrence, files without CS have hazard rates 65% lower than files with CS. As an extension to the previous paper, Johannes et al. (2019) show the results: files without CS have hazard rates of at least 33% lower than files with CS. In Amanatidis et al. (2017) study with PHP TD, which includes CS, they find that, on average, the number of times 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) study diffusion and impact to change-proneness. They extended the study in Bessghaier et al. (2021), where they replicated studies in Java for the PHP language. They studied a total of 430 releases from 10 open-source web-based applications (5 web-apps and 5 frameworks) on 12 CS. They study the diffuseness of CS, its effects on the change- and fault-proneness in server-side code (replication of Khomh et al. 2011; Palomba et al. 2017), and the CS co-occurrences (replication of 2018). Their findings agree with these previous studies: High complex and large code components have high diffuseness and frequency rates. CS related to large size and high complexity exhibit higher co-occurrences. Smelly files are more likely to change and more vulnerable to faults than smell-free files.

2.4 Evolution Studies in PHP

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. They found that these systems undergo systematic maintenance. Later in Amanatidis and Chatzigeorgiou (2016), the authors analyze 30 PHP projects extracting their metrics to verify if Lehman's

laws of software evolution are confirmed in web applications and found that not all of them stand.

2.5 Studies in SE with Granger-causality and Entropy

Granger-causality Some studies have already used Granger-causality. Couto et al. (2014) proposed a link between source code metrics and the occurrence of defects, using the Granger Causality Test to determine if past variations in source code metrics can forecast changes in defect trends. They applied this approach to four Java-based systems in various life stages and achieved an average precision greater than 50% in three of the four systems. This suggests that source code metrics can be effective predictors of future software defects. Palomba et al. (2018) analyzed 13 code smells in 30 software systems to explore the co-occurrence of code smells, which types often appear together, and the reasons for their introduction and removal. Key findings include that 59% of classes with code smells are affected by more than one smell, six pairs of smell types frequently co-occur, method-level smells may lead to class-level smells, and that code smell co-occurrences are typically removed together during maintenance. Sharma et al. (2020) implemented detection support for seven architectural smells and analyzed 3,073 open-source repositories to study the characteristics of architectural smells and their correlation with 19 design smells. The study, which included a causation analysis within five repositories, found that smell density is not dependent on repository size, architectural smells are highly correlated with design smells, most design and architectural smell pairs do not collocate, and that design smells often lead to architectural smells.

Entropy Some studies used entropy, but in prediction models, with regressions. Gupta et al. (2018) proposed a mathematical model to predict bad smells using the concept of entropy defined by the Information Theory. They use 6 code smells and 7 releases of one open-source software (Apache Abdera). They use different measures of entropy (Shannon, Rényi and Tsallis entropy) to apply non-linear regression techniques to build a prediction model for bad (code) smells. The model is validated using goodness of fit parameters and model performance statistics and they compared the results of the prediction model with the observed results on real data in the 7 releases. Other studies used entropy-based bug prediction using support vector regression (SVR) (Singh and Chaturvedi 2012) and the complexity of code changes using entropy-based measures (Chaturvedi et al. 2014).

2.6 Related Work Conclusions

Because a great percentage of web apps have server and client code entangled, often in the same files, further studies are needed using both CS from the client and server side in web apps. Furthermore, the studies on the impact of CS are mainly at the class level (even outside the web). Therefore, there is a need for studies of the effects of CS at the system level because HTML and CSS do not have classes. Another reason is that a code smell in file A can cause a bug in file B. Studies in impact measure and correlate the CS and bugs in one file (calling it smelly and not smelly, independent of whether the CS appears in the first release or the last). Therefore, time series longitudinal studies dealing with causality inference are needed to uncover how CS in past releases impacts bugs, issues, and time to release in the same and following releases.

3 Methods and Study Design

We investigate the evolution of CS in typical (monolithic) web apps and what this evolution causes to the applications' evolution maintainability problems and time to release delays. Monolithic web apps have server-side and client-side code entangled in the same code base, sometimes mixed in the same file. Therefore, we expect some CS from one side to impact the other for several reasons, including coding culture, knowledge of web code smells, and reduced time to release the entire app.

First, we study the evolution of server-side and client-side CS and assess if they evolve similarly or if there is a difference. In larger projects, it is usual to have two teams for the development (client-side and server-side code) or a third one specialized in JavaScript. However, in other projects or small projects, one team develops all. We interviewed a group of specialists with more than ten years of experience, selected from the industry, and the most common pattern is that client-side code is done first (the templates). Secondly, we will study if CS from client-side code impact CS from the server-side and vice versa in the monolithic code base.

For the analysis of CS groups, we will consider three groups: the server-side CS and we specialized the client-side CS in two groups: embed CS - CS concerning the mixture of languages - and JavaScript CS.

Next, we aim to discover if the CS evolution of the various types of server-side and client-side CS contributes to the evolution and number of app issues, bugs, and delays in releases (or application time-to-release). We study the individual CS and CS groups. Issues are the reported "issues" in each app development's "issue tracker" tools. Bugs are issues that can be classified as bugs; some are labeled as bugs as some have the word "bug" or a synonymous in the description (Antoniol et al. 2008). Time-to-release is the number of days between two official releases.

It is expected that if CS increases, that can make an issue increase that asks for refactorings (the removal of CS). That will increase maintainability. It is also expected that an increase in CS will increase the number of bugs; this is studied in the literature. It is also expected that if CS increases, the time-to-release increases.

Thus, we translated these study topics into the following research questions.

3.1 Research Questions

- **RQ1 - How do server and client-side Code Smells evolve?** - This question will lead to uncovering the evolution of the different CS on the server-side and client-side , individually and as groups (server side-CS, client-side embed CS, and client-side JavaScript CS).
- **RQ2 - Is there a relationship between server- and client-side Code Smell evolution?** - In our target applications, the server- and client side-code is entangled in the same code base. So getting CS from all groups in the same files is expected. We want to find if the evolution of one group of CS impacts the others. The answer to this question will explain if groups of CS (server-side, client-side embed, and client-side JavaScript CS) evolve in the same way (by time-series correlation) and if there is statistical causality in the evolution of one group of CS to the other. The statistical causality is verified between the same and previous releases of variables, up to four releases behind, with linear and non-linear measures. In this article, "release" means a full release of the software to the public.

- **RQ3 - Does server- and client-side code smells evolution impact web app reported faults (bugs)?** We want to understand if the CS evolution of the various smells impacts the number of reported bugs in the evolution of the app. Furthermore, we want to study if the CS density change causes changes in the number of bugs reported (considering only the reported date) in the evolution of the web app. We study correlation, and causality between individual CS ans bugs, and causality between CS groups and bugs. We also performed the same studies for the issues not filtered as bugs.
- **RQ4 - Does CS evolution impact "time to release" in a web app?** The answer to this question will help us uncover if the evolution of CS causes delays in the "time to release" of the program, i.e., the dates of the full release of the web apps. We already know that CS increase will hinder readability (Mannan et al. 2018; Yamashita and Moonen 2012, 2013), but we want to find the answer inferred from observed data. We study causality relationships between individual CS and time to release, and between CS groups and time to release.

In summary, the first two questions analyze the evolution of server- and client-side CS and the possible relation/causality between their evolution. In full-stack development practices, the same developers work on the server and client code, while there is a clear separation of teamwork in other web applications with two or even more teams. On the other hand, client- and server-side code are intertwined in the same codebase and the same files in typical PHP web apps. Because of this, some causality relations between both sides' CS evolution are expected, especially if teams (or teams) don't avoid CS in code on both sides. Therefore, we want to understand whether the CS of the client- and server-side have a relationship or causality between them. It is expected that if the CS from the client-side rise, the server-side CS also rise in subsequent releases, and vice-versa, because the CS are on the same code base and sometimes in the same file.

For the remaining two questions, we want to verify the degree of correlation and causal statistical impact of the CS evolution with the progression of the issues, bugs and "time to release" of the application. It is expected that an increase in CS provoke an increase in the bugs or time-to-release of the app by a certain amount. To measure causality and quantity, we will use the statistical and time series methods described in the "Statistics Used" subsection, which can statistically uncover these relations and causal inferences. These statistical methods will have to deal with the irregular release dates of OSS software. We will analyze CS inference individually and in three groups.

Studying individual CS relations and statistical causalities gives each CS relative importance and allows prioritizing its removal. Studying the same but in groups gives us a macro perspective of which group/side of CS has a higher impact on the outcome variable. This can make lead developers or managers correct quality problems in code made by specific teams or developers.

3.2 Apps Sample

We built the list of applications to analyze from the most forked PHP applications on GitHub - not all were web apps. Web apps are installable in a web server. For comparing client and server CS, the web apps must be complete apps (monolitic), i.e., they must have server-side code and client-side code, witch is the norm for PHP web apps. Then, we applied the following criterion:

- Inclusion criteria:

  - open-source web apps built with PHP as the server-side language
  - code available;
  - self-contained apps (server- and client-side code mixed requirement);
  - programmed with object-oriented style (PHP can also be used in a pure procedural way, but server-side CS used in the study are for object-orient programming);
- Exclusion criteria:
  - libraries (libraries do not run alone - they are included in other apps);
  - frameworks or apps used to build other apps (the structure of frameworks to build web apps are very different from the web apps; some of them are to be used to a great extent in the command line)
  - web apps built using a framework (part of the code would be very similar).

The tool we used to detect server-side CS works with object-oriented programming (OOP) CS and code. Thus, we excluded some well-known PHP apps because their core was not OOP, and frameworks and libraries since we target typical web apps. These typical web apps must contain server and client code (monolithic) . Furthermore, we excluded older releases of the apps when the PHP code was not OOP (for example, phpMyAdmin < 3.0.0). PHP can use OOP since version 4, but some apps delayed the move to OOP for several years because of web server support (PHP Apache module version). As usual in web applications, all the applications use a database for persistence, but we only analyze the code.

Table 1 shows the complete list of apps. The KLOC and percentage of code (% Code) numbers were measured by *CLOC CLOC*. The percentages in the last three columns represent the server, client-side (HTML+CSS+JavaScript), and client-side JavaScript (JS) code and their percentages breakdown. The column "client" contains the JavaScript code, but we also have the JavaScript code percentage as a separate column. So, in summary, server-side code + client-side code =100%; the column client code includes JavaScript code, but this JavaScript

Table 1 Web apps sample used in this paper

Name	Purpose	#Releases(period)	Versions	AVG KLOC			% Code Type		
				Server	Client	JS	Server	Client	JS
phpMyAdmin	Database admin.	179 (09/2008-09/2019)	3.0.0 – 4.9.1	138	70	58	66%	34%	28%
DokuWiki	Wiki	40 (07/2005-01/2019)	2005-07-01-2018-04-22b	92	19	13	83%	17%	12%
OpenCart	Shopping cart	26 (04/2013-04/2019)	1.5.5.1 – 3.0.3.2	118	177	80	40%	60%	27%
phpBB	Forum/BBS	50 (04/2012-01/2018)	2.0.0 – 3.2.2	112	27	2	80%	20%	1%
phpPgAdmin	Database admin.	29 (02/2002-09/2019)	0.1.0 – 7.12.0	18	3	2	85%	15%	10%
MediaWiki	Wiki	138 (12/2003-10/2019)	1.1.0 – 1.33.1	135	32	24	81%	19%	15%
PrestaShop	Shopping cart	74 (06/2011-08/2019)	1.5.0.0 – 1.7.6.1	210	145	81	59%	41%	23%
Vanilla	Forum/BBS	63 (06/2010-10/2019)	2.0 – 3.3	61	46	24	57%	43%	23%
Dolibarr	ERP/CRM	83 (02/2006-12/2019)	2.0.1 – 10.0.5	310	26	8	92%	8%	2%
Roundcube	Email Client	31 (04/2014-11/2019)	1.0.0 – 1.4.1	102	51	30	67%	33%	19%
OpenEMR	Medical Records	33 (06/2005-10/2019)	2.7.2 – 5.0.2.1	271	370	225	42%	58%	35%
Kanboard	Project manag.	65 (02/2014-12/2019)	1.0.0 – 1.2.13	49	6	2	90%	10%	3%

Sizes and code percentages are averaged by all releases studied

code is shown again in another column to analyze its percentage in the client code. The numbers do not include external library/third-party folders.

3.3 Web CS Catalog

The next three sections present the web server catalog for the studies. Besides the apparent separation between the server-side and client-side, we have further specialized our CS catalog on the client-side into two categories: the embedded part (mixture of languages) and the programming part (JavaScript). Most CS used in this paper, covering the server and client-side, were defined by other researchers, and tool collection availability was a relevant selection criterion. Nevertheless, as described later, we had to develop a collection tool for client-side CS. In the following subsections, we will briefly describe each adopted CS.

For the selection of the PHP and JavaScript CS, we interviewed another group of specialists (CS specialists selected in the same investigation center), and only CSs that were not ambiguous were considered. Another reason to consider the CS was the possibility of detection. Examples of Ambiguous in PHP: Superglobals, CamelCaseClassName, CamelCasePropertyName, GoTo; Ambigous in JavaScript: Closure, Empty Catch, switch. Examples of not ambiguous - all the others included in the catalog.

3.3.1 Server-side CS Catalog

For the server CS, we used *PHPMD*, an open-source tool that can detect CS in PHP (Dusch et al. 2021). The chosen subset of server-side CS, presented in Table 2, corresponds to the

Table 2 Characterization of server-side Code Smells

Code Smell	Description	Threshold
(Excessive)CyclomaticComplexity	Method number decision points plus one	10
(Excessive)NPathComplexity	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 long parameter list	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	Exc. Sum complexities all methods class	50
(Excessive)NumberOfChildren	Class with an excessive number of children	15
(Excessive)DepthOfInheritance	Class with many parents	6
(Excessive)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

Names of Code Smells are the ones presented by the tool. "(Excessive)" was added to denote a CS and not a metric. Original thresholds from the tool

Table 3 Characterization of Client embedded CS

Code Smell	Code Smell Description
embedded JS	JavaScript inside HTML page inside <script>tag
inline JS	JavaScript inside HTML page in the elements themselves
embedded CSS	CSS inside HTML page inside a <style>tag
inline CSS	CSS inside the HTML page in the elements themselves
CSS in JS	CSS code in JavaScript
CSS in JS: jQuery	CSS code in jQuery

more recurrently used in the literature, although sometimes with different names. Although they may be disputable, we did not change the proposed thresholds used by *PHPMD* for CS detection (3rd column in Table 2) for comparability's sake with other studies using the same tool. The names of the CS are presented by the tool used, *PHPMD*, with the word "Excessive" in parenthesis, to indicate that it is a CS and not a metric.

3.3.2 Client-embedded CS Catalog

The client-embedded CS were among the first reported in the literature for web apps (Nguyen et al. 2012; Fard and Mesbah 2013). The first article introduces the CS and performs an empirical study revealing their implications. We asked several experts in the field for confirmation, and they agreed that the embedded client CS hinders several aspects of web development. Since we could not obtain the collection tool mentioned in Nguyen et al. (2012), we developed another one dubbed *eextractor*[2] and allowed us to collect the CS represented in Table 3. We propose the last smell, which refers to jQuery, a widely used JavaScript library designed to simplify HTML DOM tree traversal and manipulation, event handling, CSS animation, and Ajax.

3.3.3 Client JavaScript CS Catalog

JavaScript CS for client-side were first reported here (Fard and Mesbah 2013), but since then used in other studies besides the client-side programming, for example Saboury et al. (2017) that uses mostly JavaScript libraries as sample. We faced difficulties running the initial version of JSNose (Fard and Mesbah 2013) with dependency issues at the time of our study. Therefore, for extracting the client JavaScript CS described in Table 4, we used *ESLint*, a pluggable and configurable linter tool for identifying and reporting on patterns in JavaScript (Zakas et al. 2021).

3.4 Study Design - Data Extraction

We fully automated the workflow of our study through shell and PHP command-line scripts. First, we downloaded all versions/releases of the selected web apps from GitHub, Sourceforge, or the app's official site, except the alpha, beta, release candidates, and corrections for old versions.

[2] available in https://github.com/studywebcs/eextractor"

Table 4 Characterization of Client JavaScript Code smells

Code Smell	Description	Threshold
max-lines	exceeds number lines per file	300
max-lines-per-function	exceeds number line of code in a function	20
max-params	exceeds number parameters in function	3
(Excessive)complexity	exceeds cyclomatic complexity allowed	10
max-depth	exceeds depth	4
max-nested-callbacks	exceeds depth nested callbacks	3

Names of Code Smells are the ones presented by the tool. "(Excessive)" was added

Server-side CS Then, using *PHPMD*, we extracted the location, dates, and other indicators of the CS from all versions and stored that data in XML format, which is one of the outputs of this tool. We excluded some directories not part of the web app (e.g., vendor libraries, images). The excluded folders in the example apps in 2 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.

Client-side Embedded CS For the embedded CS, we developed a tool, dubbed *eextractor* (embedded extractor), that scraps the release, gets the CS, and records their occurrences in a database. We excluded the same folders from the analysis.

Client-side JavaScript CS For the JavaScript CS, we first separate the embedded JavaScript that is inside the HTML and PHP files and put them on files in a folder named "embed". Thus, each release of the apps will have a folder "embed" with the embedded JavaScript code. Then, we extract the CS from the external (regular .js files) and embedded JavaScript (.js files in the embed folder). This extraction is performed using *ESLint*.

Size Metrics We collected *Size Metrics* by release with the tool *CLOC* [3]and stored them in tables of the database used, where later we exported them to CSV format. *CLOC*, and another tool that we used recently, *PHPLoc*, counts client code in PHP files as PHP, so we had to use an older tool, *SLOCCount*, to provide the correct count of PHP lines, only inside PHP tags. We made the difference between these two counts and counted the code outside PHP tags as client code. The code from templates is also counted as client code. We used the lines from the ".js" files and the embedded JavaScript code we previously separated for the JavaScript count.
`PHP lines of code = CLOC PHP lines of code`[4] `- SLOCCount lines of code` [5] We excluded the same third-party folders in the CS extraction for all the size counts (in all languages).

Issues For most applications, the issues were taken from *GitHub* issues, using their API. For phpBB, we extract the issues from *Jira*. The *phpMyAdmin* devs lost some issues during an early period, so we retrieved them from *Sourceforge* and joined them with the GitHub one,

[3] https://github.com/AlDanial/cloc

[4] including HTML, CSS, JavaScript

[5] measured with *cloc –use-sloccount*

removing the duplicates. We elaborated small scripts/programs that retrieve the issues from the various APIs and insert them into the database, to be exported later to CSV format files.

Bugs Bugs are the reported issues classified as bugs by label or in words inside the description (Antoniol et al. 2008). Part of the applications studied classify the bugs with labels, while other applications do not do this. Since we have all data in a database, we devised an algorithm that reunites the labeled issues with syntax detection (by word) from studies in the literature (Śliwerski et al. 2005; Ayari et al. 2007; Antoniol et al. 2008). The "Methodology for each RQ" section will explain this search criterion.

Time to Release we computed the "time to release" (RQ5) from the release dates of the apps.

Figure 2 shows the study design to collect data, extract CS and analysis.

3.5 Study Design - Data Pre-processing

After we collected the CS, we stored them in a database, each application having three tables for CS, 'server', 'client', and 'client_javascript'. In another DB table, we stored the Lines of code for the various languages and the Total LOC, as described in the "data extraction" sub-section. HTML files allow for CSS and JS inclusion, so we had to count the lines with this in mind. On the other hand, PHP files allow for HTML, CSS, and JS, so the size count must consider and count lines inside PHP tags (as referred to in the study design).

Because we want to check for causality in the time series, we count every CS by release (using a script developed by us) and export them in .csv format. In this dataset (one file per app), we have the release date, 18 server-side CS, and 12 client-side (embed CS and JavaScript). We also have the metrics for each language.

CS density - we performed all the studies with CS density. CS density is calculated in the following way:

- Server CS density = #CS server / PHP lines of code (this excludes HTML lines in ".php" files as denoted in the study design section).
- Client CS density = #CS client / Client lines of code (HTML, CSS, Javascript, Templates)
- JavaScript CS density = #CS JavaScript / JavaScript lines of code (.js files + embed lines of code)

Using another script, we extracted the CS densities. At this point, we had the data for the first two RQ and the RQ4.

For the RQ3, we will have two sets of studies; in the first set (correlation of irregular time series with different granularity), we compare the CS by release with issues by day, and in the second set (statistical causality) have to aggregate the bugs by release. After we downloaded the issues and put them in a separate database for issues, we extracted the bugs from the issues, as described in the methodology of RQ3. Then, we had to aggregate the daily data by release to allow for causality statistics inference (it is required to have the same number of observations in the independent and dependent variables) However, for the 'irregular time-series correlation' (cor_ts), we summed the number of bugs by reported day, and there was no need to aggregate by release. After, we exported the bugs by day and the aggregated bugs by release to two datasets (per app) available in the replication package. Please see Fig. 4 for the two time-series of bugs, one by release and the other by day. For some applications, the

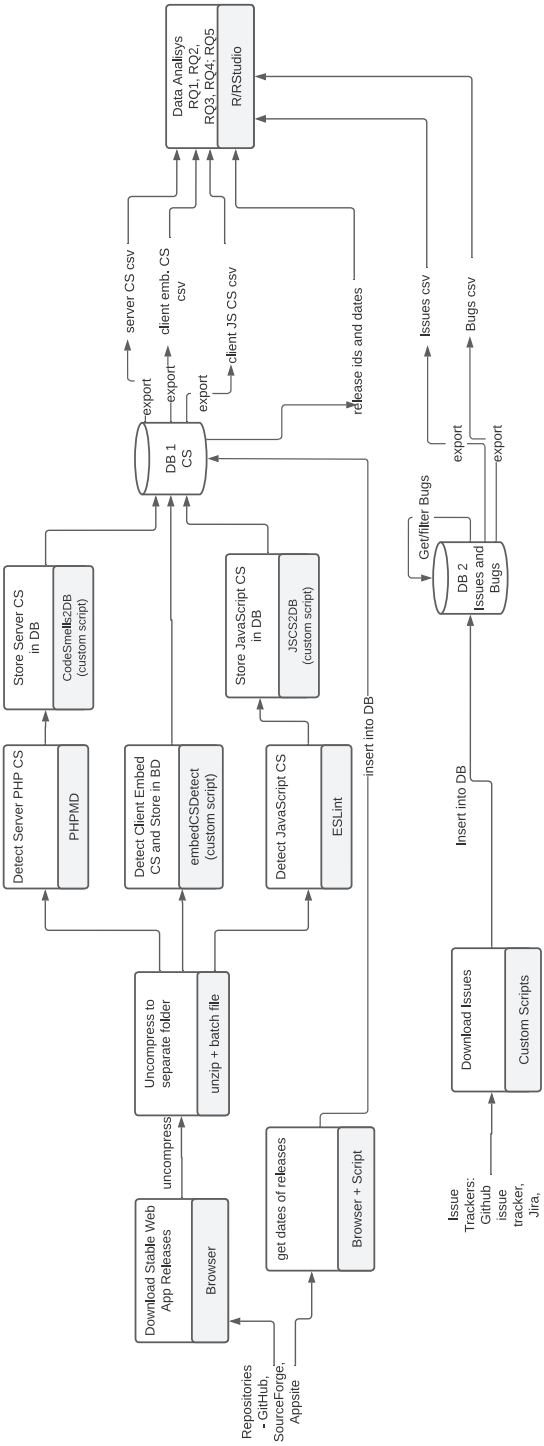


Fig. 2 Study design/methodology - Universal Project Notation (UPN)

bugs' initial dates differed from the releases' initial dates. So we had to remove some values of the bugs time-series to have intervals with both CS and bugs.

3.6 Statistics Used - Irregular Time Series Analysis and Causality

The software releases from the projects studied do not have releases equally spaced in time, leading to irregular time series, where the data is not equally spaced in time. Because we detect the CS from the software releases, they are irregular time series. The reported issues and bugs are also irregular time series but have a different granularity (the date of the report), and they are much more frequent in time.

Several challenges are implicit in the study of raw longitudinal data, especially when the observed data are short and irregularly sampled. Therefore, we must access some unconventional methodologies to obtain validated results to overcome these problems.

Causal inference between irregular time series, with linear and nonlinear relations, can be detected and quantified through various methods. The techniques include the Granger Causality Test and Vector Autoregression (VAR) Models, which assume linear relationships, and nonparametric methods like Convergent Cross Mapping (CCM) for nonlinear systems. In addition, transfer Entropy helps detect linear and nonlinear relationships in irregular time series. At the same time, Dynamic Time Warping (DTW) can establish relations between irregular series [6]. Recurrence Networks help detect complex synchronizations, especially useful for non-stationary series, and Deep Learning Models like LSTM networks or RNNs can handle irregular series and nonlinear relationships.

We used the Granger-causality method to detect linear relations between the time series and Transfer entropy to detect both linear and non-linear relations. These relations relate the present of one time series with the past of the other. We selected TE because it can quantify in percentage the transfer of information between two time series. Moreover, both of these methods can work with irregular time series.

To relate the time series at the same time (same release), we used correlations (the standard *correlation* if time series have the same granularity, and cor_ts for time series with different granularity) and linear regression.

In what follows, we summarize the main statistical tools and the R libraries used to attain our purposes.

3.6.1 Time series Correlation (cor_ts) - with irregular series and with different timescales

To calculate correlations between unevenly sampled time series (Reschke et al. 2019), we used the *cor_ts*[7] function from the R package **Bincor** (Josue et al. 2019). This function estimates Pearson and Spearman correlations for binned time series, employing the native R function *cor.test* (from the **stats** package). Binning refers to resampling the time series on a regular grid and assigning mean values within those bins (Josue et al. 2019; Mudelsee 2014). The *cor_ts* function can be applied to irregular time series where observations occur at the same time moments and series with different timescales.

[6] https://towardsdatascience.com/inferring-causality-in-time-series-data-b8b75fe52c46

[7] https://www.rdocumentation.org/packages/BINCOR/versions/0.2.0/topics/cor_ts

Unrelated time series can display spurious correlations if they share drift in the long-term trend. Time series data commonly depend on time, and Pearson correlation is reserved for independent data. This problem is related to the so-called spurious regression. The simple solution to this problem is to model the data (linear regression) and then analyze the produced residuals. If the residuals are stationary, that is,

$$P\{y_{t_1} \leq b_1, \ldots, y_{t_n} \leq b_n\} = P\{y_{t_1+m} \leq b_1, \ldots, y_{t_n+m} \leq b_n\} \tag{1}$$

i.e., the probability measure for the sequence y_t is the same as that for y_{t+m}, $\forall m$ (the distribution of the values does not change with time), then the regression/correlation it is not spurious. Otherwise, we need to apply the first order difference operator, $\Delta y_t = y_t - y_{t-1}$, to eliminate time dependency and subsequently compute the correlations between the transformed variables in the dataset.

Time series non-stationarity will be analyzed by appropriate statistical tests, like unit root and stationary tests (Shin and Sarkar 1996; Mills 2019).

3.6.2 Granger Causality

The understanding of cause-effect relationships is a meaningful task for the perception of the functionality/consequences of CS and the various evolution metrics studied. There are several studies in the scientific literature related to time-series methods based on the notion of Granger causality (Bahadori and Liu 2012; Heerah et al. 2020; Siggiridou and Kugiumtzis 2016). This causality concept is based on the idea that causes must precede their effects in time. Temporal precedence alone is insufficient to prove cause-effect relationships, and skipping relevant variables can lead to spurious causality (falsely detected causality).

Otherwise, it is said that a spurious relation between two variables occurred if the statistical summaries show significant relations, where, from the theoretical point of view, there is no reason for these relations to exist. Another reason for spurious results is the non-stationary property of time series. The unit root and co-integration analysis were developed to cope with the problem of spurious regression.

The Granger causality test (Granger 1969, 1988) is a statistical test for determining whether a time series offers valuable information in forecasting another time series. Proposed by Nobel Prize winner Clive Granger (1969), the causality hypothesis could be tested by measuring the ability to predict the future values of a time series using prior values of another time series, or "causes must precede their effects in time." It is a concept that can be applied to stationary time series.

More formally, let's consider the case of two variables x_t and y_t. Then x_t does not Granger cause y_t if, in a regression model of y_t on lagged values of x_t and y_t, that is,

$$y_t = \sum_{i=1}^{p} \alpha_i y_{t-i} + \sum_{i=1}^{p} \beta_i x_{t-i} + u_t \tag{2}$$

all the coefficients of the former are zero. So, x_t does not Granger cause y_t, if $\beta_i = 0, i = 1, ..., p$. Straightforward, the null hypothesis of the Granger non-causality can be formulated as below:

$$H_0 : \beta_i = 0, \quad i = 1, \cdots, p \tag{3}$$

This test is only valid asymptotically since the regression includes lagged dependent variables, but standard F tests are often used in practice.

Granger Causality [8] is an algorithm that takes into account the information content of signals and can be applied to stationary time series. Therefore, we need to remove all components that can be predicted from its own history; that is, we convert the given data into (unpredictable) noise which, as Shannon described (Shannon 2001), represents the information contained within it. Then, we test if we can predict the noise time series from prior values of a second-time series. If possible, the second series is a predictive cause of the first one. This latter view of Granger causality bridges the third methodology we employed in our analysis, based on entropy transfer. The Transfer Entropy is considered to be a non-linear version of *Granger causality*, and it is a robust approach for estimating bi-variate causal relationships in real-world applications (Edinburgh et al. 2021).

3.6.3 Transfer Entropy

Information-theoretic measures, such as entropy and mutual information, can quantify the amount of information necessary to describe a dataset or the information shared between two datasets. For dynamical systems, the actions of some variables can be closely coupled, such that information about one variable at a given instance may supply information about other variables at later instances in time. This flow of information can expose cause-effect relationships between the state variables.

Transfer Entropy (Behrendt et al. 2019; Schreiber 2000) is already established as an important tool in the analysis of causal relationships in nonlinear systems since it permits the detection of directional and dynamical information without assuming any functional form to describe interactions between variables/systems. The measurement of information transfer between complex, nonlinear, and irregular time series is the basis of research questions in various research areas, including biometrics, economics, ecological modeling, neuroscience, sociology, and thermodynamics. The quantification of information transfer commonly relies on measures that have been derived from subject-specific assumptions and restrictions concerning the underlying stochastic processes or theoretical models. Transfer entropy is a non-parametric measure of directed, asymmetric information transfer between two processes.

Given a coupled system (X, Y), where $P_Y(y)$ is the probability density function (pdf) of the random variable Y and $P_{X,Y}$ is the joint pdf between X and Y, the joint entropy between X and Y is given by:

$$H(X, Y) = -\sum_{x \in X} \sum_{y \in Y} P_{X,Y}(x, y) \log P_{X,Y}(x, y). \tag{4}$$

and the conditional entropy is defined by the following:

$$H(Y \mid X) = H(X, Y) - H(X) \tag{5}$$

We can interpret $H(Y \mid X)$ as the uncertainty of Y given a realization of X.

The Transfer Entropy (Schreiber 2000) can be defined as the difference between the conditional entropy of each variable in the system:

$$TE(X \to Y \mid Z) = H\left(Y^F \mid Y^P, Z^P\right) - H\left(Y^F \mid X^P, Y^P, Z^P\right) \tag{6}$$

which can be rewritten as a sum of Shannon entropies:

$$TE(X \to Y) = H\left(Y^P, X^P\right) - H\left(Y^F, Y^P, X^P\right) + H\left(Y^F, Y^P\right) - H\left(Y^P\right) \tag{7}$$

[8] https://www.rdocumentation.org/packages/lmtest/versions/0.9-38/topics/grangertest

where Y^F is a forward time-shifted version of Y at lag Δt relatively to the past time-series X^P, Y^P and Z^P. Within this framework we say that X does not G-cause Y relative to side information Z if and only if $H\left(Y^F \mid Y^P, Z^P\right) = H\left(Y^F \mid X^P, Y^P, Z^P\right)$, i.e., when $TE\left(X \to Y, Z^P\right) = 0$.

Transfer-entropy is an asymmetric measure, i.e., $T_{X\to Y} \neq T_{Y\to X}$, and it thus allows the quantification of the directional coupling between systems. The Net Information Flow is defined as

$$\widehat{TE}_{X\to Y} = TE_{X\to Y} - TE_{Y\to X}. \tag{8}$$

One can interpret this quantity as a measure of the dominant direction of the information flow. In other words, a positive result indicates a dominant information flow from X to Y compared to the other direction.

We used the R package RTransferEntropy [9], which can quantify the information flow between two stationary time series and its statistical significance using Shannon transfer entropy (Shannon 2001) and Renyi transfer entropy (Rényi 1970). A core aspect of the provided package is to allow statistical inference and hypothesis testing in the context of transfer entropy.

3.6.4 Mapping between Granger-causality and Transfer Entropy

It has been shown (Barnett et al. 2009) that linear Granger causality and Transfer Entropy are equivalent if all processes are jointly Gaussian. In particular, by assuming the standard measure (l2-norm loss function) of linear Granger causality for the bivariate case as follows:

$$GC_{X\to Y} = \log\left(\frac{\text{var}\left(\epsilon_t\right)}{\text{var}\left(\hat{\epsilon}_t\right)}\right)$$

the following can be proved (Barnett et al. 2009):

$$TE_{X\to Y} = GC_{X\to Y}/2.$$

This result provides a direct mapping between the Transfer Entropy and the linear Granger causality measures.

3.7 Methodology for each RQ

3.7.1 RQ1 - Evolution of CS in Web Apps on the Server and Client Sides

We started by analyzed the evolution of the various individual CS in the proposed catalog. i.e., server-side CS, client-side embed CS, and client JavaScript CS, and next, we measured the correlation of the various CS within their group for the three groups, in each application. We used the standard R (cor) correlation and cross-correlation tables. We also computed the average correlation on all apps aggregated.

Finally, we analyzed the evolution of CS as a group/programming scope (server-side programming, client-side, and client-side JavaScript programming). We calculated the trend with linear regression and plotted the graphs of this evolution to visually understand code smells' group evolution.

[9] https://cran.r-project.org/web/packages/RTransferEntropy/vignettes/transfer-entropy.html

3.7.2 RQ2 - Relationship between Client-side CS and Server-side CS Evolution

This research question aimed to find relationships between CS groups' evolution and causality relations between the same groups, with and without lags. All time series use "code smells density" to avoid the problems posed by the increase or decrease in the application size.

First, we studied the correlation between CS groups with the standard "cor" function of R because they are measured in the same release, and the time series have the same number of observations. As a result, we obtain the following three combinations: server and client, server and client_js, and client and client_js.

Next, we studied for causal inference from one group of CS to the other, using the six combinations possible for the three groups: Server =>Client; Server =>client_js; Client =>Server ; Client =>Client_js; Client =>Server; Client_js =>Client. Because typically the client code is developed before the server code, we would expect the client-side CS to appear before the server CS and even contribute to their appearance. For the same reason, we would expect the client embed smells to appear before the JavaScript smells. However, not all applications are developed in the same order.

We used Linear regression models to test the causal inference between CS in the same release. Next, we applied Granger-causality to test for causal inference for CS data, considering lags (time delays) between one and four. The "lags" mean that we question if a group of CS's past values (in a previous software release) would impact the present values of another group of CS. Figure 3 shows this impact. Finally, we measure nonlinear causality using the same lags (between one and four). To this end, we used Transfer Entropy since it can detect if there is information transfer from one "time series" to another and quantify it. The Transfer Entropy methodology also serves as a double-check of causality but extending it because it can detect linear and nonlinear causality.

The Granger causality requires time series to be stationary, so we tested this before other studies. However, considering the absolute number of CS, the time series are mostly non-stationary, and we have to make some transformations (differentiating, logarithms, etc.) to stabilize the variables. Nevertheless, because we wanted to study the evolution of CS density (divided by KLOC), the time series turned out to be stationary.

3.7.3 RQ3 - Impact of CS Intensity Evolution on Faults (bugs)

We wanted to find relations between code smells and faults/bugs in the web applications on the date the issue was reported, so we filtered the "issues" we had already collected. Some applications label the faults/bugs as such, while others do not mark them. So we consulted

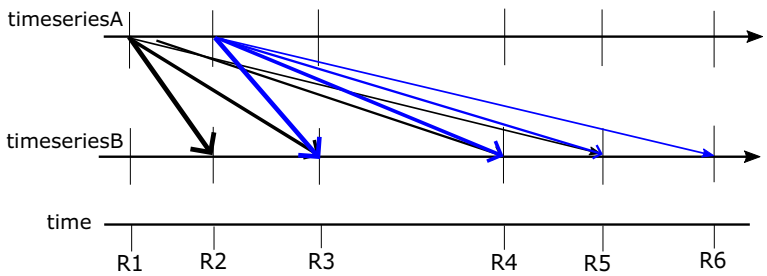


Fig. 3 Previous releases in timeseries 1 impact next release on timeseries2

the literature (Śliwerski et al. 2005; Ayari et al. 2007; Antoniol et al. 2008) and the words more common to find the bugs were: **bug, problem, error, defect, fail, fix** (but not fixed or fixes, or suffixes) and we constructed the following database query:

```
insert table_bugs_more SELECT * FROM table_issues WHERE
( labels like '%bug%' or title like '%bug%' or title like '%problem%'
or title like '%error%' or title like '%defect%' or title like '%fail%'
or title like '[[:<:]]fix[[:>:]]' or body like '%bug%' or body like '%problem%'
or body like '%error%' or body like '%defect%' or body like '%fail%' or body
like '[[:<:]]fix[[:>:]]' ) and title not like '%fixe%'
and body not like '%fixe%' and not labels like '%duplicate%' order by created
```

This query ensures that the date is the creation/report date of the bug, not the fixing date of the bug. Since we are searching for causality relations, we want to ensure we relate the bugs when reported. Hence we search for "fix" but not "fixed" or "fixes".

We could populate a bug table for each application with this query, which we later exported to CSV. However, for the analysis with "cor_ts", we could only use releases with bugs in the issue tracker, which was not the case in the early releases of every application. So we had to find the minimum release with bugs reported. In the other studies with regression, Granger causality, and Transfer Entropy, we did the same minimum date removal after the aggregation (aggregation referred to in RQ3) - we found the minimum release with both bugs (or issues) and CS, and used only data after that release. Next, we performed the same studies as the previous RQ, first for individual CS and after as a group.

Next, to answer the research question, we analyzed the impact of separated CS first and later as a group. In the individual CS assessment, we had 30 "time series," one for each code smell, and did the assessments:

Specialized time-series correlations (cor_ts) - correlations between two unevenly spaced time-series or time-series not on the same time grid. In the present study, we also had different timescales, one on the release date for the CS and another by day for the issues. The issues are in absolute numbers because we did not have the total app size or the lines of code for each day to calculate the CS density.

Causality inference from the CS to the bugs: Similarly to RQ2, we analyze causality by handling Linear regression, Granger Causality, and Transfer entropy.

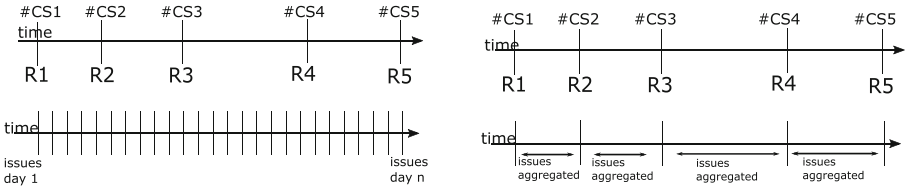
For the causality studies, we had to aggregate the issues by release date, as shown in what follows:

Bugs Aggregation: The CS time series data, including metrics, is keyed by the version release date, while the issues are by days, so they have different granularity. Fortunately, the *cor_ts* [10] R package allows calculating relations between such time series and allow us to employ the original time series in the calculation. However, for the other statistical methods, we aggregated the issue data on the same (release) dates as the CS time series.

Figure 4 shows the feasible structure of the irregular data and how we can use it for correlation and causality effects. When using the cor_ts function, which can compare irregular time series with different intervals and measures, we used the time series on the left image (a) with the bugs reported by day. For the Linear Regression, Granger Causality, and Entropy Transfer, we aggregated the "issues" by the release date, as shown in the figure on the right (b).

After, for the CS 3 groups (server, embed client, and JavaScript client) impact on bugs, we only used the transfer entropy (lags 1 to 4) because is the most complete statistics (as referred before) - we show the p-values and the percentage of information transfer (TE) between the two time series.

[10] https://www.rdocumentation.org/packages/BINCOR/versions/0.2.0/topics/cor_ts



(a) CS on release date and bugs by reported date

(b) CS on release date and bugs aggregated by release date

Fig. 4 Comparison of irregular series of CS and bugs

We did the very same studies to the issues, not filtered as bugs. Due to the lack of space we only show the results. We show all the data for this quastion in the appendix.

3.7.4 RQ4 - Impact of CS Intensity Evolution on "time to release"

We wanted to find statistical causality between CS density and delays in releasing a new version of an app ("time to release"). To this end, we made the same analysis (Linear regression, Granger Causality, and Transfer Entropy) without aggregating the data because the time series have the same observations (on the app release date).

For the CS 3 group's impact on time to release, we only used the transfer entropy (lags 1 to 4) and showed statistical significance and value on information transfer.

3.8 Function Selection and Parameter Estimation

Correlations We tested R standard correlation and time series correlation *cor_ts*, which can be applied when the time series are of different granularity and irregular. Because of this, we used *cor* in RQ1 and RQ2, and *cor_ts* in RQ3. The alternative of using cor_ts in RQ3 would be aggregating (binning) the bugs by release and using the standard correlation. When the dates axis in the time series are coincident (by release date), *cor_ts* will give the same results as *cor*.

Statistical Causality We used the standard linear regression (lm function in R) to measure causal inference in the same release. Linear regression can be used for prediction and causal analysis [11]. In the present study, we are more concerned with causal inference.

We used Granger causality (grangertest R function) in RQ2-RQ4, for lags 1 to 4. These lags mean that we are detecting Granger-causality from time-series X to time-series Y, but with a previous value of X, using only linear methods. This method should agree with linear regression if the X in the regression were from a previous release (the same lag). Granger Causality shows the significant value that allows us to conclude whether Granger causality exists.

On the other hand, Transfer Entropy measures the flow of information from a time-series X to time-series Y (in each direction), giving both the significance and the amount of information transferred, using linear and non-linear methods. This means it should agree with the Granger Causality values and add more information to the relation (the non-linear values). When measuring with Transfer entropy ("Rtransferentropy" function), we tested the parameter q. For q=1, Rényi entropy converges to Shannon entropy, and no areas of the time series are

[11] https://statisticalhorizons.com/prediction-vs-causation-in-regression-analysis

given more weight. With q less than 1, some areas of the distribution with less probability can be emphasized, and with q>1, some areas with more probability can be emphasized. We tried for values 0.5 to 0.9, 1.0 (Shannon), and 1.1, and found the best values that agree with Granger causality to be q=0.8.

Differences between Granger-causality and Transfer Entropy Granger-causality measures the impact of the past values of a time series in another, using linear models. Transfer Entropy can detect if there is information transfer from the past values of a "time series" to another, using liner and non-linear methods, and also quantify it.

We studied the times series with lags of 1 to 4 for both Granger causality and Transfer Entropy. In the study results, we present values with lags 1 and 2, but the values of the rest of the lags are in the replication package. As introduced, we are more interested in detecting causal inference between time series than in making prediction models. Nevertheless, we present all the results from the study (including the entropy transfer values from CS to issues, bugs, and "time to release" time series) in the appendixes up to lag2 and in the replication package up to lag 4 [12].

4 Results and Data Analysis

This section presents the results, data analysis, and findings of the research questions. We cannot represent all the correlations, and for the causality inference, all p-values and all the Transfer Entropy values (lack of space). So we opted for a *plus* notation (correlations) and *dot* notation(causalities), and the reader can consult the total values in the appendices or the replication package.

4.1 RQ1 - Evolution of CS in Web Apps on the Server and Client Sides

To answer this question, we studied individual CS timeseries evolution and correlate them within the same group. Then, we study the evolution and trends of the 3 groups of CS timeseries.

4.1.1 Individual Code Smell Density Timeseries Evolution and Correlation

We first analyzed the evolution of the various individual CS divided as server-side, client-side embed, and client JavaScript CS. The complete data is shown in the appendix and the replication package. Some similarities can be observed between the CS timeseries (CS), characterized by common patterns for increasing and decreasing periods. To check this similarity, we performed a correlation table between the CS timeseries of the same group, first as individual applications and after as an average.

The averages of the correlations show that some correlations in the same group prevail in most apps. However, we intend to check for the correlation between CS of the same groups but in one application at a time to check if it is possible to treat them as a group, and the answer is yes. This make possible evaluate the CS as a group for this and the next questions. The correlation data and graphs of all applications are shown in the appendix.

[12] available in https://github.com/studywebcs/data

4.1.2 Evolution of CS as a Group

Next, we evaluate the evolution of CS grouped.

Figure 5 represents the evolution of CS in the same group or programming scope, server-side programming, client-side, and client-side JavaScript programming. The CS densities for the different CS groups are all in the same order of magnitude and can be represented on the same scale. However, some peaks, especially in JavaScript code smells density evolution, are explained in the discussion section.

We had anticipated that all CS density (by size) would exhibit a downward trend over time, but this expectation has not been met, and some have even shown an increase. Nevertheless, there is some similarity in the evolution of the code smells belonging to the same group in some applications, which we will study in more detail in the following research question. By visual inspection, we can see this similarity in Fig. 5, in the client-embed smells and client-JavaScript smells, on *OpenCart*.

Fig. 5 Evolution of Code Smell groups in web apps (Y axis: CS density by LOC ; X axis: years)

Table 5 represents the linear trends and the average and standard deviation of the CS density values (by KLOC). Averages were calculated through all the releases of the applications, dividing per respective size (Lines of code) of the language or section studied. The linear trend was calculated by making a linear regression. It does not capture all the evolution of the CS density but serves to analyze the main tendency. The arrows mean the tendency of the time series. The arrows with the "equal" after mean that the increase or decrease is not so steep ("==" in *OpenEMR* means almost stable).

The density of server CS (PHP) increases for three apps, *phpMyAdmin*, *phpBB*, and *Dolibarr*. There are two peaks for *phpMyAdmin*, and there is refactoring, but the overall tendency still increases. In *phpBB*, there is a reduction in the end, but the shape is similar to an inverted U. In *Roundcube*, there is a steady increase. However, the step is not as steep as the other two apps (increases 50% in the time series). The server CS density increases or decreases in the other applications, but the step is not stiff, and some are almost stable. The density of code smells ranges from 8.5 to 20 CS per KLOC (if we do not count with *Kanboard*, and outlier with only 1.4 CS/KLOC).

Regarding client embed CS density, *phpMyAdmin*, *DokuWiki*, *OpenCart*, *MediaWiki*, *PrestaShop*, and to a lesser degree, *Vanilla* and *OpenEMR* all decrease their value. This is the type of evolution we were expecting (to make the quality increase). However, *phpMyAdmin*, *Dolibarr*, and to a lesser degree, *roundcube*, increase the client embed CS density. For *phpBB*, there is a stable trend. The client embed CS density values vary between 9 and 50 CS/KLOC).

For the JavaScript CS, the central tendency is increase, except for two considerable decreases (*OpenCart* and *Dolibarr*) and two small decreases (*phpMyAdmin* and *Roundcube*). The values vary from 29 to 159 CS/KLOC, if we consider *Kanboard* again an outlier with an excessive number of JavaScript CS, probably due to the tiny size of the application.

Summary for RQ1: The dominant trend, for most applications, in server-side CS density (by KLOC) is slowly decreasing. For client-side embed CS density, the main tendency is to decrease. However, for JavaScript CS density, the central tendency is to increase.

Table 5 Averages and trends of Code Smell densities (CS/KLOC)

app	server tendency	serve avg	server sd	client tendency	client avg	client sd	client js tendency	client js avg	client js sd
phpMyAdmin	↗	8.5	1.89	↘	20.14	9.72	↘=	159.36	480.52
DokuWiki	↘=	10.41	2.16	↘	17.37	8.34	↗	38.61	12.98
OpenCart	↗=	15.01	0.48	↘	12.16	6.39	↘	47.92	12.07
phpBB	↗	9.29	3.41	→	25.66	8.96	↗	29.39	23.25
phpPgAdmin	↘=	18.93	5.91	↗	52.37	31.78	↗=	19.11	6.43
MediaWiki	↘=	14.71	2.57	↘	13.7	6.07	↗=	28.59	4.18
PrestaShop	↗=	10.49	1.62	↘	25.89	10.92	↗=	40.1	9.85
Vanilla	↘=	19.98	2.14	↘=	8.89	1.62	↗	53.46	13.78
Dolibarr	↗	14.41	1.43	↗	27.29	9.01	↘	13.86	8.25
Roundcube	↗=	8.8	2.07	↗=	10.7 1.42	↘=	39.18	2.8	
OpenEMR	↗==	14.89	2.49	↘=	19.61	6.68	↗	29.74	6.88
Kanboard	↘=	1.4	0.69	↗	9.83	3.34	↗=	862.48	1406.87

Table 6 Correlation between Code Smells groups

correlation	phpMyAdmin	DokuWiki	OpenCart	phpBB	phpPgAdmin	MediaWiki	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
server,client		+		+		+		+	+	+		
server,client_js				+								
client,client_js	+		+									+

A plus sign indicates correlation > 0.3

4.2 RQ2 - Relationship between Server- and Client-side Code Smells

We studied the possible relationships between the evolution of CS groups and aimed to find statistical causality relations between variables in the same groups, with and without lags. All time series use "code smells density."

4.2.1 Correlation between Code Smell Groups Timeseries

Table 6 represents the correlations among the density of CS groups/types (in CS/KLOC). The plus signs represent positive correlations greater than 0.3, which is a moderate relation (Akoglu 2018; Ratner 2009; Kim et al. 2018). A more complete table with p-values can be found in the appendixes. Half of the applications exhibit a positive linear correlation between server-side and client-side CS. Furthermore, we find positive correlations between embed client CS and JavaScript client CS in a quarter of the applications. Of course, it is not necessary to have correlations between the evolution of code smells' density of different groups, but it can describe the team's expertise. We identify three groups, the third being applications with no correlations between CS time-series. Possible explanations for this phenomenon are put together in the discussion.

4.2.2 Causal Inference between Code Smell Groups' Timeseries

This section presents the results from the implementation of regression models to study if causal inferences exist. First, we use linear regression, which can be used to this end if criteria are met, as explained in Amanatidis and Chatzigeorgiou (2016) for metrics evolution, but here for Code Smells evolution. Next, we present the results from the implementation of dynamic regression models, Granger Causality, and Transfer Entropy between CS groups to study if cause-effect relations exist. This cause-effect relation is measured with lags. For example, Lag 1 means we are measuring the effect of release i in release $i+1$. Lag 2 means we are searching for the effect of release i in the release $i+2$. This "X->lag Y" can also be described as "Y" being one version ahead (lag).

Table 7 represents the resume of our measurements. Dots represent statistical significance in one web app; the solid dot means a statistical significance of 0.05, and the white dot means a statistical significance of 0.10. The column Linear Mod. shows the causal inference measured from linear regression. The columns GClag 1 (Granger Causality lag 1) and TE lag 1(transfer entropy lag 1) show the Granger causality/information transfer between the group on the left on the arrow and the one on the right. Lag 1 means the time series on the right is shifted one release (and Lag 2 two releases). While Granger Causality only captures the existence of causality in linear relations, transfer entropy also captures the non-linear

Table 7 Statistical causality between CS density groups using Linear regression, Granger causality (lag 1 and 2) and Transfer Entropy (lag 1 and 2)

CS	Linear Mod.	GClag1	GClag2	TElag1	TElag2
server =>client	●●●●●●●●●●	○	●○	●●○○○	●○
server =>client_js	●●●●●○	●		●●	●
client =>server	●●●●●●●●●●	●●●	●●○	●●●●●○○	●●●●
client =>client_js	●●●●●●●●●○	●●●○	●●○	●●●●○	●●●●
client_js =>server	●●●●●○	●○	●	●●●●●○	●●
client_js =>client	●●●●●●●●●○	●●○○○	●●○	●●●○○	●●

Black dots and white dots represent statistical significance of 0.05 and 0.10, respectively, in one application

relations (both the existence and the value). So we expect them to be in concordance, but TE should uncover more relations. The column GClag2 and TELag2 are the same, but for lag 2.

The column *Linear Mod.* of Table 7 represents the sum of the apps with statistical significance in the linear regressions among groups. The relations that have significance in most apps are: Server =>Client and Client =>Server; Client =>Client_js and Client_js =>Client. In the appendix, we show the separated values for all applications. For the applications *phpBB*, *PrestaShop*, *Vanilla*, and *Roundcube*, the regression models have statistical significance in the regression among all CS groups. This significance can indicate a strong relation among all the groups of CS, in these apps, in the same release. If, for example, the server-side CS increase, the other two CS groups follow the same pattern in that same release.

Analyzing Granger causality with lag 1, in 3 applications, we measured causal inference from CS of a release *i* to CS belonging to a release *i+1* from client to both server and client JavaScript. We found fewer causal relations using lag 2 (from two releases behind).

We can observe that there is a significant flow of information from the series with the lag 1 of the other groups (shift to the date of the next version), especially in Client->Server and Client_js->Server, followed by Client->Client_js. This result means that if there is a rise in Client CS density or JavaScript CS density in a release *i* of the app, the next release will be followed by a rise in Server-side CS density in half of the applications. Also, the rise in the Client CS density will impact the rise in JavaScript-only CS density. For lag 2 (two releases before), the Client CS density impacts both Server and JavaScript CS smells. The TE (value of transfer entropy from one to the other) and p-values are in the replication package. We studied GC and TE with up to 4 lags, but we only show two lags in the table for simplification. The remaining info is in the appendix and replication package. Both lags 3 and 4 on GC and TE have fewer apps with statistical significance.

These results mean a transference of information exists between the groups of CS, so they contribute to the behavior of the other time series. Possible explanations are put forward in the discussion.

Summary for RQ2: **Timeseries correlation analysis** - Found correlation between the server-side CS and client-side CS timeseries on half of the applications and between client CS and JavaScript client CS timeseries in a quarter of the applications.
Causal Inference between CS timeseries - With Lag1 (the preceding release), the most significant Transfer Entropy (TE) was from the client-side to server-side CS and from the client-side JavaScript to server-side CS, followed by the TE from client to client-side JavaScript. With Lag2 (from two releases before), the most notable TE was from the client to the client-side JavaScript and the server-side CS.

4.3 RQ3 - Impact of CS (server and client) on the Faults/reported Bugs of a Web App

This section presents the results for the impact of CS on faults/reported bugs in a web application. First, with CS time series individually (correlation and causality) and after, we evaluate causality from CS grouped to bugs. We used only ten apps for this study because 2 of the apps did not have enough data in the issue tracker.

4.3.1 Relationships between CS and Daily Reported Bugs

To find the relationships, we used the specialized time-series correlation (cor_ts) with CS density and the absolute number of bugs.

Table 8 represents the significant correlations between CS and daily bugs. Two of the apps (*OpenCart* and *phpBB*) only have 1 CS each that correlates to bugs. This result could be

Table 8 Time-series correlation (cor_ts) between CS and bugs (10 apps - positive corr. > 0.3

Code Smells	#correl.
(Excessive)CyclomaticComplexity	+ + + + +
(Excessive)NPathComplexity	+ + + + +
ExcessiveMethodLength	+ + + + +
ExcessiveClassLength	+ + + + +
ExcessiveParameterList	+ + ++
ExcessivePublicCount	+ + ++
TooManyFields	+ + + + ++
TooManyMethods	+ + +
TooManyPublicMethods	+ + ++
ExcessiveClassComplexity	+ + + + +
(Excessive)NumberOfChildren	+ + + + +
(Excessive)DepthOfInheritance	
(Excessive)CouplingBetweenObjects	++++
DevelopmentCodeFragment	+ + +
UnusedPrivateField	++
UnusedLocalVariable	+ + +
UnusedPrivateMethod	+ + ++
UnusedFormalParameter	+ + +
embed.JS	+ + ++
inline.JS	+ + ++
embed.CSS	+ + ++
inline.CSS	+ + + + +
css.in.JS	+ + +
css.in.JS..jquery	+ + ++
max.lines	+ + ++
max.lines.per.function	+ + ++
max.params	+ + + + +
(Excessive)complexity	+ + + + ++
max.depth	+ + + + +
max.nested.callbacks	+ + ++

related to delays in correcting bugs, closing the issues, longer time to release, or other issues. However, the remaining eight applications show a strong correlation between CS and bugs; in some CS, the correlation is present in almost all applications.

4.3.2 Causality Relationships Between CS and Bugs

To uncover causality relationships between CS and bugs, we employed Linear regression, Granger Causality, and Transfer Entropy, using CS density timeseries. For these causality inference studies, we had to aggregate the bugs, as we described in the subsection *Methodology for each RQ*.

Table 9 Statistical causality from CS density to Bugs relations (10 apps) using Linear regression, Granger causality (lag 1 and 2) and Transfer Entropy (lag 1 and 2)

CS	LM	GClag1	GClag2	TElag1	TElag2
(Ex.)CyclomaticComplexity	●●●	●●○	●○○	●●●○○○	●○
(Ex.)NPathComplexity	●●●	●●●	●●○	●●●○○○	●●
ExcessiveMethodLength	●●●●●	●●●●	●●○	●●●●○	●○
ExcessiveClassLength	●●●	●●●	●●	●●●●●	○
ExcessiveParameterList	●●●	●○	●●	●●●●○	●●●●
ExcessivePublicCount	●●○	●●	○	●●●○	●
TooManyFields	●●●○	●○	●●	●●●	○
TooManyMethods	●●●○○	●○○○		●●●●○	●●
TooManyPublicMethods	●●●●	●●○○	●●	●●○	●○
ExcessiveClassComplexity	●●●●●	●●●○○	●●○	●●●●●○	●●○○○
(Ex.)NumberOfChildren	●●●●	●●●	●●●	●●●●●	●○
(Ex.)DepthOfInheritance				●	●
(Ex.)CouplingBetweenObjects	●●○	●○○	●	●●●●●●●	●
DevelopmentCodeFragment	●○	●●	●	●●●●●	●
UnusedPrivateField	●●●	●○	●	●●●●○	●●●○○
UnusedLocalVariable	●●●●	●●○	●	●●○	○
UnusedPrivateMethod	●●○○○	●○		●●●○○	●
UnusedFormalParameter	●●●○	●	●○○	○○○	●●●
embed.JS	●●●○	●●●○	●●	●●●●●	○
inline.JS	●●○	●●○	●●○	●●●●○	●●●
embed.CSS	●●○	●●	●●○	●●●●○	●●
inline.CSS	●●●●	●●●	●●○	●●●●●○	●●●○
css.in.JS	●○○○	●●○	○	●●●●○	○
css.in.JS..jquery	●●●	●●	●●●	●●●○	●○
max.lines	●●●	●●●	●●	●●●○○○	●
max.lines.per.function	●●●	●●	●●	●●	●●
max.params	●○○	●○	●●	●●●●●●●○	●●●●●
(Ex.)complexity	●●○	●●	●●○	●●●●○	●●○
max.depth	●		●○	●●●●○○	●●●●●○
max.nested.callbacks	●●	●●○	●	●●●○○	●●

Black dots and white dots represent statistical significance of 0.05 and 0.10, respectively, in one application

Table 9 shows our measurements. The column LM is the **linear regression**, where the various CS densities are the independent variables and bugs are the dependent variable. Almost all the CS time series have a causal inference (measured with linear regression) with the bugs, and the ones in more apps are *ExcessiveMethodLength*, *ExcessiveClassComplexity*, *TooManyPublicMethods*, *UnusedLocalVariable*, from the server-side, and inline.CSS, from the client-side. For 2 of then apps, *Roundcube* and *OpenCart*, there are no significant relations.

For the causality of CS in previous releases to bugs, we have **Granger causality** lag1 and lag2 (CS in the previous release and two releases before). The column "GC1lag" shows Granger causality from CS density (lag 1) to bugs, where we can observe several G-causality relations, being the top with *ExcessiveMethodLength*. The column "GClag2" shows the Granger causality from CS from 2 releases before to the bugs in the current release. We can observe some G-causality relations, but less than with 1 lag. We also studied Granger causality for lag 3 and 4; the number decreases, except for the code smell *"max.nested.callbacks"*, which increases. Again, for the same two apps, *Roundcube* and *OpenCart*, there is no significant relations.

The column TE 1lag shows the **Transfer Entropy** from the CS in the previous release to the bugs. Column TE 2lag shows the Transfer Entropy from the CS from 2 releases before the bug's actual release. As we know, TE captures both the linear and non-linear information transfers, so we expect more than in the Granger causality columns. In TE 1lag, almost all the CS significantly impact the bug's evolution, as shown by the "bullets" in the column. However, the CS with higher information transfer to the bugs are: *CouplingBetweenObjects, ExcessiveMethodLength, ExcessiveClassLength, ExcessiveParameterList, TooManyMethods, ExcessiveClassComplexity, NumberOfChildren, DevelopmentCodeFragment, UnusedPrivateField*; all client CS but *"css.in.JS..jquery"*; and in JavaScript CS, *max.params, complexity, max.depth*. In TE lag 2, almost all decrease the number of apps, but the smell "max.depth" is 5 in lag2, lag3, and lag4. The max.nested.callbacks increases in lag 4 also. All the apps have values; however, the apps with fewer values are *PrestaShop* and *roundcube* with just 3 CS showing significant TE.

As introduced , we are more interested in the existence of causality relations from individual CS than the values themselves. However, as an example, we show the interval ratio in the contribution (TE) of some significant CS to bugs of the 3 areas, in the apps that TE was statistical significant with an error of 0.05: *CouplingBetweenObjects* 10% to 52%, *ExcessiveMethodLength* 11% to 51%, *ExcessiveClassLength* 11% to 52%, *ExcessiveParameterList* 11% to 52%; *embed.JS* 13% to 57%, *embed.CSS* 12% to 20%, *css.in.JS* 19% to 48%; *max.lines* 15% to 42%, *max.params* 14% 53%. All percentage values in rep. package.

We also tested the TE from bugs to CS (inverse Transfer Entropy), but we found no values worth referencing. This result indicates that information transfer goes from the CS to the bugs and not from the bugs to the CS (once more, the data is online, in the replications package).

One CS, *(Excessive)DepthOfInheritance*, is only present in residual numbers in 3 applications, explaining the empty line in the tables.

4.3.3 Causality Relationships between CS Groups and Bugs

Table 10 represents statistical significance and weighted amount of information transfer from CS groups to bugs. In half of the applications, server-side CS statistically caused bugs, measured with a significance of 0.05. If we expand the significance to 0.10, in 60% of the apps, JavaScript CS can statistically cause bugs, while the other client-side CS cause this in 40% of the apps. In lag2 and other lags, all start to decrease. The weighted average of

Table 10 Statistical causality from CS density grouped to Bugs (10 apps) using Transfer Entropy (lag 1 to 4)

CS	TElag1	TElag1%	TElag2	TElag2%	TElag3	TElag3%	TElag4	TElag4%
server	●●●●●	18%	●●○	10%	●●	6%	●●	4%
client	●●●○	9%	●●●	15%	●●●	16%	●●●	7%
client_js	●●●○○○	17%	●●	6%	●●	7%	●●	3%

Black dots and white dots represent statistical significance of 0.05 and 0.10, respectively, in one application. Columns with % have the weighted information transfer values

information transfer measured with TE in lag1 from server CS to bugs is around 18%, from embed client CS 9%, and from client JavaScript CS is around 17%.

Summary for RQ3: We detected significant correlations between all the CS densities and bugs timeseries in almost all ten applications evaluated except 2. We find statistical causality from almost all the CS densities to bugs timeseries, being the top PHP *(Excessive)CouplingBetweenObjects* and JavaScript *(Excessive)max.params*. The TE from bugs to CS density timeseries (inverse TE) is almost non-existent. There is significant TE from CS densities as groups (Server, Client, and JavaScript) to bugs timeseries in Lag 1, decreasing in the subsequent lags.

4.4 RQ4 - Impact of CS (server and client) on the Time to Release of a Web App

We analyzed the causality between CS density and delays in releasing a new version of an app (time to release). To this end, we used the same approach as before: Linear regression, Granger Causality, and Transfer Entropy. No aggregation was needed because "time to release" is measured in the app's release date.

4.4.1 Causality Relationships between Individual CS and Time to Release

Table 11 represents the LM, Granger-causality, and Transfer entropy of each code smell to "time to release." Each dot represents a web app with statistical significance (0.05 black, 0.10 white) in the statistic performed in the column.

In the **Linear regression** (column LM - from CS density to 'Time to release'), which measures the impact of CS in the same release - as causal inference - we found several regressions with statistical significance. However, in 2 of the apps (*DokuWiki* and *roundcube*) we found 0, and in *phpPgAdmin*, we found just 1, so the maximum possible of apps (and dots) would be 9. These three applications behave differently, i.e., the CS do not impact in 'Time to release'. However, for most apps, CS impacts 'Time to release' in the same release of CS. The CS with the most impact are *ExcessiveMethodLength , (Excessive)CyclomaticComplexity, (Excessive)NPathComplexity, ExcessiveClassComplexity, TooManyMethods , UnusedFormalParameter* from server-side; *embed.JS, inline.CSS, inline.JS, embed.CSS* from client-side; *max.lines.per.function and max.lines* from JavaScript.

The column 'GClag1' represents the linear impact of CS from the previous release in 'Time to release' of the current release. We found some Granger causalities, but this number is lower than TE. However, we found some values with statistical significance, being the max.lines in JavaScript the top one. The column 'GClag2' represents the Granger causality values for lag 2, i.e., for CS from 2 versions before the current time to release. With two

Table 11 Statistical causality from CS density to *Time to release* relations using Linear regression, Granger causality (lag 1 and 2) and Transfer Entropy (lag 1 and 2)

CS	LM	CGlag1	CGlag2	TElag1	TElag2
(Ex.)CyclomaticComplexity	●●●●●●	●●●○○	●●	●●●●○○○	●○○
(Ex.)NPathComplexity	●●●●●●	●●●○○	●●	●●●●○○	●●○○
ExcessiveMethodLength	●●●●●●●	●●●●○○	●●●	●●●●○○○	●●●
ExcessiveClassLength	●●●●	●●●●○○	●●	●●●●○○	●●
ExcessiveParameterList	●●●●	●●●○○	●●	●●●●●○	●●○
ExcessivePublicCount	●●●○○	●○○	●●	●●○○	●
TooManyFields	●●○	●○	●●	●●●●○	○
TooManyMethods	●●●●●○	●●●●○○	●●●○	●●●●●○	●●○○
TooManyPublicMethods	●●●○	●●●○	●●	●●●●●	●●○
ExcessiveClassComplexity	●●●●●●	●●●○	●●●	●●●●●○○○	●●
(Ex.)NumberOfChildren	●●○	●●●○	●●○○	●●●●●○○	●
(Ex.)DepthOfInheritance	●			○	
(Ex.)CouplingBetweenObjects	●●●	●●●○○	●●●○	●●●●	●●
DevelopmentCodeFragment	●●●●○	●●●○	●●	●●○○	
UnusedPrivateField	●●○	●●○	●●○	●●●●○○	
UnusedLocalVariable	●●●●○	●●●○	●●●●	●●●○○○	○
UnusedPrivateMethod	●●●●	●●●●○	●●○○	●●●●●●●	●●
UnusedFormalParameter	●●●●●○	●●●●○○	●●●●	●●●●●●	●●●●
embed.JS	●●●●●●	●●●●○○	●●●○	●●●●○○○	●●●○○
inline.JS	●●●●●○	●●●○	●	●●●●●●●	●●○○
embed.CSS	●●●●●○	●●●	●●○	●●●●●●○	●○
inline.CSS	●●●●●●○	●●●●○	●●○	●●●●●●●●○	●●●●○○○
css.in.JS	●●●	●●	●●	●●●●●○	●●○○
css.in.JS..jquery	●●○	●●●○○	●○	●●●●○	●○
max.lines	●●●●○○○	●●●●●○	●●●●	●●●○○○○	●●●●
max.lines.per.function	●●●●●●○	●●●○	●●○○	●●○	●○
max.params	○	●●○	●●○	●●●●●●○○	●●
(Ex.)complexity	●●○	●○○	●●○	●●●●○○	●●○○
max.depth	●○	○	●●○	●●●●●●○○○	●●●○○
max.nested.callbacks	●●●○○	●●●○○	●●●○○	●●●●●●●	●●●●

Black dots and white dots represent statistical significance of 0.05 and 0.10, respectively, in one application

lags, we find more causality values than in lag 1. The top ones are *UnusedLocalVariable, UnusedFormalParameter*, and *max.lines* from JavaScript. A probable cause: this is related to time spent in program comprehension, which takes longer.

The column 'TElag1' represents the Transfer entropy from CS to TTR(time to release) with 1 lag. The TE measures more than linear methods, so we expected to have more values here. And indeed, we have, being the top ones: *UnusedPrivateMethod, UnusedFormalParameter, TooManyMethods, ExcessiveParameterList, TooManyPublicMethod, ExcessiveClassComplexity , (Excessive)NumberOfChildren, (Excessive)CouplingBetweenObjects, UnusedPrivateField* from the server-side of the app's code; all from the client embed CS, but one with slightly less significance: *css.in.JS..jquery*; and *max.nested.callbacks, max.params,*

max.depth from Javascript CS. The app *DokuWiki* does not have any CS with statistical significance. For TE CS->TTR with lag 2, the values decrease, but some still have some significance and even rise: *UnusedFormalParameter* (server), *inline.CSS* (client), *max.lines* and *max.nested.callbacks* in JavaScript. In the replication package and the appendix, we have these values up to lag 4 in detail (Granger causality and Transfer Entropy). The CS with almost no values is the *(Excessive)DepthOfInheritance* because only three apps have this CS, which does not vary much.

Inverse TE: We also studied inverse TE, the impact from **Time to release** to CS in the next version (lag1). None has a particular impact except *ExcessiveMethodLength* that has statistical significance in 3 apps: *Roundcube*, *OpenEMR*, and *Kanboard*. These values mean that if the 'Time to release' - the time between releases - increases for these apps, some class methods of the subsequent releases can have excessive lines of code. Specifically, if the TTR increases several days in these three apps, there is a tendency to write more code in the methods, making them too long (which raises a CS) instead of dividing them into two methods.

4.4.2 Causality Relationships between CS Groups and Time to Release

Table 12 represents statistical significance and weighted amount of information transfer from CS groups to "Time to release." In half of the applications, server-side CS statistically caused a delay in the release date, measured with a significance of 0.05. If we expand the significance to 0.10, in 60% of the apps, JavaScript CS can statistically cause delays in "Time to release," while the other client-side CS cause this in 40% of the apps. In lag2 and other lags, all start to decrease, except for the client-side embed CS which still statistically causes delays in TTR in 60% of the apps. The average of information transfer measured with TE in lag1 from server CS to TTL is around 14%, from embed client CS 17% and from client JavaScript CS is around 10%. In lag2, it decreases except for the client-side CS. In lag3 and lag4 it decreases.

Summary for RQ4: There are causal relations from individual CS densities to **'Time to release'** timeseries of the apps, especially in the same release or with lag1, where the top CS are the ones from the client-side (both embed and JavaScript). With lag 2, the values overall decrease, but *inline.CSS* still presents a high value. There is significant TE from all CS as groups (server, client, and JavaScript) to *time to release* timeseries in Lag 1, and it decreases in the other lags, except for lag2:client CS.

Table 12 Statistical causality from CS density grouped to Time to Release (12 apps) using Transfer Entropy (lag 1 to 4)

CS	TElag1	TElag1%	TElag2	TElag4%	TElag3	TElag3%	TElag4	TElag4%
server	●●●●●○○	14%	●○	6%	●○	5%	●	2%
client	●●●●●○○○○	17%	●●●●○○	18%	●●●	9%	●●●●	8%
client_js	●●○○○○	10%	●●	5%	●●●	9%	●○	4%

Black dots and white dots represent statistical significance of 0.05 and 0.10, respectively, in one application. Columns with % have the weighted information transfer values

5 Discussion

The set of studies presents the following contributions in the methodology and data: an evolution and statistical inference study with both server CS and client CS in web apps; the use of (irregular) time-series specialized correlations (cor_ts); inferring statistical causality with Transfer Entropy, that can measure the flow of information with non-linear relations and comparing it with linear relations (with Granger-causality and Linear Regressions). In addition, we provide a tool to extract the embed class of code smells on the client side. We excluded third-party folders in the server- and the client-side CS and metrics (for some apps, this makes up around half of the code). Finally, we provide datasets with server-side and client-side code smells (30 in total) for the 12 applications' consecutive releases and the related metrics.

Next, we discuss the RQs' findings separately and their implications. To explain some findings (as also select the CS with no ambiguity problems, as previously described), we assembled a group of 3 experienced developers with more than ten years of experience in web development, two software engineering professors, and a web development professor.

Possible Comparisons to Desktop Studies As refereed, PHP web apps have server-side and client-side code in the same codebase and sometimes in the same file, and the server code runs in the web server and the client code runs in the browser (sometimes the same file runs twice). This dual processing is entirely different from desktop apps where the code runs on one platform, and there is no difference in client-side and server-side code. The results of RQ3 to RQ4 could be compared to desktop counterparts, but only the individual CS and only from the server-side code.

On the other hand, in desktop CS studies, Granger-causality was used differently, for example, to verify that method-level code smells may be the root cause for the introduction of class-level smells (Palomba et al. 2018); causality analysis also revealed that design smells cause architecture smells (Sharma et al. 2020), so it is not comparable. Lastly, entropy was used to predict CS from existing CS (Gupta et al. 2018), a different approach than ours.

5.1 RQ1 - Evolution of CS in Web Apps on the Server and Client Sides

Tendency of Individual CS within the Same Group The CS density evolution is more important than the absolute number of CS because the application, during its life, can be refactored, and the size can increase or decrease (removal of code). We already know that in some apps, we can observe the removal of server CS during the lifespan, while in others, the CS never get removed (Rio and e Abreu 2019).

There are similarities between the individual CS of the same type (Server, Client embed, Client JavaScript), so we could examine the CS individually and grouped. This is why we made the correlation between CS of the same group, to study the possibility of further studies in CS groups. Consequently, our investigation extended to each distinct CS and their collective behavior within their respective groups. The motive behind this approach was to explore any existing correlations within the same CS group, thereby paving the way for studies on groups of CS.

CS Groups The most observed trend in server-side CS is "slowly decreasing." In a previous study (Rio and e Abreu 2023), we found that the server code smells density (by Logical lines of code - or Effective lines of code) is almost stable. The values are almost the same but

different because the metric is different. In this study, we use PHP "lines of code" for the size - the reason for this is explained in the study design. We omitted folders from third parties (external code) in both studies. The server-side CS result aligns more with the Java language results from desktop apps.

The most observed tendency for client embed Code Smells' density is to decrease. This is because the client "embed" CS group comprehends mainly design smells that concern the mixing of several languages in the same file. So it makes sense that it decreases as the developers learn how to separate the concerns (HTML for content, CS for formatting, and JavaScript for client programming), which should be in separate files whenever possible (Gilbert and Gilbert 2019; Nguyen et al. 2012; Fard and Mesbah 2013).

Regarding JavaScript CS density, the central tendency is to increase. A possible explanation for this, given by interviews with developers with more than ten years of experience, is the moving of server-side functionality to client-side code. We plan to expand these interviews and publish them.

5.2 RQ2 - Relationship between Server- and Client-side Code Smell Evolution

Depending on the web application team expertise and knowledge, the development is made with just one team, two teams (for the server code and client code), and often a third team for the client programming, JavaScript. Furthermore, when they have just one team, they can specialize more in server development or client development; thus, there will be a difference in the quality of the code. On the other hand, when there is more than one team, the development quality can be at the same level, but often not. On the other hand, distributed applications or systems often have different development teams. Due to this diversity, we expected to find different relations between the development quality and, therefore, between groups of CS from app to app.

According to the correlation and correlation type, we divided the applications into three groups, and according to our group of specialists, we can learn the group constitution possibilities:

- 6 apps with correlation in server-side and client-side CS: one team or various teams but with similar server- and client-side code knowledge
- 3 apps with correlation in the two client groups (both "embed" and JavaScript) CS: this means that the server development quality related to CS is different in the server side, from the two different teams with different know-how in server and client-side or a small team specializing in one part (server or client code) more than the other.
- 1 app - no correlation - almost the same as before, except that all developments are done with different levels of quality - probably three developer groups.

Figure 6 shows the transference of information from time series A to time series B, with the label $TE_{A->B}$ (Transfer Entropy from A to B).

While we can measure the time-series correlations, we can not measure the team's constitution from projects from GitHub, where all the developers commit to the same project. Since most PHP applications are complete applications with server and client-side code, the code is in just one codebase on control version systems (and so the links to developers), we would need different studies to assess the constitutions on teams and the expertise of individual developers on the client-side or server-side code. However, measuring relations and causalities between groups of CS makes an excellent tool to characterize the team's quality-wise behavior in the server and client parts of the code.

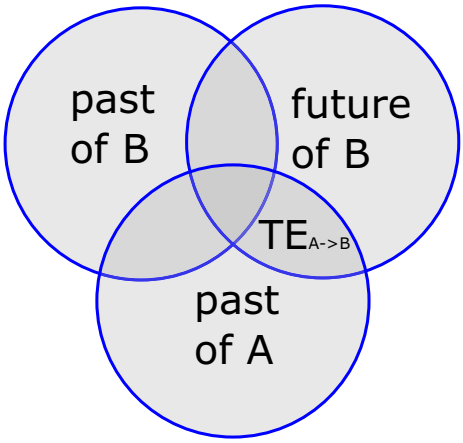


Fig. 6 Venn diagram showing the transfer entropy of Timeseries A to Timeseries B

Regarding the Transfer Entropy measurements: the quality of the code, or lack of it, measured with CS density proxies, propagates from client code to server code and JavaScript in half of the applications. This result is consistent with the typical development pipeline of monolithic web applications, in which client-side development is typically done before server-side development (as referred before).

5.3 RQ3 - Relation between CS and Bugs

As mentioned before, CS alone are not necessarily responsible for increasing bugs numbers. However, they contribute to their appearance, as shown by our measurements.

Figure 7 shows the transference of information from CS time series to the bugs time series. There is significant TE from CS from the previous release (lag1) to Bugs, but for CS from two releases before (lag2), these values decrease, which can indicate that the bugs are quickly corrected in the same release.

However, there are some exceptions. Some of the server-side CS that still have an impact with lag2 are the CS that makes the code difficult to read. As an example, the *UnusedFormalParameter*, in PHP, is possible to initialize a parameter in a method: `mtd1($a, $b, $c=initial)`. This method can be called as `$a->mtd1(a,b)`, thus possibly making developers wonder why there is a third parameter not used. For the client-side, the prevailing

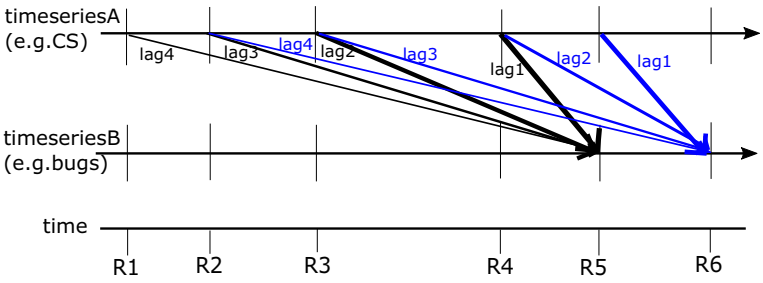


Fig. 7 Previous releases in CS timeseries impact next releases on bugs timeseries

ones in lag2 are also the problematic CS that hinder readability. However, the phenomena do not happen so clearly in the "issues" impact studies.

We also calculate (in the replication package and appendix) the value of the contribution of individual CS time series to the bugs. However, we already knew this was not the only factor contributing to bugs. However, in a few individual CS, we found some large values (around 50%), but usually, these values are small. Nevertheless, our study aimed only to find causality inference from CS to bugs, which we found, as the results show.

When analyzed in a group, the server-side CS contribute more to bugs but also the JavaScript client ones. This number slowly decreases when measuring older CS. The group of client-side embed CS still impacts up to 3 releases before. This can be explained by the difficulty of reading and modifying the embed client code. After some releases, a learning factor about code knowledge establishes itself (according to our specialist group). In general, the information transfer from the CS to the bugs is more than 15% (lag1), 10% (lag2), and then decreases.

5.4 RQ4-Relation between CS and Time to Release

The CS can be the cause directly or because they jeopardize "program comprehension," which, in the long term, causes delays. We found these contribution values for the delays in the releases in almost all individual CS, especially with lag 1, but also in lag 2 for the CS that impact readability directly.

Analyzing the CS grouped, the results indicate that if the CS increases (especially for some of them in the individual CS), the web app release date will have a delay in the next release. When the CS are grouped, we can infer the same conclusion.

5.5 Implications for Researchers

The number of studies on web applications is still tiny compared to desktop applications, especially in Java. The main difference between web and desktop applications is that they get processed on at least two platforms: the web server and the browser. Even serverless applications are processed on the server; the difference is that they implement a web server in the server-side code (usually a web service). There are even more differences, but this difference alone implies the necessity for more studies.

Because of the diversity of the languages, it is necessary to complete the catalog or list of web code smells further. Previous studies in web apps just used the server- or the client-side.

There is a gap in studies on the evolution of CS on both server- and client-side of web applications and their relations. This type of study is a crucial investigation line that should have more studies.

Another important research topic is the implications to issues, bugs, and "time to release." We show that almost all the code smells from the three groups are responsible for these variables in the web app evolution, and the ones from the client-side are not to be neglected (sometimes their contribution is even more significant).

Another line of investigation is to build a complete prediction model with other variables. Some studies aimed this for the desktop word (Java) with only CS similar to the server-side but on a file/class basis and not an evolutionary perspective (with time series).

Concerning web application code, 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 release. Therefore, if we do not perform this step, some analyzed code comes from other programs. We have performed this step since (Rio and e Abreu 2019). Therefore, we must omit third-party folders from the investigation.

Another concern for investigators is that when measuring size/lines in PHP, not all programs will count only PHP code lines inside PHP files. In a previous study on server-side CS only, we used *phpLOC* that counts HTML lines in PHP files as PHP code for the LOC. One way to avoid this problem was to use LLOC (logical lines of code). In the present study, we also had to consider this, but we used LOC measured with CLOC but with a SLOCCount plugin to count only the PHP lines inside PHP tags ("<? ?>" and ("<?php ?>").

Researchers can use our tool to detect "embed CS" (we can make a version without database requirements, outputting to .csv, if needed).

Researchers can also use our dataset with three groups of CS, server-side, embed client-side, and JavaScript client-side, to replicate the study or build other studies.

5.6 Implications for Practitioners

Practitioners should avoid code smells, even if they did not implicate a rise in issues and bugs or "time to release," but because they make the code harder to read, among other problems.

Due to the findings in causal inference between types of CS (RQ2), developers should correct the templates (HTML, CSS, and JavaScript) and JavaScript code, both inside HTML and in external .js files, before implementing server code (in the study case, PHP code, but it could be code in C#, Java, ruby, python, or even server-side Javascript with nodejs runtime).

After the findings in causal inference from web CS to bugs, issues, and "time to release," as a general rule, developers should try to avoid CS; they increase the number of bugs and even the number of issues and time to release. However, if the time to refactor is scarce, **developers can prioritize CS removal** that have more impact on bugs or "time to release" after the results we found.

Another implication for the developers would be to try to separate client code, especially CSS inside HTML and JavaScript inside HTML. We found this group of embed CS to be a casual inference in bugs and delays to the releases. So, always aim to use external ".css" files and ".js" files in monolithic web apps. As a plus of implementing this separation, it would be possible to simultaneously develop the HTML, CSS, and JavaScript files by different developers. Of course, this advice is different for systems with micro-frontends (a front-end made of parts that glue together), as the JS/HTML/CSS code typically is together in the same module, but this is out of our scope.

5.7 Implications for Educators

Most Software Engineering or Software Quality courses taught at Universities already have Code Smells in their curricula. However, they use mainly the Java language for desktops. On the other hand, courses in Web development tend to leave software engineering issues like CS out of the syllabus, especially if it is only one course (in some university Computer Science related degrees, there are two "web development" courses). Because web programming involves many technologies simultaneously, this and other SE concerns are often omitted, and this makes sense because CS are absent from general programming courses. However, because of web ubiquity (lately, universities found that significant percentages of CS students do some form of web development after graduation), adding web development quality concerns, like CS, to the Software Eng. syllabus makes sense.

5.8 Threats to Validity

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*, where we detected 18 CS. We could compare the detection with other tools for PHP, but most of them are based on *PHPMD*. We used *ESLint* to detect the 6 JavaScript CS; we rely on their accuracy. We built a tool to detect the client CS, test it by manually inspection, and perform very well in our tests - no false positives. However, we would like some feedback on the tool. We could expand this study to consider even more CS. When filtering the bugs from the issues with the devised filter, we had two results: for the applications that did not label the bugs, the results were around 93-95%; for the applications that labeled the bugs, the results were close to 100%. When studying Granger-causality and TE with issues and bugs, we had to aggregate them by the sum in intervals similar to the releases. This aggregation could influence the result. However, when we measured the correlation with *cor_ts* (time-series correlation), we did not aggregate (CS by release date, Issues, and Bugs by day). The "time to release" was already in the same irregular interval as the CS.

There should be a balance between statistical significance with the magnitude of effect, the quality of the study, and findings from other studies (Feise 2002). A p-value adjustment is necessary when performing multiple tests of significance where only one significant result will lead to the rejection of an overall hypothesis, or sequential testing during which significance calculations are performed a number of times during the A/B test until a decision boundary is reached [13]. False discovery rate control (Benjamini and Hochberg 1995) is substituting the less power (Verhoeven et al. 2005; Glickman et al. 2014). However, for studies with hundreds or more comparisons, these methods are not recommended (Bender and Lange 2001). Although we should not implement the FDR (the conditions are not met, and there is an excessive number of comparisons), we experimented with FDR in the three groups of CS, and an application at a time, in the Bugs Transfer Entropy study and did not find differences.

Threats to internal validity concern external factors we did not consider that could affect the investigated variables and relations. We can say that *PHPMD* allows us to change the metrics thresholds of some CS (some do not come from metrics), but we worked with the default values for comparing between apps. These values can, however, be questioned for different apps. We can say the same for JavaScript CS measured with ESLint. However, for the 'embed client CS,' the CS is there or not, so this problem does not exist.

Metrics that can influence the results: We tried to make the study independent of metrics, so we worked with CS density by lines of code. It is possible to work by class (as studies in desktop apps do), but HTML or CSS do not have classes, so the normalization had to be the lines of code. Other metrics worth exploring would be the total size or longevity of the application, which can be used in a future study. Some authors use the metric "Lines of code affected by CS." This metric is subjective in cases like "Long Method," in which the number of lines to count as affected is uncertain. Additionally, if a line has 2 or 3 CS, it just counts as one affected line.

It would also be desirable to have more JavaScript code smells. We plan to do this in the future. Some of the CS in JavaScript and PHP were considered ambiguous by the teams of specialists, so we did not try to detect them.

Threats to conclusion validity concern the relation between the treatment and the outcome. CS are often considered by absolute number or normalized by *LOC*, or other metric such as by class - that we could not do here, because HTML and CSS do not have classes. However,

[13] https://www.analytics-toolkit.com/glossary/p-value-adjustment/

our experiments have shown that the CS normalised by *LOC* gives a better understanding of the effect of the CS on outcome variables, because its the density of the CS system wide. Another reason that the CS are system wide is because a CS in file1 can impact a CS or a bug in file 2, if we just consider within a class or a file we would loose those relations. To calculate the Transfer Entropy, we made several tests to find the correct parameters, to put it in concordance with the Granger causality. After the tests, we used the parameter q=0.8 for all applications; however, this could be analyzed for all applications separately.

Threats to external validity concern the generalization of results. We recognize that having just 12 web apps may not be enough for generalization's sake. Also, the apps are different in evolving, resolving issues and bugs, and removing CS. However, most evolution studies consider just a small number of apps because it is very computation-intensive to collect all CS from consecutive app releases and relate them with other outcome variables in consecutive releases.

6 Conclusions

We studied the evolution and interaction of 18 server-side CS and 12 client-side CS (6 in JavaScript) in 12 widely used PHP web apps over many years. The CS collection resulted in datasets of 18 million server, 0.9 million client embed and 1.27 million JavaScript CS. Furthermore, we measured the impact between them and their impact on web app reported issues, defects/bugs, and delays/time to release the web applications. We presented an initial catalog of CS for web apps (aka web smells), both from the server and client-side, for applications built with PHP language on the server-side. In the scope of the investigation, we also developed a client embed CS extractor (*eextractor*), available to download.

We found that most of the CS in the same group have the same tendency in evolution. The primary trends for CS in most applications are: server-side: slowly decrease; client-embed: decrease; client-JavaScript: increase. The most significant TE with lag1 between CS groups happens from both CS client-side groups to server-side, followed by client-embed to client-JavaScript. With lag2, the TE from client-embed to client-JavaScript and server-side CS are the most significant.

We found a transference of information (with Transfer Entropy methods) between the time series of almost all the individual CS and the time series of the bugs, implying a causal inference from CS to bugs. In conclusion, almost all CS contribute to bugs' evolution, especially in lag1; The inverse TE, from bugs to CS density, is practically non-existent. We also found statistical evidence of causal inference using the various methods between CS and time-to-release for almost all the individual CS. Developers should remove all CS, but if time is scarce, they can prioritize its removal based on the study. We also found that individual client-side CS contribute more to issues' density evolution, but CS from the server-side code also impact - this result is only on the appendix.

When analyzing the three groups of CS, the results are similar: from CS to issues, values of entropy transfer are between 9% and 14% (highest JavaScript CS) - values only in the appendix; from CS to bugs, the values of entropy transfer are from 9% (client embed) to 17%,18% for JavaScript and server CS respectively; and from CS to time-to-release the highest value is obtained with the client embed CS.

We can conclude that although all CS groups contribute to issues, bugs and TTR, JavaScript CS can cause more issues, (server)PHP and JavaScript(from the client side) can cause more bugs, and the time to release (delays) is more impacted with the density of client embed CS.

These results and percentages concord with bug prediction models for desktop apps (Palomba et al. 2019). However, we invite further independent confirmation of this work or similar studies. It is essential to study the aspects of code that developers can change or avoid, including code smells.

6.1 Future Work

Regarding future work, we would like to increase the catalog of web smells and build detection tools for these CS. A CS candidate could be "embed HTML in PHP files." However, one of the strengths of PHP is that it doubles as a server-side language and as a template language (hence the variables with $ to allow putting them inside strings - this is a BASH, PERL heritage). So, this is a topic to be discussed. Furthermore, we want to increase the number of applications and CS studied, provided that more computing power is available. Collecting and analyzing CS across many consecutive releases and for server-side and client-side (30 CS in total) is a computationally heavy task.

Another research opportunity would be to elaborate the study in distributed apps (web apps/systems compromising several applications, both frontend/backend and with microservice architecture). However, because these applications no longer use only one code base, the RQ2 -Interactions of server and client code smells would not be possible to study for those systems - at least in the same way - mainly because the releases of client and server (only one or various micro-services) could be not coincident in dates.

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

Funding Open access funding provided by FCT|FCCN (b-on).

Data Availability For replication purposes, we provide the collected dataset, available at https://github.com/studywebcs/data and https://zenodo.org/doi/10.5281/zenodo.10838018, where we also have all the values not shown in the article due to lack of space.

Declarations

Conflicts of interest The authors declare that they have no conflict of interest.

References

Akoglu H (2018) User's guide to correlation coefficients. Turk J Emerg Med 18(3):91–93. https://doi.org/10.1016/j.tjem.2018.08.001

Amanatidis T, Chatzigeorgiou A (2016) Studying the evolution of PHP web applications. Inf Softw Technol 72:48–67. https://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. https://doi.org/10.1016/j.infsof.2017.05.004

Antoniol G, Ayari K, Penta MD, Khomh F, Guéhéneuc YG (2008) Is it a bug or an enhancement? A text-based approach to classify change requests. In: Proceedings of the 2008 conference of the center for advanced studies on collaborative research meeting of minds - CASCON '08. ACM Press. https://doi.org/10.1145/1463788.1463819

Ayari K, Meshkinfam P, Antoniol G, Penta MD (2007) Threats on building models from CVS and Bugzilla repositories. In: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research - CASCON '07. ACM Press. https://doi.org/10.1145/1321211.1321234

Bahadori MT, Liu Y (2012) Granger Causality Analysis in Irregular Time Series. In: Proceedings of the 2012 SIAM international conference on data mining. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9781611972825.57

Bán D, Ferenc R (2014) Recognizing antipatterns and analyzing their effects on software maintainability. In: Computational science and its applications – ICCSA 2014, pp 337–352. Springer International Publishing. https://doi.org/10.1007/978-3-319-09156-3_25

Barnett L, Barrett AB, Seth AK (2009) Granger Causality and Transfer Entropy Are Equivalent for Gaussian Variables. Phys Rev Lett 103(23). https://doi.org/10.1103/physrevlett.103.238701

Behrendt S, Dimpfl T, Peter FJ, Zimmermann DJ (2019) Rtransferentropy - quantifying information flow between different time series using effective transfer entropy. SoftwareX 10:1–9. https://doi.org/10.1016/j.softx.2019.100265

Bender R, Lange S (2001) Adjusting for multiple testing-when and how? J Clin Epidemiol 54(4):343–349. https://doi.org/10.1016/S0895-4356(00)00314-0

Benjamini Y, Hochberg Y (1995) Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. J R Stat Soc Ser B (Methodol) 57(1):289–300. https://doi.org/10.1111/j.2517-6161.1995.tb02031.x

Bessghaier N, Ouni A, Mkaouer MW (2020) On the diffusion and impact of code smells in web applications. In: Services computing – SCC 2020, pp 67–84. Springer International Publishing. https://doi.org/10.1007/978-3-030-59592-0_5

Bessghaier N, Ouni A, Mkaouer MW (2021) A longitudinal exploratory study on code smells in server side web applications. Softw Qual J 29(4):901–941. https://doi.org/10.1007/s11219-021-09567-w

Chaturvedi KK, Kapur PK, Anand S, Singh VB (2014) Predicting the complexity of code changes using entropy based measures. Int J Syst Assur Eng Manag 5(2):155–164. https://doi.org/10.1007/s13198-014-0226-5

Chatzigeorgiou A, Manakos A (2013) Investigating the evolution of code smells in object-oriented systems. Innov Syst Softw Eng 10(1):3–18. https://doi.org/10.1007/s11334-013-0205-z

Couto C, Pires P, Valente MT, Bigonha RS, Anquetil N (2014) Predicting software defects with causality tests. J Syst Softw 93:24–41. https://doi.org/10.1016/j.jss.2014.01.033

D'Ambros M, Bacchelli A, Lanza M (2010) On the impact of design flaws on software defects. In: 2010 10th International conference on quality software. IEEE. https://doi.org/10.1109/qsic.2010.58

Digkas G, Ampatzoglou A, Chatzigeorgiou A, Avgeriou P (2020) On the temporality of introducing code technical debt. In: Communications in computer and information science, pp 68–82. Springer International Publishing. https://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: Software architecture, pp 51–66. Springer International Publishing. https://doi.org/10.1007/978-3-319-65831-5_4

Dusch V et al (2021) Pmd - php mess detector. Online. https://phpmd.org

Dwivedi YK, Williams MD, Mitra A, Niranjan S, Weerakkody V (2011) Understanding advances in web technologies: evolution from web 2.0 to web 3.0. In: European conference on information systems

Edinburgh T, Eglen SJ, Ercole A (2021) Causality indices for bivariate time series data: a comparative review of performance. Chaos: Interdiscip J Nonlinear Sci 31(8):5. https://doi.org/10.1063/5.0053519

Fard AM, Mesbah A (2013) JSNOSE: Detecting JavaScript code smells. In: 2013 IEEE 13th international working conference on source code analysis and manipulation (SCAM). IEEE. https://doi.org/10.1109/scam.2013.6648192

Feise RJ (2002) Do multiple outcome measures require p-value adjustment? BMC Med Res Methodol 2(1). https://doi.org/10.1186/1471-2288-2-8

Gharachorlu G (2014) Code smells in cascading style sheets: an empirical study and a predictive model. Ph.D. thesis, Vancouver, BC V6T 1Z4, Canada

Gilbert RM, Gilbert RM (2019) Accessibility, content, html, javascript, css, and the land of accessible rich internet applications. Inclusive design for a digital world: designing with accessibility in mind, pp 21–42

Glickman ME, Rao SR, Schultz MR (2014) False discovery rate control is a recommended alternative to Bonferroni-type adjustments in health studies. J Clin Epidemiol 67(8):850–857. https://doi.org/10.1016/j.jclinepi.2014.03.012

Granger C (1988) Some recent development in a concept of causality. J Econom 39(1–2):199–211. https://doi.org/10.1016/0304-4076(88)90045-0

Granger CWJ (1969) Investigating causal relations by econometric models and cross-spectral methods. In: Ghysels E, Swanson NR, Watson MW (eds) Essays in econometrics, pp 31–47. Cambridge University Press. https://doi.org/10.1017/cbo9780511753978.002

Gupta A, Suri B, Kumar V, Misra S, Blažauskas T, Damaševičius R (2018) Software Code Smell Prediction Model Using Shannon, Rényi and Tsallis Entropies. Entropy 20(5):372. https://doi.org/10.3390/e20050372

Habchi S, Rouvoy R, Moha N (2019) On the survival of android code smells in the wild. In: 2019 IEEE/ACM 6th international conference on mobile software engineering and systems (MOBILESoft). IEEE. https://doi.org/10.1109/mobilesoft.2019.00022

Heerah S, Molinari R, Guerrier S, Marshall-Colon A (2020) Granger-causal testing for irregularly sampled time series with application to nitrogen signaling in arabidopsis. https://doi.org/10.1101/2020.06.15.152819

Herraiz I, Rodriguez D, Robles G, Gonzalez-Barahona JM (2013) The evolution of the laws of software evolution: A discussion based on a systematic literature review. ACM Comput Surv 46(2):1–28. https://doi.org/10.1145/2543581.2543595

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. https://doi.org/10.1007/s11219-019-09442-9

Josue MPM, Martin AME, Maria Fernanda SG (2019) Mudelsee M () BINCOR: An R package for Estimating the Correlation between Two Unevenly Spaced Time Series. R J 11(1):170. https://doi.org/10.32614/rj-2019-035

Khomh F, Penta MD, Guéhéneuc YG, Antoniol G (2011) An exploratory study of the impact of antipatterns on class change-and fault-proneness. Empir Softw Eng 17(3):243–275. https://doi.org/10.1007/s10664-011-9171-y

Kim J, Hong H, Kim KI (2018) Adaptive Optimized Pattern Extracting Algorithm for Forecasting Maximum Electrical Load Duration Using Random Sampling and Cumulative Slope Index. Energies 11(7):1723. https://doi.org/10.3390/en11071723

Kyriakakis P, Chatzigeorgiou A (2014) Maintenance patterns of large-scale PHP web applications. In: 2014 IEEE international conference on software maintenance and evolution. IEEE. https://doi.org/10.1109/icsme.2014.60

Li W, Shatnawi R (2007) An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J Syst Softw 80(7):1120–1128. https://doi.org/10.1016/j.jss.2006.10.018

Mannan UA, Ahmed I, Sarma A (2018) Towards understanding code readability and its impact on design quality. In: Proceedings of the 4th ACM SIGSOFT international workshop on NLP for software engineering. ACM. https://doi.org/10.1145/3283812.3283820

Marinescu R, Marinescu C (2011) Are the clients of flawed classes (Also) defect prone? In: 2011 IEEE 11th international working conference on source code analysis and manipulation. IEEE. https://doi.org/10.1109/scam.2011.9

Mesbah A, Mirshokraie S (2012) Automated analysis of CSS rules to support style maintenance. In: 2012 34th International conference on software engineering (ICSE). IEEE. https://doi.org/10.1109/icse.2012.6227174

Mills T (2019) Applied time series analysis: a practical guide to modeling and forecasting. Elsevier Science. https://books.google.pt/books?id=1daEDwAAQBAJ

Mudelsee M (2014) Climate Time Series Analysis: Classical Statistical and Bootstrap Methods, 2nd edn. Springer-Verlag, Atmospheric and Oceanographic Sciences Library

Nguyen HV, Nguyen HA, Nguyen TT, Nguyen AT, Nguyen TN (2012) Detection of embedded code smells in dynamic web applications. In: Proceedings of the 27th IEEE/ACM international conference on automated software engineering. ACM. https://doi.org/10.1145/2351676.2351724

Olbrich S, Cruzes DS, Basili V, Zazworka N (2009) The evolution and impact of code smells: A case study of two open source systems. In: 2009 3rd International symposium on empirical software engineering and measurement. IEEE. https://doi.org/10.1109/esem.2009.5314231

Palomba F, Bavota G, Penta MD, Fasano F, Oliveto R, Lucia AD (2017) On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empir Softw Eng 23(3):1188–1221. https://doi.org/10.1007/s10664-017-9535-z

Palomba F, Bavota G, Penta MD, Fasano F, Oliveto R, Lucia AD (2017) On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empir Softw Eng 23(3):1188–1221. https://doi.org/10.1007/s10664-017-9535-z

Palomba F, Bavota G, Penta MD, Fasano F, Oliveto R, Lucia AD (2018) A large-scale empirical study on the lifecycle of code smell co-occurrences. Inf Softw Technol 99:1–10. https://doi.org/10.1016/j.infsof.2018.02.004

Palomba F, Zanoni M, Fontana FA, Lucia AD, Oliveto R (2019) Toward a Smell-Aware Bug Prediction Model. IEEE Trans Softw Eng 45(2):194–218. https://doi.org/10.1109/tse.2017.2770122

Peters R, Zaidman A (2012) Evaluating the Lifespan of Code Smells using Software Repository Mining. In: 2012 16th European conference on software maintenance and reengineering. IEEE. https://doi.org/10.1109/csmr.2012.79

Radjenović D, Heričko M, Torkar R, Živkovič A (2013) Software fault prediction metrics: A systematic literature review. Inf Softw Technol 55(8):1397–1418. https://doi.org/10.1016/j.infsof.2013.02.009

Rani A, Chhabra JK (2017) Evolution of code smells over multiple versions of softwares: An empirical investigation. In: 2017 2nd international conference for convergence in technology (I2CT). IEEE. https://doi.org/10.1109/i2ct.2017.8226297

Ratner B (2009) The correlation coefficient: Its values range between+ 1/- 1, or do they? J Target Meas Anal Market 17(2):139–142. https://doi.org/10.1057/jt.2009.5

dos Reis JP, e Abreu FB, de Figueiredo Carneiro G, Anslow C (2021) Code smells detection and visualization: a systematic literature review. Arch Comput Methods Eng 29(1):47–94. https://doi.org/10.1007/s11831-021-09566-x

Reschke M, Kunz T, Laepple T (2019) Comparing methods for analysing time scale dependent correlations in irregularly sampled time series data. Comput Geosci 123:65–72. https://doi.org/10.1016/j.cageo.2018.11.009

Ricca F, Tonella P (2003) Web application quality: supporting maintenance and testing. In: Information modeling for internet applications, pp 231–258. IGI Global. https://doi.org/10.4018/978-1-59140-050-9.ch011

Rio A, e Abreu FB (2019) Code smells survival analysis in web apps. In: Communications in computer and information science, pp 263–271. Springer International Publishing. https://doi.org/10.1007/978-3-030-29238-6_19

Rio A, e Abreu FB (2021) Detecting sudden variations in web apps code smells' density: a longitudinal study. In: Communications in computer and information science, pp 82–96. Springer International Publishing. https://doi.org/10.1007/978-3-030-85347-1_7

Rio A, e Abreu FB (2023) PHP Code Smells in Web Apps: Evolution, Survival and Anomalies. J Syst Softw 200:111644. https://doi.org/10.1016/j.jss.2023.111644

Rio A, Brito e Abreu F (2021) PHP code smells in web apps: survival and anomalies. CoRR. https://doi.org/10.48550/arxiv.2101.00090. arXiv:2101.00090

Rényi A (1970) Probability Theory. North-Holland Publ. Co, Amsterdam, The Netherlands

Saboury A, Musavi P, Khomh F, Antoniol G (2017) An empirical study of code smells in JavaScript projects. In: 2017 IEEE 24th International conference on software analysis, evolution and reengineering (SANER). IEEE. https://doi.org/10.1109/saner.2017.7884630

Schreiber T (2000) Measuring Information Transfer. Phys Rev Lett 85(2):461–464. https://doi.org/10.1103/physrevlett.85.461

Shannon CE (2001) A mathematical theory of communication. ACM SIGMOBILE Mobile Comput Commun Rev 5(1):3–55. https://doi.org/10.1145/584091.584093

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. https://doi.org/10.1007/s10664-020-09847-2

Shin DW, Sarkar S (1996) Testing for a unit root in an AR(1) time series using irregularly observed data. J Time Ser Anal 17(3):309–321. https://doi.org/10.1111/j.1467-9892.1996.tb00278.x

Siggiridou E, Kugiumtzis D (2016) Granger Causality in Multivariate Time Series Using a Time-Ordered Restricted Vector Autoregressive Model. IEEE Trans Signal Process 64(7):1759–1773. https://doi.org/10.1109/tsp.2015.2500893

Singh V, Chaturvedi K (2012) Entropy based bug prediction using support vector regression. In: 2012 12th International conference on intelligent systems design and applications (ISDA). IEEE. https://doi.org/10.1109/isda.2012.6416630

Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? ACM SIGSOFT Softw Eng Notes 30(4):1–5. https://doi.org/10.1145/1082983.1083147

Tufano M, Palomba F, Bavota G, Oliveto R, Penta MD, Lucia AD, Poshyvanyk D (2017) When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). IEEE Trans Softw Eng 43(11):1063–1088. https://doi.org/10.1109/tse.2017.2653105

Verhoeven KJ, Simonsen KL, McIntyre LM (2005) Implementing false discovery rate control: increasing your power. Oikos 108(3):643–647. https://doi.org/10.1111/j.0030-1299.2005.13727.x
Vern R, Dubey SK (2014) A review on appraisal techniques for web based maintainability. In: 2014 5th International conference - confluence the next generation information technology summit (Confluence). IEEE. https://doi.org/10.1109/confluence.2014.6949320
Yamashita A, Moonen L (2012) Do code smells reflect important maintainability aspects? In: 2012 28th IEEE international conference on software maintenance (ICSM). IEEE. https://doi.org/10.1109/icsm.2012.6405287
Yamashita A, Moonen L (2013) Do developers care about code smells? An exploratory survey. In: 2013 20th Working conference on reverse engineering (WCRE). IEEE. https://doi.org/10.1109/wcre.2013.6671299
Zakas N et al (2021) Eslint - an open source javascript linting utility. Online. https://eslint.org/
Zazworka N, Vetro' A, Izurieta C, Wong S, Cai Y, Seaman C, Shull F (2013) Comparing four approaches for technical debt identification. Softw Qual J 22(3):403–426. https://doi.org/10.1007/s11219-013-9200-8
Zhang M, Hall T, Baddoo N (2010) Code Bad Smells: A review of current knowledge. J Softw Maint Evol: Res Pract 23(3):179–202. https://doi.org/10.1002/smr.521

Américo Rio is an Invited Assistant Professor at NovaIMS/UNL, Lisbon, Portugal, in Programming, Web, and Mobile Development. He has taught at several universities since 2009 and holds a Ph.D. in Computer Science from ISCTE-IUL, where he is also an investigator.
On the industry side, over the last 25 years, he has developed and managed numerous projects for national and international companies involving web, mobile, and desktop applications and systems.

Fernando Brito e Abreu holds a Ph.D. in Computer Engineering from IST/UL and is a Professor in the Department of Information Sciences and Technologies at Iscte-IUL (University Institute of Lisbon) and is a researcher in the Software Systems Engineering group, that he founded at ISTAR-IUL (Information Sciences, Technologies and Architecture Research Center).
He has produced more than two hundred scientific and technical texts on topics such as empirical software engineering, software quality, model-driven engineering, smart tourism, and digital transformation. He is currently chairman of the QUATIC conference Scientific Steering Committee and editor of the Software Quality Journal (Springer). Previously he was an associate editor of Software Quality Professional (American Society for Quality) and, the Portuguese delegate to IFIP TC2 (Software: Theory and Practice).

Diana Mendes is an Associate Professor at the Department of Quantitative Methods for Management and Economics, ISCTE-IUL Lisbon, Portugal. She holds a Ph.D. in mathematics and has broad experience in time series analysis, machine learning, computational economics, and finance. She has authored several technical papers and reports and actively participated in strengthening the link between academia and industry.