INSTITUTO
UNIVERSITÁRIO
DE LISBOA

Web Systems Quality Evolution: a web smells approach

José Américo Alves Sustelo Rio

PhD in Information Science and Technology

Supervisor:
Doctor Fernando Brito e Abreu, Associate Professor,
Iscte – Instituto Universitário de Lisboa

December, 2022

TECNOLOGIAS
E ARQUITETURA

Department of Information Science and Technology

**Web Systems Quality Evolution:
a web smells approach**

José Américo Alves Sustelo Rio

PhD in Information Science and Technology

Jury:

Doctor Jorge Louçã, Full Professor
Iscte – Instituto Universitário de Lisboa

Doctor Alexander Chatzigeorgiou, Full Professor
University of Macedonia

Doctor Alberto Rodrigues da Silva, Associate Professor with Habilitation
IST - Universidade de Lisboa

Doctor Carlos Jorge Costa, Associate Professor
ISEG - Universidade de Lisboa

Doctor Fernando Brito e Abreu, Associate Professor
Iscte – Instituto Universitário de Lisboa

December, 2022

Web Systems Quality Evolution:
a web smells approach

[This page has been intentionally left blank]

To my wife and daughter.

[This page has been intentionally left blank]

Acknowledgements

A doctoral thesis is often described as a long and solitary endeavor; however, the following aims to refute the second claim. Nevertheless, words are not enough to express my gratitude.

First and foremost, I am deeply grateful to my supervisor, Prof. Fernando Brito e Abreu, for his irreplaceable advice, continuous support, constant trust, and pushing the extra mile during this Ph.D. His extensive knowledge and ample experience have encouraged me in my academic research and daily life.

I also sincerely thank Prof. Diana Mendes for her statistics advice and supervision on this critical part of my study. In addition, she participated in the investigation and was always available for support on the inference statistics, which was crucial for this endeavor. She raised the bar very high.

I am grateful to Prof. Rui Menezes (deceased 14 May 2019), whose contribution to part of this work was of great significance. He encouraged and supported using survival analysis techniques and inspired me with his enthusiasm.

I want to thank Prof. Emilia Mendes for her ideas and support during the elaboration of the developer's survey. Without her, we would go directly to the operation of the collaborative web platform.

I extend my gratitude to the monitoring committee, Prof. Alberto Silva and Prof. Carlos Costa, for their opinions, insights, and call to reality on the work of the thesis. They help keep the feet on the ground.

I express my extreme gratitude to my Ph.D. colleague and friend, José Reis, for the discussions, opinions, readiness to help, and more. He was always available to exchange ideas.

I also thank my other Ph.D. colleagues, especially João Caldeira and Nuno António, for their encouragement in this long journey.

I thank my colleagues at NOVAIMS, Vitor Santos and Manuela Aparício, for their support and incentive. Yes, after the Ph.D., there is another life.

I thank Prof. Vasco Amaral for the help with some survey questions.

I also thank Ana Margarida Alexandre from ISCTE Career Services and all the Professors that helped disseminate the survey to former students.

I want to thank my parents, sister, and her family, for their continued love and support. For the missing weekends and holidays not together.

And finally, to my wife and daughter, for their patience, support, and understanding. For the holidays that were to be. For the moments when I was not present in their lives due to this thesis. Thank you.

Lisboa, December of 2022
José Américo Alves Sustelo Rio

Abstract

Context. Web applications and systems are seldom studied, probably due to the heterogeneity of platforms (server and client) and languages. Code smells (CS) are symptoms of poor design and implementation choices in software development that may lead to increased defect incidence, decreased code comprehension, and longer times to release. Web code smells are smells in the context of web development. To study web code smells, we need to consider CS covering the diversity. Understanding their evolution and how long they stay in code is necessary. 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 build a CS catalog for web apps. Next, to discover how CS evolve, their survival/lifespan in typical PHP web systems, globally, by scope or period, and sudden variations in density. Finally, to study the evolution and inner relationship of CS in web apps on the server- and client-sides and their impact of CS (from both sides) on maintainability (web app issues and bugs) and app time-to-release.

Method. We collected and analyzed 18 server-side and 12 client-side CS from 12 typical PHP web apps, summing 811 releases and their metrics, maintenance issues, reported bugs, and release dates. We analyzed CS evolution, distribution and lifespan using survival analysis techniques, and standardized CS evolution anomalies detection criteria. We used several methodologies to identify causality relationships among the considered irregular time series, such as Granger-causality and Information Transfer Entropy(TE) with CS from previous releases (lag 1 to 4).

Results. We discovered evolution trends for server- and client-side CS. We identified the smells that survived the most. CS live around 37% of applications life, almost 4 years, on average; around 61% of CS introduced are removed. We found anomalies in 5 apps' evolution. The most significant TE between CS groups is from client-side to server-side. Individual client-side CS contribute more to issues' evolution, followed by server-side CS. Almost all CS contribute to bugs' evolution; There is causal inference between individual CS and time-to-release(TTR). All CS groups contribute to issues, bugs, and time-to-release (delays).

Conclusions. We presented a catalog of web code smells and a web platform to its expansion. We found that CS stay a long time in code; the removal rate is low and did not change substantially in recent years. An effort should be made to avoid this bad behavior. There is evidence of statistical inference between client- and server-side CS and from the CS to web applications' issues, bugs, and time to release, that welcomes further independent confirmation.

Keywords: web apps, code smells, software evolution, survival analysis, causal inference

Resumo

Contexto. As aplicações e os sistemas Web raramente são estudados, provavelmente devido à heterogeneidade das plataformas (servidor e cliente) e das linguagens. Os cheiros de código (CS) são sintomas de más escolhas de conceção e implementação no desenvolvimento de software que podem levar a um aumento da incidência de defeitos, à diminuição da compreensão do código e a tempos mais longos para o lançamento. Os cheiros de código da Web são cheiros no contexto do desenvolvimento da Web. Para estudar os cheiros de código Web, é necessário considerar os CS que abrangem a sua diversidade. É necessário compreender a sua evolução e o tempo que permanecem no código. Além disso, a literatura fornece poucas provas da afirmação de que os CS são um sintoma de má conceção, conduzindo a futuros problemas nas aplicações Web.

Objectivo. Construir um catálogo de CS para aplicações web. Em seguida, descobrir como os CS evoluem, a sua sobrevivência/vida em aplicações web típicas em PHP, globalmente, por âmbito ou período, e variações súbitas na sua densidade. Finalmente, para estudar a evolução e relação interna de CS em aplicações web no lado do servidor e do cliente e o seu impacto de CS (de ambos os lados) na capacidade de manutenção (problemas e defeitos da aplicação web) e no tempo de lançamento da aplicação.

Método. Recolhemos e analisámos 18 CS do lado do servidor e 12 CS do lado do cliente de 12 aplicações web típicas de PHP, somando 811 lançamentos, e as suas métricas, problemas de manutenção (*issues*), defeitos(*bugs*) reportados, e datas de lançamento. Analisámos a evolução e distribuição de CS, o tempo de vida dos CS usando técnicas de análise de sobrevivência, e critérios padronizados de deteção de anomalias de evolução de CS. Utilizámos várias metodologias para identificar relações de causalidade entre as séries temporais consideradas irregulares, tais como *Granger-causality* e *Information Transfer Entropy*(TE) com CS de lançamentos anteriores (com atrasos *lags* de 1 a 4).

Resultados. Descobrimos tendências da evolução de CS do lado do servidor e do lado do cliente. Identificámos os cheiros que mais sobreviveram. Os CS vivem cerca de 37% da vida útil das aplicações, quase 4 anos, em média; cerca de 61% dos CS introduzidos são removidos. Encontrámos anomalias na evolução de 5 aplicações. O TE mais significativo entre grupos de CS é do lado do cliente para o lado do servidor. Os CS individuais do lado do cliente contribuem mais para a evolução dos problemas, seguidos pelos CS do lado do servidor. Quase todos os CS contribuem para a evolução dos defeitos; Há uma inferência causal entre CS individuais e TTR (*time-to-release*). Todos os CS contribuem para problemas, defeitos, e tempo para lançamento.

Conclusões. Apresentamos um catálogo de *web code smells* e uma plataforma web para a sua expansão. Os CS permanecem muito tempo no código; a taxa de remoção é baixa e não se alterou substancialmente nos últimos anos. Deve ser feito um esforço para evitar este mau comportamento. Há provas de inferência estatística entre cheiros de código do lado do cliente e do lado do servidor e entre os CS e aos problemas das aplicações web (ou sistemas web), defeitos, e tempo para lançamento, que convida a confirmação independente por outros investigadores.

Palavras-chave: aplicações web, cheiros de código, evolução do software, analise de sobrevivência, inferência causal

Contents

List of Figures

[This page has been intentionally left blank]

List of Tables

[This page has been intentionally left blank]

Listings

[This page has been intentionally left blank]

Acronyms

CS	Code Smell.
CSS	Cascading Style Sheets.
CSV	Comma Separated Values.
DB	Database.
FE/BE	Frontend/Backend.
GC	Granger Causality.
HTML	Hypertext Markup Language.
IOT	Internet Of Things.
JS	Javascript.
OSS	Open Source Software.
SE	Software Engineering.
SLR	Systematic Literature Review.
TD	Technical Debt.
TE	Transfer Entropy.
TTR	Time To Release.

[This page has been intentionally left blank]

Part I.

Fundamentals

This part covers the motivation, scope, research problems and main contributions of this work and highlights the fundamental topics, such as: web applications, web code smells, and why and how to study them. It also presents a Systematic Literature Review on web application software evolution studies.

Chapter 1

Introduction

Contents

This chapter introduces the motivation and scope, describes web applications and web systems, and what to study in their evolution.

1.1 Motivation and Scope

Web systems or applications differ from desktop development projects because they run in two places (web server and web browser), encompass a mix of programming languages, and are generally multidisciplinary projects. In the long term, the main goal of this research is to reduce the time and resources spent in the initial development and maintenance of a web project and to improve its global quality. Like any software development project, a web project evolves with time due to changes in the environment and requirements. Therefore, mitigating the problems and malformations in code that cause delays in releases and bugs is essential. Software evolution is a well-established research area inside Software Engineering, but not in web systems and applications. Next, we will detail these concepts. Afterward, we will present the research drivers, problems, questions, objectives, expected contributions, and a list of publications produced in the scope of this thesis.

1.1.1 Web systems or apps

Typical web apps or systems are built with a server-side language, but another part of the app runs in the browser. While a single language is usually used on the server-side (e.g., PHP, C#, Ruby, Python, Java, or JavaScript with nodejs), several languages are used to build the client-side (e.g., JavaScript for the programmatic part, Hypertext Markup Language (HTML) for content, and Cascading Style Sheets (CSS) for formatting).

Web apps usually focus on specific tasks or functionalities, while web systems are comprehensive, interconnected web components or apps designed for more complex tasks or processes. While "web systems" meaning is more embracing than "web apps," we will use both terms interchangeably.

There are three main architectural styles for building web apps: monolithic apps, distributed apps with separated front-end and back-end (Frontend/Backend (FE/BE))[1], and micro-services architecture. Regarding the front-end, the app can be a multiple-page or single-page app (SPA for short). Web applications usually start by having a monolithic architecture (with the server-side code and the client-side code in the same codebase). Later, if needed, some of them change the architecture to FE/BE (distributed app, easy to make front-ends for different devices) or even to micro-services architecture [2] (apps divided vertically and horizontally that aim to increase the scalability, especially on cloud platforms - a team should be responsible by one vertical stack of the application). There are more variants although all have server-side and client-side code. GitHub, one of the major players in the open-source software (Open Source Software (OSS)) ecosystem, acknowledges that diversity by publishing the corresponding language percentages for each project in the bottom right of its page, determined by the *Linguist* library[3].

When an application moves to another divided/distributed architecture, the scalability and maintainability increase, but the required resources, cost, team size, and code size also increase. Managers must consider such aspects, especially when estimating the cost of building a web

[1] https://micro-frontends.org/
[2] https://www.martinfowler.com/articles/microservices.html
[3] https://github.com/github/linguist

app. To increase scalability, monolithic web apps can still be served by multiple web servers, each with several treads (example: Apache webserver with load balancing), and they still are the recommended way to build smaller applications or to start a more complex application [4].

To allow studying both the client- and server-side code and their interrelations, we had to choose monolithic applications because the code is together in one code base. On the other hand, the distributed applications are in separate code bases. They are no longer one application but the front or back end of an application or a part of an application or system, in the case of microservices. Another reason is that monolithic applications span more years for evolution studies. When applications grow bigger or add other platforms (e.g., mobile), they can migrate to other architectures, but this is not always the case [143].

1.1.2 Web systems quality

Since the nineties, we have been observing the migration of some desktop systems to their web counterparts. The web nature, always on and not requiring software installation, gives users flexibility and obvious advantages not encountered in the desktop equivalent.

Regarding development and maintenance, web systems or applications are usually more complex than their desktop counterparts because they have server and client parts. They are also more complex because they encompass a variety of programming languages, such as PHP, C#, JavaScript, and Java, and formatting and content languages, such as HTML and CSS. Another reason is that they require a more diverse range of competencies, with a designer, and a conceptual marketer, in addition to the usual stakeholders of software development (see chapter3.1).

We will study problems in the code or the project itself such as bugs, release delays, and other problems or artifacts. These problems can have roots in poorly written code or architectural problems.

1.1.3 Web code smells

We aim to research if maintainability and reliability problems found in web projects may be due to the violation of fundamental software design and coding principles. Code smells (aka bad smells) typically indicate those violations, require refactoring, and, if spotted early, can improve quality and save time [58]. Code smells are not bugs since they do not prevent a program from functioning, but rather symptoms of software maintainability and reliability problems.

Web smells are code smells (Code Smell (CS)) in the specific context of web systems. Since awareness is fundamental for prevention, we need a web smells catalog. Such documented ontology of web smells can become a vital teaching instrument and a challenge for the research community since we will also look forward to ways of automatically detecting web smells and refactoring web systems code to get rid of them or, at least, mitigate their effect.

[4] https://martinfowler.com/bliki/MonolithFirst.html

Table 1.1: Example of 4 code smells

side	Language(s)	Code Smell name	Explanation
Server	PHP/Java others	Long/Excessive Parameter List	A method/function with an excessive number of parameters
Server	PHP/Java others	Excessive Method Length	(Long Method) A method that has too many lines
Client	JavaScript	max-depth	exceeds maximum depth that blocks can be nested
Client	CSS, JavaScript	CSS in JavaScript	CSS rules are inside JavaScript – difficult to change

Table 1.1 shows 4 examples of web code smells.

Listing 1.1: Excessive(long) Parameter List

```
<?php
function($var1, $var2, $var3, $var4, $var5, $var6$){
    //do something
}
?>
```

Listing 1.1 gives an example of a *Excessive(long) Parameter List*, which is a server-side PHP code smell.

Listing 1.2: Excessive Method Length/Long method (or function)

```
<?php
function($var1, $var2, $var3){
    // do something
    // 200 lines of code
    // and more
}
?>
```

Listing 1.2 is yet another example of a server-side PHP code smell, *Excessive Method Length-/Long method* (or function) with excessive lines of code.

Listing 1.3: JavaScript max-depth

```

function foo() {
    for (;;) { // Nested 1 deep
        while (true) { // Nested 2 deep
            if (true) { // Nested 3 deep
                if (true) { // Nested 4 deep
                    if (true) { // Nested 5 deep
                    }
                }
            }
        }
    }
}
```

Listing 1.3 is an example of a client-side JavaScript code smell, *JavaScript max-depth*, that exceeds the maximum depth that blocks can be nested to reduce code complexity.

Listing 1.4: CSS in JavaScript

```
1
2 function bar() {
3     document.getElementById("div1").style.backgroundColor = 'red';
4 }
```

Listing 1.4 is another example of a client-side embed code smell, *CSS in JavaScript*, which detects CSS rules inside JavaScript, making it difficult to change the code.

1.1.4 Web systems evolution

Software evolution became an accepted research area in Software Engineering. The evolution of software systems is a phenomenon worth studying because it poses severe problems to software projects, encompassing many dimensions and affecting, among others, all phases of the software process, managerial and economic aspects, and programming languages and environments [101].

Software quality evolution (a subset of software evolution) is an active research topic, and published works often focus on the longitudinal observation of defects [107, 168, 171]. Indeed, monitoring software quality evolution for defects over time can help quality assurance teams to forecast and prioritize their efforts. Another research thread in this area is analyzing the trend of software design quality degradation due to long-term maintenance activities, such as functionality extension and bug fixing [173]. Again, monitoring the degradation trends of software design may provide helpful feedback for evolution decisions. Software evolution studies, where we have to consider a series of software system versions, have serious threats, such as the fact that the environmental setup, including the human factor (development/maintenance team), most probably changed throughout the observation period.

We will address the quality of web systems from a longitudinal perspective. This requires choosing metrics that can serve as adequate surrogates for software quality. That choice is not consensual for various reasons [45]. To mitigate this problem, several authors have chosen to study the evolution of code smells[9, 28, 96, 112, 113, 119]. We will do that, but in the scope of web systems.

1.2 Research Drivers

This section presents and details the main problems in the research that lead to the research questions, in the following section. Then, the following section presents the main contributions and, after, the various articles produced during the thesis.

1.2.1 Research Problems

The research problems that originated our research were the following:

- RP1 - Poor characterization of code smells in web systems on the server- and client-side.
- RP2 - Limited knowledge about the life cycle and lifespan of code smells in web systems.
- RP3 - Ignorance of the effects caused by code smells in the evolution of web systems.

1.2.2 General Research Questions

In our research on web systems quality evolution, we will try to answer the following general research questions:

(GRQ1) Which are the most relevant web smells? - Some preliminary attempts have been made to define web smells, as seen in related work of sections 4, 5 and 6, but they have limited coverage, and their validation was almost exclusively done through peer review in the corresponding publication fora. To the best of our knowledge, no comprehensive web smells catalog addresses both sides. Setting up a web smells catalog can be (should be) more than an experienced practitioner's exercise based on "gut feeling." As scientists, we should produce evidence that the presence of those web smells causes problems in web projects. For that purpose, empirical experiments are required.

(GRQ2) Can web smells location be detected automatically? - Collecting web smells location data manually is unfeasible. Depending on the answer to the previous question (i.e., depending on each concrete web smell), different detection techniques may be required [55]. Although there is a considerable amount of research on code smells detection techniques, their application in the context of web systems is a largely uncovered topic.

(GRQ3) Is web systems quality evolution, in terms of maintainability and reliability, influenced by the presence of web smells? - Based on our preliminary literature review, confirmed by the final SLR in chapter 2, we could conclude that the evolution of web systems quality is mostly an unknown phenomenon. On the other hand, if code smells are a catalyst for reduced reliability or maintainability, detecting and removing those smells is expected to improve those quality characteristics. A derived question from this is: **(GRQ3a) What is the behavior of code smells in the evolution of web systems?**

(GRQ4) How can we deal with unevenly time-spaced software development data? - OSS projects, which are amenable as targets for research purposes, usually evolve at an irregular pace since they depend on the availability of their team members. The latter are, most times, unpaid volunteers that commit their contributions whenever they can, not as an obligation, like in commercial projects. Consequently, major and minor software releases – the development cycles of web systems projects – are not evenly time-spaced [164, 165]. If we take data from committed releases, then the precondition for regular time intervals, a precondition for the standard time-series analyses techniques that are usually used in longitudinal studies, like ARMA and ARIMA, is not met, therefore rendering conclusions useless. Most of the basic theory for time series analysis was developed when limitations in computing resources favored an analysis of equally spaced data since, in this case, efficient linear algebra routines can be used and many problems have an explicit solution.

1.2.3 Expected Contributions

In our quest to answer the mentioned research questions, we expect to produce the following contributions:

(EC1) Systematic literature review (SLR) of evolution studies in web systems - SLRs are secondary studies that provide researchers and practitioners a vehicle to gain access to distilled evidence synthesized from results of multiple original studies (aka primary studies). SLRs thus

substantially reduce the time and expertise it would take to locate and subsequently appraise and synthesize primary studies. During our preliminary research on the related work, we could not find any secondary study (systematic review or mapping study) focusing on empirical longitudinal studies on web systems. Producing such an SLR will be an opportunity to deeply understand software evolution experiments' potential, limitations, and pitfalls before defining the experimental designs required for answering RQ3 and RQ4. We will follow the guidelines in [82] to delimit bias in our SLR.

(EC2) Web smells catalog - This catalog will have two parts: one for the client side and another for the server side. The server part will cover PHP, which accounts for around 80% of the market (see section II). A problem to be faced here is that PHP can be used in a procedural or object-oriented manner. The catalog client part will cover JavaScript, HTML, and CSS. To validate our catalog and hopefully obtain an initial consensus on the relevance of each web smell contained in it, we will conduct a survey on practitioners' communities and the industry and develop a collaborative platform to aid in the expansion of the catalog.

(EC3) Collection tools for the proposed catalog - For feasibility's sake, detecting web smells is expected to be as automated as possible. Therefore, we plan to use or extend the open-source PHPMD tool and make some automation mechanisms for the server side. As for the client-side languages, we will use a tool already developed, and for another part, we will develop a tool.

(EC4) Descriptive studies on web systems evolution - Descriptive studies on team and product data will help understand the web systems development phenomenon. Our sample of web systems will be taken from only the final releases, their support pages, or the open-source repositories such as GitHub or Sourceforge. We have to extract each full release and detect the web code smells. For team data, we have to mine the control versioning systems of the projects. Additionally, we will mine those repositories, and their issue-tracking systems, to obtain data on issues and defects. The first prominent aspect we will address is the relative distribution of web smells. If that distribution varies throughout time, it is worth understanding why. If there is a co-occurrence of two smells' evolution, are they assessing the same aspect, or is there some causality effect? The relevance of each code smell or code smell group and its relative frequency will be an interesting decision factor for refactoring. In other words, relevant web smells that occur more often are the ones that should be considered as the first candidates for refactoring. If defects are classified in the issue tracking system, we can also relate/predict the relative frequency of defects to a particular code smell (or group). These descriptive studies will also allow us to assess the validity of the code smells collection tools (EC3).

(EC5) Inference studies of web systems quality evolution - These software evolution studies will be aimed at observing how web smells manifest themselves in sizeable open-source web systems, namely if they have some impact on maintainability and reliability problems, such as release delays, issues occurrence, and faults. In other words, EC5 is expected to answer RQ4.

The expected outcome of these quasi-experimental studies will help increase awareness on detecting web systems' smells as early as possible. Furthermore, removing them is expected to reduce the failure potential and the time spent developing new features, in other words, improving web systems' reliability and maintainability. Longitudinal studies in Software Engineering

have a significant drawback: software systems (and web systems are not distinct in this facet) are released at unequally spaced time intervals. Therefore, as referred in GRQ4, traditional time series techniques (e.g., ARMA and ARIMA) are inappropriate since they assume that data is collected constantly. To mitigate this issue and consequently answer RQ4, we plan to use irregular time series techniques that have been used, for instance, to predict the stock market volatility [43, 44] and in electronic commerce research [74]. This is an active research area in statistics, and new algorithms are being proposed [49, 50]. Its application in the context of software evolution studies is innovative.

1.3 List of publications

The majority of contents included in this thesis are from the following articles, elaborated during the thesis work:

- Rio, A. and Brito e Abreu, F., "Web Systems Quality Evolution," 2016 10th International Conference on the Quality of Information and Communications Technology (QUATIC), 2016, pp. 248-253, doi: 10.1109/QUATIC.2016.060.
- Rio, A. and Brito e Abreu, F., "Analyzing web applications quality evolution," 2017 12th Iberian Conference on Information Systems and Technologies (CISTI), 2017, pp. 1-4, doi: 10.23919/CISTI.2017.7975959.
- Rio, A. and Brito e Abreu, F. (2019). "Code Smells Survival Analysis in Web Apps." In: Piattini, M., Rupino da Cunha, P., García Rodríguez de Guzmán, I., Pérez-Castillo, R. (eds) Quality of Information and Communications Technology. QUATIC 2019. Communications in Computer and Information Science, vol 1010. Springer, doi:10.1007/978-3-030-29238-6_19
- Rio, A. and Brito e Abreu, F. (2021). "Detecting Sudden Variations in Web Apps Code Smells' Density: A Longitudinal Study." In: Paiva, A.C.R., Cavalli, A.R., Ventura Martins, P., Pérez-Castillo, R. (eds) Quality of Information and Communications Technology. QUATIC 2021. Communications in Computer and Information Science, vol 1439. Springer, doi: 10.1007/978-3-030-85347-1_7
- Rio, A. and Brito e Abreu, F., (2023) "PHP code smells in web apps: evolution, survival and anomalies," Journal of Systems and Software, Jun 2023, doi.org/10.1016/j.jss.2023.111644
- Rio, A., Brito e Abreu, F. and Mendes, Diana, "Causal inference of server- and client-side code smells in web apps evolution," under evaluation
- Rio, A. and Brito e Abreu, F. and Mendes, Emília, "Web application software evolution studies: a systematic review", under evaluation
- Rio, A. and Brito e Abreu, F. and Mendes, Emília, "Web development and code smells: an industry survey" - in preparation

1.4 Thesis Outline

The following table shows the correspondence of this thesis chapters and the previous articles:

Table 1.2: Correspondence of chapters to articles

Chapter Name	Article
1-Introduction	-"Web Systems Quality Evolution" and "Analyzing web applications quality evolution"
2-State of the Art	-"Web application software evolution studies: a systematic review"
3-Web development and code smells survey	-"Web development and code smells: an industry survey"
4-Web code smells	-To be published as a technical report
5-PHP code smells in web apps: evolution, survival, and anomalies	-"PHP code smells in web apps: evolution, survival and anomalies"
6-Causal inference of server- and client-side code smells in web apps evolution	-"Causal inference of server- and client-side code smells in web apps evolution"
7-Conclusion	-Various articles plus general conclusions

The advantage of submitting and publishing the chapters was the feedback received from the reviewers that allowed us to improve some of the studies and communications substantially.

This thesis is organized into four parts as follows:

Part I - Fundamentals

Chapter 1 - Introduction

Provides context to the global work, identifies research problems, and details the questions, answers, and solutions prescribed to mitigate them. Finally, it summarizes the benefits and highlights the thesis structure.

Chapter 2 - State of the Art

The state of the art is based on an SLR, and it gives an overview of the related work, summarizing it and identifying research gaps.

Part II - Code smells on web systems

Chapter 3 - Web Development and Code Smells: an industry survey

Presents the results of a survey to web developers to access their education and experience, the projects developed, the team's constitution and their knowledge of web code smells.

Chapter 4 - Web code smells

Proposes a web smells catalog and a working catalog used in the various studies developed through this thesis. Additionally, it presents a collaborative platform for extending and validating the web smells catalog.

Part III - Web systems evolution studies

Chapter 5 - PHP code smells in web systems: evolution, survival and anomalies
Code smells in web apps: evolution, survival, and anomalies - presents a set of studies on server-side (PHP) code smells. The evolution and possible causes, the survival, and anomalies in the evolution of code smell density.

Chapter 6- Causal inference of server- and client-side code smells in web systems evolution
This chapter presents as extensive set of studies to answer the descriptive and the causal inference questions, with code smells both form server- and client-side. First, it presents the evolution of the individual smells and, as a group, studies if they are correlated between the same groups and verifies if one group of code smells can statistically cause the other. After, it assesses if code smells for both sides, individually or as a group, can statistically cause changes in issues, bugs, and time to release values.

Part IV - Conclusion

Chapter 7- Conclusion and Future Work
Concludes and summarizes the achievements of the thesis and discusses future work.

1.5 Chapter conclusions

The first chapter provided an overview of the research performed during the thesis. It began with the characterization of web systems apps, their quality, and their differences from desktop apps. Then, the topic of code smells on web systems or apps, and the necessity to perform evolution studies was introduced. The following section presented the main research problems, general questions, principal objectives, and expected contributions. The subsequent section presented the list of publications elaborated on during the thesis work. Finally, we present the thesis outline with a resume of each chapter's contents.

Chapter 2.

State of the Art

Contents

This chapter presents the protocol design, execution and findings of a Systematic Literature Review (SLR) on Web application software evolution studies.

2.1 Introduction

2.1.1 Motivation

According to the target platform, application software development has three main areas: desktop, web, and mobile. In addition, there are other software and applications, such as operating systems development, cyber-physical, and Internet Of Things (IOT); however, these two examples are more connected to hardware, and some of the software runs embedded.

In desktop/standalone development and native mobile applications, software programs run mainly on one computer or system and can be connected to the internet with web services or other means. However, web applications typically run on 2 (or more) computers: the server-side part runs remotely on a server or servers, and the client-side part runs in a browser (whether the code is in the same code base or not).

One of the prominent areas of Software Engineering is the study of application evolution to understand what happens to an application through time, i.e., longitudinal studies. Early evolution studies regarded desktop applications, namely in Java [26, 78], as well as secondary studies synthesizing the former [20, 24]. While web development has been a significant slice of software development in the last two decades [92], our initial literature review revealed that software evolution studies in this area are scarce.

Secondary studies do exist in web development[88, 109]. However, there is a knowledge gap in secondary studies concerning web systems or app evolution. In the longitudinal web evolution studies, we expect to find aspects like metrics, maintainability issues, defects, and other quality aspects like code smells or technical debt, typical to software evolution studies in general. But also aspects related to web development, like the evolution of server and client code, databases, security aspects through releases, and even web server logs evolution. We will consider longitudinal studies in applications with more than ten continuous releases or units of time.

We aim to characterize the knowledge on evolution in web applications, to understand how they evolve regarding their various aspects during their life cycle. Furthermore, we aim to understand what variables, periods, applications, methodologies, and techniques are being used and studied, so we need to synthesize the current state of the art, allowing us to resume the knowledge and pinpoint the necessities of the area. Therefore, we propose a systematic literature review (SLR) of Web application evolution studies or Web application longitudinal studies. An SLR is *"a form of secondary study that uses a well-defined methodology to identify, analyze and interpret all available evidence related to a specific research question in a way that is unbiased and (to a degree) repeatable"* [82].

2.2 Related work

We are unaware of the existence of any other SLR published to date treating the crosscutting concerns of web development and software evolution. Therefore, we present related SLRs in each of the two topics alone.

2.2.1 Studies of web applications

We did not find any SLR aimed at the general study of web applications but only at some concerns in particular.

Shuaibu and Selamat [109] present an SLR to investigate the security development models used in web applications, the security approaches or techniques used in the process, the stages in the development model, and the tools and mechanisms used to detect vulnerabilities. They found that no development model is considered a standard or preferred for web application development, but agile development models have gained more attention. However, there is consistency in the threat-modeling technique due to its effectiveness in dealing with different vulnerabilities.

Lawal et al. [88] give a review on SQL injection attack, detection, and prevention techniques. They evaluate the techniques to check the effectiveness of each technique based on how many methods of attack it can detect and prevent. They also consider the resources used, such as memory and processing time.

Garousi et al. [60] present a systematic mapping study of the area of web application testing, where they observe the trends in terms of types of papers, sources of information to derive test cases, and types of evaluations used in papers, as well as papers demographics. Finally, the authors discuss emerging web application testing trends and implications for researchers and practitioners.

2.2.2 Software evolution studies

In [20], Breivold et al. presents a systematic review of open source software (OSS) evolution to understand how software evolvability is addressed during the development and evolution of open source software. Based on the studies' research topics, the authors identified four main categories of themes: software trends and patterns, evolution process support, evolvability characteristics addressed in OSS evolution, and examining OSS at the software architecture level. A comprehensive overview and synthesis of these categories and related studies were presented, including the metrics used.

Syeed et al. [152] present a systematic literature survey to identify and structure research on the evolution of Open Source Software (OSS) projects. The study outcome provides insight into what constitutes the field's main contributions as gaps, opportunities, and future research directions. Specifically, they analyze the target, approach (method and metrics), target group, and outcome of the studies (contributions).

Chahal and Saini [24] report an SLR on the OSS evolution process topic, categorizing the research studies into nine categories. In the first category, "techniques used for OSS evolution analysis," there is evidence of a shift in the metrics and methods for OSS evolution analysis over the period. OSS systems grew at a superlinear rate in the initial studies. However, later studies revealed that branches of an OSS system grow at different rates. The studies use Software size metrics, metrics related to change activity, and Techniques and tools to aid in software evolution analysis and prediction. In part 2 of the SLR [25], they analyzed the remaining categories and found that there needs to be a uniform approach to analyze and interpret the results. Some techniques include the use of prediction techniques that extrapolate the historical trends into

the future, but it is observed that there are no long-term correlations in the data of such systems. Finally, they observed that OSS evolution as a research area is still nascent.

2.3 Review questions

The SLR objective is to get the current state of the knowledge in "evolution studies in web applications." To our knowledge, no SLR covers this topic, as shown in the related work. Therefore, the SLR will help practitioners and researchers get state-of-the-art rapidly and become supported by scientific evidence from the studies.

"Evolution Studies" in Software Engineering refers to systematically analyzing and understanding how software systems and projects change, adapt, and evolve over time. These are often longitudinal studies that look at the development and adaptation of software over a certain period or across a set of releases [26, 78]. These studies help in understanding: codebase changes, defects, design, process and practice, and dependency evolution (dependency from libraries, frameworks, or others). The results of these studies can improve future project planning, understand and manage technical debt, improve the quality of the software, and in general, guide the decisions in software projects. Methods used include mining software repositories, metrics analysis, change analysis, and trend analysis. Tools used may include version control systems, bug tracking systems, static analysis tools, and other data collection and analysis tools[20, 24].

2.3.1 Primary review questions

Taken from the objective, the primary research questions to be answered in the review are the following:

RQ1 **What are the software evolution studies with web software?** – This is the central question that allows filtering the studies, with the inclusion and exclusion criteria. Evolution or longitudinal studies will measure variables in a software system over time.

RQ2 **What causes problems in web software evolution quality?** – Answering this question will enable us to discover the independent and dependent variables that explain web development quality evolution.

RQ3 **How can we deal with unevenly time-spaced software development data?** – Usually, in a study with applications with code available to the public, the applications will have a non-periodic release frame. How do researchers deal with that irregular data?

2.3.2 Secondary review questions

From the primary questions, we extract the secondary review questions that are going to aid in the synthesis of the extracted data, as follows: the second primary question is divided into secondary questions 1 and 2, and are the most revealing in the SLR; the third secondary question comes from RQ3; the remaining secondary questions come from RQ1 and aim to characterize the studies.

SRQ1 What are the attributes or the dependent variables to describe the evolution of the quality of web software? - The answer to this question will show us the outcome variables most studied in the selected evolution studies.

SRQ2 What are the factors or the independent variables to influence the evolution of the quality of web software? - The answer to this question will uncover the causal variables most studied in the selected evolution studies. The first two questions come from RQ2.

SRQ3 Does it use techniques for unevenly time-spaced data? If not, time series techniques? - Software projects, especially open-source projects, usually release at irregular time intervals since they depend on the availability of their team members. Synthesis of this data will let us understand if statistical techniques for regular or irregular interval time series or even different approaches are used.

SRQ4 Is collected data made available for replication? - This question aims to check if the study data is available to allow for replication.

SRQ5 What are the analysis techniques used in the study? - The answer to this question will reveal the analysis techniques used to evaluate the applications' evolution numerically.

SRQ6 What is the methodology used? - In this question, we check the methodology used in the study: methods, statistical analyses or other analysis, synthesis of data, or comparisons.

SRQ7 What is the period of the study? - Evolution studies can span months or decades of releases. The answer to this question will show us the periods used in the study.

SRQ8 What are the languages used in the applications of the study? - A plethora of programming languages can be used in web programming on the server side. However, for the client side, only JavaScript and the formatting languages(HTML and CSS), since the almost retirement of ActionScript (flash). Therefore, we aim to find the most studied languages in web evolution studies.

SRQ9 Is automatization of the study possible? - Studies can be done manually or automatically. In this question, we aim to answer if automatization of the study is possible for future scalability.

SRQ10 Does it test real cases? What is the number of applications used in the studies? - This question will answer if the study's applications are real and can be used by the general public or if they are experiments with, for example, students. We also count the number of applications used in the studies.

SRQ11 Is the study and conclusions general or domain-dependent? - Are the study's conclusions only suitable for a particular area of development or a specific language, or can they be generalized to all web applications?

Next, we present the SLR methodology to select and extract the data that allows us to answer the review questions.

2.4 Review methods

In this chapter, we present the review protocol used. Part of the review protocol was presented in [132]. We followed Kitchenham et al. [82] guidelines to perform systematic literature reviews in software engineering, with minor changes. The method is divided into different phases that are presented next.

2.4.1 Data sources and search strategy

The search string intersects the terms "web application" and its synonymous, and "evolution" with its synonymous, with the terms meaning a "longitudinal study," i.e., a study during a period, as follows:

("web application" OR "web-based application" OR "web development"
OR "web system" OR "web software" OR "web quality")
AND ("longitudinal" OR "time series" OR "software evolution")

As a first attempt to build the search string, we tried the term "longitudinal study," but the term "study" can be replaced by a synonymous, and thus the final string was the one presented. Furthermore, we had to use digital database filters and modify the search string in some databases due to their inner mechanisms.

2.4.2 Study selection

We used a database to aid in managing SLR data B.4, with forms B.5 for the easy fill of the classification of each article in the database. First, we joined all articles in sheets within a spreadsheet and joined them with a column for "no duplicates." After removing duplicates, we no longer divide by the article's source (journal or conference).

We also used Mendeley[1] as our reference manager.

2.4.2.1 Phases of the selection

Phase 1 - Search the databases - Search the electronic databases (**Springer, Scopus, IEEE, ACM, ISI, Science Direct, and Willey online**) with the search string. Join all articles retrieved and remove duplicates. Then, insert the articles in a local database.

Phase 2 - Analyze title, keyword, abstract - Read title, keyword, and abstract and mark articles for the next phase, using database forms for displaying each article data separately for easier reading. We marked the article in this stage with a *maybe* or *yes* for later review.

Phase 3 - Read the complete article. - Apply inclusion and exclusion criteria. – In this phase, the article is evaluated, and we apply the inclusion and exclusion criteria. At this phase, we had to retrieve the articles to read them. This phase had two steps: First, we read the articles to remove the ones that were out of scope and that we could not identify just by the abstract and title. After, we employed the inclusion and exclusion criteria, which are shown next. Studies of the same authors can be present if the research questions differ.

Inclusion Criteria

[1] https://www.mendeley.com/

- Origin: Conference, Journal, Book
- If studies are duplicated or referenced, only the most recent paper will be included.
- Studies published in time-frame 2002 / 2020 (we started until 2017, then extended the upper limit)
- Empirical research (not theory or lessons learned)
- Evolution/Longitudinal study (min. 10 releases or measures)
- Study with applications

Exclusion Criteria

- Duplicates found
- Full text not available
- Papers not in English
- Out of scope (e.g., desktop and native mobile software)

Phase 4 – Snowball - Perform backward snowball [163] from the selected articles: Check the relevant articles referenced in the selected articles, reading the related work and the bibliography for relevant titles. In this phase 4 (snowball), we have to repeat the tasks done in phase 2 (read title, abstract, and keywords) and in phase 3(read article). After applying the criterion, the studies enter phase 5, quality assessment.

Phase 5 -Study quality assessment - For the study quality assessment, the primary studies are considered according to reporting, rigor, credibility, and relevance [48, 79]. To that end, we classify the studies according to the following parameters: Reporting: (1,2), Rigor: (3,4,5), Credibility (6,8), and Relevance (7). The complete criterion quality list follows:

1. The aim and objectives are clearly defined.
2. The study provides an adequate description of the context.
3. The study design is appropriately aligned with the research aim.
4. The sample is adequately described and of suitable size.
5. The techniques and methodology are well-defined.
6. The study presents clearly stated findings, credible results, and justified conclusions.
7. The study provides valuable contributions to the scholarly community.
8. The study includes data suitable for replication.

The first three parameters are the minimum threshold for the study to be accepted. The other parameters will classify each study with 0 or 1. The values for the scholar community are measured using the article citations.

2.4.2.2 Concordance of study selection

The principal investigator performed the phases of the selection of articles. To reduce bias in study selection, we referred once again to Kitchenham[82], and from the articles in phase 3 (*YES* and *MAYBE* articles) we selected a random sample of 30 articles and a second investigator selected the articles independently. Then, we measured the concordance finding the values of Cohen's Kappa [34].

The values used for Cohen's Kappa Level of Agreement are the fowling: 0%-20% None; 21%-39% Minimal; 40%-59% Weak; 60%-79% Moderate; 80%-90% Strong and Above 90% Almost Perfect.

2.4.3 Data extraction

The data were extracted from the studies using a paper form, after inserted into the database using a form B.5. The form in the figure answers the secondary questions, almost replicating those questions, plus additional questions belonging to the inclusion criterion. The data was stored in a table "analyses" connected to the "studies" table B.4.

2.4.3.1 Concordance of data extraction

To measure the concordance of the data extraction, according to the Kitchenham method to reduce bias in data extraction [82], a third of the selected articles were selected randomly, and then a second investigator extracted the data independently. Then, the concordance values were again calculated with Cohen's Kappa.

2.4.4 Data synthesis

We extracted the data from the database and exported it to a spreadsheet file for the data syntheses. This procedure was straightforward because the data was divided into secondary question fields. We used Meta-ethnographic methods to perform the synthesis [111] by identifying similarities and differences across the studies, looking for patterns and relationships, and interpreting the findings related to the research questions. Data were organized in a tabular format (spreadsheet) with lines representing the studies and columns representing the secondary questions data. We found common words (synonyms) for achieving the same concept from a different description for qualitative data. In some cases (columns), we transformed the data into interval data for the quantitative data. Finally, we counted the cases and built tables and charts, including qualitative and interval data percentages.

2.5 Included and excluded studies

Figure 2.1 shows the phases 1-5 of the study selection. The search on the online electronic databases resulted in 933 articles; after removing duplicates, the end result was 844. First, duplicates were removed in a spreadsheet with a specialized function. Then, we used this spreadsheet to import the data to the articles table in the database.

The result of phase 2 (*analyze the title, keyword, and abstract*) was 135 articles. Next, phase 3 (*read complete article*) was divided into two subparts to ease the work. The former *read abstract and title* resulted in 70 articles selected. In the latter, we read the complete article and employed the inclusion and exclusion criteria, and upon its completion, we had 19 studies.

In phase 4, we performed snowball and found 37 articles of possible interest. From the 37 snowball articles found, we selected 10 in phase 2 and only two after phase 3.

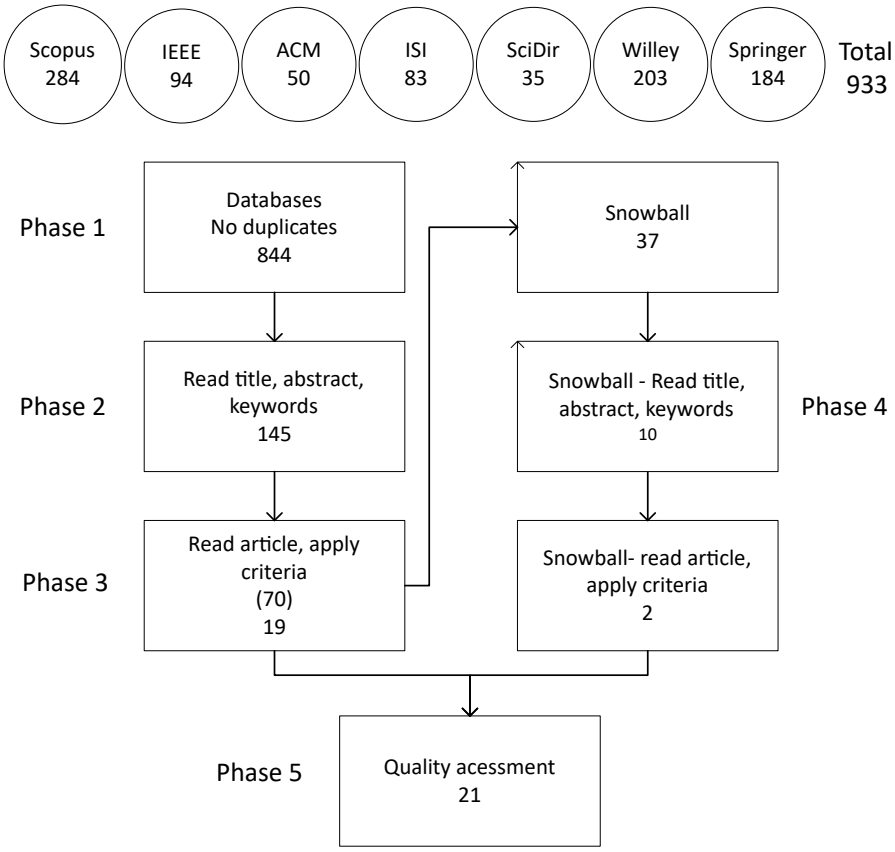


Figure 2.1: Systematic literature review Study collection phases

Table 2.1: Number of articles found by electronic database

Database	#Articles
Scopus	284
IEEE	94
ACM	50
ISI	83
Science direct	35
Willey online library	203
Springer Link	184
Total	933
Remove duplicates	844

In total, of phase 1, we had 844 studies, with no duplicates (from 849). Summing up the 37 snowball studies, we analyzed 881 articles. However, we only read 145+10= 155 studies. After the criterion's appliance, we had 21 studies to enter phase 5, quality assessment.

Phase 5 consists of applying the quality assessment. Table 2.2 resumes values and percentages for the quality assessment of the selected articles before phase 5. All articles obtained a score equal to or greater than 6. Since we considered the selection threshold for quality to be 6, after phase 5, we ended with the same 21 studies.

Table 2.2: Quality table numbers and percentages

Quality score	# studies	% studies
6	4	19%
7	6	29%
8	11	52%

2.5.1 Concordance on selection of articles

After phase 2, 30 random articles were selected to be classified by an independent reviewer. After this classification with "YES" and "NO," we used the statistic program "R" to measure the Choens' Kappas concordance statistic. Cohen's Kappa value was Kappa = 0.783, which was due to an article not being selected by the second investigator and selected by the principal investigator. After carefully reading by the second investigator, the article passed the inclusion test (making a kappa of 100%), but we included the first kappa found independently.

2.6 Results

This section presents the results of the data extraction and synthesis for the secondary questions. The main questions are answered at the end of the section by synthesizing the findings. To characterize state of the art on web application software evolution, we selected the 21 studies on B.1, following the earlier selection process.

2.6.1 Study overview

We begin by characterizing the studies, venue, and year of publication.

Table 2.3: Venues of the studies

Venue	Number Studies	Percentage
Journal Article	4	19%
Conference Paper	16	76%
Book Chapter	1	5%

Table 2.3 shows the studies' venue, i.e., where the studies were published. The selected studies consist of 4 Journal articles (19%), 16 conference papers(76%), and one book chapter(5%).

Figure 2.2 presents the study years of publication. In the first decade, fewer web evolution studies were published. However, in the second decade, the panorama changed. The trend is up until the peak in 2019.

2.6.2 Studies resume

We present a brief resume of the selected longitudinal studies. The studies are grouped into research focus categories, similar to the process used in [48], which we identified when extracting the data:

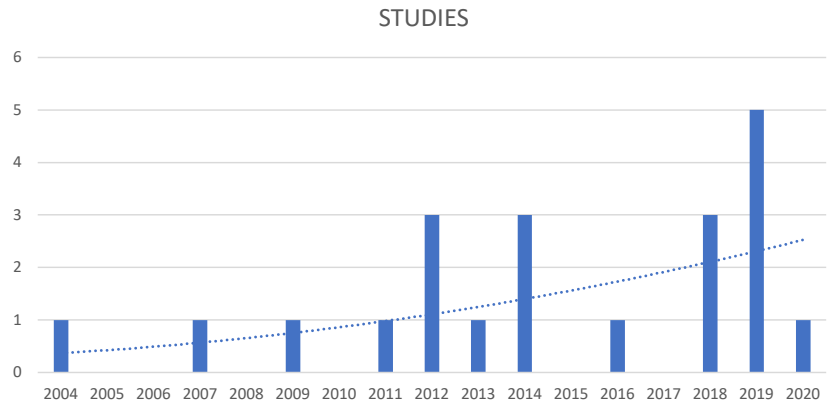


Figure 2.2: Publication years

- Laws of Evolution - Longitudinal studies focused on Lehman's software evolution laws
- Maintenance and Bugs - Longitudinal studies about how maintenance measures and bugs change over time in the apps studied
- Development techniques - Studies focused in evolution of development techniques though the period and apps selected
- Server - Longitudinal studies with the web server
- Security - Security longitudinal studies
- Code Smells (CS) and Technical Debt (TD) - Empirical evolution studies (through a definite period of time) that the variable are CS or TD
- Includes - Longitudinal studies about the inclusion of libraries in the apps studied though time

As the name longitudinal suggests, all the studies are empirical, i.e., they have data.

2.6.2.1 Laws of Evolution

S1 analyzes the evolution of open-source PHP projects to investigate if Lehman's laws of software evolution are confirmed. Data (changes and metrics) were collected for successive versions of 30 PHP projects, and statistical(trend) tests were employed to evaluate each law's validity. The authors found that all laws of evolution hold, except law VII (inconclusive). Conclusions: web applications exhibit constant growth and are under continuous maintenance. They did not find evidence that quality deteriorates over time.

S14 analyses the evolution of 20 JavaScript applications and libraries. The authors investigate Javascript (JS) applications' quality and changeability trends over time by examining the relevant Laws of Lehman. The results show that JS applications continuously change and grow; there are no apparent signs of quality degradation, while the complexity remains the same over time, although the understandability of the code deteriorates.

S18 performs a large-scale study on the evolution of database schema, part of 8 larger open source projects, over 10-12 years, and reports on the validity of Lehman's laws of software evolution, like size, growth, and amount of change per version. The findings are diverse and depend on the application database.

2.6.2.2 Maintenance and Bugs

In the study (S2), the authors investigate the maintenance patterns of five large and well-known PHP applications' evolution. Several historical aspects are examined, including the amount of unused code, removal of functions, use of libraries, stability of their interfaces, migration to object orientation, and evolution of complexity. Results show that these systems undergo systematic maintenance driven by targeted design decisions.

S3 presents a longitudinal quantitative study of test script maintenance in open-source web applications in Java (47 apps, eight high quality) and a longitudinal qualitative study of the changes kind. They propose categorizing the changes within each commit, thus identifying the elements of a test that are most prone to change. Although out of our scope, S3 presents a cross-sectional quantitative study of the prevalence and extension of SELENIUM-based tests among 283 open-source web applications. They propose two metrics based on whether a commit touches a test file.

S5 describes the technical issues in detecting dead PHP code and proposes an identification and removal approach based on dynamic analysis. They examine the approach in an industry-scale web system (with six sub-systems during 1-6 years) and discuss lessons learned.

Study S11 presents a large-scale study of client-side JavaScript code over time. The authors collected and analyzed a dataset containing daily snapshots of JavaScript code from Alexa's Top 10000 websites for nine consecutive months. Findings: scripts often change every few days, indicating a rapid pace in web applications development; the time between JavaScript changes is also short; most changes are related to the introduction of new functions and online configuration management; code reuse: widespread reliance on third-party libraries. The authors observed how quality issues evolve by employing static analysis tools to identify potential software bugs, whose evolution they tracked over time, and results show that quality issues persist over time while vulnerable libraries tend to decrease.

Study S17 presents a time series analysis of defects reported during software evolution. Using data from monthly defect reports for eight open-source software projects (3 web projects) over five years. This study builds and tests time series models for each sampled project. The research discovered that a single time series model, ARIMA(0,1,1), accurately predicts the pattern of software evolution defects for each project, providing a basis for both descriptive and predictive software defect analysis that is computationally efficient, comprehensible, and easy to apply.

2.6.2.3 Development techniques

S4 is an empirical study/experiment that evaluates the effectiveness of applying four different AO interfaces to 2 programs (one is a web app) with 50 releases (the other is a mobile app with 35 releases). They perform a comparative analysis using Chidamber and Kemerer metrics showing the strengths and weaknesses of these AO interface proposals. They also show the design stability with impact analysis.

S6 presents an analysis of the evolution of a Web application project developed with object-oriented technology and an agile process. During the development, they measured Chidamber and Kemerer metrics, Lines of Code metrics, and metrics from the class dependency graph

(including Social Network Analysis metrics). The application development evolved through phases characterized by adopting essential agile practices (pair programming, test-based development, and refactoring). They show that a few metrics are enough to characterize with high significance the various phases of the project and conclude that software quality, as measured using these metrics, seems directly related to agile practices adoption.

2.6.2.4 Server

S7 reports on an automated runtime anomaly detection method at the application layer of multi-node computer systems. The authors model a Web-based system as a weighted graph, where each node represents a "service," and each edge represents a dependency between services. They first extract a feature vector representing all of the services of the two applications studied. Then they derive a probability distribution for an anomaly measure in the time series. Next, the threshold value for critical probability is adaptively updated using a novel online algorithm. Finally, they demonstrate that a fault in a Web application can be automatically detected, and the faulty services are identified without using detailed knowledge of the system's behavior.

2.6.2.5 Security

Study S8 presents an investigation of the evolution of security vulnerabilities, reporting results about experiments performed on 31 versions of phpBB. The authors assess the evolution of percentages of vulnerable Database (DB) access and scripts from the "/admin" directory. Results show that the vulnerability analysis can be used to observe and monitor the evolution of security vulnerabilities in subsequent versions of the same software package.

The paper S9 defines Property Satisfaction Profiles (PSP) as the satisfaction values of properties computed on the extracted models. It investigates the evolution of the changes in the PSP computed on security models of 31 successive versions of phpBB, using an original algorithm.

S12 examines software vulnerabilities in Python packages from the PyPI package repository (mainly used for web development) and Safety DB, a database of vulnerabilities in the repository packages. The methodological approach builds on a release-based time series analysis of the conditional probabilities for the releases of the packages to be vulnerable. Results: many Python vulnerabilities seem to be only modestly severe; most typical vulnerabilities are *input validation* and *cross-site scripting*. For the time series analysis based on the release histories, they found that only the recent past is relevant for statistical predictions.

In S15, the authors determine privilege-level security impacts of code changes (automatically and statically) using privilege protection changes and apply a set-theoretic classification. To do so, they analyze code and determine the security privilege protection models of Web applications written in PHP using Pattern Traversal Flow Analysis (PTFA). Finally, they present the distribution of privilege protection changes and their classification over 147 release pairs of WordPress, spanning from 2.0 to 4.5.1. Findings: code changes had no impact on privilege protection in the 82 (56%) release pairs; the remaining 65 (44%) release pairs are affected by privilege protection changes, and from those, only 0.30% of code is affected by privilege protection changes; most common change categories are complete gains (40.81%), complete losses (17.99%) and substitution (20.10%).

S16 presents a refinement of the methodology of S15 to reduce to work necessary to identify protection-impacting changes. Results of experiments present the occurrence of protection impacting changes over 210 successive release pairs of WordPress (a more comprehensive study than S15). Findings: only 41% of the release pairs present protection-impacting changes; for these affected release pairs, protection-impacting changes can be identified and represent a median of 47.00 lines of code(27.41% of the total changed lines of code); protection-impacting changes amounted to 10.89% of changed lines of code; an average of about 89% of changed source code have no impact on Role-Bases Access Control (RBAC) security and thus need no re-validation nor investigation. The proposed method reduces the number of candidate causes of protection changes that developers need to investigate.

2.6.2.6 CS and TD

S10 presents MTV-Checker, a tool to automatically detect five design violations in Django-based web applications. The authors conducted an empirical study in the context of a large-scale Django-based information system. The results present the most recurrent violations, how they evolve along software evolution (12 releases), and the opinions and experiences of software architects regarding these violations.

S13 analyzed the evolution of 6 code smells(CS) in 4 web applications built with PHP and Code Smells (CS) survival, that is, how long it takes from their introduction to their removal. In the study, CS are divided into two categories: scattered smells and localized smells. The results provide some evidence that the survival of PHP code smells depends on their spreadness. Next, it is analyzed whether the survival curve for the same web application varies in the long term. The results show that there is indeed a change for all applications except one.

S19 analyzes the evolution of technical debt in 44 Python open-source software projects (7 web apps) from the Apache Software Foundation, focusing on the type and amount of technical debt that is paid back. Findings: most of the repayment effort goes into testing, documentation, complexity, and duplication removal; more than half of the Python technical debt is short-term, being repaid in less than two months; the observations that a minority of rules account for the majority of issues fixed and spent effort suggest that addressing those kinds of debt in the future is important for research and practice.

2.6.2.7 Includes

S20 reports on a large-scale crawl of more than three million pages of the top 10,000 Alexa sites and identifies these sites' trust relationships with their remote library providers (JavaScript library includes) over ten years. First, they show the evolution of JavaScript inclusions over time and develop a set of metrics to assess each JavaScript provider's maintenance quality. In some cases, top Internet sites trust remote providers that determined attackers could successfully compromise and subsequently serve malicious JavaScript. Next, they identify four previously unknown vulnerabilities that attackers could use to attack popular websites. Finally, they review some proposed ways of protecting a web application from malicious remote scripts and show that some may not be as effective as previously thought.

S21 performs an empirical study of technical lag (outdated dependencies) in the npm dependency network (JavaScript) by investigating its evolution for over 1.4M releases of 120K packages and 8M dependencies between these releases. They explore how technical lag increases over time, taking into account the release type and the use of package dependency constraints. They also discuss how technical lag can be reduced by relying on the semantic versioning policy.

2.6.3 Findings

The section presents the findings of the SLR. For the first two secondary questions, we additionally related the findings (variables) with the studies, as done in [23].

2.6.3.1 SRQ1 - Attributes or the dependent variable to describe the evolution of quality of web software

Table 2.4: Attributes or Dependent variables used in the studies

Atributes	#	%	Primary Studies
Security vulnerabilities	4	17%	S8,S11,S12,S20
Lehmann Laws validity	3	13%	S1,S14,S18
Maintenance metrics	3	13%	S2,S3,S20
Security access privileges	3	13%	S9,S15,S16
CS evolution/survival	2	8%	S10,S13
Bugs	2	8%	S11,S17
Maintenance tests	1	4%	S3
Code metrics	1	4%	S4
Dead code elimination	1	4%	S5
Agile practices adoption	1	4%	S6
Server anomaly/fault	1	4%	S7
TD resolved (paidback)	1	4%	S19
Technical lag	1	4%	S21

From the data obtained, the most studied attributes or dependent variables in web application evolution are the *Security vulnerabilities* (4) followed by the *Lehmann Laws*, *Maintenance metrics* and *Security model* with three studies each. The topics *Code Smells evolution/survival* and *Bugs* have two studies each. The remaining topics have 1 study each: Maintenance tests, code metrics, Dead code elimination, Agile practices, Server anomaly/faults, TD resolved (paid back), and Technical lag.

Studies S8,S11,S12,S20 research **Security vulnerabilities** evolution as attributes. Study S8 tackles the evolution of security vulnerabilities and the percentage of vulnerable DB accesses on 31 phpBB versions (6 years). S11 researches potential bugs and vulnerable JS libraries in the top 10000 Alexa sites for nine months. S12 studies vulnerabilities in apps/libs from the PyPI package repository (335 packages) over ten years. S20 uses vulnerability and security to describe the evolution of JavaScript for ten years.

Lehmann Laws are used in S1,S14,S18 as attributes. S1 tries to prove Lehmann Laws 30 PHP web projects in the 5-10 evolution. S14 studies 5 of the Laws from 20 JavaScript apps and

libs during their lifespan (max ten years). S18 studies Lehmann Laws for web apps' database schema evolution, using eight apps during 10-12 years.

S2, S3, and S20 use **Maintenance metrics** as the dependent attributes. S2 tests whether large PHP web apps have been maintained systematically during ten years of evolution. S2 studies functional test changes (maintenance metrics) through the evolution of several Java applications (47 web apps). S20 appears again in this subject and uses the quality of maintenance metric to describe the evolution of JavaScript *includes* during ten years.

Security model is studied in S9,S15,S16 to describe the evolution. S9 uses PSP (property satisfaction profiles) to describe the security model in 31 versions of a PHP web app over six years. S15 studies one PHP web app's privilege protection changes (in RBAC) in 147 release pairs (148 versions). S16 builds on S15 and proposes a method to only study the protection impact changes (percentage of privileged protection changes/Code changes), using 210 version pairs (211 versions).

CS evolution/survival is studied in S10 and S13. S10 studies five design violations (code smells) in 12 releases of one Django (python) web app (4 years and four months). S13 studies 6 code smells in the evolution of 4 PHP web apps ((8-14 years).

S11 and S17 use **bugs** as the dependent variable to study. S11 uses potential JavaScript bug measures for nine months with *JS HINT*. S17 tries to predict defects (bugs) in 8 apps (3 of their web apps) monthly from 66 to 90 months.

There are other attributes used in one study only. **Code metrics** are used as the dependent variable in S4, a study that includes a web app in 50 releases. S5 uses **dead code elimination** in a daily 5-month study with one PHP project. S6 tries to evaluate the **agile practice** used in the development from a study in 30 weeks within one project. Study S7 checks **anomaly(fault) detection** in a web server response serving a Javas web app. S19 studies **technical debt (TD) paid-back(resolved)** in various Python applications from the Apache software ecosystem, including seven web apps, through a maximum of 11 years. S21 studies the evolution of **technical lag**, the measure of outdated JavaScript libraries in the npm network over eight years. Study S3 appears again because it also studies **Maintenance test**s in 8 web apps (a different study).

Some papers include cross-correlational studies besides the longitudinal ones in the same paper, and we only refer to the latter.

2.6.3.2 SRQ2 - Factors or the independent variables to influence the evolution of quality of web software

The most studied factors or independent variables in web application studies are (Code) metrics with five studies and code Changes with four studies. The includes/dependencies topic comes next, with three studies. Then, with two studies, we find Dead code, Security flaws/Vulnerabilities, and Code Smells. Finally, with one study each, we find the topics: Function/class usage/survival, Commits, Test changes, Aspect Oriented interfaces, Server anomaly/fault detection, Security levels, Defects/bugs, and TD (Technical Debt).

Studies S1, S2, S6, S14, and S19 use Software Metrics as independent variables. Study S1 uses software metrics, including complexity metrics, to study the evolution of 30 PHP web apps and verify Lehmann Laws. S2 measures various metrics (function usage, function

Table 2.5: Factors or independent variables in the study

Factors	#	%	Primary Studies
Metrics	5	19%	S1,S2,S6,S14,S19
Code changes	4	15%	S11,S15,S16,S18
Includes/dependencies	3	12%	S20,S21,S2
Dead code	2	8%	S5,S2
Security flaws/Vulnerabilities	2	8%	S9,S12
Code Smells	2	8%	S10,S13
Function/class usage/survival	1	4%	S2
Commits	1	4%	S3
Test code changes	1	4%	S3
Aspect-Oriented interface changes	1	4%	S4
Web server metrics	1	4%	S7
Security levels	1	4%	S8
Defects/bugs	1	4%	S17
Technical Debt	1	4%	S19

removal, library usage, interface stability, classes invasion, evolution of complexity) to test if systems have Systematic maintenance. S6 measures Software Metrics: Chidamber and Kemerer Suite (group of metrics), Lines of Code Metrics, Class Metrics, and 3 SNA metrics to infer the development phase (agile practices related). S14 again uses metrics to verify the Laws of Evolution in JavaScript's web apps. Finally, S19 uses metrics (among others) to explain the TD paid-back in Python applications.

Code Changes are used as independent variables or factors in studies S11,S15,S16 and S18. S11 measures JavaScript changes in top Alexa sites to explain potential bugs and vulnerable libraries. S15 and measure code changes to explain privilege protection changes in a PHP web app. S16 measures only part of the code changes (the code changes with privilege protection changes) to detect the protection impact changes. S18 quantifies metrics (number of changes per version and schema size - number tables) of the database of web apps to justify the laws of evolution.

Studies S20, S21, and S2 use **Includes/dependencies** as factors that influence the evolution of web apps. S20 measures JavaScript library includes, and S21 measures releases of the package and dependencies in JavaScript npm to describe the technical lag.

Studies S5 and S2 use **Dead code** as independent variables. S5 studies the evolution of a PHP web app's dead code to describe the dead code elimination. S2 also uses dead code (among other metrics) to test if 5 PHP systems have Systematic maintenance.

Security flaws/Vulnerabilities are used in studies S9 and S12 as independent variables. S9 measures security flaws to explain PSP - property satisfaction profiles in a PHP web app for six years. S12 computes vulnerabilities in packages and libraries in Python libs from the PyPI package repository (mostly web).

Studies S10 and S13 use **code smells** as independent variables. Study S10 defines and measures five violations (MVT smells) in the evolution of one Django (python) based system. S13 measures code smells to infer the evolution and survival by type and period on 4 PHP web apps.

Other independent variables used in only one study are *Function/class usage/survival* in study S2; *Commits* in S3; **test changes** in S3; **Aspect oriented (AO) interfaces** changes in S4; and **Server metrics** (server response time) are used in study S7, to explain server anomalies. **Security levels** are used in S8. **Defects/bugs** are measured in S17 (later to model defect prediction) and **Technical Debt (TD**) is measured in s21 to infer TD paid-back (resolved).

2.6.3.3 SRQ3 - Techniques for unevenly time spaced data or time series techniques

Table 2.6: Studies use of timeseries in the data

data	#	%	Primary Studies
timeseries /data based	10	48%	S5,S6,S7,S11,S12,S13,S17,S18,S20,S21
versions/releases	9	43%	S1,S2,S4,S8,S9,S10,S14,S15,S16
commits	2	10%	S3,S19

Table 2.6 shows the use of time series, versions, or commits in the studies. Studies S5, S6, S7, S11, S12, S13, S17, S18, S20, and S21 use times series. Studies S1, S2, S4, S8, S9, S10, S14, S15, and S16 use versions/releases of the software (the term versions in the context of the release means a public versioned release, e.g., 1.1.1. Finally, studies and S3 and S19 use commits.

The studies that use "versions/releases" do not use dates in the series but label them with a release number or a similar one.

We intended to discover if any studies use irregular time series (time series that do not happen at regular intervals) techniques, but none mention this.

2.6.3.4 SRQ4 Is collected data made available for replication?

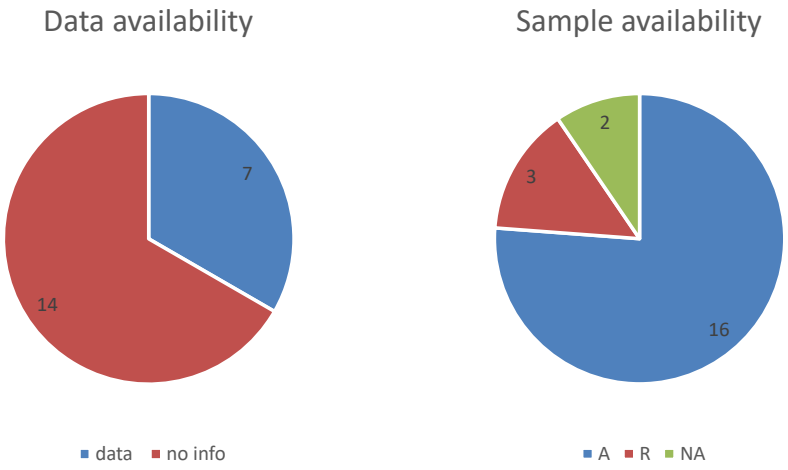
In this question, we investigate whether the applications used in the sample are available (e.g., open-source) and if the study data is available on a site or platform.

Figure 2.3 shows the replication possibilities. The left figure, 2.3a, shows data directly available for further studies or for replication. Seven studies (33%) published the data in some form on the internet or platform (S1, S3, S11, S13, S18, S19, and S21), while the other 14 studies do not have information in the paper about data availability (67% of the studies).

The right figure 2.3b shows the availability of the applications or possibility of reconstruction of the sample used in the studies. Sixteen studies (76%) use open source or free applications, so even if no data is available, it is possible to reconstruct the sample and after the data. Three of the studies (S11, S12, and S20), making 14% of the studies, do not have the sample (the applications are not available to download) but disclose the methods to recreate the sample, so in theory, it is possible to recreate the sample by following the steps in the study method. For the two remaining studies (10%), S5 (an industrial project) and S10 (a web app used in universities), we can not use the same applications.

2.6.3.5 SRQ5 - Analysis techniques used in the study

This section presents the statistical or other techniques used in the primary studies.



a Study data availability

b Sample availability

Figure 2.3: Data replication possibilities of the studies

Table 2.7: analysis techniques

Techniques used	**Studies**
descriptive statistics	S2,S4,S5, S8, S9, S10, S11,S18,S20,S21
inferential statistics	S1, S2,S6,S7,S12, S13, S14, S15, S16, S17,S20,S21
analysis tool	S19

Table 2.7 shows the analysis techniques used in the study. Twelve studies (more than half) use inferential statistics, ranging from linear regression, Mann -Kendal test, probability distributions, survival, log-rank, correlation, ARIMA, and auto-correlation. In addition, ten studies use descriptive statistics, and one study (S19) uses a tool to calculate (SonarQube).

2.6.3.6 SRQ6 - Methodology used

This section presents the methodology used in the primary studies. The experiments were classified as such by the authors.

Table 2.8: Methodology

Methodology	Studies
observational	S1-S5, S8-S21
experiment	S5,S6,S7

Table 2.8 represents the methodologies type used in the studies. Three studies, S5, S6, and S7, are experiments with evolution, while the remaining studies are observational. According to [82], studies can be one of the following four types: Quantitative Empirical Studies(no specific type), Correlation(observational studies), Surveys, and Experiments.

The observational evolution studies in the SLR typically consist of downloading software releases and taking measurements directly or measuring applied statistics. Experiments include

the treatment change (the independent variable) between groups. When both are allocated randomly (the independent variable and the groups) is a true experiment; when the researcher chooses not randomly, it is a quasi-experiment.

2.6.3.7 SRQ7 - Period of the study

In this section, we present the duration of the studies.

Table 2.9: Average study observation period (years)

Range	Studies
10 or more	S1,S2,S12,S13,S14,S18,S19,S20
5 to less than 10	S8,S9,S17,S21
1 to less than 5	S6,S10
less than 1	S5,S7,S11

Table 2.8 reveals the period of the evolution studies. Eight studies, S1, S2, S12, S13, S14, S18, S19, and S20, use ten years or more; 4 studies, S8, S9, S17, and S21, use a period of 5 to less than ten years; Study S6 and S10 are on the period of 1 to less than five years. Studies S5, S7, and S11 have a duration of less than one year. Finally, studies S3, S4, S15, and S16 are version based with no exact info about the period.

2.6.3.8 SRQ8 - Languages used in the applications of the study?

In this section, we present the languages used in the web apps that are the object of the studies.

Table 2.10: Languages used in the studies

Language	# Studies	Studies
PHP	8	S1,S2,S5,S8,S9,S13,S15,S16
Java	4	S3,S4,S6,S7
JavaScript	4	S11,S14,S20,S21
Python	3	S10,S12,S19
Various	2	S17,S18

Table 2.10 shows the most used languages in the study. With eight studies, PHP is the most studied language in web applications evolution studies. Java and JavaScript follow with four studies; next is Python with three languages. Two studies include multiple languages.

Figure 2.4 shows the percentages of the languages studied. As the table indicates, PHP is found in 38% for the studies, Java and JavaScript in 19%, and Python in 14%. These results are consistent with the use of web languages for server-side development [2] and client-side development [3].

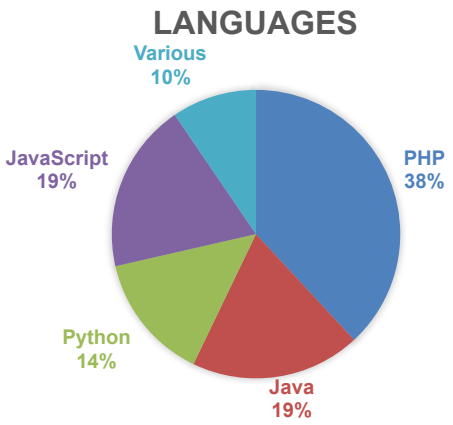


Figure 2.4: Languages used in the studies

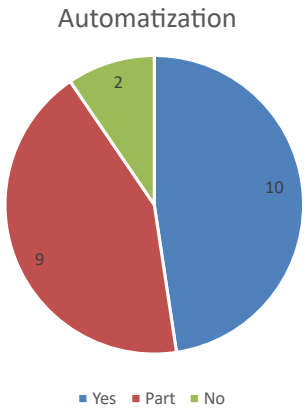


Figure 2.5: Automatizing

2.6.3.9 SRQ9 - Is automatizing of the study possible?

Figure 2.5 examines the automation or possibility of automation for the studies. Ten studies claim they have an automatic process (S1, S3, S8-S11, S13, S14, S18, and S19), while the others have partial automation, except for two studies (S6 and S7). Most studies with partial automation can achieve full automation with some effort.

2.6.3.10 SRQ10 - Does it test real cases? Number of applications in the study

According to the data extracted, all the studies use real cases. We also investigate the number of applications used in the studies.

Table 2.11 shows the number of web applications used in the studies. 1/3 of the studies use only one web app/system, and another third use more than 15 web apps. The rest of the studies are divided as follows: studies with 2-3 web apps with 10%; studies with 4-5 web apps with 14%; studies with 6-10 web apps with 10%; and no study uses the interval between 11 and 15 web apps.

[2] https://w3techs.com/technologies/history_overview/programming_language
[3] https://w3techs.com/technologies/history_overview/client_side_language/all

Table 2.11: Number of apps used in the studies

Num. Apps	Studies	Count	Percentage
1	S5,S6,S8,S9,S10,S15,S16	7	33%
2-3	S7,S17	2	10%
4-5	S2,S13,S19	3	14%
6-10	S3,S18	2	10%
11-15		0	0%
>15	S1,S4,S11,S12,S14,S20,S21	7	33%

2.6.3.11 SRQ11 - Study and conclusions: general or domain-dependent

Studies can be general domain if they can be applied to another language. On the contrary, they are considered domain-specific if they possess some characteristic that can only be used in a language or a method.

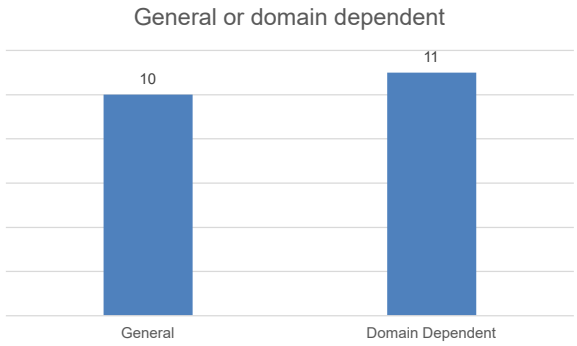


Figure 2.6: Study: general or domain specific

The chart 2.6 shows that almost half of the studies are domain-specific, and almost half are general domain studies that can be applied to other languages or applications.

Table 2.12: General or domain specific studies

Domain	Studies
General	S1,S2,S7,S11,S12,S13,S14,S17,S18,S20
Domain-dependent	S3,S4,S5,S6,S8,S9,S10,S15,S16,S19,S21

Table 2.12 shows the general or domain-specific studies. General studies include S1, S2, S7, S11, S12, S13, S14, S17, S18, S20, while dependent domain studies consist in the S3, S4, S5, S6, S8, S9, S10, S15, S16, S19 and S21 studies.

2.7 Discussion

2.7.1 Main findings

In this section, we present the main findings. For RQ1 - What are the software evolution studies with web software? – we presented the SLR method used to select 21 studies (19 by employing

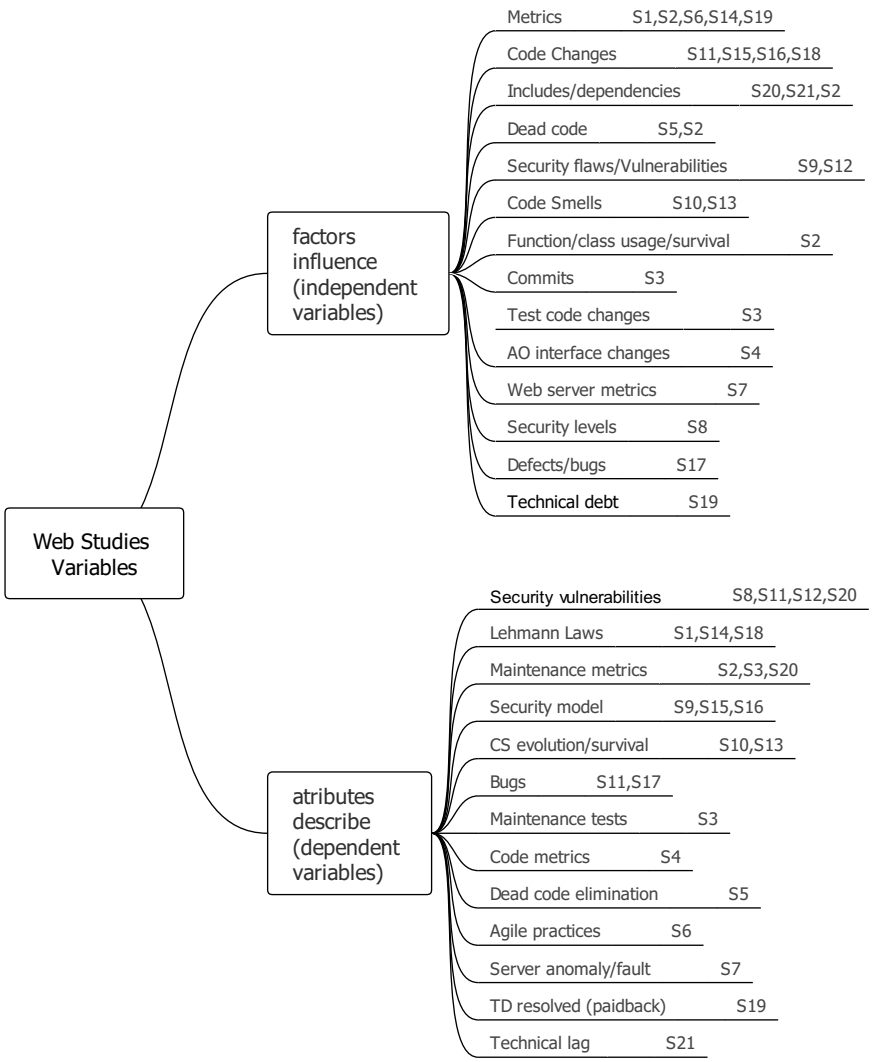


Figure 2.7: Studies Variables

the search string plus two by snowball), summarized in the results section and on a table in Appendix 1.

The aim of "RQ2 - What causes problems in web software evolution quality?" was to discover the independent and dependent variables in the evolution studies. Figure 2.7 summarizes these variables. The factors that influence evolution are the independent variables, while the attributes that describe evolution are the dependent variables. We found 13 different factors and 13 different attributes.

When investigating the question *RQ3 - How can we deal with unevenly time-spaced software development data?* We found that none of the studies investigates the irregular nature of the releases (the interval between releases is not regular). However, some studies use regular time series, others use releases/versions without dates, and others use commits.

Figure 2.8 shows the percentage of studies using time series, releases/versions (labels only), and commits. From the select studies, 48% of the studies use time series (48%), 43% of the

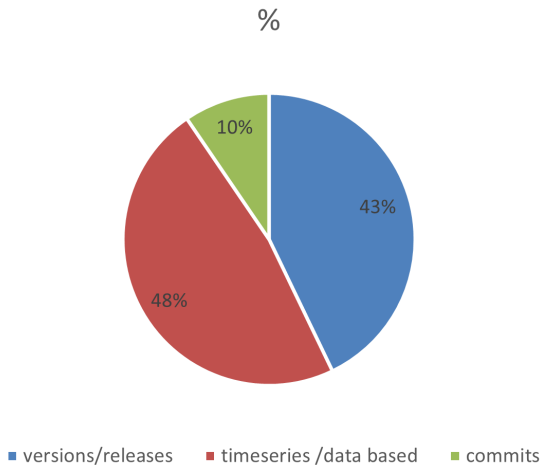


Figure 2.8: Percentage of studies using time series, releases, and commits

studies are based on releases, and only 10% of the studies use commits.

2.7.2 Other findings

Only 33% of the studies published the data on the web or repositories. However, 16 studies (76%) use open-source or free applications, and it is possible to reconstruct the sample used and, with that sample, collect the data. Thus, a third of the applications allow direct replication, but with some work, it is possible to reconstruct the sample and data in 3/4 of the studies. The *replication capability* is therefore 76%.

The techniques are divided primarily into inferential and descriptive statistics (automatic or not). One exception (study S19) uses a tool for all the measurements. The inferential statistical techniques are diverse and range from regressions, correlations, auto-correlations, ARIMA, survival, probability distributions, and trend analysis.

Regarding the methodology, three studies are experiments, and 18 are observational. Reports claim that experiments produce a higher quality of data than observational studies CITE. However, in some situations is not possible to conduct experiments, for instance, with evolution studies that aim to measure periods of 10 years or more.

The period used in the study ranges from less than one year to 10 or more years. However, the median of years studied is eight years.

We found that PHP is the most studied language in evolution studies using web apps. Java and JavaScript came next, and after Python. Two studies use various languages.

Almost all the studies achieve some form of automation except for two studies. Half of the studies claim they are fully automated, so extending the sample (usually the number of apps studied) would be more accessible. However, this question is more interesting if one extends or replicates the study. The authors probably made the automation to make the extraction and calculations effortless and not for the replication possibilities.

All the studies test real cases. The number of apps used ranges from 1 to more than 15, and the median is three apps for study. Half of the studies are domain-specific, while the other half can be interpreted as having general domain conclusions.

2.7.3 Strengths and weaknesses

2.7.3.1 Strengths of evidence

According to Atkins [5], four components can be analyzed for the strength of evidence: study design, quality of the studies, consistency of the studies, and directness of the studies.

Regarding study design, three studies are experiments, while the rest are observational studies. However, given that evolution studies are best made by observing the evolution of metrics, observational studies supported with real data and real samples provide a higher strength of evidence than an experiment with a reduced sample.

On the quality of the studies, when we selected the studies, we graded them qualitatively with eight parameters. In general, the methods were described in the studies, and so were the techniques. A third of the studies offer the data and results online, while the others (minus two studies that study closed-source apps) describe the sample and the methodology so they can be replicated.

We did not evaluate the consistency of the studies because they are heterogeneous. Regarding the directness of the studies, a third of the studies are concerned with security; however, some are concerned with security flaws, and others are concerned with permission problems. Some studies are concerned with the evolution of metrics, maintenance, Lehmann laws, and bugs, giving a wide range of possible directions.

As a conclusion for the strength of evidence, we need more studies in each identified area to evaluate the consistency and increase this strength.

2.7.3.2 Weaknesses of evidence

The following threads to validity are to consider [162].

Threats to construct validity concern the statistical relation between the theory and the observation, in our case, the review protocol, and bias in selecting the articles. This is addressed with the review protocol, which includes the research questions, search string and strategy, the filtering of the articles (phases and include and exclude criterion), and the data extraction method. First, in selecting the studies, we measured the kappa between the two investigators to minimize subjectivity. Then, in the backward snowball phase, we performed reference checking and repeated the initial phase to ensure all prior studies were selected. We used a form to answer questions and measure the concordance between investigators in the data extraction.

Threats to internal validity concern external factors we did not consider that could affect the investigated variables and relations. To minimize selection bias, we measured the second investigator's concordance with a sample of "maybe" papers he reviewed independently. There can be sample bias - samples too small - a third of the studies have one application. However, some samples have more than 15, and others have between 4 and 10. Regarding reporting bias, some studies only report statistically valid results when they are encouraged to share the study's data. Only a third of the studies share the data, but all but two of the studies give methods to rebuild the data, with minimizes this possible threat.

Threats to conclusion validity concern the relation between the treatment and the outcome. There can be a threat in data extraction. To minimize this threat, a sample of the select articles

was selected, and the second author extracted the data independently. Another threat can be in conclusion themselves. This was addressed by carefully discussing the conclusions and comparisons among all the authors.

Threats to external validity concern the generalization of results. A threat is the reduced sample size in some studies (a third have one app) that can exaggerate results. Another threat to the generalization is the number of languages studied. However, the most common languages in web development have studies in the SLR sample. Another threat is the reduced number of articles, reinforcing the conclusion that a reduced number of evolution studies exist in this area.

2.7.4 Meaning of findings

The studies were divided into seven main areas. The Laws of Evolution area mainly studied if the laws of software evolution stand for web but for different languages. For Security, the main aspects studied were the evolution of vulnerabilities and the evolution of vulnerable authorizations. In Maintenance and Bugs area studies, maintenance patterns and evolution of bugs are studied, each for one language. The same phenomenon happens for CS and TD evolution studies, where there are only a few studies (not comparable), and each study could be replicated for a new language. The studies related to the inclusion of libraries typically concern JavaScript libraries. Development techniques study mixed apps' metrics with process metrics. More studies of this kind need to be conducted for web evolution. Finally, server metrics area studies usually explore the web server logs to retrieve metrics, and there was only one study we could consider as an evolution study.

Generally, web evolution studies are in their infancy. This is because they appeared later than desktop application studies, for example, in Java. More evolution studies should exist to achieve a more significant comparison in each area found and even in areas not covered in the SLR. Most of the studies can be done for other web languages.

More studies should focus on the variables that developers/managers can control: Size metrics, Teams metrics, and Code Smells, thereby improving software quality.

2.8 Conclusions

The search string identified 933 candidate studies, from which 19 were selected. Snowball produced 37 candidates, and from them, two studies were selected. The studies were evaluated for quality, and 21 emerged as primary research studies subject to data synthesis.

The studies fell into seven areas of research: Laws of Evolution, Maintenance and Bugs, Development techniques, Server, Security, CS and TD, and Includes. The attributes or dependent variables studied in the selected papers were Lehmann Laws, Maintenance metrics, Maintenance tests, Code metrics, Dead code elimination, Agile practices, Server anomaly/fault, Security vulnerabilities, Security model, CS evolution/survival, Bugs, TD resolved (paid back) and Technical lag. The Factors or Independent variables that influence the evolution of the quality of web software are Metrics, Function/class usage/survival, Commits, Test changes, AO interfaces (changes), Dead code, Server metrics, Security levels, Security flaws/Vulnerabilities,

CS, Code Changes, Defects/bugs, Includes/dependencies, and TD. None of the studies use irregular series, but 48%

The SLR also revealed some secondary questions results: 33% of the studies have ready data for replication available on the web, but for 90% is possible to recreate the sample and the results. Tree of the studies are experiments, and the rest are observational studies (the most common for several years duration). To make the measurements and conclusions, more than half of the studies (12/21) used statistical methods; one of the studies used a tool, while the rest used an algorithm to measure. Most studies use periods of the years or more (hence the observational studies), while five studies are between 4 and 9 years, and three studies use less than one year. Four studies are evolution of releases based and do not disclose the period. The most studied language is PHP, followed by Java, JavaScript, and Python. Almost all the studies can be automatized, at least partially. All the studies use real cases; one-third use just one app, another third use more than 15 apps, and the rest use between 2 and 10 apps. Half of the studies have domain-specific conclusions, while the other half can be generalized.

A few studies use closed-source software. In those studies, we had to rely on the authors' descriptions.

2.8.1 Recommendations

Most software evolution studies are based on desktop applications (e.g., Java-based), so there is room for similar evolution studies and studies specific to web development evolution. In the web area, most evolution studies are on security, covering vulnerability and access privilege (authorization) issues. Studies covering CS and other technical debt topics, refactoring, testing, and evolution of persistence (databases and more) still need to be explored and only cover a few of the development languages used, both on the server and client sides.

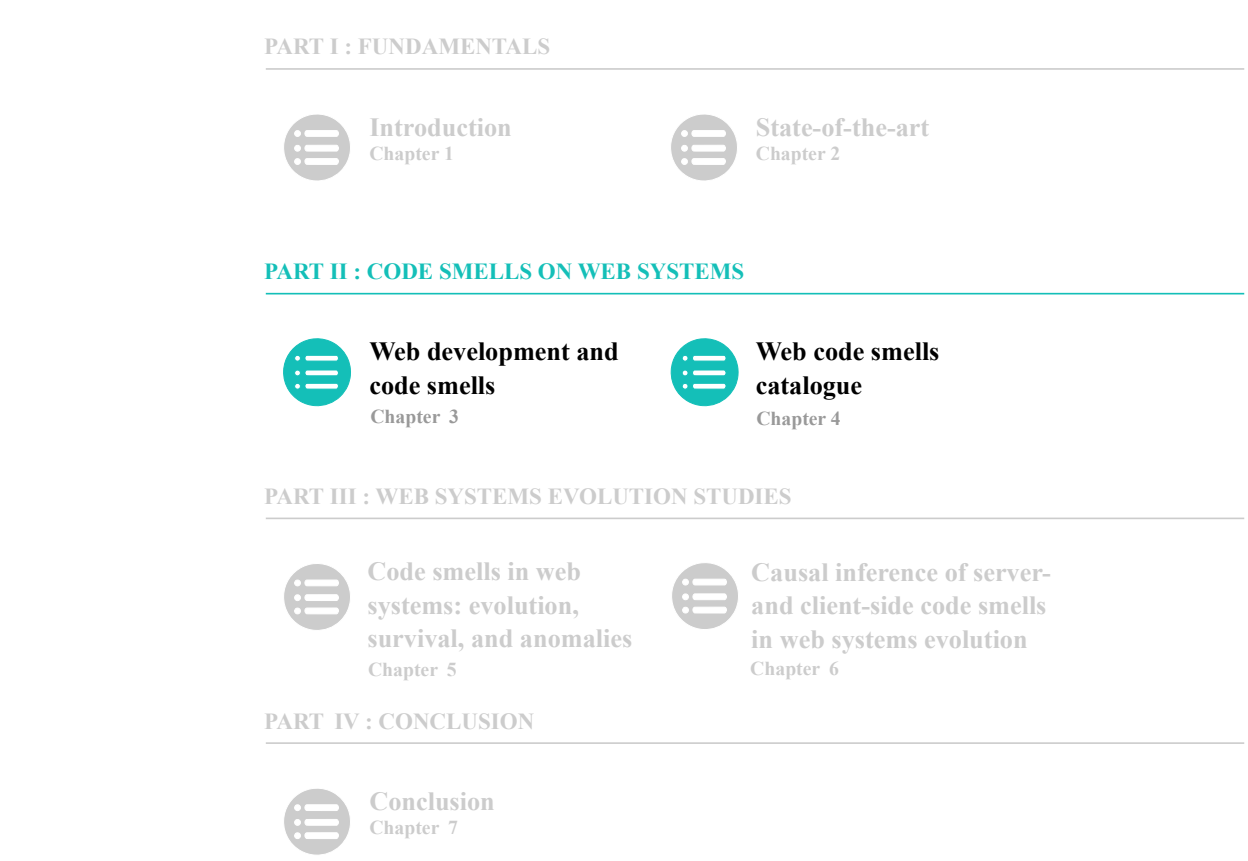
A significant percentage of studies have a low number of applications studied. It is more time-consuming to make evolution studies, i.e., to have several releases to be considered in each application on evolution studies. However, an effort could be made to have more applications in the studies to increase internal and external validity.

Almost all the studies are observational. There should be more experiences; however, as noted before, software evolution is a complex topic to experiment with and apply treatments to software and developers.

Most studies analyzed a small number of versions and did not consider the inherent non-seasonality of version releases. In other words, web app evolution studies should analyze more extended time series and consider using data analysis techniques suited to unevenly spaced time series (aka irregular time series). The studies with just versions/releases should characterize the series with dates.

Part II.

Code smells on web systems



This part includes the web development survey and the web code smells catalog characterization

Chapter 3

Web Development and Code Smells: an industry survey

Contents

This chapter presents the results of a survey to characterize the web development community, teams and projects and their knowledge of code smells.

3.1 Introduction

Web developers must deal with various programming languages that run in two environments, the client or browser and the web server. They must also possess other skills to build a web application, like communication with web designers and marketers. Around two decades ago, as web development evolved and matured, Software Engineering tailored for web development, coined as Web Engineering, became a mainstream topic [77, 99, 100, 108]. Some aspects of Software Engineering that are more important to study are the aspects the developer can change effectively, such as code smells. Code smells are not bugs but potential problems in code that can lead to further problems, such as poor comprehension, delays in implementation, and even bugs.

Code smells on the web are just in their infancy, and we intend to access their knowledge and level of implementation by developers with this survey. To this end, we divided the survey into four main areas. First, we characterize the developers' education and experience, then their teams and projects. The third part aims to understand the programming languages and tools used, and lastly, the referred knowledge of code smells in web apps or systems.

We implemented the survey in the Qualtrics tool [1]. Around 70% of the respondents were from the ISCTE School of Technologies and Architecture database of IT companies participating in the "FISTA"[2] recruiting event.

This chapter is divided as follows: section 2 presents relevant related work. Section 3 presents the methodology, and section 4 reports the results. Section 5 provides the discussion and section 5 ends with the conclusion.

3.2 Related Work

We did not find any survey on code smells in web applications. Therefore, this section shows related work separately on "web development" and "code smells", and a survey in Software Engineering (SE) practices, which is included because of the "tools" and "teams" group questions.

3.2.1 Web Developers Surveys

In [136], the authors describe a survey of web developers in which they collected over 300 responses from individuals with varying levels of experience and training, characterized as informal web developers. They report on web development projects, tool usage, development process, reuse, and learning and collaboration. In addition, they compare the responses of developers who self-identify as programmers with those who do not.

Continuing the survey report in [137], the authors report a subset of findings: The prototypical web developer from the sample is meticulous about the quality of the websites he/she produces and considers usability issues but neglect accessibility concerns; web developers have many similar interests regarding web applications or features such as authentication, databases, online surveys or forms; they value ease of use as the essential property of a web development

[1] https://www.qualtrics.com/
[2] https://fista.iscte-iul.pt/

tool but mention many other needs such as integration with other tools, robust code editing features, or WYSIWYG facilities.

3.2.2 Code Smells Surveys

In [167], the authors research if code smells are essential for developers and the reasons for it: Relevance of the underlying concepts, awareness about code smells on the developers' side, and the existence of appropriate tools for code smell analysis or removal. Finally, they report on the results obtained from an *exploratory survey* involving 85 professional software developers, aiming to understand better the knowledge and interest in code smells and their perceived criticality: A significant 32% of respondents were not aware of code smells. Technical blogs, forums, colleagues, and industry seminars were cited as primary information sources, suggesting researchers should target these channels for dissemination. Opinions on the criticality of code smells were divided, with the majority being moderately concerned. Duplicated code, large class, long method, and accidental complexity were the most frequently mentioned smells and anti-patterns. Respondents expressed a need for better tools, specifically user-friendly, real-time support for code inspections and refactoring.

The article [115] represents an empirical investigation combined with a direct survey, aimed at empirically establishing the perceptions of developers regarding code smells. This study involved presenting developers with code entities, both afflicted and unafflicted by 12 identified code smells, from three different systems. The developers were then tasked with identifying if the code was potentially problematic in terms of design and, if so, characterizing the nature and severity of the issue. The study involved both 10 original developers from the three projects and 14 outsiders, namely industrial developers and Master students. The results were:(I) some smells related to object-oriented programming practices are not perceived as design problems; (II) the problem's "intensity" influences whether a bad smell(another denomination for CS) is problematic; (III) smells related to complex/long source code are considered a significant threat by developers; and (IV) developers' experience and system knowledge play a crucial role in identifying some smells.

Expanding on the findings of the two preceding studies, [154] conducts two survey studies, each engaging a group of highly experienced developers - 63 in the first and 41 in the second. These studies aim to evaluate the developers' perceived criticality of code smells and their proficiency in identifying such smells within the code they examined. Results show developers are very concerned about code smells in theory, nearly always considering them as harmful or very harmful (17 out of 23 smells). However, when they were asked to analyze an infected piece of code, only a few infected classes were considered harmful, and even fewer were considered harmful due to the smell. The authors conclude that code smells are perceived as more critical in theory but less critical in practice. Developers' perceptions change over time, so researchers should periodically investigate these perceptions to better understand their impact on development practices.

3.2.3 Other surveys

The survey [59] reports on the type of SE practices in the Turkish software industry, with 202 practicing software engineers. Some findings: 54% of the participants reported not using any software size measurement methods, while 33% mentioned that they had measured lines of code (LOC); In terms of effort, after the development phase (on average, 31% of overall project effort), software testing, requirements, design and maintenance phases come next and have similar average values (14%, 12%, 12%, and 11% respectively); Respondents experience the most challenge in the requirements phase; Waterfall, as a rather old but still widely used lifecycle model, is the model that more than half of the respondents (53%) use. The following most preferred lifecycle models are incremental and Agile/lean development models with usage rates of 38% and 34%, respectively; The Waterfall and Agile methodologies have slight negative correlations, denoting that if one is used in a company, the other will be less likely to be used.

3.3 Methodology

Our survey aims to assess four key areas in web development and code smells for web projects: the education and experience of web developers, the development characteristics of the teams and projects, languages and tools used, and finally, their knowledge of code smells.

Section "Education and Experience of Web Developers" focuses on understanding web developers' educational background and professional experience. It includes questions related to their degrees, years of experience in the industry and areas of expertise. We also ask the country where they work and their relation to a company or not. Gathering this information will help to identify the broad skill set and qualifications of web developers in the market.

Section "Development Characteristics of Teams and Projects" intends to characterize various aspects of web development teams and projects. The questions include evaluating team and project sizes or durations, the composition of groups involved in web development projects, and whether the work is framework based or not. These questions aim to illuminate the web projects' distinct factors.

"Languages and Tools Used" delves into the technologies, programming languages, and tools web developers use to build and maintain their projects. The questions in this area inquire about preferences and reasons for choosing specific languages or tools, as well as developers' level of proficiency with them. This information will provide insight into the industry's most popular and effective technologies and help identify emerging trends.

"Knowledge of Code Smells" explores web developers' awareness and understanding of code smells. These potential code issues could lead to future problems or difficulties in maintaining the software. Questions focus on their ability to recognize code smells, strategies for addressing them, and the impact of code smells on the overall quality of the software. These questions will help to gauge the importance of code quality and best practices among web developers and identify areas for improvement.

The survey had two phases, one in 2021 and another in 2022. The first phase was to the industry developers through direct contact and ISCTE/ISTA base of contacts. The second

was through former students in various universities and Linkedin groups, although the last produced fewer results than the first.

Next, we describe the complete questions used in the survey.

3.3.1 Survey questions

3.3.1.1 1st part: Education and Experience of Web Developers

Q1 What is your level of education? (option):
1- High school or lower; 2-Technical / Professional (please fill in the duration in years); 3-Bachelor / Licentiate (please fill in the duration in years); 4-Master; 5-PhD or Higher.

The question "level of education" aims to agree or not with the perception of a self-taught web developer [136], or a more educated web developer from current days, where all programs are connected to the web.

Q2 What is the country that you are based in? (open)

Q3 How many years of experience do you have in Web development? (number)

This question will help gauge the experience level of web developers in the field.

Q4 What can best describe you as a developer? (check all that apply):
a. student; b. professional within a company; c. freelance; d. other (please specify)
While some developers work in companies, others work individually for several companies. This question aims to uncover web developers' various roles and work arrangements, highlighting the differences between those working in companies, as freelancers, or in other capacities.

Q5 Which of the following best characterizes you?
a. Full-stack developer; b. Only Client developer (HTML/CSS/JS) ; c. Only Server developer; d. Web services developer; e. other (please specify)
This question seeks to classify respondents based on their specific area of expertise within web development, allowing for a better understanding of the distribution of skills among developers.

3.3.1.2 2nd part : Development Characteristics of Teams and Projects

Q6 How do you characterize your Web development production as a whole? - slider: Open-source / Proprietary
This question aims to understand the balance between open-source and proprietary production in the web development work of respondents, shedding light on the prevalence of each approach.

Q7 How do you characterize your Web development production? - slider
No framework / Fully Framework Based
This question seeks to gauge the extent respondents rely on frameworks in their web development work, from not using any frameworks to being fully dependent on them.

Q8 What is your average project duration in weeks? - slider
1-2 weeks 3-4 weeks 1-3 months 3-6 months 6 months or more
This question aims to identify the typical duration of web development projects, offering

insights into the varying scales and complexities of these projects.

Q9 What is the average size of your team? -slider
1 2 3-5 6-10 10 or more
This question seeks to determine the average team size for web development projects, providing a perspective on the web development different team structures.

Q10 What types of people are in your team in a typical Web project?
Software Engineer / Web Developer; Creative Designer / Web Designer; Business Expert / Marketer; Team Manager; Domain Expert / Area Specialist; Other
This question focuses on understanding the composition of teams involved in web development projects, highlighting the various roles and expertise required to complete a project successfully.

3.3.1.3 3rd part: Languages and Tools Used

Q11 What is your experience – in number of years - in the following server-side languages? multiple - slider
for every language the 5 intervals: 0 1 2 3-4 5 or more
languages: PHP; C#; Ruby; Java; Scal; Python; JavaScript (nodejs); PERL ;Other.
This question seeks to evaluate respondents' experience with various server-side programming languages, providing insights into their proficiency and the popularity of each language.

Q12 What is your experience – in number of years - in the following client-side languages? multiple sliders for every language the 5 intervals: 0 1 2 3-4 5 or more
languages: JavaScript; CSS; HTML; ActionScript; Other.
This question aims to assess respondents' experience with different client-side programming languages, offering insights into the level of expertise and the popularity of each language in the field.

Q13 What tool types do you use for Web development? (choice)
Requirements / Change management; Project Planning / Project tracking; Version Control; Issue tracking; Testing (unit or functional testing); Code quality (code review, code smells, metrics, etc) ; Other(s)-separate by comma.
This question identifies web developers' various tools for the development process, such as project management, version control, and testing. Understanding the tools can offer insights into the industry's most influential and widespread solutions.

3.3.1.4 4th part : Knowledge of Code Smells

Q14 How familiar are you with code smells? - slider
I have never heard of them ; I have heard about them, but I am not so sure what they are ; I have a general understanding, but do not use these concepts; I have a good understanding,

and use these concepts sometimes; I have a strong understanding, and use these concepts frequently.
This question aims to gauge respondents' familiarity with code smells, which can provide insights into the general awareness of code quality and best practices in the industry.

Q15 Do you understand clearly the difference between code smells and adherence to code standards? - slider
Don't understand; Fully understand
This question seeks to determine whether respondents can differentiate between code smells and code standards, highlighting their understanding of these essential concepts in maintaining code quality.

Q16 How concerned are you with the presence of code smells in your code?
Extremely important Very important Moderately important Slightly important Not at all important
This question aims to assess respondents' level of importance in addressing code smells in their code, offering insights into the priority given to code quality.

Q17 Please briefly motivate your previous answer
This open-ended question allows respondents to explain their reasoning behind the importance of code smells, providing additional context for their perspective on code quality.

Q18 Please briefly describe 3 situations that you would consider a code smell in Web development - in server-side or client side development. If you are not sure please check code smells1 (link) or code smells2(link) for code smells in other areas - (open-3 text boxes)
This question seeks specific examples from respondents about code smells they have encountered in web development, offering insights into common issues and their understanding of the concept.

Q19 Would you be interested in participating in a more detailed survey about code smells/refactoring for the Web? If yes, please leave your email address here, so we can contact you:
This question gauges respondents' interest in participating in further research on code smells and refactoring in web development, allowing for a more in-depth exploration of these topics.

3.4 Results

This section presents the results of the survey.

3.4.1 1st part: Education and Experience of Web Developers

Q1 What is your level of education?

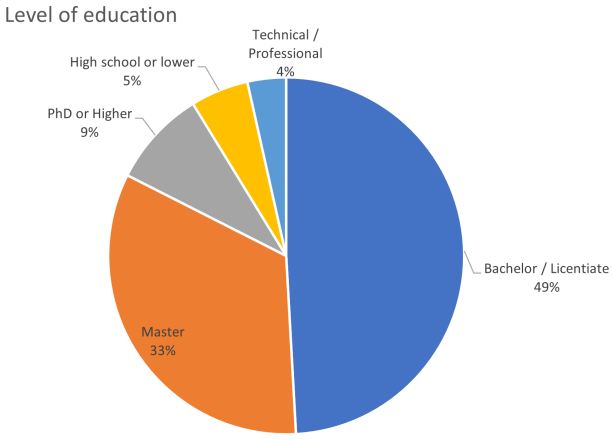


Figure 3.1: Level of Education

Graph 3.1 shows the percentages, where half of the respondents have a Bachelor's or Licentiate degree, and one-third have a Master's degree.

Q2 What is the country that you are based on?

We had 57 respondents organized as follows: Portugal 75% , Brazil 16%, Denmark 4%, Ukraine 2% and Spain 2%. One answer was invalid.

Q3 How many years of experience do you have in Web development?

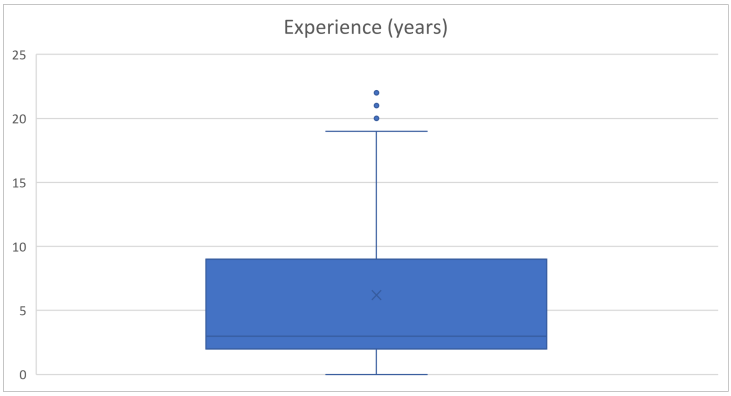


Figure 3.2: Years of Experience

Figure 3.2 shows the experience of developers in years. The more recurrent experience is 1-3 years, but 6 and 4 years follow next. There are, however, developers with 20 years of experience.

Q4 What can best describe you as a developer?

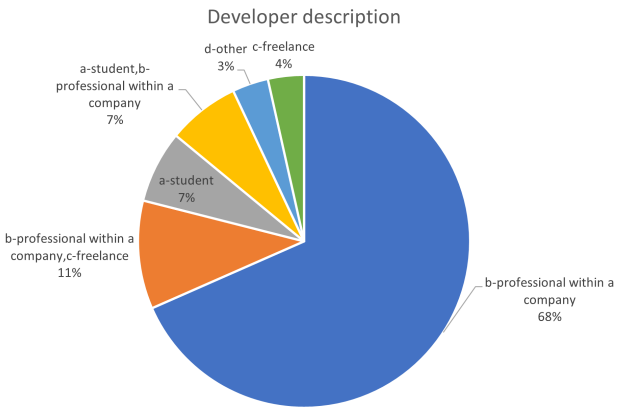


Figure 3.3: Developer description

Figure 3.3 shows the best description for developers. Two-thirds of developers are professionals in a company, while 11% are also professionals but do freelance work. 7% are students only, and the other 7% are students who are professionals in a company. So professional in a company sums up to 86% of the respondents. Freelance are only 4%.

Q5 Which of the following best characterizes you?

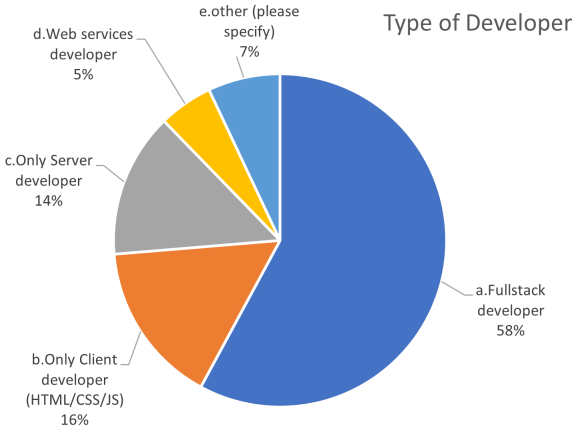


Figure 3.4: Developer Type

Figure 3.3 shows the best description for developers. Two-thirds of developers are professionals in a company, while 11% are also professionals but do freelance work. 7% are students only, and the other 7% are students who are professionals in a company. So professional in a company sums up to 86% of the respondents. Freelance are only 4%.

3.4.2 2nd part : Development Characteristics of Teams and Projects

Q6 How do you characterize your Web development production as a whole?

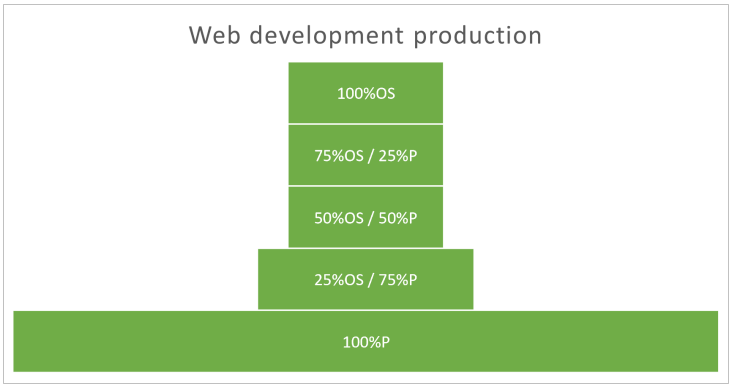


Figure 3.5: Web development production: open source/proprietary

Figure 3.5 reveals if web production is proprietary or open-source. In the appendix, there is a table with the values. The developers in this survey develop mainly proprietary code 51% while the other cases are divided equally. 11% of developers develop only open-source code.

Q7 How do you characterize your Web development production ?

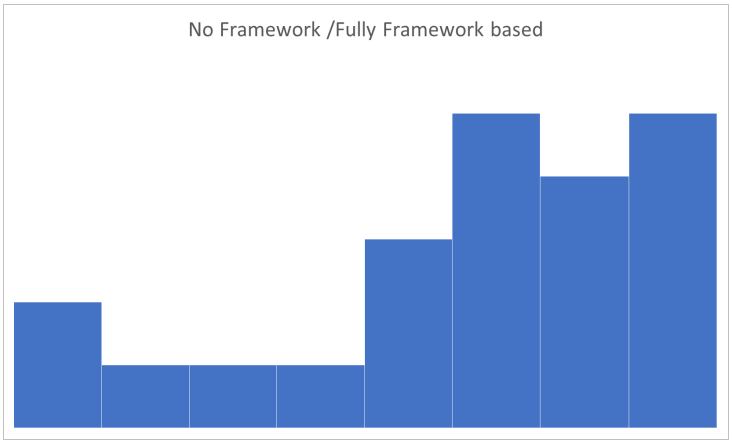


Figure 3.6: Web development : no framework/ framework based

Figure 3.6 shows the distribution of not using or using frameworks in the development. Most developers use frameworks, but some developers still do not use frameworks. In numbers 51% only do development with frameworks and 31% use mix (75% frameworks and 25% no frameworks).

Q8 What is your average project duration in weeks?

Figure 3.7 describes the average project duration. More than half of developers (55%) say projects take more than six months, while 20% say 3-6 months, 12% say 1-3 months, and 12%

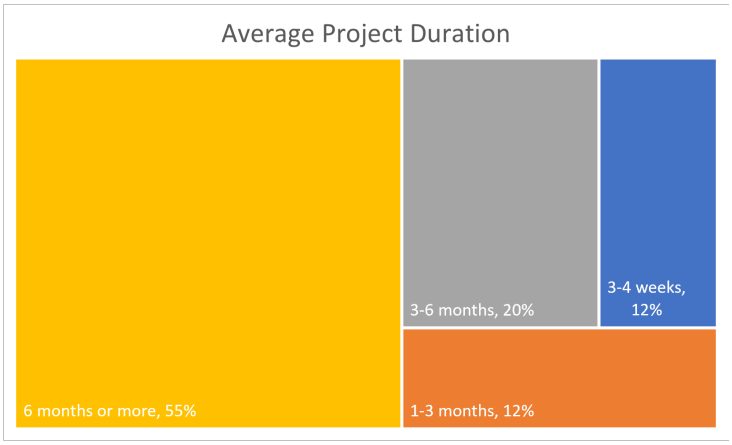


Figure 3.7: Web development average project duration

say 3-4 weeks.

Q9 What is the average size of your team?

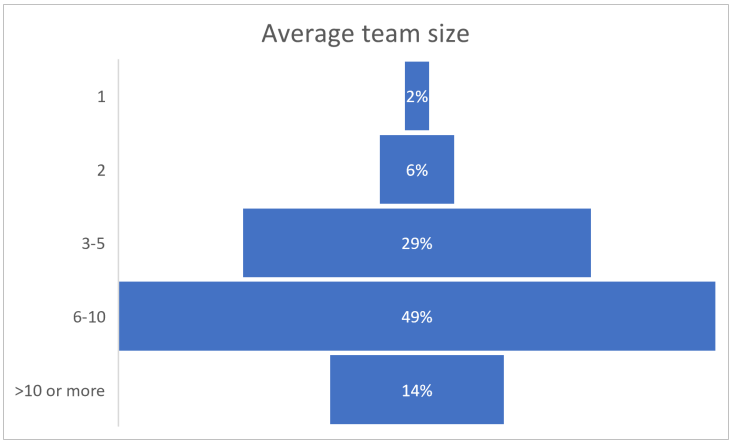


Figure 3.8: Web development team size

Figure 3.7 reveals the developers' team size. Half (49%) have teams of 6 to 10 people. 29% have teams of 3 to 5 people. 14% are in teams of more than ten people. The remains are the smaller teams, 6% have teams of two people, and 2% develop alone.

Q10 What types of people are in your team in a typical Web project?

Table 3.1: Top combinations in Web development team constitution

team constitution	percentage
Software Engineer/Web Developer	18%
Software Engineer/Web Developer,Creative Designer/Web Designer,Team Manager	12%
Software Engineer/Web Developer,Other	10%
Software Engineer/Web Developer,Team Manager	8%
Software Engineer/Web Developer,Business Expert/Marketter,Team Manager	8%
Software Engineer/Web Developer,Creative Designer/Web Designer,Business Expert/Marketter,Team Manager	8%

Table 3.1 shows the teams' top constitution (full list in the appendix). Most teams (18%) have Software Engineer/Web Developer only. In second place of the more recurrent teams constitution (12%) are the teams formed by three types of people: Software Engineer/Web Developer, Creative Designer/Web Designer, and Team Manager.

3.4.3 3rd part: Languages and Tools Used

Q11 What is your experience – in number of years - in the following server-side languages?

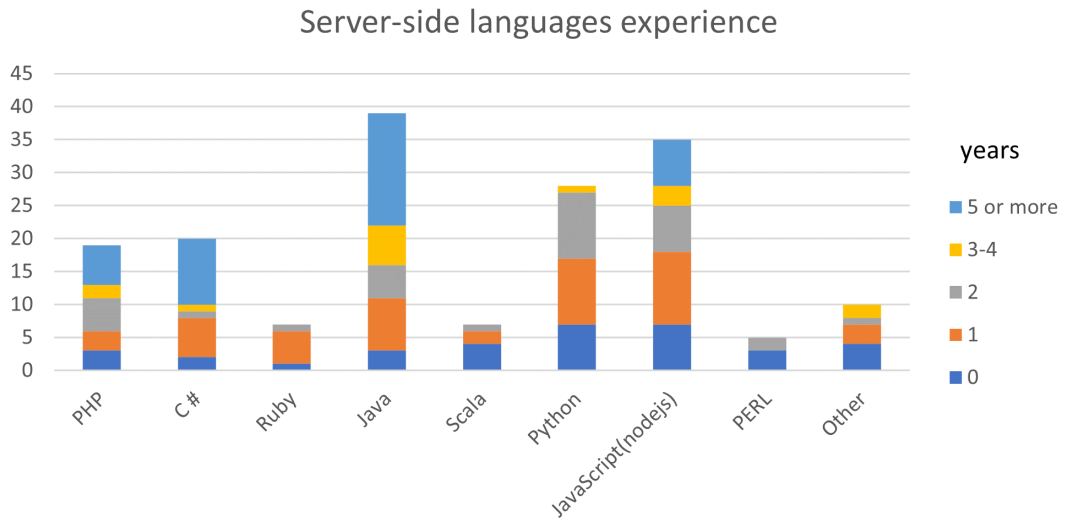


Figure 3.9: Server-side languages experience

Figure 3.9 shows the server-side languages experience. The language with more developers' experience is Java, with five years or more experience. After comes JavaScript (nodejs), Python, C# and PHP.

Q12 What is your experience – in number of years - in the following client-side languages?

Figure 3.10 shows the client-side languages experience. In client-side only or full-stack development, there is no way around it, and the developer must know HTML, CSS, and JavaScript, which is confirmed in the chart. Some developers still know ActionScript (the language used in Flash, similar to JavaScript), while others know other languages, namely TypeScript.

Q13 What tool types do you use for Web development?

Table 3.2 describes the most used tools. Because of the high number of combinations, we only show the percentages and not the combinations. More than 60% use version control tools, Project Planning/tracking, Testing, Code quality, and Issue tracking. Around half use

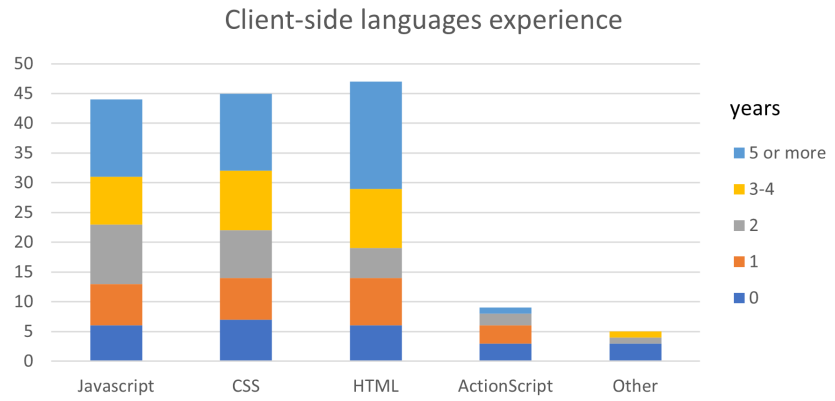


Figure 3.10: Client-side languages experience

Table 3.2: Tools used

Tools	Number	Percentage
Requirements / Change management	25	51%
Project Planning / Project tracking	35	71%
Version Control	43	88%
Issue tracking	30	61%
Testing (unit or functional testing)	35	71%
Code quality (code review, code smells, metrics, etc)	33	67%
Other(s)	1	2%

Requirements/Change management. This result means there is systematic development with great tool support.

3.4.4 4th part : Knowledge of Code Smells

Q14 How familiar are you with code smells?

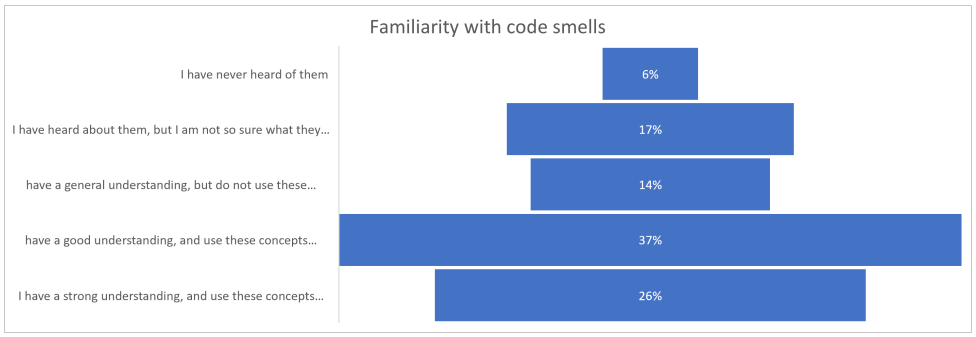


Figure 3.11: Code smells familiarity

Figure 3.11 describes developers' familiarity and use of code smells. Roughly two-thirds use the concepts of CS, while the remaining third do not use them. As for the understanding of

what code smells are, around 23% do not know what code smells are.

Q15 Do you understand clearly the difference between code smells and adherence to code standards?

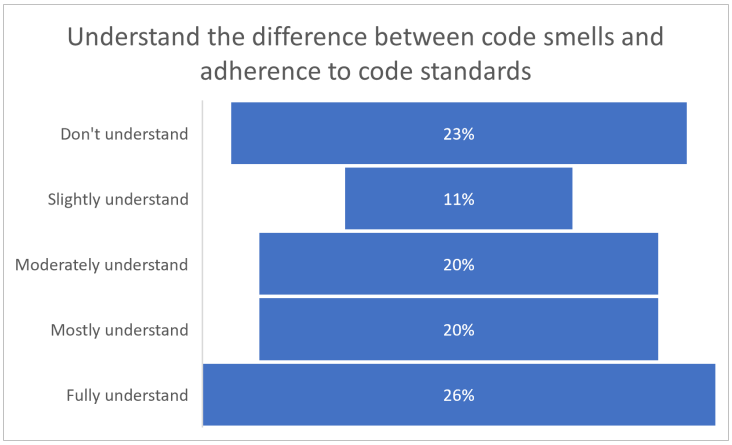


Figure 3.12: Code smells / Code standards

Figure 3.12 measures if the developers know the difference between code smells and code standards. 26% fully understand the difference, while 23% do not understand. The middle field is around half of the developers (20% mostly understand, 20% moderately understand, and 11% slightly understand the difference).

Q16 How concerned are you with the presence of code smells in your code?

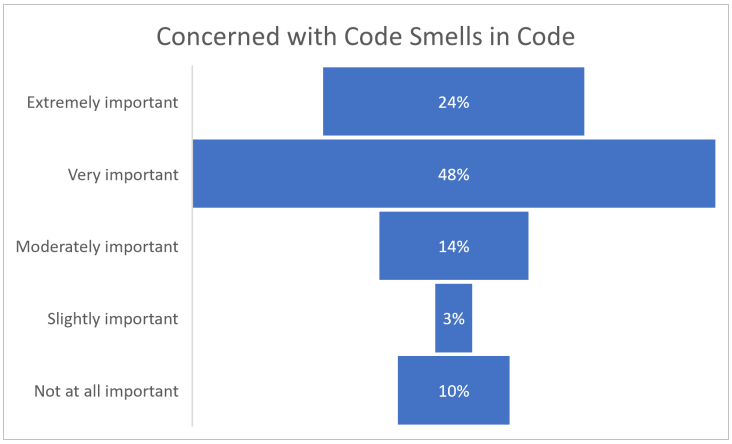


Figure 3.13: Concern with CS

Most developers are concerned with the presence of code smells in code, as shown by 3.13, and 24% consider them extremely important while 40% consider them very important. Still, 14% consider CS moderately important and 3% slightly important. Only 10% consider CS, not at all important.

Q17 Please briefly motivate your previous answer

Just a few developers answered this question. A bigger percentage of answers were related to code quality (5). Other answers: Risky code/potential bugs/security flaws(2), the concern is using linters/sonarqube(2), code readability(2), increased code complexity(1), and slow release dates(1).

Q18 Please briefly describe 3 situations that you would consider a code smell in Web development - in server-side or client-side development.

As with the previous question, just a few developers answered this question. Therefore, we will divide the consideration in CS for server- and client-sides, server-side only, and client-side only.

CS that can happen on both sides of the code: Most of the smells appointed are *long function/method*, *too complex code*, and *too many parameters* . Following, we had duplicated code and comments. The remaining situations are just referred to once: *Difficult to test*, *variable naming*, *excessive coupling*, *Large file*, *mixture of languages*.

Ambiguous answers: *inconsistent data*(when accessing various times) and *inconsistent variable names between server and client*.

Large and god classes are the most referred to on the server-side. Additionally, *passwords in code*, *no constants*, *no single responsibility*, *inconsistent architecture between components* (example web services), *no data abstraction layer* (classes), and *excessive coupling* are also referred.

On client-side, only the situations referred to were: *Large component* (for example react), *!important on css*, *long time to load*, and *web app opens new tabs*. For client-side JavaScript only, too many async calls, writing different JavaScript for different browsers, and "any" type.

3.5 Discussion

Education and Experience of Web Developers - Half of the respondents have a bachelor's or licentiate degree, and one-third have a MSc degree. This survey developers' education contrasts with earlier web development surveys where education was not so high [136]. Most of the experience claimed is between one and eight years. Two factors can contribute to this number: they get promoted to team managers, and web development more recent than other software development. As for the constitution of the developer respondents, most are professionals within a company, and more than half are full-stack developers. This position can be explained by the following: even if a developer develops mainly on the server-side, he/she has to know about the client-side development. The contrary is not true (asking three specialists in the area with more than three years of experience), i.e., a specialist on the client-side code does not necessarily know how to develop server-side code.

Development Characteristics of Teams and Projects - Half of the developers in the survey develop proprietary code, while the remaining have a mixture of proprietary and open-source

code. Most companies use or even develop some framework to accelerate the code. In open-source (OSS), there are some well-conceived frameworks nowadays for web development [3]. The duration of more than half of the projects is longer than six months, 20% is 3 to 6 months, and the rest is less than three months. Web projects are longer than their desktop counterparts because of server-side and client-side development (they run on two platforms). Half of the respondents work in teams of 6 to 10 people, and 30% in teams of 3 to 5. We used the term "people" because they are not all developers; some are designers, marketers, team managers, and others. The survey [59] that deals with Software Engineering practices takes a look at teams and development models, but the questions are different.

Languages and Tools Used - We asked about the programming language experience of web developers: on the server-side, Java, Python, JavaScript(nodejs), C#, and PHP; on the client side, we already knew the answer, and there is no way around HTML, CSS, and JavaScript. The server-side question should be more specific, as we should have asked for language experience on the web server-side. This work experience is consistent with the statistics for the server-side [4] and client-side languages [5]. However, the answers for the server-side languages also include experience outside of the web, so the statistics do not correspond with the links given. Another reason for this non-correspondence is that more than 70% of the respondents work in industry, and some server-side languages do not have the same representation outside industry. In addition, most developers use Software Engineering tools to aid development, which are the most relevant in the field now. This was not the case in older surveys like [136, 137] almost two decades ago, as developers revealed gaps in tool support. Therefore, we can conclude that tool support dramatically increased.

Knowledge of Code Smells - Most developers are familiar with CS, but some do not apply them. Some do not understand the difference between CS and adherence to code standards. On the other hand, most developers think CS are very important or extremely important, and some developers can motivate the answers with problems that CS can cause. A group of developers could identify examples or situations of CS on the web. In conclusion, developers are aware of CS, but some do not apply refactorings, most because of a lack of time, especially on industry projects with specific timelines.

This is consistent with earlier studies about CS awareness in the desktop area, as [167], where most respondents were moderately concerned about CS in code. The most identified CS are similar to the most identified in the server-side code on the present survey. At the time, respondents complained about having more CS-related tools. In [115], not all CS are perceived as problems but smells related to complex/long source code are considered a significant threat by developers, which in the current survey were among the ones referred most. In a more recent study [154] found that CS are perceived as more critical in theory but less critical in practice (classes with some smells were not considered harmful), which is the same result that we found but for different reasons - in our study, the lack of time to refactor was the culprit.

Implications for investigators - Compared with older desktop CS surveys, the project tools are no longer an issue. However, we need CS detection and refactoring tools specially

[3] https://www.geeksforgeeks.org/top-10-frameworks-for-web-applications/
[4] https://w3techs.com/technologies/overview/programming_language/
[5] https://w3techs.com/technologies/overview/client_side_language/

tailored for web applications with server-side and client-side code and the mixing of this code in monolithic web apps (the majority of the web apps).

Investigators should conduct this or a similar survey for CS on the web within a greater audience, in the open-source and freelance world, and among students. We intend to repeat this survey to a broader audience. However, there is not so much information about CS for the web and why they should be avoided. There should be more studies about the problems web CS can cause in web projects and how to remove them from the code.

Implications for practitioners - Developers, in general, should write clean code and avoid CS. In web projects, this is more complex because of server-side CS, client-side CS, and the mixing of them in the case of monolithic apps. We can infer that most developers know what CS are and know they should avoid them. However, most of them do not have the time to remove or avoid them in the coding phase. Hopefully, with more impact and characterization studies, especially with more tools embedded in the IDEs , the future web code will contain fewer CS.

3.5.1 Limitations

The major limitation of the survey is the limited size of the sample of participants. We disseminate the questionnaire in two phases: one for developers in industry, with a contact database from ISCTE, and the other with professors from the academia, to contact former students working now in the area. We disseminated the survey in Linkedin groups related to web development, but we have yet to get many answers from this side.

3.6 Chapter conclusions

We presented a survey to characterize web developers from the industry. We asked for education and experience; most have a degree or even a master's and are experienced in web development. We raised information about the project duration and how web development teams are formed. Some of the teams are very heterogeneous, as expected. We studied the expertise in languages on the server and client sides and the tools used; again, they are very experienced and use the essential tool groups to aid in software development nowadays.

The developers in the survey have some knowledge of CS for the web, but most do not apply them. In web projects, we must count the server-side, client-side, and interaction CS between client and server code for monolithic applications or systems. In addition, we must also count some factors related to web servers (e.g., speed), security, and database assessment. The Web CS contrasts clearly with desktop development, where we only have CS for one language.

For the developers that responded to CS situations questions, the most referred situations were the CS related to excessive size, excessive complexity, and excessive coupling on the server-side - some of this CS can refer to JavaScript client-side also. Client-side examples were: large components that took a long time to load. Client-side JavaScript referenced "too many async calls" and "writing different JavaScript for different browsers."

Web smells awareness is just in its infancy. More work must be done, both from investigators and practitioners. There should be more awareness of CS for the web in several curricular units in academia.

[This page has been intentionally left blank]

Chapter 4

Web code smells

Contents

This chapter introduces a primer code smells catalog and a community platform for the expansion of this catalog

4.1 Introduction

We organized a comprehensive catalog of web code smells to systematically analyze their prevalence and impact on web development. We extensively reviewed the literature and tool documentation of code smells for the web and selected code smells that do not generate controversy. The catalog is divided into server- and client-side code smells, and the client-side is further divided into embed CS and JavaScript CS. Additionally, we introduce a few novel code smells that have not yet been discussed in the literature.

The initial aim was to introduce these new web code smells in the context of evolutionary studies articles. However, recognizing the lengthy review and publication process, we decided to expedite the process by consulting with a group of code smell experts to establish a more concise "Web Smells catalog." This working catalog, the focus of our current research, enabled us to perform a comprehensive set of statistical evolution studies, with detailed findings in chapters 5 and 6.

To later expand the code smells catalog and get feedback from researchers and practitioners, we built a collaborative code smells catalog platform, described later in this chapter. This platform will allow to propose new CS, vote, and comment on others.

4.2 Related work

"Bad Smells in Code" was an essay by Kent Beck and Martin Fowler, published as a chapter of the famous book "Refactoring – Improving the Design of Existing Code" [12]. Since then, the term has gained popularity, and there are many studies about the subject, usually using Java, although the main ideas can be applied to any object-oriented programming language. A comprehensive list of code smells can be found in a paper by Mantila et al. [95, 96], as well as online in various sources [6, 57, 94]. The published work on web systems smells is considerably scarce. Some papers on web applications have focused on clone detection [84] (just one among many smells) and tools for code smell detection on client-side JavaScript [52, 72]. However, these studies mainly concern client-side programming issues, covering only some of the relevant issues. As for web systems smells on the server-side, published work is even scarcer. We will review and comment on published works on both sides in the following two subsections.

4.2.1 Web Programming - Client-side smells

Hung Viet Nguyen et al. [72] proposed a list of 6 client-side smells, mainly concerning JavaScript (JS) and CSS: They claim that WebScent is a tool for detecting embedded code smells in server code, but detected smells lie only on the client side (i.e., the smells are only in the part of code that runs in the browser – HTML, CSS, JavaScript). However, to detect code duplication, one must examine the server code because much code will be perceived as duplicated in the browser, but it is a server-side include and not repeated (i.e., a false clone).

A year later, Fard et al. [52] proposed a set of 13 JavaScript code smells. This set of code smells is considerably based on Fowler's catalog [12]. In addition, the authors developed a tool

– JNose – to automate their collection. This tool uses a web crawler, so it can only analyze the client side.

4.2.2 Web Programming - Server-side smells

The most used server-side programming language, PHP, accounts for almost 80% of the server programming in the world [161]. A good choice for our study will then be PHP. Besides its representativeness, PHP projects are typically open-source, therefore allowing code analysis.

As discussed in [127], most of Martin Fowler's code smells [58] still make sense for PHP. However, we could not find published work on the corresponding detection algorithms. The closest we could find was the PHPMD[1] tool [46]. The latter is a rule-based static analyzer of the PHP source code base and looks for several potential problems within that source that can be code smells, and one can define a ruleset and use it accordingly.

4.3 Web smells catalog

Web applications or the larger web systems can be built in three main forms, as introduced in chapter 1: monolithic applications or systems (the great majority), with server- and client-side code mixed, and frontend/backend or with a micro-service architecture. The last two types are no longer one atomic application, and normally are distributed among code repositories. We need a web code smells catalogue to cover both programming sides, the server-side and the client-side, and their mixture in the code files.

Some preliminary attempts have been made to define web smells, as described in related work, but they have limited coverage (only the server-side or only the client-side) and their validation was almost exclusively done through peer review in the corresponding publication fora. However, and to the best of our knowledge, there is no comprehensive web smells catalog that addresses both client and server sides. Next, we introduce the initial web smells catalog that resulted from literature selection and proposal of new web code smells.

4.3.1 Web smells catalog - initial

The next table shows the referred initial web smells catalog, with code smells from articles, detectors, and some CS proposed by us:

[1] https://phpmd.org/

Table 4.1: Initial web smells catalog

area	name	origin	language(s)
C	JS in HTML	NGuyen12	JS, HTML
C	CSS in JS	NGuyen12	CSS, JS
C	CSS in HTML	NGuyen12	CSS, HTML
C	Scattered Sources	NGuyen12	CSS, JS, HTML
C	Duplicate JS	NGuyen12	JS
C	HTML Syntax Error	NGuyen12	HTML
C	Closure smell (Function),	Fard13	JS
C	Empty catch - Lines of code (Code block),	Fard13	JS
C	Excessive global variables (Code block),	Fard13	JS
C	Large object (Object),	Fard13	JS
C	Lazy object (Object),	Fard13	JS
C	Long message chain (Code block),	Fard13	JS
C	Long method/function (Function),	Fard13	JS
C	Long parameter list (Function),	Fard13	JS
C	Nested callback (Function),	Fard13	JS
C	Refused bequest (Object),	Fard13	JS
C	Switch statement (Code block),	Fard13	JS
C	Unused/dead code (Code block).	Fard13	JS
C	Not supported Html code	Proposed	HTML
C	Layout with tables (HTML)	Proposed	HTML
C	Layout not responsive or adaptive	Proposed	HTML, CSS
C	Repeated Css	Proposed	CSS
S	BooleanArgumentFlag	PHPMD	PHP
S	ElseExpression	PHPMD	PHP
S	StaticAccess	PHPMD	PHP
S	CyclomaticComplexity	PHPMD	PHP
S	NPathComplexity	PHPMD	PHP
S	ExcessiveMethodLength	PHPMD	PHP
S	ExcessiveClassLength	PHPMD	PHP
S	ExcessivePublicCount	PHPMD	PHP
S	TooManyFields	PHPMD	PHP
S	TooManyMethods	PHPMD	PHP
S	ExcessiveClassComplexity	PHPMD	PHP
S	ExitExpression	PHPMD	PHP
S	EvalExpression	PHPMD	PHP
S	GotoStatement	PHPMD	PHP
S	NumberOfChildren	PHPMD	PHP
S	DepthOfInheritance	PHPMD	PHP
S	CouplingBetweenObjects	PHPMD	PHP
S	ShortVariable	PHPMD	PHP
S	LongVariable	PHPMD	PHP
S	ShortMethodName	PHPMD	PHP
S	ConstructorWithNameAsEnclosingClass	PHPMD	PHP
S	ConstantNamingConventions	PHPMD	PHP
S	BooleanGetMethodName	PHPMD	PHP
S	UnusedPrivateField	PHPMD	PHP
S	UnusedLocalVariable	PHPMD	PHP
S	UnusedPrivateMethod	PHPMD	PHP
S	UnusedFormalParameter	PHPMD	PHP
SC	ExcessiveNumberOfLanguages	proposed	all

In table 4.1, the area refers to server- or client-side (S or C), the name is the name of code smell, the column origin where the CS was first proposed (NGuyen12 [72], Fard13 [52], PHPMD [46]) or if we propose it. The last column is the target language. Some of the PHPMD code smells were already used by us in preliminary studies [133, 134]. In this initial catalog, the names of the CS are the names that appear in the source.

The code smells proposed are:

- *Not supported HTML code*- when the HTML code corresponds to a deprecated version - the browser will interpret it, but it may pose problems in the future.
- *Page Layout with tables (HTML)* - all layouts should be made with elements like divs or the new HTML 5 elements
- *Layout not responsive or adaptive* - if the layout is not responsive, this may become a problem for the apps as the user cannot see them in smaller devices
- *Repeated CSS* - not only repeated rules but also repeated included files
- *ExcessiveNumberOfLanguages* - sometimes the server code uses two or even three languages (if the server can process them). This linguistic profusion may reduce understandably, therefore, maintainability.

Nevertheless, this catalog has two challenges: First, a few code smells require further clarification, as they may not necessarily be classified as code smells but as bad practices. Second, the newly proposed code smells have not yet been validated through formal articles or studies; their legitimacy has only been established through consultation with expert developers.

To address these concerns and proceed with the evolution studies, we assembled a group of experienced developers to curate a refined list of web code smells that met two criteria: i) a clear consensus among the developers, eliminating any ambiguity in their classification, and ii) the ability to detect them effectively, even if it required the development of new tools for detection.

4.3.2 Working Web smells catalog

Next, we present the working web smells catalog used in chapters 5 and 6 studies. This catalog is divided into three parts like the former initial catalog, and we presented them in 3 sections.

4.3.2.1 Server-side smells

All the following code smells came from the tool PHPMD[46], which in turn came from the PDM java tool [120].

- (Excessive)CyclomaticComplexity - Complexity on method>10 - number of decision points in a method plus one (method entry). Decision points:'if', 'while', 'for', and 'case labels'.
- (Excessive)NPathComplexity - NPathComplexity on method>200 - number acyclic execution paths in a method (how many possible outcomes the method has)
- ExcessiveMethodLength (Long method) - method lines >100 - method is doing too much - method is too long
- ExcessiveClassLength (Long Class) - class lines > 1000 - class does too much
- ExcessiveParameterList (Long parameter list) - nr. parameters > 10 - Method with too long parameter list
- ExcessivePublicCount - Public count >45 - Excess public methods/attributes in a class
- TooManyFields fileds > 15 - Class with too many fields (atributes)

- TooManyMethods - Methos > 25 - Class with too many methods
- TooManyPublicMethods - Public Methods > 10 - Class with too many public methods
- ExcessiveClassComplexity - Weighted Method Count(WMC)>50: WMC is the sum of complexities of all methods declared in a class.
- (Excessive)NumberOfChildren NOC>15 - Class with an excessive number of children
- (Excessive)DepthOfInheritance DEI >6 - Class with many parents
- (Excessive)CouplingBetweenObjects CBO > 13 - Class with too many dependencies
- DevelopmentCodeFragment (Forgoten?) development Code:var_dump,print_r,debug_zval_dump,debug
- UnusedPrivateField - Unused private field
- UnusedLocalVariable - Unused local variable
- UnusedPrivateMethod - Unused private method
- UnusedFormalParameter - Unused parameters in methods

These rules explanation were adapted from https://phpmd.org/rules/. The four "unused" CS, in Java, is grouped as *unused code*.

4.3.2.2 Client-side smells

The first client CS are inspired in [52, 72]. They are divided in *embed* and *inline* - inline denotes a CSS or JavaScript inside an HTML tag. The sixth embed CS is proposed by us. The last six came from[52] and from *JSLint* tool.

- embedded JS - JavaScript inside HTML page, inside a <script>tag
- inline JS JavaScript inside HTML page in the HTML elements themselves
- embedded CSS - CSS inside HTML page inside a <style>tag
- inline CSS - CSS inside the HTML page in the elements themselves (on *style* attribute)
- CSS in JS - CSS code inside JavaScript code
- CSS in JS:jQuery - CSS code inside jQuery JavaScript code
- max-lines - lines on file > 300 - exceeds number lines per file
- max-lines-per-function - lines on function > 20 - exceeds number of lines of code in a function
- max-params - number of parameters in a function >3 - exceeds number of parameters in function
- (Excessive)complexity - CC > 10 - exceeds cyclomatic complexity allowed
- max-depth - depth > 4 - exceeds the depth that blocks of code can be nested
- max-nested-callbacks - callback depth > 10 - exceeds the depth that callbacks can be nested

The first half, 6 code smells, are embed code smells, denoting the mixture of languages, and the last 6 are CS in JavaScript.

4.3.2.3 Detection Tools

To detect server-side code smells, we use the PHPMD tool (mostly embedded in batch scripts). Unfortunately, for the client side, the tools are not available anymore, the JavaScript one (from

[52]) because of the use of an ancient Firefox platform (we contacted the author). Regarding the other tool, "embed" (from [72]), we could not find it online, and the author did not respond to our contact.

Therefore, we developed a tool for detecting web embed code smells, as shown in chapter 6 and the appendix. This tool detects six code smells and outputs the code directly to a database or .csv.

As for the JavaScript detection tool, we used ESLint [2] with some adaptations embedded in batch files.

4.3.3 Web smells collaborative platform

Setting up a web smells catalog should be more than an experienced practitioner's exercise based on "gut feeling." To validate our catalog and hopefully obtain an initial consensus on the relevance of each web smell, we have developed a collaborative web platform to support a large-scale survey on practitioners, both from academia and industry. In this platform, it is possible to propose new web smells and vote on the existing ones. This will serve two purposes: to reduce the amount of subjectivity in the catalog proposal; to increase the external validity of each proposed web smell through peer assessment.

Adding new web smells and issuing votes are logged operations to avoid tampering with the catalog construction and allow recording its author/issuer and corresponding affiliation. We hope that this collaborative construction of the catalog yields better results than its elaboration by just one or two researchers. The metadata on each web smell includes, but is not limited to, the following attributes:

Table 4.2: Web smells attributes

Attribute Name	Description
Name	Web smell name
Short description	Brief description / abstract
Long description	Detailed characterization of web smell
Detection algorithm	Algorithm in pseudocode
Author name	Author of the proposed web smell
Author affiliation	Author affiliation
Proposal date	Date when the web smell was proposed
Votes	Number of votes the smell gets (external table)

Table 4.2 describes the web smell metadata used platform's database.

This catalog will include parts for the client-side and the server-side. In the initial phase, the server-side will cover PHP, which accounts for almost 80% of the market. A problem to be faced here is that PHP can be used in a procedural or object-oriented manner. The client-side will cover JavaScript, HTML, and CSS. Our preliminary web smells identification task, described earlier in this chapter, serves as a bootstrap for our platform, which is online on https://websmells.org.

[2] https://eslint.org/

4.3.3.1 Platform Architecture

In our first development of the web smells platform, we did not use any framework; however, we used the Laravel framework [3] for the second version. The architecture of the application is the one shown in the appendix. Next, we briefly demonstrate the operation of the platform.

4.3.3.2 Platform operation

The web app aims to build a collaborative web smells catalog. Its users can see, propose, vote, and comment on web smells. The user must register to do these operations, but he/she can use social networks to ease the process. The web smells must be approved to appear on the catalog.

Following, we present the application's main screens.

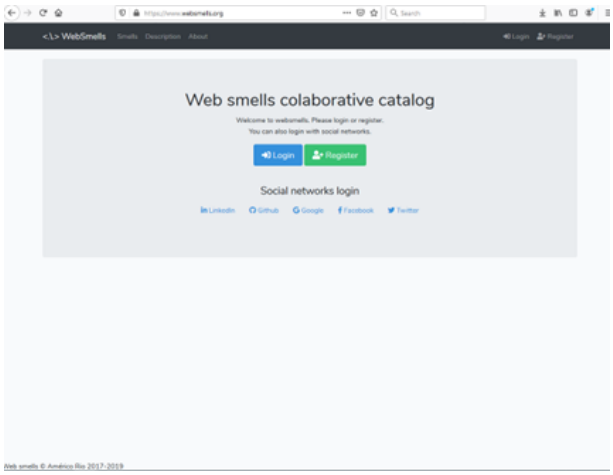


Figure 4.1: Web Smells platform login

Figure 4.1 shows the login page of the platform. Users can register and login with email and password or use one of the five social networks the login system supports.

[3] https://laravel.com/

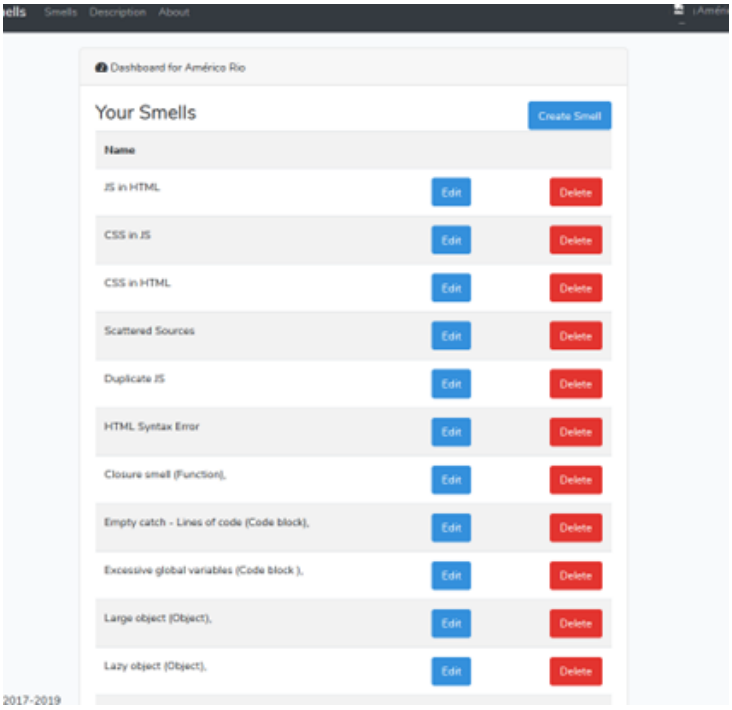


Figure 4.2: Web Smells platform CS list (author view)

Figure 4.2 shows the dashboard for authors with permission to edit their smells. In addition, there is a button to create a new smell.

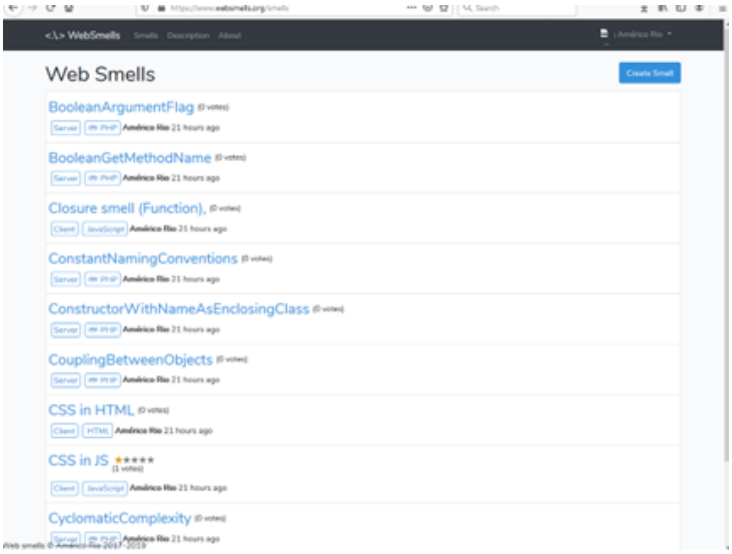


Figure 4.3: Web Smells platform (CS list general view

Figure 4.1 represents the web smells list from the menu "Web smells." A user can consult all the smells in this list and click to go to the detail view.

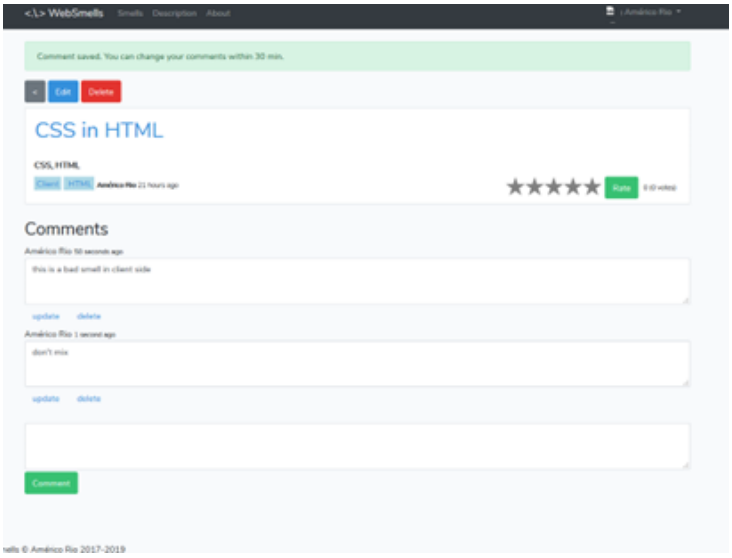


Figure 4.4: Web Smells platform (CS detail view)

The user can comment and vote in the code smell in the view 4.1. If the user is the author of the code smell, then he/she can edit and delete it.

4.4 Chapter conclusions

Web smells are code smells in web development, enclosing client languages (HTML, CSS, JS) and server languages and their interactions. We intended to build a web catalog because catalogs exist but for separate languages (especially Java) and not for Web languages, including several languages simultaneously and their interactions. Therefore, we presented an initial catalog of the web (code) smells, covering server- and client-side code.

This initial catalog was used to bootstrap a web smells collaborative platform where users can propose, vote, and comment on web code smells to validate and grow the catalog.

We also introduced a subset of web code smells called the web smells working catalog. This working catalog includes only code smells that have consensus among the group of developers used as consultants and are possible to detect with tools. It is used in various studies, including chapters 5 and 6.

Part III.

Web systems evolution studies

This part presents a set of studies on server-side code smells evolution and survival, plus probable causes for their appearance and anomalies in the evolution. Next, it presents the causal inference studies of server- and client-side code smells between themselves and on outcome variables, such as issues, bugs and time-to-release

Chapter 5

PHP code smells in web systems: evolution, survival and anomalies

Contents

This chapter presents the studies on code smells in web systems evolution, survival and anomalies

5.1 Introduction and Motivation

"Code smell" is a term introduced by Kent Beck in a chapter of the famous Martin Fowler's book [58] to describe a surface indication in source code that usually corresponds to a deeper problem. Code smells (CS) are symptoms of poor design and implementation choices that may lead to increased defect incidence, decreased code comprehension, and longer times to release. CS come in different shapes, such as a method with too many parameters or a complex body. Their detection may be subject to some subjectivity [21, 118], but that issue is beyond the scope of this paper. Keeping CS in the code may be considered a manifestation of technical debt, a term coined by Ward Cunningham [38]. Usually, to remove a CS, a refactoring operation is performed [58].

The Software Engineering community has been proposing new techniques and tools, both for CS detection and refactoring [54, 128, 172], in the expectation that developers become aware of their presence and get rid of them. However, a good indicator of the success of that quest is the reduction of CS survival time (lifespan), the elapsed time since one CS appears until it disappears due to refactoring or code dropout. Therefore, software evolution (longitudinal) studies are required to assess CS survival time while revealing other aspects of *CS evolution*, such as possible causes, trends, and evolution anomalies.

The most frequent target in CS studies are Java desktop apps [125, 128, 148]. However, few CS studies exist in other domains (web, mobile) and other languages. Web applications differ from desktop and mobile applications since some components or code segments run on a web server and others on a browser. The code that runs on a server is called the server-side code (using PHP, C#, Java, Python, Node.js' JavaScript, or other languages). The latter can communicate with other servers, such as a database or email servers, besides the host file system (e.g., storing files). The client-side code is the code that renders or runs inside a browser (HTML, CSS, JavaScript). Typical web apps have both server and client code. However, depending on the application's architecture, this code can be together as a monolithic app, separated as a distributed app (Frontend/Backend), or as a microservices architecture[1].

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

This chapter extends and updates our previous research work reported in [133, 134]. Since then, we collected more metrics from a larger sample of PHP web apps, which allowed us to address more research questions by considering more factors in data analysis. We also added novel discussions and conclusions on the evolution of CS and its relation to the evolution of app and team sizes.

We structured this chapter as follows: After the introduction with the motivation in section

[1] https://www.martinfowler.com/articles/microservices.html
[2] https://w3techs.com/technologies/overview/programming_language

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

5.2 Related Work

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

5.2.1 Evolution on CS

The evolution of 2 CS in 2 open-source systems is analyzed in Olbrich et al. [112]. The authors compare the increase and decrease of classes infected with CS and total classes in windows of 50 commits in a *SVN (Subversion)* repository. Their results show different phases in the evolution of CS and that CS-infected components exhibit a higher change frequency. Later, in [113], they investigate if the higher change frequency is true for God Classes and Brain Classes in 7-10 years of 3 systems. Without normalization, God and Brain Classes were changed more frequently and contained more defects than other kinds of classes, but when normalized by size, they were less subject to change and had fewer defects than other classes.

The lifespan of CS and developers' refactoring behavior in 7 systems mined from a *SVN* repository is discussed in Peters et al. [119]. The authors' conclusions are: a) CS lifespan is close to 50% of the lifespan of the systems; b) engineers are aware of CS but are not very concerned with their impact, given the low refactoring activity; c) smells at the beginning of the systems life are prone to be corrected quickly.

Rani et al. [124] perform an empirical study on the distribution of CS in 4 versions of 3 software systems. The study concludes that: a) the latest version of the software has more design issues than older ones; b) the "God" smell has more contribution to the overall status of CS, and "Type Checking" less. However, they also note that the first version of the software is cleaner.

Digkas et al. [42] analyze 66 Java open-source software projects on the evolution of technical debt over five years with weekly snapshots. They calculate the technical debt time-series trends and investigate the components of this technical debt. Their findings are: a) technical debt together with source code metrics increases for most of the systems; b) technical debt normalized to the size of the system decreases over time in most systems; c) some of the most frequent and time-consuming types of technical debt are related to improper exception handling and code duplication. Later in [41], the authors investigate the reasons for introducing technical debt, within 27 systems, in 6-month sliding temporal windows. Their findings are: a) the number of Technical Debt Items introduced through new code is a stable metric, although

it presents some spikes; b) the number of commits performed is not strongly correlated to the number of introduced Technical Debt Items. They propose to divide applications into *stable* and *sensitive* (if they have spikes). They use SMF (Software Metrics Fluctuation) to perform this classification, defined as the average deviation from successive version pairs.

5.2.2 Evolution on CS, with survival analysis

Chatzigeorgiou et al. [29] study the evolution of 3 CS in a window of successive versions of 2 Java systems. They extend the work in [30] to use four smells and survival analysis with survival curves. Their conclusions are: a) in most cases, CS persist up to the latest examined version, thus accumulating; b) survival analysis shows that smells "live" for many versions; c) a significant percentage of the CS was introduced in the creation of a class/method; d) very few CS are removed from the projects, and their removal was not due to refactoring activities but a side effect of adaptive maintenance.

The change history of 200 projects is reported by Tufano et al. [157] and later extended to include survival analysis in [158]. Reported findings are: a) most CS instances are introduced when an artifact is created and not because of its evolution, b) 80 percent of CS survive in the system, and c) among the 20 percent of removed instances, only 9 percent are removed as a direct consequence of refactoring operations. In [156], the authors analyze when test smells (TS) occur in Java source code, their survivability, and if their presence is associated with CS. They found relationships between TS and CS. They include survival analysis to study the lifespan of TS.

The survival of (Java) Android CS with 8 Android CS is analyzed by Habchi et al. [67]. The authors conclude that: a) CS can remain in the codebase for years before being removed; b) in terms of commits, it takes 34 effective commits to remove 75% of them; c) Android CS disappear faster in bigger projects with higher releasing trends; d) CS that are detected and prioritized by linters tend to disappear before other CS.

5.2.3 Cross-sectional or mixed studies in web apps or web languages

These studies include Saboury et al. [140], whose authors find that, for *JavaScript* applications, files without CS have hazard rates 65% lower than files with CS. As an extension to the previous paper, Johannes et al. [75] conclude that files without CS have hazard rates of at least 33% lower than files with CS. Amanatidis et al. [3], a study with PHP TD (Technical Debt) which includes CS, the authors find that, on average, the number of times that a file with high TD is modified is 1.9 times more than the number of times a file with low TD is changed. In terms of the number of lines, the same ratio is 2.4. Bessghaier et al. [16], the authors find: a) complex and large classes and methods are frequently committed in PHP files; b) smelly files are more prone to change than non-smelly files. Studies in Java [116] report similar findings to the previous two studies.

5.2.4 Evolution with PHP, without CS

Studies of this type include Kyriakakis et al. [83], where authors study 5 PHP web apps, and some aspects of their history, like unused code, removal of functions, use of libraries, stability of interfaces, migration to OOP, and complexity evolution. In addition, they found that these systems undergo systematic maintenance. Later, in Amanatidis et al. [2], they expanded the study to analyze 30 PHP projects and found that not all of Lehman's laws of software evolution [89] were confirmed in web applications.

5.2.5 Studies comparing types of CS

The following studies compare CS at different abstraction levels or different scopes. In Fontana et al. [56], the authors try to understand if architectural smells are independent of CS or can be derived from a CS or one category of them. The method used was to analyze correlations among 19 CS, 6 categories, and 4 architectural smells. After finding no correlation, they conclude that they are independent of each other. The paper by Sharma et al. [145] aims to study architecture smell characteristics and investigate correlation, collocation, and causation relationships between architecture and design smells. They used 19 smells, mining C# repositories. Their findings are: a) smell density does not depend on repository size; b) architecture smells are highly correlated with design smells; c) most of the design and architecture CS pairs do not exhibit collocation; d) causality analysis reveals that design smells cause architecture smells (with statistical means).

5.3 Methods and Study Design

5.3.1 Research questions

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

Survival studies can be further specialized. For instance, CS's effect can vary widely in breadth (localized or scattered). In localized ones, the scope is a method or a class (e.g., in Java Long Method, Long Parameter List, God Class), while the influence of others may be scattered across large portions of a software system (e.g., in Java Shotgun Surgery, Deep Inheritance Hierarchies, or Coupling Between Classes). The other factor we are concerned with regards a

superordinate temporal analysis: we want to investigate whether the CS survival time, introduction, and removal changed over time, possibly caused by CS awareness by the developers or help in detecting CS from recent IDE's or tools.

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

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

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

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

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

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

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

We performed a longitudinal study encompassing 12 web apps and 18 CS to answer the research questions. For RQ3 only, we used a subset of 6 code smells as surrogates of more scattered or localized scopes. The study will help researchers, practitioners and software managers to increase their knowledge of the evolution behavior of PHP web apps CS and help project

managers keep CS's density under control. Next, we describe the applications and the CS used in the study.

5.3.2 Applications sample

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

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

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

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

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

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

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

Table 5.1: Characterization of the target web apps (* on last release)

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

Table 5.1 shows the complete list of applications. The LOC and Classes numbers are from the last release and were measured by the PHPLOC[4] tool. As with detecting code smells, we excluded each app's folders from other vendors in the LOC measures. As an example, the excluded folders for two of the apps were: *phpMyAdmin: doc, examples, locale, sql, vendor, contrib, pmd* and for *Vanilla: cache, confs, vendors, uploads, bin, build, locales, resources*. The excluded folders for the other applications are on the replication package (file *excluded folders.txt*).

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

5.3.3 Code smells sample

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

Table 5.2: Characterization of the target code smells - original CS names used by *PHPMD*. The added (Exc.) means excessive, denoting that is a CS and not a metric

Code Smell	Description	Threshold
(Exc.)CyclomaticComplexity	Excess-method number decision points plus one	10
(Exc.)NPathComplexity	Excess-method number acyclic execution paths	200
ExcessiveMethodLength	(Long method) method is doing too much	100
ExcessiveClassLength	(Long Class) class does too much	1000
ExcessiveParameterList	Method with too many parameters	10
ExcessivePublicCount	Excess public methods/attributes class	45
TooManyFields	Class with too many fields	15
TooManyMethods	Class with too many methods	25
TooManyPublicMethods	Class with too many public methods	10
ExcessiveClassComplexity	Excess-sum complexities all methods in class	50
(Exc.)NumberOfChildren	Class with an excessive number of children	15
(Exc.)DepthOfInheritance	Class with too many parents	6
(Exc.)CouplingBetweenObjects	Class with too many dependencies	13
DevelopmentCodeFragment	Development Code:var_dump(),print_r()	1
UnusedPrivateField	Unused private field	1
UnusedLocalVariable	Unused local variable	1
UnusedPrivateMethod	Unused private method	1
UnusedFormalParameter	Unused parameters in methods	1

A brief characterization of all CS used is presented in table 5.2. The CS names are the ones used by *PHPMD*, but we added (Exc.) to mean excessive, denoting that it is a CS and not a metric. The thresholds used are the default ones used in *PHPMD*, which in turn came from PMD[6], and are generally accepted[7] from the references in the literature [18, 85, 98]. These thresholds should be considered baselines and could be optimized using an approach like the one proposed in Herbold [70]. The latter concludes that metric thresholds often depend on the properties of the project environment, so the best results are achieved with thresholds tailored to the specific environment. In our case, where we have 12 web apps, each developed by a different team, such optimization would lead to specific thresholds for each app, adding confounding effects to the comparability between apps we want to carry out in this study.

5.3.4 Data collection and preparation workflow

The workflow of our study (see figure 5.1) included a data collection and preparation phase before the data analysis phase and was fully automated using several tools. We performed the following steps in the data collection and preparation phase:

1. We downloaded the source code of all releases of the selected web apps, in *ZIP* format, from *GitHub*, *SourceForge*, or private repositories, **except** the alpha, beta, release candidates, and

[5] https://phpmd.org/
[6] https://pmd.github.io/
[7] https://pmd.github.io/latest/pmd_java_metrics_index.html

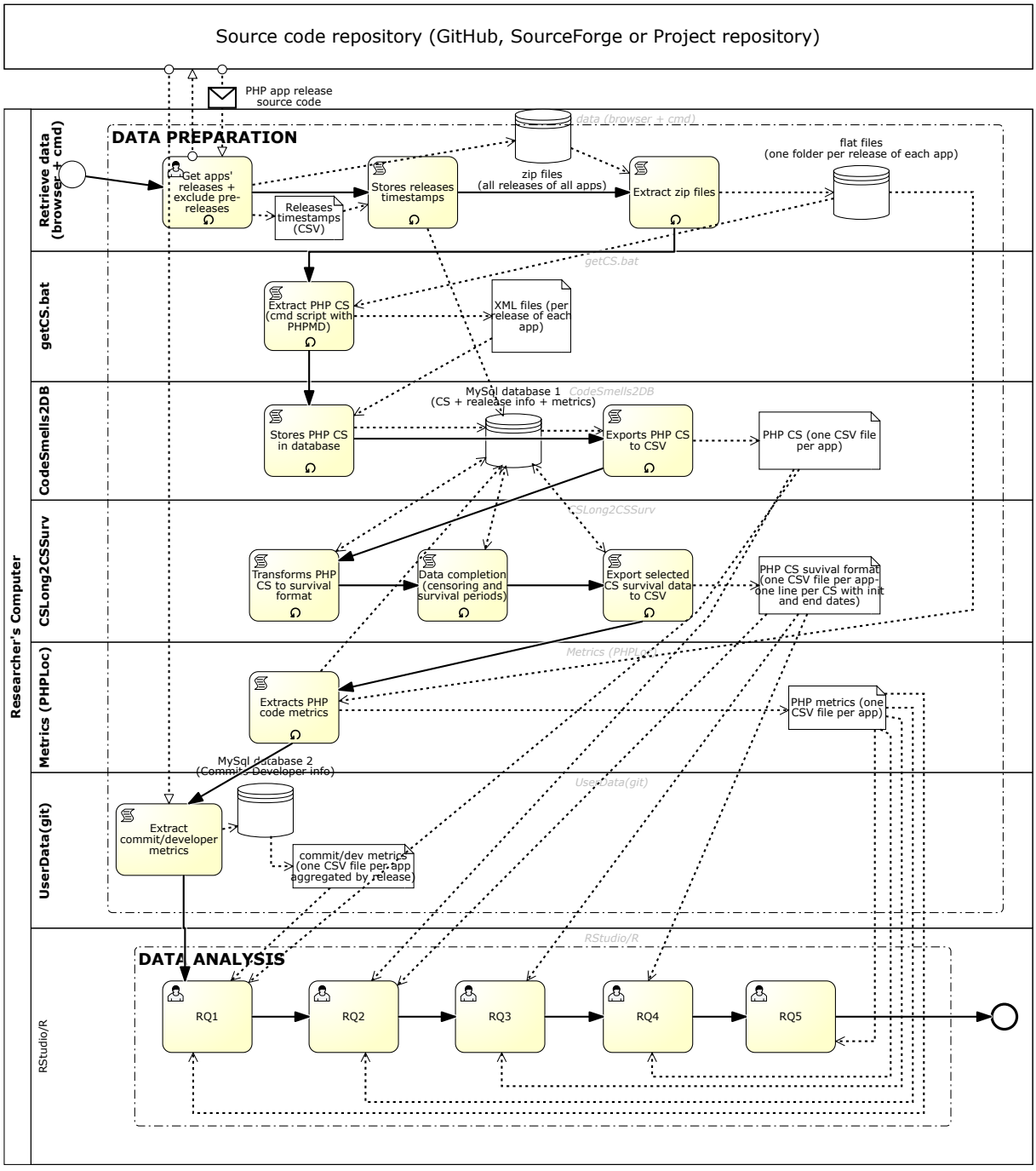


Figure 5.1: Workflow of the data preparation and analysis phases

corrections for old releases, i.e., everything out of the main branch. We considered only the stable branch when we had two branches in parallel. We used only the higher one when we had two releases on the same day. During this step, we created a database table with the application releases, which were later exported to a *Comma Separated Values (CSV)* file containing the timestamps for each downloaded release. Next, we extracted the *ZIP* files (one per each release of each app) to flat files on the file system of our local computer (one folder per each release of each app).

2. Using *phpMyAdmin*, we imported the *CSV* file with the releases' timestamps created in the first step to the CS database, a *MySQL* database.

3. Using *PHPMD*, we obtained the CS and respective attributes from all releases and stored them in XML file format (one file per application release). We *excluded some directories* not part of the applications (vendor libraries, third-party code). The zips from GitHub and other locations had these folders to make the application run without additional downloads.
4. We used the *CodeSmells2DB* PHP script, developed by the first author, to read the previous XML files and, after some format manipulation, store the corresponding information in the CS *MySQL* database. The data records at this point are divided by release/smell.
5. With the script *CSLong2CSSurv*, developed by the first author, we transformed the code smells information stored in the *MySQL* database into a format suitable for survival analysis by statistics tools. That includes the date when each unique code smell was first detected and when it disappeared if that was the case. The script stores the results of this transformation back in the database in other tables. Then, the script performed a *data completion* step, where it also calculates the censoring (removal=1 or not removal=0) and survival periods. Later, we exported the results to *CSV* format (one file per app) in preparation for the data analysis phase.
6. We used *PHPLOC* to extract several code metrics from the source code of each release of each app, storing them in the same database and later exporting them to a CSV file per app. We excluded the folders from third-party code and libraries (different folders for each application).
7. Finally, we cloned all the git repositories of the applications and retrieved developers and commits data that we aggregated by app release, using a conjunction of *command line*, *git*, and *SQL* commands. This data was stored in a second database and later exported to *CSV* files (one per app), concluding this data preparation phase.

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

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

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

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

5.3.5 Statistics used

We used the R statistics tool with several packages for all the analysis studies to perform the evaluation, correlations, and output of the graphics. For the correlations, we used the standard R *cor* function (p-values given by *cor.test*).

We used survival analysis in RQ2 to RQ4. *Survival analysis* encompasses a set of statistical approaches that investigate the time for an event to occur [33]. The questions of interest can be the average survival time (the one that interests us the most) and the probability of survival at

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

a certain point, usually estimated by the Kaplan-Meier method. In addition, the log-rank test can be used to test for differences between survival curves for groups. Also, we can calculate the hazard function, i.e., the likelihood of the event occurring (not used in the present study).

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

$$S(t_i) = S(t_{i-1})(1 - \frac{d_i}{n_i}) \tag{5.1}$$

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

We used the R packages **survival**[9] and **survminer**[10]. Regarding survival time, the two average measures are the median (50% probability of end of the life of the subject to occur) or the restricted mean ("rmean"), .i.e., the mean taking into account the end of the life of the subject (in our case the removal of the code smell). Because the median survival time is insensitive to outliers, it better describes the mean lifespan of the CS. However, in some cases, the median cannot be calculated (it requires that the survival curve goes under 50%), and the restricted mean ("rmean") can typically be calculated. Kaplan-Meier survival curve is a plot of Kaplan-Meier [76] survival probability against time and provides a valuable summary of the data that can be used to estimate measures such as median survival time[11].

In question 3 and 4 we used the **log-rank test** [142] and the two different co-variables (type/scope and time). The log-rank test is used to compare survival curves of two groups, in our case, two types of CS. It tests the null hypothesis that survival curves of two populations do not differ by computing a p-value. If the p-value is less than 0.05, the survival curves are different.

5.3.6 Methodology for each RQ

5.3.6.1 RQ1 – How to characterize the evolution of CS?

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

[9] https://cran.r-project.org/web/packages/survival/
[10] https://cran.r-project.org/web/packages/survminer/
[11] http://www.sthda.com/english/wiki/survival-analysis-basics
[12] Thousand's logical lines of code
[13] https://mattstauffer.com/blog/how-to-count-the-number-of-lines-of-code-in-a-php-project

$$CSdensity = \frac{CSabsolutenumber}{kLLOC} \tag{5.2}$$

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

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

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

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

```
1 git shortlog -sne HEAD
2     --after=<date_release_n>
3     --before=<date_release_n_plus_1>
```

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

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

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

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

5.3.6.2 RQ2 – What is the distribution and survival/lifespan of CS?

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

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

Next, we performed survival analysis by code smell. We use the non-parametric method Kaplan-Meier survival time estimate [76] to achieve this. Next, we calculate and present various averages per code smell (distribution, intensity by kLLOC), median, restricted mean (not used, just shown to compare with other values when there is no median), and percentage of CS removal. Finally, we show the survival curves plot, the values per CS in a table, and averages in violin plots. Continuing, we calculate each app's CS survival time (lifespan) values and *all apps*. We also show the plots of survival curves for the apps and the tables with values per app. Finally, we calculated the median with all code smells and the average lifespan in all apps, in which we calculated weighted averages (because of the different size and longevity of apps) and compared it with the simple average (to avoid larger applications skewing).

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

Figure 5.2 shows how to calculate the probabilities and median using survival curves (also called Kaplan-Meier plots). On the left are 6 example lines representing CS with various timespans and initial dates (in days). On the right are the same lines ordered by timespan. This way, the Y-axis is the survival probability, and the X-axis remains the days, beginning with 0. The blue line is the survival curve. The median divides the smells into halves and is the value when the probability of survival is 0.5 (or 50%). Getting the value of the curve in the X-axis when the probability is 0.5 gives the median survival of the CS in days.

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

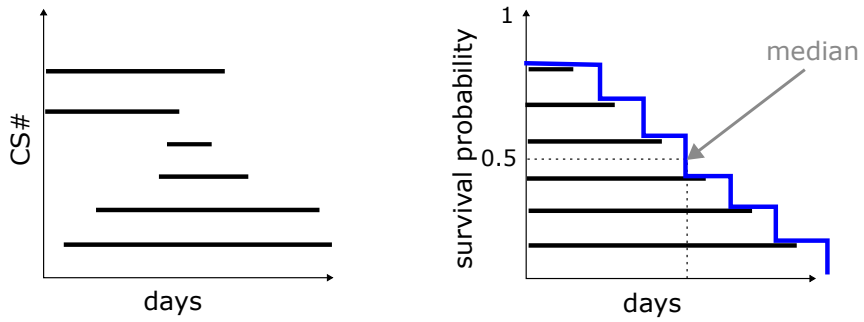


Figure 5.2: Survival curves example. Left: 6 CS in survival format; Right: the same 6 smells in survival curves/Kaplan-Meier plots, with ordered CS. The median divides the ordered CS in half and the value represents also the probability of 50% survival (Y axis)

5.3.6.3 RQ3 – Is the survival of *localized* CS the same as *scattered* CS?

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

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

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

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

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

5.3.6.4 RQ4 – Does the survival of CS vary over time?

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

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

If the survival curves are not different, their survival should be the same around the application's life (no change); if they are different, we will measure the survival variables of interest

[15] https://phpmd.org

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

[17] *DepthOfInheritance*, *CouplingBetweenObjects*, *NumberOfChildren*

(survival time median, restricted mean, and the number of introduced and removed CS in both periods).

To test the hypothesis, we also used the log-rank test and created a co-variate *timeframe*, with two values '1' and '2', '1' for the first half of the collection period, and '2' for the second half. In other words, the variable time frame will have the value '1' in the CS introduced in the first half and the value '2' in the CS introduced in the second half. For the first period, we truncated the study's variables as if they were in a sub-study ending in this period. Therefore, at the end of the first period, we filled the value of the censored column (with values described in the "statistics used" section). By doing this, we created two independent time frames for each application to analyze the code smells survival. If the p-value is less than 0.05, the CS survival differs between the two time frames.

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

5.3.6.5 RQ5 – How to detect anomalous situations in CS evolution?

When we performed the first part of the study (the CS evolution), we observed releases in which there was a refactoring on file names and location in folders (see figure 5.6), but the smells prevailed, as shown by the histograms (see figure 5.3. Our algorithm considered them new because they are in a different file/folder. Our investigation question is: how do we check those releases for anomalies in the number of CS and quantify them? In other words, how to check for peaks in the evolution of CS?

The *anomalies* occur when there is a significant increase (or decrease) in the number of CS, and the size does not grow (or reduce) accordingly. We can divide the #CS by the size for a direct way to spot anomalies. For the size measure, we can use the "Lines of Code" or "Number of Classes" [69]. However, if we use the number of classes, we could misrepresent the size of the programs with big classes (a CS itself). So a more accurate measure of the size is the number of lines. However, in a PHP file, it is possible to have other code than PHP (HTML. CSS, JavaScript), and the PHP code is enclosed in tags (<?php and ?>), so the PHP interpreter processes it on the web server. However, programs like PHPLOC or similar count all the lines on those files, even outside the PHP tags. Consequently, a better indicator of lines of code would be the LLOC, logical (effective) lines of code (the statement lines).

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

$$\Delta\rho cs = \frac{\rho cs_i - \rho cs_{i-1}}{\rho cs_{i-1}} = \frac{\rho cs_i}{\rho cs_{i-1}} - 1 \tag{5.3}$$

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

The anomalies in the evolution of CS can also be defined as sudden variations in CS density. We made automatisms via scripts to detect these outliers or anomalies in the CS evolution numerically. However, we also present the graphical evolution of CS density for each application,

which makes it easy to pinpoint the anomalies or outliers that we show with a label with the release number for easier visualization. By inspecting each anomaly (peak) in the evolution of the application code and CS, we explain what happened in that release.

5.4 Results and data analysis

5.4.1 RQ1-Evolution of code smells

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

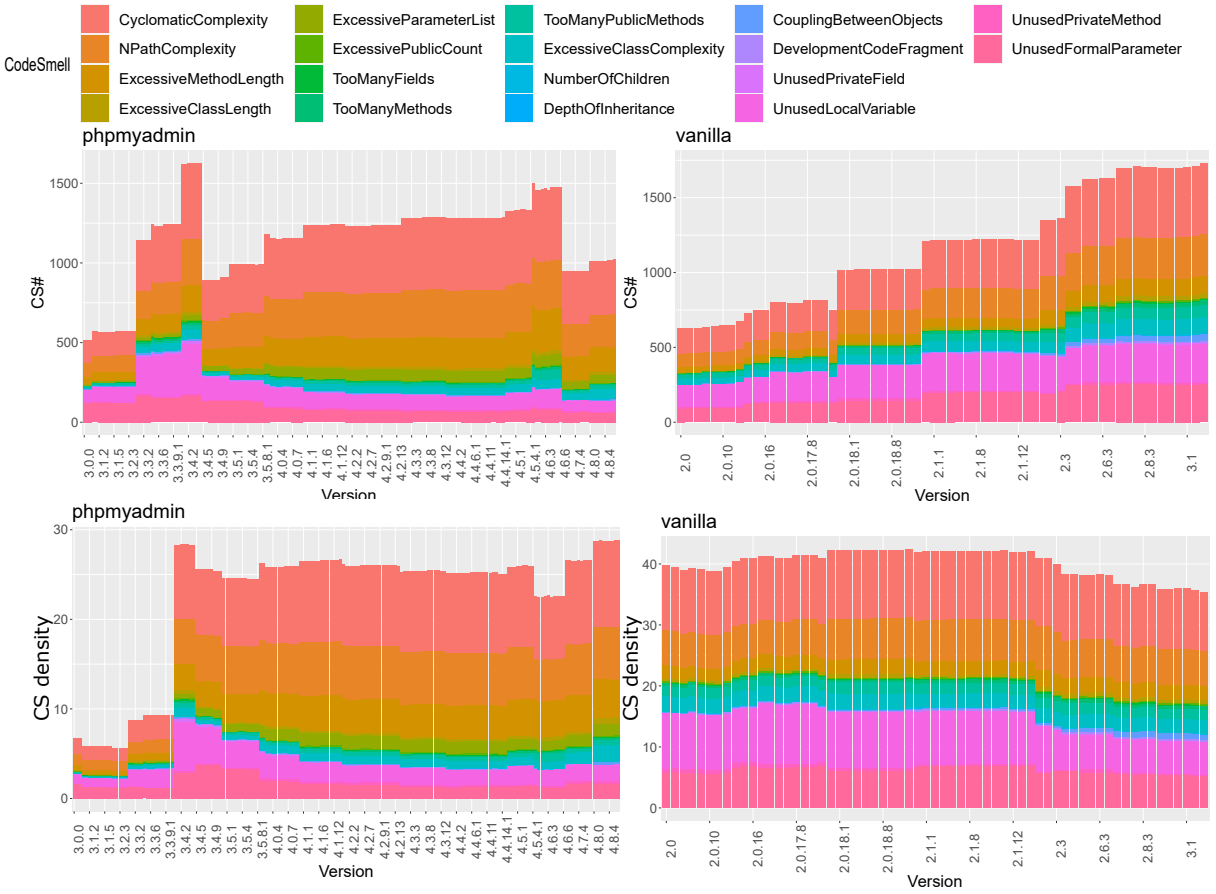


Figure 5.3: Evolution of the absolute number and density of 18 code smells, for 2 applications

Figure 5.3 shows the stack bar charts for 2 of the applications studied, with the 18 CS stacked in the bars, each release represented by a vertical bar. The complete charts for all the applications are in the replication package[18]. In addition, we analyzed the absolute evolution of CS and the density for all the applications.

For the first application in the figure 5.3, *phpMyAdmin*, CS density increases in a first period up to a peak, and after is more or less stable, showing some reductions along the way. As we see later, this peak, in the beginning, is probably related to an increase in the team size. The absolute number of CS for this application ranges from 500 to 1500 (remembering we excluded third-party folders). The density varies from 5 to almost 30 CS per kLLOC.

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

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

Table 5.3: Qualitative evolution trends of absolute CS(CS number), size(LLOC), CS density(CS/LLOC)

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

Table 5.3 shows the qualitative evolution of CS, for all the applications. *inc* and *dec* are abbreviations to *increases* and *decreases*, while *alm. stable* is an abbreviation for *almost stable*. *Short dur* is an abbreviation to *short duration*. The most common trend in the evolution of the total absolute number of CS is the steady increase of the code smells (denoted by the word "increases/inc" in the table 5.3). This trend is very similar to the evolution of the application size (second column).

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

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

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

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

5.4.1.1 Probable causes for the CS evolution

This section shows the correlation with metrics time series that can cause CS evolution trends. Figure 5.4 shows the evolution in a graphical way for the two example apps allowing us to inspect the correlation between CS number and app size. We have the remaining graphics in the replication package for the rest of the applications.

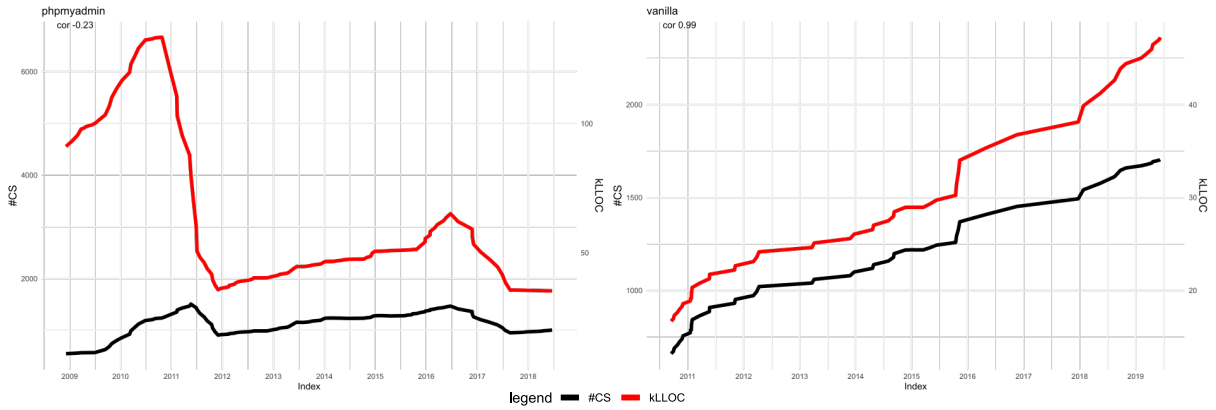


Figure 5.4: Evolution and correlation of CS and app size (LLOC), for 2 applications

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

Table 5.4: Average metrics by app

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

Table 5.4 shows the average values of the developers' team metrics and code metrics for each app: *CS density* is the average density of code smells (by kLLOC); *releases* is the number of releases in total per app; *release age* is the average time in days between releases (also called the *release frequency*); *num commits* is the average number of commits between releases; *num devs* is the average number of users between releases; *commits per dev* is the average commits per developer between releases and *commits per day* is the average commits by day between release - we used the number of measured days between releases which varies a lot; finally, *new devs* is the average number of new devs between release, i.e., the number of devs that did not commit to the app before.

The table with average metrics is shown to compare and check if there are outliers in the average values. The complete time series of the metrics in the table 5.4 for the 12 web apps are in the replication package. The CS density varies from 20 to 50 CS/kLLOC, except for *Kanboard*, which has a minimal CS density. The two outliers in the average number of commits

between releases are *Kanboard* and *phpPgAdmin* with very low commits per release. The same two applications have the lowest team size and the new dev numbers. The *number of devs* ranges from 10 to 40, being *phpPgAdmin* clearly an outlier with just 5.7 in average. The average number of *new devs* ranges from 5 to 20, if not counting the outlier. An important metric is the *average commits per dev between releases*, which ranges from 5 to 60.

Table 5.5: Correlations between code smells density and the column metric - in bold if greater than 0.3

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

We measure the correlations between the code smells density time series and the time series for each of the variables referred before for each application. Table 5.5 shows the correlation values. Each column represents the correlation with the metric of the column's name. The positive time series correlations greater than 0.3 and with a p-value of less than or equal to 0.05 are in bold. We can observe that the density of code smells correlates with the developers (column *num devs*) in a given release, except for four apps - ranging from 0.47 to 0.94). This correlation is also strong with the "*new devs*", except for two apps with negative correlation and one with almost no correlation. We observe that two of the apps (*phpPgAdmin* and *Kanboard*) are much smaller than the others, and also small is the number of developers in those apps (table 5.4).

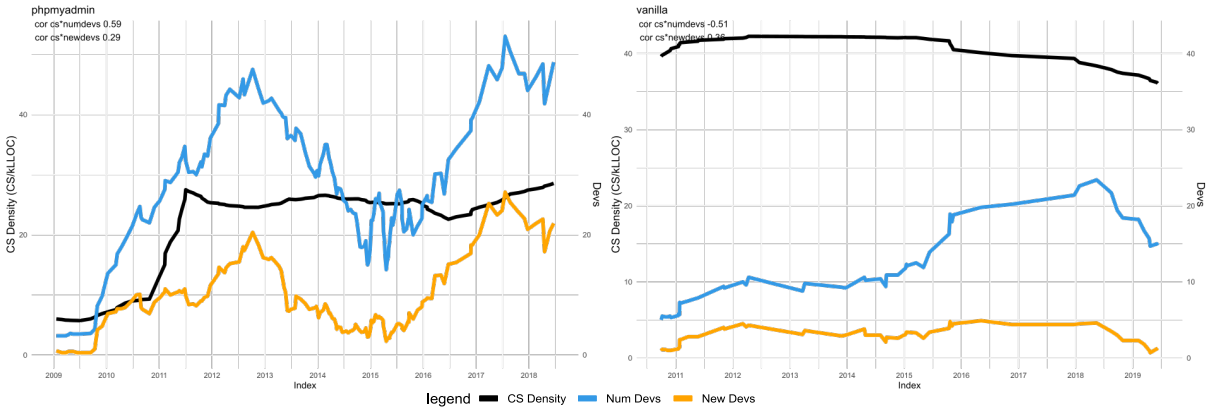


Figure 5.5: Correlation and evolution of CS density (CS/kLLOC) and team size, for 2 applications

Figure 5.5 presents the evolution and correlation between CS density, team size, and CS

density in a graphical way for the two selected apps. We can find the graphics for the rest of the applications in the replication package. For the application on the left, *phpMyAdmin*, the correlation between CS density and *"num devs"* is 0.59, while the correlation between CS density and *"new devs"* is 0.29. For the application on the right, *Vanilla*, the correlation between CS density and *"num devs"* is 0.51, while the correlation between CS density and *"newdevs"* is 0.36. Those are not the highest correlations, as we can see in the table 5.5.

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

5.4.2 RQ2-PHP code smells distribution and lifespan

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

5.4.2.1 CS lifespan

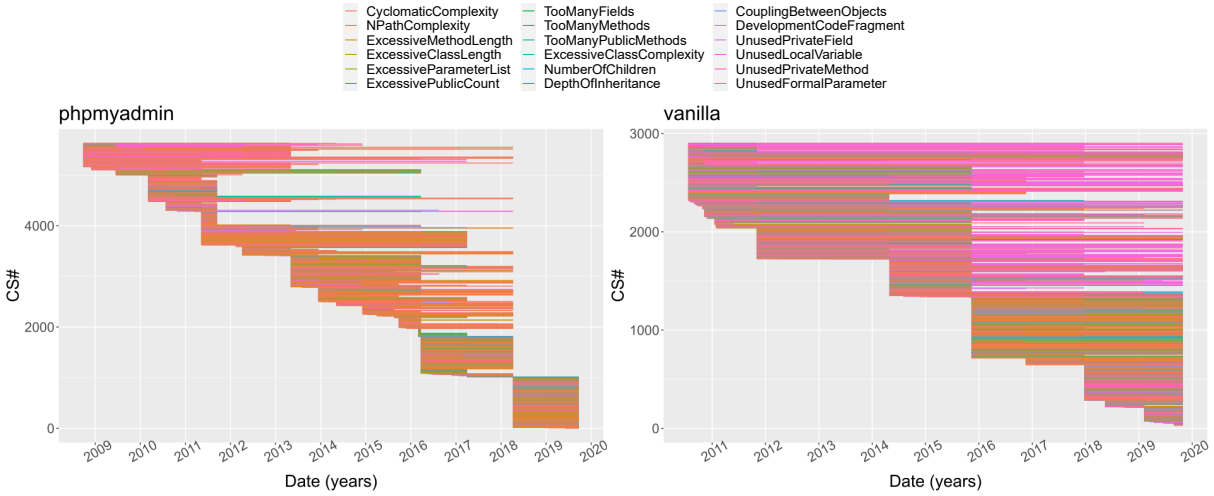


Figure 5.6: Life of unique CS in 2 of the apps

Figure 5.6 shows the unique CS for two selected applications. The horizontal lines represent the lifespan of the code smells. Each line represents one CS from its appearance on the application code until its removal. The colors represent the different CS (keyed in the legend), but this graph is more useful to observe if we look for the overall status of the CS evolution. In the application on the left, we can see a large removal of code smells, denoted when a large number of lines end at the same time (2011, 2016, 2017, 2018), while in the application on the right, this removal happens to a much lesser degree (but still happens in 2015 and 2018).

5.4.2.2 Values by CS

Figure 5.7 shows the survival curves or Kaplan-Meier plots for the 18 CS. With the curves, it is possible to calculate the survival probability at various times by crossing the X-axis with the

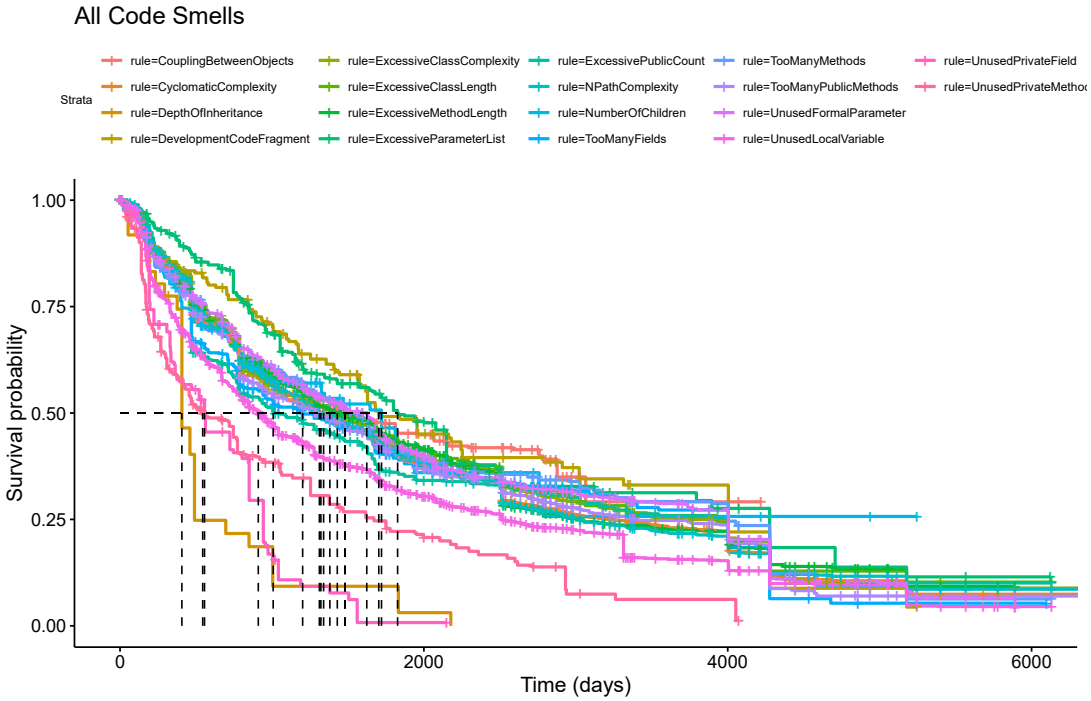


Figure 5.7: Survival curves for 18 code smells. Dashed lines denote the median.

Y-axis in the line. The medians (probability of survival =0.5) for each code smell are shown in dashed lines. Next, we show the values in a table.

Table 5.6: Average distribution, density, survival time (median and restricted mean) and % removal of Code Smells (by CS)

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

Table 5.6 represents several indicators: *CS Distribution* is the distribution of code smells averaged by CS averaged by app; *CS Density* is the CS intensity (by kLLOC) by CS averaged by app; *Survival Time Median (days)* is the *median* averaged by application; we also show the *restricted mean by CS*/averaged by application - only for comparison. For the three last columns,

we measured the values for all applications separately and then calculated the average to account for the problem of bigger applications with more code smells, which could skew the distribution. Next, we plot the table's top values and present the analysis for both the table and the graphics.

Figure 5.8 shows the distribution mean, quartiles, and variance of the most seen code smells. Figure 5.8a represents the same statistics for the density of the most seen code smells. Figure 5.8b shows the mean, quartiles, and variance for the CS survival time, and figure 5.8c shows the same statistics for the removal percentage by app for the most seen smells.

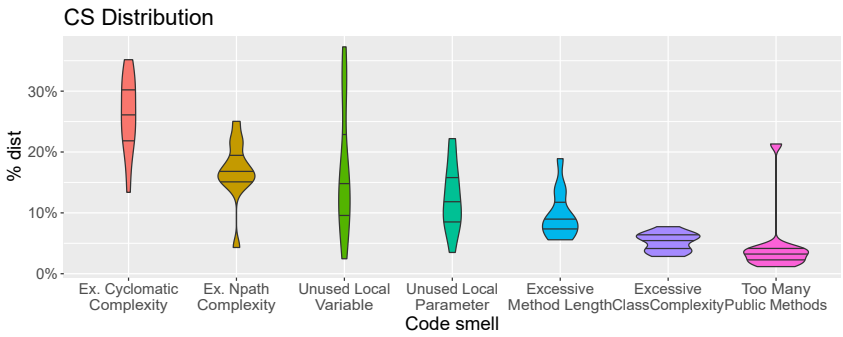
CS distribution: The most prevalent server-side CS in the studied web apps are by *(Excessive)CyclomaticComplexity* with 26.14% , followed by *(Excessive)NPathComplexity* with 16.90%. The other CS that appear more in the code are: *UnusedLocalVariable* :16.88%; *UnusedFormalParameter*: 12.46%; *ExcessiveMethodLength:* 9.60%; *ExcessiveClassComplexity* :5.22% and *TooManyPublicMethods* with 4.45%. The others are less than 2%, as shown in the table 5.6. The variance is not the same in the those CS, being the *UnusedLocalVariable* the one with most variance (figure 5.8). The *ExcessiveClassComplexity* and TooManyPublicMethods have a small variance, but the last one has a huge outlier in one app that goes to 20% - remembering that these values are averaged per app.

CS density (CS/kLLOC) - We also calculate the average density of the smells per kLLOC. This value is calculated first per app and then averaged. We can have these values as a reference when analyzing a new app. The CS that are more intense by average are the same as before, but the order is slightly different. The values are for the same top 7 in the figure 5.8a): 8.17; 5.30; 6.43; 3.54; 2.96; 1.47 and 1.09. The first tree CS have long tails (greater variance), and the following four are more concentrated in the medium value. It does not make statistical sense to calculate the average for these values, but if we sum all the smells together, the average value in all apps for the density of the code smells is around 31 CS per 1000 logical lines of code (considering the 18 code smells and the 12 web apps studied).

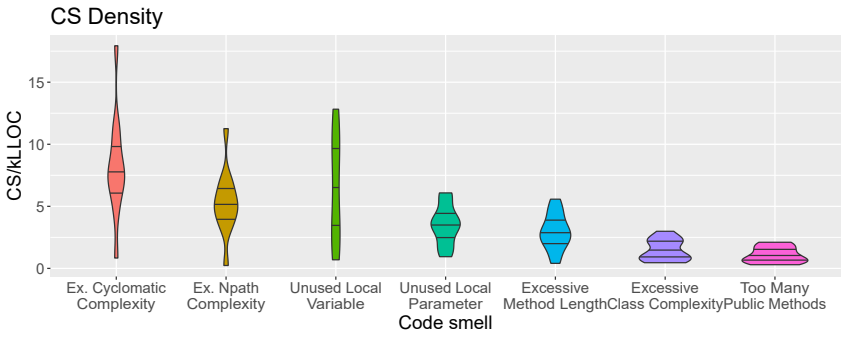
Survival time median: We want to know the *median* of the survival time of the CS. This value ranges from 500 days (*Unused Private Method*) to 1609 (*Unused Formal Parameter*). The code smells with longevity greater than 1400 days on average are *Unused Formal Parameter*, *Excessive Method Length*, *Excessive Class Complexity*, *Too Many Public Methods*. The code smells with the longevity of fewer than 1000 days on average are *Unused Private Method*, *Unused Private Field*, *(Excessive)Depth Of Inheritance*. The tails are long for both sides in the same top 7 CS (figure 5.8b), except for *Excessive Class Complexity*, which has a strange distribution on the top caused by outlier apps.

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

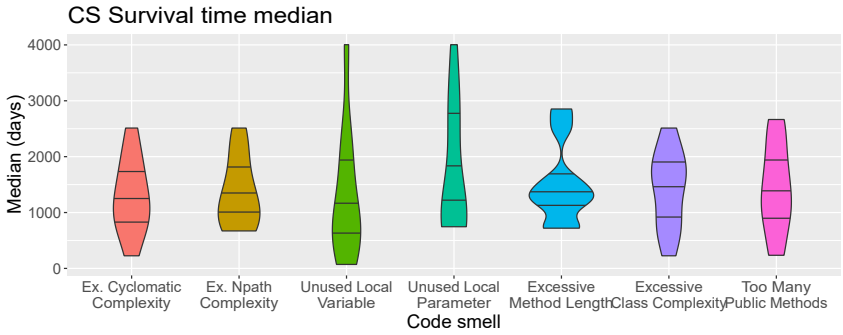
Percentage of CS removal: The values range from 40% to 70% with two outliers, one in the left (*CouplingBetweenObjects*) at 30% and other on the right (*DepthOfInheritance*) at 85%. The variances for the removal of the CS between apps are quite normal (figure 5.8c), as the tails are not too short nor too long.



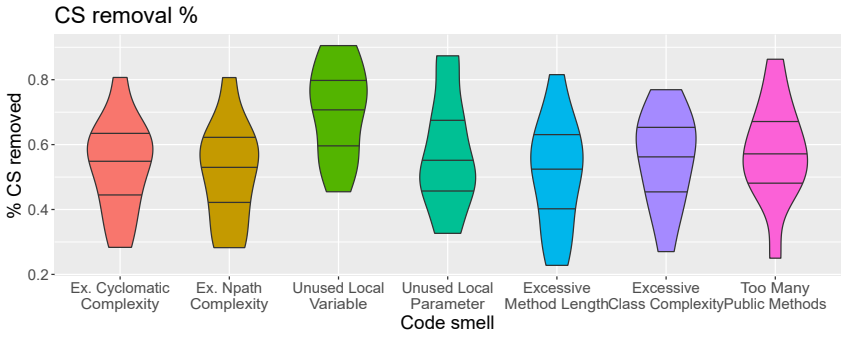
a Average CS distribution percentage



b Average CS density (CS/kLLOC)



c Average CS survival time (days)



d Average CS removal %

Figure 5.8: Average plot for top 7 CS: Distribution, Density, Survival Time and removal percentage

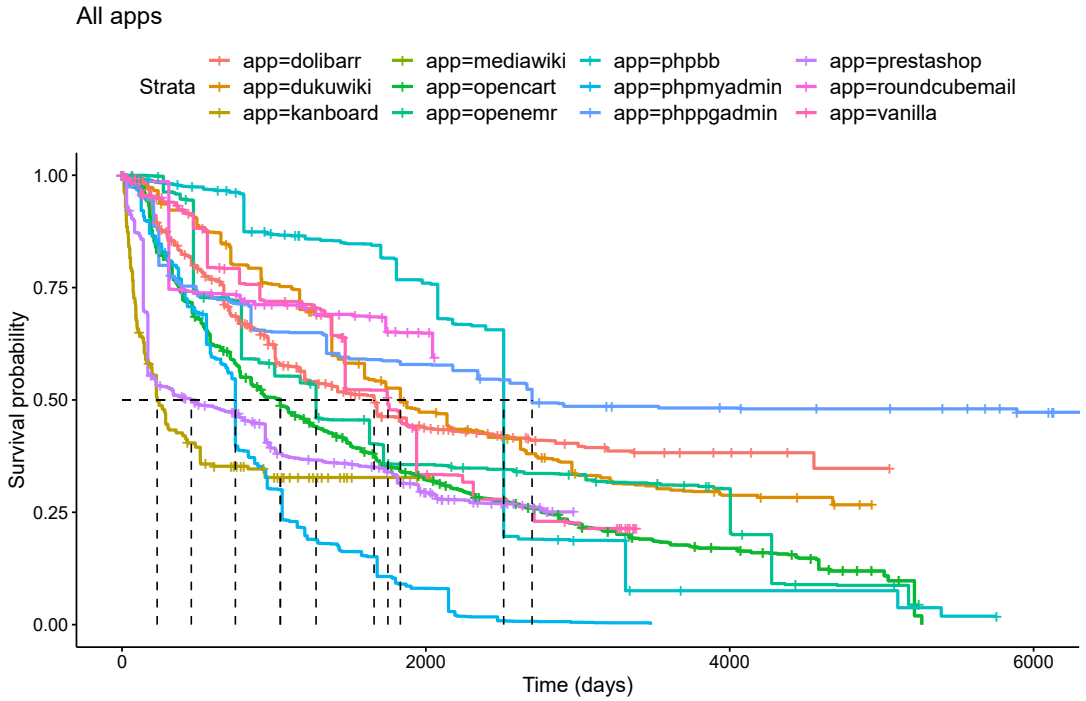


Figure 5.9: Survival curves for 12 apps. Dashed lines denote the median.

5.4.2.3 Values by application

Figure 5.9 shows the survival curves or Kaplan-Meier plots for the 12 apps. The medians (probability of survival =0.5) for each code smell are shown in dashed lines.

Table 5.7: Values of survivability of code smells, by app and all applications

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

Table 5.7 shows the survival variables of interest by applications and their average. *#CS* is the number of unique CS in the file of the application, *#removed* is the number of removed unique CS, *%removed* is the percentage of unique CS removed. The *CS survival time median (days)* is the same as for the other table 5.6 by code smell, but here by application. We show the restricted mean column for comparing when there is no median (column *CS survival time *rmean (days)*), i.e., the average survival counting both the removed smells and the not removed ones, from when the CS first appeared until the end of the study. Because of the high rate of

not removed smells, this value is much higher. We show this column because, for one app, it is not possible to calculate the median with the function *survfit* (not sufficient removed smells).

We analyzed in total 67635 unique CS in all the applications. In table 5.7, the line "all apps" represents the averages for the apps studied, where only 61% of smells are removed.

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

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

We calculated the median of all 67635 CS, which is 1266 days (around 3.5 years); however, if an application contains a high CS density, this will skew this "all apps median" value entirely. We calculated average values for the apps *median*; the average survival time is 1386 days/3.8 years (simple average) or 1323 days/3.65 years (weighted average with size and app total age). Depending on how we calculate, the survival value for all apps and code smells in our study is between 3.5 and 3.8 years.

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

Restricted mean: the last column represents the restricted mean, explained before, and considers the CS not removed. We don't use the value, but the column is shown to serve as comparison, when it is not possible to calculate the median.
We also compared the CS lifespan median to the application's studied lifespan, and this value is around 37%. The percentage value gives more information for generalization than the absolute value.

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

5.4.3 RQ3 - Survival curves for different scopes of CS: *Localized* vs. *Scattered*

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

Table 5.8: Log-rank significance test - comparison of scattered and localized smells : a p-value less than 0.05 means different survival curves

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

Table 5.8 presents the Log-Rank test significance for all the apps. A p-value less than 0.05 means that the survival curves differ between CS types with a 95% confidence level. For half of the web apps(*phpMyadmin*, *OpenCart*, *phpPgAdmin*, *MediaWiki*, *PrestaShop*, *OpenEMR*), the significance is less than 0.05 (in bold), meaning the survival life of the scattered smells differs from the survival life of the local smells. However, this analysis will not suffice because of the low removal rate of code smells, as shown next.

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

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

Table 5.9 presents the numbers and percentages for the CS survival time and CS introduced and removed, separated by scope (localized vs. scattered), given by the column *CS scope*. The column *CS found* and *CS removed* are the code smells found and removed, while the *% removed* is the percentage of the CS removed. The *CS survival time median* is the value in days at which the survival probability is 0.5, and when the removal is small, it is impossible to calculate. Therefore, we also show the *CS survival time restricted mean*, that is, the mean including smells that are not removed at the end of the study and censored with 1 or 0 (for those smells, the ending time is the end of the study).

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

Table 5.9: Code smells found, removed and survival time in days, by scope

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

the application *OpenEMR* because the removal percentages are very close.

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

Analyzing the table 5.9 numerically for the survival time median and for the apps that have the survival time median for both scopes, we find: *MediaWiki* and *PrestaShop* have shorter survival times for the localized CS, and on the contrary, *phpMyadmin* and *OpemEMR* have shorter survival times for the scattered CS. Table 5.9 and table 5.8 need a complementary graphical analysis, because of the CS's low removal rate, especially for the scattered CS.

Figure 5.10 presents the curves of the probability of survival of the two scopes of code smells, localized and scattered, for two selected apps. The graph shows the probability of survival(y-axis) vs. time (x-axis - in days). For the left app (*phpMyAdmin*), the median of the scattered CS (556 days) differs clearly from one of the localized CS (746 days), as the p-value of 0.033 also shows. However, for the second application(*Vanilla*), the median is coincident (1469 days in both cases), as also observed by the p-value of 0.95).

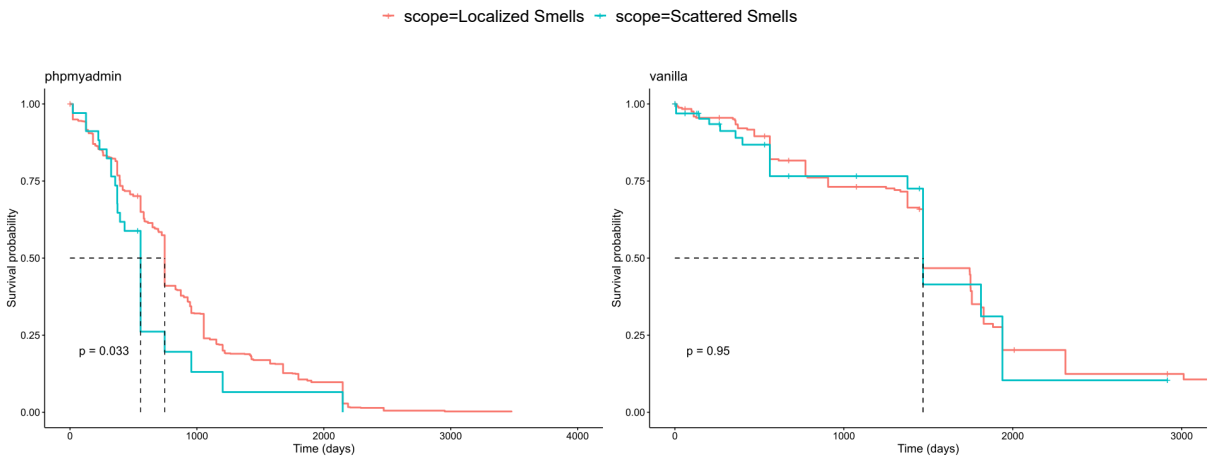


Figure 5.10: Survival curves for localized and scattered code smells, for 2 applications. Dashed lines denote the median.

Analysing the extended plot of graphical curves by scope, on the replication kit, folder RQ3, witch has 12 applications instead of 2 applications (figure 5.10) we find: *DokuWiki*, *OpenCart*, *phpBB*, *phpPgAdmin*, *MediaWiki*, *PrestaShop* all have shorter survival times for the localized CS; *phpMyAdmin*, *phpPgAdmin* and *Openemr* have have shorter survival times for the scattered CS. For *Vanilla*, *Kanboard*, *Dolibarr* and *Roundcube* the study remains inconclusive because even graphically, the survival curves are very similar.

Combining the numerical with graphical analyses, we find: that for five applications(*DokuWiki*, *OpenCart*, *phpBB*, *MediaWiki* and *PrestaShop*), localized CS live less than the scattered; for three applications (*phpMyAdmin*, *phpPgAdmin* and *OpenRMR*], the contrary happens, and for four applications (*Vanilla*, *Dolibarr*, *Roundcube* and *Kanboard*), there is no difference between the two scopes. Half of the applications remove 30% or less scattered CS (table 5.9).

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

5.4.4 RQ4 - Survival curves for different time frames

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

Table 5.10 shows the Log-rank test significance for all the apps. For almost all the applications, except *DokuWiki* and *Roundcube*, the CS survival is different in the first half and second half (p-value less than 0.05).

Table 5.11 contains the values found (CS found and removed including percentage, and survival life median and restricted mean) from the function *survfit grouped by timeframe*. The median of the survival life in days is shorter in the second half for the apps *Dukuwiki* and *phpBB*

Table 5.10: Log-rank test significance - comparison of code smells in two timeframes

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

Table 5.11: Code smells found, removed and survival time in days by timeframe

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

(CS with a shorter lifespan on timeframe 2); however, for the former, we cannot conclude that survival life is different (p-value on table 5.10). For *MediaWiki*, and *phpMyAdmin*, the median in the first half is lower than in the second half (CS in code live fewer days in the first half), although for the second apps, just by a small margin (23 days). For the other applications, we cannot calculate the **median** in the second timeframe because of the low removal rate of the CS, which we show and analyze later. We could use the "restricted mean", but this value includes the censored and the outlier values and gives different values from the survival time (usually higher). Therefore, we will perform the graphical analysis to complete this numerical analysis.

All the applications except one (*OpenEMR*) introduce more smells in the first half of their life. This result is coherent with other studies [30] that found that more smells are introduced in creating the files/classes. Likewise, all the applications except one (the same) remove more smells in absolute number and in percentage (the percentage gives a better measure of density) in the first half of their life. We remember that we treat the time frames as independent (the censoring is also made at the end of the first timeframe). From the values on table 5.11, in the

first half of the life of the applications 64% CS are removed on average, and in the second half, only 26%.

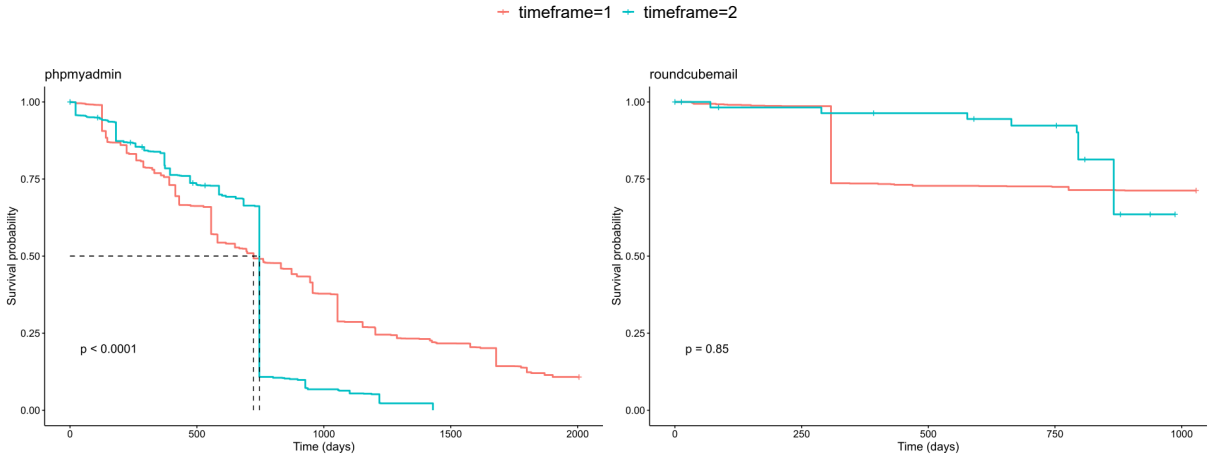


Figure 5.11: Survival curves for code smells in the two timeframes, for 2 applications. Dashed lines denote the median.

Figure 5.11 shows the survival curves for the two timeframes of two selected applications. The graphical representation of the survival curves for all the web apps is in the replication package (in folder RQ4). Analyzing the complete set of the graphs, for 10 of the 12 apps, the survival curves of the first timeframe differ significantly from the second one, being the exception **RoundCube** and *DokuWiki*, as seen in the table 5.10.

We performed the graphical analyses for the ten applications with different survival curves. For *OpenEMR*, the survival curve of timeframe 2 is different and descends much quicker than the one from timeframe 1. The graphical analysis of *phpBB* agrees with the numerical analysis from the table 5.11, so for these two apps, the CS survival time is shorter in the second timeframe (we observe a reduction in the CS survival life in the long term). However, for the remaining 8 of the ten apps, the survival curves are different, and the timeframe 1 survival curve descends much quicker to the lower probability values, indicating that CS's survival time in the first timeframe is shorter.

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

5.4.5 RQ5 - Anomalies in code smells evolution

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

Figure 5.12 represents the relative change of CS number from the previous release and the relative change, in kLLOC, from the previous release. However, as shown in the study design, these absolute values are not enough to spot these anomalies because there can be

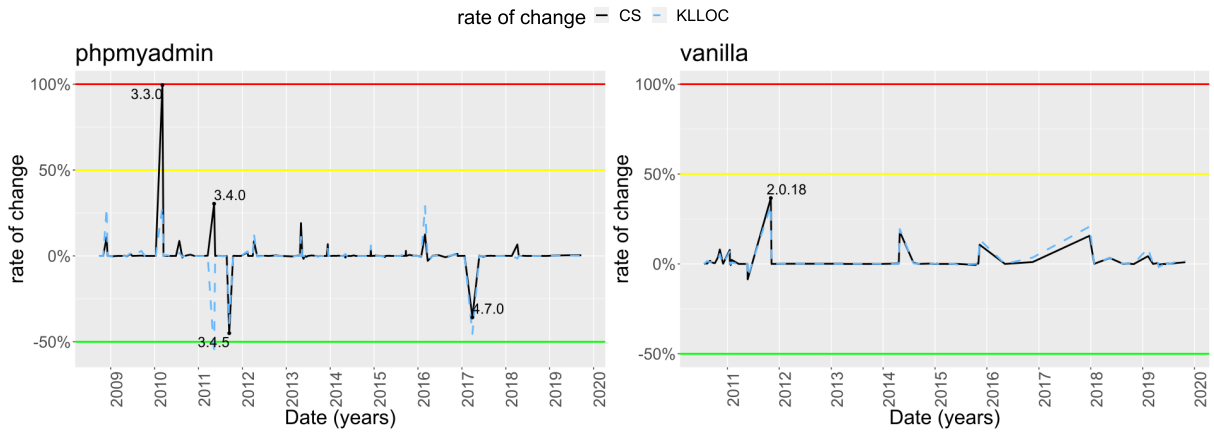


Figure 5.12: Changes of cs and kLLOC

sudden variations in the number of CS accompanied by the same variation in the size of the app, making no change in density.

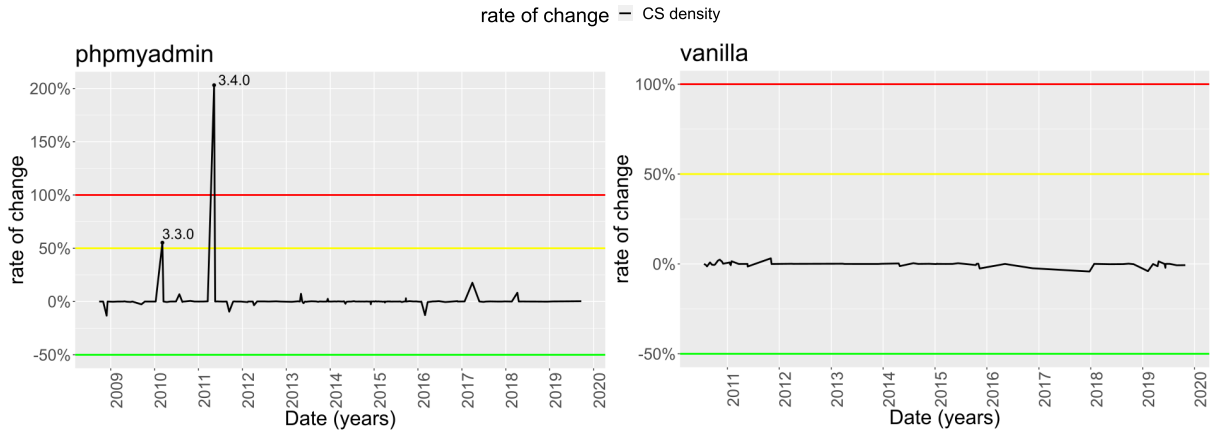


Figure 5.13: Anomaly detection in number of code smells: changes in CS density (CS per LLOC

Figure 5.13 shows the CS density evolution for two select apps, the first with anomalies and the second without anomalies. In the figure, we use lines representing thresholds, signaling an increase of 50% and 100% and a reduction of 50% in the CS density change rate. However, the values of the proposed thresholds can be changed according to application, team, quality, and company, if applicable. The managers or leading developers should choose this.

Table 5.12: Code smells sudden increases

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

Table 5.12 shows the CS density anomalies found, the variance from the previous release to the current release for *CS by kLLOC*, and *Cyclomatic Complexity by LLOC* (aka Cyclomatic

Complexity Density), a long-used objective metric for maintainability prediction [62]. We find those anomalies in 5 applications, but here we represent only the *sudden increases*. For example, *phpMyAdmin* has two anomalies, where the CS rise without the corresponding app size rise. This sudden change can indicate that the code has some problems or big changes in those releases.

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

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

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

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

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

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

5.5 Threats to validity

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

Threats to construct validity concern the statistical relation between the theory and the observation, in our case, the measurements and treatment of the data. We detected the CS using *PHPMD*, and one can question this tool for its accuracy. However, *PHPMD* is used in most of the other tools now (*PHPStorm*, *Codacy* and others), or when they first started (*sonarqube*).

Also, because *PHPMD* runs in the command line, our workflow could use it easily. One way to evaluate accuracy is to use 2 or 3 different threshold sets, but we used the default ones because we wanted to compare different applications.

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

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

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

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

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

5.6 Discussion

5.6.1 RQ1- CS Evolution

For most applications, the absolute number of code smells shows the same tendency as the app size (lines of PHP code, or LLOC as we measured), and the common trend is to increase.

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

If we compare this study to evolution studies in Java, Digkas et al. [42] studied TD (Technical Debt), which is not the same but includes CS, increases for most observed systems, while TD normalized to the size of the system decreases over time in most systems. However, this is not the behavior in the CS in web apps, as most of the CS density is stable over time (with fluctuations).

5.6.2 RQ2- CS Survivability and distribution

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

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

The PHP web apps in the study remove around 61% of the CS inserted. However, in Java, according to Tufano et al. [158], the removal rate is 80%. This value is probably because *Java* is the most studied language regarding CS and has more tools to detect and automatically refactor some CS. Another result, in Java applications, from Peters et al. [119] is that CS lifespan is close to 50% of the lifespan of the systems. Nevertheless, in our findings, PHP web apps' CS survival time median is around 37% of the life of the systems.

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

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

5.6.3 RQ3- CS Survivability by scope

CS occurrences may be removed by explicit refactoring actions or, much less frequently, due to code dropout. Therefore, PHP project managers need to have an evolutionary perspective

on the survival of CS to decide on the allocation of resources to mitigate their technical debt effects. Furthermore, since CS are of different types regarding their scope, project managers must be aware of their evolution, with a particular concern for scattered CS since their spreads may cause more harm and may be harder to refactor without appropriate tooling.

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

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

5.6.4 RQ4- CS Survivability by timeframe

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

For most apps, the survival curves of the two timeframes differ; for most, the CS survival time is shorter in the first timeframe. Studies in Java language [119], found that smells at the beginning of the systems life are prone to be corrected quickly. For the majority of the web apps, our study agrees with these findings.

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

5.6.5 RQ5- Anomalies in code smells evolution

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

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

When is the CS number/density change too high? In factories using control charts, to control a measurement attribute that goes around a value, there are limits equal to 3 times the

variance. In this case, we do not have variances around a value/metric, so we have to define thresholds that development teams or managers can tailor. We believe that a threshold of 50% will be sufficient to raise maintainability alerts. Knowing that we can never remove all the CS from an application, a 50% increase would raise a yellow flag, and a 100% increase would raise a red flag (stop immediately). Looking at table 5.12 we can see that peaks also affect the *cyclomatic complexity per LLOC*, which in turn affects maintainability [62].

We also tried other methods to measure the CS variations, and among them, to apply SPC (Statistical Process Control) techniques with 2 or 3 standard deviations as limits, but we could not get limits due to the nature of the evolution (for long periods, the number of CS was the same, then this value sudden increases). The main problem was that the standard deviation was 0 or close to 0. Moreover, with methods that use the average, for example, a metrics study [41], one must know all the project history, while in the method shown here, the computation at each point in time is based on data collected from the previous and current release.

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

5.6.6 Implications for researchers

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

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

When making evolutionary studies, investigators must inspect the software being analyzed to avoid the folders from other vendors when collecting the sample. For example, in *phpMyAdmin*, if we remove the folders from different vendors from the analysis, we have only 50% of the original code in the zip release. If this step is missed, part of the analyzed code comes from other programs. We perform this step since [133].

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

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

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

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

5.6.7 Implications for practitioners

Removing code smells in a web app without spending time is impossible. Therefore, there will always be a trade-off between releasing quickly and more quality in the code. However, this and other studies suggest several effective ways to mitigate the problem.

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

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

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

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

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

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

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

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

5.7 Chapter conclusions

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

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

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

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

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

Data Availability

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

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

Chapter 6

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

Contents

This chapter presents the studies on server- and client-side code smells: evolution, causal interaction, and statistical causality on issues, bugs and delays to release

6.1 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[47], 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[130, 160].

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 [116]. 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 [71, 122].

PHP is the most used server-side programming language in web development [1]. The research results on CS within the server-side code only (e.g., in [16, 133]) are very similar to the ones reported for desktop apps in the literature [128, 172]. 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). There are three primary forms of building web apps: monolithic apps, distributed apps with separated front-end and back-end (Frontend/Backend), and with micro-services architecture [2].

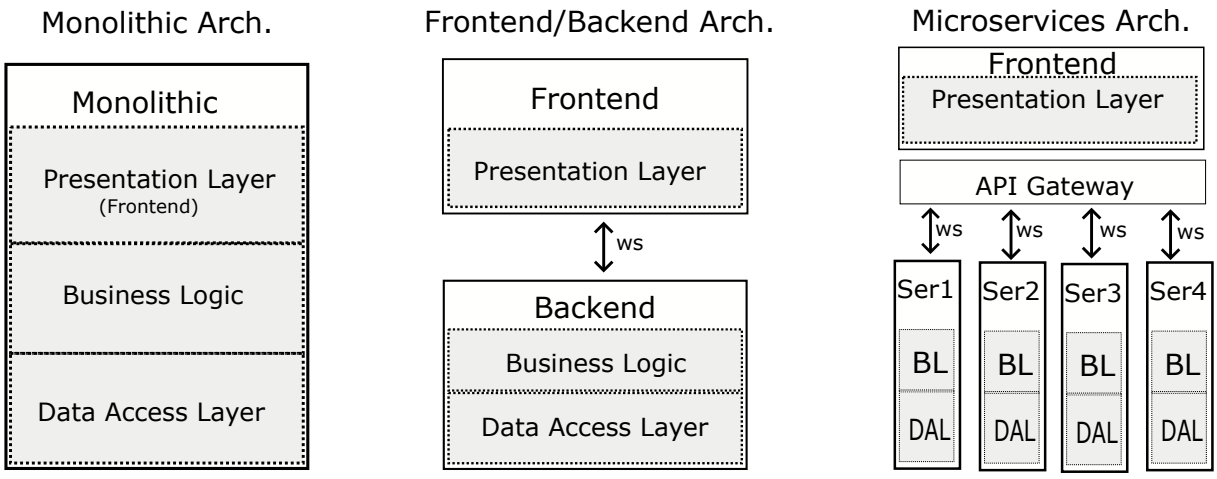


Figure 6.1: Most used architectures of web apps or systems.

Figure 6.1 shows the most used architectures of web apps or systems. The Monolithic architecture (all code in the same codebase) is the most used by far because of less resource usage and historical reasons. When applications grow bigger or add other platforms, they can

[1] https://w3techs.com/technologies/overview/programming_language
[2] https://micro-frontends.org/

need to go to other architectures [3], but this is not always the case [143]. Another form of this picture regarding teams is shown in [4].

We study monolithic applications for several reasons: they are the most common by far; the client and server code are in one codebase, so studying the relations between client and server code is more straightforward. Another reason is that monolithic applications span more years for a longitudinal study like this one.

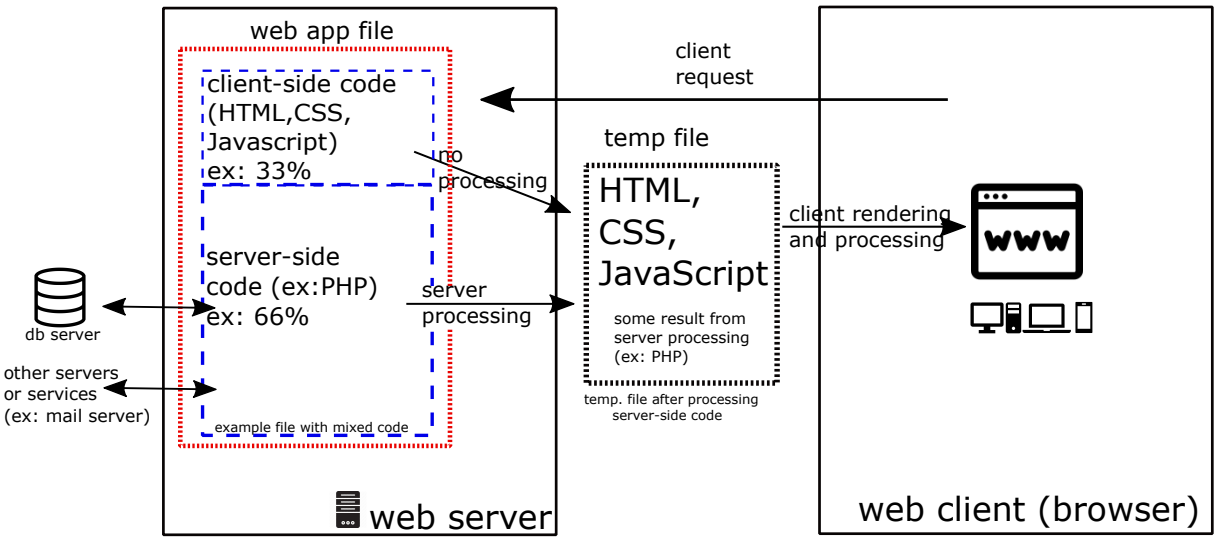


Figure 6.2: Anatomy of a monolithic web app, containing server- and client-side code - very simplified

Fig. 6.2 shows a 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). Percentages in the figure are examples. Exact percentages are in table 6.1. JavaScript is a part of the client code, often more than half of this code. Server code can be PHP, C#, Ruby, Java, Python, JavaScript(node.js) or others. For some of these server-side languages, the server code can separated from the client code, but sometimes the code is entangled. An example of a file with mixed client- and server-side code is shown on listing 6.1.

Listing 6.1: PHP file code example, with mixed client- and server-side code

```
1  <?php $db= new $db(<db credentials>);    //some php code
2  $list=$db->get("table1");
3  ?>
4  <!doctype html>
5  <html>
6      <head>
7          <style>h1{font:somefont;}</style>
8      </head>
9      <body>
10         <script>alert("hello");</script>
```

[3] https://www.martinfowler.com/articles/microservices.html
[4] https://micro-frontends.org/

```
11          <h1>Display List</h1>
12          <?php foreach($list as $line){ ?>
13          <h2 style="font-size:30px"><?php print $line['title']?></h2>
14          <?php } ?>
15          <div onclick="dosomething()">do something</div>
16          <script src="lib.js"></script>
17      </body>
18  </html>
```

Several files of monolithic web applications, especially the ones related to the presentation layer, can contain server code that gets parsed in the server and transformed into client code and client code that runs only on the browser, as in listing 6.1. Thus 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 browser (by different parsers/compilers).

Detection of CS on client-side has been studied [52, 61, 72, 103]. On the server-side some studies have been performed on the survival of CS [133], evolution and sudden variations of CS [135], dissemination and impact of CS in file changes [16], the impact of CS in bugs[17]. These server-side CS studies show similar results to desktop studies done in Java when the questions asked are similar [80, 116].

Therefore, web app CS can be found on the server and the client-side code, but little is known about their evolution. We are interested in learning if these CS evolve similarly and if the evolution of client-side CS impacts the evolution of server-side CS or vice-versa. For example, we want to know if an increase in code smells from client-side code makes CS from the server-side code increase in the same or subsequent releases. Furthermore, we want to analyze if code smells contribute to the evolution of maintainability issues, like reported issues and bugs, in the same or some releases after. Some studies in desktop Java applications already conclude that several metrics contribute to the bugs forecast, and the CS improves the prediction (E.g. [117]). However, they used a cross-correlation approach within a file/class that does not include the possibility of CS in file 1 impacting bugs in file 2. Therefore, an analysis covering these types of relations, system-wide, should be made. Lastly, CS are known to affect the readability and maintainability of an app. Therefore, we want to assess if they impact the app's time-to-release.

In recap, we aim to discover if CS evolution impacts outcome variables (other CS, issues, bugs, and time-to-release) using time-series techniques to assess causality in the same release or previous releases (with lags. with the delays of the time series). The time series are irregular because the release of open-source apps is not regular (a regular time series has measurements taken at regular intervals). The causality inferred from data, if found, does not mean that CS exclusively determines the cause of the outcome but rather that they contribute to the outcome variability and predictability.

The current chapter addresses the evolution of CS in the server-side and client-side code of monolithic web apps, study the relations among the CS themselves, and analyzes the possible causality relations between the various types of CS and reported issues, reported bugs, and time to release of the applications.

The main novelties in the study are: An evolution study with both server and client CS in web apps; the Use of irregular time-series and special correlation techniques for time-series with

different observation granularity; Use of novel statistic techniques to infer statistical causality (e.g., Transfer Entropy) and comparing it with other causality methods(e.g., Granger-Causality). Use typical and monolithic 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.

This chapter 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.

6.2 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.

6.2.1 Evolution of CS

Olbrich et al. [112] described different phases in the evolution of CS and reported that components infected with CS have a higher change frequency. Later, Peters and Zaidman[119] results indicate that CS lifespan is close to 50% of the lifespan of the systems. In Chatzigeorgiou and Manakos[30], it is 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.[158] sustain that most CS are introduced when artifacts are created and not because of their evolution. In Digkas et al.[124], the authors claim that the latest versions of the observed application have more CS/design issues than the oldest ones. They also note that the first version of the software is cleaner. In [42], the authors found that TD (Technical Debt) increases for most observed systems. However, TD normalized to the size of the system decreases over time in most systems. In Habchi et al.[67], the authors conclude that CS can remain in the application code for years before removal, and CS detected and prioritized by linters disappear from code before other CS. Recently, Digkas et al.[41] find that the number of TD items introduced through new code is a stable metric, although it presents some spikes; and also that the number of commits is not strongly correlated to the number of introduced TD items.

6.2.2 CS Impact in issues or defects/bugs

Li and Shatnawi [91] analyzed classes for 6 smells in 3 versions (independently) of 1 open-source system and confirmed a correlation between 3 of the C.S. (God Class, God Method, and Shotgun Surgery) and class error probability. D'Ambros et al. [39] studied the relationship between software defects and the number of design flaws (CS) in 6 open-source software tools and used multiple versions (considered 1 version every two weeks), each with bugs in a subsequent 6-month window. The study was class level. They found that CS (and CS in the evolution of classes) correlate with software defects but not strongly; there is not a design flaw that

consistently correlates more then the others. Olbrich et al. [112] (also referred in the evolution paragraph), studied the correlation between the code smells God Class and Brain Class with the frequency of defects detected in the post-release. They study CS by class, Bugs by class, with a version every 50th commit. The results showed that without taking the class size into account, there is a higher defect rate in the God and Brain classes compared to other classes, but when normalizing a defect rate concerning class size, God and Brain Classes had even smaller values of defect rates.

Marinescu and Marinescu [97] studied 3 versions of Eclipse, 4 CS (class-based), and defects - each of the 3 versions with pre-release and post-release defects. A direct correlation between specific code smell types and defect rates was not confirmed; but when their clients use code smell affected classes, the probability of these clients having defects increases, especially on post-release defects. Zazworka et al. [170] studied four approaches to structural flaw detection, including CS, in 13 releases of 1 system, with 10 types of CS (class and method based). They could correlate 2 of them (Dispersed coupling and God Classes) with higher defect-proneness. The type of defects used was bugs number in the version, bugs number fixed in the version, and bugs number between versions. Ban and Ferenc [10] studied the relationship between antipatterns (CS), bug, and maintainability. They used 1 version of 328 systems for maintainability but just 34 systems for bugs (cross-correlational study) and detected 9 CS. Using the total number of CS and the total number of bugs, they found a significant positive correlation between the number of bugs and the number of antipatterns (CS).

Khomh et al. [80] investigate the impact of antipatterns (CS) on classes in object-oriented systems, studying the relation between the presence of antipatterns and the change- and fault-proneness of the classes and detecting 13 antipatterns in 54 releases of 4 systems. They found that classes participating in antipatterns are more change- and fault-prone than others, and size alone cannot explain this; structural changes affect more classes with antipatterns than others. Paloma et al. [116] extends the previous article and presents a large-scale empirical investigation on the diffuseness of code smells and their impact on code change- and fault-proneness. They used 395 releases of 30 open-source projects and considered 17,350 manually validated instances of 13 different code smell kinds. The results show that smells characterized by long and/or complex code (e.g., Complex Class) are highly diffused and that smelly classes have a higher change- and fault-proneness than smell-free classes.

Another trend in the study of bug prediction models, including metrics from product, process, and CS [117] and previous studies. However, this would be a posterior step in web apps; in the article, we were concerned with assessing CS's relative importance only in the context of bugs and other variables.

6.2.3 CS in web apps or web languages

Detection studies: Nguyen et al. [72] that 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 et al. [52] 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 [103], 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. [61] 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: In Rio and Abreu[133], authors study the survival of CS in web apps and later [134] they study the sudden variations of CS evolution in the life of 8 web applications. This study was extended in [135] for 8 apps, and the final version was [131] 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 in 5 apps.

Impact: These studies include Saboury2017 et al. [140], 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, Johannes2019 et al. [75] show the results: files without CS have hazard rates of at least 33% lower than files with CS. In Amanatidis et al.[3] 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. [16] study diffusion and impact to change-proneness. They extended the study in [17], 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 [80, 116]), 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.

6.2.4 Evolution studies in PHP

Studies of this type include [83], 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 [2], 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.

6.2.5 Studies in SE with Granger-causality and Entropy

Granger-causality: Some studies have already used Granger-causality. In Couto et al. [37], they propose evidence between source code metrics (as predictors) and the occurrence of defects, using Granger Causality Test to evaluate whether past variations in source code metrics values can be used to forecast changes in time series of defects. They evaluated the approach in several life stages of four Java-based systems and reached an average precision greater than 50% in three of the four systems evaluated.

Paloma et al. [114] analyzed 13 code smells in 30 software systems to assess the extent to which code smells co-occur, which code smells co-occur together, and how and why they are introduced and removed by developers. They found: 59% of smelly classes are affected by more than one smell; six pairs of smell types co-occur frequently; method-level code smells may be the root cause for the introduction of class-level smells; code smell co-occurrences are generally removed together as a consequence of maintenance activities.

Sharma et al. [145] implemented smell detection support for seven architecture smells and mined 3 073 open-source repositories to study architecture smells characteristics, investigate correlation, and empirically investigate the causation relationships between seven architecture and 19 design smells (causation study within 5 repositories, showing values for one). Findings: smell density does not depend on repository size; architecture smells are highly correlated with design smells; most design and architecture smell pairs do not exhibit collocation; causality analysis reveals that design smells cause architecture smells.

Entropy: Some studies used entropy, but in prediction models, with regressions. Gupta et al. [66] 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. 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) [149] and the complexity of code changes using entropy based measures [27].

6.2.6 Related work conclusions

There need to be more studies 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. Furthermore, studies in impact deal with releases of systems and subsequent bugs to that release, making them cross-correlational studies (or, in some cases, mixed studies - we asked a statistical expert).

We need longitudinal studies dealing with causality inference to uncover how CS in past releases impacts bugs, issues, and time to release in the same and following releases.

6.3 Methods and Study Design

In this study, we investigate the evolution of CS in 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.

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. According to our group of specialists with more than ten years of experience, 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.

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

6.3.1 Research questions

- **RQ1 – How to characterize the evolution of CS in web apps on the server and client-sides?** – This question will lead to uncovering: 1) the evolution of the different CS on the server-side and client-side (further divided in embed and JavaScript), both in absolute numbers and density; 2) the evolution of the individual CS in their group, for the three groups ; 3) the evolution of the CS as groups (server side-CS, client-side embed CS, and client-side JavaScript CS).

- **RQ2 – What is the 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) 1) evolve in the same way (by time-series correlation) and 2) 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 - What is the impact of server- and client-side code smell evolution on web app' reported issues?** – The issues reported measuring the potential quantity of work/maintainability tasks a team has to perform in an app. This question will help us verify if CS hinders the number of issues and which code smells affect the number of issues evolution in the same release or with CS from previous releases (up to 4 releases behind). We will study: 1) the relationship between individual CS and issues (with time series correlations); 2) the causality relationships between individual CS and issues; and 3) the causality relationships between CS groups and issues.

- **RQ4 - What is the impact of CS evolution on a web app's 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 intensity change causes changes in the intensity of the bugs reported (we consider the date of the report on the bug) in the evolution of the web app. We divide this study into 3 parts: 1) Relationship between individual CS and reported bugs; 2) Causality relationships between individual CS and bugs; 3) Causality relationships between CS groups and bugs.

- **RQ5 - What is the impact of CS evolution on "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 [93, 166, 167], but we want to find the answer inferred from observed data. This study is divided into two parts: 1) Causality relationships between individual CS and time to release and 2) 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 monolithic web apps. Because of this, some causal relations between both sides' CS evolution are expected. Therefore, we want to understand if the code smells (CS) of the client- and server-side of the code evolve together or are independent and have some relationship or causality between them.

For the remaining three 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. To that end, we will use the statistical and time series methods described in the appropriate subsection capable of statistically uncovering these relations and causal inference. 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.

6.3.2 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 apps must also have client-side code -i.e., monolithic web applications. 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/monolithic 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 6.1: Web apps sample used in studies. Sizes and code percentages are averaged by all releases studied

				AVG KLOC			% Code Type		
Name	Purpose	#Releases(period)	Versions	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%

Table 6.1 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 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.

6.3.3 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 study, 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 PHP and JavaScript CS, we assembled a group of CS specialists, and only the CS that were not ambiguous were considered. Another reason to consider the CS was the detection possibility.

6.3.3.1 Catalog Server-side CS

Table 6.2: Characterization of server-side Code Smells. Names of Code Smells are the ones presented by the tool. "(Excessive)" was added to denote a CS and not a metric.

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

For the server CS, we used *PHPMD*, an open-source tool that can detect CS in PHP [46]. The chosen subset of server-side CS, presented in table 6.2, corresponds to the 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 6.2) for comparability 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.

6.3.3.2 Catalog Client-embedded CS

Table 6.3: 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

The client-embedded CS were among the first reported in the literature for web apps [52, 72]. 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 [72], we developed another one dubbed *eextractor*[5] and allowed us to collect the CS represented in table 6.3. The last smell is proposed by us and refers to jQuery, a widely used JavaScript library designed to simplify HTML DOM tree traversal and manipulation, event handling, CSS animation, and Ajax.

6.3.3.3 Catalog Client JavaScript CS

JavaScript CS for client-side were first reported here [52], but since then used in other studies besides the client-side programming, for example [140] that uses mostly JavaScript libraries as sample. For extracting the client JavaScript CS described in table 6.4 we used *ESLint*, a pluggable and configurable linter tool for identifying and reporting on patterns in JavaScript [169].

Table 6.4: Characterization of Client JavaScript Code smells. Names of Code Smells are the ones presented by the tool. "(Excessive)" was added.

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

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

6.3.4 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.

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* 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`[6] `- SLOCCount lines of code` [7] 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, 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 - Part of the applications 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 [4, 7, 151]. 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.

[6] including HTML, CSS, JavaScript
[7] measured with *cloc –use-sloccount*

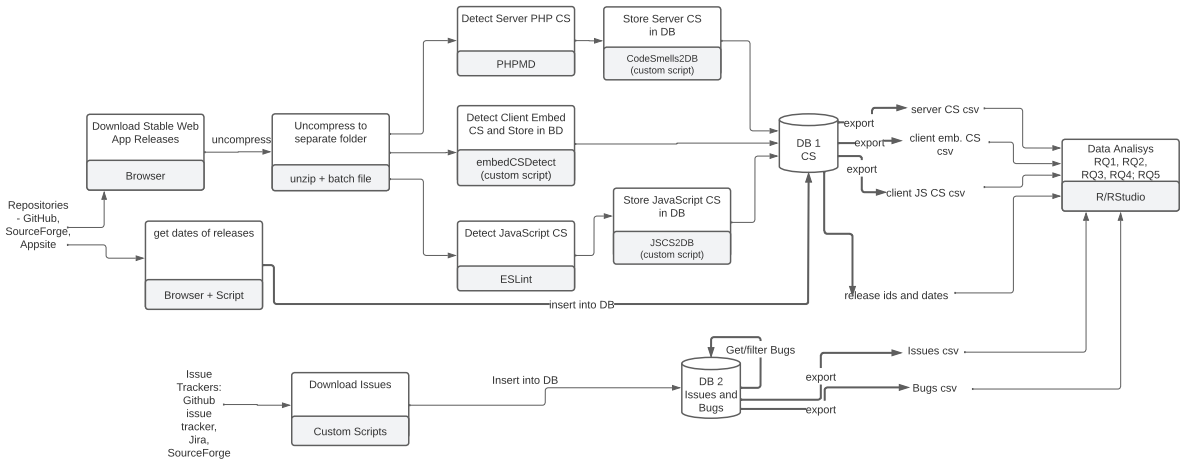


Figure 6.3: Study design/methodology - Universal Project Notation (UPN)

The figure 6.3 shows the study design to collect data, extract CS and analysis.

6.3.5 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 RQ5.

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 issues by release. After we downloaded the issues and put them in a separate database for issues, we had to aggregate the data to make the second set of studies methodologies functional (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 issues by reported day, and there was no

need to aggregate by release. After, we exported the issues by day and the aggregated issues by release to two datasets (per app) available in the replication package. Please see figure 6.5 for the two time-series of issues, one by release and the other by day.

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

We extracted the bugs from the issues, as described in the methodology of RQ4, and we performed data preparation operations similar to the issues data preparation to analyze the CS relation to bugs.

6.3.6 Irregular time series analysis and causality

The software releases are not equally spaced in time, leading to irregular time series. Because we detect the CS from the software releases, they are irregular time series, i.e., not equally spaced in time. 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.

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 [8]. 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 (Granger Causality (GC)) method to detect linear relations between the time series and Transfer Entropy (Transfer Entropy (TE)) 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.

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

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

To calculate correlations between unevenly sampled time series [129], we used the *cor_ts*[9] function from the R package **Bincor** [121]. 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 [121], [106]. The *cor_ts* function can be applied to irregular time series where observations occur at the same time moments and series with different timescales.

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,

$$P\{y_{t_1} \leq b_1, ..., y_{t_n} \leq b_n\} = P\{y_{t_1+m} \leq b_1, ..., y_{t_n+m} \leq b_n\} \tag{6.1}$$

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 do 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 [146], [104].

6.3.6.2 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 [8], [68], [147]. 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 [64, 65] 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.

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

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,

$$y_t = \sum_{i=1}^{p} \alpha_i y_{t-i} + \sum_{i=1}^{p} \beta_i x_{t-i} + u_t \tag{6.2}$$

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:

$$H_0 : \beta_i = 0, \quad i = 1, \cdots, p \tag{6.3}$$

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

Granger Causality 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 [144], 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 [51].

6.3.6.3 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 [13, 141] 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:

$$H(X, Y) = -\sum_{x \in X} \sum_{y \in Y} P_{X,Y}(x, y) \log P_{X,Y}(x, y). \tag{6.4}$$

and the conditional entropy is defined by the following:

$$H(Y \mid X) = H(X,Y) - H(X) \tag{6.5}$$

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

The Transfer Entropy [141] can be defined as the difference between the conditional entropy of each variable in the system:

$$TE(X \to Y \mid Z) = H\left(Y^F \mid Y^P, Z^P\right) - H\left(Y^F \mid X^P, Y^P, Z^P\right) \tag{6.6}$$

which can be rewritten as a sum of Shannon entropies:

$$TE(X \to Y) = H\left(Y^P, X^P\right) - H\left(Y^F, Y^P, X^P\right) + H\left(Y^F, Y^P\right) - H\left(Y^P\right) \tag{6.7}$$

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

$$\widehat{TE}_{X\to Y} = TE_{X\to Y} - TE_{Y\to X}. \tag{6.8}$$

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, which can quantify the information flow between two stationary time series and its statistical significance using Shannon transfer entropy [144] and Renyi transfer entropy [139]. A core aspect of the provided package is to allow statistical inference and hypothesis testing in the context of transfer entropy.

6.3.6.4 Mapping between Granger-causality and Transfer Entropy

It has been shown [11] 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:

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

the following can be proved [11]:

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

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

6.3.7 Methodology for each RQ

6.3.7.1 RQ1 – Evolution of CS in web apps on the server and client sides

To answer this question, we performed several analyses. First, we analyzed the evolution of the various CS in the proposed catalog. i.e., server-side CS, client-side embed CS, and client JavaScript CS, in absolute values and density.

Then, we measured the correlation of the various CS within their group for the three groups (server-side, client-side embed, and client-side JavaScript). We used the standard R (cor) correlation and cross-correlation tables. We also computed the average correlation between all the apps.

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.

6.3.7.2 RQ2 – Relationship between client-side CS and server-side CS evolution

In monolithic web applications and systems, the server-side and client-side code are in the same code base and sometimes in the same file (although they run at different times). So we want to study if the evolution of one of them impacts the other. In most web projects, because they share the same code base or even the same file, the client code is developed prior to the server code, so we can expect past relations of client code to impact the most recent versions of server code. However, not all projects use the same method; in some, the server code (and the respective CS) can appear before the client code. The client CS are divided into two groups because sometimes, not always, the developing teams are different. These assumptions are based on interviews with developers with over ten years of experience.

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 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.

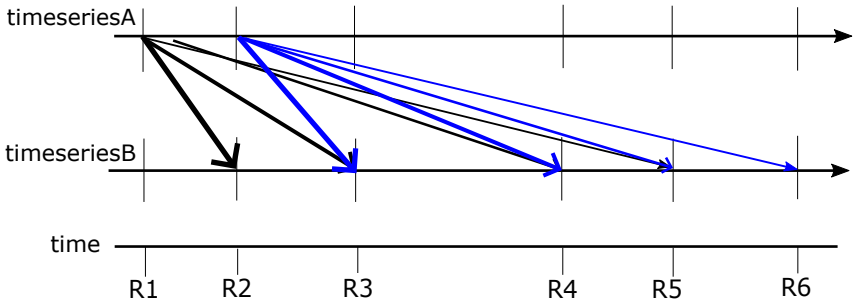


Figure 6.4: Previous releases in timeseries 1 impact next release on timeseries2

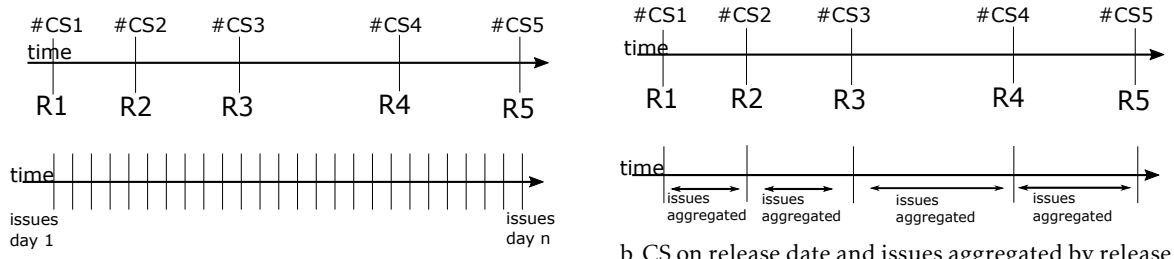
We used Linear regression models to test the causal inference between CS in the same release. We employed the R function "lm", which stands for "linear model". Next, we applied Granger-causality to test for causal inference for CS data, considering lags (time delays) between one and four. The "lags" 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 6.4 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.

We employed *grangertest* (from *lmtest* R package) and *transfer entropy* (from the R package *RTransferEntropy*) functions to achieve the two studies referred to above. 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.

6.3.7.3 RQ3 - Impact of CS intensity evolution (server and client) on the maintainability

The number of issues measures the potential work to be done by a team. To answer this impact 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 various 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 issues: Similarly to RQ2, we analyze causality by handling Linear regression, Granger Causality, and Transfer entropy. Linear regression assesses the impact in the same release, while Granger causality and transfer entropy evaluate the impact on previous releases, where we considered historical moments from lag 1 to lag 4. For the causality studies, we had to aggregate the issues by release date, as shown in what follows:
Issues 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*

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.



a CS on release date and issues by reported date

b CS on release date and issues aggregated by release date

Figure 6.5: Comparison of irregular series of CS and issues

Figure 6.5 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 issues 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 issues, 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.

6.3.7.4 RQ4 - 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 the literature [4, 7, 151] 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.

For correlation and causality inference, we used the same methods as in the questions before.

6.3.7.5 RQ5 - 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.

6.3.8 Function selection and parameter estimation

Correlations: We tested standard correlation and time series correlation *cor_ts*, which can be applied when the time series are of different granularity and irregular. However, when the dates axis in the time series are coincident (by release date), *cor_ts* will give the same results as *cor*. Because of this, we used *cor* in RQ1 and RQ2, and *cor_ts* in RQ3 and RQ4. The alternative of using cor_ts in RQ3 and RQ3 would be aggregating (binning) the issues and bugs by release and using the standard correlation.

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 [10]. 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 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.

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

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 [11].

6.4 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.

6.4.1 RQ1 – Evolution of CS in web apps on the server and client sides

To answer this question, we performed several analyses, individual and group CS evolution characterization and the evolution and trends of the 3 groups.

6.4.1.1 Individual Code Smell density evolution

We first analyzed the evolution of the various individual CS divided as server-side, client-side embed, and client JavaScript CS.

Figure 6.6 illustrates the individual CS density evolution for two applications chosen from the dataset in the corresponding period. Some of the names of the code smells do not have the word *"Excessive"*; we used the original name from the tools(*PHPMD* and *ESlint*) names of the code smells. The replication package contains the complete evolution charts for all applications, absolute CS and CS density, and application size. There is no global deterministic trend for the app on the left because this application suffered a lot of refactoring during its life. However, the server-side CS density dynamics of other applications (including the application on the right) are stable around the mean with few volatility peaks. Nevertheless, the major trend in CS (client and client JavaScript) seems to be decreasing. We can observe some peaks in the JavaScript evolution in some apps, which will be explained in the discussion.

Some similarities can be observed between the variables (CS), characterized by common patterns for increasing and decreasing periods. To check this similarity, we performed a correlation table between the smells of the same group.

6.4.1.2 Relation of CS density evolution within the same group

Figure 6.7 represents the correlation tables between CS of the same group for one application, *phpMyAdmin*. There is a high degree of correlation between CS of the same group/type.

[11] avaliable in https://github.com/studywebcs/data

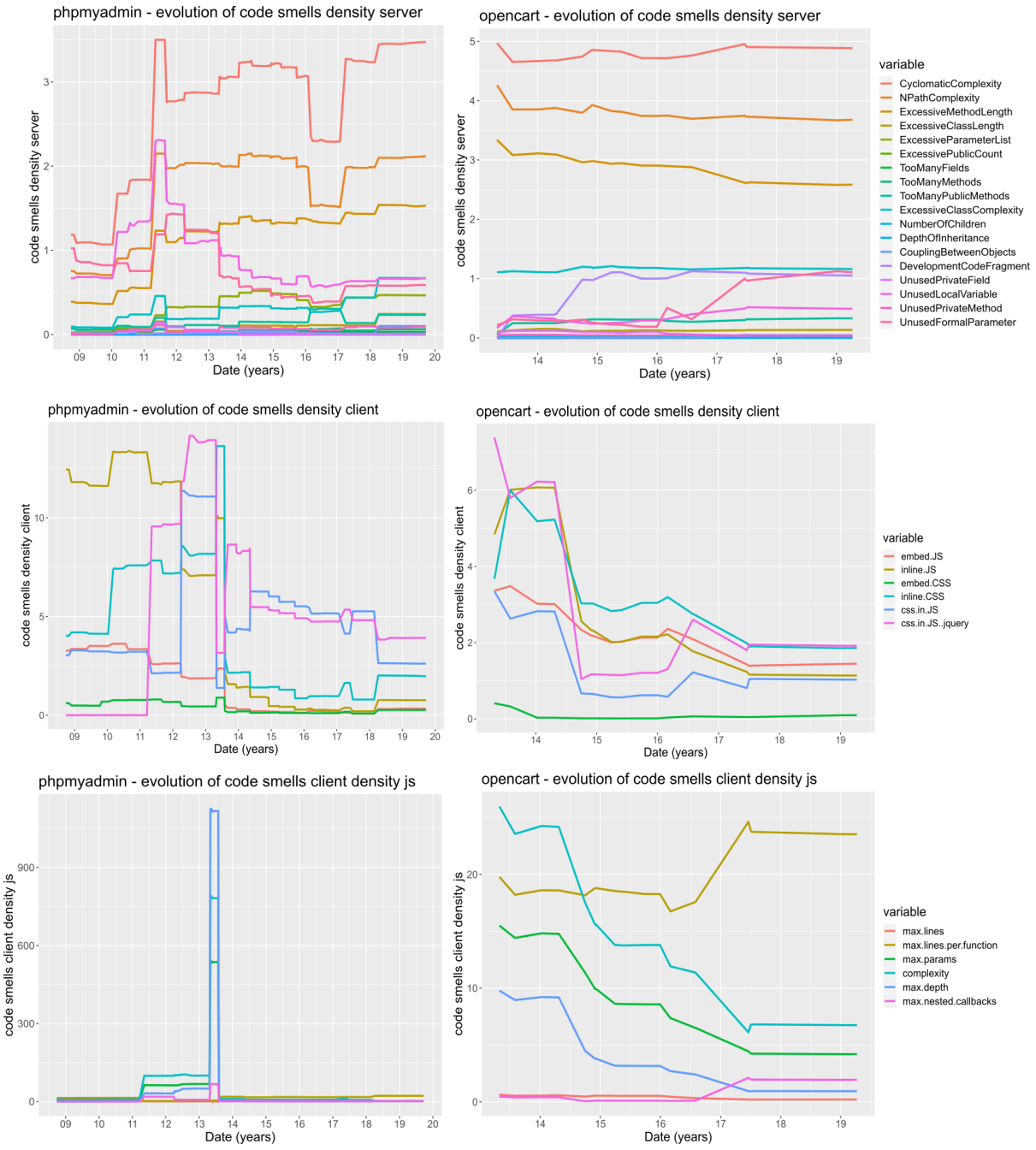


Figure 6.6: Individual Code Smells density evolution for 2 apps

However, in the server correlation table, some CS are less correlated, namely the scattered CS (among classes: *NOC Coupling*), *"development code fragment"*, and *"too many fields"*. In addition, some of the "unused" CS are correlated between them but not with the other CS. Similar to the correlation table in *phpMyAdmin*, there is a high degree of correlations in the server CS groups, with a few exceptions, namely some localized CS related to unused code. In the client-side CS, the number of CS that correlate is less, as some CS are independent. The most independent ones are the ones related to CSS. In JavaScript CS, there are a lot of high correlations. However, function size *("max.lines")* does not correlate with the other CS in some apps.

Average of correlations of all apps, CS within the same group - Tables 6.5, 6.6 and 6.7

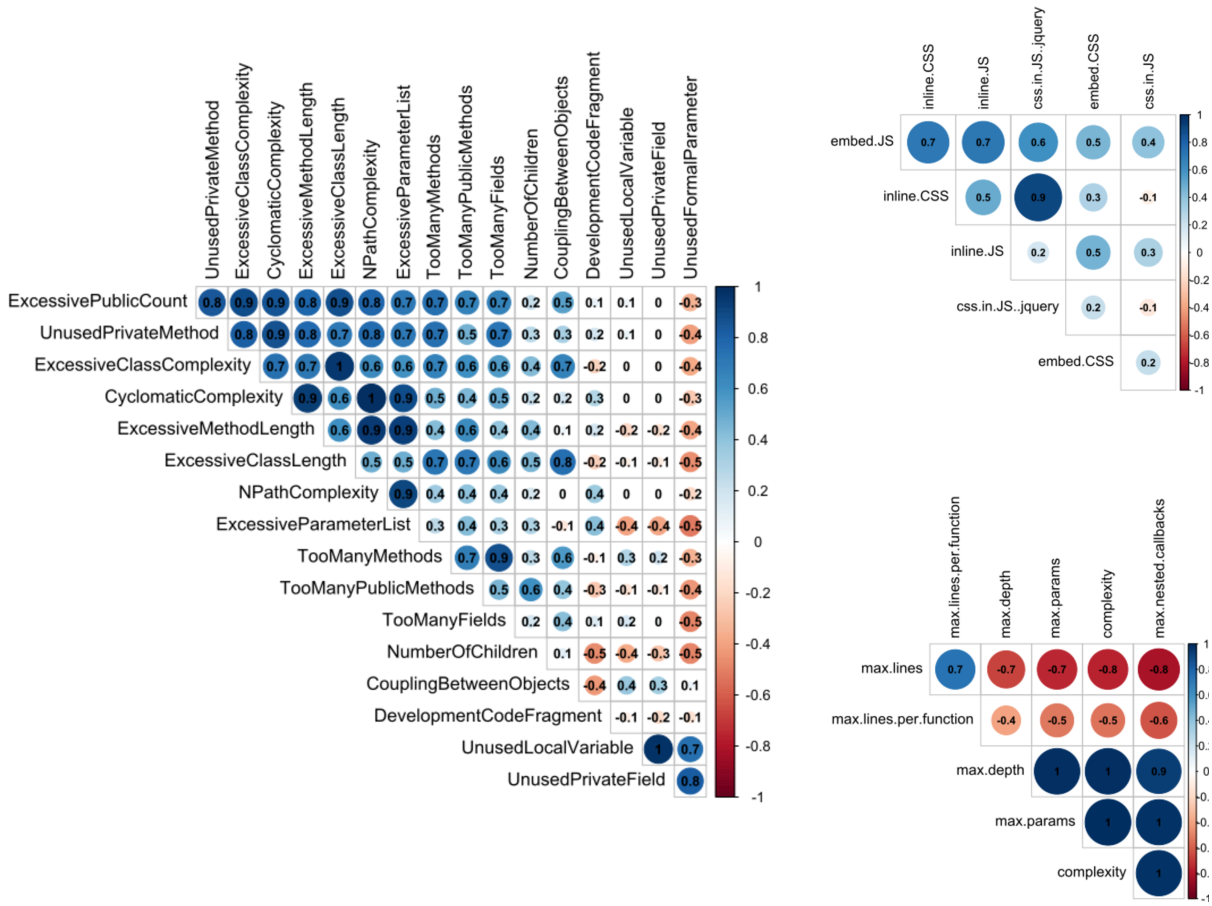


Figure 6.7: Correlation of Code Smells of same group in phpMyAdmin.

present the averages for all applications of the previous study.

Table 6.5: Averages of correlation of individual server-side Code Smells in all applications. Bold: averages > 0.3; bold with gray background: averages > 0.5

	(Ex.) CyclomaticComplexity	(Ex.) NPathComplexity	ExcessiveMethodLength	ExcessiveClassLength	ExcessiveParameterList	ExcessivePublicCount	TooManyFields	TooManyMethods	TooManyPublicMethods	ExcessiveClassComplexity	(Ex.) NumberOfChildren	(Ex.) DepthOfInheritance	(Ex.) CouplingBetweenObjects	DevelopmentCodeFragment	UnusedPrivateField	UnusedLocalVariable	UnusedPrivateMethod	UnusedFormalParameter
(Ex.) CyclomaticComplexity		**0.84**	**0.68**	**0.41**	**0.35**	0.2	**0.32**	**0.33**	**0.41**	**0.69**	**0.37**	0.04	**0.37**	**0.32**	-0.1	0.18	0.21	**0.31**
(Ex.) NPathComplexity			**0.8**	**0.49**	0.3	0.15	**0.42**	0.16	0.2	**0.55**	0.28	0	**0.33**	0.18	0.08	0.02	**0.32**	0.1
ExcessiveMethodLength				**0.55**	**0.32**	0.16	**0.42**	0.22	0.24	**0.52**	0.22	0.04	0.29	0.16	0.07	0.07	**0.32**	0.08
ExcessiveClassLength					0.19	**0.46**	**0.5**	**0.49**	0.3	**0.52**	0.22	0	**0.38**	0.17	0.12	0.09	0.3	0.07
ExcessiveParameterList						0.07	-0.02	0.04	-0.06	0.2	0.14	0.02	0.23	0.09	0.04	-0.04	0.25	0.01
ExcessivePublicCount							**0.49**	**0.61**	**0.46**	**0.31**	0.23	-0.06	0.26	0.3	0.05	0.09	0.22	0.22
TooManyFields								**0.53**	**0.34**	0.3	0.21	0.02	0.28	0.3	0.12	-0.03	0.21	0.09
TooManyMethods									**0.56**	**0.34**	**0.31**	0	0.18	**0.42**	-0.11	0.15	0.06	0.26
TooManyPublicMethods										**0.55**	0.27	-0.03	0.1	**0.44**	-0.2	0.27	-0.05	**0.46**
ExcessiveClassComplexity											**0.34**	0	**0.34**	0.25	-0.02	0.2	0.3	0.28
(Ex.) NumberOfChildren												0.05	0.28	0.14	-0.15	-0.03	0.11	0.3
(Ex.) DepthOfInheritance													-0.02	0.04	0.03	-0.02	-0.04	-0.01
(Ex.) CouplingBetweenObjects														0.09	0.1	-0.04	0.18	0.06
DevelopmentCodeFragment															-0.08	0.09	-0.08	0.27
UnusedPrivateField																0.02	0.24	-0.03
UnusedLocalVariable																	0.09	**0.5**
UnusedPrivateMethod																		-0.03
UnusedFormalParameter																		

Table 6.5 shows the averages of correlations between server-side CS. The bold values are averages greater than 0.3, and the bold with a gray background are averages greater than 0.5.

The highest correlations are between complexity CS and CS related to excessive size.

Table 6.6: Averages of correlation of individual client-side embed Code Smells in all applications. Bold: averages > 0.3; bold with gray background: averages > 0.5

	embed.JS	inline.JS	embed.CSS	inline.CSS	css.in.JS	css.in.JS..jquery
embed.JS		**0.71**	**0.47**	**0.77**	0.11	-0.05
inline.JS			**0.33**	**0.57**	0.18	-0.07
embed.CSS				**0.42**	-0.04	-0.13
inline.CSS					0.04	0.08
css.in.JS						0.01
css.in.JS..jquery						

Table 6.6 shows the averages of correlations between client-side embed CS. The highest correlations are between both JavaScript embed CS(inline and embed) and *inline.CSS* with both *embed.JS* and *inline.JS*.

Table 6.7: Averages of correlation of individual client-side JavaScript Code Smells in all applications. Bold: averages > 0.3; bold with gray background: averages > 0.5

	max.lines	max.lines.per.function	max.params	(Ex.)complexity	max.depth	max.nested.callbacks
max.lines		0.03	0.04	0.08	0.22	-0.29
max.lines.per.function			0.01	-0.09	0	0.11
max.params				0.82	0.55	**0.39**
(Ex.)complexity					0.6	**0.35**
max.depth						0.24
max.nested.callbacks						

Table 6.7 shows the averages of correlations between client-side JavaScript CS. The highest correlations are between both *(Excessive)Complexity* and *max.parameters* and between *max.depth* with both *(Excessive)Complexity* and *max.parameters*.

The averages of the correlations show that some correlations 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 the next questions.

We show the correlation graphs of all applications in the appendix.

6.4.1.3 Evolution of CS as a group

Next, we evaluate the evolution of CS grouped.

Figure 6.8 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 hoped that all the CS would decrease with time, but this is not the case; some even increased. 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 figure 6.8, in the client-embed smells and client-JavaScript smells, on *OpenCart*.

Figure 6.8: Evolution of Code Smell groups in web apps (density/by size)

Table 6.8: 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 6.8 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: CS within the same group: Most of the Code Smells on the same side/group have the same tendency, except on server-side. We identified the CS that correlate more between them in all applications and on average. CS as groups: The dominant trend, for most applications, in server-side CS 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.

6.4.2 RQ2 - Relationship between server- and client-side Code Smell evolution?

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." To avoid overloading the table information, we used symbols to show the statistically relevant relations, and the exact values can be found in the appendixes.

6.4.2.1 Correlation between Code Smell groups

Table 6.9: Correlation between Code Smells groups. A plus sign indicates correlation > 0.3

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

Table 6.9 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 [1, 81, 126]. 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.

6.4.2.2 Causal inference between Code Smell groups

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 [2] 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 6.10: Statistical causality between CS density groups using Linear regression, Granger causality (lag 1 and 2) and Transfer Entropy (lag 1 and 2). Black dots and white dots represent statistical significance of 0.05 and 0.10, respectively, in one application.

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

Table 6.10 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 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 6.10 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 separate 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: **Correlation analysis** within the same release - We observed that half of the applications correlated between the server-side CS and client-side CS. Interestingly, we noted a correlation between embedded client CS and JavaScript client CS in a quarter of the applications. **Causal Inference** from previous releases - From our analysis of 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. In the case of Lag2 (from two releases before), the most notable TE was identified from the client to the client-side JavaScript and the server-side CS.

6.4.3 RQ3 - Impact of server- and client-side CS evolution on web app' reported issues

This section presents the results for the impact of CS on issues, a measure of the team's potential work. First, we present the results of *cor_ts*, the specialized correlation for time series. Next, we present the results for the causal inference from the CS to issues. For the causality inference studies, we had to aggregate the issues, as we described in the subsection *Methodology for each RQ*. Finally, we evaluate the same causality from CS groups to issues.

6.4.3.1 Relationships between CS and issues

Table 6.11: Time-series correlation (cor_ts) between CS and issues (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	+++

Code Smells	#correl.
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	++++++

Two of the applications (*phpPgAdmin* and *MediaWiki*) do not have a large enough number of reported issues, so for this study and the next (RQ3 and RQ4), we only use ten applications. Table 6.11 shows the resume of the correlations between the 30 CS individually (server-side, client-side, and client-side JavaScript) with release data measurements and the issues time series (with daily measurements). Each "plus sign" means a correlation (made with cor_ts) for

an app greater than 30%. In the replications package, in folder RQ3, tables present the numbers: the statistical significance of cor_ts and additional classifications between correlations (more than 0.3 and more than 0.5). We observe several correlations between CS and issues. Prior, we studied this linear correlation (cor), aggregating the issues by release, and the results were lower than those presented with "cor_ts" in table 6.11.

The complete table (in the appendix) shows that three of the applications have almost no correlation at all (*OpenCart* and *phpBB* with just one, and *Kanboard* with only three). These applications behave differently than the others regarding the issues.

These results mean that in table 6.11, when we have 6 or 7 "plus" signs, there is a correlation between CS and issues in 6 or 7 apps stronger than 0.3. From the server-side, almost all CS have correlations except CS's "unused group." On the server-side, the *(excessive)CyclomaticComplexity* and *ExcessiveMethodLength* are the CS within seven apps exhibiting correlation, followed by *(excessive)NPathComplexity*, *TooManyFields*, and *ExcessiveClassComplexity* within six apps. On the client-side, the *embed.CSS* (mixture of CSS in HTML) is the CS with correlation in more apps, seven. Finally, the CS *"max.nested.callbacks"* on client-side JavaScript is the CS that more applications exhibit correlations.

6.4.3.2 Causality relationships between CS and issues

We also wanted to study the causality relationships between CS and Issues. To this end, we used the same methodologies as before (Linear regression, Granger-causality, and Transfer entropy) but with the issues aggregated as explained before. In these three analyses, we can work with densities (divided by size) both in CS and issues, as release dates aggregate them, and we can measure the app size on the release date. The density helps avoid the possible relation increase because of the trend (both having the same trend, for example) and raises the probability of them being stationary without any transformations (a requisite for entropy transfer analyses). We also performed the study with the absolute value of the issues (in the appendix or the replication package).

Table 6.12 shows the linear regression(LM), Granger causality (lag1 and lag2), and Transfer Entropy of CS/KLOC to Issues density (lag1 and lag2). Lag1 means one release behind, and lag2 means two releases behind the release of the issues. Each black dot represents one app with statistical significance ($p<0.05$), and the white dot means an app with a statistical significance of 0.10 in the statistic used in the column. The division by KLOC was made the same way before; for the CS, we only divided the density of the respective language or set of languages. However, issues are divided by total KLOC (except the folders of the third-party libraries) for issue density. The first 18 CS are from the server-side/PHP, the following 6 are the client-side embed CS, and the last 6 are the client-side JavaScript CS.

Linear regression can verify causality from CS to issues in the same release. We also measured the $r\hat{2}$ in the data (Replication package). From the values measured, the CS that have a more immediate impact on the density of issues are from the client-side, namely: *embed.CSS* and *inline.CSS*, from embed client CS, and *max.lines*, *max.lines.per.function*, *max.param*, *max.depth*, *and max.nested.callbacks* from the JavaScript client CS. Almost all the apps have significant values, except two of the apps, *OpenCart* and *Dolibarr*, do not have any relations with statistical

Table 6.12: Statistical causality from CS density to Issues density relations (10 apps) using Linear regression, Granger-causality (lag 1 and 2) and Transfer Entropy (lag 1 and 2). Black dots and white dots represent statistical significance of 0.05 and 0.10, respectively, in one application

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	••••	•	•	•••○○	•••

significance in LM.

In the **Granger causality** columns, we found some causalities from CS density to issues density in columns *Cau lag1* and *Cau lag2*. There are more causality relations in the lag1 column than in the second column. The overall trend for lag3 and lag4 (replication package and appendix) is that the number of causal relations decreases with lag. This result is consistent with the cross-correlation study we made before inspecting the maximum correlations to perform in the study (not shown here). However, there are some exceptions, for example, *UnusedFormalParameter* and *(Ex.)CyclomaticComplexity*; their effect is smaller with lag1 than with lag2. The first can be explained by a "forgetting of the meaning" effect. Almost all the apps have CS with statistical significance, except for *OpenCart* and *Roundcube* with none.

Before studying for **Transfer entropy**, we must ensure that the time series is stationary. The 30 CS time series for each app in absolute number are not stationary, but the CS density

time series are. Both the issues (in absolute number) and issues density are stationary, and we studied both; in the appendix, we present both studies, with lags 1 to 4, and the inverse study (from issues to CS).

In table 6.12, the columns TE lag1 and TE lag2 represent the transfer entropy from the CS density to issues density, with lag 1 and 2, respectively (being the CS from one release and two releases before). We found values with statistical significance in several apps, being the top for lag 1: *max.params(JavaScript)*, followed by *DevelopmentCodeFragment*, and then *ExcessiveMethodLength*, *ExcessiveClassLength*, *UnusedLocalVariable*, *inline.JS*, *embed.CSS*, *inline.CSS*, and *max.depth*. Almost all applications contribute to TE measurements, but *PrestaShop* and *roundcubemail* only have 1 and 2 values with statistical significance. For lag 2 (CS from 2 releases before), it begins to happen a history effect; while some CS decrease the causality effect, others increase: *ExcessiveClassComplexity* and *UnusedFormalParameter*, from server-side, *inline.CSS* from client embed side, and *max.params*, *complexity*, *max.depth* from client JavaScript.

We studied the inverse Transfer Entropy from the issues to the CS time series, and we found some values, but not worth mentioning, except the *UnusedLocalVariable* (server) that can cause an impact from issues to this CS. A possible explanation is some quick patch that leaves unused variables in the code.

This causality study differs from the cor_ts (time series correlations) study at the beginning of this RQ, where we have the absolute numbers of both CS and issues. Client-side CS contribute more to issue density in more apps because it is the first thing the "issue reporters" see. However, some CS from the server-side also show a more outstanding contribution than the average, especially the *development code fragment*, and *excessive size* and *unused code*.

6.4.3.3 Causality relationships between CS groups and issues

Table 6.13: Statistical causality from CS groups density to Issues (10 apps) using Transfer Entropy (lag 1 to 4). Black dots and white dots represent statistical significance of 0.05 and 0.10, respectively, in one application. The columns % denote the weighted amount of information transfer from CS to Issues

CS	TElag1	TElag1%	TElag2	TElag1%	TElag3	TElag3%	T4lag4	TElag4%
server	•••○	9%	•○○	12%	•	1%	••	5%
client	••••○	15%	••••	19%	••	9%	•••	6%
client_js	••••••	14%	••○○	11%	••○	8%	••	2%

Table 6.13 represents statistical significance and weighted amount of information transfer from CS groups to issues. Black dots and white dots represent statistical significance of 0.05 and 0.10, respectively, for one application of the ten studied app issues. JavaScript CS in the group statistically cause more issues (in 60% of the applications), followed by client-side embed CS (half of the apps) and server-side CS(40% of apps). In lag2, the client embed CS still have an impact, while the others start to decrease.

The weighted average of information transfer measured with TE in lag1 from server CS to issues is around 9%. From client embed client CS we find higher values of 15%, and from JavaScript CS, we find 14%. In lag2, the server, client, and JavaScript values are 12%, 19%

and 11%. The weighted average is calculated by getting the percentage of the apps with statistically significant TE and then multiplying it by the average TE found for apps with statistical significance.

Summary for RQ3: Many individual CS from seven apps correlate with issues, while this number is low in three apps. We find statistical causality from server- and client-side individual CS (both embed and JS) to issues (top CS: *"(Ex.)max.params"* in JS). Grouped CS show statistical causality on issues; the highest values are from client CS.

6.4.4 RQ4 - Impact of CS (server and client) on the faults/reported bugs of a web app

Next, we show the studies on the impact of individual CS on faults/reported bugs in a web application. Again, we used only ten apps for this study because 2 of the apps did not have enough data in the issues tracker.

6.4.4.1 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 6.14: 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	+++

CS	#correl.
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	++++

Table 6.14 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 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.

6.4.4.2 Causality relationships between CS and bugs

To uncover causality relationships between CS and bugs, we employed the same methods again as with the issues: Linear regression, Granger Causality, and Transfer Entropy. Again, as with issues, we used CS density.

Table 6.15: 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). Black dots and white dots represent statistical significance of 0.05 and 0.10, respectively, in one application

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	●●	●●○	●	●●●○○	●●

Table 6.15 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.

6.4.4.3 Causality relationships between CS groups and bugs

Table 6.16: Statistical causality from CS density grouped to Bugs (10 apps) using Transfer Entropy (lag 1 to 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.

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%

Table 6.16 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 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 RQ4: We detected significant correlations between all the CS and bugs in almost all ten applications evaluated except 2. We find statistical causality from almost all the CS to bugs, being the top PHP *(Excessive)CouplingBetweenObjects* and JavaScript *(Excessive)max.params*. The TE from bugs to CS density (inverse TE) is almost non-existent. There is significant TE from CS as groups (Server, Client, and JavaScript) to bugs in Lag 1, decreasing in the subsequent lags.

6.4.5 RQ5 - 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.

6.4.5.1 Causality relationships between individual CS and time to release

Table 6.17 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 (TTR)(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 TTR 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 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.

Table 6.17: 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). Black dots and white dots represent statistical significance of 0.05 and 0.10, respectively, in one application

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	●●●○○	●●●○○	●●●○○	●●●●●●●	●●●●

The column 'TElag1' represents the Transfer entropy from CS to TTR 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.

6.4.5.2 Causality relationships between CS groups and time to release

Table 6.18: Statistical causality from CS density grouped to Time to Release (12 apps) using Transfer Entropy (lag 1 to 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.

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%

Table 6.18 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 RQ5: There are causal relations from individual CS to **'Time to release'** 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* in Lag 1, and it decreases in the other lags, except for lag2:client CS.

6.4.6 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, and perform very well in our tests - no false positives. However, we would prefer 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[53]. 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 [12]. False discovery rate control [15] is substituting the less power Holm and Bonferroni[63, 159]. However, for studies with hundreds or more comparisons, these methods are not recommended [14]. 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.

Threats to conclusion validity concern the relation between the treatment and the outcome. One can argue that CS are often considered by absolute number or normalized by *LOC*. However, our experiments have shown that the normalization of CS by *LOC* gives a better understanding of the effect of the CS on outcome variables. 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.

6.5 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

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

(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), 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.

We chose to study monolithic web apps (the great majority of web apps) because they have server-side and client-side code in the same codebase and sometimes in the same file. This makes the server code interpreted in the web server and the client code interpreted 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 RQ5 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 [114]; causality analysis also revealed that design smells cause architecture smells [145], so it is not comparable. Lastly, entropy was used to predict CS from existing CS [66], a different approach than ours.

Follows the discussion of the findings, and their insights to the community.

6.5.1 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[133].

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 [131], 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.

Regarding JavaScript CS density, the central tendency is to increase. In the 2010s, computers became very powerful, and some of the functionality processed on the server moved to JavaScript code processed in the users' computer as client-side code (JavaScript). This transition from server-side to client-side operations, particularly from PHP to JavaScript, was marked by significant peaks in the evolution of JavaScript individual CS. The JavaScript evolution peaks occur when moving this functionality from server-side to client-side code (from PHP to JavaScript). Over time, as developers became more familiar with writing high-quality client-side code, the peaks in the evolution reduced. This decline can be attributed to an evolutionary learning process whereby an increase in familiarity with the language and its best practices reduced the peak occurrence of code smells. However, this process of moving functionality from server PHP to JavaScript is still ongoing, and this can explain why the density of code smells still increases in general. We interviewed developers with more than ten years of experience in this explanation.

Studies in JavaScript CS are not comparable because they do not target web apps specifically (primarily libraries) and are not evolution studies [140].

6.5.2 RQ2 - Relationship between server- and client-side Code Smell evolution

Depending on the monolithic web application, 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.

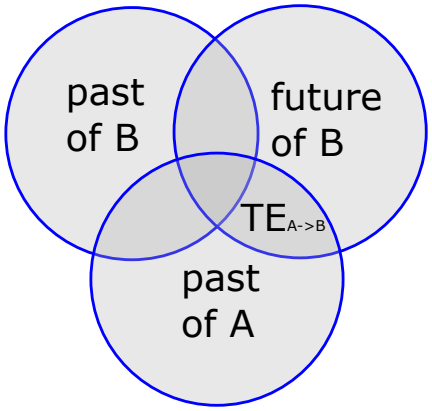


Figure 6.9: Venn diagram showing the transfer entropy of Timeseries A to Timeseries B

Figure 6.9 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).

Because in monolith applications, the code is in just one codebase (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. 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.

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 (we consulted web development specialists with more than ten years of experience in monolithic web apps about this order).

6.5.3 RQ3 - Impact of server- and client-side CS evolution on web app' reported issues

As mentioned before, CS alone are not necessarily responsible for increasing issue numbers. However, we found they play a contributing role in the occurrence of issues, and although the value is very high in percentage, it is not negligible. Furthermore, we found that all groups of CS play a causality role.

CS from the client-side show higher relations / Transfer entropy with the issues, which makes sense because these issues are the first to be spotted (the code is available readily in the browser by mouse right-click, then view source) and reported in an issues tracker when installing a web app. Remembering that issues can also contain problems other than bugs (new functionalities requests, other maintenance tasks), we do not expect high levels of correlation and TE here. However, issues are a good measure of potential work to be done by a team (backlog), and the CS evolution impacts this.

6.5.4 RQ4 - Relation between CS and Bugs

As with issues, the bugs are not caused only by CS. However, they contribute to their appearance, as shown by our measurements. The correlations number is even greater than with "issues," which makes sense (because issues can also contain new functionality requests, among others).

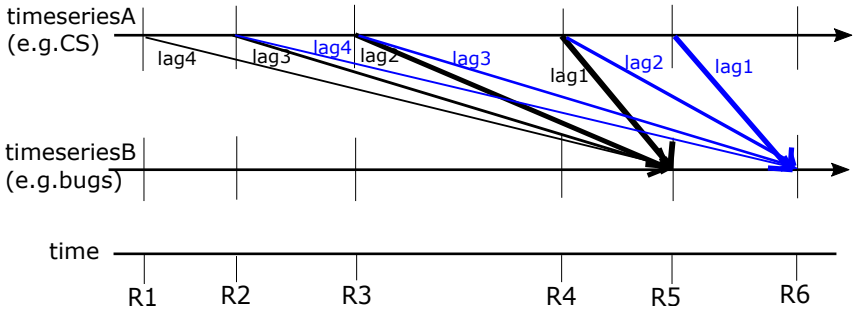


Figure 6.10: Previous releases in CS timeseries impact next releases on bugs timeseries

Figure 6.10 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 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.

Cross-relational studies exist that compare server-side CS with bugs; they mark files smelly and not smelly (if they have one CS during all the history) and find the smelly classes have 1.7 more bugs[17]. This is an entirely different approach, and they include frameworks and third-party libraries in evaluating the apps. However, the results are similar to cross-correlation studies in Java language[80, 116].

In another study, [117] study if the inclusion of CS in a bug prediction model[153], improves the accuracy and they find that the results improve the accuracy by at least 13%, showing that their impact exists, but it is not so high, as expected. This is more in line with our results; using a longitudinal study and measuring causal inference, we found that CS impact bugs with values that can be higher than 15% with CS from the previous release, or around 10% when considering two releases before, decreasing after that.

6.5.5 RQ5 - 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.

6.5.6 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.

To the best of our knowledge, there is a void 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 be filled with 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 [133]. 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.

6.5.7 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, 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 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.

6.5.8 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.

6.6 Chapter 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. 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 also found that individual client-side CS contribute more to issues' density evolution, but CS from the server-side code also impact. Furthermore, 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, even more than the issues. 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 found statistical evidence of causal inference using the various methods between CS and time to release for almost all the individual CS.

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); from CS to bugs, the values 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 [117]. 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.

Data Availability

For replication purposes, we provide the collected dataset, available at "https://github.com/studywebcs/data," where we also have all the values not shown in the article due to lack of space.

Part IV.

Conclusion

This part concludes this thesis.

Chapter 7

Conclusion and Future Work

Contents

This chapter summarizes the main contributions of this work, draws opportunities for further research and concludes the thesis.

7.1 Conclusion

7.1.1 Introduction

Software application development is mainly for three platforms, desktop, web, and mobile. Most Software Engineering concepts and studies are proven and tried on the desktop area, mainly with the Java language. While some are valid for the other platforms, others are not. The studies of mobile applications for Android, built in the Java language, are close to the desktop studies because the platform changes (from a PC to a smartphone) but the code is still processed in one device.

Enter web development, where the code is processed in two places, the web server and the web client (browser). In some monolithic applications, in part of the files, the server- and client-side code are intertwined in the same file, making the analysis even more difficult than in applications that use one platform only. As mentioned before, web applications can move to other architectures, such as distributed (FE/BE) or even microservices architecture, but trading the separation of concerns with higher resources and longer time to develop. Thus, most, if not all, of the applications start in a monolithic application architecture (some of them even include web services). Moreover, most of them never move to other architecture.

The most important aspects one should study in Software Engineering are the aspects the developer can effectively change by modifying the code or the development process. One such aspect is code smells, which have been studied in recent years but primarily for applications built in Java. However, instead of replicating the studies of Java to Web, we wanted to study the evolution of code smells in web applications to uncover the impact on quality evolution.

We performed a systematic literature review to characterize evolution studies in web apps. Then, using a survey, we characterized developers and their knowledge of code smells. Next, we presented an initial web code smells catalog to be extended via an online collaborative platform.

We used client and server-side code smells to study them in monolithic web applications. However, because applications have a higher percentage of server-side code (see Chapter 6, sample table), we started by studying the server-side. We studied the CS evolution, the survival/lifespan, and sudden changes in their density. We also studied possible causes for the introduction of code smells.

After, we studied all the code (server- and client-side). We studied the evolution of the individual CS, CS as a group, and possible relations/impact between them. On monolithic web applications, even if the different code gets processed at different times, the mixture of code can cause problems, which we tried to find. In these applications, the server- and client-side code sometimes reside in the same inter-wined file (the file gets processed twice).

Finally, the principal set of studies was the impact studies on maintainability, and delays supposedly caused by code smells in the evolution of web apps.

During the studies, and apart from the results themselves, several data sets and tools were produced. The following sections will synthesize the results obtained and referred data sets and tools.

7.1.2 Synthesis

Chapter 1 introduced the scope, research problems, and our proposal to mitigate them as global research questions, and the expected contributions, at the beginning of the investigation.

Chapter 2 presented a systematic literature review to characterize the state-of-the-art on web application software evolution studies. The studies fell into seven research areas, and we found the most used dependent and independent variables studied. Most web evolution studies used time series, followed by releases, and a few used commits. We concluded that, compared with evolution studies in desktop applications, for example, Java, more work needs to be done, especially in Code Smells, Technical Dept, and evolution of persistence (databases and more). Since the web development server-side can use different languages, there is an opportunity to replicate several of these studies in other languages in the web field.

Chapter 3 presented a survey to characterize web developers from the industry. The survey disclosed respondents' education and experience, average project duration and team constitutions, server- and client-side languages expertise and the tools used. The developers in the survey have some knowledge of code smells for the web, but some do not apply this knowledge. Developers referred to some situations they consider to be web code smells. We concluded that web smells awareness is just in its infancy. More work has to be done, both by investigators, practitioners, and in academia curricular units syllabus.

Chapter 4 introduced a web smells catalog covering server- and client-side code smells. A collaborative web smells platform to propose, vote, and comment on web code smells was also introduced, along with its operation. Finally, we showed a reduced catalog of web code smells used in the studies performed in chapters 5 and 6. The following two chapters (5 and 6) describe the evolution studies we performed with web code smells.

Chapter 5 presents the studies on the evolution and distribution of web code smells, their survival in the evolution of the software, and the sudden variations in their density. We analyzed the evolution of 18 CS in 12 PHP web applications and compared it with changes in the app and team size to investigate the possible causes of code smell evolution. We characterized the distribution of CS and used survival analysis techniques to study CS' lifespan. We specialized the survival studies into localized (specific location) and scattered CS (spanning multiple classes/methods) categories. We further split the observations for each web app into two consecutive time frames. As for the CS evolution anomalies, we standardized their detection criteria. We concluded that an effort should be made to change the CS density trend to decrease; team constitution is related to the web CS evolution: CS stay a long time in code; the removal rate is low and did not change substantially in recent years; we described a normalized technique for detecting anomalies in specific releases during the evolution of web apps, that can be used by project managers. A description of the main findings is in the next section.

Chapter 6 presents an extended set of studies on the evolution and inner relationship of CS in web apps on the server- and client-sides, and their impact of CS (from server and client-sides) on maintainability (web app issues and bugs) and app time-to-release.

We collected and analyzed 18 server-side, and 12 client-side code smells (aka web smells) from 12 typical PHP web apps, summing 811 releases. Additionally, we collected metrics, maintenance issues, reported bugs, and release dates. Finally, 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 releases (lag 1 to 4).

We found the primary trends in evolution for each CS and each CS group. We studied correlations and statistical causality between groups of code smells (server and client). We measured the latter in three forms: linear models and Granger-causality, measuring the linear relations; and Transfer entropy, which also measures the non-linear relations and can quantify the explainable values from one time series to the other. We also studied the impact of web code smells on issues, bugs, and time-to-release, individually by code smell and as a group in three groups (server, client embed, and client JavaScript). There is evidence of statistical inference between client- and server-side code smells and from the code smells to web applications' issues, bugs, and time to release that requires further confirmation. In the next section, we present the main findings.

7.2 Results/Findings

C2-RQ1 - What are the software evolution studies with web software? – The research areas are Laws of Software Evolution, Maintenance and Bugs, Development techniques, Server, Security, CS and TD, and Includes. The most studied area in web application evolution is security related.

C2-RQ2 - What causes problems in web software evolution quality? – We aimed to discover the independent and dependent variables in the evolution studies, and figure 2.7 presents these variables as a summary. The factors that influence evolution are the independent variables, while the attributes that describe evolution are the dependent variables. We found 13 different factors and 13 different attributes.

C2-RQ3 - How can we deal with unevenly time-spaced software development data? – We found that none of the studies investigates the irregular nature of the releases (the interval between releases is not regular). However, almost half of the studies use regular time series, others use releases/versions, and a tenth use commits.

The SLR (chapter 2) also revealed some secondary questions results.

C3-P1 - Education and experience - In the performed survey of web development and code smells, half of the respondents have a bachelor's or licentiate degree, and one-third have a master's degree. Most of the experience claimed is between one and eight years. Most are professionals within a company, and more than half are full-stack developers.

C3-P2 -Development - Half of the developers in the survey develop proprietary code, and most use frameworks in development. The project duration of more than half is longer than six months, 20% is 3 to 6 months, and the rest is less than three months. Half of the developers work in teams of 6 to 10 people, and 30% in teams of 3 to 5.

C3-P3 -Languages and Tools - Programming language experience of web developers: on the server-side, Java, Python, JavaScript(nodejs), C#, PHP; on the client side, HTML, CSS, and JavaScript (no way around these 3 languages). Most developers use Software Engineering tools to aid in the development.

C3-P4 -Code Smells - Most developers are familiar with code smells, but some do not apply them. Some do not understand the difference between code smells and adherence to

code standards. On the other hand, most developers think code smells are very important or extremely important, and some developers can motivate the answers with problems that code smells can cause. A group of developers could identify examples or situations of code smells on the web. In conclusion, developers are aware of code smells, but some do not apply the refactoring, probably because of lack of time.

C5-RQ1 – How to characterize the evolution of CS? - In chapter 5, we studied the evolution of CS, both in absolute number and density, to unveil the trends and patterns of CS evolution. We found that the absolute number of code smells increases according to the application size. However, in web apps, the evolution trend of server-side code smell density is mainly stable.

C5-RQ1a – What are the possible causes for CS evolution? Looking at team metrics, we found that the evolution of the absolute number of code smells correlated to the code size. Likewise, the evolution of the density of code smells correlated to *number of developers* and *number of new developers* in the release.

C5-RQ2 – What is the distribution and survival/lifespan of CS? - We studied the absolute and relative distribution (by CS and app), the survival time of CS (the time from CS introduction in the code until their removal) and the removal percentage of CS. We found that CS live an average of about 37% of the life of the applications; depending on how we calculate, the survival time of all CS in all apps is between 3.5 years and 3.8 years; The CS that live more days in the apps studied are *UnusedFormalParameter*, *ExcessiveMethodLength*, *ExcessiveClassComplexity*, *TooManyPublicMethods*; On average, only 61% of the server code smells in web apps are removed.

C5-RQ3 – Is the survival of localized CS the same as scattered CS? - We compared the lifespan and survival curves for localized CS (CS that are solely in a specific location) and scattered CS (CS that span over multiple classes or files). For 2/3 of the applications, localized and scattered CS survival is different. For five applications, localized CS live less than the scattered CS; for three applications, the contrary happens. Four applications have no difference in CS survival by scope. All applications remove localized CS, while half of the applications remove 30% or less scattered CS.

C5-RQ4 – Does the survival of CS vary over time? - We divided each application into two consecutive equal time frames and assessed if survival time (lifespan) is the same across these time frames. For almost all the applications, except two, the CS survival is different in the first and second half. For 8 of the ten apps with different CS survival times, the survival time of CS is shorter in the first half of the apps' life, while for two apps, the survival of CS is shorter in the second half. All the applications, excluding one, introduce more CS and remove more CS in absolute number and percentage in the first half of their lives.

C5-RQ5 – How to detect anomalous situations in CS evolution? - We present a method to detect anomalies in CS evolution. Before publishing the release live, this detection method can be put to work in a test automation server. We detected 7 anomalies/sudden changes in the density of CS in five of the studied applications.

C6-RQ1 - How to characterize the evolution of CS in web apps on the server and client sides? - In chapter6, we studied if the evolution of the different CS on the server-side is the same as on the client-side (embed and JavaScript), both in absolute numbers and density; if the

evolution of the CS belonging to the same group is the same; and what is the evolution of the CS as a group (server side-CS, client-side embed CS, and client-side JavaScript CS). We found that most of the Code Smells on the same side/group have the same tendency, except on the server-side. We identified the CS that correlates more between them. The primary trend, for most applications, in server-side CS is slowly decreasing. For client-side embed CS density, the central tendency is to decrease. However, for JavaScript CS density, the central tendency is to increase.

C6-RQ2 - What is the relationship between server- and client-side Code Smell evolution - We questioned if groups of CS (server-side, client-side embed, and client-side JavaScript CS) will evolve in the same way 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. **Correlation** In the same release - On half of the applications, there is high time-series correlation (cor_ts) between CS server-side CS and client-side CS; in a quarter of the applications, high time-series correlation (cor_ts) between embed client CS and JavaScript client CS.
Causal inference From past releases - Lag1 (previous release): more significant TE from client-side to server-side and client_JavaScript to server-side, and next client to client_JavaScript; Lag 2: TE from client to client_JavaScript and Server-side CS are the most significant.

C6-RQ3 - What is the impact of server- and client-side CS evolution on web app' reported issues? - We tested if CS hinders the number of issues and which code smells affect the number of issues evolution in the same release or with CS from previous releases (up to 4 releases behind). We only used ten applications due to two apps' lack of issues reports. We found many individual CS from seven apps correlate with issues, while this number is low in three apps. We find statistical causality from server- and client-side individual CS (both embed and JS) to issues (top CS: *"(Ex.)max.params"* in JS). **Grouped CS** show statistical causality on issues; the highest values are from client-side CS. With TE, we could quantify the impact: from CS groups to bugs is 9%(server), 14%(client JS), and 15%(client embed) on lag 1.

C6-RQ4 - What is the impact of CS evolution on a web app's reported faults (bugs) - We wanted 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 intensity change causes changes in the intensity of the bugs reported in the evolution of the web app. Again, we only used ten applications. We detected significant correlations (cor_ts time-series correlations) between all the CS and bugs in 8 of the ten applications evaluated. We find statistical causality from almost all the CS to bugs, being the top PHP *(Excessive)CouplingBetweenObjects* and JavaScript *(Excessive)max.params*. The TE from bugs to CS density (inverse TE) is almost non-existent. There is significant TE from CS as groups (Server, Client, and JavaScript) to bugs in Lag 1, decreasing in the subsequent lags, showing that CS as groups impact bugs and their evolution can explain in part the evolution of bugs. With TE, we could quantify the impact: from CS groups to bugs is 9%(client embed), 17%(client JS), and 18%(server) on lag 1.

C6-RQ5 - What is the impact of CS evolution on "time to release" in a web app? - We wanted to 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. The findings show there are causal relations from individual CS to **'Time to release'** 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* in Lag 1, and it decreases in the other lags, except for lag2:client CS. In summary, the extra delays in *time-to-release* of a web application can, in part, be explained with the CS time-series, and Transfer Entropy analysis gives this percentage: from CS groups to TTR is 10%(client embed), 14%(server) and 17%(client JS) on lag 1.

To measure the statistical causality in the last three questions, we used linear models (regression), Granger causality (also linear regressions but with lags), and Transfer entropy (that can measure if there is causality and works with lags and nonlinear relations), but we only show the numbers for TE. The other numbers can be found in the appendixes, but they are lower than for TE. The individual CS values are also on the appendixes or the replication kit.

As the main conclusion of chapter 6, CS contribute globally to the quality of web applications, but this contribution is low. Depending on the outcome variable (issues, bugs, time-to-release), the contribution quantity is between 10% and 20%. As a principle, developers should avoid code smells, but they are not the only cause of these outcomes. They are, however, something that developers can change or avoid in the first place.

7.2.1 Global questions

This section maps the thesis global questions to their answers.

(GRQ1) Which are the most relevant web smells? - is answered by chapter Web code smells, and C6-1 to 5.

(GRQ2) Can web smells location be detected automatically? - is answered by the tools used and proposed, and the data sets generated. This detection was also the base of chapters 5 and 6, and their RQs.

(GRQ3a) What is the behavior of code smells in the evolution of web systems? - is answered by C5-Rq1 to RQ5 and C6-R1

(GRQ3) Is web systems quality evolution, in terms of maintainability and reliability, influenced by the presence of web smells? - is answered by C6RQ3, C6RQ4 and C6RQ5

(GRQ4) How can we deal with unevenly time-spaced software development data? - is answered by C6RQ1 to RQ5 .

7.3 Main Contributions

This section presents a summary of the main contributions delivered with the development of this thesis:

C1-Web smells catalog and platform - Catalogue of web code smells - we presented a catalog of web code smells spanning the server- and client-side code. We elaborated an online platform where the community can grow this catalog by proposing, voting, or commenting on the existing smells. We explored a subset of this catalog in evolution studies and could validate that all the code smells in use impact issues, bugs and delays to release (utilizing correlation and statistical causality).

C2-Datasets - We provide a longitudinal dataset of web code smells to the evolution of applications, spanning 811 versions of 12 typical web apps. This dataset comprises the server-side code smells, client-side embed code smells, and client-side JavaScript code smells https://github.com/studywebcs/data. - We provide a dataset of the server code smells from a survival perspective, where each smell has an introduction date and a removal date along with the other parameters (file, class, method, and metrics) https://github.com/studydatacs/servercs. All the code smells datasets were collected removing the folders of third-party libraries, so the CS and metrics only evaluate the target applications.

C3-Tools

- We developed a tool, *eextractor*, to detect embed web code smells https://github.com/studywebcs/eextractor.

- We built all the scripts to automatize the detection of code smells recursively and automatically from all applications, using PHPMD(for server-side CS) and ESLint (for JavaScript client-side CS), inserting them in a database, and exporting to CSV https://github.com/studywebcs/data and https://github.com/studydatacs/servercs.

C3-Studies The following studies were produced during this thesis:

a) on the density and evolution of server-side code smells
b) of the possible causes for the evolution of code smells
c) on survival of server-side code smells - this study is done as a whole and further specialized by scope (localized and scattered CS) and time (two periods).
d) on the sudden variation in the evolution of the density of server-side code smells - method and results
e) of the evolution trend and correlation of individual CS on the same group/side: 18 server-side CS among them; 6 client-side embed CS among them; 6 client-side JavaScript among them
f) of the evolution trend of CS groups: server-side, client-side embed, and client-side JavaScript CS
g) on correlation between CS groups
h) on causal inference between CS groups - Linear Regression, G-Causality (lag1-4), Transfer entropy(TE) (lag1-4)
i) on time-series correlation (cor_ts) between CS and issues
j) on causal inference between 18 individual CS and issues - Linear Regression, G-Causality (lag1-4), TE (lag1-4)
k) on causal inference between 3 CS groups and issues - TE (lag1-4)
l) on time-series correlation (cor_ts) between CS and bugs
m) on causal inference between 18 individual CS and bugs - Linear Regression, G-Causality (lag1-4), TE (lag1-4)
n) on causal inference between 3 CS groups and bugs - TE (lag1-4)
o) on causal inference between 18 individual CS and time-to-release - Linear Regression, G-Causality (lag1-4), TE(lag1-4)
p) on causal inference between 3 CS groups and time-to-release - Transfer entropy (lag1-4)

C5-Systematic Literature review (SLR) of studies on web systems/apps evolution. This study was required to understand the state-of-the-art web systems or app quality evolution and

identify research gaps/niches.

C6 Survey on web developers and their knowledge of code smells - Survey on web developers, teams, projects, and their knowledge of code smells.

7.3.1 Main conclusions

In this section, and for reference, all the studies' main conclusions are presented.

1. In the SLR about web systems evolution studies, we found that more studies should be performed about variables that developers can effectively change, such as code smells and more.
2. From the survey, developers know about code smells on the web, but most do not consider them. Instead of avoiding them, other developers use tools to verify before release.
3. CS stay a long time in code. The removal rate is low and did not change substantially in recent years. An effort should be made to avoid this bad behavior and change the CS density trend to decrease.
4. We proposed a method to avoid peaks in the code smell density.
5. Most significant statistical causality between CS groups: CS client-side to CS server-side; after client-embed to client-JavaScript;
6. Individual client-side CS density contribute more to reported issues number evolution, followed by server-side CS.
7. Almost all CS contribute to detected bugs number evolution, especially in lag1;
8. We found statistical evidence of causal inference between CS and time to release (TTR) for almost all the individual CS.
9. All CS groups contribute to issues, bugs, and TTR; JavaScript CS can cause more issues, (server)PHP and JavaScript CS can cause more bugs, and the TTR (delays) is more impacted by the density of client embed CS. In conclusion, there is evidence of statistical inference between client- and server-side code smells and from the code smells to web applications' issues, bugs, and TTR.

Implications to practitioners - There are web smells on both sides (server- and client-side) of typical web apps. We showed that they impact issues, bugs, and time-to-release and stay in code for long periods. Therefore, when writing the code, an effort should be made to avoid these smells. When this is not possible, using web code smell detectors before the releases is encouraged. We also showed a method to avoid peaks in code smell density that could be used by developers and by project managers.

Client-side smells can cause server smells. Therefore, before inserting server code (in typical web apps where the code is tangled in files), it is essential to reduce the client-side code smells.

Implications to investigators - in chapters 5 and 6, we already showed implications to investigators, namely the invitations to replicate our studies, make studies with the web smells catalog, and use our tools (e.g., the client CS detector). We also call for collaboration in the web smells platform. In the next section, we identify some research opportunities.

Implications to educators Present topics in CS taught in universities cover desktop code smells. Educators should add web code smells topics to either Software Engineering courses or

advanced web development courses.

7.4 Research Opportunities

The most apparent research opportunity is the extension of the catalog of code smells. In chapter 3 and on the platform itself, we already proposed new web code smells, but we welcome confirmation and studies by investigators and developers.

Another research opportunity is to perform studies like those present here, but in distributed apps (FE/BE) and apps with a microservice architecture. For these applications, only the front-end or front-end parts (if we speak about micro-frontends) are web apps. However, if an application is transformed into a set of small apps, the studies must comply and become a set of micro studies. In other words, the study will have to be multipart.

The application of some of the statistics used in the thesis studies is innovative in the context of longitudinal studies of software quality evolution. For instance, we used specific time series correlations that can be used between time series of different granularity and even irregular series (series without regular sample rate). Using a panel set to infer causal inference, with linear methods (linear regression and Granger-causality for previous releases) and Transfer entropy for extra linear relations (and also for previous releases) gives a power of concordance that should be used in desktop or mobile studies also. We are not aware that Transfer Entropy was ever used in similar studies, and it can infer if a time series can be explained with the help of the other, like Granger causality, but it can quantify the information transfer between them, and also the nonlinear causalities. Even if not using a panel of concordance causal methods, studies with TE in other areas are a solid research opportunity.

As demonstrated by the studies, CS impact the quality of web apps, but their impact is moderate. Therefore, another research opportunity is building a complete model for quality web apps. This will be useful for two reasons. Firstly, it will help assess and predict the quality of a web app. Secondly, it will help quantify the causes that impact the quality of a web app and choose what we can effectively change or avoid. Work has been done in models of Java's desktop apps, with CS and metrics (product and process), but not in an evolutionary way. In the web area, more factors are to consider (web server, database server, network connection, and code that runs on two platforms, the server-side and the client/browser side), so the challenge is even more significant.

Bibliography

[1] H. Akoglu. "User's guide to correlation coefficients." In: *Turkish journal of emergency medicine* 18.3 (2018), pp. 91–93. DOI: 10.1016/j.tjem.2018.08.001.

[2] T. Amanatidis and A. Chatzigeorgiou. "Studying the evolution of PHP web applications." In: *Information and Software Technology* 72.April (Apr. 2016), pp. 48–67. DOI: 10.1016/j.infsof.2015.11.009.

[3] T. Amanatidis, A. Chatzigeorgiou, and A. Ampatzoglou. "The relation between technical debt and corrective maintenance in PHP web applications." In: *Information and Software Technology* 90 (Oct. 2017), pp. 70–74. DOI: 10.1016/j.infsof.2017.05.004.

[4] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. "Is it a bug or an enhancement? A text-based approach to classify change requests." In: *Proceedings of CASCON'2008 conference*. Centre for Advanced Studies on Collaborative Research (CASCON). 2008, pp. 304–318. DOI: 10.1145/1463788.1463819.

[5] D. Atkins, D. Best, P. A. Briss, M. Eccles, Y. Falck-Ytter, S. Flottorp, G. H. Guyatt, R. T. Harbour, M. C. Haugh, D. Henry, et al. "Grading quality of evidence and strength of recommendations." In: *BMJ* (2004). DOI: 10.1136/bmj.328.7454.1490.

[6] J. Atwood. *Code Smells on Code Horror Blog*. https://blog.codinghorror.com/code-smells/. Accessed: 2015-07-30. 2015.

[7] K. Ayari, P. Meshkinfam, G. Antoniol, and M. Di Penta. "Threats on building models from CVS and Bugzilla repositories." In: *Proceedings of CASCON'2007 conference*. Centre for Advanced Studies on Collaborative Research (CASCON). 2007, pp. 215–228. DOI: 10.1145/1321211.1321234.

[8] M. T. Bahadori and Y. Liu. "Granger Causality Analysis in Irregular Time Series." In: *Proceedings of the 2012 SIAM International Conference on Data Mining*. SIAM. 2012, pp. 660–671. DOI: 10.1137/1.9781611972825.57.

[9] T. Bakota, R. Ferenc, and T. Gyimothy. "Clone smells in software evolution." In: *2007 IEEE International Conference on Software Maintenance*. IEEE. 2007, pp. 24–33. DOI: 10.1109/icsm.2007.4362615.

[10] D. Bán and R. Ferenc. "Recognizing antipatterns and analyzing their effects on software maintainability." In: *International Conference on Computational Science and Its Applications*. Springer. 2014, pp. 337–352. DOI: 10.1007/978-3-319-09156-3_25.

[11] L. Barnett, A. B. Barrett, and A. K. Seth. "Granger causality and transfer entropy are equivalent for Gaussian variables." In: *Physical review letters* 103.23 (2009), p. 238701. DOI: 10.1103/physrevlett.103.238701.

[12] K. Beck, M. Fowler, and G. Beck. "Bad smells in code." In: *Refactoring: Improving the design of existing code*. Addison-Wesley Professional, 1999.

[13] S. Behrendt, T. Dimpfl, F. J. Peter, and D. J. Zimmermann. "RTransferEntropy — Quantifying information flow between different time series using effective transfer entropy." In: *SoftwareX* 10 (2019), pp. 1–9. DOI: 10.1016/j.softx.2019.100265.

[14] R. Bender and S. Lange. "Adjusting for multiple testing—when and how?" In: *Journal of clinical epidemiology* 54.4 (2001), pp. 343–349. DOI: 10.1016/s0895-4356(00)00314-0.

[15] Y. Benjamini and Y. Hochberg. "Controlling the false discovery rate: a practical and powerful approach to multiple testing." In: *Journal of the Royal statistical society: series B (Methodological)* 57.1 (1995), pp. 289–300. DOI: 10.1111/j.2517-6161.1995.tb02031.x.

[16] N. Bessghaier, A. Ouni, and M. W. Mkaouer. "On the Diffusion and Impact of Code Smells in Web Applications." In: *Lecture Notes in Computer Science*. Vol. 12409 LNCS. Springer Verlag. Springer, June 2020, pp. 67–84. DOI: 10.1007/978-3-030-59592-0_5.

[17] N. Bessghaier, A. Ouni, and M. W. Mkaouer. "A longitudinal exploratory study on code smells in server side web applications." In: *Software Quality Journal* 29.4 (2021), pp. 901–941. DOI: 10.1007/s11219-021-09567-w.

[18] J. M. Bieman and B.-K. Kang. "Cohesion and reuse in an object-oriented system." In: *ACM SIGSOFT Software Engineering Notes* 20.SI (Aug. 1995), pp. 259–262. DOI: 10.1145/223427.211856.

[19] H. Boomsma, B. V. Hostnet, and H. G. Gross. "Dead code elimination for web systems written in PHP: Lessons learned from an industry case." In: *IEEE International Conference on Software Maintenance, ICSM* (2012), pp. 511–515. DOI: 10.1109/ICSM.2012.6405314.

[20] H. P. Breivold, M. A. Chauhan, and M. A. Babar. "A systematic review of studies of open source software evolution." In: *2010 Asia Pacific Software Engineering Conference*. IEEE. 2010, pp. 356–365.

[21] S. Bryton, F. Brito e Abreu, and M. Monteiro. "Reducing subjectivity in code smells detection: Experimenting with the long method." In: *7th International Conference on the Quality of Information and Communications Technology (QUATIC'2010)*. IEEE. IEEE, 2010, pp. 337–342. DOI: 10.1109/QUATIC.2010.60.

[22] R. D. R. Cai. "Language Features for Software Evolution and Aspect-Oriented Interfaces: An Exploratory Study." In: *Transactions on Aspect-Oriented Software Development X* (2013), pp. 148–183. DOI: 10.1007/978-3-642-36964-3_5.

[23] A. S. Cairo, G. d. F. Carneiro, and M. P. Monteiro. "The impact of code smells on software bugs: A systematic literature review." In: *Information* 9.11 (2018), p. 273. DOI: 10.3390/info9110273.

[24] K. K. Chahal and M. Saini. "Open source software evolution: a systematic literature review (Part 1)." In: *International Journal of Open Source Software and Processes (IJOSSP)* 7.1 (2016), pp. 1–27. DOI: 10.4018/ijossp.2016010101.

[25] K. K. Chahal and M. Saini. "Open Source Software Evolution: A Systematic Literature Review (Part 2)." In: *International Journal of Open Source Software and Processes (IJOSSP)* 7.1 (2016), pp. 28–48. DOI: 10.4018/ijossp.2016010102.

[26] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan. "Types of software evolution and software maintenance." In: *Journal of software maintenance and evolution: Research and Practice* 13.1 (2001), pp. 3–30. DOI: 10.1002/smr.220.

[27] K. Chaturvedi, P. Kapur, S. Anand, and V. Singh. "Predicting the complexity of code changes using entropy based measures." In: *International Journal of System Assurance Engineering and Management* 5.2 (2014), pp. 155–164. DOI: 10.1007/s13198-014-0226-5.

[28] A. Chatzigeorgiou and A. Manakos. "Investigating the evolution of bad smells in object-oriented code." In: *2010 Seventh International Conference on the Quality of Information and Communications Technology*. IEEE. 2010, pp. 106–115. DOI: 10.1109/quatic.2010.16.

[29] A. Chatzigeorgiou and A. Manakos. "Investigating the evolution of bad smells in object-oriented code." In: *Proceedings of the 7th International Conference on the Quality of Information and Communications Technology (QUATIC'2010)*. 2010, pp. 106–115. DOI: 10.1109/QUATIC.2010.16.

[30] A. Chatzigeorgiou and A. Manakos. "Investigating the evolution of code smells in object-oriented systems." In: *Innovations in Systems and Software Engineering* 10.1 (2014), pp. 3–18. DOI: 10.1007/s11334-013-0205-z.

[31] A. Chatzimparmpas, S. Bibi, I. Zozas, and A. Kerren. "Analyzing the Evolution of Javascript Applications." In: *ENASE*. 2019, pp. 359–366. DOI: 10.5220/0007727603590366.

[32] L. Christophe, R. Stevens, C. De Roover, and W. De Meuter. "Prevalence and maintenance of automated functional tests for web applications." In: *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014* (2014), pp. 141–150. DOI: 10.1109/ICSME.2014.36.

[33] T. G. Clark, M. J. Bradburn, S. B. Love, and D. G. Altman. "Survival analysis part I: basic concepts and first analyses." In: *British journal of cancer* 89.2 (2003), pp. 232–238. DOI: 10.1038/sj.bjc.6601118.

[34] J. Cohen. "A coefficient of agreement for nominal scales." In: *Educational and psychological measurement* 20.1 (1960), pp. 37–46. DOI: 10.1177/001316446002000104.

[35] G. Concas, M. Marchesi, G. Destefanis, and R. Tonelli. "An empirical study of software metrics for assessing the phases of an agile project." In: *International Journal of Software Engineering and Knowledge Engineering* 22.04 (2012), pp. 525–548. DOI: 10.1142/s0218194012500131.

[36] R. Correia and E. Adachi. "Detecting Design Violations in Django-based Web Applications." In: *Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse*. ACM. ACM, Sept. 2019, pp. 33–42. DOI: 10.1145/3357141.3357600.

[37] C. Couto, P. Pires, M. T. Valente, R. S. Bigonha, and N. Anquetil. "Predicting software defects with causality tests." In: *Journal of Systems and Software* 93 (2014), pp. 24–41. DOI: 10.1016/j.jss.2014.01.033.

[38] W. Cunningham. "The WyCash portfolio management system." In: *ACM SIGPLAN OOPS Messenger* 4.2 (1992), pp. 29–30. DOI: 10.1145/157709.157715.

[39] M. D'Ambros, A. Bacchelli, and M. Lanza. "On the impact of design flaws on software defects." In: *2010 10th International Conference on Quality Software*. IEEE. 2010, pp. 23–31. DOI: 10.1109/qsic.2010.58.

[40] A. Decan, T. Mens, and E. Constantinou. "On the evolution of technical lag in the npm package dependency network." In: *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*. Institute of Electrical and Electronics Engineers Inc., Nov. 2018, pp. 404–414. DOI: 10.1109/ICSME.2018.00050. arXiv: 1806.01545.

[41] G. Digkas, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou. "On the temporality of introducing code technical debt." In: *Communications in Computer and Information Science*. Vol. 1266 CCIS. Springer, Sept. 2020, pp. 68–82. DOI: 10.1007/978-3-030-58793-2_6.

[42] G. Digkas, M. Lungu, A. Chatzigeorgiou, and P. Avgeriou. "The evolution of technical debt in the apache ecosystem." In: *Lecture Notes in Computer Science*. Vol. 10475 LNCS. Springer, 2017, pp. 51–66. DOI: 10.1007/978-3-319-65831-5_4.

[43] A. Dionisio, R. Menezes, and D. A. Mendes. "Mutual information: a measure of dependency for nonlinear time series." In: *Physica A: Statistical Mechanics and its Applications* 344.1-2 (2004), pp. 326–329. DOI: 10.1016/j.physa.2004.06.144.

[44] A. Dionisio, R. Menezes, and D. A. Mendes. "An econophysics approach to analyse uncertainty in financial markets: an application to the Portuguese stock market." In: *The European Physical Journal B-Condensed Matter and Complex Systems* 50.1 (2006), pp. 161–164. DOI: 10.1140/epjb/e2006-00113-2.

[45] N. Drouin, M. Badri, and F. Toure. "Metrics and software quality evolution: A case study on open source software." In: *International Journal of Computer Theory and Engineering* 5.3 (2013), pp. 523–527. DOI: 10.7763/ijcte.2013.v5.742.

[46] V. Dusch et al. *PMD - PHP Mess Detector*. https://phpmd.org. Accessed: 2021-04-04. Apr. 2021.

[47] Y. K. Dwivedi, M. D. Williams, A. Mitra, S. Niranjan, and V. Weerakkody. "Understanding advances in web technologies: evolution from web 2.0 to web 3.0." In: *European Conference on Information Systems*. 2011.

[48] T. Dybå and T. Dingsøyr. "Empirical studies of agile software development: A systematic review." In: *Information and software technology* 50.9-10 (2008), pp. 833–859. DOI: 10.1016/j.infsof.2008.01.006.

[49] A. Eckner. "A framework for the analysis of unevenly spaced time series data." In: *Preprint. Available at: http://www. eckner. com/papers/unevenly_spaced_time_series_analysis* (2012), p. 93.

[50] A. Eckner. "Algorithms for unevenly-spaced time series: Moving averages and other rolling operators." In: *Working Paper*. 2012.

[51] T. Edinburgh, S. J. Eglen, and A. Ercole. "Causality indices for bivariate time series data: A comparative review of performance." In: *Chaos* 31.8 (2021), pp. 1–14. DOI: 10.1063/5.0053519.

[52] A. M. Fard and A Mesbah. "JSNOSE: Detecting JavaScript Code Smells." In: *Source Code Analysis and Manipulation (SCAM), 13th International Working Conference on*. IEEE. 2013, pp. 116–125. DOI: 10.1109/SCAM.2013.6648192.

[53] R. J. Feise. "Do multiple outcome measures require p-value adjustment?" In: *BMC medical research methodology* 2.1 (2002), pp. 1–4. DOI: 10.1186/1471-2288-2-8.

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

[55] F. A. Fontana, P. Braione, and M. Zanoni. "Automatic detection of bad smells in code: An experimental assessment." In: *J. Object Technol.* 11.2 (2012), pp. 5–1. DOI: 10.5381/jot.2012.11.2.a5.

[56] F. A. Fontana, V. Lenarduzzi, R. Roveda, and D. Taibi. "Are architectural smells independent from code smells? An empirical study." In: *Journal of Systems and Software* 154 (Aug. 2019), pp. 139–156. DOI: 10.1016/j.jss.2019.04.066. arXiv: 1904.11755.

[57] K. Fowler. *Code smells on sourcemaking.com*. https://sourcemaking.com/refactoring/smells. Accessed: 2015-07-30. 2015.

[58] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999, p. 464.

[59] V. Garousi, A. Coşkunçay, A. Betin-Can, and O. Demirörs. "A survey of software engineering practices in Turkey." In: *Journal of Systems and Software* 108 (2015), pp. 148–177. DOI: 10.1016/j.jss.2015.06.036.

[60] V. Garousi, A. Mesbah, A. Betin-Can, and S. Mirshokraie. "A systematic mapping study of web application testing." In: *Information and Software Technology* 55.8 (2013), pp. 1374–1396. DOI: 10.1016/j.infsof.2013.02.006.

[61] G. Gharachorlu. "Code smells in Cascading Style Sheets: an empirical study and a predictive model." Doctoral dissertation. Vancouver, BC V6T 1Z4, Canada: University of British Columbia, 2014. DOI: 10.14288/1.0167067.

[62] G. K. Gill and C. F. Kemerer. "Cyclomatic complexity density and software maintenance productivity." In: *IEEE transactions on software engineering* 17.12 (1991), pp. 1284–1288. DOI: 10.1109/32.106988.

[63] M. E. Glickman, S. R. Rao, and M. R. Schultz. "False discovery rate control is a recommended alternative to Bonferroni-type adjustments in health studies." In: *Journal of clinical epidemiology* 67.8 (2014), pp. 850–857. DOI: 10.1016/j.jclinepi.2014.03.012.

[64] C. W. Granger. "Investigating causal relations by econometric models and cross-spectral methods." In: *Econometrica: journal of the Econometric Society* (1969), pp. 424–438. DOI: 10.2307/1912791.

[65] C. W. Granger. "Some recent development in a concept of causality." In: *Journal of econometrics* 39.1-2 (1988), pp. 199–211. DOI: 10.1016/0304-4076(88)90045-0.

[66] A. Gupta, B. Suri, V. Kumar, S. Misra, T. Blažauskas, and R. Damaševičius. "Software code smell prediction model using Shannon, Rényi and Tsallis entropies." In: *Entropy* 20.5 (2018), p. 372. DOI: 10.3390/e20050372.

[67] S. Habchi, R. Rouvoy, and N. Moha. "On the survival of android code smells in the wild." In: *Proceedings of the 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft'2019)*. IEEE. IEEE, May 2019, pp. 87–98. DOI: 10.1109/MOBILESoft.2019.00022.

[68] S. Heerah, R. Molinari, S. Guerrier, and A. Marshall-Colon. "Granger-Causal Testing for Irregularly Sampled Time Series with Application to Nitrogen Signaling in Arabidopsis." In: *bioRxiv* (2020). DOI: 10.1101/2020.06.15.152819. eprint: https://www.biorxiv.org/content/early/2020/06/17/2020.06.15.152819.full.pdf.

[69] B. Henderson-Sellers. *Object-oriented metrics | Guide books*. Prentice-Hall, 1995.

[70] S. Herbold, J. Grabowski, and S. Waack. "Calculation and optimization of thresholds for sets of software metrics." In: *Empirical Software Engineering* 16.6 (Dec. 2011), pp. 812–841. DOI: 10.1007/s10664-011-9162-z.

[71] I. Herraiz, D. Rodriguez, G. Robles, and J. M. Gonzalez-Barahona. "The evolution of the laws of software evolution: A discussion based on a systematic literature review." In: *ACM Computing Surveys (CSUR)* 46.2 (2013), pp. 1–28. DOI: 10.1145/2543581.2543595.

[72] N. Hung Viet, N. Hoan Anh, N. Tung Thanh, N. Anh Tuan, and T. N. Nguyen. "Detection of embedded code smells in dynamic web applications." In: *Proceedings of the 27th International Conference on Automated Software Engineering (ASE)*. IEEE/ACM. 2012, pp. 282–285. DOI: 10.1145/2351676.2351724.

[73] T Idé and H Kashima. "Eigenspace-based anomaly detection in computer systems." In: (2004), pp. 440–449. DOI: 10.1145/1014052.1014102.

[74] W. Jank and G. Shmueli. "Functional data analysis in electronic commerce research." In: *Statistical Science* 21.2 (2006), pp. 155–166. DOI: 10.1214/088342306000000132.

[75] D. Johannes, F. Khomh, and G. Antoniol. "A large-scale empirical study of code smells in JavaScript projects." In: *Software Quality Journal* 27.3 (Sept. 2019), pp. 1271–1314. DOI: 10.1007/s11219-019-09442-9.

[76] E. L. Kaplan and P. Meier. "Nonparametric Estimation from Incomplete Observations." In: *Journal of the American Statistical Association* 53.282 (1958), pp. 457–481. DOI: 10.1080/01621459.1958.10501452.

[77] G. Kappel, B. Pröll, S. Reich, and W. Retschitzegger. *Web engineering*. Wiley New York, 2006.

[78] C. F. Kemerer and S. Slaughter. "An empirical approach to studying software evolution." In: *IEEE transactions on software engineering* 25.4 (1999), pp. 493–509. DOI: 10.1109/32.799945.

[79] S. Keshav. "How to read a paper." In: *ACM SIGCOMM Computer Communication Review* 37.3 (2007), pp. 83–84.

[80] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol. "An exploratory study of the impact of antipatterns on class change-and fault-proneness." In: *Empirical Software Engineering* 17.3 (2012), pp. 243–275. DOI: 10.1007/s10664-011-9171-y.

[81] J. Kim, H. Hong, and K.-I. Kim. "Adaptive optimized pattern extracting algorithm for forecasting maximum electrical load duration using random sampling and cumulative slope index." In: *Energies* 11.7 (2018), p. 1723. DOI: 10.3390/en11071723.

[82] B. Kitchenham, S. Charters, et al. "Guidelines for performing systematic literature reviews in software engineering version 2.3." In: *Engineering* 45.4ve (2007), p. 1051.

[83] P. Kyriakakis and A. Chatzigeorgiou. "Maintenance Patterns of Large-Scale PHP Web Applications." In: *Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME'2014)* (2014), pp. 381–390. DOI: 10.1109/ICSME.2014.60.

[84] F. Lanubile and T. Mallardo. "Finding function clones in web applications." In: *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.* IEEE. 2003, pp. 379–386. DOI: 10.1109/csmr.2003.1192447.

[85] M. Lanza and R. Marinescu. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer, 2007.

[86] M.-A. Laverdière and E. Merlo. "Classification and Distribution of RBAC Privilege Protection Changes in Wordpress Evolution." In: *2017 15th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 2017, pp. 349–3495. DOI: 10.1109/pst.2017.00048.

[87] M.-A. Laverdière and E. Merlo. "Detection of protection-impacting changes during software evolution." In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 434–444. DOI: 10.1109/saner.2018.8330230.

[88] M. Lawal, A. B. M. Sultan, and A. O. Shakiru. "Systematic literature review on SQL injection attack." In: *International Journal of Soft Computing* 11.1 (2016), pp. 26–35.

[89] M. M. Lehman. "Laws of software evolution revisited." In: *Lecture Notes in Computer Science*. Vol. 1149. Springer Verlag. Springer, 1996, pp. 108–124. DOI: 10.1007/BFb0017737.

[90] D. Letarte, F. Gauthier, and E. Merlo. "Security Model Evolution of PHP Web Applications." In: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation* (2011), pp. 289–298. DOI: 10.1109/ICST.2011.36.

[91] W. Li and R. Shatnawi. "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution." In: *Journal of systems and software* 80.7 (2007), pp. 1120–1128. DOI: 10.1016/j.jss.2006.10.018.

[92] Y.-F. Li, P. K. Das, and D. L. Dowe. "Two decades of Web application testing—A survey of recent advances." In: *Information Systems* 43 (2014), pp. 20–54. DOI: 10.1016/j.is.2014.02.001.

[93] U. A. Mannan, I. Ahmed, and A. Sarma. "Towards understanding code readability and its impact on design quality." In: *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering*. 2018, pp. 18–21. DOI: 10.1145/3283812.3283820.

[94] M. Mantyla. *A Taxonomy for "Bad Code Smells"*. https://mmantyla.github.io/BadCodeSmellsTaxonomy. Accessed: 2022-10-01. 2015.

[95] M. Mantyla, J. Vanhanen, and C. Lassenius. "A taxonomy and an initial empirical study of bad smells in code." In: *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* IEEE. 2003, pp. 381–384. DOI: 10.1109/icsm.2003.1235447.

[96] M. V. Mantyla and C. Lassenius. "Subjective evaluation of software evolvability using code smells: An empirical study." In: *Empirical Software Engineering* 11.3 (2006), pp. 395–431. DOI: 10.1007/s10664-006-9002-8.

[97] R. Marinescu and C. Marinescu. "Are the clients of flawed classes (also) defect prone?" In: *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*. IEEE. 2011, pp. 65–74. DOI: 10.1109/scam.2011.9.

[98] T. J. Mccabe. "A Complexity Measure." In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837.

[99] E. Mendes. "A systematic review of Web engineering research." In: *2005 International Symposium on Empirical Software Engineering, 2005.* IEEE. 2005, 10–pp. DOI: 10.1109/isese.2005.1541857.

[100] E. Mendes and N. Mosley. *Web engineering*. Springer Science & Business Media, 2006.

[101] T. Mens and S. Demeyer. *Software Evolution*. Ed. by S. T. Mens and Demeyer. Berlin, Heidelberg: Springer, 2008, pp. 1–347. DOI: 10.1007/978-3-540-76440-3.

[102] E. Merlo, D. Letarte, and G. Antoniol. "SQL-Injection security evolution analysis in PHP." In: *Proceedings - 9th IEEE International Symposium on Web Site Evolution, WSE 2007* (2007), pp. 45–49. DOI: 10.1109/WSE.2007.4380243.

[103] A. Mesbah and S. Mirshokraie. "Automated analysis of CSS rules to support style maintenance." In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 408–418. DOI: 10.1109/icse.2012.6227174.

[104] T. Mills. *Applied Time Series Analysis: A Practical Guide to Modeling and Forecasting*. Elsevier Science, 2019.

[105] D. Mitropoulos, P. Louridas, V. Salis, and D. Spinellis. "Time present and time past: analyzing the evolution of JavaScript code in the wild." In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 126–137. DOI: 10.1109/msr.2019.00029.

[106] M. Mudelsee. *Climate Time Series Analysis: Classical Statistical and Bootstrap Methods*. 2nd ed. Atmospheric and Oceanographic Sciences Library. Springer-Verlag, 2014. DOI: 10.1111/jtsa.12002.

[107] A. Murgia, G. Concas, S. Pinna, R. Tonelli, and I. Turnu. "Empirical study of software quality evolution in open source projects using agile practices." In: *Proc. of the 1st International Symposium on Emerging Trends in Software Metrics*. Vol. 11. 2009.

[108] S. Murugesan, Y. Deshpande, S. Hansen, and A. Ginige. "Web engineering: A new discipline for development of web-based systems." In: *Web engineering: Managing diversity and complexity of Web application development* (2001), pp. 3–13. DOI: 10.1007/3-540-45144-7_2.

[109] B. Musa Shuaibu, N. Md Norwawi, M. H. Selamat, and A. Al-Alwani. "Systematic review of web application security development model." In: *Artificial Intelligence Review* 43.2 (2015), pp. 259–276. DOI: 10.1007/s10462-012-9375-6.

[110] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. "You are what you include: Large-scale evaluation of remote JavaScript inclusions." In: *Proceedings of the ACM Conference on Computer and Communications Security*. New York, New York, USA: ACM Press, 2012, pp. 736–747. DOI: 10.1145/2382196.2382274.

[111] G. W. Noblit, R. D. Hare, and R. D. Hare. *Meta-ethnography: Synthesizing qualitative studies*. Vol. 11. Sage, 1988.

[112] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka. "The evolution and impact of code smells: A case study of two open source systems." In: *Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM'2009)*. IEEE. IEEE, 2009, pp. 390–400. DOI: 10.1109/ESEM.2009.5314231.

[113] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøøberg. "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems." In: *Proceedings of the International Conference on Software Maintenance (ICSM'2010)*. IEEE. IEEE, 2010. DOI: 10.1109/ICSM.2010.5609564.

[114] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia. "A large-scale empirical study on the lifecycle of code smell co-occurrences." In: *Information and Software Technology* 99 (2018), pp. 1–10. DOI: 10.1016/j.infsof.2018.02.004.

[115] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, and A. De Lucia. "Do they really smell bad? a study on developers' perception of bad code smells." In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE. 2014, pp. 101–110. DOI: 10.1109/icsme.2014.32.

[116] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia. "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation." In: *Empirical Software Engineering* 23.3 (June 2018), pp. 1188–1221. DOI: 10.1007/s10664-017-9535-z.

[117] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto. "Toward a smell-aware bug prediction model." In: *IEEE Transactions on Software Engineering* 45.2 (2017), pp. 194–218. DOI: 10.1109/tse.2017.2770122.

[118] J. Pereira dos Reis, F. Brito e Abreu, and G. de Figueiredo Carneiro. "Code smells detection 2.0: Crowdsmelling and visualization." In: *2017 12th Iberian Conference on Information Systems and Technologies (CISTI)*. IEEE. 2017, pp. 1–4. DOI: 10.23919/CISTI.2017.7975961.

[119] R. Peters and A. Zaidman. "Evaluating the lifespan of code smells using software repository mining." In: *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*. IEEE. IEEE, 2012, pp. 411–416. DOI: 10.1109/CSMR.2012.79.

[120] PMD. *PMD static source code analyzer*. https://pmd.github.io/. Accessed: 2020-01-03. 2020.

[121] J. M. Polanco-Martinez, M. A. Medina-Elizalde, G. Sanchez, M. Mudelsee, et al. "BINCOR: an R package for estimating the correlation between two unevenly spaced time series." In: *R Journal* 11.1 (2019), pp. 1–14. DOI: 10.32614/rj-2019-035.

[122] D. Radjenović, M. Heričko, R. Torkar, and A. Živkovič. "Software fault prediction metrics: A systematic literature review." In: *Information and Software Technology* 55.8 (2013), pp. 1397–1418. DOI: 10.1016/j.infsof.2013.02.009.

[123] U. Raja, D. P. Hale, and J. E. Hale. "Modeling software evolution defects: a time series approach." In: *Journal of Software Maintenance and Evolution: Research and Practice* 21.1 (2009), pp. 49–71. DOI: 10.1002/smr.398.

[124] A. Rani and J. K. Chhabra. "Evolution of code smells over multiple versions of softwares: An empirical investigation." In: *Proceedings of the 2nd International Conference for Convergence in Technology (I2CT'2017)*. Vol. 2017-January. Institute of Electrical and Electronics Engineers Inc. IEEE, Dec. 2017, pp. 1093–1098. DOI: 10.1109/I2CT.2017.8226297.

[125] G. Rasool and Z. Arshad. "A review of code smell mining techniques." In: *Journal of Software: Evolution and Process* 27.11 (Nov. 2015), pp. 867–895. DOI: 10.1002/smr.1737.

[126] B. Ratner. "The correlation coefficient: Its values range between+ 1/- 1, or do they?" In: *Journal of targeting, measurement and analysis for marketing* 17.2 (2009), pp. 139–142.

[127] D. Reiersøl. *Code smells in PHP*. Karlsruhe Congress Center, 2009.

[128] J. Pereira dos Reis, F. Brito e Abreu, G. de Figueiredo Carneiro, and C. Anslow. "Code Smells Detection and Visualization: A Systematic Literature Review." In: *Archives of Computational Methods in Engineering* (Mar. 2021). DOI: 10.1007/s11831-021-09566-x.

[129] M. Reschke, T. Kunz, and T. Laepple. "Comparing methods for analysing time scale dependent correlations in irregularly sampled time series data." In: *Computers & Geosciences* 123 (2019), pp. 65–72.

[130] F. Ricca and P. Tonella. "Web application quality: Supporting maintenance and testing." In: *Information modeling for internet applications*. IGI Global, 2003, pp. 231–258. DOI: 10.4018/978-1-59140-050-9.ch011.

[131] A. Rio and F. Brito e Abreu. "PHP Code Smells in Web Apps: Evolution, Survival and Anomalies." In: *Journal of Systems and Software* 200.C (May 2023). DOI: 10.1016/j.jss.2023.111644.

[132] A. Rio and F. Brito e Abreu. "Web systems quality evolution." In: *Proceedings of the 10th International Conference on the Quality of Information and Communications Technology (QUATIC'2016)*. IEEE. IEEE, Sept. 2017, pp. 248–253. DOI: 10.1109/QUATIC.2016.060.

[133] A. Rio and F. Brito e Abreu. "Code Smells Survival Analysis in Web Apps." In: *Proceedings of the 12th International Conference on the Quality of Information and Communications Technology (QUATIC'2019)*. Springer International Publishing. Springer, 2019, pp. 263–271. DOI: 10.1007/978-3-030-29238-6_19.

[134] A. Rio and F. Brito e Abreu. "Detecting Sudden Variations in Web Apps Code Smells' Density: A Longitudinal Study." In: *Proceedings of the International Conference on the Quality of Information and Communications Technology (QUATIC'2021)*. Ed. by A. C. R. Paiva, A. R. Cavalli, P. Ventura Martins, and R. Pérez-Castillo. Springer, 2021, pp. 82–96. DOI: 10.1007/978-3-030-85347-1_7.

[135] A. Rio and F. Brito e Abreu. "PHP code smells in web apps: survival and anomalies." In: *CoRR* abs/2101.00090 (2021). DOI: 10.48550/arxiv.2101.00090.

[136] M. B. Rosson, J. Ballin, and J. Rode. "Who, what, and how: A survey of informal and professional web developers." In: *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*. IEEE. 2005, pp. 199–206. DOI: 10.1109/vlhcc.2005.73.

[137] M. B. Rosson, J. F. Ballin, J. Rode, and B. Toward. "Designing for the web revisited: a survey of informal and experienced web developers." In: *International Conference on Web Engineering*. Springer. 2005, pp. 522–532.

[138] J. Ruohonen. "An empirical analysis of vulnerabilities in python packages for web applications." In: *2018 9th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. IEEE. 2018, pp. 25–30. DOI: 10.1109/iwesep.2018.00013.

[139] A. Rényi. *Probability Theory*. North-Holland Publ. Co, Amsterdam, The Netherlands, 1970. DOI: 10.1017/9781108884013.006.

[140] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol. "An empirical study of code smells in JavaScript projects." In: *Proceedings of the 24th International Conference on Software Analysis, Evolution, and Reengineering (SANER'2017)*. IEEE. IEEE, Mar. 2017, pp. 294–305. DOI: 10.1109/SANER.2017.7884630.

[141] T. Schreiber. "Measuring information transfer." In: *Physical review letters* 85.2 (2000), p. 461.

[142] D. Schuette. *Survival Analysis in R for Beginners*. https://www.datacamp.com/community/tutorials/survival-analysis-R. Accessed: 2019-05-19. Datacamp, Jan. 2021.

[143] I. Shabani, E. Mëziu, B. Berisha, and T. Biba. "Design of modern distributed systems based on microservices architecture." In: *International Journal of Advanced Computer Science and Applications* 12 (2021), pp. 153–159. DOI: 10.14569/ijacsa.2021.0120220.

[144] C. E. Shannon. "A mathematical theory of communication." In: *The Bell system technical journal* 27.3 (1948), pp. 379–423.

[145] T. Sharma, P. Singh, and D. Spinellis. "An empirical investigation on the relationship between design and architecture smells." In: *Empirical Software Engineering* 25.5 (Sept. 2020), pp. 4020–4068. DOI: 10.1007/s10664-020-09847-2.

[146] D. Shin and S. Sarkar. "Testing for a unit root in an AR(1) time series using irregularly observed data." In: *Journal of Time Series Analysis* 17 (1996), pp. 309–321. DOI: 10.1111/j.1467-9892.1996.tb00278.x.

[147] E. Siggiridou and D. Kugiumtzis. "Granger Causality in Multivariate Time Series Using a Time-Ordered Restricted Vector Autoregressive Model." In: *IEEE Transactions on Signal Processing* 64.7 (2016), pp. 1759–1773. DOI: 10.1109/TSP.2015.2500893.

[148] S. Singh and S. Kaur. "A systematic literature review: Refactoring for disclosing code smells in object oriented software." In: *Ain Shams Engineering Journal* 9.4 (Dec. 2018), pp. 2129–2151. DOI: 10.1016/j.asej.2017.03.002.

[149] V. Singh and K. Chaturvedi. "Entropy based bug prediction using support vector regression." In: *2012 12th international conference on intelligent systems design and applications (ISDA)*. IEEE. 2012, pp. 746–751. DOI: 10.1109/isda.2012.6416630.

[150] I. Skoulis, P. Vassiliadis, and A. Zarras. "Open-source databases: within, outside, or beyond Lehman's laws of software evolution?" In: *International Conference on Advanced Information Systems Engineering*. Springer, 2014, pp. 379–393. DOI: 10.1007/978-3-319-07881-6_26.

[151] J. Śliwerski, T. Zimmermann, and A. Zeller. "When do changes induce fixes?" In: *ACM sigsoft software engineering notes* 30.4 (2005), pp. 1–5. DOI: 10.1145/1083142.1083147.

[152] M. M. Syeed, I. Hammouda, and T. Systä. "Evolution of open source software projects: A systematic literature review." In: *J. Softw.* 8.11 (2013), pp. 2815–2829. DOI: 10.4304/jsw.8.11.2815-2829.

[153] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, and M. Nagappan. "Predicting bugs using antipatterns." In: *2013 IEEE International Conference on Software Maintenance*. IEEE. 2013, pp. 270–279. DOI: 10.1109/icsm.2013.38.

[154] D. Taibi, A. Janes, and V. Lenarduzzi. "How developers perceive smells in source code: A replicated study." In: *Information and Software Technology* 92 (2017), pp. 223–235. DOI: 10.1016/j.infsof.2017.08.008.

[155] J. Tan, D. Feitosa, P. Avgeriou, and M. Lungu. "Evolution of technical debt remediation in Python: A case study on the Apache Software Ecosystem." In: *Journal of Software: Evolution and Process* 33.4 (2021), e2319. DOI: 10.1002/smr.2319.

[156] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. "An empirical investigation into the nature of test smells." In: *Proceedings of the 31st International Conference on Automated Software Engineering (ASE'2016)*. IEEE/ACM. IEEE, Aug. 2016, pp. 4–15. DOI: 10.1145/2970276.2970340.

[157] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. "When and why your code starts to smell bad." In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 403–414. DOI: 10.1109/ICSE.2015.59.

[158] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. De Lucia, and D. Poshyvanyk. "When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away)." In: *IEEE Transactions on Software Engineering* 43.11 (Nov. 2017), pp. 1063–1088. DOI: 10.1109/TSE.2017.2653105.

[159] K. J. Verhoeven, K. L. Simonsen, and L. M. McIntyre. "Implementing false discovery rate control: increasing your power." In: *Oikos* 108.3 (2005), pp. 643–647. DOI: 10.1111/j.0030-1299.2005.13727.x.

[160] R. Vern and S. K. Dubey. "A review on appraisal techniques for web based maintainability." In: *2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence)* (2014), pp. 795–799. DOI: 10.1109/confluence.2014.6949320.

[161] W3techs. *Usage Statistics and Market Share of Server-side Programming Languages for Websites, May 2019*. https://w3techs.com/technologies/overview/programming_language/all. Accessed: 2019-05-19.

[162] C Wohlin, P Runeson, M Host, M. Ohlsson, and B Regnell. *Experimentation in Software Engineering: An Introduction*. 2000.

[163] C. Wohlin. "Guidelines for snowballing in systematic literature studies and a replication in software engineering." In: *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. 2014, pp. 1–10. DOI: 10.1145/2601248.2601268.

[164] J. Wu and R. C. Holt. "Linker-based program extraction and its uses in studying software evolution." In: *Proceedings of the International Workshop on Foundations of Unanticipated Software Evolution*. 2004, pp. 1–15.

[165] J. Wu, C. W. Spitzer, A. E. Hassan, and R. C. Holt. "Evolution spectrographs: Visualizing punctuated change in software evolution." In: *Proceedings. 7th International Workshop on Principles of Software Evolution, 2004.* IEEE. 2004, pp. 57–66. DOI: 10.1109/iwpse.2004.1334769.

[166] A. Yamashita and L. Moonen. "Do code smells reflect important maintainability aspects?" In: *2012 28th IEEE international conference on software maintenance (ICSM)*. IEEE. 2012, pp. 306–315. DOI: 10.1109/icsm.2012.6405287.

[167] A. Yamashita and L. Moonen. "Do developers care about code smells? An exploratory survey." In: *2013 20th working conference on reverse engineering (WCRE)*. IEEE. 2013, pp. 242–251. DOI: 10.1109/wcre.2013.6671299.

[168] L. Yu and A. Mishra. "An empirical study of Lehman's law on software quality evolution." In: *International Journal of Software and Informatics* (2013).

[169] N. Zakas et al. *ESLint - An open source JavaScript linting utility*. https://eslint.org/. Accessed: 2021-05-04. May 2021.

[170] N. Zazworka, A. Vetro, C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull. "Comparing four approaches for technical debt identification." In: *Software Quality Journal* 22.3 (2014), pp. 403–426. DOI: 10.1007/s11219-013-9200-8.

[171] H. Zhang and S. Kim. "Monitoring software quality evolution for defects." In: *IEEE software* 27.4 (2010), pp. 58–64. DOI: 10.1109/ms.2010.66.

[172] M. Zhang, T. Hall, and N. Baddoo. "Code Bad Smells: A review of current knowledge." In: *Journal of Software Maintenance and Evolution* 23.3 (2011), pp. 179–202. DOI: 10.1002/smr.521.

[173] T. Zhu, Y. Wu, X. Peng, Z. Xing, and W. Zhao. "Monitoring software quality evolution by analyzing deviation trends of modularity views." In: *2011 18th Working Conference on Reverse Engineering*. IEEE. 2011, pp. 229–238. DOI: 10.1109/wcre.2011.35.

Appendix A.

Web code smells detection

The following listing is the main algorithm used in the detection of client-side embed CS, in the application developed:

```
Pseucode for the embed CS detector

-configure array with apps
-configure array with omitted folders (3rd party code)
-get all files
-for each file get extension
    if extention html
        parse\_html\_file
    else if extension in 'twig' 'tpl', 'mustache' //or other template
        parse\_template\_file
    else if extention php // for the interwinned code
        parse\_php\_file
    else if extension js
       parse\_js\_file

------------------------
specialized functions for cs:

parse\_file\_html\_common
    smell="embed JS"
    find ('//script[not(@src)]')

    // inline js, onlick onmouseover etc
    smell="inline JS"
    find('//*/@*[starts-with(name(), "on")]')

    //get css
    smell="embed CSS"
    find('style');

    /// inline css
    smell="inline CSS"
    find('*[style]');

-------------------------
parse  \_js\_file

    smell="css in JS";
    for each js file
        for each line
            if line contains ".style."
                add

    smell="css in JS: jquery";
    for each js file
        for each line
            if line contains ".css("
                add
```

Appendix B.

Supplementary Systematic Literature Review Materials

B.1 Included studies

The following table presents the details of the included studies in the SLR:

Table B.1: Included studies

Study number	Title	Authors	Source	Year	Type	Ref
1	Studying the evolution of PHP web applications	Amanatidis T., Chatzigeorgiou A.	Information and Software Technology	2016	Article	[2]
2	Maintenance patterns of large-scale PHP web applications	Kyriakakis P., Chatzigeorgiou A.	Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014	2014	Conference Paper	[83]
3	Prevalence and maintenance of automated functional tests for web applications	Christophe L., Stevens R., De Roover C., De Meuter W.	Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014	2014	Conference Paper	[32]
4	Language features for software evolution and aspect-oriented interfaces: An exploratory study	Dyer R., Rajan H., Cai Y.	Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)	2013	Conference Paper	[22]
5	Dead code elimination for web systems written in PHP: Lessons learned from an industry case	Boomsma H., Hostnet B.V., Gross H.-G.	IEEE International Conference on Software Maintenance, ICSM	2012	Conference Paper	[19]
6	An empirical study of software metrics for assessing the phases of an agile project	Concas G., Marchesi M., Destefanis G., Tonelli R.	International Journal of Software Engineering and Knowledge Engineering	2012	Article	[35]
7	Eigenspace-based anomaly detection in computer systems	Idé T., Kashima H.	KDD-2004 - Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining	2004	Conference Paper	[73]
8	SQL-Injection Security Evolution Analysis in PHP	E. Merlo; D. Letarte; G. Antoniol	2007 9th IEEE International Workshop on Web Site Evolution	2007	IEEE Conference Publications	[102]
9	Security Model Evolution of PHP Web Applications	D. Letarte; F. Gauthier; E. Merlo	2011 Fourth IEEE International Conference on Software Testing, Verification and Validation	2011	IEEE Conference Publications	[90]
10	Detecting design violations in django-based web applications	Correia R., Adachi E.	ACM International Conference Proceeding Series	2019	Conference Paper	[36]
11	Time present and time past: Analyzing the evolution of javascript code in the wild	Mitropoulos D., Louridas P., Salis V., Spinellis D.	IEEE International Working Conference on Mining Software Repositories	2019	Conference Paper	[105]
12	An empirical analysis of vulnerabilities in python packages for web applications	Ruohonen J.	Proceedings - 2018 9th International Workshop on Empirical Software Engineering in Practice, IWESEP 2018	2019	Conference Paper	[138]
13	Code Smells Survival Analysis in Web Apps	Rio A., Brito e Abreu F.	Communications in Computer and Information Science	2019	Conference Paper	[133]
14	Analyzing the evolution of JavaScript applications	Chatzimparmpas A., Bibi S., Zozas I., Kerren A.	ENASE 2019 - Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering	2019	Conference Paper	[31]
15	Classification and distribution of RBAC privilege protection changes in wordpress evolution (short paper)	Laverdiere M.-A., Merlo E.	Proceedings - 2017 15th Annual Conference on Privacy, Security and Trust, PST 2017	2018	Conference Paper	[86]
16	Detection of protection-impacting changes during software evolution	Laverdiere M.-A., Merlo E.	25th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2018 - Proceedings	2018	Conference Paper	[87]
17	Modeling software evolution defects: a time series approach	Raja, Uzma; Hale, David P.; Hale, Joanne E.	Journal of Software Maintenance and Evolution: Research and Practice	2009	Conference Paper	[123]
18	Open-Source Databases: Within, Outside, or Beyond Lehman's Laws of Software Evolution?	Ioannis SkoulisPanos Vassiliadis-Apostolos Zarras	Advanced Information Systems Engineering	2014	Chapter	[150]
19	Evolution of technical debt remediation in Python: A case study on the Apache Software Ecosystem	Tan J;Feitosa D;Avgeriou P;Lungu M	Journal of Software: Evolution and Process	2021	Journal Article	[155]
20	You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions	Nikiforakis, Nick and Invernizzi, Luca and Kapravelos, Alexandros and Van Acker, Steven and Joosen, Wouter and Kruegel, Christopher and Piessens, Frank and Vigna, Giovann	Proceedings of the ACM Conference on Computer and Communications Security	2012	Conference Paper	[110]
21	On the evolution of technical lag in the npm package dependency network	A. Decan, T. Mens and E. Constantinou	Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018	2018	Conference Paper	[40]

B.2 Techniques and methods used

The following table presents the techniques and methodology used in each SLR study:

Table B.2: Techniques and methodology used in the studies

Study	Techniques used	Methodology
1	(trend , slope) linear reg, Mann -Kendal(M-K trend test)	choose metrics for each law (phpdepend); investigate trend and slope
2	numeric(percentage), survival analysis	tool- download versions - analyse (observational) - percentages, survival
3	measure nº changes, number of commits	1- cross-sec, 2,3 longitudinal study- observational
4	measure changes (aspect interfaces ,metrics)	observational -quantify changes
5	measure unused files	visualize tree map, eclipse file decorator plugin (everyday)
6	measure, correlate (metrics)	experience with 4 phases (adoption agile practices)
7	anomaly/fault detection/probability distribution from eigenvector	anomaly detection/feature vector (services), matrix with probability distribution
8	statistics-trends	observational - measure access level(visualise trends)
9	measure changes (evolution)	obs- php parser- CF graph- graph rewriting rules - security model
10	measure evolution CS	tool (developed), detect , interview, evolution study (the important)
11	measure changes, nature of changes, survival of js	observation evolution, survival (download wget .js and inline js)
12	compare; calculate probability (Markov chain)	verify vulnerabilities, compare with db of vulnerabilities; calculate vulnerability prob
13	survival - log rank	download releases, extract code smells, compare survival by type, time (log-rank)
14	measure; trends(M-K test)	download releases - measure metrics;statistics- find trends (Mann-Kendall)
15	distribution, correlation	obs: measure number, correlation, classification of privilege changes
16	distribution, quartiles	obs: divide quartiles; measure percentages and abs; histograms (find protection impact changes)
17	ARIMA, auto correlation	(defects) time series, ACF, PACF; arima to predict (last 4 months); test MSE, MAPE, MAD
18	measure changes, effort, trend	download releases, measures changes in schema, effort, trend
19	measure with tool, rates, effort and survival(from tool)	medians, percentages, stats
20	count inclusions (same, new, quality, xss) distribution?	count, check elements of quality and security
21	measure, distribution, median, mean, classify	measure technical lag

B.3 Quality Analysis

Table B.3: Studies quality analysis

StudyNumber	Aim Objectives	Context defined	Study Design	Sample description	Tecniques method defined	Findings results conclusions	Value of Study	Data replication	SUM
S1	1	1	1	1	1	1	1	1	8
S2	1	1	1	1	1	1	1	1	8
S3	1	1	1	1	1	1	1	1	8
S4	1	1	0	1	0	1	1	1	6
S5	1	1	1	1	1	1	1	1	8
S6	1	0	1	1	1	1	1	1	7
S7	1	1	1	1	1	0	1	1	7
S8	1	1	1	1	0	0	1	1	6
S9	1	1	1	1	0	0	1	1	6
S10	1	1	1	1	1	1	1	1	8
S11	1	1	1	1	1	1	1	1	8
S12	1	1	1	1	1	0	1	1	7
S13	1	1	1	1	1	1	1	1	8
S14	1	1	1	1	1	1	1	1	8
S15	1	1	1	1	1	1	1	1	8
S16	1	1	1	1	0	1	1	1	7
S17	1	1	1	1	1	0	1	1	7
S18	1	1	1	1	0	0	1	1	6
S19	1	1	1	1	1	1	1	1	8
S20	1	1	1	1	1	0	1	1	7
S21	1	1	1	1	1	1	1	1	8

B.4 Database used

The following diagram presents the database built to store the information on the SLR research articles.

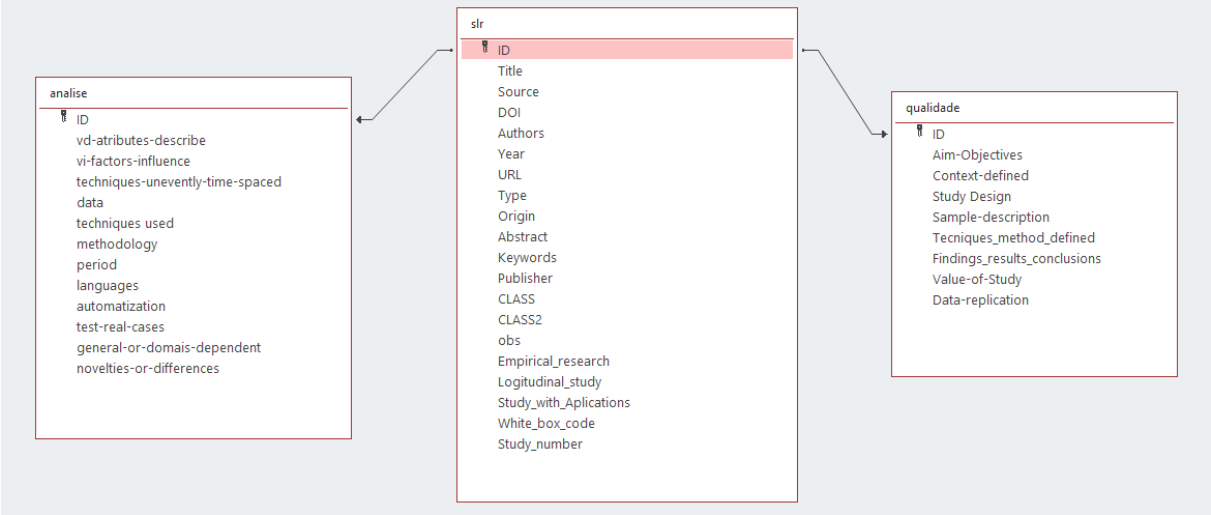


Figure B.1: Database built to manage the SLR articles data

In the diagram B.1, the "slr" table represents the table used to store the articles, where most fields are self-explanatory. The CLASS and CLASS2 fields were used to classify the article in ten first two selection phases. The fields "Empirical research," "Longitudinal study," "Study with applications," and "White box code" are Boolean. The table "analise" (*analysis*) was used to answer the secondary questions using the form shown next. Finally, the table "qualidade" (*quality*) was used to classify the quality of the articles, also filled with a form. In both tables, the fields are self-explanatory.

B.5 Forms

The following figure is the form used to visualize information about the article and to classify it. Some information is stored in the table "slr" and some in the table "analise". The form is used so we can visualize only one record at once.

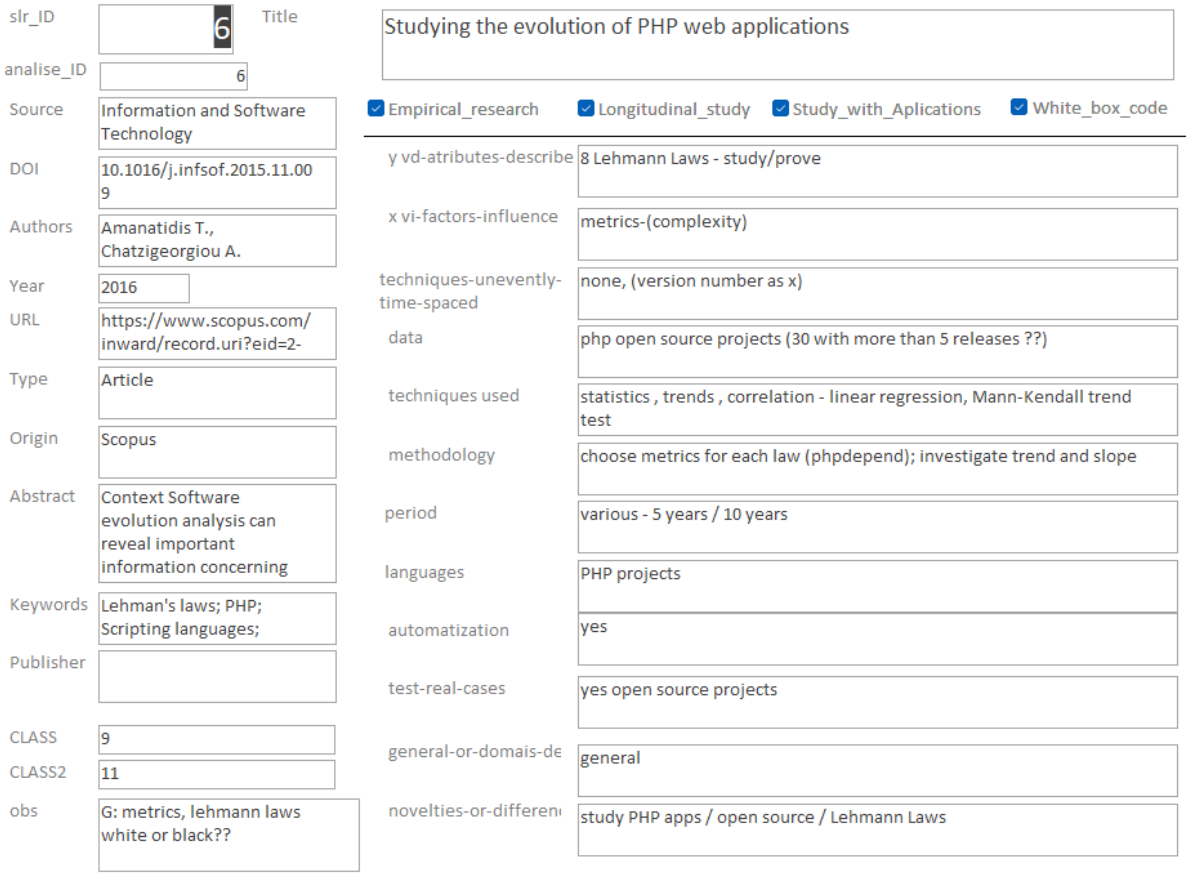


Figure B.2: Form for data extraction of SLR articles

Appendix C.

Web Development and Code Smells: an industry survey - Extended Data

C.1 Education and experience

Q1-Education

Table C.1: Level of education

Level of education	Count	Percentage
Bachelor / Licentiate	28	49%
Master	19	33%
PhD or higher	5	9%
High school or lower	3	5%
Technical / Professional	2	4%
Total	57	100%

Q2-Country

Table C.2: Country Based

Country	Count	Percentage
Portugal	43	75%
Brazil	9	16%
Denmark	2	4%
Ukraine	1	2%
Unknown	1	2%
Spain	1	2%
Total	57	100%

Q3-Experience

Table C.3: Years of Experience

years	count	Percentage
0	3	5%
1	8	14%
2	7	12%
3	11	19%
4	4	7%
5	1	2%
6	5	9%
7	1	2%
8	2	4%
9	2	4%
10	2	4%
11	1	2%
13	2	4%
15	2	4%
17	1	2%
19	1	2%
20	2	4%
21	1	2%
22	1	2%
	57	100%

Q4-Education description - professional or other

Table C.4: Developer Description

Developer description	Count	Percentage
b-professional within a company	39	68%
b-professional within a company,c-freelance	6	11%
a-student	4	7%
a-student,b-professional within a company	4	7%
d-other	2	4%
c-freelance	2	4%
Total	57	100%

Q5-Developer type (scope)

Table C.5: Developer Type

Type of Developer	Type of Developer	Percentage
a. Fullstack developer	33	58%
b. Only Client developer (HTML/CSS/JS)	9	16%
c. Only Server developer	8	14%
e. other (please specify)	4	7%
d. Web services developer	3	5%
	57	100%

C.2 2nd part : Development

Q6-Web development production Open-source / Proprietary

Table C.6: Software production license

value	count	Percentage
100%OS	5	11%
75%OS / 25%P	5	11%
50%OS / 50%P	5	11%
25%OS / 75%P	7	16%
100%P	23	51%
Total	45	100%

Q7 Production - Frameworks based nor no frameworks ? slider:
1-No framework
5-Fully Framework Based

Table C.7: Software production: frameworks or no framework based

Web development production (No framework / Fully Framework Based)	Count
1	1
1.3	1
1.8	1
2.3	1
3	2
3.2	1
3.3	1
3.5	1
3.6	1
3.7	3
3.8	1
3.9	3
4	4
4.1	4
4.2	3
4.3	3
4.4	1
4.6	3
4.7	3
4.8	1
4.9	1
5	9

Q8 Average project duration

Table C.8: Average project duration

Weeks	count	percentage
1-2 weeks	0	0%
3-4 weeks	6	12%
1-3 months	6	12%
3-6 months	10	20%
6 months or more	27	55%
	49	100%

Q9 Average team size

Table C.9: Average team size

team_nr	count	%
1	1	2%
2	3	6%
3-5	14	29%
6-10	24	49%
>10 or more	7	14%
	49	100%

Q10 Types of people in team in a typical Web project

Table C.10: Types of people in a web project

team constitution	count	perc.
Software Eng./W.Developer	9	18%
Software Eng./W.Developer,Creative/Web Designer,Team Manager	6	12%
Software Eng./W.Developer,Other	5	10%
Software Eng./W.Developer,Team Manager	4	8%
Software Eng./W.Developer,Business Expert/Marketter,Team Manager	4	8%
Software Eng./W.Developer,Creative/Web Designer,Business Exp./Marketter,Team Manager	4	8%
Team Manager	2	4%
Software Eng./W.Developer,Creative/Web Designer,Business Exp./Marketter,Team Manager,Domain Exp./Area Spec.	2	4%
Software Eng./W.Developer,Creative/Web Designer,Business Exp./Marketter,Domain Exp./Area Spec.	2	4%
Software Eng./W.Developer,Creative/Web Designer	2	4%
Software Eng./W.Developer,Creative/Web Designer,Business Exp./Marketter,Other	1	2%
Creative Designer/Web Designer	1	2%
Software Eng./W.Developer,Creative/Web Designer,Other	1	2%
Software Eng./W.Developer,Business Exp./Marketter,Team Manager,Domain Exp./Area Spec.	1	2%
Software Eng./W.Developer,Domain Expert / Area Specialist	1	2%
Business Exp./Marketter	1	2%
Software Eng./W.Developer,Creative/Web Designer,Business Exp./Marketter	1	2%
Software Eng./W.Developer,Creative/Web Designer,Business Exp./Marketter,Team Manager,Domain Exp./Area Spec.,Other	1	2%
Software Eng./W.Developer,Creative/Web Designer,Business Exp./Marketter,Team Manager,Other	1	2%
Software Eng./W.Developer,Creative/Web Designer,Domain Exp./Area Spec.	1	2%
	51	100%

C.3 3rd part: Languages and Tools

Q11 Experience – server-side languages?

Table C.11: Experience server-side languages

years	PHP	C #	Ruby	Java	Scala	Python	JavaScript(nodejs)	PERL	Other
0	3	2	1	3	4	7	7	3	4
1	3	6	5	8	2	10	11	0	3
2	5	1	1	5	1	10	7	2	1
3-4	2	1	0	6	0	1	3	0	2
5 or more	6	10	0	17	0	0	7	0	0
total	19	20	7	39	7	28	35	5	10

Q11 Experience – client-side languages?

Table C.12: Experience client-side languages

years	Javascript	CSS	HTML	ActionScript	Other
0	6	7	6	3	3
1	7	7	8	3	0
2	10	8	5	2	1
3-4	8	10	10	0	1
5 or more	13	13	18	1	0

Q13 tool types used for Web development?

Table C.13: Tools used in web development

Tools	Number	Percentage
Requirements / Change management	25	51%
Project Planning / Project tracking	35	71%
Version Control	43	88%
Issue tracking	30	61%
Testing (unit or functional testing)	35	71%
Code quality (code review, code smells, metrics, etc)	33	67%
Other(s)	1	2%
responses	49	

C.4 4th part - Code Smells

Q14 familiarity with code smells

Table C.14: CS familiarity

familiar	count	
I have never heard of them	2	6%
I have heard about them, but I am not so sure what they are	6	17%
have a general understanding, but do not use these concepts	5	14%
have a good understanding, and use these concepts sometimes	13	37%
I have a strong understanding, and use these concepts frequently	9	26%
	35	

Q15 Understand difference between Code Smells and Code standards adherence

Table C.15: Understand difference between CS and Code Standards

Undestand dif CS and Cstandards	Count	Percantage
Don't understand	8	23%
Slightly understand	4	11%
Moderately understand	7	20%
Mostly understand	7	20%
Fully understand	9	26%
	35	

Q16 Concerned code smells in code

Table C.16: Concerned code smells in code

Concerned CS in Code	count	percentage
Extremely important	7	24%
Very important	14	48%
Moderately important	4	14%
Slightly important	1	3%
Not at all important	3	10%
	29	

Q17 Justifications

Some justifications in the answers

- avoid bad code
- good software quality
- If something is wrong, it should be addressed.
- I believe knowing how to detect, mitigate and potentially remove code smells is one way to improve code quality, but they should be seen as a guide since in the real world sometimes they cannot be fixed
- We strive for quality code

- Smells are extremely uncertain: can be a symptom of a big or small problem. That uncertainty is extremely risky and at least a bigger comprehension of the underlying problem is needed, even if a solution is not implemented right away.
- My team runs linting tools (ESLint and Credo) both locally and on our CI pipeline. However, we have not looked deeply into configuring them to be more strict.
- When working on software that has to be working and maintained for a longer period of time, it is crucial to understand malfunctions and make changes as efficient as possible. Code smells might make the work on software more complex than it has to be.
- code smells point us to architectural issues, tech debt, things that can - and should - be improved on our code.
- They can pose potential bugs or security flaws.
- We use sonar qube in order to check for code smells and try to fix it
- Code smells make it difficult for other developers or even the original author to understand the code after a while. Some code smells also generate larger bundle sizes which makes them heavier and less efficient.
- I understand the importance of it, but not exactly how to use it.
- I understand that code smells indicate parts of our codebase that might be refactored to be more clear to a reader or that might endanger the workings of the system. Therefore, these parts of the code should be reviewed and, possibly, reworked for improvement.
- increase technical debt, slow the team/future releases

[This page has been intentionally left blank]

APPENDIX D.

WEB SMELLS CATALOGUE ARCHITECTURE

D.1 Web Smells catalogue database architecture

The following diagram presents the database architecture of the database that supports the web applications to extend the web code smells catalogue, available in https:\websmells.org.

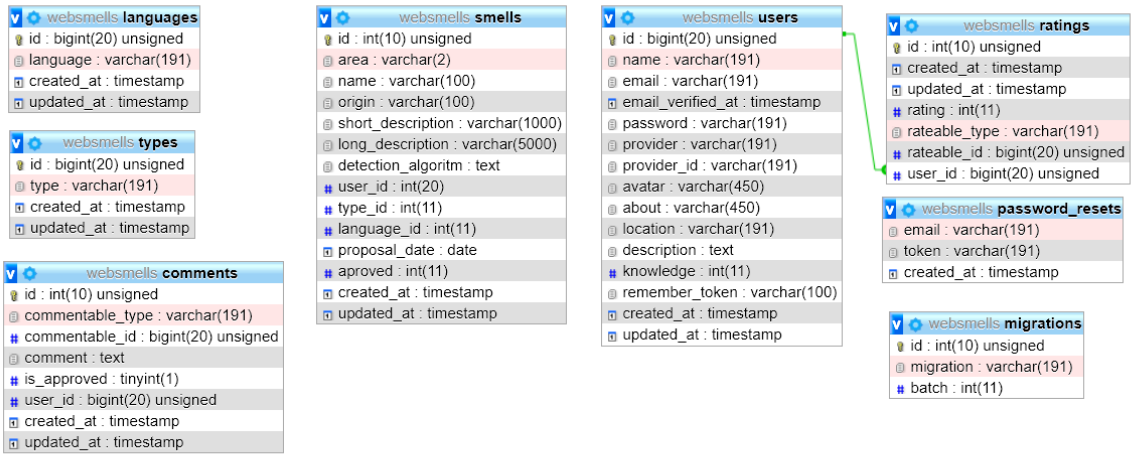


Figure D.1: Web Smells catalogue web app database architecture

The web app is described in Chapter 4. This is the second version, built with the framework Laravel. The first version was built with no framework. In the diagram D.1, all the relations are done in code, i.e., the classes in code that represent the models have the correspondent relations between tables.

The table "smells" contains the code smells proposed. If an admin approves the CS, the Boolean approved gets a "true" value. The table "languages" represents the programming languages, and the type table represent the scope (server-side or client-side). Both have foreign keys in the table smells.

The "users" table stores the users and login information. Users can comment and rate, and their comments and ratings are stored in the respective tables, where a foreign key represents the author (user). In the table smells, a foreign key also represents the user that proposes the smell.

The last two tables, "password resets" and "migrations," are necessary for Laravel. The former stores the password reset requests and the latter is necessary to transform the classes representing the tables (the models) into tables on databases via migrations.

Appendix E.

Evolution and Survival Extended Data (Chapter 5)

E.1 RQ1 - Evolution of CS characterization

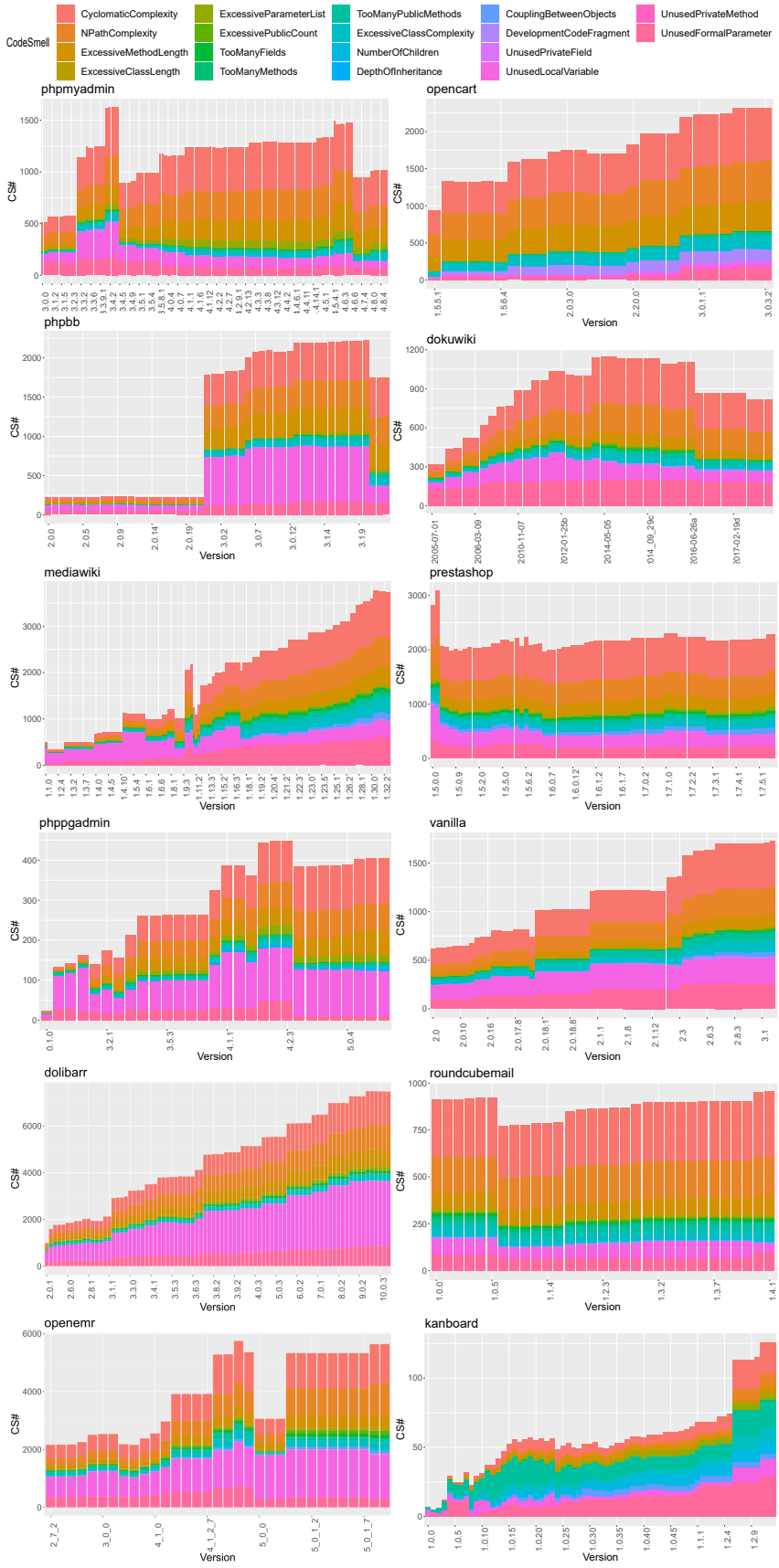


Figure E.1: Evolution of CS in 12 web apps (absolute value)

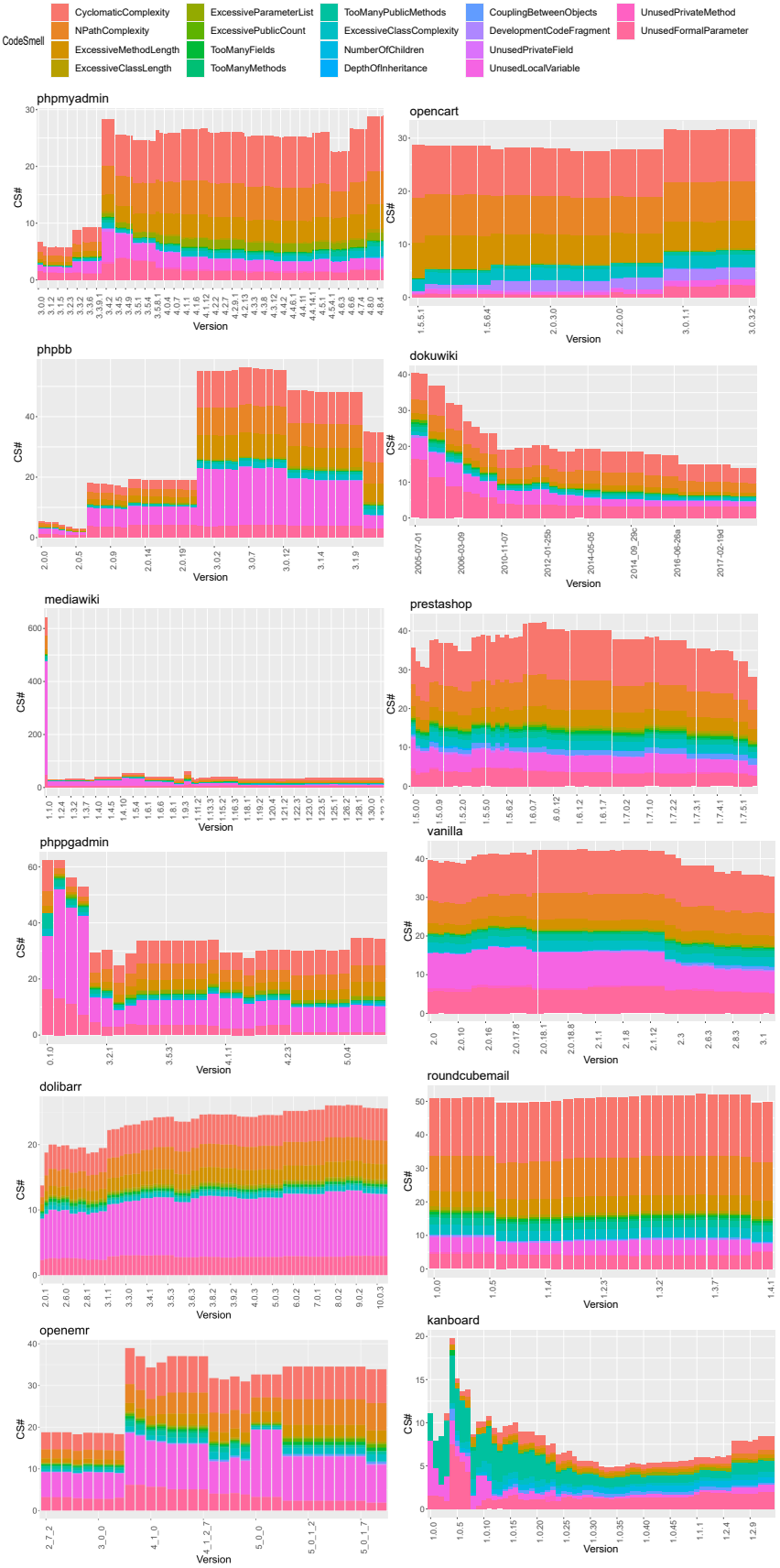


Figure E.2: Evolution of CS in 12 web apps (CS density -by size)

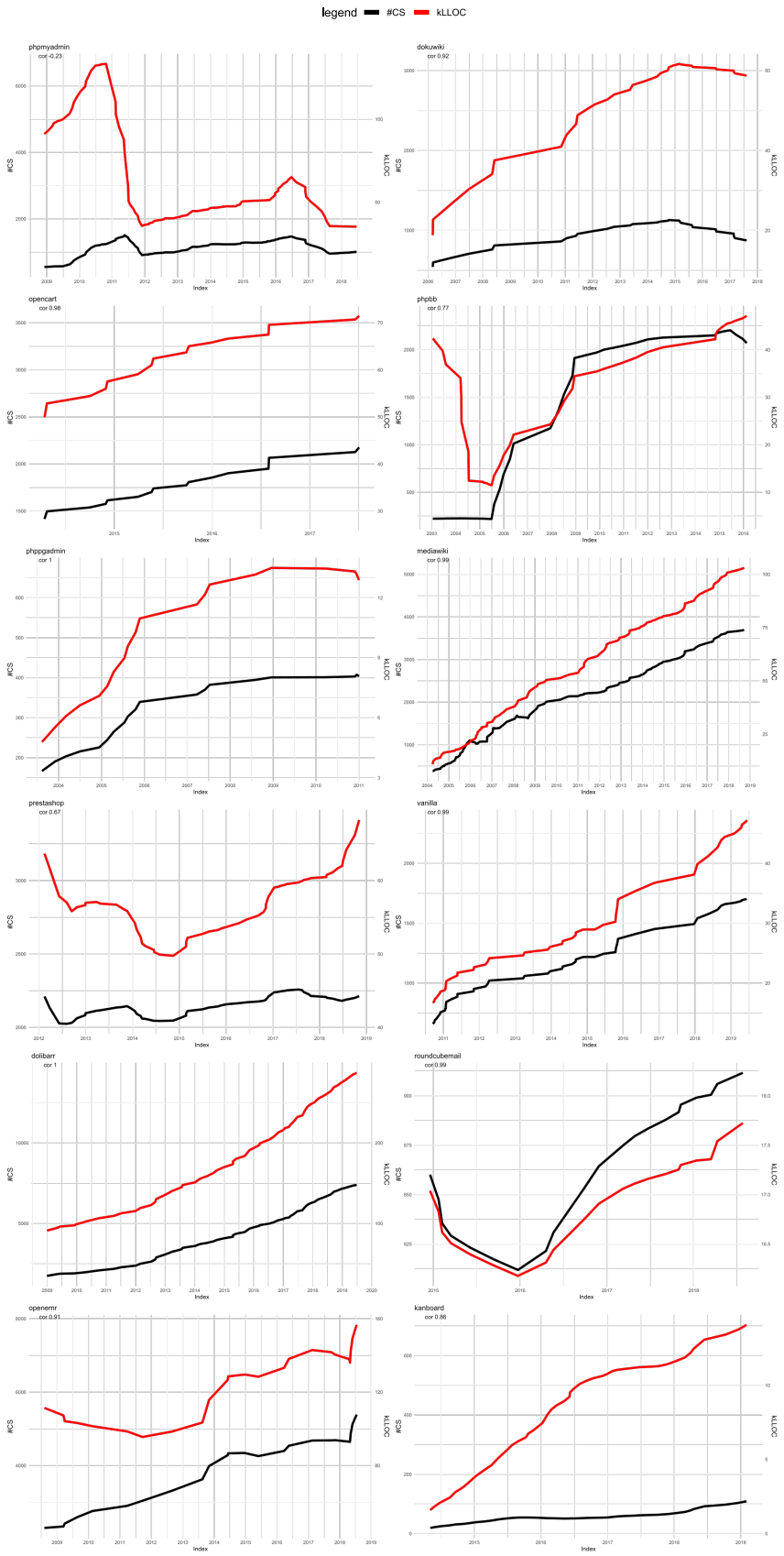


Figure E.3: Compare evolution of absolute CS with size (LLOC) - 12 apps

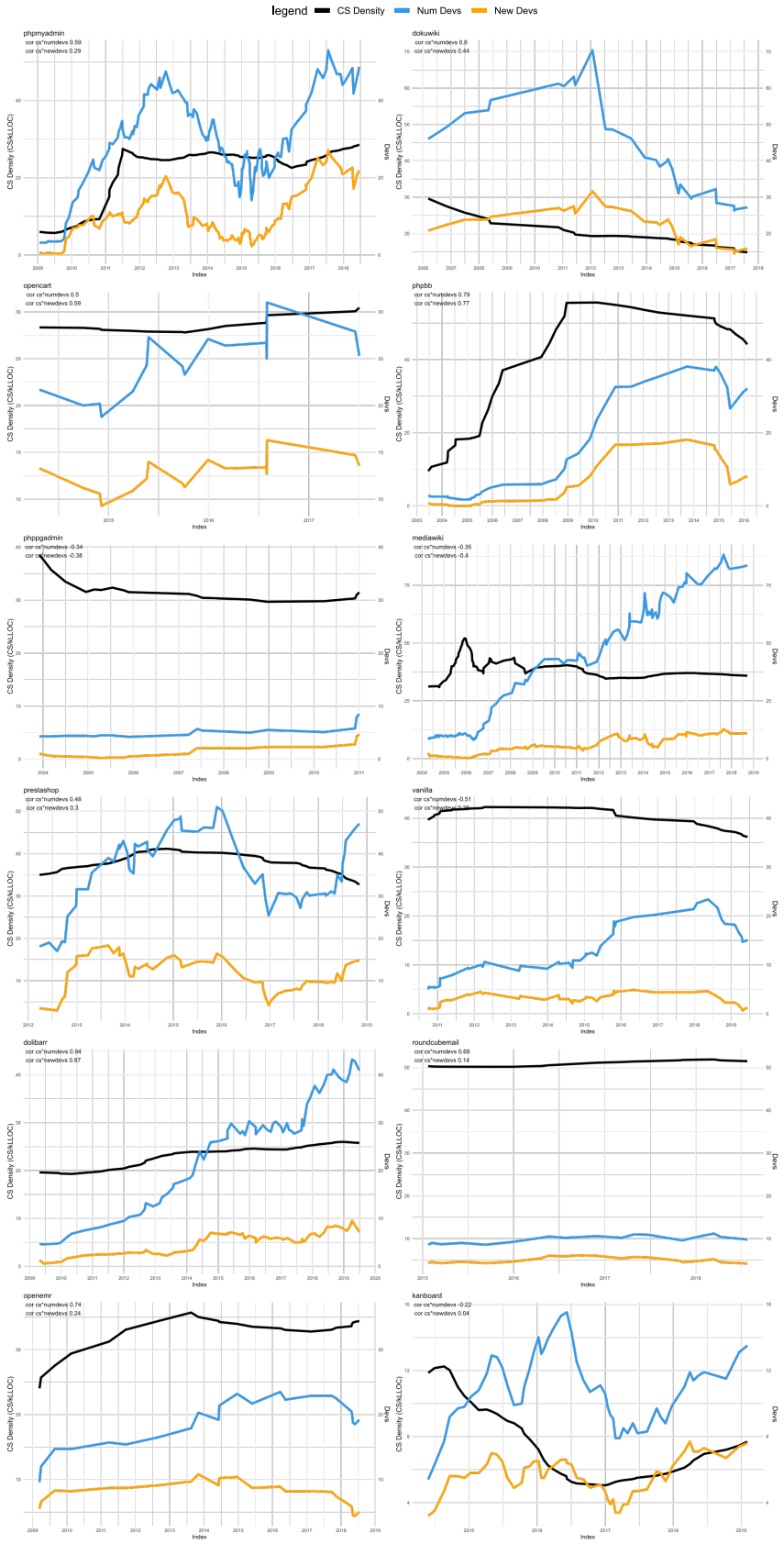


Figure E.4: Compare evolution: CS Density vs #developers and #new developers

E.2 RQ2 - Distribution and survival/lifespan of CS

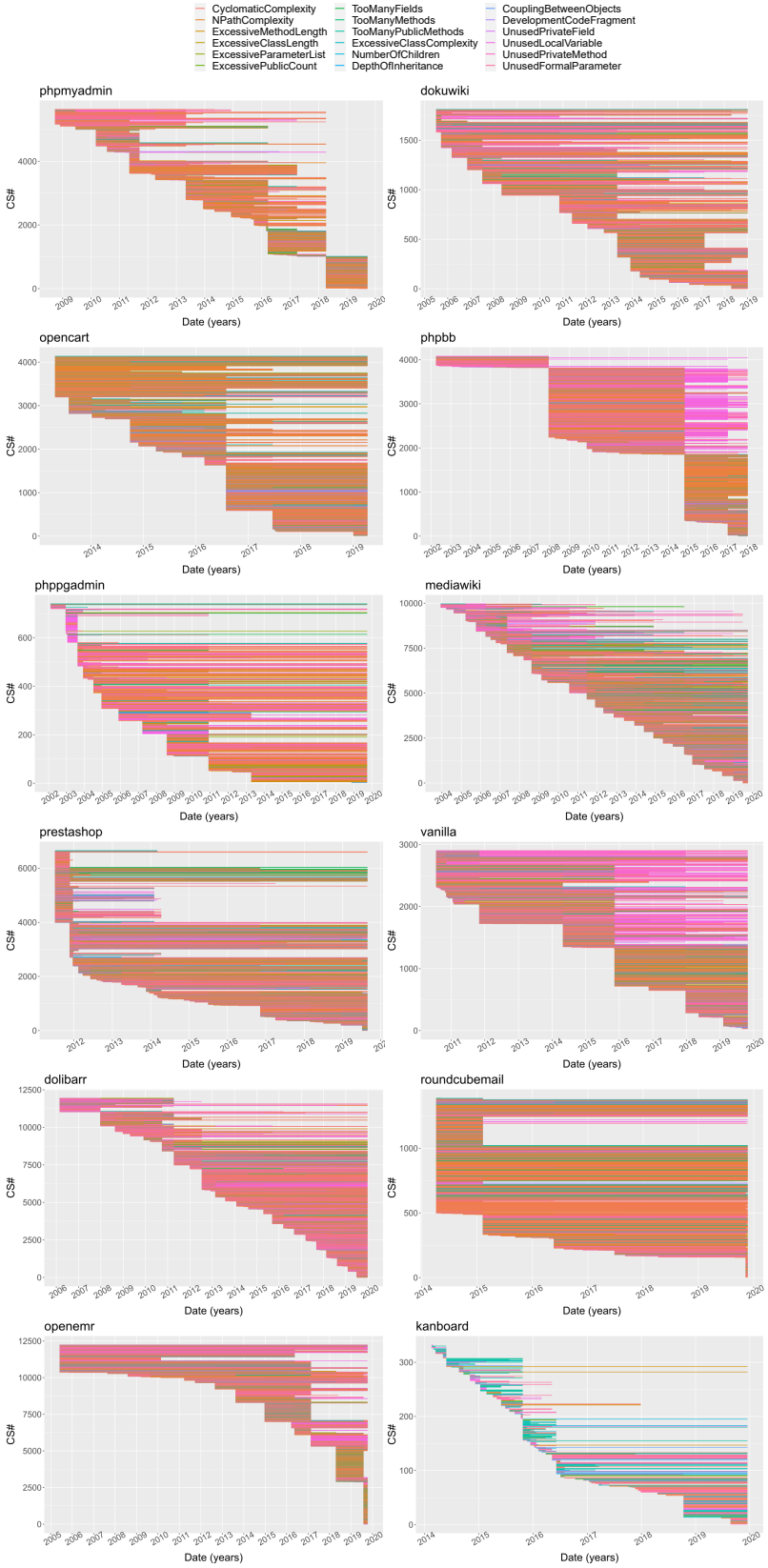


Figure E.5: Individual CS Lifespan (12 applications)

E.3 RQ3 - Survival of localized CS vs scattered CS

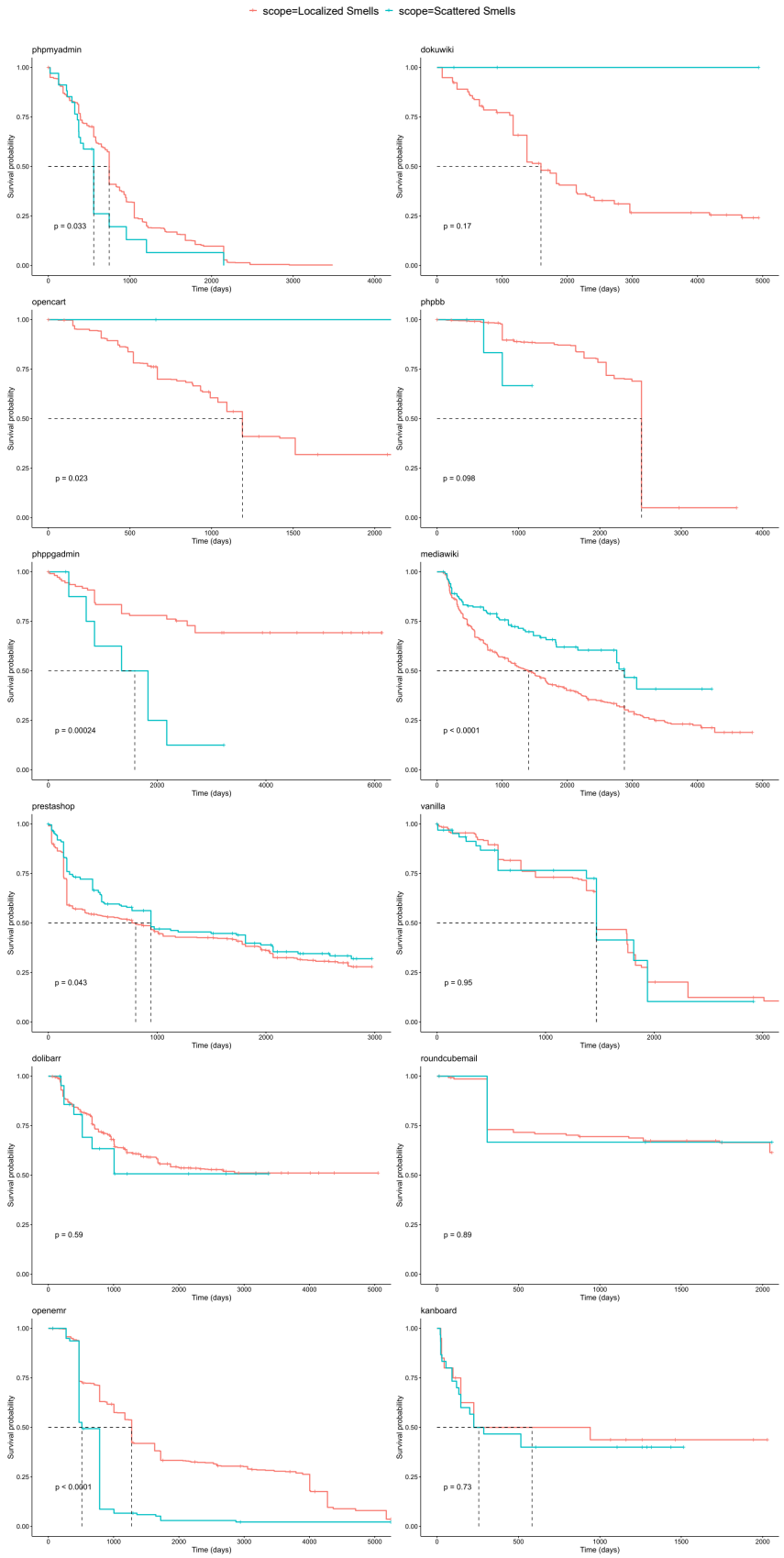


Figure E.6: Survival curves of 6 CS by scope (localized CS vs scattered CS) in 12 apps

E.4 RQ4 - Survival of CS by timeframe

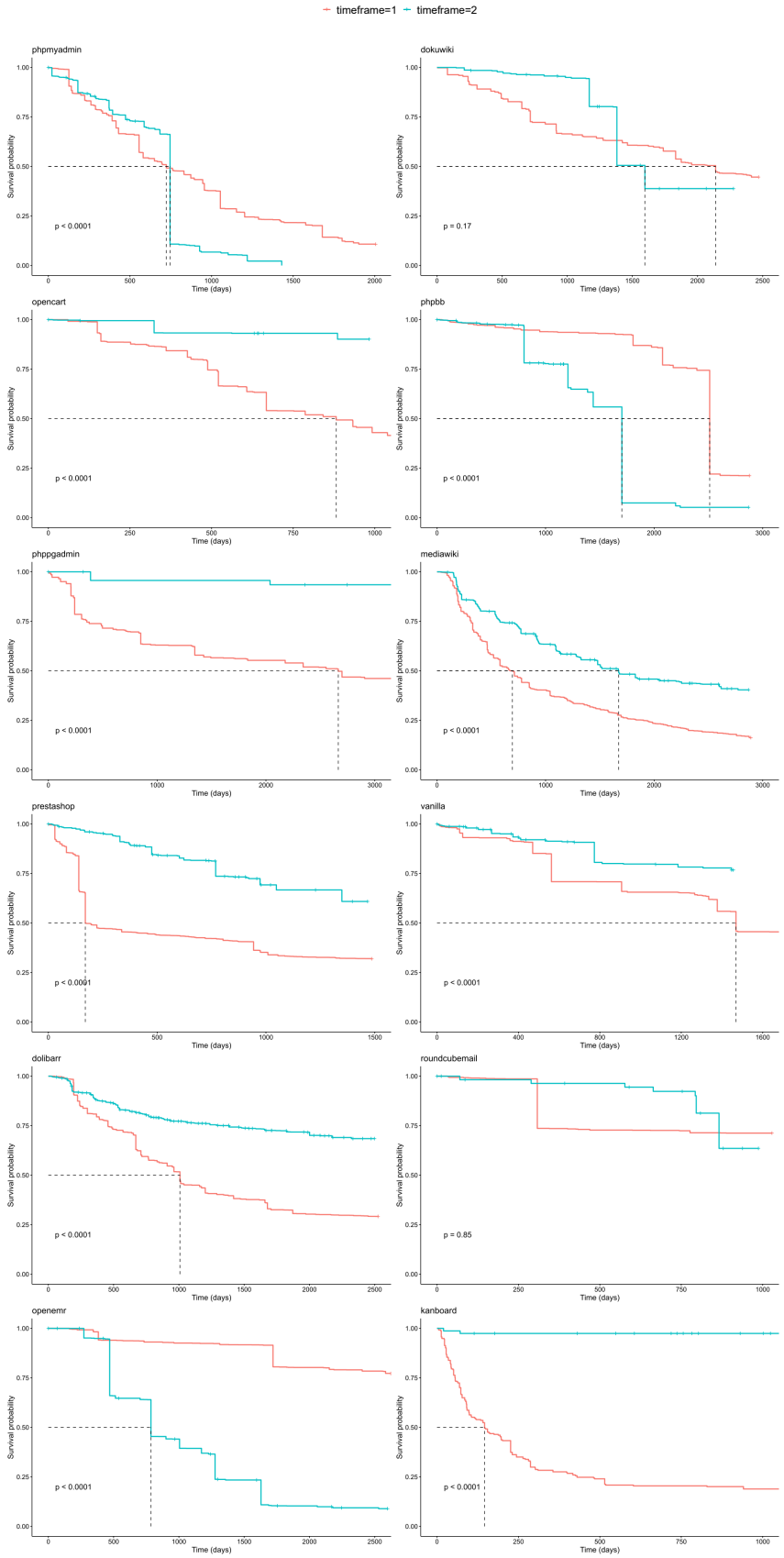


Figure E.7: Survival curves of 18 CS by timeframe in 12 apps

E.5 RQ5 - CS Evolution anomalies/sudden changes

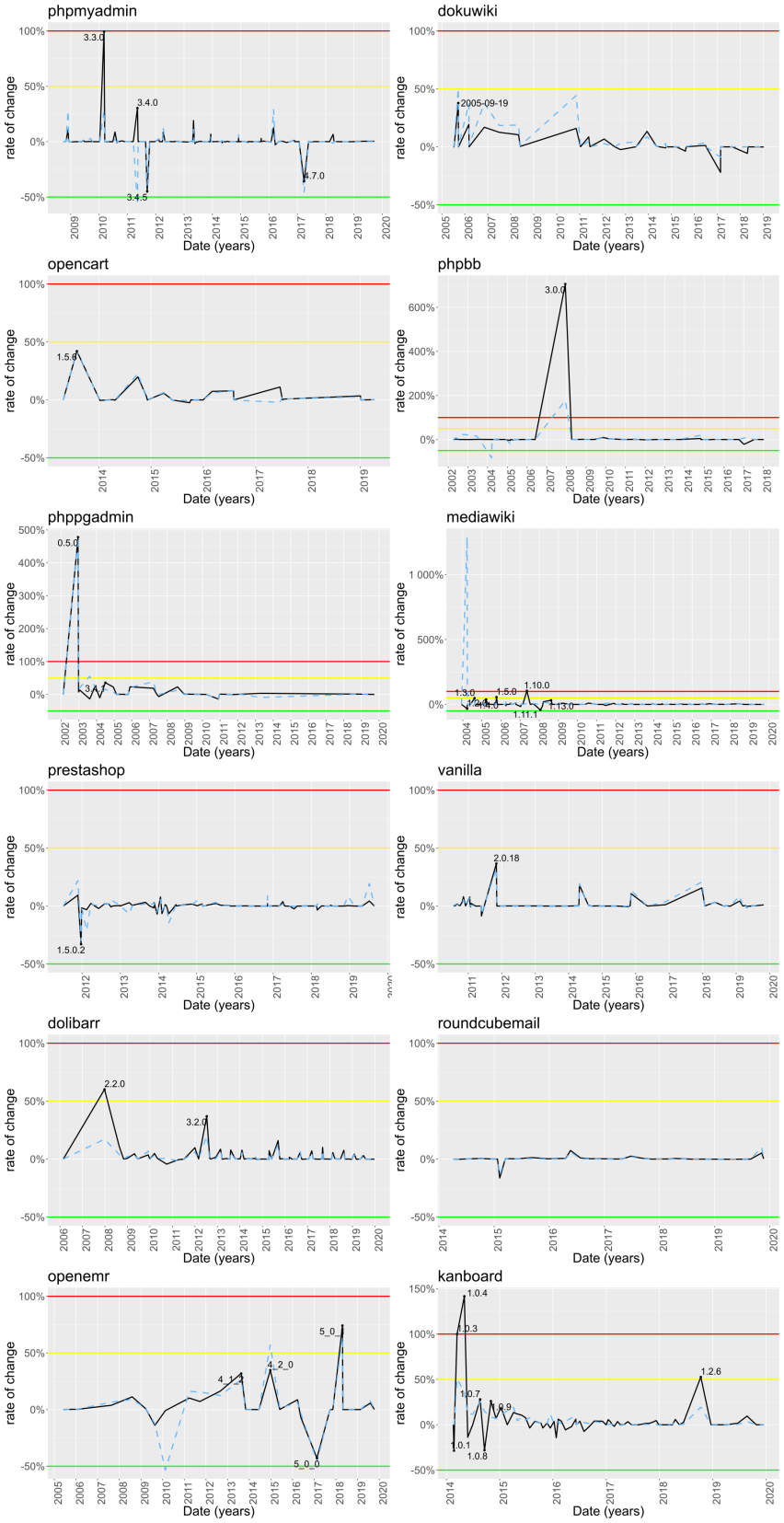


Figure E.8: CS change from last release of CS and size(kLOC) separated - 12 apps

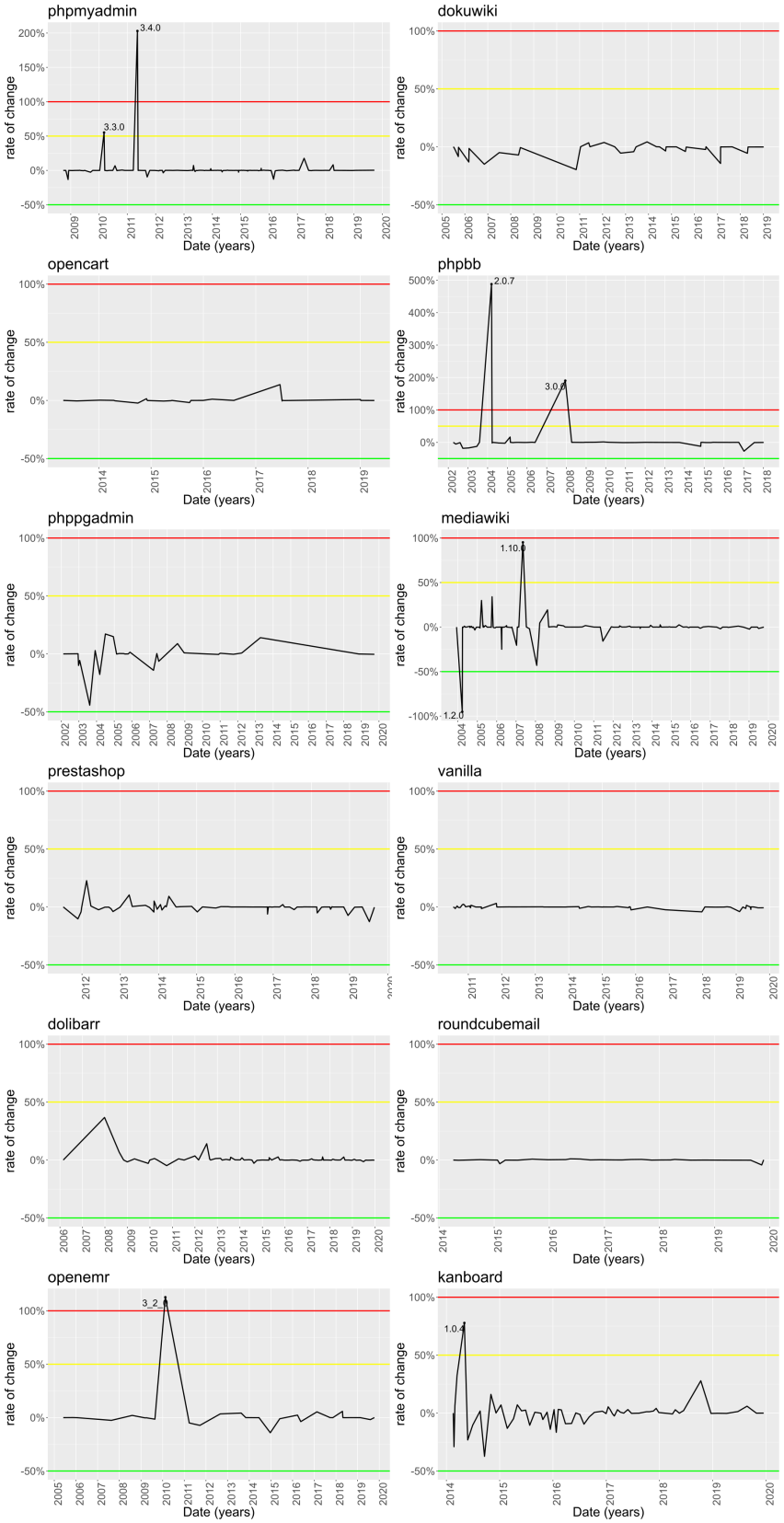


Figure E.9: CS density change (by kLOC) from last release and peaks - 12 apps

Appendix F.

Causal Inference Extended Data I - Graphics and tables up to lag4 (Chapter 6)

F.1 RQ1 - Server- and client-side Code Smell evolution

F.1.1 Server- and client-side CS + metrics evolution extended graphics

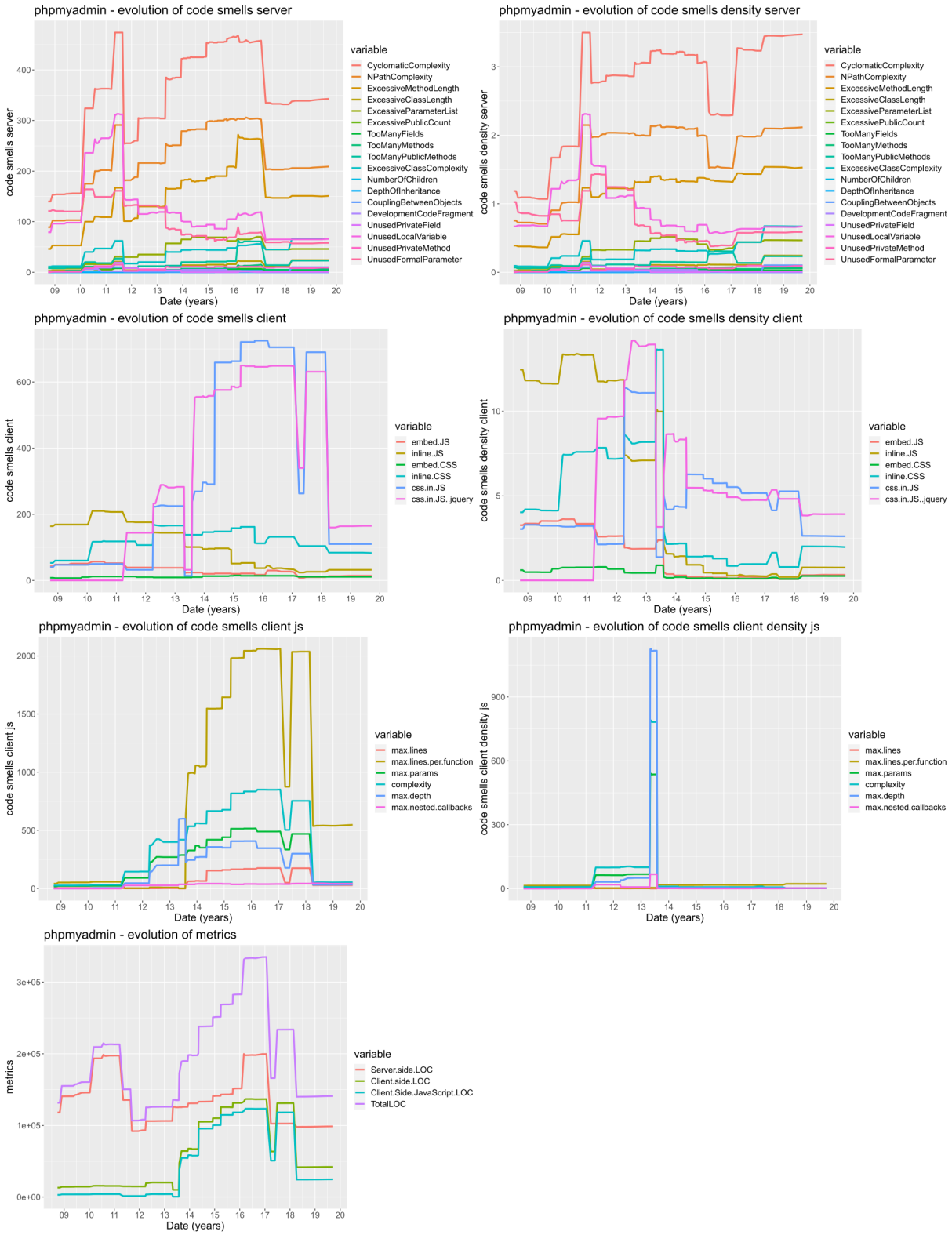


Figure F.1: CS and metrics evolution for phpMyAdmin

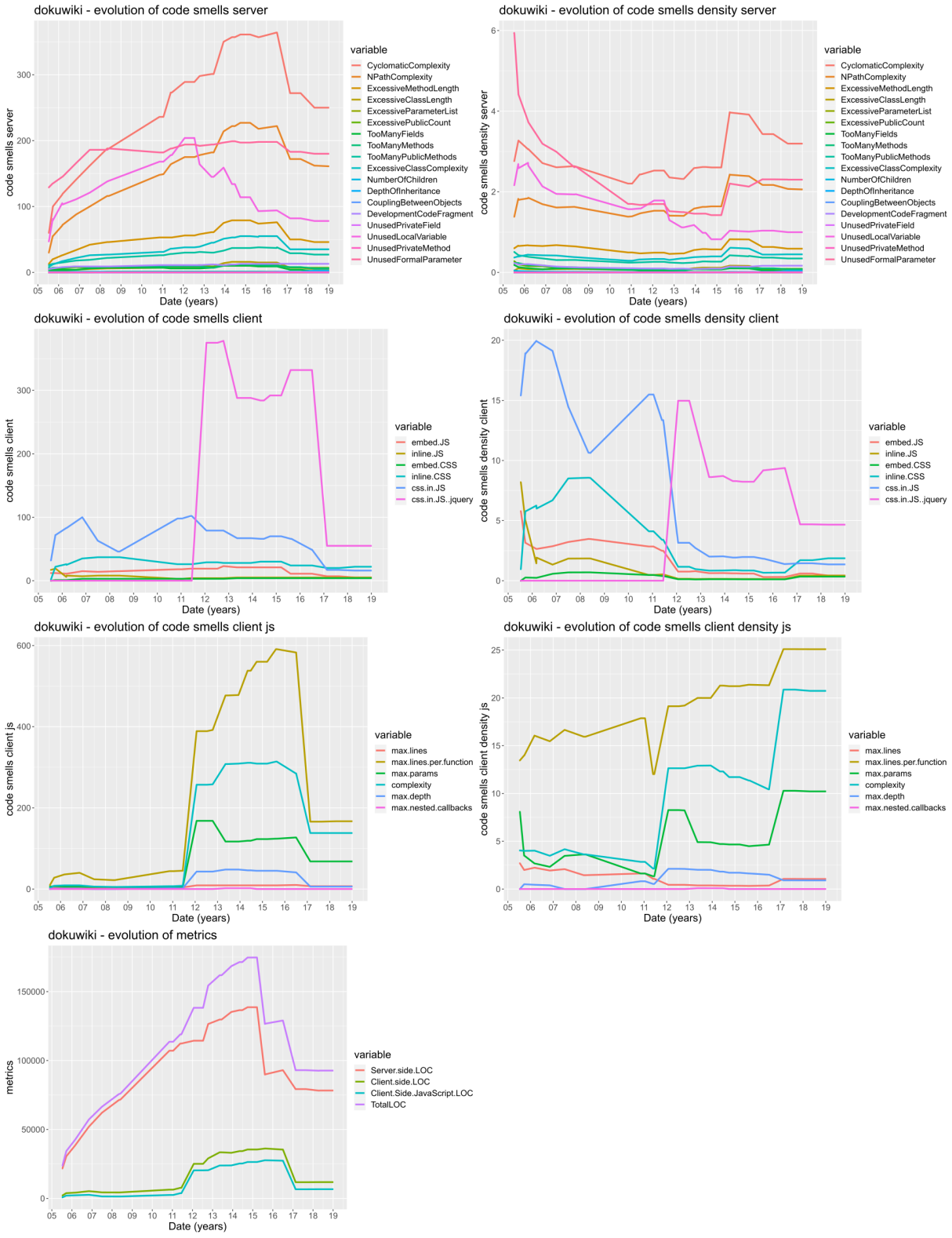


Figure F.2: CS and metrics evolution for DokuWiki

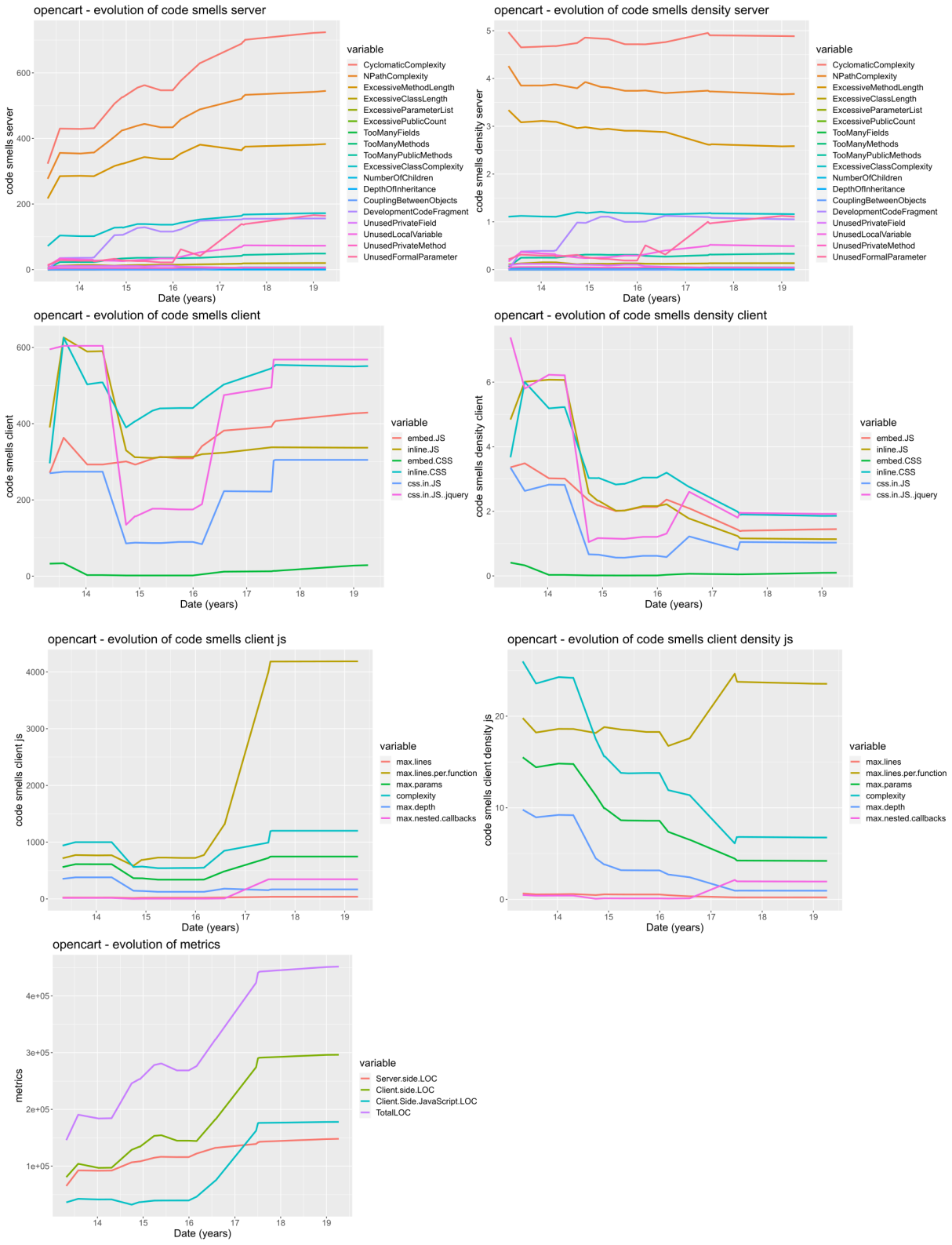


Figure F.3: CS and metrics evolution for OpenCart

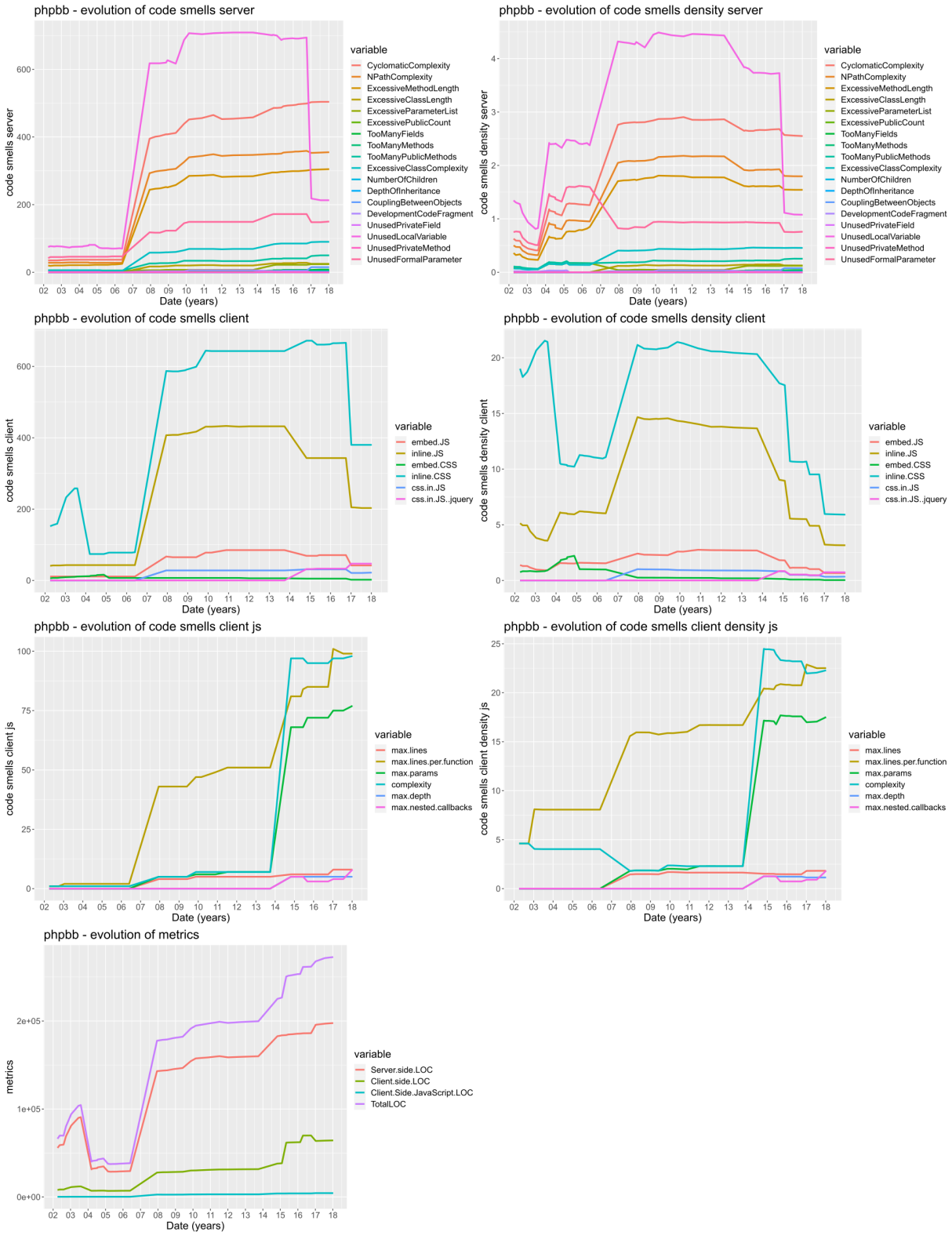


Figure F.4: CS and metrics evolution for phpBB

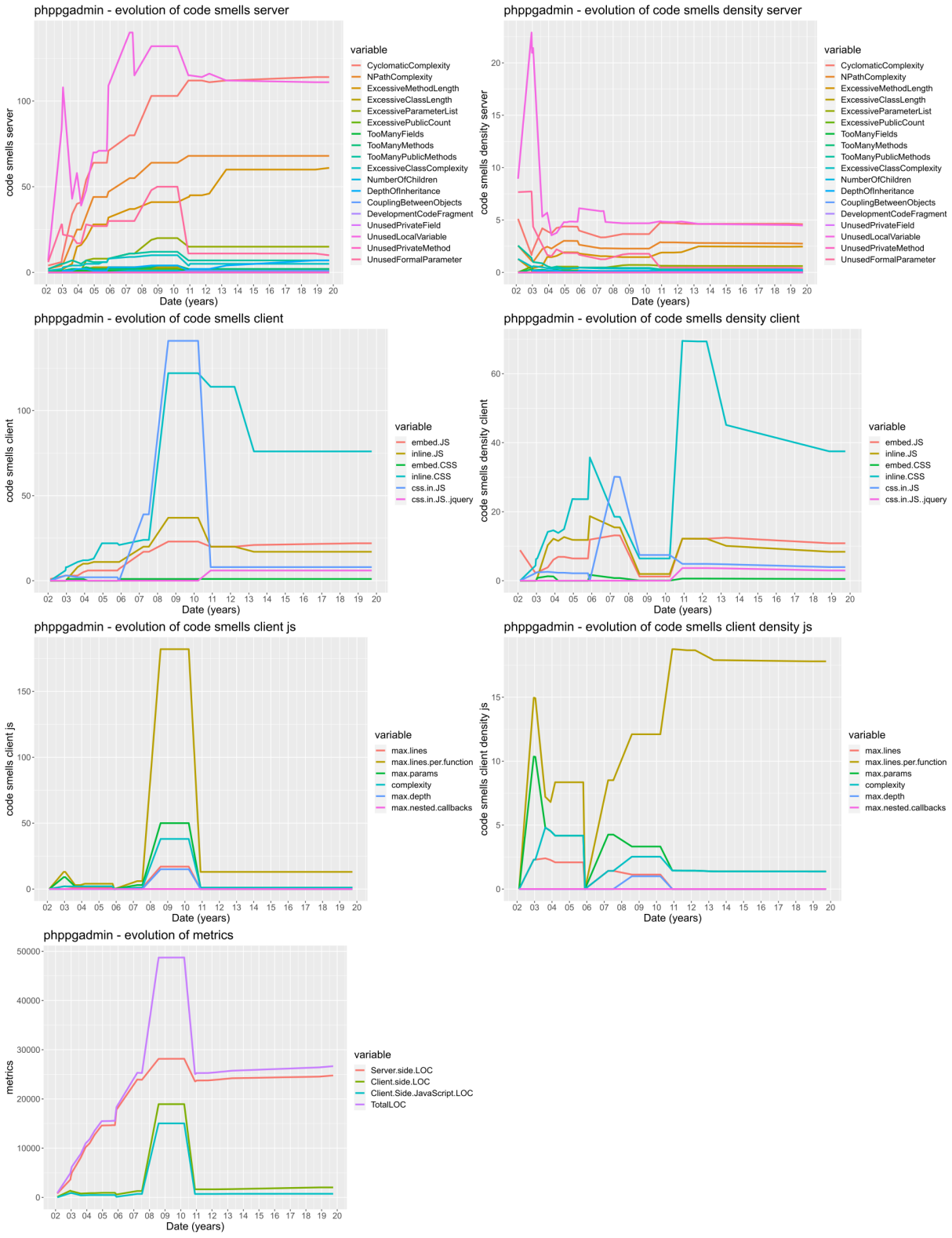


Figure F.5: CS and metrics evolution for phpPgAdmin

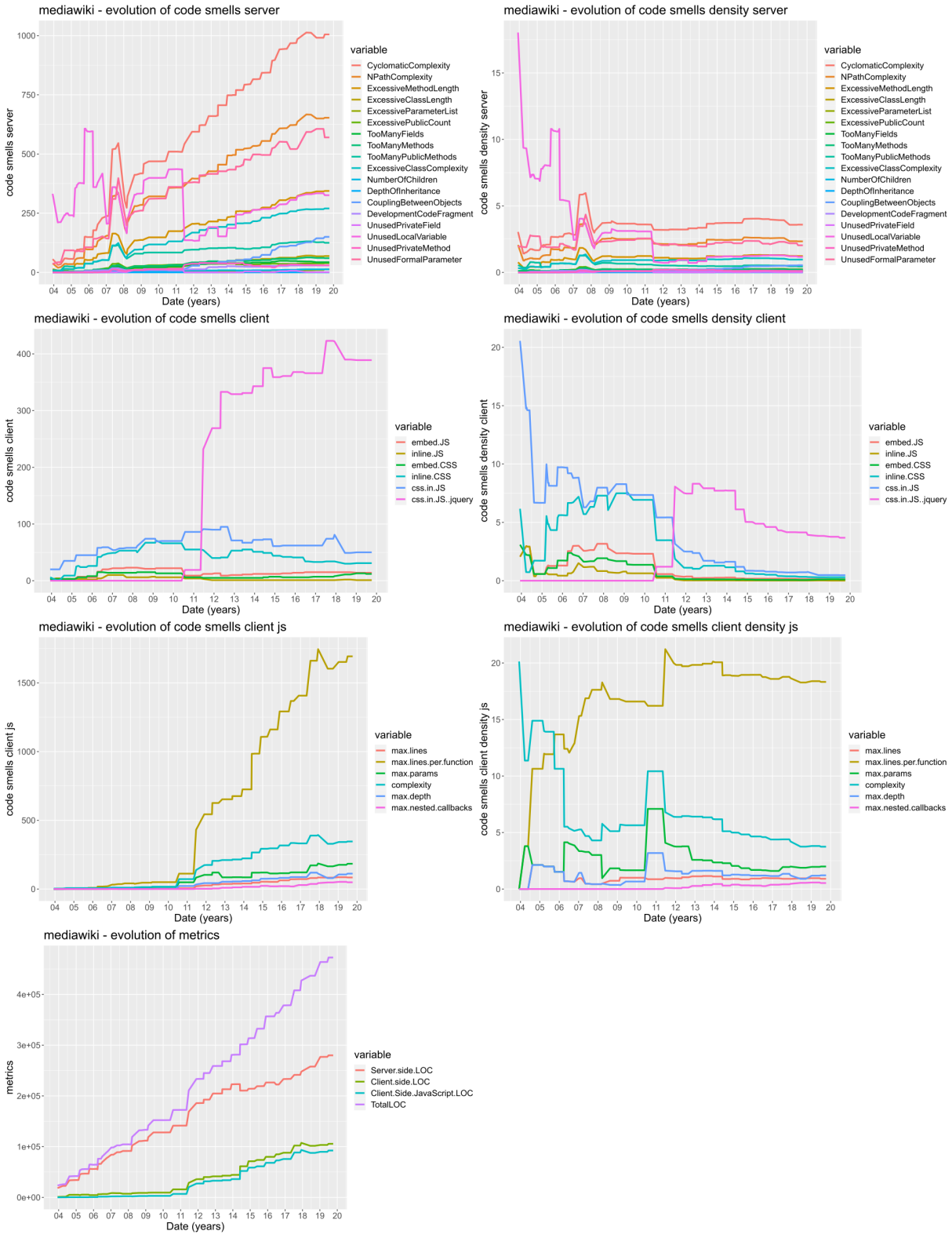


Figure F.6: CS and metrics evolution for MediaWiki

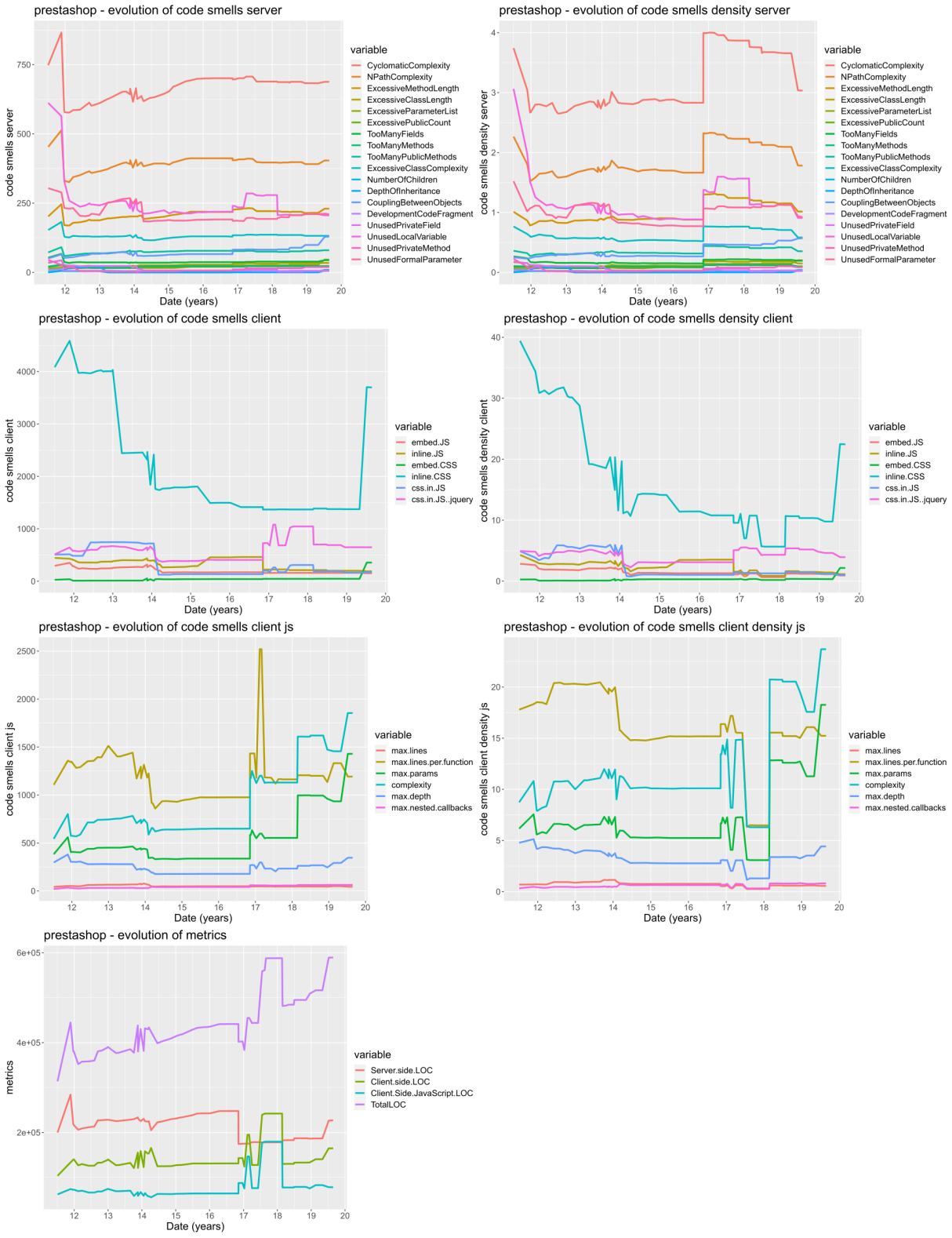


Figure F.7: CS and metrics evolution for PrestaShop

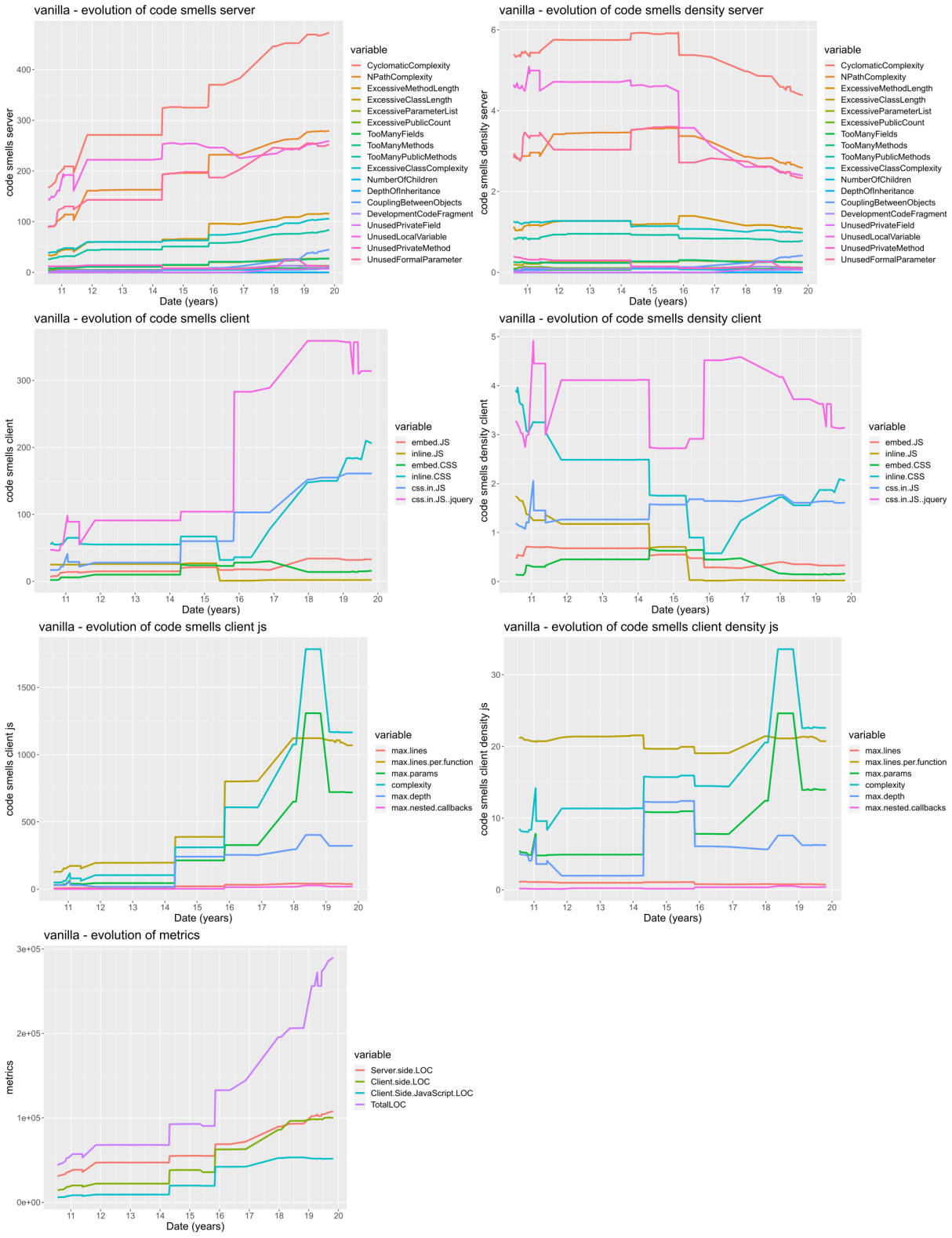


Figure F.8: CS and metrics evolution for Vanilla

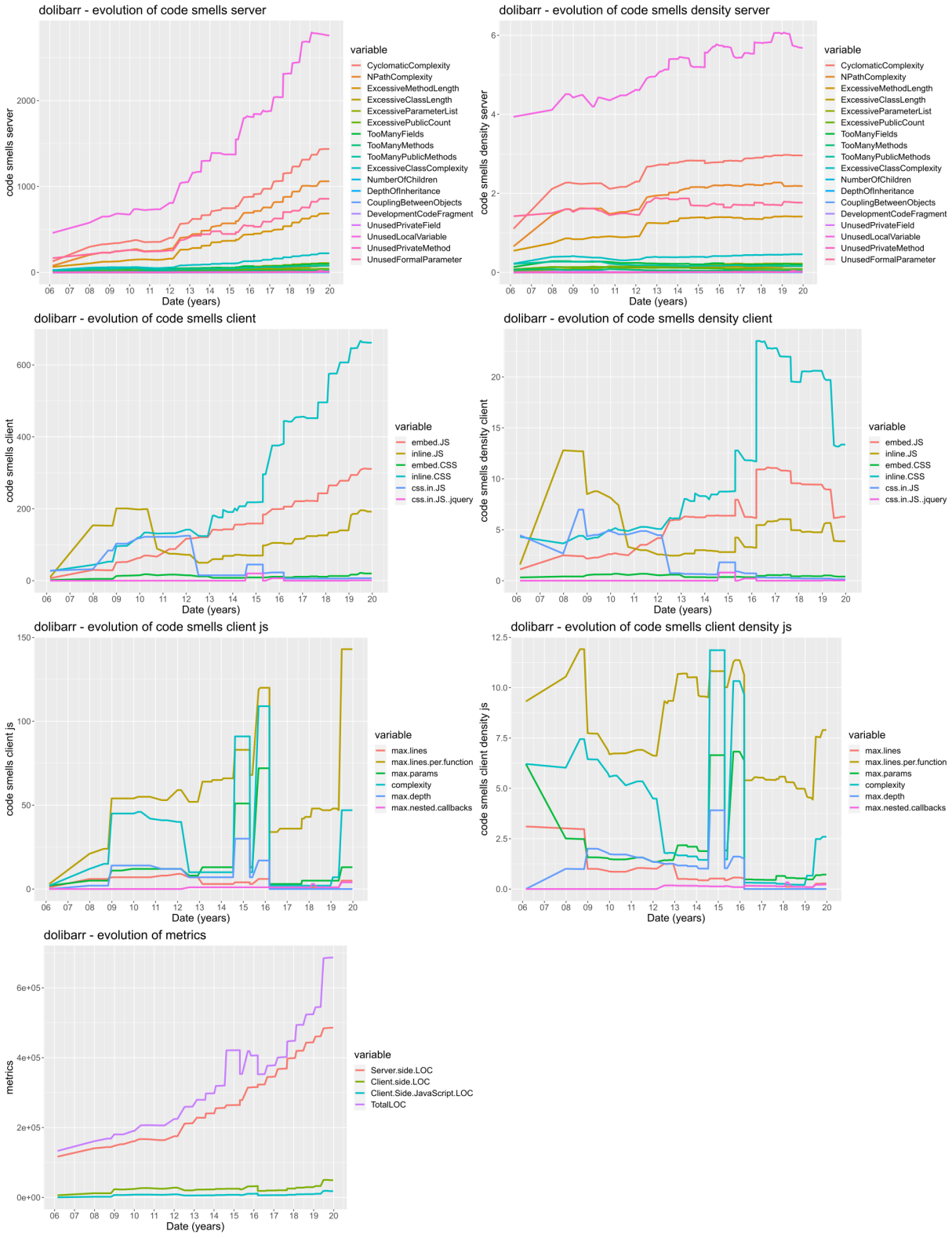


Figure F.9: CS and metrics evolution for Dolibarr

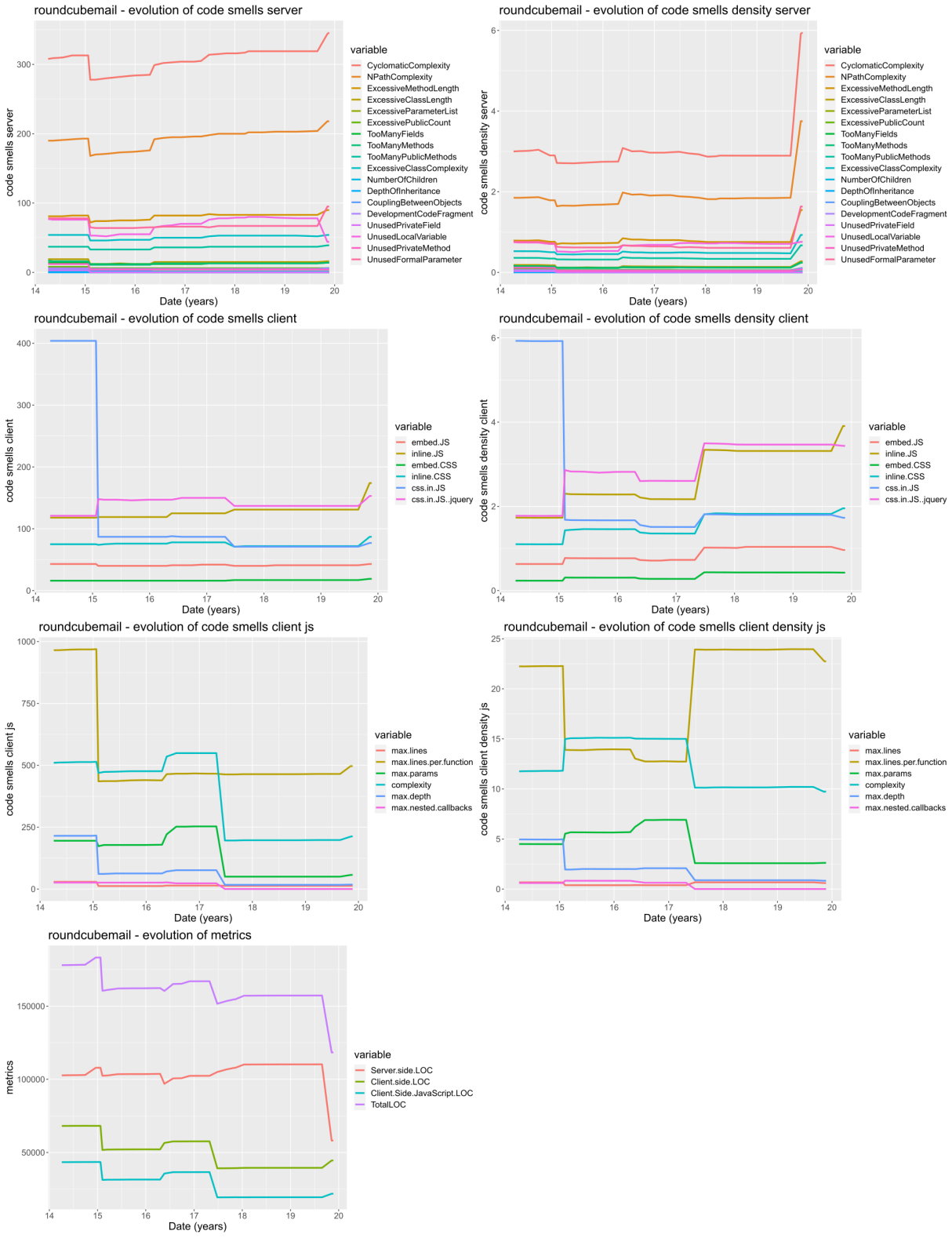


Figure F.10: CS and metrics evolution for Roundcube

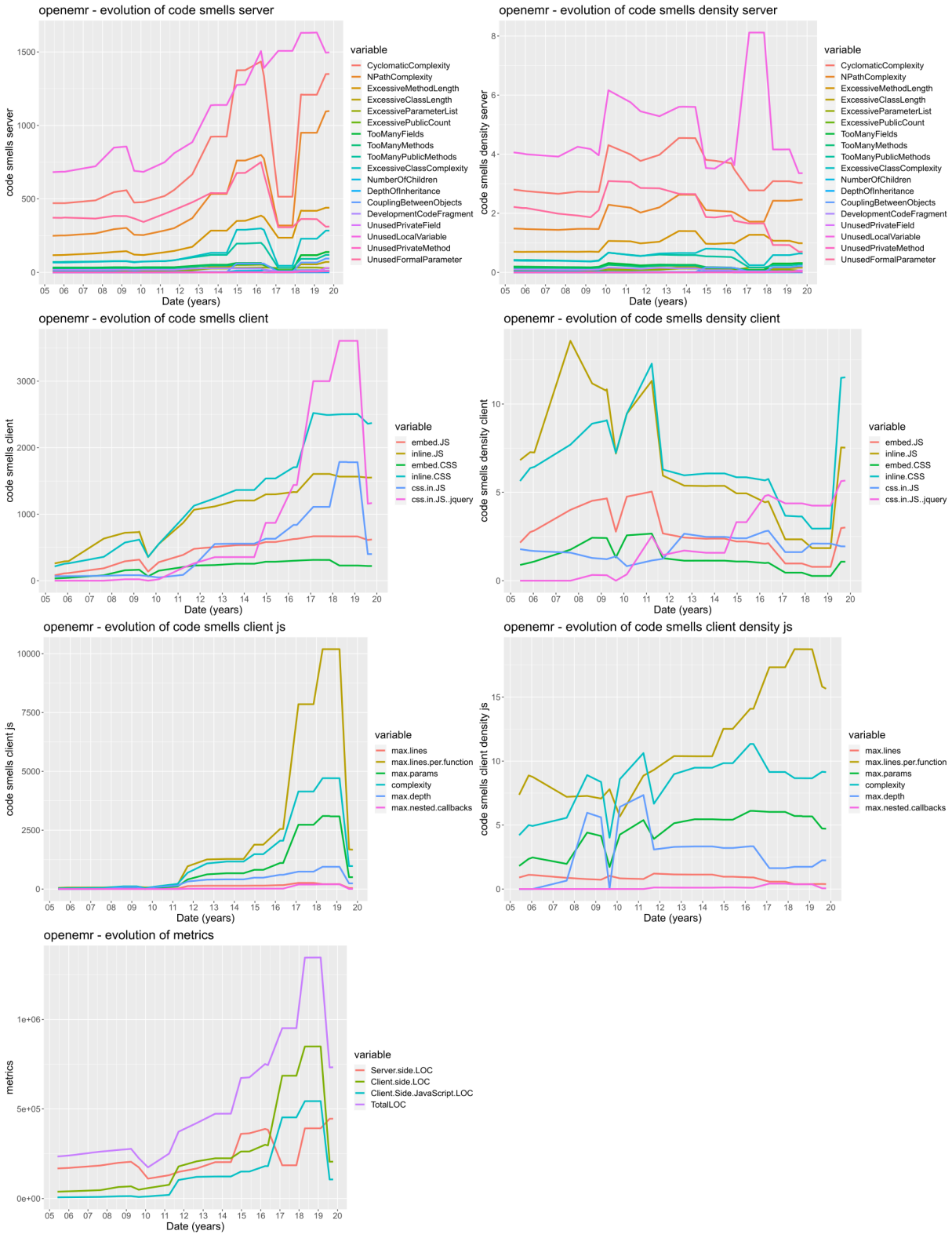


Figure F.11: CS and metrics evolution for OpenEMR

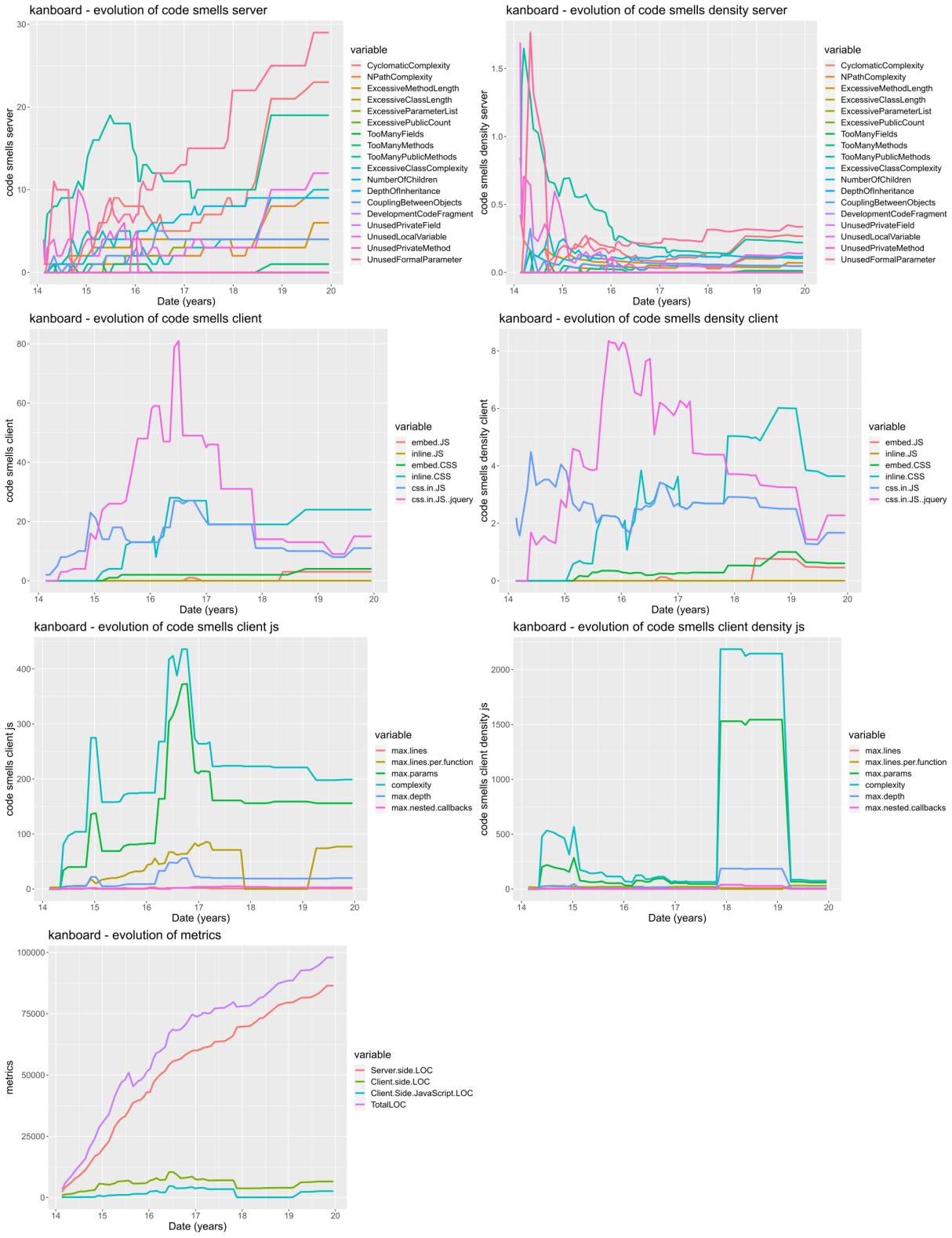
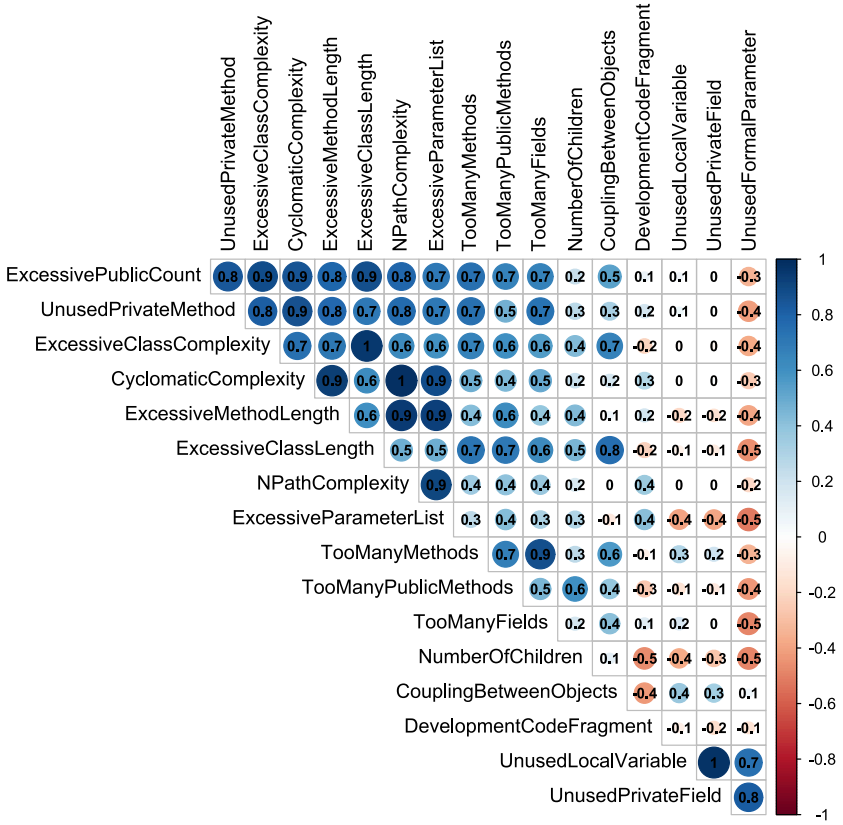
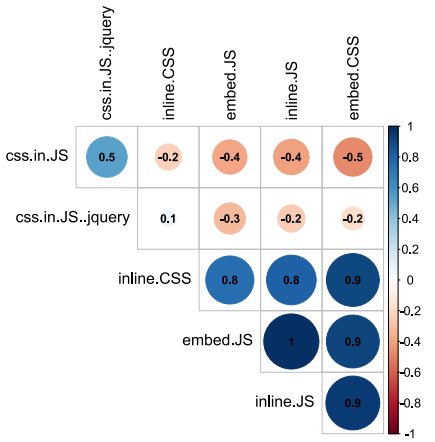


Figure F.12: CS and metrics evolution for Kanboard

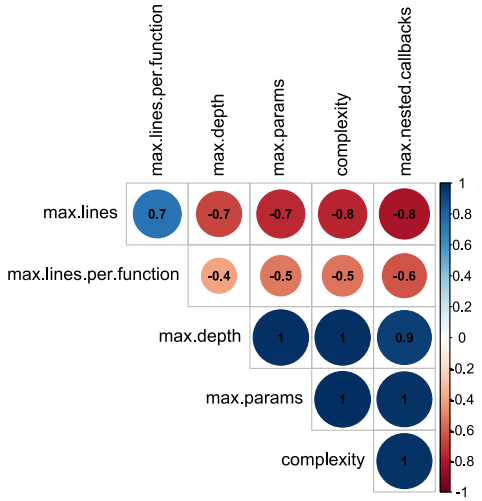
F.1.2 Correlation matrices for CS in the same group



a CS server-side x CS server-side

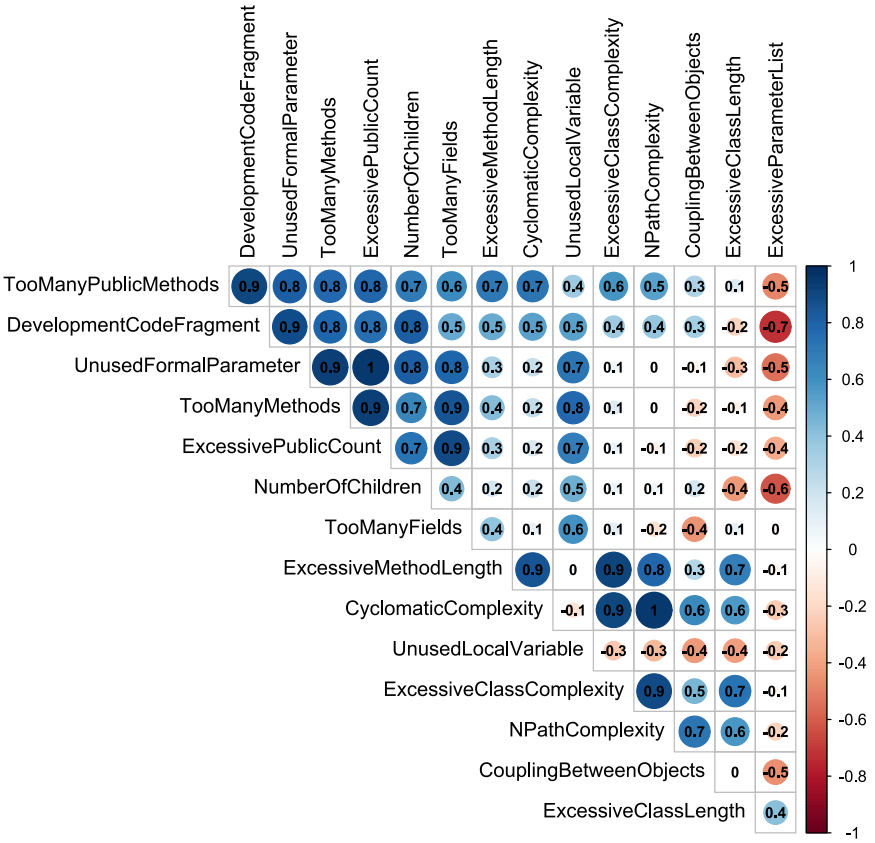


b CS client-side x CS client-side

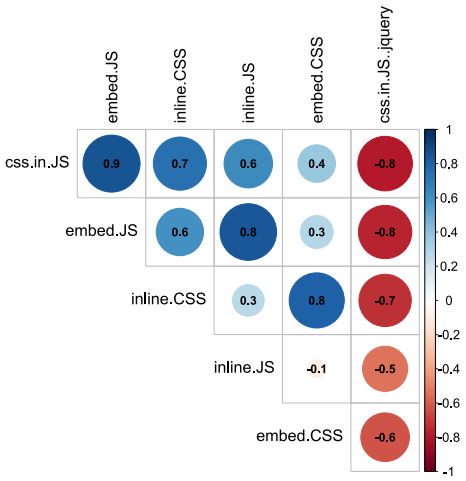


c CS client-side js x CS client-side js

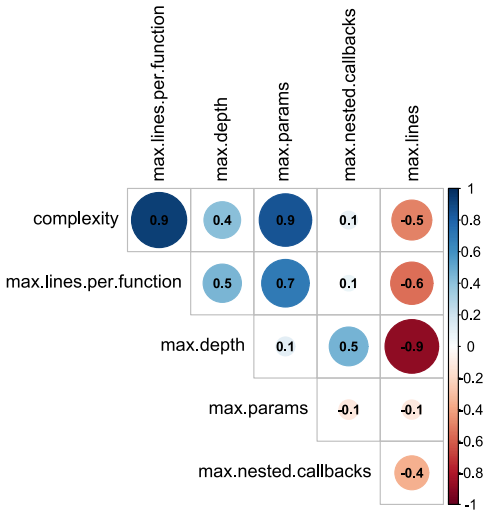
Figure F.13: phpMyAdmin - Correlation matrices for CS in the same group



a CS server-side x CS server-side

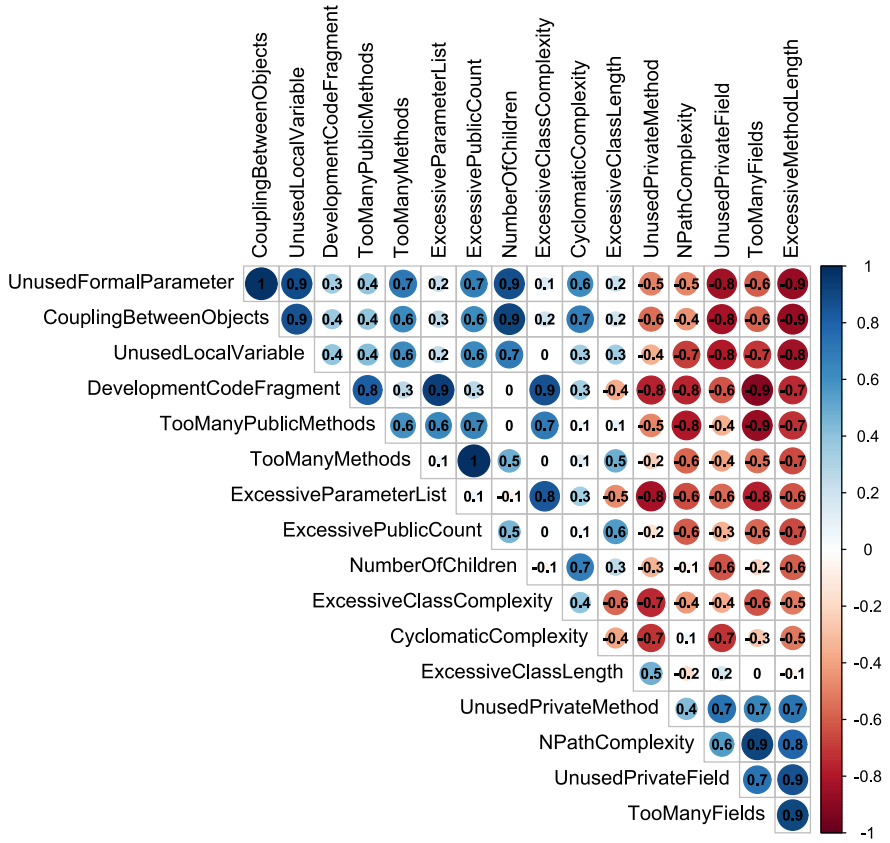


b CS client-side x CS client-side

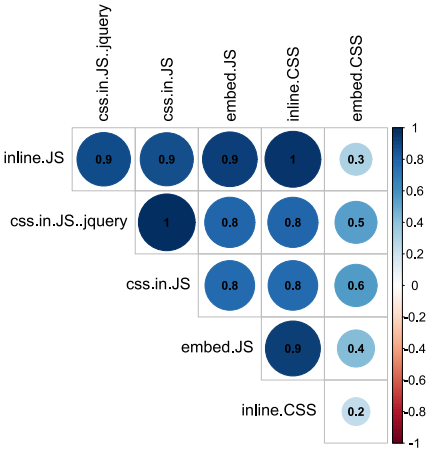


c CS client-side js x CS client-side js

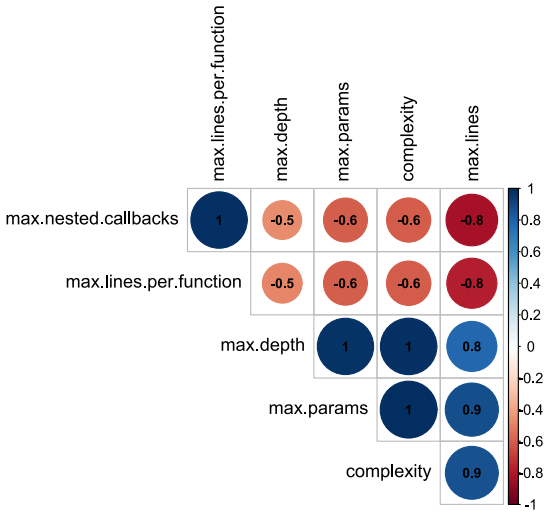
Figure F.14: DokuWiki - Correlation matrices for CS in the same group



a CS server-side x CS server-side

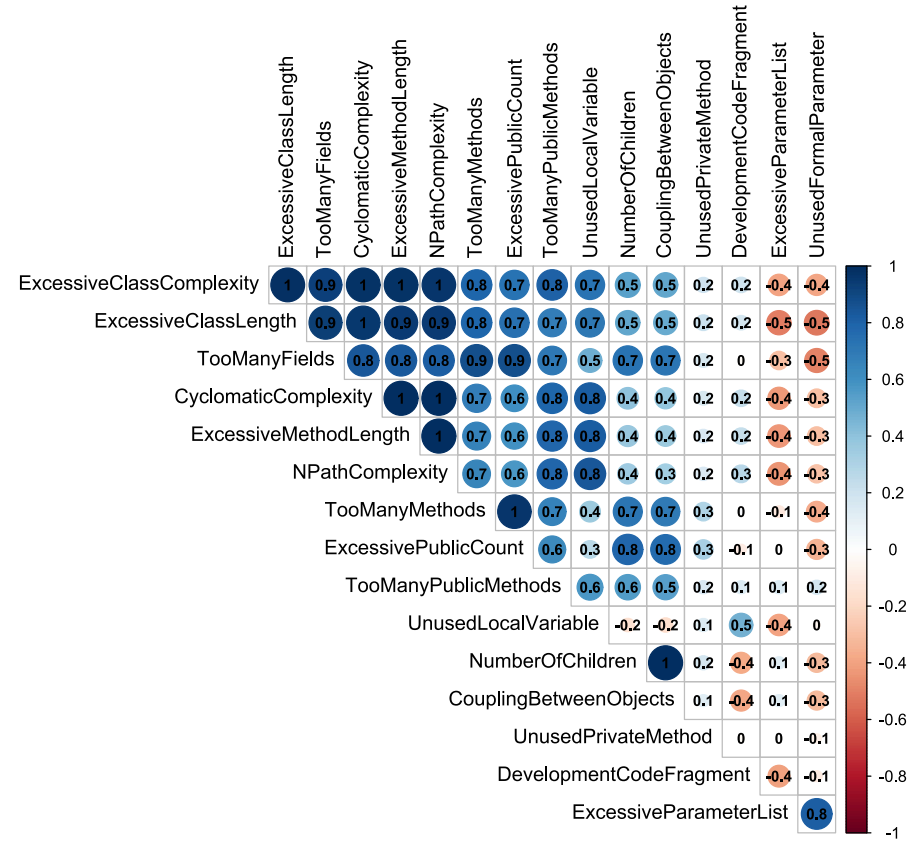


b CS client-side x CS client-side

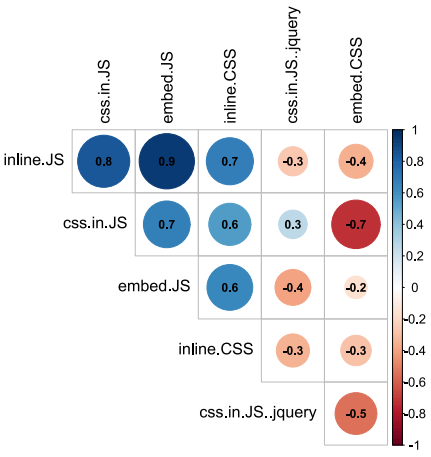


c CS client-side js x CS client-side js

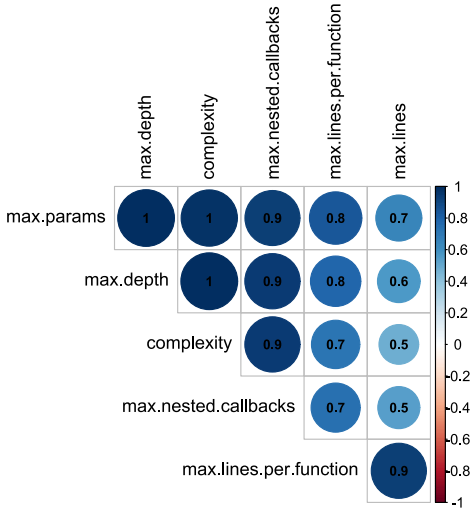
Figure F.15: OpenCart - Correlation matrices for CS in the same group



a CS server-side x CS server-side

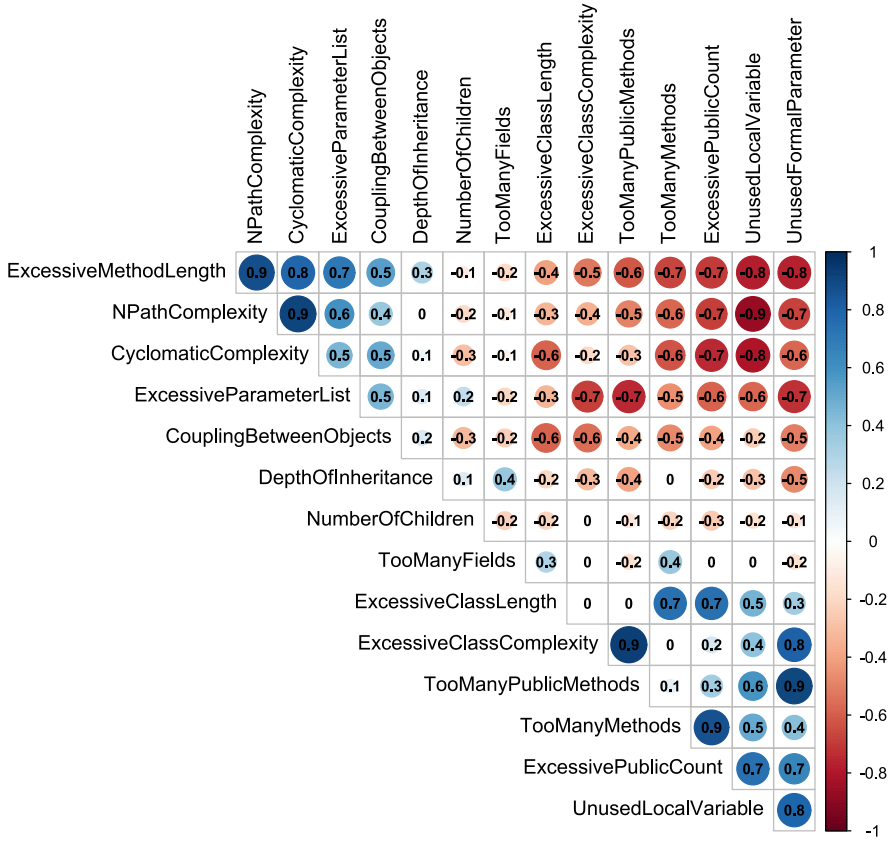


b CS client-side x CS client-side

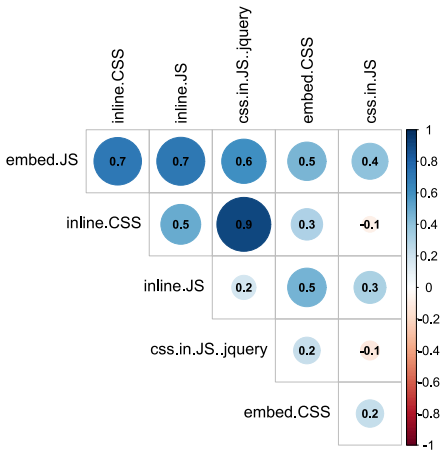


c CS client-side js x CS client-side js

Figure F.16: phpBB CS - Correlation matrices for CS in the same group



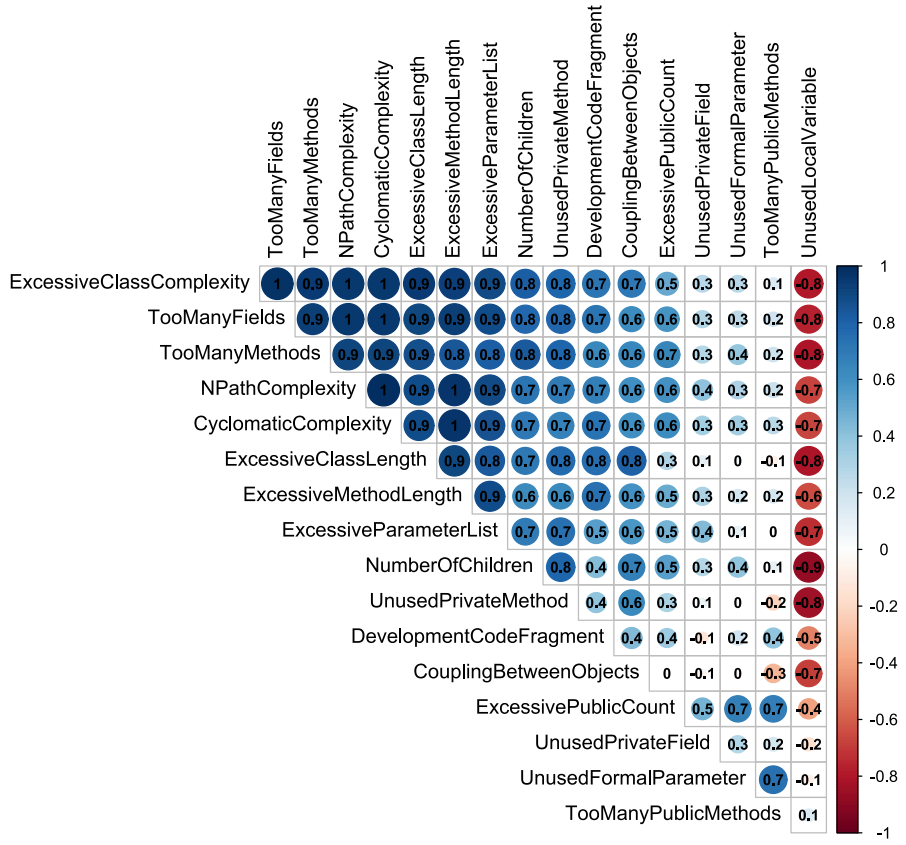
a CS server-side x CS server-side



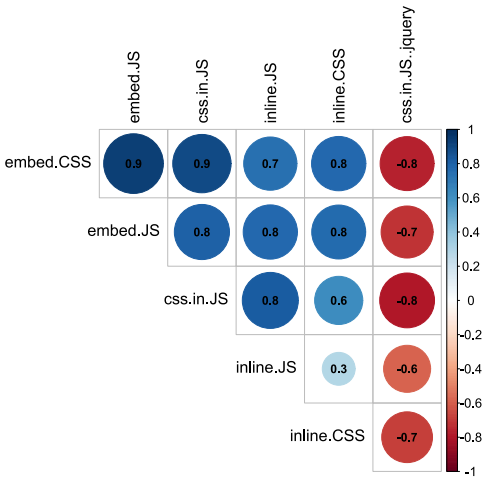
b CS client-side x CS client-side

c CS client-side js x CS client-side js

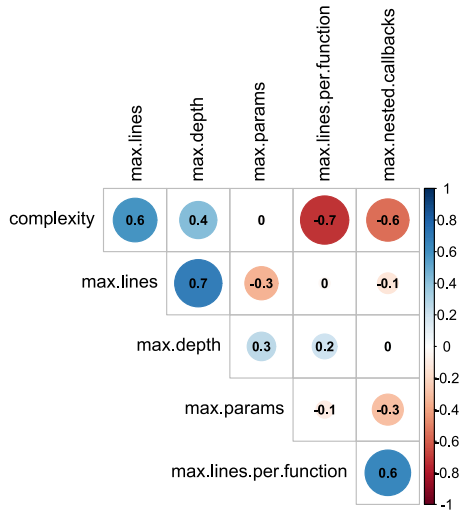
Figure F.17: phpPgAdmin - Correlation matrices for CS in the same group



a CS server-side x CS server-side

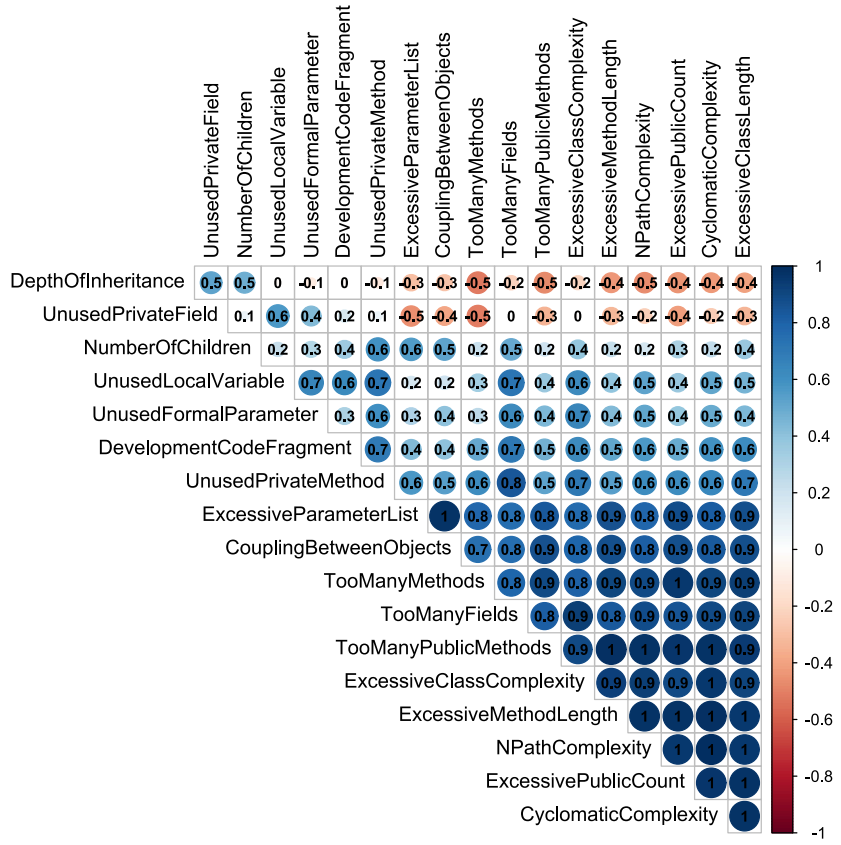


b CS client-side x CS client-side

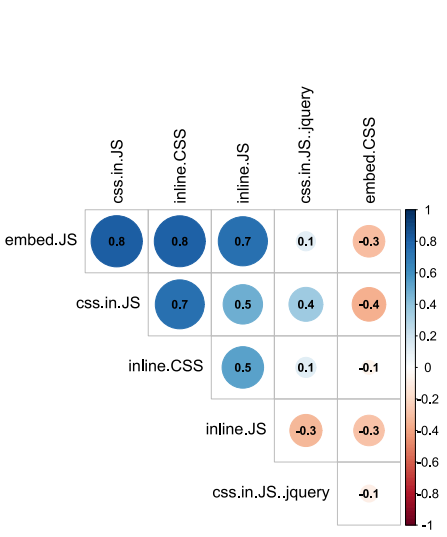


c CS client-side js x CS client-side js

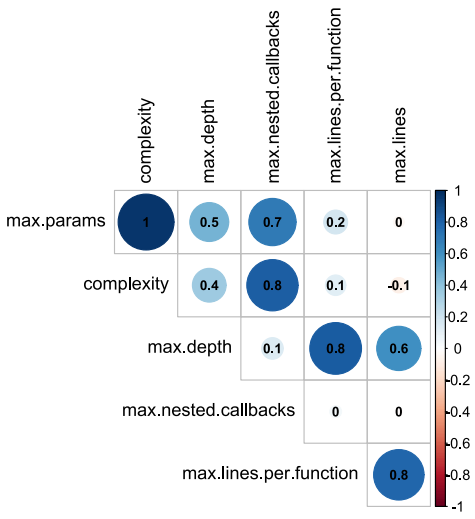
Figure F.18: MediaWiki - Correlation matrices for CS in the same group



a CS server-side x CS server-side

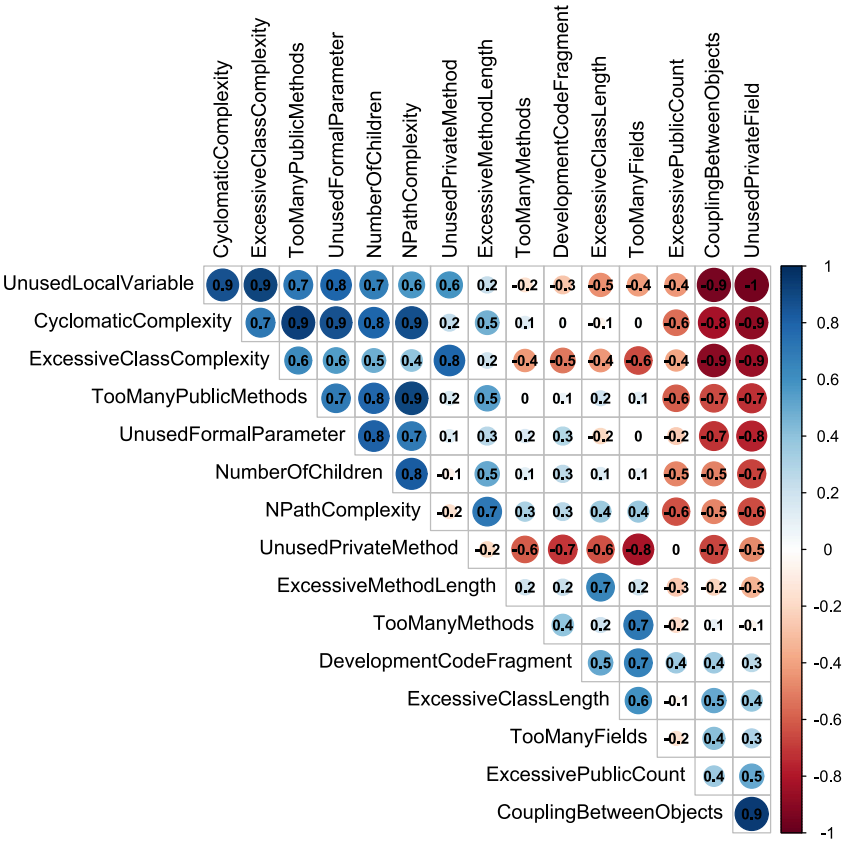


b CS client-side x CS client-side

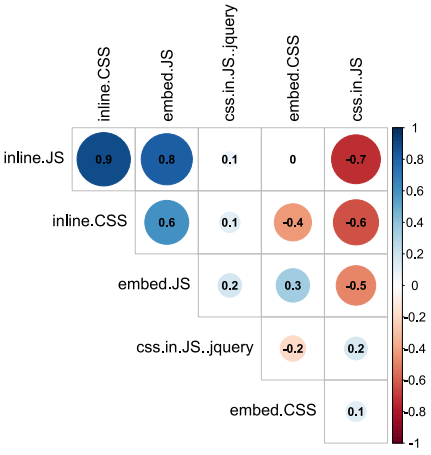


c CS client-side js x CS client-side js

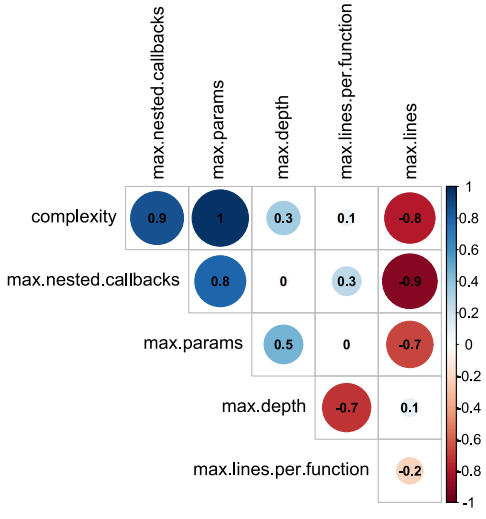
Figure F.19: PrestaShop - Correlation matrices for CS in the same group



a CS server-side x CS server-side

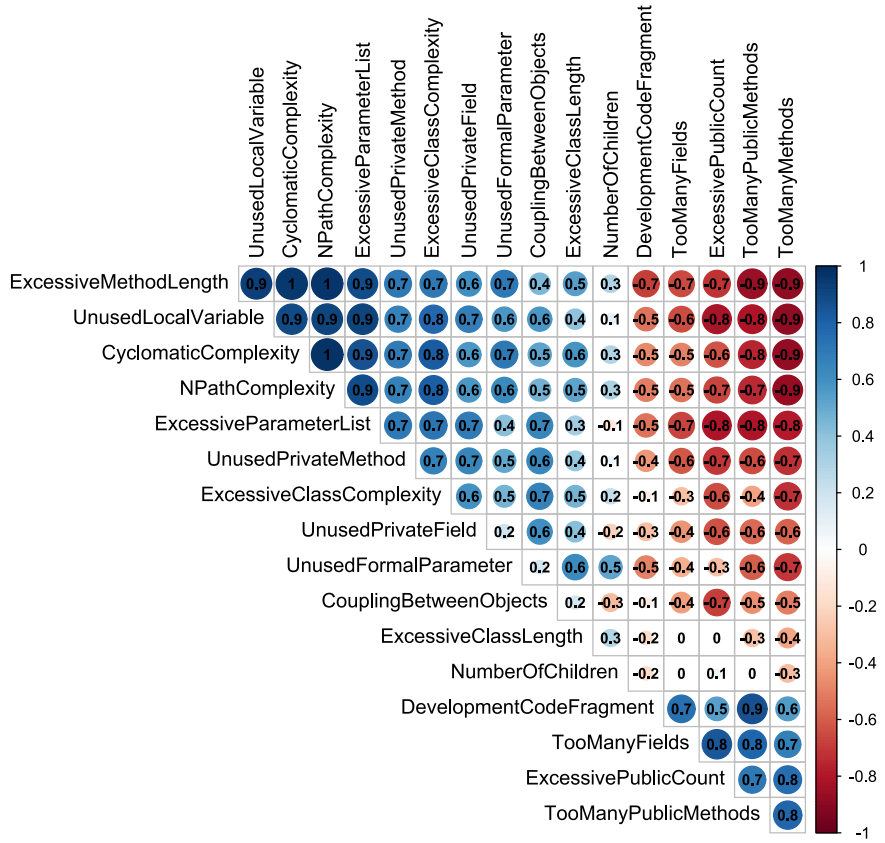


b CS client-side x CS client-side

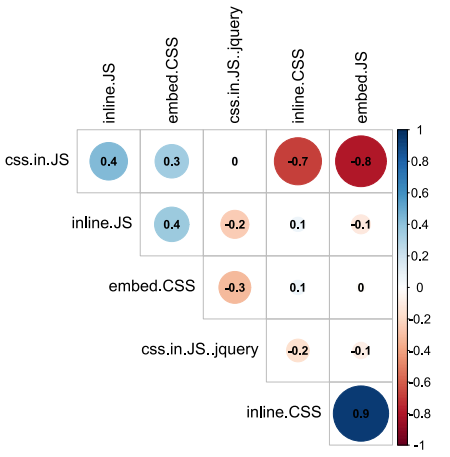


c CS client-side js x CS client-side js

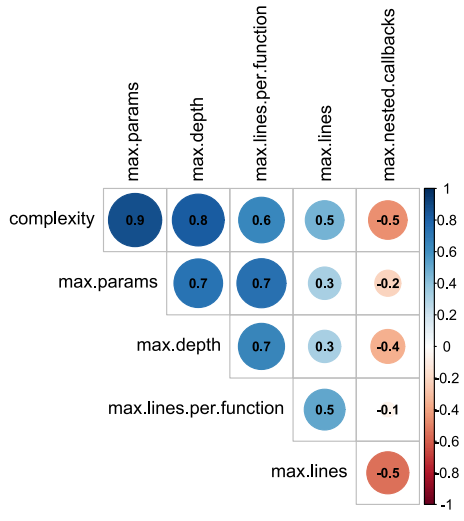
Figure F.20: Vanilla - Correlation matrices for CS in the same groupp



a CS server-side x CS server-side

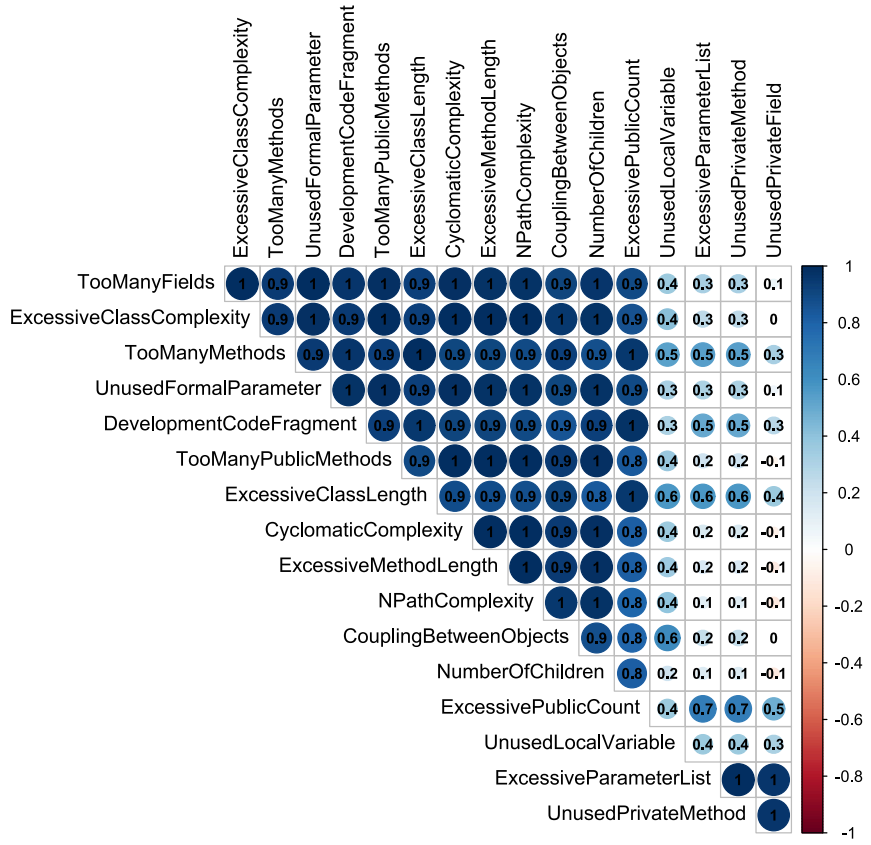


b CS client-side x CS client-side

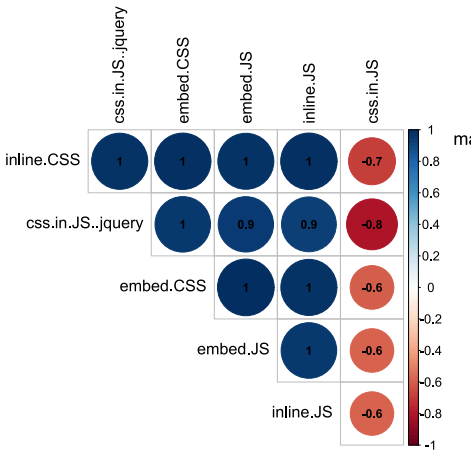


c CS client-side js x CS client-side js

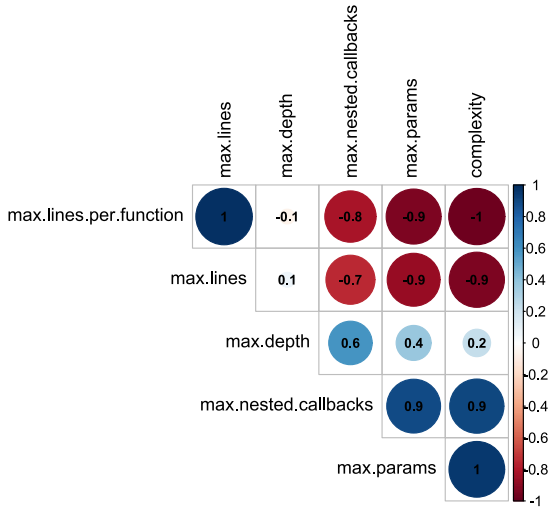
Figure F.21: Dolibarr - Correlation matrices for CS in the same group



a CS server-side x CS server-side

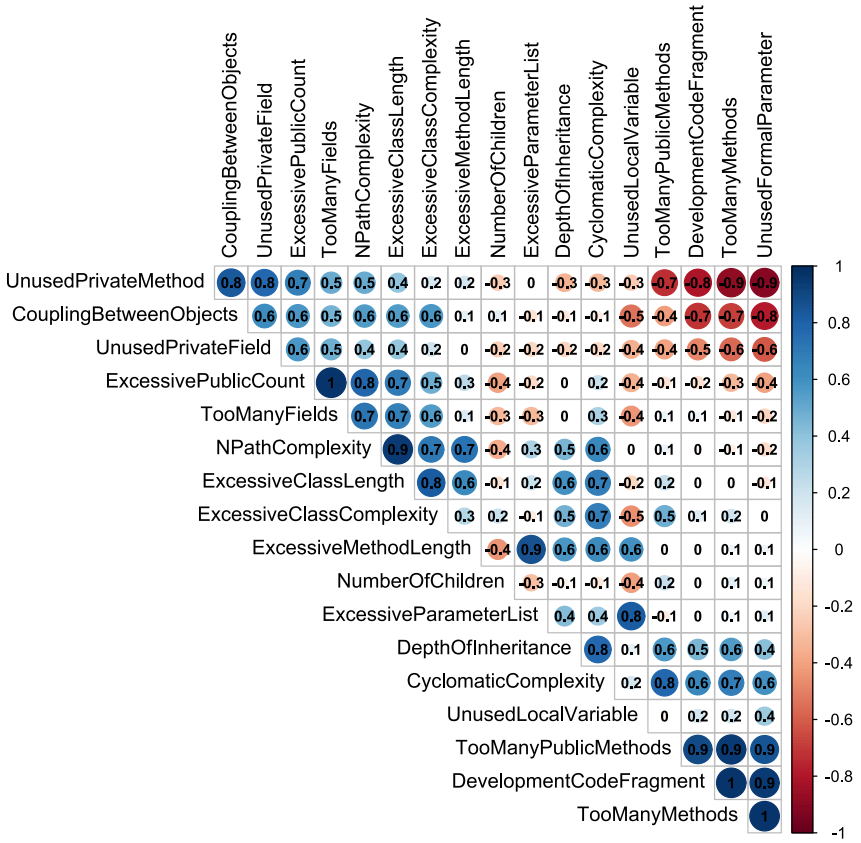


b CS client-side x CS client-side

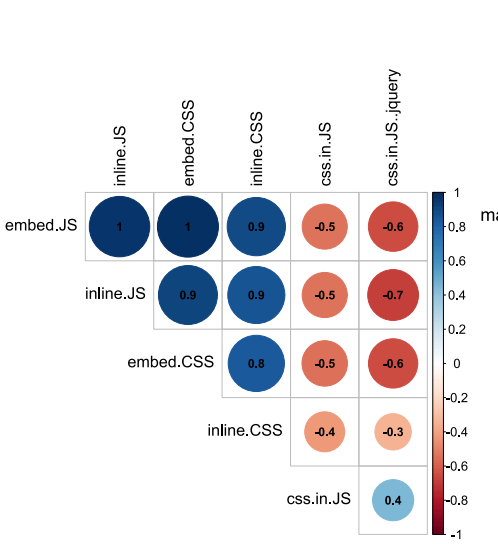


c CS client-side js x CS client-side js

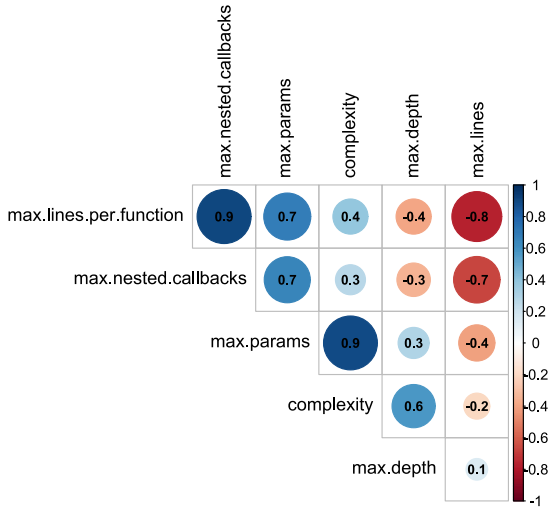
Figure F.22: Roundcube - Correlation matrices for CS in the same group



a CS server-side x CS server-side

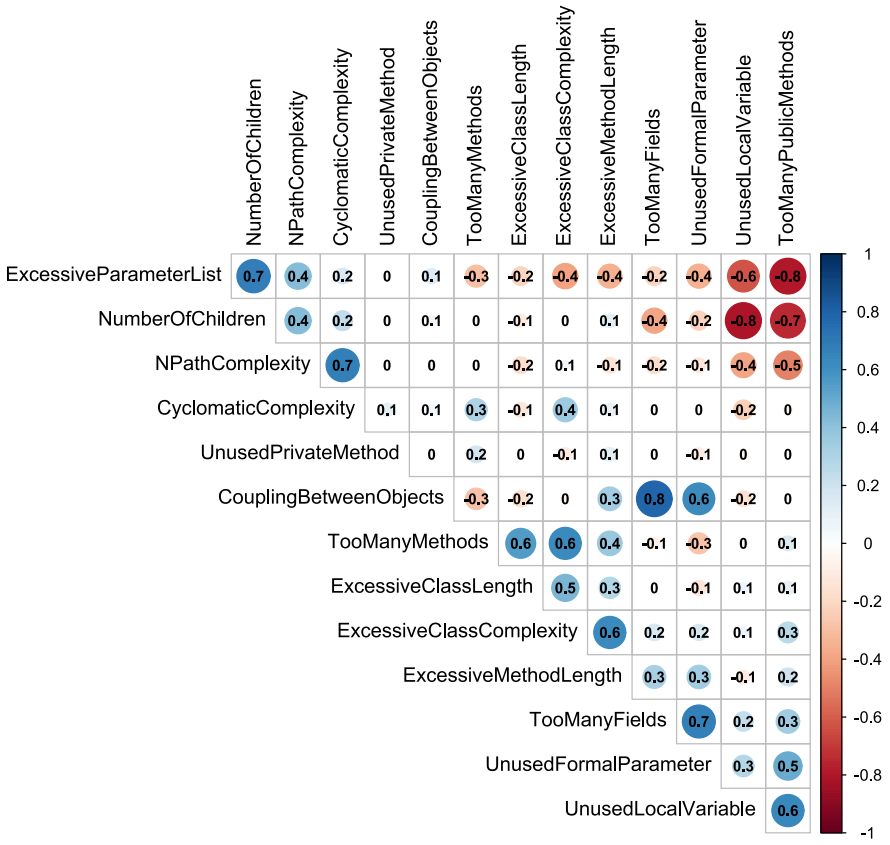


b CS client-side x CS client-side



c CS client-side js x CS client-side js

Figure F.23: OpenEMR - Correlation matrices for CS in the same group



a CS server-side x CS server-side

Figure F.24: Kanboard - Correlation matrices for CS in the same group

F.2 RQ2 - Relationship between server- and client-side CS evolution

F.2.1 Correlation - Tables with more detail in data

Table F.1: Correlation (cor_ts) between CS groups in apps - detail

Correlation	phpMyAdmin	DokuWiki	OpenCart	phpBB	phpPgAdmin	MediaWiki	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
server,client cor	-0.05	0.53	-0.78	0.46	-0.54	0.47	-0.34	0.4	0.57	0.37	-0.03	-0.76
server,client p-value	0.533	0	0	0.001	0.002	0	0.003	0.001	0	0.042	0.887	0
server,client_js cor	0.14	-0.17	-0.79	0.52	0.28	-0.25	0.03	-0.53	-0.42	-0.13	0.2	-0.16
server,client_js p-value	0.071	0.285	0	0	0.142	0.004	0.824	0	0	0.475	0.265	0.201
client,client_js cor	0.35	-0.81	0.95	-0.26	-0.08	-0.53	0.27	-0.76	-0.68	0.26	-0.35	0.34
client,client_js p-value	0	0	0	0.068	0.692	0	0.021	0	0	0.16	0.044	0.006

F.2.2 Linear regression between Code Smell groups

Table F.1 shows the detail of correlation cor_ts, with all applications represented. The line cor represents the value of the correlation, and the following line the p-value. The correlation shows only relations on the same release of the app.

Table F.2: Linear regression (X=>Y) between Code Smell groups. A dot represents statistical significance.

Linear regression	phpMyAdmin	DokuWiki	OpenCart	phpBB	phpPgAdmin	MediaWiki	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
server =>client		•	•	•	•	•	•	•	•	•		•
server =>client_js	○		•	•		•		•	•			
client =>server		•	•	•	•	•	•	•	•	•		•
client =>client_js	•	•	•	○		•	•	•	•		•	•
client =>server	○		•	•		•		•	•			
client_js =>client	•	•	•	○		•	•	•	•		•	•

The table F.2 shows the linear regressions from groups A->B, with all apps represented. We used the "lm" R function to capture regression models with statistical significance, represented with the dot in the table. 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 follow the same pattern in that same release.

In the next table F.3 the first column represents the sum of the apps with statistical significance in the linear regressions among groups (a resume from table F.2). The relations that have

significance in most apps are: Server =>Client and Client =>Server; Client =>Client_js and Client_js =>Client.

F.2.3 Linear regression, Granger causality, Transfer Entropy up to lag 4

Table F.3: Statistical causality from between CS groups (12 apps) using Linear regression, Granger-causality (lag 1-4) and Transfer Entropy (lag 1-4).

rel	LM	GC1	GC2	GC3	GC4	TE1	TE2	TE3	TE4
Server =>Client	●●●●●●●●●●	○	●○	●	●	●●○○○	●○	●	●
Server =>client_js	●●●●●○	●		○	●	●●	●	●	●
Client =>Server	●●●●●●●●●●	●●●	●●○	●●●	●●	●●●●●○○	●●●●	●●	●●●
Client =>Client_js	●●●●●●●●●○	●●●○	●●○	●	●○	●●●●○	●●●●	●●○	●●●
Client =>Server	●●●●●○	●○	●	●	●	●●●●●○	●●	●●	●●
Client_js =>Client	●●●●●●●●●○	●●○○○	●●○	●●	●●●○	●●●○○	●●	●●	●●

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

F.3 RQ3 - Impact of server- and client-side CS intensity evolution on web app' reported issues

F.3.1 Time-series correlation (cor_ts) between CS and issues per app

Table F.4: Resume of time-series correlation (cor_ts) between CS and issues, per app

cs type	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
server	8	7	1	0	12	15	17	10	12	0
client	4	2	0	1	1	5	4	4	5	2
client_js	6	4	0	0	4	6	1	0	6	1
total	18	13	1	1	17	26	22	14	23	3

Number indicates the number of positive correlations greater than 0.3 between CS and issues.

Table F.4 resumes the correlations by application. Seven of the ten applications have many timeseries correlations between CS and issues. However, three of the applications have almost no correlation at all (*OpenCart* and *phpBB* with just one, and *Kanboard* with only three). These applications behave differently than the others regarding the issues.

F.3.2 Causality relationships between CS and issues - up to lag 4

Reproduction of the extended tables in the chapter 6, but with more lags, both for Granger-causality and Tansfer Entropy.

F.3.2.1 Causality relationships between CS and issues density

Table F.5: Statistical causality from CS density to Issues density relations (10 apps) using Linear regression, Granger-causality (lag 1-4) and Transfer Entropy (lag 1-4).

CS	LM	GC1	GC2	GC3	GC4	TE1	TE2	TE3	TE4
CyclomaticComplexity	•••○○	•	•••	•••	••	•••○○	••••○	•	••
NPathComplexity	••○	•○○	•	••	○	•••○	•••	••	••○
ExcessiveMethodLength	•••○	•••	•••	••○	••	••••	•○○	••○	••
ExcessiveClassLength	•••	••○	•○	•○	••	••••○	••		
ExcessiveParameterList	•○	••○○	•○○○	•••	••○	••○○	••○		
ExcessivePublicCount	•			○		•○○	•○○		
TooManyFields	••○	••○	••○	•○	○	••○	•○○		
TooManyMethods	•○	•	••		○	•••○	••○○	•	•
TooManyPublicMethods	•••○○	•••	•○	•○	•	•••	•••	••	•
ExcessiveClassComplexity	•••○	••	••○○	•○	•○	••○○	••••○○	••	••
NumberOfChildren	•••	•••○	••	•	•○	•••○	•••		
DepthOfInheritance							○		
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	••••	••○	•○	•○	○	•••••••○○	•••••	•••	••
complexity	•••○	•••	•	••○	•○	•••○○	•••••	•○	••
max.depth	••••		•	•••	•••	••••○○○	•••••	•••••	••••
max.nested.callbacks	••••	•	•	••○	••	•••○○	•••	•	•

Columns: Linear regression(LM), Granger-causality (lag1 to 4), and Transfer Entropy (lag 1 to 4). The first 18 CS are from the server-side/PHP, the following 6 are the client-side embed CS, and the last 6 are the client-side JavaScript CS. Black dots and white dots represent statistical significance of 0.05 and 0.10, respectively, in one application.

F.3.2.2 Causality relationships between CS and issues absolute number

Table F.6: Statistical causality from CS density to Issues relations (10 apps) using Linear regression, Granger-causality (lag 1-4) and Transfer Entropy (lag 1-4).

CS	LM	GC1	GC2	GC3	GC4	TE1	TE2	TE3	TE4
(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	••○	•	•	•••	•••	•••○	•••	•	•

Columns: Linear regression(LM), Granger-causality (lag1 to 4), and Transfer Entropy (lag 1 to 4). The first 18 CS are from the server-side/PHP, the following 6 are the client-side embed CS, and the last 6 are the client-side JavaScript CS. Black dots and white dots represent statistical significance of 0.05 and 0.10, respectively, in one application.

F.4 RQ4 - Impact of CS intensity evolution on a web app's bugs

F.4.1 Time-series correlation (cor_ts) between CS and bugs per app

Table F.7: Resume of time-series correlation (cor_ts) between CS and bugs, per app

	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
server	9	9	1	0	7	16	14	11	2	1
client	4	2	0	1	0	4	3	4	4	2
client_js	6	3	0	0	4	6	0	0	5	4
total	19	14	1	1	11	26	17	15	11	7

Number indicates the number of positive correlations greater than 0.3 between CS and bugs

Table F.7 shows the correlations divided by app and by type of code smell (server, client, and client_js). The table shows that both the server and client CS are related to the bugs, except for two applications. These correlations can be not linear because cor_ts can measure non-linear correlations between 2 irregular time series with different granularity's/number of observations (check figure 6.5). This correlation only measures similarity between time series. For causality, we have to do some other studies that we show in the next section.

F.4.2 Causality relationships between CS and bugs - up to lag 4

F.4.2.1 Causality relationships between CS and bugs absolute number

Table F.8: Statistical causality from CS density to Bugs relations (10 apps) using Linear regression, Granger-causality (lag 1-4) and Transfer Entropy (lag 1-4).

CS	LM	GC1	GC2	GC3	GC4	TE1	TE2	TE3	TE4
(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	●●	●●○	●	●●●○	●●●	●●●○○	●●	●	●

Columns: Linear regression(LM), Granger-causality (lag1 to 4), and Transfer Entropy (lag 1 to 4). The first 18 CS are from the server-side/PHP, the following 6 are the client-side embed CS, and the last 6 are the client-side JavaScript CS. Black dots and white dots represent statistical significance of 0.05 and 0.10, respectively, in one application.

F.4.2.2 Causality relationships between CS and bugs density

Table F.9: Statistical causality from CS density to Bugs density relations (10 apps) using Linear regression, Granger-causality (lag 1-4) and Transfer Entropy (lag 1-4).

CS	LM n	GC1	GC2	GC3	GC4	TE1	TE2	TE3	TE4
(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	•	••	•	•••	••	••••○○	•••	•○	•

Columns: Linear regression(LM), Granger-causality (lag1 to 4), and Transfer Entropy (lag 1 to 4). The first 18 CS are from the server-side/PHP, the following 6 are the client-side embed CS, and the last 6 are the client-side JavaScript CS. Black dots and white dots represent statistical significance of 0.05 and 0.10, respectively, in one application.

F.5 RQ5 - Causality relationships between CS and time to release

Next table shows values with lags up to 4.

Table F.10: Statistical causality from CS density to Time To Release relations (12 apps) using Linear regression, Granger-causality (lag 1-4) and Transfer Entropy (lag 1-4).

CS	LM	CG1	CG2	CG3	CG4	TE1	TE2	TE3	TE4
(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	●●●○○	●●●○○	●●●○○	●●●●●	●●●●	●●●●●●●	●●●●	●	●

Columns: Linear regression(LM), Granger-causality (lag1 to 4), and Transfer Entropy (lag 1 to 4). The first 18 CS are from the server-side/PHP, the following 6 are the client-side embed CS, and the last 6 are the client-side JavaScript CS. Black dots and white dots represent statistical significance of 0.05 and 0.10, respectively, in one application.

[This page has been intentionally left blank]

Appendix G.

Causal Inference Extended Data II - Tables with values tables (Chapter 6)

This appendix presents the tables with the full values. In the chapter 6 they are represented by dots to resume and due to the lack of space.

G.1 RQ1 - Server- and client-side Code Smell evolution

All values are in the chapter 6 already.

G.2 RQ2 – Relationship between server- and client-side CS evolution

G.2.1 Correlation between Code Smell groups

Values in Appendix 6 - reproduced again on next table:

Table G.1: Correlation (cor_ts) between CS groups - all apps

Correlation	phpMyAdmin	DokuWiki	OpenCart	phpBB	phpPgAdmin	MediaWiki	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
server,client	-0.05	0.53	-0.78	0.46	-0.54	0.47	-0.34	0.4	0.57	0.37	-0.03	-0.76
server,client_js	0.14	-0.17	-0.79	0.52	0.28	-0.25	0.03	-0.53	-0.42	-0.13	0.2	-0.16
client,client_js	0.35	-0.81	0.95	-0.26	-0.08	-0.53	0.27	-0.76	-0.68	0.26	-0.35	0.34

G.2.2 Causal inference between Code Smell groups

G.2.2.1 Linear Regression - all apps

Table G.2: Linear Regression between CS groups only - all apps

p-val CS only	phpMyAdmin	DokuWiki	OpenCart	phpBB	phpPgAdmin	MediaWiki	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
server,client	0.53	0	0	0	0	0	0	0	0	0.04	0.89	0
server,client_js	0.07	0.29	0	0	0.14	0	0.82	0	0	0.47	0.27	0.2
client,server	0.53	0	0	0	0	0	0	0	0	0.04	0.89	0
client,client_js	0	0	0	0.07	0.69	0	0.02	0	0	0.16	0.04	0.01
client,server	0.07	0.29	0	0	0.14	0	0.82	0	0	0.47	0.27	0.2
client_js,client	0	0	0	0.07	0.69	0	0.02	0	0	0.16	0.04	0.01

G.2.2.2 Granger-causality

Table G.3: Granger causality between CS grups only, up to lag 4 - p-values for all apps

Lag	Granger Causality	phpMyAdmin	DokuWiki	OpenCart	phpBB	phpPgAdmin	MediaWiki	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
Lag 1	Server =>Client	0.19	0.09	0.98	0.63	0.45	0.9	0.41	0.76	0.7	0.71	0.47	0.58
	Server =>client_js	0.66	0.76	0.92	0.18	0.01	0.35	0.6	0.22	0.49	0.96	0.55	0.84
	Client =>Server	0.96	0.3	0.78	0.42	0.66	0.71	0	0.05	0.59	0.45	0.86	0
	Client =>Client_js	0.03	0.04	0.46	0.26	0.36	0.98	0.72	0.14	0.05	0.79	0.09	0.83
	Client =>Server	0.87	0.55	0.36	0.52	0.02	0.78	0.68	0.1	0.51	0.48	0.16	0.97
	Client_js =>Client	0	0.91	0.65	0.08	0.84	0.07	0.9	0.13	0.1	0.26	0.02	0.5
Lag 2	Server =>Client	0.4	0.34	0.92	0.88	0.64	0.21	0.93	0.95	0.25	0.8	0.09	0.01
	Server =>client_js	0.89	0.95	0.94	0.38	0.53	0.71	0.93	0.41	0.47	0.63	0.19	0.75
	Client =>Server	1	0.61	0.79	0.68	0.96	0.16	0.28	0.02	0.94	0.66	0.09	0
	Client =>Client_js	0.04	0.07	0.62	0.51	0.95	0.8	0.73	0.23	0.04	0.97	0.29	0.88
	Client =>Server	0.99	0.84	0.75	0.8	0.28	0.56	0.88	0.21	0.87	0.87	0.03	0.99
	Client_js =>Client	0	0.92	0.86	0.18	0.88	0.1	0.24	0.29	0.18	0.51	0.02	0.65
Lag 3	Server =>Client	0.61	0.45	0.99	0.98	0.72	0.33	0.69	1	0.41	0.93	0.16	0.05
	Server =>client_js	0.96	0.84	0.98	0.53	0.98	0.82	0.93	0.06	0.95	0.61	0.47	0.76
	Client =>Server	1	0.62	0.95	0.8	0.76	0.25	0.65	0.02	0.96	0.79	0.03	0.01
	Client =>Client_js	0.04	0.28	0.99	0.66	0.92	0.93	0.88	0.22	0.15	1	0.79	0.93
	Client =>Server	1	0.7	0.81	0.91	0.15	0.69	0.95	0.26	0.77	0.97	0.05	0.98
	Client_js =>Client	0	0.83	0.93	0.3	0.98	0.17	0.26	0.54	0.2	0.68	0.02	0.84
Lag 4	Server =>Client	0.74	0.61	1	0.99	0.86	0.25	0.82	0.98	0.17	0.95	0.39	0.05
	Server =>client_js	0.99	0.94	1	0.65	0.63	0.77	0.96	0.01	0.92	0.79	0.31	0.84
	Client =>Server	1	0.62	0.96	0.85	0.48	0.7	0.79	0.02	0.99	0.89	0.05	0.11
	Client =>Client_js	0.02	0.4	0.92	0.78	0.85	0.97	0.95	0.15	0.18	1	0.1	0.8
	Client =>Server	1	0.72	0.95	0.92	0.55	0.98	0.99	0.26	0.9	0.99	0.05	0.96
	Client_js =>Client	0	0.01	0.95	0.42	0.84	0.19	0.47	0.7	0.09	0.77	0.05	0.88

G.2.2.3 Transfer entropy

Table G.4: Transfer entropy between CS groups only - p values - all apps

Lag	TE p-value	phpMyAdmin	DokuWiki	OpenCart	phpBB	phpPgAdmin	MediaWiki	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
Lag 1	Server =>Client	0.01	0.62	0.06	0.6	0.08	0.5	0.02	0.49	0.06	0.39	0.62	0.58
	Server =>client_js	0.66	0.53	0.03	0.29	0.47	0.22	0.48	0.35	0.33	0.46	0.36	0
	Client =>Server	0	0	0.12	0.01	0.18	0	0.08	0	0.67	0.53	0.35	0.06
	Client =>Client_js	0.11	0	0.12	0.05	0.09	0.18	0.81	0.05	0.58	0.5	0.13	0
	Client =>Server	0	0.41	0	0.46	0.02	0.4	0.02	0.01	0.6	0.12	0.26	0.1
	Client_js =>Client	0.03	0.89	0	0.51	0.07	0.8	0.04	0.69	0.06	0.24	0.11	0.54
Lag 2	Server =>Client	0.61	0.8	0.87	0.91	0.93	0.56	0.45	0	0.09	0.94	0.91	0.81
	Server =>client_js	0.99	0.37	0.12	0.98	0.71	0.61	0.76	0.04	0.75	0.7	0.33	0.5
	Client =>Server	0	0	0	0.02	0.87	0.45	0.6	0.19	0.91	0.92	0.9	0.7
	Client =>Client_js	0.01	0	0	0.02	0.73	0.47	0.94	0.63	0.94	0.85	0.63	0.51
	Client =>Server	0.13	0.88	0	0.94	0.38	0.39	0	0.88	0.94	0.35	0.74	0.66
	Client_js =>Client	0.34	0.98	0	0.93	0.37	0.92	0.01	0.83	0.9	0.44	0.73	0.84
Lag 3	Server =>Client	0.85	0.55	0.81	0.99	0.78	0.87	0.56	0	0.26	0.95	0.97	0.67
	Server =>client_js	1	0.82	0.55	1	0.36	0.91	0.59	0.01	0.81	0.79	0.75	0.7
	Client =>Server	0.33	0	0.97	0	0.91	0.76	0.99	0.54	1	1	0.96	0.95
	Client =>Client_js	0.08	0	0.87	0	0.66	0.92	0.98	0.91	1	0.84	0.56	0.89
	Client =>Server	0.97	0.74	0	0.99	0.94	0.91	0	0.99	0.98	1	0.81	0.7
	Client_js =>Client	0.75	0.93	0	0.99	0.3	1	0.01	0.99	1	0.76	0.5	0.88
Lag 4	Server =>Client	0.98	0.47	0.8	0.96	0.95	0.95	0.71	0.01	0.88	0.91	1	0.93
	Server =>client_js	1	0.98	0.72	0.91	0.71	0.98	0.66	0.01	0.96	0.67	0.95	0.85
	Client =>Server	0.74	0	0	0	0.97	0.87	0.86	0.9	0.99	0.98	0.74	0.87
	Client =>Client_js	0.22	0	0	0	0.89	0.82	0.69	0.76	1	0.69	0.89	0.88
	Client =>Server	1	0.93	0	0.97	1	0.99	0	0.99	1	1	0.86	0.39
	Client_js =>Client	0.95	0.92	0	0.98	0.88	1	0	0.99	1	0.9	0.97	0.88

Table G.5: Transfer entropy between CS groups only - information transfer values (in %) - all apps

Lag	TE Evalues	phpMyAdmin	DokuWiki	OpenCart	phpBB	phpPgAdmin	MediaWiki	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
Lag 1	Server =>Client	0.11	0.11	0.37	0.09	0.4	0.08	0.19	0.11	0.1	0.16	0.16	0.09
	Server =>client_js	0.04	0.13	0.41	0.14	0.26	0.11	0.09	0.12	0.06	0.16	0.21	0.27
	Client =>Server	0.11	0.11	0.33	0.17	0.32	0.13	0.16	0.35	0.07	0.15	0.25	0.19
	Client =>Client_js	0.07	0.27	0.33	0.14	0.37	0.09	0.06	0.2	0.09	0.15	0.37	0.26
	Client =>Server	0.11	0.16	0.49	0.12	0.53	0.08	0.14	0.26	0.07	0.21	0.24	0.21
	Client_js =>Client	0.08	0.06	0.42	0.1	0.42	0.05	0.11	0.1	0.16	0.17	0.29	0.13
Lag 2	Server =>Client	0.09	0.19	0.2	0.1	0.3	0.16	0.24	0.27	0.17	0.15	0.27	0.21
	Server =>client_js	0.04	0.41	0.46	0.09	0.43	0.16	0.13	0.1	0.08	0.26	0.52	0.3
	Client =>Server	0.18	0.15	0.28	0.19	0.29	0.14	0.19	0.38	0.09	0.14	0.29	0.24
	Client =>Client_js	0.13	0.35	0.34	0.18	0.34	0.13	0.12	0.3	0.11	0.2	0.46	0.29
	Client =>Server	0.11	0.25	0.38	0.15	0.58	0.17	0.23	0.17	0.09	0.26	0.35	0.29
	Client_js =>Client	0.07	0.1	0.17	0.12	0.52	0.1	0.23	0.24	0.16	0.24	0.39	0.24
Lag 3	Server =>Client	0.1	0.29	0.24	0.12	0.3	0.2	0.29	0.23	0.22	0.16	0.24	0.34
	Server =>client_js	0.05	0.36	0.32	0.12	0.47	0.18	0.2	0.17	0.12	0.24	0.41	0.35
	Client =>Server	0.13	0.11	0.2	0.28	0.24	0.17	0.16	0.41	0.1	0.09	0.23	0.27
	Client =>Client_js	0.13	0.14	0.22	0.28	0.34	0.13	0.19	0.31	0.12	0.25	0.48	0.31
	Client =>Server	0.06	0.35	0.2	0.19	0.35	0.19	0.22	0.17	0.14	0.07	0.35	0.39
	Client_js =>Client	0.07	0.16	0.14	0.13	0.54	0.12	0.19	0.24	0.15	0.23	0.51	0.31
Lag 4	Server =>Client	0.1	0.28	0.21	0.13	0.16	0.19	0.28	0.15	0.16	0.15	0.05	0.31
	Server =>client_js	0.05	0.21	0.21	0.23	0.3	0.18	0.21	0.16	0.12	0.26	0.2	0.37
	Client =>Server	0.11	0.06	0	0.23	0.12	0.12	0.24	0.29	0.11	0.09	0.24	0.35
	Client =>Client_js	0.12	0.06	0.21	0.3	0.17	0.13	0.3	0.37	0.17	0.29	0.23	0.35
	Client =>Server	0.06	0.24	0	0.22	0.05	0.16	0.2	0.15	0.15	0.08	0.24	0.46
	Client_js =>Client	0.07	0.15	0.21	0.18	0.23	0.12	0.21	0.24	0.14	0.21	0.21	0.34

G.3 RQ3 – Impact of CS on issues evolution

G.3.1 Correlation

Table G.6: Correlations (Cor_ts) between CS and issues - p-values for all apps

CS\cor_ts	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	0.55	0.39	0.07	-0.82	0.37	0.83	0.71	0.64	0.32	-0.47
(Ex.)NPathComplexity	0.58	0.46	0.11	-0.82	0.23	0.81	0.73	0.64	0.54	-0.54
ExcessiveMethodLength	0.5	0.33	0.19	-0.82	0.37	0.86	0.73	0.7	0.62	0
ExcessiveClassLength	0.23	0.35	-0.1	-0.78	0.49	0.84	0.69	0.28	0.45	-0.01
ExcessiveParameterList	0.61	-0.32	0.23	-0.43	0.55	0	0.73	-0.05	0.69	-0.42
ExcessivePublicCount	0.37	0.02	-0.27	-0.48	0.48	0.84	0.7	-0.05	0.52	0
TooManyFields	0.23	0.15	1	-0.58	0.52	0.85	0.75	0.38	0.46	-0.08
TooManyMethods	0.16	0.22	-0.32	-0.58	0.34	0.87	0.71	0.31	-0.53	0.09
TooManyPublicMethods	0.13	0.37	0.13	-0.77	0.49	0.82	0.72	0.61	-0.18	-0.14
ExcessiveClassComplexity	0.32	0.39	0.14	-0.79	0.26	0.81	0.71	0.48	0.3	-0.57
(Ex.)NumberOfChildren	0.09	0.18	-0.21	-0.37	0.48	0.65	0.63	1	0.28	-0.44
(Ex.)DepthOfInheritance	0	0	0	0	0.01	0	0	0	-0.39	0
(Ex.)CouplingBetweenObjects	-0.15	0.26	-0.21	-0.37	0.54	0.76	0.56	0.6	0.57	-0.28
DevelopmentCodeFragment	0.39	0.49	0.26	-0.24	0.44	0.81	0.52	-0.05	-0.66	0
UnusedPrivateField	-0.19	0	0.22	0	-0.47	0.74	0.62	-0.05	0.33	0
UnusedLocalVariable	-0.15	-0.57	-0.14	-0.28	-0.48	0.45	0.73	-0.04	0.82	-0.46
UnusedPrivateMethod	0.46	0	0.17	0.15	0.5	0.17	0.39	-0.05	0.67	0.24
UnusedFormalParameter	-0.41	0.14	-0.19	-0.77	-0.07	0.79	0.69	0.57	-0.38	-0.61
embed.JS	-0.48	-0.24	-0.2	-0.44	-0.33	0.77	0.74	0.33	0.81	-0.54
inline.JS	-0.39	0.78	-0.44	-0.31	-0.18	-0.83	0.7	0.72	0.85	0
embed.CSS	0.44	0.82	-0.54	0.41	0.38	0.34	0.46	0.65	0.15	-0.35
inline.CSS	0.57	-0.14	-0.43	-0.46	-0.02	0.71	0.74	0.41	0.87	-0.2
css.in.JS	0.35	-0.54	-0.53	-0.57	-0.31	0.88	-0.4	-0.06	0.75	0.32
css.in.JS..jquery	0.46	0.08	-0.56	-0.38	-0.02	0.88	-0.06	0.08	0.84	0.69
max.lines	0.33	0.4	-0.34	-0.75	-0.29	0.89	-0.24	-0.02	0.54	0.5
max.lines.per.function	0.35	0.32	-0.21	-0.64	0.03	0.89	0.25	0	0.79	0.08
max.params	0.52	0.18	-0.45	-0.4	0.55	0.76	0.07	-0.3	0.77	-0.02
(Ex.)complexity	0.49	0.42	-0.48	-0.38	0.5	0.81	0.11	-0.33	0.78	0.14
max.depth	0.62	0.18	-0.47	-0.36	0.35	0.79	-0.3	-0.16	0.72	0.06
max.nested.callbacks	0.51	0.43	-0.23	-0.32	0.46	0.85	0.41	-0.39	0.74	-0.55

G.3.2 Causal inference

G.3.2.1 Linear Regression

Table G.7: Linear regression between CS and issues - p-values for all apps

CS\LM p	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	0.05	0.08	0.36	0.00	0.55	0.64	0.74	0.90	0.03	0.09
(Ex.)NPathComplexity	0.13	0.05	0.72	0.00	0.36	0.18	0.19	0.78	0.20	0.00
ExcessiveMethodLength	0.01	0.28	0.48	0.00	0.54	0.00	0.18	0.92	0.58	0.05
ExcessiveClassLength	0.00	0.91	0.75	0.00	0.79	0.01	0.88	0.66	0.38	0.00
ExcessiveParameterList	0.47	0.00	0.55	0.07	0.13	0.00	0.46	0.00	0.75	0.00
ExcessivePublicCount	0.03	0.67	0.58	0.16	0.94	0.94	0.64	0.00	0.54	0.00
TooManyFields	0.02	0.13	all cs=1	0.01	0.95	0.07	0.93	0.81	0.26	0.00
TooManyMethods	0.76	0.86	0.53	0.11	0.86	0.00	0.71	0.64	0.06	0.00
TooManyPublicMethods	0.02	0.08	0.59	0.04	0.66	0.71	0.79	0.96	0.05	0.00
ExcessiveClassComplexity	0.00	0.09	0.65	0.00	0.40	0.25	0.97	1.00	0.18	0.04
(Ex.)NumberOfChildren	0.00	0.00	0.00	0.36	0.01	0.89	0.37	all cs=1	0.83	0.00
(Ex.)DepthOfInheritance	0.00	0.00	0.00	0.00	0.51	0.00	0.00	0.00	0.00	0.00
(Ex.)CouplingBetweenObjects	0.00	0.00	0.00	0.38	0.16	0.88	0.11	0.64	0.40	0.04
DevelopmentCodeFragment	0.00	0.02	0.56	0.13	0.12	0.43	0.06	0.00	0.03	0.00
UnusedPrivateField	0.11	0.00	0.55	0.00	0.01	0.44	0.78	0.00	0.02	0.00
UnusedLocalVariable	0.64	0.13	0.60	0.22	0.02	0.31	0.31	0.25	0.75	0.00
UnusedPrivateMethod	0.27	0.00	0.31	0.00	0.09	0.04	0.53	0.00	0.10	0.00
UnusedFormalParameter	0.05	0.09	0.67	0.00	0.25	0.77	0.22	0.73	0.09	0.03
embed.JS	0.93	0.07	0.34	0.80	0.57	0.04	0.45	0.01	0.95	0.02
inline.JS	0.94	0.79	0.50	0.47	0.76	0.05	0.68	0.03	0.87	0.00
embed.CSS	0.17	0.00	0.69	0.00	0.01	0.17	0.42	0.02	0.62	0.00
inline.CSS	0.02	0.41	0.43	0.46	0.42	0.00	0.50	0.03	0.29	0.00
css.in.JS	0.70	0.09	0.72	0.03	0.20	0.34	0.70	0.10	0.44	0.06
css.in.JS..jquery	0.08	0.97	0.73	0.27	0.56	0.06	0.36	0.02	0.01	0.00
max.lines	0.00	0.86	0.62	0.00	0.97	0.03	0.90	0.37	0.12	0.01
max.lines.per.function	0.46	0.02	0.40	0.00	0.63	0.00	0.36	0.28	0.20	0.04
max.params	0.03	0.05	0.48	0.17	0.00	0.91	0.13	0.13	0.94	0.03
(Ex.)complexity	0.03	0.02	0.48	0.32	0.01	0.94	0.27	0.16	0.67	0.07
max.depth	0.05	0.70	0.51	0.00	0.05	0.97	0.44	0.03	0.23	0.03
max.nested.callbacks	0.03	0.00	0.53	0.28	0.01	0.11	0.80	0.03	0.78	0.03

G.3.2.2 Granger-causality lag1 (previous release)

Table G.8: Granger-causality lag 1 between CS and issues density - p-values for all apps

CS\GC lag1 p	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	0.35	0.16	0.49	0.00	0.88	0.52	0.90	0.84	0.42	0.34
(Ex.)NPathComplexity	0.45	0.10	0.40	0.00	0.72	0.13	0.40	0.76	0.78	0.08
ExcessiveMethodLength	0.15	0.53	0.36	0.00	0.84	0.00	0.42	0.85	0.11	0.02
ExcessiveClassLength	0.00	0.56	0.54	0.00	0.69	0.08	0.84	0.64	0.29	0.63
ExcessiveParameterList	0.87	0.04	0.33	0.06	0.25	1.00	0.65	0.17	0.07	0.00
ExcessivePublicCount	0.12	0.99	0.56	0.19	0.89	0.74	0.94	0.38	0.55	1.00
TooManyFields	0.26	0.05	1.00	0.01	0.88	0.16	0.50	0.89	0.62	0.00
TooManyMethods	0.71	0.54	0.56	0.13	0.67	0.02	0.71	0.67	0.29	0.42
TooManyPublicMethods	0.02	0.15	0.34	0.04	0.93	0.58	0.88	0.90	0.74	0.00
ExcessiveClassComplexity	0.00	0.19	0.26	0.00	0.80	0.55	0.69	0.95	0.17	0.19
(Ex.)NumberOfChildren	0.03	0.00	0.91	0.40	0.06	0.57	0.58	1.00	0.17	0.00
(Ex.)DepthOfInheritance	1.00	1.00	1.00	1.00	0.91	1.00	1.00	1.00	0.54	1.00
(Ex.)CouplingBetweenObjects	0.00	0.03	0.58	0.42	0.28	0.75	0.12	0.84	0.04	0.00
DevelopmentCodeFragment	0.00	0.02	0.23	0.11	0.36	0.27	0.09	0.57	0.17	1.00
UnusedPrivateField	0.22	1.00	0.34	1.00	0.05	0.62	0.68	0.17	0.00	1.00
UnusedLocalVariable	0.55	0.21	0.63	0.16	0.18	0.98	0.58	0.87	0.04	0.00
UnusedPrivateMethod	0.81	1.00	0.51	0.76	0.21	0.12	0.60	0.17	0.14	0.93
UnusedFormalParameter	0.20	0.10	0.46	0.00	0.77	0.99	0.11	0.92	0.17	0.00
embed.JS	0.85	0.13	0.30	0.75	0.75	0.37	0.95	0.17	0.42	0.37
inline.JS	0.83	0.92	0.31	0.45	0.86	0.19	0.74	0.20	0.33	1.00
embed.CSS	0.28	0.79	0.51	0.00	0.02	0.28	0.22	0.21	0.77	0.00
inline.CSS	0.12	0.56	0.35	0.41	0.72	0.02	0.99	0.17	0.06	0.00
css.in.JS	0.68	0.15	0.26	0.02	0.36	0.28	0.63	0.17	0.39	0.22
css.in.JS..jquery	0.26	0.80	0.28	0.32	0.95	0.13	0.07	0.15	0.01	0.06
max.lines	0.01	0.70	0.39	0.00	0.77	0.33	0.84	0.90	0.44	0.24
max.lines.per.function	0.40	0.03	0.67	0.00	0.92	0.03	0.23	0.98	0.64	0.72
max.params	0.15	0.05	0.32	0.20	0.02	0.88	0.02	0.68	0.66	0.27
(Ex.)complexity	0.15	0.02	0.29	0.38	0.03	0.96	0.04	0.76	0.18	0.31
max.depth	0.22	0.57	0.28	0.29	0.22	0.79	0.11	0.13	0.89	0.28
max.nested.callbacks	0.12	0.89	0.68	0.32	0.05	0.56	0.67	0.34	0.17	0.28

G.3.2.3 Granger-causality lag2 (two releases before)

Table G.9: Granger causality lag between CS and issues density- p-values for all apps

CS\GC lag2 p	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	0.90	0.59	0.49	0.00	0.83	0.05	0.10	0.81	0.90	0.01
(Ex.)NPathComplexity	0.95	0.51	0.91	0.00	0.82	0.25	0.51	0.95	0.83	0.31
ExcessiveMethodLength	0.65	0.59	0.78	0.00	0.84	0.00	0.30	0.79	0.24	0.00
ExcessiveClassLength	0.06	0.34	0.64	0.00	0.67	0.14	0.89	0.49	0.36	0.17
ExcessiveParameterList	0.93	0.06	0.64	0.15	0.55	1.00	0.02	0.30	0.09	0.08
ExcessivePublicCount	0.47	0.64	0.93	0.42	0.66	0.82	0.98	0.20	0.65	1.00
TooManyFields	0.58	0.06	1.00	0.01	0.79	0.54	0.55	0.24	0.62	0.00
TooManyMethods	0.83	0.71	0.92	0.30	0.38	0.04	0.98	0.50	0.68	0.02
TooManyPublicMethods	0.15	0.58	0.76	0.08	0.83	0.31	0.70	0.71	0.30	0.00
ExcessiveClassComplexity	0.09	0.43	0.60	0.00	0.81	0.58	0.22	0.67	0.09	0.00
(Ex.)NumberOfChildren	0.21	0.00	0.86	0.70	0.21	0.23	0.17	1.00	0.23	0.01
(Ex.)DepthOfInheritance	1.00	1.00	1.00	1.00	0.29	1.00	1.00	1.00	0.64	1.00
(Ex.)CouplingBetweenObjects	0.01	0.31	0.83	0.73	0.53	0.80	0.22	0.97	0.14	0.00
DevelopmentCodeFragment	0.00	0.16	0.57	0.21	0.72	0.60	0.12	0.15	0.56	1.00
UnusedPrivateField	0.71	1.00	0.55	1.00	0.16	0.57	0.91	0.32	0.00	1.00
UnusedLocalVariable	0.91	0.74	0.88	0.17	0.61	0.50	0.28	0.98	0.05	0.00
UnusedPrivateMethod	0.90	1.00	0.74	0.95	0.58	0.44	0.62	0.30	0.42	0.97
UnusedFormalParameter	0.55	0.25	0.84	0.00	0.97	0.02	0.03	0.24	0.54	0.00
embed.JS	0.81	0.75	0.60	0.93	0.84	0.42	0.34	0.27	0.00	0.47
inline.JS	0.97	0.64	0.60	0.73	0.65	0.42	0.06	0.29	0.00	1.00
embed.CSS	0.69	0.74	0.89	0.00	0.12	0.03	0.56	0.32	0.02	0.12
inline.CSS	0.41	0.92	0.66	0.67	0.62	0.14	0.09	0.26	0.00	0.01
css.in.JS	0.50	0.73	0.41	0.03	0.75	0.71	0.48	0.34	0.51	0.01
css.in.JS..jquery	0.76	0.87	0.48	0.62	0.97	0.21	0.00	0.25	0.01	0.38
max.lines	0.19	0.73	0.75	0.00	0.91	0.04	0.13	0.95	0.64	0.45
max.lines.per.function	0.84	0.26	0.88	0.00	0.93	0.09	0.16	0.95	0.03	0.98
max.params	0.67	0.10	0.61	0.46	0.10	0.76	0.00	0.81	0.77	0.77
(Ex.)complexity	0.67	0.11	0.54	0.69	0.14	0.93	0.00	0.87	0.23	0.79
max.depth	0.75	0.69	0.53	0.59	0.42	0.19	0.00	0.23	0.35	0.75
max.nested.callbacks	0.58	0.54	0.86	0.63	0.21	0.30	0.70	0.49	0.00	0.81

G.3.2.4 Transfer entropy lag1 (previous release)

Table G.10: Transfer entropy lag1 between CS and issues density - p-values for all apps

CS\TE lag1 p	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
CS	p	p	p	p	p	p	p	p	p	p
(Ex.)CyclomaticComplexity	0.00	0.62	0.00	0.02	0.12	0.06	0.33	0.54	0.11	0.08
(Ex.)NPathComplexity	0.00	0.62	0.49	0.02	0.33	0.10	0.14	0.54	0.33	0.01
ExcessiveMethodLength	0.00	0.88	0.01	0.02	0.36	0.04	0.32	0.54	0.22	0.16
ExcessiveClassLength	0.00	0.04	0.01	0.06	0.47	0.00	0.35	0.48	0.63	0.00
ExcessiveParameterList	0.00	0.00	0.10	0.33	0.10	0.00	0.27	0.00	0.22	0.00
ExcessivePublicCount	0.00	0.84	0.08	0.07	0.49	0.21	0.18	0.00	0.96	0.00
TooManyFields	0.00	0.07	all cs=1	0.42	0.45	0.14	0.01	0.41	0.31	0.00
TooManyMethods	0.03	0.36	0.08	0.13	0.52	0.14	0.03	0.41	0.01	0.00
TooManyPublicMethods	0.00	0.73	0.01	0.29	0.18	0.21	0.48	0.47	0.04	0.30
ExcessiveClassComplexity	0.00	0.62	0.05	0.06	0.27	0.02	0.12	0.41	0.17	0.15
(Ex.)NumberOfChildren	0.02	0.00	0.00	0.02	0.08	0.13	0.48	all cs=1	0.11	0.02
(Ex.)DepthOfInheritance	0.00	0.00	0.00	0.00	0.48	0.00	0.00	0.00	0.00	0.00
(Ex.)CouplingBetweenObjects	0.02	0.00	0.00	0.06	0.00	0.12	0.01	0.47	0.58	0.30
DevelopmentCodeFragment	0.00	0.01	0.00	0.02	0.25	0.09	0.00	0.00	0.24	0.00
UnusedPrivateField	0.00	0.00	0.01	0.00	0.28	0.01	0.45	0.00	0.79	0.00
UnusedLocalVariable	0.13	0.04	0.02	0.02	0.67	0.17	0.14	0.41	0.23	0.03
UnusedPrivateMethod	0.00	0.00	0.00	0.00	0.13	0.03	0.38	0.00	0.48	0.00
UnusedFormalParameter	0.24	0.23	0.37	0.06	0.85	0.01	0.11	0.30	0.00	0.01
embed.JS	0.34	0.22	0.01	0.18	0.57	0.01	0.03	0.46	0.20	0.17
inline.JS	0.00	0.51	0.00	0.00	0.77	0.00	0.59	0.61	0.20	0.00
embed.CSS	0.00	0.00	0.40	0.00	0.54	0.20	0.01	0.61	0.63	0.00
inline.CSS	0.34	0.42	0.01	0.19	0.81	0.01	0.03	0.72	0.80	0.00
css.in.JS	0.28	0.12	0.13	0.00	0.88	0.12	0.18	0.43	0.00	0.10
css.in.JS..jquery	0.00	0.21	0.09	0.16	0.71	0.07	0.00	0.61	0.21	0.02
max.lines	0.02	0.11	0.00	0.32	0.31	0.12	0.04	0.20	0.29	0.06
max.lines.per.function	0.19	0.00	0.00	0.60	0.34	0.27	0.07	0.51	0.19	0.30
max.params	0.00	0.00	0.00	0.09	0.58	0.03	0.06	0.04	0.04	0.02
(Ex.)complexity	0.02	0.00	0.06	0.14	0.49	0.14	0.02	0.66	0.22	0.09
max.depth	0.00	0.00	0.01	0.00	0.82	0.11	0.00	0.08	0.07	0.09
max.nested.callbacks	0.00	0.00	0.09	0.27	0.43	0.08	0.28	0.01	0.03	0.50

Table G.11: Transfer entropy lag1 between CS and issues density - TE values(%) for all apps

CS\TE lag1 te	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
CS	te	te	te	te	te	te	te	te	te	te
(Ex.)CyclomaticComplexity	10%	18%	48%	41%	15%	11%	8%	13%	37%	17%
(Ex.)NPathComplexity	11%	18%	23%	41%	10%	9%	9%	13%	29%	20%
ExcessiveMethodLength	11%	10%	48%	41%	10%	17%	11%	13%	37%	14%
ExcessiveClassLength	10%	36%	51%	24%	10%	24%	10%	13%	19%	0%
ExcessiveParameterList	12%	0%	28%	25%	16%	0%	12%	0%	33%	19%
ExcessivePublicCount	10%	9%	39%	24%	9%	13%	14%	0%	9%	0%
TooManyFields	9%	30%	100%	23%	9%	15%	18%	13%	25%	0%
TooManyMethods	10%	19%	39%	22%	10%	14%	16%	13%	25%	0%
TooManyPublicMethods	14%	15%	50%	25%	10%	12%	8%	13%	44%	9%
ExcessiveClassComplexity	11%	18%	42%	24%	10%	14%	7%	13%	34%	14%
(Ex.)NumberOfChildren	9%	0%	0%	33%	12%	14%	7%	100%	31%	20%
(Ex.)DepthOfInheritance	0%	0%	0%	0%	2%	0%	0%	0%	0%	0%
(Ex.)CouplingBetweenObjects	10%	0%	0%	29%	24%	13%	18%	13%	19%	12%
DevelopmentCodeFragment	13%	40%	60%	41%	12%	15%	24%	0%	33%	0%
UnusedPrivateField	13%	0%	51%	0%	7%	19%	8%	0%	11%	0%
UnusedLocalVariable	6%	34%	43%	41%	8%	14%	9%	13%	39%	17%
UnusedPrivateMethod	10%	0%	53%	0%	13%	16%	9%	0%	18%	0%
UnusedFormalParameter	6%	21%	25%	40%	6%	15%	13%	13%	41%	18%
embed.JS	6%	18%	34%	30%	8%	22%	11%	14%	25%	12%
inline.JS	13%	12%	42%	35%	6%	29%	6%	12%	25%	0%
embed.CSS	12%	0%	25%	24%	9%	12%	22%	11%	15%	20%
inline.CSS	6%	13%	48%	29%	5%	16%	13%	10%	16%	21%
css.in.JS	7%	20%	36%	40%	6%	14%	11%	12%	62%	17%
css.in.JS..jquery	11%	13%	42%	22%	8%	16%	32%	11%	34%	16%
max.lines	9%	20%	54%	28%	9%	15%	10%	19%	28%	14%
max.lines.per.function	7%	7%	60%	18%	8%	13%	16%	14%	31%	9%
max.params	15%	7%	42%	33%	7%	20%	15%	24%	44%	16%
(Ex.)complexity	9%	7%	42%	28%	7%	12%	18%	11%	33%	13%
max.depth	13%	15%	48%	0%	5%	13%	19%	24%	47%	13%
max.nested.callbacks	9%	0%	39%	19%	9%	16%	13%	12%	44%	9%

G.3.2.5 Transfer entropy lag2 (two releases before)

Table G.12: Transfer entropy lag2 between CS and issues density - p-values for all apps

CS\TE lag2 p	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
CS	p	p	p	p	p	p	p	p	p	p
(Ex.)CyclomaticComplexity	0.04	0.00	0.07	0.54	0.26	0.02	0.20	0.77	0.85	0.34
(Ex.)NPathComplexity	0.19	0.00	0.93	0.54	0.51	0.02	0.11	0.77	0.65	0.04
ExcessiveMethodLength	0.00	0.09	0.65	0.54	0.19	0.31	0.15	0.77	0.26	0.07
ExcessiveClassLength	0.01	0.00	0.26	0.17	0.48	0.39	0.87	0.63	0.71	0.00
ExcessiveParameterList	0.01	0.00	0.72	0.53	0.02	0.00	0.80	0.00	0.53	0.07
ExcessivePublicCount	0.01	0.09	0.55	0.09	0.45	0.71	0.37	0.00	0.48	0.00
TooManyFields	0.09	0.00	all cs=1	0.15	0.71	0.45	0.09	0.72	0.69	0.00
TooManyMethods	0.01	0.08	0.55	0.09	0.50	0.57	0.67	0.72	0.00	0.00
TooManyPublicMethods	0.00	0.02	0.59	0.73	0.35	0.57	0.87	0.73	0.33	0.03
ExcessiveClassComplexity	0.00	0.01	0.17	0.03	0.61	0.06	0.04	0.77	0.22	0.08
(Ex.)NumberOfChildren	0.04	0.00	0.00	0.67	0.05	0.31	0.31	all cs=1	0.35	0.01
(Ex.)DepthOfInheritance	0.00	0.00	0.00	0.00	0.10	0.00	0.00	0.00	0.00	0.00
(Ex.)CouplingBetweenObjects	0.26	0.00	0.00	0.89	0.05	0.54	0.32	0.73	0.59	0.02
DevelopmentCodeFragment	0.00	0.28	0.36	0.54	0.18	0.55	0.05	0.00	1.00	0.00
UnusedPrivateField	0.11	0.00	0.19	0.00	0.19	0.11	0.25	0.00	0.84	0.00
UnusedLocalVariable	0.45	0.32	0.86	0.60	0.91	0.36	0.23	0.77	0.72	0.10
UnusedPrivateMethod	0.07	0.00	0.00	0.00	0.08	0.16	0.38	0.00	0.92	0.00
UnusedFormalParameter	0.00	0.02	0.46	0.65	0.48	0.02	0.45	0.38	0.00	0.01
embed.JS	0.31	0.05	0.40	0.19	0.34	0.07	0.16	0.25	0.54	0.52
inline.JS	0.32	0.12	0.00	0.01	0.35	0.01	0.68	0.33	0.54	0.00
embed.CSS	0.16	0.00	0.61	0.00	0.11	0.26	0.42	0.45	0.74	0.00
inline.CSS	0.03	0.01	0.60	0.00	0.28	0.08	0.20	0.68	0.39	0.00
css.in.JS	0.62	0.20	0.58	0.33	0.45	0.55	0.03	0.23	0.11	0.46
css.in.JS..jquery	0.01	0.16	0.21	0.86	0.41	0.28	0.06	0.45	0.34	0.00
max.lines	0.29	0.10	0.00	0.36	0.32	0.29	0.18	0.66	0.80	0.19
max.lines.per.function	0.19	0.00	0.70	0.25	0.51	0.57	0.02	0.42	0.55	0.54
max.params	0.00	0.00	0.00	0.55	0.39	0.50	0.02	0.77	0.51	0.01
(Ex.)complexity	0.00	0.00	0.00	0.31	0.32	0.76	0.01	0.91	0.36	0.03
max.depth	0.00	0.00	0.65	0.00	0.42	0.39	0.00	0.00	0.20	0.03
max.nested.callbacks	0.04	0.00	0.72	0.61	0.03	0.65	0.39	0.00	0.55	0.87

Table G.13: Transfer entropy lag2 between CS and issues density - TE values(%) for all apps

CS\TE lag2 te	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
CS	te	te	te	te	te	te	te	te	te	te
(Ex.)CyclomaticComplexity	15%	73%	62%	40%	21%	21%	18%	17%	42%	25%
(Ex.)NPathComplexity	12%	73%	41%	40%	15%	19%	18%	17%	46%	33%
ExcessiveMethodLength	11%	61%	45%	40%	21%	26%	27%	17%	53%	30%
ExcessiveClassLength	16%	88%	50%	43%	20%	24%	15%	19%	42%	0%
ExcessiveParameterList	12%	0%	34%	43%	40%	0%	17%	0%	50%	23%
ExcessivePublicCount	16%	52%	54%	52%	17%	17%	24%	0%	51%	0%
TooManyFields	11%	76%	100%	65%	16%	23%	30%	17%	40%	0%
TooManyMethods	22%	62%	54%	52%	17%	21%	16%	17%	40%	0%
TooManyPublicMethods	18%	69%	45%	37%	13%	21%	12%	17%	58%	31%
ExcessiveClassComplexity	18%	71%	53%	57%	15%	14%	14%	15%	54%	30%
(Ex.)NumberOfChildren	19%	0%	0%	34%	26%	26%	20%	100%	31%	39%
(Ex.)DepthOfInheritance	0%	0%	0%	0%	10%	0%	0%	0%	0%	0%
(Ex.)CouplingBetweenObjects	18%	0%	0%	36%	36%	20%	21%	17%	47%	36%
DevelopmentCodeFragment	12%	50%	49%	40%	27%	18%	29%	0%	17%	0%
UnusedPrivateField	15%	0%	62%	0%	16%	33%	24%	0%	33%	0%
UnusedLocalVariable	12%	42%	35%	44%	12%	27%	18%	15%	51%	28%
UnusedPrivateMethod	14%	0%	54%	0%	29%	22%	20%	0%	24%	0%
UnusedFormalParameter	23%	70%	54%	42%	20%	22%	21%	16%	38%	35%
embed.JS	16%	55%	36%	55%	17%	37%	16%	35%	32%	19%
inline.JS	15%	47%	54%	35%	21%	35%	12%	28%	32%	0%
embed.CSS	17%	0%	50%	29%	21%	22%	24%	28%	32%	36%
inline.CSS	21%	53%	45%	54%	23%	23%	18%	19%	62%	36%
css.in.JS	13%	40%	51%	40%	16%	19%	33%	28%	59%	26%
css.in.JS..jquery	15%	31%	53%	29%	25%	27%	36%	28%	58%	33%
max.lines	13%	41%	45%	54%	21%	26%	17%	22%	35%	22%
max.lines.per.function	18%	11%	36%	54%	14%	24%	39%	26%	50%	18%
max.params	25%	11%	54%	44%	23%	23%	35%	18%	48%	38%
(Ex.)complexity	22%	11%	62%	52%	21%	14%	40%	13%	57%	28%
max.depth	16%	21%	45%	0%	20%	23%	40%	20%	65%	28%
max.nested.callbacks	11%	0%	46%	29%	33%	20%	23%	24%	41%	17%

G.4 RQ4 – Impact of CS on bugs evolution

G.4.1 Correlation

Table G.14: Correlations(cor_ts) between CS and bugs - p-values for all apps

CS\cor_ts	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	0.58	0.48	0.03	-0.87	0.19	0.67	0.41	0.4	-0.17	-0.21
(Ex.)NPathComplexity	0.61	0.56	0.06	-0.87	0.04	0.65	0.46	0.33	-0.01	-0.27
ExcessiveMethodLength	0.5	0.37	0.13	-0.87	0.2	0.73	0.45	0.43	0.03	0.16
ExcessiveClassLength	0.24	0.41	-0.17	-0.84	0.38	0.72	0.35	0.36	-0.05	-0.15
ExcessiveParameterList	0.6	-0.33	0.18	-0.48	0.41	0	0.46	0.22	0.32	0.01
ExcessivePublicCount	0.4	0.03	-0.31	-0.56	0.33	0.75	0.36	0.22	0.03	0
TooManyFields	0.3	0.07	1	-0.62	0.39	0.68	0.41	0.41	-0.03	-0.22
TooManyMethods	0.21	0.13	-0.36	-0.66	0.21	0.75	0.46	0.41	-0.34	-0.01
TooManyPublicMethods	0.15	0.52	0.09	-0.82	0.21	0.68	0.44	0.41	-0.3	-0.08
ExcessiveClassComplexity	0.31	0.32	0.09	-0.85	0.21	0.73	0.42	0.31	-0.11	-0.36
(Ex.)NumberOfChildren	0.05	0.3	-0.21	-0.38	0.4	0.53	0.48	1	0.12	0.02
(Ex.)DepthOfInheritance	0	0	0	0	0.03	0	0	0	-0.74	0
(Ex.)CouplingBetweenObjects	-0.13	0.46	-0.21	-0.38	0.42	0.77	0.23	0.41	0.05	0.15
DevelopmentCodeFragment	0.45	0.72	0.23	-0.27	0.2	0.63	0.19	0.22	-0.59	0
UnusedPrivateField	-0.18	0	0.22	0	-0.34	0.73	0.3	0.22	0.06	0
UnusedLocalVariable	-0.09	-0.69	-0.17	-0.33	-0.34	0.31	0.43	-0.24	0.37	-0.35
UnusedPrivateMethod	0.49	0	0.07	0.01	0.39	0.38	0.29	0.22	0.17	0.4
UnusedFormalParameter	-0.37	0.21	-0.22	-0.85	-0.01	0.61	0.38	0.68	-0.31	-0.24
embed.JS	-0.42	-0.45	-0.23	-0.49	-0.21	0.64	0.44	0.5	0.53	-0.32
inline.JS	-0.33	0.83	-0.43	-0.33	-0.26	-0.66	0.39	0.5	0.42	0
embed.CSS	0.49	0.83	-0.51	0.44	0.26	0.06	0.12	0.37	0.08	0
inline.CSS	0.56	-0.27	-0.47	-0.52	0.02	0.7	0.47	0.56	0.42	0.26
css.in.JS	0.4	-0.62	-0.52	-0.63	-0.17	0.75	-0.28	0.22	0.29	0.46
css.in.JS..jquery	0.49	0.16	-0.55	-0.42	0.06	0.73	0.1	-0.01	0.4	0.74
max.lines	0.38	0.5	-0.35	-0.79	-0.2	0.7	-0.24	0.24	0.27	0.63
max.lines.per.function	0.4	0.44	-0.21	-0.69	0.02	0.74	0.1	0.25	0.33	0.12
max.params	0.53	0.25	-0.44	-0.46	0.48	0.72	0.18	0.02	0.35	0.4
(Ex.)complexity	0.51	0.47	-0.47	-0.44	0.46	0.74	0.18	0.03	0.36	0.52
max.depth	0.59	0.21	-0.45	-0.42	0.32	0.63	-0.1	0.18	0.3	0.42
max.nested.callbacks	0.45	0.07	-0.23	-0.38	0.37	0.76	0.17	0	0.32	-0.17

G.4.2 Causal inference

G.4.2.1 Linear Regression

Table G.15: Linear regression between CS and bugs - p-values for all apps

CS\LM p-val	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	0.12	0.02	0.78	0.00	0.81	0.11	0.02	0.49	0.20	0.17
(Ex.)NPathComplexity	0.31	0.02	0.47	0.00	0.96	0.79	0.00	0.54	0.59	0.22
ExcessiveMethodLength	0.01	0.04	0.40	0.01	0.79	0.00	0.00	0.48	0.82	0.70
ExcessiveClassLength	0.00	0.18	0.58	0.00	0.20	0.00	0.98	0.46	0.75	0.97
ExcessiveParameterList	0.43	0.24	0.56	0.18	0.00	0.00	0.00	0.49	0.93	0.04
ExcessivePublicCount	0.02	0.17	0.83	0.09	0.36	0.27	0.00	0.31	0.87	0.00
TooManyFields	0.97	0.84	all cs=1	0.01	0.27	0.03	0.02	0.41	0.71	0.08
TooManyMethods	0.07	0.47	0.84	0.06	0.36	0.00	0.01	0.43	0.34	0.03
TooManyPublicMethods	0.00	0.03	0.55	0.03	0.61	0.22	0.14	0.48	0.37	0.00
ExcessiveClassComplexity	0.00	0.01	0.55	0.00	0.89	0.00	0.01	0.47	0.46	0.76
(Ex.)NumberOfChildren	0.00	0.01	0.00	0.24	0.00	0.15	0.77	all cs=1	0.77	0.01
(Ex.)DepthOfInheritance	0.00	0.00	0.00	0.00	0.34	0.00	0.00	0.00	0.18	0.00
(Ex.)CouplingBetweenObjects	0.00	0.13	0.00	0.26	0.00	0.05	0.40	0.67	0.96	0.61
DevelopmentCodeFragment	0.00	0.07	0.42	0.24	0.12	0.28	0.53	0.33	0.21	0.00
UnusedPrivateField	0.97	0.00	0.39	0.00	0.00	0.15	0.04	0.63	0.04	0.00
UnusedLocalVariable	0.53	0.04	0.51	0.33	0.10	0.00	0.00	0.69	0.76	0.00
UnusedPrivateMethod	0.08	0.00	0.61	0.61	0.00	0.01	0.07	0.49	0.32	0.07
UnusedFormalParameter	0.48	0.20	0.42	0.00	0.94	0.03	0.06	0.37	0.42	0.00
embed.JS	0.07	0.02	0.44	0.91	0.29	0.00	0.00	0.44	0.19	0.42
inline.JS	0.06	0.37	0.39	0.77	0.19	0.00	0.00	0.68	0.13	0.00
embed.CSS	0.35	0.41	0.81	0.00	0.00	0.79	0.70	0.55	0.41	0.10
inline.CSS	0.74	0.15	0.41	0.71	0.36	0.00	0.00	0.62	0.05	0.02
css.in.JS	0.49	0.04	0.45	0.07	0.20	0.09	0.05	0.61	0.31	0.67
css.in.JS..jquery	0.30	0.60	0.41	0.14	0.72	0.03	0.73	0.55	0.03	0.00
max.lines	0.10	0.45	0.33	0.00	0.44	0.00	0.00	0.76	0.48	0.26
max.lines.per.function	0.16	0.03	0.69	0.00	0.90	0.03	0.25	0.79	0.96	0.30
max.params	0.82	0.18	0.38	0.08	0.00	0.08	0.80	0.74	0.25	0.34
(Ex.)complexity	0.83	0.06	0.38	0.16	0.00	0.03	0.92	0.77	0.81	0.37
max.depth	0.85	0.80	0.37	0.12	0.02	0.79	0.22	0.53	0.74	0.30
max.nested.callbacks	0.67	0.99	0.60	0.15	0.00	0.00	0.29	0.55	0.45	0.52

G.4.2.2 Granger-causality lag1 (previous release)

Table G.16: Granger-causality lag 1 between CS and bugs - p-values for all apps

CS\GClag1 p-val	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	0.28	0.04	0.70	0.00	0.68	0.44	0.09	0.66	0.83	0.20
(Ex.)NPathComplexity	0.49	0.02	0.49	0.00	0.81	0.73	0.00	0.69	0.66	0.34
ExcessiveMethodLength	0.04	0.12	0.32	0.00	0.72	0.02	0.02	0.64	0.18	0.26
ExcessiveClassLength	0.00	0.51	0.70	0.00	0.32	0.02	0.89	0.52	0.25	0.99
ExcessiveParameterList	0.66	0.15	0.73	0.13	0.07	1.00	0.01	0.57	0.15	0.32
ExcessivePublicCount	0.05	0.50	0.62	0.11	0.45	0.60	0.01	0.50	0.59	1.00
TooManyFields	0.79	0.41	1.00	0.00	0.38	0.08	0.16	0.58	0.59	0.70
TooManyMethods	0.08	0.99	0.61	0.08	0.38	0.07	0.05	0.58	0.68	0.53
TooManyPublicMethods	0.00	0.05	0.60	0.02	0.59	0.61	0.28	0.64	0.81	0.06
ExcessiveClassComplexity	0.00	0.02	0.88	0.00	0.71	0.05	0.09	0.62	0.37	0.55
(Ex.)NumberOfChildren	0.00	0.00	0.45	0.28	0.00	0.58	0.63	1.00	0.61	0.17
(Ex.)DepthOfInheritance	1.00	1.00	1.00	1.00	0.81	1.00	1.00	1.00	0.64	1.00
(Ex.)CouplingBetweenObjects	0.00	0.09	0.33	0.30	0.07	0.30	0.70	0.67	0.18	0.67
DevelopmentCodeFragment	0.00	0.04	0.63	0.17	0.41	0.17	0.52	0.52	0.54	1.00
UnusedPrivateField	0.80	1.00	0.43	1.00	0.06	0.26	0.22	0.62	0.00	1.00
UnusedLocalVariable	0.89	0.05	0.29	0.23	0.44	0.11	0.02	0.87	0.21	0.05
UnusedPrivateMethod	0.33	1.00	0.46	0.68	0.06	0.03	0.24	0.57	0.29	0.99
UnusedFormalParameter	0.66	0.18	0.41	0.00	0.71	0.20	0.17	0.61	0.45	0.29
embed.JS	0.26	0.03	0.38	0.95	0.54	0.04	0.06	0.72	0.02	0.62
inline.JS	0.24	0.46	0.50	0.63	0.25	0.01	0.05	0.85	0.01	1.00
embed.CSS	0.70	0.57	0.88	0.00	0.01	0.91	0.64	0.80	0.12	0.33
inline.CSS	0.97	0.19	0.45	0.56	0.73	0.01	0.03	0.79	0.00	0.41
css.in.JS	0.94	0.05	0.84	0.04	0.43	0.08	0.34	0.61	0.59	0.83
css.in.JS..jquery	0.45	0.78	0.77	0.19	0.63	0.12	0.18	0.70	0.01	0.01
max.lines	0.19	0.62	0.32	0.00	0.57	0.01	0.05	0.81	0.83	0.44
max.lines.per.function	0.45	0.03	0.40	0.00	0.99	0.36	0.55	0.83	0.65	0.45
max.params	0.94	0.15	0.38	0.11	0.00	0.37	0.34	0.93	0.06	0.50
(Ex.)complexity	0.93	0.04	0.41	0.23	0.01	0.22	0.37	0.88	0.25	0.50
max.depth	0.94	0.95	0.46	0.17	0.20	0.96	0.97	0.65	0.47	0.48
max.nested.callbacks	0.97	1.00	0.38	0.19	0.05	0.05	0.94	0.92	0.04	0.60

G.4.2.3 Granger-causality lag2 (two releases before)

Table G.17: Granger-causality lag2 between CS and bugs - p-values for all apps

CS\GClag2 p-val	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	0.79	0.10	0.46	0.00	0.62	0.17	0.14	0.81	0.75	0.09
(Ex.)NPathComplexity	0.76	0.07	0.34	0.00	0.64	0.79	0.03	0.95	0.72	0.66
ExcessiveMethodLength	0.32	0.12	0.16	0.00	0.69	0.05	0.04	0.76	0.11	0.65
ExcessiveClassLength	0.01	0.13	0.82	0.00	0.39	0.12	0.79	0.65	0.18	0.57
ExcessiveParameterList	0.76	0.04	0.78	0.25	0.32	1.00	0.01	0.63	0.18	0.65
ExcessivePublicCount	0.30	0.10	0.49	0.24	0.49	0.96	0.10	0.54	0.75	1.00
TooManyFields	0.86	0.05	1.00	0.00	0.58	0.38	0.37	0.46	0.67	0.76
TooManyMethods	0.30	0.37	0.46	0.17	0.32	0.11	0.19	0.67	0.79	0.43
TooManyPublicMethods	0.00	0.12	0.38	0.03	0.61	0.59	0.75	0.75	0.38	0.29
ExcessiveClassComplexity	0.02	0.08	0.83	0.00	0.58	0.27	0.20	0.72	0.30	0.73
(Ex.)NumberOfChildren	0.02	0.02	0.37	0.54	0.04	0.68	0.36	1.00	0.71	0.20
(Ex.)DepthOfInheritance	1.00	1.00	1.00	1.00	0.24	1.00	1.00	1.00	0.84	1.00
(Ex.)CouplingBetweenObjects	0.01	0.60	0.20	0.58	0.31	0.69	0.97	0.93	0.21	0.98
DevelopmentCodeFragment	0.00	0.14	0.59	0.28	0.46	0.50	0.44	0.49	0.97	1.00
UnusedPrivateField	0.99	1.00	0.29	1.00	0.25	0.65	0.61	0.67	0.00	1.00
UnusedLocalVariable	0.97	0.25	0.17	0.21	0.86	0.49	0.01	0.93	0.20	0.43
UnusedPrivateMethod	0.90	1.00	0.38	0.90	0.25	0.20	0.44	0.63	0.62	0.78
UnusedFormalParameter	0.66	0.16	0.24	0.00	0.77	0.06	0.09	0.57	0.88	0.73
embed.JS	0.67	0.48	0.22	0.94	0.90	0.22	0.16	0.78	0.00	0.01
inline.JS	0.68	0.87	0.41	0.81	0.49	0.10	0.03	0.76	0.00	1.00
embed.CSS	0.95	0.96	0.54	0.00	0.08	0.12	0.97	0.82	0.00	0.61
inline.CSS	0.68	0.89	0.29	0.73	0.59	0.07	0.04	0.75	0.00	0.60
css.in.JS	0.69	0.44	0.76	0.06	0.84	0.37	0.91	0.66	0.46	0.30
css.in.JS..jquery	0.83	0.97	0.71	0.42	0.73	0.32	0.00	0.74	0.00	0.12
max.lines	0.68	0.96	0.11	0.00	0.87	0.03	0.15	0.95	0.41	0.68
max.lines.per.function	0.77	0.21	0.28	0.00	1.00	0.46	0.34	0.97	0.02	0.71
max.params	0.95	0.24	0.30	0.26	0.02	0.63	0.00	0.96	0.20	0.84
(Ex.)complexity	0.95	0.10	0.31	0.48	0.05	0.71	0.00	0.96	0.07	0.84
max.depth	0.94	0.89	0.36	0.38	0.53	0.28	0.00	0.68	0.06	0.81
max.nested.callbacks	0.99	0.97	0.27	0.42	0.21	0.28	0.88	0.88	0.00	0.75

G.4.2.4 Transfer entropy lag1 (previous release)

Table G.18: Transfer entropy lag1 between CS and bugs - p-values for all apps

CS\TElag1 p-val	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	0.00	0.07	0.27	0.02	0.71	0.06	0.01	0.54	0.06	0.15
(Ex.)NPathComplexity	0.00	0.07	0.48	0.02	0.78	0.06	0.00	0.54	0.10	0.13
ExcessiveMethodLength	0.00	0.00	0.08	0.02	0.58	0.46	0.18	0.54	0.41	0.01
ExcessiveClassLength	0.00	0.00	0.04	0.18	0.72	0.29	0.05	0.48	0.00	0.61
ExcessiveParameterList	0.00	0.00	0.52	0.14	0.10	0.00	0.04	0.14	0.07	0.00
ExcessivePublicCount	0.00	0.00	0.03	0.09	0.93	0.13	0.14	0.30	0.38	0.00
TooManyFields	0.00	0.00	all cs=1	0.19	0.89	0.79	0.03	0.41	0.38	0.10
TooManyMethods	0.00	0.00	0.03	0.09	0.79	0.39	0.03	0.41	0.11	0.77
TooManyPublicMethods	0.00	0.16	0.23	0.22	0.57	0.22	0.05	0.47	0.06	0.69
ExcessiveClassComplexity	0.00	0.00	0.20	0.18	0.80	0.10	0.02	0.41	0.05	0.00
(Ex.)NumberOfChildren	0.00	0.00	0.00	0.01	0.04	0.71	0.10	all cs=1	0.01	0.36
(Ex.)DepthOfInheritance	0.00	0.00	0.00	0.00	0.39	0.00	0.00	0.00	0.00	0.00
(Ex.)CouplingBetweenObjects	0.02	0.00	0.00	0.02	0.02	0.24	0.04	0.47	0.05	0.04
DevelopmentCodeFragment	0.00	0.19	0.02	0.02	0.68	0.50	0.01	0.30	0.01	0.00
UnusedPrivateField	0.00	0.00	0.10	0.00	0.00	0.15	0.03	0.00	0.11	0.00
UnusedLocalVariable	0.32	0.09	0.24	0.05	0.97	0.25	0.01	0.41	0.29	0.51
UnusedPrivateMethod	0.00	0.00	0.10	0.62	0.19	0.15	0.05	0.14	0.03	0.09
UnusedFormalParameter	0.55	0.07	0.50	0.13	0.98	0.07	0.07	0.30	0.10	0.72
embed.JS	0.00	0.00	0.02	0.28	0.96	0.44	0.01	0.46	0.01	0.32
inline.JS	0.13	0.19	0.00	0.00	0.90	0.01	0.09	0.61	0.01	0.00
embed.CSS	0.00	0.19	0.75	0.00	0.36	0.46	0.02	0.61	0.10	0.00
inline.CSS	0.23	0.00	0.09	0.04	0.85	0.01	0.10	0.72	0.01	0.01
css.in.JS	0.14	0.00	0.48	0.01	0.99	0.02	0.01	0.43	0.20	0.06
css.in.JS..jquery	0.00	0.00	0.23	0.09	0.73	0.23	0.00	0.61	0.11	0.37
max.lines	0.08	0.03	0.00	0.24	0.81	0.06	0.00	0.20	0.22	0.09
max.lines.per.function	0.30	0.00	0.42	0.51	0.86	0.17	0.00	0.51	0.18	0.47
max.params	0.00	0.00	0.00	0.00	0.56	0.04	0.01	0.04	0.06	0.19
(Ex.)complexity	0.01	0.00	0.10	0.11	0.50	0.12	0.00	0.66	0.01	0.16
max.depth	0.00	0.00	0.08	0.00	0.87	0.72	0.00	0.08	0.14	0.16
max.nested.callbacks	0.00	0.10	0.70	0.12	0.54	0.07	0.19	0.01	0.00	0.12

Table G.19: Transfer entropy lag1 between CS and bugs - TE values(%) for all apps

CS\TElag1 te	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	10%	34%	22%	38%	7%	10%	14%	13%	42%	13%
(Ex.)NPathComplexity	15%	34%	23%	38%	3%	10%	16%	13%	46%	13%
ExcessiveMethodLength	11%	51%	37%	38%	7%	10%	14%	13%	32%	19%
ExcessiveClassLength	11%	52%	40%	20%	7%	11%	16%	13%	80%	2%
ExcessiveParameterList	10%	60%	14%	31%	16%	0%	18%	14%	52%	16%
ExcessivePublicCount	10%	51%	41%	23%	3%	15%	14%	13%	32%	0%
TooManyFields	10%	78%	100%	30%	3%	7%	17%	13%	32%	4%
TooManyMethods	13%	59%	41%	23%	7%	10%	17%	13%	36%	6%
TooManyPublicMethods	16%	30%	29%	27%	3%	13%	15%	13%	42%	4%
ExcessiveClassComplexity	10%	59%	31%	20%	3%	8%	13%	13%	57%	31%
(Ex.)NumberOfChildren	13%	50%	0%	37%	15%	6%	12%	100%	68%	10%
(Ex.)DepthOfInheritance	0%	0%	0%	0%	3%	0%	0%	0%	32%	0%
(Ex.)CouplingBetweenObjects	10%	43%	0%	33%	19%	11%	15%	13%	52%	21%
DevelopmentCodeFragment	12%	28%	46%	38%	7%	9%	18%	13%	55%	0%
UnusedPrivateField	13%	0%	36%	0%	23%	11%	17%	12%	36%	0%
UnusedLocalVariable	5%	28%	26%	38%	3%	13%	14%	13%	36%	7%
UnusedPrivateMethod	13%	0%	36%	2%	12%	10%	16%	14%	58%	7%
UnusedFormalParameter	5%	30%	21%	35%	3%	9%	15%	13%	46%	6%
embed.JS	13%	55%	33%	26%	3%	11%	13%	14%	58%	9%
inline.JS	8%	19%	53%	36%	3%	17%	13%	12%	58%	0%
embed.CSS	12%	19%	15%	20%	11%	9%	19%	11%	35%	15%
inline.CSS	7%	43%	37%	39%	4%	16%	10%	10%	61%	14%
css.in.JS	8%	48%	22%	38%	3%	19%	19%	12%	41%	19%
css.in.JS..jquery	10%	45%	31%	27%	7%	11%	33%	11%	50%	8%
max.lines	8%	26%	42%	30%	3%	17%	15%	19%	30%	13%
max.lines.per.function	6%	14%	22%	20%	3%	15%	21%	14%	29%	7%
max.params	15%	14%	53%	50%	7%	17%	21%	24%	49%	10%
(Ex.)complexity	10%	14%	37%	30%	6%	13%	24%	11%	66%	11%
max.depth	11%	30%	37%	24%	4%	6%	35%	24%	46%	11%
max.nested.callbacks	13%	18%	17%	23%	8%	17%	14%	12%	76%	14%

G.4.2.5 Transfer entropy lag2 (two releases before)

Table G.20: Transfer entropy lag2 between CS and bugs - p-values for all apps

CS\TElag2 p-val	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	0.06	0.42	0.11	0.57	0.15	0.02	0.54	0.77	0.26	0.65
(Ex.)NPathComplexity	0.04	0.42	0.99	0.57	0.31	0.02	0.37	0.77	0.80	0.83
ExcessiveMethodLength	0.00	0.51	0.10	0.57	0.27	0.54	0.59	0.77	0.56	0.41
ExcessiveClassLength	0.06	0.66	0.18	0.17	0.60	0.58	0.96	0.63	0.43	0.62
ExcessiveParameterList	0.03	0.00	0.53	0.55	0.28	0.00	0.51	0.32	0.00	0.05
ExcessivePublicCount	0.04	0.60	0.62	0.26	0.43	0.67	0.69	0.38	0.73	0.00
TooManyFields	0.11	0.36	all cs=1	0.27	0.57	0.32	0.06	0.72	0.73	0.11
TooManyMethods	0.04	0.62	0.62	0.25	0.62	0.50	0.88	0.72	0.00	0.98
TooManyPublicMethods	0.00	0.57	0.10	0.51	0.16	0.82	0.63	0.73	0.28	0.80
ExcessiveClassComplexity	0.00	0.36	0.09	0.04	0.36	0.07	0.07	0.77	0.33	0.15
(Ex.)NumberOfChildren	0.02	0.49	0.00	0.81	0.09	0.57	0.56	all cs=1	0.52	0.97
(Ex.)DepthOfInheritance	0.00	0.00	0.00	0.00	0.85	0.00	0.00	0.00	0.00	0.00
(Ex.)CouplingBetweenObjects	0.39	0.44	0.00	0.97	0.15	0.84	0.23	0.73	0.69	0.05
DevelopmentCodeFragment	0.00	0.89	0.24	0.57	0.40	0.71	0.21	0.38	0.11	0.00
UnusedPrivateField	0.00	0.00	0.23	0.00	0.04	0.68	0.88	0.00	0.05	0.00
UnusedLocalVariable	0.08	0.41	0.79	0.47	0.68	0.36	0.63	0.77	0.35	0.47
UnusedPrivateMethod	0.11	0.00	0.00	0.88	0.48	0.38	0.86	0.32	0.27	0.36
UnusedFormalParameter	0.00	0.51	0.85	0.41	0.20	0.00	0.86	0.38	0.00	0.21
embed.JS	0.07	0.81	0.61	0.42	0.33	0.34	0.50	0.25	0.15	0.38
inline.JS	0.00	0.30	0.00	0.01	0.21	0.29	0.78	0.33	0.15	0.00
embed.CSS	0.03	0.30	0.75	0.00	0.53	0.21	0.70	0.45	0.11	0.45
inline.CSS	0.01	0.18	0.09	0.00	0.54	0.00	0.28	0.68	0.48	0.55
css.in.JS	0.70	0.82	0.32	0.39	0.25	0.13	0.09	0.23	0.65	0.33
css.in.JS..jquery	0.02	0.06	0.13	0.90	0.88	0.39	0.11	0.45	0.14	0.70
max.lines	0.62	0.44	0.00	0.46	0.25	0.28	0.45	0.66	0.48	0.44
max.lines.per.function	0.36	0.00	0.79	0.20	0.47	0.63	0.05	0.42	0.14	0.88
max.params	0.01	0.00	0.00	0.61	0.65	0.23	0.05	0.77	0.04	0.77
(Ex.)complexity	0.22	0.00	0.00	0.40	0.58	0.69	0.08	0.91	0.19	0.57
max.depth	0.00	0.00	0.10	0.00	0.71	0.38	0.00	0.00	0.29	0.57
max.nested.callbacks	0.00	0.68	0.80	0.70	0.34	0.26	0.52	0.00	0.21	0.65

Table G.21: Transfer entropy lag2 between CS and bugs - TE values(%) for all apps

CS\TElag2 te	phpMyAdmin	DokuWiki	OpenCart	phpBB	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	15%	47%	54%	43%	23%	19%	13%	17%	57%	17%
(Ex.)NPathComplexity	15%	47%	20%	43%	18%	21%	13%	17%	37%	12%
ExcessiveMethodLength	11%	45%	62%	43%	18%	19%	19%	17%	35%	17%
ExcessiveClassLength	15%	45%	52%	46%	17%	19%	11%	19%	53%	2%
ExcessiveParameterList	11%	45%	39%	46%	23%	0%	23%	20%	53%	22%
ExcessivePublicCount	16%	37%	45%	47%	17%	16%	18%	16%	41%	0%
TooManyFields	12%	47%	100%	65%	17%	24%	33%	17%	41%	8%
TooManyMethods	22%	40%	45%	46%	14%	20%	12%	17%	53%	10%
TooManyPublicMethods	21%	41%	62%	51%	17%	15%	17%	17%	57%	8%
ExcessiveClassComplexity	17%	51%	55%	58%	19%	12%	12%	15%	57%	26%
(Ex.)NumberOfChildren	21%	42%	0%	33%	21%	19%	17%	100%	45%	6%
(Ex.)DepthOfInheritance	0%	0%	0%	0%	2%	0%	0%	0%	26%	0%
(Ex.)CouplingBetweenObjects	17%	46%	0%	35%	27%	13%	25%	17%	42%	31%
DevelopmentCodeFragment	15%	28%	50%	43%	20%	14%	23%	16%	53%	0%
UnusedPrivateField	21%	0%	56%	0%	22%	20%	13%	17%	57%	0%
UnusedLocalVariable	17%	43%	35%	53%	17%	25%	12%	15%	57%	17%
UnusedPrivateMethod	14%	0%	62%	4%	18%	17%	12%	20%	53%	7%
UnusedFormalParameter	26%	40%	33%	56%	24%	27%	14%	16%	53%	21%
embed.JS	21%	28%	27%	51%	17%	26%	11%	35%	51%	19%
inline.JS	27%	44%	62%	37%	22%	18%	11%	28%	51%	0%
embed.CSS	23%	44%	39%	34%	12%	22%	20%	28%	66%	11%
inline.CSS	23%	44%	62%	65%	17%	30%	17%	19%	53%	12%
css.in.JS	14%	22%	56%	40%	19%	26%	28%	28%	35%	26%
css.in.JS..jquery	16%	44%	54%	28%	14%	23%	34%	28%	53%	6%
max.lines	11%	31%	62%	55%	21%	24%	12%	22%	48%	15%
max.lines.per.function	17%	12%	27%	62%	14%	21%	33%	26%	66%	9%
max.params	22%	12%	62%	47%	17%	27%	32%	18%	47%	8%
(Ex.)complexity	16%	12%	63%	53%	15%	14%	33%	13%	66%	12%
max.depth	14%	27%	62%	35%	14%	22%	39%	20%	57%	12%
max.nested.callbacks	16%	15%	37%	28%	20%	26%	22%	24%	52%	19%

G.5 RQ5 – Impact CS time to release (TTR)

G.5.1 Causal inference

G.5.1.1 Linear Regression

Table G.22: Linear regression between CS and TTR - p-values for all apps

CS\LM p-value	phpMyAdmin	DokuWiki	OpenCart	phpBB	phpPgAdmin	MediaWiki	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	0.00	0.19	0.14	0.00	0.55	0.00	0.05	0.71	0.00	0.30	0.48	0.00
(Ex.)NPathComplexity	0.00	0.37	0.29	0.00	0.74	0.00	0.03	0.16	0.00	0.34	0.19	0.00
ExcessiveMethodLength	0.00	0.43	0.05	0.00	0.11	0.00	0.04	0.00	0.00	0.29	0.61	0.46
ExcessiveClassLength	0.00	0.41	0.61	0.01	0.17	0.00	0.13	0.00	0.39	0.10	0.19	0.00
ExcessiveParameterList	0.00	0.00	0.69	0.00	0.27	0.00	0.26	0.00	0.00	0.00	0.92	0.70
ExcessivePublicCount	0.00	0.22	0.02	0.43	0.58	0.06	0.10	0.63	0.00	0.00	0.10	0.00
TooManyFields	0.11	0.28	all cs=1	0.08	0.00	0.00	0.19	0.24	0.00	0.20	0.21	0.00
TooManyMethods	0.01	0.24	0.01	0.18	0.41	0.00	0.15	0.75	0.00	0.09	0.03	0.00
TooManyPublicMethods	0.00	0.11	0.32	0.14	0.32	0.11	0.05	0.16	0.00	0.24	0.06	0.26
ExcessiveClassComplexity	0.00	0.29	0.77	0.01	0.17	0.00	0.03	0.65	0.03	0.22	0.67	0.00
(Ex.)NumberOfChildren	0.00	0.00	0.00	0.78	0.00	0.00	0.89	0.99	0.72	all cs=1	0.67	0.09
(Ex.)DepthOfInheritance	0.00	0.00	0.00	0.00	0.01	0.00	0.94	0.00	0.00	0.00	0.00	0.00
(Ex.)CouplingBetweenObjects	0.00	0.00	0.00	0.78	0.00	0.00	0.20	0.67	0.53	0.23	0.03	0.35
DevelopmentCodeFragment	0.00	0.26	0.53	0.06	0.00	0.01	0.95	0.48	0.00	0.00	0.04	0.00
UnusedPrivateField	0.01	0.00	0.08	0.00	0.00	0.11	0.88	0.57	0.02	0.00	0.23	0.00
UnusedLocalVariable	0.00	0.79	0.08	0.01	0.53	0.00	0.30	0.35	0.00	0.40	0.37	0.19
UnusedPrivateMethod	0.00	0.00	0.32	0.00	0.00	0.00	0.43	0.26	0.00	0.00	0.02	0.00
UnusedFormalParameter	0.00	0.27	0.01	0.07	0.26	0.41	0.04	0.44	0.00	0.21	0.02	0.36
embed.JS	0.00	0.78	0.16	0.00	0.57	0.00	0.41	0.56	0.00	0.24	0.01	0.00
inline.JS	0.00	0.25	0.35	0.00	0.62	0.00	0.07	0.17	0.00	0.48	0.00	0.00
embed.CSS	0.00	0.05	0.71	0.03	0.00	0.00	0.54	0.51	0.47	0.32	0.01	0.00
inline.CSS	0.00	0.23	0.27	0.03	0.67	0.00	0.47	0.07	0.00	0.33	0.03	0.00
css.in.JS	0.35	0.44	0.63	0.00	0.85	0.00	0.70	0.47	0.00	0.23	0.39	0.63
css.in.JS..jquery	0.62	0.64	0.63	0.98	0.00	0.00	0.13	0.01	0.89	0.23	0.13	0.06
max.lines	0.38	0.77	0.07	0.00	0.12	0.00	0.56	0.04	0.00	0.85	0.07	0.07
max.lines.per.function	0.00	0.86	0.03	0.03	0.09	0.00	0.97	0.65	0.02	0.97	0.03	0.89
max.params	0.11	0.25	0.15	0.79	0.33	0.55	0.70	0.80	0.29	0.76	0.41	0.06
(Ex.)complexity	0.10	0.37	0.13	0.62	0.33	0.00	0.41	0.44	0.01	0.98	0.74	0.06
max.depth	0.20	0.88	0.24	0.00	0.00	0.65	0.58	0.27	0.04	0.22	0.11	0.06
max.nested.callbacks	0.07	0.00	0.02	0.99	0.00	0.00	0.54	0.12	0.00	0.61	0.06	0.14

G.5.1.2 Granger-causality lag1 (previous release)

Table G.23: Granger causality lag1 between CS and TTR - p-values for all apps

CS\GClag1 p-value	phpMyAdmin	DokuWiki	OpenCart	phpBB	phpPgAdmin	MediaWiki	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	0.00	0.43	0.18	0.28	0.37	0.00	0.29	0.49	0.00	0.08	0.65	0.06
(Ex.)NPathComplexity	0.01	0.58	0.57	0.23	0.60	0.00	0.31	0.10	0.00	0.09	0.12	0.11
ExcessiveMethodLength	0.00	0.70	0.16	0.20	0.10	0.00	0.14	0.00	0.00	0.08	0.19	0.28
ExcessiveClassLength	0.00	0.71	0.83	0.41	0.09	0.00	0.35	0.00	0.00	0.06	0.22	0.49
ExcessiveParameterList	0.00	0.08	0.70	0.06	0.40	0.00	0.20	1.00	0.00	0.28	0.44	0.93
ExcessivePublicCount	0.00	0.62	0.84	0.94	0.29	0.05	0.19	0.51	0.56	0.06	0.14	1.00
TooManyFields	0.14	0.95	1.00	0.77	0.70	0.00	0.87	0.35	0.27	0.06	0.44	0.28
TooManyMethods	0.02	0.85	0.78	0.57	0.20	0.00	0.37	0.05	0.00	0.05	0.02	0.64
TooManyPublicMethods	0.00	0.39	0.88	0.88	0.69	0.81	0.15	0.25	0.01	0.08	0.03	1.00
ExcessiveClassComplexity	0.00	0.78	0.92	0.43	0.66	0.00	0.27	0.82	0.00	0.07	0.91	0.48
(Ex.)NumberOfChildren	0.00	0.62	0.14	0.80	0.91	0.00	1.00	0.54	0.01	1.00	0.06	0.36
(Ex.)DepthOfInheritance	1.00	1.00	1.00	1.00	0.16	1.00	0.81	1.00	1.00	1.00	0.55	1.00
(Ex.)CouplingBetweenObjects	0.01	0.36	0.08	0.91	0.10	0.00	0.22	0.89	0.04	0.10	0.05	0.16
DevelopmentCodeFragment	0.00	0.54	0.53	0.57	1.00	0.01	0.28	0.86	0.88	0.05	0.02	1.00
UnusedPrivateField	0.00	1.00	0.07	1.00	1.00	0.13	0.43	0.42	0.01	0.44	0.11	1.00
UnusedLocalVariable	0.00	0.27	0.09	0.46	0.53	0.00	0.25	0.56	0.00	0.53	0.87	0.51
UnusedPrivateMethod	0.00	1.00	0.30	0.09	1.00	0.00	0.15	0.24	0.00	0.28	0.01	0.47
UnusedFormalParameter	0.00	0.79	0.08	0.48	0.39	0.03	0.90	0.65	0.00	0.06	0.01	0.84
embed.JS	0.00	0.32	0.20	0.10	0.21	0.00	0.17	0.31	0.00	0.56	0.07	0.00
inline.JS	0.00	0.70	0.29	0.12	0.92	0.00	0.05	0.23	0.35	0.91	0.06	1.00
embed.CSS	0.00	0.00	0.81	0.24	0.36	0.00	1.00	0.17	0.85	0.68	0.10	0.10
inline.CSS	0.00	0.00	0.19	0.26	0.43	0.06	0.12	0.03	0.00	0.72	0.20	0.20
css.in.JS	0.22	0.33	0.85	0.23	0.92	0.00	0.54	0.93	0.00	0.43	0.30	0.81
css.in.JS..jquery	0.99	0.38	0.79	0.27	0.10	0.00	0.32	0.03	0.78	0.54	0.02	0.07
max.lines	0.20	0.58	0.06	0.20	0.04	0.02	0.65	0.05	0.00	0.76	0.01	0.12
max.lines.per.function	0.00	0.26	0.11	0.58	0.45	0.00	0.82	0.07	0.31	0.77	0.01	0.55
max.params	0.15	0.18	0.18	0.45	0.13	0.76	0.69	0.71	0.03	0.96	0.04	0.07
(Ex.)complexity	0.14	0.13	0.19	0.22	0.11	0.00	0.41	0.98	0.06	0.74	0.11	0.07
max.depth	0.30	0.21	0.27	0.29	0.86	0.75	0.32	0.37	0.55	0.49	0.99	0.07
max.nested.callbacks	0.08	0.72	0.10	0.22	1.00	0.00	0.50	0.15	0.00	0.98	0.03	0.16

G.5.1.3 Granger-causality lag2 (two releases before)

Table G.24: Transfer entropy lag2 between CS and TTR - p-values for all apps

CS\GClag2 p-value	phpMyAdmin	DokuWiki	OpenCart	phpBB	phpPgAdmin	MediaWiki	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	0.50	0.65	0.29	0.23	0.69	0.00	0.52	0.45	0.00	0.27	0.82	0.22
(Ex.)NPathComplexity	0.65	0.53	0.40	0.27	0.83	0.00	0.45	0.12	0.00	0.38	0.34	0.22
ExcessiveMethodLength	0.15	0.69	0.11	0.26	0.14	0.00	0.51	0.00	0.00	0.27	0.22	0.33
ExcessiveClassLength	0.00	0.92	0.98	0.23	0.20	0.00	0.57	0.10	0.23	0.28	0.57	0.82
ExcessiveParameterList	0.54	0.33	0.87	0.16	0.53	0.00	0.73	1.00	0.00	0.54	0.25	0.99
ExcessivePublicCount	0.04	0.92	0.58	0.89	0.61	0.15	0.64	0.70	0.00	0.28	0.50	1.00
TooManyFields	0.41	0.71	1.00	0.15	0.89	0.00	0.42	0.82	0.00	0.22	0.88	0.60
TooManyMethods	0.42	0.78	0.55	0.56	0.43	0.00	0.78	0.03	0.00	0.25	0.10	0.68
TooManyPublicMethods	0.00	0.85	0.75	0.42	0.58	0.57	0.51	0.61	0.00	0.25	0.16	0.57
ExcessiveClassComplexity	0.01	0.82	0.91	0.20	0.37	0.00	0.40	0.86	0.00	0.22	1.00	0.80
(Ex.)NumberOfChildren	0.00	0.89	0.18	0.94	0.99	0.00	0.99	0.54	0.07	1.00	0.09	0.84
(Ex.)DepthOfInheritance	1.00	1.00	1.00	1.00	0.19	1.00	0.93	1.00	1.00	1.00	0.41	1.00
(Ex.)CouplingBetweenObjects	0.00	0.68	0.07	0.99	0.13	0.00	0.82	0.61	0.01	0.43	0.27	0.22
DevelopmentCodeFragment	0.10	0.41	0.65	0.83	1.00	0.01	0.39	0.24	0.01	0.26	0.12	1.00
UnusedPrivateField	0.40	1.00	0.10	1.00	1.00	0.01	0.90	0.17	0.04	0.58	0.22	1.00
UnusedLocalVariable	0.22	0.13	0.04	0.29	0.82	0.00	0.53	0.02	0.00	0.63	0.39	0.72
UnusedPrivateMethod	0.74	1.00	0.55	0.08	1.00	0.00	0.97	0.74	0.00	0.54	0.07	0.54
UnusedFormalParameter	0.19	0.86	0.05	0.68	0.50	0.05	0.32	0.79	0.00	0.20	0.05	0.39
embed.JS	0.18	0.29	0.28	0.26	0.48	0.04	0.93	0.06	0.00	0.82	0.27	0.00
inline.JS	0.12	0.67	0.49	0.31	0.90	0.01	0.64	0.75	0.52	0.71	0.26	1.00
embed.CSS	0.26	0.02	0.99	0.44	0.64	0.02	0.87	0.45	0.08	0.85	0.44	0.11
inline.CSS	0.33	0.00	0.37	0.53	0.05	0.25	0.90	0.23	0.00	0.71	0.17	0.35
css.in.JS	0.89	0.50	0.95	0.43	0.97	0.00	0.70	0.80	0.00	0.57	0.67	0.29
css.in.JS..jquery	0.75	0.43	0.92	0.27	0.12	0.00	0.56	0.20	0.98	0.69	0.12	0.06
max.lines	0.48	0.67	0.04	0.40	0.12	0.42	0.68	0.10	0.01	0.77	0.03	0.03
max.lines.per.function	0.15	0.31	0.12	0.29	0.19	0.00	0.90	0.03	0.38	0.88	0.06	0.08
max.params	0.97	0.46	0.21	0.07	0.35	0.94	0.92	0.78	0.65	0.99	0.03	0.02
(Ex.)complexity	0.97	0.33	0.26	0.11	0.26	0.00	0.88	0.89	0.14	0.94	0.09	0.02
max.depth	1.00	0.21	0.40	0.09	0.98	0.97	0.81	0.72	0.04	0.64	0.71	0.03
max.nested.callbacks	0.92	0.86	0.09	0.14	1.00	0.00	1.00	0.22	0.00	0.98	0.03	0.08

G.5.1.4 Transfer entropy lag1 (previous release)

Table G.25: Transfer entropy lag1 between CS and TTR - p-values for all apps

CS\TElag1 p-value	phpMyAdmin	DokuWiki	OpenCart	phpBB	phpPgAdmin	MediaWiki	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	0.01	0.59	0.00	0.10	0.66	0.00	0.11	0.08	0.00	0.48	0.48	0.05
(Ex.)NPathComplexity	0.02	0.52	0.61	0.08	0.40	0.00	0.11	0.08	0.00	0.48	0.58	0.02
ExcessiveMethodLength	0.00	0.56	0.08	0.08	0.20	0.00	0.12	0.01	0.01	0.48	0.37	0.07
ExcessiveClassLength	0.01	0.78	0.17	0.10	0.00	0.00	0.18	0.14	0.00	0.09	0.38	0.00
ExcessiveParameterList	0.00	0.00	0.00	0.13	0.08	0.00	0.77	0.00	0.01	0.00	0.36	0.00
ExcessivePublicCount	0.08	0.23	0.52	0.00	0.49	0.10	0.18	0.25	0.01	0.00	0.36	0.00
TooManyFields	0.01	0.55	all cs=1	0.14	0.00	0.04	0.13	0.07	0.05	0.05	0.58	0.00
TooManyMethods	0.13	0.62	0.52	0.00	0.50	0.00	0.46	0.00	0.00	0.05	0.06	0.00
TooManyPublicMethods	0.00	0.24	0.01	0.34	0.00	0.00	0.01	0.11	0.22	0.25	0.70	0.16
ExcessiveClassComplexity	0.00	0.59	0.07	0.20	0.03	0.00	0.42	0.05	0.00	0.09	0.84	0.09
(Ex.)NumberOfChildren	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.12	0.00	all cs=1	0.08	0.09
(Ex.)DepthOfInheritance	0.00	0.00	0.00	0.00	0.38	0.00	0.08	0.00	0.00	0.00	0.00	0.00
(Ex.)CouplingBetweenObjects	0.25	0.00	0.00	0.01	0.00	0.00	0.02	0.26	0.00	0.25	0.18	0.14
DevelopmentCodeFragment	0.00	0.25	0.05	0.00	0.00	0.34	0.46	0.13	0.10	0.00	0.26	0.00
UnusedPrivateField	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.03	0.00	0.11	0.00
UnusedLocalVariable	0.09	0.41	0.21	0.11	0.75	0.00	0.14	0.10	0.00	0.09	0.67	0.01
UnusedPrivateMethod	0.00	0.00	0.01	0.00	0.00	0.00	0.05	0.00	0.00	0.00	0.00	0.00
UnusedFormalParameter	0.71	0.11	0.00	0.37	0.00	0.46	0.94	0.15	0.00	0.01	0.00	0.01
embed.JS	0.08	0.57	0.00	0.14	0.23	0.03	0.11	0.01	0.00	0.42	0.09	0.06
inline.JS	0.00	0.55	0.00	0.21	0.05	0.00	0.02	0.03	0.03	0.56	0.21	0.00
embed.CSS	0.00	0.20	0.01	0.02	0.00	0.01	0.34	0.21	0.00	0.50	0.09	0.00
inline.CSS	0.00	0.23	0.01	0.04	0.10	0.00	0.00	0.00	0.00	0.56	0.55	0.00
css.in.JS	0.00	0.97	0.01	0.00	0.05	0.00	0.47	0.31	0.10	0.32	0.17	0.56
css.in.JS..jquery	0.00	0.48	0.22	0.00	0.00	0.00	0.66	0.00	0.59	0.50	0.25	0.08
max.lines	0.01	0.46	0.00	0.27	0.14	0.01	0.07	0.07	0.06	0.27	0.31	0.06
max.lines.per.function	0.50	0.23	0.60	0.32	0.45	0.00	0.13	0.10	0.41	0.46	0.21	0.01
max.params	0.00	0.25	0.00	0.07	0.06	0.05	0.05	0.17	0.58	0.05	0.13	0.03
(Ex.)complexity	0.00	0.25	0.06	0.01	0.16	0.00	0.05	0.17	0.42	0.29	0.17	0.08
max.depth	0.00	0.18	0.08	0.00	0.00	0.00	0.00	0.02	0.02	0.07	0.00	0.08
max.nested.callbacks	0.00	0.00	0.16	0.00	0.00	0.00	0.01	0.49	0.00	0.00	0.04	0.24

Table G.26: Transfer entropy lag1 between CS and bugs - TE values(%) for all apps

CS\TElag1 TE	phpMyAdmin	DokuWiki	OpenCart	phpBB	phpPgAdmin	MediaWiki	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	7%	9%	53%	17%	21%	15%	13%	9%	21%	12%	17%	17%
(Ex.)NPathComplexity	7%	9%	21%	17%	24%	14%	13%	9%	31%	12%	17%	20%
ExcessiveMethodLength	20%	10%	38%	18%	28%	16%	13%	21%	15%	12%	21%	15%
ExcessiveClassLength	7%	7%	32%	14%	41%	22%	12%	13%	26%	24%	23%	0%
ExcessiveParameterList	8%	0%	58%	16%	30%	18%	7%	0%	15%	0%	20%	17%
ExcessivePublicCount	5%	12%	21%	25%	21%	9%	12%	12%	18%	0%	22%	0%
TooManyFields	7%	7%	100%	14%	0%	12%	13%	16%	17%	24%	15%	0%
TooManyMethods	7%	7%	21%	25%	25%	13%	10%	30%	35%	24%	11%	0%
TooManyPublicMethods	20%	12%	53%	12%	31%	15%	17%	15%	11%	16%	15%	11%
ExcessiveClassComplexity	9%	9%	42%	13%	42%	15%	8%	10%	24%	20%	11%	15%
(Ex.)NumberOfChildren	21%	0%	0%	24%	0%	21%	23%	14%	20%	100%	28%	15%
(Ex.)DepthOfInheritance	0%	0%	0%	0%	29%	0%	12%	0%	0%	0%	0%	0%
(Ex.)CouplingBetweenObjects	6%	0%	0%	24%	0%	26%	16%	11%	22%	16%	18%	15%
DevelopmentCodeFragment	21%	14%	41%	34%	0%	7%	8%	13%	12%	0%	26%	0%
UnusedPrivateField	8%	0%	54%	0%	0%	14%	17%	12%	15%	0%	22%	0%
UnusedLocalVariable	7%	11%	29%	18%	17%	15%	13%	16%	22%	20%	18%	22%
UnusedPrivateMethod	9%	0%	52%	0%	0%	14%	15%	22%	46%	0%	36%	0%
UnusedFormalParameter	4%	12%	59%	13%	31%	6%	4%	7%	30%	32%	29%	22%
embed.JS	8%	8%	40%	16%	37%	12%	14%	21%	25%	13%	30%	14%
inline.JS	13%	8%	57%	15%	37%	16%	17%	14%	14%	12%	23%	0%
embed.CSS	21%	15%	54%	17%	0%	11%	12%	12%	22%	11%	30%	22%
inline.CSS	12%	15%	51%	12%	30%	13%	20%	14%	26%	12%	19%	18%
css.in.JS	14%	5%	51%	18%	33%	16%	10%	11%	14%	12%	30%	10%
css.in.JS..jquery	8%	6%	32%	24%	0%	10%	8%	28%	7%	11%	22%	13%
max.lines	9%	10%	42%	11%	31%	12%	15%	16%	12%	16%	19%	15%
max.lines.per.function	5%	7%	18%	13%	20%	17%	13%	17%	9%	13%	23%	22%
max.params	21%	7%	44%	14%	35%	11%	17%	13%	6%	24%	23%	16%
(Ex.)complexity	15%	7%	40%	25%	29%	15%	17%	11%	7%	16%	24%	13%
max.depth	11%	16%	38%	0%	0%	12%	20%	19%	15%	25%	31%	13%
max.nested.callbacks	11%	0%	34%	24%	0%	14%	20%	10%	36%	19%	24%	11%

G.5.1.5 Transfer entropy lag2 (two releases before)

Next, the p-values and percentages for Transfer entropy of individual CS from two previous releases of software to "time-to-release", for the various applications.

Table G.27: Transfer entropy lag2 between CS and TTR - p-values for all apps

CS\TElag2 p-value	phpMyAdmin	DokuWiki	OpenCart	phpBB	phpPgAdmin	MediaWiki	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	0.14	0.10	0.35	0.10	0.24	0.50	0.57	0.03	0.55	0.90	0.09	0.42
(Ex.)NPathComplexity	0.69	0.10	0.25	0.24	0.03	0.45	0.57	0.02	0.25	0.90	0.07	0.49
ExcessiveMethodLength	0.00	0.27	0.46	0.03	0.17	0.17	0.44	0.12	0.19	0.90	0.05	0.18
ExcessiveClassLength	0.03	0.22	0.16	0.48	0.18	0.02	0.41	0.30	0.14	0.32	0.29	0.00
ExcessiveParameterList	0.00	0.00	0.13	0.69	0.38	0.04	0.72	0.00	0.16	0.00	0.10	0.40
ExcessivePublicCount	0.14	0.19	0.65	0.20	0.04	0.83	0.12	0.21	0.14	0.00	0.57	0.00
TooManyFields	0.34	0.25	all cs=1	0.67	0.00	0.63	0.34	0.21	0.09	0.15	0.46	0.00
TooManyMethods	0.06	0.51	0.65	0.34	0.27	0.67	0.25	0.07	0.05	0.15	0.00	0.00
TooManyPublicMethods	0.01	0.13	0.44	0.87	0.00	0.45	0.15	0.29	0.19	0.14	0.18	0.06
ExcessiveClassComplexity	0.03	0.14	0.14	0.75	0.27	0.32	0.41	0.03	0.32	0.58	0.30	0.73
(Ex.)NumberOfChildren	0.91	0.00	0.00	0.85	0.00	0.00	0.13	0.60	0.32	all cs=1	0.40	0.54
(Ex.)DepthOfInheritance	0.00	0.00	0.00	0.00	0.81	0.00	0.61	0.00	0.00	0.00	0.00	0.00
(Ex.)CouplingBetweenObjects	0.75	0.00	0.00	0.94	0.00	0.00	0.70	0.78	0.32	0.14	0.01	0.36
DevelopmentCodeFragment	0.21	0.62	0.11	0.28	0.00	0.21	0.49	0.63	0.38	0.00	0.14	0.00
UnusedPrivateField	0.40	0.00	0.38	0.00	0.00	0.52	0.29	0.46	0.50	0.00	0.87	0.00
UnusedLocalVariable	0.26	0.50	0.25	0.15	0.13	0.16	0.49	0.36	0.09	0.58	0.42	0.55
UnusedPrivateMethod	0.12	0.00	0.00	0.00	0.00	0.27	0.70	0.14	0.00	0.00	0.33	0.00
UnusedFormalParameter	0.73	0.23	0.44	0.89	0.00	0.49	0.13	0.01	0.38	0.04	0.00	0.55
embed.JS	0.07	0.03	0.56	0.36	0.35	0.05	0.12	0.08	0.17	0.44	0.29	0.03
inline.JS	0.73	0.52	0.00	0.65	0.13	0.06	0.12	0.05	0.03	0.45	0.66	0.00
embed.CSS	0.52	0.19	0.59	0.01	0.00	0.48	0.71	0.56	0.91	0.47	0.29	0.05
inline.CSS	0.50	0.06	0.42	0.01	0.02	0.00	0.28	0.07	0.22	0.07	0.66	0.05
css.in.JS	0.92	0.07	0.35	0.00	0.04	0.07	0.36	0.15	0.34	0.63	0.41	0.55
css.in.JS..jquery	0.33	0.40	0.05	0.56	0.00	0.29	0.59	0.06	0.82	0.47	0.44	0.20
max.lines	0.82	0.89	0.00	0.71	0.05	0.03	0.55	0.55	0.61	0.01	0.68	0.55
max.lines.per.function	0.93	0.52	0.13	0.71	0.20	0.00	0.37	0.43	0.88	0.24	0.10	0.05
max.params	0.87	0.57	0.00	0.46	0.30	0.54	0.58	0.34	0.70	0.13	0.19	0.05
(Ex.)complexity	0.88	0.57	0.00	0.34	0.23	0.08	0.58	0.71	0.89	0.00	0.06	0.57
max.depth	0.02	0.10	0.46	0.00	0.00	0.05	0.50	0.21	0.40	0.00	0.00	0.57
max.nested.callbacks	0.16	0.00	0.39	0.58	0.00	0.00	0.41	0.59	0.05	0.00	0.02	0.86

Table G.28: Transfer entropy lag2 between CS and TTR - TE values(%) for all apps

CS\TElag2 te	phpMyAdmin	DokuWiki	OpenCart	phpBB	phpPgAdmin	MediaWiki	PrestaShop	Vanilla	Dolibarr	Roundcube	OpenEMR	Kanboard
(Ex.)CyclomaticComplexity	11%	34%	40%	35%	66%	16%	18%	19%	20%	11%	60%	20%
(Ex.)NPathComplexity	7%	33%	70%	28%	74%	18%	18%	23%	21%	11%	67%	18%
ExcessiveMethodLength	8%	29%	49%	36%	51%	20%	20%	29%	23%	11%	61%	22%
ExcessiveClassLength	14%	28%	55%	20%	57%	20%	22%	26%	31%	26%	54%	0%
ExcessiveParameterList	13%	0%	59%	18%	46%	19%	18%	0%	19%	0%	61%	13%
ExcessivePublicCount	11%	22%	46%	28%	59%	13%	29%	27%	29%	0%	35%	0%
TooManyFields	8%	22%	100%	19%	0%	18%	22%	27%	31%	33%	41%	0%
TooManyMethods	19%	18%	46%	24%	53%	12%	27%	34%	32%	33%	48%	0%
TooManyPublicMethods	14%	21%	49%	14%	36%	16%	27%	25%	28%	34%	60%	27%
ExcessiveClassComplexity	12%	35%	57%	15%	59%	17%	18%	20%	20%	18%	51%	13%
(Ex.)NumberOfChildren	7%	0%	0%	18%	0%	22%	27%	20%	21%	100%	40%	13%
(Ex.)DepthOfInheritance	0%	0%	0%	0%	47%	0%	14%	0%	0%	0%	0%	0%
(Ex.)CouplingBetweenObjects	11%	0%	0%	19%	0%	22%	14%	14%	19%	34%	60%	18%
DevelopmentCodeFragment	4%	17%	58%	33%	0%	20%	18%	16%	19%	0%	62%	0%
UnusedPrivateField	10%	0%	49%	0%	0%	16%	13%	23%	21%	0%	30%	0%
UnusedLocalVariable	12%	22%	61%	35%	63%	21%	20%	26%	26%	18%	50%	16%
UnusedPrivateMethod	12%	0%	49%	0%	0%	10%	14%	23%	33%	0%	45%	0%
UnusedFormalParameter	10%	19%	55%	15%	39%	17%	25%	25%	23%	32%	39%	16%
embed.JS	18%	38%	28%	27%	58%	24%	26%	35%	19%	24%	51%	32%
inline.JS	10%	18%	40%	21%	61%	23%	26%	27%	24%	23%	37%	0%
embed.CSS	12%	27%	49%	31%	0%	12%	20%	16%	12%	23%	51%	23%
inline.CSS	12%	39%	49%	19%	66%	20%	19%	22%	21%	38%	42%	26%
css.in.JS	9%	41%	59%	18%	63%	21%	24%	26%	26%	15%	48%	21%
css.in.JS..jquery	8%	10%	66%	19%	0%	15%	21%	33%	13%	23%	39%	16%
max.lines	8%	11%	49%	18%	69%	18%	18%	19%	15%	46%	32%	14%
max.lines.per.function	8%	8%	65%	21%	65%	23%	24%	26%	13%	30%	55%	29%
max.params	8%	10%	40%	19%	51%	18%	19%	25%	16%	37%	44%	27%
(Ex.)complexity	8%	10%	49%	25%	58%	21%	19%	13%	12%	53%	54%	12%
max.depth	11%	33%	49%	0%	0%	18%	19%	25%	17%	36%	59%	12%
max.nested.callbacks	8%	0%	60%	18%	0%	22%	18%	20%	24%	31%	59%	15%

The values for lag 3 and 4, for all the studies, are the the replication package, available online.