An Empirical Analysis on the Relationship between Code Smells and Fault-Proneness In JavaScript Projects: A Replication Study

Kevin Pacifico,[1] Giammaria Giordano,[1] Valeria Pontillo,[2]
Massimiliano Di Penta,[3] Damian Andrew Tamburri[3,4] Fabio Palomba[1]

[1] *Software Engineering (SeSa) Lab - Department of Computer Science, University of Salerno, Italy*
giagiordano@unisa.it, fpalomba@unisa.it,
[2] *Gran Sasso Science Institute (GSSI), L'Aquila, Italy*
valeria.pontillo@gssi.it
[3] *University of Sannio, Italy*
dipenta@unisannio.it, datamburri@unisannio.it
[4] *JADS/NXP Semiconductors, Netherlands*

Corresponding author:

An Empirical Analysis on the Relationship between Code Smells and Fault-Proneness In JavaScript Projects: A Replication Study

Kevin Pacifico,[1] Giammaria Giordano,[1] Valeria Pontillo,[2]
Massimiliano Di Penta,[3] Damian Andrew Tamburri[3,4] Fabio Palomba[1]

[1] *Software Engineering (SeSa) Lab - Department of Computer Science, University of Salerno, Italy*
giagiordano@unisa.it, fpalomba@unisa.it,
[2] *Gran Sasso Science Institute (GSSI), L'Aquila, Italy*
valeria.pontillo@gssi.it
[3] *University of Sannio, Italy*
dipenta@unisannio.it, datamburri@unisannio.it
[4] *JADS/NXP Semiconductors, Netherlands*

Abstract

Lorem Ipsum is simply dummy text of the printing and typesetting industry. Lorem Ipsum has been the industry's standard dummy text ever since the 1500s, when an unknown printer took a galley of type and scrambled it to make a type specimen book. It has survived not only five centuries, but also the leap into electronic typesetting, remaining essentially unchanged. It was popularised in the 1960s with the release of Letraset sheets containing Lorem Ipsum passages, and more recently with desktop publishing software like Aldus PageMaker including versions of Lorem Ipsum. Contrary to popular belief, Lorem Ipsum is not simply random text. It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source. Lorem Ipsum comes from sections 1.10.32 and 1.10.33 of "de Finibus Bonorum et Malorum" (The Extremes of Good and Evil) by Cicero, written in 45 BC. This book is a treatise on the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum, "Lorem ipsum dolor sit amet..", comes from a line in section 1.10.32.

Keywords: TBD

1. Introduction

2. Background and Related Work

This section provides the necessary information to understand the rest of this paper and survey the state-of-the-art in the context of code smells.

2.1. Background

Table 1 shows the code smells detectable by the framework of Johannes *et al.* [1] with a brief description. The code smells detectable by the framework encompasses multiple categories of potential issues, including *syntactic concerns* (*e.g.,* Lengthy Lines, Chained Methods), *structural complexity* (*e.g.,* Nested Callbacks, Depth Smell), and *semantic misuses* (*e.g.,* Variable Re-assign, Assignment in Conditional Statements).

To give a tangible example of code smells, let us consider *Variable Re-assign* smell. This smell occurs when developers reuse the same variable but change the type or semantic role of the value it holds within the same scope. Such practice can significantly reduce code readability and increase the likelihood of runtime errors. Listing 1 shows an example of *Variable Re-assign* smell.

Table 1: Code Smells Detectable by the Framework of Johannes *et al.*

Code Smell	Description
Lengthy Lines	Occur when a single line of code contains too many characters.
Chained Methods	This smell occurs when there is excessive use of method chaining—repeatedly invoking multiple methods in a single statement.
Long Parameter List	This smell arises when a function is defined with too many parameters, which can complicate its usage, reduce readability, and increase the likelihood of errors.
Nested Callbacks	This smell arises in the code when multiple asynchronous tasks are executed in sequence.
Variable Re-assign	The smell refers to when a variable is re-assigned with a different type in the same scope.
Assignment in Conditional Statements	The smell occurred when the assign operator is used in an "if" statement.
Complex Code	This smell appears when a JavaScript file exhibits high cyclomatic complexity, meaning the code contains a large number of linearly independent paths.
Extra Bind	This smell typically occurs when a function is unnecessarily bound using .bind(ctx) even after the this keyword has been removed from its body.
This Assign	This smell occurs when the "this" keyword is assigned to another variable (e.g., var self = this) in order to access the parent scope's context.
Long Method	This smell occurs when a method is composed of too many lines of code.
Complex Switch Case	This smell refers to when there are too many switch case statements.
Depth Smell	This smell occurs when the code contains too many nested blocks or deep levels of indentation.

To illustrate the *Variable Re-assign* code smell, consider the following JavaScript example:

```
1 function calculateDiscount(userInput) {
2   let discount = userInput; // discount
        holds a string (e.g., "20")
3
4   if (!isNaN(discount)) {
5     discount = parseFloat(discount); //
```

```
        now a number
  }

  if (discount > 12) {
    discount = "MAX";// now a string
        again
  }

  return discount;
}

console.log(calculateDiscount("60")); //
    Outputs: "MAX"
```

Listing 1: Example of Variable Re-assign Code Smell.

As we can see from the example, the variable DISCOUNT changes multiple times—first holding a string, then a number, and finally being reassigned to a string again. This change can potentially decrease code clarity and maintainability, as the variable's type and purpose become ambiguous throughout the function.

A possible mitigation strategy in this case consists of avoiding variable reuse for semantically distinct purposes. For example, the original user input can be stored in one variable (*e.g.,* RAWINPUT), the parsed numeric value in another (*e.g.,* NUMERICDISCOUNT), and any final label or status in a third (*e.g.,* FINALDISCOUNT). As illustrated in Listing 2.

```
function calculateDiscount(userInput) {
  const rawInput = userInput; // original
      input as string
  const numericDiscount = parseFloat(
      rawInput); // parsed numeric value

  let finalDiscount;
  if (!isNaN(numericDiscount) &&
      numericDiscount > 12) {
    finalDiscount = "MAX";  // label
        assigned
  } else {
    finalDiscount = numericDiscount; //
        return the numeric value
  }

  return finalDiscount;
}

console.log(calculateDiscount("60")); //
    Outputs: "MAX"
```

Listing 2: Refactored Version Avoiding Variable Re-assign

2.2. State-of-the-art

Fowler and Beck defined code smells as indicators of suboptimal software design that may suggest the presence of deeper structural issues within the source code [2]. In the last decades, code smells have been investigated from different perspectives. In the context of Java systems, code smells were investigated from multiple angles. From an evolutionary standpoint, Giordano *et al.* investigated their relationship with respect to design patterns [3] and inheritance and delegation [4]. In both cases, they discovered that some code smells are positively correlated with best practices. Tufano *et al.* [5] investigates *when* and *why* code smells evolve in Java systems, discovering that smells are commonly introduced in the first stages of the project, and their removal is typically related to file removal. Li and Shatnawi [6] investigated the relationship between code smells and bugs across three versions of Eclipse and revealed a positive correlation between the presence of code smells and an increased likelihood of errors. Sjoberg *et al.* [7] investigated the relationship between code smells and maintenance effort, concluding that code smells have a limited impact on maintenance activities. In contrast, Abbes *et al.* [8] found that the presence of code smells can adversely affect code understandability. Khomh *et al.* [9] examined the relationship between code smells and both change- and fault-proneness across 54 releases of four widely used Java open-source systems—ArgoUML, Eclipse, Mylyn, and Rhino. Their findings indicate that classes affected by code smells are generally more susceptible to changes and faults compared to those without such smells.

Moving on to JavaScript-specific code smells, several tools have been developed over the years. Fard *et al.* [10] introduced a technique called JNOSE for detecting 13 distinct types of code smells in JavaScript systems. Their study identified lazy object and long method as the most prevalent code smells across the analyzed systems. Nguyen *et al.* [11] developed a tool that focuses on detecting issues such as the intermixing of HTML, CSS, and JavaScript. Additionally, tools such as ESLint [1], JSLint [12], and JSHint [13] utilize rule-based static code analysis to validate source code against established best practices.

In comparing our replication study with the original work by Johannes *et al.* [1], the primary differences lie in the expansion of the experimental dataset. Specifically, we extended the scope of the original study by analyzing 50 projects, as opposed to the 15 considered in the original work, and by selecting a broader project domain. Through this replication, our objective is to either confirm or refute their findings, thereby contributing to the robustness and generalizability of their conclusions.

3. Research Method

The *goal* of this study is to investigate the evolution of code in JavaScript systems, and their relationship with fault-proneness. The *quality focus* of this investigation is the fault proneness of the source code, as greater fault proneness can significantly impact the cost of software maintenance and evolution.

The study is carried out from the *perspective* of practitioners and researchers. Practitioners are interested in identifying development practices that may lead to the accidental introduction of code smells, as well as understanding how these smells relate to the quality and maintainability of software systems. Researchers, on the other hand, aim to deepen the understanding of code smells by analyzing their characteristics and investigating their potential relationship with fault-proneness in software systems.

The *context* of this study is based on the analysis of 12 types of code smells detectable using the framework of Johannes *et al.* [1] across 50 real-world JavaScript systems.

In compliance with our goal, we formulated the following research questions:

> **RQ$_1$. On code smell survivability.**
>
> *What is the lifespan of code smells in JavaScript projects?*

The **RQ1** explores the longevity of code smells by analyzing when they are introduced and how long they persist in the code. The study by Johannes *et al.* conducted on 15 JavaScript projects, revealed that many code smells are present from the moment a file is created and tend to persist over time. Among them, *Variable Re-assign* emerged as the most long-lived code smell. By expanding the analysis to a larger number of projects, it will be possible to assess whether the lifespan of code smells varies depending on the type or complexity of the project.

> **RQ$_2$. Comparison of Fault-Proneness Between Smelly and Non-Smelly Files.**
>
> *Is the risk of fault higher in files containing code smells compared to files without smells?*

The goal of the **RQ2** is to compare the time to failure between JavaScript files that contain code smells and those that do not. The study conducted by Johannes *et al.* [1] on 15 projects revealed that files without code smells have a 33% lower risk of failure. When accounting for dependencies between code elements, the risk reduction increases to 45%. By expanding the analysis to 50 projects, it will be possible to confirm or revise these percentages using a broader and more diverse dataset.

> **RQ$_3$. On the equally of code smells in fault-proneness.**
>
> *Are JavaScript files containing code smells equally fault-prone?*

The **RQ3** aims to identify which code smells have the greatest impact on software quality, helping to determine which should be prioritized during refactoring. The study conducted by Johannes *et al.* found that the smells *Variable Re-assign*, *Assignment in Conditional Statements*, and *Complex Code* are among those most strongly associated with a high risk of failure. By extending the analysis to 50 projects, it will be possible to verify whether these categories remain critical or if new risk patterns emerge.

When conducting our empirical experiment, we adopted the software engineering practices described by Wholin *et al.* [14]. In terms of reporting, we leverage the *ACM/SIGSOFT Empirical Standards*[1], specifically, given the connotation of our study, we used the "General Standard", "Data Sicence", and "Repository Mining" guidelines.

First, we selected JavaScript repositories from GitHub using GitHub Search [15], then, we ran the framework of Johannes *et al.* [1] and extracted information about smells, including information about the introduction and removal of the smell, and information about fault. Once the data were extracted, the framework applied the *Cox Proportional Hazards* (COX) model [16] to evaluate the correlation between smells and fault-proneness from a temporal point of view.

Figure 1 overviews the research method of this study. In the next sections, we will elaborate in detail on the steps applied to perform our study.

3.1. Project Selection

Aware that project selection is one of the most critical phases of the study, we took steps to mitigate potential threats to validity by using GitHub Search [15]—a specialized platform designed to identify suitable open-source projects hosted on GitHub. Given the focus of our analysis, we restricted the selection to projects written in JavaScript. At the end of this step, we identified 50 projects. Table 2 provides the statistical description of the projects analyzed.

Table 2: Statistical Description of Projects Analyzed.

Statistic	Issues	Stars	Contributors	LOC
Mean	1,143.06	17,553.28	199.14	138,621.4
Standard Deviation	768.18	15,614.79	154.98	206,071.1
Min	2	864	39	3,041
25th Percentile	648.25	10,425	90	29,808.25
Median (50%)	1,227	14,900	157.5	65,314
75th Percentile	1,504.75	19,800	265.5	146,356
Max	4,752	106,000	836	1,252,611

As can be seen from the table, the distribution of project characteristics is notably skewed. For instance, while the median number of stars is 14,900, the maximum reaches 106,000, suggesting the presence of outliers. The number of contributors also shows significant variability, with a median of 157.5 and a maximum of 836. The distribution of lines of code is particularly broad: although the median project has around 65,314 LOC, some projects reach over 1.2 million, while others remain below 10,000. These results highlight the heterogeneous nature of the dataset, which includes both lightweight and extremely large-scale projects in terms of popularity, collaboration, and codebase size.

3.2. Survival Analysis and the Cox Proportional Hazards Model

The framework employs survival analysis to model the time until the occurrence of well-defined events, such as the introduction of a code smell or the appearance of a fault. Among the various techniques available for this purpose, the Cox Proportional Hazards Model was selected due to its flexibility and ability to incorporate multiple explanatory variables without requiring strong assumptions about the baseline hazard distribution.

To estimate the risk associated with faults or code smells, the model defines the hazard function as $\lambda_i(t) = \lambda_0(t) \cdot e^{\beta \cdot F_i(t)}$, where

[1] https://github.com/acmsigsoft/EmpiricalStandards

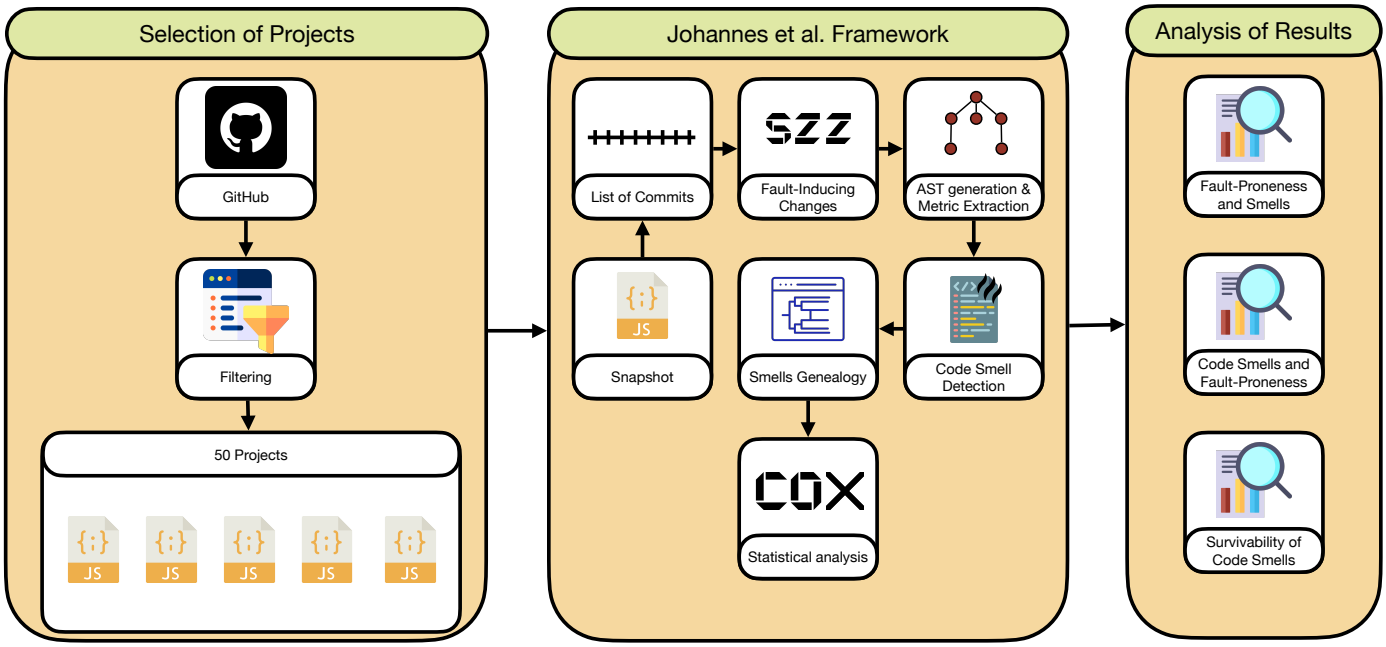

Figure 1: Research Method Overview.

$\lambda_i(t)$ represents the instantaneous risk for file i at time t, $\lambda_0(t)$ is the baseline hazard, $F_i(t)$ is the vector of covariates associated with the file at that time, and β denotes the coefficients indicating the effect size of each covariate. The model assumes a linear relationship between the log-hazard and the covariates. When this assumption does not hold—for instance, when the effect of a covariate changes over time—a link function is applied to transform the variables appropriately, thereby maintaining the model's validity.

To further ensure that the proportional hazards assumption is satisfied, the framework performs non-proportionality tests prior to model fitting. Additionally, stratification is employed to control for confounding variables that are not of direct interest by grouping observations into homogeneous subsets. This approach enables a more accurate estimation of the effects of primary variables while accounting for background variability.

The relative risk between two files i and j at a specific point in time is computed as $\lambda_i(t)/\lambda_j(t) = e^{\beta\cdot(F_i(t)-F_j(t))}$. This formulation reflects the multiplicative nature of the model, where each covariate contributes proportionally to the hazard, assuming its effect remains constant over time.

The survival analysis is applied across all commits and files in the studied projects. For each file, the model estimates the hazard of experiencing either a fault or the introduction of a code smell, based on independent variables such as the presence of smells, file size (measured in lines of code), and code complexity.

The Cox model offers several advantages that make it particularly well-suited for this study. It accommodates censored data, meaning that files that never experience a fault or smell within the observation window can still be included in the analysis. It also supports comparisons between smelly and non-smelly files by incorporating binary covariates into the hazard function. Furthermore, the model captures the evolution of file characteristics over time, allowing for longitudinal analysis. Finally, because files may exhibit multiple faults throughout their lifespan, the model's ability to handle recurrent events is essential for accurately characterizing fault dynamics in evolving software systems.

3.3. Framework Execution

Once the project selection was completed, we utilized the framework of Johannes et al. [1] to extract information about faults, code smells, and survivability analysis. The tool is equipped with three dedicated modules to detect these types of information.

RQ1. Code Smell Identification and Survivability. To detect code smells, the framework generates an abstract syntax tree (AST) and analyzes the code using the *ESLint*[2] library. For each commit, it checks for the presence of code smells as defined in Section 2, using threshold-based criteria. The severity of a smell is computed as the degree to which it exceeds the corresponding threshold.

The framework tracks all commits that modify smelly files, enabling the monitoring of smell evolution over time. Specifically, given two consecutive commits, C_1 and C_2, on a file F, if a smell appears in C_2 but is absent in C_1, then C_2 is marked as a smell-introducing commit. Conversely, if a smell is present in C_1 but not in C_2, C_2 is marked as a smell-removal commit. If a smell is never removed, it is assumed to persist throughout the lifespan of the project.

To assess similarity between smells, the framework considers two factors: (1) whether the smell categories match, and (2) a textual similarity score between descriptions. If the categories

[2] `eslint.org/`

align, a similarity score ranging from 0 to 1 is computed. If this score exceeds a predefined threshold, the smells are considered equivalent.

RQ2. Comparison of Fault-Proneness Between Smelly and Non-Smelly Files. To address **RQ2**, a survival analysis was conducted to compare the time to failure between files that contain code smells and those that do not. The Cox proportional hazards model was employed to estimate the relative risk of failure as a function of multiple independent variables. The analysis was performed at both the commit level and the line-of-code level.

Files are categorized into two groups *i.e.,* those containing at least one of the 12 types of code smells considered, and those that are smell-free. For each commit, the framework records both the time between revisions and the presence or absence of code smells. The Smelly metric is used as an independent variable to quantify the impact of smells on failure risk.

To ensure that the failures analyzed are truly associated with code smells, the framework incorporates the SZZ Fischer implementation [17] algorithm, which traces each defect back to the commit in which it was introduced. Bug-introducing commits that focus exclusively on JavaScript files, ignoring commits that modify only empty or comment lines. To reduce false positives, the tool applies the Median Absolute Deviation (MAD) metric proposed by Da Costa *et al.* [18], filtering out anomalous values produced by SZZ. This enables verification of the co-occurrence between faults and code smells.

Once a bug-introducing commit is identified, the framework uses the *diff* command to compare versions of the affected file and extract a set of *candidate fault lines*. Finally, it generates an Abstract Syntax Tree (AST) to determine dependencies among variables, functions, and modules, enabling the identification of *extended candidate fault lines*, *i.e.,* external modules or functions potentially impacted by the fault.

Finally, the difference in failure rates between smelly and non-smelly files is analyzed to determine whether code smells are associated with a statistically significant increase in failure risk.

RQ3. On the equality of code smells in fault-proneness. To evaluate whether all code smells contribute equally to fault-proneness, the framework adopts a methodology similar to that used for **RQ2**, but applies it in a more fine-grained and differentiated manner. Rather than considering code smells as a single aggregated factor, this analysis models the effect of each individual code smell category separately, enabling the assessment of the specific impact each type of smell has on fault occurrence.

For each smell category, the framework defines a binary independent variable, denoted as *Smelly*, which indicates whether the corresponding smell is present in a given file at a particular commit. This disaggregated representation allows the survival model to estimate the unique contribution of each smell type to the overall hazard of fault introduction.

In addition to the presence or absence of specific smells, the model accounts for several control variables that are known to influence fault likelihood. These include the file's size, measured in lines of code (LOC); the extent of code churn, captured by the number of lines added, removed, or modified across commits; and the number of previously reported faults associated with the file, which serves as a proxy for its historical defectiveness.

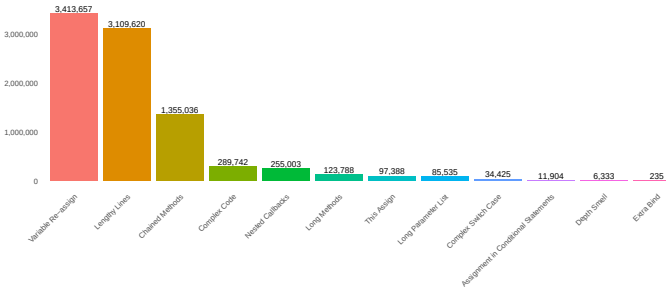

Figure 2: Diffusion of Code Smells in JavaScript Systems.

For each subject system, the framework constructs a Cox proportional hazards model incorporating both the individual smell variables and the aforementioned covariates. Prior to estimation, the framework performs standard tests for non-proportionality to verify that the assumptions underlying the model are satisfied and that the hazard ratios can be interpreted reliably.

This approach makes it possible to determine whether certain code smells are more strongly correlated with software faults than others, and whether some types of smells represent a disproportionately higher risk to software reliability. By isolating and quantifying the effect of each smell type, the framework offers detailed insights into the relative severity of different forms of technical debt, thereby supporting more targeted and effective refactoring strategies.

4. Analysis of the Results

This section describes the main results of our research question. To increase readability, this section will discuss the main findings of each **RQ** individually.

Before analyzing the **RQs** results, we provide some preliminary considerations regarding the diffusion of smells into JavaScript systems. Figure 2 illustrates the diffusion of code smells into the analyzed projects.

As it is possible to see from the figure, the three most dominant smells are *Variable Re-assign*, *Lengthy Lines*, and *Chained Methods*, respectively. These three categories represent the vast majority of occurrences, indicating that stylistic and structural issues such as variable reassignments and overly long or complex statements are particularly prevalent across the analyzed codebases. Following these, other smells such as *Complex Code*, *Nested Callbacks*, and *Long Methods* appear with significantly lower frequency. These patterns suggest that while deeper structural issues exist, they are less common compared to more surface-level readability concerns. At the lower end of the spectrum, smells like *Assignment in Conditional Statements*, *Depth Smell*, and *Extra Bind* are relatively rare, potentially reflecting either better developer awareness of these

issues or limited applicability depending on the programming paradigm used.

4.1. RQ1 – Survivability of Code Smells.

Table 3: Survivability and Status of Entities per Project.

Project	% Removed	% Active	Mean Survivability (Days)	Mean Survivability (Commits)	% Smells from Day One
JSDoc	99.28%	0.72%	425	424	59.65%
ExcelJS	97.86%	2.14%	618	342	62.25%
Axios	86.43%	13.57%	599	204	16.38%
Chrome Extensions Samples	9.59%	90.41%	3228	1048	95.48%
Flux	72.49%	27.51%	604	115	70.28%
Blockly Samples	41.82%	58.18%	776	725	81.06%
Highlight.js	99.36%	0.64%	825	759	56.21%
Handlebars.js	39.5%	60.5%	3350	1212	53.66%
Jasmine	96.11%	3.89%	813	373	42.36%
Prism	27.46%	72.54%	1448	1329	51.16%
Async	74.4%	25.6%	989	351	50.13%
Karma	92.87%	7.13%	441	504	81.2%
Browserify	78.83%	21.17%	282	504	36.48%
JQuery Validation	50.4%	49.6%	2264	303	78.4%
Emotion	88.37%	11.63%	413	220	51.16%
Forever	76.04%	23.96%	755	160	37.99%
Gulp	66.67%	33.33%	520	207	23.73%
Validator.js	100%	0%	394	159	30.59%
LocalForage	76.46%	23.54%	1005	200	71.69%
Markdown	100%	0%	1201	328	34.75%
Popmotion	93.45%	6.55%	170	155	59.95%
Stf	45%	55%	1085	891	41.31%
Reactstrap	97.35%	2.65%	504	261	94.64%
Gpu.js	98.77%	1.23%	319	269	51.33%
Sweetalert2	99.62%	0.38%	122	155	50.35%
Nodemailer	99.85%	0.15%	285	84	49.46%
Fetch	47.79%	52.21%	829	154	8.56%
Anime	26.81%	73.19%	1538	235	75.84%
Standard	64.94%	35.06%	469	369	46.82%
Sortable	48.02%	51.98%	1136	171	35.3%
Winston	91.71%	8.29%	1007	384	38.34%
Ws	99.96%	0.04%	919	399	26.79%
Yargs	97.09%	2.91%	316	289	19.89%
Piskel	60.58%	39.41%	1522	525	66.94%
Katex	98.83%	1.17%	365	161	55.77%
Prepack	23.08%	76.92%	999	730	85.03%
Ungit	99.99%	0.01%	64	139	96.22%
Recompose	94.12%	5.88%	221	214	47.06%
Places	98.63%	1.37%	210	98	100%
Just	99.01%	0.99%	59	58	96.05%
Razzle	21.05%	78.95%	686	703	93.78%
Bpmn-js	98.68%	1.32%	517	351	50.02%
Artillery	44.24%	55.76%	469	512	83.47%
Gridsome	61.54%	38.46%	554	600	89.74%
Parsley.js	50.03%	49.97%	1732	438	77.53%
NLP.js	4.12%	95.88%	1409	1479	99.85%
Aura	97.87%	2.13%	202	109	79.72%
VisBug	100%	0%	832	1069	100%
A Dark Room	77.76%	22.24%	1067	226	31.42%
Diagram.js	100%	0%	442	216	46.51%

Table 3 provides an overview of smell survivability across the analyzed projects. It reports the percentage of smells that were removed or remained active at the time of the last analyzed commit, along with the average survivability of smells expressed in both days and commits.

Based on the percentage of smells removed, the projects can be grouped into three categories: those with a high removal rate (over 70%), such as *JSDoc*, *ExcelJS*, and *Axios*, totaling 32 projects; those with a low removal rate (below 45%), including *Handlebars.js* and *Prepack*, comprising 10 projects; and those with a moderate removal rate (between 45% and 70%), such as *Piskel* and *Gridsome*, accounting for 8 projects.

The survivability values reveal notable differences in how long smells persist across codebases. Projects such as *Handlebars.js* and *Prism* exhibit some of the highest mean survivability—exceeding 3,000 days and over 1,000 commits. This suggests that smells in these systems are not only tolerated, but may become deeply embedded in the codebase, possibly due to architectural inertia, limited refactoring, or a perception that they are not harmful enough to warrant prompt removal.

At the opposite end, projects like *Sweetalert2*, *Just*, and *Ungit* exhibit extremely low survivability—sometimes under 100 days or commits—indicating the need for more proactive maintenance, where issues are addressed promptly. This may reflect agile development practices or stronger emphasis on quality assurance.

Interestingly, projects with moderate removal rates exhibit mixed survivability trends. For example, *Piskel* and *Gridsome* show relatively high survivability in both time and number of commits, suggesting that although smells are eventually removed, they often persist for a considerable period.

Looking at the percentage of smells introduced at file creation (last column), we observe three distinct groups: 14 projects introduced fewer than 45% of their smells from the beginning, 17 projects introduced between 45% and 70%, and 19 projects introduced more than 70%. When cross-referencing these groups with the smell removal categories, a clear pattern emerges: projects with low removal rates tend to also have a higher proportion of smells introduced from day one (average 76%), while those with moderate or high removal rates introduce significantly fewer smells at file creation (averaging around 53–56%). This may suggest that projects with more persistent smells also suffer from initial code quality issues, which in turn may hinder long-term maintainability.

Results RQ1

The results show that code smell survivability varies significantly across projects, with some removing over 70% of smells promptly, while others allow them to persist for years. Projects like Handlebars.js exhibit high survivability, indicating limited refactoring or tolerance of smells, whereas others like Sweetalert2 address issues quickly. Additionally, projects with low smell removal rates tend to introduce more smells at file creation, suggesting a link between poor initial code quality and long-term maintainability challenges.

4.2. RQ2 – Risk Coefficients and Significance.

Considering **RQ2**, the results of the survivability analysis reveal three distinct patterns. In 18 projects, files affected by a code smell exhibit a higher probability of long-term survival compared to their non-smelly counterparts. This may indicate that smelly components are modified or refactored less frequently, potentially due to their perceived stability or neglect. In contrast, 21 projects display the opposite behavior: smelly files are more likely to fail or be removed earlier, suggesting lower survivability and possibly higher maintenance effort. In the remaining 11 projects, no substantial difference in survivability is observed between smelly and non-smelly files, indicating a similar evolutionary behavior.

These findings partially support the hypothesis that code smells can be associated with fault-proneness. However, the variation across projects highlights that the impact of code smells on survivability is not consistent and may depend on several contextual factors, such as the type of smell, system architecture, team practices, or project lifecycle.

Figure 3 illustrates representative examples of each of the three observed trends. The full set of survival plots, along with project-level classifications and detailed statistical results, is available in our publicly accessible replication packageX.

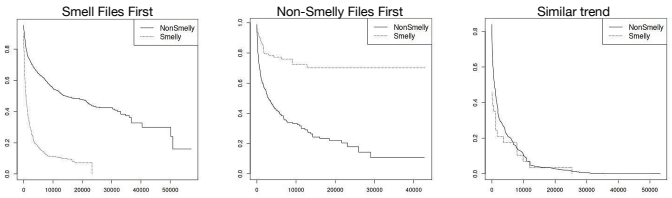

Figure 3: TBD.

Results RQ2

The survivability analysis reveals three patterns: (1) in 18 projects, smelly files survive longer, possibly due to neglect or perceived stability; (2) in 21 projects, they are more likely to be removed earlier, indicating lower survivability and higher maintenance; (3) in 11 projects, no significant difference is observed. These mixed results suggest that the impact of code smells on survivability is context-dependent and not uniform across projects.

4.3. RQ3 – On the fault-proneness in files containing smells.

Table 4 presents the results of the COX proportional hazards model applied to each code smell. As shown in the table, different smells exhibit varying degrees of impact on fault proneness. For example, "Long Methods" and "Nested Callbacks" consistently show higher risk coefficients across multiple projects, suggesting a stronger correlation with fault-prone components. In contrast, smells such as "Complex Switch" and "Assign in Condition" tend to display lower or more variable coefficients, indicating a weaker or more context-dependent relationship with software faults.

These findings align with the results reported by Johannes et al. [1] who identified "Variable Re-assign" as the most critical smell, associated with a 34.60% increase in the likelihood of fault introduction. Similarly, "Complex Code" and "Assignment in Conditional Statements" were linked to elevated fault risks of 31.40% and 30.00%, respectively, reinforcing their relevance as indicators of error-proneness and maintenance difficulties.

On the other hand, smells such as "Depth Smell" and "Complex Switch Case" were found to have lower associated risk levels, with average fault probabilities of 16.25% and 17.60%. Although these smells may still affect code readability and maintainability, their actual contribution to fault occurrence appears notably less severe compared to the more impactful smells.

Results RQ3

The results indicate that different code smells have varying levels of impact on fault-proneness. The three most impactful smells are "Variable Re-assign", "Complex Code", and "Assignment in Conditional Statements", each associated with a notably higher risk of fault introduction. These findings are consistent with prior research, reinforcing the idea that certain smells are more strongly correlated with software defects.

5. Discussion and Implications

6. Threats to Validity

In this section, we elaborate on threats to validity and the mitigation strategies we applied.

Construct Validity. The main threat is related to the relationship between theory and observation. As in the original work, we estimated the number of previous faults in each source code file by identifying those committed during fault-fixing revisions. This identification was performed through mining commit logs for specific keywords (e.g., "fix", "#", and "gh-") and referencing bug IDs. While effective, this heuristic approach is not without limitations. It may fail to detect fault-fixing commits when messages are omitted, keywords are misspelled, or bug identifiers are recorded in uncommon formats. Nonetheless, this method has been widely adopted and validated in prior software engineering studies (*e.g.,* Jaafar *et al.* [19]; Shihab *et al.* [20]), lending credibility to its use.

Similarly, the SZZ algorithm used to trace fault-inducing commits is known to have imperfections. To mitigate these, we applied the improvements recommended by Da Costa [18], including the exclusion of commits that only modify blank or comment lines, as well as those occurring too far from the issue date. In line with the original study, we limited our analysis to the master branch of each project. Although this branch often consolidates the evolution of all others, temporal mismatches in the introduction of smells or bugs across branches could still introduce minor inaccuracies. However, we believe this effect is marginal within the scope of our data.

When reconstructing smell genealogies, we set a 70% similarity threshold to determine whether two instances of the same smell type should be considered equivalent. While this threshold may occasionally conflate distinct smells or separate similar ones, we reused the same threshold and settings defined in the original work.

Internal Validity. We adopted the same metric extraction approach as the original work, relying on the AST generated by ESLint. Consequently, our findings are inherently dependent on the correctness and precision of ESLint's analysis. However, we maintain a reasonable level of confidence in its reliability. As in the original study, we employed a logarithmic link function for certain covariates in the survival analysis. While we acknowledge that other link functions might yield

Table 4: Risk Coefficient for Smells.

Project	Long Methods	Depth Smell	Complex Code	Lengthy Lines	Long Param. List	Nested Callbacks	Complex Switch	This Assign	Chained Methods	Var Re-assign	Extra Bind	Assign in Cond.
JSDoc	0.344	2.254e-06	2.364	0.885	0.885	0.829	0.771	0.631	0.584	1.31	4.538e-05	0.64
ExcelJS	3e-07	8.271e-07	2.98e-07	0.108	8.258e-07	1.083	3.485e-08	3.63e-08	0.034	0.0212	–	–
Axios	0.308	6.131e-06	0.527	0.168	0.095	0.252	0.268	0.1	0.167	0.158	–	0.198
Chrome Extensions Samples	0.569	0.83	0.57	0.232	0.711	0.616	0.601	0.59	0.535	0.431	0.536	0.563
Flux	2.986e-07	2.994e-07	2.981e-07	0.237	1.052e-07	1.047e-07	3.754e-08	1.079e-07	3.595e-08	1.192e-08	–	6.13e-06
Blockly Samples	0.417	0.188	0.243	0.227	0.148	0.183	0.198	0.253	0.188	0.212	0.143	0.199
Highlight.js	1.0	0.000001	0.759	0.688	0.688	0.726	0.799	0.784	0.711	0.812	0.602	0.727
Handlebars.js	0.345	0.419	1.270	0.614	0.614	0.702	0.776	0.682	0.654	0.632	0.544	0.875
Jasmine	0.118	0.000002	0.373	0.199	0.199	0.234	0.280	0.175	0.289	0.327	0.154	0.198
Prism	1.020	0.004	1.329	1.051	1.051	0.931	1.182	1.208	1.112	1.275	0.782	0.563
Async	0.683	0.418	0.351	0.171	0.171	0.199	0.236	0.122	0.204	0.235	0.158	0.138
Karma	0.441	0.504	0.504	0.441	0.441	0.504	0.441	0.441	0.441	0.504	0.441	0.504
Browserify	0.283	0.504	0.504	0.283	0.283	0.504	0.283	0.283	0.283	0.504	0.283	0.504
JQuery Validation	1.001	0.498	0.498	0.498	0.498	0.498	0.498	0.498	0.498	0.498	0.498	0.498
Emotion	0.181	0.427	0.427	0.181	0.181	0.427	0.181	0.181	0.181	0.427	0.181	0.181
Forever	0.055	0.055	0.055	0.055	0.055	0.055	0.055	0.055	0.055	0.055	0.055	0.055
Gulp	0.207	0.207	0.207	0.207	0.207	0.207	0.207	0.207	0.207	0.207	0.207	0.207
Validator.js	0.159	0.159	0.159	0.159	0.159	0.159	0.159	0.159	0.159	0.159	0.159	0.159
LocalForage	0.200	0.200	0.200	0.200	0.200	0.200	0.200	0.200	0.200	0.200	0.200	0.200
Markdown	0.328	0.328	0.328	0.328	0.328	0.328	0.328	0.328	0.328	0.328	0.328	0.328
Popmotion	0.170	0.170	0.170	0.170	0.170	0.170	0.170	0.170	0.170	0.170	0.170	0.170
Stf	0.891	0.891	0.891	0.891	0.891	0.891	0.891	0.891	0.891	0.891	0.891	0.891
Reactstrap	0.261	0.261	0.261	0.261	0.261	0.261	0.261	0.261	0.261	0.261	0.261	0.261
Gpu.js	0.269	0.269	0.269	0.269	0.269	0.269	0.269	0.269	0.269	0.269	0.269	0.269
Sweetalert2	0.155	0.155	0.155	0.155	0.155	0.155	0.155	0.155	0.155	0.155	0.155	0.155
Nodemailer	0.084	0.084	0.084	0.084	0.084	0.084	0.084	0.084	0.084	0.084	0.084	0.084
Fetch	0.154	0.154	0.154	0.154	0.154	0.154	0.154	0.154	0.154	0.154	0.154	0.154
Anime	0.235	0.235	0.235	0.235	0.235	0.235	0.235	0.235	0.235	0.235	0.235	0.235
Standard	0.369	0.369	0.369	0.369	0.369	0.369	0.369	0.369	0.369	0.369	0.369	0.369
Sortable	0.171	0.171	0.171	0.171	0.171	0.171	0.171	0.171	0.171	0.171	0.171	0.171
Winston	0.384	0.384	0.384	0.384	0.384	0.384	0.384	0.384	0.384	0.384	0.384	0.384
Ws	0.399	0.399	0.399	0.399	0.399	0.399	0.399	0.399	0.399	0.399	0.399	0.399
Yargs	0.289	0.289	0.289	0.289	0.289	0.289	0.289	0.289	0.289	0.289	0.289	0.289
Piskel	0.525	0.525	0.525	0.525	0.525	0.525	0.525	0.525	0.525	0.525	0.525	0.525
Katex	0.161	0.161	0.161	0.161	0.161	0.161	0.161	0.161	0.161	0.161	0.161	0.161
Prepack	0.73	0.73	0.73	0.73	0.73	0.73	0.73	0.73	0.73	0.73	0.73	0.73
Ungit	0.139	0.139	0.139	0.139	0.139	0.139	0.139	0.139	0.139	0.139	0.139	0.139
Recompose	0.214	0.214	0.214	0.214	0.214	0.214	0.214	0.214	0.214	0.214	0.214	0.214
Places	0.098	0.098	0.098	0.098	0.098	0.098	0.098	0.098	0.098	0.098	0.098	0.098
Just	0.06	–	8.271e-07	0.06	–	8.271e-07	3.808e-08	0.06	–	8.271e-07	–	–
Razzle	1.669e-05	1.669e-05	1.669e-05	0.356	–	–	1.669e-05	–	2.252e-06	0.37	–	–
Bpmn-js	0.727	–	0.686	1.506	0.799	0.613	0.575	0.657	0.634	0.995	1.669e-05	–
Artillery	2.251e-06	–	2.255e-06	0.103	2.255e-06	3.227e-07	4.538e-05	4.538e-05	0.025	0.038	–	–
Gridsome	6.135e-06	–	–	0.63	–	–	–	6.135e-06	6.135e-06	1.845	1.669e-05	–
Parsley.js	0.131	3.012e-07	2.99e-07	0.536	0.251	0.685	2.248e-06	0.146	0.38	0.581	–	0.099
NLP.js	4.538e-05	4.538e-05	4.538e-05	5.384	1.154	7.927	38.104	7.457	4.538e-05	1.154	–	–
Aura	1.266	2.435	0.813	0.842	0.577	3.876e-08	0.813	1.132	0.538	0.822	–	0.75
VisBug	1.668e-05	–	1.668e-05	6.124e-05	–	1.668e-05	1.668e-05	1.668e-05	–	1.668e-05	–	–
A Dark Room	3.799e-08	3.696e-08	3.665e-08	0.105	3.737e-08	3.834e-08	1.171e-08	–	0.049	0.089	–	–
Diagram-js	1.289	1.387	1.255	3.542	1.125	0.999	1.323	1.434	1.66	3.019	–	6.324

improved model fit for specific variables, the results of non-proportionality tests indicate that the models remain appropriate for our dataset.

External Validity. We analyzed 50 large-scale JavaScript projects, all of which are open-source. While these projects span diverse domains and vary in size, the scope remains limited. Therefore, additional validation involving a broader range of JavaScript systems and a more comprehensive set of code smell types would be beneficial.

Conclusion Validity. Threats to conclusion validity in this replication primarily concern the statistical techniques and interpretations used to support our findings. As in the original study, we relied on the Cox Proportional Hazards Model to assess the relationship between code smells and fault-proneness. While this model is well-established and suitable for survival analysis, its conclusions depend heavily on correct specification and the validity of underlying assumptions. We performed non-proportionality tests to verify model fit, and the results suggest that the assumptions were adequately met.

7. Conclusions

Acknowledgment

TDB

Declaration of Interest

The authors declare that they have no financial or personal conflicts of interest that could have influenced the research presented in this paper.

Data Availability Statement

All data supporting the findings of this study are provided as electronic supplementary material. This includes datasets, comprehensive results, scripts, and other resources needed to replicate the study. These materials are accessible via our online appendix on Figshare at: X, and on the following GitHub repository: X.

Credits

References

[1] D. Johannes, F. Khomh, G. Antoniol, A large-scale empirical study of code smells in javascript projects, Software Quality Journal 27 (2019) 1271–1314.

[2] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.

[3] G. Giordano, G. Sellitto, A. Sepe, F. Palomba, F. Ferrucci, The yin and yang of software quality: On the relationship between design patterns and code smells, in: 2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2023, pp. 227–234. doi:10.1109/SEAA60479.2023.00043.

[4] G. Giordano, A. Fasulo, G. Catolino, F. Palomba, F. Ferrucci, C. Gravino, On the evolution of inheritance and delegation mechanisms and their impact on code quality, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2022, pp. 947–958. doi:10.1109/SANER53432.2022.00113.

[5] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk, When and why your code starts to smell bad (and whether the smells go away), IEEE Transactions on Software Engineering 43 (2017) 1063–1088.

[6] R. Shatnawi, W. Li, An investigation of bad smells in object-oriented design, in: Third International Conference on Information Technology: New Generations (ITNG'06), IEEE, 2006, pp. 161–165.

[7] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, T. Dybå, Quantifying the effect of code smells on maintenance effort, IEEE Transactions on Software Engineering 39 (2012) 1144–1156.

[8] M. Abbes, F. Khomh, Y.-G. Gueheneuc, G. Antoniol, An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, in: 2011 15Th european conference on software maintenance and reengineering, IEEE, 2011, pp. 181–190.

[9] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change-and fault-proneness, Empirical Software Engineering 17 (2012) 243–275.

[10] A. M. Fard, A. Mesbah, Jsnose: Detecting javascript code smells, in: 2013 IEEE 13th international working conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2013, pp. 116–125.

[11] H. V. Nguyen, H. A. Nguyen, T. T. Nguyen, A. T. Nguyen, T. N. Nguyen, Detection of embedded code smells in dynamic web applications, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, 2012, pp. 282–285.

[12] D. Crockford, Jslint: The javascript code quality tool, URL http://www.jslint. com 95 (2011).

[13] S. Bin Uzayr, N. Cloud, T. Ambler, JavaScript Frameworks for Modern Web Development, Springer, 2019.

[14] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, et al., Experimentation in software engineering, volume 236, Springer, 2012.

[15] O. Dabic, E. Aghajani, G. Bavota, Sampling projects in github for msr studies, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 560–564.

[16] J. Fox, S. Weisberg, Cox proportional-hazards regression for survival data, An R and S-PLUS companion to applied regression 2002 (2002).

[17] M. Fischer, M. Pinzger, H. Gall, Populating a release history database from version control and bug tracking systems, in: International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings., IEEE, 2003, pp. 23–32.

[18] D. A. Da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, A. E. Hassan, A framework for evaluating the results of the szz approach for identifying bug-introducing changes, IEEE Transactions on Software Engineering 43 (2016) 641–657.

[19] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, F. Khomh, Mining the relationship between anti-patterns dependencies and fault-proneness, in: 2013 20th Working Conference on Reverse Engineering (WCRE), 2013, pp. 351–360. doi:10.1109/WCRE.2013.6671310.

[20] E. SHIHAB, Studying re-opened bugs in open source software, empirical software engineering, http://link. springer. com/article/10.1007%2Fs10664-012-9228-6 (????).