CrossMark

A large-scale empirical study of code smells in JavaScript projects

David Johannes[1] · Foutse Khomh[1] (ID) · Giuliano Antoniol[2]

Abstract

JavaScript is a powerful scripting programming language that has gained a lot of attention this past decade. Initially used exclusively for client-side web development, it has evolved to become one of the most popular programming languages, with developers now using it for both client-side and server-side application development. Similar to applications written in other programming languages, JavaScript applications contain *code smells*, which are *poor* design choices that can negatively impact the quality of an application. In this paper, we perform a large-scale study of JavaScript code smells in server-side and client-side applications, with the aim to understand how they impact the fault-proneness of applications, and how they are evolved by the developers of the applications. We detect 12 types of code smells in 1807 releases of 15 popular JavaScript applications (i.e., express, grunt, bower, less.js, request, jquery, vue, ramda, leaflet, hexo, chart, webpack, webtorrent, moment, and riot) and perform survival analysis, comparing the time until a fault occurrence, in files containing code smells and files without code smells. We also examine the introduction and removal of the code smells in the applications using survival models. All our analysis are conducted at the granularity of the line of code. Results show that (1) on average, files without code smells have hazard rates at least 33% lower than files with code smells. (2) Among the studied smells, "Variable Re-assign," "Assignment In Conditional statements," and "Complex Code" smells have the highest fault hazard rates. (3) Code smells, and particularly "Variable Re-assign," are often introduced in the application when the files containing them are created. Moreover, they tend to remain in the applications for a long period of time; "Variable Re-assign" is also the most prevalent code smell. Overall, code smells affect negatively the quality of JavaScript applications and developers should consider tracking and removing them early on before the release of applications to the public.

✉ Foutse Khomh
foutse.khomh@polymtl.ca

Extended author information available on the last page of the article.

1 Introduction

> *"Any application that can be written in JavaScript, will eventually be written in JavaScript."*
> — Jeff Atwood —

JavaScript is a highly dynamic scripting programming language that is becoming one of the most important programming languages in the world. Recent surveys by Stack Overflow (Stackoverflow 2016) show JavaScript topping the rankings of popular programming languages for four years in a row. Many developers and companies are adopting JavaScript related technologies in production and it is the language with the largest number of active repositories and pushes on Github (2016). JavaScript is dynamic, weakly typed, and has first-class functions. It is a class-free, object-oriented programming language that uses prototypal inheritance instead of classical inheritance. Objects in JavaScript inherits properties from other objects directly and all these inherited properties can be changed at run-time (Fard and Mesbah 2013). This trait can make JavaScript programs hard to maintain. Moreover, JavaScript being an interpreted language, developers are not equipped with a compiler that can help them spot erroneous and unoptimized code. As a consequence of all these characteristics, JavaScript applications often contain code smells (Fowler 1997), i.e., poor solutions to recurring design or implementation problems. However, despite the popularity of JavaScript, very few studies have investigated code smells in JavaScript applications, and to the best of our knowledge, there is no work that examines the impact of code smells on the fault-proneness of JavaScript applications. This paper aims to fill this gap in the literature. Specifically, we detect 12 types of code smells in 1,807 releases of 15 popular JavaScript applications (i.e., express, grunt, bower, less.js, request, jquery, vue, ramda, leaflet, hexo, chart, webpack, webtorrent, moment, and riot) and perform survival analysis, comparing the time until a fault occurrence, in files containing code smells and files without code smells. Similar to our previous work Saboury et al. (2017), we conduct a survival analysis of code smells in these JavaScript applications, but at the granularity of the line, i.e., considering the lines of code where the code smells and the faults appeared. We also conduct a survival analysis of code smells aiming at understanding how long they survive in applications. We answer the following three research questions:

(RQ1) Is the risk of fault higher in files with code smells in comparison with those without code smell? Previous works (Khomh et al. 2012; Jaafar et al. 2013) have found that code smells increase the risk of faults in Java classes. In this research question, we compare the time until a fault occurrence in a JavaScript file that contain code smells and JavaScript files that do not contain a code smell, computing their respective hazard rates. This research question replicates our previous work (i.e., Saboury et al. 2017) but at the granularity of the line of code. In fact, in Saboury et al. (2017), we analyzed the risk of fault by considering fault occurrences everywhere in a JavaScript file, i.e., without verifying that the fault occurred on the lines of code affected by the code smells. However, a precise matching of faulty and smelly code lines is important if we want to understand the possibility of reducing the risk of fault by refactoring code smells; which is why we ensure a matching between faulty and smelly code lines in this research question (we refer to this analysis as *line grain*). We also examined code lines (possibly contained in other JavaScript files) that need to be

modified in order to fix the fault. For example, if faulty code lines include the use or the declaration of a code element (which means an object, a function, or a variable), we add in the faulty lines the portion of code where this element is declared and used. We refer to this other analysis as *line grain analysis considering dependencies*. Results show that on average, across our 15 studied applications, JavaScript files without code smells have hazard rates 33% lower than files with code smells in our line grain analysis, and 45% lower in our line grain analysis considering dependencies. These results are consistent with the findings of Saboury et al. (2017).

(RQ2) Are JavaScript files with code smells equally fault-prone? A major concern of developers interested in improving the design of their application is the prioritization of code and design issues that should be fixed, considering their limited resources. This research question examines faults in files affected by different types of code smells (with at least one match between faulty lines and smelly lines of the corresponding type of smell), with the aim of identifying code smells that developers should refactor in priority. We conduct our analysis at the granularity of the line of code as in **RQ1**. Results show that "Variable Re-assign," "Assignment in Conditional Statements," and "Complex Code" smells are consistently associated with high hazard rates across the 15 studied systems. Developers should consider removing these code smells, in priority since they make the code more prone to faults.

(RQ3) How long do code smells survive in JavaScript projects? It is interesting to know how long the smells of a project survive, when they are introduced (at the creation of a file or during a revision), and what type smell are likely to live longer. Indeed, having a good knowledge of the lifetime of code smells in JavaScript projects can help developers identify the most dangerous smell types. Our findings suggest that JavaScript code smells are often created at the file birthdate and remain in systems for a long period of time. In fact, a considerable proportion of the studied code smells still survive today in the analyzed systems. They have a high chance to survive for a very long time after their introduction into the code base. Especially, "Variable Re-assign" which has been found to be fault-prone has one of the highest probability of surviving over time. Hence, we strongly recommend that developers remove this code smell, as soon as possible, from their projects.

This paper is an extension of our previous work Saboury et al. (2017). As for this previous work, we conduct a survival analysis of code smells in JavaScript applications to answer (RQ1) and (RQ2). Unlike this previous work, which made these survival analysis at the granularity of the file and on five large Javascript applications, we make our analysis at the granularity of the line of code. We also added ten more Javascript applications to our analysis. Our line grain analysis is used to refine the conclusions of this previous work : in fact, if a file is smelly and faulty, we intersect the faulty lines and the smelly lines of the file to see if a match occurs; if there is no match between faulty and smelly lines, the match obtained by Saboury et al. (2017) at the file level is removed from our analysis. Additionally, we examine the evolution of the studied code smells (from their introduction in the code to their last occurrence in the code) to understand how they are currently being evolved and maintained in JavaScript projects (RQ3).

The remainder of this paper is organized as follows. Section 2 describes the type of code smells we used in our study. Section 3 describes the design of our case study. Section 4

presents and discusses the results of our case study. Section 5 discusses the limitation of our study. Section 6 discusses related works on code smells and JavaScript systems, while Section 7 concludes the paper.

2 Background

To study the impact of code smells on the fault-proneness of server-side JavaScript applications, and to study the smells's survival, we first need to identify a list of JavaScript bad practices as our set of code smells. Hence, we select 12 popular code smells from different JavaScript Style Guides (Fard and Mesbah 2013; NpmJS 2016; GitHub 2016b; 2016a; jQuery 2016; Eslint 2015). These smells are popular because, according to Fard and Mesbah (2013), NpmJS (2016), GitHub (2016b), GitHub (2016a), jQuery (2016), and Eslint (2015), they widely appear in JavaScript codes, make the code unreadable, and are an obstacle to a good development of JavaScript projects. These code smells are presented in details in Saboury et al. (2017), with their frequency of occurrences in popular JavaScript projects like JQuery:

– **Lengthy Lines** appear when there are too many characters in a single line of code.
– **Chained Methods** appear when there is a "chain" of method calls (repeated chaining method). Chaining method is a common practice in object-oriented programming languages, that consists in using an object returned from one method invocation to make another method invocation.
– **Long Parameter List** happens when a function has too many parameters.
– **Nested Callbacks** are introduced in the code when multiple asynchronous tasks are invoked in sequence (i.e., the result of a previous one is needed to execute the next one) (Brodu et al. 2015; Gallaba et al. 2015).
– **Variable Re-assign** corresponds to the reuse of variables in the same scope for different purposes.
– **Assignment in Conditional Statements** occurs when the = operator is used in conditions. For example: $if (a = b)$
– **Complex Code** smell appears when a JavaScript file is characterized by high cyclomatic complexity values (i.e., high numbers of linearly independent paths through the code (McCabe 1976)).
– **Extra Bind** occurs most of time when we let ".bind(ctx)" on a function after removing a this variable from the body of the inner function, which is an unnecessary overhead.
– **This Assign** happens specifically when a this variable is stored in another variable to access to the parent scope's context.
– **Long Methods** is a well-known code smell (Marinescu and Lanza 2006; Fard and Mesbah 2013; Fontana et al. 2012) which consists in writing a method with too many statements.
– **Complex Switch Case** happens when there are too many switch statements.
– **Depth** smell occurs when the number of nested blocks of code (or the level of indentation) is too high.

Table 1 indicates the number of smells, by type, that we found in each studied JavaScript project (including all their commits). We can notice that all the studied JavaScript projects, even those proposing a Style Guide to avoid code smells (like JQuery), contain a large

Table 1 Smells count of the studied systems

Module	Long Methods	Depth	Complex Code	Lengthy Lines	Long Parameter List	Nested Callbacks	Complex Switch Case	Chained Methods	Variable Re-assign	Extra Bind	Assignment in Conditional Statements	This Assign	Total
Express	3313	104	5507	51802	2737	25834	949	28287	73116	0	1159	3034	195842
Request	4682	0	8006	69496	374	1146	625	17595	119215	0	0	7214	228353
Less.js	17086	1799	46940	361171	11082	868	4819	55608	482752	0	22655	4075	1008855
Bower.io	1919	0	3017	101901	356	25687	8	143999	72712	123	0	2145	351867
Grunt	1393	28	1901	54269	433	1039	10	17931	29734	3	113	22	106876
Jquery	39002	3389	45664	1197334	12397	14036	897	409512	518637	3	811	3913	2245595
Vue.js	25009	1509	64842	230084	21458	159	259	7338	462302	0	1201	9780	823941
Ramda	1754	9	3855	427837	5237	3013	974	10700	158314	0	5	526	612224
Leaflet	7382	0	32934	263223	7648	10249	2800	138734	357790	0	733	244	821737
Hexo.io	2023	45	4626	34250	1433	12263	580	71734	57196	0	243	1100	185493
Chart.js	14776	22	33706	457036	7643	4720	2795	357635	409079	0	2005	13961	1303378
Webpack	4337	176	7354	122926	2869	9806	2249	66051	77397	69	28	608	293870
Webtorrent.io	18956	1059	48636	42989	11913	3437	2449	418806	653364	0	631	44645	1246885
Moment	23927	107	47334	1177712	10530	4743	6263	551804	453806	0	422	1064	2277712
Riot	13412	795	34536	209389	11209	2732	4301	28250	357150	91	7028	5652	674545

number of code smells. Plus, each type of code smell is highly diffused in at least one studied JavaScript project.

3 Study design

The *goal* of our study is to investigate the relation between the occurrence of code smells in JavaScript files and the fault-proneness of the part of the JavaScript files containing the smelly code lines. We also aim to understand the survival of code smells during the evolution of the projects. The *quality focus* is the source code fault-proneness, which, if high, can have a negative impact on the cost of maintenance and evolution of the system. The *perspective* is that of researchers, interested in the relation between code smells and the quality of JavaScript systems. The results of this study are also of interest for developers performing maintenance and evolution activities on JavaScript systems since they need to take into account and forecast their effort, and to testers, who need to know which files should be tested in priority. Finally, the results of this study can be of interest to managers and quality assurance teams, who could use code smell detection techniques to assess the fault-proneness of in-house or to-be-acquired systems, to better quantify the cost-of-ownership of these systems. The *context* of this study consists of 12 types of code smells identified in 15 JavaScript systems. In the following, we introduce our research questions, describe the studied systems, and present our data extraction approach. Furthermore, we describe our model construction and model analysis approaches.

(RQ1) Is the risk of fault higher in files with code smells in comparison with those without code smell? Prior works show that code smells increase the fault-proneness of Java classes (Khomh et al. 2012; Jaafar et al. 2013). Since JavaScript code smells are different from the code smells investigated in these previous studies on Java systems, we are interested in examining the impact that JavaScript code smells can have on the fault-proneness of JavaScript applications. In our previous work (Saboury et al. 2017), we showed that JavaScript files with code smells are more likely to be fault-proneness than those without code smells. In this research question, we will refute or confirm this conclusion by replicating the analysis at the granularity of code lines. An analysis at the granularity of code lines is important to verify if the faults observed in files containing code smells occur on the portion of code affected by the code smell(s). Indeed, in our previous work, when a file is marked as smelly and faulty, there is no guarantee that the fault occurred on the part of the file containing the code smell. It is possible that the faulty code lines are different from the smelly code lines. Our current analysis at the granularity of the line do not consider this to be a match and hence refines the link between the smelliness and the fault-proneness of JavaScript files. By performing a line grain analysis, we expect the observed impact of code smells on fault-proneness to be more accurate.

(RQ2) Are JavaScript files with code smells equally fault-prone? During maintenance and quality assurance activities, developers are interested in identifying parts of the code that should be tested and–or refactored in priority. Hence, we are interested in identifying code smells that have the most negative impact on JavaScript systems, i.e., making JavaScript applications more prone to faults.

(RQ3) How long do code smells survive in JavaScript projects? We are interested here in knowing the genealogy of code smells in JavaScript projects. Behind this, we aim to

understand when smells are introduced (i.e., at the birthday of files, or during changes), how long they survive, and when they are generally removed (i.e., at the deletion of files, or through refactorings).

3.1 Studied Systems

In order to address our research questions, we perform a case study with the following 15 open-source JavaScript projects. Table 2 summarizes the characteristics of our subject systems.

Express[1] is a minimalist web framework for Nodejs. It is one of the most popular libraries in NPM (Mardan 2014) and it is used in production by IBM, Uber and many other companies.[2] Its GitHub repository has over 5500 commits and more than 220 contributors. It has been forked 7100 times and starred more than 41,500 times. Express is also one of the most dependent upon libraries on NPM with over 8800 dependents. There are more than 2800 closed GitHub issues on their repository.

Bower.io[3] is a package manager for client-side libraries. It is a command line tool which was originally released as part of Twitter's open-source effort[4] in 2012 (Bower 2016). Its GitHub repository has more than 2700 commits from more than 210 contributors. Bower has been starred over 15,200 times on GitHub and has over 1,600 closed issues.

LessJs[5] is a CSS[6] pre-processor. It extends CSS and adds dynamic functionalities to it. There are more than 2800 commits by over 210 contributors on its GitHub repository. LessJs's repository has more than 2,300 closed issues and it is starred more than 15,800 times and forked over 3,400 times.

Request[7] is a fully featured library to make HTTP calls. More than 8300 other libraries are direct dependents of Request. Over 2100 commits by more than 270 contributors have been made into its GitHub repository and 16,000+ users starred it. There are more than 1200 closed issues on its GitHub repository.

Grunt[8] is one of the most popular JavaScript task runners. More than 1600 other libraries on NPM are direct dependents of Grunt. Grunt is being used by many companies such as Adobe, Mozilla, Walmart, and Microsoft (Grunt 2016). The GitHub repository of Grunt is starred by more than 11,900 users. More than 60 contributors made over 1400 commits into this project. They also managed to have more than 1000 closed issues on their GitHub repository. We selected these projects because they are among the most popular NPM libraries, in terms of the number of installs. They have a large size and possess a Github repository with issue tracker and wiki. They are also widely used in production.

---

[1] https://github.com/expressjs/express

[2] https://expressjs.com/en/resources/companies-using-express.html

[3] https://github.com/bower/bower

[4] https://engineering.twitter.com/opensource

[5] https://github.com/less/less.js

[6] Cascading Style Sheet

[7] https://github.com/request/request

[8] https://github.com/gruntjs/grunt

Table 2 Descriptive statistics of the studied systems

Module	Domain	# Commits	# Contributors	# GitHub stars	# Releases	# Closed issues	# Forks	Project start date
Express	Web framework	5500+	220	41500+	277	2800+	7100+	Jun 26, 2009
Request	HTTP client utility	2200+	285	21300+	144	1700+	2500+	Jan 23, 2011
Less.js	CSS pre-processor	2800+	217	15800+	74	2300+	3400+	Feb 20, 2010
Bower.io	Package manager	2700+	210	15200+	105	1600+	1900+	Sep 7, 2012
Grunt	Task Runner	1400+	67	11900+	13	1000+	1500+	Sep 21, 2011
Jquery	JavaScript library	6300+	273	50500+	148	1700+	17000+	Apr 3, 2009
Vue.js	JavaScript framework	2700+	240	122300+	233	7100+	17000+	Jul 29, 2013
Ramda	JavaScript library	2700+	212	14800+	49	1100+	900+	Jun 21, 2013
Leaflet	JavaScript library	6600+	590	23500+	41	3500+	3900+	Sep 22, 2010
Hexo.io	Blog framework	2600+	125	24500+	130	2700+	3300+	Sep 23, 2012
Chart.js	JavaScript charting	2600+	287	40800+	41	3900+	9200+	Mar 17, 2013
Webpack	JavaScript bundler	7400+	500	45800+	331	5500+	5800+	Mar 10, 2012
Webtorrent.io	Streaming torrent client	2300+	116	18200+	277	1000+	1700+	Oct 15, 2013
Moment	JavaScript date manager	3600+	480	39500+	74	2900+	5900+	Mar 1, 2011
Riot	Component-based UI library	3200+	169	13300+	122	1800+	1000+	Sep 27, 2013

JQuery[9] is a famous JavaScript library, created to make easier the writing of client-side scripts in the HTML of web pages. It makes also easier the way to write Ajax (asynchronous JavaScript and XML) code. More than 6300 commits have been made into its Github repository by over 270 contributors, and 50,500+ users starred it. Plus, it is forked more than 17,000 times, and there are more than 1700 closed issues. JQuery is likely one of the most popular and biggest JavaScript project.

VueJs[10] is a performant and progressive JavaScript framework for building user interfaces. It has the big advantage (in comparison with other JavaScript frameworks) to be incrementally adoptable. Over 240 contributors made over 2700 commits into its Github repository, and they closed more than 7100 issues. It is forked more than 17,000 times and starred more than 122,300 times, which makes it so popular.

Ramda[11] is a functional library, which makes easier the creation of functional pipelines and functions (as sequences for example), and does not mutate user data. It is starred more than 14,800 times, and 212 contributors made over 2700 commits into its Github repository.

Leaflet[12] is used for mobile-friendly interactive maps, and is designed in order to be simple, efficient, easily extended (with plugins), easy to use, and usable across desktop and mobile platforms. Its Github repository is starred by more than 23,500 users and forked by over 3900 users. More than 590 people contribute to over 6600 commits, and managed to have more than 3500 closed issues on their Github repository.

Hexo.io[13] is a very fast, powerful, and simple framework designed for blog's creation. It has 125 contributors, who made more than 2600 commits, and closed over 2700 issues. Its Github repository is forked over 3300 times and starred over 24,500 times.

ChartJs[14] is a flexible and very simple HTML5 charting that offers to designers and developers the chance to see their data in 8 different ways, possibly scalable, customizable and animated. Its Github repository joins over 280 contributors, who closed more than 3900 issues in over 2600 commits. Plus, more than 9200 users forked it and over 40,800 users starred it.

Webpack[15] is a module blunder designed for modern applications. It allows the browser to load only a few number of bundles as small as possible. Those bundles correspond to the packaged modules that the application needs. Webpack is easy to configure and to take in hand. Its Github repository has over 7400 commits and more than 500 contributors, who closed more than 5500 issues. It has been forked 5800 times and starred more than 45,800 times.

---

[9]https://github.com/jquery/jquery

[10]https://github.com/vuejs/vue

[11]https://github.com/ramda/ramda

[12]https://github.com/Leaflet/Leaflet

[13]https://github.com/hexojs/hexo

[14]https://github.com/chartjs/Chart.js

[15]https://github.com/webpack/webpack

Webtorrent.io[16] is a streaming torrent client especially designed for the desktop and the web browser. Almost 120 contributors made over 2300 commits and helped to solve and close more than 1000 issues on its Github repository. It is starred more than 18,200 times.

Moment[17] allows users to do whatever they want with dates and times in JavaScript (which means manipulate, parse, validate, display, etc.) in a very easy way. Its Github repository has more than 3600 commits, over 480 contributors, and more than 2900 closed issues. It is forked more than 5900 times and starred more than 39,500 times.

Riot[18] is a simple, minimalistic, and elegant component-based UI library that offers to users the necessary building blocks for modern client-side applications, some custom tags, and an elegant syntax and API. Almost 170 people contributed to its Github repository, and made more than 3200 commits, and closed more than 1800 issues. It is starred more than 13,300 times.

The criteria of selection, for the studied systems, are based on projects' popularity and activity. Indeed, in order to make our study relevant, we need popular projects to analyze, which is indicated by the Github stars of the projects. In our case, all the studied systems have more than 5000 Github stars, which make them widely used and diffused in JavaScript community. Also, as we want our analysis and conclusions pertinent, we need active systems, that evolve a lot and all the time. This is mainly shown first by the number of commits recorded by each project, which exceed 1000 for all the studied systems ; and second by their actual activity, because commits are recently pushed on each proposed system.

3.2 Data extraction

To answer our research questions, we need to mine the repositories of our 15 selected systems. Given a system, for each commit we extract information about the *smelliness* of each file, identifying if the file contains code smells and the location of the smelly lines. In addition, we need to know for each commit, if the commit introduces a bug, fixes a bug, or just modifies the file in a way that a code smell is removed or added. Plus, fault-proneness lines of files related to bugs need to be extracted in order to correlate them with the extracted smelly lines of the file, if they exist. Figure 1 provides an overview of our approach to answer RQ1 and RQ2, and Fig. 2 presents an overview of the approach followed to answer RQ3. We describe each step in our data extraction approaches below. We have implemented all the steps of our approaches into a framework available on Github.[19]

Snapshot generation Since all the 15 studied systems are hosted on Github, at the first step, the framework performs a `git clone` to get a copy of a system's repository locally. It then generates the list of all the commits and uses it to create snapshots of the system that would be used to perform analysis at commits level.

Identification of fault-inducing changes Our studied systems use Github as their issue tracker and we use Github APIs to get the list of all the resolved issues on the systems. We

---

[16]https://github.com/webtorrent/webtorrent

[17]https://github.com/moment/moment

[18]https://github.com/riot/riot

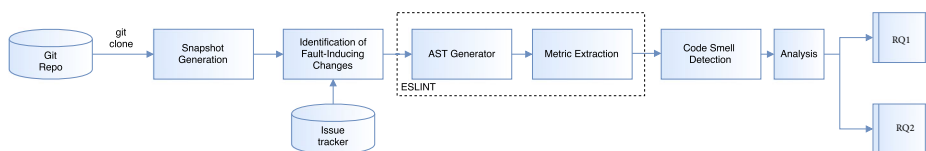[19]https://github.com/DavidJohannesWall/smells_project

Fig. 1 Overview of our approach to answer RQ1 and RQ2

leverage the SZZ algorithm (Śliwerski et al. 2005) to detect changes that introduced faults. We first identify fault-fixing commits using the heuristic proposed by Fischer et al. (2003), which consists in using regular expressions to detect bug IDs from the studied commit messages. The regular expressions are used first to detect the information that refers to a number associated to a fixed bug ID (preceded generally by a "#" or a "gh-"), and then to identify common words related to an issue fixing (e.g., the word "fix"). To assess the effectiveness of this heuristic, we randomly selected 100 identified fault-fixing commits from the JQuery project and examined them manually. All the 100 commits were indeed fault-fixing commits. This result is not impressive since the best way for developers to refer to an issue, when they fix it in a commit they push on GitHub, is by using "#" or "gh-," followed by the bug ID. In that way, especially for active projects like the studied ones, it is easy to retrieve the fault-fixing commits. However, it is possible to miss some other fault-fixing changes, either because the developers did not refer to the associated issue, or because they did it in a non common way (for example by referring to an issue number without preceding it by "#" or "gh-"), or because the corresponding bug has no number ID (for these ones, this is generally because the bug is not relevant). For the JQuery project, some references to issue numbers are different (preceded by "/issues/"), but they represent less than 5% of the referred issues, and often refer to already closed issues because they just document the developer's comment. Next, we extract the modified files of each fault-fixing commit through the following Git command:

```
git log [commit-id] -n 1 --name-status
```

We only take modified JavaScript files into account. Given each file F in a commit C, we extract C's parent commit C'. Parent commit corresponds to the previous commit in the master branch of the project. We in fact only consider the master branch of the project during our analyses. Then, we use Git's `diff` command to extract F's deleted lines. We apply Git's `blame` command to identify commits that introduced these deleted lines, noted as the "candidate faulty changes." We eliminate the commits that only changed blank and comment lines. Then, we filter the commits that were submitted after their corresponding bugs' creation date. In order to avoid false positives when identifying the fault-inducing changes, we followed the framework of Da Costa et al. (2017) by: (1) removing bugs in bugs list of a fault-introducing commit, when these bugs are too far away from the first bug appearance (in fact, if a commit is responsible of introducing many bugs, it is unlikely that this commit introduces other bugs a long time after the first bug appearance); (2) removing commits potentially responsible of a specific bug, when they are too far away from this bug appearance (indeed, a bug is rarely identified several years after it was really introduced). We
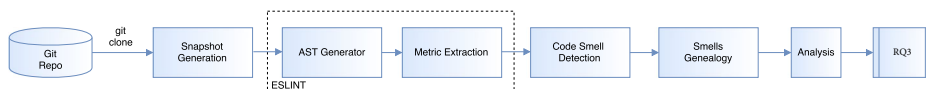

Fig. 2 Overview of our approach to answer RQ3

discard these false positives by using the metric of Da Costa, the MAD (*Median Absolute Deviation*), and by removing them when they are above the upper MAD of the SZZ generated data. Considering the file F in a fault-fixing commit and its commit that introduced faults, we use again Git's `diff` command to extract F's changes between both commits, in order to retrieve the "candidate fault lines" of the file F (useful for our line grain analysis). For the next step, we use UglifyJS[20] to get an F's Abstract Syntax Tree (AST) that gives the dependencies of all F's variables, objects, and functions (which means their declaration and use lines). We then match F's dependencies with the "candidate fault lines" to extend them: given an F's element (variable, object, or function), if one of its declaration or use lines is found into the "candidate fault lines," then we add this declaration and use lines to the "candidate fault lines." We finally obtain the "extended candidate fault lines" of the file F (useful for our line grain analysis that include dependencies). The line grain analysis including dependencies consist in extended the faulty identified lines, in order to consider all the impacts of the faulty lines inside a file. For example, if a bug appears in a function of a JavaScript file, and this function is used many times in other places of this file, we report not only the faulty lines but also the lines where the faulty lines are implied in the file. The motivation behind this is when lines code are associated to a particular bug (because these lines have been modified in order to correct the bug), there are in realty affecting other parts of the code file, and these other parts need to be reported because they are intimately linked to the faulty lines, hence to the bug. In that way, if fault-proneness lines include the use or the declaration of a code element (which means an object, a function, or a variable), we add in the faulty lines the portion of code where this element is declared and used. This allows us to take into account parts of the code that are indirectly impacted by a fault occurrence, and hence potentially by a code smell occurrence.

AST generation and metric extraction To automatically detect code smells in the source code, we first extract the Abstract Syntax Tree from the code. AST are being used to parse a source code and generate a tree structure that can be traversed and analyzed programmatically. ASTs are widely used by researchers to analyze the structure of the source code (Neamtiu et al. 2005; Baxter et al. 1998; Pfenning and Elliott 1988). We used ESLint[21] which is a popular and open-source lint utility for JavaScript as the core of our framework. Linting tools are widely used in programming to flag the potential non-portable parts of the code by statically analyzing them. ESLint is being used in production in many companies like Facebook, Paypal, and Airbnb. ESLint uses espree[22] internally to parse JavaScript source codes and extracts Abstract Source Trees based on the specs.[23] ESLint itself provides an extensible environment for developers to develop their own plugins to extract custom information from the source code. We modified ESLint built-in plugins to traverse the source tree generated by ESLint to extract and store the information related to our set of code smells described in Section 2 (i.e., their weight and their lines location). Table 3 summarizes all the metrics our framework reports for each type of code smell. For each file of each commit of the studied systems, ESLint is used to parse the file and report information about the smells appearing in the file. Basically, when a file is parsed by ESLint, nodes are created (e.g., a node is a line of the parsed file when ESLint try to identify lengthy line

---

[20]https://github.com/mishoo/UglifyJS

[21]http://eslint.org/

[22]https://github.com/eslint/espree

[23]https://github.com/estree/estree

Table 3 Metrics computed for each type of code smell

Smell type	Type	Metric
Lengthy lines	Number	The number of characters per line considering the exceptions described in Section 2.
Chained methods	Number	The number chained methods in each chaining pattern.
Long parameter list	Number	The number of parameters of each function in source code.
Nested callbacks	Number	The number of nested functions present in the implementation of each function.
Variable re-assign	Boolean	The uniqueness of variables in same scope.
Assignment in conditional statements	Boolean	The presence of assignment operator in conditional statements.
Complex code	Number	The cylcomatic complexity value of each function defined in the source code.
Extra bind	Boolean	Whether a function is explicitly bound to a context while not using the context.
This assign	Boolean	Whether `this` is assigned to another variable in a function.
Long methods	Number	The number of statements in each function.
Complex switch case	Number	The number of case statements in each switch-case block in the source code.
Depth	Number	The maximum number of nested blocks in each function.

smells), and if a condition is respected by a node (e.g., for lengthy line smells, the number of characters of a given line exceeds a metric number, like 65), the node is reported as smelly and information about this smell are reported (e.g., the weight, meaning how much the smell exceeds the metric number, and the line or lines range of the smell in the file obtained thanks to the corresponding node).

Smells genealogy Thanks to our previous extraction methods, we easily get, for each JavaScript file of a project, the history of the commits that modified those files. Given the history H of a JavaScript file F, we identify and track F's smells through each commit of H. Given two consecutive commits $C1$ and $C2$ of H: if one smell appears in $C2$ (and not in $C1$), we consider it as a new smell and keep its date of creation ($C2$'s date); if one smell disappears in $C2$ (and was present in $C1$), we consider that it was killed, and keep its date of destruction ($C2$'s date). If a smell was never killed (i.e., is present in the last commit of H), we consider its presence until the last project's commit. To measure the degree of similarity between two smells, they first need to be from the same smell type, and then we use SequenceMatcher[24] from difflib (a Python library) that gives us a number between 0 and 1 as the degree of similarity (1: both smells are the same; 0: they are totally different). The similarity between two code smells is based on their text, thanks to this SequenceMatcher, which relies on the Ratcliff and Obershelp's algorithm, published in 1980, named "gestalt pattern matching." The main idea of the algorithm is to find the longest contiguous matching subsequence between two compared sequences. We consider two smells as the same if they are from the same smell type (among the 12 studied code smells), and if their similarity degree is greater than 0.7. If one smell of $C1$ gets a similarity degree greater than 0.7 with two smells of $C2$, we match it with the one with the highest similarity value. We

---

[24] https://docs.python.org/2/library/difflib.html

repeated our survival analysis of smells with different thresholds of similarity degree (0.8 and 0.9), but we observed no significant difference with the use of the 0.7 threshold. Also, we did the same with lower thresholds, and observe a deterioration of the results, because two smells were naively considered as the same while they were really different. In that way, if a smell is corrected in a commit, while another smell of the same type is introduced, they are more likely to be considered as the same and the lifetime of this type smell will abnormally increase. Therefore, we decided to only report our results for the 0.7 threshold.

Code smell detection Among the 12 metric values reported by our framework, 4 are boolean. The boolean metrics concern *This Assign*, *Extra Bind*, *Assignment in Conditional Statements*, and *Variable Re-assign* smells. The 8 remaining metrics are integers. To identify code smells using the metric values provided by our framework, we follow the same approach as previous works (Saboury et al. 2017; Marinescu 2004; Mazinanian and Tsantalis 2016), defining threshold values above which files should be considered as having the code smell. We define the thresholds relative to the systems using boxplot analysis, by being sure that all the chosen thresholds correspond to a smell population above the third quartile (75%) in the boxplots. Figures 3 show some relevant boxplots we found (meaning some with several values significantly taken by the smells), with the associated chosen threshold value in the title. All the boxplots appear in the GitHub repository of the project. We chose to define threshold values relative to the projects because design rules and programming styles can vary from one project to another, and hence it is important to compare the
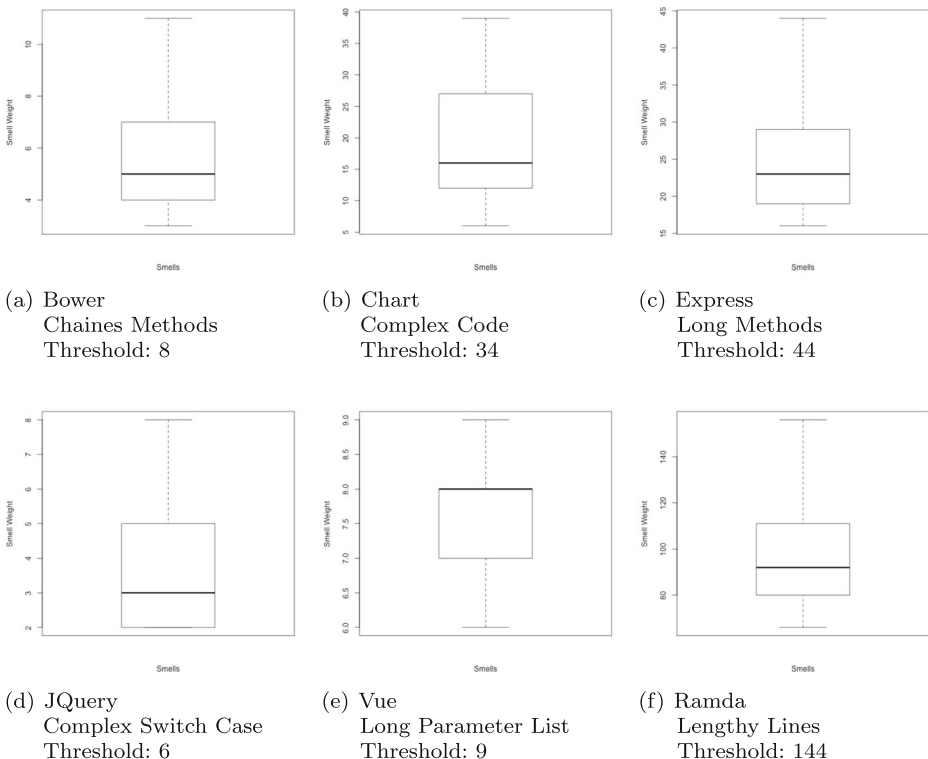

(a) Bower
    Chaines Methods
    Threshold: 8

(b) Chart
    Complex Code
    Threshold: 34

(c) Express
    Long Methods
    Threshold: 44

(d) JQuery
    Complex Switch Case
    Threshold: 6

(e) Vue
    Long Parameter List
    Threshold: 9

(f) Ramda
    Lengthy Lines
    Threshold: 144

Fig. 3 Boxplots of some smells

characteristics of files in the context of the project. For each system, we obtain the threshold values as follows. We examined the distribution of the metrics and observed a big gap around the first 70% of the data and the top 10%. Hence, we decided to consider files with metric values in the top 10% as containing the code smell. For files that contain multiple functions, we aggregated the metric values reported for each functions using the maximum to obtain a single value characterizing the file. For our line grain analysis, since some smells (like long methods smells) appear on several consecutive lines, we record the lines range for these specific type smells, which is provided by the framework ESLint. For the other smells (like re-assign smells), we only report the line location). We manually examined a sample of JQuery project files marked as "smelly" by our algorithm to verify that they are indeed smelly. Among almost 9000 smelly files reported by our algorithm, we randomly selected a sample of 100 files (to ensure a confidence interval of at least 90%). We manually inspected the files and found that they all contain at least one smell with a weight above the chosen corresponding threshold.

3.3 Analysis

In this section, we detail the data analysis methods for each research question. Since Survival models are the cornerstone of our analysis, we first provide background information about them.

Survival analysis is used to model the time until the occurrence of a well-defined event (Fox and Weisberg 2010). One of the most popular models for survival analysis is the Cox Proportional Hazards (Cox) model. A Cox hazard model is able to model the instantaneous hazard of the occurrence of an event as a function of a number of independent variables (Koru et al. 2008; Singer and Willett 2003). Particularly, Cox models aim to model how long subjects under observation can *survive* before the occurrence of an event of interest (e.g., a fault occurrence or a smell occurrence in our case) (Singer and Willett 2003; Selim et al. 2010).

Survival models were first introduced in demography and actuarial sciences (Westergaard 1932). Recently, researchers have started applying them to problems in the domain of Software Engineering. For example, Selim et al. (2010) used the Cox model to investigate characteristics of cloned code that are related to the occurrence of faults. Koru et al. (2007) also used Cox models to analyze faults in software systems. In Cox models, the hazard of a fault occurrence at a time t is modeled by the following function:

$$\lambda_i(t) = \lambda_0(t) * e^{\beta * F_i(t)} \tag{1}$$

If we take log from both sides, we obtain:

$$log(\lambda_i(t)) = log(\lambda_0(t)) + \beta_1 * f_{i1}(t) + ... + \beta_n * f_{in}(t) \tag{2}$$

where

- $F_i(t)$ is the time-dependent covariates of observation i at the time t.
- β is the coefficient of covariates in the function $F_i(t)$.
- λ_0 is the baseline hazard.
- n is the number of covariates.

When all the covariates have no effect on the hazard, the baseline hazard can be considered as the hazard of occurrence of the event (e.g., a fault or a smell). The baseline hazard

would be omitted when formulating the relative hazard between two files (in our case) at a specific time, as shown in the following (3).

$$\lambda_i(t)/\lambda_j(t) = e^{\beta*(f_i(t)-f_j(t))} \tag{3}$$

The proportional hazard model assumes that changing each covariate has the effect of multiplying the hazard rate by a constant.

Link function As (2) shows, the log of the hazard is a linear function of the log of the base-line hazard and all the other covariates. In order to build a Cox proportional model, a linear relationship should be available between the log hazard and the covariates (Therneau and Grambsch 2000). Link functions are used to transform the covariates to a new scale if such relationship does not exist. Determining an appropriate link function for covariates is necessary because it allows changes in the original value of a covariate to influence the log hazard equally. This allows the proportionality assumption to be valid and applicable (Therneau and Grambsch 2000).

Stratification In addition to applying a link function, a stratification is sometimes necessary to preserve the proportionality in Cox hazard models (Koru et al. 2008). For example, if there is a covariate that needs to be controlled because it is of no interest or secondary, stratification can be used to split the data set so that the influence of more important covariates can be monitored better (Koru et al. 2008).

Model validation Since Cox proportional hazard models assume that all covariates are consistent over time and the effect of a covariate does not fluctuate with time, hence, to validate our model, we apply a non-proportionality test to ensure that the assumption is satisfied (Selim et al. 2010; Therneau and Grambsch 2000).

In this paper, we perform each of our analysis on each file of each commit of the studied projects, and refine the correlations between faults and smells experienced by a file with our line grain analysis. Then, for each file, we use Cox proportional hazard models to calculate the risk of a fault or a smell occurrence over time, considering a number of independent covariates. We chose the Cox proportional hazard model for the following reasons:

(1) In general, not all files in a commit experience a fault. Cox hazard models allow files to remain in the model for the entire observation period, even if they don't experience the event (i.e., fault occurrence). (2) In Cox hazard models, subjects can be grouped according to a covariate (e.g., smelly or non-smelly). (3) The characteristics of the subjects might change during the observation period (e.g., size of code), and (4) Cox hazard models are adapted for events that are recurrent (Therneau and Grambsch 2000), which is important because software modules evolve over time and a file can have multiple faults during its life cycle.

(RQ1) Is the risk of fault higher in files with code smells in comparison with those without code smell?

To assess the impact of code smells on the fault-proneness of JavaScript files, we perform survival analysis, comparing the time until a fault occurrence, in files containing code smells with match between faulty and smelly lines, and files without code smells. For each file and for each revision r (i.e., corresponding to a commit), we also compute the following metrics:

- **Time:** the number of hours between the previous revision of the file and the revision r. We set the time of the first revision to zero.
- **Smelly:** this is our covariate of interest. It takes the value 1 if the revision r of the file contains a code smell and 0 if it does not contain any of the 12 studied code smells.
- **Event:** For the line grain analysis (respectively the line grain analysis including dependencies), this metric takes the value 1 if the revision r is a fault-fixing change and if there is at least one match between the faulty lines (respectively the extended faulty lines described in Section 3.2) and the code smell lines, and 0 otherwise. Indeed, if there is no matching, we consider that the fault-fixing change doesn't relate to code smells. We use the improved SZZ algorithm to insure that the file contained a code smell when the fault was introduced.

Using the smelly metric, we divide our dataset in two groups: one group containing files with code smells (i.e., smelly = 1) and another group containing files without any of the 12 studied code smells (i.e., smelly = 0). For each group we create an individual Cox hazard model. In each group, the covariate of interest (i.e., smelly) is a constant function (with value either 1 or 0), hence, there is no need for a link function to establish a linear relationship between this covariate and our event of interest, i.e., the occurrence of a fault. We use the *survfit* and *coxph* functions from R (Therneau 2000) to analyze our Cox hazard models.

In addition to building Cox hazard models, we test the following null hypothesis: H_0^1: *There is no difference between the probability of a fault occurrence in a file containing code smells and a file without code smells*. We use the *log-rank* test (which compares the survival distributions of two samples), to accept or refute this null hypothesis.

(RQ2) Are JavaScript files with code smells equally fault-prone?

Similar to **RQ1**, for each file and for each revision r (i.e., corresponding to a commit), we compute the **Time** and **Event** metrics defined in **RQ1**. For each type of code smell i we define the metric **Smelly**$_i$**:** which takes the value 1 if the revision r of the file contains the code smell i and 0 if it does not contain any of the 12 studied code smells. Also, similarly to **RQ1**, for the line grain analysis (respectively the line grain analysis including dependencies), we define the metric **Event**$_i$**:** which takes the value 1 if the revision r is a fault-fixing change and if the code smell i is in the intersection between the faulty lines (respectively the extended faulty lines described in Section 3.2) and the smelly lines, and 0 otherwise. When computing the **Event** and the **Event**$_i$ metrics, we used the improved SZZ algorithm to ensure that the file contained the code smell i when the fault was introduced. Because size, code churn, and the number of past occurrence of faults are known to be related to fault-proneness, we add the following metrics to our models, to control for the effect of these covariates : (i) LOC: the number of lines of code in the file at revision r; (ii) Code Churn: the sum of added, removed and modified lines in the file prior to revision r; (iii) No. of Previous-Bugs: the number of fault-fixing changes experienced by the file prior to revision r.

We perform a stratification considering the covariates mentioned above, in order to monitor their effect on our event of interest, i.e., a fault occurrence. Next, we create a Cox hazard model for each of our 15 studied systems. In order to build an appropriate link function for the new covariates considered in this research question (i.e., LOC, Code churn, and No. of Previous-Bugs), we follow the same methodology as Koru et al. (2008) and Selim

Table 4 Fault hazard ratios for each project with the line grain approach

Module	$exp(coef)$	p value (Cox hazard model)	p value (proportional hazards assumption)
express	1.603	4.415e−08	0.046
request	2.328	3.737e−4	0.977
less	1.801	7.896e−7	0.085
bower	1.512	6.833e−5	0.826
grunt	0.994	0.961	0.068
jquery	3.091	0	1.294e−11
vue	0.110	0	0.001
ramda	0.780	0.020	0.390
leaflet	0.707	6.076e−9	0.752
hexo	1.439	2.891e−5	0.700
chart	1.043	0.771	0
webpack	1.134	0.006	0
webtorrent	1.444	0.180	0.814
moment	0.996	0.954	1.878e−7
riot	0.973	0.717	0.167

$exp(coef)$ values means higher hazard rates

et al. (2010) and plot the log relative risk vs. each type of code smell, the No. of Previous-Bugs, LOC and Code Churn in each of our 15 datasets (corresponding to the 15 subject systems). We generated summaries of all our Cox models and removed insignificant covariates, i.e. those with p values greater than 0.05. Finally, for each system, we performed a non-proportional test to verify if the proportional hazards assumption holds.

(RQ3) How long do code smells survive in JavaScript projects?

To assess the impact of smells survival over project lifetime, we perform survival analysis, comparing the time between the creation of a type of code smell and either their removal, or the last revision of the studied projects. For each studied system and for each smell type, we compute the following metrics:

- The number of created smells.
- The number of killed smells (over the system lifetime).
- The number of survived smells, which means the number of smell that are still in the system.
- The number of smells created at the file birthdate.
- The median days of survival of the smells.
- The average days of survival of the smells.

For each smell created (which means that it was never encountered before), we also compute the **Time** and **Event** metrics defined below:

- **Time:** the time in days since the smell creation.
- **Event:** this metric takes the value 1 if the studied smell is present at this time (which means that it has not been removed), and 0 otherwise.
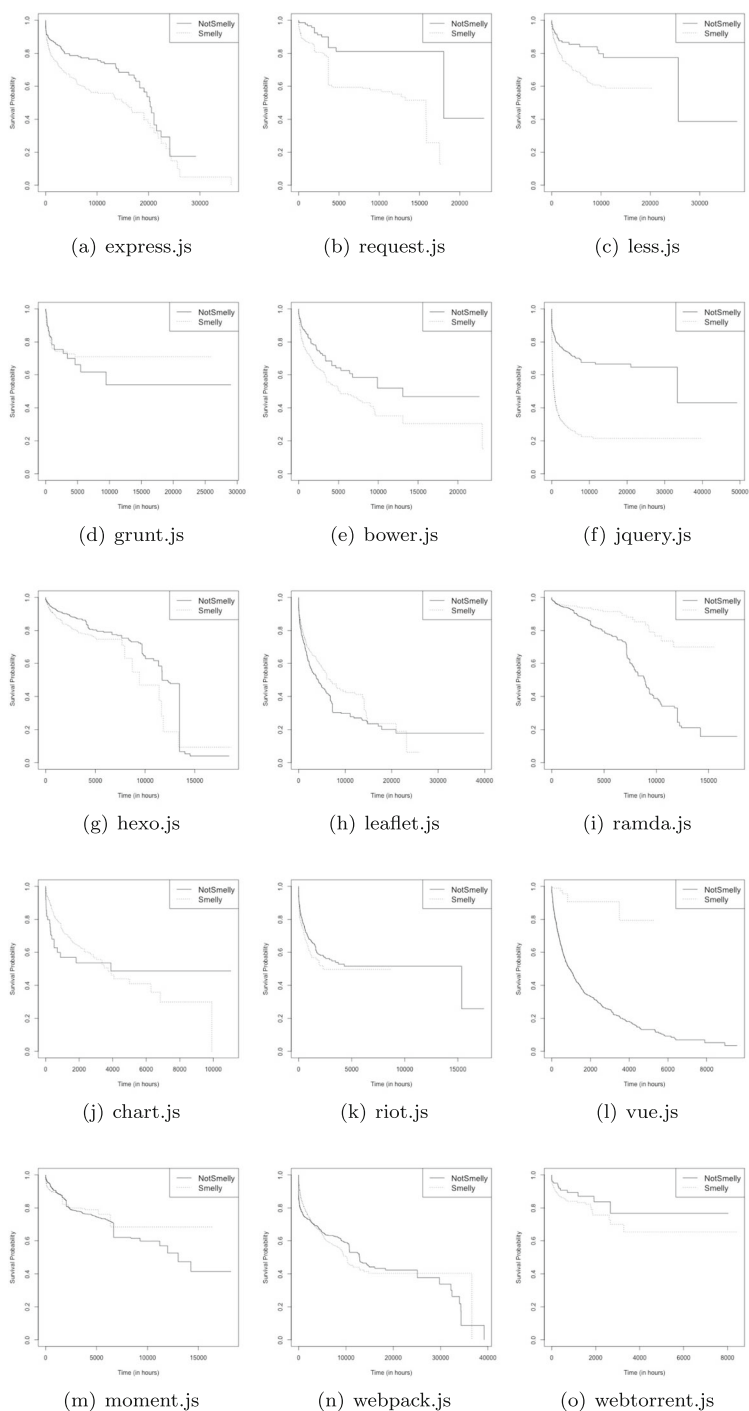
Fig. 4 Survival probability trends of smelly codes vs. non-smelly codes in our 15 JavaScript projects with the line grain including dependencies approach

In this way, if a particular smell s is removed x days after its introduction, we will have the corresponding event metric equal to 1 from 0 to $x - 1$, and equal to 0 at the time x and after. When we report these informations for a studied system, the maximum time that we take in account, for a particular smell type, corresponds to the maximum lifetime of the smells of this type. Thereby, for each smell type of the system and for each time, we will know the proportion of smells alive relatively to the number of smells created. This will particularly help us in the Cox survival model design.

Similar to **RQ1**, for each of the 12 studied smells, and for each of the 15 studied systems, we create an individual Cox survival model using the **Time** and **Event** metrics defined above. We use the *survfit* and *coxph* functions from R (Therneau 2000) to analyze our Cox survival models.

4 Case study results

In this section, we report and discuss the results for each research question.

(RQ1) Is the risk of fault higher in files with code smells in comparison with those without code smell?

Approach We use our framework described in Section 3.2 (Fig. 1) to collect information about the occurrence of the 12 studied code smells in our 15 subject systems. Then, we apply the data analysis model described in Section 3.3 (RQ1) to our collected data.

Findings Table 4 (line grain results) and Fig. 4 (line grain including dependencies results) show that files containing code smells experience faults faster than files without code smells, which confirms the finding of Saboury et al. (2017) obtained at the file level, without matching faulty and smelly code lines. The Y-axis in Fig. 4 represents the probability of a file experiencing a fault occurrence. Hence, a low value on the Y-axis means a low *survival* rate (i.e., a high hazard or high risk of fault occurrence). For all 15 projects, we calculated relative hazard rates (using (3) from Section 3.3) between files containing code smells and files without code smells. Results show that, on average, files without code smells have hazard rates 33% lower than files with code smells in our line grain analysis, and 45% lower in our line grain analysis considering dependencies. It is normal to see this proportion decreasing in line grain analysis (in comparison with the file grain analysis of Saboury et al. 2017), because we add an additional matching condition to set the *event* to 1. Between the line grain analysis and the line grain including dependencies analysis, the proportion increased slightly because considering dependencies increases the number of faulty lines that are related to code smells. We performed a *log-rank* test comparing the survival distributions of files containing code smells and files without any of the studied code smells and obtained p values lower than 0.05 for most of the 15 studied systems. Hence, we reject H_0^1. Since our detection of code smells depends on our selected threshold value (i.e., the top 10% value chosen in Section 3.2), we conducted a sensitivity analysis to assess the potential impact of this threshold selection on our result. More specifically, we rerun all our analysis with threshold values at top 20% and top 30%. We observed no significant differences in the results. Hence, we conclude that:

Table 5 Hazard ratios for each type of code smells (for the line grain analysis)

Module	Covariate	$exp(coef)$	p value (Cox hazard model)	p value (proportional hazards assumption)
express	No.Previous-Bugs	1.030	0	0.244
	Code Churn	1	0.677	0.688
	LOC	1.002	0	0.02
	Variable Re-assign	1.45	6.681e−6	0.265
grunt	No.Previous-Bugs	1.017	9.883e−5	0.445
	Code Churn	0.999	0.375	0.77
	LOC	1	0.649	0.586
	Assign. in Cond. State.	0.509	0.0346	0.349
	Chained Methods	0.42	4.479e−4	0.44
	Lengthy Lines	0.408	8.779e−6	0.597
bower	No.Previous-Bugs	1.053	0	0.559
	Code Churn	1	0.639	0.028
	LOC	1.001	1.01e−14	0.477
	Variable Re-assign	1.556	1.392e−5	0.679
	This Assign	0.583	3.399e−5	0.034
less	No.Previous-Bugs	1.022	0	0.373
	Code Churn	0.999	8.959e−4	0.456
	LOC	1	1.059e−5	0.197
	Variable Re-assign	1.638	1.658e−5	0.039
	Assign. in Cond. State.	1.263	0.056	0.953
request	No.Previous-Bugs	1.06	0	0.496
	Code Churn	1	0.928	0.748
	LOC	1.001	0	0.772
	This Assign	1.422	0.053	0.858
	Variable Re-assign	1.92	0.002	0.752
jquery	Code Churn	1	2.237e−4	2.021e−14
	LOC	1	0	0.125
	Lengthy Lines	1.524	0	0.422
	Assign. in Cond. State.	1.414	0.001	0.016
	Complex Code	1.283	1.39e−5	0.911
hexo	No.Previous-Bugs	1.197	0	0.05
	Code Churn	1.001	0.005	0.271
	LOC	1.001	0	0.082
	Complex Code	1.59	0.0756	0.247
	Long Methods	1.357	0.214	0.554
leaflet	No.Previous-Bugs	1.015	0	0.017
	Code Churn	1	0.708	1.327e−4
	LOC	1.0001	1.108e−6	0.124

Table 5   (continued)

Module	Covariate	$exp(coef)$	p value (Cox hazard model)	p value (proportional hazards assumption)
	Variable Re-assign	0.701	2.436e−9	0.567
	Complex Code	0.472	6.459e−4	0.048
	Chained Methods	0.462	8.513e−6	0.77
ramda	Code Churn	1	0.619	0.436
	LOC	1	0.002	0.328
	Nested Callbacks	0.364	5.916e−6	0.004
	Complex Code	0.603	0.475	0.446
	Chained Methods	0.261	0.021	0.582
chart	No.Previous-Bugs	1.056	0	0.993
	Code Churn	1	0.563	0.102
	LOC	1	0.466	0.454
	Nested Callbacks	0.287	0	5.257e−8
	This Assign	0.404	0	3.453e−6
	Long Parameter List	0.248	4.915e−9	0.057
riot	Code Churn	0.994	4.415e−7	0.267
	LOC	1	8.598e−5	0.555
	Variable Re-assign	1.02	0.794	0.103
	This Assign	0.135	4.648e−11	0.131
	Nested Callbacks	0.244	0.047	0.036
vue	Code Churn	0.998	1.464e−7	0.146
	LOC	1	1.11e−16	0.03
	Variable Re-assign	0.083	0	0.035
	This Assign	0.018	1.399e−8	0.145
moment	No.Previous-Bugs	1.015	0	0.474
	Code Churn	1	0.17	0.442
	LOC	1	0.383	0.969
	Complex Switch Case	2.849	9.324e−7	0.925
	Long Methods	1.061	0.776	0.698
webpack	Code Churn	1	0.622	7.173e−6
	LOC	1	0.44	0
	Variable Re-assign	1.131	0.008	0
	Extra Bind	0.554	0.238	0.628
webtorrent	No.Previous-Bugs	1.043	1.576e−5	0.27
	Code Churn	1	0.997	0.441
	LOC	1.001	7.963e−4	0.257
	Variable Re-assign	1.488	0.114	0.637
	Nested Callbacks	0.258	0.02	0.761

Higher $exp(coef)$ values means higher hazard rates

> *JavaScript files without code smells have hazard rates 33% lower than JavaScript files with code smells when considering fault occurrences on code smell lines (i.e., the line grain granularity), and this difference is statistically significant. If we consider the extended candidate fault lines described in Section 3.2 the hazard rate increases to 45%.*

JavaScript developers should be careful to avoid introducing code smells because they increase the probability of introducing faults. Also, refactoring should be considered to remove code smells since it could help reduced the fault-proneness of the systems.

(RQ2) Are JavaScript files with code smells equally fault-prone?

Approach Similar to **RQ1**, we use our framework from Section 3.2 (Fig. 1) to collect information about the occurrence of the 12 studied code smells in our 15 subject systems. Then, we apply the data analysis model described in Section 3.3 (RQ2) to our collected data.

Findings Tables 5 and 6 summarize the fault hazard ratios for the 12 studied code smells for respectively the line grain, and line grain including dependencies approach. The value in the column exp(coef) shows the amount of increase in hazard rate that one should expect for each unit increase in the value of the corresponding covariate. The last column of Tables 5, and 6 show that the p values obtained for the non-proportionality tests are above 0.05 for all the 15 systems; meaning that the proportional hazards assumption is satisfied for all the 15 studied systems.

Overall, the hazard ratios of the studied code smells vary across the systems and across the approaches (line grain and line grain including dependencies). With our line grain approach, *Variable Re-assign* has one of the highest hazard ratio in most systems, that is to say in nine out of 15 systems (60%); *This Assign* has one of the highest hazard rate in five out of 15 systems (33%); *Complex Code* and *Nested Callbacks* have one of the highest hazard rate in four out of 15 systems (27%); *Assignment in Conditional Statements* and *Chained Methods* have one of the highest hazard rate in three out of 15 systems (20%); *Lenghty Lines*, and *Long Methods* are the most hazard code smells in only two out of 15 systems (13%); *Long Parameter List*, *Extra Bind*, and *Complex Switch Case* are the most hazard code smells in only one out of 15 systems (7%); the other smells do not appear in any of the studied systems as having a high hazard ratio. With the line grain analysis that includes dependencies, the results are a little bit different. *Variable Re-assign* is still one of the most hazard code smell in most systems, in 14 out of 15 systems (93%); *Complex Code* has one of the highest hazard rate in six out of 15 systems (40%); *Assignment in Conditional Statements* has one of the highest hazard rate in five out of 15 systems (33%); *Long Methods* has one of the highest hazard rate in four out of 15 systems (27%); *Lenghty Lines* has one of the highest hazard rate in 3 out of 15 systems (20%); *Complex Switch Case*, *This Assign*, and *Chained Methods* are the most hazard code smells in only two out of 15 systems (13%); *Nested Callbacks* has one of the highest hazard rate in only one out of 15 systems (7%); the other smells do not have high hazard rates in any of the studied systems. Overall, the code smells with the highest hazard rates are *Variable Re-assign*, *Assignment in Conditional Statements*, and *Complex Code*. This is consistent across the two approaches (i.e., line grained and line grained including dependencies approaches), and consistent with the findings of Saboury and al. (2017).

Table 6 Hazard ratios for each type of code smells (for the line grain analysis that consider dependencies)

Module	Covariate	$exp(coef)$	p value (Cox hazard model)	p value (proportional hazards assumption)
express	No.Previous-Bugs	1.03	0	0.237
	Code Churn	1	0.542	0.695
	LOC	1.002	0	0.01
	Complex Code	3.504	0	0.184
	Long Methods	2.972	4.857e−12	0.25
	Variable Re-assign	1.572	2.581e−8	0.371
	No.Previous-Bugs	1.018	8.412e−6	0.495
	Code Churn	0.999	0.293	0.691
	LOC	1	0.787	0.542
	Variable Re-assign	1.123	0.284	0.079
	Complex Code	1.275	0.191	0.949
bower	No.Previous-Bugs	1.05	0	0.597
	Code Churn	1	0.972	0.017
	LOC	1.001	0	0.58
	Long Methods	2.664	7.421e−10	0.478
	Complex Code	1.768	0.001	0.966
	Variable Re-assign	1.72	7.232e−8	0.954
less	No.Previous-Bugs	1.022	0	0.312
	Code Churn	0.999	6.801e−4	0.457
	LOC	1	2.068e−5	0.172
	Variable Re-assign	1.74	1.102e−6	0.049
	Assign. in Cond. State.	1.37	0.008	0.973
request	No.Previous-Bugs	1.055	0	0.544
	Code Churn	1	0.785	0.595
	LOC	1.001	0	0.805
	Long Methods	1.886	0.053	0.711
	Complex Code	2.215	0.016	0.441
	Variable Re-assign	2.138	1.814e−4	0.731
jquery	Code Churn	1	1.551e−4	6.439e−15
	LOC	1	0	0.147
	Lengthy Lines	1.55	0	0.325
	Assign. in Cond. State.	1.414	0.001	0.016
	Variable Re-assign	2.996	0	6.306e−13
hexo	No.Previous-Bugs	1.199	0	0.077
	Code Churn	1.001	0.0156	0.262
	LOC	1.001	0	0.081
	Assign. in Cond. State.	2.321	0.404	1
	Complex Code	1.59	0.076	0.247
	Variable Re-assign	1.536	3.307e−7	0.725

Table 6 (continued)

Module	Covariate	$exp(coef)$	p-value (Cox hazard model)	p-value (Proportional hazards assumption)
leaflet	Code Churn	1	0.727	8.189e−5
	LOC	1	8.948e−14	0.039
	Lenghty Lines	1.422	2.957e−4	2.139e−5
	Complex Switch Case	2	0.001	0.014
	Assign. in Cond. State.	2.838	1.11e−15	0.013
ramda	Code Churn	1	0.614	0.436
	LOC	1	0.002	0.334
	Complex Code	0.603	0.475	0.446
	Variable Re-assign	0.977	0.831	0.818
chart	No.Previous-Bugs	1.057	0	0.867
	Code Churn	1	0.458	0.071
	LOC	1	0.537	0.325
	Variable Re-assign	0.921	0.502	0
	This Assign	0.452	0	4.319e−6
riot	Code Churn	0.994	1.548e−7	0.242
	LOC	1	4.61e−5	0.52
	Variable Re-assign	1.134	0.096	0.037
	Chained Methods	0.253	1.101e−4	0.664
	Nested Callbacks	0.244	0.047	0.036
vue	Code Churn	0.998	1.441e−7	0.146
	LOC	1	1.11e−16	0.031
	Variable Re-assign	0.088	0	0.039
	Chained Methods	0.108	6.842e−7	0.007
moment	No.Previous-Bugs	1.014	0	0.465
	Code Churn	1	0.137	0.38
	LOC	1	0.38	0.964
	Complex Switch Case	3.349	9.775e−10	0.948
	Long Methods	1.062	0.776	0.698
	Variable Re-assign	1.187	0.006	8.558e−9
webpack	Code Churn	1	0.456	2.715e−6
	LOC	1	0.18	0
	Variable Re-assign	1.231	3.232e−6	0
	Lengthy Lines	1.002	0.984	1.112e−6
	Assign. in Cond. State.	1.096	0.873	0.787
webtorrent	Code Churn	1	0.783	0.46
	LOC	1.001	1.711e−6	0.189
	Variable Re-assign	1.785	0.019	0.672
	This Assign	1.512	0.034	0.557

Higher $exp(coef)$ values means higher hazard rates

(a) Variable Re-Assign   (b) Chained Methods   (c) Lengthy Lines
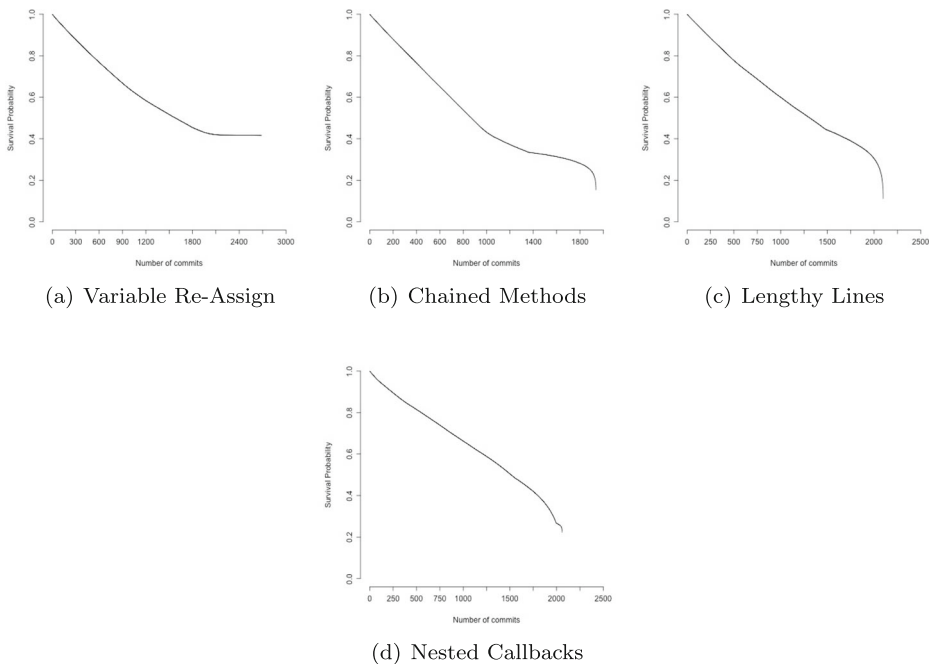
(d) Nested Callbacks

Fig. 5 Survival analyzes of the largest smells of bower.js, with commit scale

As we expected, in both analysis, the covariates No.Previous-Bugs is significantly related to fault occurrence; it appears in eight out of 15 systems with an *exp(coef)* greater than 1 and significant *p* values. However, its hazard rate is lower than those of many of the studied code smells. LOC is significantly related to fault occurrence in seven systems (less than half of the studied systems) with very low hazard rates, meaning that JavaScript developers cannot simply control for size and monitor files with previous fault occurrences, if they want to track fault-prone files effectively. Since *Variable Re-assign*, *Assignment in Conditional Statements*, and *Complex Code* are related to high hazard ratios in respectively 77%, 27%, and 33% of the studied systems. We strongly recommend that developers prioritize files containing these three types of code smells during testing and maintenance activities.

> *JavaScript files containing different types of code smells are not equally fault-prone. Developers should consider refactoring files containing either* Variable Re-assign *code smell, or* Assignment in Conditional Statements *code smell, or Complex Code smell in priority since they seem to increase the risk of faults in the system.*

A distinctive characteristic of JavaScript is its highly dynamic nature, which allows developers to perform some shortcuts like reassign variables, or verifying a condition while assigning variables. We hit in our study a paradox because one of the strength of JavaScript seems to be also one of its weakness, as re-assign and assignment in conditional statements

Table 7 Descriptive statistics on survival over time of the largest smells of the studied systems

System	Smell	Not survived	Survived	Number created at file birth	Median days or survival	Average days of survival
express	Variable Re-assign	6743	425	5783 (80.7%)	74	209
	Nested Callbacks	314	728	417 (40%)	1101	1152
	Complex Code	374	5	353 (93.1%)	74	122
	Long Methods	283	9	260 (89%)	74	143
	SUM	8430	1238	7348 (76%)		
grunt	Variable Re-assign	2210	317	1636 (64.7%)	248	411
	Lengthy Lines	243	108	172 (49%)	248	681
	Complex Code	91	5	83 (85.4%)	248	292
	Long Methods	55	5	41 (68.3%)	248	334
	SUM	2732	448	2017 (63.4%)		
bower	Variable Re-assign	1427	1801	1235 (38.3%)	797	777
	Chained Methods	82	96	35 (19.7%)	1231	819
	Lengthy Lines	32	50	28 (34.1%)	644	719
	Nested Callbacks	38	32	27 (38.6%)	163	656
	SUM	1647	2087	1368 (36.6%)		
less	Variable Re-assign	36979	3779	37303 (91.5%)	56	281
	Assign. in Cond. State.	1349	4	1261 (93.2%)	405	477
	Lengthy Lines	547	26	509 (88.8%)	36	139
	This Assign	392	33	387 (91.1%)	129	314
	SUM	40241	3927	40391 (91.4%)		
request	Variable Re-assign	1362	667	1140 (56.2%)	365	534
	This Assign	28	50	34 (43.6%)	874	743
	Chained Methods	32	22	38 (70.4%)	256	557
	Nested Callbacks	1	28	1 (3.4%)	774	592
	SUM	1455	777	1232 (55.2%)		
jquery	Variable Re-assign	13076	5156	12122 (66.5%)	528	694
	Complex Switch Case	146	15	130 (80.7%)	179	435
	This Assign	118	37	120 (77.4%)	539	716
	Chained Methods	59	58	35 (29.9%)	657	785
	SUM	13743	5356	12675 (66.4%)		
hexo	Variable Re-assign	19023	823	17626 (88.8%)	2	86
	Lengthy Lines	768	12	675 (86.5%)	10	138
	Long Parameter List	755	3	728 (96%)	2	20
	Complex Code	599	19	576 (93.2%)	2	51
	SUM	22522	980	20584 (87.6%)		
leaflet	Variable Re-assign	5856	498	2241 (35.3%)	789.5	734
	Lengthy Lines	733	270	358 (35.7%)	203	354
	Nested Callbacks	123	489	114 (18.6%)	986	1029
	Long Methods	77	5	32 (39%)	911	752
	SUM	6997	1278	2855 (34.5%)		

Table 7 (continued)

System	Smell	Not Survived	Survived	Number created at file birth	Median days or survival	Average days of survival
ramda	Variable Re-assign	4720	1078	4656 (80.3%)	375	391
	Chained Methods	365	101	344 (73.8%)	241	372
	Long Parameter List	176	90	208 (78.2%)	206	396
	Complex Code	194	28	200 (90.1%)	375	364
	SUM	5668	1340	5599 (79.9%)		
chart	Variable Re-assign	5297	5696	4538 (41.3%)	406	365
	This Assign	199	388	86 (14.7%)	406	339
	Lengthy Lines	119	14	34 (25.6%)	12	154
	Complex Switch Case	53	30	31 (37.3%)	169	258
	SUM	5740	6207	4746 (39.7%)		
riot	Variable Re-assign	8331	2625	7866 (71.9%)	52	188
	Lengthy Lines	193	33	206 (91.2%)	7	150
	This Assign	63	92	61 (39.4%)	43	110
	Assign. in Cond. State.	107	47	80 (51.9%)	100.5	196
	SUM	9119	2937	8637 (71.6%)		
vue	Variable Re-assign	4199	5833	6587 (65.7%)	139	175
	Lengthy Lines	2947	4208	3518 (49.2%)	125	143
	Complex Code	414	675	679 (62.4%)	139	171
	Long Methods	259	417	421 (62.3%)	139	171
	SUM	8612	11712	12129 (59.7%)		
moment	Variable Re-assign	5642	11063	6154 (36.8%)	450	486
	Nested Callbacks	19	335	12 (3.4%)	492	464
	Complex Switch Case	117	69	114 (61.3%)	299	634
	Chained Methods	113	42	74 (47.7%)	243	389
	SUM	6135	11590	6561 (37%)		
webpack	Variable Re-assign	4643	627	3192 (60.6%)	352	593
	Nested Callbacks	379	54	276 (63.7%)	104	359
	Chained Methods	371	16	174 (45%)	492	589
	Lengthy Lines	182	38	86 (39.1%)	236	518
	SUM	5970	813	3987 (58.8%)		
webtorrent	Variable Re-assign	709	424	471 (41.6%)	335	370
	This Assign	108	53	88 (54.7%)	335	427
	Nested Callbacks	28	55	23 (27.7%)	453	406
	Chained Methods	12	2	9 (64.3%)	19.5	122
	SUM	869	535	591 (42.1%)		

smells increase significantly the fault-proneness of JavaScript projects. Developers should be careful to not use JavaScript dynamism to justify bad development practices (a slightly longer piece of code is better than a smelly piece of code!).

(a) Variable Re-Assign   (b) Nested Callbacks   (c) Complex Code

(d) Long Methods
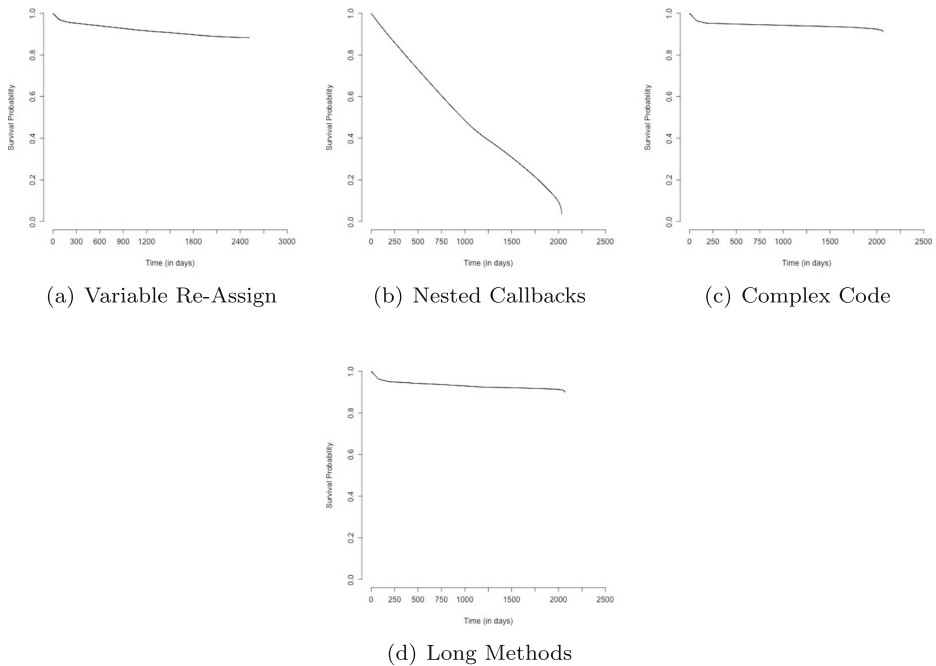
Fig. 6 Survival analyzes of the largest smells of express.js

Similar to **RQ1**, we conducted a sensitivity analysis to assess the potential impact of our threshold selection (performed during the detection of code smells) on the results; rerunning the analysis using threshold values at top 20% and top 30%. We did not observed any significant change in the results.

(RQ3) How long do code smells survive in JavaScript projects?

Approach We use the framework described in Section 3.2, Fig. 2, to collect information about the appearance of the 12 studied code smells in our 15 subject systems, as well as their line localization, their content and their genealogy (which means their evolution over time from their creation to either their destruction, or the last revision of the studied system). Then, we apply the data analysis model described in Section 3.3 (RQ3) to our collected data. As mentioned earlier, the time scale of our survival analysis is the number of days, and could have been the number of commits. Indeed, knowing how many days a smell survives in the system is more demonstrative than knowing how many commits it survives, and the number of commits scale will not bring more relevant information since the studied systems are regularly active almost everyday. Figures 8 and 5 represent the main survival smells results for the project bower.js, respectively, with number of days and commits scales, and show that both scales imply the same trend about these results. All the results with both scales appear in the GitHub repository of the project. For the results and the conclusions of this section, the number of days scale is used.

Findings Our results are presented in Table 7 for a density analysis, and in Figs. 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, and 20 for a survival analysis. For Table 7, for

(a) Variable Re-Assign   (b) Lengthy Lines   (c) Complex Code

(d) Long Methods
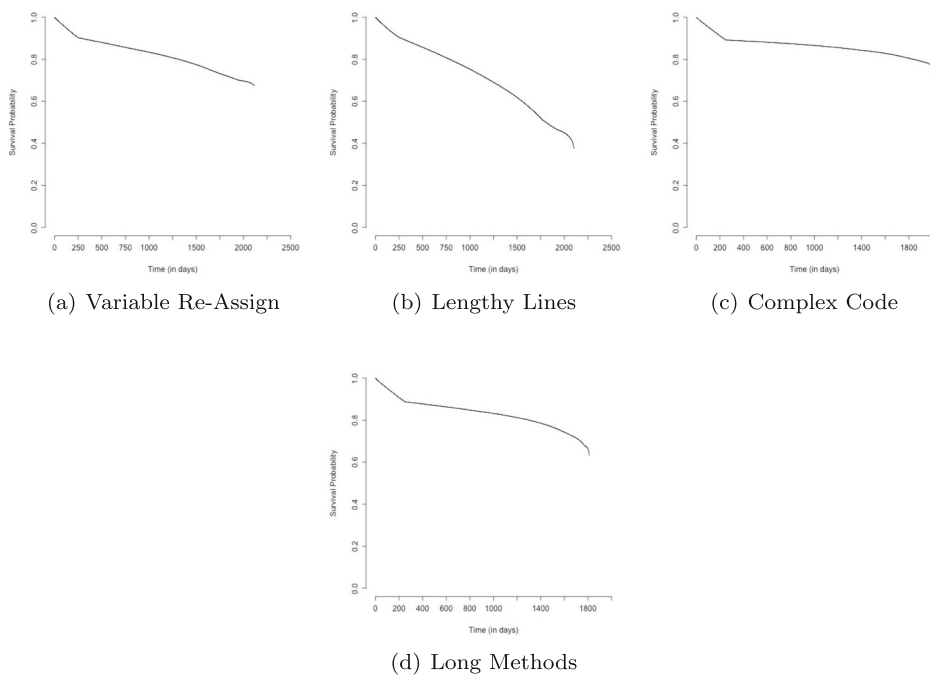
Fig. 7 Survival analyzes of the largest smells of grunt.js

(a) Variable Re-Assign   (b) Chained Methods   (c) Lengthy Lines

(d) Nested Callbacks

Fig. 8 Survival analyzes of the largest smells of bower.js

(a) Variable Re-Assign   (b) Nested Callbacks   (c) Complex Code
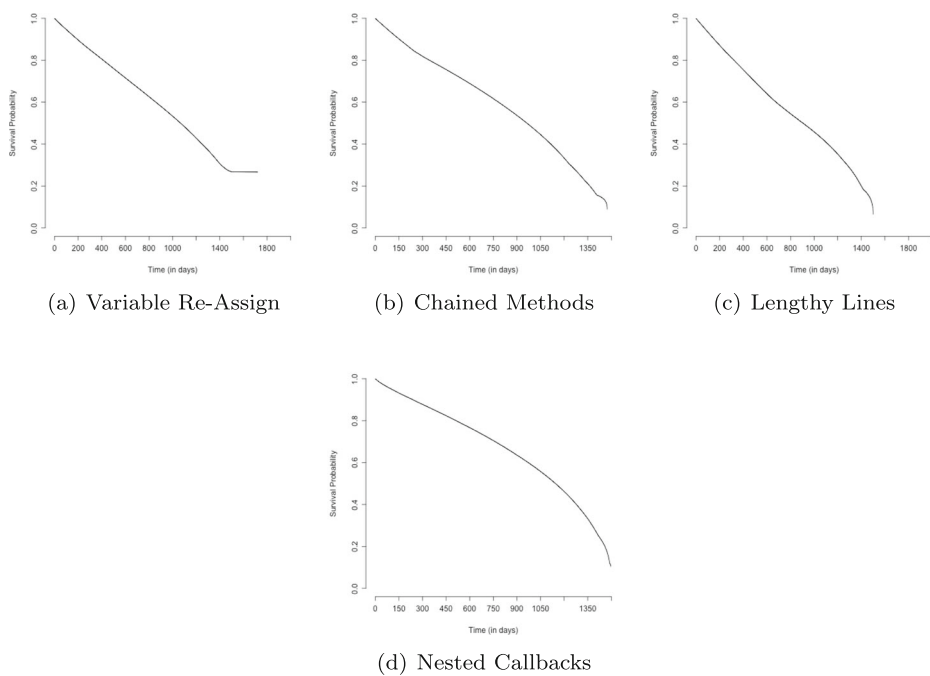
(d) Long Methods
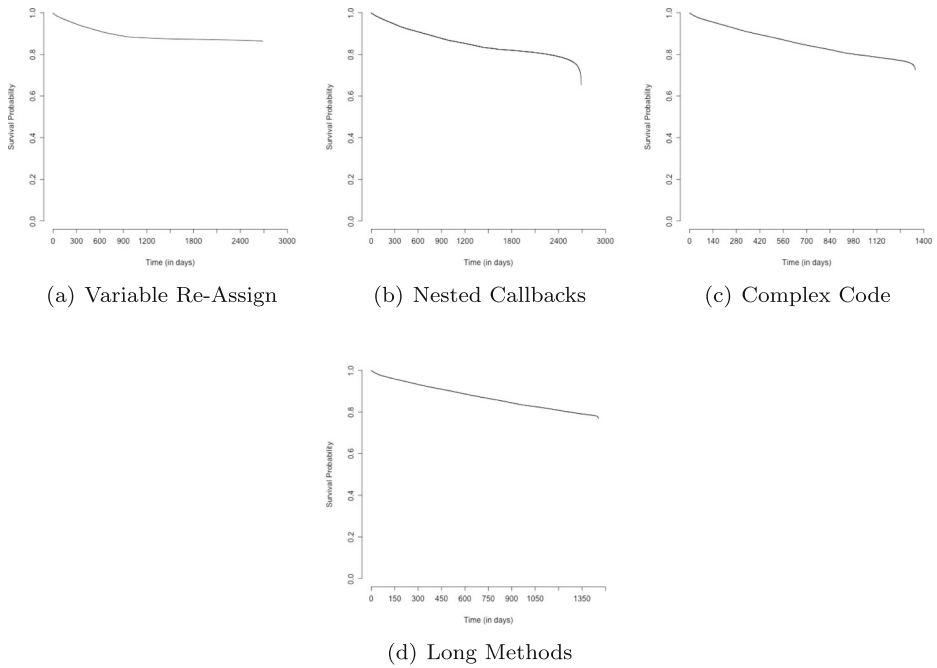
Fig. 9 Survival analyzes of the largest smells of less.js

(a) Variable Re-Assign   (b) This Assign   (c) Chained Methods

(d) Nested Callbacks
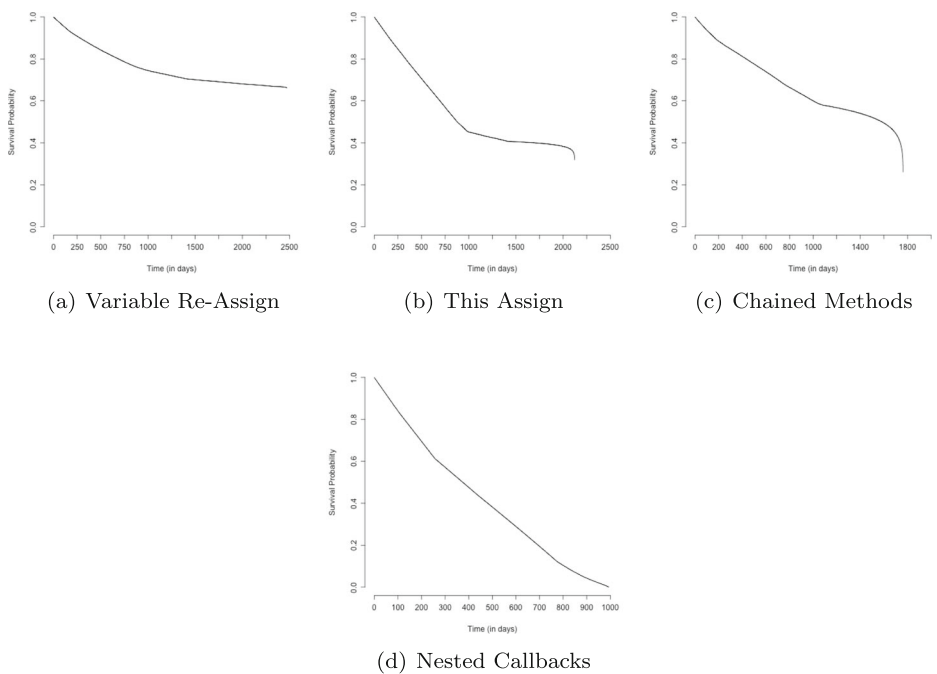
Fig. 10 Survival analyzes of the largest smells of request.js

(a) Variable Re-Assign

(b) Complex Switch Case

(c) This Assign

(d) Chained Methods
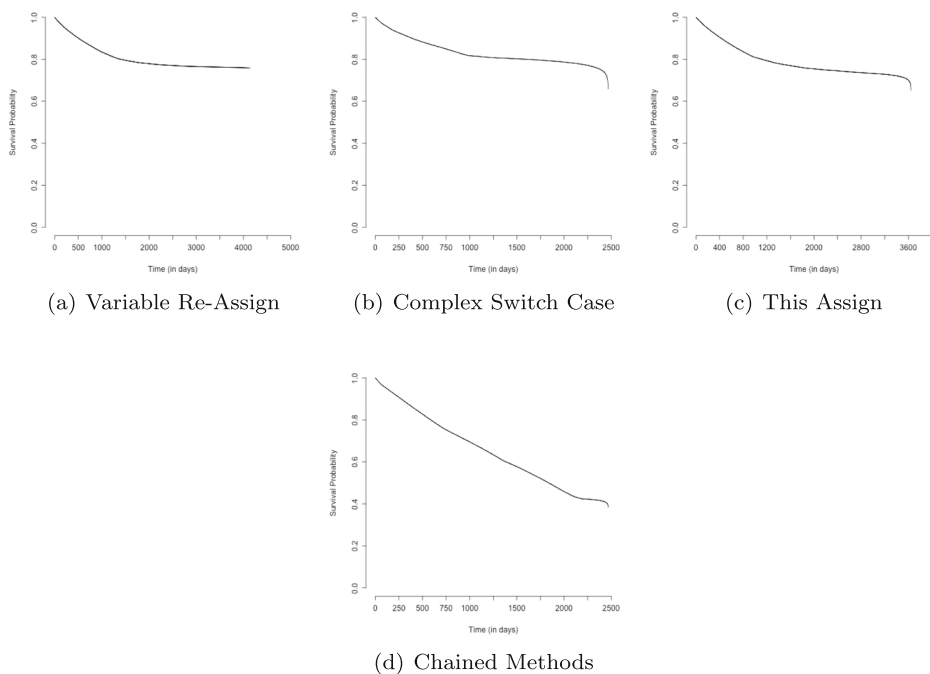
Fig. 11 Survival analyzes of the largest smells of jquery.js

(a) Variable Re-Assign

(b) Lengthy Lines

(c) Long Parameter List

(d) Complex Code
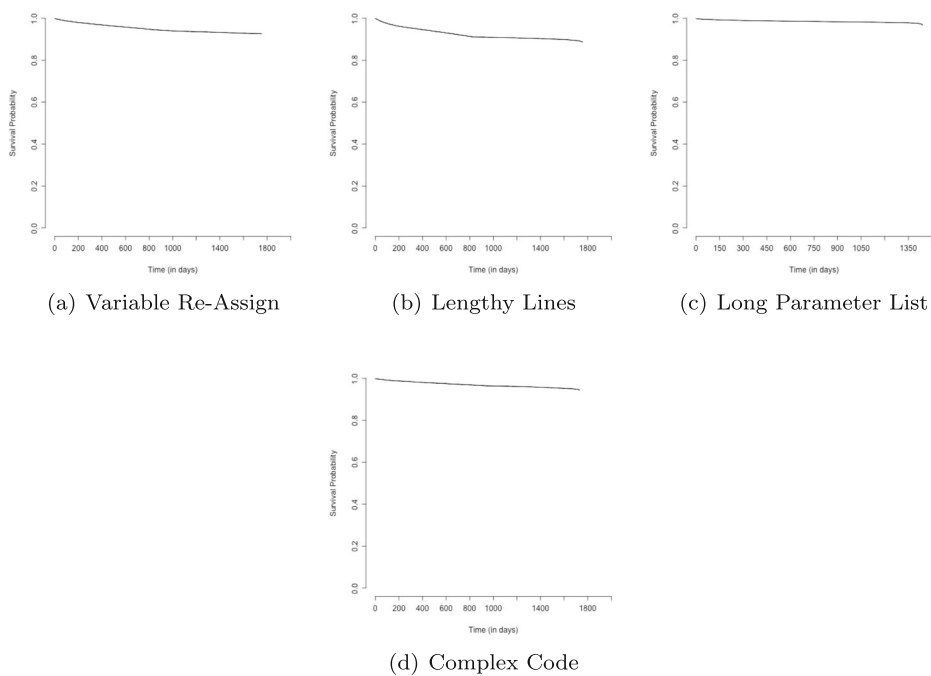
Fig. 12 Survival analyzes of the largest smells of hexo.js

(a) Variable Re-Assign   (b) Lengthy Lines   (c) Nested Callbacks

(d) Long Methods
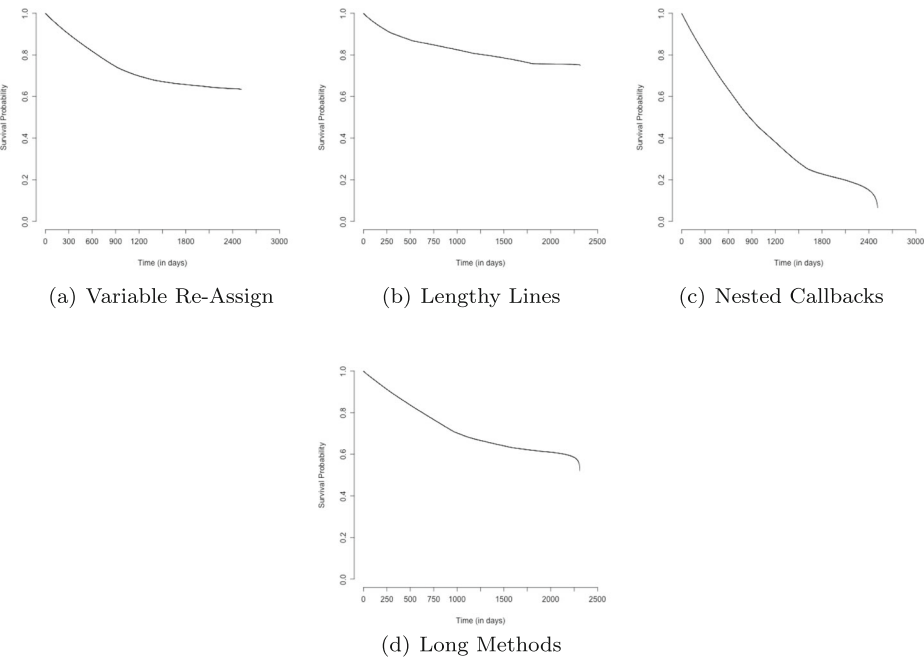
Fig. 13 Survival analyzes of the largest smells of leaflet.js

(a) Variable Re-Assign   (b) Chained Methods   (c) Long Parameter List

(d) Complex Code
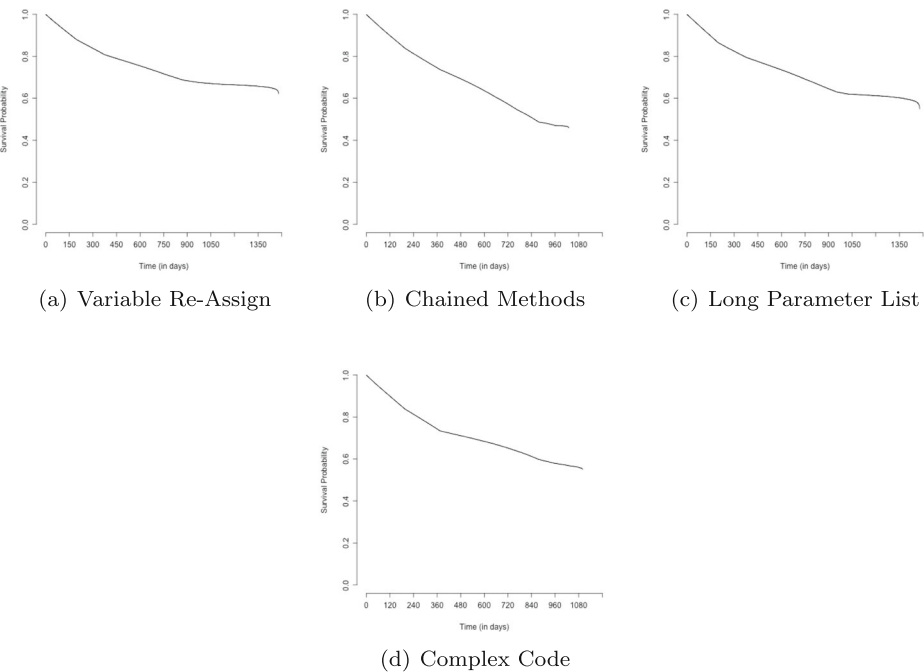
Fig. 14 Survival analyzes of the largest smells of ramda.js

(a) Variable Re-Assign   (b) This Assign   (c) Lengthy Lines

(d) Complex Switch Case
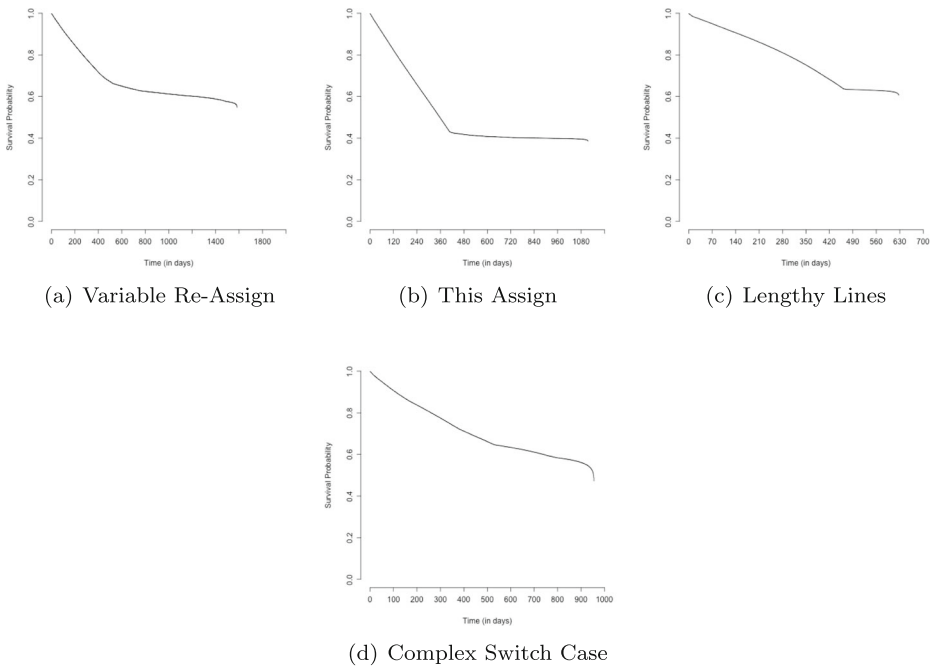
Fig. 15 Survival analyzes of the largest smells of chart.js

(a) Variable Re-Assign   (b) Lengthy Lines   (c) This Assign

(d) Assignment in Conditional
Statement
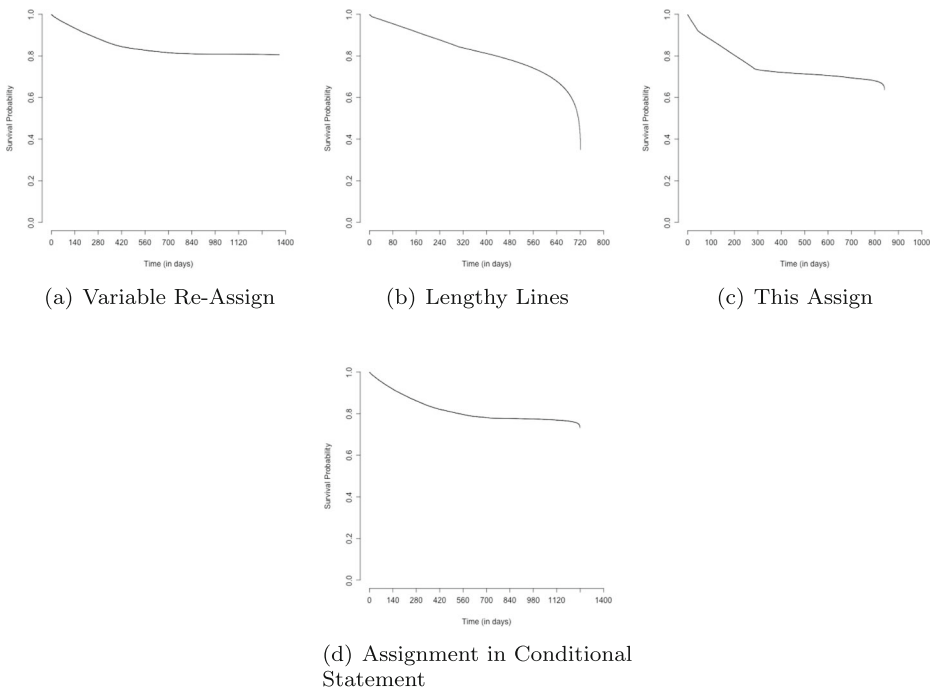
Fig. 16 Survival analyzes of the largest smells of riot.js

each system and each smell, the third column corresponds to the number of killed smells (i.e.,smells that have been removed), and the fourth to the number of survived smells. The sum of both columns gives us the number of created smells. The fifth column reports, in percentage, the proportion of smells created at the files birthdate, relatively to the number of created smells. In order to not overload the presentation of our results, we only report the descriptive statistics for the four most relevant smells, which means those for which the number of created is the most considerable. Finally, Table 7 reports, for each studied system, general statistics (*SUM* lines), computed by summing the statistics of the 12 studied smells. For Figs. 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, and 20, we plot the survival analysis for each studied system, and for each smells reported in Table 7, still in order to not overload the presentation of our results. The Y-axis corresponds to the chance of surviving of a given smell type, x days after its introduction into the code base.

The results presented in Table 7 show that smells are not often introduced during files evolution and changes, but rather at the creation of files. Indeed, when we look at the *SUM* lines, from 34.5% (for *leaflet*) to 91.4% (for *less*) of the smells are introduced at the file birthdate, meaning that developers should pay attention to the quality of their code when they create a JavaScript file, because it is precisely at this moment that most of the smells are introduced into the system. We also notice that, for the major part of the studied systems (eight out of 15), more than 20% of the smells created still survive presently; and for 13 systems, more than 10% of the smells created are now present in these systems. It reveals that a significant part of the smells are never removed from the system once they are introduced in the code. Plus, after analyzing the commits of the studied systems, it is interesting to notice that most of the time, the killed smells are removed at the same time as the file containing them (and not because of a refactoring).

Table 7 gives us also an overview of the smells lifetime, and we observe that for most of the systems (nine out of 15), the median and average days of survival of the most significant smell types are greater than 100 days; and for 14 systems out of 15 (except *hexo*), at least one of the most sizable smell types has an average and median lifetime greater than 100 days. This observation highlights that in general, smells tend to survive a very long time inside the system once they are introduced. Finally, Table 7 also presents an interesting result, which is that the smell *Variable Re-assign* is always the most considerable smell type (in our 15 studied systems) in term of number of created smells, and its survival rate follows the trend of the sum of the smells (when we consider all the smells created in the system). For every studied system, over 1000 *Variable Re-assign* smells are created, and for eight systems out of 15, the number of created smells of this type exceeds 10,000. Once again, *Variable Re-assign* is at the heart of our analysis, because as said previously, it is one of the most risky smell in terms of fault-proneness.

According to Figs. 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, and 20, the four most significant smell types of each studied system have a considerable chance of surviving 500 days after their introduction. This is indeed the case for all the most significant smell types for nine systems out of 15 (except *request*, *chart*, *moment*, *webtorrent*, and *vue*), with over 50% chance of surviving 500 days after the introduction of their largest smell types. Also, for 14 studied systems out of 15 (except *vue*), at least one of the most sizable smell types has more than 50% chance of surviving 500 days after its smells introduction. Plus, for 12 systems out of 15 (except *bower*, *vue*, and *webtorrent*), the *Variable Re-assign* smell type has more than 50% chance of surviving 1500 days after its introduction. These observations show the trend of the smells of the studied systems to be persistent and survive a long time after their introduction in the code, and also the significance of *Variable Re-assign*, which
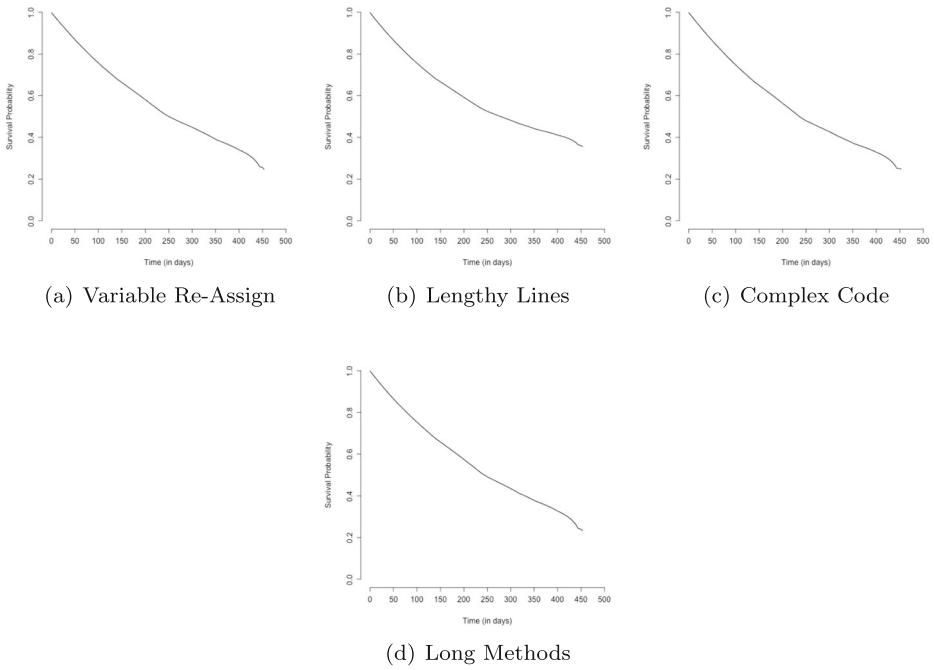
(a) Variable Re-Assign   (b) Lengthy Lines   (c) Complex Code

(d) Long Methods

Fig. 17 Survival analyzes of the largest smells of vue.js

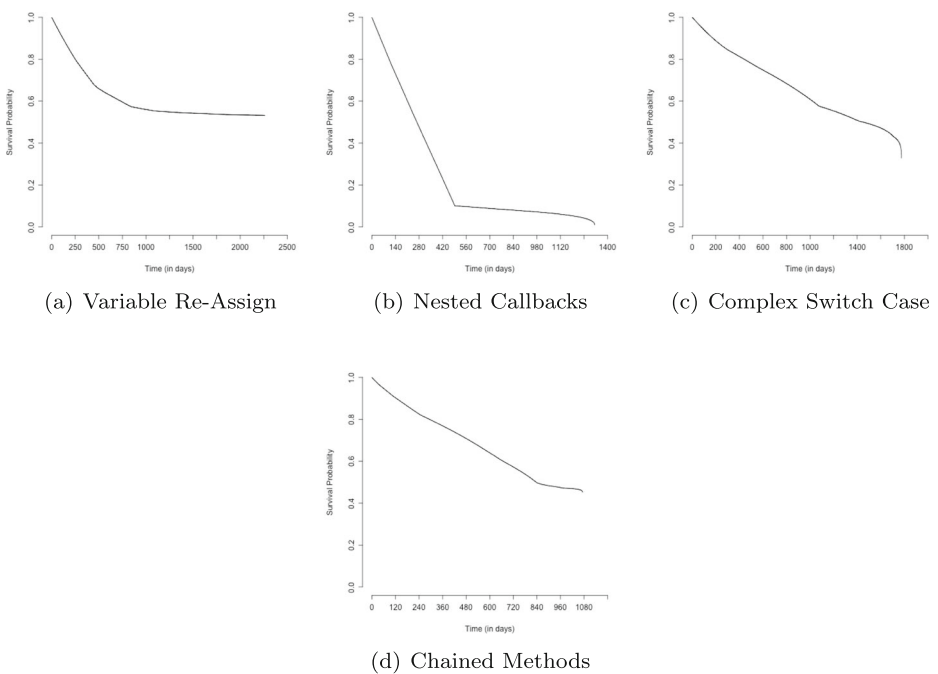(a) Variable Re-Assign   (b) Nested Callbacks   (c) Complex Switch Case
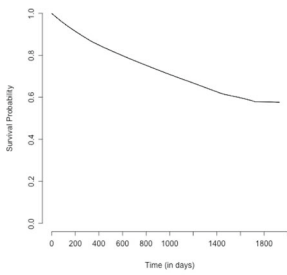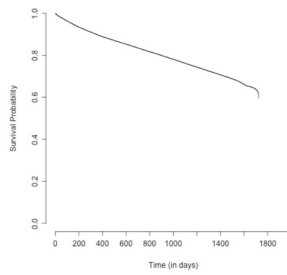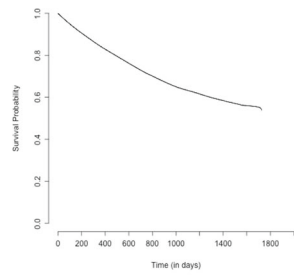
(d) Chained Methods

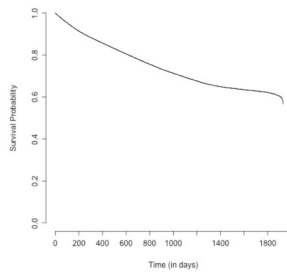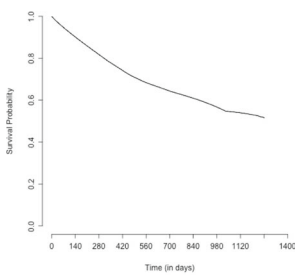**Fig. 18** Survival analyzes of the largest smells of moment.js

(a) Variable Re-Assign

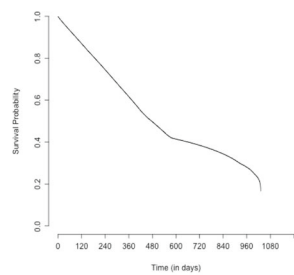(b) Nested Callbacks

(c) Chained Methods

(d) Lengthy Lines

Fig. 19 Survival analyzes of the largest smells of webpack.js

(a) Variable Re-Assign
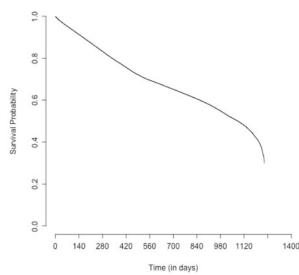
(b) This Assign

(c) Nested Callbacks
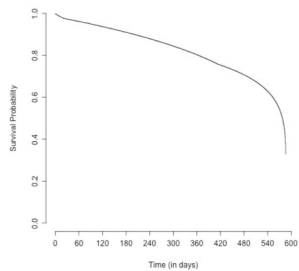
(d) Chained Methods

Fig. 20 Survival analyzes of the largest smells of webtorrent.js

is strongly linked to fault-proneness, and is the most proliferated smell type in the studied systems, with a very high chance of surviving over time.

> *Most of the studied smells (from 34.5% to 91.4%) are intro-duced during the creation of JavaScript files. Once intro-duced, in most of the systems (eight systems), more than 20% of the studied smells are not removed and have a high chance of surviving a very long time. Plus, Variable Re-assign, which is the most fault-prone smell is also the smell with the highest chance of surviving over time.*

Even if JavaScript is highly dynamic and allows reassign variables, we show here that re-assign smells are bad practices widely used, rarely removed from the projects, and more likely to introduce faults in systems. This observation is not so much surprising, as reassign statements make the code particularly unreadable, especially when they are widely present in a same portion of code. Hence, developers need to pay attention when they introduce reassign variables. They should consider refactoring their code to remove re-assign smells regularly.

5 Threats to validity

In this section, we discuss the threats to validity of our study following common guidelines for empirical studies (Yin 2002).

Construct validity threats concern the relation between theory and observation. In our study, threats to the construct validity are mainly due to measurement errors. The number of previous faults in each source code file was calculated by identifying the files that were committed in a fault fixing revision. This technique is not without flaws. We identified fault fixing commits by mining the logs searching for certain keywords (i.e.,"fix," "#," and "gh-") and certain bug IDs, as explained in Section 3.2. Following this approach, we are not able to detect fault fixing revisions if the committer either misspelled the keywords, or failed to include any commit message, or includes in a non common way the bug IDs. Nevertheless, this heuristic was successfully used in multiple previous studies in software engineering (Jaafar et al. 2013; Shihab et al. 2013). The SZZ heuristic used to identify fault-inducing commits is not 100% accurate. However, it has been successfully used in multiple previous studies from the literature, with satisfying results. In our implementation, we remove all fault-inducing commit candidates that only changed blank or comment lines, and all of those that were too far from the issue date. We followed all the recommendations proposed by Da Costa et al. (2017) to improve the accuracy of the SZZ heuristic. Also, as explained in Section 3.2, we only analyzed commits from the master branch of each project. Even if all the other branches information are represented in the master branch, the time introduction of bugs and smells can be different. However, the issue of time difference between master branch introduction and other branches introduction of the studied smells or bugs concerns a few data of our extraction. When tracking smells over time during the construction of the genealogies of the smells, we set a similarity threshold of 70%, meaning that if two smells of the same type have a similarity greater than 70%, they are considered to

be the same. Obviously, this threshold is not perfect and can associate two different smells together, or dissociate two smells, which are in reality the same. However, we performed a sensitivity analysis, with threshold values at 70%, 80%, and 90% and obtained similar results.

Internal validity threats concern our selection of systems and tools. The metric extraction tool used in this paper is based on the AST provided by ESLint. The results of the study are therefore dependent on the accuracy of ESLint. However, we are rather assured that this tool functions properly as it is being used widely by big companies, e.g., Facebook, Paypal, and Airbnb. We chose a logarithmic link function for some of our covariates in the survival analysis. It is possible that a different link function would be a better choice for these covariates. However, the non-proportionality test implies that the models were a good fit for the data. Also, we do not claim causation in this work, we simply report observations and correlations and tries to explain these findings.

Threats to conclusion validity address the relationship between the treatment and the outcome. We are careful to acknowledge the assumptions of each statistical test.

Threats to external validity concern the possibility to generalize our results. In this paper, we have studied 15 large JavaScript projects. We have also limited our study to open-source projects. Still, these projects represent different domains and various project sizes. Table 2 shows a summary of the studied systems, their domain and their size. Nevertheless, further validation on a larger set of JavaScript systems, considering more types of code smells, is desirable.

Threats to reliability validity concern the possibly of replicating our study. In this paper, we provide all the details needed to replicate our study. All our 15 subject systems are publicly available for study. The data and scripts used in this study is also publicly available on GitHub.[25]

6 Related work

In this section, we discuss the related literature on code smell and JavaScript systems. Code Smells (Fowler 1997) are poor design and implementation choices that are reported to negatively impact the quality of software systems. They are opposite to design patterns (Gamma et al. 1995) which are good solutions to recurrent design problems. The literature related to code smells generally falls into three categories: (1) the detection of code smells (e.g., Khomh et al. 2011; Fard and Mesbah 2013); (2) the evolution of code smells in software systems (e.g., Chatzigeorgiou and Manakos 2010; Olbrich et al. 2009; Peters and Zaidman 2012; Tufano et al. 2015) and their impact on software quality (e.g., Shatnawi and Li 2006; Khomh et al. 2012; Abbes et al. 2011; Jaafar et al. 2013; Tufano et al. 2015; D'Ambros et al. 2010; Palomba et al. 2018); and (3) the relationship between code smells and software development activities (e.g., Sjoberg et al. 2013; Abbes et al. 2011). Our work in this paper falls into the second category. We aim to understand how code smells affect the fault-proneness of JavaScript systems. Li and Shatnawi (2006) who

---

[25] https://github.com/DavidJohannesWall/smells_project

investigated the relationships between code smells and the occurrence of errors in the code of three different versions of Eclipse reported that code smells are positively associated with higher error probability. In the same line of study, Khomh et al. (2012) investigated the relationship between code smells and the change- and fault-proneness of 54 releases of four popular Java open-source systems (ArgoUML, Eclipse, Mylyn, and Rhino). They observed that classes with code smells tend to be more change- and fault-prone than other classes. Tufano et al. (2015) investigated the evolution of code smells in 200 open-source Java systems from Android, Apache, and Eclipse ecosystems and found that code smells are often introduced in the code at the beginning of the projects, by both newcomers and experienced developers. Sjoberg et al. (2013), who investigated the relationship between code smells and maintenance effort reported that code smells have a limited impact on maintenance effort. However, Abbes et al. (2011) found that code smells can have a negative impact on code understandability. Recently, Fard et al. (2013) have proposed a technique named JNOSE to detect 13 different types of code smells in JavaScript systems. The proposed technique combines static and dynamic analysis. They applied JNOSE on 11 client-web applications and found "lazy object" and "long method/function" to be the most frequent code smells in the systems. WebScent (Nguyen et al. 2012) is another tool that can detect client-side smells. It identifies mixing of HTML, CSS, and JavaScript, duplicate code in JavaScript, and HTML syntax errors. ESLint (2015), JSLint (2019), and JSHint (2019) are rule-based static code analysis tools that can validate source codes against a set of best coding practices. Despite this interest in JavaScript code smells and the growing popularity of JavaScript systems, to the best of our knowledge, there is no study that examined the effect of code smells on the fault-proneness of JavaScript server-side projects. This paper aims to fill this gap. To remove code smells in software systems, developers often rely on refactorings, which are behavior preserving code transformations. Gatrell and al. (2015) explain how refactoring can be help improve the quality of software systems and reduce fault-proneness.

7 Conclusion

In this study, we examine the impact of code smells on the fault-proneness of JavaScript systems. We also examine the survival of code smells in JavaScript systems. Regarding fault-proneness, we compare the time until a fault occurrence in JavaScript files that contain code smells and files without code smells, tracking faults and smells at the line level of each file, in order to ensure that the considered faults affected smelly code lines. Results show that JavaScript files without code smells have hazard rates at least 33% lower than JavaScript files with code smells. In other terms, the survival of JavaScript files against the occurrence of faults increases with time if the files do not contain code smells. We further investigated hazard rates associated with different types of code smells and found that "Variable Re-assign," "Assignment in Conditional Statements," and "Complex Code" smells have the highest hazard rates. Regarding the survival of code smells in the systems, results show that code smells are generally introduced in the JavaScript files when the files are created. "Variable Re-assign" which occur frequently in JavaScript systems can survive for a very long time in a system. JavaScript developers should consider removing *Variable Re-assign* code smells from their systems in priority since this code smell is consistently associated with a high risk of fault. They should also prioritize *Assignment in Conditional Statements*, *Complex Code*, *This Assign*, *Nested Callbacks*, and *Long Parameter List* code smells for refactoring.

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

References

Abbes, M., Khomh, F., Guéhéneuc, Y.-G., Antoniol, G. (2011). 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 (CSMR)* (pp. 181–190).

Bower (2016). About Bower [Online; accessed 4-October-2016]. [Online]. Available: https://bower.io/docs/about/.

GitHub (2016a). Airbnb JavaScript style guide [Online; accessed 17-October-2016]. [Online]. Available: https://github.com/airbnb/javascript.

Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L. (1998). Clone detection using abstract syntax trees. In *1998. Proceedings of the International Conference on Software Maintenance. IEEE* (pp. 368–377).

Brodu, E., Frénot, S., Oblé, F. (2015). Toward automatic update from callbacks to promises. In *Proceedings of the 1st Workshop on All-Web Real-Time Systems. ACM* (p. 1).

Chatzigeorgiou, A., & Manakos, A. (2010). Investigating the evolution of bad smells in object-oriented code. In *2010 7th International Conference on the Quality of Information and Communications Technology (QUATIC). IEEE* (pp. 106–115).

D'Ambros, M., Bacchelli, A., Lanza, M. (2010). On the impact of design flaws on software defects. In *2010 10th International Conference on Quality Software (QSIC). IEEE* (pp. 23–31).

da Costa, D.A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., Hassan, A.E. (2017). A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, *43*(7), 641–657.

Eslint (2015). The pluggable linting utility for javascript and jsx. http://eslint.org/.

Fard, A.M., & Mesbah, A. (2013). Jsnose: Detecting javascript code smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE* (pp. 116–125).

Fischer, M., Pinzger, M., Gall, H. (2003). Populating a release history database from version control and bug tracking systems. In *2003. ICSM 2003. Proceedings. International Conference on Software Maintenance. IEEE* (pp. 23–32).

Fontana, F.A., Braione, P., Zanoni, M. (2012). Automatic detection of bad smells in code: an experimental assessment. *Journal of Object Technology*, *11*(2), 5–1.

Fowler, M. (1997). Refactoring: improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*.

Fox, J., & Weisberg, S. (2010). An R companion to applied regression Sage.

Gallaba, K., Mesbah, A., Beschastnikh, I. (2015). Don't call us, we'll call you: characterizing callbacks in Javascript. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM). IEEE* (pp. 1–10).

Gatrell, M., & Counsell, S. (2015). The effect of refactoring on change and fault-proneness in commercial c# software. *Science of Computer Programming*, *102*, 44–56.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). Design patterns: elements of reusable object oriented software.

GitHut (2016). Discover languages in GitHub [Online; accessed 11-August-2016]. [Online]. Available: http://githut.info/.

Jaafar, F., Guéhéneuc, Y.-G., Hamel, S., Khomh, F. (2013). Mining the relationship between anti-patterns dependencies and fault-proneness. In *WCRE* (pp. 351–360).

jQuery (2016). jQuery JavaScript style guide [Online; accessed 17-October-2016]. [Online]. Available: https://contribute.jquery.org/style-guide/js/.

JSLint (2019). JSLint: the JavaScript code quality tool. http://www.jslint.com.

JSHint (2019). JSHint: a static code analysis tool for JavaScript. http://jshint.com/.

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., Sahraoui, H. (2011). BDTEX: a GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software*, *84*(4), 559–572.

Khomh, F., Penta, M.D., Guéhéneuc, Y.-G., Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, *17*(3), 243–275. [Online]. Available: https://doi.org/10.1007/s10664-011-9171-y.

Khomh, F., Di Penta, M., Guéhéneuc, Y.-G., Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering*, *17*(3), 243–275.

Koru, A.G., Zhang, D., Liu, H. (2007). Modeling the effect of size on defect proneness for open-source software. In *Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering. IEEE Computer Society* (p. 10).

Koru, A.G., El Emam, K., Zhang, D., Liu, H., Mathew, D. (2008). Theory of relative defect proneness. *Empirical Software Engineering*, *13*(5), 473–498.

Mardan, A. (2014). Express.. js Guide: the comprehensive book on express. js. Azat Mardan.

Marinescu, R. (2004). Detection strategies: metrics-based rules for detecting design flaws. In *2004. Proceedings. 20th IEEE International Conference on Software Maintenance. IEEE* (pp. 350–359).

Marinescu, R., & Lanza, M. (2006). Object-oriented metrics in practice.

Mazinanian, D., & Tsantalis, N. (2016). Migrating cascading style sheets to preprocessors by introducing mixins. In *Proceedings of the 31st IEEE/ACM International Cconference on Automated Software Engineering. ACM* (pp. 672–683).

McCabe, T.J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, *4*, 308–320.

Neamtiu, I., Foster, J.S., Hicks, M. (2005). Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes*, *30*(4), 1–5.

Nguyen, H.V., Nguyen, H.A., Nguyen, T.T., Nguyen, A.T., Nguyen, T.N. (2012). Detection of embedded code smells in dynamic web applications. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE* (pp. 282–285).

GitHub (2016b). Node.js style guide [Online; accessed 17-October-2016]. [Online]. Available: https://github.com/felixge/node-style-guide.

NpmJS (2016). npm-coding-style [Online; accessed 17-October-2016]. [Online]. Available: https://docs.npmjs.com/misc/coding-style.

Olbrich, S., Cruzes, D.S., Basili, V., Zazworka, N. (2009). The evolution and impact of code smells: a case study of two open source systems. In *3rd International Symposium on Empirical Software Engineering and Measurement, ESEM 2009* (pp. 390–400).

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., De Lucia, A. (2018). On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, *23*(3), 1188–1221.

Peters, R., & Zaidman, A. (2012). Evaluating the lifespan of code smells using software repository mining. In *2012 16th European Conference on Software Mmaintenance and Reengineering (CSMR). IEEE* (pp. 411–416).

Pfenning, F., & Elliott, C. (1988). Higher-order abstract syntax. In *ACM SIGPLAN Notices, vol. 23, no. 7. ACM* (pp. 199–208).

Saboury, A., Musavi, P., Khomh, F., Antoniol, G. (2017). An empirical study of code smells in JavaScript projects. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE* (pp. 294–305).

Selim, G.M., Barbour, L., Shang, W., Adams, B., Hassan, A.E., Zou, Y. (2010). Studying the impact of clones on software defects. In *2010 17th Working Conference on Reverse Engineering. IEEE* (pp. 13–21).

Shatnawi, R., & Li, W. (2006). An investigation of bad smells in object-oriented design. In *2006. ITNG 2006. 3rd International Conference on Information Technology: New Generations. IEEE* (pp. 161–165).

Shihab, E., Ihara, A., Kamei, Y., Ibrahim, W.M., Ohira, M., Adams, B., Hassan, A.E., Matsumoto, K.-I. (2013). Studying re-opened bugs in open source software. *Empirical Software Engineering*, *18*(5), 1005–1042.

Singer, J.D., & Willett, J.B. (2003). Applied longitudinal data analysis: modeling change and event occurrence, Oxford University Press, Oxford.

Sjoberg, D.I.K., Yamashita, A., Anda, B., Mockus, A., Dyba, T. (2013). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, *39*(8), 1144–1156.

Stackoverflow (2016). Developer survey results 2016, [Online; accessed 11-August-2016]. [Online]. Available: http://stackoverflow.com/research/developer-survey-2016.

Śliwerski, J., Zimmermann, T., Zeller, A. (2005). When do changes induce fixes? In *ACM Sigsoft Software Engineering Notes, vol. 30, no. 4. ACM* (pp. 1–5).

Therneau, T. (2000). R survival package.

Therneau, T.M., & Grambsch, P.M. (2000). *Modeling survival data: extending the Cox model*. Berlin: Springer.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., Poshyvanyk, D. (2015). When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering-volume 1. IEEE Press* (pp. 403–414).

Westergaard, H. (1932). *Contributions to the history of statistics*. London: P.S King.

Grunt (2016). Who uses grunt [Online; accessed 4-October-2016]. [Online]. Available: http://gruntjs.com/who-uses-grunt.

Yin, R.K. (2002). Case study research: design and methods, 3rd, SAGE Publications, USA.

David Johannes is Software Engineer at GIRO. His research interests include software maintenance and evolution, empirical software engineering, and software analytic.

Foutse Khomh Foutse Khomh is an associate professor at Polytechnique Montréal, where he heads the SWAT Lab on software analytics and cloud engineering research (http://swat.polymtl.ca/). He received a Ph.D in Software Engineering from the University of Montreal in 2010, with the Award of Excellence. His research interests include software maintenance and evolution, cloud engineering, service-centric software engineering, empirical software engineering, and software analytic. He has published several papers in international conferences and journals, including ICSM(E), ASE, ISSRE, SANER, ICWS, HPCC, IPCCC, JSS, ASEJ, JSEP, EMSE, and TSE. His work has received three ten-year Most Influential Paper (MIP) Award, and four Best/Distinguished paper Awards. He has served on the program committees of several international conferences including ICSM(E), SANER, MSR, ICPC, SCAM, ESEM and has reviewed for top international journals such as SQJ, JSS, EMSE, TSE and TOSEM. He is program chair for Satellite Events at SANER 2015, program co-chair of SCAM 2015, ICSME 2018, and ICPC 2019, and general chair of ICPC 2018. He is one of the organizers of the RELENG workshop series (http://releng.polymtl.ca) and Associate Editor for IEEE Software.

Giuliano Antoniol received his Laurea degree in electronic engineering from the Universita' di Padova, Italy, in 1982. In 2004, he received his Ph.D. in Electrical Engineering at Polytechnique Montreal. He worked in companies, research institutions, and universities. In 2005, he was awarded the Canada Research Chair Tier I in Software Change and Evolution. He has participated in the program and organization committees of numerous IEEEsponsored international conferences. He served as program chair, industrial chair, tutorial, and general chair of international conferences and workshops. He is a member of the editorial boards of four journals: the Journal of Software Testing Verification & Reliability, the Journal of Empirical Software Engineering and the Software Quality Journal and the Journal of Software Maintenance and Evolution: Research and Practice. Dr Giuliano Antoniol served as Deputy Chair of the Steering Committee for the IEEE International Conference on Software Maintenance. He contributed to the program committees of more than 30 IEEE and ACM conferences and workshops, and he acts as referee for all major software engineering journals. He is currently Full Professor at Polytechnique Montreal, where he works in the area of software evolution, software traceability, search-based software engineering, software testing, and software maintenance.

Affiliations

David Johannes[1] · Foutse Khomh[1] ⓘ · Giuliano Antoniol[2]

David Johannes
david.johannes@polymtl.ca

Giuliano Antoniol
giuliano.antoniol@polymtl.ca

[1]   SWAT Laboratory, Polytechnique Montréal, Montréal, QC, Canada

[2]   SOCCER Laboratory, Polytechnique Montréal, Montréal, QC, Canada