Titre: Title:	On the Fault-Proneness of Javascript Code Smells
Auteur: Author:	Amir Saboury
Date:	2016
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Saboury, A. (2016). On the Fault-Proneness of Javascript Code Smells [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2445/

UNIVERSITÉ DE MONTRÉAL

ON THE FAULT-PRONENESS OF JAVASCRIPT CODE SMELLS

AMIR SABOURY
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2016

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ON THE FAULT-PRONENESS OF JAVASCRIPT CODE SMELLS

présenté par : SABOURY Amir
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. MERLO Ettore, Ph. D., président
M. KHOMH Foutse, Ph. D., membre et directeur de recherche
M. ANTONIOL Giuliano, Ph. D., membre et codirecteur de recherche
M. GUÉHÉNEUC Yann-Gaël, Doctorat, membre

DEDICATION

To my family and friends...

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my supervisors Dr. Foutse Khomh and Dr. Giuliano Antoniol for their helps, support, positive attitude, and all the valuable feedbacks they have provided me during my studies.

I would also like to thank Dr. Yann-Gaël Guéhéneuc and Dr. Bram Adams for all the knowledge they shared, and their kind support.

I would like to thank the members of the jury for evaluating my thesis.

And I would like to thank all my colleagues specially Armstrong who helped me with the French translation, my family, and my friends specially Afshin and Saba, for all of their positive thoughts and for believing in me.

RÉSUMÉ

JavaScript est un langage de script qui a gagné beaucoup en popularité cette dernière décennie. Initialement utilisé exclusivement pour le développement Web côté client, il a évolué pour devenir l'un des langages de programmation les plus populaires. Les développeurs l'utilisent aujourd'hui aussi bien pour le développement Web côté client que côté serveur. Comme pour les applications écrites dans d'autres langages de programmation, les applications JavaScript peuvent contenir des mauvaises odeurs de code, qui sont des mauvais choix de conception ou d'implémentation pouvant affecter négativement la maintenabilité et la qualité des applications.

Dans ce mémoire, nous étudions les mauvaises odeurs de code dans les applications serveur JavaScript, dans le but de comprendre l'impact des mauvaises odeurs de code sur la fiabilité des applications JavaScript. Grâce à des modèles d'analyse de survie, nous examinons le risque d'occurrence de fautes dans les fichiers contenant des mauvaises odeurs de code et les fichiers ne contenant pas de mauvaise odeur de code. Au total, nous avons analysé 12 types de mauvaises odeurs de code contenues dans 537 versions de cinq bibliothèques JavaScript parmi les plus populaires, c'est-à-dire : express, grunt, bower, less.js et request. Les résultats obtenus montrent qu'en moyenne, le risque d'occurrence de faute dans les fichiers sans mauvaise odeur de code est 65% inférieur à celui des fichiers contenants des mauvaises odeurs de code. Parmi les mauvaises odeurs étudiées "Variable Reassign" et "Assignment in conditional statements" sont celles qui présentent le plus grand risque d'occurrence de faute.

Afin de comprendre la perception des développeurs vis-à-vis des 12 types de mauvaises odeurs de code étudiés, nous avons effectué un sondage auprès de 1484 développeurs JavaScript. Les résultats montrent que les développeurs considèrent les mauvaises odeurs de code "Nested Callbacks," "Variable Re-assign" et "Long Parameter List" comme étant de sérieux problèmes de conception qui entravent la maintenabilité et la fiabilité des applications JavaScript. Une évaluation qui corrobore les résultats de notre analyse quantitative.

Globalement, les mauvaises odeurs de code augmentent le risque d'occurrence de fautes dans les applications JavaScript. Nous recommandons aux développeurs de les corriger tôt avant la mise en marché de leurs applications.

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 cost of maintainability and the quality of a software.

In this thesis, we investigate code smells in JavaScript server-side applications with the aim to understand how they affect the software reliability.

We detect 12 code smells in 537 releases of five in demand JavaScript libraries (*i.e.,* express, grunt, bower, less.js, and request) and perform survival analysis, comparing the time until a fault occurrence, in files containing code smells and files without code smells. Results show that (1) on average, files without code smells have hazard rates 65% lower than files with code smells. (2) Among the studied smells, "Variable Re-assign" and "Assignment In Conditional statements" code smells have the highest hazard rates. Additionally, we conduct a survey among 1,484 JavaScript developers, to understand the perception of developers towards our studied code smells. We found that developers consider "Nested Callbacks," "Variable Re-assign" and "Long Parameter List" code smells to be serious design problems that hinder the maintainability and reliability of applications ; assessment in line with the findings of our quantitative analysis.

Overall, code smells affect negatively the fault-proneness of JavaScript applications. Therefore, developers should consider tracking and removing them early on before the release of software to the public.

CO-AUTHORSHIP

Earlier study in the thesis was submitted as follows :

— An Empirical Study of Code Smells In JavaScript Projects
Amir Saboury, Pooya Musavi, Foutse Khomh and Giulio Antoniol, submitted to the *24th IEEE International Conference on Software Analysis, Evolution, and Reengineering.*

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

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

A Couple of years ago, JavaScript might not have been the first choice of programming language. But now, you wouldn't be considered a serious developer if you didn't know how to program in JavaScript.
JavaScript is a highly dynamic scripting programming language that is becoming one of the most important programming languages in the world. Recent surveys by Stackoverflow [1] show JavaScript toping 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 [2]. 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 [3]. A trait that makes 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 [4], *i.e.,* poor solutions to recurring design or implementation problems.

Code smells are conjectured in the literature to decrease the quality of systems [4]. Yet, despite the many studies on code smells, few studies have empirically investigated the impact of code smells on the quality of JavaScript applications.

1.1 Concepts and Definitions

In this section, brief summaries of essential concepts related to the research topic are presented :

1.1.1 JavaScript

JavaScript invented by Brendan Eich in 1995, an employee of Netscape, hired to create a scripting language in order to make web pages interactive. It was originally named "Mocha," then renamed to "LiveScript" and then changed to "JavaScript." JavaScript was taken to

European Computer Manufacturers Association (ECMA) in 1996 to get standard specification so other browsers could follow and implement their own runtime environments. Over the years multiple versions of *ECMAScript* have released. *ActionScript 3* is also one of the implementations of ECMAScript. The latest version of ECMAScript (in short *ES*) is the version 7 (*ES7*) which finalized on June 2016.

1.1.2 Node.js

Although the origins of JavaScript goes back to browsers and client-side of a web application, it is not limited only to the browsers. Back in 1996, Netscape introduced an application server named "Enterprise Server 2.0" which enabled support for server-side execution of JavaScript application as CGI applications. Particularly in 2009, Ryan Dahl introduced Node.js, an open-source, JavaScript run time environments.

Node.js (also called Node) is an open source platform to run JavaScript on the server side. The core is based on Google's JavaScript runtime engine called V8. Both Node and V8 are mostly written in C and C++. Although V8 mainly supports running client-side JavaScript codes, Node uses its power to support long-running, real-time, scalable and memory efficient server-side network applications [5]. Node is also famous for its event-driven nature and non-blocking I/O model, which makes it a perfect fit for data-intensive and network applications. On the first quarter of 2015, Joyent, IBM, Microsoft, PayPal, Fidelity, SAP and The Linux Foundation joined forces to bring neutral and open governance support to the Node community [6]. The growing popularity of JavaScript raises the demand for good design practices and style-guides.

Now, JavaScript is the most commonly used programming language on both front-end and back-end with more than 90% usage on the front-end and more than 54% on the back-end [1]. It also has the most active repositories and total pushes on Github [2].

JavaScript is growing fast, and many developers and companies are adopting JavaScript related technologies in production. Although the popularity of JavaScript in the industry is increasing rapidly, fewer scientists in academia studied the code quality and code smells on JavaScript and its community packages.

1.1.3 NPM

Nodejs comes with its own package manager called NPM (Node Package Manager). NPM enables developers to install community libraries and their dependencies easily through the command-line interface [5]. By the time of writing this thesis, there are more than 350,000

distinct modules registered on NPM. With the average growth rate of more than 400 modules per day, it is the most popular registry on the web [7]. This shows how the popularity of using community libraries are increasing. NPm has increased the usage of community libraries drastically, thanks to its simple and easy to use APIs.

Developers can add third-party modules as dependency of their application using simple commands. But, this raises the questions about the quality and security of those modules. In March 2016, a developer removed all of his modules from NPM. One of the modules called `left-pad`[1] had been used by thousands of other modules living in NPM. The removal of this small module with only 11 lines of code, broke thousands of other modules [8, 9]. This event was a strong illustration of the impact that community modules can have on JavaScript applications.

1.1.4 Code Smell

A good and maintainable object-oriented software design requires best practices and proper guides for the design [10, 11]. Violations of these rules and design principles are generally referred as code smells [12]. Code smell or bad smell was first introduced by Fowler and Beck as the pieces of the codes that scream out to be refactored [4].

1.2 Research Objectives

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 server-side applications. This thesis aims to fill this gap in the literature.

In this thesis we propose the following research questions :

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

Previous works [13, 14] 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 JavaScript files that contain code smells and files without code smells, computing their respective hazard rates. Results show that on average, across our five studied applications, JavaScript files without code smells have hazard rates 65% lower than JavaScript files with code smells.

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

1. https ://www.npmjs.com/package/left-pad

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, giving their limited resources. This research question examines faults in files affected by different types of code smells, with the aim to identify code smells that developers should refactor in priority. Our findings show that "Variable Re-assign" and "Assignment in Conditional Statements" code smells are consistently associated with high hazard rates across the five studied systems. Developers should consider removing these code smells, in priority since they make the code more prone to faults. We also conducted a survey with 1,484 JavaScript developers, to understand the perception of developers towards the 12 studied code smells. Results show that developers consider "Nested Callbacks", "Variable Re-assign" and "Long Parameter List" code smells to be the most hazardous code smells. Developers reported that these code smells negatively affect the maintainability and reliability of JavaScript applications.

1.3 Thesis Overview

The remainder of this thesis is organized as follows.

— Chapter 2 describes the related literature on code smell and JavaScript systems, and also the type of code smells we extracted from literature and used in our study.
— Chapter 3 introduces our subject systems and describes the design of our case study and analysis.
— Chapter 4 presents and discusses the results of our case study and the findings of our qualitative study. It also discusses the limitation of our analysis.
— Chapter 5 concludes the thesis.

CHAPTER 2 LITERATURE REVIEW

In this section, we discuss the related literature on code smell and JavaScript systems.

2.1 JavaScript Code Smells

There have been previous researches on JavaScript. Most of them focusing on the security of client side JavaScript applications.
Saxena et al. developed a system to explore the execution space of client side JavaScript applications to find security vulnerabilities [15]. On the other study, they proposed a dynamic analysis technique to discover validation vulnerabilities in client side JavaScript applications systematically [16]. Richards et al. studied the dynamic behavior of JavaScript by performing an empirical study on how and why the dynamic features of JavaScript are being used [17]. Bielova conducted a survey on JavaScript security policies for client side applications and proposed a detailed comparision of the runtime monitoring based security technique for JavaScript applications [18]. Tripp and Weisman presented a hybrid-analysis solution to automate the assessment of JavaScript client side application by combining white-box and black-box methodologies [19]. Hallaraker et al. also proposed an approach based on monitoring JavaScript code execution to detect malicious code behavior [20].

Code Smells [4] are poor design and implementation choices that are reported to negatively impact the quality of software systems. They are opposite to design patterns [21] 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., [3, 22]).
2. the evolution of code smells in software systems (e.g., [23, 24, 25, 26]) and their impact on software quality (e.g., [14, 26, 27, 28, 29]).
3. the relationship between code smells and software development activities (e.g., [29, 30]).

Our work in this thesis falls into the second category. We aim to understand how code smells affect the fault-proneness of JavaScript systems. Jaafar et al. [14] examined the fault-proneness of classes sharing a static or a co-change relationship with a class containing a code smell. They conduct an empirical study to analyze anti-patterns dependencies in more than 165000 commits of three Java open source software (ArgoUML, JFreeChart, and XerecesJ). They showed that, classes that have static relationship as well as the classes that have co-

change relationship with anti-patterns are more likely to be fault-prone than others. Li and Shatnawi [27] who 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. [28] 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. [26] 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. [30], 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. [29] found that code smells can have a negative impact on code understandability. Recently, Fard et al. [3] 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 [31] 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 [32], JSLint [33] and JSHint [34] are rule based static code analysis tools that can validate source codes against a set of best coding practices.

These previous studies raised the awareness of the community about the potential negative impact of code smells on the quality of object oriented systems and recommended that developers apply refactorings to remove code smells from their systems.

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 thesis aims to fill this gap.

2.2 Types Of Code Smells

To study the impact of code smells on fault-proneness of JavaScript server-side applications, we first need to identify a list of JavaScript bad practices as our set of code smells. The following 12 code smells are extracted from the literature and JavaScript style guides by big organizations[3, 35, 36, 37, 38].

2.2.1 Lengthy Lines

Too many characters in a single line of code would decrease readability and maintainability of the code. Lengthy lines of code also make the code review process harder. There are different limits indicated in different JavaScript style guides. NPM's coding style[35] and node style guide[36] suggest that 80 characters per line should be the limit. Airbnb's JavaScript style guide[37] which is a popular one with around 42,000 Github stars, suggests a number of characters per line of code less than 100. Wordpress's style guide[39] encourages jQuery's 100-character limit[38]. All the style guides include white spaces and indentations in the limit. As mentioned in jQuery's style guide, there are some cases that should be considered exceptions to this limit : (i) comments containing long URLs and (ii) regular expressions [38].

2.2.2 Chained Methods

Method chaining 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. This process can be repeated indefinitely, resulting in a "chain" of method calls. The nature of JavaScript and its dynamic behavior have made creating chaining code structures very easy. jQuery [1] is one of the many libraries utilizing this pattern to avoid overuse of temporary variables and repetition [40]. Chained methods allow developers to write less code. However, overusing chained methods makes the control flow complex and hard to understand [3]. Below is an example of chained methods from a jQuery snippet :

```
$('a').addClass('reg-link')
      .find('span')
      .addClass('inner')
      .end()
      .end()
      .find('div')
      .mouseenter(mouseEnterHandler)
      .mouseleave(mouseLeaveHandler)
      .end()
      .explode();
```

2.2.3 Long Parameter List

An ideal function should have no parameters [41]. Long lists of parameters make functions hard to understand [42]. It is also a sign that the function is doing too much. The alternatives

1. jquery.com

are to break functions into simpler and smaller functions that do more specific tasks or to create better data structures to encapsulate the data. To handle a large amount of configurations passing to functions, JavaScript developers tend to use a single argument containing all the configurations. This is a better practice since it eliminates the order of parameters when the function calls, and it is easier to add more parameters later on while maintaining the backward compatibility. Bellow are examples of this code smell and suggested refactorings.

```
// considered bad
function distance(x1, y1, x2, y2) {
  return Math.sqrt(Math.pow(x1-x2, 2) +
      Math.pow(y1-y2, 2));
}

// alternative
function distance(p1, p2) {
  return Math.sqrt(Math.pow(p1.x-p2.x, 2) +
      Math.pow(p1.y-p2.y, 2));
}
```

```
// considered bad
function send(from, to, subject, body) {
  // ...
}

// alternative
function send(options) {
  // using options.from, options.to
  //     options.subject, options.body
}
```

2.2.4 Nested Callbacks

JavaScript I/O operations are asynchronous and non-blocking [43]. Developers use callback functions to execute tasks that depend on the results of other asynchronous tasks. When multiple asynchronous tasks are invoked in sequence (*i.e.,* the result of a previous one is needed to execute the next one), nested callbacks are introduced in the code [44, 45]. This structures could lead to complex pieces of code which is called "callback hell" [3, 44, 46]. There are several alternatives to nesting callback functions like using Promises [44] or the newest ES7 features [47].

ES6 added a new feature to JavaScript, called *Generator Functions* which are essentially functions that return iterators. Generators enable more control over the execution of JavaS-

cript functions by introducing `yield` keyword. `yield` pauses the execution of the function and moves the control back to the caller function. Using third-party libraries which implemented *coroutines*, generators can be used to control the execution of asynchronous functions. Also, ES7 removes the need for the third-party coroutine libraries by implementing built-in flow-control mechanisms using `async` and `await` keywords.

Bellow is an example of Nested Callbacks smell and alternative implementations with Promises and generators.

```
// considered bad
db.getUser({id: 1}, function (err, user) {
  twitter.getTweets({handle: user.twitter}, function (err, tweets) {
    sendEmail(tweets, function (err, done) {
      console.log('Done')
    })
  })
})

// Alternative implementation using Promises
db.getUser({id: 1})
  .then(function (user) {
    return twitter.getTweets({handle: user.twitter});
  })
  .then(function (tweets) {
    return sendEmail(tweets);
  })
  .then(function() {
    console.log('Done')
  })

// Alternative implementation using Generators
co(function*() {
  let user = yield db.getUser({id: 1});
  let tweets = yield twitter.getTweets({handle: user.twitter});
  yield sendEmail(tweets);
  console.log('Done')
})

// Alternative implementation using async/await
async function f() {
  let user = await db.getUser({id: 1});
  let tweets = await twitter.getTweets({handle: user.twitter});
  await sendEmail(tweets);
  console.log('Done')
```

```
}
```

2.2.5 Variable Re-assign

JavaScript is dynamic and weakly-typed language. Hence, it allows changing the types of the variables at run-time, based on the assigned values. This allows developers to reuse variables in the same scope for different purposes. A mechanism that can decrease the quality and the readability of the code. It is recommended that developers use unique names, based on the purpose of the variables [3]. Bellow is an example of Variable Re-assign code smell and a suggested refactoring.

```
// considered bad
function parse(url) {
  url = url.split('/'); // bad practice
  var page_id = url.pop();
  var category = url.pop();
  url = url[0]; // bad practice
  return {
    id: page_id,
    category: category,
    url: url
  };
}
parse('example.com/article/12');

// using unique names
function parse(url) {
  const url_parts = url.split('/');
  const page_id = url_parts.pop();
  const category = url_parts.pop();
  const domain = url_parts[0];
  return {
    id: page_id,
    category: category,
    url: domain
  };
}
parse('example.com/article/12');
```

2.2.6 Assignment in Conditional Statements [2]

JavaScript has three kinds of operators that use the = character.

— "=" For assignment.

```
const pi = 3.14;
```

— "==" For comparing values.

```
if (username == "admin") {}
```

— "===" For comparing both values and types.

```
if (input === 5) {}
```

The operator == compares only values and allows different variable types to be equal if their value is the same. While the operator === compares both the types and the values of variables and evaluates to false if operands' types are different even if their values are equal.

```
'5' == 5  // true
'5' === 5 // false
```

The operator = not only assigns a value to a variable but also returns the value. This allows multiple assignments in a single statement :

```
var a, b, c;
a = b = c = 5;
```

Which translates into :

```
var a, b, c;
(a = (b = (c = 5)));
```

The = operator also could be used in conditions :

```
function getElement(arr, i) {
  if (i < arr.length) return arr[i];
  return false;
}
var element;
if (element = getElement(arr, 5)){
  console.log(element);
}
```

Sometimes developers use assignments in conditional statements to write less code. It could also happen by mistyping = instead of ==. IDEs [3] often flag the usage of assignment in

2. http ://eslint.org/docs/rules/no-cond-assign
3. Integrated Development Environment

conditions with a warning sign. Compilers like `g++` will warn about these patterns if `-Wall` switch is passed to it. It is a common pattern for iterating over an array or any other iterable object and extracting values from them. Iterating over the result of executing a regular expression on a string. Bellow is an example of Assignment in Conditions code smell and a suggested refactoring.

```
var str = 'this is a string';
var rx  = /\w+/g;
var word;
while(word = rx.exec(str)){
    console.log(word[0]); // matched word
    console.log(word.index); // matched index
}

// better approach
var str = 'this is a string';
var rx  = /\w+/g;
var word;
while(true){
  word = rx.exec(str);
  if (!word) break;
    console.log(word[0]); // matched word
    console.log(word.index); // matched index
}
```

While assignment in conditions could be intentional, it is often the result of a mistake, *i.e.,* = is used instead of == [48].

2.2.7 Complex code

The cyclomatic complexity of a code is a the number of linearly independent paths through the code [49]. JavaScript files with the Complex code smell are characterized by high cyclomatic complexity values.

2.2.8 Extra Bind [4]

The "`this`" keyword in JavaScript functions is contextual and is initialized with the context which the function is being called within.

```
var obj = {
  a: 5,
```

4. http ://eslint.org/docs/rules/no-extra-bind

```
  f: function () {
    return this.a;
  }
}
obj.f(); // 'this' in f is 'obj'
```

This design of JavaScript leads to `this` to be bound to a global scope whenever the function is called as a callback if not bound explicitly. So the scope of variable `this` is not lexical. In other words `this` in inner functions is not going to be bound to the `this` of the outer function [3]. Using "`.bind(ctx)`" on a function will change the context of the function and should be used with caution.

The example below shows the usage of `.bind(ctx)` to explicitly bind the context of the callback function to the context of its outer function.

```
function downloader(id) {
  this.path = '/' + id;
  this.result = null;
  function callback(data) {
    this.result = data;
    console.log('done', this.path);
  }
  download(this.path, callback.bind(this)); // note the usage of 'this'
}
```

Sometimes the `this` variable is removed from the body of the inner function in the course of maintenance or refactoring. Keeping `.bind()` in these cases is an unnecessary overhead. In ES6, there is another type of functions called *arrow functions* which solved the problem mentioned above. In *arrow functions* the scoping of `this` is lexical.

The example below shows how *arrow functions* could be used to have lexical `this` inside functions.

```
function downloader(id) {
  this.path = '/' + id;
  this.result = null;
  download(this.path, (data) => {
    this.result = data;
    console.log('done', this.path);
  });
}
```

2.2.9 This Assign [5]

If the context in a callback function is not bound at the definition level, it will be lost. When there are large numbers of inner functions or callbacks in which the context should be preserved, developers often use a hacky solution such as storing `this` in another variable to access to the parent scope's context. If the context of the parent scope is stored in another variable besides `this`, usually named `self` or `that` [50], it wouldn't be overridden and it is going to be bound to the same variable for all the defined functions in the same scope tree.

The example below is an example of storing `this` in another variable to be used in callback functions.

```
function User(id) {
  var self = this;
  self.id = id;
  getPropertiesById(id, function(props) {
    // self is bound to its value on parent scope
    // since there is no self in the current scope
    self.props = props;
  });
}
```

Assigning `this` to other variables could work for small classes, but it decreases the maintainability of code as the size of the project grows. Having a substitute variable for `this` could also break if the substitute variable is overridden by a callback function. It is a bad practice to use this hacky solution since there are other built-in language features to have lexical `this`.

The code below shows how to use built-in language features to achieve to lexical `this` in callback functions.

```
function User(id) {
  this.id = id;
  getPropertiesById(id, function(props) {
    this.props = props;
  }.bind(this)); // note the .bind
}

// ES6 feature:
function User(id) {
  this.id = id;
  // arrow functions use lexical ‘this‘
  getPropertiesById(id, props => {
    this.props = props;
```

5. https ://github.com/amir-s/eslint-plugin-smells

```
14   });
15 }
```

2.2.10 Long Methods

Long method is a well-known code smell [3, 42, 51]. Long methods should be broken down into several smaller methods that do more specific tasks.

2.2.11 Complex Switch Case

Complex switch-case structures are considered a bad practice and could be a sign of violation of the Open/Close principle [52]. Switch statements also induce code duplication. Often there are similar switch statements through the software code and if the developer needs to add/remove a case to one of them, it has to go through all the statements, modifying them as well [3, 53, 54].

2.2.12 Depth [6]

The depth or the level of indentation is the number of nested blocks of code. Higher depth means more nested blocks and more complexity. The following statements are considered as an increment to the number of blocks if nested : `function`, `If`, `Switch`, `Try`, `Do While`, `While`, `With`, `For`, `For in` and `For of`.

These two functions have the same functionality. But the depth of the second implementation is less than the first one.

```
1 // max depth = 4
2 function get(array, cb) {
3     var result = [];
4     for (var i=0;i<array.length;i++) {
5         download(array[i], function (data) {
6             result.push(data);
7             if (result.length == array.length) {
8                 cb(result);
9             }
10        })
11    }
12 }
13
14 // max depth = 2
```

6. http ://eslint.org/docs/rules/max-depth

```
function get(array, cb) {
    var result = [];
    function inner_cb(data) {
        result.push(data);
        if (result.length != array.length) return;
        cb(result);
    }
    for (var i=0;i<array.length;i++) {
        download(array[i], inner_cb)
    }
}
```

CHAPTER 3 METHODOLOGY AND DESIGN

The *goal* this study is to investigate the relation between the occurrence of code smells in JavaScript files and files fault-proneness. The *quality focus* is the source code fault-proneness, which, if high, can have a concrete effect 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 team, 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 code smells identified in five 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 smell ? Prior works show that code smells increase the fault-proneness of Java classes [13, 14]. 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 projects.

(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 projects more prone to faults.

3.1 Studied Systems

In order to address our research questions, we perform a case study with the following five open source JavaScript projects. Table 3.1 summarizes the characteristics of our subject systems.

3.1.1 Express

Express[1] is a minimalist web framework for Nodejs. It is one of the most popular libraries in NPM [55] and it is used in production by IBM, Uber and many other companies[2]. Its Github repository has over 5,200 commits and more than 190 contributors. It has been forked 5,000 times and starred more than 28,000 times. Express is also one of the most dependent upon libraries on NPM with over 8,800 dependents. There are more than 2,300 closed Github issues on their repository.

3.1.2 Bower.io

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 [56]. Its Github repository has more than 2,600 commits from more than 200 contributors. Bower has been starred over 14,500 times on Github and has over 1,500 closed issues.

3.1.3 LessJs

Less.js[5] is a CSS[6] pre-processor. It extends CSS and adds dynamic functionalities to it. There are more than 2,600 commits by over 200 contributors on its Github repository. LessJs's repository has more than 2,000 closed issues and it is starred more than 14,000 times and forked over 3,200 times.

3.1.4 Request

Request[7] is a fully-featured library to make HTTP calls. More than 8,300 other libraries are direct dependents of Request. Over 2,000 commits by more than 260 contributors have been made into its Github repository and 12,000+ users starred it. There are more than 1,100 closed issues on its Github repository.

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

3.1.5 Grunt

Grunt[8] is one of the most popular JavaScript task runners. More than 1,600 other libraries on NPM are direct dependents of Grunt. Grunt is being used by many companies such as Adobe, Mozilla, Walmart and Microsoft [57]. The Github repository of Grunt is starred by more than 11,000 users. More than 60 contributors made over 1,300 commits into this project. They also managed to have more than 1,000 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.

Table 3.1 Descriptive statistics of the studied systems.

Module	Domain	# Commits	# Contributors	# Github stars	# Releases	Project start date
Express	Web framework	5200+	+190	+28000	260	Jun 21, 2009
Request	HTTP client utility	2000+	+200	+12000	118	May 2, 2010
Less.js	CSS pre-processor	2600+	+200	+14000	48	Feb 21, 2010
Bower.io	Package manager	2600+	+200	+14500	100	Sep 2, 2012
Grunt	Task Runner	1300+	+60	+11000	11	Sep 18, 2011

3.2 Data Extraction

To answer our research questions, we need to mine the repositories of our five selected systems to extract information about the *smelliness* of each file at commit level, identifying whether the file contains a code smell or not. 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. Figure 3.1 provides an overview of our approach. We describe each step in our data extraction approach below. We have implemented all the steps of our approach into a framework available on Github[9].

3.2.1 Snapshot Generation

Since all the five 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.

8. https ://github.com/gruntjs/grunt
9. https ://github.com/amir-s/smelljs

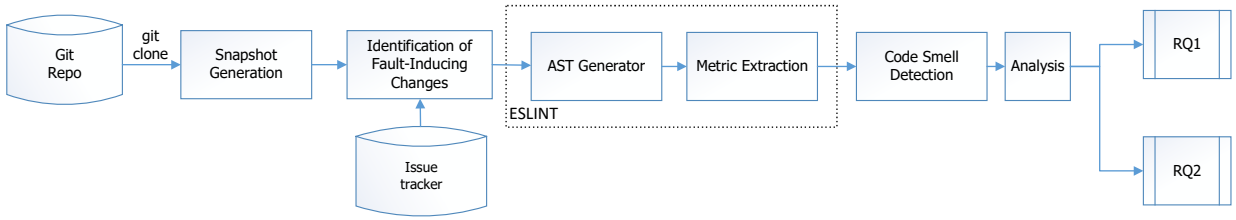


Figure 3.1 Overview of our approach to answer our research questions.

3.2.2 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 issues on the systems. We leverage the SZZ algorithm [58] to detect changes that introduced faults. We first identify fault-fixing commits using the heuristic proposed by Fischer et al. [59], which consists in using regular expressions to detect bug IDs from the studied commit messages. 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'. 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. Finally, we filter the commits that were submitted after their corresponding bugs' creation date.

3.2.3 AST Generation and Metric Extraction

To automatically detect code smells in the source code, we first extract the Abstract Syntax Tree from the code. Abstract Syntax Trees (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 [60, 61, 62]. We used ESLint [10] 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

10. http ://eslint.org/

companies like Facebook, Paypal, Airbnb, etc. ESLint parses JavaScript source codes and extracts Abstract Source Trees based on the specs [11]. ESLint itself provides an extensible environment for developers to develop their own plugins to extract custom information and transform the source code. We developed our own plugins and 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 2.2. Table 3.2 summarizes all the metrics our framework reports for each type of code smell.

Table 3.2 Metrics computed for each type of code smell.

Smell Type	Type	Metric
Lengthy Lines	Number	The number of characters per line.
Chained Methods	Number	The number of parameters of each function in source code.
Long Parameter List	Number	The number of parameters of each function in source code.
Nested Callbacks	Number	The number of nested functions present in 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.
Extra Bind	Boolean	If a function is not using explicitly bounded context.
This Assign	Boolean	If `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.
Depth	Number	The maximum number of nested blocks in each function.

3.2.4 Code Smell Detection

Among of 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 [63, 64], defining threshold values above which files should be considered as having the code smell. We define the thresholds relative to the systems using Box-plot analysis. 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 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.

11. https ://github.com/estree/estree

3.2.5 Analysis

At the end, we feed the data to our *R* scripts to generate the result. We wrapped the R scripts into our framework to automate the whole process. In the following section we will discuss about our analysis approach.

3.3 Survival Analysis

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 and files without code smells.

Survival analysis are used to model the time until the occurrence of a well-defined event [65]. 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 [66] [67]. Particularly, Cox models aim to model how long subjects under observation can *survive* before the occurrence of an event of interest (a fault occurrence in our case) [67] [68].

Survival models were first introduced in demography and actuarial sciences [69]. Recently, researchers have started applying them to problems in the domain of Software Engineering. For example, Selim et al. [68] used the Cox model to investigate characteristics of cloned code that are related to the occurrence of faults. Koru et al. [70] 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 :

$$\lambda_i(t) = \lambda_0 * e^{\beta * F_i(t)} \tag{3.1}$$

If we take log from both sides, we obtain :

$$log(\lambda_i(t)) = log(\lambda_0(t)) + \beta_l * f_{il}(t) + ... + \beta_n * f_{in}(t) \tag{3.2}$$

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 (*i.e.,* a fault). 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 Equation 3.3.

$$\lambda_i(t)/\lambda_j(t) = e^{\beta * (f_i(t) - f_j(t))} \tag{3.3}$$

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

3.3.1 Link function

. As Equation 3.2 shows, the log of the hazard is a linear function of the log of the baseline 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 [71]. 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 [71].

3.3.2 Stratification

. In addition to applying a link function, a stratification is sometimes necessary to preserve the proportionality in Cox hazard models [66]. 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 [66].

3.3.3 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 [71] [68].

In this thesis, we perform our analysis at commit level. For each file, we use Cox proportional hazard models to calculate the risk of a fault occurrence over time, considering a number of independent covariates. We chose 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 [71], which is important because software modules evolve over time and a file can have multiple faults during its life cycle.

CHAPTER 4 STUDY RESULT

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

4.1 Fault-proneness of Smelly Codes vs. non-Smelly Codes

In this section, we discuss the approach and the result for our first research question : **Is the risk of fault higher in files with code smells in comparison with those without smell ?**

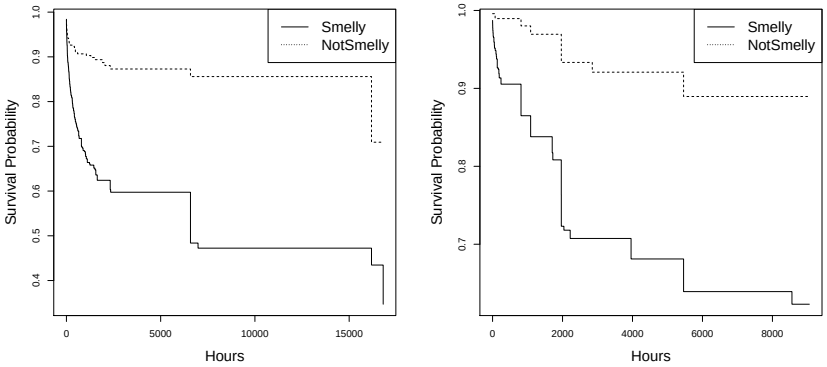
4.1.1 Approach

We use our framework described in Section 3.2 to collect information about the occurrence of the 12 studied code smells in our five subject systems. 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 doesn't contain any of the 12 studied code smells.
- **Event :** this metric takes the value 1 if the revision r is a fault-fixing change and 0 otherwise. We use the SZZ algorithm to insure that the file contained a code smell when the fault was introduced.

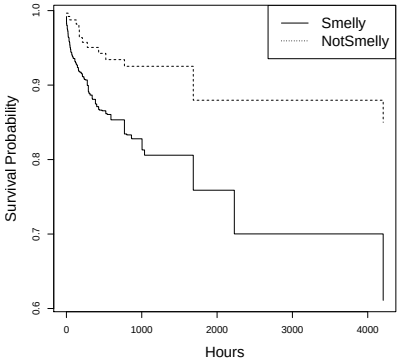
Using the smelly metric, we divide our dataset in two groups : one group containing files with code smells (*i.e.,* smelly = 1) and another group containing files without any of the 12 studied code smells (*i.e.,* smelly = 0). For each group we create an individual Cox hazard model. In each group, the 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 the R [72] to analyze our Cox hazard models.

In addition to building Cox hazard models, we test the following null hypotheses : 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.

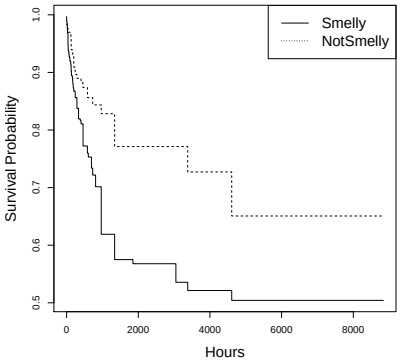


(a) express.js

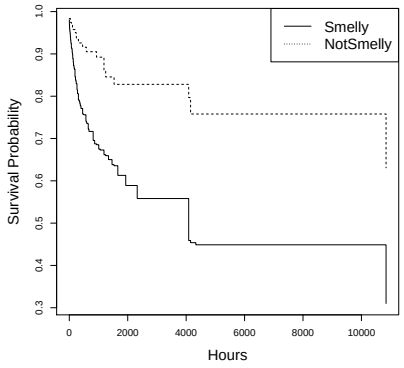
(b) request.js



(c) less.js



(d) grunt.js



(e) bower.js

Figure 4.1 Survival probability trends of smelly codes vs. non-smelly codes in our five JavaScript projects.

4.1.2 Findings

Results presented in Figure 4.1 show that files containing code smells experience faults faster than files without code smells. The Y-axis in Figure 4.1 represents the probability of a file *surviving* 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 five projects, we calculated relative hazard rates (using Equation 3.3 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 65% lower than files with 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 all the five 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 :

> *JavaScript files without code smells have hazard rates 65% lower than JavaScript files with code smells and this difference is statistically significant.*

4.2 The impact of each Code Smell type on fault-proneness

In this section, we discuss the approach and the result for our second research question : **Are JavaScript files with code smells equally fault-prone ?**

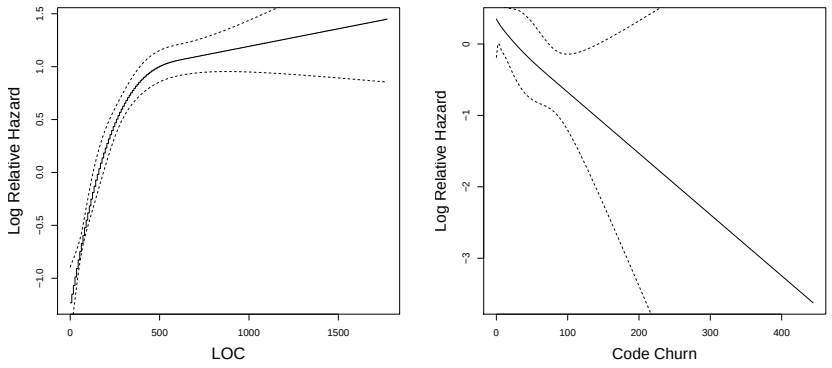
4.2.1 Approach

Similar to **RQ1**, we use our framework from Section 3.2 to collect information about the occurrence of the 12 studied code smells in our five subject systems. For each file and for each revision r (*i.e.,* corresponding to a commit), we also 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 doesn't contain any of the 12 studied code smells. When computing the **Event** metric, we used the 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

Table 4.1 Hazard ratios for each type of code smells. Higher exp(coef) values means higher hazard rates.

module	covariate	exp(coef)	*p*-value (Cox hazard model)	*p*-value (Proportional hazards assumption)
express	No.Previous-Bugs	1.013	0.05e-3	0.870
	Chained Methods	7.931	0.003	0.961
	This Assign	2.584	0.038e-8	0.716
	Variable Re-assign	1.488	0.007	0.253
grunt	Nested Callbacks	3.534	0.002	0.204
	Variable Re-assign	1.514	0.039	0.913
	Assign. in Cond. State.	2.212	0.001	0.829
bower	No.Previous-Bugs	1.019	0.019	0.451
	Depth	7.786	0.065e-4	0.910
	LOC	1.008e-1	0.029	0.241
less	No.Previous-Bugs	1.036	0.02e-14	0.741
	Complex Switch Case	0.481	0.027	0.417
	Assign. in Cond. State.	1.646	0.019e-2	0.940
request	No.Previous-Bugs	1.067	0.002	0.407
	Depth	0.172	0.052e-3	0.620
	Variable Re-assign	3.277	0.088e-2	0.733

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.



(a) express.js (b) request.js

Figure 4.2 Determining a link function for express.js and grunt.js modules for two covariates : LOC and Code Churn respectively.

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 five 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 [66] [68] and plot the log relative risk vs. each type code smell, the No. of Previous-Bugs, LOC and Code Churn in each of our five datasets (corresponding to the five subject systems). For all types of code smells and No. of Previous-Bugs covariates, we observed that a linear relationship exists. Since the plots for LOC and Code Churn covariates were similar to each other, for of all the five systems, and because of space limitation, in this thsis, we present only the plot of LOC (obtained on *express.js*) and Code Churn (obtained on *grunt.js*) covariates (see Figure 4.2). Figure 4.2 shows that for LOC, we do not have a linear relationship, hence we visually identified a suitable function (*i.e.,* a logarithmic function) to establish a linear relationship. In the case of Code Churn, we identified that a negative linear function should be applied. 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 of the system, we performed a non-proportional test to verify if the proportional hazards assumption holds.

4.2.2 Findings

Table 4.1 summarizes the hazard ratios for the 12 studied code smells. 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 Table 4.1 shows that the p-values obtained for the non-proportionality tests are above 0.05 for all the five systems ; meaning that the proportional hazards assumption is satisfied for all the five studied systems.

Overall, the hazard ratios of the studied code smells vary across the systems, with *Chained Methods*, *This Assign*, and *Variable Re-assign* having the highest hazard ratios in *express* ; *Nested Callbacks*, *Assignment in Conditional Statements*, and *Variable Re-assign* having the highest hazard rates in *grunt*, and *Depth* being the most hazard code smell in *bower*. *Assignment in Conditional Statements* has the highest hazard ratio in *less* and *Variable Re-assign* has the highest hazard ratio in *request*.

As we expected, the covariates No.Previous-Bugs is significantly related to fault occurrence, however, its hazard rate is lower than those of many of the studied code smells. LOC is significantly related to fault occurrence in only one system (*i.e., bower*), meaning that JavaScript developers cannot simply control for size and monitor files with the previous fault occurrences, if they want to improve the reliability of their system effectively. Since *Variable Re-assign* and *Assignment in Conditional Statements* are related to high hazard ratios in three out of five systems (60%), we strongly recommend that developers prioritize files containing these

two type 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* Variable Re-assign *code smell or* Assignment in Conditional Statements *code smell in priority since they seem to increase the risk of faults in the system.*

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.

4.3 Discussion

To understand the perception of developers towards our studied code smells, we conducted a qualitative study with JavaScript developers. In total 1,484 developers took part in our qualitative study. The survey consisted of 3 questions about the participant background and 15 questions about the studied code smells. We designed a website [1] to run the survey. The study took place between October 4th and October 17th, 2016. The link to the survey was shared within the *Hacker News community* [2] and the *EchoJS community* [3]. Participants were free to skip any question and they could leave the survey at any time. 68% of the participants to our survey had more than 3 years of experience writing JavaScript applications. We asked the participants about their usages of JavaScript and found that 92% of them use JavaScript to write client-side applications and 51% use it for server side applications. Over 63% of participants were familiar with the concept of *code smell* and 19% never heard of it.

The results of our survey showed that 20% of participants use pure callbacks to handle asynchronous logic, while 66% use *Promises* and 13% use the newest ES6 and ES7 features to control the flow of asynchronous codes. 92% of participants indicated that nesting the callbacks makes the code harder to maintain.

86% of our participants reported that they prefer codes using `const` instead of `var` to declare variables and not re-using them in the same scope. 73% indicated that re-using variables makes the code harder to maintain.

Surprisingly, 74% of our participants said they preferred having assignments in conditional statements while using *Regular Expressions*, however, 54% of them acknowledged that this practice makes the code harder to maintain.

55% of our participants reported that they prefer using `.bind(this)` instead of assigning `this` to other variables. However, only 16% of the participants indicated that they use `.bind`. 55% of the participants indicated that they use *arrow functions* to have lexical `this`.

Although the JavaScript documentation lists *Complex Switch Case* as a code smell, only 14% of our participants preferred `if/else` structures over `switch/case`.

We also divided participants into two groups based on the years of experience they had developing javascript applications. The first group had less than two years of experience and the second group, more than two years of experience. Comparing the result shows the answers of two groups follow the same pattern in most of the cases. However, 75% of experienced deve-

1. https ://srvy.online/js
2. https ://news.ycombinator.com/
3. http ://www.echojs.com/

lopers considered “Variable Re-assign” a hazardous code smell, while 63% of less experienced developers had the same opinion. The difference between the two groups in other answers was less than 10%.

On another analysis, we divided the participants into two groups based on how they use JavaScript. We considered front-end developers as the first group and back-end developers as the second group. The answers of the two groups followed the same pattern in most of the cases as well. However, 26% of front-end developers still use callbacks while only 14% of back-end developers use them, and only 8% of front-end developers use newest ES6 and ES7 features to write asynchronous codes. On the other hand, 20% of server-side developers utilize these new features to write asynchronous codes. 51% of server-side developers were strictly against re-using variables, while only 39% of front-end developers agreed that “variable re-assign” is a dangerous code smell.

In the survey, we asked participants to rank the 12 studied code smells on a Likert scale from 1 to 10, based on their impact on the software understandability, debugging and maintenance efforts. Results show that participants consider *Nested Callbacks* to be the most hazardous code smells (with a rating of 8.1/10), followed by *Variable Re-assign* (with a rating of 6.5/10) and *Long Parameter List* (with a rating of 6.2/10). They claimed that these code smells negatively affect the maintainability and reliability of JavaScript systems ; Assessment in line with the findings of our quantitative analysis.

4.4 Threats to validity

In this section, we discuss the threats to validity of our study following common guidelines for empirical studies [73].

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.,* "bug",“fix"’,“defect" and “patch") 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. Nevertheless, this heuristic was successfully used in multiple previous studies in software engineering [14, 74]. 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. When analyzing the *smelliness* of files that experienced fault-inducing changes, we only tracked the presence of the smell in the file as a whole. Hence, the smell contained in the file may not have been involved in the changed lines that induced the fault.

Internal validity threats concern our selection of systems and tools. The metric extraction tool used in this study is based on the AST provided by ESLint. The results of the study are therefore dependent on the accuracy of the results provided by ESLint. However, we are rather assured that this tool functions properly as it is being used widely by big companies like facebook [4], Paypal [5], Airbnb [6] etc. 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 thesis, we have studied five large JavaScript projects. We have also limited our study to open-source projects. Still, these projects represent different domains and various project sizes. Table 3.1

4. http ://facebook.com/
5. http ://paypal.com/
6. http ://airbnb.com/

shows a summary of the studied systems, their domain and their size. More studies on more JavaScript systems should be done to further validate our results.

Threats to reliability validity concern the possibly of replicating our study. In this thesis, we provide all the details needed to replicate our study. All our five subject systems are publicly available for study. The data and scripts used in this study is also publicly available and can be downloaded from Github [7].

7. https ://github.com/amir-s/smelljs

CHAPTER 5 CONCLUSION

In this study, we examine the impact of code smells on the fault-proneness of JavaScript systems. We presents a quantitative study of five JavaScript systems that compare the time until a fault occurrence in JavaScript files that contain code smells and files without code smells. Results show that JavaScript files without code smells have hazard rates 65% 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" and “Assignment in Conditional Statements" code smells have the highest hazard rates. In addition, we conducted a survey with 1,484 JavaScript developers, to understand the perception of developers towards our studied code smells, and found that developers consider *Nested Callbacks*, *Variable Re-assign*, *Long Parameter List* to be the most hazardous code smells. 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*, *Nested Callbacks*, and *Long Parameter List* code smells for refactoring.

REFEFENCES

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

[2] Githut, "Discover languages in github," 2016, [Online; accessed 11-August-2016]. [Online]. Available : http://githut.info/

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

[4] M. Fowler, "Refactoring : Improving the design of existing code," in *11th European Conference. Jyväskylä, Finland*, 1997.

[5] S. Tilkov and S. Vinoski, "Node. js : Using javascript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, p. 80, 2010.

[6] Joyent, "Joyent moves to establish node.js foundation," 2016, [Online; accessed 11-August-2016]. [Online]. Available : https://www.joyent.com/about/press/joyent-moves-to-establish-nodejs-foundation

[7] E. DeBill, "Modules count," 2016, [Online; accessed 11-August-2016]. [Online]. Available : http://www.modulecounts.com

[8] C. Williams, "How one developer just broke node, babel and thousands of projects in 11 lines of javascript," 2016, [Online; accessed 30-August-2016]. [Online]. Available : http://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/

[9] A. Koçulu, "Ive just liberated my modules," 2016, [Online; accessed 30-August-2016]. [Online]. Available : https://medium.com/@azerbike/i-ve-just-liberated-my-modules-9045c06be67c#.w2hqddox0

[10] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of object-oriented programming*, vol. 1, no. 2, pp. 22–35, 1988.

[11] A. J. Riel, *Object-oriented design heuristics*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[12] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, "Building empirical support for automated code smell detection," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2010, p. 8.

[13] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012. [Online]. Available : http://dx.doi.org/10.1007/s10664-011-9171-y

[14] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, and F. Khomh, "Mining the relationship between anti-patterns dependencies and fault-proneness." in *WCRE*, 2013, pp. 351–360.

[15] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 513–528.

[16] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "Flax : Systematic discovery of client-side validation vulnerabilities in rich web applications." in *NDSS*, 2010.

[17] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 1–12.

[18] N. Bielova, "Survey on javascript security policies and their enforcement mechanisms in a web browser," *The Journal of Logic and Algebraic Programming*, vol. 82, no. 8, pp. 243–262, 2013.

[19] O. Tripp and O. Weisman, "Hybrid analysis for javascript security assessment," in *ESEC/FSE*, vol. 11, 2011.

[20] O. Hallaraker and G. Vigna, "Detecting malicious javascript code in mozilla," in *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*. IEEE, 2005, pp. 85–94.

[21] E. Gamma, R. Helm, R.Johnson, and J. Vlissides, *Design Patterns : Elements of Reusable Object Oriented Software*, 1995.

[22] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "Bdtex : A gqm-based bayesian approach for the detection of antipatterns," *J. Syst. Softw.*, vol. 84, no. 4, pp. 559–572, Apr. 2011.

[23] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *Quality of Information and Communications Technology (QUATIC), 2010 7th Int'l Conf. on the*. IEEE, 2010, pp. 106–115.

[24] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka, "The evolution and impact of code smells : A case study of two open source systems," in *3rd Int'l Symposium on Empirical Software Engineering and Measurement, ESEM 2009,*, 2009, pp. 390–400.

[25] R. Peters and A. Zaidman, "Evaluating the lifespan of code smells using software repository mining," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conf. on*. IEEE, 2012, pp. 411–416.

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

[27] R. Shatnawi and W. Li, "An investigation of bad smells in object-oriented design," in *Information Technology : New Generations, 2006. ITNG 2006. 3rd Int'l Conf. on.* IEEE, 2006, pp. 161–165.

[28] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[29] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conf. on*, March 2011, pp. 181–190.

[30] D. I. K. Sjoberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1144–1156, Aug. 2013.

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

[32] "Jslint : The javascript code quality tool. http ://www.jslint.com."

[33] "Eslint : The pluggable linting utility for javascript and jsx. http ://eslint.org/."

[34] "Jshint : A static code analysis tool for javascript. http ://jshint.com/."

[35] "npm-coding-style," 2016, [Online ; accessed 17-October-2016]. [Online]. Available : https://docs.npmjs.com/misc/coding-style

[36] "Node.js style guide," 2016, [Online ; accessed 17-October-2016]. [Online]. Available : https://github.com/felixge/node-style-guide

[37] "Airbnb javascript style guide," 2016, [Online ; accessed 17-October-2016]. [Online]. Available : https://github.com/airbnb/javascript

[38] "jquery javascript style guide," 2016, [Online ; accessed 17-October-2016]. [Online]. Available : https://contribute.jquery.org/style-guide/js/

[39] "Wordpress javascript coding standards," 2016, [Online ; accessed 17-October-2016]. [Online]. Available : https://make.wordpress.org/core/handbook/best-practices/coding-standards/javascript/

[40] J. Chaffer, *Learning JQuery 1.3 : Better Interaction and Web Development with Simple JavaScript Techniques.* Packt Publishing Ltd, 2009.

[41] R. C. Martin, *Clean code : a handbook of agile software craftsmanship.* Pearson Education, 2009.

[42] F. A. Fontana, P. Braione, and M. Zanoni, "Automatic detection of bad smells in code : An experimental assessment." *Journal of Object Technology*, vol. 11, no. 2, pp. 5–1, 2012.

[43] L. Griffin, K. Ryan, E. de Leastar, and D. Botvich, "Scaling instant messaging communication services : A comparison of blocking and non-blocking techniques," in *Computers and Communications (ISCC), 2011 IEEE Symposium on.* IEEE, 2011, pp. 550–557.

[44] E. Brodu, S. Frénot, and F. Oblé, "Toward automatic update from callbacks to promises," in *Proceedings of the 1st Workshop on All-Web Real-Time Systems.* ACM, 2015, p. 1.

[45] K. Gallaba, A. Mesbah, and I. Beschastnikh, "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, 2015, pp. 1–10.

[46] M. Ogden, "Callback hell," 2015, [Online ; accessed 14-August-2016]. [Online]. Available : http://callbackhell.com

[47] J. Archibald, "Es7 async functions," 2014, [Online ; accessed 14-August-2016]. [Online]. Available : https://jakearchibald.com/2014/es7-async-functions/

[48] "Sei cert c++ coding standard - exp19-cpp. do not perform assignments in conditional expressions," [Online ; accessed 23-August-2016]. [Online]. Available : https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=975

[49] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[50] D. Crockford, *JavaScript : The Good Parts : The Good Parts.* " O'Reilly Media, Inc.", 2008.

[51] R. Marinescu and M. Lanza, "Object-oriented metrics in practice," 2006.

[52] R. C. Martin, "The open-closed principle," *More C++ gems*, pp. 97–112, 1996.

[53] F. Martin, B. Kent, and B. John, "Refactoring : improving the design of existing code," *Refactoring : Improving the Design of Existing Code*, 1999.

[54] J. Kerievsky, *Refactoring to patterns.* Pearson Deutschland GmbH, 2005.

[55] A. Mardan, *Express. js Guide : The Comprehensive Book on Express. js.* Azat Mardan, 2014.

[56] "About bower," 2016, [Online; accessed 4-October-2016]. [Online]. Available : https://bower.io/docs/about/

[57] "Who uses grunt," 2016, [Online; accessed 4-October-2016]. [Online]. Available : http://gruntjs.com/who-uses-grunt

[58] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM sigsoft software engineering notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

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

[60] I. Neamtiu, J. S. Foster, and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[61] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Software Maintenance, 1998. Proceedings., International Conference on.* IEEE, 1998, pp. 368–377.

[62] F. Pfenning and C. Elliott, "Higher-order abstract syntax," in *ACM SIGPLAN Notices*, vol. 23, no. 7. ACM, 1988, pp. 199–208.

[63] R. Marinescu, "Detection strategies : Metrics-based rules for detecting design flaws," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on.* IEEE, 2004, pp. 350–359.

[64] D. Mazinanian and N. Tsantalis, "Migrating cascading style sheets to preprocessors by introducing mixins," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* ACM, 2016, pp. 672–683.

[65] J. Fox and S. Weisberg, *An R companion to applied regression.* Sage, 2010.

[66] A. G. Koru, K. El Emam, D. Zhang, H. Liu, and D. Mathew, "Theory of relative defect proneness," *Empirical Software Engineering*, vol. 13, no. 5, pp. 473–498, 2008.

[67] J. D. Singer and J. B. Willett, *Applied longitudinal data analysis : Modeling change and event occurrence.* Oxford university press, 2003.

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

[69] H. Westergaard, *Contributions to the History of Statistics.* P.S. King, London, 1932.

[70] A. G. Koru, D. Zhang, and H. Liu, "Modeling the effect of size on defect proneness for open-source software," in *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*. IEEE Computer Society, 2007, p. 10.

[71] T. M. Therneau and P. M. Grambsch, *Modeling survival data : extending the Cox model*. Springer Science & Business Media, 2000.

[72] T. Therneau, "R survival package," 2000.

[73] R. K. Yin, *Case Study Research : Design and Methods - Third Edition*, 3rd ed. SAGE Publications, 2002.

[74] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Studying re-opened bugs in open source software," *Empirical Software Engineering*, vol. 18, no. 5, pp. 1005–1042, 2013.