Pós-Graduação em Ciência da Computação

Luana Martins dos Santos

A Study of JavaScript Error Handling

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2019

Luana Martins dos Santos

A Study of JavaScript Error Handling

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: engenharia de software e linguagens de programação
Orientador: Fernando José Castor de Lima Filho

Recife
2019

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

S237s Santos, Luana Martins dos
A study of JavaScript error handling / Luana Martins dos Santos. – 2019.
72 f.: il., tab.

Orientador: Fernando José Castor de Lima Filho.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2019.
Inclui referências.

1. Engenharia de software. 2. Tratamento de erros. I. Lima Filho, Fernando José Castor de (orientador). II. Título.

005.1 CDD (23. ed.) UFPE- MEI 2019-058

Luana Martins dos Santos

"A Study of JavaScript Error Handling"

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 14/02/2019.

BANCA EXAMINADORA

Dedico este trabalho a minha família e aos meus amigos que me lembraram que eu preciso cuidar de mim, apesar de eu mesma esquecer disso.

ACKNOWLEDGEMENTS

I would like to thank and dedicate this work to the following people:

To my advisor Fernando Castor, for being such a great professor, advisor and patient at my endless questions and doubts.

To Everton Lacerda, Erico Teixeira, and Felipe Ebert for our discussions for this dissertation.

To my friends, specially Luiz and Jeff, for help me to not forget that challenges may build a person, but it is important to reserve a moment to drink coffee.

To my family, for supporting me through this academical life.

ABSTRACT

JavaScript is in widespread use for both Web-based and Standalone software development. A large number of production quality, robust software systems are currently built using it. Because of its popularity, JavaScript has been the subject of a several empirical studies in the last few years. Previous research has analyzed uses of the `eval` function, how callbacks are employed, and other aspects of the language. In this work, we contribute to the existing body of knowledge by studying how developers employ error handling mechanisms in JavaScript systems. JavaScript provides two different mechanisms for handling errors, `try-catch` blocks and callback functions. These mechanisms are employed along with a number of abstractions that have not been previously studied in the context of error handling, namely: promises, events, and `async`hronous functions. In addition, we evaluated the usage of global event handlers, which is applicable for scenarios where an error occurred and no handler was found. We analyzed 192 popular JavaScript repositories from Github, comprising more than 60 thousand files and 11 million lines of code. We also classified them as Web-based or Standalone, depending on Node.js framework usage. We analyzed how the error handling mechanisms of the language are employed, what error handling strategies are typically used, and how Web-based and Standalone systems differ regarding the error handling. Errors impact differently in Web-based and Standalone systems. Users may not concern about the errors in Web-based as it generally occurs in the console. Standalone systems deal differently with errors, once the system finds an error, it crashes and does not allow any further operation. Our findings indicate that pure callbacks are the predominant error handling mechanism in JavaScript systems (64.500 callback functions in our dataset), although `try-catch` blocks are also frequently used (51.200 `try-catch` blocks). We found 22.44% of the `try-catch` blocks are empty, 15.48% of the error handlers ignore the error parameter (from `catch` clause). In callback functions, 8.66% ignore any error parameter it receives, and 5.54% reassign an error parameter. Web-based systems have a greater number of try-catch blocks than callback functions for error handling compared to standalone systems. Web-based systems have a greater number of handlers that ignore error parameters than standalone systems. In summary, our analysis shows that error-handling strategies are generally simplistic, mostly ignoring the error (11.5%) or leaving it empty (8.22%).

Keywords: Error handling. Exception Handling. Empirical Study. JavaScript.

RESUMO

JavaScript é amplamente usado para desenvolvimento de software tanto em sistemas *Web-based* quanto *Standalone*. Existe uma grande quantidade de sistemas desenvolvidos nessa linguagem. Devido à sua popularidade, o JavaScript tem sido objeto de vários estudos empíricos nos últimos anos. Pesquisas anteriores analisaram o uso da função `eval`, assim como *callback functions* e outros aspectos da linguagem. Neste trabalho, estudamos como os desenvolvedores empregam mecanismos de tratamento de erros em sistemas JavaScript. JavaScript fornece dois mecanismos diferentes para tratar erros: blocos `try-catch` e *callback functions*. Esses mecanismos são empregados em conjunto com um número de abstrações que não foram estudadas anteriormente no contexto do tratamento de erros, a saber: *promises*, eventos e funções asíncronas. Além disso, avaliamos o uso de tratadores de eventos globais, que é aplicável a cenários nos quais ocorreu um erro para o qual nenhum tratador foi encontrado. Analisamos 192 repositórios populares de JavaScript do Github, com mais de 60 mil arquivos e 11 milhões de linhas de código. Também os classificamos como *Web-based* ou *Standalone*, dependendo do uso do framework Node.js. Analisamos como os mecanismos de tratamento de erros da linguagem são empregados, quais estratégias de tratamento de erros são normalmente usadas e como os sistemas *Web-based* e *Standalone* diferem em relação ao tratamento de erros. Erros impactam diferentemente em sistemas *Web-based* e *Standalone*. Os usuários não percebem imediatamente erros ocorridos em sistemas *Web-based*, por geralmente aparecerem no console. Sistemas *Standalone* lidam de maneira diferente com erros, pois uma vez que o sistema encontra um erro, ele falha, não permitindo qualquer operação adicional. Nossas descobertas indicam que *callbacks functions* são o mecanismo predominante de tratamento de erros de sistemas JavaScript (existem 64,500 *callbacks functions* em nosso conjunto de dados), embora blocos `try-catch` também sejam usados com frequência (51,200 blocos `try-catch`). Encontramos que 22.44% dos blocos `try-catch` estão vazios, 15.48% dos tratadores de erro ignoram o parâmetro de erro (`catch` *clause*). Em *callback functions*, 8.66% ignoram qualquer parâmetro de erro que a função recebe, e 5.54% reatribuem um parâmetro de erro (como o argumento de erro de uma `catch` *clause* ou o argumento de uma *callback function*) a algum outro valor. Os sistemas *Web-based* possuem um maior número de blocos `try-catch` do que *callback functions* para tratamento de erros, em comparação com os sistemas *Standalone*. Os sistemas *Web-based* apresentam um maior número de tratadores que ignoram os parâmetros de erro do que sistemas *Standalone*. Em resumo, nossa análise mostra que as estratégias de tratamento de erros geralmente são simplistas, envolvendo principalmente ignorar o erro (11.5%) ou deixá-lo vazio (8.22%).

Palavras-chaves: Tratamento de erros. Exceções. Estudo Empírico. JavaScript.

LIST OF ABBREVIATIONS AND ACRONYMS

AST	Abstract Syntax Tree
JS	JavaScript

CONTENTS

1 INTRODUCTION

Modern Web-based applications make extensive use of JavaScript (Gallaba et al. (2017)). A survey from StackOverflow [1], in 2018, presents JavaScript as the most widely used programming language. This result has been consistent throughout the last six editions of the survey. Besides that, the three most often cited frameworks and libraries by survey respondents are Node.js, Angular, and React, all of them based on JavaScript. In particular, Node.js, a framework and runtime system for the construction of standalone JavaScript applications, is used by almost 50% of the respondents. Furthermore, JavaScript appears in the first place of the January 2018 Redmonk programming languages ranking[2], which measures popularity based on the number of questions on StackOverflow and repositories on Github. These numbers highlight the relevance of JavaScript, in general, and Node.js, in particular.

Exception handling mechanisms are designed to detect an occurrence and recover from errors. Poor quality in the design of exception-handling (the developers approaches to deal with errors) can impact on system robustness. Exception handling is an important quality attribute of software, however, it is one of the less understood and neglected parts of software development (Chen et al. (2009)). Mikkonen and Taivalsaari (2007) claimed that this design decision make tracing and debugging JavaScript systems even harder. They presented examples of error-tolerance of JavaScript: misspell a variable name results in the creation of the variable, the developers can access to non-existent properties in the objects, the developer may omit the `return` statement in a function which will turn the value `undefined`. Any of those scenarios may lead to unexpected behavior in the system. Equivalently, syntax errors, as using square brackets instead of parentheses, in a function call is not reported and can bring problems to trace the occurrence of the error.

Due to the widespread adoption of JavaScript, several empirical studies have been conducted to analyze different aspects of the language and its usage. For instance, Richards et al. (2011) analyzed more than 500,000 calls to the `eval` function. They found out that most of the popular websites use this function and, in up to 2/3 of the cases, these are actually misuses. Gallaba, Mesbah and Beschastnikh (2015) studied 138 JavaScript programs to understand how they use callbacks. They discovered that one in ten function definitions take a callback function as an argument, most of those callbacks are nested, and more than half are asynchronous. In the work of Gallaba, Mesbah and Beschastnikh (2015), an asynchronous callback is a callback that is eventually passed to an asynchronous API call. More recently, Wang et al. (2017) analyzed 57 concurrency bugs in Node applications and discovered that 2/3 of those bugs are caused by atomicity violations. According

[1]https://insights.stackoverflow.com/survey/2018
[2]http://redmonk.com/sogrady/2018/03/07/language-rankings-1-18/

to Hong, Park and Kim (2014), an atomicity violation is an unintended race condition that an operation may be scheduled between two operations that should be executed consecutively.

Previous work analyzed the usage of error handling constructs is JS programs. Jakobus et al. (2015) have focused exclusively on exception handling. They did not study the potential differences between Web-based and Standalone systems. In addition to error handling in JavaScript applications, Jakobus et al. (2015) have analyzed 50 software projects written in C++, PHP, Java and C#. They analyzed 9 million lines of code and over 20,000 error handlers. In addition, they analyzed both exception scopes and handlers, which are code that may raise exceptions and handlers for those exceptions, respectively. We did not study error scopes as Jakobus et al. (2015) did, but we did a more in-depth study of error handlers, mainly because in the Jakobus' study, although comprehensive in relation to the programming languages analyzed, only considers `try-catch` blocks.

In face of the high popularity of Node, we aim to identify not only the error handling approach of JavaScript community as a whole, but also the differences between the applications that use Node and those that do not use it. Thus, we classified them as Web-based and Standalone, considering the Node usage.

1.1 RESEARCH OBJECTIVES

In this work, we contribute to the existing body of knowledge by studying how JavaScript projects use the error handling mechanisms of the language (`try-catch` blocks and callback functions). Additionally, we also analyzed the employment of other abstractions that uses the error handling mechanisms namely: promises, events, and `async`hronous functions. Previous work of Jakobus et al. (2015) on error handling, analyzed five programming languages (including JavaScript). They focused solely on the `try-catch` blocks without regarding callback functions. However, callback functions are a commonplace in real-world JavaScript applications and an integral part of software development on Node.js (Gallaba, Mesbah and Beschastnikh (2015)).

We organize this master dissertation through two lines of investigation. Firstly, we explore how the developers commonly employ error handling mechanisms on JavaScript projects (thus, we analyzed the error handlers and the strategies they employed). Secondly, we compare both mechanisms and strategies for handling errors in Web-based and Standalone projects, particularly considering the Node usage.

The main objectives of this empirical study are:

- Gaining an understanding of JavaScript error handling in practice. Characterize the error handling mechanisms (and their abstractions: `try-catch` blocks, callback functions, promises, events and async-await), which error handling mechanism are more employed, how often they are employed, and the error handling strategies are being employed considering the repositories in Github from the dataset that we analyzed;

- Understand and compare the differences between Web-based and Standalone systems, regarding the error handling mechanisms (and their abstractions: `try-catch` blocks, callback functions, promises, events and async-await) and the error handling strategies they employ.

1.2 CONTRIBUTIONS

In this work, we analyzed error handling mechanisms and strategies of JavaScript, and the differences between Web-based and Standalone JavaScript systems. To the best of own knowledge, this is the first attempt to analyze error handling in JavaScript considering the error handling mechanisms besides `try-catch` blocks, and also regarding another ways to apply error handling mechanism (like callback functions, promises, events and async-await functions). This work makes the succeeding contributions:

- **An understanding of error handling of JavaScript projects.** We conduct an analysis of the error handlers in JavaScript systems, and compare to what JavaScript community expects errors to be handled. In order to make this comparison we used Google to retrieve guides about error handling for both JavaScript and Node projects;
- **A tool for extracting error handling strategies.** We use Esprima and Estraverse libraries to create Abstract Syntax Tree (AST) for the files of JavaScript applications. The objective of this tool is to extract information about the error handlers (mechanisms and strategies employ) to construct the dataset.
- **A dataset of error handlers extracted of JavaScript applications from GitHub.** We retrieve error handling mechanisms (and abstractions) and error handling strategies from Github that may be used for further research in the academic community;

1.3 STRUCTURE OF THE DISSERTATION

The remainder of this dissertation is organized as:

Chapter 2 reviews the literature and documentation of the JavaScript programming language, regarding the mechanisms and abstractions used to handle errors. Firstly, we present the error handling concepts and the mechanisms used in JavaScript. In this chapter, we also explain the syntax of the mechanisms and abstractions, and the intrinsic relation between asynchronicity and error handling in JavaScript.

Chapter 3 reports the methodology adopted in this thesis. It introduces the target systems of the study, the premises we have assumed for the extraction of software metrics from these systems, and the rationale for those premises. In this section, we explains the statistical analyses we performed.

Chapter 4 presents the results of the analysis of the study. The chapter presents an overview of the data analysis concepts applied and the results obtained by this analysis.

Chapter 5 provides a discussion about our findings presented in Chapter 4 and the implications they have in the JavaScript community.

Chapter 6 presents the conclusions of this dissertation, and indicates several paths for future work.

2 THEORICAL FOUNDATION

In this chapter, we introduce the main concepts of error handling and the mechanisms used in JavaScript. Those mechanisms allow developers to signal the occurrence of errors and to change the program control flow in order to handle those errors when they occur (Garcia et al. (2001)), ideally separating error signaling code from error handling code (Lee and Anderson (1990)).

Error handling constructs define two parts of the behaviour program: *error scope* and *error handler*. We consider the definition presented by Lee and Anderson (1990), *error scope* is the *normal behavior* of the system, and *error handler* deals with exceptional situations, is the the code that handles with errors. The terms used by Lee and Anderson (1990) to categorize systems for exception handling is *normal part* and *abnormal part*. They claim that the most of the refactorings is done aiming to improve the software belonging to *normal part*. The robustness of the system is enhanced by changing the exceptional behaviour without changing the normal system behaviour.

Furthermore, by knowing that systems have *normal part* and *abnormal part*, the programming languages should provide mechanisms to change the flow appropriately between these two categories. JavaScript provides two mechanisms to recover from errors: an exception handling mechanism (`try-catch` blocks) and callback functions, which is a function designed to be invoked later with a result currently unavailable (Brodu, Frénot and Oblé (2015)). The decision of using one or another is mainly based on the nature of functions inside the error handling scope. Callback functions are employed mainly to handle errors in potentially asynchronous code. Whereas, `try-catch` blocks are used in either async and sync code, through async-await functions.

This chapter is organized as follows. Section 2.1 presents the terminology required to understand error handling in general. Section 2.2 introduces JavaScript error handling mechanisms and, in the subsections, presents the concepts and examples of the error handling abstractions using the error handling mechanism. Section 2.3 presents how global event handlers works. Although, global event handlers employ callback functions to handle errors, they are different from the error handlers presented in the earlier sections, as the goal of a global event handler is to deal with any errors that appears in the application in the lack of any other error handler. In Section 2.4, we present the concepts of Web-based and Standalone applications, and in Section 2.5, we present previous research works in error handling and JavaScript that are related to this work.

2.1 ERROR HANDLING TERMINOLOGY

In this section, we present the terminology used throughout this master dissertation. The concepts presented here are the foundation to understand error handling approaches used by JavaScript developers to recover from errors.

Error: is a part of an erroneous state which compose a difference from a valid state according to Lee and Anderson (1990).

Stack trace: consists of a runtime exception and a function call sequence at the moment of the crash (Gu et al. (2019)).

Handle type: JavaScript is weakly typed, and does not apply any restrictions on the types the variables can refer to (Ocariza, Pattabiraman and Zorn (2011)). Thus, developers are not forced to define specific types of errors, any type even a string, number, or NaN (not a number), may be used for signaling an error. Although a native class Error was created for this propose, developers might define their own error classes.

Error handling strategy: is the set of statements created to recover from an error raised on the application Sena et al. (2016). This concept involves actions like printing a message on the console, throwing (or re-throwing) an error, executing a more complex set of actions in order to maintain the application working properly, available (crucial for web applications), and in a consistent state.

Employed mechanism: has the goal of generalizing operations of an object and make them able to handle errors (Miller and Tripathi (1997)). In this study, we analyzed `try-catch` blocks and callback functions of JavaScript applications.

Error recovery: involves changing the program flow in order to execute actions to handle the error and putting the program back into the expected execution path.

Asynchronous functions: A function that uses an asynchronous callback is defined by having one of its parameters as the callback argument (Gallaba et al. (2017)). Although it is not directly related to error handling, it plays a key role on the decision of which mechanism a developer should use in JavaScript.

2.2 EXCEPTION HANDLING MECHANISMS

According to the Mozilla's documentation[1], JavaScript has a compact set of statements to define the control flow of the application. These statements allows developers to handle interaction users usually do towards the interface of the system in the web.

Considering error handling, whenever an error occurred, the programming language should provide ways to change the flow in order to execute a different set of statements from the expected path. The representation of an error are commonly an object, as in Java or C# (Ben-Assuli and Jacobi (2012)). The documentation explains that strings and numbers are often `thrown` as errors, although there is a recommendation to use one

[1]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling

of the objects designed for this purpose, *e.g.*, Error object[2]. A list of error objects is available in ECMAScript and two interfaces designed for this purpose: DOMException and DOMError.

Although the above is also valid to Node.js[3], the framework presents different objects to signal an error and applications may present the errors objects:

1. Standard JavaScript errors such as `EvalError`, `SyntaxError`, `RangeError`, `ReferenceError`, `TypeError`, and `URIError`.

2. System errors triggered by underlying operating system constraints such as attempting to open a file that does not exist, or to send data over a closed socket.

3. User-specified errors triggered by application code.

4. AssertionErrors are a special class of error that can be triggered when Node.js detects an exceptional logic violation that should never occur. These are raised typically by the `assert` module.

All error types in Node.js are inherited from JavaScript `Error` object, and have all the properties of this object. The documentation also states that all errors raised by `throw` statement should be handled by `try-catch` constructions, otherwise the process will stop immediately. This is also are valid for synchronous processing.

For asynchronous processing, in agreement with Node.js documentation[4], there are three approaches for handling errors:

- the definition of a method that accept callback functions and the first argument is an `Error` object. This approach is also called "first-error callback" (Gallaba, Mesbah and Beschastnikh (2015)), where the first argument stores the error object, if it exists. Otherwise it receives the `null` value. This allows the developer to check the existence of such an error by employing an `if` statement.

- An asynchronous method may be called from an object inherited from `EventEmitter`, such errors are routed to `error` events.

- Some other methods in the Node.js API, although asynchronous, may still use the `throw` mechanism and therefore should be handled by `try-catch` mechanism.

The first two approaches are callback functions being used to handle errors in two different ways. The last approach uses `async-await` abstraction which will be presented in the next sections.

[2]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error
[3]https://nodejs.org/api/errors.html#errors_class_error
[4]https://nodejs.org/api/errors.html#errors_error_propagation_and_interception

In this section, we discuss how those error handling mechanisms are structured, how they work, and finally, we present an example. We also introduce additional error handling abstractions that are build upon these mechanisms, namely promises, events and `async-await` constructions.

2.2.1 Try-catch blocks

`Try-catch` block separates the code that may throw an error from the code that handles the error. In the `try` block, resides the code that may throw errors, and in the `catch` block resides the code responsible for bringing the system back to a consistent state. An error is an object that is signaled through a `throw` statement. When an exception is thrown, JavaScript looks for an appropriate handler for the error, first within the current method. When none is found, the error is propagated up through the call stack until a handler is found. Only exceptions `thrown` from within a `try` block will be handled by the associated `catch` block.

Code 1 presents an example of a `try-catch` block. Line 2 invokes the `eval` function, which receives a text string and interprets it as JavaScript code. That line has a parse error due to a missing quote (") and thus throws an exception. At that moment, the control will jump to the `catch` block, which will handle the error by printing an error message to the console.

Listing 1 – Synchronous code in JavaScript using `try-catch`

```
1  try {
2    eval('alert("Hello world)');
3    // prints SyntaxError: Invalid or unexpected token
4  } catch(error) {
5    console.log(error);
6  }
```

2.2.1.1 Async functions

Async functions were introduced into Node.js 8 and are part of ECMAScript 2017. Async function performs error handling like promises, through the `try-catch` mechanism Wilson (2018). It is basically syntactic sugar for the promises. However, all promises declared inside an asynchronous function must be preceded by the `await` keyword. Whenever `await` appears during the execution of a program. Execution is paused until the expression is completed by resolution or rejection of the promise[5]. A return of `async-await` is an AsyncFunction object which represents the asynchronous function.

[5]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

Code 2 shows an example extracted from Stackoverflow[6]. It presents the `getProcessedData` method which is an `async-await` function that downloads some data and returns a promise through `processDataInWorker` method. Any error occurred in the `try` block is recovered by `catch` scope. Code 3 shows a version of `getProcessedData` method using promises, implemented by the author in Stackoverflow.

Listing 2 – Processing data from URL using `async-await` abstraction

```
 async function getProcessedData(url) {
  let v;
  try {
    v = await downloadData(url);
  } catch(e) {
    v = await downloadFallbackData(url);
  }
  return processDataInWorker(v);
}
```

Listing 3 – Processing data from URL using promise

```
function getProcessedData(url) {
  return downloadData(url) // returns a promise
    .catch(e => {
      return downloadFallbackData(url) // returns a promise
    })
    .then(v => {
      return processDataInWorker(v); // returns a promise
  });
}
```

2.2.2 Callback functions

A callback is a function passed as an argument to another function to be invoked some time later (Gallaba, Mesbah and Beschastnikh (2015)). Callbacks are usually employed in scenarios where a function is expected to have long execution time, *e.g.*, to access an external resource. Whenever a caller f calls a function g, it passes a callback function to be invoked by g upon its completion. Most importantly, f does not stay blocked waiting for g to complete, *i.e.*, g may be executed asynchronously. Callback functions often take two parameters, one is an error indicating that g could not successfully complete and the other one is the result of a successful execution of function g.

[6]https://stackoverflow.com/questions/44029927/how-can-i-use-aync-await-in-javascript

Callbacks can be synchronous (when executed after the invoked function returned) or asynchronous (when deferred to execute some time later after the invoked function returned). The JavaScript community employs a code practice called “error-first protocol” Gallaba, Mesbah and Beschastnikh (2015), reserving the first parameter of a callback function for errors and the second argument is reserved for any successful request data. In this practice, an error occurs when the first parameter is not `null`. Code 4 shows an example of “error-first protocol”. In this example, a product is removed from a database by calling the `remove` function. This function receives an anonymous callback function whose parameters are `err` and `removedProd` (Line 1). If an error occurs, the callback will be invoked with a non-`null` argument corresponding to `err` (as define by the error-first protocol). Thus, the condition of the `if` statement of Line 2 will be `true` and the error will be handled by function `handleError`. Otherwise, if `err` is `null`, the id of the removed product will be printed to the console. In this example, the `if` statement of Line 2 acts as the error handler.

Listing 4 – An example of callback usage to handle an error

```
1   product.remove(function (err, removedProd) {
2     if (err) return handleError(err);
3     console.log(removedProd._id + " removed.")
4   })
```

Callback functions, as any function definition in JavaScript, may be defined using one of three approaches available for this purpose. The first approach consists of using the `function` keyword to declare a named function[7]. The argument function passed to `remove` in Code 4 is an example of a function expression. When the function name is omitted, they are called anonymous functions or “function expressions”[8]. Lastly, it is possible to define “arrow functions”[9], a more compact version of function expressions. Line 6 of Code 5 exhibits an example of arrow function through the assignment of variable `getRectArea` that receives an arrow function. This function receives two parameters, `width` and `height`[10], and return the area of the rectangle.

Listing 5 – A function declaration and an arrow function

```
1   // A named function declaration
2   function calcRectArea(width, height) {
3     return width * height;
4   }
5   // An arrow function
6   var getRectArea = (width, height) => width * height;
```

[7]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function
[8]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function
[9]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions
[10]http://mongoosejs.com/docs/api.html#Query

Callbacks can also be employed in event handlers and in promise objects. We discuss them in detail in the remainder of this section.

2.2.2.1 Events

Events use the publisher/subscriber paradigm to perform asynchronous execution. This paradigm relies on publishers and subscribers that interact to each other by sending and receiving a notification of an occurrence of a event. Publishers publish a notification of an event (like a click in a button), and this event is asynchronously notified to all subscribers registered to be notified in face of that event. Eugster et al. (2003) present the concept of "event service", which represents the mediator between publishers and subscribers. They also declare that the dissociation between publishers and subscribers provided by "event service" is composed by the following concepts:

Space decoupling: publishers and subscribers do not need to hold references for each other, and do not even need to know how many entities (either publishers or subscribers) are involved in the system.

Time decoupling: the interaction between publishers and subscribers does not have to happen at the same time. This is a basic premise for asynchronous execution.

Synchronization decoupling: publishers and subscribers are not blocked while publishing or being notified of the occurrence of an event. Thus, the interaction occurs asynchronously.

To understand how the event parties are defined in JavaScript, we list important concepts about events and how developers may publish or notify events:

Event type is the object that represents the event itself and defines what type of error has occurred. For instance, an event that represents the mouse is moving towards to a specific component is called "mouse-move".

Event target is the object the event is associated to. On the example of a button click, the event target is the button itself where the click happened.

Event handler is the function called when an event occur.

Event registers is a function designed to register an event and associate event handler and event target. Through this function, an event target is able to subscribe to an event, or to unsubscribe depending on the function called.

Event triggering is the process of executing all event handlers to an event in a first-in first-out (FIFO) order of subscription.

In this style of programming, an event is propagated in the system to indicate the completion of a task or an user interaction. Although the events are usually of string type, there is no restriction made of the language for another type to be used for that purpose, that is, any object can be used as an event.

It was also stated that both Web-based and Standalone applications use an event-driven model[11,12]. There are natively predefined events, both by Node.js and by ECMAScript Flanagan (1998). In Node.js, all objects that emit events are instances of `EventEmitter` class. They can subscribe to an event by calling methods `on` or `once`. The `on` method receives two parameters: an event (usually a string defined which event is related to) and an event handler (a function that should be called whenever the event occurs), and the second method `once` do exactly the same, however after the first occurrence of the event, the handler is automatically unsubscribed. To publish an event, in Node.js, the developer should call `emit` method which receives an arbitrary number of parameters, those are the events.

Code 6 shows an example of the events usage of `on` method. The object `http` is created in line 3, through reference of the library. Then, the `get` method is called to execute the request, receiving an object representing the request parameters as the first parameter and a callback function to handle the response. While the response is being received, an event called `data`, triggers the function defined on line 14. On the conclusion of sending the response, an event called `end` triggers the function defined on line 18.

Listing 6 – A GET request is created using `http` library in Node

```
1   'use strict'
2
3   const http = require('http');
4
5   http.get({
6     hostname: 'localhost',
7     path: '/user?name=jv&jovem=true&agr=19',
8     port: 3000,
9     agent: false
10  }, function(response) {
11    let body = "";
12    console.log(response.statusCode);
13    console.log(response.headers);
14    response.on('data', function (data) {
15      body += data;
16    });
```

[11]https://nodejs.org/api/events.html
[12]https://developer.mozilla.org/en-US/docs/Web/API/Event

```

    response.on('end', function () {
      console.log('Resposta:', body);
    });
  });
```

Mozilla's documentation presents methods for handling events[13], however the methods present are not designed directly to handle errors. Some of those methods are currently deprecated (`Event.createEvent` and `Event.initEvent`), and others are available to use: `Event.composedPath`, `Event.preventDefault`, `Event.stopImmediatePropagation`, and `Event.stopPropagation`. These methods do not necessarily handle errors. Libraries design their own API methods, events, variables, including the name of the functions that handle events. We were not able to accurately identify the specificities of each library used in each one of the projects. For this reason, we consider only the three methods defined by Node.js for listening and handling events, the methods: `on` and `once`.

2.2.2.2 Promises

Promises are an extension of JavaScript and first appeared on ECMAScript 6. They are objects that represent a future result, whenever is successfully computed, the promise is called resolved, if any errors emerge, the promise is called rejected[14]. Brodu, Frénot and Oblé (2015) declare that promises also combines synchronous and asynchronous execution, allowing the sequentially of control flow, while providing use of continuations. Code 7 presents an example of a promise creation using the `new` keyword. It receives two parameters: `resolve` and `reject` callbacks functions employed to handle the promise state. Whenever a promise completes the operation successfully, it should call `resolve` function passing the result, otherwise, `reject` callback is called passing the error.

Listing 7 – A Promise object creation

```
  let promise = new Promise(function(resolve, reject){
    // do a thing, possibly async, then
    if(/* everything turned out fine */){
      resolve("Successfully completed!");
    }else{
      reject(Error("Error occurred!"));
    }
  });
```

Through the exposure of the method `then`, developers may nest callback functions, and perform a conglomerate of functions executing one after the other by passing the

[13]https://developer.mozilla.org/en-US/docs/Web/API/Event
[14]http://documentup.com/kriskowal/q/

result of the previous promises in the order defined in the chain. Another term used to refer to promises are the “thenable” objects for supporting developers to attach callbacks on a promise chain (Kambona, Boix and Meuter (2013)).

Errors may occur in any of the callbacks involved in the chain. Legibility is negatively impacted when more callback functions are nested into the chain. The creation of the concept “promise”, is highly due to “callback hell”[15]. For nesting callback functions, promises uses the `then` method. It receives two arguments, the first is the callback that is called whenever a result is successfully computed. The second argument, which is optional, is a callback function call when a failure occurred in the execution of a promise chain. There is also a syntax sugar for `then(null, failureCallback)`, it is the method `catch`. When a promise chain fails, the following step is to search for a promise in the chain that the second parameter is not undefined, or `catch` calling. Code 8 shows both method signatures, using the second parameter to pass the callback function to handle errors (line 10), and omitting the handler function on method `then` (line 12), using the method `catch` for error handling.

Listing 8 – Method signature `then` for promises

```
1   function onSuccess(result){
2       // "Successfully completed!"
3       console.log(result);
4   }
5
6   function onError(err){
7       console.log(err);
8   }
9
10  promise.then(onSucess, onError);
11
12  promise.then(onSuccess)
13      .catch(onError);
```

2.3 GLOBAL EVENT HANDLERS

The mechanisms we presented are Functions objects, in fact, all JavaScript function is actually a Function object[16]. This allow developers to create their own objects, such as definitions of promises (Promise object[17], and creation of new `async` functions (Async-Function[18]).

[15]http://callbackhell.com/
[16]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function
[17]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises)
[18]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

Global event handlers are mixins, which is an interface that some or all its methods and/or properties are unimplemented[19]. Global event handlers describe event handlers for specific interfaces such as `HTMLElement`, `Document`, or `Window`. There are several event handlers natively defined in JavaScript, categorized as Global event handler according to Mozilla's documentation[20].

For error handling, global event handlers may be used for trackle situations such as shutdown of server or failing to load data, also useful for automated collection of error reports. There are two ways to fire an `error` event[21]: (i) a JavaScript runtime error (such syntax errors thrown in the handlers), (ii) a resource failed to load.

2.3.1 JavaScript Global event handlers

For JavaScript applications, developers may define global event handlers through the window and element objects. We present the code signature of the error event handlers in Code 9.

Listing 9 – Global event handlers signature for JavaScript

```
// MDN web docs
window.onerror = function(message, source,
    lineno, colno, error) { ... }

window.addEventListener('error', function(event) { ... });

element.onerror = function(event) { ... }
```

2.3.2 Node Global event handlers

In Node, there is an event listener method named *on* called by the *process* object to handle events in a global way. Code 10 shows the code syntax for globally handle errors in Node. The "uncaughtException" event is emitted whenever an uncaught JavaScript exception bubbles up to the event loop. By default, Node prints the stack trace to stderr and exits with code 1, overriding any previously set `process.exitCodeNode`. Any exception thrown inside this global event handler is not caught. The Node documentation[22] recommends to not resume the application after an "uncaughtException" event.

Listing 10 – Global event handlers signature for Node

```
// Node.js documentation
process.on('uncaughtException', function(event) { ... });
```

[19]https://developer.mozilla.org/en-US/docs/Glossary/Mixin
[20]https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers
[21]https://developer.mozilla.org/en-US/docs/Web/API/GlobalEventHandlers/onerror
[22]https://nodejs.org/api/process.html#process_event_uncaughtexception

2.4 WEB-BASED AND STANDALONE SYSTEMS

Besides the error handling mechanisms and their abstractions, to deal with errors in JavaScript applications, we present two concepts directly related to the interaction of the applications to other applications. Applications may use a browser as an environment even if they never actually perform a request to a server. In this work, we consider that Web-based applications are applications that do not connect to another application (server) and otherwise, we classified the application as Standalone.

We believe that these two classes may have to deal with different types of errors. Standalone applications have to deal with errors that happens in the browser independently of an outside server. In Standalone applications, in general, when an error is not handled, it is printed in the console. Web-based applications may send information to a server, and errors may occur in different parts of this system. Web-based applications rely on servers to retrieve and store information. Once they are needed, Web-based applications must send a request to the server. Similarly, when the applications need to save an information, it must send it to the server. In this case, Web-based applications may deal with errors like the server is down. Node applications crash without any notice in a lack of a global handler (in the Node applications, in specifically, through the calling of `process.on('uncaughtException', callbackFunction)`). The details in how we classified the applications considering the concepts presented here, are described in the next chapter.

2.5 RELATED WORK

In this section, we present a short description of the research which has been conducted in areas related to our work.

2.5.1 JavaScript

Gallaba, Mesbah and Beschastnikh (2015) studied JavaScript callback functions through 138 JavaScript applications, with over 5 million lines of JavaScript code. They claimed that developers are often frustrated by “callback hell” problem, which refers to nested and anonymous callback functions and asynchronous callback scheduling. They found that one in ten function definitions takes a callback argument, and that over 43% of all callback-accepting function callsites are anonymous. They analyze the adoption of first-error callbacks in the JavaScript programs, and found that one in five callback functions adheres to the error-first protocol. Besides that, they compared client and server applications, and server-side code employ first-error callbacks twice as often than client-side code. In this work, we did not explore callback functions in their general usage, but in their usage for handling errors, although they investigate the error-first protocol. They also divided the client-side and server-side code in sub-categories (DataViz, Engines, Frameworks, Games, Webapps, NPM), which is an idea that we could investigate in our dataset in the future.

Ocariza, Pattabiraman and Zorn (2011) provide an analysis of the error messages printed by JavaScript code in web applications, and investigate the root causes. They found that approximately 93% of the errors are categorized as such: Permission Denied (52%), Undefined Symbol (28%), Null Exception (9%), and Syntax Errors (4%), and about 72% of the errors are non-deterministic (i.e., vary across executions). They also analyzed the source of the errors, through the correlation of applications' static and dynamic characteristics (such as number of calls to `eval`). We conceive that our work may be enhanced whether we consider the nature (dynamic or static) of the target systems we analyzed.

2.5.2 Exceptions

Cabral and Marques (2007) investigated error handling mechanisms employed in 32 different applications in Java and C# systems. They noticed that error handlers are usually not specialized enough, programmers usually do one of the following approaches: logging an error message, notify the error to the user, and application termination. Although they found that the exception objects of these error handlers are very specific types and bound to the system logic. Besides error handling, they also evaluate `finally` clauses in `try-catch` blocks. We do not investigate `finally` clauses in our work, however it may be interesting investigate them in the future.

Jakobus et al. (2015) examined commonalities and differences of both exception scopes and handlers, through of analysis of 50 software projects, containing code developed in C++, JavaScript, PHP, Java and C#. They evaluated over 20,000 exception scopes and handlers. Their results reinforced the current belief that developer are used to employ a simplistic approach in exceptions handlers, regardless of programming language used. In our work, we took a deeper analysis in error handlers for JavaScript, as we evaluated callback functions for handling errors besides `catch` blocks. We do not analyze the strategies in the error scope (code inside `try` blocks, in the method `then` of promises and inside functions that raise error events) that may be an interest idea. However, it needs more investigation.

Kery, Goues and Myers (2016) analyzed 11 million `try-catch` blocks in Java from Github applications. They found 1,515,523 catch empty handlers, 12.4% of the total. They found that exception are mostly handled locally in `catch` blocks, instead of being propagated by throwing an Exception. They also found that the handlers use mostly actions like Log, Print, Return, or Throw in `catch` blocks. They claimed that bad practices like empty `catch` blocks or catching Exception are widely spread.

3 METHODOLOGY

In this chapter, we report our methodology for this empirical study. This includes how we build the dataset, the assumptions we present due to specificities of the JavaScript language, and how we elicited the software metrics to analyze the error handling usage on the applications.

In Section 3.1, we present our research questions. Section 3.2 explains how we selected the repositories we study in this work. In Section, 3.3, we discuss some of the obstacles to automatically extracting information about error handling from JavaScript projects and how we identified error handling code. In Section 3.4 presents how we elicit the error handling strategies, and in Section 3.5 how we perform the extraction of those strategies from the JavaScript projects. In Section 3.6, we compose a set of recommendations and antipatterns for error handling in JavaScript systems we extracted from guides in the grey literature (Auger (1975)). And, finally, Section 3.7 presents the threats to validity.

3.1 OVERVIEW

In this work, we aim to get a deeper understand on error handling in JavaScript through an analysis of the mechanisms and the strategies used to deal with errors. In this study, we address the following research questions:

- **RQ1.** How do developers handle errors in JavaScript applications?
- **RQ1.1.** Which error handling mechanisms do developers employ?
- **RQ1.2.** Which error handling strategies do developers employ?
- **RQ2.** Are there any differences regarding error handling in Web-based and Standalone JavaScript applications?

RQ1 is broken into two different and complementary questions. To answer **RQ1.1**, we elicited software metrics that the systems may employ, based on previous research studies. Then, we collected the software metrics pertaining to the use of the two mechanisms from 192 projects comprising both Web-based and standalone JavaScript systems. Furthermore, we measure the extent to which different abstractions (details in Section 2.2) are employed in these projects. For **RQ1.2**, we measure the prevalence of well-known error handling patterns and anti-patterns (Cabral and Marques (2007)) in these systems. Finally, to answer **RQ2.** we compare the mechanisms and strategies usage by both Web-based and Standalone applications for handling errors.

3.2 PROJECT SELECTION

To build the dataset of projects to be analyzed, we collected a sample that aims to represent a population of engineered, non-trivial, popular open-source JavaScript systems from the Github. We followed recommendations presented by Kalliamvakou et al. (2014) for mining repositories on Github. The authors indicate perils in mining repositories. Among the recommendations and perils, the ones relevant to this study are: some projects may be not very active, have a small number of commits, or be personal projects. Bearing those recommendations in mind, we used the Github web page to query repositories, by using the "advanced search" page[1]. We select repositories with at least 1,000 stars, 500 forks, and whose last commit occurred less than a year ago (between November 2017 and November 2017). We also selected repositories from[2], a curated list of JavaScript systems which describes itself as a *"collection of awesome browser-side JavaScript libraries, resources and shiny things"*. The projects we selected from this list also meet the previously mentioned criteria. We disregard forks in our analysis. As a result, our dataset comprises 192 repositories.

3.2.1 Classification of Web-based and Standalone systems

We manually classified each repository as Web-based or Standalone since there is no reliable automatable approach to perform this classification. We aimed at classifying the repositories without the need to execute all of them. Thus, we examine the package.json file of each project, which is a manifest file of JavaScript projects, and analyzed meta information such as the project name, dependencies (and their versions), commands, version, etc[3]. Among those properties, only *name* and *version* are mandatory. The full list of properties a package.json file may have is publicly available[4]. We highlight the properties *scripts* and *engines*. The first one is a dictionary which stores script commands that are recurrently executed during the application life cycle. The key is the life cycle event and the value is the command. An example of usage of the property *scripts* is presented in Code 11 in Line 8. The *engines* property is also a dictionary to specify Node and other commands the package/app work on. An example of the *engines* property is shown in Code 12. To enhance the classification method, we also manually checked the README.md file on the repositories searching for the project's purpose (either Web-based or Standalone). Finally, we found 86 repositories classified as Web-based and 106 classified as Standalone projects.

[1]https://github.com/search/advanced
[2]https://github.com/sorrycc/awesome-javascript
[3]https://docs.npmjs.com/creating-a-package-json-file
[4]https://docs.npmjs.com/files/package.json

3.3 IDENTIFYING ERROR HANDLING CODE IN JAVASCRIPT

After the classification of the repositories, we analyzed the mechanisms for error handling available in JavaScript. Similarly to other languages such as Java, C++, and Python; JavaScript also employs `try-catch` blocks for exception handling. In this case, identifying exception handling code is straightforward, as `try-catch` blocks explicitly separate error handling scope, we only need to retrieve the statements in `catch` blocks. `Try-catch` blocks explicitly separates code responsible for the normal activity of the system from the error handling code, differently from callback functions. In the subsections 3.3.1 and 3.3.2, we discuss the main obstacles to identify error handling callbacks. In the subsection 3.3.3, we present our approach to overcome these obstacles.

Listing 11 – Example of package.json for *scripts* property

```
{
    "name": "ethopia-waza",
    "description": "a delightfully fruity coffee varietal",
    "version": "1.2.3",
    "devDependencies": {
        "coffee-script": "~1.6.3"
    },
    "scripts": {
        "prepare": "coffee -o lib/ -c src/waza.coffee"
    },
    "main": "lib/waza.js"
}
```

Listing 12 – Example of package.json for *engines* property

```
{
    "engines" : {
        "node" : ">=0.10.3 <0.12"
    }
}
```

3.3.1 Dynamic typing in JavaScript

JavaScript is a prototype-based, multi-paradigm (object-oriented, imperative, and declarative), dynamic language[5]. As a dynamic programming language, variables are not directly related to a specific type and can be re-assigned to another type at runtime. There are seven native types available: boolean, null, undefined, number, string, Symbol and Object.

[5]https://developer.mozilla.org/en-US/docs/Web/JavaScript

Developers can create their own types by deriving from objects. Although JavaScript has included an `Error` object constructor[6] for this purpose, in practice any object, no matter its type, can be used to define an error. As a consequence, it is not possible to reliably verify if a complex object stores an error without executing the program.

The problem of determining which parameters were defined to store the error information directly impacts the identification of callback functions for error handling. As mentioned in Section 2.2.2, a callback function is a function f passed to another function g as an argument, which is then invoked within g to perform some type of task or action. Due to the lack of typification, this imposes an obstacle for identifying error handling in callback functions.

3.3.2 Single-use callbacks

JavaScript, similarly to other programming languages, defines functions or methods so that the developers may improve modularization to their systems. In a Web environment, there is a tag <script>, used to bind a JavaScript file to an HTML file. In order to reuse functions defined in different script files, those functions must be exported by this script. In order to import a script file, there are a variety of syntactic approaches[7]. In Node.js, the only way to import functions from other scripts is through the `require` method. This method may receive a string representing the name of the module (when importing a module directly from npm) or the path of a local script from the current project. There are some libraries such as RequireJS[8], Neuter[9], Browserify[10] and others that import modules likewise Node.js applications even if they may not be using Node.js at all. There are also some differences on importing script files when the project uses an transpiler (like Babel.js). After importing, module functions are now visible for each other. There is also the problem of determining that a symbol used within a function actually refers to a function even if is locally declared. Doing that with precision would require type information, which is unavailable in JavaScript. This is the reason why we also do not account for locally-declared functions used as callbacks.

Due to the dependency on libraries used on the projects, we are unable to ensure how a function was imported, and we cannot properly retrieve the function body to evaluate how errors are handled. As this obstacle directly impacts the retrieval of callback functions, we decided to disregard this case and only consider functions passed to the callback as lambda or arrow functions. Using this approach, we gain in precision but lose cases of callback usage.

[6]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error
[7]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import
[8]https://requirejs.org/docs/start.html
[9]https://www.npmjs.com/package/grunt-neuter
[10]http://browserify.org/

3.3.3 Tackling error handling code

In order to undertake the problem of error handling callbacks, we adopted an approach similar to the one employed by Gallaba, Mesbah and Beschastnikh (2015), searching for conventions in variable naming, although with some adaptations. They considered that an error parameter name should match the regular expression $e|err|error$ to consider that a callback function follows the error-first protocol (Gallaba, Mesbah and Beschastnikh (2015)).

We extend this assumption to classify any callback function (not to adheres to the error-first protocol). We also extend the regular expression used by Gallaba, Mesbah and Beschastnikh (2015) by performing a manual analysis of a sample of our dataset to check which words could be interesting to be used to identify callback functions for error handling. We randomly selected three repositories and retrieved a hundred functions. Then, we manually checked the callback function for error handling, and searched for naming patterns on the parameters passed to these functions.

We compile a list of the most often used words for naming parameters, object constructors, functions, or events (such as *uncaughtException*). The list consists of the words *reject*, *error*, *exception*, *reason*, *err*, *er*, and *e*. The words *reject* and *reason* usually refer to errors during the execution of a promise, i.e. the promise was *rejected* for a specific *reason*. We consider that a function is handling an error whenever at least one of its parameters name is included in the aforementioned list (ignoring capitalization). We also consider whether the function parameter name partially matches to at least one of the list of error parameter names that we build. To mitigate false positives, for the last two error parameters of our list (*er* and *e*), we require the currently function parameter we evaluate to be strictly equal. For the other words, we checked whether the keyword is part of the parameter name. For instance, a parameter named `connectionError`, *er*, or *rejected* would be assumed to refer to errors but `marker` would not.

We took a different approach for event handlers. As presented in Chapter 2, developers must register callback functions as listeners for events and they may not receive any parameters. Thus, for identifying handlers for error events we must complement the parameter name-based approach. Events are uniquely identified by either a `string` or a `Symbol`. A `Symbol` is *"... a unique and immutable primitive value and may be used as the key of an Object property..."*[11]. Moreover, according to the Node documentation[12], events that represent the occurrence of an error should have string type and be named *'error'*. In this work, we consider that any event whose name includes one of the defined keywords, represents an error. As a consequence, handlers for such events are considered to be error handling callbacks. In addition, for events, we only consider events defined through `strings`, since we have not found a single example of a `Symbol` event representing

[11]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#Symbol_type
[12]https://nodejs.org/api/events.html

an error. Code 13 shows an example of a callback that works as both an event handler and an error handler. The callback itself has no parameters, but it is identified as an error handler because of the name of the event it handles, the `reconnect_error`.

Listing 13 – Event listener creation for reconnect errors

```
    socket.on('reconnect_error', function () {
        console.log('attempt to reconnect has failed')
    });
```

3.4 CODE IN ERROR HANDLERS

Besides analyzing the mechanisms employed in JavaScript, to answer **RQ1.2**, we need to elicit the actions developers can take for error handling. Cabral and Marques (2007) studied error handling in Java and .NET systems. They collected a number of static software metrics for actions that may be performed for error recovery in the aforementioned programming languages. They named the software metrics (the approaches of the error handlers) by actions. Table 1 presents the name and description of the action categories used in the work of Cabral and Marques (2007). They found that 60% to 75% of the error handlers relies on three categories: Empty, Log and Alternative/Static Configuration. The first category refers to the error handlers that have no statements, the second category refers to error handlers that log a message only, and the last category refers to an event of an error or in the execution of a finally block some kind of pre-determined (alternative) object state configuration is used.

In this study, we renamed the action categories for strategies, and partially consider the actions categories of the work of Cabral and Marques (2007). Their actions categories were defined for Java and .NET applications. JavaScript presents different challenges which make some of those metrics not applicable for this study or demand a deeper analysis inside each code handler. From the actions categories considered by Cabral and Marques (2007), we regard in our study Empty, Log, Throw, Continue, Return. We also consider the action category Others that correspond to any other actions that do not relies into the previous categories defined by Cabral and Marques (2007).

We disconsider the following actions categories in this study: (i) Alternative/Static Configuration, (ii) Rollback, (iii) Close, (iv) Assert and (v) Delegates (only for .NET). The action categories (i) to (iv) are intrinsically bind to the different libraries that the projects may use. Libraries in JavaScript design their classes, objects and methods using different nomenclature, so that we could not properly determine their objectives only by their names, and JavaScript do not define typification. Thus, we could not properly classify the JavaScript error handlers in those categories. And, the action category (v) is exclusively of .NET.

Table 2 shows the strategies that we also consider to analyze the error handlers, based on the actions categories of the work of Cabral and Marques (2007) and in the rules of ESLint[13]. The first category (Single statement) consider if the error handler has only one statement. Break is a strategy that was defined by Cabral and Marques (2007) inside the action category Return. Differently from the work of Cabral and Marques (2007), that add break statements inside the action category Return, we decided to separate Return and Break in two different strategies. There is a specifically rule in ESLint to oblige developers to use return statement inside callbacks[14]. The strategy No usage of error parameters is referent to the rule *no-unused-vars*, more specifically to *caughtErrors*[15], with regard to not usage of any error parameter. Finally, the strategy Reassign parameters refers to the reassignment of an error parameter, losing the error information. Additionally, we retrieve the abstraction employed for each error handler, the number of lines, the number of statements, the number of statements, and the file we found the error handler.

[13]https://eslint.org
[14]https://eslint.org/docs/rules/callback-return
[15]https://eslint.org/docs/rules/no-unused-vars#caughterrors

Table 1 – Description of the Handler's actions categories

Category	Description
Empty	The handler is empty, is has no code and does nothing more than cleaning the stack
Log	Some kind of error logging or user notification is carried out
Alternative/Static Configuration	In the event of an error or in the execution of a finally block some kind of pre-determined (alternative) object state configuration is used
Throw	A new object is created and thrown or the existing exception is re-thrown
Continue	The protected block is inside a loop and the handler forces it to abandon the current iteration and start a new one
Return	The handler forces the method in execution to return or the application to exit. If the handler is inside a loop, a break action is also assumed to belong to this category
Rollback	The handler performs a rollback of the modifications performed inside the protected block or resets the state of all/some objects (e.g. recreating a database connection)
Close	The code ensures that an open connection or data stream is closed. Another action that belongs to this category is the release of a lock over some resource
Assert	The handler performs some kind of assert operation. This category is separated because it happens quite a lot. Note that in many cases, when the assertion is not successful, this results in a new exception being thrown possibly terminating the application
Delegates(only for .NET)	A new delegate is added
Others	Any kind of action that does not correspond to the previous ones

Source: Cabral and Marques (2007)

Table 2 – Description of code handlers strategies

Category	Description
Single statement	The handler has only one statement
Break	There is a `break` inside the handler
No usage of error parameters	The handler does not use any error parameters
Reassign parameters	The handler reassign an error parameter to another value

Source: Made by the author

3.5 DATA COLLECTION AND PROCESSING

We developed an extractor tool[16] using Node.js and libraries Esprima[17] and Estraverse[18]. Esprima was used to build the Abstract Syntax Tree (AST) for each file of the repositories and Estraverse is employed to visit the nodes of the AST.

The extractor tool clones the repositories from Github and performs various analysis on the source code. To ensure the files follow the same indentation rules, we perform some static code transformations using using Babel[19] and Uglify.js[20]. The changes we perform in the code of the applications are: i) remove the blank lines, ii) reformat the code so the indentation rules are the same throughout the files of the dataset, iii) transform function declarations into arrow functions (both function declarations and arrow functions are syntaxes of callback functions and are presented in the subsection 2.2.2), and iv) add brackets whenever an *if*, *for*, *do*, *while*, or *with* statement has only one line. We retrieved the number of `try-catch` blocks, `async-await` functions, promises, error event handlers, callback functions, and the previously specified action categories presented in Table 1 and Table 2.

The extractor creates an AST for each JavaScript file on the project. We do not consider files whose extension is *.min.js*, since those files are automatically generated by a process known as minification[21]. The purpose of minification is to obscure the code or to make the source code smaller, and hence, transfer a smaller amount of data over the network. We are not considering code that resides on HTML files. JavaScript code may be included inside HTML files, between the tags <script>. And, we also do not analyzed files under the `node_modules` directory, which is used by npm package manager[22] to store libraries code.

We recorded the results on spreadsheet files and performed statistical analysis using Python (libraries: numpy [23], scipy [24] and pandas [25]) for descriptive and inferential statistics. To compare the classes Web-based and Standalone applications, in both the error handling abstraction usage and the strategies employed by the classes, to increase the statistical power of the tests, we applied T-Student (Senn and Richardson (1994)) tests whenever the samples followed normal distributions, otherwise we applied statistical test defined by Mann and Whitney (1946).

[16]https://github.com/luanamartins/project-analysis/tree/master/extract-metrics
[17]http://esprima.org/
[18]https://github.com/estools/estraverse
[19]https://babeljs.io/
[20]https://www.npmjs.com/package/uglify-js
[21]https://en.wikipedia.org/wiki/Minification_(programming)
[22]https://www.npmjs.com/
[23]http://www.numpy.org/
[24]https://www.scipy.org/
[25]https://pandas.pydata.org/

3.6 RECOMMENDATIONS AND ANTIPATTERNS IN JAVASCRIPT ERROR HANDLING

To answer the research question **RQ1**, we believe that we need to compare the "how" developers handle errors with the way they (as a community) expect the errors to be handled. Besides that, referring to the **RQ2**, this analysis is also useful to identify the differences between Web-based and Standalone systems.

To analyze which programming practices the community expect errors to be handled, we adopt the methodology employed by Cassee et al. (2018) to elicit recommendations and antipatterns for error handling. We start by querying Google using the strings "javascript node.js error handling best practices" and "javascript node.js error handling anti-patterns" (both without the quotes). We then select the first 20 items of each query. We only selected documents that are directly related to error handling, disregarding questions from Q&A sites, and recommendations for clean code (related to best practices of software development). Barbosa et al. (2016) presents the following description of error propagation: "it specifies the specific places in the source code where specific exceptions are raised and handled, and also which specific exception types may flow between these places". We are unable to properly identify the places where an error has been thrown and where it is caught, thus we decided to leave out any recommendations or antipatterns pertaining to exception flow. In overall, we analyze 8 guides: Goldberg (2018), Joyent (2019), Nemeth (2017), Schardt (2018), Ershov (2018), Soares (2017), Notna (2017), Syed (2018).

Next, we study the selected guides to identify recommendations of practices that should be followed and that should be avoided pertaining to JavaScript error handling. We also retrieve these rules of ESLint[26] about error handling: Empty Handler[27], Ignore error[28], and Reassign error[29]. Table 3 shows the recommendations and antipatterns retrieved from the guides and from ESLint. Returning from within a callback do not appear in the previous works related to error handling as it is very specific to JavaScript. The rationale for the latter is that a problem may arise whenever the execution inside a callback function is not interrupted in face an error. Even if the developer uses an `if` statement, the callback function will continue the execution without a return statement after the identification of the error. The guide Nemeth (2017) declares that such behaviour may lead to unexpected scenarios, and recommends to always return inside callback functions.

Some of the recommendations and antipatterns such as Empty Handler and Log error appear before in the context of other programming languages (Cassee et al. (2018), Cabral and Marques (2007)), but others such as Ignore errors, Async-await/Promise usage have not appear in the other studies. The most cited recommendation is Async-await/Promise usage, endorsed by these guides: Goldberg (2018), Schardt (2018), Ershov (2018), Soares (2017),

[26]https://eslint.org/
[27]https://eslint.org/docs/rules/no-empty
[28]https://eslint.org/docs/rules/no-unused-vars
[29]https://eslint.org/docs/2.0.0/rules/no-param-reassign

Table 3 – Recommendations and anti-patterns

Recommendations	
Name	Guides
Async-await/Promise usage	Use Async-Await or promises for async error handling.
Error object usage	Use Error objects (or subclasses) for all errors.
Error-first protocol	Functions should expose an error-first callback interface.
Log error	Log the error — and do nothing else.
Return error	Return inside a callback function.
Antipatterns	
Name	Description
Empty Handler	Empty catch block.
Ignore error	Ignore errors in callbacks.
Reassign error	Reassignment of error parameter.
Throw error	Throw an error inside a callback function.

Source: Made by the author

Table 4 – Recommendations and anti-patterns

Recommendations	
Name	Guides
Async-await/Promise usage	Goldberg (2018), Schardt (2018), Ershov (2018), Soares (2017), Notna (2017)
Error object usage	Goldberg (2018), Joyent (2019)
Error-first protocol	Nemeth (2017), Syed (2018)
Log error	Joyent (2019)
Return error	Nemeth (2017), Ershov (2018)
Antipatterns	
Name	Guides
Empty Handler	ESLint (2019a)
Ignore error	Nemeth (2017), ESLint (2019c)
Reassign error	ESLint (2019b)
Throw error	Ershov (2018)

Source: Made by the author

Notna (2017). Error object usage is recommended by the guides: Goldberg (2018), Joyent (2019). Error-first protocol appears in guides Goldberg (2018), Syed (2018), and Log error is recommended only by Joyent (2019). The last recommendation is Return error, appears in Nemeth (2017), Ershov (2018). In the antipatterns, Empty Handler appears in ESLint (2019a), Ignore error is appears in Nemeth (2017), ESLint (2019c), and Reassign error appears in ESLint (2019b). In Throw error appears in Ershov (2018).

3.7 THREATS TO VALIDITY

Our work is subject to a number of threats to validity. We identified three kinds of threats to its validity: internal, external and construct, all of which are discussed below.

3.7.1 Internal validity

The threats to internal validity concern external factors we did not consider that could affect the variables and the relations being investigated. The main threat we faced is whether the repositories from our dataset are representative of real applications. A large number of Github projects are personal, not very active, or serve as storage purposes. We tried to avoid this threat by following the guidelines from Kalliamvakou et al. (2014) by selecting repositories based on metrics as forks, number of commits, watchers, the date of last commit. The repositories from our dataset have almost 600 watchers, at least 908 stars and more than a thousand closed issues in Github, showing a high interaction of the community.

3.7.2 External validity

The threats to external validity are related to the generalizability of the study results. Our results only apply to JavaScript projects on Github. It does not cover software projects in other source code hosting websites. Furthermore, our results are limited by our selection of repositories, hence, we used a set of guidelines proposed by Kalliamvakou et al. (2014) to query them.

3.7.3 Construct validity

The threats to construct validity are related to how properly a measurement actually measures the concept being studied. As JavaScript is a prototype-based language, we cannot properly retrieve information of attributes on objects, by checking if they really are related to errors. In order to mitigate this threat, we performed a manual analysis of a subset, to elicit the naming pattern used for error handling.

Another threat to validity is that we do not analyzed the JS code inside HTML documents, and the impact of them on the results are unknown.

4 RESULTS

In this chapter, we present the results of this study. We focus on the 179 projects that show at least one error handler. Bearing in mind the research questions presented in Chapter 3, we present the data analysis considering each research question in the next sections. Section 4.1 provides an overview of the repositories analyzed based on the methodology presented in Chapter 3. Section 4.2 presents the results of analysis for RQ1, on how developers deal with errors in JavaScript, first in terms of the error handler abstractions, and then by discussing the strategies employed by these error handlers. Section 4.3 shows the results of data analysis for RQ2, considering both error handler abstractions and strategies taken for handling errors on Web-based and Standalone systems.

4.1 OVERVIEW

In this section, we provide an overview of the analyzed repositories. Table 5 summarizes the projects that we analyze, and classified into Web-based or Standalone. Our tool for metrics' collection was unable to process 226 files due to Esprima library limitations. Errors may happen whenever an invalid program is parsed by Esprima, thus during processing we found: unexpected tokens (such as usage of `export` statement), the strict mode code may not include a specific statement used (e.g., the usage of the `with` statement is forbidden by strict mode rules). Esprima provides a mode called "tolerant" that may handle better these invalid programs. We decide not to use it, as this approach may still leave out some files, as the Esprima's documentation claims that the library is unable to robustly handle every possible invalid program[1]. Furthermore, 13 repositories (6.77% of our sample) do not employ any mechanism to handle errors. These repositories are very small in number of JavaScript files. We calculate the number of files with extension *.js* and not ending with *.min.js* for those repositories considering the criteria to analyze JavaScript code. In total, we found one repository with no js file, eight repositories with only one or two js files, two repositories with three files, one with four js files and one with 14 js files. Table 6 shows a summary of the 192 repositories. It presents the total and median numbers of stars, forks, watchers, open issues, closed issues, open pull-requests, and closed pull-requests of each class (Web-based and Standalone) and the entire dataset. The projects in the dataset we built are popular, having a median of 9,367 stars and 1,035 forks. The repositories in the dataset have an overall of 11,576,639 LoC, 6,097,966 LoC in Web-based systems, and 5,478,673 LoC in Standalone ones.

[1]https://media.readthedocs.org/pdf/esprima/4.0/esprima.pdf

Table 5 – Summary of the repositories

Metrics	**Web-based**	**Standalone**	**Overall**
Overall Repositories	86	106	192
Analyzed repositories	81	98	179
Overall files	60,090	55,722	115,812
Analyzed files	59,972	55,614	115,586
Median of files per project	82	93	175
Overall LoC	6,097,966	5,478,673	11,576,639

Source: Made by the author

Table 6 – Statistics about the repositories in the dataset

Metrics	**Web-based**	**Standalone**	**Overall**
# of stars	1,211,116	1,428,911	2,684,414
# of stars (median)	8,593	9,501	9,367
# of forks	267,252	192,539	468,419
# of forks (median)	1,031	1,020	1,035
# of watchers	54,361	53,810	109,460
# of watchers (median)	307	311.5	309.5
# open issues	20,530	17,682	38,369
# open issues (median)	91	108	99.5
# closed issues	129,324	173,489	305,165
# closed issues (median)	580	881	777
# open pull requests	2172	2392	4670
# open pull requests (median)	11	13.5	12
# closed pull requests	66,739	97,629	165,847
# closed pull requests (median)	272	429	348.5

Source: Made by the author

4.2 ERROR HANDLING IN JAVASCRIPT APPLICATIONS

In this section, we describe the analysis aimed to answer **RQ1** and **RQ2**.

4.2.1 Error handling abstractions

We investigate the mechanisms and abstractions have been used by the JavaScript community. The previous work focusing on Java and .NET of Cabral and Marques (2007) has considered the amount of source code by comparing the number of lines of code inside error handlers to the total number of lines of the program. We adopt this approach and complement it by also comparing the total number of lines for handling errors in each abstraction with the total number of lines for handling errors. The results are shown in Table 7. Developers reserve a small code percentage for handling errors. We found that 10.42% of LoC resides in callback functions that receive one or more error parameters. We need to highlight that those callback functions are not totally dedicated to error handling and may have statements inside the scope that aim to do something else besides error handling. In the other abstractions, we found only 1% of code residing in `catch` blocks (on `try-catch` mechanisms) and less than 1% residing in events handlers and promises. In these abstractions, we have a clear distinction of error handler and error scope.

Table 7 presents the result of this analysis in the column "% of handlers". We found that callback functions are used to handle errors in 60.2% of the handlers, and the second most employed abstraction is the `try-catch` blocks, which comprise 36.63% of the handlers. Promises, events, and `async-await` functions comprise less than 4% of the error handlers.

We also compare the amount of error handling code for each abstraction considering the total LoC of error handling. Callback functions comprise 93.16% of LoC in the error handlers. We reason this to the association of the code of error scope and the code error handler in the same block. Thus, this percentage includes code related to error scope, which is not intended to error handling. In order to check the existing of an error, developers can use an `if` statement. We can not ensure the intention behind the callback functions defined, the callback function may receive an error handling to both handle the

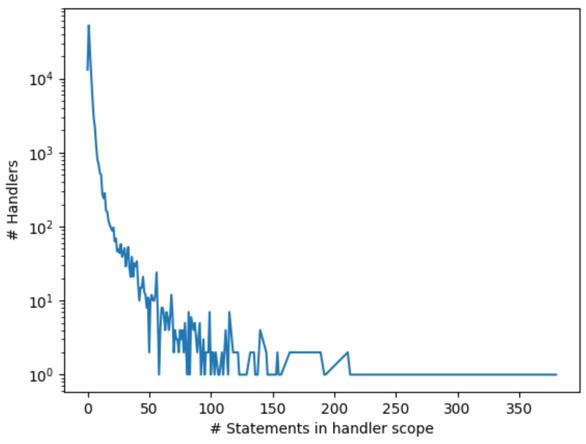
Table 7 – Percentage of EH abstractions by number of structures and number of handlers

EH abstraction	% by LoC	% by EH LoC	% of handlers	# of instances
pure callbacks	10.42%	93.16%	60,2%	61,678
pure try-catch blocks	0.69%	6.17%	36.63%	37,528
promises	0.045%	0.40%	1.92%	1,971
events	0.028%	0.25%	1.2%	1,230
async-await functions	0.0013%	0.012%	0.04%	49

Source: Made by the author

error and do something else, or the callback function may be defined intentionally for error handling only. Considering the other abstractions, which have a clear separation between the error scope and the error handler, we found only 0.69% of the error handling code, in number of lines, consists of `try-catch` blocks.

Figure 1 – Frequency of error handlers by number of statements within the handlers

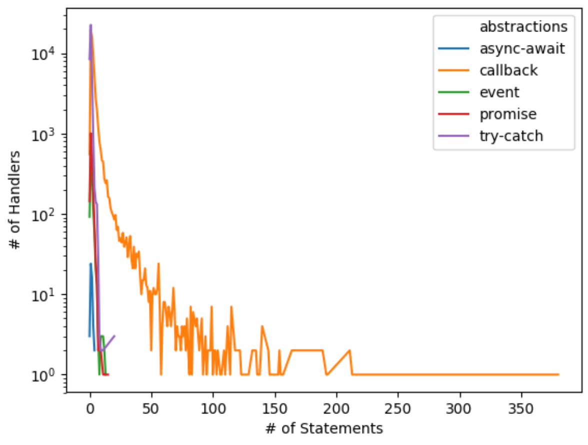


Source: Made by the author

Table 7 also shows the percentage of LoC of the analyzed projects that appear within error handlers. For all the abstractions, the percentage of LoC pertaining to error handling is less than 1% of the overall LoC of these projects, except for callback functions, which comprise more than 93.16%. Callback functions have an expressive percentage compared to the other abstractions because they usually have only part of their LoC dedicated to error handling. However, it is not possible to gauge the intention of the developers in an automated, and precise manner. Thus, we consider that, whenever a callback includes an error parameter, it is an error handling callback and accounted for it. About 10% of the LoC inside callbacks receive at least one error parameter.

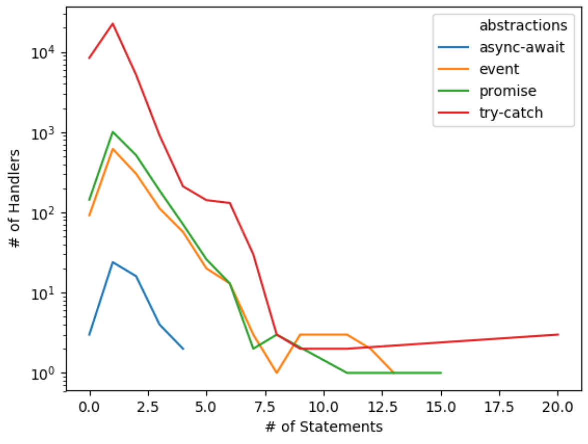
We also compare the number of statements that reside in the error handlers. The results are presented in Figure 1. Here, we highlight that most of the error handlers are very small (*i.e.*, with a small number of statements), regardless of the employed abstraction. More specifically, 51.64% of the error handlers (52,913 out of 102,456) have zero or one statement only. The third quartile is 111 and 90 percentile is 154. The largest error handler we identified has 380 statements. We also analyzed if this trend still maintain considering the abstractions. Figure 2 shows a comparison between the number of error handlers and the number of statements considering the abstraction. Regardless of the abstraction employed, the majority of the error handlers are small with regards to the number of statements. As the number of statements of callback functions are not totally dedicated to error handling, we also present in Figure 3 the frequency of error handlers and their respective number of statements removing the data related to error handlers in callback functions. A higher number of error handlers among the abstractions varies into 1 to 2 statements, even if we disregard callback functions in this analysis, error handlers are generic with regards to the number of statements.

Figure 2 – Frequency number of statements by abstraction



Source: Made by the author

Figure 3 – Frequency number of statements by abstraction removing error handlers from callback functions



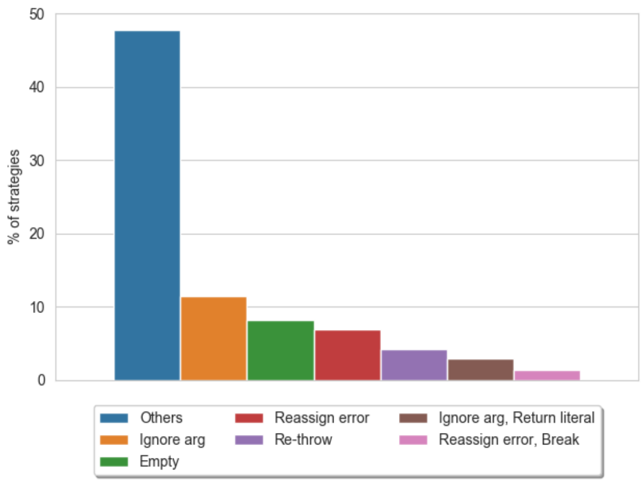
Source: Made by the author

Besides measuring the number of lines within error handlers, we compared the usage of error handling abstractions in the analyzed projects. We calculate the total number of handlers by error handling abstraction. This information is presented in the rightmost column of Table 7. Approximately 60% of all the error handlers are pure callbacks (not promises nor events). In the analyzed projects, there were on average 31 callbacks for each promise and 50 callbacks for each event. Overall, 36.62% of the error handlers are `try-catch` blocks. Almost all the usages of `try-catch` blocks employ the synchronous variety. The `async-await` abstraction is rarely used and so are `try-catch` blocks in that asynchronous context; only 0.1% of the `try-catch` blocks are associated with `async-await` and only 12 projects use this abstraction. We highlight that `async-await` has been added to Node.js only recently, in the 8th edition of ECMAScript, released in June 2017[2]. We believe this is the main reason for the low frequency of its usage.

4.2.2 Error handling strategies

Developers may use different strategies to handle errors. By *strategies* we mean the actions performed by the handlers in order to address the error. A handler may employ more than one strategy, for instance, logging an error and returning a literal value, such as an error code. We accounted for a number of different error handling strategies and also analyzed combinations of strategies as separate categories. We calculated the percentage of each combination of strategies. As the number of combinations of strategies is very large, and some of them do not comprehend a significant part of the handlers, we reported the percentages of combinations of strategies that comprehend more than 1% of the error handlers, shown in Figure 4. In overall, 11.5% of the handlers do not use the error arguments, 8.2% are empty, and 6.9% assign some other value to an error argument. When a handler does not use the error argument, it implements a generic combination of error handling strategies that ignores the type of the error caught. Finally, almost half the usages of error handling strategies, approximately 48%, employ other combinations of strategies.

Figure 4 – Error handling combination of strategies among the error handler abstractions

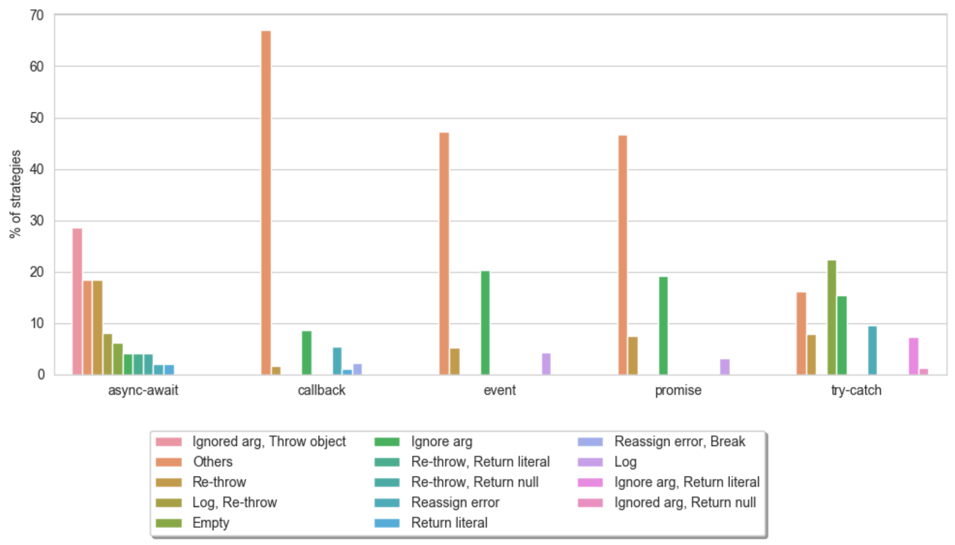


Source: Made by the author

[2]http://www.ecma-international.org/ecma-262/8.0/index.html

We also analyzed which strategies developers use when leveraging each error handling abstraction. We present the combination of strategies separated by error handling abstraction in Figure 5. We found a small number of error handlers, there are only 49 instances of error handlers that employ `async-await` abstraction in our dataset. Error handlers in callback functions present a percentage of almost 67% of other combination of strategies not foreseen. Furthermore, 8.66% of those callback functions completely ignore any error received as parameter, and 5.54% reassign an error parameter. Error handlers that use event abstractions present a scenario similar to promises. Among the 1,230 event error handlers, 582 (47.31%) exhibit varied combinations of strategies, 249 (20.24%) of them do not use the error argument, 64 (5.2%) just re-throw an error, and 53 (4.31%) print a message on the console. For error handlers defined through promise objects, the majority is more complex and use other combinations of strategies to handle errors (46.68%), 378 of them (19.17%) do not use the error argument, 148 (7.5%) re-throw an error, and 63 (3.19%) only print the the error on console. In `try-catch` blocks, most of them are either empty (22.44%), employ a generic error handling strategy that ignores the error argument (15.49%), or reassign an error from `catch` clause (9.67%). In general, the strategies are simplistic, *i.e.*, 8.22% are empty, 11.5% do not use error arguments, and 6.89% reassign an error parameter. Although, we found a high percentage (47.73%) of a complex strategy usage, which falls into "Others" category, this happens due to the statements found into error handlers defined by callback functions.

Figure 5 – Percentage of combination of strategies in error handlers



Source: Made by the author

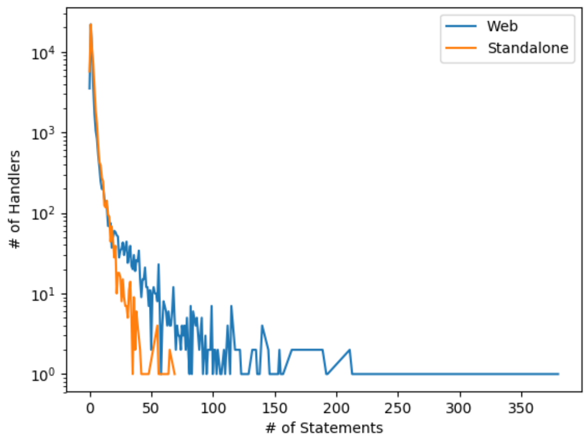
4.3 WEB-BASED AND STANDALONE SYSTEMS

In this section, we answer **RQ2** by identifying the main differences between two classes of javascript systems: Web-based and Standalone. We performed an analysis similar to the one presented in Section 4.2.

Firstly, we analyzed the error handling code for each class (Web-based or Standalone) based on the number of statements inside the error handlers. Figure 6 shows the frequency of error handlers and number of statements for each class. We do not differentiate the error handling abstraction for plotting this line plot. We found a higher number of handlers in the Standalone class. We calculate one sample for Web-based and one for Standalone project, based on the division of number of statements by number of error handlers. Then, we perform a Kolmogorov-Smirnov test for the null hypothesis that 2 independent samples are drawn from the same continuous distribution. We found a p-value of $1,57 \times 10^{-4}$, thus we should reject the hypothesis that both samples were drawn from the same distribution.

Boxplots for percentage of anonymous callback-accepting function callsites per category, across client/server, and in total.

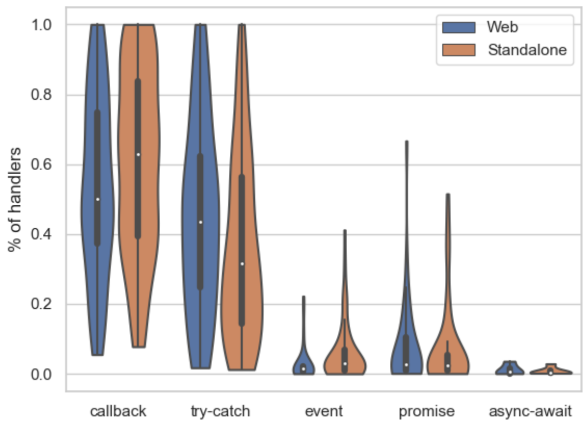
Figure 6 – Line plots for the number of statements and the number of handlers by class (Web-based or Standalone)



Source: Made by the author

To analyze error handling mechanism usage among the systems, we aggregate the error handlers by mechanism (or abstraction), class, and project, and then calculate the percentage of each error handling abstraction per repository. Figure 7 presents violin plots for each abstraction and class analyzed. The sample used to plot the violin plots are the percentage of error handlers of each abstraction per project. There is a prevalence of pure callback functions over `try-catch` blocks for error handling in projects classified as Standalone. Standalone systems present approximately two error handling callbacks for each `try-catch` blocks. Web-based systems make more balanced use of both abstractions, even though they also use callbacks more often. This result is consistent with the asynchronous, preferably non-blocking nature of operations in Standalone systems[3]. Although both numbers are small, the usage of events is more frequent than in Standalone systems. As for promises, it is well balanced.

Figure 7 – Violinplots for percentage of error handler in the repositories. Each data point corresponds to percentage of error handler in a project



Source: Made by the author

Additionally to identify which abstractions are most often employed in the projects, we analyze the strategies employed by the developers in those systems. Firstly, we measure the number of occurrences of each strategy within the error handlers. As aforementioned, an error handler may employed more than one strategy, for instance, it may log an error message, call a method to put the system into a consistent state, or throw the error. Therefore, this handler includes instances of three categories: Log, Others, and Throw. Previous work of Sena et al. (2016) considers the concept of handler action as the statements in the catch clauses responsible for performing any recovery action. Some of those handler actions were classified as a combination of actions that a handler may have. We classified the error handlers strategies in the same manner.

In order to increase the power of statistical test, we first analyze the data distribution to check if the data follows normal distribution. We perform a test based on the works D'Agostino (1971) and D'Agostino and Pearson (1973) that combines skew and kurtosis to produce an omnibus test of normality. Based on the results of this test, we perform a T-Student's test [4] when both samples follow a gaussian distribution and Mann-Whitney's test otherwise. Mann-Whitney[5] is a non-parametric statistical test for comparing two samples of different populations which do not need to follow a specific distribution. Be-

[3]https://nodejs.org/en/about/
[4]https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html
[5]https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.mannwhitneyu.html

sides the normality test, we perform homogeneity variance tests for equality of variances, whenever the samples follow a normal distribution. We used Bartlett's test[6] when the samples follow a normal distribution and Levene test[7] otherwise to test variance. In case the variance is different, instead of performing T-Student's test, we performed Welch's t-test.

Considering the strategies presented in Chapter 3, we calculate the existence of a specific strategy for each error handler. Then, we group the error handler strategies, and count the number of error handlers are found employing a combination of strategies in files and in projects. We perform the statistical tests twice, first keeping the zeroes observations and re-performed the tests removing the zeroes observations from the sample. These blank data are relative to files or projects that do not employ any error handler or do not employ the strategy. In all hypothesis tests, we consider a significance level of 5%. When we compared Web-based and Standalone projects, for empty `try-catch` blocks (p-value $= 9.315 \times 10^{-5}$), which means Standalone projects present a higher trend to have error handlers empty, re-throwing an error (p-value = 0.0026), and for reassignment of an error (p-value $= 1.337 \times 10^{-6}$). We found that Web-based projects tend to throw error objects, compared with Standalone projects, with p-value 8.6×10^{-3}. On callback functions, we found that Web-based projects tend to reassign an error compared with Standalone projects with probability value of 0.0245. For callback functions, we found the p-value of 0.02296 for a higher adoption of Web-based applications handlers rather than Standalone error handlers into ignore an error parameter. For promises, we found a similar result on probability value of 0.03377 that Web-based error handlers tend to ignore error parameters when compared with Standalone projects.

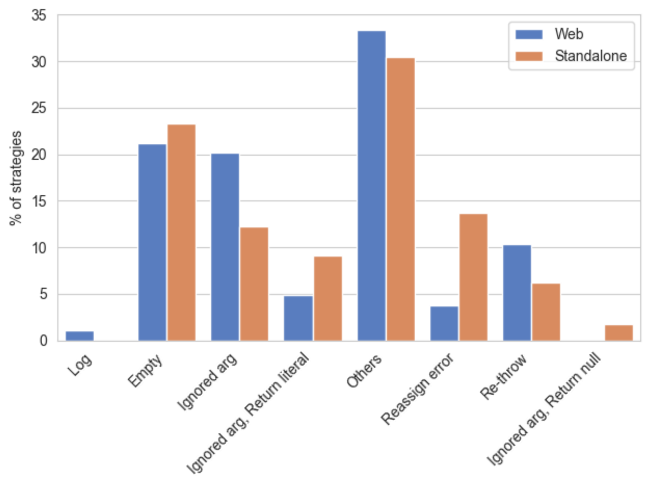
Figure 8 shows the combinations of strategies in the error handlers for each class present in `try-catch` blocks, per class (Web-based and Standalone). One of most representative combination of strategies of the `try-catch` blocks are empty for both classes (21.18% of the handlers for Web-based systems, and 23.3% for Standalone systems). Standalone systems are more likely to reassign an error parameter than Web-based systems. We found that 13.7% of Standalone systems change error parameters to other values, while only 3.8% of the error handlers in Web-based systems apply this strategy.

Async-await functions have not been widely adopted in the JavaScript community. This abstraction has a low number of error handler in both classes, and approximately four out of every five `async-await` handlers appear in Standalone systems. We believe that it may still be too early to study error handling strategies for this abstraction in more depth.

Differently from `async-await` functions, callback functions are pervasive to JavaScript systems. Figure 9 shows the percentage of the combination of strategies of the classes in

[6]https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.bartlett.html
[7]https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.levene.html

Figure 8 – Percentage of handlers that implement some type of handler action combination in `try-catch` blocks



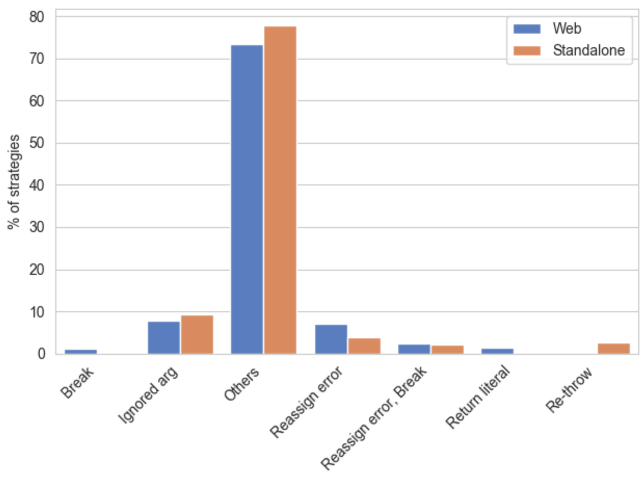
Source: Made by the author

callback functions. Although callbacks are used more often in Standalone systems, there is not much difference between Standalone and Web-based systems regarding how often they employ each error handling strategy. Besides employing a strategy that is outside of the ones listed before, Standalone systems has more error handlers (9.36%) that ignore an error than Web-based systems (7.93%). Similarly to what we observed for `try-catch` handlers, the greatest difference can be observed for handlers that reassign error parameters. However, conversely to what `try-catch` error handlers present, for callback functions, Web-based systems reassign error parameters more often: 7.2% of error handlers whereas 3.94% of the handlers do the same in Standalone systems. This trend can still be observed when also accounting for instances of the Reassign error (p-value: 0.0245) and break (p-value: 0.032) strategies combined.

We found that error handlers implemented as event handlers are more common in Standalone systems (996 handlers) than in Web-based systems (234 handlers). Figure 10 shows the barplots of combination of strategies for event handlers. Despite of this difference, the distribution of the strategies across the handlers follow similar trends. For example, 25.64% of the handlers in Web-based systems do not even use any error parameter and 18.98% of Standalone systems ignore the error parameter too (this result do not include parameterless error handlers). We also found that 6.41% of the handlers of Web-based systems just re-`throw` an error whereas 4.92% of handlers in Standalone systems do the same. `Throw`ing an error from an error event handler triggers the publication of the `uncaughtException`, which, if not captured, causes the system to crash.

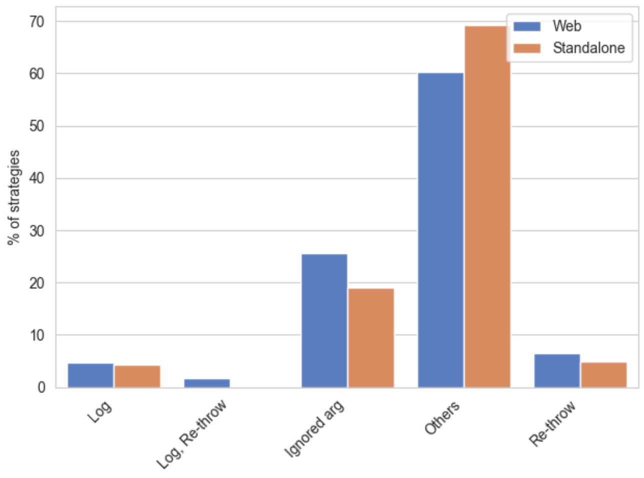
Figure 11 shows the percentages for strategies in promises. For those error handlers we found that 274 out of 1,007 (27.2%) handlers in Web-based systems and 104 out of 964 (10.79%) handlers in Standalone systems just ignore the error parameter. This is a

Figure 9 – Percentage of handlers that implement some type of handler action combination in callback functions



Source: Made by the author

Figure 10 – Percentage of handlers that implement some type of handler action combination in events



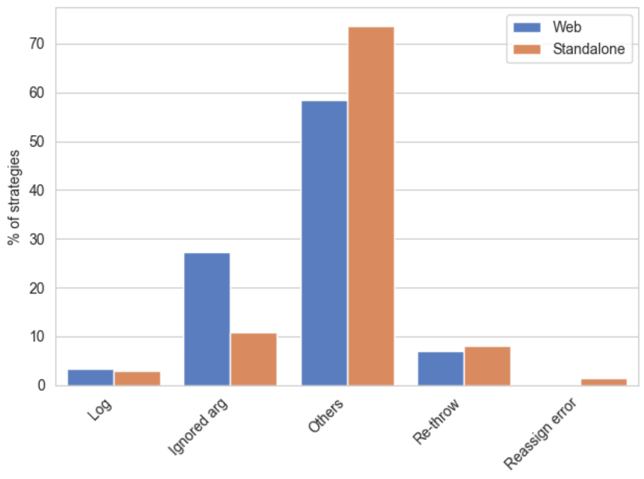
Source: Made by the author

considerable difference and suggests that promises in Web-based systems are more generic: developers are worried about the occurrence of an error but not about what kind of error occurred. We also found that 6.95% of error handlers in Web-based systems and 8.09% of error handlers in Standalone systems rethrow an error from a `catch` method inside a promise. This means that the next promise in the chain will be in a rejected state, thus, precluding it from executing. Further investigation is necessary to understand the reasons why developers throw the error instead of calling the `reject` method.

Error-first callbacks[8] are callbacks whose first parameter is an object representing an

[8]http://callbackhell.com

Figure 11 – Percentage of handlers that implement some type of handler action combination in promises



Source: Made by the author

Table 8 – Dispersion of error handling abstractions among the projects

EH abstraction	# of projects	# of handlers
callbacks	171	61,678
try-catch	161	37,528
promises	82	1,971
events	90	1,230
async-await	10	49

Source: Made by the author

error. This is an important convention because it encourages developers to think about error signaling and handling whenever they write callbacks. In our analysis of the JavaScript projects, we found that 50,806 out of the 57,293 (88.7%) callback functions that have an error parameter follow the error-first protocol. They are predominantly employed in Standalone systems, the percentage of error handlers among the systems are 81.8% in mean, while the percentage in Web-based systems are 68.15% in mean.

We investigate the dispersion of the error handling considering the error handling abstraction and the projects. Table 8 shows the number of projects which error handling abstractions appears and the number of error handlers in the error handling abstractions. We notice that the number of callback functions and `try-catch` is the highest among the error handling abstractions, they are also scattered in the 179 projects. Callback functions and `try-catch` are employed in 171 and 161 projects respectively. Promises and events are employed in almost half of the projects and async-await appears in only 10 projects. Table 9 also presents the dispersion of the errors handlers among the projects, however

Table 9 – Dispersion of error handling abstractions among the projects per class.

EH abstraction	# of projects (Web-based)	# of projects (Standalone)	# of handlers (Web-based)	# of handlers (Standalone)
callbacks	75	96	30,200	31,478
try-catch	76	85	15,289	22,239
promises	35	47	1,007	964
events	30	60	234	996
async-await	4	6	8	41

Source: Made by the author.

considering which class they were pertain. We found that error handlers appear in more frequently in Standalone projects.

5 DISCUSSIONS

In this chapter, we discuss the results presented in this dissertation. In the Section 5.1, we discuss error handling mechanisms usage. In the next two sections, we discuss about how the projects in the dataset adhere to the recommendations (Section 5.2) and antipatterns (Section 5.3), presented in in the Chapter 3 (Section 3.6).

5.1 USAGE OF ERROR HANDLING MECHANISMS

In this section, we discuss about the error handling mechanisms (more specifically, about the mechanisms and their abstractions). We analyze the proportions of LoC and statements against the number of error handlers for each of the abstractions (pure callback functions, pure `try-catch` blocks, async-await functions, promises and events). Table 10 shows the proportions of both LoC and statements per error handler. The proportions are much smaller for `try-catch` blocks (1.706 and 0.939) than for the other abstractions, *e.g.*, for promises, these proportions are 2.433 and 1.605 and for events 2.455 and 1.723. We notice that pure `try-catch` blocks seem to be second-class citizens among the error handling abstractions, although they are available in JavaScript since its initial version.

The JavaScript programming culture is highly reliant on callback functions, where errors are often not signaled by means of exceptions and thus, cannot be handled with pure `try-catch` blocks. This applies to both Web-based and Standalone applications. Notwithstanding, 42.53% of the handlers are pure `try-catch` blocks, which means that they are commonly employed. Furthermore, as mentioned before, `try-catch` blocks are part of the language since its inception. Thus, developers are used to employ this abstraction and it is not a matter of lack of familiarity.

Among the guides we analyzed, five recommended the use of promises of async-await functions instead of pure callbacks, in particular in the cases where errors may occur. This recommendation enforces a stronger separation between the error scope and the handler scope, since promises have a specific function for handling errors (`catch`) and async-await functions employ `try-catch` blocks. Arguably, this leads to better separation of concerns (Lee and Anderson (1990), Parnas and Würges (1976)) and improved readability. Pure callback functions do not delimit error scope and handler scope, *i.e.*, the handler scope statements are generally blended within the scope of the callback function. This forces developers to adopt an ad-hoc solution, such as the error-first idiom or even not deal with the error at all.

In spite of this recommendation and its potential benefits, we found only moderate adoption among the analyzed systems. Promises amount to 0.4% of all the error handling lines of code of the analyzed systems (Section 4.2.1, Table 7), whereas `try-catch`

Table 10 – Number of LoC and statement per error handler

EH abstraction	LoC per handler	Statements per handler
pure callback	-	-
pure try-catch	1.706	0.939
promise	2.433	1.605
event	2.455	1.723
async-await	-	-

Source: Made by the author

Table 11 – Number of handlers by abstraction per project

EH abstraction	Min	Max	Mean	Median	% by EH LoC
pure callback	1	13059	360.69	47	93.16%
pure try-catch	1	11709	233.09	38	6.17%
promise	1	306	24.037	6	0.40%
event	1	231	13.66	6	0.25%
async-await	1	26	4.9	2	0.012%

Source: Made by the author

blocks within `async-await` functions comprise only 0.001%. For the sake of comparison, `try-catch` blocks not appearing in `async-await` functions account for 6.17% of all the error handling lines of code. It can be argued that these features are not in widespread use because they were introduced in JavaScript only a few years ago, *i.e.*, promises only became part of JavaScript in 2015 (ECMAScript 6) and `async-await` in 2017 (ECMAScript 8). However, the former has seen wider adoption than events, which are available in Node.js since 2011 (version 0.1.26).

Table 11 shows statistics about the error handlers by abstraction per project. We analyze the mean of percentage of error handling abstractions per project. Callback functions and `try-catch` blocks are the most employed abstractions. The projects present a mean of 360.69 pure callback functions, and a mean of 233.09 pure `try-catch` blocks. The number of callback functions and `try-catch` blocks are not higher considering the median of both samples for those error handling abstractions.

5.2 RECOMMENDATIONS

In this section, we analyze how error handlers practices adheres to recommendations that we describe in Section 3.6.

Table 12 – Percentage of handlers throwing error as error object

EH abstraction	Error object usage (%)	Error object usage ignoring error (%)
pure callback	0.56%	0.07%
pure try-catch	0.02%	0.28%
promise	0.20%	0.1%
event	0%	0.33%
async-await	0%	28.57%

Source: Made by the author.

5.2.1 Error Object Usage

In this strategy, we evaluate the usage of throwing errors. Whenever a developer decides to throw[1] an error, the execution is suspended for the current function, and the control flow changes for the next handler. The program will terminate if a handler is not found. The syntax of JavaScript for throwing an error is "throw expression", such expression may be any type (as claimed by Mozilla documentation [2]), like a string, an integer, object or even the built-in Error object. However, it should be avoided to use any other type besides Error object, as the result will not include information about the call stack, neither the property "name" and other properties that describe the error. Throwing an error inside a callback function is considered a "mistake" as this mechanism is asynchronous. It was designed to be called in a point in the future and, thus, even if a developer wraps a snippet code using `try-catch` blocks, it is unable to handle any errors that occur on it.

Firstly, we analyze handlers not rethrowing any error parameter. We find 659 out of 61.678 callback functions that throw only an error object or apply another strategy together, representing 1.07% of callback functions that handle errors. Among those 659 handlers, 348 of them throws an error object only. This may lead the system to a unstable state. We need to study even further the impact those handlers may have in the systems. In general, a small percentage of handlers throws errors objects for all abstractions. Table 12 presents the percentage of handlers that throws error, and less than 1% of the handlers in all EH abstractions employ this strategy. The only deviation of this scenario is async-await, which we found 28.57% of the handlers throw errors ignoring any error parameter. This means the callback functions do not generally throw errors.

Handlers could also re-throw an error after caught it, propagating it up to another handler to treat it. We found 10.4% of `try-catch` blocks that only re-throws an error in Web-based applications, and 6.23% on Standalone applications.

[1]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw
[2]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw

5.2.2 Error-first protocol

Error-first protocol is defined by the work of Gallaba, Mesbah and Beschastnikh (2015) as callback functions which their first argument represents the occurrence of an error, if it is non-`null` or the absence thereof in case it is `null`. They analyzed callback functions in 138 JavaScript systems, and they found 20% of callback functions adopt the error-first protocol. They also claim that error-first callbacks are not widely employed.

Gallaba, Mesbah and Beschastnikh (2015) also claimed that the error-first protocol is similar to the solutions employed in languages that do not include dedicated error handling mechanisms, such as C. According to Bruntink, Deursen and Tourwé (2006), some of the disadvantages of C's approach for error signaling and handling are the following: (i) when a function needs to return a value, it must be through a parameter, instead of the function's exit point; (ii) there is no clear convention for what it means to be successful or not, *e.g.*, sometimes 0 means success, sometimes it is a failure; (iii) it is necessary to check what the method returns for every call, which may lead to chained checks when there are multiple calls; (iv) since an error is an integer, little information about the error is available for error handlers.

For JavaScript, however, one could argue that the problems are not the same. In (i), both errors and correct results are reported through parameters in callback functions. The (ii) is arguably a problem, as both kinds of results are reported in the same manner, unlike of what usually happens in C. Furthermore, the error-first protocol is widely known, even if not widely used (Gallaba, Mesbah and Beschastnikh (2015)). For example, the official Node.js documentation for the `Error` object starts out by discussing error-first callbacks[3]. In (iii), it can actually occur in JavaScript and stems from the well-known "callback hell" problem Alimadadi, Mesbah and Pattabiraman (2016). Callback hell occurs when requests rely on the result of previous requests turning these callbacks profoundly nested Philips et al. (2016). Empirical studies considered the impact of callback hell in JavaScript applications in different contexts Kambona, Boix and Meuter (2013), Fukuda and Leger (2015). Nonetheless, the language includes mechanisms to avoid this problem, namely, promises and `async-await` functions, and, as pointed out in Section 4.2.1, guides on JavaScript error handling actively recommend the use of these approaches. Finally, in (iv), an error in JavaScript is an object and, as a consequence, can convey more information than a single integer.

Although Gallaba, Mesbah and Beschastnikh (2015) has not analyzed Node projects specifically, and used a different classification in the JavaScript systems, we present their results, to introduce the reader to this topic. In our work, 30% of all the callback functions in Standalone systems employed this approach, whereas only 16% of the Web-based systems did so. We observed that error-first protocol is highly employed in callback functions to handle error. We calculate the percentage of error-first callbacks for each project in

[3]https://nodejs.org/api/errors.html

Web-based and Standalone. The percentage in Web-based systems are 68.14% and 81.77% in Standalone systems. Thus, there is a trend for Standalone systems adopt more often this protocol comparing to Web-based systems.

The bottom line of this discussion is that tools for analyzing problems in JavaScript error handling cannot start from the assumption raised by previous work in error handling that the problems of the error-first protocol are the same as those found in C. Furthermore, regardless of the aforementioned issues, the use of pure callback for error handling, whether employing the error-first protocol or not, incurs in the problem of not promoting separation of concerns between error scope and handler scope. This negates one of the fundamental benefits of using an error handling mechanism (LEE; ANDERSON, 1990; PARNAS; WÜRGES, 1976).

5.2.3 Log error

Log error information is useful during the system lifetime, specially for software instrumentation. For this reason, there are some libraries with this purpose such as: *morgan* (1.212.616 weekly downloads on NPM), *loglevel* (2.370.906 weekly downloads on NPM), *winston* (2.472.945 weekly downloads on NPM), and others. This strategy allows developers to code trace and debug major events in the execution of the application.

Kery, Goues and Myers (2016) analyzed Java projects and found that print statements and log methods represent 10% each of the error handlers. Cabral and Marques (2007) claimed that logging is one of the most common actions (in our work called strategies). It suggests that programmers do not have a specific approach for error handling, just registering the error, and notifying the user.

Table 13 shows the percentage of `Log error` strategy for the error handling abstractions that we analyzed. In our analysis, considering the usage of "console.log" method in JavaScript applications on among the abstractions evaluated, the usage is not common as in Object-oriented programming languages as Java and C#, presented by Cabral and Marques (2007) and Kery, Goues and Myers (2016). Async-await functions presented the highest percentage (8.16%) for logging a message and rethrowing an error parameter, that represents only 4 catch blocks of 49, which makes difficult to generalize this as a trend for the abstraction, once it is not representative. Event handlers and promises present a keen percentage for logging errors, 4.3% and 3.2%, respectively.

We found that less than 1% of `try-catch` blocks and callback functions call "console.log" method only.

5.2.4 Return error

One of the practices recommended by the tutorials is to return an error through a callback. The lack of this strategy in callback functions, it leads to an error to be swallowed for the

Table 13 – Percentage of error handlers using log error as strategy

EH abstraction	Log error (%)
pure callback	0.78%
pure try-catch	0.76%
promise	3.12%
event	4.3%
async-await	0%

Source: Made by the author

Table 14 – Percentage of error handlers using Return error as strategy

EH abstraction	Return error (%)	Return literal (%)
pure callback	0.79%	0.24%
pure try-catch	0.76%	7.39%
promise	3.2%	1.01%
event	4.31%	0.08%
async-await	0%	2.04%

Source: Made by the author

application, preventing the developers on identifying the error occurred due to the lack of indication of its occurrence.

Table 14 shows the percentage of error handlers that return an error parameter and return an literal (like string, number, undefined or null values). We found only 448 out of 30,200 callback functions, which represents approximately 1.5% of the callbacks. When we analyze error handlers that return an error parameter or an literal, we notice an approximately 1% of the pure callback functions employs these strategy only. This indicates a small adoption in returning inside callback functions. `Try-catch` block is the abstraction that most uses the strategy of `return` inside error handler scope. On 2,772 `try-catch` blocks, 7.39% of them return a literal, and 378 `try-catch` blocks (1.7%) return `null`. The adoption of returning on callback functions is low for Web-based compared to Standalone projects.

5.3 ANTIPATTERNS

In this section, we discuss how error handlers adheres into the antipatterns presented in Chapter 3 (Section 3.6).

5.3.1 Empty Handler

Empty handlers means that an error is caught in the application and there is no code to handle it. For callback functions, we consider that a callback function is empty when follows one of the scenarios: (i) it was declared as an empty block through a function declaration using this structure (*i.e.*, function(){}); (ii) it was not defined at all, when a function whose a parameter is a callback function may omit this callback functions leave this value to "undefined", and (iii) it calls a function inside the callback function (proposed to handle an error) and the scope of that function is empty. Recalling our methodology, we retrieve the callback functions that dwell in (i). In order to leave a callback function empty, a developer must deliberately define a callback function, and leave it empty. We were not able to detect the cases (ii) and (iii).

Table 15 shows the percentage of error handlers that are empty by error handling abstraction. We found a high number of empty `try-catch` blocks (approximately 22.44%). We found a number of empty catch blocks in async-await functions (6.12%), and did not find any callback function that were empty (either pure callbacks, event handlers, or promise objects), since these error handlers would hardly be created. Arguably, the absence of empty callback functions arise from the nonexistence of syntactic separation between error handling and non-error handling code within a callback. Even if the callback does not handle errors, it is expected to do something else besides error handling.

`Try-catch` blocks are more often empty, reassign error and rethrow in Web-based projects than in Standalone projects, p-values: $9,3 \times 10^{-5}$, $1,34 \times 10^{-}6$ and $2,6 \times 10^{-3}$ respectively.

The study of Cabral and Marques (2007) found a lower number of empty handlers in Java rather than C#. They suggested that checked exceptions may impact less on the decision of leave a handler empty, and the developers may leave empty handlers due to use error handling mechanisms for control/execution flow besides error handling. They also stated that sometimes Java API forces this when a detection of EOF (end of file) must be done through throwing an exception. JavaScript do not enforce any type of error handling usage as in Java and C#. Kery, Goues and Myers (2016) claimed that there are situations that empty catches are applicable to the program logic, however they acknowledge this is considered a bad practice.

In JavaScript applications, we found that 8,424 error handlers (8.22% of all error handlers regardless of the error abstraction employed) are empty and all of them are inside `try-catch` blocks or async-await functions. We found 3,238 empty handlers on `try-catch` (21.17%) exists in Web-based systems and 5,183 empty handlers on `try-catch` (23.31%) exists in Standalone systems. In async-await functions, we found approximately 4.88% of error handlers are empty in Standalone systems and 12.5% in Web-based systems, they are not statistically significant as they represent only 3 handlers out of 49 error handlers.

Table 15 – Percentage of empty error handlers

EH abstraction	Empty error (%)
pure callback	0%
pure try-catch	22.44%
promise	0%
event	0%
async-await	6.12%

Source: Made by the author

We found that less than 1% of `try-catch` blocks and callback functions call "console.log" method only.

5.3.2 Ignore error

Developers may completely ignore the error information. The consequences for this approach interfere directly on the state of the system. Wherever an error occurs in a point of the system, the developers cannot ensure the remaining of the execution is stable. Additionally, this specific error will not be added into the stack trace, possibly making the identification of the problem in a debugging process more difficult.

We found a high percentage of handlers that just ignore an error in all abstractions. Specifically, approximately 11.5% of all handlers (regardless the abstraction employed) neglect error parameters and take a generic strategy in occurrence of the error. By generic strategy, we consider the definition of the work of Cassee et al. (2018): error handler that "capture any kind of error". When we took the abstraction employed into account, at least 5% of the handlers ignore error information for each abstraction. Table 16 shows the percentage of error handlers by error handling abstraction. Pure callback functions exhibit 8.66% of handlers ignoring errors, and promises present about 19.18% of the catch methods ignore errors. Nearly 20.24% event handlers employed for error handling ignore error parameters. We found more than 4% of handlers of the async-await functions that neglect errors, however this represents only 2 handlers out of 49.

We also compared the usage of Ignore errors strategy between `try-catch` blocks and callback functions. Aggregating data by file in the repositories, we found that at 5% of confidence interval, developers are less likely to ignore errors in pure callbacks than when using `try-catch` blocks (p-value $= 1.382 \times 10^{-21}$). Even if we consider the other abstractions (events and promises as they also apply callback functions as an error handling mechanism) and async-await functions, we found that at 5% of confidence interval, callbacks are less likely to neglect errors than in `try-catch` blocks (p-value of 2.513×10^{-28}).

Table 16 – Percentage of Ignore error on error handlers

EH abstraction	Ignore error (%)
pure callback	8.66%
pure try-catch	15.48%
promise	19.18%
event	20.24%
async-await	4.08%

Source: Made by the author

Although we are calling ignoring errors as a strategy, it is not directly attached to a specific structure or statement (as `throw` statement for instance). We claimed that based on error handlers that although are not empty, do not use any of the error parameters (either parameters of the `catch` clauses or of the callback functions). We found 2,394 callback functions (on 30,200 handlers) that ignore errors in Web-based projects, and 2,946 (on 31,478 handlers) in Standalone projects. They represent 7.93% and 9.36% respectively. Even though `try-catch` blocks presents a less number of handlers in any class, the percentage of handlers that neglect the error parameter are higher. We found 3,090 in 15289 handlers (20.21%) in Web-based, and 2,719 in 22,239 handlers (12.23%) in Standalone projects.

By default, ESLint has a rule that disallow the definition of empty block statements[4]. To make clear this behaviour is intentionally, the tool recommends to leave a comment. JSLint[5] introduces a new reserved word: ignore. Whenever catch clause receives a parameter called "ignore", it indicates that a catch block may be empty without raise warnings.

5.3.3 Reassign error

Error parameters could be passed to code handlers directly by developers or detected by the engine or the library. Reassign an error parameter may lead the system to a problematic behaviour by changing the error raised. In JavaScript applications, a built-in variable called *arguments* is an array-like object which stores function parameters. Reassigning parameters also changes directly the behaviour of arguments variable. ESLint has two rules that fall into in this category: no-param-reassign[6] and no-global-assign[7]. The first rule aims at checking when a function parameter is assigned, and according to its documentation, this assignment is unintended and indicates a mistake or a programmer error. Side effects on parameters are counterintuitive and could make detection of errors more difficult. The second rule is more specific, and is also related to reassign of

[4]https://eslint.org/docs/rules/no-empty
[5]http://www.jslint.com/help.html
[6]https://eslint.org/docs/rules/no-param-reassign
[7]https://eslint.org/docs/rules/no-global-assign

Table 17 – Percentage of Reassign error on error handlers

EH abstraction	Reassign error (%)
pure callback	5.54%
pure try-catch	9.67%
promise	0.96%
event	0%
async-await	2.04%

Source: Made by the author

variables, it is related to assignment of JavaScript built-in environment variables, for example `window` in browsers, and `process` in Node.js applications. Loss of some functionally happens when those type of variables are reassigned and there is no native mechanism to prevent reassignment of these variables environment. We would like to highlight that even the possibility of handling unforeseen errors raised in the application through `window` or `process` object is not empowered by the reassignment of those objects.

Table 17 shows the percentage of error handlers by abstraction that reassign an error parameter. We found 9.67% of `try-catch` blocks reassigning at least one of the catch block parameters, and 5.54% of callback functions reassigning a parameter. Promises and async-await functions apply this strategy in a less percentage of 0.96% and 2.04% respectively. We did not find a single error handler using event abstraction that reassign a variable. This suggests a high usage of this strategy by `try-catch` blocks, and non-representative adoption of this strategy in asynchronous abstractions.

5.3.4 Throw error

Sometimes an error may bubble up to the stack and there is no handler to deal with it. `Throw` an error can crash the Node.js process due to asynchrony of callback functions. According to Node.js documentation, a common mistake of new developers is the employment of throw inside an error-first callback. `Try-catch` mechanism cannot properly intercept errors in asynchronous APIs, besides it relies inside an async-function[8]. Global event handlers are applied either for Web-based and Standalone systems to intercept these errors in JavaScript applications. For Node applications, there are two options available: a domain can be created (which is currently deprecated) or to register an event handler for `uncaughtException` through method `on`. In JavaScript applications, there are three options: (i) on object `window` by calling methods `onerror`, (ii) still on object `window`, calling `addEventListener` method, and (iii) on object `element` by calling `onerror`.

In our dataset, we found 32.81% of the projects (63 out of 192 repositories) that do not employ any global handler at all and present at least one callback functions that

[8]https://nodejs.org/api/errors.html

throws an exception. Any unhandled error in the project will not be caught, and the error is printed into the console (when an application is running in a web browser) or shutdown the Node.js project. The projects that may fail to recover from errors, as do not employ any global handler, consists in mean of almost 70k lines (minimum number of lines is 200 and maximum of 900711), and 388.2 files in mean (minimum number of files is 1 and maximum of 7782). From the 63 projects, 23 projects are Web-based classified and 40 projects are Standalone. The number of projects that neglect the usage of global event handlers in Standalone projects is higher that the number of Web-based projects, we calculate the number of projects that neglect the usage of global event handlers by the number of projects in a specific class. We found the value of 0.267 for Web-based projects and 0.377 for Standalone. Therefore, Standalone applications tend to neglect the global event handlers usage.

As definition of a global handler, it does not need to be created twice, however we found 23 projects that create a global event handler more than once. The majority of the projects has less than ten global event handlers, and the highest numbers of global event handlers found are 16, 32 and 51, corresponding to the projects with the highest number of lines (172138, 1137763, 932438 lines, respectively) and files (691, 6483, 9201 files, respectively). We reason this to the projects in Github may include more than one module in the same repository, and this need further investigation, as this approach may represent software engineering practices in specific projects.

According to the Node.js documentation[9], `uncaughtException` handlers were designed to perform asynchronous tasks for deallocate resources, such as file descriptors, handlers, etc, and shutdown the system afterwards. It is not recommended to resume normal operation after “uncaughtException” event.

We calculate the number of projects that do not correspond to the event global handler from its class. We find 127 projects that has at least one event handler, and have no global event handler specific for their class. That means the possibility of finding an *uncaughtException* handler in Web-based projects, and handlers from window object on Standalone projects. In total, we found nine projects in this situation, six projects are Web-based and three are Standalone. Although *uncaughtException* handlers are used by Standalone applications, we found six Web-based projects that have *uncaughtException* handler, and nine Standalone projects that use global event handler instead of *uncaughtException* handlers. In fact, there are six repositories in Web-based applications which have only *process.on(‘uncaughtException’)* handlers and three repositories in Standalone applications which do not have a single handler to uncaughtException event.

[9]https://nodejs.org/api/process.html#process_event_uncaughtexception

6 CONCLUSIONS

This chapter concludes the investigation of JavaScript error handling and approaches employed by developers to cope with errors. We perform an empirical study of popular JavaScript repositories from Github comprising of more than 60 thousand files and 11 million LoC. To the best of our knowledge, this is the first investigation of error handling mechanisms in JavaScript.

We found out that callbacks are the predominant error handling mechanism of JS systems (64.5k callback functions in our dataset), although `try-catch` blocks are also frequently used (51.2k `try-catch` blocks). Some well-known error handling anti-patterns observed in other languages (Cabral and Marques (2007), Cassee et al. (2018)) are commonplace in JavaScript error handling. Among the `try-catch` blocks, 22.44% are empty and 49.7% have a single line of code. Moreover, JavaScript is a dynamic programming language, error parameters are not forced to be of a specific type, leading `catch` blocks to be generic, in terms of the typification of error objects applied on error handling. Generic `catch` blocks can lead to errors being captured accidentally (Robillard and Murphy (2003)). In callback functions, only 0.88% are empty and 1.7% of the error handlers employed for error handling just ignore the error argument at all. These results highlight an opportunity to improve existing static analysis tools for JavaScript since, to the best of our knowledge, none of the more well-known static analysis tools for JavaScript[1] include rules to detect these anti-patterns.

Our results bring us to the conclusion that the error handling is a topic generally neglected by developers and JavaScript community (considering the dataset we analyze) as most of the errors handlers are empty (details in Section 5.3.1), ignore (details in Section 5.3.2) or reassign an error parameter (details in Section 5.3.3). A large proportion (**22.44**%) of the `try-catch` blocks is empty. For `async-await` functions, the percentage is **6.12**%. On the other hand, no error handler implemented in pure callbacks, promises, and event handlers was empty. When we consider cases where the handler ignores its error parameter, something akin to an empty catch block in the context of pure callbacks, the percentages are 8.66% for pure callbacks, 19.18% for promises, and 20.24% for event handlers. In contrast, 15.8% of the `try-catch` blocks make no use of their error parameter. For error handlers that deal with asynchronism, those who ignore some error parameter appear frequently in Standalone projects than in Web-based projects, with the exception of promises, which in Standalone projects show a difference of 16.41%.

We highlight a high adoption of callback functions for asynchronous execution (Section 5.1), although we found a high number of guides (details in Section 3.4) that recommend the employment of promises and async-await functions in these cases.

[1]https://hackernoon.com/the-ultimate-list-of-javascript-tools-e0a5351b98e3

6.1 IMPLICATIONS

The results of this study present that developers employ, in general, mechanisms and simplistic strategies for error handling that are not recommended by the own JavaScript community.

Developers: we believe that our findings could function as guidelines. As we present in Chapter 5 (Section 5.1), we found that callback functions, even though is a construct of the language (not specifically designed to error handling) is often employed instead of `try-catch` blocks. One of the recommendations (of the own community) is to employ async-await or promises to handle errors. We also found that `try-catch` blocks are more often empty, reassign error and rethrow in Web-based projects than in Standalone projects. It suggests that developers should aware about the error handling strategies they are employing to a specific type of system they are developing.

Researchers: we think that this work motivates researchers to have a better understanding of the employment of error handling mechanisms on JavaScript systems. We believe that this work starts research studies in a deeper way on JavaScript error handling, as it considers other constructs of the language (callback functions), going further the `try-catch` blocks, as the current research studies do. An example of the usage of different constructions for error handling in JavaScript is the usage of generators[2] and observable objects[3]. Specially because these two concepts are being used in popular frameworks as Angular, React.js, Vue and others (Voutilainen (2017)).

Tool Developers: our study will inform the design of future JavaScript analysis and code comprehension on static analysis tools. We believe the results of our study may help on the construction of better tools for development of JavaScript programs. Besides the ESLint rules that we have analyzed, developers could add new rules to static analysis tools the recommend (or even force the developers of the applications) as the Async-await/Promise usage, Error object usage Log error and Return error for callback functions.

6.2 FUTURE WORK

We list our future work intentions as follows:

- Apply improvements in the extraction tool:

[2]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*
[3]https://www.oreilly.com/library/view/learning-javascript-design/9781449334840/ch09s05.html

Callback functions: we considered a limitation on the retrieval of the callback functions designed to handle errors, only if it is directly defined on the callee function. This may mean that we miss some of the callback functions designed to handle errors, but we were currently unable to retrieve callback functions defined in another place of the application. In Code 14, we present a version of the Code 4. The function named `removeAnProduct` could be defined in the same script as the calling of `product.remove` or in any other JS files of the system. We plan to investigate how to identify such cases, to reach the callback functions not covered by the methodology we used.

Listing 14 – An example of callback usage to handle an error

```
product.remove(removeAnProduct);

function removeAnProduct(err, removedProd) {
  if (err) return handleError(err);
  console.log(removedProd._id + " removed.")
}
```

Event handlers: we also plan to investigate how to identify the event handlers more precisely. In this work, we have a limitation in identify error event handlers, and we decide adopt only the nomenclature of event handlers in Node. As JavaScript libraries define the API, with no common interface, to properly identify error events handlers. This impacts directly on the total number of error handlers and the strategies error handlers that uses events employs. We aim to investigate how to retrieve event handlers more properly, considering the specificities of the API of the libraries of the applications of our dataset.

- Extend the study regarding the impacts of the current approach for error handling of JavaScript community:

Analysis of JS code inside HTML files: Some JavaScript code may be found inside HTML files, as firstly we did not analyzed those source code, we plan to investigate the impact of those code in the results we presented in this dissertation in the future.

Error handling bugs: another future work derived from this study is in how the usage of specific error handling mechanisms might impact systems with regards to bugs, due to existence of antipatterns employed in the error handlers. Additionally, another investigation is about software bugs impacts the classes Web-based and Standalone.

Web-based and Standalone strategies: Although the use of strategies considered as antipatterns for both classes (Web-based and Standalone) is high, we highlight that the motivation of the systems may impact on how applications handle errors (as Gallaba et al. (2017) did categorizing the error handlers), but it is need more investigation.

Async-await usage: we noticed a high recommendation from the guides to adopt the async-await abstraction to solve both asynchrony and error handling issues in one structure. We found a very small adoption of async-await functions from developers of either async-await functions and promises in the systems. We plan to investigate the reasons behind this trend.

REFERENCES

ALIMADADI, S.; MESBAH, A.; PATTABIRAMAN, K. Understanding asynchronous interactions in full-stack javascript. In: *Proceedings of the 38th International Conference on Software Engineering*. New York, NY, USA: ACM, 2016. (ICSE '16), p. 1169–1180. ISBN 978-1-4503-3900-1. Available at: <http://doi.acm.org/10.1145/2884781.2884864>.

AUGER, C. *Use of Reports Literature*. Archon Books, 1975. (Butterworths guides to information sources). ISBN 9780208015068. Available at: <https://books.google.com.br/books?id=gfa3AAAAIAAJ>.

BARBOSA, E. A.; GARCIA, A.; ROBILLARD, M. P.; JAKOBUS, B. Enforcing exception handling policies with a domain-specific language. *IEEE Transactions on Software Engineering*, v. 42, n. 6, p. 559–584, June 2016. ISSN 0098-5589.

BEN-ASSULI, O.; JACOBI, A. Improving robustness of scale-free networks to message distortion. In: *Knowledge and Technologies in Innovative Information Systems - 7th Mediterranean Conference on Information Systems, MCIS 2012, Guimarães, Portugal, September 8-10, 2012. Proceedings*. [s.n.], 2012. p. 185–199. Available at: <https://doi.org/10.1007/978-3-642-33244-9_13>.

BRODU, E.; FRÉNOT, S.; OBLÉ, F. Toward automatic update from callbacks to promises. In: *Proceedings of the 1st Workshop on All-Web Real-Time Systems*. New York, NY, USA: ACM, 2015. (AWeS '15), p. 1:1–1:8. ISBN 978-1-4503-3477-8. Available at: <http://doi.acm.org/10.1145/2749215.2749216>.

BRUNTINK, M.; DEURSEN, A. van; TOURWÉ, T. Discovering faults in idiom-based exception handling. In: *Proceedings of the 28th International Conference on Software Engineering*. [S.l.: s.n.], 2006. p. 242–251.

CABRAL, B.; MARQUES, P. Exception handling: A field study in java and .net. In: ERNST, E. (Ed.). *ECOOP 2007 – Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. p. 151–175. ISBN 978-3-540-73589-2.

CASSEE, N.; PINTO, G.; CASTOR, F.; SEREBRENIK, A. How swift developers handle errors. In: *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. [s.n.], 2018. p. 292–302. Available at: <https://doi.org/10.1145/3196398.3196428>.

CHEN, C.-T.; CHENG, Y. C.; HSIEH, C.-Y.; WU, I.-L. Exception handling refactorings: Directed by goals and driven by bug fixing. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 82, n. 2, p. 333–345, Feb. 2009. ISSN 0164-1212. Available at: <http://dx.doi.org/10.1016/j.jss.2008.06.035>.

D'AGOSTINO, R.; PEARSON, E. S. Tests for departure from normality. Empirical results for the distributions of b2 and b1. *Biometrika*, v. 60, n. 3, p. 613–622, 12 1973. ISSN 0006-3444. Available at: <https://dx.doi.org/10.1093/biomet/60.3.613>.

D'AGOSTINO, R. B. An omnibus test of normality for moderate and large size samples. *Biometrika*, v. 58, n. 2, p. 341–348, 08 1971. ISSN 0006-3444. Available at: <https://dx.doi.org/10.1093/biomet/58.2.341>.

ERSHOV, A. *Node.js Error Handling Patterns Demystified (with examples)*. 2018. <https://dev.to/aershov24/nodejs-error-handling-demystified-2nbo>. Online; accessed 15 November 2018.

ESLINT. *disallow empty block statements (no-empty)*. 2019. <https://eslint.org/docs/rules/no-empty>. Online; accessed 15 January 2019.

ESLINT. *disallow reassigning exceptions in catch clauses (no-ex-assign)*. 2019. <https://eslint.org/docs/rules/no-ex-assign>. Online; accessed 15 January 2019.

ESLINT. *Enforce Callback Error Handling (handle-callback-err)*. 2019. <https://eslint.org/docs/rules/handle-callback-err>. Online; accessed 15 January 2019.

EUGSTER, P. T.; FELBER, P. A.; GUERRAOUI, R.; KERMARREC, A.-M. The many faces of publish/subscribe. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 35, n. 2, p. 114–131, Jun. 2003. ISSN 0360-0300. Available at: <http://doi.acm.org/10.1145/857076.857078>.

FLANAGAN, D. *JavaScript: The Definitive Guide*. 3rd. ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1998. ISBN 1565923928.

FUKUDA, H.; LEGER, P. A library to modularly control asynchronous executions. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2015. (SAC '15), p. 1648–1650. ISBN 978-1-4503-3196-8. Available at: <http://doi.acm.org/10.1145/2695664.2696034>.

GALLABA, K.; HANAM, Q.; MESBAH, A.; BESCHASTNIKH, I. Refactoring asynchrony in javascript. In: *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.: s.n.], 2017. p. 353–363.

GALLABA, K.; MESBAH, A.; BESCHASTNIKH, I. Don't call us, we'll call you: Characterizing callbacks in javascript. In: *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.: s.n.], 2015. p. 247–256.

GARCIA, A. F.; RUBIRA, C. M. F.; ROMANOVSKY, A. B.; XU, J. A comparative study of exception handling mechanisms for building dependable object-oriented software. In: *Journal of Systems and Software 59(2)*. [S.l.: s.n.], 2001. p. 197–222.

GOLDBERG, Y. *Checklist: Best Practices of Node.JS Error Handling (2018)*. 2018. <https://goldbergyoni.com/checklist-best-practices-of-node-js-error-handling/>. Online; accessed 15 November 2018.

GU, Y.; XUAN, J.; ZHANG, H.; ZHANG, L.; FAN, Q.; XIE, X.; QIAN, T. Does the fault reside in a stack trace? assisting crash localization by predicting crashing fault residence. *Journal of Systems and Software*, v. 148, p. 88–104, 2019. Available at: <https://doi.org/10.1016/j.jss.2018.11.004>.

HONG, S.; PARK, Y.; KIM, M. Detecting concurrency errors in client-side java script web applications. In: *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*. Washington, DC, USA: IEEE Computer Society, 2014. (ICST '14), p. 61–70. ISBN 978-1-4799-2255-0. Available at: <http://dx.doi.org/10.1109/ICST.2014.17>.

JAKOBUS, B.; BARBOSA, E. A.; GARCIA, A. F.; LUCENA, C. José Pereira de. Contrasting exception handling code across languages: An experience report involving 50 open source projects. In: *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*. [S.l.: s.n.], 2015. p. 183–193.

JOYENT. *Production Practices*. 2019. <https://www.joyent.com/node-js/production/design/errors>. Online; accessed 15 November 2018.

KALLIAMVAKOU, E.; GOUSIOS, G.; BLINCOE, K.; SINGER, L.; GERMAN, D. M.; DAMIAN, D. The promises and perils of mining github. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2014. (MSR 2014), p. 92–101. ISBN 978-1-4503-2863-0.

KAMBONA, K.; BOIX, E. G.; MEUTER, W. D. An evaluation of reactive programming and promises for structuring collaborative web applications. In: *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. New York, NY, USA: ACM, 2013. (DYLA '13), p. 3:1–3:9. ISBN 978-1-4503-2041-2. Available at: <http://doi.acm.org/10.1145/2489798.2489802>.

KERY, M. B.; GOUES, C. L.; MYERS, B. A. Examining programmer practices for locally handling exceptions. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2016. (MSR '16), p. 484–487. ISBN 978-1-4503-4186-8. Available at: <http://doi.acm.org/10.1145/2901739.2903497>.

LEE, P. A.; ANDERSON, T. *Fault Tolerance: Principles and Practice*. 2nd. ed. Berlin, Heidelberg: Springer-Verlag, 1990. ISBN 0387820779.

MANN, H. B.; WHITNEY, D. R. On a test of whether one of two random variables is stochastically larger than the other. *Annals of Mathematical Statistics*, v. 18, 11 1946.

MIKKONEN, T.; TAIVALSAARI, A. *Using JavaScript As a Real Programming Language*. Mountain View, CA, USA, 2007.

MILLER, R.; TRIPATHI, A. Issues with exception handling in object-oriented systems. *ECOOP97 — Object-Oriented Programming Lecture Notes in Computer Science*, p. 85–103, 1997.

NEMETH, G. *Node.js Best Practices | RisingStack*. [S.l.]: RisingStack Engineering - Node.js Tutorials Resources, 2017. <https://blog.risingstack.com/node-js-best-practices/>. Online; accessed 15 November 2018.

NOTNA, A. *Async patterns in Node.js: only 5 different ways to do it!* 2017. <https://codeburst.io/async-patterns-in-node-js-only-4-different-ways-to-do-it-70186ee83250>. Online; accessed 15 November 2018.

OCARIZA, F. S.; PATTABIRAMAN, K.; ZORN, B. G. Javascript errors in the wild: An empirical study. *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, p. 100–109, 2011.

PARNAS, D. L.; WÜRGES, H. Response to undesired events in software systems. In: *Proceedings of the 2nd International Conference on Software Engineering*. [S.l.: s.n.], 1976. p. 437–446.

PHILIPS, L.; KOSTER, J. D.; MEUTER, W. D.; ROOVER, C. D. Dependence-driven delimited cps transformation for javascript. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. New York, NY, USA: ACM, 2016. (GPCE 2016), p. 59–69. ISBN 978-1-4503-4446-3. Available at: <http://doi.acm.org/10.1145/2993236.2993243>.

RICHARDS, G.; HAMMER, C.; BURG, B.; VITEK, J. The eval that men do a large-scale study of the use of eval in javascript applications. In: *ECOOP 2011 – Object-Oriented Programming*. [S.l.: s.n.], 2011. p. 52–78.

ROBILLARD; MURPHY. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, p. 191–221, Apr. 2003. ISSN 1049-331X.

SCHARDT, B. *Error Handling In Node/Javascript Sucks. Unless You Know this. 2018*. 2018. <https://medium.com/front-end-hacking/error-handling-in-node-javascript-suck-unless-you-know-this-2018-aa0a14cfdd9d>. Online; accessed 15 November 2018.

SENA, D.; COELHO, R.; KULESZA, U.; BONIFáCIO, R. Understanding the exception handling strategies of java libraries: An empirical study. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2016. (MSR '16), p. 212–222. ISBN 978-1-4503-4186-8. Available at: <http://doi.acm.org/10.1145/2901739.2901757>.

SENN, S.; RICHARDSON, W. The first t-test. *Statistics in Medicine*, v. 13, n. 8, p. 785–803, 1994. Available at: <https://onlinelibrary.wiley.com/doi/abs/10.1002/sim.4780130802>.

SOARES, W. *9 Ways To Avoid Pitfalls Using Node.js*. 2017. <http://blog.avenuecode.com/9-ways-to-avoid-pitfalls-using-nodejs>. Online; accessed 15 November 2018.

SYED, B. *Exception Handling*. 2018. <https://basarat.gitbooks.io/typescript/docs/types/exceptions.html>. Online; accessed 15 November 2018.

VOUTILAINEN, J. *Evaluation of Front-end JavaScript Frameworks for Master Data Management Application Development*. Master's Thesis (Master's Thesis) — Metropolia University of Applied Sciences, The address of the publisher, 12 2017. An optional note.

WANG, J.; DOU, W.; GAO, Y.; GAO, C.; QIN, F.; YIN, K.; WEI, J. A comprehensive study on real world concurrency bugs in node.js. In: *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.: s.n.], 2017. p. 52–78.

WILSON. *Node.Js 8 the Right Way: Practical, Server-Side JavaScript That Scales*. 1st. ed. [S.l.]: Pragmatic Bookshelf, 2018. ISBN 168050195X, 9781680501957.