Institute of Software Engineering
Universitätsstraße 38
70569 Stuttgart
Germany

Bachelorarbeit

Do TypeScript Applications Show Better Software Quality than JavaScript Applications? A Repository Mining Study on GitHub

Manuel Merkel

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Dr. Justus Bogner
Commenced:	May 10, 2021
Completed:	November 10, 2021

Abstract

JavaScript, the most widely used programming language, is *the* language for developing the client-side of web applications and, more recently, smartphone and desktop applications. However, due to its dynamic and flexible nature, it often has the reputation of delivering poor software quality. The emerging popular language TypeScript, which evolved from JavaScript, offers features to eliminate these prejudices. Therefore, this thesis addresses the arising question: Do TypeScript applications show better software quality than JavaScript applications? Furthermore, the impact of type safety on the software quality of TypeScript applications is investigated and whether the chosen framework has an impact on the software quality of the two programming languages.
To be able to make a statement about these properties, a large-scale mining software repository study was conducted on GitHub. A total of 604 GitHub JavaScript and TypeScript repositories with over 16 million lines of code were examined. Three research questions were created to examine four software quality metrics: 1. code smells per LoC as an indicator of code quality 2. bug-fix commit ratio, which reflects bug proneness 3. average time a bug issue is open on GitHub, which represents the bug resolution time and 4. cognitive complexity per LoC to analyze code understandability. These metrics were categorized and evaluated differently for each research question.
The data showed that TypeScript applications exhibit significantly better code quality and understandability than JavaScript applications. Contrary to expectations, there was not enough evidence that TypeScript applications are less bug prone and take less time in bug resolution than JavaScript applications. For TypeScript applications, it appears that insisting on type safety is beneficial, as it leads to significantly better quality in three out of four quality metrics. Lastly, it can be summarized that the choice of the framework has only a modest impact on the software quality of the programming languages.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

This chapter gives the motivation for the thesis and describes the research questions and the methods used in the study. Finally, the structure of the thesis is provided.

1.1 Motivation

In 1995, JavaScript[1] was developed and introduced for small client-side tasks in the browser. With the growth of the Internet, JavaScript spread rapidly and, according to a recent Stackoverflow survey[2], is now the most widely used programming language. Potential reasons may include its versatility, flexibility, and ease of use. Typescript[3], a superset of JavaScript, is also growing in popularity[4] and some claim it will be *the* programming language for developing next-generation web apps, mobile apps, Node.js projects, and IoT devices [Che19]. The main difference between the two programming languages is that TypeScript extends JavaScript with optional type annotations. Optional means that there is an "any" type in addition to the other types, such as number, string, boolean, etc. If a variable is annotated with the "any" type, the type safety is lost and, thus, one of the main reasons for using TypeScript. Consequently, we have the emerging statically typed TypeScript on one side and the long-proven dynamically typed JavaScript on the other. The former performs type checking at compile-time, while the latter performs type checking at run-time [Jan15].

JavaScript, and increasingly TypeScript, are the languages for developing the client-side of web applications and, more recently, desktop applications through frameworks like Electron[5]. With nearly two billion websites online[6] in total, the two languages have proven themselves very often for many pages. Many of these applications are extensive and have complex business logic. Especially in the information age, we use these sites frequently and they are an indispensable part of our everyday life. For example, the client-side of the most used search engine on the Internet with 1.6 billion visits per month, Google[7], was developed with TypeScript and JavaScript. For eBay[8], an online auction house, and LinkedIn[9], the largest professional network, not only the front-end was

[1]JavaScript, https://www.javascript.com
[2]Stackoverflow survey, https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages
[3]TypeScript, https://www.typescriptlang.org
[4]GitHub statisticy, https://octoverse.github.com/
[5]Electron, https://www.electronjs.org/
[6]Internet statistics, https://www.internetlivestats.com/
[7]Google, https://www.google.com/
[8]eBay, https://www.ebay.com/
[9]LinkedIn, https://www.linkedin.com/

written in JavaScript, also a part of the back-end[10].
In fact, software is the most commonly used product, as Jones et al. point out in their book "The economics of software quality"[JBA12]. However, it would also be the product that has the highest failure rates, mainly due to poor quality. Moreover, they emphasize that poor quality leads to a higher demand for software developers for maintenance, and thus higher costs. Therefore, it is even more important to examine the software quality and bring it to the highest possible level. JavaScript, though, does not have the best reputation when it comes to delivering very good software quality. Due to its beginner-friendliness, because the language is easy to learn, and due to its dynamic nature without a compiler, it is assumed that JavaScript delivers poor code quality [FM13].
Especially with the importance of software quality combined with the large number of applications written in JavaScript, the question arises: Does using the emerging programming language TypeScript result in better software quality than using JavaScript?

However, there is insufficient empirical evidence to support the claim that TypeScript leads to better software quality than JavaScript. Indeed, the question of whether statically typed languages have a positive impact on software quality or not has been preoccupying the experts for several decades, with mostly contradictory results. Proponents of static type systems insist, that:

- "[...] it allows early detection of some programming errors."[Pie02]
- there is a "[...] better documentation in the form of type signatures [...]"[MD04]

One of these points may have even prompted Microsoft to develop TypeScript in 2012.

Whereas advocates of dynamic typing claim, that:

- "[...] static typing is too rigid, and that the softness of dynamically languages makes them ideally suited for prototyping systems with changing or unknown requirements, or that interact with other systems that change unpredictably."[MD04]

Many studies have evaluated the benefits of static typing in terms of software quality [PT98][GBB17][RPFD14][ZLH+20], however, some papers have proven the opposite, for instance, [Han10]. Nevertheless, there are many threats to validity, especially in large-scale studies [Rob10], which makes replications of them all the more important [Has08]. However, many of these studies are not replication friendly, making it difficult to verify the validity of the results, as shown in this replication study [BHM+19] to the work of Ray et al. [RPFD14]. More on this can be found in chapter 3.
To the best of our knowledge, no study analyzes and compares JavaScript and TypeScript in terms of software quality on a large scale. Thus, we want to add another part of research in the field of software quality.

[10]Programming languages used in most popular websites, https://en.wikipedia.org/wiki/Programming_languages_used_in_most_popular_websites

1.2 Research Questions and Process

The aim of this thesis is therefore to validate if TypeScript applications, despite its proximity to JavaScript, delivers better software quality. To prove this claim, a large set of JavaScript and TypeScript applications need to be empirically analyzed and compared. With the help of this data collection, insights into the software quality of the two programming languages are possible. Furthermore, a possible influencing factor on the analyzed properties in TypeScript projects is investigated.

1.2.1 Research Questions

To successfully prove the properties described above, this thesis seeks to answer the following three research questions:

RQ_1 Do TypeScript applications exhibit higher software quality than JavaScript applications?

RQ_2 Do TypeScript applications that less frequently use the "any" type show better software quality?

RQ_3 Does the chosen framework lead to software quality differences between TypeScript and JavaScript applications?

The motivation behind the first research question is to generally show that TypeScript has the better software quality of the two programming languages. For the second research question, the focus is on TypeScript and checks whether the software quality decreases as the number of used "any" types increases. Finally, the last research question is exploratory and examines whether there is a software quality difference between the two languages within the frameworks. Therefore, the data is categorized according to the well-known frameworks Angular[11] and Vue[12] and the library React[13].

There is a wide range of possibilities to answer the research questions [WRH+12]. On the one hand, there is a controlled approach, where as few factors as possible are changed, usually by sacrificing scalability, and on the other hand, there is a less controlled approach on a large scale to make a statement about the big picture from its large sample size, but mostly with many threats to validity.

1.2.2 Method

This research uses the latter method and conducts a mining software repository (MSR) study [Has08][KGB+15]. GitHub[14], a network-based version management service for software development projects, serves as the source of open-source JavaScript and TypeScript applications. With GitHub comes the possibility of large scale. According to our research, there are more than 300,000 JavaScript repositories with a popularity rating of more than five stars in a period from early 2012 to

[11]Angular, `https://angular.io/`
[12]Vue, `https://vuejs.org/`
[13]React, `https://reactjs.org/`
[14]GitHub, `https://github.com`

mid-2021; for TypeScript, the same selection criteria lead to more than 60,000 repositories. These repositories are rich in data, as access to the complete history of these projects is readily available. By selecting suitable repositories, we will extract the following information to measure software quality:

1. Bug proneness: The number of bug-fix commits is used as a key indicator of bug proneness. For evaluation, the entire commit history is scanned to count the bug-fix commits.
2. Bug resolution time: The average time a bug issue is open is determined via the GitHub issue tracking system[15].
3. Code quality and understandability: Code smells, which determine code quality, and cognitive complexity, which represent code understandability, are reported by the static code analysis tool SonarQube[16].

The measured properties are an indicator of software quality and will be used to answer all three research questions. Data collection is automated as much as possible to achieve the best possible reproducibility.

1.3 Structure of the Thesis

The thesis is divided into eight chapters: first, chapter 2 explains the required technical background knowledge. Chapter 3 provides the state of the research and describes the work that inspired this thesis. Subsequently, chapter 4 describes the methods used in the study. The starting point of chapter 5 is the analysis and evaluation of the gathered data. Very crucial in such a large-scale study is to address the threats to validity, which are considered in chapter 7. Chapter 6 discusses the results of the study, and chapter 8 concludes with a summary.

[15]Issue tracking system, `https://guides.github.com/features/issues/`
[16]SonarQube, `https://www.sonarqube.org/`

2 Background

First, the two programming languages are discussed and their main difference, the type system, is compared. In addition, the term software quality is specified. Finally, an overview is made of three widely used frameworks that are considered in the context of this work in relation to software quality.

2.1 JavaScript

The following information was taken from the book “Javascript: The Definitive Guide - Master the World’s Most-Used Programming Language” by the author Flanagan [Fla20].
JavaScript is a scripting language and was developed by Netscape in 1995 for dynamic HTML in web browsers. Through user interaction, such as moving the mouse or typing on the keyboard, HTML and CSS changed, reloaded, or generated content. The programming language is dynamically typed, interpreted and supports the style of object-oriented or functional programming. However, through this dynamic, flexible and messy nature, prejudices often arise that JavaScript delivers poor software quality [FM13][SMKA17]. JavaScript was standardized by ECMA[1], a standards organization, and was given the name “ECMAScript”. Today, all modern browsers support version 5 of ECMAScript from 2010, in which Node was released as a new host environment for JavaScript in addition to the browser. This opened a new door for JavaScript and since then, it can access the complete operating system. From 2015, when ECMAScript version 6 (ES6) was released and new features have been added annually. For example, classes and modules were added, which made larger JavaScript applications possible. In this manner, ECMA tries to correct problems from previous versions of JavaScript. Since older browsers usually cannot read the new features of JavaScript, there is the compiler Babel[2]. This translates the new syntax into an older syntax and can be used even though the target system cannot support the new features.
This is how JavaScript became the language of the web. The programming language continued to evolve and is now also used for the development of desktop, tablet and smartphone applications.

2.2 TypeScript

Information for this section was taken from the book “TypeScript Quickly” by authors Fain and Moiseev [FM20].
TypeScript is a superset of JavaScript and was released by Microsoft in 2012. Superset means that it uses all ECMAScript features plus the latest ones that are being developed and more. TypeScript is

[1]ECMA, `https://www.ecma-international.org/`
[2]Babel, `https://babeljs.io/`

Listing 2.1 Javascript dynamic typing

```
var string = 'I am a string'
string = 42 // Evaluates string to 42 and becomes a number

var obj = {}
obj.foo // Evaluates to undefined

function multiply(x){
    return x * x
}
multiply('a') // Evaluates to NaN
```

statically typed and transpiled, i.e. the source code is converted to another programming language, namely JavaScript. However, the type annotation is optional by the "any" type. The "any" type is used, for example, when type information is not available due to the use of third-party libraries. Another use case is the migration of existing JavaScript code to TypeScript, where the "any" type is gradually replaced by the appropriate type. However, type safety is lost when using the "any" type and is therefore not recommended. Therefore, any JavaScript program can be a valid TypeScript program if it does not contain any errors that only become apparent during transpilation, such as type-related ones. This way, popular JavaScript libraries can also be used in TypeScript. Babel also finds its use for TypeScript code, as the code is transpiled into JavaScript.
Wherever JavaScript is applied, TypeScript can also be applied. When choosing a programming language, the advantages and disadvantages of static versus dynamic typing need to be weighed, and whether the few extra features, such as interfaces that only TypeScript has, are relevant.

2.3 Type System

The information from this section comes from the book "Programming with Types" by Riscutia [Ris19].
A type system is a set of rules that assigns and enforces types to variables, functions, or objects in a programming language. The type defines which operations can be applied to these objects, what they represent, and the set of allowed values. Either the type is specified by the developer and the type system obtains the information from there, or it obtains the type from the context of the code. Type checking is done at compile time or during code execution to ensure that the type system rules are respected by the program.

2.3.1 Static and Dynamic Typing

Static typing checks the types at compile time, which guarantees the correct types at runtime. Dynamic typing checks the types later, at runtime, which can lead to runtime errors due to incorrectly assigned types. This also imposes no type constraints at compile time. What this means can be seen in the JavaScript code snippet in listing 2.1.

Listing 2.2 TypeScript static typing

```
let string: String = 'I am a string'
string = 42 // Type 'number' is not assignable to type 'string'.

let obj = {}
obj.foo // Property 'foo' does not exist on type '{}'.

function multiply(x: Number): Number{
    return x * x
}
multiply('a') // Argument of type 'string' is not assignable to parameter of type 'Number'.
```

Instead of throwing an error, JavaScript evaluates invalid operations to a (wrong) value and, as the last two examples show, ends with a runtime error. The following code snippet in listing 2.2 does the same as listing 2.1, but it was implemented with TypeScript.

What can be seen in the last code snippet is that none of the operations compile and there is always a compilation error. Furthermore, TypeScript gives very useful error messages in the editor while developing.
Ultimately, both typing systems have their advantages and disadvantages, and it is important to decide which type is more suitable for each application. For smaller applications, the dynamic type is often more suitable due to its flexibility, while for larger ones, the static type appears more scalable.

2.4 Software Quality

Very early on, software quality was a concern. Already in 1976, there was one of the first studies, which examined this area. Boehm et al. defined three questions that should be asked when acquiring software products:

- “How well (easily, reliably, efficiently) can I use it as-is?”
- “How easy is it to maintain (understand, modify, and retest)?”
- “Can I still use it if I change my environment?”[BBL76]

These questions were derived from their defined tree of software quality characteristics and reflect the most important attributes of that tree: portability, reliability, efficiency, human engineering and maintainability. The attributes were further refined into sub-characteristics. The leaves of the tree are the primitive characteristics, which are the foundation for quantitative measurement.
However, there is not one single definition of software quality. Nevertheless, there is now already the second ISO standard, after the ISO 9126, the quality model ISO 25010 was created in 2011. The ISO brings methods for the evaluation of the quality of software products, which can serve as guidelines for the development of good software. The model addresses the internal and external quality of the software product. The former is from the point of view of the developer, who attaches importance to, for example, good maintainability and portability, and the latter is from the point of

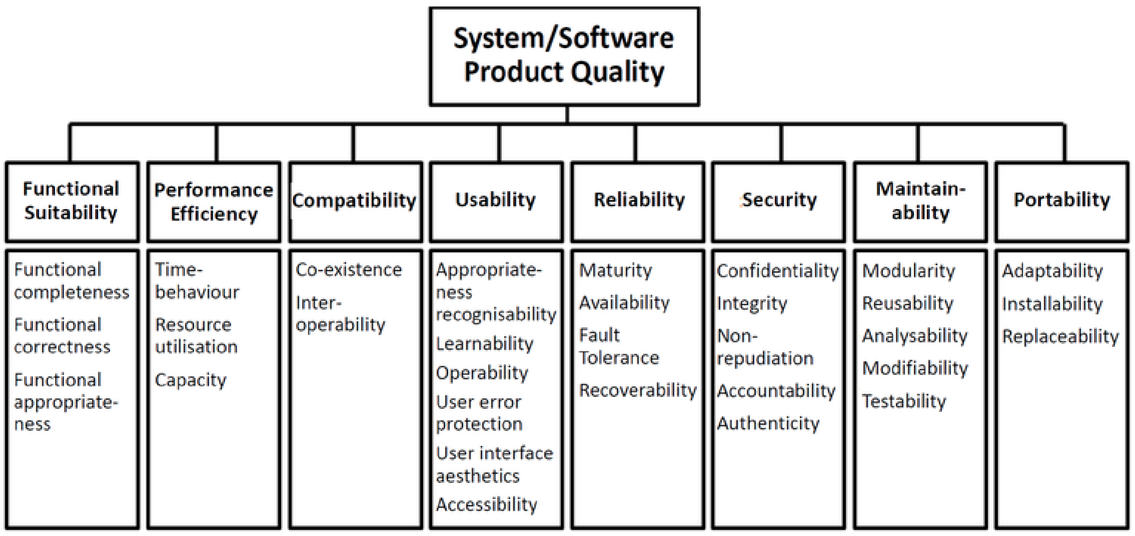


Figure 2.1: Quality model according to ISO/IEC 25010, taken from [Tie18]

view of the end-user, who attaches more importance to usability [HM16]. Thus, software quality was standardized through characteristics and sub-characteristics, which can be seen in figure 2.1, taken from Tiemeyer [Tie18].

For an exact definition of the attributes, the web page of ISO 25010[3] is recommended, since this describes the characteristics very well. As can be seen in the figure, the characteristics of the model from Boehm et al. are mostly included in the modern definition, however, they have been extended and adapted.
As already mentioned with the model of Boehm et al. [BBL76], the sub-characteristics of ISO 25010 also provide a foundation to create metrics to measure the main software quality characteristics. Since the measurement of such metrics can mostly only be automated on a large scale, tools have been developed to measure them and provide feedback on the software quality. Most of the tools analyze the code statically. SonarQube, for example, which was used to obtain the data for this study, is one such tool. Among other things, this can be used to measure the maintainability through code smells and the time required to fix them. In addition, it also makes a statement about reliability, by measuring bugs and the effort to fix them. Also included in the ISO 25010 is security, which SonarQube measures, among other things, by the number of vulnerabilities and the time it takes to fix them. Furthermore, SonarQube measures data based on the source code, such as complexity, duplications, size of multiple attributes such as classes, comment lines, etc., and lastly a lot about tests. A list of metrics with explanations can be found on their website[4]. However, these tools are designed to set the quality of the software as high as possible to facilitate development and maintainability.

[3]ISO/IEC 25010, `https://iso25000.com/index.php/en/iso-25000-standards/iso-25010`

[4]SonarQube metrics, `https://docs.sonarqube.org/latest/user-guide/metric-definitions/`

2.5 Frameworks

A framework is a reusable architecture or infrastructure that is an integrated collection of components. It is not a complete program, but provides the framework in which the application is programmed. They specify the control flow of the application and an interface, for the concrete classes, that need to be created and registered, and that are reusable anywhere within the software under development. Frameworks are developed and used for the reuse of architectural patterns, which is mostly domain specific [Edw14].

2.5.1 Web Framework

Formerly, applications were developed manually, which was a lot of work. As the web apps became more and more complex and the demand on them became higher, web frameworks were introduced, that are used for the development of static and dynamic web pages. There are two categories, client-side frameworks, which are used to develop and improve graphical user interfaces, and server-side frameworks, which provide tools and libraries to, for example simplify the databases access [CJYF19]. For this study, only the client-side frameworks were of interest, since JavaScript and TypeScript are mainly used there. In the following, the two well-known frameworks Angular and Vue are explained and additionally the library React, because it is also used to create full-fledged web applications and is very often used. When choosing the right framework, it is necessary to choose based on the task, because all of them have their weaknesses and strengths.
Many of these frameworks are primarily designed to replace the old concept of multi-page applications, where a new HTML document is loaded with every user interaction, with the single-page concept, where only the changed UI parts are updated by the server. The latter principle is more performant. The following information was used from Saks [Sak19] and for additional publication years, etc. the data from the respective Wikipedia page was used.

React

React is an open-source library released by Facebook in 2013 primarily for developing user interfaces of single-page applications. In 2015, however, React Native was added as a library for developing iOS and Android applications. When developing applications, React makes it easier to update the UI when the user interacts by dividing the view into small components. For these components, the JSX syntax is used, which is a mix of JavaScript and HTML. In the end, React is only responsible for state management and rendering of the state, which is why for example for routing extra third-party libraries need to be installed with the Node Package Manager.

Angular

In 2011, the open-source framework AngularJS (version 1) for developing single-page applications was released by Google employees. The second version (from now on simply Angular) was completely redeveloped in 2016 and is now based on TypeScript instead of JavaScript. Today, in addition to web applications, mobile and desktop applications can also be created with Angular. Almost any HTML syntax is valid Angular template syntax, but not the other way around. Angular extends HTML with JavaScript function expressions, by double curly braces JavaScript can be rendered in the view. As architecture Angular partly uses the MVC (Model View Controller) pattern, i.e. the division of the software into model view and controller.

Vue

Vue was released in 2014 primarily to create single-page applications. Thereby it partially evolved from Angular, which the developer found too heavy for some tasks, so he extracted the features he liked to create Vue. It also has many similarities with React, especially in terms of data binding and the components and is generally easy to learn if the basic knowledge of JavaScript, HTML and CSS are there. A Vue file contains HTML-based template syntax that is bound to the Vue instance JavaScript. Through JavaScript code, data can be passed to the template. The applications are mostly built as a tree of nested Vue components.

3 Related Work

This is not the first study that compares programming languages with respect to their impact on software quality. Often, statically and dynamically typed languages were compared with each other. In this chapter, different works that have motivated and inspired this thesis are described.

3.1 Debate between Static and Dynamic Type Systems

In 2010, Stefan Hanenberg devoted himself to the study of type systems. In his paper, [Han10], he presents an empirical study with 49 ungraduate students who programmed a parser. A between-subject design was used because he assumed that when students "program statically" and then switch to "dynamically programming" they are still "thinking static". For the experiment, he wrote a new programming language, with similarities to Java, Ruby and Smalltalk, with two different versions: one is statically typed, the other is dynamically typed. At the beginning of the study, the students were divided into two groups, one assigned to the static type system and one to the dynamic type system. They were all introduced to the new programming language and were then able to program the parser in a controlled manner within 27 hours, spread over four days. Hanenberg analyzed two metrics, the programming time, from the beginning of programming until all tests for a minimal scanner were fulfilled, and the resulting quality of the parser, which was evaluated by 400 tests. Contrary to his expectations, the results showed that using the static type system did not have a significant positive impact on the programming time, as well as on the quality. An interesting point that Hanenberg evaluated through debugging times was that, for small tasks, there was no significant difference between the static and dynamic type systems, but when correcting type errors in the parser, the statically typed group required significantly more time.

Similar to Hanenberg, Kleinschmager et al. also examined in their paper [KRS+12] the development time for a programming task with a dynamically typed language, Groovy, and a statically typed language, Java. For the Java application, a video game was used, for Groovy all classes, methods, etc. from this video game were renamed to be in another domain, a simple email client. In the study from 2012, a within-subject design was used with a total of 36 subjects. The participants were divided into two groups, one of them performing first the programming tasks with the static type system and then the one with the dynamic type system, the other group vice versa. The programming time of three different kinds of programming tasks was investigated:

1. using the code from an existing system, where documentation is only provided by the source code;
2. fixing type errors (no-such-method-errors for dynamically typed applications) in existing code;
3. fixing semantic errors in existing code [KRS+12].

A total of nine programming tasks per programming language were completed by each participant. As a result, Kleinschmager et al. found a positive influence of the static type system for six of the nine tasks. This was true for all tasks 2 and almost all tasks 1. For tasks 3, no influence of the type system was found. Especially the result that fixing type errors (2) for every programming task the static type system has a positive influence is in contrast to the study of Hanenberg [Han10] described above. Particularly these two works show how much the results diverge between the two type systems.

3.2 Software Quality Differences between JavaScript and TypeScript

One of the few papers that directly compared JavaScript with TypeScript in software quality is "To Type or Not to Type: Quantifying Detectable Bugs in JavaScript" [GBB17] published by Gao et al. in 2017. The goal of this study was to investigate whether bugs could have been prevented by static programming in projects that were programmed with JavaScript. Gao et al. examined 398 JavaScript projects from GitHub. To get the percentage of preventable bugs from these projects, fix commits and issues from the issue tracking system were used to search bugs in the history of the code. This narrowed down a region in the code with the bug, which was then typed. As a very interesting result, it was found that using TypeScript would have prevented 60 of the 400 bugs examined, i. e. 15%.

3.3 Large-Scale MRS Studies

Ray et al. completed a large-scale mining software repository (MSR) study [RPFD14] in 2014 and examined the effect of different programming languages on software quality. They investigated the top 19 programming languages from GitHub, removed CSS, Shell script, and Vim script and added JavaScript and TypeScript. For each programming language, the top 50 repositories, sorted by stars, were examined (altogether 850 projects in 17 programming languages). Using the GitHub Archive database, where all public activity of GitHub projects was stored, Ray et al. analyzed with a mixed-methods approach a large amount of data and investigated among other things 1. whether programming languages are more defect prone than others and 2. which language properties relate to defects. After the evaluation, they concluded: "The data indicates [...] that static typing is better than dynamic;" [RPFD14].

Critical towards the results from Ray et al. are Berger et al. with their replication study [BHM+19]. In 2019, they investigated the same research questions and tried to use the same methods to obtain the same results. However, they partially failed to do so, two of the four research questions could not be replicated due to missing code and irreconcilable differences in the data. In the reanalysis of the first research question, they found that TypeScript had to be removed from the data set. A large part of these projects is from the period before 2012, i.e. before the release of TypeScript. The largest three remaining projects contained only declarations and no code. In addition, the classification by similar characteristics of the programming languages was mislabeled in the original study. Berger et al. highlighted several threats to validity and summarized that the conclusion of the original study does not hold.

Related to Ray et al. [RPFD14], Zhang et al. completed a large-scale mining software repository study [ZLH+20]. In 2020, they examined a total of 600 GitHub projects, written in an equally distributed manner in the ten most popular programming languages, to exploring the bug-fixing characteristics between the different languages. They analyzed the different projects for, among other things, bug resolution time and compared them under the different programming languages, as well as under static and dynamic typing. The data obtained through the GitHub API was statistically analyzed and showed that dynamically typed languages consume 59.5% more time in bug resolution than statically typed languages.

Another large-scale study was conducted in 2019 by Roehm et al. [RVWJ19] where they examined maintainability prejudices on open-source projects. Through GHTorrent, a mirror of the data offered by GitHub, 6,897 open-source GitHub repositories with the programming languages C, C++, C#, Java and JavaScript were selected. To be able to make a statement about the maintainability of the different programming languages, ten hypotheses were validated. For this purpose, five metrics were measured by ConQuat: clone coverage, comment incompleteness, too long files, too long methods, and nesting depth. The collected data indicated interesting results, among others that "[...] JavaScript code is not worse than other code".

3.4 Summary

In particular, the first two papers in section 3.1 describe the long debate between dynamically typed and statically typed programming languages. Such debates motivate researchers to complete studies with fewer threats and improved methods. So far, one of the very few studies between JavaScript and TypeScript that focused on software quality is described in section 3.2. The results are very interesting. Nevertheless, the study, with its gathered data, needs to be replicated, perhaps even on a larger scale. Many ideas about the study design of mining software repositories and how to avoid threats to validity, among others, were well described by the related work in section 3.3 and have been adopted and adapted for this thesis. However, the study by Ray et al., in particular, shows that replications of such a large-scale study are important.
None of these studies address in a large scale whether TypeScript applications deliver better software quality than JavaScript applications. Specifically, the interesting results of the study in section 3.2 raise the question of whether such results can be found on a larger scale. For this reason, and because there is still no definitive and universal answer to the question of whether static typing is better than dynamic typing, this thesis attempts to shed light on whether TypeScript applications deliver better software quality than JavaScript applications at a large scale.

4 Methodology

This chapter focuses on the study procedure and the methodology used.
A sketch of the study process can be seen in figure 4.1. After creating the research questions and hypotheses, the next step was to select the samples. More on this can be found in section 4.1. Following the manual review of the projects, the data collection was started, which is described in detail in section 4.2. The last step was to analyze the data, i.e. statistically evaluate it and validate the hypotheses. The hypotheses, including the motivation behind them, can be found in section 4.3 and the methods used for the analysis are described in section 4.4. The source code for the process of sampling, data collection, and statistical analysis can be found behind the link[1].

4.1 Study Objects

This section elaborates on the first part of the figure 4.1, the study objects and the sampling. As the title of this thesis mentions, JavaScript and TypeScript applications were examined and compared regarding their software quality. The term “applications” refers to web, desktop and smartphone applications with JavaScript or TypeScript as the primary language.

4.1.1 Selection of the Mining Platform

In a mining software repository study, there are several possibilities to obtain the objects. The first step was to select a version control system with repositories. In the past, first mining attempts were made with CVS[2] and SVN[3], however, today Git[4] is very dominant, which was selected in this study. The next step to retrieve the applications data was to find a platform from which to mine them. There are several possibilities for this, but most of them get their data from GitHub, which is a cloud-based Git repository hosting service where the majority of existing Git projects can be found. The main advantages and disadvantages of the top three platforms that were considered for this study are listed in table 4.1.

The first option was GHArchive with BigQuery, which is a mirror of GitHub and uses SQL queries to retrieve the desired data. The main reason why this platform was not chosen is that the editor is hard to understand and only one terabyte of processed query data is free per month. However, the

[1]Source code, https://figshare.com/articles/software/_/16958389
[2]Concurrent Versions System, https://www.nongnu.org/cvs/
[3]Subversion, https://subversion.apache.org/
[4]Git, https://git-scm.com/
[5]GHArchive, https://www.gharchive.org/
[6]GHSearch, https://github.com/seart-group/ghs

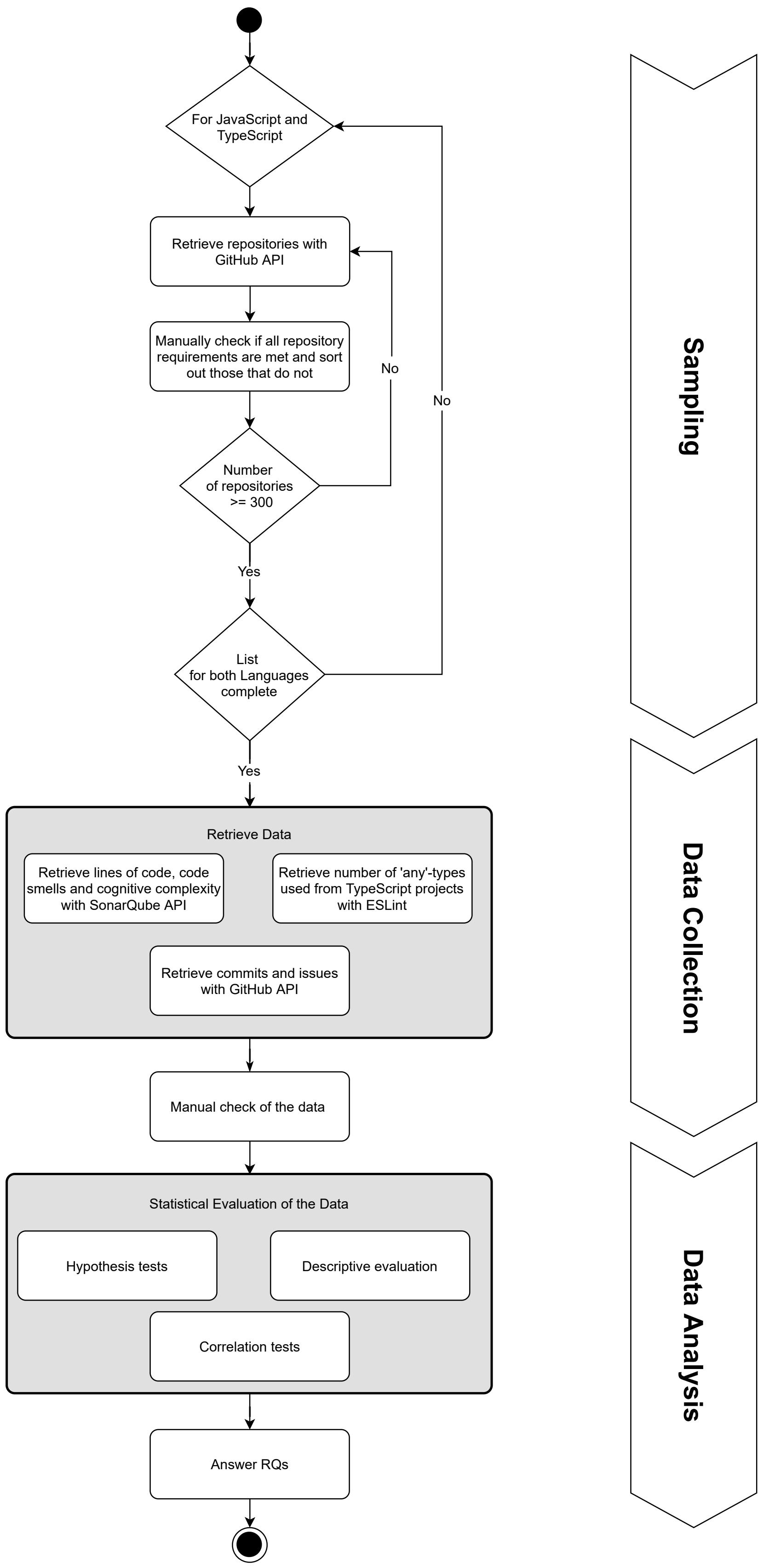

For JavaScript and TypeScript
Retrieve repositories with GitHub API
Manually check if all repository requirements are met and sort out those that do not
No
No
Number of repositories >= 300
Yes
List for both Languages complete
Yes
Retrieve Data
Retrieve lines of code, code smells and cognitive complexity with SonarQube API
Retrieve number of 'any'-types used from TypeScript projects with ESLint
Retrieve commits and issues with GitHub API
Manual check of the data
Statistical Evaluation of the Data
Hypothesis tests
Descriptive evaluation
Correlation tests
Answer RQs
Sampling
Data Collection
Data Analysis

Figure 4.1: Research process

	GHArchive with BigQuery[5]	GHSearch[6]	GitHub
Pro	• Records the complete public GitHub timeline • Data set is updated several times a day • Requests per SQL	• Easy to use GUI	• Direct access to complete public GitHub timeline • Requests to API with parameters • Easily integrated into python script
Con	• Editor of BigQuery is hard to understand • Processed query data is limited	• Database includes “only” 970k repositories in total • Only superficial data (no access to commit history) • Only one language can be searched (Not Vue and TS/JS)	• Requests to API per hour limited • No GUI

Table 4.1: Platforms through which data from repositories can be requested

question arose whether this amount was sufficient or whether an alternative retrieving method had to be found in the process. The second option was GHSearch, which impressed with its simple and easy-to-understand GUI. Unfortunately, it is not possible to access the complete data set from GitHub, which is why this platform was also not selected. The last of the top three options is GitHub itself, which was finally selected. The main reasons was its flexibility and the possibility to integrate the retrieval of the repositories and the data into a Python[7] script. Nevertheless, a significant disadvantage of the system is that an authenticated user with an access token can only send 5000 requests per hour. GHTorrent, historically also a commonly used tool to retrieve repositories data, as of July 2019 does not offer shell access to MongoDB and MySQL anymore.

4.1.2 Requirements

To filter out a large number of projects, only repositories with more than five stars[8] were selected, since projects below that usually have little or no activity. Furthermore, the repositories were not allowed to be a fork, i.e. a copy. In addition, only projects from 2012 to mid-2021 were examined, as TypeScript was released there. If there is a file with extension .ts from years before 2012, these are XML files that contain human language translation [BHM+19]. Lastly, the language part of the primary programming language had to be at least 60%. This number is relatively low because

[7]Python, https://www.python.org/

[8]GitHub stars, https://docs.github.com/en/get-started/exploring-projects-on-github/saving-repositories-with-stars

applications always have content of other languages than JavaScript and TypeScript, like HTML, CSS, etc. or an additional back-end. An overview of the projects with the respective percentage of the primary programming language can be found at the end of the section. The code of the framework Vue.js is listed on GitHub as its own programming language, therefore, these projects were also examined.

4.1.3 Sampling

A Python script was created to automate the sampling as much as possible. The procedure was divided into two steps: In the first step, a rough pre-selection of the samples was made and in the second step, the selection was refined.

First Step

Based on the requirements defined above, the repositories were requested through the GitHub API. The API is very well documented and allows parameters in the request to completely customize it. This yielded a list of possible JavaScript and TypeScript applications, sorted by popularity, i.e. stars [BHV16].

To calculate the size of the sample to be examined, several values are needed, which were then inserted into formulas. There are also online tools[9] that do the calculation by entering three values: population size, confidence level, and precision level. A confidence level, i.e. the probability that the data of the selected samples reflect the data of the population, of 95% was chosen. The level of precision, i.e. the range at which the data of the population deviates from the samples, was set to 5% [Isr92]. These values are common in empirical research and have been adopted here. The population size was more difficult to determine because it had to be estimated. Therefore, the sample size was calculated after the end of the first step, as this allowed a better estimation. Between 2012 and mid-2021, there are more than 300,000 JavaScript GitHub repositories with the characteristics described above as a prerequisite; for TypeScript, there are more than 60,000. Among these repositories, however, a large part was also plugins, collections, components, engines, databases, frameworks, libraries, templates, etc. Therefore, it was assumed that the population size of the applications would result in less than one-twentieth of the above number of repositories. After inputting the three parameters into the online tool, the sample size for each programming language was calculated to be between 341 and 375 projects. The tool ends up using the same formulas from Isreal's paper [Isr92], but it handles the retrieval of dependent variables, making it easier to calculate.

Second Step

The list of JavaScript and TypeScript repositories was then sorted out from popular to less popular until the sample size was reached. The algorithm 4.1 was executed for the two lists and shows the filtering process of each repository. In the following, every step of the algorithm is introduced and described.

[9]Sample size calculator, https://www.surveymonkey.com/mp/sample-size-calculator/

Algorithm 4.1 Pseudocode of sampling

```
def sampling(jsonList):
    reposWithCriteria = {}

    for repository in jsonList:
        # 1st criterion
        if repository is not application:
            continue

        # 2nd criterium
        if repository has bug issues:
            save bug issues to repository
        else:
            continue

        # 3rd criterium
        if repository has more than 30 commits:
            save bug commits to repository
            add repository to reposWithCriteria

    return reposWithCriteria
```

1st criterion. To verify that the repository is an application, the “README.md” file[10] and project description were examined. A README is supposed to explain your work to other users, however, this is not always the case and sometimes it was created poorly, or not at all. To automate this review step, the Latent Dirichlet Allocation[11] (LDA) topic modeling algorithm was used. As input LDA gets one or more texts, in this case, the “README.md” file and the project description. After cleaning the data and tokenizing the text, the LDA algorithm was executed. It outputted two topics, where each topic is a composite of keywords and each keyword has a specific weighting for the topic. Since only repositories that contained an application were accepted, projects that returned plugin, module, extension, API, database, framework, library, etc. from the LDA algorithm were not examined further.

2nd criterion. Due to a metric to be examined, only repositories that contained closed bug issues written in english were considered. The closed bug issues were saved in a JSON file for further analysis. More about this in section 4.2, in the data collection.

3rd criterion. As a final requirement for the samples, they needed more than 30 commits to sort out inactive projects or those where all the code was pushed within a few commits. The bug-fix-commits were saved in a JSON file for further analysis. Again, more on this in the next section.

If one of these criteria did not apply, the repository was not investigated further. Parts of sampling and data collection were conducted in parallel and before the sample was finally selected, data has already been cached. The reason for caching this data was the limitation of API calls on GitHub. This enforces an avoidance strategy to multiple API calls to get one and the same data.

[10]README, https://guides.github.com/features/wikis/
[11]Latent Dirichlet Allocation, https://radimrehurek.com/gensim/models/ldamodel.html

Language	Projects	Lines of Code	Total Commits	Total Issues
JavaScript	299	7,496,726	235,535	56,578
TypeScript	305	8,683,600	340,164	157,497
Total	604	16,180,326	575,699	214,075

Table 4.2: Study objects statistics

Language	PL $\geq$ 90%	90%> PL $\geq$ 80%	80%> PL $\geq$ 70%	70%> PL $\geq$ 60%	Total
JavaScript	125	84	49	41	299
TypeScript	138	82	45	40	305

Table 4.3: Overview of the percentage of the primary programming language (PL)

As shown in figure 4.1, the sampling process was an iterative step. Only 300 repositories at a time were automatically checked for the criteria, since otherwise the JSON file would have become too large due to all the data and could no longer be opened. After that, the repositories were manually examined one by one to see if the script incorrectly outputted projects as valid. After sorting out non-valid samples, the algorithm was executed again. In total, the script was running for the TypeScript sampling for 14 hours with interruptions. Until the final selection of TypeScript samples, over 1200 repositories were manually examined, i.e. the file "Readme.md" was read, the number of issues and the language used as well as the number of commits were checked. For JavaScript, the Python script was adjusted by refining the output criterion so that fewer repositories had to be checked manually. Therefore, this step took less time than the selection of the TypeScript sample.

Ultimately, the calculated sample size had to be adjusted in the second step of sampling, due to an overly generous estimate of the population size. Out of the 66,000 TypeScript repositories, more than 57,000 were examined by the Python script. Of these, 305 met the defined characteristics. From the more than 300,000 JavaScript repositories, only about 41,000 were examined by the Python script, with 299 valid samples returned. If this number of valid samples were extrapolated to 300,000, a population size of about 2,200 would result. Since, according to our observation, the density of available applications becomes smaller as the popularity decreases, this value would be much smaller. As a result of the adjusted population size, the sample size of 299 for JavaScript applies quite well; for TypeScript, the sample size applies even more.

Table 4.2 shows statistics on the projects mined, with some interesting values. A total of 604 applications with over 16 million lines of code, 575,699 commits and 214,075 issues were examined. Commits and lines of code were only counted if the corresponding programming language was used. As already mentioned, the part of the programming languages in applications other than JavaScript or TypeScript is relatively high. Table 4.3 shows the overview of projects with the percentage of the primary programming language. Of this part of the programming language other than JavaScript or TypeScript was thereby often HTML, CSS, etc. which does not serve as back-end; however, among the JavaScript applications, 12% had a back-end with a percentage of more than 10%. Whereas almost 10% of the TypeScript applications had a back-end that contained more than 10% content of the projects.

4.2 Data Collection

This section covers the second part of figure 4.1 and describes its details. For the three research questions, four metrics were defined to measure software quality. These were structured and reused differently for the individual research questions. The entire data collection was automated using a Python script.

4.2.1 Metrics

Two data sources were used to obtain the metrics, one is GitHub and its API and the other is the static code analysis tool SonarQube with its API[12].
For some of the metrics, the source code was required, thus all TypeScript and JavaScript applications were downloaded via the GitHub API.

Code Smells per LoC

For this metric, two values had to be measured, first code smells and second lines of code (LoC). Code smells are an indicator of low code quality and are associated with weak program design. By detecting these smells, critical areas in the code that may even end up in a bug could be found and fixed earlier [EM02][EM12]. Therefore, code smells can also be related to maintainability [Yam13], as they are by SonarQube.
To obtain the number of code smells, SonarQube was used. By applying certain rules to the code, the smells were determined. However, the number of rules varies, but is fairly balanced, with 141 for JavaScript and 147 for TypeScript. As an example, there are rules like “switch statements should have at least 3 case clauses” that indicate bad practices, or conventions like “An open curly brace should be located at the end of a line”. In addition, there are many other types of rules with labels such as confusing, obsolete, suspicious, redundant, clumsy, etc.
Since several steps were required for the analysis, obtaining the data was automated in the Python script. Consequently, requests were made to the SonarQube API as follows for each application individually:

1. A unique token was created for each project.
2. With the project key, in this case simply the name of the repository, and the token obtained, the command to start the analysis was executed in the project root.
3. With the project key, the code smells for each file could be issued. Thus, the exact number was returned for TypeScript and JavaScript applications.

After manually checking the data, it was found that for TypeScript projects many files were not scanned. This was due to SonarQube having issues with the “tsconfig.json” file in which the compiler configuration and root files are defined. Possible reasons were that the file was not found or was not located in the root directory of the project. Therefore, for each TypeScript project, the “tsconfig.json” file, if present, was deleted and replaced with a default file in the root directory.

[12]SonarQube API, https://docs.sonarqube.org/latest/extend/web-api/

To make the individual projects comparable, the code smells data was stored per line of code. The SonarQube API also provides the measurement of ncloc, which is the number of physical lines of code without comments and without empty lines. Again, only lines that actually involve JavaScript and TypeScript code were stored.

Cognitive Complexity per LoC

This metric also requires two data sets, Cognitive Complexity and Lines of Code. Cognitive complexity is a score based on source code and evaluated according to three basic rules:

1. Readable shorthanded structures are ignored. In other words, structures where multiple statements could be combined into a readable shorthanded one. Thus, the score is not increased if, for example, null-coalescing operators, i.e. checking by ?? whether the variable is null or undefined, are used.
2. The score is increased for each break in the linear flow of the code. This includes any loop structure, conditionals (ternary operators, if, ...), switches and catches, sequences of logical operators, recursion and jumps to labels.
3. The score is also increased for nested flow-break structures. So if there is a conditional in a loop, the nesting is increased by one, so the score is already at two. [Cam18]

Cognitive complexity has been adopted as a promising metric to measure code understandability and seems to be the first code-related one [MWW20].
The calculation of the score is performed automatically by SonarQube. Since the analysis of the individual projects was already done when determining the code smells, the metric could be easily queried with the help of the SonarQube API. The number of lines of code was also already saved when the previous metric was obtained.

Percentage of Bug-Fix Commits

To measure bug proneness, the complete commit history of each repository was examined. The following was done with the commits returned from the GitHub API:

1. A counter was incremented for each commit.
2. If the commit additionally contained a bug-fix, it was saved. For this, the commit message was searched for “bug” and “fix”, which according to Zhang et al. is the most accurate method with the least false positives and a precision of 95% [ZLH+20]. However, after own research of the dataset, this confirmed the number with just less than 94% precision. The number of bug-fix commits was measured by a counter.

Finally, the number of bug-fix commits was divided by the number of total commits to calculate the percentage of bug-fix commits.
This process already happened during the selection of the samples. After this was completed, all commits were checked again for relevant files. That is, for each commit, all involved files were requested and the extension was checked. If .vue (Vue.js framework), .js (plain JavaScript), .jsx (library React), .ts (plain TypeScript) or .tsx (library React) was not included, the commit/bug-fix

counter was decreased. This step could not already be performed during the sampling process, as projects sometimes have several thousands commits. With the 5000 requests per hour limit, a project with 35,000 commits would therefore also make at least 35,000 requests to GitHub to check each file for its extension and would run for a minimum of seven hours. To prevent this step from taking so long, a total of six tokens were created with multiple accounts, each of which then took turns making the requests. This avoided any timeout due to the limited number of API calls.
After following the iterative sampling step and sorting out the non-valid projects for this study, this second step was performed. As a result, there are about 15 JavaScript as well as TypeScript applications that have less than 30 commits and are therefore not included in the data collection of the bug-fix commits ratio.

Average Time a Bug Issue is Open

To measure this metric, a feature of GitHub was accessed, the issue tracking system. This is designed to keep track of tasks, enhancements and, importantly for this study, bugs. For a better overview, the issues can be labeled, such as bug, front-end, back-end, etc. Each repository has the possibility to document such issues, so that everyone in the project has the same status. The issues can be assigned to developers and closed after completion. Through the GitHub API, the following information was extracted about the issues:

1. date of opening,
2. date of the last comment,
3. date of closing,
4. title and description,
5. label.

When requesting issues, it is important to note that the GitHub API also returns pull requests[13]. Pull requests are used when pushing code to a branch to show other developers in the project what has been changed. These also have a date when they were opened and a date when they were closed. Unfortunately, this information was not meant to be examined and can incorrectly sneak into the dataset.
Since not every issue describes a bug, only those that either have a bug label or include “bug” in the title or description were examined. The latter was done because, according to Bissyandé et al., only 30% of their dataset had a label [BLJ+13]. However, after a manual check of the own dataset, it turned out that 70.3% of the total bug issues have a label. This is a huge difference to Bissyandé et al., though they examined all issue types, not only bugs.
To measure the resolution time of the bug, inspired by Zheng et al. [ZMZ15], the time from the opening of the issue to the time from the last comment under the issue was calculated. After receiving the data, it turned out that issues were partially opened and directly closed again. One reason for this may be that the bug has already been fixed without opening an issue, but should be kept in the system. Therefore, issues that were opened for less than two minutes were deleted from

[13]Pull requests, `https://docs.github.com/en/github/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests`

the data. Furthermore, issues that were open for more than a year were also not considered, as they were probably forgotten and simply closed after a long time. After cleaning the data, the average of all bug resolution times of one project was calculated to make the metric comparable to other projects.
Since some JavaScript and TypeScript applications have very few issues, projects that have less than 5 were not included in the data collection of this metric. The reason for this is that with a very small number of issues, the average is dependent on only this few bugs and therefore the dataset can be skewed. Hence, a limit was set at 5 bug issues. It could not already be taken into account in the sampling, as this would otherwise limit the population size too much.

4.2.2 Retrieval of the Number of Used “any” Types per LoC

For the second research question, in addition to the collected metrics, the number of annotated “any” types was also needed. To determine the number, ESLint[14] was enabled for each TypeScript project using the “typescript-eslint” plugin. ESLint inspects the code for certain rules and issues a warning or an error if this rule is violated. Since using the “any” type is not recommended, there is the “no-explicit-any” rule.
To automate the retrieval of the data in the Python script, the following steps were performed sequentially for each TypeScript project:

1. Already existing ESLint configurations were deleted.
2. A prebuilt ESLint configuration with only one rule, “no-explicit-any”, was copied to the root of the repository.
3. ESLint and the TypeScript plugin were installed and then executed in the project root.
4. The output ESLint report was read and saved.

To make the values comparable between the projects, the number of “any” types used per line of code was calculated. Since this step is very time-consuming, the results were only sporadically checked manually. Furthermore, if a value appeared incorrect, ESLint was reinstalled and executed again.

4.2.3 Framework Identification

For the last research question, the data was sorted and analyzed according to the framework used for the development of the application. With a total of 604 projects, the detection of the framework cannot be done manually and it was automated in the Python script. Therefore, in the repositories already downloaded, the “package.json” file was examined. The “package.json” file is a kind of manifest that documents the configuration of the project, as well as the packages installed by the package manager npm[15] and yarn[16]. For front-end projects with JavaScript or TypeScript and

[14]ESLint, https://github.com/typescript-eslint/typescript-eslint
[15]npm, https://www.npmjs.com/
[16]Yarn, https://yarnpkg.com/

projects with Node.js[17] runtime, it is almost impossible to work without this file.
Thus, for the most part, the "package.json" file defines the framework used and was simple to read out automatically. If more than one framework was specified, it was manually checked which one was used. However, if this was not possible, the project was not examined further in the last research question. After assigning a framework to each project (if not, categorized under "other"), it was sporadically checked whether the script did the correct selection.

4.3 Hypotheses

This section again briefly recaps the research questions and defines its hypotheses. To get the connection to the metrics defined in section 4.2, these were also listed at the corresponding hypothesis.

RQ$_1$ Do TypeScript applications exhibit higher software quality than JavaScript applications?

Table 4.4 lists the null hypotheses and their alternative hypotheses. As can be seen, the complete collected data set of the four metrics was used to make a statement about the first research question.

Metric	Null hypothesis	Alternative Hypothesis
Code smells per LoC	H_0^1 TypeScrip applications show less or equal code quality than JavaScript applications.	H_1^1 TypeScript applications show better code quality than JavaScript applications.
Percentage of bug-fix commits	H_0^2 TypeScript applications are more or equally prone to bugs than JavaScript applications.	H_1^2 TypeScript applications are less prone to bugs than JavaScript applications.
Average time a bug issue is open	H_0^3 TypeScript applications take more or equal time in bug resolution than JavaScript applications.	H_1^3 TypeScript applications take less time in bug resolution than JavaScript applications.
Cognitive complexity per LoC	H_0^4 TypeScript applications show less or equal code understandability than JavaScript applications.	H_1^4 TypeScript applications show better code understandability than JavaScript applications.

Table 4.4: Null hypotheses with their alternative hypotheses for RQ1

The motivation behind the first hypothesis H_1^1 was to show that TypeScript applications have better code quality than JavaScript applications. However, JavaScript does not have the best reputation in terms of code quality. Reasons for this could be that the language is easy to learn and thus popular for beginners. Moreover, it is an interpreted and dynamically typed programming language, so it does not have a compiler that warns the developer about erroneous and unoptimized code. Therefore, code smells from JavaScript and the compiled TypeScript applications were investigated.

[17]Node.js, https://nodejs.org/en/

The second hypothesis H_1^2 has a look at the bug proneness of the two programming languages. The motivation behind this was that TypeScript finds errors at compile-time and, since JavaScript only prints errors at runtime or evaluates code to a strange value (see chapter 2), it was assumed that the language is less bug prone.

As with bug proneness, researchers have already been concerned with the bug resolution time between the dynamic and static programming languages. It is presumed that static typing makes the code self-documenting and therefore easier to find bugs. Hence, the motivation behind the third hypothesis H_1^3 is to show that TypeScript applications take less time in bug resolution than JavaScript applications.

The last hypothesis H_1^4 of the first research question focuses on code understandability and has a similar motivation as the first hypothesis. Especially with the dynamic, flexible and messy nature of JavaScript, it was assumed that TypeScript provides better code understandability.

RQ$_2$ Do TypeScript applications that less frequently use the "any" type show better software quality?

Table 4.5 defines the null hypotheses and the alternative hypotheses of the second research question. As can be seen in the research question, this hypotheses focused only on TypeScript projects. The data from the same four metrics were reused from the first research question. Since the use of the "any" type is associated with the loss of type safety, the motivation behind the four hypotheses was to see if software quality would decrease as a result. Therefore, it was checked whether there was a positive relationship between each of the metrics and the "any" type used.

Metric	Null hypothesis	Alternative Hypothesis
Code smells per LoC	H_0^5 The frequency of using the "any" type correlates not or positively with code quality in TypeScript applications.	H_1^5 The frequency of using the "any" type correlates negatively with code quality in TypeScript applications.
Percentage of bug-fix commits	H_0^6 The frequency of using the "any" type correlates not or negatively with bug proneness in TypeScript applications.	H_1^6 The frequency of using the "any" type correlates positively with bug proneness in TypeScript applications.
Average time a bug issue is open	H_0^7 The frequency of using the "any" type correlates not or negatively with time in bug resolution in TypeScript applications.	H_1^7 The frequency of using the "any" type correlates positively with time in bug resolution in TypeScript applications.
Cognitive complexity per LoC	H_0^8 The frequency of using the "any" type correlates not or positively with code understandability in TypeScript applications.	H_1^8 The frequency of using the "any" type correlates negatively with code understandability in TypeScript applications.

Table 4.5: Null hypotheses with their alternative hypotheses for RQ2

RQ$_3$ Does the chosen framework lead to software quality differences between TypeScript and JavaScript applications?

This research question is exploratory and therefore has no hypotheses. Nevertheless, the motivation behind it is explained here.

Examined were software quality differences in:

1. code quality,
2. bug proneness,
3. time in bug resolution,
4. code understandability.

There is little to no research on the topic of which framework delivers better software quality, as they are difficult to compare and each is tailored for a different type of application. Nonetheless, the differences in software quality between JavaScript and TypeScript within the frameworks Vue and Angular, and the library React are examined.

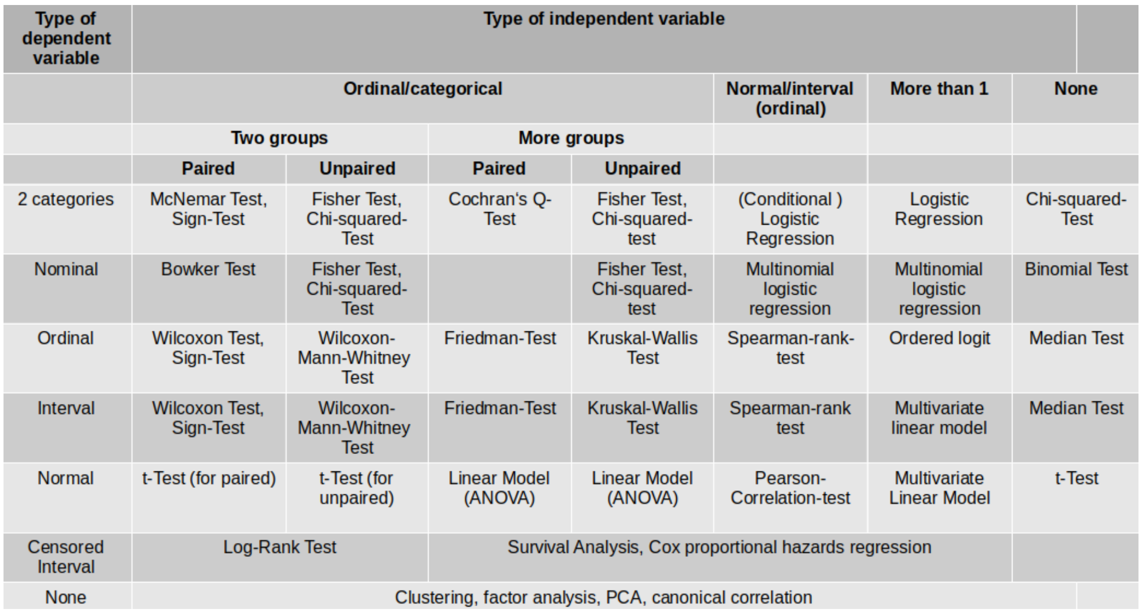
4.4 Statistical Analysis

The last part of figure 4.1, the statistical evaluation, is covered in this section.
Different statistical analysis methods were considered for each of the three research questions. A statistical significance level $p < 0.05$ was used, but reduced to 0.00625 by the Bonferroni correction, which is applied when several tests are performed in succession. Therefore, p was simply divided by the number of tests (i.e. hypotheses) [Arm14]. Thus, if p was less than 0.625%, the null hypothesis was rejected and the alternative hypothesis accepted.

4.4.1 Hypothesis Testing

The four hypotheses of the first research question were evaluated using hypothesis tests. Figure 4.2[18] gives an overview of the different hypothesis tests and helped to select the most appropriate one. The first step in determining the right test was to identify the type of the independent and dependent variables. JavaScript and TypeScript as independent variables are categories that are unpaired. More difficult to determine is the level of measurement of the dependent variable, i.e. the metrics. The first requirement that was checked was normality. If this would be satisfied, the parametric two-sample t-test could be performed, otherwise, a non-parametric one, which does not need this assumption [Bos13]. Therefore, the Shapiro-Wilk test[19] was integrated into the Python script. The test checks the null hypothesis, which states that the data set does not come from a normal distribution [HTZ16]. As input, a data set was given and as output, the test statistic and the p-value were returned. If the p-value was greater than 5%, the null hypothesis was rejected and the alternative hypothesis that the data set is normally distributed was accepted. This p was

[18]Overview of statistical tests, `https://www.crashkurs-statistik.de/welchen-statistischen-test-soll-ich-waehlen/`
[19]Python Shapiro-Wilk test, `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.shapiro.html`



Type of dependent variable	Type of independent variable						
	Ordinal/categorical				Normal/interval (ordinal)	More than 1	None
	Two groups		More groups				
	Paired	Unpaired	Paired	Unpaired			
2 categories	McNemar Test, Sign-Test	Fisher Test, Chi-squared-Test	Cochran's Q-Test	Fisher Test, Chi-squared-test	(Conditional) Logistic Regression	Logistic Regression	Chi-squared-Test
Nominal	Bowker Test	Fisher Test, Chi-squared-Test		Fisher Test, Chi-squared-test	Multinomial logistic regression	Multinomial logistic regression	Binomial Test
Ordinal	Wilcoxon Test, Sign-Test	Wilcoxon-Mann-Whitney Test	Friedman-Test	Kruskal-Wallis Test	Spearman-rank-test	Ordered logit	Median Test
Interval	Wilcoxon Test, Sign-Test	Wilcoxon-Mann-Whitney Test	Friedman-Test	Kruskal-Wallis Test	Spearman-rank test	Multivariate linear model	Median Test
Normal	t-Test (for paired)	t-Test (for unpaired)	Linear Model (ANOVA)	Linear Model (ANOVA)	Pearson-Correlation-test	Multivariate Linear Model	t-Test
Censored Interval	Log-Rank Test		Survival Analysis, Cox proportional hazards regression				
None	Clustering, factor analysis, PCA, canonical correlation						

Figure 4.2: Overview of statistical tests

independent of the significance level chosen above.
After testing each data set of each metric for normality, none were found to be normally distributed. Therefore, the t-test could not be considered and the non-parametric Mann-Whitney U test[20] was chosen as a substitute (see figure 4.2). As a requirement, at least 10 samples per category were needed [Bos13]. The test investigated whether the distribution underlying the data of TypeScript is the same than the distribution underlying the data of JavaScript. As input, Python received the data sets from JavaScript and TypeScript and the parameter "alternative", which specifies the direction of the hypotheses. The output was the test statistic and the p-value, which makes a statement about the hypotheses.
In addition to the significance calculation, the effect size was also calculated, which represents the magnitude of the difference between the two data sets. To obtain the effect size, Cohen's d is calculated by an online tool[21] based on the test statistic (U-Value). The interpretation of the value is as follows:

1. **< 0.5** - small effect
2. **0.5 - 0.8** - medium effect
3. **> 0.8** - large effect

[20]Mann-Whitney U test, `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html`
[21]Cohen's d calculator Formula 11, `https://www.psychometrica.de/effect_size.html`

4.4.2 Correlation Tests

The second research question investigated whether the software quality of TypeScript applications decreases as the number of “any” types used per LoC increases. Besides scatter plots, the statistical estimation of the relationship and its significance was tested by the Spearman correlation test. The Pearson correlation test was not applied, as normal distribution is required to assess significance [AVM02]. Some studies claim that the test is robust to a violation of normality [HP76], however, others are rather critical of this [Kow72].
Therefore, Python was used to calculate the non-parametric Spearman correlation coefficient[22], since the normal distribution was not assumed by either data set. As input, the script received the two data sets, first the metric and second the number of used “any” types per LoC, and the parameter “alternative”, which specifies the direction of the hypotheses. Returned is the correlation coefficient and the p-value, which makes a statement about the hypotheses. The coefficient varies between -1 and 1, where 0 implies no correlation, 1 and -1 an exact monotonic relationship [AVM02]. A positive number means that if the number of “any” types used per LoC increases, the metric values also increases, a negative number vice versa.

4.4.3 Descriptive Evaluation

The third research question is exploratory and sorted the collected data on software quality according to the framework used. Therefore, to get a good overview of the data descriptive statistics with box plots were used to compare the software quality between JavaScript and TypeScript within the frameworks. If the data showed an irregularity to the results of the first research question, tests for significance were performed. For this, the Mann Whitney U test, described in subsection 4.4.1, was performed.

[22]Spearman correlation coefficient, `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.spearmanr.html`

5 Analysis and Results

In this chapter, each research question is addressed and the statistically analyzed data is shown. At the end of the first two sections there is a small summary of the results of the hypotheses for a better overview. The collected data can be found behind the link[1].

5.1 RQ1: Do TypeScript applications exhibit higher software quality than JavaScript applications?

The research question includes four hypotheses that were validated through hypothesis tests. For each hypothesis, the analyzed data were given in tables and presented in box plots for a first overview.

5.1.1 H_1^1 TypeScript applications show better code quality than JavaScript applications.

To validate the first hypothesis, the code smells per line of code were examined to make a statement about the code quality. Some of the data collected can be found in table 5.1. For a better overview of the data, box plots for the two programming languages are shown in figure 5.1.

Language	Projects	Code Smells	Lines of Code	Mean	Median
JavaScript	299	217,957	7,496,726	0.025545	0.020132
TypeScript	305	95,132	8,683,600	0.013368	0.011143

Table 5.1: Code smells (CS) per LoC results

Directly noticeable in the box plots is that TypeScript has a lower median and a lower scatter of the data. Furthermore, JavaScript has significantly stronger outliers and a left-skewed distribution.
In total, 313,098 code smells were detected in the 604 applications. Thereby, an interesting fact is that JavaScript has more than twice as many code smells compared to TypeScript with more than one million lines less code. As a result, JavaScript also has an average of about 12 code smells per 1000 lines of code more.
As the box plots and the difference of the median of about 0.009 code smells per line of code more for JavaScript suggests, the assumption behind the alternative hypothesis could prove true. However, to make a significant statement, the Mann-Whitney U test was performed. The result was a p-value

[1]Collected data, https://figshare.com/articles/dataset/Collected_Data/16958344

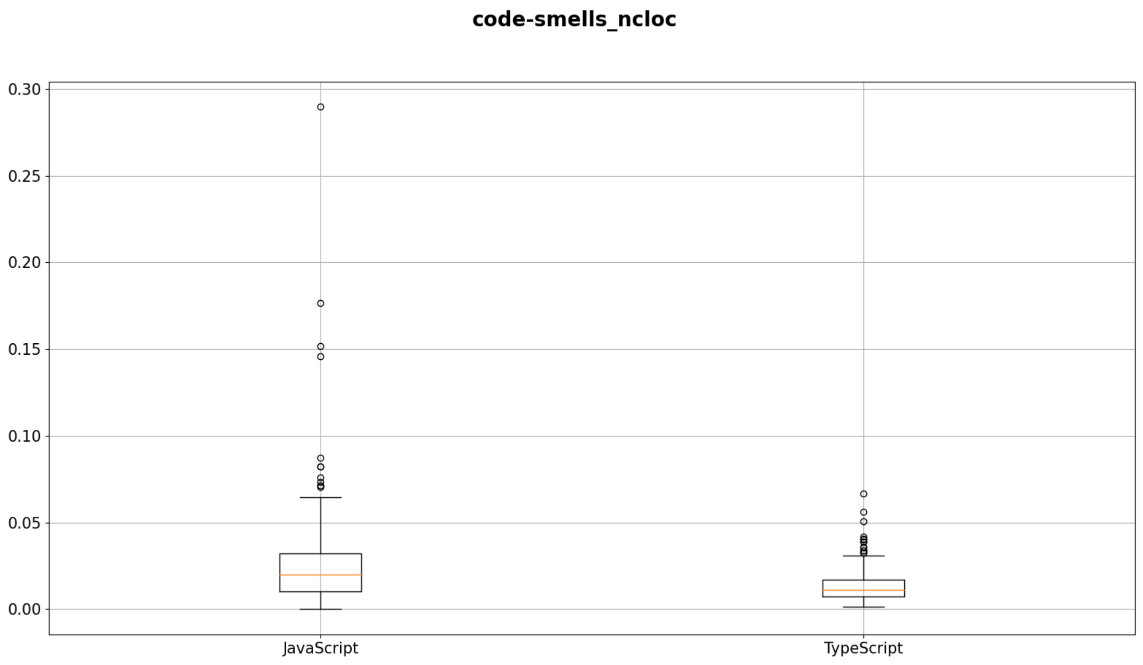


Figure 5.1: Box plots of code smells per LoC

of 2.7e-15, which is smaller than the adjusted significance level of 0.00625. Therefore the null hypothesis can be rejected and alternative hypothesis is accepted. Cohen's d yields a value of 0.671, which is a medium effect of the difference between the two data sets.

Result 1: TypeScript applications show better code quality than JavaScript applications.

5.1.2 H_1^2 TypeScript applications are less prone to bugs than JavaScript applications.

This hypothesis has looked at the commit history and checked what ratio of them contained bug-fixes to address the bug proneness. Parts of the measured data are presented in table 5.2 and in the box plots in figure 5.2. As already described in the data collection in subsection 4.2.1, not all projects could be included in the validation of this hypothesis.

Language	Projects	Commits	Bug-Fix Commits	Mean	Median
JavaScript	285	235,535	30,717	0.126364	0.109375
TypeScript	288	340,164	66,488	0.206436	0.163461

Table 5.2: Bug-fix commit ratio results

The comparison of the boxplots directly shows the lower median of the JavaScript applications and a larger scatter of the data for the TypeScript applications. Furthermore, both have quite a few outliers, however, TypeScript has more.

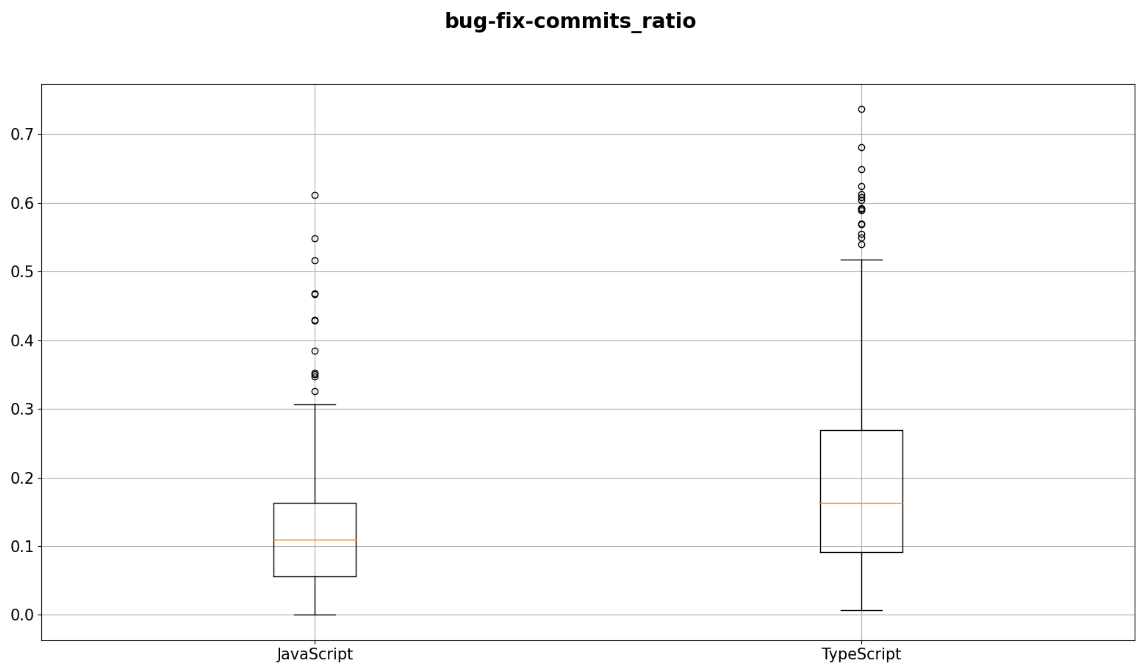


Figure 5.2: Box plots of bug-fix commit ratio

575,699 commits, which include only JavaScript and TypeScript files, were examined in total, of which 97,205 contained a bug-fix. After reviewing the data from the table, it turns out that TypeScript has a third more commits, although more than twice as many bug fixes. For JavaScript applications, on average one in eight commits contains a bug-fix and thus has 8% fewer bug fixes than the TypeScript applications.

The lower median of the bug-fix commit ratio from the JavaScript applications of 0.109375 indicates that the idea behind the alternative hypothesis does not prove true, however, the Mann-Whitney U test had to be performed again. The result was a p-value of 0.99, which is clearly above the significance level of 0.00625, and thus the null hypothesis cannot be rejected.

Result 2: TypeScript applications are more or equally prone to bugs than JavaScript applications.

5.1.3 H_1^3 TypeScript applications take less time in bug resolution than JavaScript applications.

The third hypothesis examined the average time a bug issue was open and thus makes a statement about the bug resolution time. An excerpt of the collected data can be found in table 5.3 and in the box plots in figure 5.3. Again, the complete data set was not used, as already described in the data collection in subsection 4.2.1

When directly comparing the box plots, it is not clear that TypeScript applications have a lower bug resolution time than JavaScript applications. JavaScript shows a slightly lower median, but they both have about the same distribution of data. TypeScript, however, shows stronger outliers. Since the evaluation of the box plots does not reveal a lot, the easier to read values in the table and the hypothesis test provide more accurate insights into the bug resolution time.

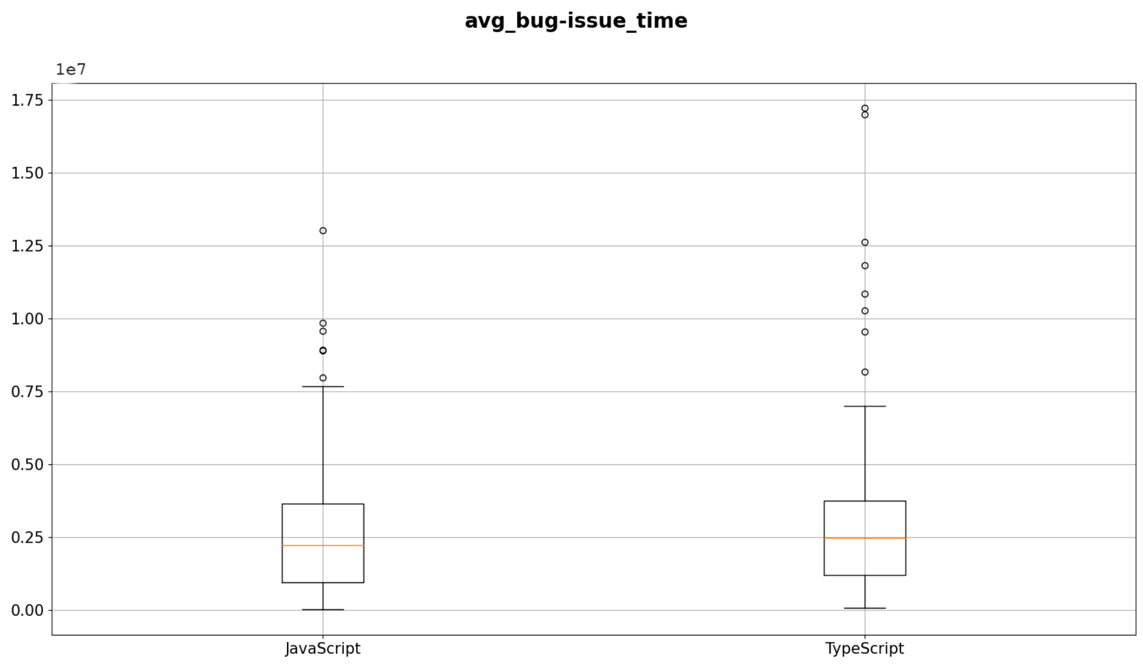


Figure 5.3: Box plots of average time a bug issue is open in seconds

Language	Projects	Bug Issues	Mean	Median
JavaScript	183	15,894	31.86d	25.59d
TypeScript	245	51.817	33.04d	28.49d

Table 5.3: Bug resolution time results

Of the total 214,075 issues collected (see table 4.2), 67,711 contain a bug. It is very interesting to notice that the feature of GitHub to document the issues seems to be used a lot more by TypeScript projects, as they documented almost three times as many issues and therefore more than three times as many bug issues as JavaScript projects. JavaScript developers needed on average 31.86 days to fix an issue, which is more than one day shorter than TypeScript developers. The median again indicates a false assumption in the alternative hypothesis, though only the Mann-Whitney U test provides certainty. As a result, a p-value of 0.75 was returned from the test which is greater than the significance level of 0.00625, therefore the null hypothesis cannot be rejected.

Result 3: TypeScript applications take more or equal time in bug resolution than JavaScript applications.

5.1.4 H_1^4 TypeScript applications show better code understandability than JavaScript applications.

The final hypothesis of the first research question addressed code understandability by examining cognitive complexity per line of code. An overview of the data collected is given in table 5.4. This data is also shown in the box plots in figure 5.4 and are first analyzed descriptively.

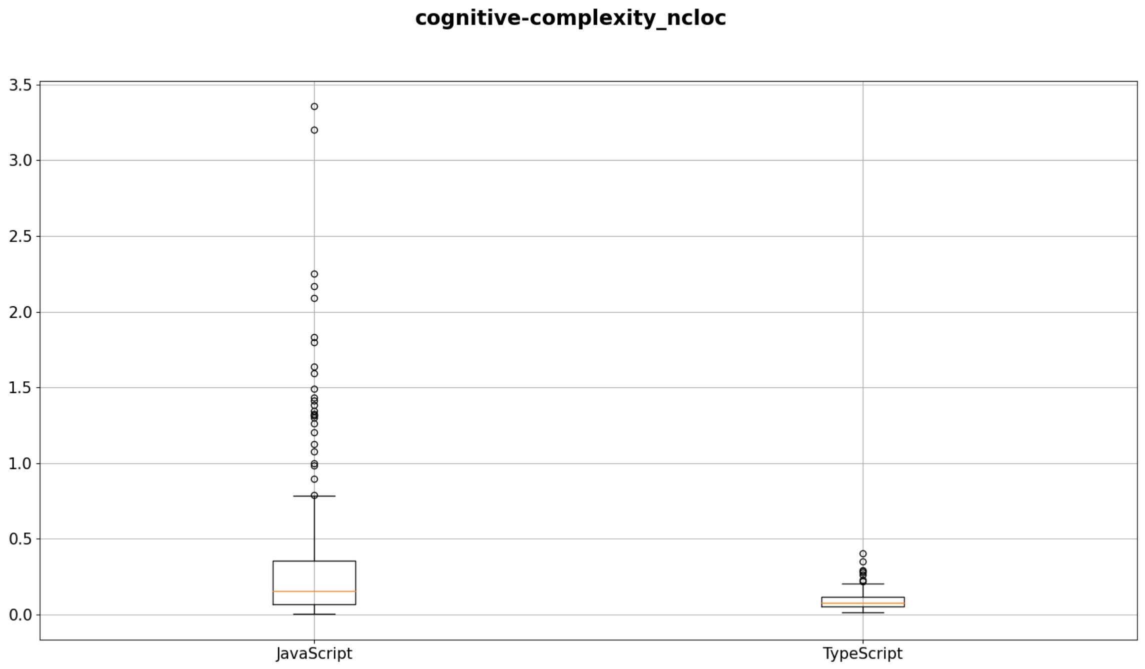


Figure 5.4: Box plots of cognitive complexity per LoC

Language	Projects	CC Score	Lines of Code	Mean	Median
JavaScript	299	3,435,760	7,496,726	0.321999	0.157013
TypeScript	305	722,687	8,683,600	0.089552	0.077416

Table 5.4: Cognitve complexity (CC) per LoC results

The box plots show a lower median and a very small scatter in the data from the TypeScript applications. In contrast, JavaScript applications have a higher scatter and a lot of strong outliers.

With over 16 million lines of code, the total cognitive complexity score was over 4 million. Thereby, a cognitive complexity score almost five times as high at less than one million lines of code, JavaScript clearly shows a higher value than TypeScript. This is also reflected in the mean value. TypeScript has an average score of 90 per kLoC, whereas JavaScript, with 322 per kLoC, has more than three times as much. A twice as high median of JavaScript indicates a correct assumption in the alternative hypothesis. The Mann-Whitney U test proved this with a p-value of 5.14e-18, which is lower than the significance level of 0.00625 and thus, the null hypothesis was rejected. However, Cohen's d yields a value of 0.339, which is a small effect of the difference between the two data sets.

Result 4: TypeScript applications show better code understandability than JavaScript applications.

5.1.5 Summary RQ1

Table 5.5 shows a summary of the collected values of the hypothesis tests. In addition to the returned values, the test statistic and the p-value, of the Mann-Whitney U test, the related alternative hypotheses are listed and whether they were accepted. For those that were accepted, the effect size (d_{Cohen}) is also provided. Ultimately, of the first research question, two of the four null hypotheses were rejected and the corresponding alternative hypothesis was accepted.

	Mann-Whitney U test			
Alternative Hypothesis	**Test Statistics**	**p-Value**	**d_{Cohen}**	**Accepted**
$\mathbf{H}_1^1$ TypeScript applications show better code quality than JavaScript applications.	28842.5	2.78e-15	0.671	Yes
$\mathbf{H}_1^2$ TypeScript applications are less prone to bugs than JavaScript applications.	54395.5	0.99	-	No
$\mathbf{H}_1^3$ TypeScript applications take less time in bug resolution than JavaScript applications.	23281.0	0.75	-	No
$\mathbf{H}_1^4$ TypeScript applications show better code understandability than JavaScript applications.	27219.0	5.14e-18	0.339	Yes

Table 5.5: Overview of hypothesis test results

5.2 RQ2: Do TypeScript applications that less frequently use the “any” type show better software quality?

This research question deals with the correlation between the individual metrics and the number of “any” types used. To get a general overview of the data, a scatter plot was created for each of the four hypotheses. However, since the Spearman test converts the values into ranks, the correlation coefficient cannot be interpreted geometrically and there is no need to pay attention to outliers. Therefore, the scatter plots are only there for a rough overview.

Table 5.6 shows the collected values of the “any” types used. In total 79,735 times the “any” type was measured by ESLint, the average per project is 261. However, this value does not reveal a lot, because the dependency to the lines of code is missing. This yields, on average, one variable notation with the type “any” per 100 lines of code. For the fact that the use of the “any” type is not advised, the highest measured number is rather large, with one annotation in every tenth line of code. Nevertheless, this seems to be an outlier, as the median shows only seven uses of the “any” type per kLoC.

	Minimum	Maximum	Mean	Median	Total
Used “any”-types	0	5326	261.43	70	79,735
Used “any”-types per LoC	0	0.106281	0.010300	0.007169	-

Table 5.6: Overview number of “any”-types used

5.2.1 H_1^5 The frequency of using the “any” type correlates negatively with code quality in TypeScript applications.

To validate this hypothesis, it was investigated whether the code quality correlates negatively with the number of “any” types used. In other words, it is checked whether the number of code smells used per line of code correlates positively with the number of “any” types used. To get an overview of the data, they are plotted in a scatter plot in figure 5.5.

As can be seen from the plotted line in the scatter plot, there seems to be a weak correlation, but first the Spearman correlation test had to be done. The p-value returned was 2.5e-06 and a correlation coefficient of 0.26. With a p-value lower than the specified significance level of 0.00625, the null hypothesis was rejected and the alternative hypothesis was accepted. The correlation coefficient, though, showed only a weak to moderate positive correlation.

Result 5: The frequency of using the “any” type correlates negatively with code quality in TypeScript applications.

5.2.2 H_1^6 The frequency of using the “any” type correlates positively with bug proneness in TypeScript applications.

This hypothesis tested whether there is a relationship between the bug-fix commit ratio and the number of “any” types used to make a statement about the bug proneness. Again, a scatter plot was created for the overview of the data, which can be seen in figure 5.6.

By the plotted line in the scatter plot it could be assumed that there is even a slight negative correlation and therefore the alternative hypothesis could not be accepted. However, by performing the Spearman correlation test that returned a p-value of 0.77 and a correlation coefficient of -0.04, confirmed this assumption to be correct. Therefore, there is not enough evidence to reject the null hypothesis.

Result 6: The frequency of using the “any” type correlates not or negatively with bug proneness in TypeScript applications.

5.2.3 H_1^7 The frequency of using the “any” type correlates positively with time in bug resolution in TypeScript applications.

The correlation of bug resolution time was validated in the third hypothesis of the second research question by checking whether the average time a bug issue is open has a positive relationship with the number of “any” types used. The scatter plot with the plotted values can be found in figure 5.7.

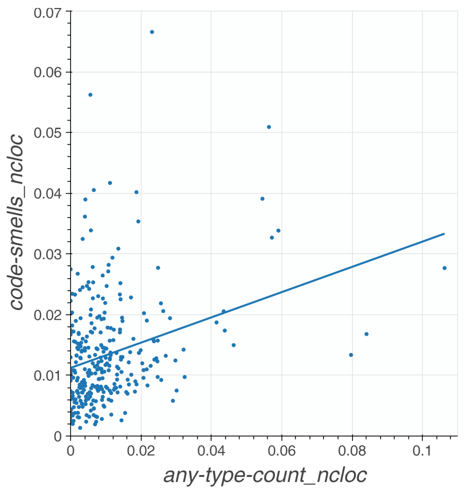


Figure 5.5 Scatter plot H_1^5

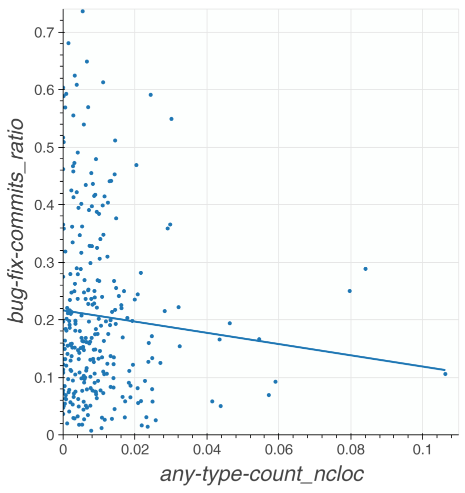


Figure 5.6 Scatter plot H_1^6

In this case, the line drawn in the scatter plot indicates a slightly positive correlation. As mentioned above, there is no geometric statement possible, so the Spearman correlation test was performed here as well. The result was a p-value of 0.0034 and a correlation coefficient of 0.17. Due to the adjusted significance level of 0.00625, the null hypothesis can still be rejected and the alternative hypothesis can be accepted. However, the coefficient indicates that the two values have only a weak positive correlation.

Result 7: The frequency of using the "any" type correlates positively with time in bug resolution in TypeScript applications.

5.2.4 H_1^8 The frequency of using the "any" type correlates negatively with code understandability in TypeScript applications.

Good code understandability is indicated by a low score of the cognitive complexity per line of code; hence, the final hypothesis examined whether there is a positive relationship between the cognitive complexity per line of code and the number of "any" types used. The data recorded for this can be found in the scatter plot in figure 5.8.

By looking at the line in the scatter plot, a slight correlation can be seen. However, since only the Spearman correlation test can indicate this, it was performed. The result was a p-value of 4.7e-04 and a correlation coefficient of 0.19. Since the p-value is below the significance level of 0.00625, the null hypothesis was rejected and the alternative hypothesis was accepted. Nevertheless, the correlation coefficient reveals that there is only a weak positive correlation between the two values.

Result 8: The frequency of using the "any" type correlates negatively with code understandability in TypeScript applications.

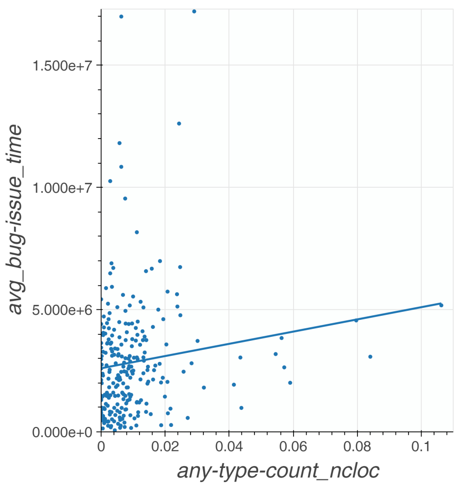


Figure 5.7 Scatter plot H_1^7

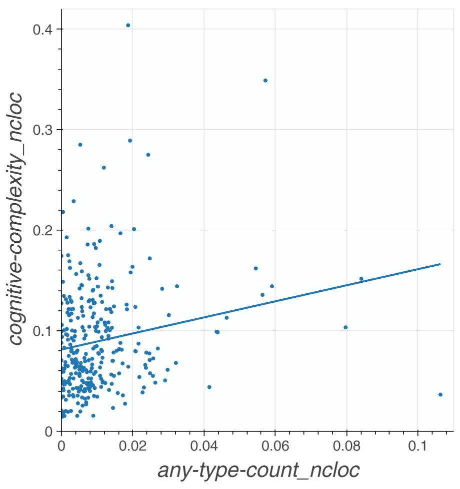


Figure 5.8 Scatter plot H_1^8

5.2.5 Summary RQ2

Table 5.7 shows all values from the Spearman correlation test and whether the alternative hypothesis was accepted in each case. Given these results, three of the four null hypotheses of the second research question were rejected and the corresponding alternative hypothesis was accepted.

	Spearman Test		
Alternative Hypothesis	**p-Value**	**Correlation Coefficient**	**Accepted**
H_1^5 The frequency of using the “any” type correlates negatively with code quality in TypeScript applications.	2.48e-06	0.26	Yes
H_1^6 The frequency of using the “any” type correlates positively with bug proneness in TypeScript applications.	0.76	-0.04	No
H_1^7 The frequency of using the “any” type correlates positively with time in bug resolution in TypeScript applications.	0.0034	0.17	Yes
H_1^8 The frequency of using the “any” type correlates negatively with code understandability in TypeScript applications.	4.7e-04	0.19	Yes

Table 5.7: Overview of correlation test results

5.3 RQ3: Does the chosen framework lead to software quality differences between TypeScript and JavaScript applications?

In this research question, the collected data on software quality was sorted by the popular frameworks Angular and Vue and the library React. Since it is exploratory and has no hypotheses, a boxplot was created for each framework from each language and then they were compared side by side. In case of any subsequent irregularities to the first research question, which generally examined JavaScript applications and TypeScript applications for software quality, an extra test for significance was performed.
Table 5.8 shows the distribution of the frameworks/libraries used in the projects. Interesting here is that React was by far the most used for JavaScript as well as TypeScript applications. The large difference in the number of uses of Angular between the two languages is because the new development from version 1 to version 2 in 2016 changed the support from JavaScript to TypeScript. However, Vue and Ember were used quite rarely.

Language	React	Angualar	Vue	Preact[2]	Ember[3]	Backbone[4]	Other
JavaScript	107	15	13	6	4	7	147
TypeScript	178	65	20	3	2	0	37

Table 5.8: Overview frameworks used for application development

5.3.1 Software Quality Differences in Code Quality

First, the difference of code quality between JavaScript and TypeScript within the frameworks was examined. In addition to the box plots in figure 5.9, the average and median are also listed in table 5.9 to make it easier to read the exact data. Values highlighted in italics indicate the lowest value between the frameworks and the programming languages.

	Mean CS per LoC			Median CS per LoC		
Language	**React**	**Angular**	**Vue**	**React**	**Angular**	**Vue**
JavaScript	0.013443	0.029708	*0.011409*	0.011796	0.019089	*0.006548*
TypeScript	0.012235	0.013275	0.016396	0.010393	0.011065	0.015125

Table 5.9: Overview of frameworks with code smells per LoC as metric

As can be seen in the box plots and the table, the median varies a lot among the JavaScript frameworks. However, JavaScript combined with Vue has the lowest measured median with 6.5 code smells per kLoC. Interestingly, this median is also clearly lower than the median of TypeScript combined with Vue with 15 code smells per kLoC. A little recap, from research question one

[2]Preact, https://preactjs.com/
[3]Ember.js, https://emberjs.com/
[4]Backbone.js, https://backbonejs.org/

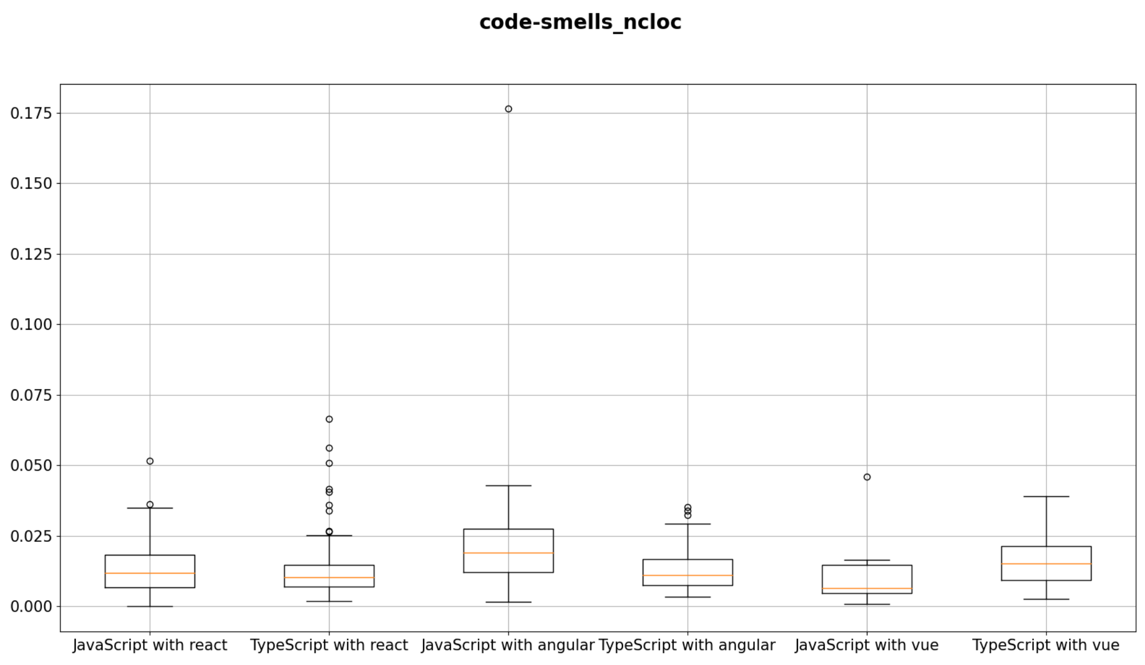


Figure 5.9: Box plots of frameworks with code smells per LoC as metric

it is known that TypeScript applications show less code smells per line of code than JavaScript applications. Therefore, the Mann-Whitney U test was performed between the Vue datasets of the two programming languages to see if they contradict the result of the first hypothesis. Since this is an exploratory research question and no longer related to the hypotheses of the first two research questions, a significance level of 0.05 was used here instead of the adjusted one of 0.00625. The test returned a p-value of 0.043 (d_{Cohen} = 0.632), meaning that the JavaScript applications that used Vue returned significantly fewer code smells per line of code than TypeScript applications that used Vue. This is a contradiction to the first hypothesis. From this, it can be deduced that other frameworks with JavaScript deliver a lot more code smells per line of code and Vue is an exception. Angular projects have with 19 Code smells per kLoC the highest median among the JavaScript applications. Through an extreme outlier also, among others, a very high average, and thus also eight code smells per kLoC more than the TypeScript applications with Angular. For React, the median (JavaScript 0.011796, TypeScript 0.010393) is quite balanced between the two languages with only a bit more than one code smell per kLoC more for JavaScript applications. TypeScript applications using React have a lot of outliers, nevertheless still the smallest median among the frameworks of the programming language.

5.3.2 Software Quality Differences in Bug Proneness

The differences in bug proneness, i.e. the bug-fix commit ratio, between JavaScript and TypeScript within the frameworks were examined. The data obtained can be seen in the box plots in figure 5.10 and table 5.10.

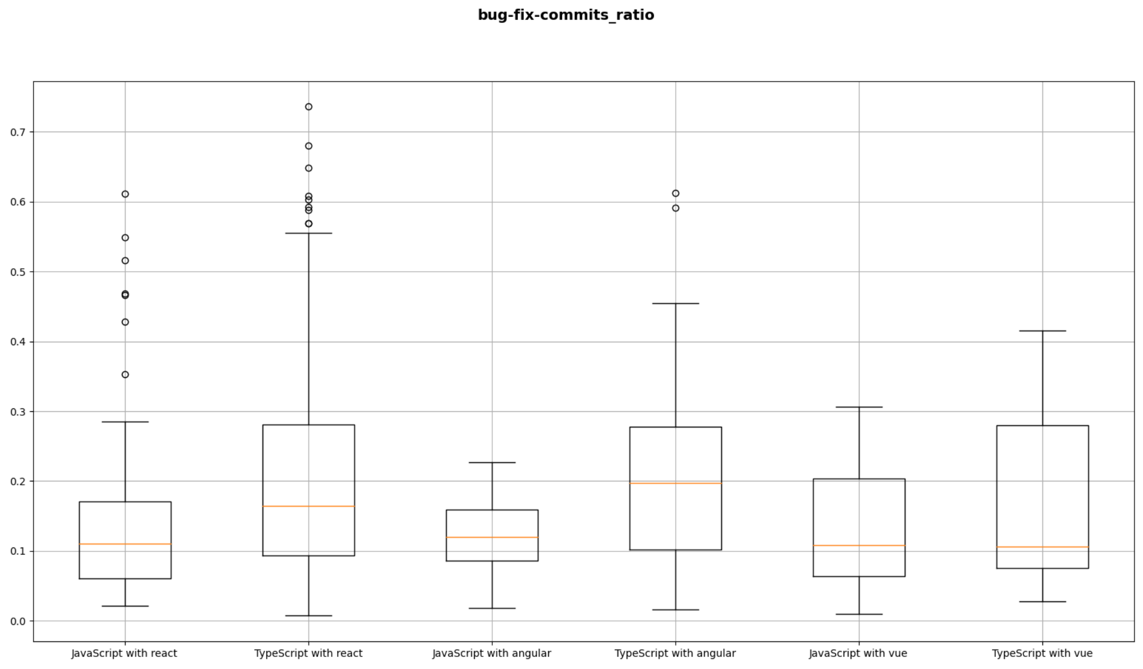


Figure 5.10: Box plots of frameworks with bug-fix commit ratio as metric

	Mean			Median		
Language	**React**	**Angular**	**Vue**	**React**	**Angular**	**Vue**
JavaScript	0.139394	*0.121550*	0.132907	0.110243	0.120116	0.108229
TypeScript	0.214683	0.213572	0.172017	0.164717	0.197565	*0.106430*

Table 5.10: Overview of frameworks with bug-fix commit ratio as metric

What becomes visible in the direct comparison of the box plots between the respective frameworks of the two programming languages is that JavaScript applications mostly have a lower median than TypeScript applications, except for Vue, there they have a minimally higher one. By the validation of the second hypothesis, it was found that TypeScript applications are equally or more bug prone than JavaScript applications. This is also reflected in the box plots and none of the frameworks contradict it. Therefore, no further tests were necessary.
On average, Angular with JavaScript has the lowest bug proneness with 12% bug-fix commits, however, the lowest median has Vue with TypeScript. With only a very small difference of 0.2%, JavaScript has almost the same. The difference of the median of the bug fix commit ratio is roughly the same for React (JavaScript 11%, TypeScript 16%) and Angular (JavaScript 12%, TypeScript 19%) with once 5% and the latter almost 7%. However, it looks different for the average, where TypeScript applications have a much higher ratio, which is due to their left-skewed distribution. The highest median among the frameworks is seen in Angular for both programming languages, with a ratio of 12% for JavaScript and 19% for TypeScript. Overall, React shows many outliers for both programming languages. It is also interesting to note that the scatter is very small when Angular is used with JavaScript.

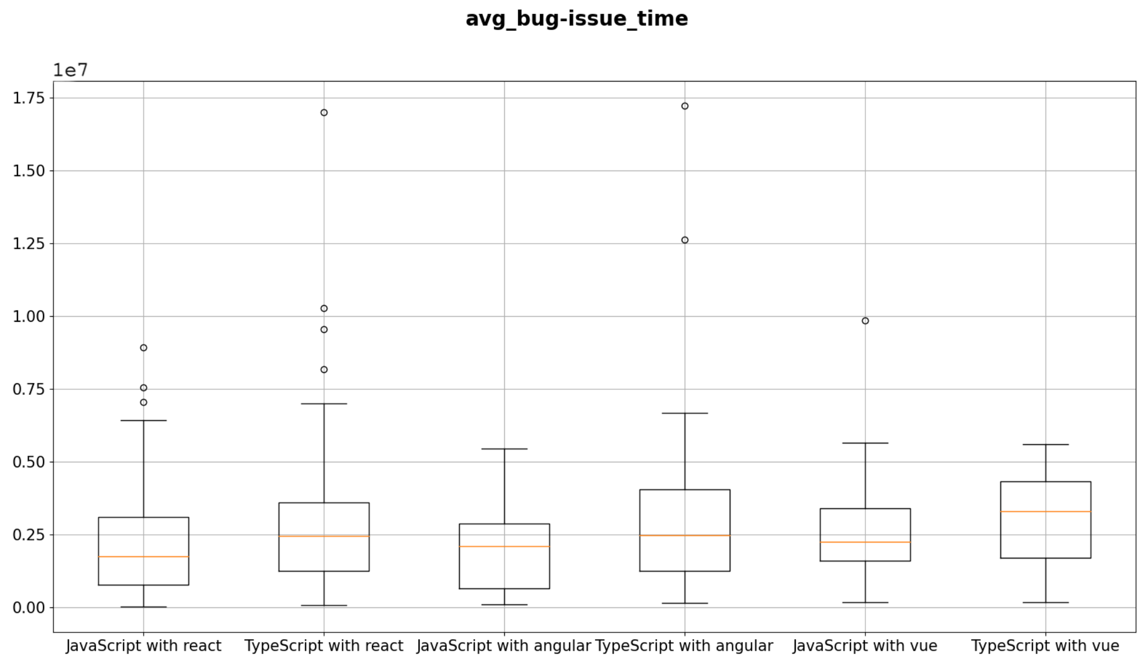


Figure 5.11: Box plots of frameworks with average time a bug issue is open in seconds as metric

5.3.3 Software Quality Differences in Time in Bug Resolution

In the box plot in figure 5.11, the categorized data on bug resolution time, i.e., the average time a bug issue is open, is presented by framework and language in box plots. To make it easier to read the data, it is also listed in table 5.11.

	Mean			**Median**		
Language	**React**	**Angular**	**Vue**	**React**	**Angular**	**Vue**
JavaScript	27.01d	*23.45d*	36.87d	*20.06d*	24.20d	25.88d
TypeScript	31.23d	35.95d	33.69d	28.37d	28.51d	38.09d

Table 5.11: Overview of frameworks with average time a bug issue is open as metric

The direct comparison of the box plots shows that JavaScript applications with the framework used for each have a lower median than TypeScript applications. Thus, there is no contradiction to the third hypothesis, which concluded that TypeScript applications need the same or more time in bug resolution time than JavaScript applications.
Angular has the lowest average time to fix bugs with JavaScript at 23.45 days. In contrast, developers using TypeScript take on average twelve days longer. A strong difference in fixing time can be seen especially in Vue. Bugs can be fixed in JavaScript applications with a median of 25.88 days within 13 days faster than TypeScript applications. However, in contrast, it is interesting to note that the average for TypeScript applications using Vue is three days lower than for JavaScript applications. This is because the TypeScript dataset is right-skewed and the JavaScript dataset is a bit left-skewed. JavaScript combined with React has the lowest median at just over 20 days; TypeScript has over

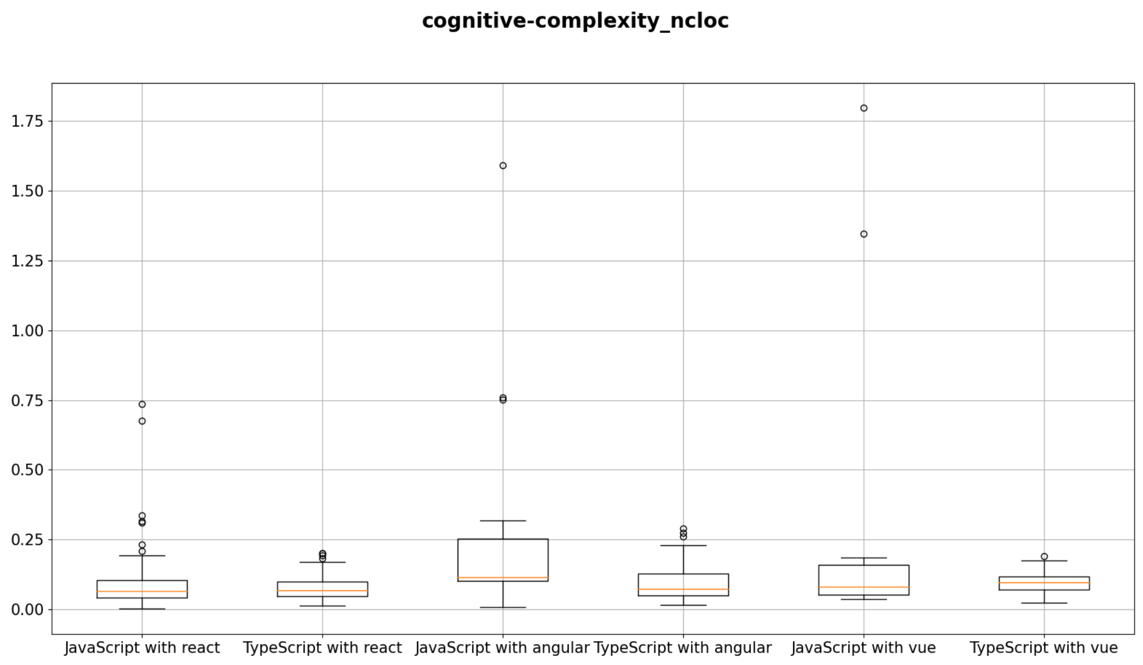


Figure 5.12: Box plots of frameworks with cognitive complexity per LoC as metric

eight days longer to fix. The highest median among the frameworks is seen for Vue with both programming languages. Noticeable in the box plots is that TypeScript has some stronger outliers with React and Angular.

5.3.4 Software Quality Differences in Code Understandability

The last metric that was compared between JavaScript and TypeScript within the frameworks was the code understandability, i.e. the cognitive complexity per line of code. The data can be easily compared in the box plot in figure 5.12, but for a better overview, they are also listed in table 5.12.

	Mean			**Median**		
Language	**React**	**Angular**	**Vue**	**React**	**Angular**	**Vue**
JavaScript	0.092589	0.304296	0.314373	*0.065919*	0.114772	0.082206
TypeScript	*0.076910*	0.094257	0.099249	0.068910	0.073144	0.098030

Table 5.12: Overview of frameworks with cognitive complexity per LoC as metric

Surprisingly, the median for React and Vue is lower for JavaScript applications than for TypeScript applications. However, by running Mann-Whitey U tests, it was found that the cognitive complexity score per line of code for neither framework is significantly lower for JavaScript than for TypeScript applications. One reason for this could be seen in the average of the respective frameworks, which is much higher for JavaScript applications, especially for Vue, than for TypeScript applications.
Overall, React shows the lowest median with JavaScript with a score of 66 per kLoC, but only with a small difference of 3 per 100 LoC to TypeScript. Angular shows a slightly higher median with

TypeScript with a score of 0.073144 per LoC, hence lower than JavaScript by 41 per kLoC. The average, on the other hand, for JavaScript with Angular and Vue is about three times higher than the median in each case, which is again due to the left-skew. Therefore, the median for JavaScript with Vue is also lower than TypeScript with Vue, the former has a value of 8.2 per 100 LoC, the latter 9.2. Lastly, what can be deduced from the box plots is that JavaScript has larger outliers for each of the frameworks and the data of TypeScript projects has a very low scattering.

6 Discussion

This chapter discusses the collected results by addressing each of the research questions with their respective hypotheses.

RQ1: Do TypeScript applications exhibit higher software quality than JavaScript applications?

A rather similar result between the two languages was found when examining the bug resolution time, whereas TypeScript applications did not show a lower value than JavaScript applications. One possible reason for this could be that the two programming languages have a lot of similarities and that static typing does not self-document the code well enough to find bugs faster. Interestingly, TypeScript developers used the feature to document issues on GitHub much more often than JavaScript developers. This suggests a skill difference, as using this feature can help further in the development process and is used less by beginners.

Contrary to expectations, TypeScript applications are not less bug prone than JavaScript applications. There could be several reasons for this. First, by compiling TypeScript, the type-related bugs are found earlier than with JavaScript and might therefore not be included in the statistics. Thus, TypeScript would have to be even more bug prone in other aspects. Second, the one-third higher number of commits implies that TypeScript developers committed more often and hence documented the changes more accurately. JavaScript developers committed less, which may also indicate that multiple bugs could be included in one commit. Hence, this could lead to an inaccurate lower error rate.

As expected, TypeScript applications delivered better code quality and understandability than JavaScript applications. One assumption was that the dynamic and flexible nature without a compiler that warns about erroneous and unoptimized code leads to more code smells and a higher cognitive complexity in JavaScript than in TypeScript. In addition, the difference in skills mentioned earlier could also become apparent here. If the GitHub feature was used more often by TypeScript developers, it could be the case that some of them already used static code analysis tools and therefore showed a better value.

Overall, two of the four alternative hypotheses of the first research question could be accepted. Two of the null hypotheses could unfortunately not be rejected, which does not imply better software quality in favor of TypeScript. Especially the hypothesis on bug proneness contradicts the data from Gao et al. [GBB17] (see related work 3.2). They found that TypeScript can prevent 15% of the bugs from JavaScript, but they did not study solely applications. Likewise, the not accepted hypothesis on bug resolution time contradicts the results of Zhang et al. [ZLH+20] (see related work 3.3). They evaluated their data and concluded that dynamically typed languages take 59.5% more time in bug resolution than statically typed languages. However, TypeScript is not present in their dataset and they did not study solely applications either.

RQ2: Do TypeScript applications that less frequently use the “any” type show better software quality?

The hypotheses of the second research question examined whether software quality is better when type safety is strongly adhered to than when it is only weakly adhered to. The strength of type safety was calculated by the number of “any” types used per line of code.

One assumption is that the static type system detects bugs earlier than a dynamic type system. However, the results of the second research question on bug proneness indicate otherwise, even a weak negative correlation. It can be concluded that bug proneness in TypeScript applications does not depend on type safety, i.e. the number of “any” types used.

A weak to moderate dependency between the metric and the strength of type safety was found for code quality and understandability and the bug resolution time. Thereby, the average time a bug issue is open, the number of code smells per line of code and the cognitive complexity per line of code increased when the “any” type was used more frequently. In addition to the effect of static typing, there could be other influences. Since the use of the “any” type is not recommended and should only be used in rare cases, there could be a skill difference between TypeScript developers. Many of them used tools like ESLint to avoid the use of this type. Then, it is also obvious to use static code analysis tools like SonarQube to keep the number of code smells and the cognitive complexity score low. The skill difference, and the related experience, may also have helped to fix the bugs faster. However, this phenomenon was not seen in bug proneness.

Since, to the best of our knowledge, there is no comparable study to this research question, the results cannot be compared. Overall, three of the four hypotheses could be accepted, which basically implies to use the “any” type as little as possible and to insist on type safety to get a better software quality. Nevertheless, this is not true for all quality characteristics.

RQ3: Does the chosen framework lead to software quality differences between TypeScript and JavaScript applications?

There has been very limited existing research on this question as well, since each framework is tailored for a different purpose. Thus, only the difference between the programming languages and the frameworks is interesting. However, there are no comparisons to other studies available either.

The data set of the first research question was categorized under the frameworks and compared in each case between the two programming languages. Interestingly, the results between the languages and the corresponding frameworks are the same as for the first research question, except for the code quality metric. Vue in combination with JavaScript shows significantly fewer code smells per LoC than Vue in combination with TypeScript. This is a contradiction to the first research question. For the other quality attributes, Vue in combination with JavaScript also shows better results than with TypeScript, except for error proneness, where both have about the same value. On the one hand, this could be caused by the very small sample size (JavaScript = 13, TypeScript = 20) and the collected samples might be exceptions. On the other hand, however, Vue as a framework might also compensate for the effects of TypeScript, such as static typing.

Another interesting value was found in the code understandability metric. In contrast to the first research question, React with JavaScript shows a marginally better median than React with TypeScript. However, this is not significant because the JavaScript data is left-skewed, which also leads to a higher average.

In summary, the data indicates that the frameworks do not have a major impact on the software quality of the programming language. To examine the outliers in Vue applications with JavaScript and to be able to make a statement about the slightly lower values, a larger sample needs to be analyzed.

7 Threats to Validity

There are many threats to validity in a large-scale study. Reasons for this include the unavoidable automation and the vast amount of data. It is important to describe these threats and explain how they were mitigated so that the results are credible and trustworthy. Furthermore, researchers can take them into account in replication studies or in further research. Therefore, the following points are listed where threats can occur and how attempts have been made to keep them to a minimum. The metrics bug-fix commit ratio, average time a bug issue is open, code smells per LoC and cognitive complexity per LoC were adopted from other studies and adjusted to incorporate the researchers' experiences and solutions to the same challenges.

Sample

Since the population size is unknown in such studies, it needs to be estimated. To get a rough overview and direction of the estimation, this was done during the sampling procedure, to subsequently calculate the sample size. However, after the procedure, only 300 applications were selected for each of the two programming languages, instead of the calculated 341 to 375. For TypeScript, almost all repositories that met the preconditions were searched for applications, so the initially calculated sample size did not fit; for JavaScript, the sample size was adjusted downwards after extrapolations. The sample size ultimately depends on how many valid samples the Python script outputs and if this is over-optimized, too many applications will be incorrectly sorted out. As mentioned in section 4.1.3, this was not the case. For TypeScript alone, more than 1200 samples were manually checked for the specified requirements, indicating that there was no over-optimization and thus, no sample size bias.

Samples were selected based on popularity. For TypeScript, this was not a problem, since there were projects from every popularity layer. For JavaScript, however, only the top 15% of the most popular repositories were examined. This is not representative of all applications. Nevertheless, this was a better approach than randomly selecting projects. Also in section 4.1.3, it was noted that the lower the popularity of the projects, the fewer applications that match the characteristics were included. Therefore, this would extend the long sampling process without changing the dataset much.

What cannot be avoided is the proportion of programming languages that are not JavaScript or TypeScript. For applications, it is not possible without HTML, CSS, etc., and often not without a back-end. When selecting the samples, the proportions of the programming languages were written down and shown in table 4.3 in section 4.1.3. For JavaScript and TypeScript, about 10% of the applications were with a back-end, which took more than 10% of the programming languages. This amount is small enough and should not affect the data set. If applications without a back-end were assumed, the population size would be too restricted.

The entire sample consists of open source projects, therefore, no statement about commercial applications can be made. This is a problem of such studies, since access is only possible to public source code on a large scale. Moreover, the number of developers, as well as their degree of skill, was not taken into account. For smaller studies, this is easier to control, for larger ones it is difficult to consider. The applications are all from the same interval of time, between 2012 and mid-2021, as TypeScript has been released since then. Further, around this initial time, the now well-known and widely used frameworks were introduced and launched. Therefore, no additional criteria were considered in the selection of the sample in terms of time. The distribution of the size of the projects was also not taken into account, nevertheless, the proportion of lines of code is rather balanced.

Data Collection

To measure code smells, SonarQube applies rules to the source code. Since the number of rules varies between programming languages, it could be assumed that this metric is not comparable. However, with 141 rules for JavaScript and 147 for TypeScript, the number is almost balanced. Therefore, the difference can be neglected, especially since TypeScript shows significantly fewer code smells per LoC than JavaScript.

Since JavaScript and TypeScript applications only had to have more than 60% as the primary language, commits and lines of code were only counted if the file with the corresponding language was also included.

The detection of bug-fix commits was automated by searching for “bug” and “fix” in the developer’s commit message. This method was adopted from Zhang et al. [ZLH+20] and then tested for validity. Therefore, 0.5% (500 commits) of the total of over 97,000 bug-fix commits were manually examined to determine the rate of false positives and how often more than one bug fix was included. 6% of the commits contained no bug/fix, and 8% contained more than one. However, this number is rather low and should not have a major impact on the results.

Another threat to validity lies in the measurement of bug resolution time. It is obvious to use the time from opening a bug issue to closing the bug issue. Nevertheless, it turned out that measuring the interval from the opening of the bug issue to the last comment below it is a more accurate method [ZMZ15]. Therefore, the latter was adopted for this study to determine the bug resolution time as accurately as possible. However, it is difficult to assess whether an incorrect time interval of bug resolution is included in the dataset, so reliance is placed on the results of Zheng et al. that this is the most accurate variant [ZMZ15]. This method has already been adopted by other studies, for instance, Zhang et al. [ZLH+20], and should minimize the number of incorrectly measured values. Thus, if there are still incorrect bug resolution intervals in the data set, they should not have an effect on the results.

Due to missing additional information, some bug-issues affecting the back-end were included. GitHub offered the possibility to link (and close) issues in a commit, which would show the connection to the code and, therefore, to the used programming language. However, this feature was rarely used and no reliable assignment was possible. Thus, the approximately 10% of applications that have a back-end were considered low enough that they would not affect the dataset.

For the detection of the number of used “any” types, ESLint with the “typescript-eslint” plugin was installed in the project root and then executed for all TypeScript applications. Since this step was very time-consuming, it was automated. To exclude errors, projects with strange values were manually checked, ESLint reinstalled and run again. For example, for repositories that ended up with a value of zero, the existing ESLint file was examined and mostly the “no-explicit-any” rule or the strict mode, which also contains it, was enabled. Since not every project could be examined manually, testing was only sporadic. However, these subsequent examinations confirmed the values. If there were undetected incorrect values in the data set, their number would be very small and would not significantly affect the results.

The detection of the framework was completely automated. For this, the “package.json” file was scanned for the most popular frameworks. If several frameworks were detected, these projects were manually examined and it usually turned out relatively fast which framework was used. After the identification was complete, projects were sporadically investigated to determine whether the detected framework was correctly assigned. The examination showed that the frameworks were always correctly assigned by the script. However, the number of incorrectly assigned frameworks to the non-examined applications should be close to zero and thus do not influence the results.

8 Conclusion

This thesis presented a large-scale repository mining study on GitHub, to show that TypeScript applications exhibit better software quality than JavaScript applications. Furthermore, a possible influencing factor on the software quality of TypeScript projects was investigated and whether the choice of the framework shows an influence on the two programming languages. To be able to make a statement about these properties, a total of 604 GitHub repositories with over 16 million lines of code were examined. Overall, three research questions were defined, in each of which the same four metrics were categorized and evaluated differently. All of them are indicators of software quality and were captured through features of GitHub and by statically analyzing the code with SonarQube and ESLint. A Python script was implemented to automate all steps from sample selection to data collection to statistical analysis.

The data suggests that TypeScript applications show better code quality than JavaScript applications because they have significantly fewer code smells. Furthermore, a significantly smaller score of the cognitive complexity per LoC showed that the code of TypeScript applications is better understandable than that of JavaScript applications. Contrary to expectations, the bug proneness, measured by the bug-fix commit ratio, and the bug resolution time, captured by the average time a bug issue is open, showed both a lower value for JavaScript projects than for TypeScript projects. Thus, for only two of the four metrics was there evidence to support that TypeScript applications exhibit higher software quality than JavaScript applications.
In general, it can be assumed for TypeScript applications that insisting on type safety leads to better software quality. This was true for three of the four metrics examined, namely code quality and understandability and bug resolution time. Nevertheless, there is not enough evidence to conclude this for bug proneness.
Lastly, no major additional differences in code quality due to the choice of framework could be identified. Only for the framework Vue in combination with JavaScript was a significantly better code quality found than for Vue with TypeScript, which is in contradiction to the conclusion above. Overall, however, it can be summarized that the framework used has only a modest influence on the software quality of the two programming languages.

In a large-scale study, further research with replications is important to minimize the threats to validity. Therefore, the source code and the collected data were provided. Further work could focus on the software quality between the programming languages among the frameworks. This is still largely unexplored and struggled in this study to some degree with a small sample size. Especially the contradiction of the framework Vue in combination with JavaScript would be interesting to investigate on a larger scale or in a controlled experiment. In addition, this study also did not resolve the long ongoing debate between static and dynamic typing, which exhibits better software quality, so further research is needed to shed more light on this topic.

Bibliography

[Arm14] R. A. Armstrong. "When to use the Bonferroni correction". In: 34.5 (Apr. 2014), pp. 502–508. DOI: 10.1111/opo.12131. URL: https://doi.org/10.1111/opo.12131 (cit. on p. 41).

[AVM02] R. Artusi, P. Verderio, E. Marubini. "Bravais-Pearson and Spearman correlation coefficients: meaning, test of hypothesis and confidence interval". In: *The International Journal of Biological Markers* 17.2 (2002), pp. 148–151. DOI: 10.5301/jbm.2008.2127 (cit. on p. 43).

[BBL76] B. W. Boehm, J. R. Brown, M. Lipow. "Quantitative evaluation of software quality". In: *Proceedings of the 2nd international conference on Software engineering*. 1976, pp. 592–605 (cit. on pp. 21, 22).

[BHM+19] E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek, J. Vitek. "On the Impact of Programming Languages on Code Quality: A Reproduction Study". In: *ACM Trans. Program. Lang. Syst.* 41.4 (Oct. 2019). ISSN: 0164-0925. DOI: 10.1145/3340571. URL: https://doi.org/10.1145/3340571 (cit. on pp. 16, 26, 31).

[BHV16] H. Borges, A. Hora, M. T. Valente. "Understanding the Factors That Impact the Popularity of GitHub Repositories". In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2016, pp. 334–344. DOI: 10.1109/ICSME.2016.31 (cit. on p. 32).

[BLJ+13] T. F. Bissyandé, D. Lo, L. Jiang, L. Réveillère, J. Klein, Y. L. Traon. "Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub". In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. 2013, pp. 188–197. DOI: 10.1109/ISSRE.2013.6698918 (cit. on p. 37).

[Bos13] S. Boslaugh. *Statistics in a Nutshell: a desktop quick reference*. OReilly, 2013 (cit. on pp. 41, 42).

[Cam18] G. A. Campbell. "Cognitive Complexity: An Overview and Evaluation". In: *Proceedings of the 2018 International Conference on Technical Debt*. TechDebt '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 57–58. ISBN: 9781450357135. DOI: 10.1145/3194164.3194186. URL: https://doi.org/10.1145/3194164.3194186 (cit. on p. 36).

[Che19] B. Cherny. *Programming TypeScript - Making Your JavaScript Applications Scale*. Sebastopol: Ö'Reilly Media, Inc.", 2019. ISBN: 978-1-492-03760-6 (cit. on p. 15).

[CJYF19] D. H. Curie, J. Jaison, J. Yadav, J. R. Fiona. "Analysis on Web Frameworks". In: 1362 (Nov. 2019), p. 012114. DOI: 10.1088/1742-6596/1362/1/012114. URL: https://doi.org/10.1088/1742-6596/1362/1/012114 (cit. on p. 23).

[Edw14] N. M. Edwin. "Software Frameworks, Architectural and Design Patterns". In: 07.08 (2014), pp. 670–678. DOI: 10.4236/jsea.2014.78061. URL: https://doi.org/10.4236/jsea.2014.78061 (cit. on p. 23).

[EM02] E. van Emden, L. Moonen. "Java quality assurance by detecting code smells". In: *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* 2002, pp. 97–106. DOI: 10.1109/WCRE.2002.1173068 (cit. on p. 35).

[EM12] E. Emden, L. Moonen. "Assuring software quality by code smell detection". In: Jan. 2012, pp. xix–xix. ISBN: 978-1-4673-4536-1. DOI: 10.1109/WCRE.2012.69 (cit. on p. 35).

[Fla20] D. Flanagan. *Javascript: The Definitive Guide - Master the World's Most-Used Programming Language*. Sebastopol: O'Reilly Media, 2020. ISBN: 978-1-491-95202-3 (cit. on p. 19).

[FM13] A. M. Fard, A. Mesbah. "JSNOSE: Detecting JavaScript Code Smells". In: *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2013, pp. 116–125. DOI: 10.1109/SCAM.2013.6648192 (cit. on pp. 16, 19).

[FM20] Y. Fain, A. Moiseev. *TypeScript Quickly*. Birmingham: Manning, 2020. ISBN: 978-1617295942 (cit. on p. 19).

[GBB17] Z. Gao, C. Bird, E. T. Barr. "To Type or Not to Type: Quantifying Detectable Bugs in JavaScript". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 2017, pp. 758–769. DOI: 10.1109/ICSE.2017.75 (cit. on pp. 16, 26, 61).

[Han10] S. Hanenberg. "An Experiment about Static and Dynamic Type Systems: Doubts about the Positive Impact of Static Type Systems on Development Time". In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '10. Reno/Tahoe, Nevada, USA: Association for Computing Machinery, 2010, pp. 22–35. ISBN: 9781450302036. DOI: 10.1145/1869459.1869462. URL: https://doi.org/10.1145/1869459.1869462 (cit. on pp. 16, 25, 26).

[Has08] A. E. Hassan. "The Road Ahead for Mining Software Repositories". In: Nov. 2008, pp. 48–57. DOI: 10.1109/FOSM.2008.4659248 (cit. on pp. 16, 17).

[HM16] G. Herzwurm, M. Mikusz. *Qualitätsmerkmale von Software*. Dec. 2016. URL: https://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/is-management/Systementwicklung/Management-der-Systementwicklung/Software-Qualitatsmanagement/Qualitatsmerkmale-von-Software/index.html (cit. on p. 22).

[HP76] L. L. Havlicek, N. L. Peterson. "Robustness of the Pearson Correlation against Violations of Assumptions". In: *Perceptual and Motor Skills* 43 (1976), pp. 1319–1334 (cit. on p. 43).

[HTZ16] Z. Hanusz, J. Tarasinska, W. Zieliński. "Shapiro–Wilk test with known mean". In: 14 (Feb. 2016), pp. 89–100 (cit. on p. 41).

[Isr92] G. D. Israel. *Determining Sample Size - GJIMT*. 1992. URL: https://www.gjimt.ac.in/wp-content/uploads/2017/10/2_Glenn-D.-Israel_Determining-Sample-Size.pdf (cit. on p. 32).

[Jan15] R. H. Jansen. *Learning TypeScript: exploit the features of TypeScript to develop and maintain captivating web applications with ease*. Packt Publishing, 2015, pp. 2 f. (Cit. on p. 15).

[JBA12] C. Jones, O. Bonsignour, T. Arroyo. *The economics of software quality*. Addison-Wesley, 2012 (cit. on p. 16).

[KGB+15] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. German, D. Damian. "The Promises and Perils of Mining GitHub (Extended Version)". In: *Empirical Software Engineering* (Jan. 2015) (cit. on p. 17).

[Kow72] C. J. Kowalski. "On the Effects of Non-Normality on the Distribution of the Sample Product-Moment Correlation Coefficient". In: 21.1 (1972), p. 1. DOI: 10.2307/2346598. URL: https://doi.org/10.2307/2346598 (cit. on p. 43).

[KRS+12] S. Kleinschmager, R. Robbes, A. Stefik, S. Hanenberg, E. Tanter. "Do static type systems improve the maintainability of software systems? An empirical study". In: *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. 2012, pp. 153–162. DOI: 10.1109/ICPC.2012.6240483 (cit. on p. 25).

[MD04] E. Meijer, P. Drayton. "Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages". In: Jan. 2004 (cit. on p. 16).

[MWW20] M. Muñoz Barón, M. Wyrich, S. Wagner. "An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability". In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM '20. Bari, Italy: Association for Computing Machinery, 2020. ISBN: 9781450375801. DOI: 10.1145/3382494.3410636. URL: https://doi.org/10.1145/3382494.3410636 (cit. on p. 36).

[Pie02] B. C. Pierce. *Types and programming languages*. The MIT Press, 2002 (cit. on p. 16).

[PT98] L. Prechelt, W. Tichy. "A controlled experiment to assess the benefits of procedure argument type checking". In: *IEEE Transactions on Software Engineering* 24.4 (1998), pp. 302–312. DOI: 10.1109/32.677186 (cit. on p. 16).

[Ris19] V. Riscutia. *Programming with Types -*. Birmingham: Manning, 2019. ISBN: 978-1-617-29641-3 (cit. on p. 20).

[Rob10] G. Robles. "Replicating MSR: A study of the potential replicability of papers published in the Mining Software Repositories proceedings". In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 2010, pp. 171–180. DOI: 10.1109/MSR.2010.5463348 (cit. on p. 16).

[RPFD14] B. Ray, D. Posnett, V. Filkov, P. Devanbu. "A Large Scale Study of Programming Languages and Code Quality in Github". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, pp. 155–165. ISBN: 9781450330565. DOI: 10.1145/2635868.2635922. URL: https://doi.org/10.1145/2635868.2635922 (cit. on pp. 16, 26, 27).

[RVWJ19] T. Roehm, D. Veihelmann, S. Wagner, E. Juergens. “Evaluating Maintainability Prejudices with a Large-Scale Study of Open-Source Projects”. In: *Software Quality: The Complexity and Challenges of Software Engineering and Software Quality in the Cloud*. Ed. by D. Winkler, S. Biffl, J. Bergsmann. Cham: Springer International Publishing, 2019, pp. 151–171. ISBN: 978-3-030-05767-1 (cit. on p. 27).

[Sak19] E. Saks. “JavaScript Frameworks: Angular vs React vs Vue.” In: (2019) (cit. on p. 23).

[SMKA17] A. Saboury, P. Musavi, F. Khomh, G. Antoniol. “An empirical study of code smells in JavaScript projects”. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2017, pp. 294–305. DOI: 10.1109/SANER.2017.7884630 (cit. on p. 19).

[Tie18] E. Tiemeyer. *Handbuch IT-Projektmanagement - Vorgehensmodelle, Managementinstrumente, Good Practices*. M: Carl Hanser Verlag GmbH Co KG, 2018. ISBN: 978-3-446-45385-2 (cit. on p. 22).

[WRH+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-29044-2. URL: https://doi.org/10.1007/978-3-642-29044-2 (cit. on p. 17).

[Yam13] A. Yamashita. “How Good Are Code Smells for Evaluating Software Maintainability? Results from a Comparative Case Study”. In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 566–571. DOI: 10.1109/ICSM.2013.97 (cit. on p. 35).

[ZLH+20] J. Zhang, F. Li, D. Hao, M. Wang, H. Tang, L. Zhang, M. Harman. “A Study of Bug Resolution Characteristics in Popular Programming Languages”. In: *IEEE Transactions on Software Engineering* (2020), pp. 1–1. DOI: 10.1109/tse.2019.2961897. URL: https://doi.org/10.1109/tse.2019.2961897 (cit. on pp. 16, 27, 36, 61, 66).

[ZMZ15] Q. Zheng, A. Mockus, M. Zhou. “A Method to Identify and Correct Problematic Software Activity Data: Exploiting Capacity Constraints and Data Redundancies”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 637–648. ISBN: 9781450336758. DOI: 10.1145/2786805.2786866. URL: https://doi.org/10.1145/2786805.2786866 (cit. on pp. 37, 66).

All links were last followed on November 10, 2021.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, 10.11.2021, [illegible]

place, date, signature