Institute of Software Engineering
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor Thesis

Does Functional Programming Improve Software Quality? An Empirical Analysis of Open Source Projects on GitHub

Daniel Abajirov

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Dr. Justus Bogner
Commenced:	November 7, 2020
Completed:	May 7, 2021

Abstract

Nowadays, there are not many studies that have empirically analyzed the effect of functional programming on software quality. Through the era of microservices and cloud-based systems, functional programming is experiencing a growing usage. This is due to the features that this paradigm offers and the benefits that follow. With this study we want to find out if there is a correlation between functional programming language and software quality.

To determine the impact of functional programming on software quality, we conducted an empirical study. This study was inspired by the lack of empirical evidence of this impact. To address this lack, we have collected a large dataset from GitHub (eight programming languages, four functional and four imperative, 400 projects, 200 using a functional languages and 200 using an imperative one) that we will analyze to gather information on various aspects of software quality such as maintainability, reliability and further to test our hypotheses.

Several tools and techniques were used for the analysis. For maintainability, we used a static analysis tool, SonarQube. For reliability, we analyzed the commit history of each project looking for bugs and categorized them. To determine the domains of each project, we implemented a classification algorithm in Python. As input for this algorithm, we used the information in each project's "README.md" to obtain a list of possible topics. With this list we were able to classify each project to its domain performing a manual classification. We also categorize bugs from the commit history into eight categories.

Above all, it seems that functional programming languages provide good reliability compared to imperative languages.
In general, it cannot be said that functional languages have a positive influence on software quality, since for maintainability there is not enough evidence to show that functional programming languages have less code smells that imperative programming languages. The results also indicate that for domains such as applications, databases, and libraries, the use of a functional language could decrease the frequency of programming bugs.

Contents

List of Figures

List of Tables

1 Introduction

In this chapter, we provide an introduction that helps to better understand the motivation behind the choice to analyze the impact of functional programming on software quality. Then, we explain the research questions we formulate to see if functional programming improves software quality. Finally, we give an overview of the structure of the paper.

1.1 Motivation

With the increasing use and development of new technologies and the expansion of cloud infrastructure, software quality is becoming more and more important [Goe19].
An important part of a project that determines its quality is modularity and as size of projects increases, modularity plays a big role in determining whether software is good or not. A study on the effect of modularity on software quality was published by Rosene et al. in "Software Maintainability - What It Means and How to Achieve It" [RCB81]. This research showed that there is a positive relation between the modularity and maintainability of a program. Modularity is also one of the key of functional languages and because of their implementation it is easily to write modular code then with a non-functional language like Java.
John Hughes claimed in his paper [Hug90] that as software becomes more complex, it is important to structure it well. In his paper he attempted to show how functional programming can help to achieve this goal. In his conclusion, he cited two important factors why functional programming can help improve the modularity of software. The first one is the use of higher order functions. The goal is to create smaller functions that take care of only part of the logic. Then combine them into more complex functions. The second is the lazy evaluation of functional languages. This is a crucial part of improving the modularity of a software and especially the software quality. Using lazy evaluation can have many benefits, such as postponing expensive calculations that may not be needed or working with large data sets that would not fit in memory. Lazy evaluation and other aspects of functional programming are discussed later in Chapter 2.

Today, there is not enough evidence to lead that functional programming has a positive impact on software quality. This is evident in the publications of studies that investigated software quality in specific domains using imperative languages or compared different programming languages to select the best language that can lead to good software quality. In their paper, "An Overview of Practical Impacts of Functional Programming", Khanfor and Yang [KY17] aimed to better understand the impact of functional programming language through a literature review. It was concluded that more studies need to be conducted by the software engineering research community that can show the impact of functional programming language on software quality. Functional programming is generally considered more difficult to learn and master than other

programming paradigm such as imperative programming. If we have enough evidence to show that learning this new paradigm is worth the effort, we can encourage more people to learn it and use it.

So the questions is whether functional languages can improve software quality. We have seen through Hughe's paper [Hug90] that there is a positive impact on modularity, but that is not enough to conclude that functional programming also has a positive impact on software quality.

From a purely academic use, functional language is slowly being adopted by many companies, such as Amazon, Twitter and others that we can see in the table 1.1. This information can be found on the website of official languages such as Elixir[1], Clojure[2], and Erlang [3]. They rely on the properties of functional languages such as the already mentioned lazy evaluation or the simplicity of writing code in functional style that make it is easier to find bugs and reduce the risk of exploits or to implement complex tasks that involve parallel actions.
At the early QConPlus[4] congress, functional programming has been addressed.

Clojure	Elixir	Erlang	F#
Amazon	Adobe	WhatsApp	Jet.com
Apple	BBC	IBM	Walmart
Cisco	Discord	Cisco	Olo
CircleCi	Frame	Rocket Journey	Large Financial Services Firm
Spotify	Payout3	Helium	Microsoft

Table 1.1: Companies using functional programming languages.

One of the presentation was "The resurgence of Functional Programming"[5] in which was discussed how functional programming could experience a resurgence due to the era of microservices and cloud-native systems. Properties of functional programming like composability, immutability or the absence of side effect can lead to a joyful development experience. Through this properties we can write more understandable code and through modularity we can program complex interactions, having a great confidence in the code.

On their paper "A Large Scale Study of Programming Languages and Code Quality in Github" [al17], Ray et al. attempted to investigate the influence of programming languages on software quality. They had as results under others that functional languages are a bit better than procedural languages and thus the choice of using a functional programming language could have a slight impact on software quality. Unfortunately, with these results we cannot assume that functional programming improves software quality over non-functional programming. This is because the number of functional projects selected is not representative. Of the 19 languages present, only

[1] ElixirCompanies, https://elixir-companies.com/en
[2] ClojureCompanies, https://clojure.org/community/companies
[3] ErlangCompanies, https://erlang-companies.org
[4] QConPlus2020, https://plus.qconferences.com/recap/plus2020
[5] ResurgenceFunctionalProgramming, https://plus.qconferences.com/plus2020/track/resurgence-functional-programming

3 were functional languages (Clojure[6], Erlang[7] and Haskell[8]). They considered Scala[9] to be functional, but it is not so easy to determine whether a Scala project is purely functional or not, due to the possibility of writing functional non-functional code in Scala [BHM+19]. This leads to a lack of empirical evidence that can support that functional programming languages could improve software quality. There is not enough data to compare functional with non-functional languages in their study to provide a reliable result. A better overview of the studies on the topic of functional programming or software quality is provided in chapter three.
Thus, as stated at the outset, there is a lack of empirical evidence to support the impact of functional programming on software quality. The few studies that have attempted to shed some light on this field are mostly poorly structured and fail to produce valid results.

1.2 Research Questions

The goal of this study is therefore to empirically analyze projects using functional programming languages and to compare them to projects with imperative languages. The comparison should provide insights into a potential influence of the functional programming paradigm on software qualities. The concrete quality aspects to be analyzed will be reliability and maintainability. In this way we can better understand, if functional programming improves software quality.

With our study, we want to find out if functional programming language have a positive impact on software quality. Then we want to see what characteristics influence the frequency of the bug categories of the analyzed projects. For this purpose, we will answer the following RQs:

1. Does using a functional programming language improve Software Quality[10]?
    1.1. Does using a functional programming language improve Reliability[11]?
    1.2. Does using a functional programming language improve Maintainability[12]?
2. What characteristics influence the frequency of the bug categories of the analyzed projects?
    2.1. Choice of the programming paradigm.
    2.2. Choice of the programming language.
    2.3. Choice of the project domain.

To better answer the two RQs, we split them into sub-questions so that we can easily examine them and gather enough information to answer the main questions.

[6]Clojure, https://clojure.org/index
[7]Erlang, https://www.erlang.org
[8]Haskell, https://www.haskell.org
[9]Scala, https://www.scala-lang.org
[10]SoftwareQuality, https://iso25000.com/index.php/en/iso-25000-standards/iso-25010
[11]Reliability, https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/62-reliability
[12]Maintainability, https://iso25000.com/index.php/en/iso-25000-standards/iso-25010/57-maintainability

For RQ1.1, we will look for bugs in each commit that a project has had in its history. For this question, we need to analyze each commit of a project and look for bugs or errors. With further analysis, we will mark a commit as a bug if it contains a bug discovery or bug fix.

To answer RQ1.2, we will use static code analysis to find code smells that each project currently has. This information plus the main line code of a project will be used to compare the two programming paradigms in order to answer this RQ. As main lines code we mean only the lines of code that are written in the main language assigned to each project.

To answer RQ2.1, we will see which programming paradigms are more prone to certain types of bugs. In order to categorise a bug, a classification algorithm will be applied to the information obtained from RQ1.1.

To answer RQ2.2, we will see which programming languages are more prone to certain types of bugs. In order to categorise a bug, a classification algorithm will be applied to the information obtained from RQ1.1.

To answer RQ2.3, we will assign to each project its domain and see which domains are more prone to certain types of errors. We will divide the domains into categories that we have chosen for this purpose and see what the main difference between them is. The assignment of domains will be done using a topic analysis algorithm on the description files of each project. In order to categorise a bug, a classification algorithm will be applied to the information obtained from RQ1.1.

1.3 Structure of the Thesis

This thesis is divided into seven chapters. Chapter 1 introduces the motivation behind the thesis and the research questions. Chapter 2 gives the essential background knowledge to better understand the paradigms and the definition we used in this thesis. In chapter 3 we present the related work. Chapter 4 describes the design and planned procedure of the study. The actual evaluation of the results and comparison are discussed in Chapter 5. Chapter 6 contains a discussion of the results as a threat validation and Chapter 7 the conclusion of this thesis and the limitations that this thesis has encountered. Future work is also discussed in this chapter.

2 Background

In order to fully understand the content of this paper and follow its logical thread, we must first explain three basic concepts, namely imperative programming, functional programming, and software quality.

2.1 Imperative Programming

The first computers were not electronic but electromechanical, requiring glass tubes to perform some operations. An example is the first computer (ENIAC, 1946) [SBCG98]. These first computers had something in common, they all needed instructions to perform operations. This summarizes the early phase of imperative programming. Instructions were simple. They manipulated data only step by step, the so-called "do this, then do that", as Salus says in his book [SBCG98]. The first major imperative programming language for electronic computers was FORTRAN. With this new language it was possible to execute complex expressions and create more complex programs [SBCG98].
In general, imperative programming is a well-defined sequence of instructions given to a computer to change a state. An example of this type of programming is the loop, where, starting from an initial state, instructions are executed through a series of iterations that change the state. The source code of imperative languages assembles the commands that determine what the computer must do and when to achieve the desired result [SIT18].
Imperative programming languages are very concrete in that they work close to the system. So on the one hand the code is easy to understand, on the other hand it takes many lines of source code to describe what can be accomplished in declarative or functional programming languages with a fraction of those commands. This has both advantages and disadvantages [SIT18].
As advantages we have that imperative programs are more easy to run on hardware [BMP13]. The first disadvantages are in solving more complex problems, which increases the amount of code produced. It remains very readable, but its size makes it unwieldy. When is produced more code, there is also the risk of making more mistakes and introducing more bugs, which makes updating an application quite complex.

2.2 Functional Programming

Functional programming languages can be traced back to the lambda calculus proposed by Alonzo Church in the 1930s [Mic11]. The lambda calculus is based on mathematical logic and forms the basis for modern functional programming languages [KI16]. In 1958, 30 years later after the introduction of lambda calculus, John McCarthy invented LISP, the first functional programming

language. Functional programming describes programs as expressions and transformations, modeling mathematical formulas, and tries to avoid mutable state. Functional programming languages categorize problems differently than imperative languages [CMHB18]. The logical categories (filter, transform, and convert) are represented as functions that implement the low-level transformation but rely on the developer to customize the low-level machinery with a higher-order function, supplied as one of the parameters [CMHB18].
Programs are built out of pure functions. A pure function has no side effects, it doesn't depend on anything but its arguments, and its only influence on the outside world is through its return value. Functional programs make heavy use of recursion and laziness. A recursion occurs when a function calls itself, either directly or indirectly. With laziness, an expression's evaluation is postponed until it's actually needed. Lazy techniques imply pure functions.
Laziness depends on the ability to replace a function call with its result at any time. Functions that have this ability are called referentially transparent. This can benefit from Memoization (automatic caching of results) and automatic parallelization, moving function evaluation to another process or machine [KI16].
We can see that some imperative languages such as Java [1] or C# [Buo17] have recently implemented some functional capabilities into their paradigm. In summary, functional programming has many advantages such as run-time optimization, lazy evaluation, and abstraction in categorizing problems [BMP13]. To get a better overview of the two implementations, we pick two implementations of the factorial function, this can be seen in listing 2.1. First we have a Java implementation of this function. The use of the for loop to iterate through immediately jumps out. The code is easy to understand and to follow. In listing 2.2 we have the same implementation using the functional language Clojure. Here it is hard to tell what is happening without further knowledge. Reduce is a core function in Clojure, which can also be found in Lisp. This function takes the first two items in the list, applies the operator to them, then takes that result along with the next item in the list and applies the operator to them, and so on. Range and inc as functions should be self-explanatory. Another difficulty is to follow the logic of this function.
In conclusion, the imperative implementation is easier to read and understand, while the functional one takes some time to be analyzed and understand.

```java
public long factorialJava(int n) {
    long factorial = 1;
    for (int i = 2; i <= n; i++) {
        factorial = factorial * i;
    }
    return factorial;
}
```

Listing 2.1: Implementation of factorial function in Java.

```clojure
(defn factorial[n]
  (reduce * (range 1 (inc n))))
```

Listing 2.2: Implementation of factorial function in Clojure.

[1] Java lambda expressions. https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

2.3 Software Quality

Software quality is not so easy to define. There are many definitions but they all point in the same direction. For example some defined software quality as measures how well an application is designed or perform, others defined it whether the software satisfies its requirements. Wagner (2013) in “Software Product Quality Control” [Wag13] notes that quality is not an easy concept to be defined and "quality is a concept that has kept philosophers occupied for more that 2000 years". Always in [Wag13], we can see how different the software quality from different organization like ISO, IEC and IEEE is defined:

- The degree to which a system, component or process meets specified requirements
- The ability of a product, service, system, component or process to meet customer or user needs, expectations or requirements
- The totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs
- Conformity to user expectations, conformity to user requirements, customer satisfaction, reliability and level of defects present
- The degree to which a set of inherent characteristics fulfils requirements
- The degree to which a system, component or process meets customer or user needs or expectations

For this thesis, we will use the definition of the ISO 25010 [2] that says "The quality of a system is the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value". In particular we will analyze the aspect of Reliability and Maintainability. The definitions according to ISO 25010 are as follows:

- **Reliability:**"Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time".
- **Maintainability:**"This characteristic represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements".

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

3 Related Work

Unfortunately, not so many empirical studies have been done on functional programming that analyzed the impact on software quality.

One of the few studies that attempted to shed some empirical light on the question of which language could improve software quality was the study performed by Harrison et al. in "Comparing programming paradigms: an evaluation of functional and object-oriented programs"[HSDL96]. In this study, a quantitative evaluation of functional and object-oriented paradigm is carried out. The aim was to find out whether the choice of one paradigm over another has an influence on the quality of the code. For the evaluation, twelve sets of algorithms were developed in SML (Standard ML) and C++. Strict constraints were placed on the development of this algorithm to improve the reliability of the results. It was found that there is a significant difference in the number of known errors per thousand lines of code. Further results showed that the SML code showed one and a half times as much reuse as the C++ code. This suggests that using a functional programming language may be better for reusability than object-oriented programming languages. But in general, no significant differences were found to the numbers of known errors, modification requests, the times to attend to these, a subjective measure of complexity, and the total development time. The finding suggested that the subjective preference may be a deciding factor.

Another interesting study that indirectly compared functional and non-functional languages in an empirical study was that of Ray et al., “A Large Scale Study of Programming Languages and Code Quality in Github” [al17]. They collected a total of 850 projects in 17 different languages. These projects were selected from GitHub, retrieving only projects primarily written in that language. Using a mixed-methods approach, they were able to combine multiple regression modelling and text analysis to investigate the influence of language features on software quality.
They found out that strong typing is better than weak typing, that static typing is better than dynamic, and that managed memory usage is better than unmanaged. Further they found that functional languages are slightly better than procedural languages, albeit with some limitations.
There are limitations to these results. The first is that they were not able to quantify the specific effects of language type on usage. Secondly, the categorisations of domains and bugs could be influenced by the initial choice of keywords. Finally, they associated defect fixing commits with language properties, even though they might reflect the reporting style or other properties of the developers.

Due to the success of this study, Berger et al. decided to conduct a replication study to validate the results thus obtained by Ray and his collaborators. In their reproduction study [BHM+19], they first carried out an experimental replication, which was only partially successful. Nevertheless, they managed to validate one part of the study, namely the association of programming languages with defects.

Due to an incomplete data set and missing codes, they had to do a complete re-analysis of the data and statistical modelling steps of the original study. Through this reanalysis, they discovered that only four out of eleven languages from the original study had a statistically significant association with defects, and even among these, the effect size was quite small.
Upon further investigation, they found that the original statistical modelling had technical omissions, such as improper handling of multiple hypothesis testing. After correction, they compared the data thus obtained with that of the original study and found a certain discrepancy. For example, the original six languages with a positive association with defects became only one.
This replication study is interesting for us because it gives us some good practice recommendations for similar efforts. Recommendations that can lead to the avoidance of mistakes that can be made when conducting such empirical studies. Such as the attention that should be paid to the modelling of the analysis pipeline. To avoid bias, errors and unwarranted interpretations, it must be implemented carefully.

But we can also find other papers that are in some way related to the empirical study and maintainability, an aspect that we have investigated in this study on software quality. One example is the study by Roehm et al., "Evaluating Maintainability Prejudices with a Large-Scale Study of Open-Source Projects", [RVWJ19].
In this study, a large set of open source projects (6,987 GitHub repositories, 402 million lines, 5 programming languages) were used to test 10 hypotheses about maintainability. The data was collected via GitHub API and analyzed using an open source tool ConQAT.
The results were that programming language has only a modest impact on maintainability and that there is no significant relationship between maintainability and development activity, repository popularity, code base size and team size.

Another example of an empirical study based on software quality is that of Kochhar et al., "A Large Scale Study of Multiple Programming Languages and Code Quality" [KWL16]. In this study, a large empirical investigation was conducted to find out whether the use of multiple programming languages to implement some functionalities has an impact on software quality.
For this purpose, a large dataset was collected. This dataset consisted of popular projects from GitHub (628 projects, 85 million SLOC, 134 thousand authors, 3 million commits, in 17 languages). Multiple regression models were built to investigate the effect of using different languages on the number of bug-fixing commits, considering factors such as project age, project size, team size, and the number of commits.
The results showed that, in general, using multiple languages to implement a project has a significant impact on the quality of the project. This is due to an increase in error-proneness. Furthermore, it was found that certain languages are more error-prone when used with other languages, such as C++, Objective-C, Java, TypeScript, Clojure and Scala.

The studies cited above are used by us to get a more detailed overview of how to measure the code quality of open source code and to see how we can analyze the data collected.
Of particular importance to us was the way they selected and analyzed the open source projects and how the data cleaning process was done to get a valid sample of the projects to be analyzed. None of the aforementioned studies, however, have directly analyzed the impact that a functional programming language could have on software quality.

4 Methodology

In this chapter, we describe the languages and GitHub projects we collected and the analysis methods we used to gather the data necessary to answer our research questions. To have a better overview of the whole data collection process, we can take a look at figure 4.1.
The first step is the data collection. Here is important to identify good sources from where we can extract our data. This can be seen in the next section, “Study Objects and Sampling”. Here we will explain what source we used, how we proceeded, and what rules we applied to get good data.
The next phase is data cleaning. In this phase, we remove all data that does not fit our rules. Through this process, we obtain valid data that we can later use for analysis. This can also be seen in the next section, “Study Objects and Sampling”.
After data cleaning comes data analysis. Here, after we have a valid collection of data, we can start analyzing it. We can see this in the section, “Data Collection”, “Identifying Project Domains”, and “Categorizing Bugs”.
The final stage of interpretation occurs later in chapter five, where we compare the data to answer the RQs formulated in chapter one.

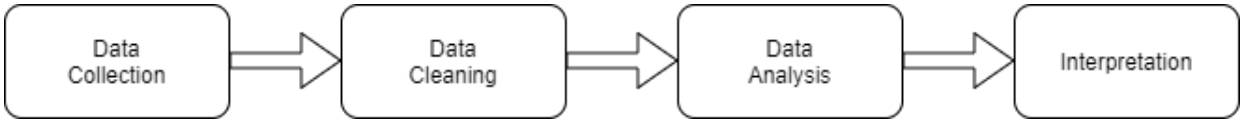

Figure 4.1: Data collection process.

4.1 Study Objects and Sampling

Study Object To understand whether functional programming languages have an impact on software quality, we analyzed four functional programming languages, which can be seen in the table 4.1, and for comparison, four imperative programming languages. This can be seen in table 4.2. For the functional languages, we chose Clojure, Haskell, Erlang, and F#, and for the imperative languages, we chose JavaScript, Python, Go, and Ruby.
For each of the selected languages, we used GitHub to find the projects we needed for our study. GitHub is a code-hosting platform where it is possible to find open-source projects used as data-mining sources by various empirical studies.
The choice of languages that we saw in the table 4.1 and 4.2 was mainly based on the supported languages that the static analysis tool, SonarQube [1], supported. This tool uses static code analysis to capture quality technical features of software and displays them in a web interface. In addition to the most famous language like Python, Java , and C , many other languages can also be analyzed

[1] SonarQube, https://www.sonarqube.org

	Project Details			Total Commits
Functional Language	**Projects**	**Authors**	**KLOC**	**Commits**
Clojure	50	3.580	21.925	63.101
Erlang	50	2.544	32.806	48.554
F#	50	3.335	46.611	82.260
Haskell	50	4.517	58.808	142.728
Summary	**200**	**13.976**	**160.152**	**336.643**

Table 4.1: Study object: Functional language.

	Project Details			Total Commits
Imperative Language	**Projects**	**Authors**	**KLOC**	**Commits**
Go	50	22.656	318.502	344.906
JavaScript	50	13.428	158.825	198.818
Python	50	32.584	216.191	306.672
Ruby	50	23.921	81.279	245.913
Summary	**200**	**92.589**	**774.799**	**1.096.309**

Table 4.2: Study object: Imperative language.

using internal or external plugins. For this work, we used four external plugins to analyze projects written in Clojure , F# , Haskell and Erlang .
At the beginning of our analysis, we selected the first 50 top projects written mainly in the analyzed languages, sorted by stars. In total, we analyze 400 projects from eight different languages, 106,565 Authors, over one million KLOC and over one million commits that can be seen in table 4.1 and 4.2.

Sampling To automatically retrieve the corresponding projects for the selected languages from GitHub, we used the GitHub REST API [2]. Using the request[3] library in Python, we were then able to use the GitHub API to automatically retrieve the data we needed. Due to the limit on the number of requests an unidentified user can make per hour, a user token was required. The data thus obtained were still in a raw state, and a further process was necessary to obtain valid data.

We conducted a process of data cleaning to obtain a valid sample of projects to analyze. This process consisted of three phases, during which we excluded the projects that did not meet our criteria. The criteria on which we excluded the projects was based on the study performed by Kalliamvakou et al. [al16]. This criteria were based on the numbers of stars that a project must have to be considered, here minimum 50, and the numbers of commits to be considered a project, more than 6. This process of data cleaning can be summarized in three phases:

[2]GitHub REST API, https://docs.github.com/en/rest
[3]request, https://requests.readthedocs.io/en/master/

- **First:** identify the first top 100 projects for each language based on the number of stars. This is done using the GitHub-API and the python library "Requests" [4]. Then we need to ensure that the main language, GitHub assigns to each project, is more than 75%.
- **Second:** manually review each project to determine if it is a collection or other type of documentation that cannot be classified as a project.
- **Third:** clean the data by removing the projects that were only for collection or documentation and did not meet the minimum number of commits. If the data thus obtained is less than 50 projects, take the next sample for 10 projects and repeat the steps.

After this process of data cleaning, we can extract the first details about these projects. These details consisted of the number of authors each project had, the number of KLOC and the number of total commits. This information can be seen in table 4.1 for functional projects and in table 4.2 for imperative one.

4.2 Data Collection

Mainly we collected two types of data coming from the same sources, the projects. The first came from using SonarQube to analyze the projects, the second from analyzing the commit history of the projects in each repository.

In order to use SonarQube, some preparations were necessary. First, we had to manually set up the correct dependencies to Clojure [5] projects and Haskell[6]. Without these preparations it was not possible to use these plugins for these languages. For Clojure, it was a case of adding the right dependencies from the plugin or making sure that each Clojure project used the right dependencies. Then, we had to manually test whether the dependencies added in this way worked for each project without causing errors. For Haskell, it was necessary to use hlint [7] to generate a report. This report was then used by SonarQube to analyze the project. For the rest of the languages, this adding and checking was not necessary. Due to the limitation of the SonarQube API, it was then not possible to automate the whole process. This meant that manual creation of the projects to be analyzed was required on the SonarQube side, including execution of the command line needed to analyze the projects.

After each project was analyzed by SonarQube, we proceeded to manually annotate the results we needed for our study. This included the number of code smells, their proportions, and the number of LOC. SonarQube did not provide an easy way to retrieve only the code for a particular language in a project. To retrieve only the code regarded to the language of our interest, we used an external tool to count the lines of code, CLOC [8]. In this way, we could associate the code smells of a particular language with its lines of code.
This manual annotation was done because we were using the basic version of SonarQube, which

[4]Requests python library https://docs.python-requests.org/en/master/
[5]SonarQube Clojureplugin, https://github.com/fsantiag/sonar-clojure
[6]SonarQube Haskellplugin, https://github.com/uartois/sonar-haskell
[7]hlint Haskell,https://github.com/ndmitchell/hlint
[8]CLOC, tool to count lines of code. https://github.com/AlDanial/cloc

did not allow us to easily export the data thus obtained. In the table 5.3 we can observe the Code Smells, the Maintainability Rating value, and the Maintainability Rating Scale of each functional languages and in the 5.4 for imperative languages. The Maintainability Rating Value is given to the project related to the value of the Technical Debt Ratio. The default Maintainability Rating grid is: A=0-0,05, B=0,06-0,1, C=0,11-0,20, D=0,21-0,5, E=0,51-1. To create these tables, for each scale we took the mean of the range, for example, if a project was classified as A, we gave it the value 0.25 ((0 + 0,05) / 2).

Due to the limitation of using external plugins to analyze functional projects, when we encountered a problem with a project, such as dependencies not working or a "NullPointExpection", we selected the next available project on our list to have the same number of valid projects for all languages.

The second type of data consisted of bug fix commits. We know that when developers fix bugs, they tend to leave important information about the fix or the bug, such as why the bug occurred or how the bug was fixed. To distinguish bug/fix commits from normal commits, we relied on keywords that developers tend to use when fixing or commenting on certain bugs. One of these keywords was the combination of the word "fix" with the word "issue", but other keywords than this one were used to ensure that mainly only bugs/fixes were collected.

After we completed the collection of bug/fix commits, to ensure the validity of the data thus obtained, a random sample of 800, 100 from each language, bug fix commits was taken and manually verified. From all the data so analyzed only five, 0.006%, commits were not bug/fix commits, two of them belonged to Python, one to Ruby, one to Clojure, and the last one to Go.

4.3 Identifying Project Domains

Normally, it is possible to assign to each project its corresponding domain. For example a project that consist on an application that users can use, fall under the domain "Application". A project that contains data and programming code that is used to develop software programs or application fall under the domain "Library". A full overview of the domains that we used in this thesis can be seen in the table 4.3. We grouped together domains that appeared individually in either functional or imperative programming into the category "Other".

The subdivision into domains allowed us to analyse projects in more detail and obtain more information. This allowed us to observe, for example, what influence a certain language has on a specific domain rather than on the whole sample. In order to realize this subdivision, we needed to classify the studied projects into different domains. This was done based on their characteristics and functionality using a mix of automated and manual techniques.

Normally, every project on GitHub have always a "Readme" file that describe its features. This means that we can use this information to classify each project into its own domain manually or through an automated technique.
We first applied Latent Dirichlet Allocation(LDA), a well-known topic analysis algorithm, to the text describing the features of the project. Topic modeling refers to the task of identifying the topics

Domain	Domain Characteristics	Total Projects
Application (APP)	A program that can be used from an user.	93
Database (DB)	An implementation of a database using sql or nosql.	17
Code Analyzer (CA)	A program for testing, analyzing and reporting information about the source code.	33
Library (LIB)	A project that contains data and programming code that is used to develop software programs or application.	132
Framework (FW)	A platform to develop software applications.	57
Programming Language (PL)	A project that implement a programming lanugage.	13
Others (OTH)	-	55

Table 4.3: Domains of imperative language projects.

that best describe a set of documents.These topics emerge only during the topic modeling process (hence called latent). Each topic represents a set of words. And the goal of LDA is to map all documents to topics in such a way that the words in each document are mostly captured by those imaginary topics [BNJ03].
Given a set of documents, LDA identifies a set of topics where each topic is represented as a probability of generating different words. For each document, LDA also estimates the probability of assigning that document to each topic. Based on this information, we were able to assign a domain to each project with a semi-manual classification. Domains like framework or programming language needed more attention and a manual investigation on the project repository. Frameworks are sometimes easily confused with Library or Application due to the poor documentation of these projects. Just like programming languages that sometimes presented words like application or library in their description that can confuse the assignment of a domain. The implementation of the algorithm was made possible through the use of the python library "Gensim" [9].

The results and classification of this mix of automated and manual techniques to classify projects into domains can be seen in the table 5.10 and 5.11.

[9]Gensim library https://radimrehurek.com/gensim/models/ldamulticore.html

4.4 Categorizing Bugs

Project commit logs contain not only information on modifications and production incidents that occurred during their development, but also other information such as the type of incidents that occurred. These types of incidents gave us a better understanding of what kind of problems a certain project has had, such as problems caused by the incorrect implementation of an algorithm or a problem due to insecurity. With this kind of categorisation we can better understand which programming languages are more prone to certain errors.
In our study, we analyzed the commit history of each project to find out what kind of bugs or errors each bug-commit had. In order to make this categorisation, we based ourselves on the studies of Tan et al. [LTW+06] and Ray et al. [al17]. From their studies, we borrowed the categories that we used to categorize our bug-commits. Of the 20 categories presented in Tan's study, we decided to use only eight that were also presented in Ray's study. To make sure that the categories we discarded were not present or not easily recognisable, we searched the commits with keywords tied to these types of categories. For example, we searched for corruption or data corruption for the category "data corruption", but nothing was found for this category.
We noticed that some bugs contained keywords linked to the discarded categories but fell more into the programming category, such as "Fix 443 port" or "Fix crash due to wrong input in display-method". So we finally decided to classify the bug commits according to these eight categories, which we can find in the table 4.4. In this table we can see the description of each category and the keywords linked to the bug/fix commits.
In order to assign a separate category to each of the almost 100 thousand bug/fix commits found, a manual classification was not possible. For this reason, we developed a natural language processing algorithm (NLP). Natural language processing is a field of artificial intelligence in which computers analyze, understand, and derive meaning from human language in a smart and useful way [LHH19].

For the implementation of the supervised learning algorithm we decided to use a linear model and a python library [10]. Linear models make predictions using linear functions of input features and are also used for classification. A main advantage of these models is that they are very fast to train and can make predictions quickly [Gui16].
To optimize this model we then used the SGD (stochastic gradient descent) method [11]. SGD Classifier fit well for large data sets

We first randomly selected 400 bug/fix commits and then manually classified each bug into one of the categories presented in the studies cited above to use as training data for supervised learning. Then randomly selected another 400 bug/fix commits to use these as supervised data. By testing the prediction, we got an accuracy of 89%. This result was quite satisfactory, so we decided to merge the training data with the supervised data into a single training data to be used against the remaining bug/fix commits that needed to be analysed. The categorization of these commits can be seen in the table 5.6 and 5.7

[10] Lienar model https://scikit-learn.org/stable/modules/classes.html#module-sklearn.linear_model
[11] SGDClassifier https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html

Bug Type	Bug Description	Search keywords
Algorithm (Alg)	Bugs caused by algorithmic or logical errors.	algorithm, implementation, implementing, logical errors, logical.
Programming (Prog)	Bugs caused by generic programming errors.	missing, missing switch case, missing case, faulty initialization, bad initialization, default value, standard value, exception handling, exception, copy-paste, copy-paste error, refactoring, type error, error handling.
Concurrency (Conc)	Bugs caused by multi-threading or multi-processing (data race, deadlock, and synchronization).	deadlock, race condition, data race, race, synchronization error,
Memory (Mem)	Bugs caused by improper memory handling.	null pointer, memory leak, buffer, buffer error, buffer overflow, heap overflow, dangling pointer, double free, segmentation fault, OutOfMemoryException, StackOverflowException, MemoryFailPoint, AccessViolationException, MemoryError.
Security (Sec)	Errors that may affect the authentication of users, the authorisation of access rights and privileges, the confidentiality of data or the integrity of data.	security buffer overflow, buffer overflow, security, security error, security bug, password, privilege, authorisation, compromise integrity of data, compromise confidentiality of data, oauth, auth, ssl, ssl error, ssl fix.
Performance (Perf)	Bugs that can lead to significant performance problems, delayed responses or to a poor user experience because the system is slow.	optimization problem, delay response, performance, performance problem, performance fix, performance bug
Failure (Fail)	Bugs that can cause the application to crash or hang.	reboot, restart, crash, crash problem, hang, hang problem, booting, booting problem, system crash, application crash.
Unknown (Unkn)	The impact cannot be identified from the bug report.	

Table 4.4: Categories of bugs and the keywords that characterise them.

4.5 Metrics

We selected static metrics which we expect to have an effect on the impact of functional programming on software quality: number of bugs per commit and code smells per LOC. We chose these metrics because they are easy to understand and improve. They are language-independent and they have been found to be suitable for making solid statements about software reliability and maintainability

and they are used in practice.
Code smells can provide hints to various maintainability factors that can be improved through refactoring [YM12]. Bug counts a metric have been used for a variety of purpose like for error-pattern discovery or the for the evaluation of products status [KMB04]. Another use was the estimation of reliability of a product [BL92].

4.6 Hypotheses

To make the impact of functional programming on software quality analyzable in an empirical study, we formulated two hypotheses about reliability and maintainability. These hypotheses can be seen in table 4.5. Our hypotheses can be divided in two categories: assumptions about the impact of the functional programming language on reliability (cf. RQ1.a) and assumptions about the impact of the functional programming language on maintainability (cf. RQ1.b). The hypotheses consider the languages Clojure, F#, Erlang, and Haskell as functional programming languages and the languages Python, JavaScript, Go, and Ruby as imperative programming languages.

	Alternative hypothesis	**Null hypothesis**
Reliability	H_{11}: Projects using a functional programming language have fewer bugs than projects using an imperative one. $mean(FunctionalBugs) < mean(ImperativeBugs)$	H_{01}: Projects using a functional programming language have more or a similar number of bugs than projects using an imperative one. $mean(FunctionalBugs) >= mean(ImperativeBugs)$
Maintainability	H_{12}: Projects using a functional programming language have fewer code smells than projects using an imperative one. $mean(FunctionalCodeSmells) < mean(ImperativeCodeSmells)$	H_{02}: Projects using a functional programming language have more or a similar number of code smells than projects using an imperative one. $mean(FunctionalCodeSmells) >= mean(ImperativeCodeSmells)$

Table 4.5: Overview of Hypotheses (each in the form of an alternative hypothesis H_{1i} and its null hypotheses H_{0i}, where i represents the RQ; i=1 for Reliability and i=2 for Maintainability). FunctionalBugs are all the bugs found in projects using a functional language. ImperativeBugs are all the bugs found in projects using an imperative language. FunctionalCodeSmells are all the code smells found in projects using a functional language. ImperativeCodeSmells are all the code smells found in projects using an imperative language.

The motivation behind H_{11} was that with the functional programming language many problems can be solved with less code and some bugs can be avoided. This is because pure functions and the immutability of data are an important aspect of functional programming. With these features, it is possible to write cleaner and, compared to some imperative implementations, shorter code. Together

with the avoidance of side effects and strict typing rules, this could lead to better reliability. For example, Haskell does not allow functions with side effects [12]. The only variables that the function can change are local to the function. This ability is a huge asset for a functional programming language.

The reason for H_{12} was the relationship that programmers who use functional languages have to the code. They tend to follow the core of functional thinking [13]. Functions need to make sense, but in an imperative implementation it is easier to hide bad design in objects and not realise what happened. The thought process of functional programming is slower, but this produces higher quality designs that the program needs to thrive and be more useful in the long run. This could led to less code smells and to a better maintainability of a project. This is also encouraged by the modularity that functional programming has to offer. Another reason behind H_{12} is modularity of functional languages. This property plays a big role for maintainability of a program. Maintanble code is easier to understand, to test, and to refactor.

4.7 Statistical Methods

To answer the first two sub-questions (RQ1.a and RQ1.b), we must first find a suitable test. To select one, we must first check whether the samples are normally distributed.
The normal distribution is a probability function that describes how the values of a variable are distributed. It is a symmetrical distribution. This means that most observations cluster around the central peak. Because it is a symmetric function, the probabilities for values farther from the mean taper equally in both directions [Tho19].
Normal distribution of sample determine which kind of tests are more appropriate. If our sample is normal distributed we can select a parametric test, like an independent t-test or an ANOVA, otherwise a non-parametric one, like the Wilcoxon or the Mann-Whitney U-test.

In order to check if the samples were normally distributed we used the Shapiro-Wilk test [Tho19]. Instead of implementing it manually we relied on a python library, "SciPy" [14]. In this way we are sure that the output is a valid result. The test needs as input an array of sample data and as output we get the test statistic and the p-value. If the p-value is less than 0.05 we can reject the null hypothesis and thus conclude that the sample is not normally distributed. For both samples, the p-value was quite smaller than 0.05, so we had to reject the null hypothesis that the sample data are normally distributed.

After performing the Shapiro-Wilk test on the sample we wanted to analyze, it turned out that all samples were not normally distributed. For this reason, we decided to use a non-parametric test, the Mann-Whitney U-test. This test is used for the comparison of two independent, non-normally distributed samples with a number of observations > 20. More details on this specific test are to be found in the book of Corder and Foreman, "Nonparametric Statistics for Non-Statisticians: A

[12] Functional Programming Using Haskell, https://www.mta.ca/~rrosebru/oldcourse/371199/haskell/paper.htm#relia
[13] Core Functional Programming Concepts, https://thecodeboss.dev/2016/12/core-functional-programming-concepts/
[14] Shapiro-Wilk test in python https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.shapiro.html

Step-by-Step Approach" [CF09]. As with the Shapiro-Wilk test, we again used the Python library "SciPy" to perform the Mann-Whitney U-test [15]. The test needs as input an array of samples and as output we get the test statistic and the p-value. If the p-value is less than or equal to 0.025 we can reject the null hypothesis and thus conclude that there is a significant difference between the two samples.
The critical value of 0.025 was calculated using the Bonferroni correction. This test is used to correct the critical value based on the number of hypotheses. "The Bonferroni correction controls the number of false positives arising in each family by using a probability threshold for each observation within the family." [Hay13].
For RQ2, we performed descriptive statistics to get an overview of the data and see if there is a relationship between the choice of programming paradigm, programming language, or project domain and the frequency of error categories of the analyzed projects.

[15] Mann-Whytney U-test python https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html

5 Results

In this chapter, we show the results we obtained from the analysis of the projects as described in chapter four. The structure follows the RQs we formulated in chapter one. The information is summarized in tables.

5.1 Software Quality Impact (RQ1)

Here, we present the results and the hypothesis that we used to answer the sub-question RQ1.1 and RQ1.2.

H_{11}: Projects using a functional programming language have fewer bugs than projects using an imperative one.

In table 5.1, we can get an overview of the bugfix commits found. For a clearer comparison, the percentages are displayed next to the numbers.
As we can see, there are not big differences within functional languages. The range of bug fixes is between 5% and 7%. In the functional languages as a whole, we can see that only 6% of the total commits are bug fixes.
On the other hand, we can see in table 5.2 that the range of bug fixes for imperative languages is between 4% and 9%. For the imperative languages, 7% of the total commits are bug fixes, 1% more than for functional languages.

Two samples were used to perform the Mann-Whitney U-test. The first contained the bug/fix per total commits of each functional project. The second contained the bug/fix per total commits of each imperative projects.
Using this two samples as input for the test, the results for the Mann-Whitney U-test indicated a significant difference between the two samples, we can see the results in table 5.5. Our effect size for the sample difference is 0.25. This value indicates a small-medium level of association between projects using a functional programming language having fewer bugs than projects using an imperative one. The median of the functional sample (median(Functional) = 0.044 bug/fix per total commits) is smaller than the median of the imperative one (median(Imperative) = 0.061 bug/fix per total commits). Moreover, the effect size for the sample difference was 0.25. This means that the null hypothesis is rejected.

Therefore, we can conclude that projects using a functional programming language have fewer bugs than projects using an imperative one.

Functional Languages	Total Commits	BugFix Commits
Clojure	63.101	3.316 (~ 5%)
Erlang	48.554	3.229 (~ 7%)
F#	82.260	4.891 (~ 6%)
Haskell	142.728	7.681 (~ 5%)
Summary	**336.643**	**19.117**(~ 6%)

Table 5.1: Overview of metric results for functional languages.

Imperative Languages	Total Commits	BugFix Commits
Go	344.906	30.140(~ 9%)
JavaScript	198.818	8.859 (~ 4%)
Python	306.672	27.368 (~ 9%)
Ruby	245.913	14.029 (~ 6%)
Summary	**1.096.309**	**80.396**(~ 7%)

Table 5.2: Overview of metric results for imperative languages.

H_{12}: Projects using a functional programming language have fewer code smells than an imperative one.

In table 5.3, we can get an overview of the code smells found. We also included the maintainability rating value/scale to provide a better overview (this information is not used to evaluate the hypothesis).
To better compare the data in the tables, we annotate the respective % near each number of code smells. As we can see, there are some big differences within functional languages. If between Clojure, F#, and Haskell there is a small difference, between Erlang and the other languages there is a huge difference, almost 0.34 code smells per LOC more.
In the functional languages as a whole, we can see that we have 0.08 code smells per LOC.
On the other hand, we can see in table 5.4, that the range of code smells for imperative languages is between 0.02 and 0.2 code smells per LOC, very low compared to the functional one. All together for the imperative languages, we have only 0.6 code smells per LOC, almost 6 code smells per loc less than for functional languages.

Two samples were used to perform the Mann-Whitney U-test. The first contained the code smells per total main lines of code of each functional project. The second contained the code smells per total main lines of code of the individual imperative projects.
Using these two samples as input for the test, the results for the Mann-Whitney U-test indicated no significant difference between the two samples, we can see the results in table 5.5. The median of the functional sample (median(Functional) = 0.0159 code smells per LOC) is bigger than the median of the imperative one (median(Imperative) = 0.0133 code smells per LOC). This means that the null hypothesis is not rejected.

Functional Languages	Lines of Code	Code Smells	Maintainability Rating Value
Clojure	321,303	2,363(~ 0.7%)	0.029(~ A)
Erlang	562,329	199,025 (~ 35%)	0.029(~ A)
F#	1,063,827	23,111 (~ 2%)	0.029(~ A)
Haskell	1,099,179	7,301 (~ 0.6%)	0.053 (~ B)
Summary	**3,046,638**	**231,800**(~ 7%)	**0.14**(~ C)

Table 5.3: Code smells of functional languages using SonarQube.

Imperative Languages	Lines of Code	Code Smells	Maintainability Rating Value
Go	19,205,881	119,777 (~ 0.6%)	0.029 (~ A)
JavaScript	11,989,449	25,257 (~ 0.2%)	0.025(~ A)
Python	3,991,981	57,981 (~ 1%)	0.029 (~ A)
Ruby	1,664,597	18,402 (~ 1%)	0.029(~ A)
Summary	**36,851,908**	**221,417**(~ 0.6%)	**0.112**(~ C)

Table 5.4: Code smells of imperative languages using SonarQube.

Therefore, we can not conclude that projects using a functional programming language have fewer code smells than an imperative one.

	median(Functional)	median(Imperative)	U-value	p-value	Effect Size	α -value
Reliability	0.044	0.061	14,114	1.7e-7	0.25	0.025
Maintainability	0.0159	0.0133	23,530	0.99	-	0.025

Table 5.5: Results of measures overview.

5.2 Analysis of Characteristics that influence Frequency of Bug Categories (RQ2)

This section presents a descriptive statistics to see which characteristics influence the frequency of the error categories of the analyzed projects. These characteristics are programming paradigms, programming language, and domains. The data is summarized in tables and bar charts.

1: Influence of the programming paradigm on the frequency of bug categories.

In table 5.6, we have an overview of all functional languages and their bug categories. To better compare these results, we have used percentages as the unit of measure. In table 5.8, we have the percentage of bug categories of the functional programming languages. In table 5.7, we have an overview of all imperative languages and their bug categories and in table 5.9, we have the percentage of error categories of the imperative ones.

Here, we look at the programming paradigms. As we can see, there is not much difference in the concurrency categories and failure categories between the two paradigms. What is different, however, is the frequency of algorithm and security errors, although the difference is not very large, only 0.3%. The frequency of programming errors is higher in the functional paradigm compared to the imperative paradigm, 93.3% versus 92.4%.

In general, there are not so many relevant differences that would allow us to draw a conclusion. What stands out is the programming category, where we can see almost 1% difference between the two paradigms. This suggests that the functional paradigm has a greater impact on the frequency of programming bugs than the imperative one.

2: Influence of the programming language on the frequency of bug categories.

The choice of programming language is another interesting feature to see how the frequency of error categories might be affected. This can be seen in table 5.6 and 5.8 for functional programming languages and in table 5.7 and 5.9 for imperative programming languages.
In the algorithm category, there is a small difference between the functional and imperative languages, almost 0.3% more for the imperative. In the concurrency category, there is no significant difference between the programming languages. It is interesting to note that in the memory category, almost all languages have a range between 0.6% and 1.2%, with the exception of the imperative language Go with 1.7%. This could mean that the choice of using the language Go can influence the frequency of memory bug in projects.
Continuing with the programming category, we can see that all imperative languages are in the range between 92% and 92.8%, while functional languages are in the range between 93% and 94.4%, with the exception of Erlang with 90.4%. This could lead to the conclusion that functional programming languages, with the exception of Erlang, have a greater impact on the frequency of programming errors than imperative languages.
For the security category there is no significant difference between functional and imperative languages. The same applies for the performance category.
For the fail category, we can see that Erlang has 4% failure errors, almost 2% more than the imperative languages. This suggests that Erlang has a greater impact on the frequency of failure errors than imperative languages. Lastly, for the unknown category we have the same value, 1,6%.

In conclusion, we found that some languages have a greater impact on frequency of specific bugs than others. For the programming category, we saw that Clojure, F#, and Haskell have a greater influence on the frequency of this type of bug than the imperative languages. Meanwhile, Erlang has a greater impact on the frequency of failure bugs than the imperative languages.

Functional Languages	Alg	Conc	Mem	Prog	Sec	Perf	Fail	Unkn	Count
Clojure	28	2	30	3077	29	29	55	66	3316
Erlang	28	1	26	2920	42	16	131	65	3229
F#	30	1	58	4619	15	25	87	56	4891
Haskell	46	1	93	7238	29	30	127	117	7681
Summary	**132**	**5**	**207**	**17.854**	**115**	**100**	**400**	**304**	**19.117**

Table 5.6: Categories of bugs for functional languages.

Imperative Languages	Alg	Conc	Mem	Prog	Sec	Perf	Fail	Unkn	Count
Go	296	18	519	27.746	223	249	664	425	30.140
JavaScript	92	1	97	8.223	72	53	184	137	8.859
Python	253	4	241	25.444	253	188	480	505	27368
Ruby	128	3	87	12.935	151	112	317	296	14.029
Summary	**769**	**26**	**938**	**74.348**	**699**	**602**	**1.645**	**1.363**	**80.396**

Table 5.7: Categories of bugs for imperative languages.

3: Influence of the project domain on the frequency of bug categories.

Here, we analyzed the obtained domains to see which type of domain influence the frequency of the bug categories of the analyzed projects. For this purpose, we created bar charts and converted the value in percent to make the comparison easier.
In table 5.10, it is possible to see the distribution of domains of the 200 analyzed functional projects. A large part is made up by the "Library" domain with 67 projects, followed by the "Application" domain with 37 projects. At the end we have four domains with almost the same number of projects, "Framework" with 22, "Other" with 23, "Database" with 11 and "Programming Language" with 12. On the other hand, in table 5.11, we have the distribution of domains of the 200 imperative projects analyzed. Again, a large part is made up by the "Library" domain with 65 projects, followed by the "Application" domain with 56 projects. Finally, we have two domains with almost the same number of projects, "Framework" with 35 and "Other" with 32. The remaining domains (Database, Code Analyzer and Programming Language) all have small numbers, between one and six.

After this first introduction, we can now proceed to analyze in detail the relationship between the different domains and categories of errors.
Starting with the "Application" domain, we can see in figure 5.1, that the most largest difference is in the amount of programming bugs between functional and imperative projects. Imperative projects have almost twice as many programming bugs as the functional projects. However, we must take into account the number of projects that fall under the "Application" domain: 37 for the functional and 56 for the imperative projects. Nevertheless, we cannot assume that with the same number of projects, the functional ones will have more % of programming bugs. In conclusion,

Functional Languages	Alg	Conc	Mem	Prog	Sec	Perf	Fail	Unkn
Clojure	0,8%	0,006%	0,9%	93%	0,9%	0,9%	1,6%	1,9%
Erlang	0,8%	0,003%	0,8%	90,4%	1,3%	0,5%	4%	2%
F#	0,6%	0,003%	1,1%	94,4%	0,4%	0,5%	1,8%	1,1%
Haskell	0,6%	0,001%	1,2%	94,2%	0,4%	0,4%	1,6%	1,5%
Summary	**0,6%**	**0,002%**	**1%**	**93,3%**	**0,6%**	**0,5%**	**2%**	**1,6%**

Table 5.8: Percentage of categories of bugs for functional languages.

Imperative Languages	Alg	Conc	Mem	Prog	Sec	Perf	Fail	Unkn
Go	1%	0,005%	1,7%	92%	0,7%	0,8%	2,2%	1,4%
JavaScript	1%	0,001%	1,1%	92,8 %	0,8%	0,5%	2,1%	1,5%
Python	0,9%	0,001%	0,8%	92,8 %	0,9%	0,6%	1,7%	1,8%
Ruby	0,9%	0,002%	0,6%	92%	1%	0,7%	2,2%	2,1%
Summary	**0,9%**	**0,003%**	**1,1%**	**92,4%**	**0,9%**	**0,7%**	**2%**	**1,6%**

Table 5.9: Percentage of categories of bugs for imperative languages.

we can say that the choice of the “Application” domain for the imperative paradigm has a greater impact on the frequency of programming bugs.

The second domain is the “Database” domain, which we can see in figure 5.2. Again, we can see that the most largest difference is in the amount of programming bugs between functional and imperative projects. Imperative projects have 8% more programming bugs than functional projects, although there are more functional than imperative projects (11 versus 6). We can say that the choice of “Database” domain for imperative paradigm has a greater impact on the frequency of programming bugs.

Proceeding with the “Code Analyzer” domain, which we can see in figure 5.3, we can note that a big part of functional programming bug is concentrated in this domain, 23.6%. Surprisingly the imperative projects contains almost no bugs of this type here. However, we must take into account the number of projects that fall under the code analyzer domain: 28 for the functional and 5 for the imperative projects, almost six times more functional projects than imperative ones. Still, we cannot assume that with the same number of projects, the imperative ones will have more % of bugs, in particular programming bugs.
To conclude, we can say that the choice of the “Code Analyzer” domain for the functional paradigm has a greater impact on the frequency of programming bugs.

In the “Library” domain, figure 5.4, we have a better overview than the other domains, since the number of projects is almost the same. As with the other domains, the bugs here are concentrated around the programming category, with a difference of 3% between functional and imperative

projects. From this difference, we can deduce that the choice of "Library" domain for imperative paradigm has a greater impact on the frequency of programming bugs.

Although 57 projects are included in the "Framework" domain, as we can see in figure 5.5, only a small % of the total bugs are found in this domain. We can see that 6% of the total bugs for functional projects and 5% of the total bugs for imperative projects. With a difference of 1%, we can say that the choice of "Framework" domain for functional paradigm has a greater impact on the frequency of programming bugs. We cannot draw any conclusions for other categories of bugs because the difference is too small.

Concluding with the "Programming language" domain, figure 5.6, we can see that only one imperative project was classified as such. The difference between the categories is so small that % is annotated with 0 (smaller than 0.01%). The only largest difference is found in programming bugs. Functional projects have 5.7% more programming bugs than imperative projects. This suggests that the choice of "Programming Language" domain for the functional paradigm has a greater impact on the frequency of programming bugs.

Functional Langauges	**APP**	**DB**	**CA**	**LIB**	**FW**	**PL**	**OTH**
Clojure	12	3	5	22	5	1	2
Erlang	8	4	4	17	5	3	9
F#	5	3	8	20	6	1	7
Haskell	12	1	11	8	6	7	5
Summary	**37**	**11**	**28**	**67**	**22**	**12**	**23**

Table 5.10: Domains of functional language projects.

Imperative Langauges	**APP**	**DB**	**CA**	**LIB**	**FW**	**PL**	**OTH**
Go	16	3	1	10	10	1	9
JavaScript	9	1	1	20	14	0	5
Python	18	1	1	15	5	0	10
Ruby	13	1	2	20	6	0	8
Summary	**56**	**6**	**1**	**65**	**35**	**1**	**32**

Table 5.11: Domains of imperative language projects.

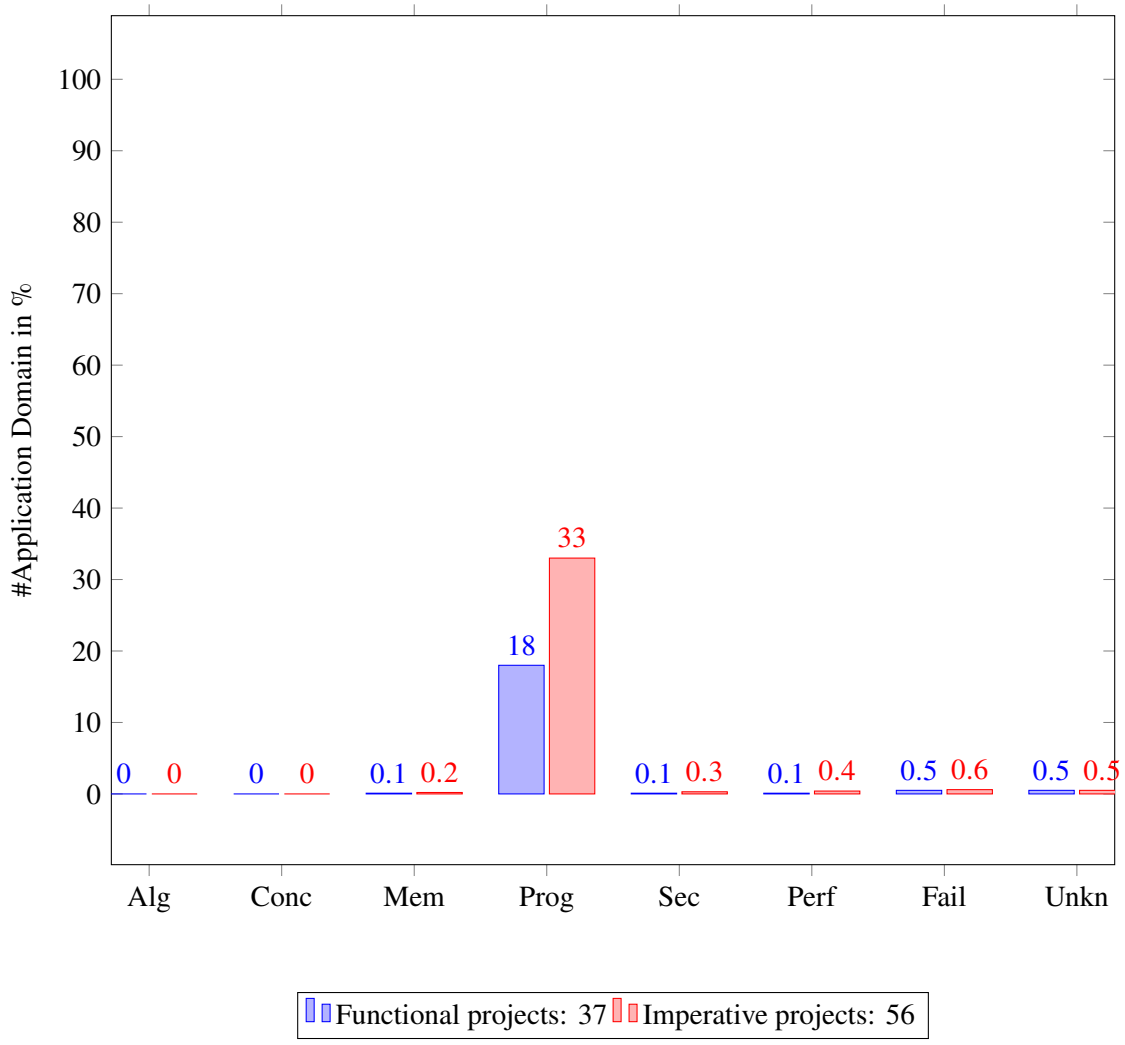

Figure 5.1: Application domain with bug categories in %

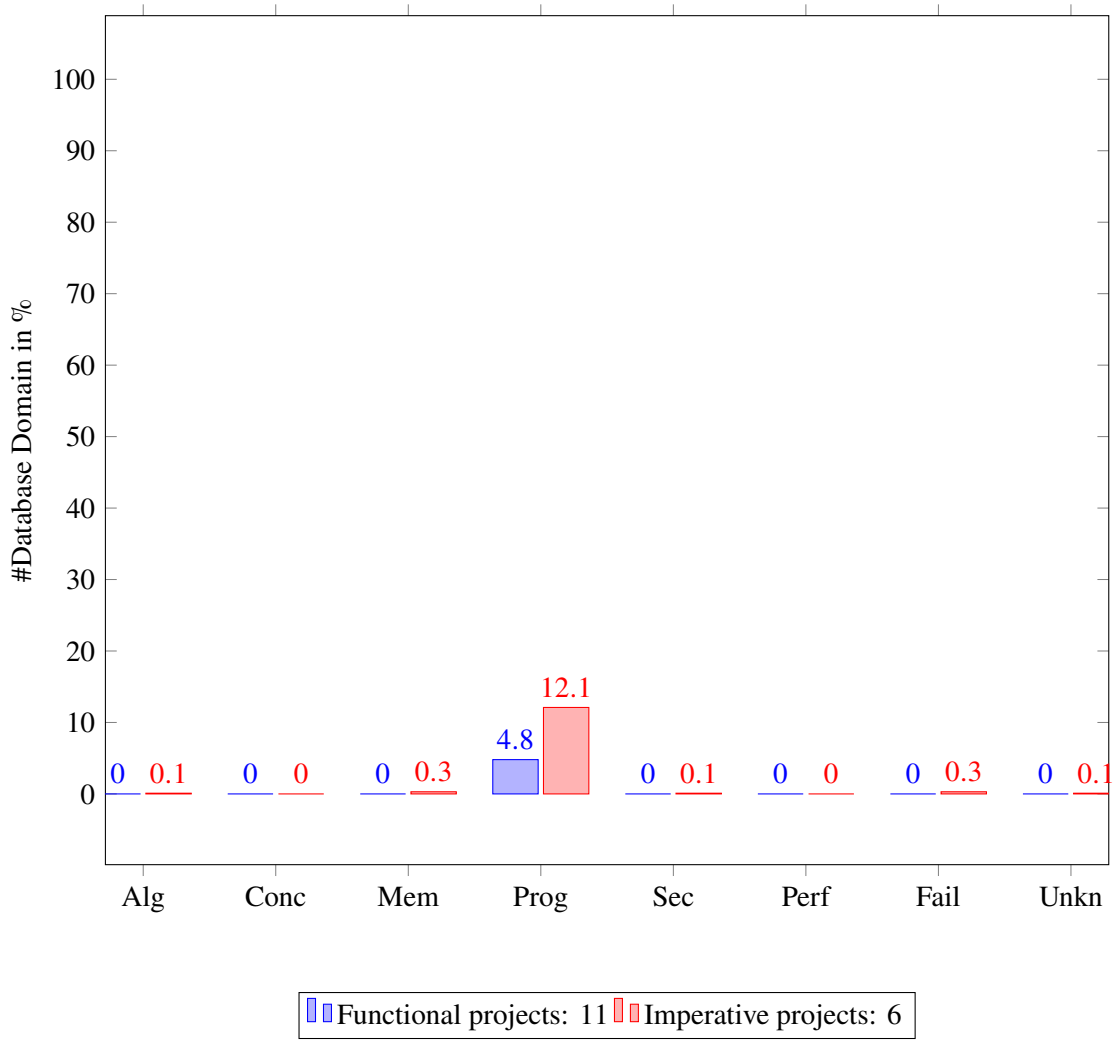

Figure 5.2: Database domain with bug categories in %

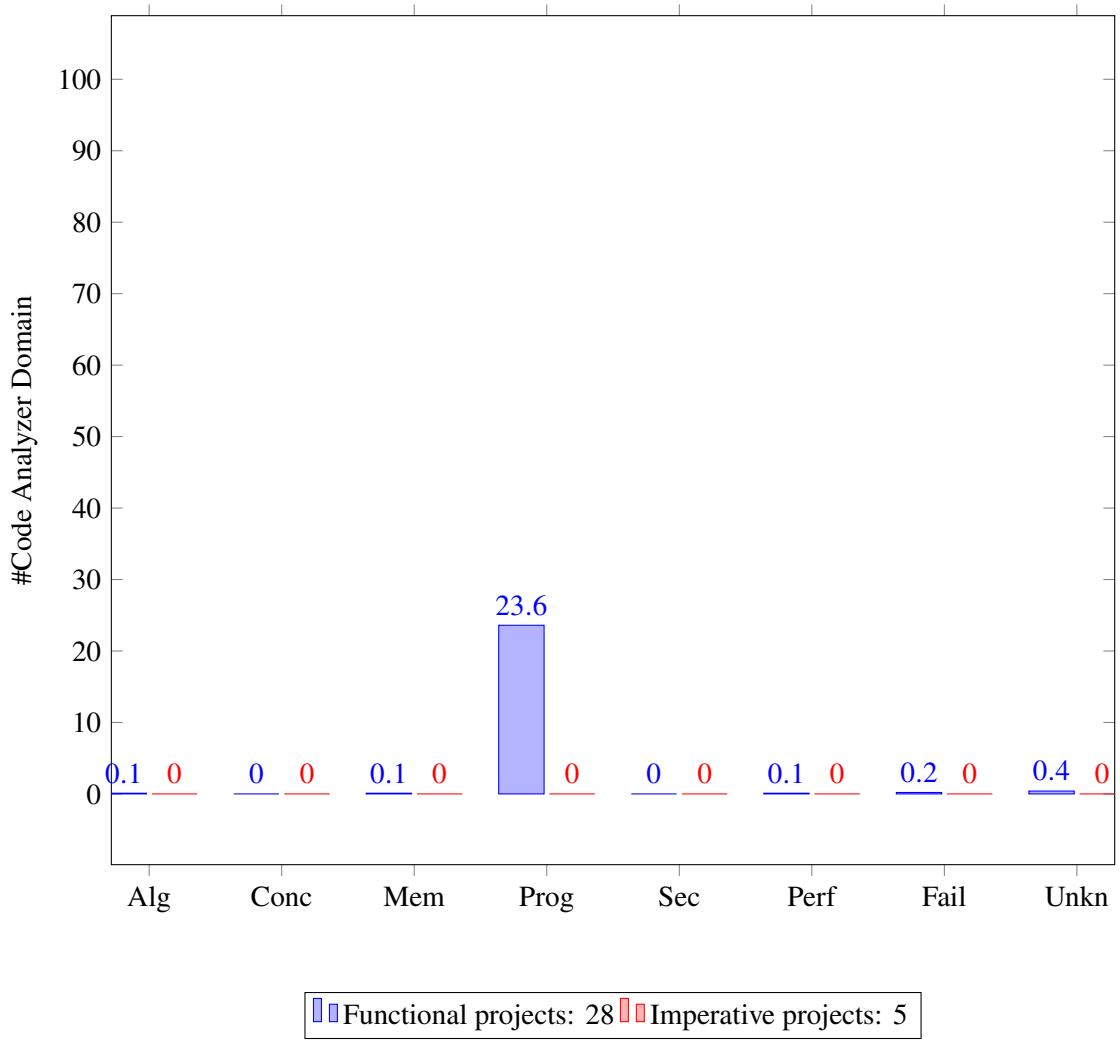

Figure 5.3: Code Analyzer domain with bug categories in %

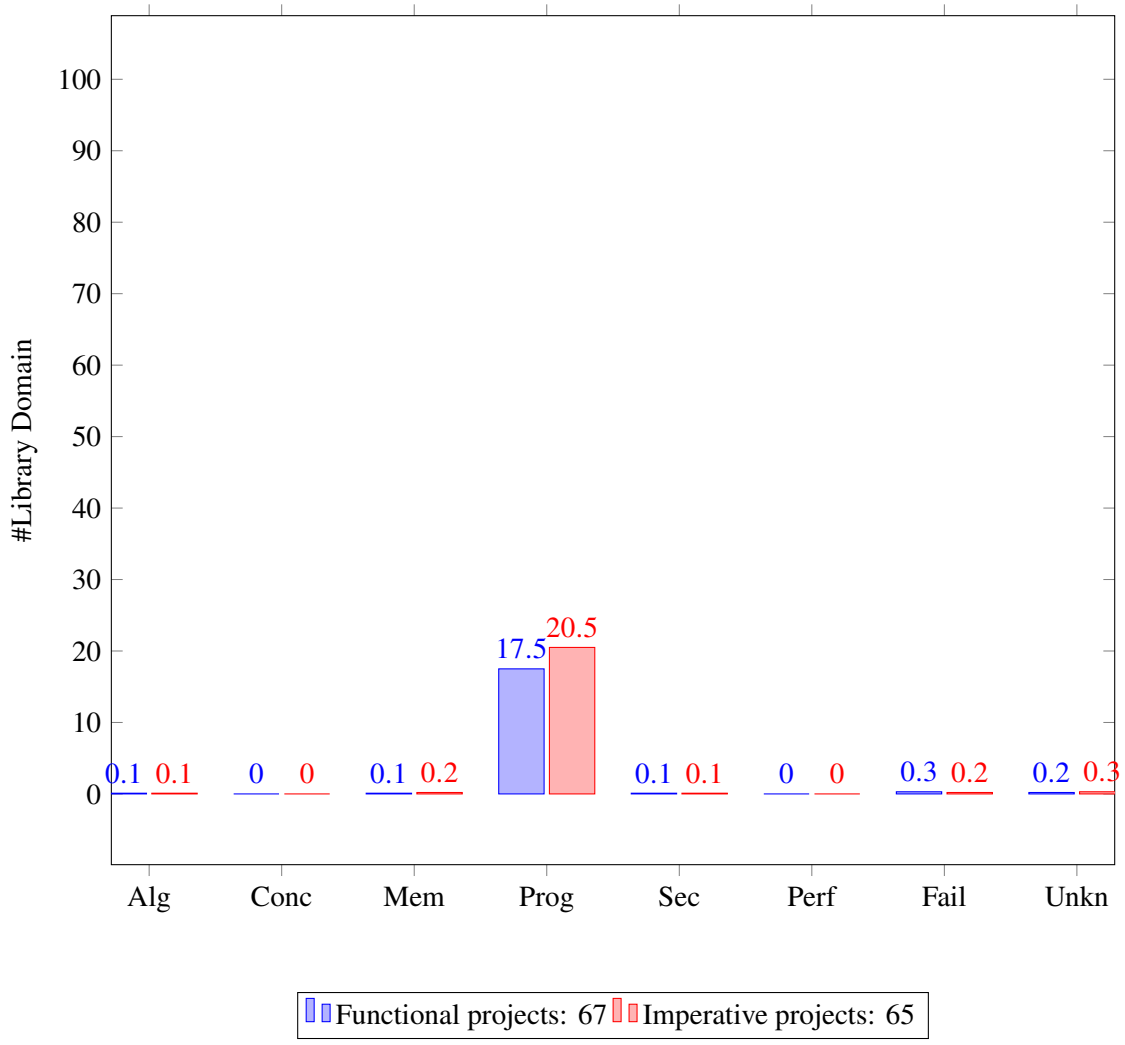

Figure 5.4: Library domain with bug categories in %

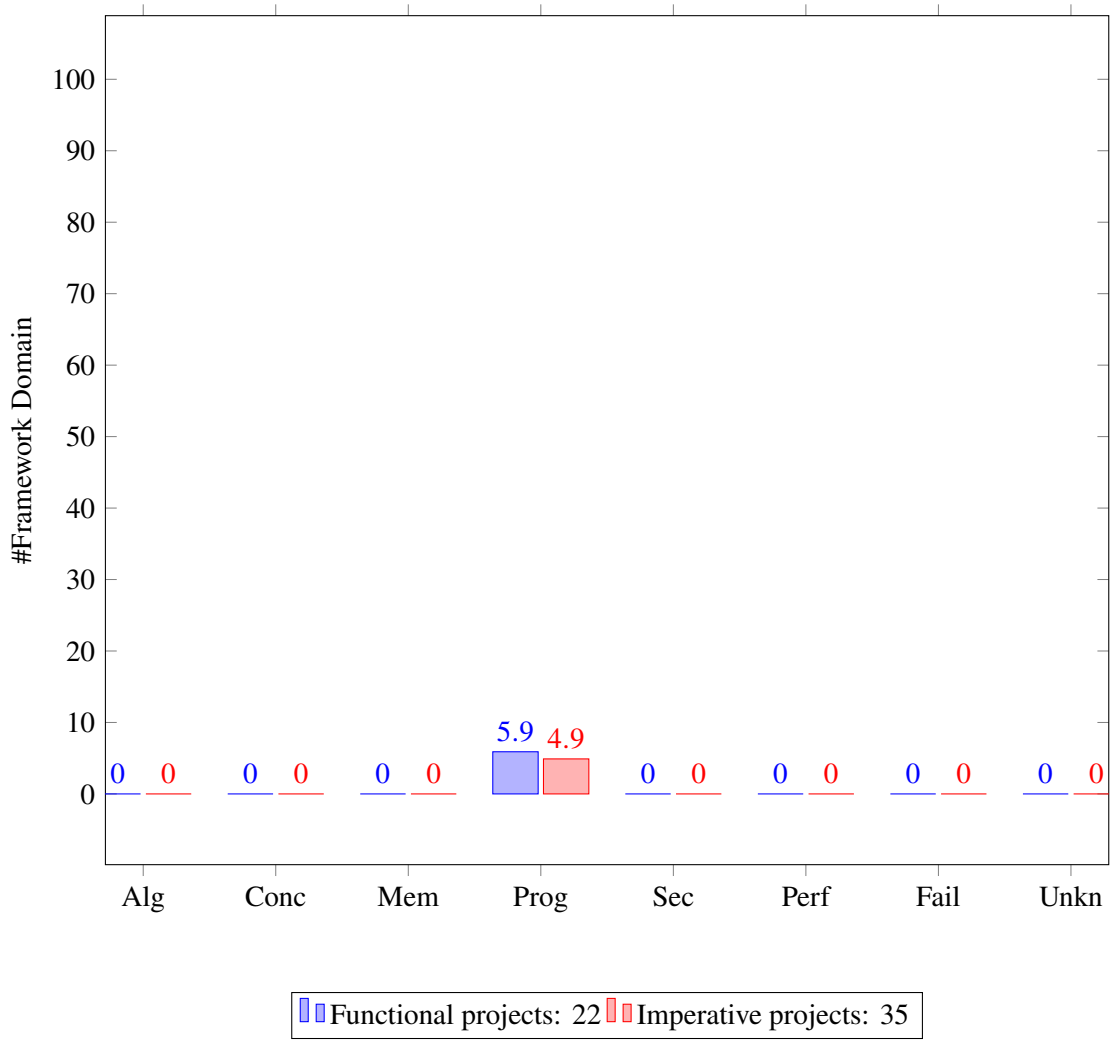

Figure 5.5: Framework domain with bug categories in %

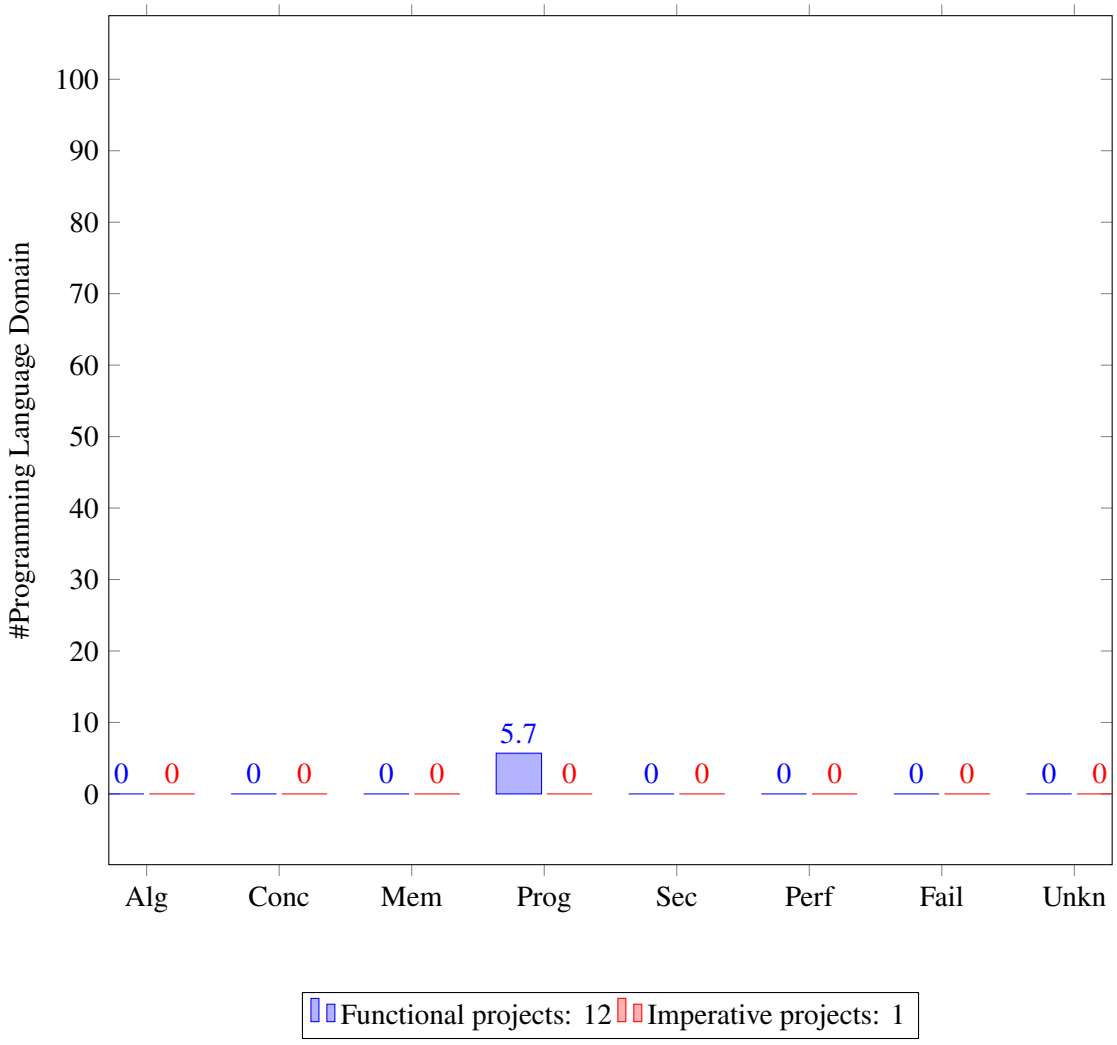

Figure 5.6: Programming Language domain with bug categories in %

6 Discussion

In this chapter we discuss the results and present threats to validity.

6.1 Results Discussion

We saw that one of our hypotheses could be supported, namely the first one about reliability. This indicated that functional programming languages have a small-medium positive impact on reliability compared to imperative programming languages. One explanation for this result could be the less and more concise code that functional programming languages provide compared to imperative languages [NF15]. More concise code could lead to less code and less code to review. This brings us to the idea that with less code, we have fewer things to test and therefore fewer bugs in production. Unfortunately, there are not so many studies that have analyzed this aspect, so we cannot generalize.

The other assumption about maintainability could not be sustained. This may be due to the rigidity of the rules that the static analysis tool for functional languages have, which may cause more code smells. We found that more than half of the projects in Clojure did not have a plugin for the static analysis tool in their project. This could lead to the assumption that the project was not tested against code smells. For imperative languages, we found that code smell discovery tools were integrated into the projects. For example, many JavaScript projects had the ESLint [1] plugin integrated. This could lead to few code smells in the final project we analyzed. Another explanation could be the use of the static analysis tool. We have seen that in Erlang almost each line of code contained on average 0.35 code smells. This could lead to the conclusion that the plugin used to analyze the projects might have too many false positives.

We also saw that the functional paradigm has a greater impact on the frequency of programming bugs that the imperative one. This result could be due to the fact that developers find it easier to detect bugs. This leads to more refactoring and thus more programming bugs discovered and fixed. Then for programming languages we had that some languages have a greater impact on frequency of specific bugs than others. Programming languages like Clojure, F#, and Haskell have a greater influence on the frequency of programming bugs than the imperative languages. As for the functional paradigm, we can say the same here. Bugs in a functional project could be easier to detect and through refactoring or general fixing we could have more bugs related to the programming category.
For Erlang we saw that this language has a greater impact on the frequency of failure bugs than the imperative languages. This result could be due to the distribution of domains of the projects we

[1] ESLint plugin. https://eslint.org

had for Erlang. We had found that 28 out of 50 projects were associated with domains that have a higher probability of containing a bug of the failure type. For example, domains such as databases or applications, where it is possible to find bugs that can crash or hang the application.

Meanwhile for the code analyzer, framework, and programming language we had that functional languages have a greater impact on the frequency of programming bugs. This could be due to the complexity of these domains, which could lead to more generic programming errors, and due to the functional languages properties, such errors could be easier to find and fix than for imperative languages.

6.2 Threats to Validity

There are a few threats to our reported results. The first is the choice of the static analysis tool we used to gather information about code smells. Because of the various plugins this tool uses to analyze projects, it is not possible to determine if one language is better analyzed than another. We based our results on the number of rules, which each language on SonarQube has. There was no significant difference between the number of rules each language had regarding code smells. Thus, we concluded that each language was analyzed in the same way.

The second is the procedure we chose to identify bugs from commits. We chose to look for keywords that might indicate a bug. This could lead to some false positives, commits that contain some keywords related to bugs or fixes but are not bugs. For this purpose, we tried to compare the bugs we found with the issue tracker that each repository has on GitHub. Unfortunately, this did not improve the validity of our data, as not all issues were bugs or marked as such. We found that some bugs were not present in the issue tracker but were present in the commits, or that issues that were labeled as bugs were not actually bugs. To give the data some validity, we chose two random commits from each project and manually inspected them to see how good the accuracy was. Out of 800 commits, only five were false positives.

The third threat to validity is bug categorization. For this task, we implemented a categorization algorithm using a linear model. We trained this algorithm with manually categorized and selected data. The accuracy was 89% to categorize a bug in the right category. Due to some bugs that could be false positives, the categorization could be affected.

Another threat to validity is the sampling process. We selected projects from GitHub based on the stars, the language of a project, and the number of commits each project must have. We did not pay attention to the domains of these projects. This could result in an unrepresentative number of projects being used for the sampling process. The biggest difference is in the domains of the applications and the code analyzers. For the functional languages, we had 37 application projects versus the 56 application projects for imperative languages. For the code analyzer domain, we had 28 projects for functional languages but only five for imperative languages.

Among the projects we used as a sample, some were actively developed and sponsored by companies. This could influence the data obtained in this way. For example “consul” [2], “Capistrano” [3], or “go-micro” [4] are some of these type of projects that we analyzed.
They may have stricter quality assurance processes than other projects hosted on GitHub, or they may have adopted a management process like Scrum to better organize the development process. This could be seen in the way the issues tracker was organized. Projects with sponsor or projects that belong to companies had a good management of the issue tracker with good labels.

[2] Consul https://www.consul.io
[3] Capistrano https://capistranorb.com
[4] go-micro https://github.com/asim/go-micro

7 Conclusion

We performed an empirical study to analyze whether functional programming improves software quality. For this study, 400 different projects were analyzed from GitHub. Through this analysis, we were able to analyze two aspects of software quality, reliability and maintainability. By categorizing bug commits and project domains, we were able to see what characteristics influence the frequency of bug categories of the analyzed projects.

The data indicate a small-medium effect that using a functional programming language improves reliability. This suggests that functional programming languages are less prone to bugs than imperative ones. For maintainability, there is not enough evidence to show that functional programming languages have less code smells than imperative programming. So all in all, it cannot be said that functional programming improves software quality. It is interesting to see that the functional paradigm has a greater impact on the frequency of programming bugs than the imperative one. Going into details, we can see how functional languages like Clojure, F#, and Haskell are more prone to bugs caused by generic programming errors than other imperative languages. We also found that Erlang has a greater impact on the frequency of failure-based bugs than imperative languages.
The results also indicate that for the application, database, and library domains, the imperative paradigm has a greater impact on the frequency of programming bugs. While for the code analyzer, framework and programming language the functional paradigm has a greater impact on the frequency of programming bugs. For the remaining categories, there was no significant difference that could lead to a comparison.

More research is needed on the impact of functional languages on software quality. Other aspects of software quality could be analyzed to get a better overview. Further studies in companies that use functional programming languages to develop applications could help gather more detailed information that would lead to more accurate conclusions. Experiments that could research how easy it is to learn and understand functional languages compared to imperative languages could also help draw some conclusions about software quality.

Bibliography

[al16] E. K. et al. “An in-depth study of the promises and perils of mining GitHub”. In: *Empirical Software Engineering* (2016), pp. 2035–2071. URL: https://link.springer.com/article/10.1007/s10664-015-9393-5 (cit. on p. 22).

[al17] B. R. et. al. “A Large-Scale Study of Programming Languages and Code Quality in Github”. In: *Reseaarch Topics in Functional Programming* 60 (2017), pp. 91–100 (cit. on pp. 12, 19, 26).

[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. 13, 19).

[BL92] S. Brocklehurst, B. Littlewood. “New Ways to Get Accurate Reliability Measures”. In: *Software, IEEE* 9 (Aug. 1992), pp. 34–42. DOI: 10.1109/52.143100 (cit. on p. 28).

[BMP13] H. (Barendregt, G. Manzonetto, R. Plasmeijer. “The Imperative and Functional Programming Paradigm”. In: (June 2013) (cit. on pp. 15, 16).

[BNJ03] D. M. Blei, A. Y. Ng, M. I. Jordan. “Latent Dirichlet Allocation”. In: *J. Mach. Learn. Res.* 3.null (Mar. 2003), pp. 993–1022. ISSN: 1532-4435 (cit. on p. 25).

[Buo17] E. Buonanno. *Functional Programming in C: How to write better C code*. German. 1st. Manning Publications, 2017 (cit. on p. 16).

[CF09] G. Corder, D. Foreman. *Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach*. German. 1. Auflage. John Wiley Sons, 2009 (cit. on p. 30).

[CMHB18] J. Carter, A. Miller, S. Halloway, A. Bedra. *Programming Clojure (The Pragmatic Programmers)*. German. 3rd ed. Pragmatic Bookshelf, 2018 (cit. on p. 16).

[Goe19] S. Goericke. *The Future of Software Quality Assurance*. New York, Vereinigte Staaten: Springer Publishing, 2019 (cit. on p. 11).

[Gui16] S. Guido. *Introduction to Machine Learning with Python: A Guide for Data Scientists*. German. 1st ed. O’Reilly UK Ltd., 2016 (cit. on p. 26).

[Hay13] W. Haynes. “Bonferroni Correction”. In: *Encyclopedia of Systems Biology*. Ed. by W. Dubitzky, O. Wolkenhauer, K.-H. Cho, H. Yokota. New York, NY: Springer New York, 2013, pp. 154–154. ISBN: 978-1-4419-9863-7. DOI: 10.1007/978-1-4419-9863-7_1213. URL: https://doi.org/10.1007/978-1-4419-9863-7_1213 (cit. on p. 30).

[HSDL96] R. Harrison, L. Smaraweera, M. Dobie, P. Lewis. “Comparing Programming Paradigms: an Evaluation of Functional and Object-Oriented Programs”. In: *Software Engineering Journal* 11 (Aug. 1996), pp. 247–254. DOI: 10.1049/sej.1996.0030 (cit. on p. 19).

[Hug90] J. Hughes. “Why Functional Programming Matters”. In: *Research Topics in Functional Programming* 30 (1990), pp. 17–42 (cit. on pp. 11, 12).

[KI16] J. Kunasaikaran, A. Iqbal. “A Brief Overview of Functional Programming Languages”. In: *electronic Journal of Computer Science and Information Technology* 6 (Dec. 2016), p. 32 (cit. on pp. 15, 16).

[KMB04] C. Kaner, S. Member, W. P. Bond. “Software Engineering Metrics: What Do They Measure and How Do We Know?” In: *In METRICS 2004. IEEE CS*. Press, 2004 (cit. on p. 28).

[KWL16] P. S. Kochhar, D. Wijedasa, D. Lo. “A Large Scale Study of Multiple Programming Languages and Code Quality”. In: Mar. 2016, pp. 563–573. DOI: 10.1109/SANER.2016.112 (cit. on p. 20).

[KY17] A. Khanfor, Y. Yang. “An Overview of Practical Impacts of Functional Programming”. In: Dec. 2017, pp. 50–54. DOI: 10.1109/APSECW.2017.27 (cit. on p. 11).

[LHH19] H. Lane, H. Hapke, C. Howard. *Natural Language Processing in Action: Understanding, analyzing, and generating text with Python*. German. 1st. Manning Publications, 2019 (cit. on p. 26).

[LTW+06] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, C. Zhai. “Have things changed now?: An empirical study of bug characteristics in modern open source software”. In: Jan. 2006, pp. 25–33. DOI: 10.1145/1181309.1181314 (cit. on p. 26).

[Mic11] G. Michaelson. *An Introduction to Functional Programming Through Lambda Calculus (Dover Books on Mathematics)*. German. Dover Publications Inc., 2011 (cit. on p. 15).

[NF15] S. Nanz, C. A. Furia. “A Comparative Study of Programming Languages in Rosetta Code”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* (May 2015). DOI: 10.1109/icse.2015.90. URL: http://dx.doi.org/10.1109/ICSE.2015.90 (cit. on p. 45).

[RCB81] A. Rosene, J. Connolly, K. Bracy. “Software Maintainability - What It Means and How to Achieve It”. In: *IEEE Transactions on Reliability* R-30.3 (1981), pp. 240–245. DOI: 10.1109/TR.1981.5221065 (cit. on p. 11).

[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. 20).

[SBCG98] P. Salus, W. Brainerd, R. Cytron, Grisworld, Ralph E." *Imperative Programming Languages (Handbook of Programming Languages, Band 2)*. German. Macmillan Technical Publishing, 1998 (cit. on p. 15).

[SIT18] K. Singh, A. Ianculescu, L. Torje. *Design Patterns and Best Practices in Java: A comprehensive guide to building smart and reusable code in Java (English Edition)*. German. Packt Publishing, 2018 (cit. on p. 15).

[Tho19] H. Thode. *Testing For Normality*. German. Routledge, 2019 (cit. on p. 29).

[Wag13] S. Wagner. *Software Product Quality Control*. German. 2013th ed. Springer, 2013 (cit. on p. 17).

[YM12] A. Yamashita, L. Moonen. “Do code smells reflect important maintainability aspects?” In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 2012, pp. 306–315. DOI: 10.1109/ICSM.2012.6405287 (cit. on p. 28).

All links were last followed on May 7, 2021.