UAM
UNIVERSIDAD AUTONOMA
DE MADRID

Escuela Politécnica Superior

20
21

# Degree work

## Open Source platform for code security and quality assessment

Javier Delgado del Cerro

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11


Campus Internacional excelencia UAM CSIC+

www.uam.es

Universidad Autónoma de Madrid

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

**Degree as Computer Engineering**

# DEGREE WORK

## Open Source platform for code security and quality assessment

**Author: Javier Delgado del Cerro**
**Advisor: Eduardo Cermeño Mediavilla**

**julio 2021**

**All rights reserved.**
No reproduction in any form of this book, in whole or in part
(except for brief quotation in critical articles or reviews),
may be made without written authorization from the publisher.

© 22 de Febrero de 2021 by UNIVERSIDAD AUTÓNOMA DE MADRID
Francisco Tomás y Valiente, n$^{o}$ 1
Madrid, 28049
Spain

**Javier Delgado del Cerro**
**Open Source platform for code security and quality assessment**

**Javier Delgado del Cerro**

PRINTED IN SPAIN

*To my grandparents, parents, and brother.*

*You can't connect the dots looking forward,*
*you can only connect them looking backwards.*
*So you have to trust that the dots will somehow connect in your future.*

*Steve Jobs.*

# Resumen

El software está cada vez más presente en nuestras vidas, y es necesario poder medir su calidad. En este trabajo, hacemos un estudio del estado del arte de la calidad del software, analizando las distintas definiciones teóricas y las herramientas prácticas utilizadas para medirla y mejorarla. Proponemos una nueva puntuación multi-parametrizable para medir la calidad del software en función de las prioridades de cada proyecto. En nuestro caso, empleamos un conjunto determinado de métricas obtenidas mediante herramientas open source, entre las que destaca la cantidad de errores reales presentes en el código, según su gravedad. Sin embargo, las métricas usadas pueden variar y la idea del método seguiría siendo válida. Con este método, hemos desarrollado una herramienta que nos permite automatizar la extracción de las métricas, y hemos analizado un total de 200 proyectos en cuatro lenguajes de programación, escogiendo los más populares por lenguaje según GitHub. Nuestros resultados muestran que la calidad asociada a cada lenguaje varía notablemente en función de la parametrización utilizada, obteniendo como resultado Java y C++ como lenguajes más recomendables, en función de si priorizamos mantenibilidad y rendimiento o un menor ratio de errores respectivamente.

# Palabras clave

lenguaje de programación, calidad del código, puntuación de la calidad, análisis estático, proyectos de código abierto, métricas software

# Abstract

Software is increasingly present in our lives, and it is necessary to measure and compare its quality. In this paper, we survey the state of the art of software quality, analyzing the different theoretical definitions and practical tools used to measure and improve it. We propose a new multi-parametrizable score to measure software quality according to the priorities of each project. In our case, we use a set of metrics obtained through open source tools, among which the number of actual errors and vulnerabilities present in the code, according to their severity, stands out. However, the metrics used may vary, and the idea of the method would still be valid. With this method, we have developed a tool that allows us to automate the extraction of the metrics, and we have analyzed a total of 200 projects in four programming languages, choosing the most popular ones per language according to GitHub. Our results show that the quality associated with each language varies considerably depending on the parametrization used, resulting in Java and C++ as the most recommended languages, depending on whether we prioritize maintainability and performance or a lower error rate, respectively.

# Keywords

programming language, code quality, quality score, static analysis, open source projects, software metrics

# Table of Contents


**1 Introduction** 1

**2 Related works** 3

2.1 Defining software quality . . . 3

2.1.1 Quality according to ISO/IEC 25000 . . . 4

2.1.2 Quality according to CISQ . . . 5

2.1.3 The problem with quality metrics . . . 5

2.2 Practical applications to measure software quality . . . 6

2.2.1 Platforms with multiple interrelated tools . . . 7

2.2.2 Platforms based on static analysis . . . 8

2.2.3 Simple tools . . . 8

**3 Proposed method** 11

3.1 Metrics . . . 11

3.1.1 Error rate by severity . . . 11

3.1.2 Cyclomatic complexity . . . 12

3.1.3 Percentage of comments . . . 12

3.1.4 Duplicated code . . . 13

3.2 Scoring parametrization . . . 14

**4 Method validation** 17

4.1 Preparation of the experiments . . . 17

4.1.1 Selection of programming languages . . . 17

4.1.2 Selection of evaluation tools . . . 19

4.1.3 Development of the evaluation platform . . . 20

4.1.4 Database description . . . 23

4.2 Experiments . . . 24

4.2.1 Experiment I: Error weights relatives to their severity . . . 24

4.2.2 Experiment II: Error weights relatives to their severity (II) . . . 25

4.2.3 Experiment III: Focus on errors . . . 25

4.2.4 Experiment IV: Ignore errors . . . 25

4.2.5 Experiment V: Uniform weights . . . 26

4.2.6 Some more collected data . . . 26

4.3 Results . . . 26

4.3.1 Parametrization-independent results . . . 27

4.3.2 Parametrization-dependent results . . . 28

**5 Discussion** **31**

5.1 Isolated metrics analysis . . . 31

5.2 Multi-parametric analysis . . . 32

5.3 Correlations analysis . . . 33

5.4 Threats to validity . . . 33

5.4.1 Threats to the errors found by the tools . . . 33

5.4.2 Threats due to the set of metrics provided by the tools . . . 34

**6 Conclusions** **37**

6.1 Future work . . . 38

**Bibliography** **40**

**Acronyms** **41**

# Lists

## List of figures


3.1 Diagram of duplicated code . . . . . 13

4.1 Stack Overflow Developer Survey 2020 . . . . . 18
4.2 TIOBE Index in March, 2021 . . . . . 18
4.3 Platform's *Errors* page . . . . . 20
4.4 Platform's *Dashboard* page . . . . . 21
4.5 Platform's *Group dashboard* page . . . . . 22
4.6 Platform's *Features* page . . . . . 23
4.7 Cyclomatic complexity and percentage of comments per language on GitHub ranking . 27
4.8 Average number of duplications and number of hundreds of lines per duplication per language on GitHub ranking . . . . . 27
4.9 Rate of errors per severity and language on GitHub ranking . . . . . 28
4.10 Weighted severity and score per language on GitHub ranking. Weights as expressed in Section 4.2.1 . . . . . 28
4.11 Weighted severity and score per language on GitHub ranking. Weights, as expressed in Section 4.2.2, . . . . . 29
4.12 Weighted severity and score per language on GitHub ranking. Weights, as expressed in Section 4.2.3, . . . . . 29
4.13 Weighted severity and score per language on GitHub ranking. Weights, as expressed in Section 4.2.5, . . . . . 30
4.14 Weighted severity and score per language on GitHub ranking. Weights, as expressed in Section 4.2.4, . . . . . 30
4.15 Matrix of correlations between the calculated metrics and some attributes intrinsic to GitHub Projects . . . . . 30


## List of tables


4.1 Top 10 projects for each language on GitHub, sorted by Stars and Forks. . . . . . 24
4.2 Weights used in Section 4.2.1 . . . . . 25
4.3 Weights used in Section 4.2.1 with slight variations. . . . . . 25
4.4 Weights used in Section 4.2.2 . . . . . 25
4.5 Weights used in Section 4.2.3 . . . . . 26

4.6 Weights used in Section 4.2.4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
4.7 Weights used in Section 4.2.5 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

# 1 Introduction

During the last decades, we have witnessed an impressive development of technology, where the appearance of computers accessible to the general public, cell phones, and smartphones, has made software increasingly present in our lives. In addition, more and more of the gadgets we use in our daily lives depend on such software: intelligent light bulbs controlled through the internet, electric toothbrushes with mobile applications, smartwatches, and even cars, such as Teslas, wholly connected to the internet, which base their entire operation on software, and that in a relatively near future, will be able to drive for us, taking care of keeping us safe, and having our lives at their disposal.

This new paradigm we are facing makes it necessary to ensure that the code used in any device is of the highest possible quality to avoid endangering our data, our devices, or even ourselves.

In addition, the rapid evolution of technology in recent years forces software to advance consistently, as fast as possible, thus forcing developers to adopt a component-based approach and use external libraries developed by other developers or companies. Although this can make working on a project much easier, it can also introduce potential bugs or errors, making it necessary to be able to assess the quality of such projects before using them.

This is one of the reasons why multiple programming languages of all kinds have been developed in the last decades. Garbage collection, object-oriented programming, and functional programming are some of the technologies that have emerged to facilitate the development process and improve software quality. Furthermore, most used programming languages are updated every couple of years, or even twice a year in the case of Java, to add new features and solve bugs. There is even an (unsuccessful) attempt to create a programming language explicitly to facilitate software quality [1].

This work aims to propose an innovative parametrizable software project evaluation method that allows the analysis of the quality of software of different programming languages. With this method, we will perform multiple experiments to try to determine the most appropriate programming language in terms of quality, depending on the project's priorities.

The rest of the work is organized as follows. In Chapter 2 we survey the state of the art of software quality to determine what is the quality of a software project, how to measure it, and what practical

utilities are currently being used to determine and improve software quality. In Chapter 3 we discuss the set of metrics we are using and design a parametrizable score for the quality of a software project based on the priorities of the developer or project. In Chapter 4 we develop a tool based on open source projects to extract those metrics and calculate the parametrizable score. We also explain the used database and the experiments performed to determine the best programming language in terms of software quality using our innovative score. The results of these experiments are shown at the end of this chapter. These experiments and their limitations are discussed in Chapter 5. Finally, we state some conclusions in Chapter 6.

# 2

# Related works

Looking at the state of the art of software quality evaluation, we can see that there is plenty of work which we can classify from two points of view. On the one hand, there are lots of papers studying how various factors affect software quality. On the other hand, many platforms and tools are used to assess software quality and help developers and companies improve their software. Therefore, in this chapter, we will analyze the most important works from both of these points of view.

## 2.1. Defining software quality

A good way to start the study of software quality is to explain this concept and determine the meaning we are going to use in this work. This will make it easier to evaluate the different tools available on the market to assess the quality of a code.

The concept of *software quality* is much more complex than it might seem at first glance and has been discussed many times throughout the short history of computer science, from different points of view. Following the model proposed by the Institute of Electrical and Electronics Engineers (IEEE) in [2], the different perspectives through which code quality can be defined would be:

- The *transcendental view*, which perceives quality as something that can be recognized but not defined.
- The *user view* perceives quality as the degree to which it is fit for purpose.
- The *manufacturing view* perceives quality as the adequacy to specification.
- The *product view* perceives quality as being linked to inherent characteristics of the product.
- The *value-based view* perceives quality as dependent on the quantity a customer is willing to pay for the product.

It is more than evident that these measures provide us with certain information about the quality of a product. However, in our case, we will understand quality from the product perspective, since it is the one that we can analyze independently of the users, the market, and the requirements. It is also the most objective one. From now on, and throughout this work, quality, as seen from the product perspective, will be identified as *code quality*.

With this perspective, we will determine which characteristics define code quality so that we can

then relate them to measurable attributes. Again, this subject has been discussed in depth by different organizations, and two approaches, commonly used in the literature to define software quality, stand out.

### 2.1.1. Quality according to ISO/IEC 25000

When talking about technology-related standards, there are two outstanding international non-governmental organizations. The first and most important one is the International Organization for Standardization (ISO) , formed by 165 different countries, which is considered the world's largest developer of international standards, with over twenty thousand standards set in all kinds of topics. If we focus on electronic, electrical and related technologies, the International Electrotechnical Commission (IEC) is the international organization involved in developing the necessary standards.

These two organizations cooperate closely, as in the case of the *ISO/IEC 9126*, the quality standard for software and systems up to 2011. In 2011, however, this standard was replaced by the *ISO/IEC 25000* family, also known as SQuaRE (System and Software Quality Requirements and Evaluation), which aims to create a common framework for evaluating software product quality. Within this family of standards, the *ISO/IEC 25010*, last updated in 2017, categorizes product quality into eight distinct characteristics (in turn divided into sub-characteristics that we will not mention as they do not provide any necessary information to this work):

- **Functional suitability:** The degree to which the product satisfies the required needs when used under the specified conditions.
- **Performance and efficiency:** Performance relative to the amount of resources used under the established conditions.
- **Compatibility:** The degree to which the product, system, or component can exchange information with other products, systems, or components and function while sharing the same hardware or software environment.
- **Usability:** The degree to which specified users can use the product or system to achieve specified objectives with effectiveness, efficiency, and satisfaction in the specified context of use.
- **Reliability:** The degree to which the system, product, or component performs the specified functions under the specified conditions and time period.
- **Security:** The degree to which the system or product protects information and data so that individuals, or other products or systems, have the appropriate degree of access to the data based on their type and level of authorization.
- **Maintainability:** The degree of effectiveness and efficiency with which a product or system can be modified by those responsible for its maintenance.
- **Portability:** The degree of effectiveness and efficiency with which a system, product, or component can be transferred from one hardware, software, or any environment, to another.

The ISO/IEC 25023 standard defines measures to quantitatively evaluate the quality of a software product in terms of these characteristics. However, these measures are made at the system or product performance level, not at the code level (which causes the problems). Furthermore, note that some of the proposed characteristics, as the *functional suitability* or *usability*, do not belong to what we have

considered as *quality of a software product*.

Even so, the current literature ( [3], [4]) shows that, among these characteristics, *functional suitability*, *usability*, *maintainability* and *portability* are highly related with some code characteristics as *complexity, coupling, documentation, degree of inheritance* and *size*.

### 2.1.2. Quality according to CISQ

Another of the leading organizations to take into account when talking about code quality is the Consortium for IT Software Quality (CISQ) . It is an organization focused on developing standards to improve software quality, formed by more than 1,500 members, as important as the European Union, the U.S. Department of Homeland Security, Amazon, Microsoft, Oracle or the IEEE .

To solve the fact that on the ISO/IEC standard the measurements are made at the system or product performance level, the CISQ decides to expand that standard by adding some measurements which can be computed directly from the code. This way, quality is related to found problems over a subset of four specified categories [5]. As we can see at [6], these four chosen categories, with their specification, would be the following ones:

- **Security:** It measures code weaknesses by representing the most important security flaws, taking into account the Top 25 of Common Weakness Enumeration (CWE) and the Top 10 of Open Web Application Security Project (OWASP) , two projects focused on determining and categorizing the most frequent software weaknesses and vulnerabilities.
- **Reliability:** It measures the weaknesses of the code that affect its availability, fault tolerance, and ease of recovering from faults.
- **Performance and efficiency:** It measures code weaknesses that influence response time, processor utilization, memory, and other resources.
- **Maintainability:** It measures code weaknesses that affect its understandability, ease of modification, testability, and scalability.

In this case, the sub-characteristics proposed in the ISO/IEC standard have been eliminated and used to determine the scope covered by each category. A series of CWE weaknesses are defined corresponding to the different categories that can be detected by analyzing the code. In total, there are about eighty-six rules.

### 2.1.3. The problem with quality metrics

Once we have defined the characteristics that every quality code should have, the ideal objective would be to be able to define a standardized metric that, for any code, would provide us with a score, say from 1 to 100. This ideal metric would be as objective as possible and perhaps broken down into sub-categories, which would provide us with accurate information about the code's quality and allow us to establish a threshold to decide when any project is or is not of quality.

This, however, is impossible, because depending on the needs of the system or software product, the complexity will be greater or lesser, the documentation will be more or less necessary, and the maintainability will be more or less critical. There are many variables that influence code quality [7], including the programming language itself: languages with strict typing or static typing, such as Java or C, are generally associated with higher quality results than those with weak typing or dynamic typing, such as Python. This relationship, however, could be because programmers with better working practices and who produce higher quality software opt for more strongly typed languages.

In the origins of code quality research, the usual procedure was the extraction of a series of metrics from the project code in an automated way, which were subsequently interpreted by an expert who manually examined the code to take into account its particularities and determine whether or not the project had the appropriate quality.

Thus, the approaches currently followed to determine code quality ( [8], [9]) are usually based on the extraction of a large number of metrics by static analysis of the code, and the use of different types of classifiers or neural networks to obtain a score for each of the categories of *complexity, coupling, documentation, degree of inheritance* and *size*. The ground truth determinant of the quality of the software product or system is usually the number of forks (times a project has been cloned) and stars (people who have given it a star to indicate that they like the project) of the code on Github.

This approach, however, also has its limitations, as it is quite questionable whether the forks or stars of a project in Github denote quality: there can be very unpopular projects with enormous quality, and trendy and useful projects with a large number of stars and forks, but very low quality.

Another approach commonly used nowadays is the analysis of commits of open source repositories. For example, it is pretty common to take open source projects hosted in GitHub and analyze the commit history, looking for messages related to bugs and fixes. In this way, the number of possible bugs that the programmers have found, and the solutions, are taken into account to compare the quality of different software projects. This approach is taken in [7] and [10], where the aim is to analyze how the programming language, or the combination of several languages within the same software project, influences its quality.

## 2.2. Practical applications to measure software quality

As there is a considerable amount of very diverse tools developed and used to measure and improve software quality, we think it makes sense to classify them according to the *completeness* of these tools. We understand *completeness* as the number of types of analysis they perform, the integration with other Continuous Integration/Continuous Deployment (CI/CD) tools and services, the number of programming languages they support, and the way they represent the information.

### 2.2.1. Platforms with multiple interrelated tools

The first type we are going to develop are platforms with a large number of tools, aimed at ensuring code quality in every possible way, but with slightly different objectives or methods, and which are integrated with each other.

The most representative and important example of this category (according to [11]) would be the multinational **Synopsys**. According to its website, it is an American software company founded in 1986 and currently has more than 15,000 employees worldwide. Among its products, it has a large number of tools to detect as many quality and safety defects as possible, all of them with integrations in multiple workflows. Some of the tools they offer are:

- **Coverity:** A static code security analysis tool that supports more than 21 programming languages such as C, C#, Java, Python, Javascript, and 70 frameworks such as Node.js or .NET Core. It integrates with a large number of IDEs such as IntelliJ, Eclipse, or Visual Studio.
- **Web scanner:** A dynamic web application analysis tool. It uses the application as a black box and looks for SQL injection, XSS vulnerabilities, etc.
- **Seeker:** An interactive web application analysis tool. It is a combination of static and dynamic tools, but more complex and modern. It works inside the application and analyzes the code and its operation, HTTP traffic, the libraries and frameworks used, and the behaviour during execution.
- **Tinfoil API Scanner:** A tool to detect vulnerabilities in APIs of any type: RESTful or Internet of Things devices, for example. It uses the documentation to detect all the endpoints with their parameters and generate the necessary tests.
- **Lack Duck:** A software composition analysis tool. It allows knowing the risks associated with libraries or third-party code used in an application. To do so, it uses a database with information about more than four million components.
- **Polaris:** A platform that brings together all the tools mentioned above to work directly on IDEs, repositories, and continuous integration tools such as Jenkins.

In general, all the companies mentioned in this section have tools for static code analysis (SAST), dynamic analysis (DAST), interactive analysis (IAST), software composition analysis (SCA), etc. Most of them are US companies such as *White hat security*, *Rapid7* (which stands out especially for its penetration testing software *Metasploit*), or *Contrast Security*. However, it is also important to mention *Checkmarx*, an Israeli company with about 500 employees, and *HCL Software*, an Indian multinational with more than 4,500 employees, which stands out for its *HCL AppScan* platform, consisting of tools of all the types specified above.

The platforms provided by these manufacturers are the most complete that exist, but also the most exclusive ones: in most cases, the price is not available on the websites, it is necessary to request a demo so that someone from the company can show you how they work if you want to use them, and in many cases, despite belonging to the same platform, they are promoted as different products, so they require separate payments. Everything seems to indicate that the price will be prohibitive for small companies, startups, or small developers who simply want to have some control over the quality and security of the code they develop.

### 2.2.2. Platforms based on static analysis

Slightly below (in terms of completeness) the platforms mentioned above are the ones mentioned in this section. These are platforms developed by smaller companies, which generally have a single tool that integrates with GitHub, GitLab, or BitBucket repositories, among others, to analyze the code and look for possible security flaws or problems that could compromise its quality. Thanks to the integration with repositories, they can explore each commit and pull request of a repository, determine which are safe or which add possible errors and should be reviewed or discarded, and even add notes to the commits and pull requests themselves. Generally, they have their own dashboard where you can view all the information obtained, such as bugs for each file, etc.

In general, they support several programming languages, among which the basic ones are always found, such as C, C++, Python, Java, and JavaScript. They do not offer any dynamic or interactive analysis. Instead, they only include static analysis, and, in some cases, component integration, detecting possible third-party libraries or modules that may introduce vulnerabilities.

Representative examples of this kind of platforms could be *SonarSource*, *Codacy*, *ShiftLeft NG SAST* o *DeepSource*, and, in general, they all offer fairly similar features mentioned above. It is common for them to offer the option of self-hosting so that large businesses can set up the platform on their own server and ensure the integrity of their data; for example, in the case of *SonarSource* is option is called *SonarQube*, and it allows integration with custom GitLab servers, or with GitHub Enterprise, while *SonarCloud* integrates directly with GitHub and GitLab, and process everything in their cloud. Finally, some of these platforms also offer integration with IDEs, such as *SonarLint* within *SonarSource*, which works as an extension to multiple IDEs like Eclipse, IntelliJ IDEA, or Visual Studio, to allow the developer to detect and fix quality issues while writing code.

Many of the solutions offer completely free plans for open source projects or small companies, and if this does not apply, the rates start at base prices of around $10-$15/month. This makes these platforms affordable for any small business.

Although some of these platforms, like *SonarQube*, are promoted as open source tools, there aren't really any of these tools that you can compile and use directly from the code deployed in their repository. They usually only have small snippets that are of no use.

### 2.2.3. Simple tools

This is the most extensive section by far, and it is composed of simple tools, most of them open source, that usually focus on analyzing code from a single programming language and looking for bugs or getting metrics. Many of these libraries are used by the platforms mentioned above to obtain their

data. For example, *Codacy* specifies in its own web page [1] which tools it uses for every language and objective.

Some of these tools simply use a database to look for potential errors on the code with the help of the GNU *grep* tool, as it could be *semgrep* [2] or *graudit* [3], which work on multiple programming languages like C, Go, Java and Python. This simplicity in their design makes them very susceptible to false positives: they detect as errors code that is actually correct.

Other tools are more complex and understand the syntax of the code, which allows them to detect errors more reliably and obtain different metrics, but forces them to specialize in fewer programming languages. For example:

- For C or C++, *flawfinder* [4] and *cppcheck* [5] detect possible security errors that may be related to pointers, overflows, out-of-index errors, etc., and tools such as *CMetrics* [6] and *cqmetrics* [7] generate metrics such as lines of code, cyclomatic complexity, etc.
- For Python, *bandit* [8] and *pylint* [9] look for programming errors, code smells, and that certain standards are satisfied, while *vulture* [10] finds unused classes, functions and variables, and *radon* [11] provides information on various metrics such as lines of code, lines of comments or cyclomatic complexity.
- For Java, *error-prone* [12], *FindSecBugs* [13] and *pmd* [14] allow us to detect possible common errors and ensure compliance with good programming practices, and we can obtain a variety of metrics with tools such as *ck* [15].
- For JavaScript, *JSPrime* [16] performs a static analysis of the code's security, while *retire.js* [17] detects the use of libraries with known vulnerabilities, and *escomplex* [18] and *yardstick* [19] allows us to obtain multiple metrics.

There are some tools that, despite being in this section (because they are not part of a platform, do not show the information clearly and simply, and do not integrate with any type of version control, integration or distribution system), are extremely interesting and complete, as is the case of *coala* [20]:

[1] https://docs.codacy.com/getting-started/supported-languages-and-tools/
[2] https://github.com/returntocorp/semgrep
[3] https://github.com/wireghoul/graudit/
[4] https://www.dwheeler.com/flawfinder/
[5] http://cppcheck.sourceforge.net/
[6] https://github.com/MetricsGrimoire/CMetrics
[7] https://github.com/dspinellis/cqmetrics
[8] https://bandit.readthedocs.io/en/latest/
[9] http://pylint.pycqa.org/
[10] https://github.com/jendrikseipp/vulture
[11] https://radon.readthedocs.io/
[12] https://errorprone.info/
[13] https://spotbugs.github.io/
[14] https://pmd.sourceforge.io/pmd-5.3.2/
[15] https://github.com/mauricioaniche/ck
[16] http://dpnishant.github.io/jsprime/
[17] http://retirejs.github.io/retire.js/
[18] https://github.com/jared-stilwell/escomplex
[19] https://github.com/calmh/yardstick
[20] https://coala.io

an open source tool that allows you to analyze and repair code from a large number of programming languages including C/C++, Python, JavaScript, CSS, and Java, to ensure that they follow different quality requirements.

Of course, in all the sections mentioned above, there are many other similar platforms and tools that we have not mentioned, as the length and time limitations of the work prevent a detailed and exhaustive analysis. However, the above tools allow us to get an idea of the state of the art of code quality analysis nowadays.

# 3 Proposed method

Our objective is to develop a parametrizable score to evaluate the quality of software projects and compare the quality resulting from the choice of one programming language or another according to our priorities. To achieve this, we select a generic set of metrics able to capture code vulnerabilities, together with some representative metrics of *complexity, documentation, maintainability* and *size*, as this information is highly related to software quality according to what we saw in Section 2.1.

With these generic metrics, we propose a parametrizable final score to get a rate from 0 to 100 for every project, based on the developer's needs. This score allows us to compare multiple programming languages according to the priorities set on the parameters.

## 3.1. Metrics

We intend to select a complete, compact, and simple set of metrics with enough evidence for the results to be rigorous. From our previous study of the state of the art, we have found three different metrics commonly used that seemed interesting: cyclomatic complexity, percentage of comments, and code duplications. Moreover, we want to innovate by using information about bugs and vulnerabilities found in the code itself. This is a pretty intuitive approach that has never been taken on the literature and can be very interesting. However, our method based on using a parametrizable score could be applied with a different set of metrics.

### 3.1.1. Error rate by severity

This metric is one of our main contributions to the state of the art of study of software quality. It is based on analyzing the code and considering the actual errors present in it, instead of estimating them using other much less specific metrics.

As we have seen in Chapter 2, there are many tools that allow us to detect errors within the code. From these errors, we will make a classification according to their severity, using an integer from 0 (least severe) to 5 (most severe). This allows us to know the number of errors a project has and how severe

they are, and how they are distributed by their severity.

Although this information is precious, it has some limitations, the most important being a problem inherent to code analysis, usually referred to as *false negatives and false positives* [12]:

- On the one hand, it is impossible to detect all the bugs contained in a code, because the tools are based on a set of rules, and if a bug is not known, there will be no rule written for it, so there will always be false negatives. We can do nothing about this issue other than updating the used tools regularly to make sure we catch as many errors as possible and adding some other metrics that work relatively well as predictors of programming errors. These metrics will be detailed in the following sections.
- On the other hand, if the static analysis is based on such rules, it is possible that the tools provide false positives: problems that the developer has considered, and in practice may not occur, even if the analyzer assumes that they do. To prevent these false positives, we allow the user to ignore specific or general errors if he thinks they are false positives. These discarded errors are not taken into account on the score explained in Section 3.2.

### 3.1.2. Cyclomatic complexity

One of the primary metrics we talked about in Section 2.1.3 is the software complexity, which is highly related to the maintainability costs of a project [13]. Also, on [14], the authors found that the complexity of a source code was a good predictor of possible vulnerabilities analyzing a complete production operation system as Windows Vista.

The main complexity metric usually used is the called **cyclomatic complexity**. It was developed in 1976 by Thomas J. McCabe, and it is a quantitative measure of the number of linearly independent paths through the source code of a program.

Through these years, much literature has been published on this metric, and some research makes it a very interesting metric to consider in our project. For example, [15] found that the cyclomatic complexity is one of the better performance predictors, and [16] states that there is an inverse relationship between cyclomatic complexity and the number of bugs in a program and programmer performance, finding that, even in UNIX, there is a correlation greater than 0.9 between the cyclomatic complexity and the number of errors.

### 3.1.3. Percentage of comments

Another of the metrics mentioned in Section 2.1.3 is the comments percentage, i.e., the percentage of lines with comments within the total number of lines of the project.
This metric is highly related to the *maintainability* of the source code; as explained code is much easier to understand and to fix, preventing misunderstanding that could lead to new errors when changes are made to the code.

### 3.1.4. Duplicated code

Finally, the last two metrics we decide to include in this work are related to the quantity of duplicated code in the project. This is, again, related to the maintainability of the project, as a duplicated code is much likely to cause problems in the future, but it also less memory efficient. It is, according to [17]: *"one of the factors that severely complicates the maintenance and evolution of large software systems"*.

If we assume that duplicated code blocks are supposed to do the same task, any simple refactoring must be duplicated too. This puts pressure on the developer, which should find every place in which to perform the refactoring. Although some automatic tools recognize these blocks as duplicates, failure to keep the code in sync would prevent them from working as expected. This means that, if in the future, an error is discovered, or a modification is required, the chances of something going wrong are really high.

Initially, our idea was to calculate the total number of duplicated lines of code and divide it by the total number of lines of code in the project so that we would have the percentage of duplicated code of the project. However, although it seemed very simple, we discovered a small subtlety that was really counter-intuitive and limiting. Suppose we have a project with three files, as in Figure 3.1. Then, the first and second files share a large block of code, in this case, represented by the orange color. This orange block could be, for example, a class, which is duplicated in the two files. On the other hand, inside the orange block, another block of code is represented by the blue color, which also appears in the third file. It could be, for example, a method. This blue block is then repeated in the three files of the project.

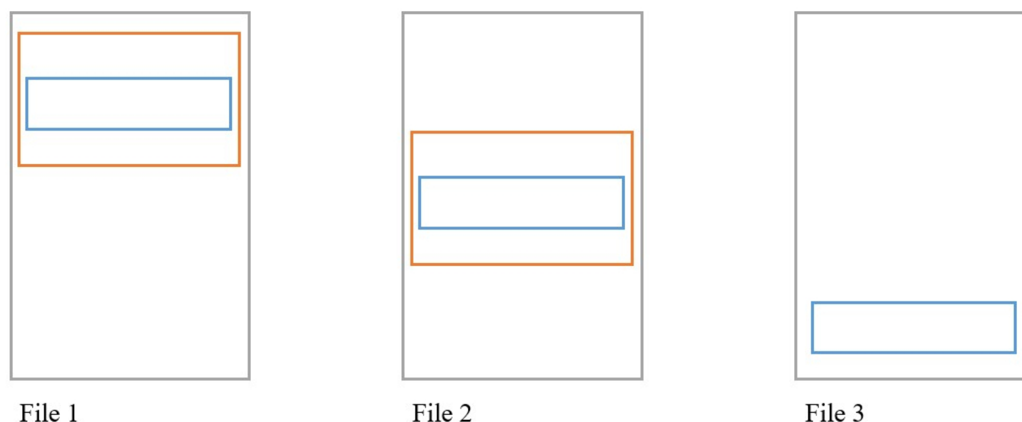



**Figure 3.1:** Example of duplications

Therefore, the number of duplicated lines would be $1 * lines(orange) + 2 * lines(blue)$. This causes the blue block to be counted several times, and, therefore, the percentage of duplicated code can exceed 100 percent.

In practice, we found some projects with percentages much greater than one, so we decided to use another approach by defining two different metrics. On the one hand, we have the number of duplica-

tions per line, which is calculated as the number of duplicated code fragments by the total number of non-blank lines of code of the project. For example, in Figure 3.1, the number of duplicated code fragments would be $3$ ($1$ orange block and $2$ blue blocks). On the other hand, we calculate the average of the hundreds of lines that each duplicated fragment has. Again, on Figure 3.1, this would be expressed as $[lines(orange)/100 + lines(blue)/100]/2$.

With these two metrics, we have information about the duplications present on the project and their size.

## 3.2. Scoring parametrization

Continuing with the development of our method, once we have selected a set of generic metrics representative of software quality, we want to generate an adjustable score from 0 to 100 configurable by the user according to the priorities of the analyzed project. This score will allow us to compare the quality provided by the different programming languages depending on the weighting used.

As we have already mentioned in Section 2.1.3, it is impossible to provide an absolute metric that assesses the quality of every project, because depending on the needs, complexity, documentation, or performance will be more or less of a priority.

Thus, to make our analysis as versatile as possible, we offer the possibility of weighting the importance given to the different proposed metrics, hence obtaining a score from 0 to 100 according to specified requirements. This will allow us to compare the various programming languages, but it could also be used to compare different projects that have the same objective in order to select one based on quality (for example, one could use this method to choose the best web framework according to his priorities).
The parametrizable weights used on the score we can be listed as follows:

- The weight of each severity, from 0 to 5, as an integer in $[0, 10]$. We use the number of errors of each severity per 10.000 lines of code. This allows the user to give more importance to errors with greater severity and ignore the errors with severity 2, for example.
- The weight of the average cyclomatic complexity, rated as an integer in $[0, 10]$.
- The weight of the number of duplicated fragments per non-blank line of code in each project, rated as an integer in $[0, 10]$.
- The weight of the average of thousands of lines per duplicated fragment of code in each project, rated as an integer in $[0, 10]$.
- The percentage of comments in each project in relation to the expected percentage. This means that the user fixes an expected percentage of comments, and we set an integer weight from 0 to 10 used with the difference between the expected and the total percentage of comments.

Now that the weights are defined, as they all go from 0 to 10, the parameters must have similar scales so that the formula makes sense and these parameters have a real influence on the score.

In general, according to the tests we have done, explained in Chapter 4, the ratio of errors per

10,000 lines of code is between 0 and 0.4, although, in exceptional examples, it can exceed 1. The cyclomatic complexity of a method is greater or equal than one and very rarely exceeds ten [18], so it is understandable to take this value divided by ten so that the scale is similar to that of the errors. The two duplication metrics are most of the time inside the interval $[0, 0{,}5]$. Finally, in the case of the percentage of comments, we understand it as a percentage expressed as a decimal number (i.e., we take 100 % as 1, not as 100).
The formula used to calculate the score can then be expressed as:

- Let $n_i,\ i \in \{0, 1, ..., 5\}$ be the number of errors of severity $i$ per 10.000 lines of code, and $w_i \in \{0, 1, ..., 10\}$ the weight assigned to that number.
- Let $c$ be the average cyclomatic complexity of the project and $w_c \in \{0, 1, ..., 10\}$ the weight assigned to that number.
- Let $nd$ be the number of duplicated code fragments of the project and $w_{nd} \in \{0, 1, ..., 10\}$ the weight assigned to that number.
- Let $ld$ be the average hundreds of lines of duplicated code fragments of the project and $w_{ld} \in \{0, 1, ..., 10\}$ the weight assigned to that number.
- Let $pc$ be the percentage of comments of the project, $epc$ the expected percentage of comments, both as real numbers in $[0, 1]$, and $w_{pc} \in \{0, 1, ..., 10\}$ the weight assigned to that number. The value $epc$ allows us to establish a reference of what the percentage of comments should be so that it is not too low nor too high.

So, the raw score, which goes from $0$ to $\infty$, can be expressed as:

$$raw_score = \frac{\left[\sum_{i=0}^{5} w_i * n_i\right] + w_c * c/10 + w_{nd} * nd + w_{ld} * ld + w_{pc} * |pc - epc|}{||w_0 + w_1 + w_2 + w_3 + w_4 + w_5 + w_c + w_{nd} + w_{ld} + w_{pc}||_2}$$

Note that, as we divide by the norm of the vector formed by all the weights, the important thing is the ratio between the weights and not their value itself. In this way, one metric can be given twice as much importance as another.

In order to turn this into an score from $0$ to $100$, where $100$ is the greatest score, let $COEFF$ be an integer value not yet determined, which we will talk about later. Thus, the score could be expressed as:

$$score = 100\left[1 - tanh(\frac{raw_score}{COEFF})\right].$$

Therefore, $COEFF$ is just a linear transformation of the input of the scoring function.

In this way, it allows us to regulate the scores to be as appropriate as possible, but it does not alter the comparisons between different projects. That is, if a project $A$ has better quality than another project $B$ for $COEFF = 5$, it will still have better quality for any other value of $COEFF$.

# 4

# Method validation

In order to validate our proposal, we have selected four programming languages to compare using five experiments, each one with a different parametrization, to look for the best language depending on the weighting given to each metric. To facilitate the collection of these metrics, we have developed a tool based on a selection of open source utilities, many of which were mentioned during the state of the art study, in Section 2.2.3.

## 4.1. Preparation of the experiments

### 4.1.1. Selection of programming languages

We want to implement some of the most popular programming languages, so, to obtain information on the most currently used languages, we turned to the data provided by the latest Stack Overflow survey [19], the default page for any developer, which in 2020 had almost 65,000 participants. From which, taking the responses of professional developers, we obtain the graph shown in Figure 4.1.

Another reference source to obtain the popularity of different programming languages is the TIOBE index, developed monthly from the popularity of programming language searches in various engines such as Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube, and Baidu ( [20]). In the March 2021 version, they offer the following ranking set out in Figure 4.2.

Based on this information and taking into account our knowledge of the different programming languages, we make our selection, in which we choose four languages: C, C++, Python, and Java. All of them are present in the Top 15 of both rankings, and most of them in quite leading positions. Moreover, they are selected to represent several paradigms: structured and strongly typed programming, such as C, strongly typed object-oriented programming (C++ or Java), and dynamically typed (Python). This allows us to apply our method to many different, very varied, and diverse projects.

However, it should be noted that the code we have developed is completely modular, and it's open source, which means that anyone could add support for the programming language or tool of their choice simply and quickly.

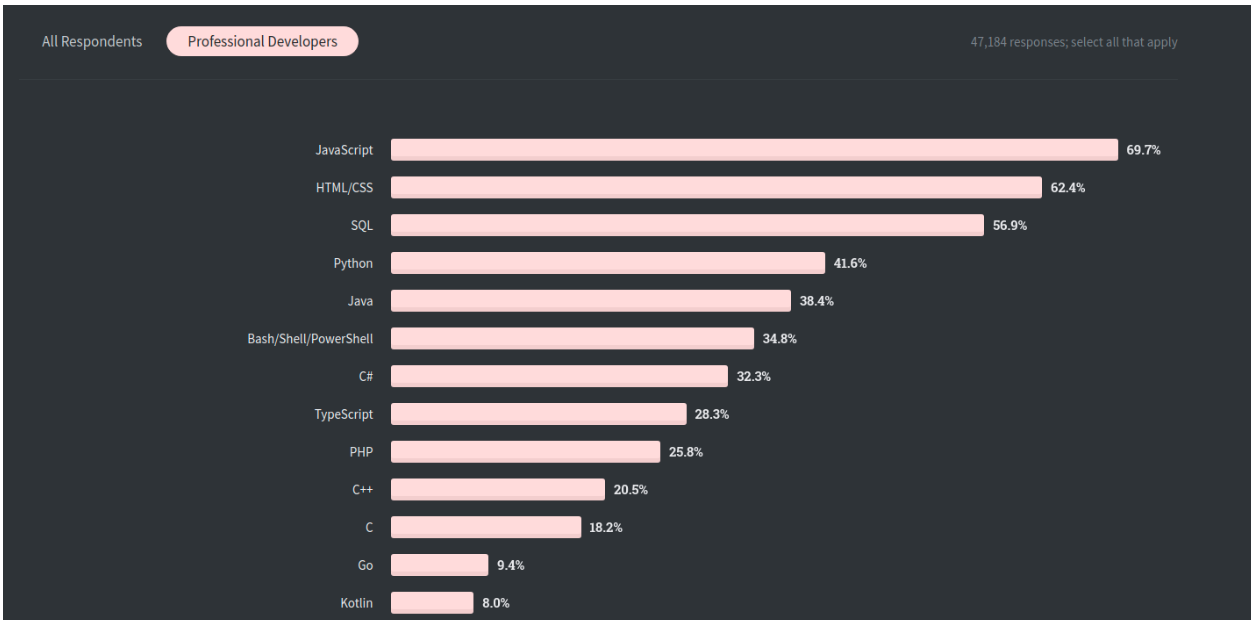



**Figure 4.1:** Stack Overflow Developer Survey 2020 - Most Popular Technologies

| Mar 2021 | Mar 2020 | Change | Programming Language | Ratings | Change |
|---|---|---|---|---|---|
| 1 | 2 | ↑ | C | 15.33% | -1.00% |
| 2 | 1 | ↓ | Java | 10.45% | -7.33% |
| 3 | 3 | | Python | 10.31% | +0.20% |
| 4 | 4 | | C++ | 6.52% | -0.27% |
| 5 | 5 | | C# | 4.97% | -0.35% |
| 6 | 6 | | Visual Basic | 4.85% | -0.40% |
| 7 | 7 | | JavaScript | 2.11% | +0.06% |
| 8 | 8 | | PHP | 2.07% | +0.05% |
| 9 | 12 | ↑ | Assembly language | 1.97% | +0.72% |
| 10 | 9 | ↓ | SQL | 1.87% | +0.03% |
| 11 | 10 | ↓ | Go | 1.31% | +0.03% |
| 12 | 18 | ⇈ | Classic Visual Basic | 1.26% | +0.49% |
| 13 | 11 | ↓ | R | 1.25% | -0.01% |
| 14 | 20 | ⇈ | Delphi/Object Pascal | 1.20% | +0.48% |

**Figure 4.2:** TIOBE Index in March, 2021

### 4.1.2. Selection of evaluation tools

The source code of a software project can be composed of hundreds of thousands of lines of code, so it is necessary to use automatic tools that allow us to collect the various metrics. Since the proposed method is somewhat tool-independent, we have chosen a set of tools according to our convenience. In this case, all the tools chosen are open source and can be freely downloaded and used. However, we consider that our proposal could be used with any combination of analysis tools the user has available.

As we mentioned earlier, the idea is to choose a set of tools that are able to detect errors and CWE vulnerabilities, together with some representative metrics of *complexity, documentation, maintainability* and *size*.

We base our selection on, among other things, the time the project has been active, the quality of the documentation, and the quality of the outputs provided. Most of these tools have already been mentioned above in Section 2.2.3, so we avoid detailing them again.

First, the tools we use to look for possible code weaknesses or security issues are as follows:

- **For C/C++** we use *flawfinder* and *cppcheck*.
- **For Python** we use *bandit* and *pylint*.
- **For Java** we use *PMD*, which analyzes Java source code directly, and *FindSecBugs*, which analyzes bytecode code, so we need to compile it before analyzing it.

Then, to look for duplications in the code in the different programming languages, we go with the Copy-Paste Detector (CPD) tool included in *PMD*, which we have already mentioned above. We use the information provided by this tool to generate the two duplication metrics explained in Section 3.1.4.

Finally, to calculate the remaining metrics, we use:

- The *radon* tool for Python code analysis, which provides a huge number of metrics including lines of code and comments, cyclomatic complexity per module and file, maintainability index, and Halstead metrics.
- For the rest of the implemented programming languages we use the *metrix++* [1] tool, which includes a smaller amount of metrics and has the disadvantage that it is necessary to use it file by file to obtain sufficiently detailed information. However, it is the only open source software project that provides metrics for C++.

With these two tools, to calculate this cyclomatic complexity per project, we calculate the average cyclomatic complexity per file and compute the average on every file, thus, getting the average cyclomatic complexity on the entire project. To calculate the percentage of comments, given the total number of lines per file and the lines of comments per file, we calculate it as

$$comments_percentage = \frac{\sum_{f \in \Delta} lines_of_comments(f)}{\sum_{f \in \Delta} lines(f)},$$

where $\Delta$ is defined as a set containing every source code file on the project. We have considered 15 %

[1] https://github.com/metrixplusplus/metrixplusplus/

as the expected percentage of comments along the different experiments, as it's not too low or too high according to our analysis.

We choose to use Python to develop the tool as it facilitates the integration of the different tools (developed in multiple programming languages) via shell calls thanks to its *subprocess* module.

### 4.1.3. Development of the evaluation platform

With the tool already developed, and after having researched the state of the art in Chapter 2, we realized that we could contribute a lot to the community by developing an open source platform. Creating a platform such as those analyzed in section 2.2.1 is tremendously complex and costly, and in no case do they mention that they make use of open source tools internally for their development. However, the tools named in Section 2.2.2 are much simpler, as they focus only on static analysis of the code by searching for security flaws and calculating different metrics. Furthermore, although some of them are promoted as open source, none of them is fully compilable or executable from public data; there are simply some available fragments of their code that in no case become usable.

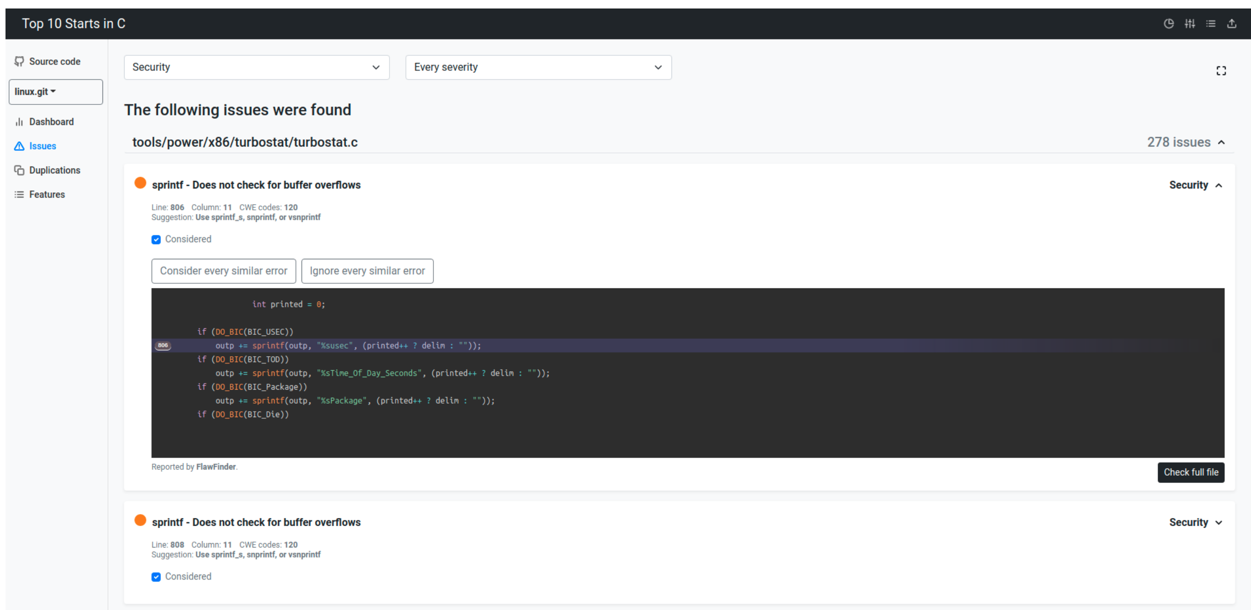



**Figure 4.3:** Screenshot of the platform's *Errors* page. Here the user can see the errors a source code has, filter them by category and severity, and even check the full file's source code. In this case, we present the results of analyzing *Git*'s source code.

We decide then to develop a platform of this style by expanding the command line tool we had developed, and we focus on the following points as our proposal of value:

- **Analysis and classification of design and safety problems:** Given code in one or multiple programming languages, it will be analyzed to list and classify its errors, depending on the severity of the problem and the category to which it belongs.
- **Code duplication analysis:** Given a code, we will detect and report duplications in that code, which may cause

problems in the future.

- **Obtaining a parametrizable metric:** By analyzing the code, we will generate a set of metrics that will allow the user to understand the main weaknesses or strengths of the code. This user will be able to weigh each metric according to the objective of the analyzed code, obtaining an overall project score according to their needs.
- **Comparison of multiple projects:** The tool will also allow comparisons between different projects by using the same parameters' settings in all of them. This can allow the user to choose one of several alternatives to be used in the project, depending on its quality, which will improve the quality of the final project.

When it comes to representing the data generated and stored in the database, among the many possibilities available, we choose to use an HTML server that allows us to generate pages dynamically. Thus, the platform can be used from any device without installing any program or extension.

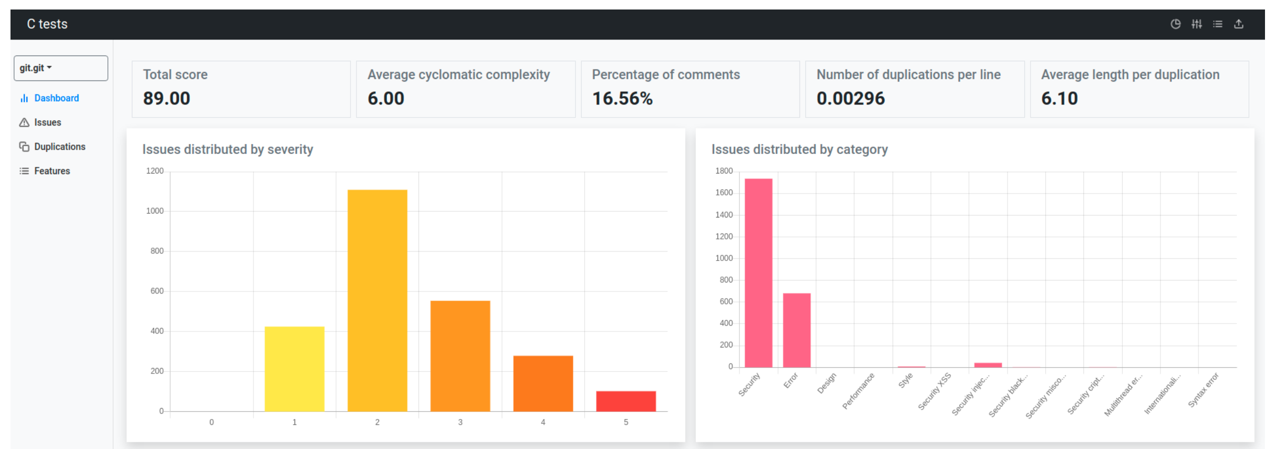



**Figure 4.4:** Screenshot of the platform's *Dashboard* page. Here, the user can get some general information about the analysis performed on a project's source code. In this case, we present the results of analyzing *Git*'s source code.

Thanks to using Python as the programming language, the development was straightforward, given our previous experience. On the one hand, it natively allows shell calls, which facilitates communication with the selected tools. On the other hand, thanks to the sqlalchemy [2] library, we can connect to SQL databases, such as PostgreSQL [3], which has been chosen in this situation to store the results of the analysis. Finally, the Flask [4] framework, with which we already have much experience, allows high-speed and efficient development of a simple server, which also generates HTML pages dynamically thanks to the jinja2 [5] library.

We should also mention the use of the Bootstrap v5.0 [6] framework to facilitate the design of the different web pages that make up the platform's interface, together with three other libraries:

[2] https://www.sqlalchemy.org
[3] https://www.postgresql.org
[4] https://flask.palletsprojects.com
[5] https://jinja.palletsprojects.com/
[6] https://getbootstrap.com

- prism.js [7], which allows us to display code of multiple programming languages within the platform, in order to visualize the context of errors, for example.
- Dropzone.js [8], which allows us to implement the typical *drag and drop* rectangle to upload files to the platform.
- charts.js [9], which allows us to display some of the generated data on animated charts, so that the user can have a better and easier understanding of the provided information.

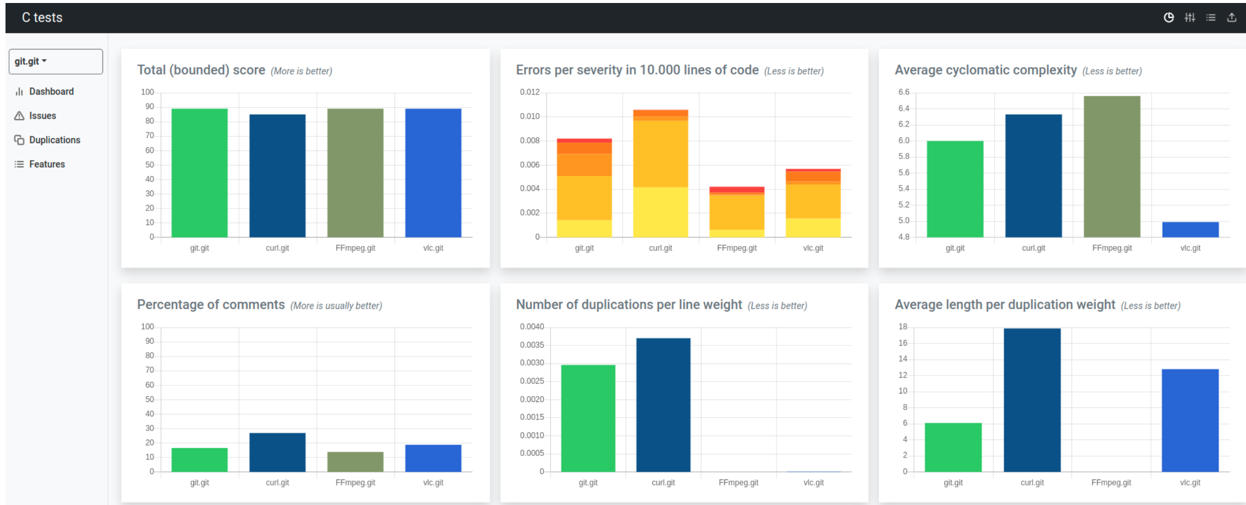



**Figure 4.5:** Screenshot of the platform's *Group dashboard* page. Here, the user can compare the analysis of a project group. In this case, we compare the results of analyzing the source codes of *Git*, *Curl*, *FFMpeg* and *VLC*.

We decide to include some other tools to look for SQL errors. In this case, we use *SQLint* [10] for generic SQL source code, and *PMD* again, which in this case is used to analyze PostgresSQL code, allowing us to analyze routines or functions more complex than simple queries. This new database management language allows our platform to be even more complete, being able to analyze projects holistically.

Apart from the metrics provided, we are also interested in obtaining some information about the operation of the code. That is, we want to be able to know what operations the code is performing: whether it works with cryptography, with input-output, with the network, and so on. In this way, the user can make sure that the code does not have undue functionalities that could jeopardize his project. For this purpose, we make use of a cross-platform open source tool developed by Microsoft called *Application Inspector* [11]. Since it is open source, we can easily modify it to integrate it into our platform fully.

It is essential to mention that there are other non-static analysis tools we could have used on this

[7] `https://prismjs.com`
[8] `https://www.dropzonejs.com`
[9] `https://www.chartjs.org`
[10] `https://github.com/purcell/sqlint`
[11] `https://github.com/microsoft/ApplicationInspector`

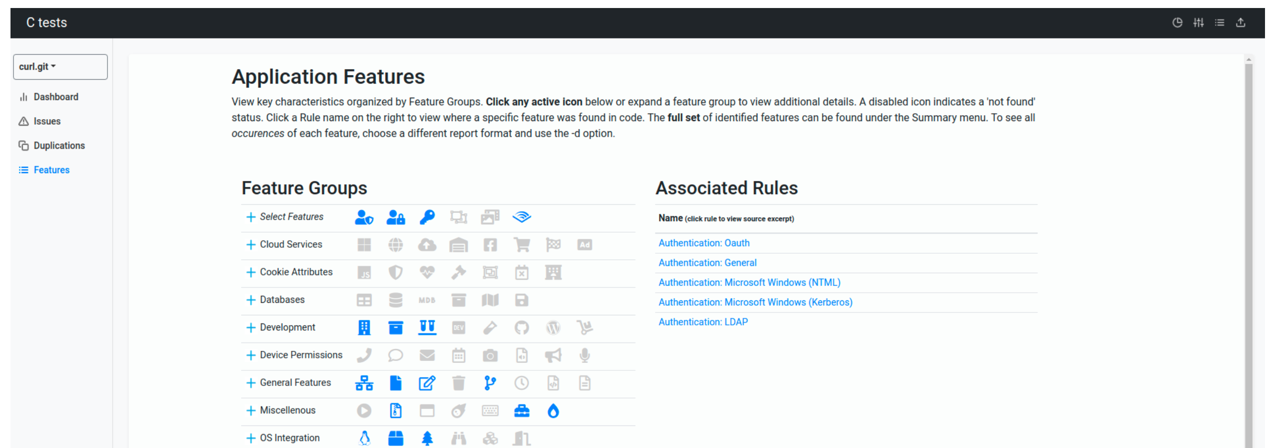



**Figure 4.6:** Screenshot of the platform's *Features* page. Here, the platform displays the information extracted with Microsoft Application Inspector. In this case, we present the results of analyzing *Curl*'s source code.

project to make the platform event more complete and valuable. However, after considering the pros and cons, we decided not to take the risk that would involve the execution of unknown code on our platform. This decision makes the system more robust and less susceptible to possible attacks.

### 4.1.4. Database description

As for the database, we decide to use production code of some of the most popular open source projects available by analyzing the Top 50 open source projects for C, C++, Java, and Python, so that we can draw some conclusions about the programming languages and how they compare with each other, or what is most valued by developers in each programming language.

We then turn to GitHub, the most extensive version control platform based on Git, which hosts more than 200 million repositories, and has more than 65 million developers (according to its official website as of June 2021), to get a list of the most popular open source projects among the community.
We take as reference the Github-Ranking repository, which daily updates the ranking of each language ordering the repositories according to the number of *stars* and *forks* in GitHub.
Note, however, that we discard repositories based on tutorials or listings of other libraries, as they are not really repositories of code to use, and those with less than 50 % of code of the language to which they belong. Thus, the top 10 projects for each language, as of June 8, 2021, are listed in Table 4.1.

Note that our approach when comparing quality achieved with different programming languages, based on the real errors and metrics of the source code, is fairly innovative. As we mentioned in 2.1.3, usually the quality is compared in two ways:

- On some studies like [7], [10] the quality of different programming languages is compared by selecting some open source projects of each language and looking for commits which names are related to bugs and errors, or by looking at bug databases [21].

| Top | C | C++ | Java | Python |
| --- | --- | --- | --- | --- |
| 1 | linux | tensorflow | spring-boot | youtube-dl |
| 2 | netdata | electron | elasticsearch | models |
| 3 | scrcpy | terminal | mall | flask |
| 4 | redis | swift | RxJava | keras |
| 5 | git | opencv | guava | ansible |
| 6 | php-src | bitcoin | MPAndroidChart | transformers |
| 7 | wrk | pytorch | glide | scikit-learn |
| 8 | obs-studio | tesseract | lottie-android | core |
| 9 | ijkplayer | godot | zxing | scrapy |
| 10 | FFmpeg | x64dbg | netty | you-get |

**Table 4.1:** Top 10 projects for each language on GitHub, sorted by Stars and Forks.

- Other studies as [8], [9] simply use some metrics to determine the software quality, without taking into account the real errors the software has.

In our case, as we have looked for actual bugs within the code, calculated different metrics, and, finally, given an overall score to each project based on various parameters, we have a much more practical, realistic, and customized approach depending on the needs that each project or developer may have.

## 4.2. Experiments

Our experiments set is made of different combinations of parameters to check how they influence the final score for each language. We have also decided to add a new intermediate metric called *weighted severity*, which is the fraction of the raw score contributed by the severity. It can be expressed as:

$$weighted\ severity = \frac{\left[\sum_{i=0}^{5} w_i * n_i\right]}{||w_0 + w_1 + w_2 + w_3 + w_4 + w_5 + w_c + w_{nd} + w_{ld} + w_{pc}||_2}.$$

The main objective of this metric is to let the reader know how the error importance varies depending on the parametrization.

### 4.2.1. Experiment I: Error weights relatives to their severity

A first fairly intuitive approximation is to consider the weights of the errors as directly proportional to their severity. Thus, the mistakes that we regard as more serious are more detrimental to the final grade; they have more weight. We then take the values represented in Table 4.2, using the notation

explained in Section 3.2.

| $epc$ | $w_c$ | $w_{pc}$ | $w_{nd}$ | $w_{ld}$ | $w_0$ | $w_1$ | $w_2$ | $w_3$ | $w_4$ | $w_5$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 2 | 2 | 2 | 2 | 0 | 2 | 4 | 6 | 8 | 10 |

**Table 4.2:** Weights used in Section 4.2.1

In this scenario, we prioritize the errors when considering the quality, giving relatively low importance to the cyclomatic complexity, the percentage of comments, and the duplications-related metrics.

We also consider in this particular experiment a small variation of the weights, now given in Table 4.3, to show how these subtle variations do not influence the final result.

| $epc$ | $w_c$ | $w_{pc}$ | $w_{nd}$ | $w_{ld}$ | $w_0$ | $w_1$ | $w_2$ | $w_3$ | $w_4$ | $w_5$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 2.4 | 1.6 | 2 | 1.8 | 0 | 2.3 | 3.8 | 5.6 | 8.5 | 9.4 |

**Table 4.3:** Weights used in Section 4.2.1 with slight variations.

### 4.2.2. Experiment II: Error weights relatives to their severity (II)

A small variation on the previous approach could be to give more importance to comments, duplications and cyclomatic complexity. In this case we would be considering that the maintainability of the code has much more importance than in the previous section.
We then take the parameters reflected in Table 4.4.

| $epc$ | $w_c$ | $w_{pc}$ | $w_{nd}$ | $w_{ld}$ | $w_0$ | $w_1$ | $w_2$ | $w_3$ | $w_4$ | $w_5$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 8 | 8 | 8 | 8 | 0 | 2 | 4 | 6 | 8 | 10 |

**Table 4.4:** Weights used in Section 4.2.2

### 4.2.3. Experiment III: Focus on errors

We can consider, in the opposite way to the previous section, the case in which we only care about errors, without taking into account duplications, complexity or comments. Here we would be ignoring all the maintainability and performance of the code, as we have already explained in sections 3.1.2, 3.1.3 and 3.1.4.

### 4.2.4. Experiment IV: Ignore errors

One more possibility is to ignore the data provided by the classification of the errors by severity, and take into account only the metrics. In this case, the used weights could be the ones represented on

| $epc$ | $w_c$ | $w_{pc}$ | $w_{nd}$ | $w_{ld}$ | $w_0$ | $w_1$ | $w_2$ | $w_3$ | $w_4$ | $w_5$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 0 | 0 | 0 | 0 | 0 | 2 | 4 | 6 | 8 | 10 |

**Table 4.5:** Weights used in Section 4.2.3

Table 4.6.

| $epc$ | $w_c$ | $w_{pc}$ | $w_{nd}$ | $w_{ld}$ | $w_0$ | $w_1$ | $w_2$ | $w_3$ | $w_4$ | $w_5$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 5 | 5 | 5 | 5 | 0 | 0 | 0 | 0 | 0 | 0 |

**Table 4.6:** Weights used in Section 4.2.4

### 4.2.5. Experiment V: Uniform weights

Finally, we can give every weight the same value so that the importance of each metric is the same. These weights are the one in Table 4.7.

| $epc$ | $w_c$ | $w_{pc}$ | $w_{nd}$ | $w_{ld}$ | $w_0$ | $w_1$ | $w_2$ | $w_3$ | $w_4$ | $w_5$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |

**Table 4.7:** Weights used in Section 4.2.5

### 4.2.6. Some more collected data

Some studies, as [22], find no correlation between software quality and the number of programmers in a project. On the other hand, in the literature, it is common to use the number of stars and forks of a GitHub project as the ground truth of the quality of that project, as done in [8] and [9].

To corroborate or disprove these claims, we have collected information provided by GitHub, intrinsic to the projects, as the number of stars, forks, collaborators, and the age of the project in days. Then, we have looked for possible correlations between the metrics obtained by our tool and these metrics provided by GitHub.

## 4.3. Results

We present the results of the experiments divided into two sections. In Section 4.3.1 we show the individual metrics, which do not depend on the parametrization chosen in a specific experiment. In Section 4.3.2 we present the results of each experiment with two different plots. On the left, we show the weighted severity (explained in Section 4.2), where less is better, while on the right, we present the

score obtained by each language, where more is better. In all cases, the graphs represent the mean and standard deviation.

### 4.3.1. Parametrization-independent results

First of all, we can look at the individual metrics obtained, which do not depend on the parametrization given for the final score. In Fig. 4.7 the mean cyclomatic complexity and percentage of comments are displayed, along with their standard deviation.

In Figure 4.8 the two metrics related to code duplications are displayed. We can see the average number of duplications per line of code on the left, while on the right, the average hundreds of lines per duplication is shown.

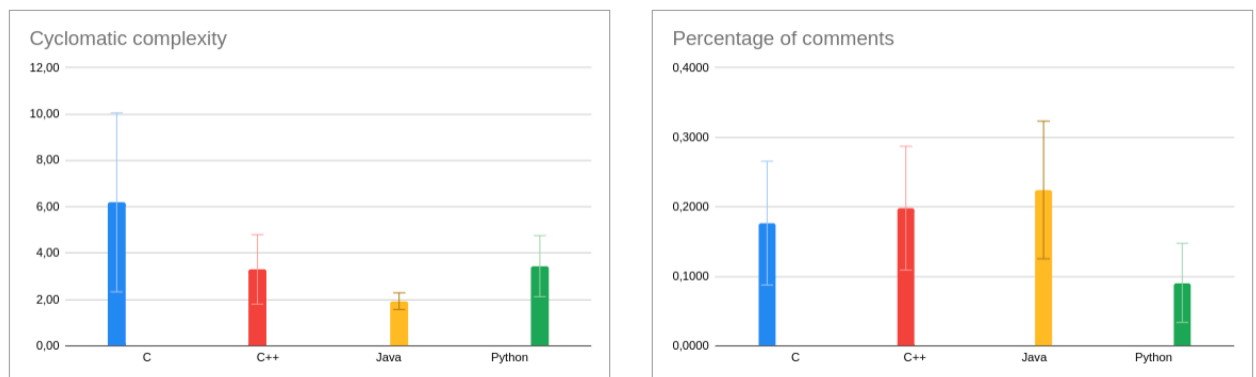



**Figure 4.7:** Cyclomatic complexity and percentage of comments on each language. Represented as the mean and standard deviation.

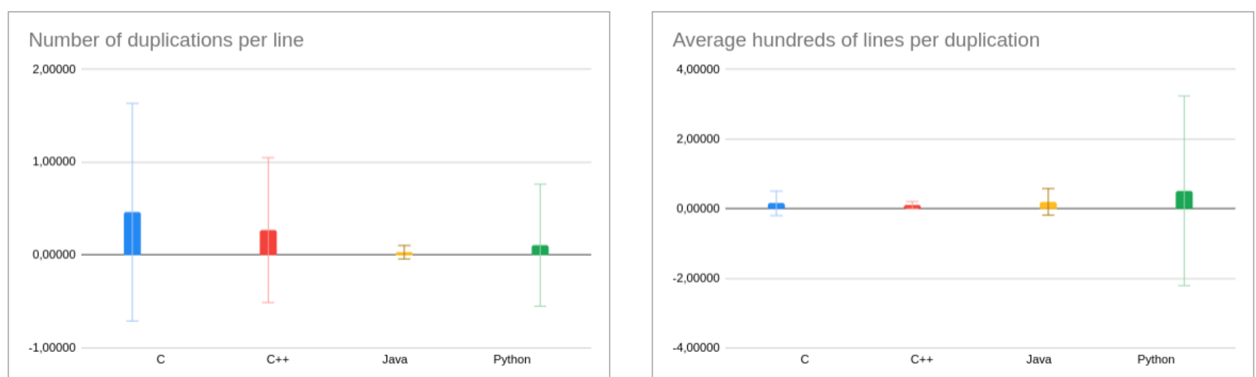
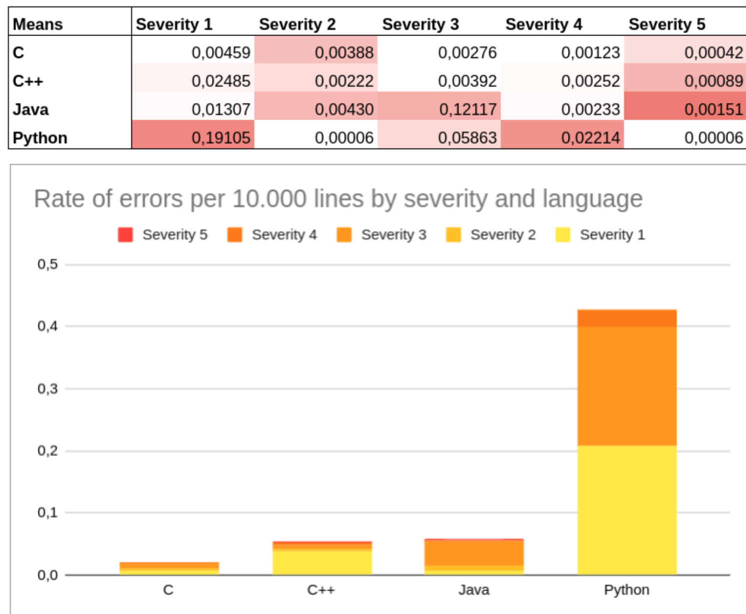



**Figure 4.8:** Average number of duplications and number of hundreds of lines per duplication per language on GitHub ranking.

In addition, we can also present the results of the errors by severity and by language, again, without taking into account any parametrization yet. Figure 4.9 shows the raw data represented in the table on the top, along with the graphical representation for straightforward interpretation. It is important to

note that most Python errors with severity 1 are due to the PEP 8 standard, which considers an error something as simple as using tabs instead of spaces.



| Means | Severity 1 | Severity 2 | Severity 3 | Severity 4 | Severity 5 |
|---|---|---|---|---|---|
| C | 0,00459 | 0,00388 | 0,00276 | 0,00123 | 0,00042 |
| C++ | 0,02485 | 0,00222 | 0,00392 | 0,00252 | 0,00089 |
| Java | 0,01307 | 0,00430 | 0,12117 | 0,00233 | 0,00151 |
| Python | 0,19105 | 0,00006 | 0,05863 | 0,02214 | 0,00006 |



**Figure 4.9:** Mean of errors by severity on each language.

## 4.3.2. Parametrization-dependent results

Now that the simple metrics are presented, we can check how the final score varies depending on different parameter settings, using $COEFF = 1{,}5$ on the scoring function explained at 3.2. Note that, although the score varies a lot between our different experiments, that does not have any importance, and could be compensated simply by changing $COEFF$, which does not affect the order of highest scores. This is something we already explained in Section 3.2.

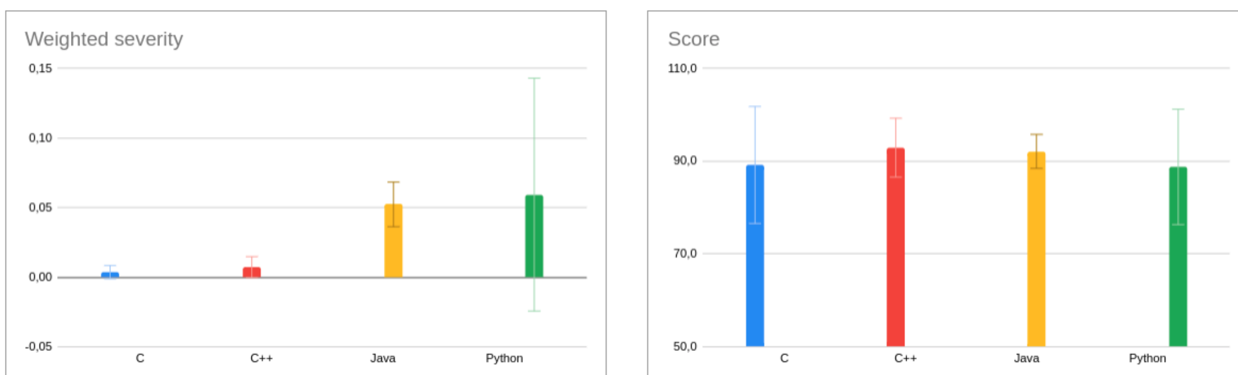



**Figure 4.10:** Weighted severity and score by language, represented as the mean and standard deviation. Weights as expressed in Section 4.2.1

Figure 4.10 shows the mean and standard deviation of the weighted severity and final score using the weights reflected in Table 4.2. We can see how in this case, with the weights of the errors as directly proportional to their severity and the rest of the weights with less importance, C++ is the language with the highest quality. Python, on the contrary, is the language with the worst score while also being the one with fewer comments. If we take the minor variations proposed in Table Table 4.3 the results are pretty much the same, with some variations on the scores but maintaining the same ranking.

We then take the parameters proposed in Section 4.2.2, which give more importance to comments, duplications, and cyclomatic complexity, and represent the results in Figure 4.11, where we can see how, in this case, Java is the language with the highest score, followed by C++.

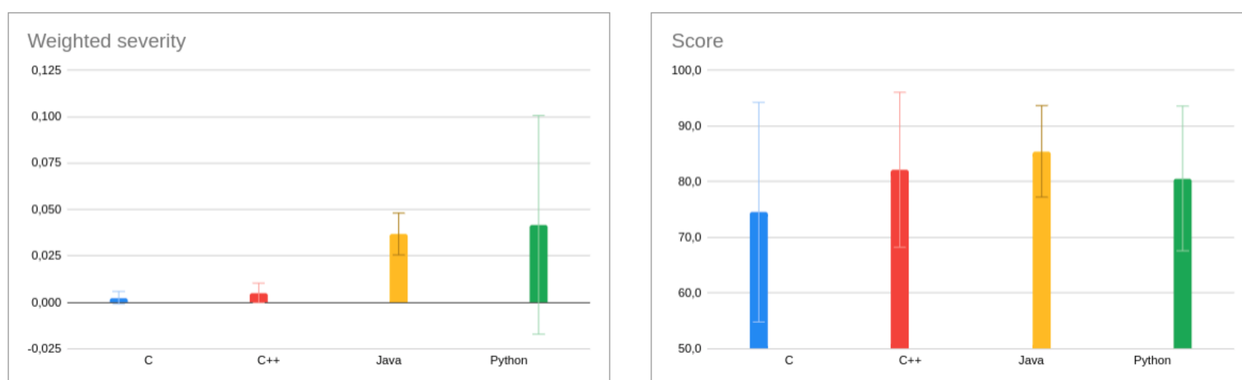



**Figure 4.11:** Weighted severity and score by language, represented as the mean and standard deviation. Weights as expressed on Section 4.2.2

Taking the parameters shown in Table 4.5, the results can be represented in Figure 4.12, where we can see how C becomes the programming language with the highest average score if we ignore the cyclomatic complexity, duplications and comments.

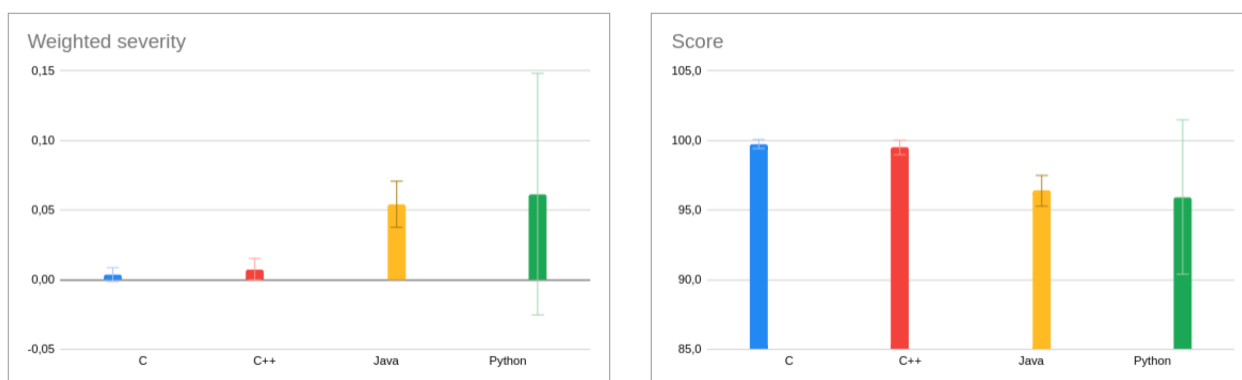



**Figure 4.12:** Weighted severity and score by language, represented as the mean and standard deviation. Weights as expressed on Section 4.2.3

Figure 4.14 shows the results of completely ignoring the errors, as expressed in Section 4.2.4, while Figure 4.13 shows the result of choosing a uniform combination of parameters, with Java being again

the language with the highest score, followed by C++, and C having the lowest one.

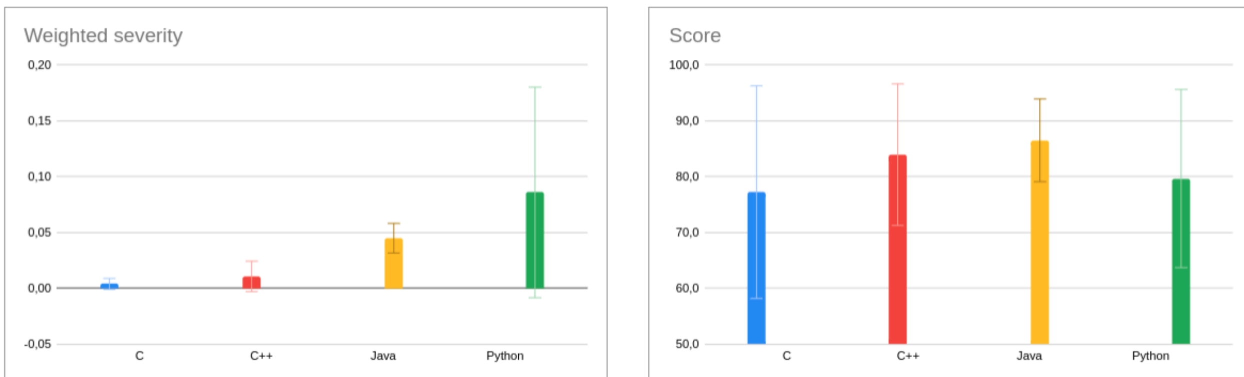



**Figure 4.13:** Weighted severity and score by language, represented as the mean and standard deviation. Weights as expressed on Section 4.2.5

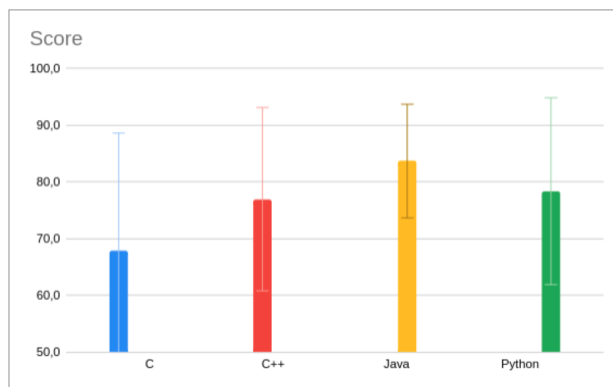



**Figure 4.14:** Weighted severity and score by language, represented as the mean and standard deviation. Weights as expressed on Section 4.2.4

Finally, the correlations between metrics and GitHub attributes are reflected in Figure 4.15. Note that, in this case, we are using the weights proposed in Section 4.2.1. However, the results are pretty much the same with any other combination of parameters. In particular, as parametrizations do not affect individual metrics, the only two columns which vary when changing the parametrizations are the *Weighted severity* and *Score* columns.

| Correlations | Cyclomatic complexity | Comments | Duplic. number | Duplic. len. avg | Severity 1 | Severity 2 | Severity 3 | Severity 4 | Severity 5 | Weighted severity | Score |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Stars | -0,10 | -0,11 | -0,07 | -0,01 | 0,15 | -0,09 | 0,02 | 0,13 | -0,08 | 0,09 | 0,09 |
| Forks | -0,11 | 0,00 | 0,04 | -0,05 | 0,13 | -0,04 | 0,03 | 0,05 | -0,02 | 0,07 | 0,03 |
| Colaborators | 0,00 | -0,05 | 0,08 | -0,01 | 0,02 | -0,11 | -0,07 | 0,01 | -0,10 | -0,06 | -0,04 |
| Age (days) | 0,08 | -0,03 | 0,08 | -0,08 | -0,17 | -0,06 | -0,05 | -0,09 | -0,01 | -0,12 | -0,02 |

**Figure 4.15:** Matrix of correlations between the calculated metrics and some attributes intrinsic to GitHub Projects.

# 5

# Discussion

From the previous results, it might not be easy to determine a programming language proclaimed as a clear winner in terms of quality. In this chapter we will discuss under which conditions it would be advisable to use each language to maximize software quality.

## 5.1. Isolated metrics analysis

Fig. 4.7 represents the mean cyclomatic complexity and percentage of comments, along with their standard deviation. We can see how the low cyclomatic complexity of Java code stands out, probably due to a large number of getters and setters, which have a cyclomatic complexity of 1. On the other hand, the cyclomatic complexity of C is considerably higher, probably because it is the oldest and most rudimentary language. As C requires memory management by the programmer, the complexity increases with the conditionals used to check whether the memory was allocated or not. Furthermore, C does not have so many native libraries or functions to simplify operations, so even checking if a string contains another substring requires a for loop, while in the other languages can be checked with a simple method.

As for the percentage of comments, it is reflected how Python code, with the lowest percentage, is self-explanatory, and probably the use of *Javadoc*-formatted comments, which is very widespread, is what raises the percentage of comments in Java.

Now, if we take a look at the rate of errors per severity, at Figure 4.9, it is clear how C has the lowest rate of errors. This could be explained due to the fact that C is an older and more complex language than C++, Java, or Python. It is not as attractive to novice developers, and programmers who choose to use it are usually experienced in the programming field, with years of experience, and, therefore, less likely to make mistakes. In contrast, languages such as Python and Java are used by many junior developers who may simply have taken a concise programming course to understand the basics. That could be the reason why Python has the most significant rate of errors. However, it is important to note how Python has the lowest rate of errors with a severity of 5, which are the most dangerous.

In Figure 4.8 the two metrics related to code duplications are displayed. We can see the average

number of duplications per line of code on the left, while on the right, the average hundreds of lines per duplication is shown. We can see how C has the biggest number of duplications per line, probably due to being the only one of the four programming languages that do not support Object-Oriented Programming and, therefore, inheritance. This could cause some functions to be duplicated with minor modifications. At the other extreme, Java has the lowest number of duplications per line, which goes in line with the fact that it is the only language from the selected ones that does not support structured programming. Python stands out with the highest average hundreds of lines per duplication, but we have not found any particular reason that could explain it.

## 5.2. Multi-parametric analysis

With the individual metrics analyzed, we can consider the experiments' results as a function of the parameters. Hence, we have that the preference for one programming language or another is given according to the ratio of priorities we give to maintainability and performance and to the error rate.

Thus, we first consider the two extreme cases analyzed. In the first one, represented in Section 4.2.3, we completely ignore the metrics and attend only to the errors. Moreover, we consider the importance of these errors as directly proportional to their severity. Therefore, more severe errors are more detrimental to the score. In this case, the language with the best score is C, followed closely by C++, which makes sense as they are the languages with the lowest rates of errors. If we take an opposite approach, considering only the metrics related to maintainability and performance, as in section 4.2.4, we obtain Java as the highest-scoring language, followed by Python. This shows how the quality score is completely dependent on the parametrization used, as the preferred language is different, and even the programming paradigm varies from a structured programming language like C to an Object-Oriented language like Java.

In the other three experiments considered, we tried different, more moderate approaches, giving different priorities to the metrics. Thus, we see in sections 4.2.5 and 4.2.2 how, whether we give equal priority to all metrics and errors, or prioritize maintainability and performance, Java is the language with the highest quality, followed by C++. Conversely, if we prioritize a lower rate of errors, as in Section 4.2.1, C++ is the clear winner.

In summary, according to our results, the best languages are Java or C++, while the language with the lowest quality in average is Python. In the case of Java, in average, it has the highest quality, and it should be the primary option whenever code maintainability and performance has to be prioritized. In the case of C++, we have seen how it is always above C, or practically at the same level, so it seems advisable to use C++ instead of C, Python, or Java whenever having as few errors as possible is a priority, without wanting to sacrifice too much code maintainability. This is in line with the findings of P. Bhattacharya et al. in [21], where they found that using C++ instead of C increased software quality and

maintainability.

In general, our findings are also consistent with the ones on [7], where they state *“disallowing type confusion is modestly better than allowing it, [...], static typing is also somewhat better than dynamic typing.”*, which goes in line with Java and C++ being the preferred languages in terms of quality. This validates our methodology as an interesting way to determine and compare the software quality of projects according to the requirements.

It is also important to mention that, as we saw with the results when using the parameters in Table 4.3, small variations in the parameters do not imply variations in the ranking of programming languages. This is due to the normalization being used on the scoring parametrization formula, explained in Section 3.2, and it is what allows us to express the weights as integers from 0 to 10.

## 5.3. Correlations analysis

Finally, when analyzing the correlations between metrics and GitHub projects attributes, represented in Figure 4.15, we found no significant correlations between the elements. In particular, we can note there seems to be no correlation between the number of collaborators and the source’s quality, something already seen on the literature [22]. Furthermore, there seems to be no correlation between the quality score and the number of stars or forks of the projects. This goes against the premises of some papers like [8] and [9], which consider the numbers of stars and forks as the ground truth for the quality perceived by developers. Therefore, either the quality perceived by developers is not related to code quality, or the number of stars and forks cannot be used as the ground truth.

## 5.4. Threats to validity

We recognize few threats to our reported results, mainly because we are using open source tools, which are not as complete as we would like, and, therefore, they have limited a lot the metrics we can work with to achieve a score. We can classify these limitations into two groups.

### 5.4.1. Threats to the errors found by the tools

There are some limitations imposed by the tools used to detect errors, vulnerabilities, and code weaknesses, mainly because these tools are different depending on the programming language. This implies that the percentage of bugs discovered in relation to the bugs in the code itself may vary between tools, which, in turn, may disfavor the scores of some of the programming languages. It is also important to consider the false positives and false negatives the tools may provide, which could make the error severity metrics less reliable.

In addition, when using different analyzers, the classification of issues and vulnerabilities into categories could be susceptible to errors. Since the categories provided by the analyzers are usually very diverse, and, although we have tried to group them as coherently as possible, errors could arise in the classification.

Finally, it is worth noting that if the used tools were better designed, all supporting the CWE nomenclature for errors, we could obtain more information about the quality of the projects.
As we mentioned in Section 2.1.2, the CISQ relates each of its categories to different errors listed as CWE weaknesses. If every tool used to detect errors could classify the found errors within these CWE weaknesses, we could assign a score to each of the categories mentioned by the CISQ from these errors. However, this is only supported by two of the tools used, *FlawFinder* and *CppCheck*, which are both for C and C++.

Nevertheless, none of these limitations affects the comparison of the evolution of the same project or the comparison of multiple projects of the same programming language, since they will be subject to the same analyzers. Therefore, the information provided by the tool will be consistent. In other words, the used methodology is perfectly valid for two primary purposes. On the one hand, it can reflect how the quality of a project evolves, thus, helping developers take the necessary measures to improve the product. On the other hand, the methodology would do a perfect job assisting the developers to choose one library or framework from a set of alternatives, all written on the same programming language, to assure that the dependencies of their software have as much quality as possible, avoiding possible errors.

### 5.4.2. Threats due to the set of metrics provided by the tools

The generic set of metrics chosen also imposes limitations, and, although our method is usable with any other group of metrics, we consider it essential to mention these limitations, as they may influence our results.
When it comes to obtaining metrics, by using two different tools and having to rely on the metrics available in both tools, we are quite limited. This is mainly due to the absence of open source tools that allow us to calculate complex metrics for C++.

As we saw earlier in Section 2.2.3, for Python we have *radon*, for Java *ck*, and for C *CMetrics* and *cqmetrics*. All of these tools offer quite complex and complete metrics, although some are relatively difficult to use. However, for C++, no tool provides sufficient metrics. Thus, we have had to resort to *Metrix++*, which covers C, C++, and Java, at the cost, however, of providing a very limited number of metrics.

It would have been interesting, for example, to be able to take into account Halstead's E metric, which is the best predictor of performance [15], or perhaps the maintainability index (which has propo-

nents and opponents). Other metrics that would have been useful are those related with *coupling* and *degree of inheritance* which, as we explained in Section 2.1.1, are related with the categories proposed on the ISO/IEC 25000 standard.

# 6 Conclusions

We have proposed an innovative method to evaluate the quality of a software project based on the weights given to different metrics. In this way, the score given to each project depends on the priorities involved in its development. In our case, we have applied a generic set of metrics that allows an approach to the concept of software quality based on the definitions given by the ISO/IEC 25000 and the CISQ standards. However, one of the main advantages of our approach is that the set of metrics chosen can vary, and the proposed scoring parametrization would still be valid.

Our methodology provides a considerable innovation regarding the approaches commonly used in the literature, advocating the analysis of GitHub commits or of some metrics obtained from the code. In our case, we use a holistic approach taking into account the metrics and the actual bugs, vulnerabilities, and code smells found in the code.

To validate this methodology, we have chosen four programming languages (C, C++, Java, and Python) and the 50 most popular open source projects available on GitHub for each language. With this dataset, we have performed five different experiments to check how the weighting of the various metrics affected the overall quality of software projects depending on the programming language. Our experiments reveal how the quality is highly dependent on the given parametrization, and result in two main languages to consider: Java and C++. Java seems to be more recommended when code maintainability is the main priority. However, when the main focus is to have a lower number of errors, at the expense of lower maintainability, C and C++ prevail, although the use of C++ seems to be recommended as it has slightly higher maintainability. In addition, the result of Python, which does not appear as a winner in any of the tests, stands out. This shows how, despite being a trendy programming language, it does not favor software quality in any case.

Throughout the work, we have encountered several difficulties when comparing metrics. On the one hand, to use the proposed scoring formula, it is necessary to consider the scales of each metric so that none has, by default, more importance than the rest. On the other hand, when comparing projects with different programming languages, the tools for extracting metrics from the code may be biased and either disadvantage or favor a specific programming language.

In addition, in the case of code duplications, we have noticed the problem of trying to consider

a percentage of duplicated code, as is usually done with the percentage of comments. To solve this unintuitive fact, we propose two metrics that describe the number and length of code duplications.

To conclude, our method improves state of the art in several key points: it considers software quality as something related to its different properties, providing a method to obtain an overall score according to a project's priorities. Furthermore, within the selection of metrics, we analyze the actual errors that a code has. With all this, we compare some of the most popular programming languages, checking how the given parametrization considerably affects the result of the language with the best quality.

## 6.1. Future work

Of course, there is still much work to be done to confirm the hypotheses put forward in our experiments definitively. For example, it would be desirable to implement a more extensive set of programming languages and a higher number of projects per language and repeat our experiments, which should be relatively simple thanks to our tool being open source.

In addition, the calculation of a larger number of metrics, such as coupling and degree of inheritance, would allow us to determine the quality more objectively and obtain much more information about the different programming languages. It would also be great to modify each of the used open source tools to make the analysis and classification more consistent between tools.

The are also lots of improvements that could be made to the platform itself. Regarding the platform's efficiency, we currently implement parallel processing of multiple projects through threads, something that could be improved by using different processes and a queue to manage all requests to avoid the server getting saturated when analyzing too many projects simultaneously.

Although we try to optimize storage by deleting code coming from GitHub so that we can access it later if necessary (for example, when the user requests to view an entire file from the platform) using the GitHub API, there are also possible improvements to be made at the storage performance level. Currently, we are employing a single SQL database, which is not the most convenient for storing all the duplicate code. A mixed approach, using a NoSQL database to keep the duplications, would be much more efficient in terms of performance.

Finally, we could also improve the platform's functionality by implementing some kind of synchronization with GitHub projects to allow the recurrent and automatic analysis, therefore, keeping track of the evolution of the quality of a specific project.

# Bibliography

[1] B. Oliver and R. G. Dromey, “Safe: a programming language for software quality,” in *Software Quality and Productivity*, pp. 227–230, Springer, 1995.

[2] B. Kitchenham and S. L. Pfleeger, “Software quality: The elusive target,” *IEEE Softw.*, vol. 13, p. 12–21, Jan. 1996.

[3] Y. Kanellopoulos, A. Panos, A. Dimitris, C. Makris, E. Theodoridis, C. Tjortjis, and N. Tsirakis, “Code quality evaluation methodology using the iso/iec 9126 standard,” *International Journal of Software Engineering & Applications*, vol. 1, 07 2010.

[4] I. Heitlager, T. Kuipers, and J. Visser, “A practical model for measuring maintainability,” pp. 30 – 39, 10 2007.

[5] CISQ, “Cisq supplements iso/iec 25000 series with automated quality characteristic measures.” `https://www.it-cisq.org/cisq-supplements-isoiec-25000-series-with-automated-quality-characteristi`
Accessed: 2021-02-23.

[6] CISQ, “Code quality and related standards.” `https://www.it-cisq.org/standards/`. Accessed: 2021-02-23.

[7] B. Ray, D. Posnett, V. Filkov, and 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, (New York, NY, USA), p. 155–165, Association for Computing Machinery, 2014.

[8] M. Papamichail, T. Diamantopoulos, and A. Symeonidis, “User-perceived source code quality estimation based on static analysis metrics,” pp. 100–107, 08 2016.

[9] V. Dimaridou, A.-C. Kyprianidis, M. Papamichail, T. Diamantopoulos, and A. Symeonidis, “Towards modeling the user-perceived quality of source code using static analysis metrics,” pp. 73–84, 07 2017.

[10] P. S. Kochhar, D. Wijedasa, and D. Lo, “A large scale study of multiple programming languages and code quality,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 563–573, 2016.

[11] Gartner, “Gartner magic quadrant for application security testing,” 04 2020.

[12] B. Chess and G. McGraw, “Static analysis for security,” *IEEE Security Privacy*, vol. 2, no. 6, pp. 76–79, 2004.

[13] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig, “Software complexity and maintenance costs,” *Commun. ACM*, vol. 36, p. 81–94, Nov. 1993.

[14] T. Zimmermann, N. Nagappan, and L. Williams, “Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista,” pp. 421–428, 01 2010.

[15] B. Curtis, S. B. Sheppard, and P. Milliman, “Third time charm: Stronger prediction of programmer performance by software complexity metrics,” in *Proceedings of the 4th International Conference on Software Engineering*, ICSE ’79, p. 356–360, IEEE Press, 1979.

[16] T. J. McCabe and C. W. Butler, “Design complexity measurement and testing,” *Commun. ACM*, vol. 32, p. 1415–1425, Dec. 1989.

[17] S. Ducasse, M. Rieger, and S. Demeyer, “A language independent approach for detecting duplicated code,” in *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM’99). ’Software Maintenance for Business Change’ (Cat. No.99CB36360)*, pp. 109–118, 1999.

[18] R. Vasa and J.-G. Schneider, “Evolution of cyclomatic complexity in object oriented software,” 01 2003.

[19] S. Overflow, “Stack overflow 2020 developer survey.” `https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages`. Accessed: 2021-03-13.

[20] TIOBE, “Tiobe index for march 2021.” `https://www.tiobe.com/tiobe-index/`. Accessed: 2021-03-13.

[21] P. Bhattacharya and I. Neamtiu, “Assessing programming language impact on development and maintenance: a study on c and c++,” in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 171–180, 2011.

[22] E. Weyuker, T. Ostrand, and R. Bell, “Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models,” *Empirical Software Engineering*, vol. 13, pp. 539–559, 10 2008.

# Acronyms

**CI/CD** Continuous Integration/Continuous Deployment.

**CISQ** Consortium for IT Software Quality.

**CPD** Copy-Paste Detector.

**CWE** Common Weakness Enumeration.

**IEC** International Electrotechnical Commission.

**IEEE** Institute of Electrical and Electronics Engineers.

**ISO** International Organization for Standardization.

**OWASP** Open Web Application Security Project.

UAM
UNIVERSIDAD AUTONOMA
DE MADRID