# POLITECNICO DI TORINO

Master's Degree Course in Computer Engineering


Politecnico
di Torino

Master's Degree Thesis

# Research, implementation and analysis of source code metrics in Rust-Code-Analysis

**Supervisors**

**Prof. Luca ARDITO**

**Prof. Maurizio MORISIO**

**Dr. Michele VALSESIA**

**Candidate**

**Marco BALLARIO**

**October 2022**

## Abstract

***Context*:** *Software metrics* are procedures to measure software processes, products, and resources. *Source code metrics* are a subset of software metrics to measure source code properties. Using these measures, developers can have a picture of the status of a codebase, identify potentially problematic parts in advance and improve the code during its evolution. *Static code analysers* are automated tools developed to analyse source code without executing it. These tools are particularly adapted to implement source code metrics and aim to be quick to give immediate feedback. *Rust-Code-Analysis* (*RCA*) is an open source static code analyser which computes source code metrics from an intermediate representation of a source code file: the *Abstract Syntax Tree* (*AST*). The tool can parse many different programming languages and generate metrics at various levels of granularity by dividing the source code into blocks called *Spaces*. RCA is a Rust library developed by Mozilla to support the Firefox browser development process. Anyone can contribute to its development and propose changes to its code.

***Objectives*:** This thesis has two main objectives. The first one consists of expanding Rust-Code-Analysis by adding support for new source code metrics. The library already supports eleven metrics for ten different programming languages. The second objective, instead, consists of studying the values produced by these new metrics for various repositories and then evaluating their results. In particular, we are interested in the significance of the measures, their relationships with the size of a codebase, and their possible thresholds.

***Method*:** We have started by selecting a set of metrics to implement in Rust-Code-Analysis from scientific articles and existing static analysis tools. The chosen metrics focus on three different aspects: the size of the codebase, object-oriented programming paradigm, and security. In parallel, we have introduced new features to the tool, adding some integration tests and the average of specific metrics to learn how the library works. Starting from the acquired knowledge, we have implemented the selected metrics for the Java language, making the necessary modifications to Rust-Code-Analysis. Each implementation includes an exhaustive set of unit tests and has been subjected to an accurate and severe review workflow before being accepted into the official repository. Using these metrics, we have measured several popular Java projects of various sizes and versions, reporting the obtained results in graphs. The results analysis has been conducted in two parts. In the first part, we have focused on the measures obtained from repositories of various sizes, pinning them to a specific version. For the second part, we have analysed the metrics values for several versions of the same codebase. By inspecting the

measures and graphs, we have examined and discussed the behaviour of each metric from distinct perspectives.

***Conclusions*:** The work produced in this thesis consisted of the implementation of six new metrics for the Java language within Rust-code-analysis. Other contributors could also implement these new metrics for additional object-oriented languages since the project is open source. At last, we have shown the behaviour of these metrics in actively maintained repositories, specifying which could be their potential uses.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

With the increasing amount of software produced daily, source code quality is becoming more crucial than ever. We can measure software quality using *software metrics*, which are standard methods defined to measure attributes of software products, processes, and resources. *Source code metrics* are a particular category of software metrics which extract some information from a source code. From that information, then, these metrics can measure source code properties. We can use source code metrics to know the maintainability status of a project, identify potential problems in advance, and improve the code during its development. Static code analysers are automated tools developed to analyse a source code without executing it. These tools are particularly adapted to implement source code metrics and aim to be quick to give immediate feedback. *Rust-Code-Analysis* (*RCA*) is a Rust library developed by Mozilla. The library is a static code analyser which can extract several source code metrics from various programming languages. RCA is an open source project born with the aim of supporting the Firefox browser development process. Anyone can contribute to its development and propose changes to its code.

In this thesis, we have performed three main consecutive tasks:

1. We have researched the state-of-the-art source code metrics, studied their implementation feasibility in RCA, and selected six metrics.

2. We have learned how RCA works by extending its functionalities. With the acquired knowledge, we have implemented the chosen metrics for the Java language.

3. We have analysed the implemented metrics by applying them to working codebases and inspecting the obtained measures represented using different graphs and tables.

We have divided the thesis into the following chapters:

1. **Software metrics**: In this chapter, we analyse the state of the art of source code metrics and describe our research process to select the six metrics to implement in RCA.

2. **Rust-Code-Analysis**: In this chapter, we describe how the RCA tool works, and we explain the few initial improvements we have applied to the library.

3. **Metrics implementation**: In this chapter, we illustrate the process we have followed to implement the six metrics in the library. We also describe in detail the solutions we have thought of to cover exceptional cases and overcome the main obstacles in our way.

4. **Metrics analysis**: In this chapter, we present and discuss the measures obtained by applying the implemented metrics to some working codebases.

5. **Conclusions**: In this final chapter, we summarise the most important results obtained and describe potential future opportunities and extensions.

# Chapter 2

# Software metrics

## 2.1 Definitions

*Software metrics*[1] are standard procedures defined to measure some attributes of software entities. They are often confused with *measures*, despite the two being distinct concepts. Metrics are functions or methods; they define a way to obtain numerical measures from a software characteristic. Measures are the result of the application of metrics to actual cases. Despite the difference, the two terms are synonyms in software engineering. Measuring is an essential part of every field of research, especially in computer science. New metrics and measurement approaches are continuously proposed and studied by researchers. The goal is to find objective procedures that anyone can reproduce to get quantifiable results. Metrics applications are numerous and more relevant because of the enormous amount of source code produced daily. The applications of software metrics are many. Software companies can use them to plan project costs, debug code, optimise application performance and assure the quality of the final product.

*Software entities* are the subject of the measures, and they can be classified into three categories: processes, products and resources. Processes are all the activities related to software production, like the design of the application, its actual development and testing. Products are the artefacts, documents and deliverables resulting from a process. Finally, resources are entities required by a process. A process can use both products and resources to complete, but it will eventually produce just resources. For example, a design process of a program may generate a UML diagram. A developing process can later require the generated UML diagram to deliver a working code.

*Software attributes* are instead the entities properties that are the measurements object. Attributes are measured using metrics definitions and can be classified into two types: internal and external. Internal attributes are properties one can measure, inspecting a single process, product or resource without considering the behaviour of the entity. External attributes can instead be measured only by taking into account how the entity behaves and relates to its environment. The size of a source code file is an example of an internal attribute because it can be measured statically by counting the lines of code. Instead, the number of failing tests is an external attribute that one can measure only by executing the code.

*Source code metrics* are a particular subset of software metrics that measure internal source code attributes. This kind of software metric is directly extracted from the source code without the need to execute it. By only relying on the source code, source code metrics can extract information about the source code like size, complexity, maintainability, quality and security.

Source code metrics history can be divided into two main ages[2]. The first age comprehends all the metrics defined before 1991. Most of the metrics defined in this period are based on code complexity. The second age instead started in 1992. The metrics defined in this period are mostly based on the object-oriented paradigm.

The first metrics designed in the age of code complexity metrics were the lines of code (LOC) and thousands of lines of code (KLOC), defined in the seventies to measure software size. Later in that decade, complexity metrics like McCabe's Cyclomatic Complexity and the Halstead metrics started becoming very popular in the software engineering field.

The most know metrics introduced in the age of object-oriented metrics were the CK metrics, defined by Chidamber and Kemerer. The CK suite was the most adopted object-oriented suite of metrics because of its exhaustive empirical evaluation. Later more metrics suites were defined to cover more object-oriented aspects not covered by the CK metrics. Examples include the Lorenz and Kidd set of metrics and the MOOD suite.

As the source code evolves in time, new approaches to source code metrics are continuously proposed. New directions are indicated, for example, by component-oriented and aspect-based metrics[3]. *Components* are reusable modules that can be combined to produce a working application, according to the component-oriented programming paradigm. *Aspects* are a way to define a common cross-cutting concerns behaviour across different software modules. The aspect-oriented programming paradigm helps reduce boilerplate code while favouriting modularization.

*Static code analysers* are automated tools developed to analyse a source code without executing it. These tools are particularly adapted to implement source code metrics and aim to be quick to give immediate feedback. Static code analysers are also used for reverse engineering and to find bugs and lints in source code. Lints are pieces of suspicious code that can contain a bug, programming errors or be stylistically wrong. Given the versatility and velocity of the instrument, a static code analyser is often integrated into projects by using continuous integration systems. A continuous integration system can execute a static code analyser over a codebase to compute source code metrics after some changes are applied to the repository. This way, developers have a constant way to monitor the quality status of a project over time.

*Static code analysis*[4] origins date back when computers did not even exist, and humans were the ones performing algorithms and computations. By defining a set of instructions to use in algorithms then, scientists could start conceptually reasoning about source code. In particular, Alan Turing proved in 1936 how hard it is to determine the result of program execution based only on its source code. He did that by analysing the *halting problem*, which is the problem of determining if a program will finish running or not, given a set of inputs. Turing demonstrated that it is impossible to find a general solution for the halting problem which works for all the possible inputs. It was the beginning of static code analysis, and it happened even before the invention of the transistors.

Given the difficulty of evaluating the overall formal correctness of a program, the first generation of static code analysers in the late seventies focuses on finding errors and bugs in a source code. It is in this period that *Lint* was born. Lint is a program to flag common mistakes in a C source code and provide programmers with warnings similar to the ones produced today by modern compilers. At the time, many programmers were using the C programming language, and Lint helped them to find errors before compiling the source code.

With time, Lint grew, and it became supporting many more languages. Despite its excellent design, however, the tool also produced a lot of false positives, which required significant time to review, so developers started looking for new solutions. In the late nineties and early years of the new century, the birth of *Coverity* marked the second generation of static source code analysers. The tool introduced new path analysis techniques to reason deeper about the runtime behaviour of a program while separating the analysis engine from the known issues database. However, the solutions still reported many false positives in some cases and were poorly scalable in others.

Later in the year two thousand, a new semantical approach was applied to overcome these issues. The third generation of static code analysers makes use of the *Abstract Syntax Tree*, which is an abstract representation of a source code structure. This representation removes many language-specific details, captures the core logic of a source code and allows a more efficient analysis. With abstract syntax trees, code starts to be treated as data from which one can extract useful information, find defects or even propose automatic refactors.

New approaches to static code analysis are continuously studied by scientists. The goal is to develop an application able to recognise if a program is correct or not and to identify all its defects. Doing this could automate software correctness, which would have a massive impact on many software engineering fields.

## 2.2 Research process

In order to find valuable and popular software metrics, several scientific articles and static code analysers have been inspected. Since the sources of information chosen are publications and tools, the research has been divided into two phases. For the first phase, the *academic review*, information about the most valuable code metrics in the scientific field has been collected from scientific articles. For the second phase, the *market review*, the documentation of a few selected static code analysers has been inspected to understand what are the most common metrics produced daily for various projects worldwide.

### 2.2.1 Academic overview

A systematic literal review[5] performed to find the most popular maintainability source code metrics has been used as a starting point for the academic part of the research. The study already provides a list of static maintainability source code metrics most cited in recent articles. The review was not the only source consulted, though. The research for scientific material continued with the help of search engines like Google and Google Scholar, provided with the necessary keywords. The academic review result is a set of metrics which have been grouped into three categories and explained below.

*Size metrics* are software metrics defined to measure the size of a project or a source code file. The size of a codebase is interesting for developers because it has many applications. Size measures can be used to estimate the quantity of work done, the efficiency of a team and the costs of a project. However, one can define the size of a repository in different ways.

The most known size metric is the *Lines Of Code* (*LOC*), obtainable counting lines, statements or instructions in source code files. However, LOC is not always reliable since programmers can write the same operations in various ways and produce diverse LOC values. Software size can also be measured from a functional point of view, even without having the source code available. *Function Points* (*FP*) are a unit of measure of the functionality of an application. One can compute the FP value for a software product by evaluating its input, output and stored data and analysing its internal and external behaviour. Many variants of the basic FP have been standardised and are used by various commercial products.

*Complexity metrics* are a category of software metrics defined to measure the complexity of a source code. A high complexity usually highlights problems in the code. Programmers should keep the complexity of the source code they write low. A project with low complexity is also more maintainable and understandable.

The *Halstead suite* is a set of metrics computed by counting the distinct and total number of operators and operands defined inside a source code. The suite includes values of the length of the program and a difficulty measure which can be considered a complexity value because it represents the difficulty of writing and understanding the program. *McCabe's Cyclomatic Complexity* (*CC*) is another complexity metric that determines the intricacy of a program by counting the independent execution paths in a source code.

*Object-oriented metrics*[6] are a type of software metric that focuses on measuring object-oriented design properties from the source code. To be effectively applied object-oriented programming requires the developer to follow specific design principles when writing its code. Object-oriented metrics can measure how much a source code follows the object-oriented design best practices. They also provide a quantitative value to allow confrontation between different designs. Programmers can use them to improve source code and design quality[7]. Specifically, object-oriented metrics can be used to evaluate five different quality attributes: efficiency, complexity, understandability, reusability and testability or maintainability.

Most object-oriented metrics focus on measuring classes and object properties and their relationships and interactions with other classes and objects in an application. According to the systematical literal review mentioned above, the most cited object-oriented metrics are the ones defined by Chidamber and Kemerer, also called the *C&K suite*. The six metrics are:

- *Weighted Methods per Class* (*WMC*): It is the sum of the Cyclomatic Complexities of the methods defined in a class. It is a measure of the complexity of a class.

- *Response For a Class* (*RFC*): It is the number of distinct methods and constructors invoked by a class. It is another complexity measure.

- *Lack of Cohesion Of Methods* (*LCOM*): It is a measure of the cohesion of a class. Cohesion measures how the elements of a class are good together. Low cohesion may indicate that a class should be divided into more classes.

- *Coupling Between Object Classes* (*CBO*): It represents the number of classes coupled to a given class. Coupling measures how much interaction a class has with other classes. High coupling prevents reusability.

- *Depth of Inheritance Tree* (*DIT*): It measures the maximum length between a node and its root node in a class hierarchy. It is another measure of class complexity.

- *Number Of Children* (*NOC*): It is the number of classes that directly extend a specific class. It is a measure of the influence of a class on the application design.

Object-oriented metrics, however, are not limited to just these six metrics. Many more minor metrics have been defined to measure almost any aspect of the object-oriented design programming paradigm.

### 2.2.2 Market overview

The documentation of several static analysers has been consulted to discover the types of software metrics commonly reported by commercial tools. The chosen tools are a selection of both open source and proprietary products of various scales:

- *SonarQube*[8] is a large open-source project developed by SonarSource. It supports many languages and can be integrated into continuous integration systems.

- *NDepend*[9] is a large proprietary tool for the .NET environment with many advanced features, including dependency virtualization and many different metrics calculation.

- *Visual Studio*[10] is a proprietary IDE developed by Microsoft which contains modules for code metrics computation.

- *JaSoMe*[11] is an open-source, object-oriented metrics analyser for Java code. It works even if the code is not compilable.

- *CK*[12] is another open-source tool for object-oriented metrics for Java code. It can produce a large set of metrics, including the CK suite.

The documentation of the selected tools reported some metrics not found during the academic overview. All the new metrics discovered have been grouped into three categories and discussed below. Some advanced tools use the metrics computed to allow the definition of quality gates for the software product to the end user.

*Security metrics* are a kind of software metric defined to evaluate the level of security of an application. Most of them rely on the concept of *bugs* or *issues* found in the code. Bugs or issues are recognised in the code using a database of known security vulnerabilities. An example of this kind of database is the *Common Weakness Enumeration*[13] (*CWE*) project, a catalogue of hardware and software common weaknesses. Advanced tools can also grade the security of an application based on the security metrics computed.

*Duplication metrics* are software metrics used to identify how much of a source code is replicated identically in other parts. *Duplication* makes a codebase challenging to maintain and should be reduced to minimal levels. Metrics like the *number of duplicated lines or blocks* help programmers to identify maintainability threats.

*Test metrics* provide information about the tests defined for an application. Developers use tests to ensure their program works properly in a simplified environment. One can test each software module singularly, with unit tests, or the whole system with integration tests. Proper tests should cover most of the cases handled by a source code and terminate successfully. *Code coverage* and the *number of completed tests* are examples of test metrics.

## 2.3 Feasibility analysis

All the software metrics found during the academic and market overview need to be carefully reviewed to evaluate the potential feasibility of a static code analyser like Rust-Code-Analysis (RCA). Many of the metrics discovered are not implementable in RCA because of the tool's characteristics, which will be explained in detail in the next chapter.

*Dynamic metrics* are a type of software metrics only obtainable by executing the source code. Unlike static metrics, which are the result of a static analysis of a source code, dynamic metrics require the source code to compile to measure its properties. They are the result of a dynamic analysis performed during the execution of a program. RCA is a static analyser and cannot support dynamic metrics. For this reason, many discovered metrics have been discarded.

In particular, most object-oriented metrics are computed from Java source code using special dynamic techniques and libraries. Some programs compute dynamic metrics using information stored in the *bytecode*, an intermediate representation of the code generated by the Java compiler. Other tools, instead, exploit the Java language's *reflective* characteristics, allowing a program to inspect itself during the execution. Unfortunately, these methods are too language-specific and complicated to be applied to RCA.

The metrics selected for the implementation into RCA have been chosen from the remaining static source code metrics and are six in total: one size metric, three object-oriented metrics and two security metrics. This choice has been made to implement both types of metrics already present and experiment with new ones.

### 2.3.1 Assignments, Branches and Conditions (ABC)

The *ABC metric*[14] is a size metric proposed by Jerry Fitzpatrick in 1997. The author proposed the ABC as an alternative code metric to the Halstead metrics and the LOC. At the time, the Halstead metrics were not much adopted, and the LOC was not adequately defined. The author proposed the ABC to overcome the disadvantages of both metrics and provide a new way to count the operation performed by a source code.

The article explains that there are only three fundamental operations in imperative languages: *storage*, *test* and *branching*. Therefore, a metric size should count at least all those fundamental operations. The metric defines an ABC value as a vector with three components:

- Assignments
- Branches
- Conditions

Each vector component identifies the number of fundamental operations in the code. *Assignments* are transfers of data into a variable, *Branches* are explicit forward program branches out of scope, and *Conditions* are logical or boolean tests.

One can write the ABC values as an ordered triplet of integers. The article defines a standard notation, where the metric expresses its components in the form <*Assignment, Branches, Conditions*>. The ABC value can also be represented as a scalar value to simplify code comparisons. Assuming that the vector components are orthogonal and have comparable scales, one can obtain the ABC scalar value by finding the magnitude of the ABC vector with the following formula:

$$|ABC| = sqrt((A * A) + (B * B) + (C * C))$$

Where $A$ are the assignments, $B$ the branches and $C$ the conditions. The author states that further research may show that the scales of the vector components are not comparable, and a weighted sum or other calculation could provide better results. However, he also recommends using the magnitude calculation if no other methods are available. One should also report the magnitude rounded to the nearest tenth with only one decimal digit and always accompanied by the distinct ABC components.

The metric author defines the rules for counting the ABC components for C, C++ and Java languages. This way, one can compare the ABC metric for code written in different programming languages. Below the counting rules for the Java language are explained. The counting rules for the other languages are similar, with minor differences.

1. Add one to the assignment count for each occurrence of an assignment operator, excluding constant declarations:
   =, *=, /=, %=, +=, −=, <<=, >>=, &=, |=, ^=, >>>=

2. Add one to the assignment count for each occurrence of an increment or decrement operation (prefix or postfix):
   ++, −−

3. Add one to the branch count for each function or class method call.

4. Add one to the branch count for each occurrence of the *new* operator.

5. Add one to the condition count for each use of a conditional operator:
   ==, !=, <=, >=, <, >

6. Add one to the condition count for each use of the following keywords:
   *else*, *case*, *default*, *try*, *catch*, *?*

7. Add one to the condition count for each unary conditional expression.

### 2.3.2 Weighted Methods per Class (WMC)

The *Weighted Methods per Class* or *Weighted Method Count* (*WMC*) is an object-oriented metric introduced by Chidamber and Kemerer in 1994. WMC is part of a suite of the C&K object-oriented metrics suite already discussed above. The metric defines a complexity measure for object-oriented code. One can compute WMC from McCabe's Cyclomatic Complexities of the methods defined in the class using the following formula:

$$WMC = \sum_{i=1}^{n} CC_i$$

Alternatively, one can use other complexities metrics instead of McCabe's Cyclomatic Complexity. The complexity of the methods and the number of methods defined in a class indicate how much time and effort is necessary to design, develop and maintain a class. Moreover, classes with many complex methods have limited reusability. Classes with low WMC values are more maintainable and reusable.

### 2.3.3 Number of Public Methods (NPM)

The *Number of Public Methods*[15] (*NPM*) is an object-oriented metric which counts the number of public methods defined inside a class. The metric consists of a single counter.

NPM may be an indicator of two different types of problems. Firstly, a high NPM value means that a class can perform many operations and has many responsibilities. In this sense, NPM may be considered a complexity measure. When encountering a high NPM value, one should also check the WMC of the class to know if the class is genuinely complex. An example of this scenario is a utility class which defines many complex methods and makes them available publicly in a module.

Secondly, a high NPM may indicate that a class is too much coupled with other parts of the project. Any public method may expose the outer classes and modules the classes that internally uses. Too much coupling makes an application complex to maintain and prevents the reusability of its modules.

### 2.3.4 Number of Public Attributes (NPA)

The *Number of Public Attributes* (*NPA*) metric is an object-oriented metric which consists of a counter of the public attributes defined in a class.

NPA provides information similar to the one described for the NPM. The metric measures the data a class exposes to the outer classes and modules. A high NPM value may indicate high complexity and coupling for a class. Generally, it is good to have a low value of NPM for a class. This way, a class can keep all of its internal data safe from any more direct and less secure access. One can also use NPM and NPA to compute more complex derivative metrics.

### 2.3.5 Classified Operation Accessibility (COA)

*UMLsec* is an extension of the Unified Modelling Language UML, which allows the integration of security information in a UML diagram. This instrument is used when designing software applications which need to control security characteristics like confidentiality, access control and information flow.

A *classified attribute* is a class property which is defined as secrecy in the UMLsec. A *classified method* is a class method which interacts with at least one classified attribute. These two definitions have been used to define two object-oriented security metrics[16]. The two metrics aim to discover security issues at an early stage and allow software designers to compare the security of different designs.

*Classified Operation Accessibility* (*COA*) is defined as the ratio of the number of classified public methods to the number of classified methods in a class. The metric is computed by dividing the number of classified public methods defined in a class (CPM) by the total number of classified methods declared in that class (CM).

$$COA = \frac{CPM}{CM}$$

COA indicates the potential attack surface size for a given class. The attack surface size is defined by all the distinct ways a malicious actor could access secret information. The objective of COA is to protect the methods that interact directly with classified attributes. By keeping the value of the low metric, one can also reduce the information flow of secret data among the classes of the application.

### 2.3.6 Classified Class Data Accessibility (CCDA)

*Classified Class Data Accessibility* (*CCDA*) is defined as the ratio of the number of classified class public attributes to the number of classified attributes in a class. CCDA is computed by dividing the number of public classified class attributes (CCPA) by the total number of classified attributes declared in the class (CA).

$$CCDA = \frac{CCPA}{CA}$$

The metric shows how much of a class secret data are exposed from outside. The higher the value, the higher the chances a malicious party has of getting confidential information. Reducing the value of this metric contributes to reducing the size of the attack surface.

# Chapter 3

# Rust-Code-Analysis

## 3.1 Rust language

*Rust*[17] is a modern programming language focused on performance and safety, even in a concurrent environment. It was first designed in 2006 by Graydon Hoare, a Mozilla Research employee, as a personal project. Mozilla started sponsoring the project in 2009 and officially announced it in 2010. Since then, Rust has grown fast thanks to its features, gaining much attention worldwide. Today, even major software engineering companies are choosing Rust for their projects. The Mozilla Foundation, in particular, is trying to adopt it for more and more of its projects. Some components of Firefox, for example, are written in Rust.

Some of the most notable language features are:

- **Performance**: The language is fast and memory-efficient, and it has no garbage collector. Rust is as performant and portable as C++ and carefully enforces memory security. Many language features are optimized and removed during the compilation and do not introduce any runtime cost. Programmers can use the language to write embedded systems software, kernels, servers or even browsers.
- **Reliability**: Rust is a strongly typed language. We can omit variable types in declarations because they will be inferred automatically during the compilation phase. This feature and many others help programmers to reduce errors when writing Rust code.
- **Productivity**: The language supports the productivity of programmers with an intelligent compiler, an integrated package manager and build tool, a tool to auto-format the code, and much more. All these instruments help programmers to write and build their code quickly. The language also comes

with extensive documentation and many examples, which are helpful both for beginners and advanced users.

The Rust language has been voted the *"most loved programming language"* by Stack Overflow's users for six years until today. To understand the reasons that make this language so loved and popular, we need to take a closer look at some of its main features.

Rust is a static-typed language and allows the definition of optional types. Due to these and many other features, the language can recognise many errors at compile time. The Rust compiler is responsible for performing checks on the code and producing error messages. However, those error messages are curated and clear to help a programmer to understand the causes of an error.

Rust does not have a garbage collector and allows the passage of variables to functions by value and reference. The language efficiently and securely manages memory and references by introducing concepts like *ownership* and *lifetime*. Ownership is a set of rules defined by the language to manage the memory of a program. Essentially each variable has one and only one owner. An owner is responsible for a variable memory and can be, for example, a function or a struct. The program drops the variable from memory when the owner goes out of scope. Most of the time, we do not want to own a variable, but borrow its value. Through *borrowing*, we can obtain a reference to a variable and work with it outside the owner by passing it as a parameter to a function, for example. Lifetimes are, instead, generic constructs introduced to ensure that all borrow operations are valid, especially in ambiguous scenarios.

A programmer can extend Rust functionalities and break the rules of the language. The language allows the violation of the memory safety guarantees because it might be necessary in some particular cases, for example, when writing an operating system. However, for common scenarios, this is often not advised. The programmer will be held responsible for the unsafe code introduced and its effects on the memory. One can introduce unsafe code using the *unsafe* keyword.

Rust implements some object-oriented design abstractions in its way, promoting composition over inheritance. Using *traits*, we can define a particular behaviour for a type and share it across multiple types, adapting it to the different necessities. The trait concept is similar to the interface one. One can also use traits to specify what a generic type can be by defining all the behaviours it must have.

### Example

For example, suppose we have a simple Rust source file called *main.rs*. This file contains a single function named *main*. The *main* function does not accept any parameters and it does not return any value. The only objective of the function is to print out the well-known string *"Hello World!"* and it achieves that by calling the *println!* macro. Once we print out the sentence, the function terminates, and the program ends. This example is very similar to the default code generated by Cargo when we create a new Rust program using the *cargo new* command.

```
1 fn main() {
2     println!("Hello World!");
3 }
```

*Macros* are a Rust feature used for *metaprogramming*, a technique that allows the code to write other code. They are more flexible and more powerful than ordinary functions. For example, macros, as opposed to functions, can take a variable number of parameters. However, macros code is often more complex to understand than functions code.

In the example, we end the macro invocation with a semicolon. It may seem insignificant, but placing a semicolon at the end of an instruction has a specific meaning in a Rust program. If an instruction ends with a semicolon, we are not interested in its return value. In Rust, we can make a function returning a value simply by not placing a semicolon at the end of an expression.

## 3.2 Rust-Code-Analysis library

*Rust-Code-Analysis*[18] (*RCA*) is a Rust library to analyze and compute metrics from source codes written in different languages. The library can generate new information and measures by inspecting an intermediate representation of a source code file called *Abstract Syntax Tree* (*AST*). RCA generates metrics at different levels of granularity, adapting the computations to the programming language of the input source code. The library is a static code analyzer, software that helps programmers to know more about their code before even compiling it. RCA can compute metrics fast, even when analyzing large repositories, by exploiting many Rust language features.

The library was initially born as a tool to support the development of the Firefox browser. With more than five thousand changes applied every month by more than five hundred developers worldwide, the Firefox codebase needs continuous

inspection. In fact, any change could introduce potentially harmful code. RCA was one of the elements developed to solve this issue and is currently used to evaluate risks introduced by changes and prevent the injection of new defects in the Firefox codebase.

Rust-Code-Analysis can compute the following metrics: *CC* (McCabe's cyclomatic complexity), *SLOC* (number of lines), *PLOC* (number of physical lines or instructions, *LLOC* (number of logical lines or statements), *CLOC* (number of comments), *BLANK* (number of blank lines), *HALSTEAD* (a suite that provides a series of complexity and maintainability measures), *MI* (a suite that provides a series of maintainability measures), *NOM* (number of functions and closures), *NEXITS* (number of possible exit points from a method/function), and *NARGS* (number of arguments of a function/method).

RCA can currently analyze source code files written in different programming languages: *C*, *C++*, *C#*, *CSS*, *Go*, *HTML*, *Java*, *JavaScript*, *Python*, *Rust*, *Typescript*. However, through its modular design, the library will be able to support many other programming languages in the future. The code structure of RCA helps the addition of new functionalities for a new programmer considerably. A contributor can apply a series of changes to a module in order to support a new language or implement a new metric, leaving the rest of the code untouched.

RCA library is publicly accessible on *GitHub*[19] and *Crates.io*[20], the Rust community's crate registry[21]. A *crate* is a compilation unit in Rust, a type of artefact managed by the Rust compiler. During a compilation process, the Rust compiler *rustc* turns a crate in source code form into a crate in a binary form: either an executable or a library. RCA is released under the Mozilla Public Licence v2.0 and can run on all the most common platforms like *Linux*, *macOS*, and *Windows*. A programmer can build a library using the standard Rust package manager *Cargo*. *Cargo*[22] manages the dependencies of a library, making the build process automatic. Alongside the library, a command-line interface (CLI) called *rust-code-analysis-cli* has been provided. The CLI allows a user to interact efficiently with the APIs exposed by the library. For example, it can be used to run RCA over a repository and print or export the generated metrics in different formats: *Cbor*, *Json*, *Toml*, *Yaml*. One can also install the CLI tool through *Cargo*.

### Example

With the following line, we can use Rust-Code-Analysis CLI to produce metrics for our *main.rs* file introduced in the previous example. We can do that by providing the CLI with the JSON output format and the location of the resulting file.

```
cargo run -p rust-code-analysis-cli -- -m -p main.rs -O json -o .
```

Once we open the JSON file generated by RCA, we can see a data structure containing all metrics associated with the different spaces. Spaces are an abstraction placed on top of the AST, and they have been introduced by RCA to report metrics at various levels of granularity for a single source code file.

```
{
    "name": "main.rs",
    "start_line": 1,
    "end_line": 3,
    "kind": "unit",
    "spaces": [
        {
            "name": "main",
            "start_line": 1,
            "end_line": 3,
            "kind": "function",
            "spaces": [],
            "metrics": {
                "nargs": {...},
                "nexits": {...},
                "cognitive": {...},
                "cyclomatic": {...},
                "halstead": {...},
                "loc": {...},
                "nom": {...},
                "mi": {...}
            }
        }
    ],
    "metrics": {
        "nargs": {...},
        "nexits": {...},
        "cognitive": {...},
        "cyclomatic": {...},
        "halstead": {...},
        "loc": {...},
        "nom": {...},
        "mi": {...}
    }
}
```

### 3.2.1 Tree-sitter

*Tree-sitter*[23] is a parser generator tool and an incremental parsing library. It can generate a *Concrete Syntax Tree* (*CST*) from an input source code file. It can even efficiently update it if the input file is being edited. A CST represents the grammar rules present in a source code file, as precise as possible, in a tree-like form. RCA uses Tree-sitter because it is:

- **Efficient**: The library is fast enough to parse every real-life keystroke in a text editor.
- **Robust**: A Concrete Syntax Tree is generated even in the presence of syntax errors.
- **General**: The tool can parse almost any programming language, and it is quite generic to support more languages in the future.
- **Dependency-free**: The runtime library written in C does not require any dependency, so programmers can embed it in any software.

RCA uses the Tree-sitter library to generate a simplified version of a Concrete Syntax Tree called *Abstract Syntax Tree*[24] (AST). While a CST contains all grammar rules recognised in a source code file, an AST stores just the essential information for the analysis process, discarding every syntactic clutter. Both a CST and an AST are tree-like structures composed of syntactic nodes. Every syntactic node corresponds to a grammar rule defined and recognised by Tree-sitter, a piece of language syntax that the library has found in our code. When Tree-sitter fails to recognise something in a source code, it generates an error node in the corresponding place within the AST. That is especially useful for identifying syntactical errors in source files. This library allows us to separate RCA into two major loosely-coupled components: language parsers and metrics computation modules. Since Tree-sitter is available on GitHub, we can report any problem related to parsing on the relative issue page. In this way, the Tree-sitter team is informed of the issue and can promptly act to solve the problem. On the other hand, RCA programmers can focus on metrics computation modules without worrying too much about any parsing issue.

#### Example

For our example, we again consider the simple *"Hello World"* Rust program introduced previously. However, instead of focusing on the characteristics of the Rust programming language, we can now focus on how Tree-sitter parses our simple Rust source code. First, we run the Tree-sitter library directly on our source code file. This way, we can show all elements the library has recognised in our code

as a tree-like structure. Then, we illustrate how to produce a more compact and meaningful representation of the same tree-like structure from RCA CLI. The metric computation inside RCA is possible using this compact representation of a source code generated by the Tree-sitter library. We expect Tree-sitter to recognise the function definition, the macro call, the printed string and all other syntactic elements that make our Rust program compilable.

We can start by focusing on the Tree-sitter command-line tool. Our first objective, for now, consists in parsing our simple source code file to produce the relative CST representation. First, we need to download the Tree-sitter Rust grammar from the official GitHub repository. A grammar is a set of rules applied to specific tokens to represent the correct syntax of a language. Once positioned inside the root directory of the grammar, we need to use the downloaded grammar to generate a Rust language parser. The parser is the component which generates the CST. This tree-like data structure carries all information about an input source code. With the *tree-sitter generate* command, we can generate a Rust language parser, which is a long C language source file called *parser.c*. In the end, passing as input to the *tree-sitter parse* command our file name, we can produce the Tree-sitter Concrete Syntax Tree and inspect it.

```
(source_file [0, 0] - [3, 0]
  (function_item [0, 0] - [2, 1]
    name: (identifier [0, 3] - [0, 7])
    parameters: (parameters [0, 7] - [0, 9])
    body: (block [0, 10] - [2, 1]
      (expression_statement [1, 4] - [1, 30]
        (macro_invocation [1, 4] - [1, 29]
          macro: (identifier [1, 4] - [1, 11])
          (token_tree [1, 12] - [1, 29]
            (string_literal [1, 13] - [1, 28])))))))
```

As we can see, the CST produced by Tree-sitter provides many information about our simple Rust program. Looking at the CST, we immediately notice that the library has recognised several items. The tool has also annotated the item position in the source code file in the form of line and column coordinates. As we can see, the line and column count starts from zero. Tree-sitter correctly recognised a source file composed uniquely by a function without parameters. In the body of that function, we then see a single expression statement composed of a macro invocation. Inside the macro, we notice a name identifier and a token list, which in our case, is a list made of just one string. The library also supports the generation of ASTs, a tree structure with only named syntax nodes[25], which are nodes with an explicit name in the grammar. RCA converts a CST into an AST by removing all the anonymous syntax nodes, which are the nodes without a name. RCA produces an

AST because it is easier to manage since it does not include some useless details instead present in a CST. We can use the *rust-code-analysis-cli* tool to parse and print out the AST generated by RCA for our simple Rust source code file by typing this command:

```
cargo run -p rust-code-analysis-cli -- -p ./path/to/file/main.rs -d
```

The AST printed out by the RCA CLI (**Figure 3.1**) is also a more readable version of the CST generated by the Tree-sitter library. In yellow, we can see the syntax node. Next to the syntax node type, the CLI tool reports in green the position of the syntax node in the source code file. RCA CLI expresses coordinates by reporting the line and column for each token. In this case, the count of lines and columns starts from one to be more user-friendly than Tree-sitter's output, so the first row or column would be the number one. As the last part, next to the coordinates, there is the corresponding token. That is the portion of the source code file that the library has recognised.

```
{source_file:138} from (1, 1) to (4, 1)
└─ {function_item:170} from (1, 1) to (3, 2)
   ├─ {fn:57} from (1, 1) to (1, 3) : fn
   ├─ {identifier:1} from (1, 4) to (1, 8) : main
   ├─ {parameters:192} from (1, 8) to (1, 10) : ()
   │  ├─ {(:4} from (1, 8) to (1, 9) : (
   │  └─ {):5} from (1, 9) to (1, 10) : )
   └─ {block:266} from (1, 11) to (3, 2)
      ├─ {{:6} from (1, 11) to (1, 12) : {
      ├─ {macro_invocation:219} from (2, 5) to (2, 29) : println!("Hello World!")
      │  ├─ {identifier:1} from (2, 5) to (2, 12) : println
      │  ├─ {!:77} from (2, 12) to (2, 13) : !
      │  └─ {token_tree:149} from (2, 13) to (2, 29) : ("Hello World!")
      │     ├─ {(:4} from (2, 13) to (2, 14) : (
      │     ├─ {string_literal:283} from (2, 14) to (2, 28) : "Hello World!"
      │     │  ├─ {":124} from (2, 14) to (2, 15) : "
      │     │  └─ {":124} from (2, 27) to (2, 28) : "
      │     └─ {):5} from (2, 28) to (2, 29) : )
      ├─ {;:2} from (2, 29) to (2, 30) : ;
      └─ {}:7} from (3, 1) to (3, 2) : }
```

**Figure 3.1:** AST produced by the RCA CLI.

In our case, the Tree-sitter Rust grammar has been able to recognise every token in our simple code snippet. It is evident because there is no error node in the AST. That means the code is syntactically correct: we can compile it without getting any error from the compiler. RCA assumes that the provided code is

syntactically correct and, therefore, compilable. The library can guarantee that the computed metrics are meaningful and precise only with those premises. Below we have reported a more readable representation of the AST (**Figure 3.2**). From it, we can see the grammar rules that have been matched in our source code and the relative tree nodes.

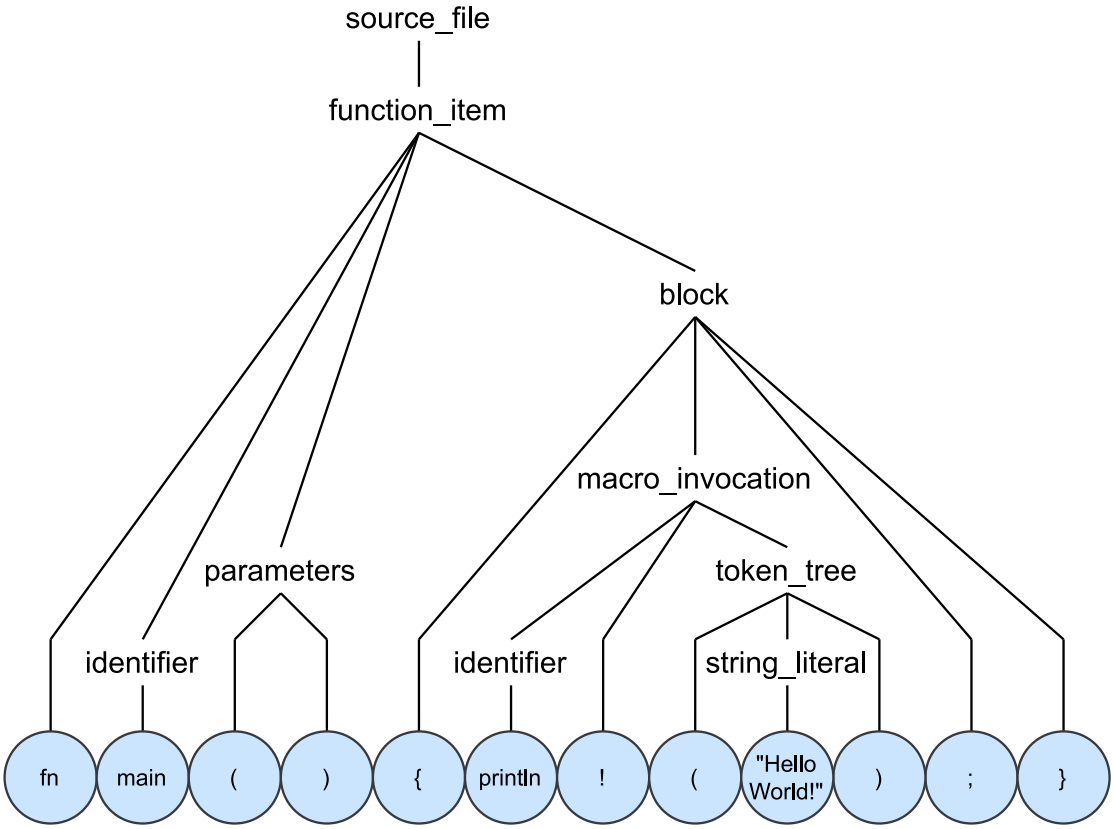


**Figure 3.2:** AST graphical representation.

## 3.3 Library integration tests

The RCA library already defines several unit tests for each implemented metric. Those unit tests verify the correct computation of each metric independently. Specifically, they analyse a few lines of ordinary code written in different languages. RCA has defined unit tests to achieve essentially two goals:

- Verifying the correctness of measures in typical use cases, such as testing source code one can find in active codebases.

- Verifying measures correctness in extreme cases, such as testing unusual and rare source codes: the corner cases present in each programming language.

Unit tests are composed of a string representing the source code of a test and the expected measures. A unit test fails when RCA does not generate the expected measurements. Unit tests, however, are not enough to test the correct behaviour of a system as a whole. By adding integration tests to RCA, we can perform more checks on all implemented metrics at once, testing complex and real workflows. Moreover, the execution of integration tests after any commit pushed in the GitHub repository guarantees that, overall, the library is still working as expected, even after some changes.

### 3.3.1 Implementation

With our integration tests, we verify if RCA generates correct metrics values for the source code of a given repository. We have selected three repositories of various sizes as subjects of the tests. We have pinned the repositories to a specific version, so tests always run on immutable code. Each one of the three repositories represents a test case for a specific source code language:

- Rust Standard Library for Rust
- PDF.js for JavaScript
- DeepSpeech for C/C++

For testing code written in Rust, we have selected the *Rust Standard Library*[26]. The Rust Standard Library defines a minimal set of primitive types, modules, macros, and keywords that compose the Rust language. The library is continuously updated, and it represents an excellent example of a working and complex codebase.

*PDF.js*[27] is a Mozilla open source project written in JavaScript that allows browsers to parse and render PDF files. The viewer is embedded in Firefox, and it is available as an extension for Chrome. The community continuously improves the code, and the project represents an excellent example of an actual medium-sized JavaScript repository.

As an example of a modern C++ repository, we have chosen the *DeepSpeech*[28] project. DeepSpeech is a Speech-To-Text engine that allows users to transcribe English voice audio files into text automatically. To do that, it uses a model trained by machine learning techniques. DeepSpeech is a Mozilla project, and it represents a massive and complex addition to our integration tests for what concerns the C++ language.

In order to have valuable data for tests, we have created an additional repository filled with thousands of JSON files generated running RCA CLI over the three chosen repositories. This additional repository represents how the JSON files generated by RCA vary over time with the evolution of the library.

### 3.3.2 Synchronization

Rust, by default, executes unit and integration tests concurrently. This choice is very convenient because we can complete tests faster, allowing programmers to focus on their results. However, this apparent beneficial choice also introduces some limits. With a parallel test execution, however, we cannot write tests that need to read and write the same information at the same time. The parallel test execution, in this case, would lead to race conditions and unexpected results. Even the introduced integration tests were initially affected by this limit. In order to run correctly, all three tests need to download the repository containing the JSON files that will be used for the comparisons. However, the three parallel tests should not start downloading all the same repository three times. Moreover, the tests should not consider the repository ready if its download is not over yet.

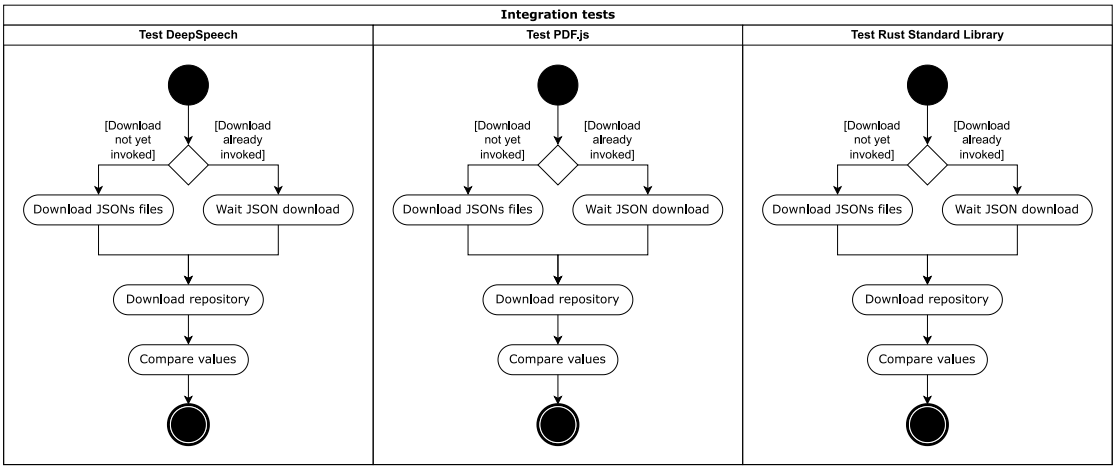


**Figure 3.3:** Integration tests threads activity diagram.

To solve this problem, we have decided to use one of the many synchronization primitives offered by Rust: *Once*. This primitive provides a practical method called *call_once*, which takes as an argument a function. As the name suggests, the function, passed as an argument to the method, is the code which must be executed *only once* by all the running threads: the initialization function. This means that only the fastest thread can execute the code passed as an argument to the *call_once* method. All remaining threads will next notice that someone has

already invoked the *call_once* function, blocking and waiting for the completion of the initialization function performed by the fastest thread. Once the fastest thread completes its initialization, the other threads unblock themselves and proceed with their execution. This way, only the fastest thread downloads the JSON files. The other threads are informed about the download status through the synchronization primitive (**Figure 3.3**).

### 3.3.3 Producer and consumer

Together with the concurrency level introduced with tests, we have implemented an additional one to explore the content of a repository directory. RCA already uses a parallel exploration of a directory content in the CLI. As we have seen previously, a user can provide an entire folder as input code to the command line tool. In that case, RCA navigates all source code files inside a mechanism called *single producer and multiple consumers* (**Figure 3.4**) and computes the metrics for each file. The solution is a variant of the *producer and consumer concurrency design pattern*. In this pattern, a producer generates data and puts them into a queue as soon as they are ready. At the same time, a consumer extracts the elements in the queue one at a time to process them in order, starting from the oldest one.

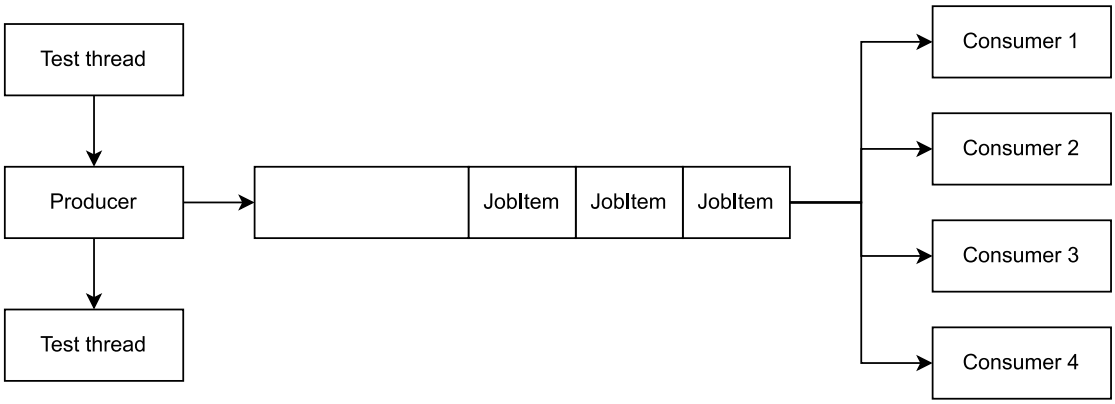


**Figure 3.4:** Single producer and multiple consumers channel.

In our case, the producer and consumer mechanism starts as soon as a single integration test thread has finished downloading the two necessary repositories. In that moment, we create a new producer thread. Its main task consists of exploring a provided path recursively. When the producer thread encounters a file, it sends its path inside a *channel* such that the first free consumer thread can work on it. A channel is a concurrent data structure specifically designed for message passing between threads.

To perform the exploration of a directory, we use a library called *Walkdir*[29]. Walkdir is a Rust library which provides an efficient and cross-platform implementation of recursive directory traversal. With it, we can access the content of a directory in a compact and straightforward way. Moreover, the library offers many options to configure the directory traversal, adapting it for any need.

After the creation of the producer thread, we have created a set of consumer threads. We have decided to set the number of consumer threads for the integration tests to four. We have made this choice because we have hypothesised that most CPUs nowadays are able to manage four threads efficiently. In case of necessity, however, one can change this number in the tests to adapt the number of test threads to more specific environments.

Consumer threads continuously check for any item in the channel. When a consumer retrieves a file from the channel, it immediately invokes the function that performs the comparisons. When the producer has finished exploring the folder content, the execution resumes from the main thread. While the producer and the consumers are still running, in fact, the main thread is blocked on a *join* call over the producer, waiting for it to terminate to continue. A join call is a way to block the execution and wait for the termination of a thread. After the producer has ended, the main thread stops the consumers using a technique called *poisoning*. The technique consists of sending a poison message to all the consumers to end their execution. After poisoning the consumers, the main thread performs a join call over them to wait for their termination.

For implementing this mechanism, we have used *Crossbeam*[30], a Rust library that provides a set of tools for concurrent programming. Among these tools, the library includes multi-producer multi-consumer channels for message passing, such as the one we have used for our integration tests.

### 3.3.4 Metrics files generation

Through RCA CLI, we can generate metrics for all the source code files contained in a repository and export them as JSON files. However, some codebases are enormous and might generate thousands of JSON files. How can we pair a JSON metrics file with the respective source code file? RCA CLI initially solved this problem by producing JSON files with long names. Each file name was the path to the source code file converted into a string. So, for example, if we consider this file:

```
1 path/to/file/main.rs
```

The corresponding JSON file is:

```
path_to_file_main.rs
```

However, what would happen if we were to encounter two source code files such as the following ones?

```
path/to/file/main.rs
path/to/file_main.rs
```

In this case, the metrics generation would produce only one file called:

```
path_to_file_main.rs
```

We would write the content of this file twice and have a single JSON metrics file for two different source code files. To solve this naming ambiguity conflict, we have changed how the command line interface generates the names of each file. We have decided to produce metrics files in a directory structure similar to the one of the input source code repository. Using this solution, the name of the metrics file is identical to the name of its relative source code file, with, in addition, the JSON extension. With this new directory generation and convention for file names, we can safely produce JSON files without naming conflicts.

### 3.3.5 Comparisons

Once we have generated the metrics for a single source code file and parsed its generated JSON file, we can proceed with the comparisons. For each source code file, we compare at runtime a *Value* and a *FuncSpace*. A *Value* is an *enum* that contains any valid JSON file content. An enum or enumeration is a data type consisting of various possible alternative values. The *FuncSpace* struct is the return value of the *get_function_spaces* API from the RCA library. The *FuncSpace* struct represents the format of the metrics produced by RCA and some more information we will explain later in the chapter.

```
pub struct FuncSpace {
    pub name: Option<String>,
    pub start_line: usize,
    pub end_line: usize,
    pub kind: SpaceKind,
    pub spaces: Vec<FuncSpace>,
    pub metrics: CodeMetrics,
}
```

When we export the metrics as JSON, we serialise the struct *FuncSpace* using a Rust framework called *Serde*[31]. The serialisation process builds a JSON file starting from the internal fields of a *FuncSpace* struct. To read the content of a JSON file for our test, we need to do the inverse operation of the serialisation, which is the deserialisation. We can do this operation with Serde as well. The deserialisation of a JSON file produces the *Value* struct described in the preceding paragraph, which contains all the *FuncSpace* fields obtained by parsing the JSON file. We can access the JSON fields using the *Value* struct as an associative array. Any index can be either the name of a deserialised field or an array index if the field to access is an array element. The fields of a *Value* struct, however, have a different type from their respective *FuncSpace* field. So it is necessary to do some type of conversion just before comparing the values. We can convert and compare the metrics values by calling the *compare_structs* function and providing references to the two structs to compare. The *compare_structs* function traverses spaces in a recursive way because a space can contain other spaces. We can then convert and compare all the fields of the structs by iterating over the spaces contained in a selected space, again through the *compare_structs* function.

```
/// Compares a FuncSpace and a Value field by field
fn compare_structs(funcspace: &FuncSpace, value: &serde_json::Value){

    // Compares struct fields
    ...

    // Recursion
    for (pos, s) in funcspace.spaces.iter().enumerate() {
        compare_structs(s, &value["spaces"][pos]);
    }
}
```

We have used *unwraps* to perform runtime checks of any conversion errors. Unwraps are a way to extract a value from an optional type in Rust. When an unwrap tries to extract a *None* value from an *Option*, the function panics, which means the execution stops with an error message. Because the program ends, there is no way to recover from the mistake. Since Rust has many ways to handle and recover errors, unwrap usage is usually not advised. In our tests, we exploit unwraps to immediately stop a test execution with an error message when a conversion fails. That means an error has been found, so our test stops and reports a failure.

In case of space names, comparing the considered string and the expected one requires some precautions. We have noticed that space names could have inside them newline characters. Newlines are treated differently in Windows and Linux. In Windows, a newline symbol is composed of two characters: a *carriage return* and

a *line feed*. On Linux, a newline symbol is composed of just a *line feed* character. So, to make the tests pass on both systems, we need a way to handle different newline characters. On Windows, we have solved the problem by removing any *carriage return* character from the space names generated at runtime by our tests. We have discovered this issue through the RCA Continuous Integration system: during the first submission of the code, only the integration tests performed on Windows failed. After applying this change, both Windows and Linux integration tests ended their execution successfully.

The actual metric values comparison is performed by extracting the *metrics* field from the two structs and by invoking the *compare_f64* function for each couple of metric values to be compared. The *metrics* field contains all the computed metrics values. Since RCA produces most metrics values as floating point numbers, we need a proper way, that considers their limited precision, to compare them. We have designed a small function for that purpose. Below, as a sample, it is reported the comparison code for the LOC metric. The produced code is straightforward to maintain and extend.

```
// Extract metrics for comparisons
let metrics1 = &funcspace.metrics;
let metrics2 = &value["metrics"];
...
let loc1 = &metrics1.loc;
let loc2 = &metrics2["loc"];
compare_f64(loc1.sloc(), &loc2["sloc"]);
compare_f64(loc1.ploc(), &loc2["ploc"]);
compare_f64(loc1.lloc(), &loc2["lloc"]);
compare_f64(loc1.cloc(), &loc2["cloc"]);
compare_f64(loc1.blank(), &loc2["blank"]);
```

Before the comparisons, we truncate the metrics values to the third decimal digit. We perform this operation to ignore any minimal precision error in the values. The serialisation and deserialisation processes can introduce these precision errors. In order to compare two floating point numbers, we need to verify this formula:

$$|f1 - f2| < \varepsilon$$

Where $f1$ and $f2$ are the two floating numbers to compare, and $\varepsilon$ is the minimal difference between a floating point number and its next one. This straightforward formula allows to compare two floating numbers with exact precision. An equality check is enough for other types of fields, such as strings or integers. Serde serialises invalid values, such as *NaN* or infinite measures, as *null* values. Therefore, to check those measures, we need to verify that the corresponding deserialised values obtained from the JSON file are *null*.

```rust
/// Compares two f64 values truncated to 3 decimals
fn compare_f64(f1: f64, f2: &serde_json::Value) {
    if f1.is_nan() || f1.is_infinite() {
        assert!(f2.is_null());
    } else {
        let ft1 = f64::trunc(f1 * 1000.0) / 1000.0;
        let ft2 = f64::trunc(f2.as_f64().unwrap() * 1000.0) / 1000.0;
        assert!((ft1 - ft2).abs() < f64::EPSILON);
    }
}
```

To summarise our solution, we can look directly at the function code. First, we check whether infinite or nan measures are null in the JSON. After that, we suppose the generated value is finite and valid, so we truncate the measure to the third decimal and apply the comparison formula. This solution allows us to cover all the possible comparisons, interrupting the text execution when a metric value is different from the one present in its JSON file.

## 3.4 Metrics computation

Since now, we have seen the output generated by RCA but have not yet explained the measuring process in detail. The first step for metrics computation is the AST generation from a source code using the Tree-sitter library. An AST stores the code information in a tree-like structure composed of syntax nodes, each carrying information about a matching piece of code. One of the most significant properties about syntax nodes is their type, which corresponds to the grammar rule matched in the source code. With it, we can distinguish a syntax node from another. A syntax node also contains information about the matched code, for example, its spatial coordinates inside the source code file.

On top of the AST, Rust-Code-Analysis builds another abstraction by dividing the source code into *spaces*. A space is any structure in the source code file that can incorporate a function, and it is represented in the code by the *FuncSpace* struct. In particular, we can define a space with the following information:

- **Space name**, which is an optional string since we can find spaces without names in source code files. We can call these spaces *anonymous spaces*.
- **Start and end lines** to locate the space position in a source code file.
- **Space kind**, which is the type of space. At the moment, the following space kinds have been defined: *Unknown*, *Function*, *Class*, *Struct*, *Trait*, *Impl*, *Unit*,

*Namespace*. Each represents a container type, which could contain particular subspaces depending on the characteristics of the considered programming language.

- **Spaces**, since a space can contain many subspaces. This field is a dynamic array of spaces.

- **Metrics**, which is a struct of type *CodeMetrics*. This struct defines the structure of the computed measures, grouped by their metric name.

### 3.4.1 Metrics function

One of the RCA library's main features is a function called *metrics*, which we can find in the *spaces.rs* file. This function parses an AST and computes the code metrics. We can describe its implementation through various steps:

1. The traversal of the AST happens through a loop and a node stack. The function pops out a single syntax node from the node stack and analyses it at each iteration of the loop. If a node has children, *metrics* adds those children nodes to the stack in order of appearance in the source code. This way, at the next iteration of the loop, the function can pop out the first child from the stack, traversing the AST nodes in the same order explained before.

2. For each node, the stack also stores a level variable. This variable represents the level of nesting of the space the node is part of. At each new iteration of the loop, the function pops out from the stack a syntax node and its level variable. If the level of the syntax node popped out from the stack is lower than the level of the syntax node retrieved in the last loop iteration, then it means we have finished exploring a space. In that case, *metrics* calls a *finalize* function to merge a subspace into its container space and then updates the variable containing the level of the previous loop iteration.

3. As the next step, *metrics* can then analyse the type and the content of a syntax node. A new space is created and stored in a state if the syntax node is a function or a container of functions declaration. An example of a container of functions is a Java class or a C++ namespace. The *metrics* function pushes the state into the state stack, increments the current level and updates the level of the previous loop iteration. This way, the function has a buffer to save all the metrics of our just discovered space, which is the last state contained in the state stack. It can extract the last state and update the metrics stored in it starting from the node we are analysing.

4. At the end of the loop, we may have more than one state in the state stack. To solve this issue, we call the *finalize* function one more time to merge all the remaining states into one state.

5. Finally, the function extracts the last state, adds the file name, and returns it.

### 3.4.2 Finalize function

The *finalize* function takes as arguments a state stack and the number of iterations to carry out. The *metrics* function calls *finalize* to compute additional metrics and perform the merge operation. We also call *finalize* inside *metrics* when we detect the ending of a space. In particular, this happens when the nesting level of the current iteration is lower than the nesting level of the last one. The merge operation is necessary to create the nested metrics structure presented above. This structure not only gives the computed metrics values in blocks but also helps to understand the source code file structure, often composed of a tree of nested spaces. In addition, the space structure can help find the parts of the code with the most strange metrics values, where programmers could find most of the issues. The *metrics* function also invokes *finalize* in the final part of its execution. In this last call, we use the maximum number of iterations instead of the difference between the nesting levels of the last two iterations. This way, we are sure that all the spaces have been merged into one, returning a single space as the last operation.

#### Example

Now we can see, with an example, how the algorithm that computes metrics works in a real case. We will simulate the execution of the *metrics* function over a Rust code containing only two function definitions. As we can see from the code, the task of the two defined functions, *foo* and *bar*, is to print out two strings.

```
1 fn foo() {
2     println!("Foo!");
3 }
4
5 fn bar() {
6     println!("Bar!");
7 }
```

We now focus on some specific iterations of the AST. In particular, we are interested in the iterations that discover new spaces and the ones that detect the ending of a space. At every step, we show the state stack content and describe the operation applied to it:

1. In step one (**Figure 3.5**), we detect a space of kind *Unit*. A space of kind *Unit* represents the entire source file content and contains all the spaces of a file. We push a state containing a space of kind *Unit* into the state stack.

2. In step two (**Figure 3.6**), we detect a space of kind *Function*. We push a state containing a space of kind *Function* into the state stack.

3. In step three (**Figure 3.7**), we detect the ending of the space of kind *Function*. We merge the state containing the space of kind *Function* into the one containing the space of kind *Unit* within the state stack.

4. In step four (**Figure 3.8**), we again detect a space of kind *Function*. We again push a state containing a space of kind *Function* into the state stack.

5. In step five (**Figure 3.9**), we again detect the ending of the space of kind *Function*. We again merge the state containing the space of kind *Function* into the one containing the space of kind *Unit* within the state stack.

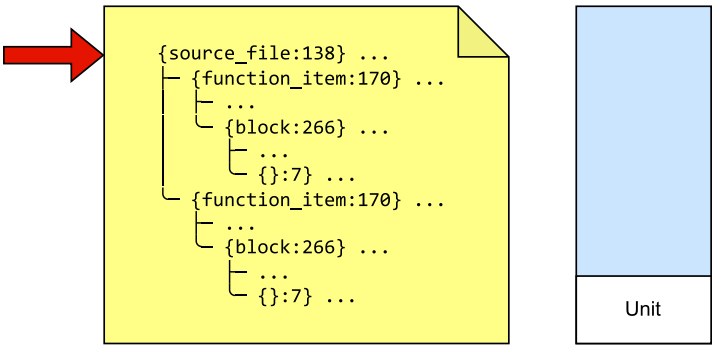


**Figure 3.5:** State stack at step 1.

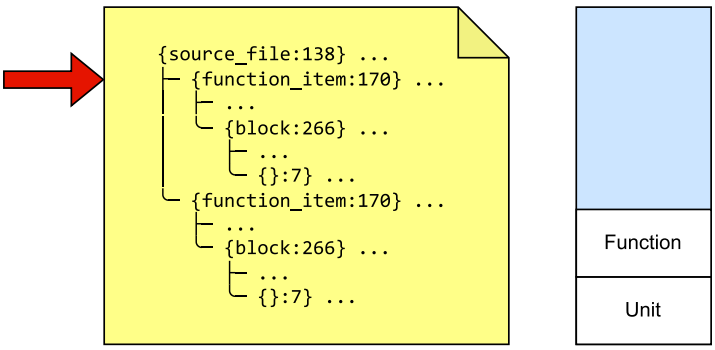


**Figure 3.6:** State stack at step 2.

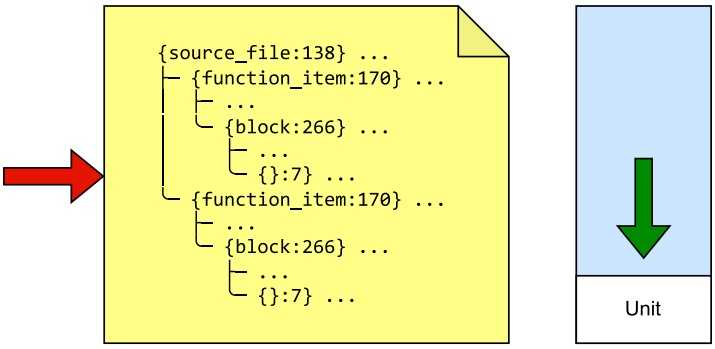


**Figure 3.7:** State stack at step 3.

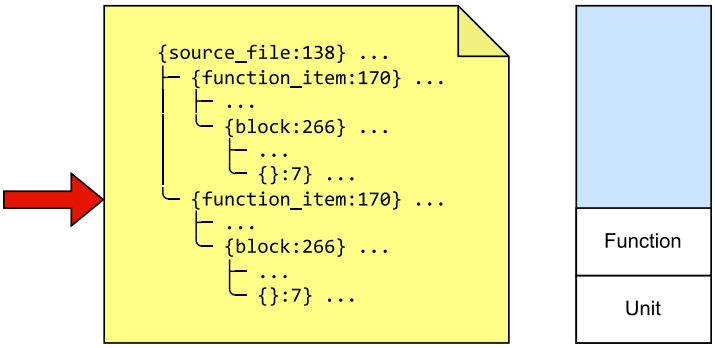


**Figure 3.8:** State stack at step 4.

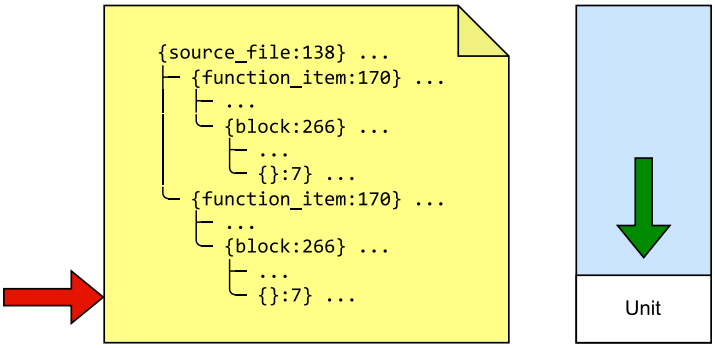


**Figure 3.9:** State stack at step 5.

### 3.4.3 Averages computation

In the *finalize* function, we have seen that when a space ends, we can compute additional metrics. We can compute at this point averages, minimum and maximum metrics values. RCA defines each metric in a specific module. The metric module contains all information and methods needed to compute a metric, plus a set of unit tests for each metric. RCA stores any data related to a metric in a struct called *Stats*. To compute the average of a metric for a space, we need essentially three components:

- The current space metric value.
- The sum of all the metric values of the nested spaces.
- A counter of spaces which contributes to the metric sum.

We can find all this information stored inside the Stats struct. The first and second elements of the list above are already present in the struct, so we only need to add the counter of spaces. The counter introduced has been initialised to one, and it increments its value every time a subspace gets merged into its container space. The increment may happen, for example, when we merge a space of kind *Method* into its parent space of kind *Class*. The counter of spaces represents the number of subspaces contained inside a space plus one, to take into account the container space itself. Below we can see an example of the NOM *Stats* struct. As a final side note, we can notice that a space represents a portion of code, and its kind depends on the programming language. Spaces of kind *Method* and *Class* are for example very common in Java source code files.

```
pub struct Stats {
    // The current space metric value
    functions: usize,
    // The sum of the current space metric value plus all its inner
   spaces metric values
    functions_sum: usize,
    ...
    // The number of the inner spaces of the current space plus one
    space_count: usize,
}
```

We can see from this code how we increase the space counter by one every time we perform a merge of a space. The increment happens because every space starts with a space counter equal to one. We increase the space counter in the *merge* function, which is a method defined in the *Stats* struct of the NOM module that performs the spaces merge operation for a metric.

```
impl Stats {
    /// Merges a second `Nom` metric suite into the first one
    pub fn merge(&mut self, other: &Stats) {
        ...
        self.space_count += other.space_count;
    }
    ...
{
```

Once the counter is available and correctly managed, we can add the functions that compute and return the average values that will be serialised into a JSON file alongside the other measures. We can compute an average value by dividing the sum of the metric values of the subspaces, which includes the metric value of the current space, by the number of subspaces plus the current one. The *#[inline(always)]* attribute optimizes the execution of the smaller functions. Inlining a function means replacing the invocation of the function with the function code in the invoker. The technique is advantageous when we invoke the function repeatedly from the same place, as it happens in our case.

```
impl Stats {
    ...
    /// Returns the average number of function definitions over all
    spaces
    #[inline(always)]
    pub fn functions_average(&self) -> f64 {
        self.functions_sum() / self.space_count as f64
    }
}
```

We now have a new metric value for each recognised space. Average values carry innovative information which we can use, for example, to detect which space contributes the most or the least to the total metric value of a source code file. Those metrics could help programmers to find which parts of the code contain unexpected metric values very quickly.

# Chapter 4

# Metrics implementation

## 4.1 Workflow and tools

The main objective of this chapter is to illustrate the process of implementing new measurements in the rust-code-analysis tool. As seen in the second chapter, each metric has its purpose, rules, advantages, and disadvantages. However, since a metric is an abstract concept, it needs to be implemented in a tool to use it and adequately analyze its upsides and downsides.

The implementation process needs the support of a precise definition of the metric and many unambiguous and complete examples. Unfortunately, this is not always the case. It is possible to find outdated or too formal metrics definitions. The examples provided can be scarce and too trivial for a satisfying implementation. Moreover, definitions can leave gaps the programmer is called to fill by taking the more profitable paths for specific cases and environments.

Depending on how many possibilities the programmer intends to cover, the personal choices during a metric implementation can be few or many. Programmer choices can often result in having many different implementations of the same metric despite starting from an identical definition. For those reasons, implementing a metric requires knowledge, documentation, motivated choices, and precision.

We have divided the chapter into three subchapters, each one dedicated to an implemented metric category. We have chosen this subchapter organization to help the reader understand the approaches adopted to implementing different metrics types in the RCA tool.

We have decided to implement each selected metric only in the *Java* language. Java is a high-level object-oriented programming language designed to work on any platform without the need to recompile the code. The language implements this feature by compiling source files into an intermediate bytecode representation, which can run on any platform with a Java Virtual Machine installed. Although born in 1995, the Java programming language is still widely used, especially for client-server applications. It is currently one of the top languages used on GitHub, according to the GitHut project[32].

To implement code metrics inside RCA, a deep knowledge of the Java language, its constructs, and how the Tree-Sitter parsing library handled them is necessary. Luckily Oracle discloses these and many more details about the Java language on its *The Java Tutorials* website[33], along with many beginner-friendly examples. For the parser integration, instead, we have consulted the tree-sitter-java[34] grammar.

As seen in the previous chapter, RCA uses the concept of space to compute and serialize metrics into different final formats. It is necessary for a contributor to have a clear understanding of how the RCA spaces abstraction works in order to implement a metric. By understanding how RCA analyses the tree-sitter nodes and computes the metrics, a developer can safely inject its code in the correct spots. Additionally, to test the implementation during the development, it is possible to produce JSON format files and the output generated by the RCA CLI. The JSON is the only format we have worked with since it is the most flexible and easy to manage. Those two methods require minor code modifications and can help verify the work produced until a particular moment.

RCA provides an engine we can use to inspect nodes and a solid structure to follow when introducing new measurements. However, the programmer must be careful, follow the provided tracks, and accept the enforced constraints to avoid damaging the existing work. In the beginning, new contributors must face at least two primary limits imposed by RCA: memory and error management. Each new implementation has to limit its memory usage as much as possible so that the tool can maintain its high computation speed. Moreover, a programmer must carefully consider and manage every error and exception to preserve the tool's reliability. RCA should never unexpectedly stop working while handling both syntactically correct and incorrect code.

For each new metric introduced in RCA, we have added a set of unit tests to demonstrate the correct behaviour of the metric computation for the most common scenarios. Writing unit tests is extremely useful for checking the correct metric

computation incrementally. Using a test-driven approach, we can start the implementation by defining a few trivial tests and adding more until we are satisfied. When passing, each test guarantees that a specific type of case is still correctly handled after some changes.

In addition to the metric unit tests, before submitting our work to be reviewed as a pull request on GitHub, we have performed a series of manual integration tests on three repositories of various sizes.

*Notepad*[35] is a simple notepad application written in Java that uses a library called Java Swing for its graphical user interfaces. The application provides elementary reading and writing operations. Its source code is composed of only four files.

*Simple-Java-Calculator*[36] is a Java application designed to help novice of the language to learn how to build a simple calculator application. The entire repository consists of a total of five source code files.

Finally, *NetBeans*[37] is a giant Java open-source repository for the popular NetBeans IDE. NetBeans is widely used worldwide for writing Java applications, and it is considered the official IDE for the Java language.

Notepad and Simple-Java-Calculator are examples of very small Java repositories. By computing the metrics for these two repositories by hand and confronting the values with the ones produced by RCA, we can adequately test our implementation on actual working code. We must highlight that we have only computed the measures by hand for the first comparisons. After checking that the RCA metric computation was correct, we saved and updated the produced JSON files into small personal GitHub repositories. That way, we could easily track the variations of the measures for different versions of our implementation and check that the measures always remain the same, even after significant implementation changes.

After we have performed a few checks on the two little repositories with a successful result, we can test our implementation on the large NetBeans repository. Running RCA over such a giant repository with thousands of files would often take minutes to complete. However, if the run is successful, we are sure our implementation does not break any existing RCA working code. Since analysing each file one by one is not humanly possible, we have just checked the metric values for a few random files. This test's main objective is not to check the behaviour of the metric computation like for the first two repositories. Instead, the objective is to test RCA reliability and verify if the tool can still manage many different cases and exceptions without unexpectedly fail.

Other than understanding and coherently inserting our work in the rust-code-analysis environment, a contributor must adhere to Mozilla's review pipeline. Each new change introduced through a pull request in the RCA GitHub codebase has to be tested by an advanced Continuous Integration (CI) system, and it also has to be reviewed by the creators of the repository. The CI system performs many tests on the updated codebase to perform many tedious checks automatically. During this phase, a hypothetical contributor can have confirmation that his changes are applicable. The proposed modifications must be syntactically correct, well-formatted, do not contain any mistakes, be accompanied by an adequate number of unit tests, and work on the most common platforms: Microsoft Windows and Debian Linux distribution.

The rust-code-analysis CI uses several tools to maintain the codebase working and avoid introducing code that may eventually lead to problems.

*Rustfmt*[38] is the instrument used to format Rust code according to the style guidelines. Formatting code consistently across a project is critical to improving its readability and making it easier to maintain. Rustfmt is an excellent tool for doing that since it works on much Rust code and, in some cases, can format code even if it does not compile.

*Clippy*[39] is a tool to recognize mistakes that programmers commonly make when writing code in the Rust programming language. Thanks to a collection of over 550 lints, Clippy can signal potential problems and highlight possible solutions to the developer, so he can fix them before introducing dangerous code in the codebase. We can define a lint as a programming error, bugs, stylistic errors, and suspicious constructs.

Finally, to keep tests organized, RCA CI uses *Codecov*[40]: a code coverage solution that provides test coverage of overall project files and of only changed files. Generally, it is good to have high test coverage on a project because that means the tests defined cover most of the code. Knowing the code coverage of the changed files may help discover the new or modified lines not covered by tests.

To summarize, we can represent the entire process of implementing a new metric for the Java language in RCA with the following image (**Figure 4.1**).

The first step is the research phase, in which we collect information about the metric to implement. Usually, we can start from the paper that contains the metric definition and perform some research online to find some existing open source implementation of the metric.

Once we have enough information, the metric implementation can start. For the implementation, we use an iterative approach. We start with simple examples and define trivial unit tests to cover basic scenarios. On each iteration, we then define new, slightly more complex tests, write the code necessary to handle them and verify the correct behaviour of the metric computation. The computation should work as expected for all the defined scenarios, both the new and the old ones. This way, we are sure we are always moving forward without breaking anything which is already working correctly.

Once we are satisfied with the quality of our work, we can submit it as a pull request on the GitHub repository to start another iterative process, the review one. The CI verifies the formal correctness and coherence of the proposed changes. Moreover, repository maintainers can give feedback to the contributor and suggest applying specific modifications to improve the overall quality of the work. It is possible to go back to the research phase if someone discovers some formal mistakes during the review. In case of implementation errors, we should modify the source code and, if necessary, the tests to correct the errors. In these cases, it is essential to ensure that all the previously defined tests can still complete without failures. Once the work has reached a satisfactory quality, review iterations stop, and codebase maintainers merge the changes into the master branch of the RCA repository.

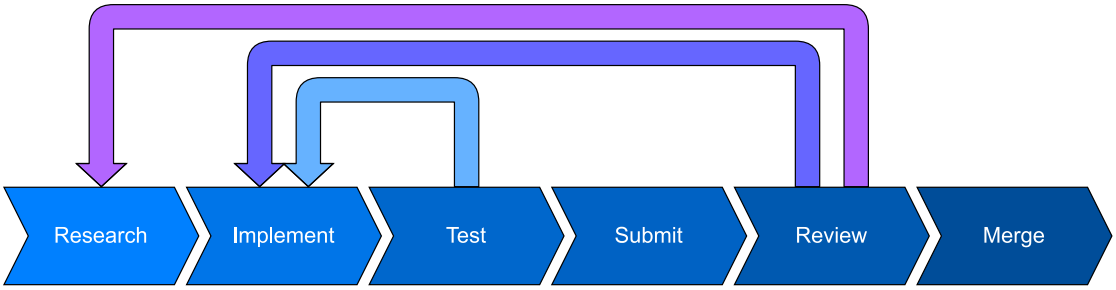


**Figure 4.1:** Metric implementation process.

## 4.2 Size metrics

The purpose of size metrics is to measure the dimensions of source code files or codebases. RCA already provides some size metrics implementation, like the LOC family measures (SLOC, PLOC, LLOC, and CLOC). We can follow these examples without derailing too much from the tracks provided. Moreover, implementing size metrics can be a great starting point for a contributor. In our case, this helped us gain the experience necessary to add more experimental metrics later.

### 4.2.1 Assignments, Branches and Conditions (ABC)

The *ABC* metric is a perfect example of size metrics. As we saw in a previous chapter, its definition is pretty straightforward. Moreover, it does not rely on resources RCA yet cannot offer. It is a metric that fits perfectly in the RCA environment and provides an alternative to other size metrics like the LOC family metrics and the Halstead. So, as the first step, we have created a separate file for it in the metrics folder, reproducing the same internal file structure as the other metrics implemented in RCA.

During the implementation, however, some issues arose from a few specific cases. Most of the problems have originated from the official paper, which has two significant flaws. Firstly, being published in 1997, it is not very recent. Secondly, it is not clear enough to specify the behaviour of the metric in some specific meaningful cases. For these reasons, we had to make calibrated decisions to overcome the paper ambiguities while reaching reasonable trade-offs between the upsides and downsides of our choices.

It is noteworthy to notice that the ABC metric is relatively unknown in the market of static code analyzers. There are very few implementations of this metric online, which is not good because that provides us with a few working examples covering just the most basic cases. We hypothesize that static analysis tools have not widely adopted the ABC metric because of its ambiguous definition. Furthermore, the metric counting rules are similar to the counting rules for another more famous and adopted code metric: the Cyclomatic Complexity.

The primary reference for our implementation was a static analyzer tool called *GMetrics*. GMetrics[41] is a code analyzer that provides calculations and reports about the size and complexity of *Groovy* codebases.

Groovy[42] is an alternative programming language for the Java platform. One of its most peculiar features is bytecode interoperability. Just like Java code, we can convert Groovy code into bytecode. However, the bytecode generated from Groovy code is the same that one would obtain by compiling the same program written in the Java standard language. Moreover, the Groovy language creators have designed it to be easy to learn and remarkably powerful. Similar to other programming languages like Python and Ruby, Groovy has dynamic and static features and supports metaprogramming and functional programming.

Given all the similarities to the standard Java language, a Groovy implementation is valuable to see how we can implement the ABC counting rules in real test

cases. Unfortunately, however, this reference implementation can only partially solve the problem. From the GMetrics source code[43], we have discovered that the tool unit tests[44] only cover the most trivial scenarios. So we must make weighted decisions to cover the most complex real cases.

The ABC metric consists of three counters: Assignments, Branches, and Conditions. We can represent the metric can using these three counters or by computing the magnitude of the vector having the three counters as components. Knowing so, the first thing to do was to define a proper *Stats* structure to contain the metric counters. Just as for the other metrics that RCA implements, the core logic of the metric is in its *compute* function. This particular function is called once for every node of the AST. It receives the node as an argument to inspect it and update the metric *Stats* accordingly.

The *compute* function translates the counting rules logic into working code that can recognise and count the required programming tokens and patterns. We can do this in RCA using a Rust *match* statement. Similar to a *switch* statement, a *match* statement defines different code segments to execute according to the type of syntax node RCA is currently analysing. We have started by implementing the most explicit and unambiguous rules for assignments, branches, and conditions, slowly preparing our way up to more complex cases.

As the metric definition states, we count any assignment operator found in the source code file as assignments. That includes the most used and known assignment operator = or equal sign and *augmented assignments or compound assignments operators*[45]. Those are particular operators that allow programmers to include some types of data modification into an assignment operation. An example of a compound assignment operator is the += operator in Java, which allows you to sum two values and store the result in the variable on the left side of the operator. Knowing precisely all the different operators to count, we then need to check what is the token name reported in the Tree-Sitter Java grammar for these assignment operators. That way we can use those tokens in a *match* statement inside the *compute* function.

Branches are the most specific objects to count between the three metric components. For this component, we need to count two types of tokens in the input source code: method calls and new operators. Both these tokens are already part of the Tree-Sitter Java grammar inside RCA. So to integrate this counting into our implementation, we have to add these two tokens to our *match* statement and increment the branch count.

After counting assignments and branches, we can finally start implementing the conditions count. Again the condition counting rules are pretty straightforward to implement given their clear definition, at least for the most common and generic cases. First of all, we need to count any usage of a conditional operator. The paper considers a conditional operator every operator used to compare not boolean variables or numerical values such as == and > or < operators. There are also specific keywords that our implementation has to count as conditions. Some of those keywords are: *else*, *try*, *catch*, and *?*. No particular problems have occurred during the implementation of these first counting rules.

Until now, we have laid down the basic structure of the ABC metric implementation. Alongside implementing the first most trivial counting rules, we have defined a small set of unit tests to check the behaviour of the new module. That way, it is evident when the system is not behaving as expected after some changes. That, however, was not enough to consider the ABC metric implementation over. Code metrics should be precise and consistent and cover as many different scenarios as possible. So, after ensuring everything was working correctly by running the unit tests, we started tackling the corner cases of the ABC metric, such as constant declarations and unary conditional expressions.

When counting assignments at the beginning, we have purposely skipped a particular case specified in the paper. The paper's author states that any occurrence of an assignment operator must be counted, *except for constant declarations*. The author makes this decision because he does not consider constant assignments a data storage operations like variable and field assignments. We initialize constants once and never modify them in the code, so they do not count as data storage operations. To exclude constant declarations, we need a way to detect them in Java code. The Java language allows constant declarations through the usage of the *final* keyword. Any variable or class property that contains the *final* modifier is considered a constant. So how exactly can we count assignment operators without counting constant declarations by accident?

The solution we propose is a stack-based one. We define in the ABC module a new enum type called *DeclKind*, a short name for declaration kind. For this enum, we then define two types: *Var* e *Const*. Then we introduce a new field in the ABC *Stats* struct, an array of *DeclKind* called *declaration*. The *declaration* array is the stack data structure we can use to solve the constant declaration exclusion problem. In Java, we can define constants in lists, each constant declaration separated from the preceding by a comma, just like it happens for variable declarations. Those lists could be indefinitely long, so a stack approach has been chosen to keep track of the environment we are analysing without sacrificing memory and performance.

### Example

We can better explain this solution by looking at an example. Suppose we have this constant declaration inside a Java class:

```
1 final String MESSAGE = "Hello World!";
```

Tree-Sitter recognises this line as a field declaration or variable declaration, according to the position of the line in a source code file. In Java, we can declare a constant in two places. Inside a class by declaring a constant attribute or inside a method by declaring a local constant. According to our solution, each time we find a field or variable declaration, we push into the stack the *Var* value (**Figure 4.2**). The pushed value indicates we are currently analysing a variable or a constant declaration.

We can then proceed with the following nodes of the AST since now we have a way to remember in which environment we are. We stumble upon a *final* node, a child node of a *modifier* node. The *final* keyword plus the indication that we are in a variable declaration tells us one thing only: we have found the declaration of one or more constants. So we push the *Const* value in the stack and go on with the following nodes, keeping in mind that we are now inside one or more constant declarations (**Figure 4.3**).

Now that we have correctly set the environment, we can start looking for assignment operators. The last element of the stack acts like a memory of the environment we are currently analysing. We can use its value to decide if an assignment operator must be counted or ignored. Finally, when we stumble upon a semicolon, we know for sure that any variable or constant assignment is over, so we can safely empty the stack (**Figure 4.4**).

The advantage of this approach is that we use little memory. At the same time, we can inspect the nodes in order by keeping track of the needed information when they are available. By doing so, we can reduce the performance impact.

The next obstacle to the complete and comprehensive implementation of the ABC metric is the interpretation of the so-called unary expressions. That has been the most complex part of the implementation, despite the paper's author dedicating just a few lines to them at the end of the metric definition.

A *unary conditional expression* or *unary condition* is an implicit condition that uses no relational operators. This definition includes cases where we treat a single variable, field, or value as a boolean value. The ABC paper also reports, together

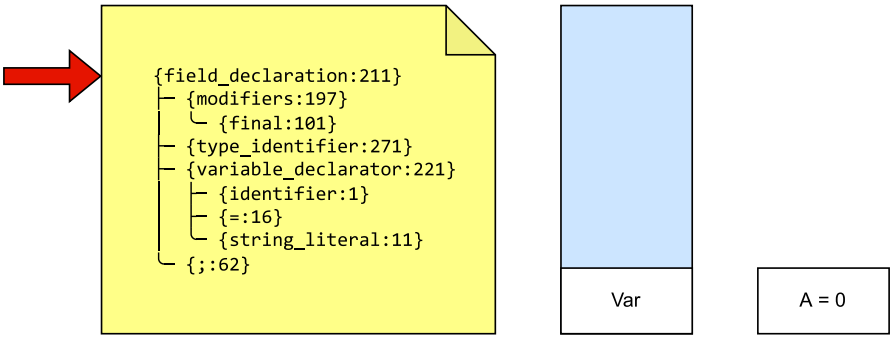


**Figure 4.2:** Declaration stack and assignments count at step 1.

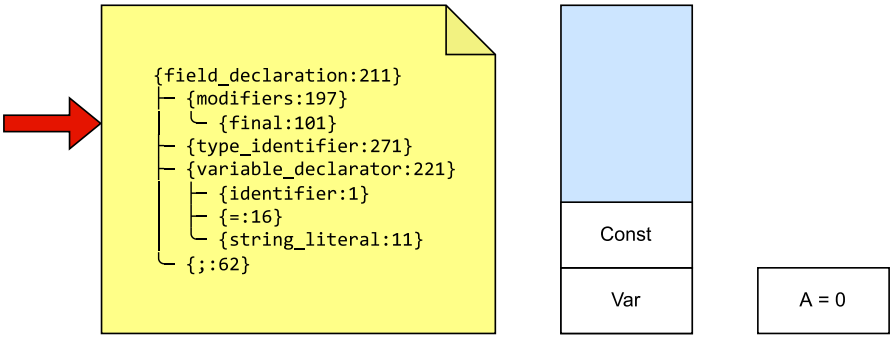


**Figure 4.3:** Declaration stack and assignments count at step 2.

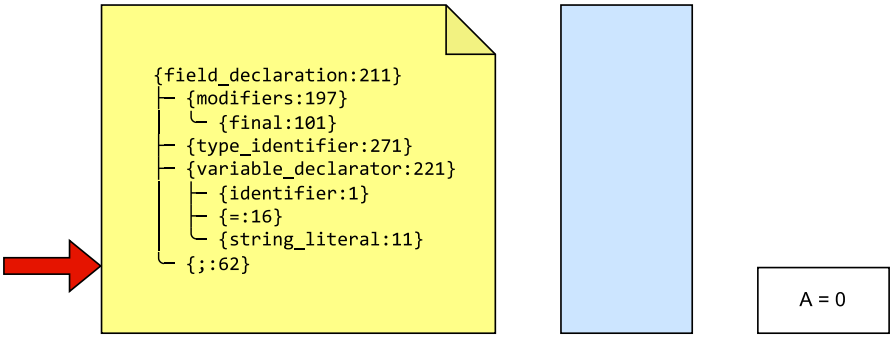


**Figure 4.4:** Declaration stack and assignments count at step 3.

with this brief definition, a small example to help us understand what a unary conditional expression is. Despite the snippet of code being in C++, we can translate it into the Java language, obtaining the following example:

```
1 if (x || y)
2     System.out.println("test failure");
```

This example shows how the variables $x$ and $y$ are both unary conditions since we test them as conditional expressions. Looking at this example, we get the idea of this type of condition. Unary conditions are boolean values placed in a specific place, making the execution flow take one path rather than another. GMetrics helps us understand more about that by providing us with a further example in its documentation online.

```
1 return !read;
```

Although the official paper does not mention this example, we now have proof that what we have assumed is correct. The return statement is just another place where unary conditions can appear. So after understanding that, we can start listing all the different locations where we could find a unary conditional expression to have a list of all the cases to cover. A unary condition can appear:

- In a *Binary Expression* using an *And* or an *Or* operator
- In the condition part of an *If* or *While* statement
- In the condition part of a *Do While* statement
- In the condition part of a *For* statement
- In the condition part of a *Ternary* expression
- As a return value in a *Return* statement
- As a return value in a *Lambda* expression
- As a value in an *Assignment* statement
- As a value in a *Variable Declaration* statement
- As an argument in a *Method Invocation*

As we can see, the first elements of the list are a direct consequence of the paper's definition. In addition to the official document, we have noticed that we can surround conditions in conditional statements with round parenthesis in Java. So

we must define a way to explore parenthesis to find and count any unary condition. Moreover, our exploration should be able to recognize normal conditions and avoid double counting them. This double count may happen in *binary expressions*, expressions composed of a couple of elements separated by an operator. If the operator is boolean, the operands can be two unary expressions. To understand that, we inspect the operands and any surrounding parenthesis until we find the operand value. Finally, we verify if the content found can be proven to be boolean.

The middle part of the list is instead about the return statements, an addition proposed by the GMetrics tool that we have decided to adopt and count. It is interesting to notice that not all returned values are eligible for being counted as unary conditions. Only boolean return values are considered unary conditions by our implementation. How can we understand a return value type without runtime information? We have solved this issue by only counting return values that can prove to be boolean by their location. So values we can find next to an *And* or an *Or* operator or instead behind a *Not* operator. So, for clarity, with these assumptions, we can count something like the following line as a unary condition.

```
1 return !x;
```

Instead, we can avoid counting as unary conditional expressions the following line.

```
1 return x;
```

In this case, we do not increment the condition counter because we cannot know the type of the variable $x$, relying only on static analysis. We could keep track of every variable type declared inside the file, storing the information in a collection. However, that would be very complex work for a static analyzer, and it would need many resources. That is why dynamic analysis exists, to simplify this kind of analysis. Our choice is the best solution we can achieve using just static analysis. Using this method, we do not reach the precision of dynamic analysis, but we can add a small piece of information to our measures and make them more precise.

The last elements of the list are cases not directly addressed by any documentation. We have derived them from our analysis of the GMetrics test cases and by reasoning about what a unary condition is. A *unary conditional expression* is an unknown implicit condition. It highlights a behaviour dependent on the content of a variable only known at runtime. Reflecting on this definition, we have decided to also include assignment operations in our list. We can assign a unary condition to a variable in a variable declaration or even pass a unary condition as a method argument. In these cases, the execution can change according to the value of the

condition at runtime. These scenarios fit our definition, so we have decided to cover them. Although we do not know the value of the condition, since we cannot act at runtime, we can still consider this static information by including them in our counts. We have reported a few examples of these cases in the following lines.

```
1 boolean x = !(true);
2 y = !!x;
3 m(!false);
```

The list of places where a unary condition can reside has helped us build the skeleton of the implementation in the *compute* function. We can directly translate each element in the list into a case in the match statement, so when each environment is detected, the research to find unary expressions starts accordingly.

To manage Java's indefinite number of parenthesis that can surround expressions, we have defined two functions. The functions implement the research of unary conditions inside the places in the code where unary conditions can reside.

We have called the first function *java_count_unary_conditions*, since it is a function for now necessary just for the Java language. The function inspects a list of elements and counts any unary conditional expression found in the list. For a list of elements, we intend two specific syntactic nodes where we can find an indefinite number of unary conditions. These two particular nodes are a binary expression using a boolean operator and a list of method arguments. The functions get as parameters the node list, which children could be a unary condition, and the condition counter to update. The function then loops over the list node children. Only children nodes of a *BinaryExpression* that are of kind *MethodInvocation*, *Identifier*, *True* or *False* though, are counted as unary conditions. The rest of the children are further inspected by the second function: *java_inspect_container*.

The objective of *java_inspect_container* is to inspect the content of Java *parenthesized expressions* and *Not* operators to find unary conditional expressions. The function takes as parameters a container node and the conditions counter. The first thing the function does is to initialize a flag called *has_boolean_content*. The flag is set to *True* if the parent of the container node is known to contain a boolean value. That happens when the container node is found inside a *binary expression* using a boolean operator and inside the condition part of *if*, *while*, *do-while*, *for* statements, and *ternary expressions*. Then the function defines a loop whose goal is to inspect the content of *parenthesized expressions* and *Not* operators. The loop is stopped when the child of the current node is not a *parenthesized expression* or a *Not* operator. If a *Not* operator is found the *has_boolean_content* is updated

and set to true. As a final test in the loop, the child node of the *parenthesized expression* or the *Not* operator is checked. If the child node is of kind *MethodInvocation*, *Identifier*, *True* or *False*, we have reached the end of the loop. If the flag *has_boolean_content* is true, we consider the element as a unary condition and update the conditions counter. In any case, the loop is stopped since it reached its end.

We have reported some code snippets of both these important functions below. We used the two functions a lot to achieve the expected behaviour for our ABC implementation. We have called *java_count_unary_conditions* any time a node of kind *BinaryExpression* using a boolean operator and a node of kind *ArgumentList*, representing the arguments of methods was found. We instead have called the *java_inspect_container* in many more places than the first function. The function has proved useful in condition parts of conditional and *return* statements and assignment operations. Particular care has been given to *for* statements and for *ternary expressions*. *For* statements in Java have a flexible syntax, while *ternary expressions* can implicitly return values that may contain unary conditions.

Below we have shown the condition for which we detect a unary condition inside the *java_count_unary_conditions* function. As we can see, we make great use of the *matches!* macro, defined to compare node kinds and Tree-Sitter tokens.

```
// Checks if the node is a unary condition
if matches!(node_kind, MethodInvocation | Identifier | True | False)
    && matches!(list_kind, BinaryExpression)
    && !matches!(list_kind, ArgumentList)
{
    *conditions += 1.;
} else {
    // Checks if the node is a unary condition container
    java_inspect_container(&Node::new(node), conditions);
}
```

This code instead shows the exit condition for the exploration that takes place in the *java_inspect_container* function. In this case, the *has_boolean_content* flags help us understand if we have stumbled upon a real unary condition.

```
// Stops the exploration when the content is found
if matches!(node_kind, MethodInvocation | Identifier | True | False)
{
    if has_boolean_content {
        *conditions += 1.;
    }
    break;
}
```

Once the *compute* function is completed, most of the work is over. As a final step, we need to decorate the *Stats* struct with the necessary methods and fields and compute the ABC magnitude based on the counters we have set. As we can see from the output below, for each ABC metric Stats, we have decided to report a set of measures: assignments, branches, and conditions counters, the magnitude of the vector composed by the counters, computed with the formula $sqrt(A^2 + B^2 + C^2)$ and a series of cumulative measures. The cumulative measures are the minimum, maximum and average, and we have computed them by considering all the inner spaces of the current measured space.

```
"abc": {
    "assignments": ...,
    "branches": ...,
    "conditions": ...,
    "magnitude": ...,
    "assignments_average": ...,
    "branches_average": ...,
    "conditions_average": ...,
    "assignments_min": ...,
    "assignments_max": ...,
    "branches_min": ...,
    "branches_max": ...,
    "conditions_min": ...,
    "conditions_max": ...
}
```

## 4.3 Object-oriented metrics

Object-oriented metrics are a kind of metrics that have never been implemented in RCA, despite them being highly used and valuable in source code development and maintenance. So, as a first step, it has been necessary to design a system to support this new metric type.

In particular, the issue with these kinds of metrics is that they are, as the name suggests, class-level metrics. RCA can already detect classes in a source code file using the spaces abstraction: classes are reported as spaces of type class in a JSON file. To implement object-oriented metrics, we need a way to show the Stats struct of a metric just for spaces of class type since the metrics have no meaning for other kinds. The new feature should fit in RCA perfectly, minimizing the impact on the library.

The solution we have developed for this problem uses the already discussed *Serde*, a library for serializing and deserializing Rust data structures. By exploiting *Serde* field attributes[46] we have a way to serialize a data structure conditionally.

Attributes[47] in Rust are a method to annotate different kinds of items. When, for example, a module, a function, or a struct is annotated with an attribute, they acquire a specific behaviour. An example of an attribute is *#[test]*, which is used to annotate test functions. Annotations allow the Rust language to apply metadata to the code[48].

The Serde framework provides us with an attribute called *skip_serializing_if* with the following syntax:

```
#[serde(skip_serializing_if = "path")]
```

The *path* string is a callable function that must return a boolean value. The function provided is usually a method of type of data on which the attribute is applied. The attribute provides a way to conditionally serialize and deserialize a field in the nested spaces structs. However, we need to specify the Stats structs that must be conditionally shown. Moreover, we must also define a method inside the Stats implementation that specifies the condition for which the structure must not appear.

Coupled with the class mechanism implementation, we have introduced another small but noteworthy change in the library. RCA contains its syntax node definition inside a specific module: the *Node* module. The choice is made to abstract and detaches the nodes used inside RCA from the Tree-Sitter nodes. This detachment is handy to extend the functionalities of Tree-Sitter nodes with custom methods. It might also be important for a future where another parsing library could replace Tree-Sitter. Inside the *Node* module, we have added a small function called *children* that returns an iterator object over the children nodes of a node. The function allows us to work with the children nodes of a node in a functional style. Using the new custom *children* function, we can finally work with children nodes in a compact, functional and effective way. This way, we do not have to worry about memory management and have access to all the known functional features like *filtering*, *mapping*, and *reducing*.

### 4.3.1 Weighted Method per Class (WMC)

As we have seen in a previous chapter, the *WMC* is a metric that measures the complexity of a source code. From its definition, we can also notice that the metric is very similar to a metric already implemented in RCA: the Cyclomatic Complexity.

The WMC is the sum of the Cyclomatic Complexities of the methods defined inside a class. However, the two metrics have two notable differences that make the WMC implementation different from the Cyclomatic Complexity one:

1. The Weighted Methods per Class is a metric associated with classes. We cannot propagate its value to any outer spaces the same way as the Cyclomatic Complexity.

2. The Cyclomatic Complexity for a class takes into account all the complex code inside a class, including any complex class properties initialization. The Weighted Methods per Class only considers the complexity of the methods.

Since RCA can already compute the Cyclomatic Complexity of Java methods, the WMC *compute* function turned out to be straightforward and short, with some differences from the other metrics. Since we already have all we need to calculate the WMC, we do not need to pass the current node to the function. We instead need to receive from outside the *SpaceKind* of the space we are currently analysing alongside the Cyclomatic Complexity Stat for the space and, as usual, the Stats of the WMC metric itself.

It is significant to notice that the *compute* function for the WMC is not invoked for all the syntax tree nodes like most of the metrics in RCA. The function is instead called at a higher level when the node level metrics are already computed, and the retrieved values only need to be finalized. We must invoke the function together with the Halstead metric finalization and the MI computation during the merge of the spaces in the *finalize* function.

Once made that clear, we have defined the *compute* to be a *switch*, or *match* in Rust, over the *SpaceKind* parameter. Any time a space kind of type *Function*, *Class*, *Interface* or *Unit* is received as parameter the space kind is saved into a metrics *Stats* field, which is initialized for all *Stats* structs to the *Unknown SpaceKind*. This special field is the one we exploit for conditionally serializing and deserializing the WMC Stats struct. In addition, if the *SpaceKind* parameter is equal to *Function*, we collect the value of the Cyclomatic Complexity from the Stats struct received as a parameter.

The reason why the *compute* is so small is because the logic of the metric computation is split between the *compute* and the *merge* function implemented for the WMC Stats structure. The *merge* function contains the the WMC propagation logic. In the method, we have specified what to do every time we try to merge a space of kind *Function* into a space of kind *Class* or *Interface*. In those cases, we need to add to the WMC of the class or interface the Cyclomatic Complexity

stored in the Stats structure of the merging space of kind *Function*. This way, we can avoid propagating the WMC values of a space to any of its outer spaces and, by doing that, preserve the definition of the metric.

We have also added a *is_disabled* function to the WMC Stats struct. This function is responsible for providing the condition for which we should hide the WMC Stats should or not. Its implementation consists of a single check of the space kind field in the Stats struct. The *is_disabled* function returns true for Stats struct which refers to spaces of kind *Function* or *Unknown* so that the WMC metric is visible just for spaces of kind *Class*, *Interface* and *Unit*.

Since the metric implementation is quite different from other metric implementations, we have given particular care to writing a complete set of unit tests. The cases we have covered with our tests are the following:

1. **Single class**: This is the most common type of Java file, consisting of just one class definition.

2. **Multiple classes**: It is possible to define multiple top-level classes inside a Java source code file. The only restriction is that it can only be one public class, and the public class name must be equal to the file name.

3. **Nested inner classes**: Java allows programmers to define classes inside other classes. The WMC value is not propagated from an inner class to an outer class.

4. **Local inner classes**: Those are classes defined inside methods. In this case, the Cyclomatic Complexity of the method is also counted and is propagated to the outer class WMC.

5. **Anonymous inner classes**: If we implement an abstract class on the fly inside a method, we are then defining an anonymous inner class. This class type is the first old and convoluted attempt for the Java language to define disposable functions. Despite not being recommended and having Lambda functions available in a later version of the Java language, these classes are still widely used. So we must consider it for the WMC computation. RCA, at the moment, does not consider anonymous inner classes as spaces. Therefore, the whole class and the methods that implement it contribute to the outer method Cyclomatic Complexity. However, the anonymous inner class is not reported as a different space and therefore does not have its WMC.

6. **Lambda expressions**: Lambda expressions are an evolution of the anonymous inner classes. For this reason, they are treated similarly to them. They

contribute to the outer method Cyclomatic Complexity but do not have their own space and, therefore, their WMC.

7. **Single interface**: A standard Java interface.
8. **Multiple interfaces**: Like classes, in Java, we can define multiple interfaces in the same file.
9. **Nested inner interfaces**: A type of interface similar to the nested inner classes.
10. **Class inside an interface**: This case verifies the behaviour of the WMC propagation from a class space to an interface one.
11. **Interface inside a class**: A further test to verify the WMC propagation between spaces of different kinds, the inner one being an interface and the outer one being a class.

The WMC implementation is our first attempt to introduce class-level metric support to RCA. Overall the result we have obtained is relatively minimal in terms of impact on the RCA codebase. The solution also defines a strategy to implement more object-oriented metrics in RCA. To show the result obtained, we have reported below the WMC Stats struct, consisting of the measures divided per classes, interfaces and total.

```
"wmc": {
    "classes": ...,
    "interfaces": ...,
    "total": ...
}
```

### 4.3.2 Number of Public Methods (NPM)

The *NPM* is a class-level metric that counts the number of public methods in a space of kind *Class*, *Interface* and *Unit*. While the WMC implementation relies on the fact that we should carefully propagate the WMC throughout the outer spaces of an inner space, for the NPM, we do not consider these kinds of problems. NPM measures of an inner space in our implementation are safely propagated upwards to the outer spaces. In this way, for example, we can have NPM measures for classes that include the public methods defined in the class and all the public methods defined in any inner class. This choice is consistent with the other metrics already implemented in RCA and is not limiting. We can still deduce the number of public methods defined in a class simply by taking the

total NPM measure of the class and subtracting to it the sum of the NPM measures of all the immediate and not immediate child spaces of kind *Class* or *Interface*.

It is also necessary to notice that NPM measures have a peculiar behaviour in Java interfaces. As we have discovered on the Java Tutorial website, all methods defined by interfaces in Java are implicitly public. So, for the Java language, the NPM of interfaces always equals the number of methods defined in the interfaces. To clarify this distinction, we have decided to report different measures for classes and interfaces. The designed generic structure can also be helpful for languages that allow declaring interface methods as not public.

Alongside the Number of Public Methods, we have also decided to count the Number of Methods (NM) defined in classes. NM can be a useful complementary metric, and it does not cost RCA much effort to compute it together with the NPM. One could argue that the NOM metric inside RCA has already calculated this value. While the NOM metric acts on spaces of any kind, NM acts only on specific spaces. Moreover, we must think generically and not focus too much on Java. In the future, for example, we could extend RCA to support a new language for which methods and functions are two different things, like C++. The NOM metric will, in that case, report together functions and methods for each space, while the NPM metric will be able to detect just the methods and not the functions.

Different from the WMC but similarly to the ABC, the NPM *compute* function acts on every syntax node because any syntax node could be a public method definition. Any time we find a node of kind *ClassBody* or *InterfaceBody*, we start searching for any public methods defined in it. Using the functional approach, we filter the children nodes of any *ClassBody* node to find method declarations. We inspect any visibility modifiers for each method declaration to determine whether the method is public. When we find a new public method, we increment the NPM counter. After the method declaration inspection, we update the NM counter by adding all the new methods inspected. For the interface, the behaviour is similar but less complex. In this case, we have to count the method declarations since any method declared inside an interface is implicitly public. We then add the resulting count to the NPM and NM values.

Another noticeable difference with the WMC metric is how NPM defines its *is_disabled* function. In the NPM *compute* function we have no way to save the current space kind. What we do at the beginning of the *compute* function is to check if the current syntax node is of kind *Program*, *ClassDeclaration* or *InterfaceDeclaration*. If we find a node of one of these types, we set a flag called *is_class_space* inside the NPM Stats struct to true, so we have a way to activate the

Stats visualization. The *is_disabled* function then just returns the *is_class_space* flag value, applying to it a not operation.

The final result is a Stats structure that gives a lot of new information to programmers. For a file and all the classes defined in it, we can retrieve both the Number of Public Methods and the Number of Methods. The measures are opportunely separated for classes and interfaces and also merged into a single total measure.

```
"npm": {
    "classes": ...,
    "interfaces": ...,
    "class_methods": ...,
    "interface_methods": ...,
    "total": ...,
    "total_methods": ...
}
```

### 4.3.3 Number of Public Attributes (NPA)

The NPA metric counts the number of public attributes defined in a class. This metric implementation presents many similarities to the NPM metric. As the NPM, we can safely propagate the NPA metric values from a nested space to its outer spaces without problems. Like the NPM, we have split the measures into classes and methods. Similarly to interface methods, interface attributes defined in Java are implicitly public. Another similarity with the NPM is the *is_disabled* function implementation: the *is_disabled* function for the NPA is practically identical to the same function implemented for the NPM metric. Again, similarly to the NPM, we have decided to count also the total Number of Attributes (NA) alongside the Number of Public Attributes.

The *compute* implementation for the NPA metric is very similar to the NPM one with some important differences. Again we have to inspect every single node that RCA provides us and update the flag to activate the Stats visualization for the current space. Even in this case, we can distinguish two behaviours: one for nodes of kind *ClassBody* and one for nodes of kind *InterfaceBody*. For each children node of type *FieldDeclaration* we have to count its children of type *VariableDeclarator*. That gives us the number of attributes of the class since we can define multiple fields in a single field declaration in Java. Once done, we have to inspect the modifier of each field declaration to know if the defined fields are public or not. If we find a public modifier, the NPM measure is updated accordingly. For the interfaces, the implementation is similar but less complex than the NPM. We only have to

count *VariableDeclarator* inside *ConstantDeclaration* and add that value both to the Number of Public Attributes and the Number of Attributes. The counting is simplified since Java interface attributes are implicitly public, static and final and considered constants. Despite the attributes of interfaces always being public, we decided to report interface measures separated from class measures. We have made this distinction for the same reasons discussed in the NPM implementation section. We want our implementation to be as general as possible and consider a future implementation in other languages. In C++, for example, interfaces can declare non-public attributes.

Again the final result is a Stats struct very similar to the NPM one. Measures are reported and separated into classes and interfaces. Alongside the NPA measures, we have also reported the total Number of Attributes. On the bottom part of the Stats then, we have added two total measures for both classes and interfaces.

```
"npa": {
    "classes": ...,
    "interfaces": ...,
    "class_attributes": ..,
    "interface_attributes": ...,
    "total": ...,
    "total_attributes": ...
}
```

## 4.4 Security metrics

Security metrics are the most experimental metrics added to RCA. Some do not even take their definition directly from a paper, but instead, they take inspiration from existing definitions, adapting them to provide practical new measures.
These kinds of metrics aim to show and highlight potential threats to security in a codebase. They are able to do that by collecting measures about strange behaviours in a source code, like, for example, a class with all its attributes public.

In our case, the effort to implement these metrics has been minimal since we have integrated them into some of the existing metric Stats structs, extending existing metrics. For some of these metrics, the experience gained by computing averages has been handy.

### 4.4.1 Class Operation Accessibility (COA)

The *Class Operation Accessibility* (*COA*) metric is an adaptation of the *Classified Operation Accessibility* (also abbreviated with *COA*) metric for not classified methods. The idea behind this adaptation is that in everyday source code, it is not much common to find classified methods. By adapting the definition to not classified methods, we are attempting to make the metric more beneficial for the average programmer and more mainstream. The Classified Operation Accessibility is computed by dividing the number of classified public methods by the number of classified methods of a class. A classified method is a method that interacts with at least one classified attribute. A classified Attribute is an attribute that is defined in UMLsec as secrecy. Our COA metric, which we have called Class Operation Accessibility, is instead defined as the number of public methods of a class divided by the total number of methods of the class.

To implement our COA metric, we have added three more average measures inside the NPM Stats struct. In this way, we have avoided introducing a new small module, and we have collected all the information about class and interface methods in one place. The final NPM Stats struct obtained is the following:

```
"npm": {
    "classes": ...,
    "interfaces": ...,
    "class_methods": ...,
    "interface_methods": ...,
    "classes_average": ...,
    "interfaces_average": ...,
    "total": ...,
    "total_methods": ...,
    "average": ...
}
```

### 4.4.2 Class Data Accessibility (CDA)

The *Class Data Accessibility* (*CDA*) metric is an adaptation of the *Classified Class Data Accessibility* (*CCDA*) metric for not classified attributes. Similarly to what we have explained above for the Class Operation Accessibility, the purpose of this adaptation is to make the metric more available and usable to the common programmer. Most developers have never used UMLsec and do not know what a confidential attribute is. For this reason, we have defined the CDA metric as the number of public attributes of a class divided by the total number of attributes of the class. This way, we have removed any reference to confidential items.

Just like the Class Data Accessibility metric, we already have all the ingredients to compute this new metric inside the NPA Stats struct. We obtain that by dividing the number of public attributes by the number of attributes defined in a class or an interface. The result is an extended and improved version of the NPA Stats struct, which includes three more security measures. Developers can use those measures to discover potential security threats in a codebase.

```
"npa": {
    "classes": ...,
    "interfaces": ...,
    "class_attributes": ..,
    "interface_attributes": ...,
    "classes_average": ...,
    "interfaces_average": ...,
    "total": ...,
    "total_attributes": ...,
    "average": ...
}
```

# Chapter 5

# Metrics analysis

## 5.1 Analysis process

The ultimate objective of a software metric is to give information about the measured attribute. At this stage of our work, we have a working implementation of several code metrics, each one with different aims. Size metrics' objective, for example, is to give information about the quantity of programming code produced by a team. Object-oriented metrics tell us how well-designed our project is according to object-oriented principles. Finally, security metrics provide reliability measures and highlight potential security flaws and dangerous threats in the code.

In this chapter, we apply our metrics implementation to real maintained codebases. The objective is to collect actual data and observe the behaviour of static metrics of different categories, highlighting the advantages and disadvantages of their usage in a working environment. To achieve that, we have represented the collected data using different graphs. All the work discussed in this chapter is publicly available in our GitHub repository[49], so anyone can inspect it and reproduce our results.

We have divided the data analysis and graphs production work into two studies: spatial and temporal analysis. Each of these studies focuses on specific sets of measures and represents a possible usage of the implemented metrics. In the spatial analysis, we inspect the measures of repositories at fixed versions. The objective is to produce a snapshot of the codebases at a specific time so we can compare and evaluate the metric values of different projects. For the temporal analysis, we consider how the measures for the repositories evolve in time. We produce graphs that follow the evolution of the metrics for different project versions. This way, we can see when significant events happened and predict how the projects will change over time.

### 5.1.1 Selected repositories

For the data collection phase, we have selected four Java code repositories of different sizes for each study: one very-large, one large, two medium and two small, for a total of six repositories. Collecting data from thousands of projects classified by size is a systematic methodology to validate a thesis in much bigger studies. We have chosen repositories of four different size categories to replicate what researchers usually do in those studies on a much smaller scale (**Table 5.1**).

| Name | Size | Version | Java Files |
|---|---|---|---|
| mockito | Very large | 4.7.0 | 949 |
| spring-kafka | Large | 2.9.0 | 502 |
| gson | Medium | 2.9.1 | 218 |
| Java-WebSocket | Medium | 1.5.3 | 175 |
| java-jwt | Small | 4.0.0 | 75 |
| FastCSV | Small | 2.2.0 | 39 |

**Table 5.1:** Repositories selected for the analysis.

Since the *FastCSV* project does not have at least ten different versions to measure at the moment of the study, we have decided to replace it with the *java-jwt* project in the temporal analysis. We have also made a similar substitution for the *Java-WebSocket* repository in the temporal analysis. By replacing the repository measure with the *gson* project ones, we increase the number of repositories we analyze and inspect. These two replacements help us achieve a more comprehensive vision of both medium and small repositories, adding two more repositories to our studies.

For our studies, we want to analyze Java repositories which are actually used and actively maintained. Luckily the Maven Community provide access to the most used Java projects online from the *Maven Central Repository*[50]. *Maven*[51] is a build automation tool developed by the Apache foundation, which helps programmers manage project dependency and build their projects faster. If a Maven project needs a repository as a dependency and the build automation tool does not find it locally, usually the dependency is searched online in the Maven Central Repository. Similarly, *Gradle*[52], another popular build automation tool used especially in Android projects, uses the Maven Central Repository as a place to find missing dependencies. We can conclude that the Maven Central Repository is the perfect place to conduct our repository research. So we have selected some of the most used repositories that meet our size criteria directly from there. Our selection results in a list of modern repositories with various application fields, used by developers and working applications every day worldwide.

*FastCSV*[53] is a small library for reading and writing CSV files. The library is known to be very fast, does not need any dependency to work, and is very reliable. Programmers can use FastCSV for big data applications that need high performance and small applications that need a lightweight solution. The library is based on the CSV standard but allows specific configurations to some extent.

*Java JWT*[54] is an implementation of the JSON Web Token[55] (JWT) standard for the Java language. The library allows programmers to sign and verify a JWT and supports many different encryption algorithms. JWT is an open standard that defines a compact way to exchange data between parties in a network as a JSON object. Before being transmitted, we digitally sign the JSON object using a secret or a key so that the data can be verified and trusted by the receiver. Modern applications use JWTs every day for authentication and information exchange. An example of the usage of JWTs is the Single Sign-On feature. This feature exploits the JWTs interoperability and small overhead to allow users to log in once to access different software systems.

*Gson*[56] is a JSON parsing library developed internally by Google and released in 2008. It allows the serialization of Java objects into their JSON representation and vice versa. Gson is different from other JSON parsing libraries because it allows the serialization of Java classes without needing any annotations and even supports generic types. In the project documentation on GitHub, Google developers state that they will probably no longer add significant new features to the library. However, the developers still actively maintain the codebase, constantly fix bugs, and discuss requests for new features using GitHub issues.

*Java WebSockets*[57] is a repository containing a basic implementation of a WebSocket server client. WebSocket[58] is a protocol that defines a full-duplex communication channel over a single TCP connection. The protocol can work side-by-side with HTTP, even sharing its ports with HTTP. After a handshake, we can transmit messages without size limits between the parties and any party can close the channel. We commonly use WebSockets for highly interactive applications, like chats, games and browsers. In these cases, other types of interactions, like HTTP, can be inefficient and introduce significant latencies.

*Spring for Apache Kafka*[59] is a Spring project that provides a template for a Kafka-based messaging solution[60]. Spring[61] is an open-source framework for building Java applications. The framework helps programmers to develop enterprise web applications by providing a vast library of modules for any need. Apache Kafka[62] is a message broker, a component responsible for validating, transforming and routing messages among applications or microservices. Thousands of banks

and companies use Apache Kafka daily for its high scalability, fault-tolerance and support for mission-critical applications. Spring for Apache Kafka includes high-level abstractions and usage examples of Apache Kafka in the Spring framework environment. Developers use the project as a reference to build their solutions.

*Mockito*[63] is a popular open-source testing framework. Mocking is a technique that allows the usage of simulated and controlled components, like objects, instead of real ones for testing purposes. By using this method, it is possible to produce readable and independent unit tests. The framework is one of the most used Java libraries, according to an analysis[64] made on ten thousand GitHub repositories in 2013. The framework is still widely used today, with over two thousand forks and thirteen thousand stars on GitHub.

### 5.1.2 Python language

Once we have selected the repositories, we can start collecting data from them. We get those data in JSON files from running RCA over all the selected codebases. However, the data produced are still raw and need further elaboration to give more understandable information and be useful for our analyses. Once the elaboration is complete, we can visualize the obtained information in graphs and tables so anyone can interpret the data visually.

We have decided to use the *Python* scripting language to automate the process of generating the JSON raw data from RCA and elaborate the data to get meaningful measures and graphs. Python[65] is a powerful high-level object-oriented programming language. The language is known for its elegant syntax, versatility and simplicity. The indentation is a fundamental part of the syntax and one of the most notable characteristics of the language. Data types are assigned dynamically, while a garbage collector automatically disposes of no more used variables. Python comes with an extensive standard library to satisfy many everyday programming needs, including the parsing of JSON files. Some of the most common language application includes: automating tasks or scripting, data elaboration and prototyping.

For graph generation, we have chosen a Python library called *Matplotlib*. Matplotlib[66] is a plotting library which provides API to create static, animated and interactive data visualizations. In particular, we have used a specific library module called PyPlot. Pyplot allows us to plot graphs very similar to the ones produced by the *MATLAB* platform, which are very recognizable in the scientific field. MATLAB is a proprietary programming language and a numerically computing environment. It allows matrix manipulations, plotting of functions and data and much more. Pyplot is a valid alternative to MATLAB for plotting graphs because it

is free, open-source, and can rely on the Python language. After we have generated the graphs, Matplotlib allows us to export them in the most common image formats.

We have organized our analysis into three Python scripts: *data-production.py*, *spatial-analysis.py* and *temporal-analysis.py*. In the *data-production.py* file, we have put the code necessary for downloading the repositories and run RCA over them to produce the raw JSON data files. In *spatial-analysis.py* and *temporal-analysis.py*, instead, we have inserted the code that takes the previously generated raw JSON files elaborates them according to the type of analysis and produces the corresponding analysis graphs. The Python scripts and the graphs are available in the GitHub repository we reported at the beginning of the chapter. In this way, anyone can reproduce the same graphs we analysed by executing the three Python scripts in the correct order.

## 5.2 Spatial analysis

In the spatial analysis, we visualize the measures of four selected repositories pinned at a specific version, the last one available. The spatial analysis is static because we measure repositories without considering their evolution in time. This analysis can provide information about a specific version of codebases by generating a report on the status of a software project according to the metrics we implemented.

To better organize the analysis, we have classified the plotted graphs into three categories: cumulative measures, metric comparisons and visibility measures.

### 5.2.1 Cumulative measures

This kind of graph represents for each selected repository three cumulative measures: sum, maximum and average. We have not measured the minimum because it is common to have in almost all repositories a file with the minimum value of a metric. Furthermore, low values often indicate normal behaviour for the metrics we have analysed. Higher values are the ones that may indicate problems. We obtain the cumulative measures for each repository by reading all the repository JSON files generated by RCA. In particular, we just read the measures reported in the first space, the most external one, which includes all the other spaces inside it: the Unit space. We compute the sum by summing together the measures read for each repository file. We derive the maximum by keeping track of the maximum value read for each repository. Finally, we compute the average by dividing the previously calculated sum by the number of files read.

To visualize the elaborated data clearly, we have reported the cumulative measures into three bar graphs, one for each cumulative value. This way, we can compare sums, maximums and averages for every repository into three distinct graphs. Each bar represents the cumulative measure we obtained for a single repository. In total, we have generated four graphs, one for each implemented metric so that we can inspect the behaviour of all the implemented metrics simultaneously.

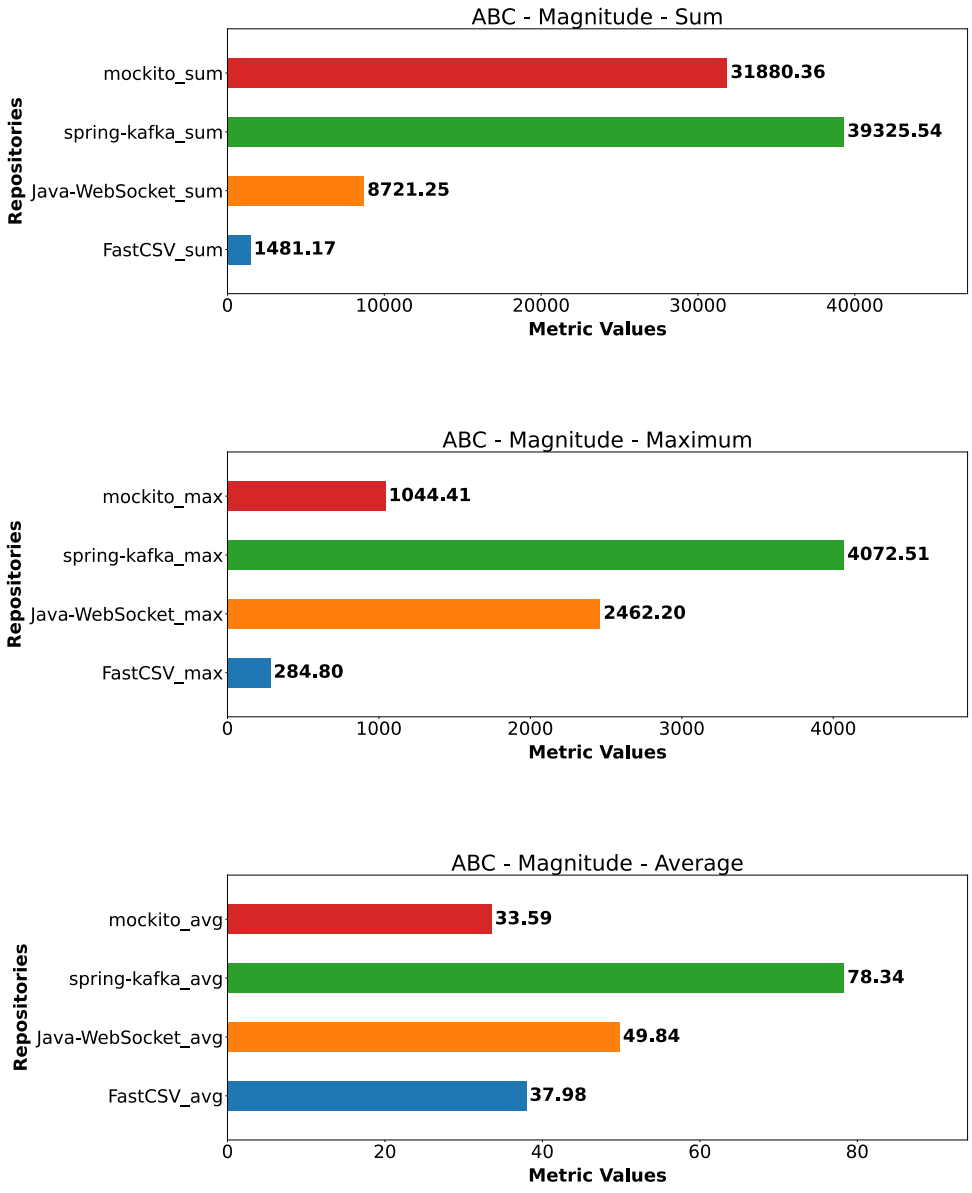


**Figure 5.1:** ABC cumulative measures.

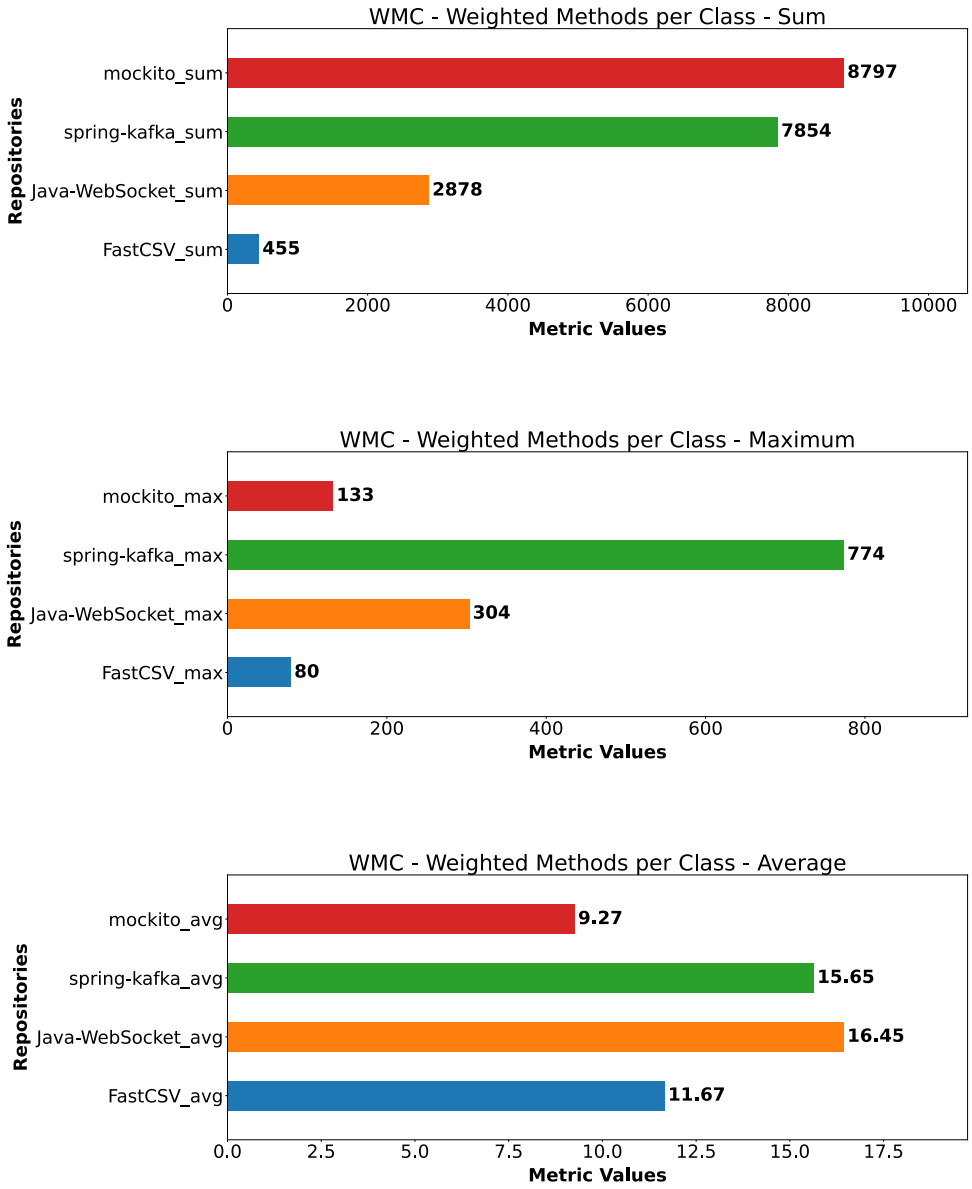


**Figure 5.2:** WMC cumulative measures.

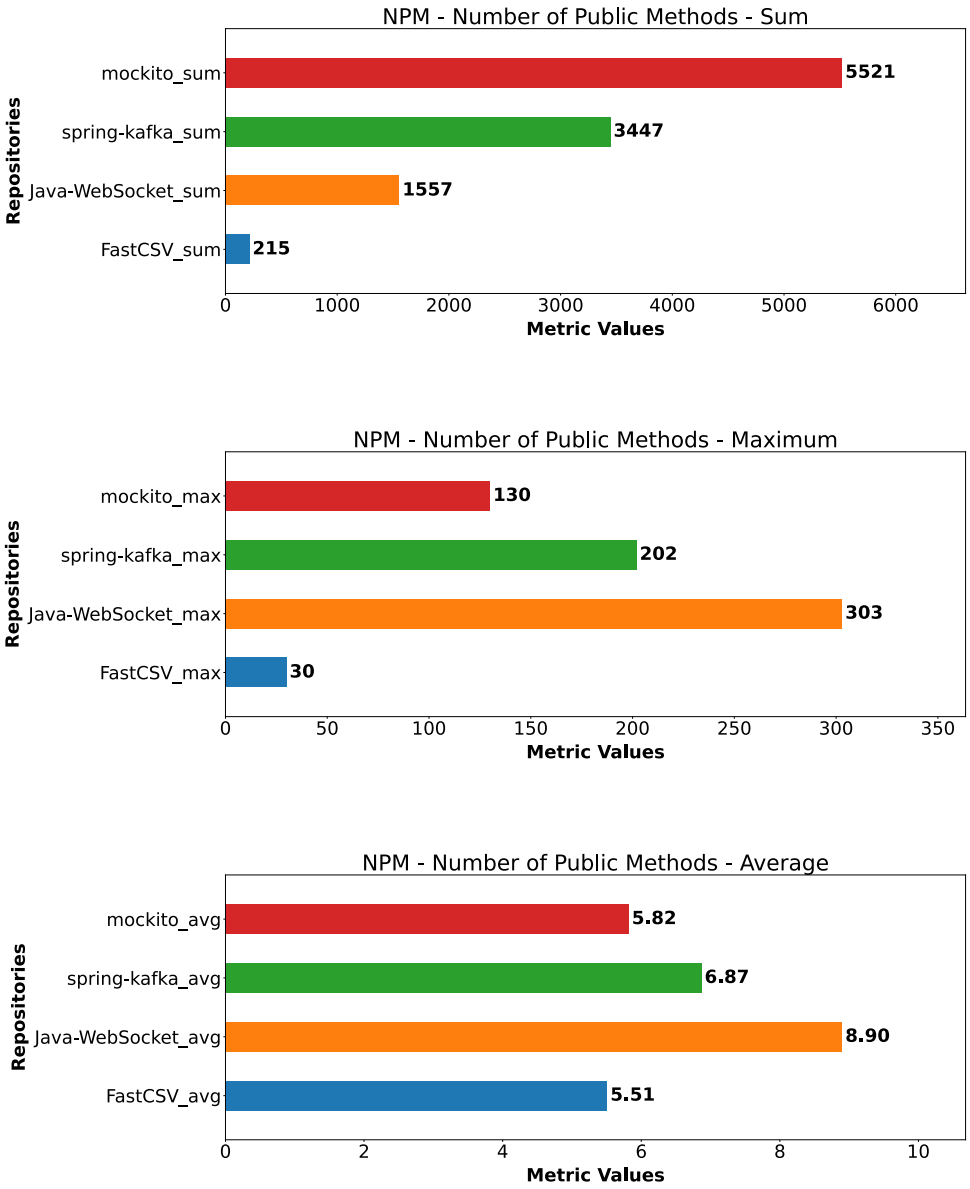


**Figure 5.3:** NPM cumulative measures.

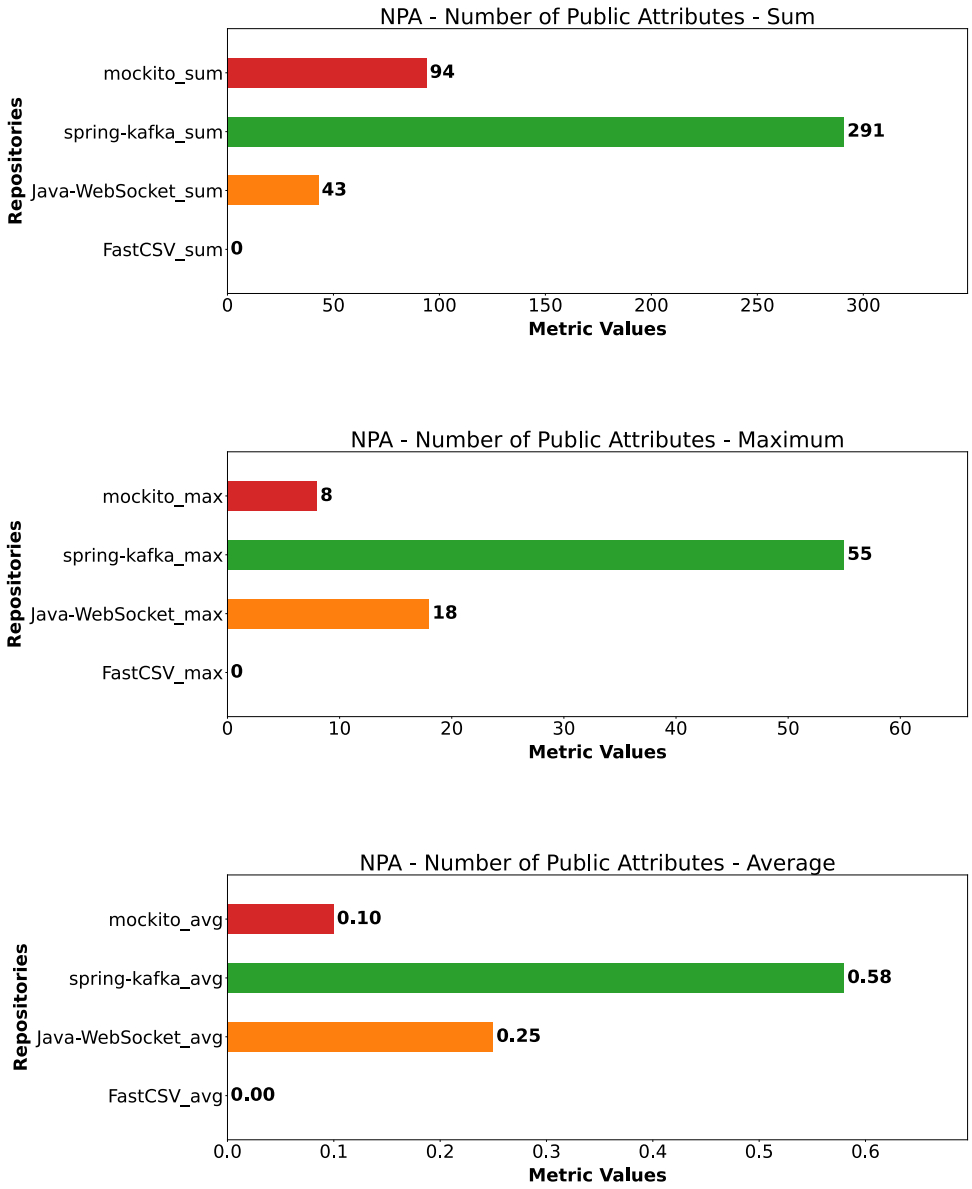


**Figure 5.4:** NPA cumulative measures.

From the ABC graphs (**Figure 5.1**), we can notice a few things. By looking at the three smallest repositories, we notice that the smaller the repository, the smaller its cumulative measures. However, this is not always the case because Mockito shows us that a large repository composed of almost a thousand source files can contain less code than a repository half its size. Moreover, Mockito code seems to be better distributed over its source files since the maximum and average values are minimal.

We can draw similar conclusions also by inspecting the WMC graphs (**Figure 5.2**). In this case, we can see that the Mockito project is the most complex project of the four analysed. However, its maximum and average values again show us that Mockito source files are generally less complex than the other repositories' source files. This consideration enforces the idea that despite being a large codebase, Mockito manages very well the content of its files, making them generally small and not complex.

When inspecting the NPM graphs (**Figure 5.3**), we can see that it seems normal to have total more significant values of NPM in larger repositories. Again the maximum and average values tell us more about the content of the singular source files. In this case, we notice that the bigger the repositories, the more attention is dedicated to reducing the number of public methods declared in singular files. The minor repository presents instead relatively low NPM values in the singular files.

Differently from the other graphs, the NPA graphs (**Figure 5.4**) show us low values. That means that all the repositories declare few public attributes, which is good from a security point of view because the object attributes visibility is somehow limited. The best performer is FastCSV; the minor project does not declare even one public attribute across all its files. Next, we have Mockito and Java WebSocket, seriously limiting their public attribute declarations. The worst performer is Spring for Apache Kafka, with the most significant cumulative values in all three graphs.

### 5.2.2 Metric comparisons

In this kind of bar chart, we represent cumulative sums for multiple metrics simultaneously. All the bars of the same colour show measures relative to the same repository. This kind of graph helps us understand the relationships between similar metrics or metrics with the same purpose.

The kind of metrics we want to compare are the size metrics and the complexity metrics. So, for this section, we produced just two graphs.

The size measures graph reports on a single bar chart cumulative sums for four different size metrics: Cyclomatic Complexity, Physical Lines Of Code, Halstead Estimated Program Length and ABC magnitude. It is essential to notice that Cyclomatic Complexity is not strictly a size metric, but since the rules for measuring it are similar to the ones used for the ABC metric, we have decided to include it in the graph.

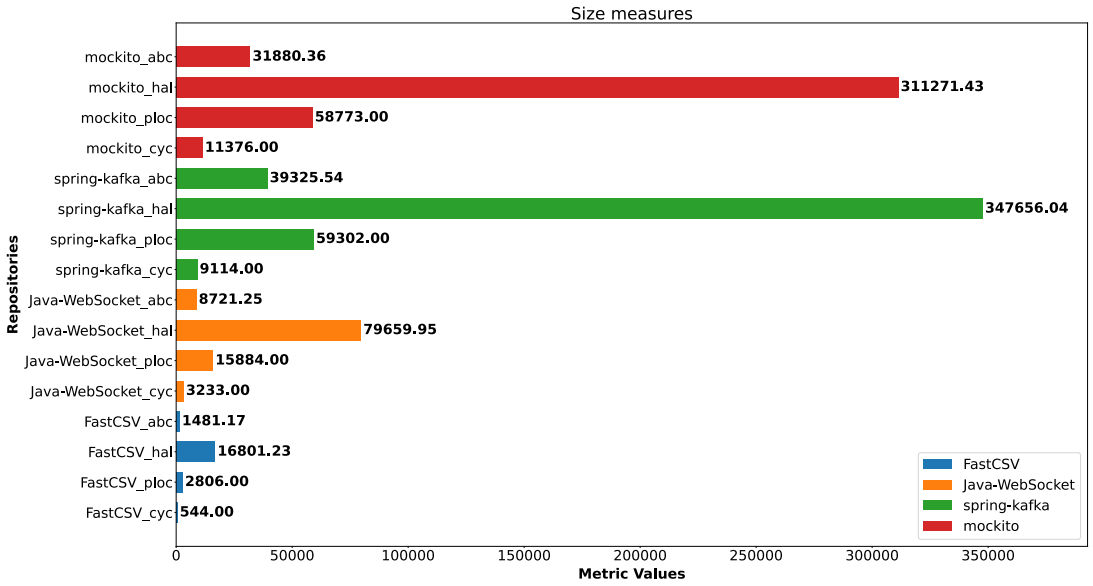


**Figure 5.5:** Size measures comparison.

Looking at the bar chart (**Figure 5.5**), we immediately notice that the cumulative sums for all four metrics follow a specific pattern. Little repositories report small metric values, while large repositories report significant ones. The only difference we can notice between the progress of the bars is their growth relative to the number of the repository source file. The Halstead Estimated Program Length seems greatly influenced by the number of source files; the Physical Lines Of Code and ABC magnitude follow. The Cyclomatic Complexity is almost not influenced by the number of project files.

The complexity measure graph reports cumulative sums on a single bar chart for two similar complexity metrics: the Cyclomatic Complexity and the Weighted Methods per Class.

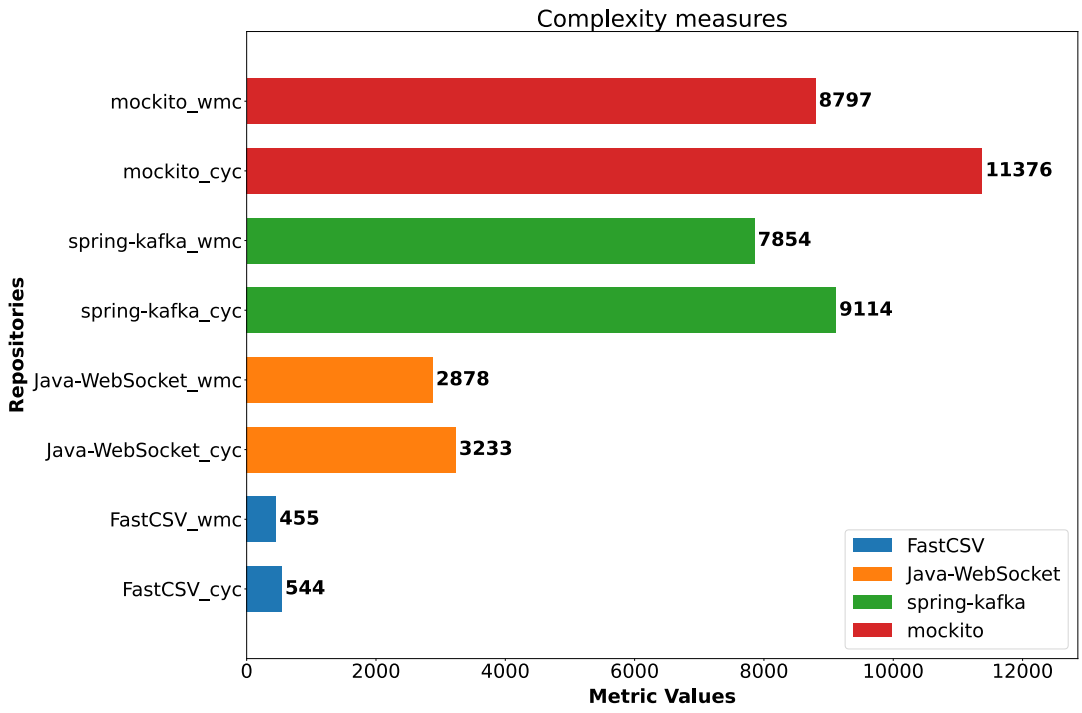


**Figure 5.6:** Complexity measures comparison.

From the bar chart (**Figure 5.6**), we notice from the bar chart that for each analysed repository, the value of the WMC cumulative sum is slightly lower than the Cyclomatic Complexity one. As we explained in the previous chapter, this happens because the WMC metrics do not consider the complexity derived from attribute initialisation, which may happen in a class body outside the class methods scope. Moreover, since Java considers Enums as classes in the latest versions of the language, we can also attribute some of the complexity the WMC lacks to any Enum class present in any source code files. By subtracting the Weighted Methods per Class sum of a repository to its Cyclomatic Complexity sum, we can retrieve the amount of complexity hidden in class attributes and Java Enums.

### 5.2.3 Visibility measures

For these last spatial analysis graphs, we have shown the number of methods and attributes, highlighting the ratio between the public and the non-public ones. We inspect these measures because we could consider too many public methods or attributes in a repository as a potential security vulnerability. In total, we have generated four graphs, divided into two categories: measures and percentages.

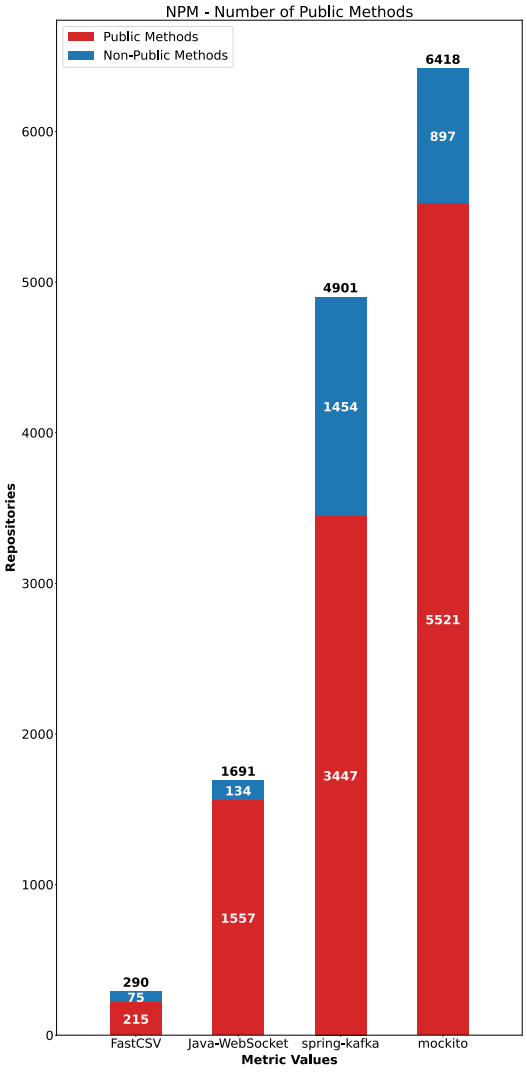


**Figure 5.7:** Number of methods.

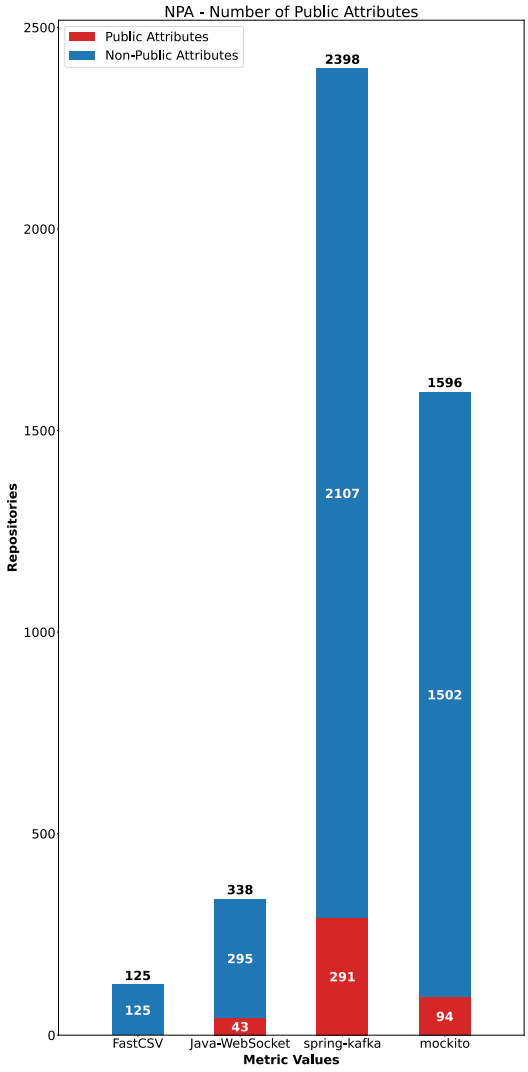


**Figure 5.8:** Number of attributes.

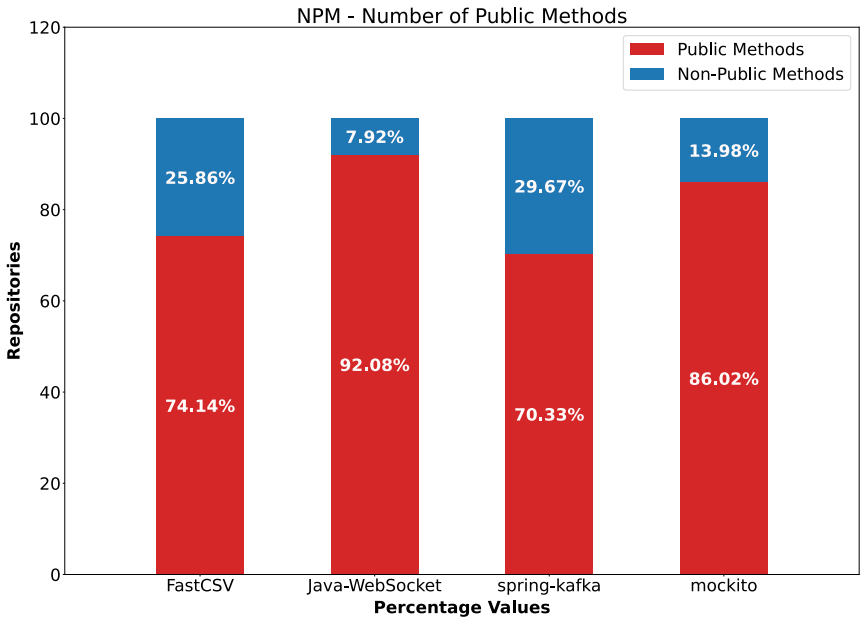


**Figure 5.9:** Percentages of methods.

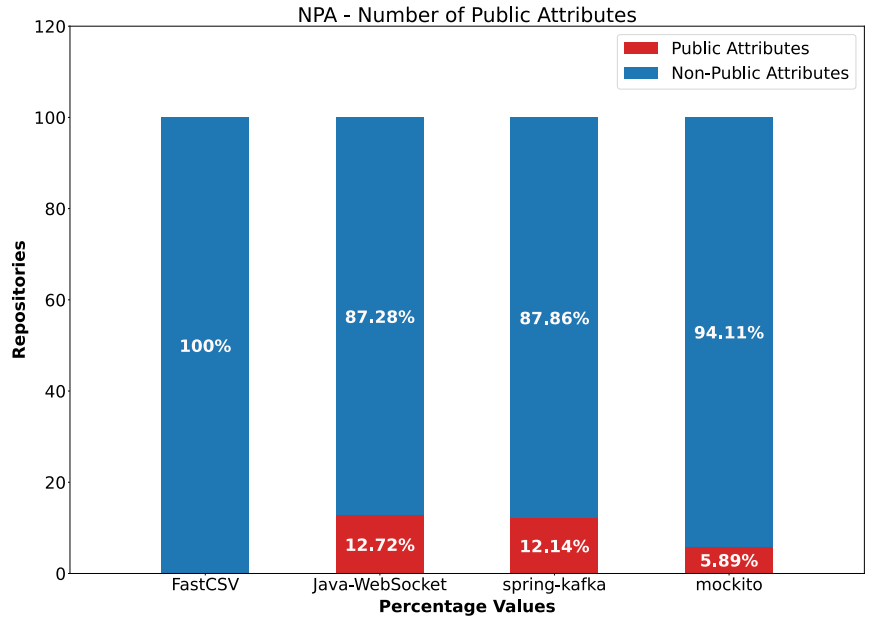


**Figure 5.10:** Percentages of attributes.

In the first category of graphs (**Figure 5.7** and **Figure 5.8**), we illustrate the total number of method and attribute declarations found in all the repositories. We have represented the number of declared methods and attributes for a repository as bars. We then divided each bar into two colours, red and blue. The red parts indicate the public declarations, while the blue ones show the remaining non-public declarations. Thanks to these graphs, we can see how the number of public and not public declarations varies across projects of different sizes.

In the second category of graphs (**Figure 5.9** and **Figure 5.10**), we illustrate the total number of method and attribute declarations found in all the repositories as percentages. In these graphs, as opposed to the previous ones, the bars have all the same height and represent the total number of method and attribute declarations. Each bar represents a repository, again divided into two colours. The red parts indicate the percentage of public declarations. In contrast, the blue ones show the remaining percentage of non-public declarations. These graphs help us see the number of public and non-public declarations as percentages of the total declarations.

From the graphs of the measures, we can start to notice several particular aspects. First of all, about the methods, we can see that the number of public methods declared in repositories is high compared to the number of non-public ones. For the attributes, instead, we notice something completely different. Each measured repository has reported a low value of public attributes compared to the non-public ones.

From the graphs of the percentages, we can instead discover further details that we can use to compare the projects. The percentage of public methods seems unrelated to the repository size. Both small and big repositories have various percentages of public methods. The graphs, however, confirm that the number of public methods declared by all repositories is very high for all the projects. For the attribute percentages, we can notice something more. The percentage of public attributes diminishes with the size of the repositories, except for the smallest one. We could assume that there is not much need for data sharing in small repositories with few modules, so public attributes are limited in these cases. Instead, medium and large projects have more modules, so the attention dedicated to the number of public attributes is higher. In these cases, the bigger the repository size, the lower the number of public attributes declared.

## 5.3 Temporal analysis

In the temporal analysis, we show the measures of four selected repositories at different versions in time. In particular, we have measured the last ten versions of the projects available at the moment of the analysis. This study can highlight positive and negative project changes and provide a report of a codebase status over time. One could even use this analysis to attempt to predict the behaviour of the repositories in the future and act in advance to stop bad changes.

We have represented the collected data with graphs and tables for this study. Again, we have divided the analysis into three categories: cumulative measures, files and classes rankings and metric thresholds.

### 5.3.1 Cumulative measures

For this first part of the temporal analysis study, we plotted two cumulative measures over time: the average and the maximum. The averages help us understand how all the overall project measures evolve. The maximums instead focus on the problems. We can see how the most critical files change by monitoring the files with the highest metric value over time for each repository. We have produced the measures using the same method adopted for the spatial analysis. This time, however, we have measured ten versions of each selected repository.

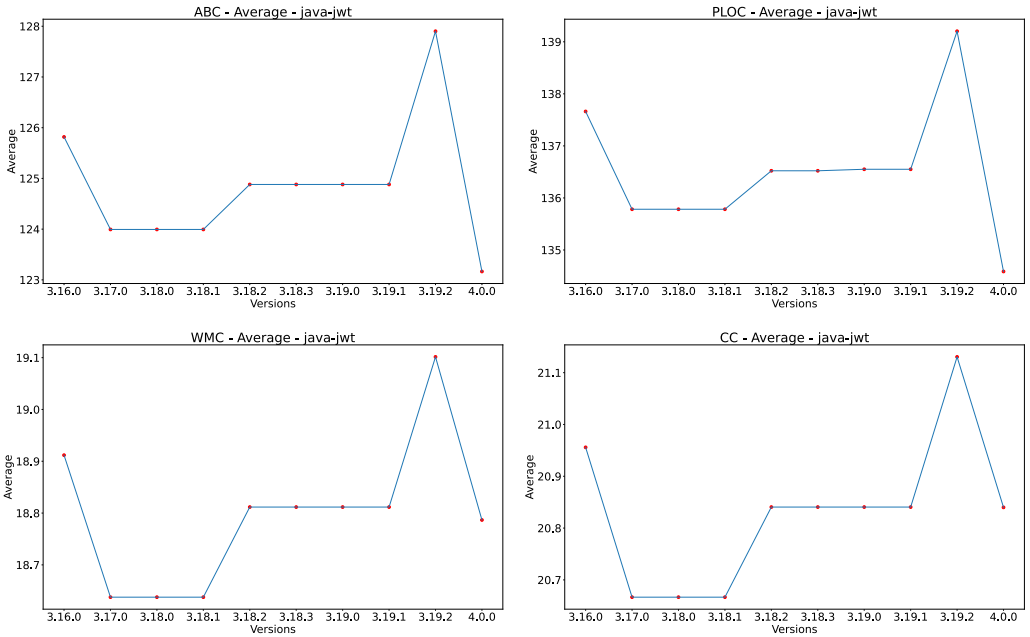


**Figure 5.11:** Java JWT average measures - Part 1.

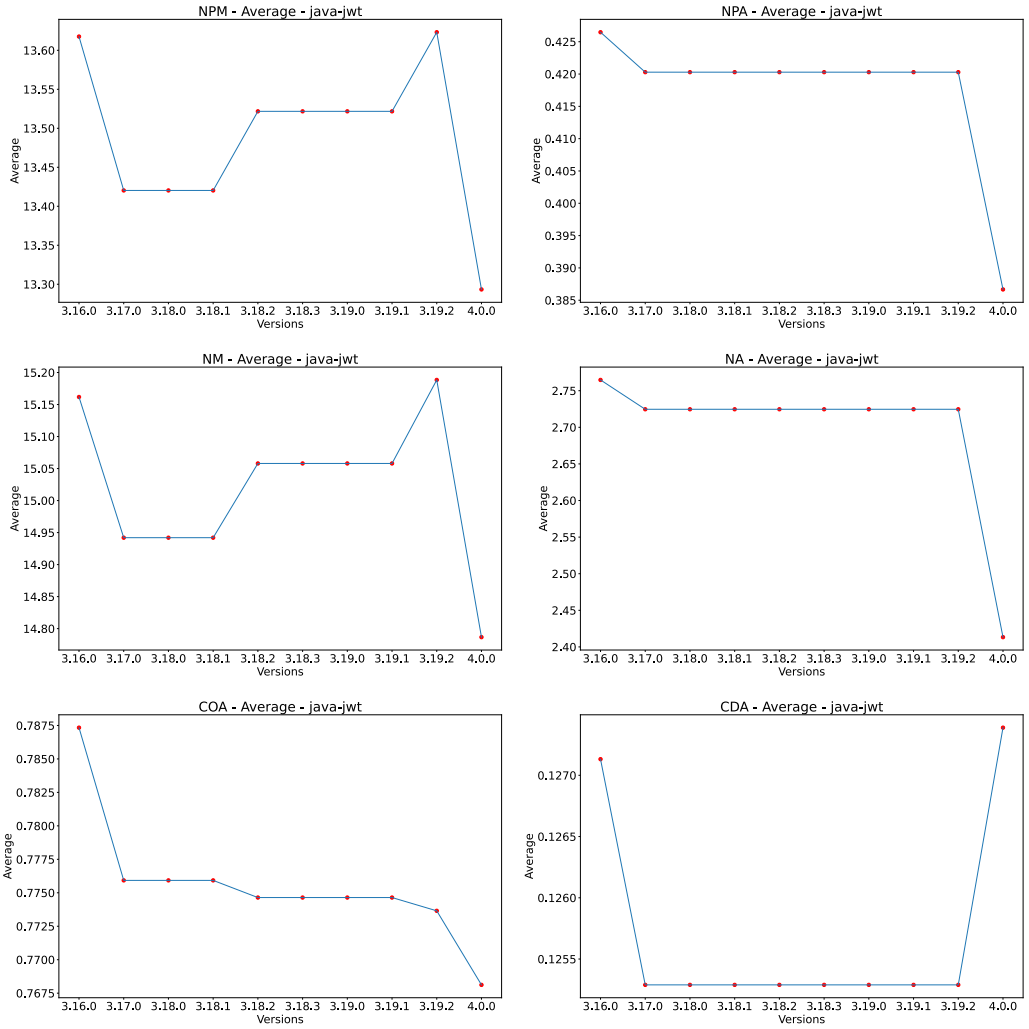


**Figure 5.12:** Java JWT average measures - Part 2.

We have represented both averages and maximums in line charts. We have generated two groups of charts for each cumulative measure and each repository. In the first group, we showed four size and complexity measures; the ABC, the PLOC, the WMC and the CC. In the second group, we have instead represented six visibility measures; the NPM, the NM, the COA, the NPA, the NA and the CDA. We have decided to call NM and NA the number of both public and non-public methods and attributes. The division helps the reader to distinguish between size and complexity graphs and visibility ones. We can use these graphs to compare the behaviour of metrics with similar purposes and possibly find correlations between metrics over time. We have reported above an example of this line charts (**Figure 5.11** and **Figure 5.12**).

In general, we can say that the averages over time for complexity measures follow a similar trend. We can say the same for the size metrics. Moreover, there is a certain resemblance between the complexity and size line charts for the small and medium repositories. Instead, the largest project presents noticeable differences between the size and complexity graphs.

In the visibility line charts for the averages, we can keep track of the number of declarations of methods and attributes over time. We have also decided to report the average values of COA and CDA over time to see if we can find any correlation between the measures. The number of public declarations and the total number of declarations somehow seem to influence the COA and CDA values. However, the correlation between the measures is unclear since COA and CDA are already average values. So we are showing averages of sums over files of measures that already are averages. For this reason, we cannot directly compute the COA and CDA values reported in the graphs by dividing the NPM and NPA measures by the NM and NA ones reported in the graphs above.

### 5.3.2 Files and classes rankings

For this second part of the temporal analysis, we have collected classes and files with the highest metric values and reported them into tables. A single table contains the ranking for a single project from the point of view of a single metric. In these tables, each row represents a file or class, while each column represents an interval of project versions. In the cells, we have reported the metric value for the file or class specified by the row for the project version specified by the column. If a cell is empty, the file or class has ceased to be one of the top three files or classes with the highest metric value for that specific interval of codebase versions. We have not collected measures for the COA and CDA metrics for this part of the study. We assumed many classes would have the maximum metric values. So it is pointless to show a ranking of classes with all identical maximum values.

We have chosen to report the four ranking tables for the Java JWT project in the following spaces, as they contain some curious results.

| File | v1 - v8 | v9 | v10 |
|---|---|---|---|
| ECDSAAlgorithmTest.java | 1085.87 | 1238.76 | 1129.75 |
| ECDSABouncyCastleProviderTests.java | 951.85 | 984.44 | |
| JWTCreatorTest.java | | | 918.20 |
| JWTVerifierTest.java | 887.37 | 887.37 | 1167.54 |

**Table 5.2:** Top 3 Java JWT files with highest ABC magnitude values.

The first table (**Table 5.2**) shows how the three files with the highest ABC magnitude have incremented their values in the ninth version analysed. We can also see that in the last version, the third file in the ranking reduces its size, so another file with a lower ABC magnitude value takes its place.

| Class | v1 - v8 | v9 | v10 |
|---|---|---|---|
| BaseVerification | | | 86 |
| ECDSAAlgorithmTest | 101 | 112 | 102 |
| ECDSABouncyCastleProviderTests | 87 | 87 | |
| JWTVerifierTest | 79 | 79 | 94 |

**Table 5.3:** Top 3 Java JWT classes with highest WMC values.

In this second table (**Table 5.3**), we observe similar behaviour for the WMC. Even for this metric, we can see that a class replaces another class in the rankings at the last version. In particular, we notice that the class replaced has the exact name of the file replaced in the table above. We can assume that the file contains the class of the same name. Project maintainers have refactored and improved the file in the last project version, making it less big and complex.

| Class | v1 - v8 | v9 | v10 |
|---|---|---|---|
| ECDSAAlgorithmTest | 84 | 91 | 81 |
| ECDSABouncyCastleProviderTests | 84 | 84 | 74 |
| JWTVerifierTest | 77 | 77 | 92 |

**Table 5.4:** Top 3 Java JWT classes with highest NPM values.

In the third table (**Table 5.4**), related to the NPM metric, we notice some classes we have already reported in the preceding tables. This detail may point to a correlation between the NPM of a class and its ABC magnitude and WMC. In this case, a high NPM value also means a high WMC for the class and, consequently, a high ABC magnitude for the file containing the class. Despite having just three classes in the rankings for all the versions, we notice that the ranking has changed internally. The codebase maintainers changed the classes in the last ten versions but have not changed the NPM of those classes drastically.

The fourth and last table (**Table 5.5**) shows classes not seen before in the preceding tables. From it, we can see that the project has just one class with an NPA value of more than one at the beginning. Moreover, from the data collection process, we noticed several classes with NPA equal to one in the project. The table shows just a few classes with NPA equal to one, reported in alphabetical order.

| Class | v1 - v9 | v10 |
|---|---|---|
| AlgorithmTest | 1 | 1 |
| BasicHeaderTest | 1 | |
| HeaderParams | | 4 |
| PublicClaims | 11 | |
| RegisteredClaims | | 7 |

**Table 5.5:** Top 3 Java JWT classes with highest NPA values.

### 5.3.3 Metric thresholds

For the last part of this temporal study, we have decided to analyse possible thresholds for the implemented metrics. Metrics thresholds have enormous applications. They can help developers to understand if a particular file or class reports normal or abnormal metrics values. With them, we can rate the quality of a codebase. So, to analyse this crucial aspect, we have designed two types of bar charts similar to the ones designed for the visibility measures in the spatial analysis.

We have represented the number of files or classes that exceed the chosen metric threshold in the first bar chart type and the percentages of files or classes that exceed the thresholds in the second one. This way, we can look at a codebase and evaluate them according to the selected thresholds. We have produced two bar graphs for each implemented metric and repository.

We have researched possible metric thresholds from scientific articles and static code analysers tool documentation. This way, we have set each threshold to values that have proved to work. We have chosen to use the worst possible value for the metrics for which we have not found possible thresholds.

Two blogs talk about the possible ABC magnitude thresholds in their articles. In the *Ten Percent Not Crap* blog[67], the author proposes the value of sixty as a threshold. Any file with an ABC magnitude above sixty needs particular attention and can be dangerous. The article specifies more granular thresholds for the metric reported by a static analysis tool for the Ruby language called *Flog*[68]. The article also cites a *Sonar ABC Metric Plugin*, a static analysis tool for Java code developed by a company named *eXcentia*. According to the article, exhaustive experiments on extensive Java projects using the plugin also have reported sixty as a possible threshold. Since the ABC metric counting rules are similar between languages, we can safely adopt sixty as a threshold value for the ABC magnitude in our study.

For the WMC threshold, we have chosen the value thirty-four, reported in a scientific paper about object-oriented metrics. The authors of the article analyse statical distributions of measures found in practice, evaluate the threshold in proprietary software, and perform a qualitative analysis. The result of the study is a list of several object-oriented metrics thresholds[69], including the one we have chosen.

We have defined a threshold of forty for the NPM and ten for the NPA. A study on forty samples of object-oriented open-source Java programs of various sizes and purposes[70] has demonstrated the validity of those values.

For the COA and CDA, since they are metrics defined by us, we have not found any possible threshold values. For this reason, we have selected the maximum value possible as a threshold for these metrics. So for COA and CDA, we have set the thresholds to one.

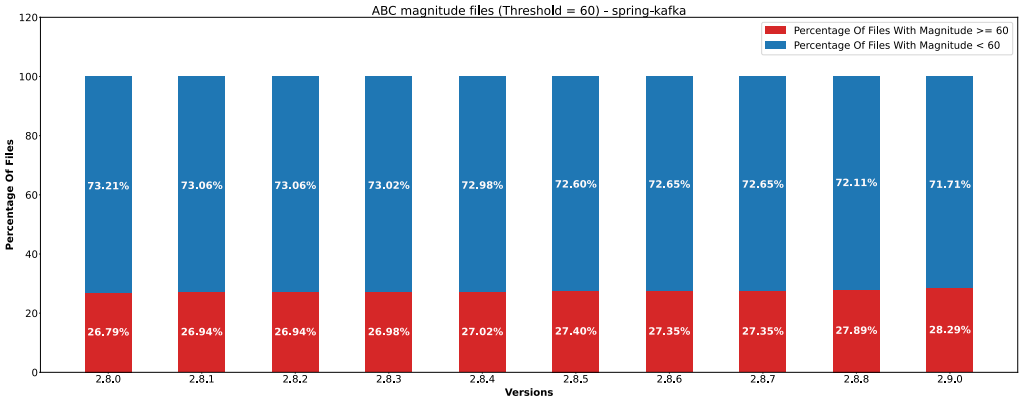


**Figure 5.13:** ABC threshold percentages for Spring for Apache Kafka.

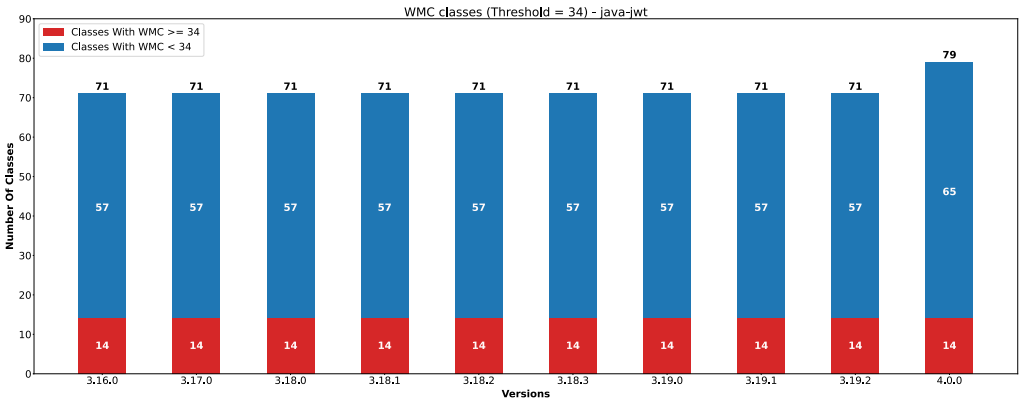


**Figure 5.14:** WMC threshold measures for Java JWT.

Looking at the ABC magnitude threshold graphs (**Figure 5.13**), we have noticed that little repositories tend to have more files that exceed the threshold. Instead, larger repositories, having more files, can manage the content of their files better, making them smaller.

We can recognise the same pattern even for the WMC values (**Figure 5.14**). By increasing the number of classes, we can also see how the number of complex classes decreases.

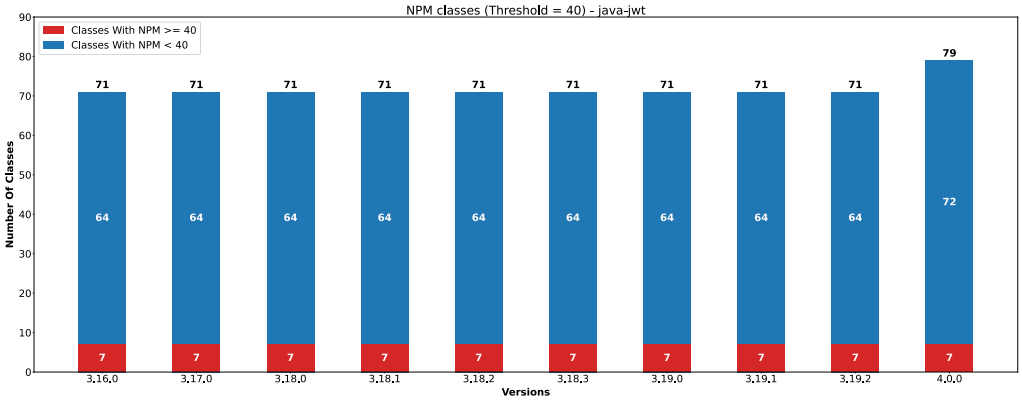


**Figure 5.15:** NPM threshold measures for Java JWT.

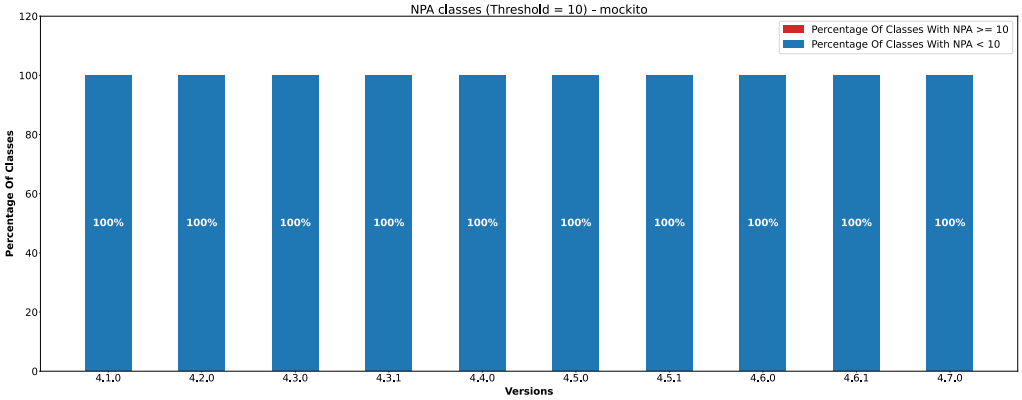


**Figure 5.16:** NPA threshold percentages for Mockito.

For the NPM metric in general, the bar graphs (**Figure 5.15**) show low numbers of classes with high NPM values, except for the smallest codebase. We can assume that developers of medium and large projects consider the NPM measures during their work. They ensure not to declare too many public methods in a single class.

We can also notice a similar but more extreme phenomenon for the NPA metric. The graphs (**Figure 5.16**) show that all the measured codebases have declared very few classes with high NPA. Programmers declare attributes as public only if necessary, paying public attributes even more attention than public methods.

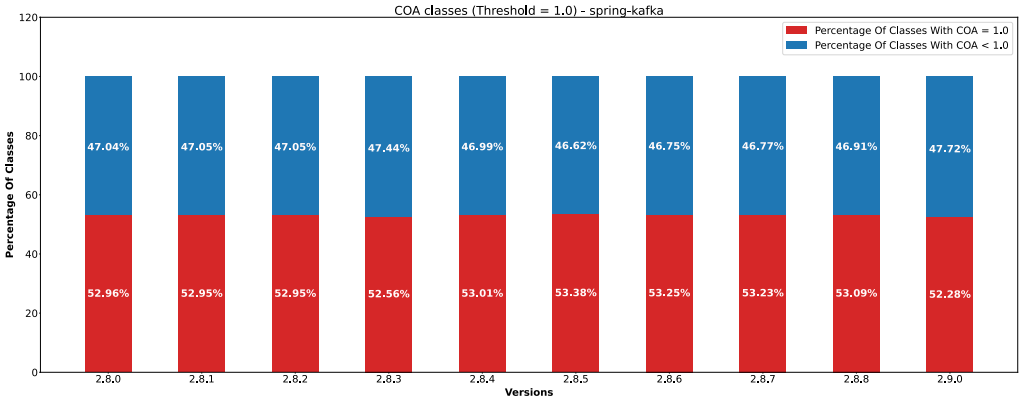


**Figure 5.17:** COA threshold percentages for Spring for Apache Kafka.

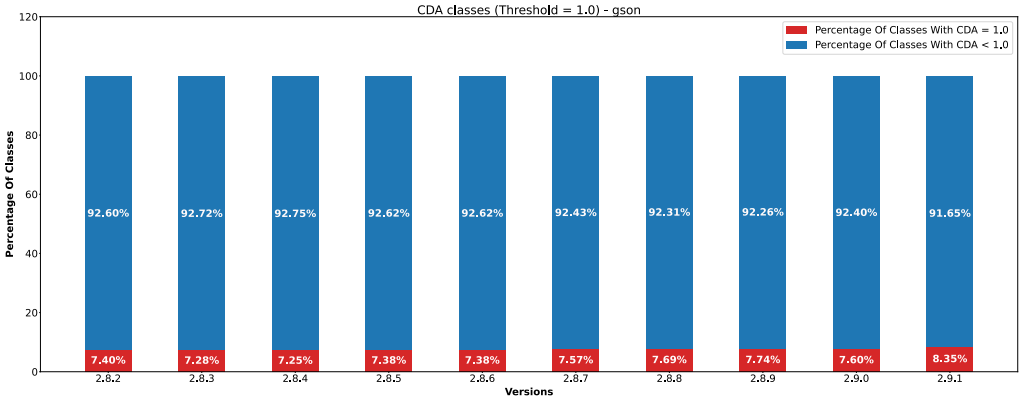


**Figure 5.18:** CDA threshold percentages for Gson.

The COA graphs (**Figure 5.17**) give us more information about the methods defined in the classes of the selected projects. All the bar graphs show many classes with all the methods defined as public. In three of the four projects, the percentage exceeds fifty per cent. We can deduce that it is common to have classes with all their methods declared public, despite the security risks. Programmers should pay more attention to this aspect during development. Alternatively, we can think of a lower threshold for the metric.

For the CDA graphs (**Figure 5.18**), we notice something different. With some exceptions, we can see a tendency to lower the percentage of classes with CDA equal to one with the growth of the codebase size. Programmers avoid declaring classes with only public attributes, especially in large projects.

# Chapter 6

# Conclusions

The work produced in this thesis consisted of the research, implementation and analysis of six new metrics for the Java language within Rust-code-analysis. Other contributors could also extend these new metrics since the project is open source. In addition to the metrics implemented, we have improved RCA functionalities by proposing a system to manage object-oriented metrics and introducing a set of integration tests.

The source code metrics implemented have different applications and purposes:

- ABC is a size metric. Programmers can use it as an additional measure to estimate the size of a source code file or a codebase.
- WMC, NPM and NPA are object-oriented metrics. We can use them to measure the design quality of classes in object-oriented applications.
- Finally, COA and CDA are security metrics. Developers can use them as an indicator of the security of a class, file or project.

We have shown potential uses of the metrics by applying them to well-known and actively maintained projects. Each implemented metric has its advantages and disadvantages, which may vary from the environment they are applied to. However, metrics are just indicators of potential problems in the code, and one should not just focus on them while developing or maintaining a source code. This concept is also called Goodhart's law, from the British economist that defined it, and it states: *"When a measure becomes a target, it ceases to be a good measure"*[71].

At the end of the thesis, we can also define a few tasks as future work:

- **Cover more cases**: The implementations and unit tests for the implemented metrics can be extended to support more specific scenarios of the Java language.

For example, in a modern version of the Java language, Enums are considered classes. A contributor could write some unit tests to cover the Enums cases and extend the implementation to pass the new tests.

- **Implement the new metrics for more languages**: All six implemented metrics can be extended by adding support for other programming languages. RCA, in fact, already provides an easy-to-learn infrastructure to do that.
- **Derived metrics**: A researcher could look for new source code metrics that can be computed using the new implemented metrics. This way, RCA could support even more complex metrics by relying on the basic metrics it produces.
- **New mechanisms to compute metrics**: More RCA functionality could be added to allow a more structured analysis of the source files of a codebase. Some metrics need to know information spread across multiple files and can only be computed after all the data are collected. Another example is duplication metrics. To work, this kind of metric needs specific features that RCA at the moment does not have.

# Bibliography


[1] Norman Fenton and James Bieman. *Software Metrics: A Rigorous and Practical Approach, Third Edition*. 3rd. USA: CRC Press, Inc., 2014. ISBN: 1439838224 (cit. on p. 3).

[2] Aline Lopes Timóteo, Re Álvaro, Eduardo Santana De Almeida, Silvio Romero De, and Lemos Meira. *Software Metrics: A Survey* (cit. on p. 4).

[3] Rajender Chhillar and Sonal Gahlot. «An Evolution of Software Metrics: A Review». In: Aug. 2017, pp. 139–143. ISBN: 978-1-4503-5295-6. DOI: 10.1145/3133264.3133297 (cit. on p. 4).

[4] *NDepend - How Has Static Code Analysis Changed Through the Years?* URL: https://blog.ndepend.com/static-code-analysis-changed-years/ (cit. on p. 5).

[5] Luca Ardito, Riccardo Coppola, Luca Barbato, and Diego Verga. «A tool-based perspective on software code maintainability metrics: a systematic literature review». In: *Scientific Programming* 2020 (2020) (cit. on p. 6).

[6] S.R. Chidamber and C.F. Kemerer. «A metrics suite for object oriented design». In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493. DOI: 10.1109/32.295895 (cit. on p. 7).

[7] Dr. Linda, H. Rosenberg, and Lawrence E. Hyatt. *Software Quality Metrics for Object Oriented System Environments, A report of SATC's research on OO metrics* (cit. on p. 7).

[8] *SonarQube - Metric Definitions*. URL: https://docs.sonarqube.org/latest/user-guide/metric-definitions/ (cit. on p. 8).

[9] *NDepend - Code Metrics Definitions*. URL: https://www.ndepend.com/docs/code-metrics (cit. on p. 8).

[10] *Code analysis documentation - Visual Studio (Windows)*. URL: https://learn.microsoft.com/en-us/visualstudio/code-quality/?view=vs-2022 (cit. on p. 8).

[11] *JaSoMe GitHub repository*. URL: https://github.com/rodhilton/jasome (cit. on p. 9).

[12] *CK GitHub repository*. URL: https://github.com/mauricioaniche/ck (cit. on p. 9).

[13] *Common Weakness Enumeration*. URL: https://cwe.mitre.org/index.html (cit. on p. 9).

[14] Jerry Fitzpatrick. «Applying the ABC metric to C, C++, and Java». In: 2000 (cit. on p. 10).

[15] *PHP Depend - NPM - Number of Public Methods*. URL: https://pdepend.org/documentation/software-metrics/number-of-public-methods.html (cit. on p. 12).

[16] Bandar Alshammari, Colin Fidge, and Diane Corney. «Security Metrics for Object-Oriented Class Designs». In: *2009 Ninth International Conference on Quality Software*. Aug. 2009, pp. 11–20. DOI: 10.1109/QSIC.2009.11 (cit. on p. 13).

[17] *Rust*. URL: https://www.rust-lang.org/ (cit. on p. 15).

[18] Luca Ardito, Luca Barbato, Marco Castelluccio, Riccardo Coppola, Calixte Denizet, Sylvestre Ledru, and Michele Valsesia. «rust-code-analysis: A Rust library to analyze and extract maintainability information from source codes». In: *SoftwareX* 12 (2020), p. 100635. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2020.100635. URL: https://www.sciencedirect.com/science/article/pii/S2352711020303484 (cit. on p. 17).

[19] *Rust-Code-Analysis GitHub repository*. URL: https://github.com/mozilla/rust-code-analysis (cit. on p. 18).

[20] *Rust-Code-Analysis crates.io crate*. URL: https://crates.io/crates/rust-code-analysis (cit. on p. 18).

[21] *Crates.io*. URL: https://crates.io/ (cit. on p. 18).

[22] *The Cargo Book*. URL: https://doc.rust-lang.org/cargo/ (cit. on p. 18).

[23] *Tree-sitter documentation*. URL: https://tree-sitter.github.io/tree-sitter/ (cit. on p. 20).

[24] *Abstract vs. Concrete Syntax Trees*. URL: https://eli.thegreenplace.net/2009/02/16/abstract-vs-concrete-syntax-trees/ (cit. on p. 20).

[25] *Tree-sitter documentation - Named vs Anonymous Nodes*. URL: https://tree-sitter.github.io/tree-sitter/using-parsers#named-vs-anonymous-nodes (cit. on p. 21).

[26] *Rust Standard Library documentation*. URL: https://doc.rust-lang.org/std/ (cit. on p. 24).

[27] *PDF.js documentation*. URL: https://mozilla.github.io/pdf.js/ (cit. on p. 24).

[28] *DeepSpeech documentation*. URL: https://deepspeech.readthedocs.io/en/r0.9/ (cit. on p. 24).

[29] *Walkdir documentation*. URL: https://docs.rs/walkdir/latest/walkdir/ (cit. on p. 27).

[30] *Crossbeam documentation*. URL: https://docs.rs/crossbeam/latest/crossbeam/ (cit. on p. 27).

[31] *Serde documentation*. URL: https://serde.rs/ (cit. on p. 29).

[32] *GitHut*. URL: https://madnight.github.io/githut/ (cit. on p. 39).

[33] *The Java Tutorials*. URL: https://docs.oracle.com/javase/tutorial/ (cit. on p. 39).

[34] *Tree-sitter-Java GitHub repository*. URL: https://github.com/tree-sitter/tree-sitter-java (cit. on p. 39).

[35] *Notepad GitHub repository*. URL: https://github.com/buiducnhat/Notepad (cit. on p. 40).

[36] *Simple-Java-Calculator GitHub repository*. URL: https://github.com/pH-7/Simple-Java-Calculator (cit. on p. 40).

[37] *NetBeans GitHub repository*. URL: https://github.com/apache/netbeans (cit. on p. 40).

[38] *Rustfmt GitHub repository*. URL: https://github.com/rust-lang/rustfmt (cit. on p. 41).

[39] *Clippy GitHub repository*. URL: https://github.com/rust-lang/rust-clippy (cit. on p. 41).

[40] *Getting Started With Codecov and Rust*. URL: https://about.codecov.io/language/rust/ (cit. on p. 41).

[41] *GMetrics documentation*. URL: https://dx42.github.io/gmetrics/ (cit. on p. 43).

[42] *The Apache Groovy programming language*. URL: https://groovy-lang.org/ (cit. on p. 43).

[43] *GMetrics GitHub repository*. URL: https://github.com/dx42/gmetrics (cit. on p. 44).

[44] *GMetrics tests*. URL: https://github.com/dx42/gmetrics/blob/master/src/test/groovy/org/gmetrics/metric/abc/AbcMetric_MethodTest.groovy (cit. on p. 44).

[45] *Compound assignment operators in Java*. URL: https://www.geeksforgeeks.org/compound-assignment-operators-java/ (cit. on p. 44).

[46] *Serde - Field attributes*. URL: https://serde.rs/field-attrs.html (cit. on p. 53).

[47] *Rust Reference - Attributes*. URL: https://doc.rust-lang.org/reference/attributes.html (cit. on p. 53).

[48] *Rust By Examples - Attributes*. URL: https://doc.rust-lang.org/rust-by-example/attribute.html (cit. on p. 53).

[49] *Metrics-Analysis GitHub repository*. URL: https://github.com/marco-ballario/metrics-analysis/ (cit. on p. 62).

[50] *Maven Central Repository*. URL: https://search.maven.org/ (cit. on p. 63).

[51] *Maven*. URL: https://maven.apache.org/ (cit. on p. 63).

[52] *Gradle*. URL: https://gradle.org/ (cit. on p. 63).

[53] *FastCSV GitHub repository*. URL: https://github.com/osiegmar/FastCSV (cit. on p. 64).

[54] *Java-JWT GitHub repository*. URL: https://github.com/auth0/java-jwt (cit. on p. 64).

[55] *Introduction to JSON Web Tokens*. URL: https://jwt.io/introduction (cit. on p. 64).

[56] *Gson GitHub repository*. URL: https://github.com/google/gson (cit. on p. 64).

[57] *Java-WebSocket GitHub repository*. URL: https://github.com/TooTallNate/Java-WebSocket (cit. on p. 64).

[58] *The WebSocket Protocol*. URL: https://www.rfc-editor.org/rfc/rfc6455 (cit. on p. 64).

[59] *Spring-Kafka GitHub repository*. URL: https://github.com/spring-projects/spring-kafka (cit. on p. 64).

[60] *Spring for Apache Kafka*. URL: https://spring.io/projects/spring-kafka (cit. on p. 64).

[61] *Spring Framework*. URL: https://spring.io/projects/spring-framework (cit. on p. 64).

[62] *Apache Kafka*. URL: https://kafka.apache.org/ (cit. on p. 64).

[63] *Mockito GitHub repository*. URL: https://github.com/mockito/mockito (cit. on p. 65).

[64] *GitHub's 10,000 most Popular Java Projects – Here are The Top Libraries They Use*. URL: https://www.overops.com/blog/githubs-10000-most-popular-java-projects-here-are-the-top-libraries-they-use/ (cit. on p. 65).

[65] *Python Beginners Guide - Overview*. URL: https://wiki.python.org/moin/BeginnersGuide/Overview (cit. on p. 65).

[66] J. D. Hunter. «Matplotlib: A 2D graphics environment». In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55 (cit. on p. 65).

[67] *Ten Percent Not Crap - Groovy Code Metrics: ABC*. URL: https://tenpercentnotcrap.wordpress.com/2013/01/14/groovy-code-metrics-abc/ (cit. on p. 81).

[68] *Jake Scruggs - What's a Good Flog Score?* URL: https://jakescruggs.blogspot.com/2008/08/whats-good-flog-score.html (cit. on p. 81).

[69] Tarcísio G. S. Filó and Mariza Bigonha. «A Catalogue of Thresholds for Object-Oriented Software Metrics». In: 2015 (cit. on p. 82).

[70] Kecia A.M. Ferreira, Mariza A.S. Bigonha, Roberto S. Bigonha, Luiz F.O. Mendes, and Heitor C. Almeida. «Identifying thresholds for object-oriented software metrics». In: *Journal of Systems and Software* 85.2 (2012). Special issue with selected papers from the 23rd Brazilian Symposium on Software Engineering, pp. 244–257. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2011.05.044. URL: https://www.sciencedirect.com/science/article/pii/S0164121211001385 (cit. on p. 82).

[71] *The Problem with Software Measurement and Metrics*. URL: https://www.techwell.com/techwell-insights/2017/02/problem-software-measurement-and-metrics (cit. on p. 86).