FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Multi-Language Software Metrics

Gil Dinis Magalhães Teixeira

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: João Bispo, PhD

Second Supervisor: Filipe Figueiredo Correia, PhD

February 12, 2021

Multi-Language Software Metrics

Gil Dinis Magalhães Teixeira

Mestrado Integrado em Engenharia Informática e Computação

February 12, 2021

Abstract

Software metrics are of utmost significance in software engineering. They attempt to capture the properties of programs, which developers can use to gain objective, reproducible and quantifiable measurements. These measurements can be applied in quality assurance, code debugging, and performance optimization. The relevance of their use is continuously increasing with software systems growing in complexity, size, and importance.

Implementing metrics is a tiring and challenging task that most companies ought to surpass as part of their development process. Also, this task usually must be repeated for every language used. However, suppose we can decompose the implementation of software metrics into a sequence of basic queries to the source code that can be implemented in a language-independent way. In that case, these metrics can be applied to any language.

Many programming languages share similar concepts (e.g., functions, loops, branches), which has led certain approaches to raise the abstraction level of source-code analysis and support multiple languages. LARA is a framework developed in FEUP (Faculty of Engineering of the University of Porto) for code analysis and transformation that is agnostic to the language being analyzed. This means that analysis strategies written in LARA have the potential of being applied to the source code of multiple languages.

A study is conducted to identify the benefits and drawbacks of software metrics as well as to select and analyze relevant and commonly used software metrics. This will provide the foundations for the implementation of software metrics in a language-independent way by using the LARA framework.

In order to be able to perform a multi-language analysis, we created a model on top of each compiler's AST, where every node is mapped to a common join point. This results in a virtual AST with the same join points for each language. Using this model, we also created a library of metrics that can be applied to multiple languages.

To validate the developed framework, we performed an internal and external validation. The internal validation, by comparing a projected implemented in different languages, ensures that the metrics results are consistent. In contrast, the external validation compares several tools' results and performance across multiple open source projects. Even though our approach underperforms in terms of execution time, it presents a flexible and straightforward approach to analyze four different languages (C, C++, Java, JavaScript) and can calculate object-oriented, complexity, and size metrics.

Resumo

Métricas de software são de extrema importância em engenharia de software, já que tentam captar propriedades dos programas, que os desenvolvedores podem utilizar para obterem medições objetivas, reproduzíveis e quantificáveis. Estas medições podem ser aplicadas para se garantir a qualidade do código, para realizar a depuração de código e para otimizar o seu desempenho. A relevância da sua utilização continua a aumentar com os sistemas de software a crescerem em complexidade, tamanho e importância.

A implementação de métricas é uma tarefa exaustiva e exigente, que a maioria das empresas utiliza como parte do seu processo de desenvolvimento. Além disso, esta tarefa tem de ser repetida para cada linguagem utilizada. No entanto, se conseguirmos decompor a implementação de métricas de software numa sequência de queries básicas ao código fonte que possam ser implementadas de forma independente da linguagem, então estas métricas podem ser aplicadas a qualquer linguagem.

Muitas linguagens de programação partilham conceitos semelhantes (por exemplo, funções, loops, branches), o que levou certas abordagens a elevar o nível de abstração da análise do código-fonte para suportar múltiplas linguagens. LARA é uma framework, desenvolvida na FEUP (Faculdade de Engenharia da Universidade do Porto), para a análise e transformação de código que é agnóstica à linguagem a ser analisada. Isto significa que as estratégias de análise escritas em LARA têm o potencial de serem aplicadas ao código fonte de múltiplas linguagens.

Realizámos um estudo para identificar os benefícios e desvantagens das métricas de software, bem como para selecionar e analisar as mais relevantes e mais utilizadas. Isto fornecerá as bases para a implementação de métricas de software de uma forma independente da linguagem, utilizando a framework LARA.

De forma a poder realizar uma análise multilinguagem, criámos um modelo em cima da AST de cada compilador, onde cada nó é mapeado para um join point comum. Isto resulta numa AST virtual com os mesmos join points para cada linguagem. Usando este modelo, também criámos uma biblioteca de métricas que pode ser aplicada a múltiplas linguagens.

Para avaliar a framework desenvolvida, realizámos uma validação interna e externa. A validação interna, comparando um projeto implementado em diferentes linguagens, assegura que os resultados das métricas são consistentes. Por sua vez, a externa compara os resultados e o desempenho de várias ferramentas em múltiplos projetos open-source. Ainda que a nossa abordagem tenha um desempenho inferior em termos de tempo de execução, apresenta uma abordagem simples e flexível para analisar quatro linguagens diferentes (C, C++, Java, JavaScript) e pode calcular métricas orientadas a objetos, mas também de complexidade e de tamanho.

Agradecimentos

Apresento o meu agradecimento sincero e profundo aos Professores Doutores João Bispo e Filipe Figueiredo Correia, pela forma persistente e cuidada com que me foram acompanhando ao longo desta aventura. Docentes universitários de reconhecido mérito científico e pedagógico, tiveram a generosidade de partilhar comigo ideias, sugestões e perspetivas de pesquisa e análise sempre oportunas e relevantes. Houve momentos em que os seus conselhos serviram também para reforçar o ânimo e ajudar a ultrapassar as hesitações próprias de alguém que se encontrava a pisar sendas desconhecidas e por isso capazes de provocar dúvidas e até situações de algum desânimo.

Agradeço aos meus pais todo o carinho e atenção que revelaram durante o longo período em que este trabalho nos ocupou. O equilíbrio familiar foi um fator determinante para que as capacidades cognitivas e o espírito crítico pudessem funcionar com fluência e naturalidade.

Agradeço à Faculdade de Engenharia da Universidade do Porto por me ter proporcionado a oportunidade de cumprir este grande sonho da minha vida.

Gil Teixeira

“Measure what is measurable,
and make measurable what is not so.”

Galileo Galilei

Contents

List of Figures

List of Tables

Abbreviations

SE	Software Engineering
OO	Object Oriented
CST	Concrete Syntax Tree
eCST	Enriched Concrete Syntax Tree
AST	Abstract Syntax Tree
LOC	Lines of Code
LCOM	Lack of Cohesion in Methods
CBO	Coupling between Objects
DIT	Depth of Inheritance Tree
NOC	Number of Children
RFC	Response For Class
WMC	Weighted Methods per Class
DAC	Data Abstraction Coupling
MPC	Message Passing Coupling
NOM	Number Of Methods
SIZE1	Size of procedures or functions
SIZE2	Size of properties defined in a class
CogniComplex	Cognitive Complexity
CycloComplex	Cyclomatic Complexity
NOL	Number of Lines
NOCl	Number of Classes
NOFi	Number of Files
NOFu	Number of Functions

Chapter 1

Introduction

This chapter gives the reader an overview of this work. Section 1.1 introduces the context of the problem. Section 1.2 refers the reasons for choosing this topic. The research questions are described in Section 1.3. Section 1.4 identifies the several steps to achieve this thesis's goal. Finally, Section 1.5 presents the overall structure of this document.

1.1 Context

Software metrics are of utmost relevance in software engineering. They try to capture the properties of programs, which developers can use to obtain objective, reproducible and quantifiable measurements. These measurements can be applied to ensure code quality, perform code debugging, and optimize performance. Their use's relevance continues to grow with software systems growing in complexity, size, and importance.

Since the 1960s and 1970s, a considerable number of metrics have been proposed to analyze the software source code, e.g., metrics that evaluate the size and complexity of programs such as Lines of Code or Cyclomatic Complexity. With the increasing popularity of the object-oriented language development paradigm, metrics that focus on this paradigm have also emerged [6], such as Lack of Cohesion in Methods (LCOM) and Coupling between object classes (CBO). [26]

There are many tools that allow developers to calculate source code metrics quickly. Even though these tools make this process easier, the measures provided may not be relevant in a given context [8], making it necessary for the development teams to have at their disposal tools to support the development of metrics. This complicated and repetitive task usually has to be repeated for all languages used to ensure that the metric calculation is consistent. However, if we can break down the implementation of software metrics into a sequence of source code queries that can be implemented in a language-independent manner, then that metric can be applied to any language.

1.2 Motivation

When analyzing software systems, one of the most used techniques consists of using software metrics. Although many metrics have been proposed and implemented, there is no standard for metrics implementation. This results in different tools having different implementations of the same metric. This problem is further amplified with the metrics having to be reimplemented in various languages. Also, several tools focus on just one language or one programming paradigm. Thus, a tool that can analyze multiple languages from different programming paradigms would reduce the time and effort the developers need to allocate to obtain relevant metrics for software development. The developed solution is built upon the LARA framework [28]. Even though this tool can do code analysis agnostic to the language being analyzed, they do not share the same specifications since the different compilers were developed independently. So this framework was extended to support additional language specifications that can be shared between LARA compilers, and it was also added software metrics measurement capabilities.

1.3 Research Questions

This dissertation aims to understand the benefits and drawbacks of developing a multi-language analysis tool capable of calculating different types of metrics. This study will therefore address the following research questions:

> *RQ1: What impact will a multi-language approach have in terms of effort to add new languages?*

The complexity of projects is continuously increasing, and one of those factors is the number of languages. Using a tool that supports a multi-language approach would decrease the number of tools required to analyze the full project.

> *RQ2: What impact will a multi-language approach have in terms of effort to add new metrics?*

The tools available do not always present the exact metrics the developer wants to calculate, so having a tool that can easily be extended to support those metrics would save both time and effort.

> *RQ3: What impact will a multi-language approach have in terms of performance?*

Using a multi-language tool will add an overhead compared to only having to analyze a single language. So it is essential to evaluate the trade-offs of the proposed approach.

> *RQ4: What impact will a multi-language approach have in terms of the consistency of metrics across languages?*

Between different tools, since they have different implementations for the same metrics, the results are inconsistent. This study aims to understand if this is still the case when there is only a single implementation.

1.4 Methodology

The ultimate goal is to demonstrate that it is possible to implement a set of code metrics to evaluate source code quality and maintainability in a language-independent approach. The proposed solution uses LARA, a framework developed at the Faculty of Engineering of the University of Porto (FEUP), for code analysis and transformation that is agnostic to the language under analysis. This means that the analysis strategies written at LARA have or can be applied to the source code of multiple languages. However, although this tool has the potential to be extended with metrics capabilities, this has not yet been realized. To accomplish this goal, the following objectives were established:

- Selection of a set of metrics that can be implemented independently of language;
- Extension of the LARA framework with software metrics measurement and analysis capabilities;
- Implementation of the metrics collected;
- Validation of the results obtained in different languages and their comparison with metrics measurement tools;

1.5 Dissertation Structure

Beyond the introduction, this report is divided into four different chapters. Chapter 2 describes the state of the art in source code analysis. In this chapter, an analysis of the most used metrics, a comparison of models for multi-language source code representation as well as an exploration of approaches and tools for language-independent code analysis is performed. Chapter 3 presents a multi-language approach for metrics calculation. Chapter 4 details the developed tool's experimental results and compares it with other analysis tools. Finally, Chapter 5 describes the conclusions taken from the work developed while also presenting research lines arising from this work that could be followed.

Chapter 2

Source Code Analysis

This chapter aims to provide an overview of the current developed work for source code analysis. The study of similar approaches and solutions helps to know what's been done and what else can be added. Hence this chapter has been divided into four sections. Firstly, a description and comparison of some of the most studied metrics is carried out to evaluate the size and complexity of the source code, as well as, metrics that only focus object-oriented systems. Since the extraction of metrics requires a model to represent the source code, it is essential to analyze several multi-languages models and what level of information is retrieved from the source code. Finally, since our work focus on developing a system to calculate metrics in a language-independent way, it is crucial to analyze the current solutions and determine their current limitations. Therefore, the last two sections focus on the literature approaches for language-independent code analysis and the currently available software metrics tools.

2.1 Source Code Metrics

In software engineering, a software metric is a standard of measure of a particular characteristic or property of a software system. Software metrics can be divided into three categories: project, process, and product metrics [18]. This section focus on a subcategory of product metrics. Source code metrics can be categorized in several ways depending on:

- Type of analysis - categorizes the analysis required for the metric, i.e., static or dynamic. While static code analysis does not require the source code's execution, dynamic analysis relies on examining the program behavior during execution.
- Topic of study - the metrics are divided according to the area of study (e.g., software complexity, code plagiarism detection, software testing).
- Level of Granularity - categorizes the metrics by the element in analysis (e.g., system, class, method).

2.1.1 Lines of Code

Lines of code (LOC) is the oldest, simplest, and most widely used software metric to measure the size of a program [26]. As the name implies, it counts the number of lines of the source code. LOC is usually used to predict the effort that the development of a program requires. However, it can also be used to calculate project-wide attributes [24] such as:

Productivity $LOC/Person$

Quality $Defects/LOC$

Cost per LOC $(TotalCost)/LOC$

The main advantage of LOC is its simplicity and ease of implementation. It is used to detect several code smells such as *Large Method* or *Large Class*, and it can indicate that a method or class requires refactoring. Nevertheless, it is not considered a robust metric since the number of lines for the same algorithm depends on the language used, the programmer and coding standard, e.g., a standard *if* statement requires three lines of code, however by using the ternary operator that same statement can be written in a single line. Furthermore, not all lines of code require the same level of effort to write (e.g., getters/setters that can be automatically generated vs complex business logic).

2.1.2 Mccabe's Cyclomatic Complexity Metric

Cyclomatic complexity [25] is a graph-theoretic software measure used to indicate a program complexity. This metric was introduced to tackle testability and maintainability issues, as both time and money are spent testing and maintaining systems.

This metric is computed using the program's control flow graph and measures the number of paths through a program. Cyclomatic complexity (CC) can be calculated as follows:

$$CC = e - n + 2p$$

where:

n is the number of vertices

e is the number of edges

p is the number of connected components

However, more straightforward ways to calculate CC were proposed [24] that present an equivalent value as the original McCabe metric. So the complexity can also be calculated by determining the number of decision statements plus one. So to calculate CC sum the following occurrences and add one:

- The number of if/then statements;
- The number of cases within a switch statement;

- The number of loops
- The number of try/catch statements.

Ideally, the complexity of the code should be low. High complexity means the system is harder to understand and harder to maintain. Studies [30] have found a correlation between the complexity of functions and methods and the number of bugs. Besides, more tests have to be written to account for all the possible paths. For example, Figure 2.1 represents the control flow graph from the snippet of code to generate the nth term of the Fibonacci sequence (Listing 2.1). The cyclomatic complexity can be calculated from this graph. The graph shows sixteen nodes, twenty edges, and only one connected component. Therefore the cyclomatic complexity is $19 - 16 + 2 = 5$. Using the alternative method, since the function has three *if* statements and one loop, the CC equals five.

```python
def fibonacci(n):
    a = 0
    b = 1
    if n < 0:
        print("Invalid Input")
    elif n == 0:
        return a
    elif n == 1:
        return b
    else:
        for i in range(2,n):
            c = a + b
            a = b
            b = c
        return b
```

Listing 2.1: Snippet of code to generate the nth term of the Fibonacci Sequence

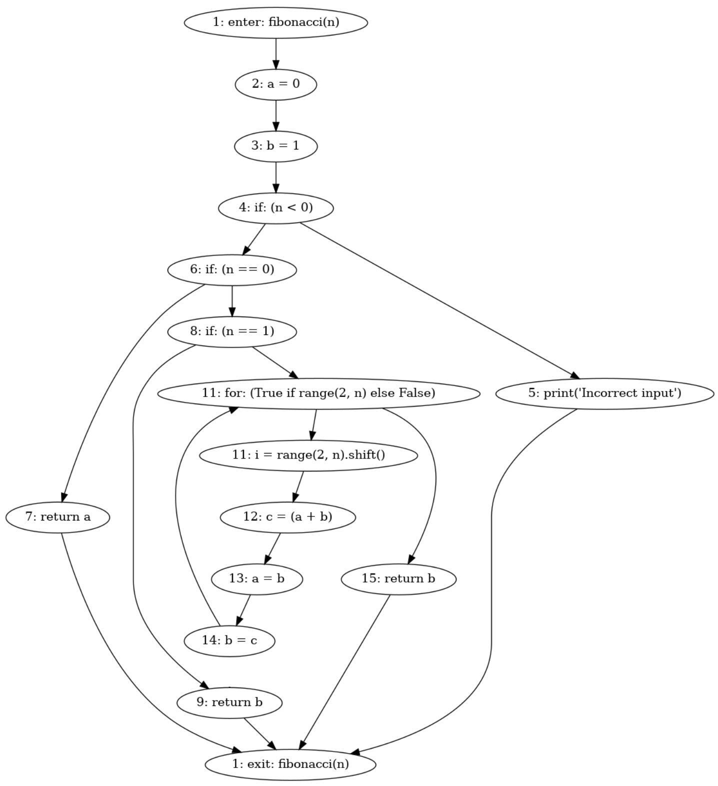

Figure 2.1: The Control Flow Graph

Cyclomatic complexity is a widely used and successfully proved metric to measure the complexity of the code. It can be applied to individual functions, methods, classes, and modules of a program. The original metric proposed by McCabe can be hard to calculate by requiring the Control Flow Graph, however, by using the number of conditional statements, the calculation becomes trivial.

2.1.3 Halstead's Complexity Metrics

Halstead complexity measures [15] emerged as an alternative to using LOC as a measure of the complexity of code. The main goal of these metrics is to obtain measurable properties of software and the relations between them. The calculation of Halstead metrics requires the following notions:

n_1 = number of unique operators.

n_2 = number of unique operands.

N_1 = total number of operators.

N_2 = total number of operands.

From the notions of operators and operands, several metrics can be calculated:

Program vocabulary is the number of unique operators and operands: $n = n_1 + n_2$

Program length is the count of the total number of operators and operands: $N = N_1 + N_2$

Program Volume is used to measure the program size: $V = N * log_2(n)$

Difficulty Level shows how difficult is to maintain the program: $D = (n1/2) * (N2/n2)$

Program Level is the inverse of the previous metric: $L = 1/D$

Programming Effort measures the mental effort needed to implement an existing algorithm in the specified program language: $E = V * D$

Programming Time shows the time in minutes required to implement an existing algorithm in the specified program language: $T = E/(60 * 18)$

Halsted's metrics are straightforward to calculate. Considering the snippet of code presented in Listing 2.1, the operators and operands were collected in Table 2.1 and the values of the measures are shown in Table 2.2. These metrics can be used to measure the quality of programs and to predict the maintenance effort and error rate. However, the significant difficulty is to define the operands and operators for multiple languages. Another issue is that these measures do not analyze the structure of the code and the interactions between modules [24].

2.1.4 Object-Oriented Metrics

The paradigm of object-oriented programming uses the concept of objects that hold data, using fields, but also can call procedures, i.e., methods. Each object can be implemented independently which leads to a modular system that allows code reuse. Some of the most popular languages, such as C++, Java and JavaScript support this paradigm. Therefore, multiple metrics were created to evaluate object-oriented systems.

Operators	Occurrences	Operands	Occurrences
`=`	5	`a`	4
`if`	1	`0`	3
`:`	5	`b`	6
`<`	1	`1`	1
`print`	1	`n`	4
`elif`	2	`"Invalid Input"`	1
`else`	1	`i`	1
`==`	2	`2`	1
`return`	3	`c`	2
`for`	1		
`in`	1		
`range`	1		
`()`	2		
`+`	1		
`,`	1		
$n_1 = 15$	$N_1 = 28$	$n_2 = 9$	$N_2 = 23$

Table 2.1: Halstead's Operators and Operands

Metric	Value
n	24.0
N	51.0
V	234.0
D	19.0
L	0.1
E	4446.0
T	4.1

Table 2.2: Halstead's metrics

2.1.4.1 The Chidamber and Kemerer Metrics

Chidamber and Kemerer [6] were pioneers in presenting object-oriented metrics. These metrics, introduced in 1994, have been widely implemented and used as a base for other measures. They provide essential information on OO systems' main concepts such as size, complexity, inheritance, coupling, and cohesion [26]. The metrics proposed were [5]:

Coupling Between Object (CBO) counts the number of non-inherited classes coupled to a particular class. A class is coupled to another if the methods of one class use the methods or access the variables of the other. This metric should be as low as possible. A system with low coupling is more modular and easier to reuse.

Response For a Class (RFC) counts the number of methods of a class and all the methods that are called by methods in that class. A high RFC may indicate a highly complex class, and therefore it becomes more complicated to understand and test that class.

Lack of Cohesion in Methods (LCOM) counts the number of pairs of methods that do not share instance variables, minus the number of pairs of methods that share instance variables of the class. The higher the LCOM of class, the more difficult it is to maintain, and it might imply the class should be split into more subclasses.

Weighted Methods per Class (WMC) sums all the complexities of the class's methods. The complexity is not specified in order to allow the most general application of this metric. If each method is assigned a complexity of one, WMC is equal to the number of methods in the class. Classes with a higher number of methods are harder to maintain and have a more significant impact on the subclasses since they will inherit all the methods.

Depth of Inheritance Tree (DIT) counts the number of classes that a particular class inherits from. The DIT of the root class is equal to zero. The deeper the inheritance tree, the grater is the number of superclasses, and harder is the class maintainability.

Number of Children (NOC) counts the number of direct subclasses of a class. A greater NOC can indicate a higher level of code reuse since class inheritance is a form of reuse. However, if a class has a large number of children, any changes to the superclass may affect all the other subclasses, making this system hard to maintain.

CK metrics are some of the most widely studied metrics. This set of measures is independent of the language syntax. Many studies validate the usefulness of CK metrics in evaluating many OO attributes. However, some criticism has been made to the definition of the LCOM metric. The fact that this measure truncates all value below zero limits the metric's ability to truly show the lack of cohesion [31].

2.1.4.2 Li and Henry's Metrics

Li and Henry [23] proposed five new OO metrics. These metrics were introduced in the early stages of OO development when very few studies of metrics had been performed in this paradigm. The validation of these metrics' relation and the maintenance effort was also accomplished in two commercial systems. The five metrics are summarized as:

Message Passing Coupling (MPC) counts the number of send statements defined in a class. If a class has a high MPC, it indicates a high coupling between classes. Therefore the classes are more dependant on each other, which introduces more complexity to the system.

Data Abstraction Coupling (DAC) counts the number of attributes in a class that have another class as their type. If a class has attributes whose type is of another class, then the more complex is the coupling between that class and the others.

Number Of Methods (NOM) counts the number of local methods. The class complexity increases with the number of methods.

Size of procedures or functions (SIZEl) counts the number of semicolons in a class. This metric is proposed as an alternative of LOC for OO oriented programs to measure or size.

Size of properties defined in a class (SIZE2) sums the number of attributes and methods of a class. This metric is also used to measure the size of a class.

The metrics DAC, NOM, and SIZE2 are straightforward to implement and can be calculated using a UML class diagram. The MPC metric can be particularly useful to calculate the coupling between classes. However, the practicality of the metric SIZE2 may depend on the language in analysis.

2.1.5 Metrics Comparison

The metrics described in previous sections are among the most cited metrics in literature. A systematic mapping of studies published select and analyzed 226 studies related to software between 2010 and 2015. Of the top 10 metrics with the highest number of occurrences, six were CK metrics, one was from Li and Henry (NOM), and the others were LOC, CC, and Number of Attributes. Figure 2.2 depicts the distribution of the metrics.

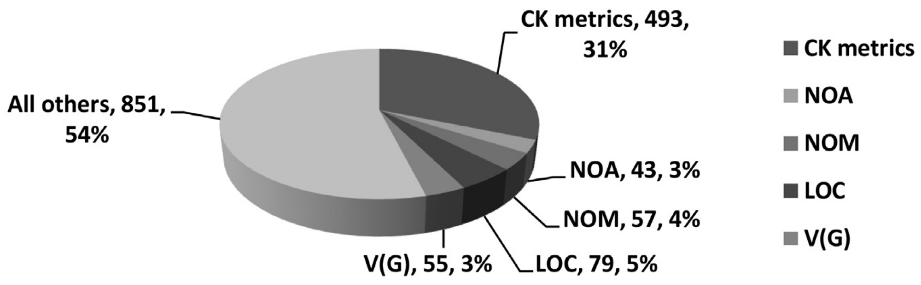

Figure 2.2: Metrics occurrences distribution as presented by Varela et al. [26]

To compare different metrics is essential to establish the level of detail of the model of the source required to obtain it. The LOC metric can be trivially calculated by inspecting the source file. McCabe's Cyclomatic Complexity requires the deepest level of information, abstract syntax tree level since it is a metric related to the program's control flow. Halsted's metrics also requires the abstract syntax tree level, since the definition of operators and operands is deeply connected to the syntax of the language. The Chidamber and Kemerer Metrics require more details about the classes and methods and the relationships between them, therefore requires information at the Program entity level. The Li and Henry's metrics can be divided into two groups: the first group has the metrics: NOM and SIZE2 which requires only a high-level overview of the classes; the second group: MPC and DAC require more details on the relationship between classes, therefore requires information at the Program entity level.

2.2 Models for Multi Language Source Code Representation

The gathering of metrics for multiple languages requires the modeling of source code. If the obtained data is more detailed, it requires higher memory consumption, and it takes more time to process. However, it provides deeper insight regarding the analyzed program. The level of information extracted from the source code can be divided into three levels of detail [12]:

- Abstract syntax tree level information — provides a complete overview of the source code. It is the deepest level, and it allows to make a control-flow analysis.

- Program entity level information — provides information on the program entities (classes, fields, and methods). However, it gives limited information about the bodies of methods and functions, consisting of only variable access and method invocation.

- Architectural level information — provides a high-level overview of the project architecture, namely, the modules, packages, and components used. It can also provide a high-level view of the classes of the project, however, it does not give information about its relationships.

2.2.1 LARA

Aspect-Oriented Programming (AOP) [20] paradigm separates concerns (e.g., logging and profiling) of a software application to improve modularization. Most AOP languages are designed to be extensions of their target language. This helps by providing a lower learning curve to the developers since they already know the target language. However, this approach has some drawbacks, such as propagating the limitations of the target language or tying the aspect language to the target language. This last point prevents code reuse by having to develop a new AOP approach for every target language.

Therefore a new language was proposed, LARA [28] is a Domain-Specific Language (DSL) for source-to-source transformations and analysis, inspired by AOP languages, such as AspectJ and AspectC++. However, while these languages are extensions of their target language, Java and C++, respectively, LARA is agnostic to the target language as much as possible. This allows to reduce the effort to add more languages as well as developing features for multiple target languages.

The LARA framework selects points of interest and expresses source code transformations by using weavers, which are tools that connect the language model and the target code representation, e.g., an Abstract Syntax Tree (AST), and allow to write aspects that add new behavior by inserting additional strings of code.

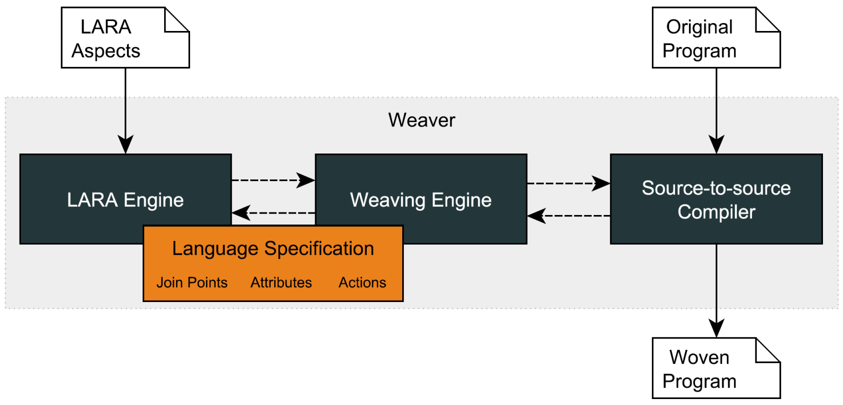

Figure 2.3: The general structure of a weaver based on the LARA framework as presented by Pinto et al. [28]

LARA uses a language model with three main components: join points, attributes, and actions (see Figure 2.3). Joint points represent the code structures that a weaver can capture (e.g., file, function, loop). Joint points define a structure, and indicate what points can be select from other points, e.g., from a file it is possible to select a function, or what points are derived from others, e.g., a call is derived from an expression. Attributes are the properties that can be accessed for each joint point, e.g., the name and type of a variable, and actions allow to change the original source code, e.g., insertions of code.

Contrary to other approaches, to add new languages its not required to implement a parser or grammar, which for most languages its not an easy task. LARA allows to reuse of existing parsers such as Spoon and Clang for Java and C++, respectively. Hence, the join points model can be implemented in a incremental way, by starting to implement a weaver that selects joint points and attributes from a file or function, without having to account for every specific concept of a language.

2.2.2 eCST

Rakic and Budimac [29] propose a structure to represent the source code of multiple languages and in which software metrics could be applied. This structure had to store information at the AST level.

So, due to the fact that software metrics are sensitive to the programming language syntax, they decided on an extension of the Concrete Syntax Tree (CSTs), designated as Enriched Concrete Syntax Tree (eCST). This eCST contains all the elements of the source code and universal nodes to be able to consider the languages' different semantics, e.g., 'else if' in javascript and 'elif' in python. For the generation of the CST, the ANTLR parser generator [27] is used. The extension from CST to eCST is performed with simple changes to the grammar that generates the original CST.

The use of universal nodes gives another task during the parsing and transformation of the source code. Nevertheless, it simplifies the calculation of certain metrics algorithms, such as the cyclomatic complexity. This metric is calculated based on the number of decision statements, and the predicates used differ from the languages used. However, by adding a unique universal node before each branching operation, the measure can be easily calculated.

2.2.3 FAMIX

Tichelaar et al. [32] propose a meta-model for representing OO systems. However, support for procedural languages was posteriorly added. It models at a program entity level, hence it allows to model the classes, methods, attributes invocations, and accesses. Information about methods bodies is limited at the invocation of methods and accesses of attributes. As a result, it does not provide a complete overview of the source code.

For the development of FAMIX, the authors focused on 3 principles: language independency, extensibility, and information exchange. The authors wanted language independence in order to support legacy systems written in languages that are currently not widely used. Therefore, they used an approach that is as independent as possible of the language and deals with the language on a more abstract level. This principle was met with support of several languages such as C++, JAVA, Smalltalk and Ada. FAMIX should be highly extensible and should not hinder the developers from doing what they need to do. Hence, the tool is able to save language-specific information, e.g., hierarchies in C++.

Finally, since the model was designed with the objective of information exchange, the obtained results must be stored in a flat, streaming-friendly way. This guarantees that the elements are self-contained and able to be transferred between tools with an incremental loading of information or in separated transfers.

2.2.4 MOON

Minimal Object-Oriented Notation or MOON [10] is a model to represent the information of an OO system. It includes an concrete and abstract syntax for every language.

It models the program's information at an instruction level, by using a hierarchical model, in which every element inherits the OBJECTMOON class, which has an id, the name of the file and the line where the object appears. This element can then be either a NAMEDOBJECT, an INHERITANCE_CLAUSE, a MODIFIER, an INSTR, or an EXPR. A NAMEDOBJECT can either be an entity, a class, a method, or type. The INHERITANCE_CLAUSE and the MODIFIER are used to represent relations of hierarchy. Finally, the instructions (INSTR) and expressions (EXPR) represent dependency relations.

This conceptual model allows to capture information of the system to, consequently, perform code refactorization. It has the downside of some specific language characteristics not being currently represented.

2.2.5 PatOIS's High-Level Description Language

Alikacem et al. [1] propose a High-Level Description Language for source-code representation based on a language-independent meta-model. This language was developed to extract from the source code necessary information for the calculation of metrics. The developed meta-model was designed to support OO languages and their programming concepts.

The model has three categories of concepts: Common concepts, Variable concepts, and Specific Concepts. Common concepts are those that are repeated across all supported languages, such as classes, methods, and attributes. Variable concepts have similar syntax but can have some differences in the semantics. The most notorious concept of this category is inheritance. To solve this problem, the methods and attributes are duplicated in the subclasses. Finally, specific concepts only exist in a particular language, for example, the concept "Entity" represents the idea of abstraction, and it is then specialized in Interface, Union, or Template.

The code representation models at an entity level, and it is generated by a module named parsing and mapping, so to add a new language, a new module must be created for that language. However, new concepts introduced with new languages releases require few changes and can be easily added.

2.2.6 Model's Comparison

The models previously described model the source code at two different levels of detail. PatOIS and FAMIX model at a program entity level, while LARA, eCST and, MOON model at the AST level. While these two groups share several similarities, there are still some differences to be aware of.

Both PatOIS and FAMIX define a meta-model that focuses primarily on object-oriented programming key concepts. This includes information about class inheritance, method invocation, and attribute access. However PatOIS model offers more information regarding languages' specific concepts such as Interface in Java or Union in C++.

The second group uses a more general approach that represents all aspects of source code, not only OO design aspects. The main difference is how this was achieved. eCST is an extension of and Concrete Syntax Tree that is enriched with universal nodes that are independent of the input language. LARA uses an abstract language model that represent points of interest (i.e., join points) and properties (i.e., attributes) of the code of the target language, on top of language-specific ASTs. Finally, MOON uses a hierarchical model to represent the objects, the relations, and the instructions.

2.3 Approaches for Language independent Code Analysis

2.3.1 SMIILE

Rakić and Budimac [29] analyzed the state of current software analysis tools and proposed a prototype that would mitigate the weaknesses of academic and market technologies.

The authors pointed out that most tools are not language-independent. Therefore, programmers have to use different tools in different projects and in projects that use more than one language. They also point out the lack of a graphical interface so that even someone not specialized in the area can understand the results and the lack of suggestions for refactoring the code. Thus, taking into account the disadvantages of the tools, they proposed a framework with the following objectives: platform and language independence, support for a wide variety of metrics, interpretation of the results, and proposal of recommendations for the user.

The tool, SMIILE [14], has three main steps (Figure 2.4). First, it would be necessary to create an intermediate representation of the source code structure, an Enriched Concrete Syntax Tree (eCST), described in Section 2.2.2. The second step follows, where different metrics are calculated, giving special attention to object-oriented metrics, and the generated values are then stored in XML format. Finally, algorithms and heuristics are applied to the calculated metrics to provide the user with information to improve the source code.

In the article [3], the authors present the tool's results after adding support for five different languages: Java, C#, Modula-2, Pascal, and COBOL. They only implemented the metrics of Cyclomatic Complexity (CC) and Lines of Code (LOC). However, they refer that the implementation of new metrics such as Halstead or object-oriented metrics would be easily achieved.

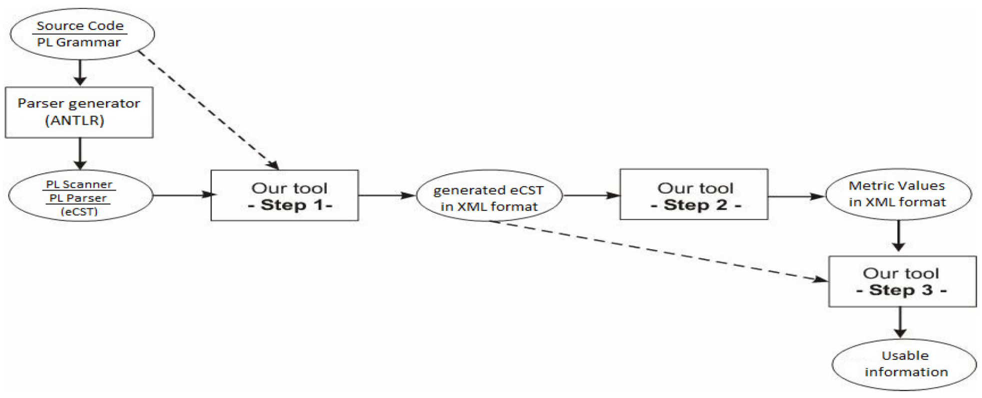

Figure 2.4: SMIILE Architecture as presented by Rakić and Budimac [29]

2.3.2 Moose Reengineering Environment

The Moose Reengineering Environment [21] provides a set of extendable tools and methodologies to re-engineer sizeable industrial software systems. This environment, written in Smalltalk and based on the FAMIX meta-model (Section 2.2.3), provides a metrics engine that allows to calculate a vast number of metrics in a language-independent way. They distinguish three types of metrics: class, method, and attribute metrics.

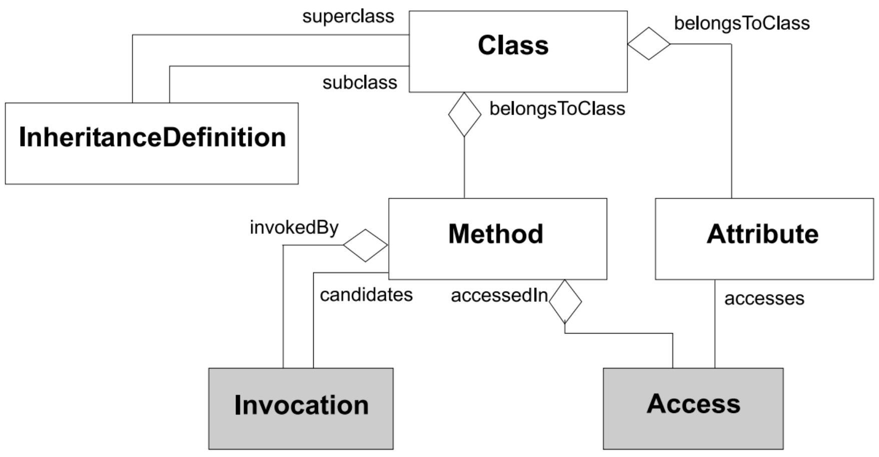

Figure 2.5: A simplified view of the FAMIX metamodel as presented by Lanza and Ducasse [21]

The FAMIX meta-model is stored in a graph, depicted in Figure 2.5, that is traversed to construct and calculate metrics. Three generic metrics are used to calculate more complex and specific measures. These are Node Count (NC), Edge Count (EC), and Path Length (PL). The first two metrics are used to calculate nodes and edges. NodeCount is defined as the number of nodes connected to a particular type of node over a specific edge, for example, the number of methods associated with a certain class. Edge Count is used to measure the number of edges connected to a node. This can be used to calculate the number of access of an attribute by a method. Finally, Path Length defines the length of the chain of edges, starting in a specific node. For instance, this can be used to calculate the depth of inheritance of a specific class.

The main strengths of this approach are the flexibility of the system, which allows the implementation of new metrics as needed and the integration of a software visualization tool, which allows to represent them. However, due to the meta-model chosen, certain metrics related to the control flow of the problem and specific OO metrics cannot be implemented.

2.3.3 ATHENA

ATHENA [9] is an automated tool implemented in C, exclusively for the UNIX operating system, aimed at assuring the quality of software by performing a wide range of software quality metrics. The authors chose to highlight the framework's following functionalities and characteristics: a measurement kernel which supports a wide range of metrics; language independence, which was achieved with the support of several languages such as Pascal and C; high customizability, with the capacity to add new metrics; generation of a report with information about the calculated software's measurements.

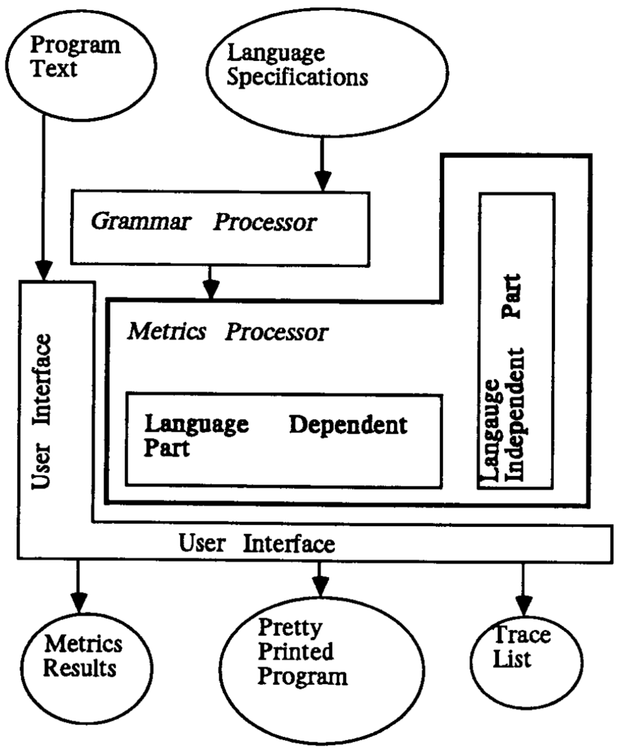

Figure 2.6: ATHENA's Measurement Kernel as presented by Christodoulakis et al. [9]

The architecture of ATHENA is divided into three main components (see Figure 2.6): The grammar processor, the metrics processor, and the user interface. The grammar processor generates a new abstract syntax tree for every supported language, at the cost of requiring the implementation of a new parser for different target languages. The generate tree is structured so that metrics can be easily calculated. The measurement of metrics is performed in the metrics processor, and it operates on the trees and graphs previously generated. The calculation of metrics is divided into two sections, the language-dependent part is used to calculate metrics that are dependent on the syntax and semantic of the language. On the other hand, the language-agnostic section focuses on more general procedures such as Lines Of Code (LOC). Ultimately, the User Interface prints graphical the obtained results by the previous component.

This system was classified as highly flexible and adaptable, being able to process new inputs and be extended with new metrics. However, more details about this tool and its development are not available anymore.

2.3.4 A Refactoring Framework based on the MOON Meta Model

The MOON meta-model [10] was designed to support the possibility of a system that would execute refactoring in a language-independent way. However, to perform those desired refactorizations it is required first to use metrics to detect the bad smells in source code. This framework mainly focuses on class and method metrics. To calculate these metrics, it is not only necessary to collect basic information of any OO language, such as classes, inheritance, methods, and attributes, but also information about the instruction's control flow.

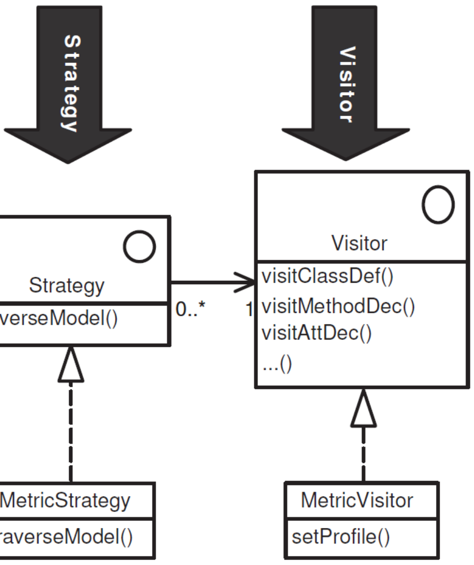

Figure 2.7: MOON's Metric Engine Core as presented by Crespo et al. [10]

The authors classify the metrics based on their granularity level: system, class, and method. Therefore to traverse the model and measure different element properties, the design patterns Visitor and Strategy were implemented (see Figure 2.7). Visitor was implemented to avoid adding a new method every time there is a need to make a new operation with all of them and to preserve the meta-model definition. The Strategy design pattern was used to define the traversal algorithm independently of the visitor. The measure calculation is performed by visiting each element of the meta-model and obtaining the desired metrics.

The significant advantage of this technique is the reuse of the framework. It currently supports a vast number of metrics and languages, however, it has the possibility of extending many more of both the former and latter. Nevertheless, it requires a new parser for any language added. Some metrics related to conditional and loop sentences cannot be implemented in a language-agnostic way. Finally, it does not present a graphical representation to interpret the obtained results.

2.3.5 PatOIS

Alikacem and Sahraoui [1] proposed a framework to calculate source code metrics. With this tool, the authors wanted to tackle several problems with current tools. Namely, the lack of formalization, with the same metric being implemented in several different ways. Moreover, most tools are black boxes that do not allow to extend to support more metrics.

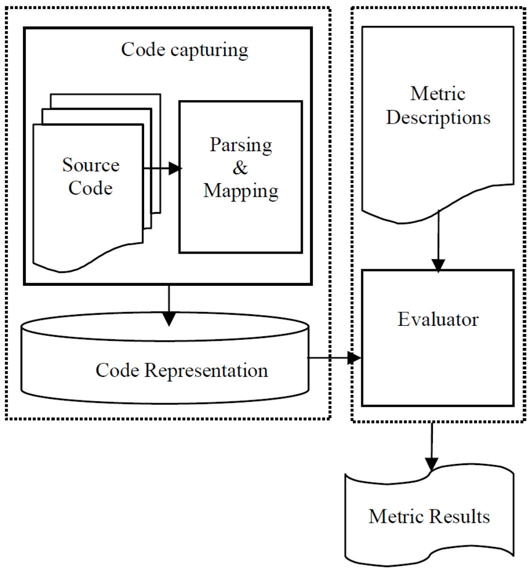

Figure 2.8: Architecture of PatOIS's metric extraction framework as proposed by Alikacem and Sahraoui [1]

The architecture of this approach, shown in Figure 2.8, uses two main mechanisms. The first is a source code mechanism described in section 2.2.5. The second is a declarative language named PatOIS (for Primitives, Operations, Iterator, and acceSsors) to report metrics. To obtain metrics, four mechanisms were developed to access and perform operations on the represented data. The first mechanism is Selection. It allows selecting the instances of a particular element, such as classes of a program. Filtering allows the examination of all elements that possess specific qualifying criteria, such as all public attributes of a class. Navigation allows to explore and iterate the properties and relations of an element. For example, iterate the set of methods of a particular class. Finally, Operations allow to perform specific operations to a set of elements to get metrics values. For instance, to get the number of parameters of a method.

The calculation of metrics requires a parser to generate an AST and a mapping module to build the representation based on the meta-model. The framework currently supports two languages and thirty-five metrics. However, the approach taken allows the specifications of more metrics that can be calculated on programs written in other languages. Nonetheless, due to the approach taken to generate the meta-model, it does not support metrics to measure the complexity of control flow.

2.3.6 AMT

AMT [19] or automated metrics computation tool was designed to evaluate object-oriented systems' quality attributes. Due to the constant evolution of software processes, there is a need for a tool that can gather metrics data. However, it must be easily modified and extended to support new and evolving metrics across multiple languages. The proposed tool has two main mechanisms: representation of the code using XML and metric collection, which uses the generated XML as input to collect each metric.

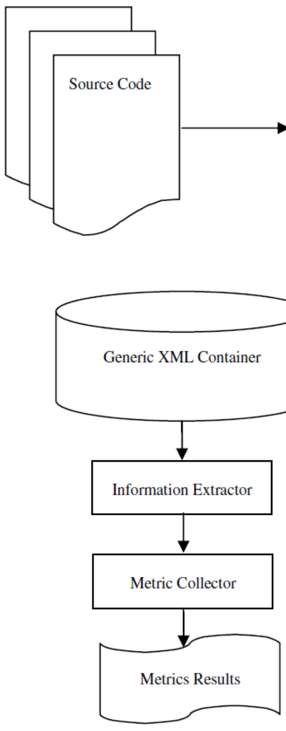

Figure 2.9: AMT framework as proposed by Kayarvizhy and Kanmani. [19]

The framework, as shown in Figure 2.9, has multiples modules. The Parser generates from the source code OO constructs such as classes, methods, and attributes. The XML Converter acts on the generated constructs and formats them by creating a standardized XML file. The Information Extractor from XML parse the XML file and extract the necessary information to create a data structure later used for metric computation. The last two modules are Metrics Calculator and Metrics Listing, which, as the name suggests, calculate the metrics and provide the user with a GUI where the cohesion coupling and inheritance metrics are categorized by levels of granularity.

The main objective of this tool was achieved by creating a framework the gives a high level of automation for obtaining relevant metrics in addition to support the addition of more metrics and languages. Currently, it can analyze Java and C# source code. However, to add support to more languages involves getting a new parser for that language.

2.3.7 Discussion

To compare the six approaches presented in previous sections the tools were analyzed according to two groups of criteria. The first group of criteria focuses on the properties of the approach, e.g., platform independency, languages support, the effort required to add support for more languages, if it has a graphical interface to display the obtained results, interpretation of calculated values and suggestions for improvements and at what level it models the source code. These proprieties are presented in Table 2.4

The second is a comparison of the supported metrics. Table 2.3 includes the following metrics:

- LOC - Lines of Code (see Section 2.1.1) measure the size of the program.
- CC - Cyclomatic Complexity (see Section 2.1.2) represents the program complexity using the control flow graph of the program.
- CK metrics - Chidamber and Kemerer Metrics (see Section 2.1.4.1) are OO metrics related to the coupling and cohesion of the program
- LH metrics - Li and Henry's Metrics (see Section 2.1.4.2) OO maintenance metrics.
- Other OO metrics - Other OO metrics different from CK and LH metrics.
- Halstead metrics - (see Section 2.1.3) measure the complexity of code using operands and operators.
- Other non-OO metrics - Other metrics that do not belong in the previous categories.

Approach	**LOC**	**CC**	**CK metrics**	**LH metrics**	**Other OO metrics**	**Halstead metrics**	**Other non-OO metrics**
SMIILE [29]	Yes	Yes	NOC, WMC	NOM	No	No	No
Moose [21]	Yes	No	NOC, WMC	NOM, SIZE2	Yes	No	No
ATHENA [9]	No	Yes	No	No	No	Yes	Yes
MOON [10]	No	Yes	NOC, DIT	No	No	No	No
PatOIS [1]	No	No	All	NOM, SIZE2	Yes	No	No
AMT [19]	No	No	All	DAC, NOM, SIZE2	Yes	No	No

Table 2.3: Source Code Aproaches Metric's Support

Almost all the analyzed tools heavily focus on OO metrics, with ATHENA being the only tool that does not focus on that paradigm. Java is the most supported language. There are two groups of tools, the ones that model at the program entity level and those that model at the AST level. This makes it to be generally easier to add new languages for the former approaches. However, the latter model saves more information about the source code at the cost of having to implement a parser or a grammar. LARA occupies a middle ground between the two approaches, by allowing the reuse of existing parsers, e.g., Java and C/C++ weaver use existing parsers, Spoon and Clang. Most tools already have a graphical interface that displays the metrics, but only one present interpretation of their meaning.

Approach	**Plat. Ind.**	**Lang. Supp.**	**Effort to add new languages**	**Graph. Rep.**	**Interp./ Improv.**	**Source Code Model Level**
SMIILE [29]	Yes	Java, C#, Module-2, Pascal, COBOL	It is necessary to define the language grammar. The ANTLR parser generator accepts language grammar as its input and produces language scanner, parser, AST and CST.	No	No	AST
Moose [21]	Yes	C++, Java, Smalltalk	It is required a new tool to extract FAMIX-based information for every language supported.	Yes	No	Program Entity
ATHENA [9]	No	Pascal, C, COBOL	For every language supported it is necessary a new grammar processor. Which processes language specification and produces the environment for the metrics processor.	Yes	No	AST
MOON [10]	Yes	Java, Eiffel	It requires a new parser to convert source code to the meta model for every new language.	No	Yes	AST
PatOIS [1]	Yes	C++, C#, Eiffel, Java	It requires a module, named parsing & mapping, that maps a program to a representation that conforms to the generic meta-model described.	Yes	No	Program Entity
AMT [19]	Yes	Java C#	It requires a new parser to break the source code into object-oriented constructs. Parsing functionality for languages is generally available as part of the compiler.	Yes	No	Program Entity

Table 2.4: Source Code Aproaches Comparison

2.4 Software Metrics tools

There are a large number of tools available to calculate different software metrics. This section presents open source tools as well as commercial tools for software metrics calculation. This analysis does not focus on tools that analyze a single language or that only perform LOC analysis.

2.4.1 Understand

Understand is a tool for source code metrics and quality analysis supporting object-oriented languages (C#, C++, Java), procedural languages (Ada, Basic, C, Pascal), as well as web languages

(CSS, HTML, JavaScript, PHP, and XML). It can also handle code bases written in more than one language. The metrics supported are divided inyo three groups: Complexity Metrics (e.g., McCabe Cyclomatic), Volume Metrics (e.g, Lines of Code), Object Oriented (e.g., Coupling Between Object Classes). The metrics can be calculated by function, class, file, and project. Understand is a standalone executable, and although it is a paid tool, the developers offer a 15-day trial.

2.4.2 CCCC

CCCC is a tool for the analysis of source code. It supports C/C++ and Java. The measures extracted are size, complexity, as well as object oriented metrics and, are displayed in a report in HTML format. The tool was developed as freeware and is released in source code form. The users are encouraged to compile the program themselves, however, an executable is also provided.

2.4.3 Analizo

Analizo is an open source tool for extensible source code analysis. It supports C, C ++ and Java, and extracts and calculate size, complexity metrics, and CK metrics as well generates dependency graphs and analyzes the project's versions by producing an evolution matrix. Analizo completely free and is available as a Debian package.

2.4.4 Rational Software Analyzer

Rational Software Analyzer is a static analysis tool for static analysis code reviews and bug detection. It features a wide set of analysis rules that enable the identification of code-level issues early in the software development life cycle. The supported languages are Java and C/C++, and it can calculate more than 40 metrics, including size, complexity, and cohesion. Even though Rational Software Analyzer is a paid proprietary software, the developers offer a demo.

2.4.5 Krakatau

Krakatau provides a command line metrics tool for C/C++ and Java software projects. It supports line counting, complexity, Halstead and object-oriented metrics at the project, file, class, interface, and method level. Reports can be generate in CSV, HTML and XML formats. This is also a paid tool but has a trial version available

2.4.6 Sonarqube

Sonarqube is an open-source automatic review tool for continuous inspection of code, which includes static analysis of code for up to 27 languages. This analysis allows for the detection of bugs, code smells, duplicated code, and security vulnerabilities. It can be integrated into an existing workflow in order to enable continuous code inspections, metrics history and evolution graphs. Sonarqube has a free Community Edition with only 15 languages and several paid commercial options.

2.4.7 BetterCodeHub

Better Code Hub is a cloud-based source code analysis service that checks a codebase against ten good software development guidelines that guarantee that the code will be easy to maintain and extend. These guidelines evaluate the size, complexity, modularity, cohesion, coupling of code, as well as, verifying that the is no duplicated or over-commented code. There is support for 17 programming languages. Better Code Hub is free for open-source projects and has a paid plan for private repositories.

2.4.8 Tools Comparison

The tools comparison focus on two groups of criteria, proprieties of the tool and the metrics supported. As seen in Table 2.5, there are four proprietary tools and three open-source tools. The proprietary tools do not allow the extension with new capabilities, however, they have a warranty from their creators. These tools, while paid, offer a trial version. On the other hand the open source tools, not only allow to check how the metrics are calculated but also to add new measures, besides being free to use. All frameworks have support for Java, and almost all have support for C and C++ (Sonarqube Community Edition does not support C/C++, however all the other paid versions of Sonarqube support it). Table 2.6 shows the metrics supported per tool. The analyzed tools heavily focus on measuring the size and complexity of the projects, with all supporting the metrics LOC and CC. Due to the analyzed languages, with primary focus on Java and C++, all tools support some OO metrics.

Tool	**Software Distribution**	**Total Lang. Supported**	**Top 10 Languages**
Understand	Proprietary	19	C, C++, C#, Java, JavaScript, PHP, Python, TypeScript
CCCC	Open Source	4	C, C++, Java
Analizo	Open Source	4	C, C++, C#, Java
Rational Software Analyzer	Proprietary	3	C, C++, Java
Krakatau	Proprietary	3	C, C++, Java
Sonarqube Community Edition	Open Source	15	C#, Java, JavaScript, PHP, Python, TypeScript
BetterCodeHub	Proprietary	17	C++, C#, Java, JavaScript, PHP, Python, Ruby, Shell, TypeScript

Table 2.5: Tools Proprieties

Tool	**LOC**	**CC**	**CK metrics**	**LH metrics**	**Other OO metrics**	**Halstead metrics**	**Other non-OO metrics**
Understand	Yes	Yes	All	NOM, SIZE1, SIZE2	Yes	No	Yes
CCCC	Yes	Yes	WMC	None	Yes	No	Yes
Analizo	Yes	Yes	All,	NOM	Yes	No	No
Krakatau	Yes	Yes	All	NOM, SIZE2	Yes	Yes	Yes
Rational Software Analyzer	Yes	Yes	N.A. [a]	N.A. [a]	N.A. [a]	Yes	Yes

[a] Rational Software Analyzer does not specifies the supported OO metrics

Table 2.6: Tools Metrics Comparison

2.5 Conclusions

This chapter provided a review of the most important aspects of source code analysis. Relatively to the metrics, most tools heavily focus on OO metrics, with particular attention given to CK metrics. To model the source code, most approaches focus on two levels of representation AST and program entity level. However, while it is generally easier to parse the source code and add support for more languages on the latter level, the former allows for a more in-depth analysis. Finally, a large number of tools are available to the user, Java is the most supported language, and while most tools support some kind of graphical representation of the metrics, they do not provide a way to interpret the results, so that the final user can improve the current state of the project.

Chapter 3

Multi-Language Approach for Metrics Calculation

This chapter describes the design of a multi-language approach for metrics calculation. This chapter starts by defining the goals of the tool. Section 3.2 describes LARA's components, which were used as the basis for the developed framework. Section 3.3 describes LARA Common Language, a model that maps language-specific nodes to common join points. Finally, Section 3.4 describes LARAM, a library written in LARA that uses the LARA Common Language to calculate metrics in multiple languages.

3.1 Introduction

This dissertation's primary goal is to design an approach based on the LARA framework that is capable of calculating metrics in multiple languages. However, to accomplish this goal, during the design stage, several other objectives were set. These objectives aim to overcome some of the limitations of the current tools. Firstly, the tool must be flexible enough to adapt to specific features and the various degrees of complexity of the languages required to analyze while also allowing to add new languages in a straightforward manner. The level of information extracted from the source code must provide the highest level of detail, i.e., an abstract syntax tree, preferably annotated with source code information. It has to support various types of metrics such as size, complexity, or object-oriented metrics, with various granularity levels: project, file, class, and function levels. Finally, the calculated results must be exported to a format that the user can understand and that can be used to compare with other tools.

3.2 LARA framework

LARA boasts a large codebase with 5.570 code files and 766.523 lines of code, mostly of Java code. Due to its size and complexity, this presented a steep learning curve requiring initial familiarization with several components before starting the phase of design and implementation of

LARAM. Some components are specific to LARA compilers for a particular programming language, while others are shared and used by all LARA compilers. Each LARA compiler supports a specific language (or set of languages), and in this work, we have used the following LARA compilers:

- Clava [2] is a C/C++ source-to-source LARA compiler with 3.321 code files and 527.706 lines of code. It uses Clang to parse code and uses a custom AST similar to Clang's own AST, but that provides source-to-source capabilities. The submodules relevant to the presented framework are ClavaAst and ClavaLaraApi. ClavaAst holds the classes of all the nodes in the Clava's AST and can be used to represent C/C++ code. ClavaLaraApi holds the APIs that can be used in LARA that are specific to C/C++ and is mostly composed of LARA scripts.

- Kadabra is a Java source-to-source LARA compiler with 436 code files and 58.066 lines of code. Kadabra uses an existing Java-to-Java compiler, Spoon, to build the AST. The submodule relevant to the presented framework is KadabraLaraApi, which holds the Java-specific APIs used in LARA.

- Jackdaw is a JavaScript source-to-source LARA compiler with 189 code files and 27.598 lines of code. Jackdaw uses Esprima to parse JavaScript code and to generate an AST.

LARA also has a set of tools and APIs to enable LARA compilers' development for additional programming languages, e.g., the Weaver Generator. The main APIs used in this project were `lara.System`, which allows to measure LARA scripts' execution time, `lara.Io`, which has utility methods related with input/output operations on files, and `weaver.Query`, a class for selection of join points that allows to select all join points of a given type from the AST root or from a certain point, while also allowing the application of filters and chain of queries.

3.2.1 Using LARA join points to calculate metrics

Since each LARA compiler already provides join points for representing AST nodes, why not use those join points to calculate metrics? To understand the problem of defining multi-language metrics this way, let us compare the steps required to obtain the superclass(es) of a class for all three compilers. Table 3.1 compares the join point representing a class, the attribute required to obtain the classes that the original class extends as well as the type that is returned. As can be seen, the join point and the attribute both have two different nomenclatures in different languages, and the corresponding attribute returns a different type in each case, i.e., an array, a string, and a join point for Clava, Kadabra, and Jackdaw, respectively.

The first solution that was considered was to implement a small set of queries for each LARA compiler. Each query would have the same name and return type. For example, Listings 3.1 and 3.2 show the query's implementation to obtain the classes that extends a class for Clava and

Compiler	Class join point	Superclass(es) attribute	What is returned
Clava	`class`	`bases`	An array with the classes that the original class extends
Kadabra	`class`	`superClass`	A string with the name of the class that the original class extends
Jackdaw	`classDeclaration`	`superClass`	A join point of the type `Identifier`

Table 3.1: Comparisons of join points in various compilers

Kadabra join points. However, this solution was discarded since we considered it was impractical to implement the required queries without the help of some kind of structure or systematic approach.

```
var MetricQuery = {};

MetricQuery.getSuperClasses = function($class) {
    var superClassName = $class.superClass;
    return Query.search("class",superClassName).get();
}
```

Listing 3.1: Implementation of MetricQuery for Kadabra join points

```
var MetricQuery = {};

MetricQuery.getSuperClasses = function($class) {
  return $class.bases;
}
```

Listing 3.2: Implementation of MetricQuery for Clava join points

3.3 LARA Common Language

As seen in the previous section, even when using each LARA compiler's join points, we need to write several versions of the same metric. This happens because each LARA compiler is an independent project that implements its own language specification. The LARA compilers being decoupled from each other has its advantages since this allows independent development of each compiler, which can adapt its language specification to the particularities of the languages it supports. Even if two programming languages (e.g., C++ and Java) share the same paradigm (e.g., OO), it is common for each language to have its own conventions (e.g., C++ functions are methods in Java) or even features (e.g., C++ supports multiple inheritance, Java supports single inheritance).

In order for two LARA compilers to have support for the same LARA code, both compilers need to support the same join points and attributes that are used in the LARA script, and the attributes must have the same interface. A solution to this could be to make the language specification of each LARA compiler support a common set of join points and attributes. However, we would need a way to enforce a Common Language Specification between several LARA compilers, which would pollute the language specification of each compiler (e.g., the LARA compiler for Java would support both the join point `method` and `function`), and it could introduce changes that could break backward compatibility.

In this thesis, we extended the LARA framework to solve this problem so that it supports additional language specifications that can be shared between LARA compilers. These language specifications are opt-in, and are enabled by a simple import, which replaces the original language specification of the LARA compiler. These language specifications work as a virtual AST, which provides the same join points and attributes across the LARA compilers that support them.

This approach presents several challenges. First, it needs a unified way to handle different ASTs. It is then necessary to define a common set of join points and attributes for a language specification that will be shared between several languages. Finally, those join points have to be mapped to the nodes of each AST.

3.3.1 Interacting with different compilers

Since every language can use its own AST, we created the Java interface `AstMethods`, that allows to interact with different ASTs that have been developed independently. By having this information centralized in this class, the LARA code only needs to communicate with this Java class to traverse any AST. The interface has the following methods:

`Object toJavaJoinPoint(Object node)` - Converts an AST node into the Java join point of the original language specification of the LARA compiler. This can be useful if we want to reuse functionality already implemented by the LARA compiler.

`String getJoinPointName(Object node)` - Maps an AST node to the type of the corresponding Common Language Specification join point.

`Object getChildren(Object node)` - Gets the children of an AST node

`Object getRoot()` - Gets the root node of the AST

`Object getParent(Object node)` - Gets the parent of an AST node

`Object[] getDescendants(Object node)` - Gets the descendants of an AST node

`Object[] getScopeChildren(Object node)` - Gets the join points inside the scope of this AST node (e.g., body of a loop)

For every compiler it was necessary to implement the methods of the interface `AstMethods`. As can be seen in Figure 3.1, it was created an implementation of this interface for every compiler, `ClavaAstMethods`, `KadabraAstMethods` and `JackdawAstMethods`. Since this class acts as a bridge of communication between Java and the JavaScript engine that runs the LARA scripts, it was also needed to handle the conversions between the two languages. For example, each method that is called by JavaScript (e.g., `AstMethods.getChildren()`) has a Java equivalent implementation (`AAstMethods.getChildrenImpl()`). This allows that the methods that are

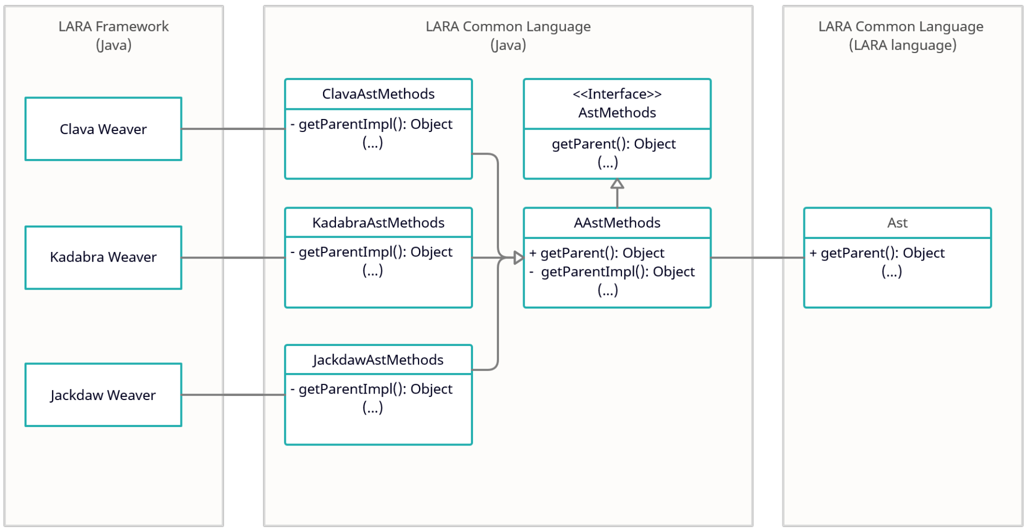
LARA Framework (Java)
Clava Weaver
Kadabra Weaver
Jackdaw Weaver
LARA Common Language (Java)
ClavaAstMethods
- getParentImpl(): Object (...)
KadabraAstMethods
- getParentImpl(): Object (...)
JackdawAstMethods
- getParentImpl(): Object (...)
<<Interface>> AstMethods
getParent(): Object (...)
AAstMethods
+ getParent(): Object
- getParentImpl(): Object (...)
LARA Common Language (LARA language)
Ast
+ getParent(): Object (...)

Figure 3.1: Interface of communication between each compiler and LARA

directly called by JavaScript to only have `Object` in the signature, since it can be a node of any AST, while the methods called directly in Java are allowed to have safe Java types of the specific AST being handled. It also handles conversions between Java and JavaScript, such as converting Java arrays and lists to JavaScript arrays and converting nulls to undefined. Finally, we implemented the class `AST` in LARA, which is the layer that interfaces the `AstMethods` object with the rest of the LARA code.

Since the `AstMethods` interface defines which join points exist and how they are traversed, each language specification that is to be shared between several LARA compilers requires its own implementation of `AstMethods`, for each LARA compiler that we want to support.

3.3.2 Common Model

Now that we already have a way to access the AST of any language, as long as there is an implementation of `AstMethods` for that language specification, it is necessary to define common join points and map them to the nodes of any AST. To define the join points and its attributes we reused the system of the LARA framework that is already being used to define the language specifications of the LARA compilers. This system requires three components. In the first component, the *join point model* specifies the join points and the relations of inheritance between them, e.g., a `class` extends a `decl`. The second, the *attribute model* defines the attributes of the join points and their interface, e.g., the name of a variable or the signature of a method, as `Strings`. Finally, the *action model* defines actions that can change the AST, e.g., code insertions. However, this last model was not used, since this project's focus was the implementation of code metrics that did not required altering the AST. The definition of join points and attributes was done in an iterative manner, since it was not necessary to immediately define the complete model. This allowed the work to progress and be tested very rapidly, as with a few join points, it was already possible to obtain relevant information, e.g., with the definition of join points `class` and `method` it is already possible to determine the number of methods per class.

In this model, every element is a join point, Figure 3.2 shows the join points of the language model we implemented for OO languages, as well as the relations of inheritance between the join points[1]. The root element is a `program`, which usually contains a set of files as its children. If a specific join point is not specified, queries start at this point. The `file` represents a source file, which usually contains the remaining elements. A `stmt` is the base join point of all statement types, such as declarations or conditional statements. The `decl` represents a declaration or definition, (e.g., a class, a variable, a function). The `vardecl` represents declared variables, which if declared inside a class become a `field`, or a `param` if the variable is part of a function signature. The `expr` represent expressions that can be evaluated to determine a value, such as a function call, a reference to a variable, an operator call, or a lambda expression. A `call` can be a `memberCall`, which can also specialize into a `constructorCall`. `varRefs` represent access to variables in the code, which become `fieldRefs` if the variables being used are fields of a class. Finally `type` defines the user defined types as well as primitive types from variables.

Note that the model does not define, for each join point, what join points can be selected, or the children that each join point has access to. This was a design decision that allowed to have a more flexible model that required less implementation effort and mapped more easily between different languages. However, this means that queries should be done based on the descendants of a join point (the default query search). Queries based on the children of a join point are not guaranteed to be consistent between LARA compilers.

This also means that this model works as a *virtual AST* that is built by traversing the AST of each language during a query. Since each AST can have its own organization, this way we do not need to know if `Stmt` join points are direct children of `File`, only that we can find statements inside files in a given source code.

This model's join points of are associated with concepts related to imperative and object-oriented languages, since these where the paradigms of the languages already supported by LARA compilers, and also because we chose to implement metrics focused on these paradigms. However, the concept of a virtual AST can be expanded to other paradigms, e.g., logic programming. In this case, we could create join points such as `fact` or `clause`, which could be added to the presented model, or could be part of a new language specification focused on this paradigm.

Now that we have the model's specification, the next step is to map the nodes for each language to the common join points. For this purpose was created a class for each language. The created class uses a specialized Java map, `FunctionClassMap`[2], which accepts Java classes as keys, and functions as values. We used this class to map a Java class that represents an AST node to the name of a common join point. Therefore, in this map the node's class is used as the key, while the value is a function that receives an instance of the node and returns a string with the name of the

[1]Documentation was also generated for the full specification and it is available at `http://specs.fe.up.pt/tools/lcl/language_specification.html`

[2]`https://github.com/specs-feup/specs-java-libs/blob/master/SpecsUtils/src/pt/up/fe/specs/util/classmap/FunctionClassMap.java`

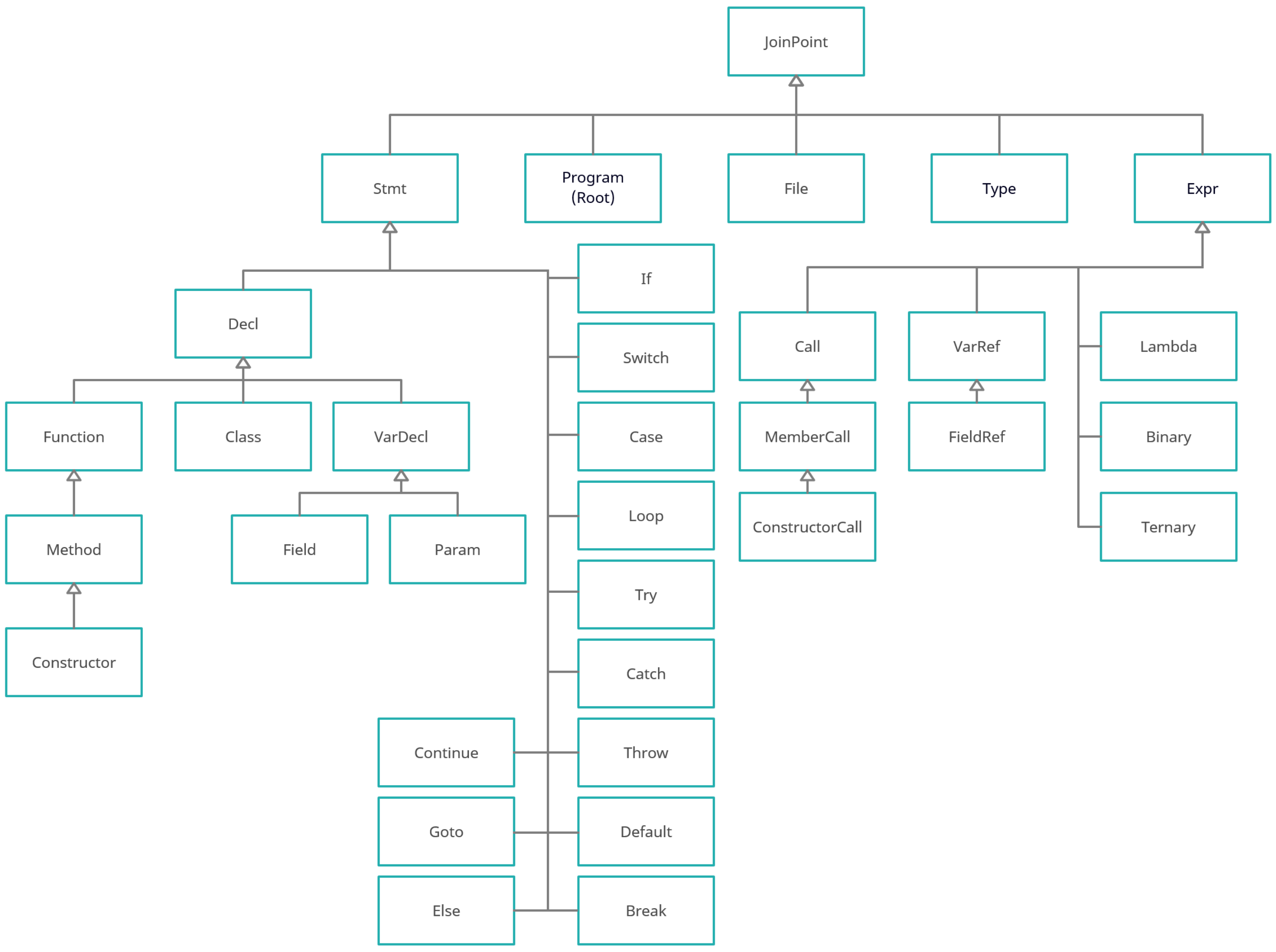

Figure 3.2: LARA Common Language Join Point Model

corresponding common join point. We used this map that accepts functions as we might need to access information about the node before returning the type of common join point. For instance, in Spoon, used by the Java compiler, the node `CtCase` is used by both the case statements and the default case statement, and we need to check if it has no corresponding expression to determine it is a default case statement. More importantly, this map also respects the hierarchy of Java classes. For example, also in Spoon, every node extends the class `CtElement`, therefore `CtElement` was mapped to the default common join point `JoinPoint`. If the compiler tries to get the value of a class that does not exist yet in the map, since that class must extend `CtElement`, the map will return the closest join point name, which in this case will be `JoinPoint`.

After defining the specification of the model and having mapped the AST nodes to the corresponding join point name, we need to create the join points themselves. When building a LARA compiler, we first define a language specification, and then we use the `WeaverGenerator`, provided by the LARA framework, that automatically generates Java code with an initial implementation of the LARA compiler. The generator creates, among other files, Java classes with skeleton implementations for each of the join points. In a traditional LARA compiler, these classes are used as *wrappers* around the AST nodes, which provide a layer of abstraction that allow to use a similar interface to manipulate the nodes of very different ASTs from LARA scripts. We reused this idea for our approach, but instead of using the language specification to generate a Java wrapper for each join point, we generate LARA wrappers. The LARA wrappers (which are mostly JavaScript) provided a flexible environment over which we implemented a solution that, when using the alternative language specification of the approach we propose, allows us to write LARA scripts in the same way as when we write scripts for the original language specification of the LARA compiler[3].

To maintain this compatibility, we automatically generate a LARA class for each join point where join point attributes with parameters are accessed as JavaScript functions, while attributes without parameters are accessed as JavaScript attributes. Since most join point attributes need an implementation for each language, by default, they throw an exception. The exceptions to this rule are attributes that can be implemented using the interface provided by the LARA class `AST`. Since this interface is shared by all common language specifications and LARA compilers, we can automatically provide an implementation (e.g., `joinPointType`, `instanceOf`, `ancestor`).

The base class join point also stores as a field the original Java AST node in order to obtain information about the node. So, for every target language it is necessary to override the methods for each attribute, since each compiler implements the attribute in its own way. For instance, Listings 3.3 show the implementation of the attribute `function` from join point `call` for C++.

```
_lara_dummy_ = Object.defineProperty(CallJp.prototype, 'function', {
  get: function () {
    var functionDecl = this.astNode.getFunctionDecl();
    if(functionDecl.isPresent())
      return CommonJoinPoints.toJoinPoint(functionDecl.get());
    else
      return null;
  }
});
```

Listing 3.3: Attribute `function` of common join point `call` for C++

3.3.3 Programming Language Peculiarities

Every single programming language was designed with a different set of goals in mind, which could be to optimize performance, make programming more safe, or do a specific task. In practice, this means that each language and compiler has some peculiarities that make them slightly different from each other, even if they use the same programming paradigm. For instance, C++, due to its roots in C, heavily depends on function declarations specified in header files, which have corresponding function definitions in separate files. This means that when searching for functions in C++, they can appear as if they are duplicated when compared with searching for the same

[3]This is true as long as the LARA script uses the API `weaver.Query` to perform queries, instead of the keyword `select`

functions in Java code. Therefore when designing a system that analyzes multiple languages, it is important that the system is flexible enough to handle these differences. We compiled below a collection of quirks we found during this work and the respective implemented solution, sorted by language/compiler.

Starting with Java and Spoon:

Spoon does not have a `Program` node Spoon does not have a node to represent a program, which aggregates the source files. The solution was to extend `CtElement` and create a new Spoon node, `CtApp`. This node has a single method that returns all the nodes that represent a source file. We considered an alternative solution that would maintain Spoon's approach, of having separate ASTs for each source file. However, this would require changing both the interface `AstMethods` and the query system, and the idea was discarded.

Methods as Functions In Java, there are no functions outside of a class, so every function defined in Java has to be a method. Since the join point `method` extends `function`, if a query asks for all the functions in the Java source code, the query will return all the methods (which are also functions).

Constructors as Methods Spoon AST does not classify constructors as methods. However, using the `FunctionClassMap` we were able to wrap Spoon constructors with a Constructor join point, and maintain this hierarchy in LARA.

The following are relative to C++ and Clang:

Declarations and Definitions In C++, every class and function can be declared and defined in different files. Since declarations do not provide as much information as definitions, when a node of a class or function is mapped to a join point, the definition is chosen over the declaration whenever possible. Another problem caused by having a definition and declaration in the language is that when a query is made for all classes or functions, certain results will appear duplicated. To solve this, the `weaver.Query` API is patched in Clava to, when searching for these join points, only return the definition if both definitions and declarations are present for the same class or function.

Super calls Base class constructors are automatically called if they do not have arguments. However, to call the constructors with arguments, it is required to use the constructor initialization list. Other languages that do not support multiple inheritance use the keyword `super`. So it was necessary to create an attribute `superCalls`, which returns an array with calls to all constructors.

Relative to JavaScript and Esprima:

Lack of Information JavaScript is a dynamically typed language, which means that it is not possible to generally know the types of variables. The AST provided by Esprima also does not have information about the types, nor ways to obtain it. So, if the JavaScript source code calls a method, it is not possible to know the class that method belongs to. This limits the information that is possible to retrieve from the AST, and it was not possible to implement join points related to types, fields or methods.

Finally, the following show some general problems and corresponding solution that applied to all languages:

Identify a function/class We defined an attribute `id` for several join points, including `method` and `class`, since this is necessary for certain tasks. We had to define signatures for each of these nodes. For classes, they are identified by the namespace and name in C++, the package and name in Java, and only the name in JavaScript.

Handling Types Types can be composed of other types, e.g., an array of integers, generic classes in Java, class templates in C++. Clang and Spoon use different strategies to represent types. Spoon uses a tree for each type, while Clang uses a graph in order to reuse type nodes (e.g., an integer variable and an array of integers have a reference to the same node that represents the integer type). So to access the type references used by types, in Java it was only required to retrieve the descendants of the type, while in C++, it was necessary to consider potential cycles. This does not apply to JavaScript, since the AST does not support types.

Then/Else join points An `If` statement can have two block of statements as children. One mandatory block is executed if the condition is true, the other optional block is executed if the condition is false. To be able to differentiate these two blocks of statements, if a block of statements has a node `If` as a parent, it is mapped to either a `then` or an `else` join point. These join points were created as they can be used to calculate metrics, such as Cognitive Complexity.

3.4 Multi Language Metrics

In literature, nearly all metrics, especially object-oriented metrics, are defined in a language-independent way (e.g., using pseudo-code). However, most tools focus on a single language and do not take advantage of this fact.

This work aims to propose a robust and scalable approach for multi-language analysis in the LARA framework. As a case study, we implemented a language metrics library over this approach in a language-agnostic way. This library can be easily extended with new or alternative definitions of metrics.

3.4.1 LARAM

LARAM is a library implemented in LARA that uses common join points to calculate metrics in multiple languages. Every language uses its own AST with its own nodes, however, by using the virtual AST and join points described in the previous section, it is possible to have a single implementation of metric to work on multiple languages. The library, which uses a patched version of Query API for common join points, instead of the default compiler join points, is able to select all join points of a given type. The metrics are calculated for the entire project, for each file, for each class, and for each function.

We created the abstract class `Metric` (see Fig. 3.3) to represent each metric. This class contains information about the metric, namely, the id, the full name, the author's name, the year it was proposed, and a short description of the metric. Since not all metrics can be calculated on all granularity levels, e.g., object-oriented metrics that are calculated only for classes, it is important to define, for each metric, the supported levels. So, for metrics that are only calculated at a class level, e.g., CBO, only the method `calculateForClass($class)` is implemented, while more general metrics such as Cyclomatic Complexity implement a method for each supported level. However, it is possible to have default implementations. For instance, the metric itself can be calculated in the method `calculateForJoinPoint($jp)`, and all the other methods call this method.

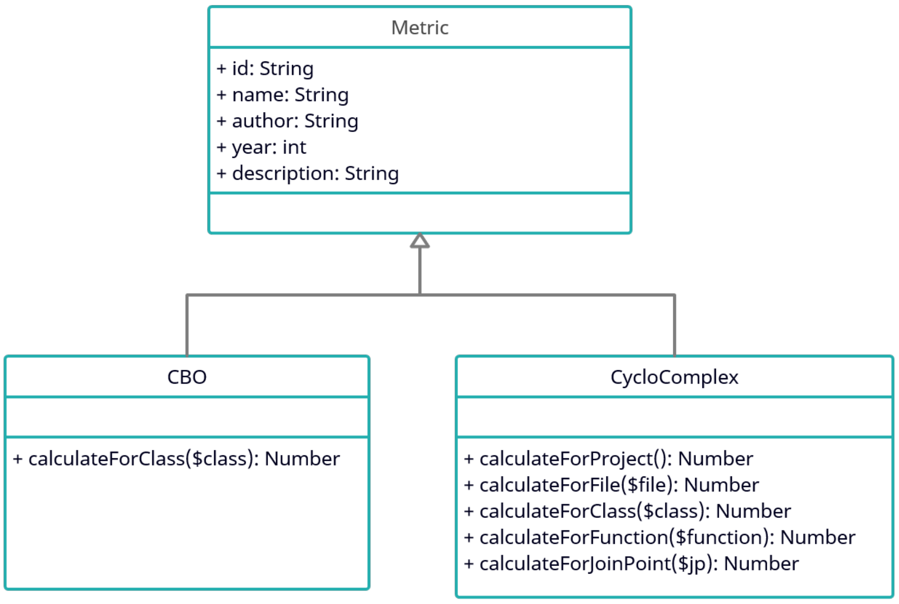
Metric
+ id: String
+ name: String
+ author: String
+ year: int
+ description: String
CBO
+ calculateForClass($class): Number
CycloComplex
+ calculateForProject(): Number
+ calculateForFile($file): Number
+ calculateForClass($class): Number
+ calculateForFunction($function): Number
+ calculateForJoinPoint($jp): Number

Figure 3.3: Metric Class Diagram

Each metric can be called individually on a specific join point (e.g., `class`). It is also possible to calculate all the metrics supported for all the granularity levels. When calculating all metrics, the results are written to four `csv` files, one for each granularity. Each contains the id of the analyzed join point (file, class, function), the id of the metric, the value of the metric, and the time of execution. There is also textual output in the console, which shows the results and times for each metric in a more friendly way. When calculating metrics, the execution times can have small variations, so the metrics can be calculated multiple times, and the final execution time is an average of the multiple runs.

3.4.2 Implemented Metrics

We selected metrics with a high number of references in the literature, and their definition can be found in Section 2.1. To assess the quality of our approach, we selected to implement metrics that are relevant and well-accepted. The metrics can be divided into three sets: object-oriented, complexity metrics, and size metrics[4].

3.4.2.1 Object-Oriented Metrics

With the rise in popularity of object-oriented languages, such as C++ and Java, various metrics were developed to analyze projects implemented in these languages. Chidamber and Kemerer's (CK) metrics were among the first set metrics to analyze these systems and are able to measure the cohesion and coupling of objects. Li and Henry's metrics continue the work Chidamber and Kemerer and proposed a set of metrics to analyze the maintainability of these systems.

Chidamber and Kemerer Metrics are a set of 6 different measures designed to evaluate several aspects of OO systems. A total of six metrics were proposed, Subsection 2.1.4.1 provides a definition of these metrics. We implemented these metrics in LARA as close as possible to the original definitions. Table 3.2 shows the join points and attributes used for each metric.

Metric	class	method	memberCall	constructor	constructorCall	fieldRef	param	type	field	varDecl
NOC	id superClasses									
DIT	superClasses									
RFC	allMethods	id isCustom instanceOf	method superCalls	id	constructor					
WMC[a]	allMethods									
LCOM	id methods	id				class field			id	
CBO	id fields methods allMethods isCustom allSuperClasses	returnType descendants	class instanceOf			class instanceOf	type	decl isClass usedTypes	type descendants	type instanceOf

[a] WMC uses the metric Cyclomatic Complexity to calculate the complexity of each method, therefore it also uses the join points and attributes defined in that metric.

Table 3.2: Join points and attributes used in each CK metric

In the definition of Weighted Methods per Class (WMC) it is not specified how to calculate the complexity of a method, so it was chosen to calculate the cyclomatic complexity. Since methods with high cyclomatic complexity are harder to test, they are more likely to have bugs.

Depth of Inheritance Tree (DIT) and Number of Children (NOC) are metrics related to the class hierarchy. In our implementation, we chose to only take into consideration explicit classes. Other similar structures that also allow inheritance, such as Interfaces in Java or Structs in C++, were not considered (although a modified version of this metric could consider them).

LCOM shows the lack of cohesion among methods of a class. This metric has many definitions and revisions and depending on the tool used, the results can be very different. The definition followed in LARAM is also referred to as LCOM94. The implementation can be seen in Listing 3.4. First, it retrieves, for each method, all the references to the fields of the analyzed class. Then,

[4]The full implementation of all metrics is available at: `https://github.com/GilTeixeira/feup-diss/tree/master/lara/Metrics`

it determines for each pair of methods if there is an intersection of the referenced fields. Finally, if the number of pairs that do not access a common variable is greater or equal than the number of pairs that do access a common variable, the difference between these values is returned. If the difference is a negative value, it returns 0.

```
LCOM94.prototype.calculateForClass = function ($class) {

    var methodFieldRefsMap = new Map();

    for ($method of $class.methods) {
        var $fieldRefs = Query.searchFrom($method, "fieldRef").get();
        $fieldRefs = $fieldRefs.filter($fieldRef => $fieldRef.field !== null &&
                     $fieldRef.class.id === $class.id);
        var fieldRefsIds = $fieldRefs.map($fieldRef => $fieldRef.field.id);
        methodFieldRefsMap.set($method.id, new Set(fieldRefsIds));

    }

    var numPairMethodCommomFieldAccess = 0; // Q
    var numPairMethodNoCommomFieldAccess = 0; // P

    var methods = $class.methods;

    for (i = 0; i < methods.length; i++)
        for (j = i + 1; j < methods.length; j++) {
            var method1Id = methods[i].id;
            var method2Id = methods[j].id;

            var setMethod1 = methodFieldRefsMap.get(method1Id);
            var setMethod2 = methodFieldRefsMap.get(method2Id);

            if (!hasIntersectionSets(setMethod1, setMethod2))
                numPairMethodNoCommomFieldAccess++;
            else numPairMethodCommomFieldAccess++;

        }

    var lcom = Math.max((numPairMethodNoCommomFieldAccess -
    numPairMethodCommomFieldAccess), 0);

    return lcom;
}
```

Listing 3.4: LCOM

The Response For a Class (RFC) measures the number class methods plus all the methods called by the class. This metric is straightforward to calculate since there is an attribute to get all methods of a class, and retrieving the calls to others methods can be easily done using the Query API to get all `memberCalls`.

Coupling Between Object (CBO) is a count of the number of classes coupled to a given class. The types of coupling counted were: call to methods of other classes, all the classes extended, the types of fields, parameters and variable in methods, and the return types of each method.

Li and Henry Metrics (see Subsection 2.1.4.2) describe a set of 5 metrics to evaluate the maintainability of OO systems. These metrics are very objective, so their implementation was more straightforward when compared with CK metrics. Table 3.3 shows the join points and attributes used for each metric.

Number Of Methods (NOM) measures the number of methods and Size of procedures or functions (SIZE2) measures the number of fields and methods, so their implementation only requires to use the size of the arrays returned by attributes `fields` and `allMethods`.

Size of procedures or functions (SIZEl) counts the number of semicolons of each class. To calculate this metric, we obtained the source code of each class, and after removing all of the comments, we simply counted the number of remaining semicolons.

Message Passing Coupling (MPC) counts the number of send statements, i.e., the number of function calls. So we searched the descendants of each function for calls to other methods and returned the number of distinct calls.

Data Abstraction Coupling (DAC) counts the number of attributes that use another class as their type. For each field, it was taken into consideration not only the base type but also all inner types, e.g., a field of the type `Pair<String,Integer>` uses the types `Pair`, `String` and `Integer`.

Metric	class	method	memberCall	type	field
NOM	allMethods				
SIZE1	code allMethods	code ancestor			
SIZE2	fields allMethods				
MPC	id isCustom allMethods		class		
DAC	id fields isCustom			decl isClass usedTypes	type

Table 3.3: Join points and attributes used in each LH metric

3.4.2.2 Complexity Metrics

The complexity metrics that we implemented were the cyclomatic and cognitive complexity. The former measures how difficult it is to test a code unit, since it counts the number of linearly independent paths through source code. The latter measures how difficult it is to read and understand the code. This is done by counting the number of structures that break the linear flow of the code, and also increment this value when they are nested. Both of these metrics can be calculated for the entire project, a file, a class, or a function.

The join points that increment the cyclomatic complexity are:

- `if` statements;
- `loop` statements, including `for`, `foreach`, `while` and `do ... while`;
- ternary operators;
- `case` inside `switch` statements;
- logical `AND (&&)` and `OR (||)` operators;
- `goto` statements;
- functions;
- `lambda` expressions;

For the cognitive complexity, there are three types of increments: general increments that simply increase the complexity by 1; nesting level increments that, as the name implies, increments the nesting level but does not increase the overall complexity; and the nesting increments which increase the overall complexity by the current nesting level. There is a general increment for the following join points:

- `if` statements;
- `else` statements, `else if` are only counted as 1 increment;
- ternary operators;
- `switch` statements;
- `loop` statements, including `for`, `foreach`, `while` and `do ... while`;
- `catch` statements;
- `goto` statements;
- sequences of the same logical `binary` operator;

These join points increase the nesting level:

- `if` statements;
- ternary operators;
- `switch` statements;
- `loop` statements, including `for`, `foreach`, `while` and `do ... while`;
- `catch` statements;
- `lambda` statements;

Finally, these join points increase the overall complexity by the current nesting level:

- `if` statements;
- ternary operators;
- `switch` statements;
- `loop` statements, including `for`, `foreach`, `while` and `do ... while`;
- `catch` statements;

```
function CancelToken(executor) {
  if (typeof executor !== 'function') {
    throw new TypeError('executor must be a function.');
  }

  var resolvePromise;
  this.promise = new Promise(function promiseExecutor(resolve) {
    resolvePromise = resolve;
  });

}
```

Listing 3.5: Example of JavaScript expression function as a class

Compared to the original SonarQube implementation [4], our implementation follows almost the same rules, with one exception. In LARAM, nested functions do not increment the overall complexity. This decision was made because before ECMAScript 6 added classes to JavaScript, classes were created using functional expressions inside a function constructor, and this would cause methods of a class to increment the overall complexity. SonarQube also has a form of mitigation where outer function are ignored if they only contain declarations at the top level, however, this does not account for all possibilities, e.g., Listing 3.5 shows an example of a case where a class implemented in JavaScript is not considered a class by SonarQube because it has an `if` statement.

3.4.2.3 Size Metrics

The last set of metrics focuses on getting information about the high-level structure of the source code. As these are the most objective and straightforward metrics, they are also the easiest to implement and understand. The implemented metrics count the number of classes, number of files, number of functions, and number of lines. Three of the metrics consist of counting the number of a given join point, so we used the `Query` API to get an array of those join points and simply returned the size of that array. For instance, Listing 3.6 shows the implementation of the number of functions for the entire project or for a specific file or class. The number of lines is calculated by doing the difference between the last and first line of a join point plus one.

```
NOFu.prototype.calculateForProject = function() {
  return Query.search("function").get().length;
}

```

```
NOFu.prototype.calculateForFile = function($file) {
  return this.calculateForJoinPoint($file);
}

NOFu.prototype.calculateForClass = function($class) {
  return this.calculateForJoinPoint($class);
}

NOFu.prototype.calculateForJoinPoint = function($jp) {
  return Query.searchFrom($jp, "function").get().length;

}
```

Listing 3.6: Implementation of the metric Number of Functions

3.5 Summary

This chapter describes the approach chosen to calculate metrics for multiple languages. We defined an interface that can be used to add new languages, a model of common join points that is mapped to the nodes of each AST. Each join point has a set of attributes that can be used to obtain relevant information. Since we represent different languages using the same join point model, it becomes possible to consistently apply the same LARA scripts to multiple languages. We also defined a library of object-oriented, complexity, and size metrics that can be applied to the whole project, or to each file, class or function. Finally, for each granularity, we generate a `csv` file where each row contains the id of the measured join point, the id of the metric, results of the metric, and the time it took to calculate.

Chapter 4

Empirical Evaluation

This chapter's purpose is twofold, to evaluate the results obtained by the proposed approach and compare it with other tools available, as well as to discuss the potential benefits and drawbacks of this approach when compared with the state of the art.

4.1 Methodology

To evaluate the proposed framework, we performed two types of validation, internal and external. The internal validation focuses on comparing the results of every tool's metrics in a small project developed in different languages. This will allow to evaluate the consistency of results. The external validation compares the results and performance of several tools across multiple open source projects, in order to determine which approach leads to better times and show how different tools use different implementations of the same metric.

4.1.1 Experimental Setup

In order to compare the several tools, we selected with LARAM, a group of open source and proprietary tools to obtain metrics of multiple projects. While most tools were capable of multi-language analysis, we also selected one tool (CKJM) that only supports Java since it uses a highly optimized approach by iterating the AST and focusing specifically on the CK metrics. The tools used for analysis are:

Analizo Analizo[1] is a suite of source code analysis tools aimed at being language-independent and extensible with support to C, C++, Java, and C#. The version tested was 1.25.0.

SonarQube Community Edition SonarQube [2] is an automatic code review tool to detect bugs, vulnerabilities and code smells in source code. For C++ analysis we used the SonarQube C++ plugin (Community)[3]. Due to compatibility reasons, we used the SonarQube Version

[1]Analizo is available at https://github.com/analizo/analizo

[2]SonarQube is available at https://www.sonarqube.org/

[3]SonarQube C++ plugin (Community) is available at https://github.com/SonarOpenCommunity/sonar-cxx

8.5.1 (build 38104) to analyze Java and JavaScript projects and SonarQube Version 7.9.5 (build 38598) with the C++ Community plugin v1.3.2 for C++ projects. The C++ plugin only supports the Long Term Support version of SonarQube, and version 7.9.5 is the one currently available.

Understand Understand [4] is an IDE for maintaining, measuring and visualizing codebases, with support for 19 different languages. The version tested was Understand 5.1 (Build 1012).

CKJM CKJM [5] is a tool to calculate the Chidamber and Kemerer object-oriented metrics by processing the bytecode of compiled Java files. The version tested was 1.0.

Every tool has a different interface and a different way to output results. We used several strategies in order to automate the comparison of results, such as changing the output of the tools to conform with the output format of our tool (LARAM), or transform the results of the tools into our format. Moreover, since the validation involves multiple tools and multiple projects to analyze, it was also necessary to automate the process of data collection and merging.

- **LARAM** can be used as a single command that accepts a directory with source files. It displays the results in the terminal and creates a file for each type of metric, in which each line shows the class/file/function analyzed, the name of the metric, the value of metric and the time it took to calculate.

- **Analizo** uses a single command that accepts a directory with the source files. The results are displayed in the terminal. However, since the tool is open source, we created a fork [6] in which the generated results are similar to LARAM. It was also added the functionality of measuring the time to calculate metrics.

- **CKJM** only supports Java and does not use the source files to calculate the metrics. Instead, it uses the compiled `.class` files, therefore it was necessary to use two commands, one to find the paths of the `.class` files and another to call the tool. CKJM is also open source, and we made a fork [7] in order to output results in a format similar to LARAM. However, since this tool uses a different approach in which it visits every class, calculating all metrics simultaneously, it was not possible to calculate the time for each individual metric. Instead, we measured the time to calculate all metrics for a single class.

- **SonarQube** provides a web-based interface to analyze projects. We manually created a SonarQube project for every project to be analyzed and used the provided Web API to obtain the metrics of a project with a given ID. Therefore, it was possible to automatize the retrieval of metrics by using this API. The data is later parsed and written into a file with a format similar to the one used by LARAM. It was not possible to measure the time it took

[4]Understand is available at `https://www.scitools.com/`
[5]CKJM is available at `https://github.com/dspinellis/ckjm`
[6]Analizo's fork is available at `https://github.com/GilTeixeira/analizo`
[7]CKJM's fork is available at `https://github.com/GilTeixeira/ckjm`

the measure each metric, but it is possible to measure the time of the complete SonarQube analysis.

- **Understand** is an IDE that provides a GUI for the user to obtain metrics of a given directory, which are saved to a `csv` file. We created a parser for this file that allows us to convert it to the LARAM format. It was not possible to automatize the metrics retrieval using this tool since it does not provide a command-line interface, so it was necessary to manually generate the metrics file using the provided GUI for every tested project.

Finally, in order to automate the complete flow we, created two Bash scripts. The first allows to replicate the obtained results. It clones all the projects tested, builds the Java projects (required to run CKJM), and sets up the analysis tools LARAM, CKJM and Analizo. Understand is a commercial tool, so it was not possible to automatically set up. SonarQube was also not possible to automatically set up since it requires different versions depending on the language of the project analysed and the plugin must be manually added. The second plugin analyses a single project and takes as parameters the path of the project analyzed, the project's language, and the id of the SonarQube project. This script creates a folder with the name of the project analyzed and the current timestamp. Inside that folder creates, for each tool, a new folder with the obtained results. After obtaining and parsing each tool's results, the script merges the results from every tool in four files, one for each type of metric and an additional file with the time that every tool took.

All the benchmarks were performed using a desktop computer using Pop!_OS 20.04 LTS with an Intel® Core™ i7-6700K CPU @ 4.00GHz.

4.1.2 Projects tested

All the open-source projects are available on GitHub and were chosen with the following criteria: it must be a relevant project, so it was only selected projects with more than 10k stars, it must be tagged with one of the languages supported for analysis with LARAM (C/C++, Java, JavaScript). Due to limitation in time and difficulties with the languages, for C/C++ there was an additional requirement of a low number of external dependencies. Since Kadabra can do analysis with an incomplete classpath, for Java, a high number of dependencies was not a problem.

The list of projects tested was:

Store A small example project of a retail store with 6 classes [8]. It was developed to help verify the values of the metrics and to compare the consistency of results between two languages, C++ and Java.

Axios A promise-based HTTP client for the browser and node.js implemented in JavaScript [9].

Libphonenumber Google's common Java, C++ and JavaScript library for parsing, formatting, and validating international phone numbers[10].

[8]The project Store is available at https://github.com/GilTeixeira/feup-diss-val/tree/main/test

[9]Axios is available at https://github.com/axios/axios

[10]libphonenumber is available at https://github.com/google/libphonenumber

Nlohmann-json A header-only library to make JSON a first-class datatype for C++11 [11].

Elasticsearch A Distributed RESTful Search Engine written in Java[12].

4.2 Internal Validation Results

The internal validation focuses on evaluating the consistency of results, that is, determining if the result of the same metric is equal for different languages. For that purpose was created a small project (Store) with six classes in two languages, C++ and Java. The implementation was as close as possible between the two languages. Then, an analysis was made using the tools that support both languages to compare the results of the same metric. Tables 4.1, 4.2, 4.3, show the metrics results of the project Store per class for tools LARAM, Understand, Analizo, respectively, and Table 4.4 shows the metrics results for the whole project calculated by SonarQube.

The results calculated by LARAM are very consistent in almost every metric. The only exceptions are the metrics size of procedures or functions (SIZE1) and Number of Lines (NOL). These discrepancies of results between languages are to be expected as these metrics measure the number of semicolons and number of lines. Analizo's results are also very consistent in both languages. The only exception is the calculation of RFC for the class `Client`, this seems to be a bug since it is counting a method of the class as both a method and a call. SonarQube, despite the fact that it has a different implementation for every language, obtains very consistent results with only the number of files and number of lines being different between the two implementations. However, Understand has very different results in almost every metric. Even though Understand is a closed-source project, it is still possible to deduce why this happens. The main reasons are due to the particularities of each language. In Java, every class has `Object` as a superclass, therefore the depth of inheritance tree will always be at least 1. In C++, for every class is created a Constructor, a Destructor, and a Copy Constructor, and since most metrics use the value of the number of methods to be calculated, the results of the metrics will not be consistent for all languages. This indicates that Understand was not developed to take into account the particularities of each language in order to have a consistent results between languages.

4.3 External Validation Results

The external validation was made on real-life open-source projects. For each tool we evaluated not only the results and execution time of each metric, but also compared the different metric implementations.

[11]Nlohmann-json is available at `https://github.com/nlohmann/json`
[12]Elasticsearch is available at `https://github.com/elastic/elasticsearch`

	Client		Date		Person		Product		Store		Transaction	
	C++	Java	C++	Java	C++	Java	C++	Java	C++	Java	C++	Java
DIT	1	1	0	0	0	0	0	0	0	0	0	0
RFC	7	7	11	11	5	5	4	4	22	22	8	8
WMC	0	0	7	7	0	0	0	0	28	28	2	2
LCOM	0	0	0	0	0	0	1	1	0	0	0	0
CBO	2	2	0	0	0	0	0	0	5	5	1	1
NOM	5	5	11	11	5	5	4	4	10	10	6	6
SIZE1	19	13	33	19	14	8	12	7	95	123	28	19
SIZE2	7	7	14	14	7	7	6	6	15	15	9	9
MPC	1	1	0	0	0	0	0	0	5	5	1	1
DAC	1	1	0	0	0	0	0	0	3	3	1	1
CyC	5	5	18	18	5	5	4	4	38	38	8	8
CoC	0	0	9	9	0	0	0	0	56	56	3	3
NOL	33	27	72	64	29	24	26	22	227	248	46	43

Table 4.1: Comparison of LARAM results for the project *Store*

	Client		Date		Person		Product		Store		Transaction	
	C++	Java	C++	Java	C++	Java	C++	Java	C++	Java	C++	Java
DIT	1	2	0	1	0	1	0	1	0	1	0	1
RFC	15	10	15	11	7	5	7	4	13	10	6	6
WMC	5	5	16	16	5	5	4	4	14	35	6	8
LCOM	68	30	66	45	71	40	71	37	84	64	58	44
CBO	1	1	0	0	0	0	0	0	3	4	1	1
NOM	8	5	15	11	7	5	7	4	13	10	6	6
CyC	1	1	1	1	1	1	1	1	1	3	1	1
NOL	33	27	72	64	29	24	26	22	227	248	46	43

Table 4.2: Comparison of Understand results for the project *Store*

	Client		Date		Person		Product		Store		Transaction	
	C++	Java	C++	Java	C++	Java	C++	Java	C++	Java	C++	Java
DIT	1	1	0	0	0	0	0	0	0	0	0	0
RFC	14	13	29	29	11	11	9	9	32	32	18	18
LCOM	1	2	2	2	1	1	1	1	1	1	1	1
CBO	2	2	0	0	0	0	0	0	1	1	1	1
NOM	5	5	11	11	5	5	4	4	10	10	6	6
CyC	2	2	2.45	2.45	2	2	2	2	4.5	4.5	2.33	2.33

Table 4.3: Comparison of Analizo results for the project *Store*

4.3.1 Metrics Results Comparison

The analyzed tools have different implementations of the same metrics. The metrics can have either revisions that change how they are calculated or can give some leeway in the way they are defined, e.g., the metric Weighted Methods per Class (WMC) does not define how the complexity

	Project	
	C++	Java
CyC	78	78
CoC	68	68
NOCl	6	6
NOFu	41	41
NOFi	12	6
OO-NOL	626	452

Table 4.4: Comparison of SonarQube results for the project *Store*

of each method should be calculated, to allow for the most general application of this metric. Therefore it is expected that the results will not be exactly the same across different tools. So, in order to determine the degree to which different implementations of the same metric are associated, we calculated the correlation between every pair of results of the same metric in different tools. The method of correlation chosen was Kendall. Even though other methods, such as Pearson, use more information in their calculations, Kendall is non-parametric, which does not require that the variables follow a normal distribution. It is also preferred to the Spearman method for being more robust when using with small samples and when handling outliers [11]. The Kendall correlation has a range of values between -1 and 1. The highest the value, the stronger is the relationship between the variables. This means that for the Kendall correlation, the value -1 means that the highest values of one variable are associated with the lowest values of the other variable. Values closer to 0 indicates that there is no relationship between the variables. After calculating the correlation between each pair of tools, we computed a heat map with the results. We compared the results obtain for metrics applied to classes and files for the Java project Libphonenumber since all considered tools support Java and these metric granularities.

Figure 4.1 shows the correlations for the coupling between classes. While all results show a high correlation level (all values are higher than 0.75), the pairs of implementations with the higher correlation are CKJM and Understand with 0.89, followed by CKJM and LARAM with 0.88. These results are to be expected as every implementation counts the number of class references in a given class. Analizo has a lower correlation with other tools since it only counts method calls and instance variables of another class, while the other tools count field accesses, method calls, inherited classes, variable types, and return types.

Figure 4.2 shows the correlations for the lack of cohesion of methods. The results show a high discrepancy of correlation between implementations, with the only two implementations showing a high level of correlation being LARAM and CKJM with 0.78. LCOM is a metric with many revisions [7, 6, 17, 16, 22, 13], so it is expected that the tools do not implement the same variation, which leads to different results. LARAM and CKJM both implement the original LCOM definition [6], so they the most similar. Analizo implements the definition purpose by M. Hitz and B. Montazeri [16]. Understand has the most dissimilar results since the metric produces normalized results between 0 and 100.

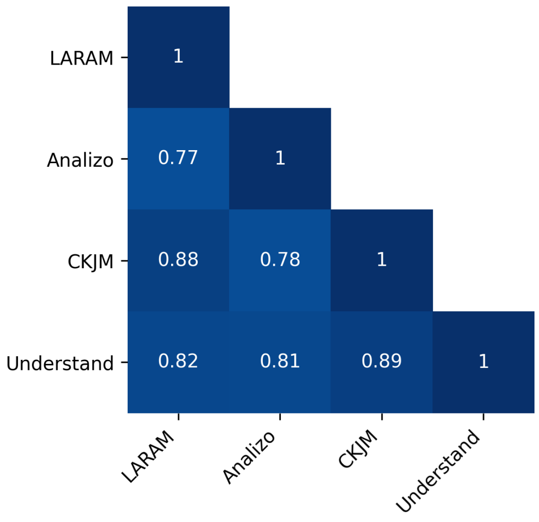

Figure 4.1: Heat Map with the correlation for CBO

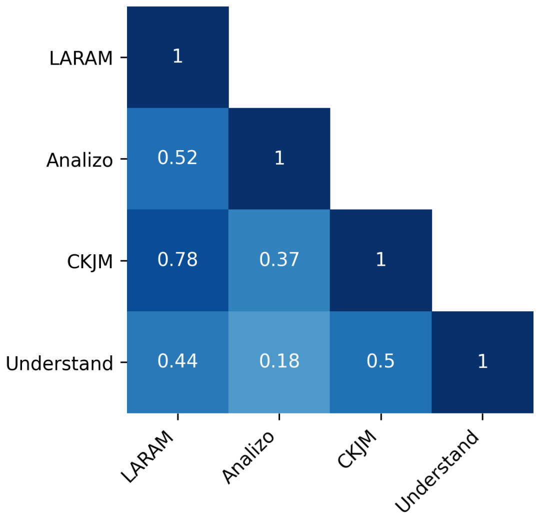

Figure 4.2: Heat Map with the correlation for LCOM

Figure 4.3 shows the correlations for the lack of Number of Children. The results show all metrics having a perfect correlation. This has two causes: most classes are not extended, and therefore the metric has the value 0, and this is a fairly simple metric, consequently most implementations are similar.

Figure 4.4 shows the correlations for the response for class. The tools Analizo, LARAM, and CKJM show a high correlation, with the Understand results being slightly less correlated with other tools. Understand has a different description of this metric and only measures the number of methods, including inherited ones, while the other tools also count the number of unique functions called.

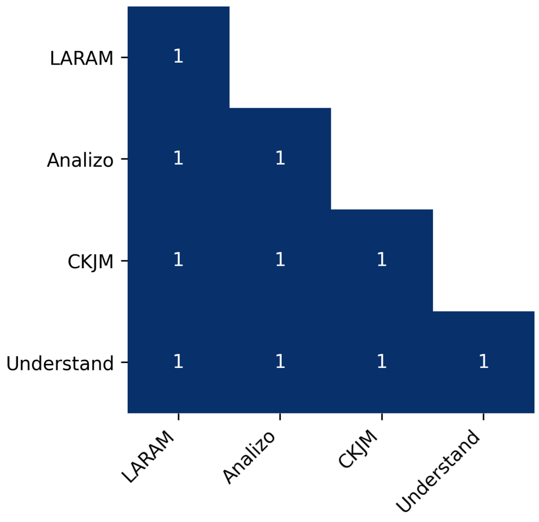

Figure 4.3: Heat Map with the correlation for NOC

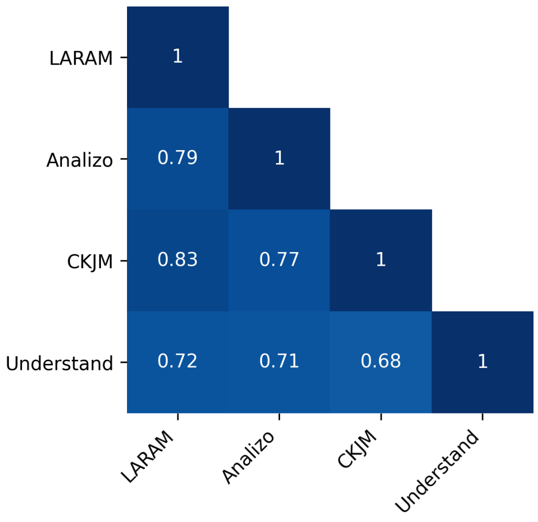

Figure 4.4: Heat Map with the correlation for RFC

Figure 4.5 shows the correlations for the weighted methods per class. Understand and CKJM are the pair of tools with the higher correlation. However, each tool has a different interpretation of this metric. For example, CKJM simply assigns a complexity value of 1 to each method, Understand sums the number of methods and the cyclomatic complexity of each one, while LARAM only sums the complexity of each method. Therefore, Understand implementations is closer to CKJM and LARAM than these tools are of each other.

Figure 4.6 shows the correlations for the number of methods. All tools have an almost perfect correlation. LARAM has a slightly lower correlation than the other tools because, similar to the Spoon model used by LARA, it counts the default constructor as a method when the class does not explicitly define a constructor.

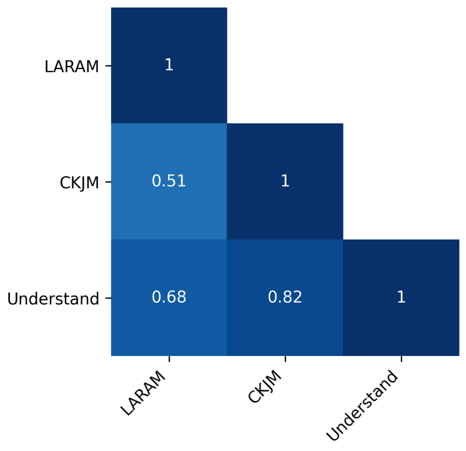

Figure 4.5: Heat Map with the correlation for WMC

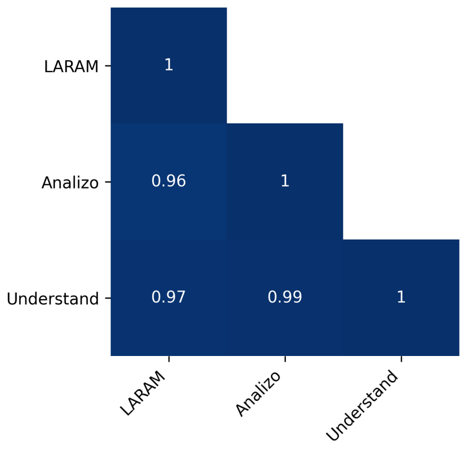

Figure 4.6: Heat Map with the correlation for NOM

Figure 4.7 shows the correlations for the depth of inheritance tree. LARAM, Understand, and Analizo have a high correlation, while CKJM negatively correlates with the other tools. The reason for these results is because CKJM uses the `.class` files to calculate metrics, and when a class `A` extends a class `B` imported from a library, CKJM attributes the class `A` the value 0 for the DIT, instead of the default 1, for the other classes. Therefore while other tools have higher values for classes that extend imported classes, CKJM has 0 as a result. This can be seen as a limitation of this tool because without analyzing the `jar` files, it cannot return the default value of 1 for this metric.

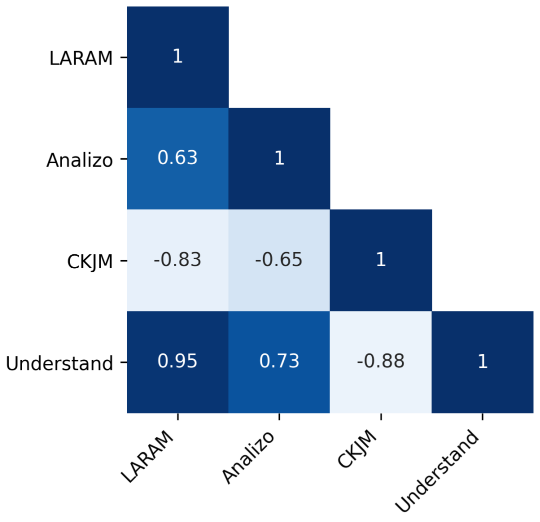

Figure 4.7: Heat Map with the correlation for DIT

Figures 4.8, 4.9, 4.10 shows the correlations for the number of functions, number of lines and number of classes. As expected, the first two matrices have a perfect correlation. The number of classes is slightly different for every tool since LARAM only counts classes, SonarQube counts classes and interfaces, and Understand counts classes, interfaces, and enums. This is not a limitation of any of the tools but a different design choice when implementing the metric.

Figure 4.11 shows the correlations for the cyclomatic complexity. Therefore, SonarQube and LARAM have a perfect correlation for these two metrics, while Analizo is not as closely related. Analizo calculates the average complexity per method while the other two calculate the sum of complexities. For the cognitive complexity, the correlation between LARAM and SonarQube is also equal to 1, and no other tool calculates this metric.

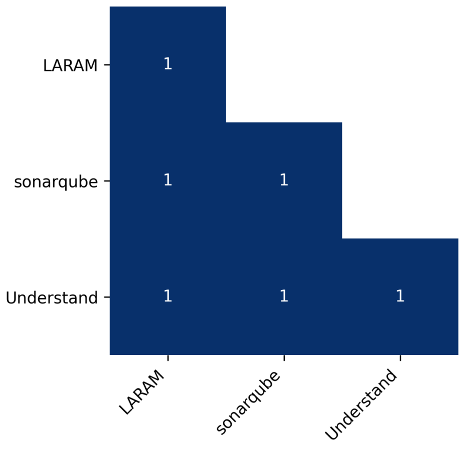

Figure 4.8: Heat Map with the correlation for Number of Functions

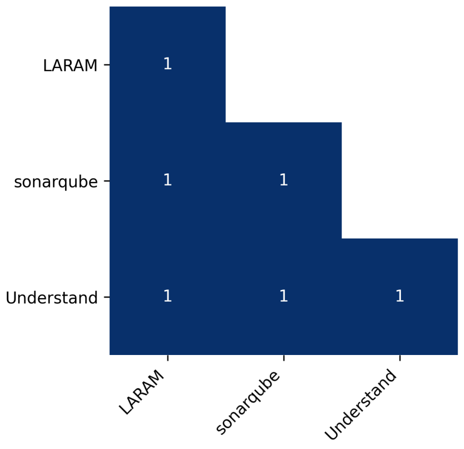

Figure 4.9: Heat Map with the correlation for Number of Lines

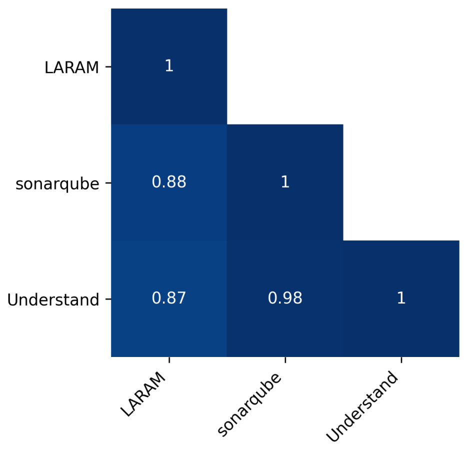

Figure 4.10: Heat Map with the correlation for Number of Classes

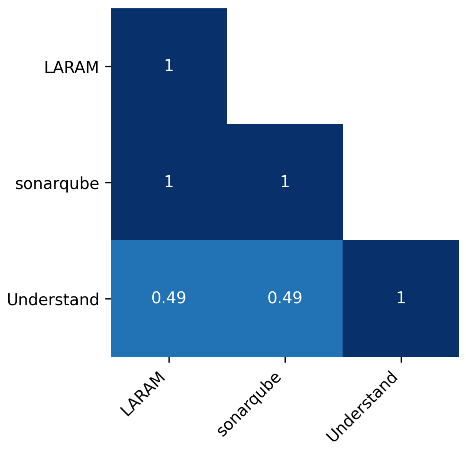

Figure 4.11: Heat Map with the correlation for Cyclomatic Complexity

4.3.2 Metrics Calculation Times Comparison

In this subsection, we compare the execution time of each tool. Since not all tools calculate the same metrics, we decided to compare the time it takes to calculate each metric. We also take a closer analysis on the execution time of several steps of LARAM.

Tables 4.5 and 4.6 shows the time in milliseconds it takes to calculates the class metrics that LARAM supports. This analysis was made using Java projects because it allows to compare with both Analizo and CKJM. As it is not possible to calculate the time it takes to calculate every metric in CKJM, the comparison is made using the sum of all the CK metrics, which are the only metrics supported by this tool. Considering only the CK metrics, Analizo manages to have a better execution time than CKJM, even though no metric takes more than one second. Analizo does not calculate metrics at the level of the AST but instead uses precomputed computation extracted from Doxygen files, this allows a faster execution time and a focus on metrics based on the relationship of entities. CKJM also has excellent results since it analyzes only one language and focuses explicitly on only calculating the CK metrics. LARAM is outperformed by other tools in every metric, however is important to emphasize that it can calculate a wide range of metrics, at different granularity levels. The fact that each metric at a higher granularity has to calculate the metric from the ground up causes the metrics at the higher level to take the most time. Therefore there is room for improvement if each level uses already calculated values, i.e., by caching the complexities of each function the complexity of a class could simply be the sum of the complexities of each function instead of calculating it again. The metrics that take the least time are the ones that do not require queries to the AST, e.g., DIT, NOM, SIZE2, while the metric that takes the longest is Cognitive Complexity.

Figure 4.12 and Table 4.7 show the times of different phases of LARAM. `InitTime` is the startup time and includes the time to load JavaScript libraries and parsing the code to the AST. As expected, as the projects get bigger, this phase takes more time. `LARAToJsTime` is the time

Metric	LARAM	Analizo	CKJM
CK-CBO	4 463.8	8.3	-
CK-DIT	37.0	1.1	-
CK-LCOM	4 815.8	226.4	-
CK-NOC	3 149.7	41.7	-
CK-RFC	4 038.0	1.7	-
CK-WMC	4 149.0	-	-
Total CK	20 653.3	279.2	619.1
LH-DAC	143.2	-	-
LH-MPC	3 554.6	-	-
LH-NOM	63.1	0.7	-
LH-SIZE1	1 563.0	-	-
LH-SIZE2	82.9	-	-
OO-CoC	11 137.9	-	-
OO-CyC	4 118.8	14.3	-
OO-NOL	144.8	-	-

Table 4.5: Time in milliseconds to calculate class metrics for all classes of Elasticsearch for each Tool

Metric	LARAM	Analizo	CKJM
CK-CBO	6 690.4	8.1	-
CK-DIT	20.0	0.3	-
CK-LCOM	7 071.2	154.3	-
CK-NOC	2 577.9	4.6	-
CK-RFC	4 679.4	0.8	-
CK-WMC	4 953.1	-	-
Total CK	23 449.9	168.0	263.2
LH-DAC	142.2	-	-
LH-MPC	4 716.2	-	-
LH-NOM	31.5	0.2	-
LH-SIZE1	3 332.5	-	-
LH-SIZE2	42.1	-	-
OO-CoC	13 982.1	-	-
OO-CyC	4 921.5	7.8	-
OO-NOL	76.7	-	-

Table 4.6: Time in miliseconds to calculate class metrics for all classes of Libphonenumber in Java for each Tool

	Axios	Elastic Search	Libphonenumber-java	Libphonenumber-js	Nlohmann-json
InitTime	1 196	4 601	3 212	10 367	8 004
LARAToJsTime	435	529	531	468	566
WeavingTime	4 786	114 563	120 203	78 446	49 357
TotalTime	7 590	120 866	125 062	90 659	59 314
Lines	1 889	19 935	20 476	37 642	25 558

Table 4.7: LARAM phases times in miliseconds

to convert LARA code, including LARA Common Language, to JavaScript and load it into the weaving engine. Per language, the times are relatively equal, with projects in Java and C+ taking more time than JavaScript because there are fewer join points and attributes implemented in the latter. `WeavingTime` is Lara's run time, in this case, the time to calculate all the metrics. The times also increase with the number of lines of code. Since in JavaScript is not possible to calculate all of the OO metrics, it takes less time compared with other languages. Finally, `totalTime` is the total execution time and will always be slightly greater than the sum of the individual times due to other tasks not being measured (e.g., debug messages, other overheads).

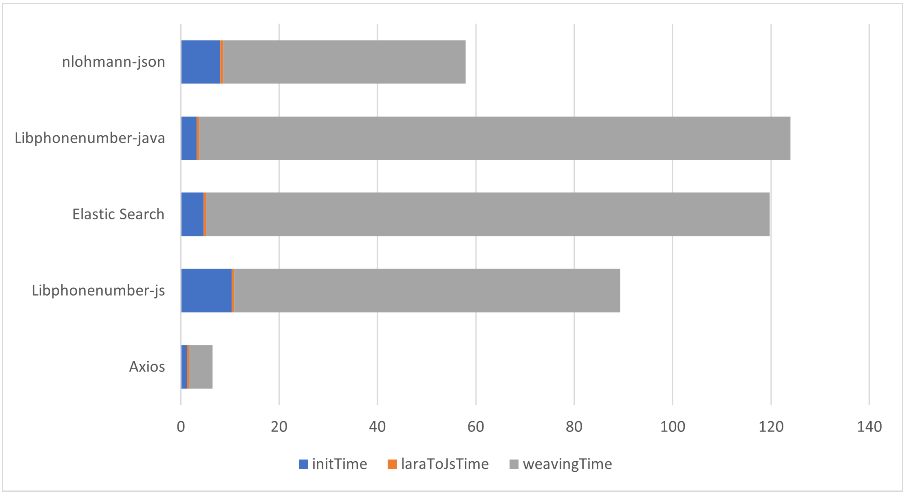

Figure 4.12: LARAM phases times in seconds as a bar chart

4.3.3 Implementations Comparison

This section compares the approaches of the analyzed tools, as well as the implementations of complexity and object-oriented metrics.

Even though each tool provides source code metrics, they use very different approaches. LARAM uses a virtual AST that maps the same set of join points and attributes to the ASTs of multiples languages. Then provides a common interface to interact with the virtual AST through JavaScript scripts, which allows to query nodes in a hierarchical way, and filter the nodes by type or attribute values. CKJM focus on calculating OO metrics for only one language, Java. During a single traversal of the AST, it can calculate multiple metrics at once, thus it is able to improve the execution time. Nevertheless, there is a trade-off with this strategy that is the more complicated implementation. In LARAM, it would also be possible to implement a similar approach, however it was chosen to define each metric separately since it is easier to add new metrics and understand the ones already defined.

Analizo's model identifies a list of existing modules (classes, files), elements define inside every module, (variables, functions) as well as the dependency information (inheritance relationships, function calls). Then uses Doxyparse, a source code parser based on Doxygen, to retrieve information about the source code. Finally, it has another module written in Perl to calculate the software metrics based on that model. SonarQube requires for each supported language a grammar, a parser, and a tree visitor. Then by using the visitor is possible to calculate different complexity metrics. To analyse Java and c/C++ projects the visitor is written in Java, while for Javascript the visitor is written in TypeScript. Understand is a closed source software, so it was not possible to verify how this tool calculates its metrics.

We have also analyzed how each tool implements the metrics. For that purpose, we analyzed the implementations of two metrics of different types, Depth of Inheritance Tree (OO metric) and Cyclomatic Complexity (Complexity Metric). LARAM, Analizo and CKJM are capable of calculating DIT and theirs implementation can be seen in Listings 4.1, 4.2 and 4.3. PatOIS (Subsection 2.3.5 at the page 19), which uses a high-level description language, is also presented in Listing 4.4.

LARAM retrieves the superclasses from join point `class`, then uses a recursive function to get the path of maximum length between the analyzed class and the root class in the class hierarchy. Analizo uses a similar approach, in which they get from their model, the parents of the class, then get the depth from each parent and return the value with the longest depth. CKJM, since it only targets Java, which can only have one superclass per class, simply obtains the array of superclasses and returns its size. PatIOS uses a primitive `inheritanceLevel`, whose value is the distance between two classes in the class hierarchy. Then, it calculates the inheritance level for all of the class's ancestors and returns the biggest value.

```
DIT.prototype.calculateForClass = function($class) {
  var depth = 0;
  for($classSuper of $class.superClasses){
    depthSuperClass = this.calculateForClass($classSuper) + 1;
    depth =  Math.max(depthSuperClass,depth);
  }
  return depth;
}
```

Listing 4.1: LARAM Implementation of DIT

```
sub calculate {
  my ($self, $module) = @_;
  my @parents = $self->model->inheritance($module);
  return 1 + $self->_depth_of_deepest_inheritance_tree(@parents) if (@parents);
  return 0;
}

```

```
sub _depth_of_deepest_inheritance_tree {
  my ($self, @parents) = @_;
  my @parent_dits = map { $self->calculate($_) } @parents;
  my @sorted = reverse(sort(@parent_dits));
  return $sorted[0];
}
```

Listing 4.2: Analizo Implementation of DIT

```
 public void visitJavaClass(JavaClass jc) {
 // (...)
  try {
      cm.setDit(jc.getSuperClasses().length);
  } catch( ClassNotFoundException ex) {
      System.err.println("Error obtaining all superclasses of " + jc);
  }
// (...)
   }
```

Listing 4.3: CKJM Implementation of DIT

```
DIT(c:class):max( forAll(d : ancestors (c);;
     SET += inheritanceLevel(c,d)))
```

Listing 4.4: PatOIS Implementation of DIT

While DIT analyzes the relationship between entities, cyclomatic complexity looks at the body of functions and classes. In fact, it measures the number of paths through the code. LARAM and SonarQube calculate the total complexity while Analizo calculates the average cyclomatic complexity per method. Each implementation is presented in the Listings 4.5, 4.6 and 4.7 for LARAM, SonarQube and Analizo, respectively.

LARAM implementation is done at the level of the class and sums the complexity of each of its methods, as well as increases the complexity value by one for each method of the class. To calculate the complexity of a method the metric searches the method's descendants and attributes the value of one for each conditional join point. Note that most of the code of the function `calculateForJoinPoint` could be replaced by a single line if we implemented an attribute `isConditional`. SonarQube has a visit method for each node of the AST, and if that node is a conditional node, it is added to the list of conditional nodes, then it returns the size of that list (`List<Tree> blame`). Analizo uses Doxygen to retrieve the number of flow conditions of each method for a class and returns the mean of those values.

```
CycloComplex.prototype.calculateForClass = function($class) {
  var complexityCounter = 0;
  for ($method of $class.allMethods){
    complexityCounter += this.calculateForFunction($method);
    complexityCounter++;
  }
  return complexityCounter;
}

CycloComplex.prototype.calculateForFunction = function($function) {
  var complexityCounter = 0;
  for(jp of $function.descendants){
    complexityCounter += this.calculateForJoinPoint(jp);
  }
  return complexityCounter;
}

CycloComplex.prototype.calculateForJoinPoint = function($jp) {
  var complexityCounter = 0;

  complexityCounter += $jp.instanceOf('if')?1:0;
  complexityCounter += $jp.instanceOf('loop')?1:0;
  complexityCounter += $jp.instanceOf('ternary')?1:0;
  complexityCounter += ($jp.instanceOf('case') && !$jp.instanceOf('default'))?1:0;
  complexityCounter += ($jp.instanceOf('binary')&& ($jp.kind === '&&' || $jp.kind
    === '||'))?1:0;
  complexityCounter += $jp.instanceOf('goto')?1:0;
  complexityCounter += ($jp.instanceOf('function') && $jp.hasBody)?1:0;
  complexityCounter += $jp.instanceOf('lambda')?1:0;

  return complexityCounter;

}
```

Listing 4.5: LARAM Implementation of Cyclomatic Complexity

```
public class ComplexityVisitor extends BaseTreeVisitor {

  private List<Tree> blame = new ArrayList<>();
  // (...)
  @Override
  public void visitMethod(MethodTree tree) {
    if (tree.block() != null) {
      blame.add(tree.simpleName().identifierToken());
    }
    super.visitMethod(tree);
  }
  // (...)
  @Override
  public void visitForEachStatement(ForEachStatement tree) {
    blame.add(tree.firstToken());
    super.visitForEachStatement(tree);
  }
   // (...)
}
```

Listing 4.6: Excerpt of SonarQube Implementation of Cyclomatic Complexity

```perl
sub calculate {
  my ($self, $module) = @_;

  my @functions = $self->model->functions($module);
  if (scalar(@functions) == 0) {
    return 0;
  }

  my $statisticalCalculator = Statistics::Descriptive::Full->new();
  for my $function (@functions) {
    $statisticalCalculator->add_data($self->model->{conditional_paths}->{$function}
     + 1);
  }

  return $statisticalCalculator->mean();
}
```

Listing 4.7: Analizo Implementation of Cyclomatic Complexity

Getting objective comparisons between the different implementations is not a straightforward task, since every tool uses a different language, the function that returns the value of the metric can call multiple functions, or even the fact that metrics are not calculating the exact same interpretation of the metric. So, because Analizo uses Perl and none of the tools available can analyze Perl, the only possible comparison with LARAM is the Lines of Code of the function that calculates the metric. That comparison can be seen in Table 4.8. LARAM has the same or less number of lines except in 2 metrics, LCOM and CyC. LARAM calculates LCOM as defined by Chidamber and Kemerer, while Analizo calculates a revision of LCOM. As for the Cyclomatic Complexity, Analizo only retrieves the number of control flow statements from Doxygen, and by doing that, it is not possible to select which statements to have in consideration. On the other hand, LARAM verifies every descendant of the functions and checks if they are conditional nodes.

	Analizo	LARAM
CBO	29	28
LCOM	17	34
RFC	16	16
DIT	13	8
NOC	13	11
NOM	6	3
OO-CyC	13	27

Table 4.8: Comparison of number of lines of each metric implementation between LARAM and Analizo

Since the SonarQube module for Java also uses Java for its implementation, it is possible to get not only the number of lines of each metric but also other complexity measures. Those measures can be seen in table 4.9, where it compares the implementations of cyclomatic and cognitive complexity. In terms of cyclomatic complexity and cognitive complexity, the two implementations have similar values, with SonarQube having a lower value in one implementation and LARAM having in another. However, since SonarQube requires a visitor for each node, the number of functions and consequently the number of lines will be of a higher magnitude compared to LARAM. It is important to note that SonarQube has a different implementation for every language while LARAM has a single implementation for multiple languages. Also, the metrics for the current LARAM implementation of cyclomatic complexity could be improved by implementing an additional attribute `isConditional`.

	Cyclo. Complex.		Cogni. Complex.		Number Functions		Lines of Code	
	Sonar	LARAM	Sonar	LARAM	Sonar	LARAM	Sonar	LARAM
CoC Impl.	48	39	23	17	27	6	207	62
CyC Impl.	18	22	6	16	12	6	76	50

Table 4.9: Comparison of several metrics applied to the implementation of metrics between LARAM and SonarQube

4.4 Validation Threats

In this section, we identify and describe some of the possible validity threats while also analyzing how they were mitigated.

Project Selection. To evaluate our approach, we chose a limited number of projects. The analyzed projects are open-source, with a high number of stars on GitHub. These criteria were chosen because it can indicate the high relevance of the project, but also allowed for a more open and reproducible comparison between tools. These projects also had a reasonable number of lines in order to demonstrate the worst-case scenario in terms of performance.

Project Dependencies. The projects chosen for C++ did not have external dependencies. For LARA is currently in development a Plugin to call Clava from CMake files, this plugin will allow to analyze more complex projects as long as it uses CMake. While for Java, it is possible to analyze projects with an incomplete classpath.

Execution Time. LARAM in terms of execution time is outperformed by other tools, however, this was not the main focus of this project. The main goal of LARAM was to create a tool that allows to easily calculate metrics using a common model for multiple languages. The model defined is easily extended and provides a simple interface to be interacted.

Relevance of Metrics. The metrics selected and implemented are some of the most referenced metrics in the literature, as seen in Chapter 2.

Comparison Tools. The tools selected were chosen because they allow to analyze multiple languages at the same time. Understand and CKJM are also some of the most referenced tools in the literature related to software metrics [26]. SonarQube is also a reference in terms of software analysis. Moreover, Analizo uses Doxygen, which is the standard tool for generating documentation.

4.5 Conclusions

After checking how different tools calculate these metrics, it is possible to see the benefits and drawbacks of each approach.

LARAM uses JavaScript to implement the metrics, and provides a simple interface to interact with the virtual AST, thus it is relatively easy to understand the implemented metrics and add new ones. The AST for every language uses the same join points, which allows to have a single implementation of a metric for multiple languages. The join points have the information from each AST node available through attributes. The join points and attributes can be iteratively added to the model, i.e., it is not necessary to define and implement all attributes immediately, this can be done according to the metrics' requirements. In terms of execution time, it falls behind compare to other approaches, however it provides a flexible and comprehensive approach to add new languages, by reusing existing grammars and parsers, and metrics.

Analizo uses the model provided by Doxygen to retrieve information about source code. This can be seen as a double-edged sword, as it does not need to create a parser for every language, but it also limits the information it can gather. Even though Doxygen gathers information on the existing modules (e.g., classes) and their elements (e.g., functions, calls), the information retrieved from the body of functions is limited. Analizo has a different class for each metric, which allows an easier understanding and addition of metrics. The language of which Analizo is implemented, Perl, is not one of the most used, especially when compared with JavaScript or Java, since 27.87% GitHub projects use JavaScript, 10.65% use Java and only 5.74% use Perl.

CKJM only analyzes Java Bytecode. The metrics are calculated using two visitors, a Class and a Method Visitor, and all the metrics are calculated with a single visit. By targeting a single language and only needing one traversing of the AST, the results are calculated very fast. However, it is not a scalable way to add new metrics, since each visitor's code can rapidly increase.

SonarQube requires a grammar, a parser, and a visitor for each supported language. Maybe due to being a commercial tool, the number of supported languages is superior to all the other tools considered in this work. However, each metric has a different implementation in each language.

Understand by being closed source software, only the developers can add new languages and metrics. However, it also provides many metrics for multiple languages.

Chapter 5

Conclusions and Future Work

This chapter overviews retrospectively the work developed in this dissertation. It answers the research questions defined at the beginning of the dissertation, explains the main contributions, and sets a few possible paths that could be used to enrich and extend the developed project.

5.1 Research Questions

Software metrics provide valuable information about the source code that could be used to improve software's maintainability. Even though these metrics are defined independently of the language, most tools do not take advantage of this fact and require a different implementation for each language. Our solution to this problem consists of developing a model that can map multiple types of AST nodes to common join points. This model can subsequently be used to calculate metrics in a language-independent way. To guide the validation of our approach, the following research questions were set:

> *RQ1: What impact will a multi-language approach have in terms of effort to add new languages?*

The level of effort to add new languages depends on how much information it is necessary to model from the source code (see Subsection 2.3.7). While modeling the source code at the entity level requires less effort to add new languages, it limits the metrics that are possible to calculate, e.g., Cognitive Complexity is not possible to calculate with only this information. Extracting information about the source code at an AST level requires a definition of a parser and grammar. Since it uses LARA, our approach allows the reuse of existing parsers, which simplifies the process of adding more languages. Consequently, it is also required to implement a version of the interface *AstMethods* to be able to perform operations in the AST. It is also necessary to map the AST nodes to the common join points and implement the attributes, however it is not necessary to implement all join points, but instead it can be done in an iterative way.

> *RQ2: What impact will a multi-language approach have in terms of effort to add new metrics?*

The conventional method consists of having a different implementation for every language. This allows to have code specific to a single language, which can lead to a highly optimized implementation (See Subsection 4.3.3. However, since it requires multiple implementations, if it is necessary to change a single implementation, in order to maintain the consistency it is also necessary to change all the others. Our approach uses a single model with common join points that allows to only need one implementation. This allows to avoid code duplication and ensures that the metric definition is consistent across languages at the cost of having to take into account the language's particularities.

RQ3: What impact will a multi-language approach have in terms of performance?

When analyzing a project, there are two main phases, parsing the source code to an AST and calculating the metrics. As expected, both of these times increase as the projects became bigger. The time it takes to calculate a metric depends on what is necessary to calculate. The metrics that take the least time are the ones that do not require to iterate the AST. The obtained results show that the porposed approach is outperformed in terms of execution time by other approaches (See Subsection 4.3.2). However, our metrics were developed as a proof of concept on the flexibility of our approach, for example, every metric is independent of one another, this penalizes performance, but it improves readability and simplifies the development of metrics.

RQ4: What impact will a multi-language approach have in terms of the consistency of metrics across languages?

Not all approaches take into account the consistency of results for their respective tools (See 4.2). Our approach, by having only one implementation of a metric for all languages, is able to mitigate the inconsistencies between languages. However, it is still necessary to have in consideration several language particularities to have truly consistent results.

5.2 Main Contributions

The end goal of this dissertation was to present an approach to calculate metrics in multiple languages. To accomplish this goal, we also contribute with the following engineering contributions:

Literature Review. The first step of this project was to review the already used approaches to model the source code and compare the level of information each approach retrieves. The review of the approaches and tools for language-independent code analysis compares the languages and metrics supported, the level of information retrieved, and the effort to add new languages.

LARA Common Language. For the LARA framework was created a new module that allows to represent nodes in multiple languages in a single join point. The model created mostly focuses on object-oriented and imperative concepts, however other models can be created based on the one developed to support concepts of other paradigms.

LARAM. A library written in LARA that calculates object-oriented, complexity, and size metrics using the model defined in LARA Common Language. It calculates metrics for the entire project but also for specific join points, file, class, and function. The library can also provide a straightforwardly way to add new metrics to the needs of the user.

Comparative Study. The validation of our tool consisted of a comparative study with other multi-language analysis tools. This study evaluated the consistency of the metric results, the execution time, and how each tool implements metrics.

5.3 Future Work

As with every project developed in a short period, this project can be further improved and extended. The languages and metrics supported were developed as a proof of concept, and the implementations of these metrics can be further improved and optimized. Also, support for more languages could be added. The following topics present some of the possible paths that can be taken into consideration:

Storing the AST. For every query starting in the root node, every node is converted into a join point. For the bigger projects, all of these transformations take a substantial amount of time. A possible optimization would be to store the entire AST converted in join points in memory. This could be done by starting with the AST root node, recursively ask for the children, turn them into join points and add them to the parent as children, resulting in an equivalent AST with join points. LARAM already measures the time that every metric takes to be calculated, therefore if this alternative approach were indeed implemented, it would be straightforward to check if it would improve the execution time.

Identification of Code Smells. The metrics calculated provide objective measurements on the source code, they do not indicate that the code is well or poorly written. The next step is to identify what metrics can be used to identify deeper problems and define thresholds that indicate a code smell.

Mutation Testing. The developed model was used to calculate metrics, but it can be used for other purposes. Since LARA is a framework source-to-source, it would be possible to generate the code again. One interesting action that could be developed is to add mutations to specific join points, e.g., replace the operator in expressions. This could be used to determine if the tests developed catch these mutations.

References

[1] E. H. Alikacem and H. A. Sahraoui. A metric extraction framework based on a high-level description language. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 159–167, 2009.

[2] João Bispo and João M.P. Cardoso. Clava: C/c++ source-to-source compilation using lara. *SoftwareX*, 12:100565, 2020.

[3] Z. Budimac, G. Rakic, M. Hericko, and C. Gerlec. Towards the better software metrics tool. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 491–494, 2012.

[4] G Ann Campbell. Cognitive complexity-a new way of measuring understandability. *SonarSource SA*, 2018.

[5] Laila Cheikhi, Rafa Al-Qutaish, Ali Idri, and Asma Sellami. Chidamber and kemerer object-oriented measures: Analysis of their design from the metrology perspective. *International Journal of Software Engineering and Its Applications*, 8:359–374, 03 2014.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[7] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '91, page 197–211, New York, NY, USA, 1991. Association for Computing Machinery.

[8] Joelma Choma, Eduardo Guerra, Tiago Da Silva, Luciana Zaina, and Filipe Correia. Towards an artifact to support agile teams in software analytics activities. pages 88–93, 07 2019.

[9] D. N. Christodoulakis, C. Tsalidis, C. J. M. van Gogh, and V. W. Stinesen. Towards an automated tool for software certification. In *[Proceedings 1989] IEEE International Workshop on Tools for Artificial Intelligence*, pages 670–676, 1989.

[10] Yania Crespo, Carlos Nozal, M. Manso, and Raúl Sánchez. Language independent metric support towards refactoring inference. *9th ECOOP Workshop on QAOOSE*, 01 2005.

[11] Christophe Croux and Catherine Dehon. Influence functions of the spearman and kendall correlation measures. *Statistical methods & applications*, 19(4):497–515, 2010.

[12] Stéphane Ducasse and S. Tichelaar. Dimensions of reengineering environment infrastructures. *Journal of Software Maintenance and Evolution: Research and Practice*, 15:345 – 373, 09 2003.

[13] Letha H. Etzkorn, C. Davis, and W. Li. A practical look at the lack of cohesion in methods metric. *J. Object Oriented Program.*, 11:27–34, 1998.

[14] Crt Gerlec, Gordana Rakić, Zoran Budimac, and Marjan Hericko. A programming language independent framework for metrics-based software evolution and analysis. *Comput. Sci. Inf. Syst.*, 9:1155–1186, 09 2012.

[15] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., USA, 1977.

[16] B. Henderson-Sellers, L. Constantine, and I. Graham. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Syst.*, 3:143–158, 1996.

[17] Martin Hitz and Behzad Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of International Symposium on Applied Corporate Computing*, pages 25–27, 1995.

[18] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., USA, 2nd edition, 2002.

[19] N. Kayarvizhy and S. Kanmani. An automated tool for computing object oriented metrics using xml. In Ajith Abraham, Jaime Lloret Mauri, John F. Buford, Junichi Suzuki, and Sabu M. Thampi, editors, *Advances in Computing and Communications*, pages 69–79, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[20] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, pages 220–242, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[21] Michele Lanza and Stéphane Ducasse. Beyond language independent object-oriented metrics: Model independent metrics, 08 2002.

[22] W. Li and S. Henry. Maintenance metrics for the object oriented paradigm. In *[1993] Proceedings First International Software Metrics Symposium*, pages 52–60, 1993.

[23] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111 – 122, 1993. Object-Oriented Software.

[24] A. Madi, O.K. Zein, and Seifedine Kadry. On the improvement of cyclomatic complexity metric. *International Journal of Software Engineering and its Applications*, 7:67–82, 01 2013.

[25] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[26] Alberto S. Nuñez-Varela, Héctor G. Pérez-Gonzalez, Francisco E. Martínez-Perez, and Carlos Soubervielle-Montalvo. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164 – 197, 2017.

[27] Terence Parr. *The Definitive ANTLR Reference: Building Domain-specific Languages*. Pragmatic Bookshelf, 05 2007.

[28] Pedro Pinto, Tiago Carvalho, Joao Bispo, Miguel Ramalho, and João Cardoso. Aspect composition for multiple target languages using lara. *Computer Languages, Systems & Structures*, 53, 09 2018.

[29] Gordana Rakić and Zoran Budimac. Problems in systematic application of software metrics and possible solution. *5th International Conference on Information Technology ICIT*, 11 2013.

[30] M. Schroeder. A practical guide to object-oriented metrics. *IT Professional*, 1(6):30–36, 1999.

[31] Ramanath Subramanyam and M. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *Software Engineering, IEEE Transactions on*, 29:297– 310, 05 2003.

[32] Sander Tichelaar, Stéphane Ducasse, and Serge Demeyer. Famix and xmi. *Reverse Engineering - Working Conference Proceedings*, pages 296 – 298, 02 2000.