Universidad Autónoma de San Luis Potosí

Facultad de Ingeniería

Centro de Investigación y Estudios de Posgrado

Source code metrics extraction through the dynamic creation of source code data models and grammatical inference

T E S I S

Que para obtener el grado de:

Doctorado en Ciencias de la Computación

Presenta:

Alberto Salvador Núñez Varela

Asesor:

Dr. Héctor Gerardo Pérez González

San Luis Potosí, S. L. P. Diciembre de 2018

Universidad Autónoma de San Luis Potosí

Facultad de Ingeniería

Centro de Investigación y Estudios de Posgrado

Extracción de métricas de código fuente mediante la creación dinámica de modelos de datos del código fuente e inferencia gramatical

T E S I S

Que para obtener el grado de:

Doctorado en Ciencias de la Computación

Presenta:

Alberto Salvador Núñez Varela

Asesor:

Dr. Héctor Gerardo Pérez González

San Luis Potosí, S. L. P. Diciembre de 2018

Dedicatoria

A mi esposa Yarin y a mi hija Emily que me acompañaron a lo largo de este proceso

Agradecimientos

A mis papás y hermanos por su apoyo durante la realización de este trabajo, a mi asesor de tesis el Dr. Héctor Gerardo Pérez González por su guía en el desarrollo de la tesis, a mi comité de tesis conformado por el Dr. Francisco Eduardo Martínez Pérez y el Dr. Juan Carlos Cuevas Tello y a los sinodales Dra. Perla Velazco Elizondo y el Dr. César Arturo Guerra García por su valiosa participación.

También agradezco a los coordinadores de posgrado, a la Dra. Liliana Margarita Félix Ávila, al Dr. José Ignacio Núñez Varela y al Dr. César Augusto Puente Montejano por su ayuda durante todo este proceso.

Abstract

Source code metrics extraction is a complex process that has to be done automatically due to the current size of the software source code and the number of existing metrics. To aid this process, this thesis presents two methodologies. The first methodology allows extracting relevant data from the source code. This extraction is achieved by creating queries from the non-terminals symbols of the language context-free grammar, and from the result of these queries, a dynamic data model will be created from which the code metrics can be computed. Querying the language grammar allows access to all available data, providing more flexibility in the metrics definition. The second methodology allows the context-free grammar inference of relevant object oriented structures. These two methodologies are created according to two problematics found with current extraction methodologies and metrics tools and are described in the paragraphs below.

Current methodologies are based on source code representations, usually in the form of a model, but the metrics that can be defined are limited by the data available in those representations. This thesis attempts to solve this issue by dynamically creating the models according to the code metric to extract. The presented methodology is not limited by the programming language or code metrics it supports, since new languages can be incorporated and new code metrics can be defined.

In order to incorporate a new language the user must provide the context-free grammar that defines it, but this could represent a complicated process if the grammar is not available and has to be written. This thesis also presents a preliminary grammatical inference methodology to automatically infer the context-free grammar of certain code structures relevant to the metrics definition, which should aid the overall metrics extraction process. Context-free grammar inference is a complex and unresolved process, especially for programming languages, given that a language can be defined by any number of grammars and, a grammar in turn, can define an infinite number of strings that are valid for that language. Rules and heuristics can be defined in order to reduce the complexity of the problem to a specific domain, from which expected results can be obtained. In this thesis a solution is proposed by limiting the domain of the problem to infer the context-free grammar of relevant object oriented structures for Java-like languages. Also, syntax rules are created for helping the inference process to create a context-free grammar as desired.

A metric tool named MQLMetrics was created implementing the proposed extraction methodology. Several source code metrics (shown in Appendix A) are initially supported

by the tool. These metrics are extracted from the object and aspect oriented paradigms, and from XML files, this in order to showcase that the tool is programming paradigm and language independent.

Given that the objective of the MQLMetrics tool is to allow the definition and modification of new metrics, a case study was developed in which a set of code metrics, and variations of those metrics, are defined and computed by the tool. The definitions of that set of metrics are those as calculated by each of three common available metrics tools (Understand, Metrics 1.3.8 and MASU). In order to validate the results, the set of metrics were extracted from the JHotDraw system, a benchmark system in source code metrics research. The results from each tool were then compared with the results obtained by the MQLMetrics tool. The MQLMetrics tool was able to effectively reproduce the results as calculated from the other tools, although it was not expected to obtain a 100% match for all metrics values for each tool given the lack of formal metrics definitions and documentation, the overall difficulty regarding the metrics computation, and the specific programming language structures which can lead to miscalculations.

The results of the case study provide evidence that the proposed extraction methodology is able to allow the extraction and definition of source code metrics as required by the practitioner.

Índice

Introducción 1
Protocolo de tesis 6
Preguntas de investigación 6
Objetivo 7
Objetivos específicos 7
Contribuciones 8
Metodología de investigación 8
Estructura de la tesis 10
1. Trabajo previo sobre métodos de extracción de métricas de código fuente 11
1.1 Metodologías de extracción de métricas de código fuente 12
1.2 Metodologías de extraccción en herramientas de métricas 15
1.3 Resumen del Capítulo 22
2. Marco teórico sobre métricas de código fuente y su extracción 23
2.1 Métricas de código fuente 23
2.2 Extracción de métricas de código fuente 27
2.2.1 Problemas encontrados en metdologías de extracción actuales 28
2.3 Inferencia gramatical 33
2.4 Resumen del Capítulo 35
3. Metodología de extracción mediante la creación de modelos dinámicos e inferencia gramatical 36
3.1 Metodologías para la extracción de métricas de código fuente e inferencia gramatical 37
3.2 Arquitectura actual de herramientas 38
3.3 Terminología 40
3.4 Metodología para la extracción de métricas de código fuente 41
3.4.1 Creación de modelos de datos dinámicos del código fuente 43
3.4.2 Consideraciones especiales en las gramáticas independientes de contexto 50
3.4.3 Metric Query Language 52
3.5 Metodología de inferencia gramatical 53
3.5.1 Patrones 55
3.5.2 Definición léxica 59

3.5.3 Definición sintáctica 61
3.5.4 Proceso de inferencia 61
3.5.5 Ejemplo de inferencia de una gramática independiente de contexto 72
3.6 Resumen del Capítulo 76
4. Resultados y comparación con otras herramientas 78
4.1 Descripción de MQLMetrics 78
4.1.1 Una consulta de ejemplo para métricas orientadas objetos comúnes 79
4.1.2 Ejemplo del peso cognitivo 82
4.2 Soporte generalizado de entradas de MQLMetrics 85
4.3 Caso de estudio 87
4.3.1 Selección de herramientas de métricas 87
4.3.2 Selección de métricas de código fuente 88
4.3.3 Selección del sistema y lenguaje de programación 90
4.3.4 Resultados 90
4.4 Inferencia gramatical 97
4.5 Respuestas a las preguntas de investigación 99
4.6 Amenzas a la validez 102
4.6.1 Metodología de extracción de métricas de código fuente 102
4.6.2 Inferencia gramatical 103
4.6.3 Complejidad 104
4.7 Resumen del Capítulo 104
Conclusiones 106
Referencias 110
Apéndice A - Métricas de código fuente 116
Apéndice B - Código de ejemplo 117
Apéndice C - Gramática de Java 118

Index

Introduction 1

Thesis protocol 6

Research questions 6

Objective 7

Specific objectives 7

Research contributions 8

Research methodology 8

Thesis structure 10

1. Related work on source code metrics extraction methodologies 11

1.1 Source code metrics extraction methodologies 12

1.2 Extraction methodologies in software metric tools 15

1.3 Chapter summary 22

2. Theoretical framework on source code metrics and their extraction 23

2.1 Source code metrics 23

2.2 Source code metrics extraction 27

2.2.1 Issues found in current extraction methodologies 28

2.3 Grammatical inference 33

2.4 Chapter summary 35

3. Extraction methodology through the creation of dynamic data models and grammatical inference 36

3.1 Methodologies for code metrics extraction and grammatical inference 37

3.2 Current tools architecture 38

3.3 Terminology 40

3.4 Source code metrics extraction methodology 41

3.4.1 Creation of dynamic data models from source code 43

3.4.2 Context-free grammars special considerations 50

3.4.3 Metric Query Language 52

3.5 Grammatical inference methodology 53

3.5.1 Patterns 55

3.5.2 Lexical definition 59

3.5.3 Syntactical definition 61
3.5.4 Inference process 61
3.5.5 Context-free grammar inference example 72
3.6 Chapter summary 76
4. Results and comparisons with other tools 78
4.1 MQLMetrics description 78
4.1.1 A query for common object oriented metrics example 79
4.1.2 Cognitive weight example 82
4.2 MQLMetrics generalized inputs support 85
4.3 Case study 87
4.3.1 Metrics tools selection 87
4.3.2 Source code metrics selection 88
4.3.3 System and programming language selection 90
4.3.4 Results 90
4.4 Grammatical inference 97
4.5 Research questions answers 99
4.6 Threats to validity 102
4.6.1 Source code extraction methodology 102
4.6.2 Grammatical inference 103
4.6.3 Complexity 104
4.7 Chapter summary 104
Conclusions 106
Future work 108
References 110
Appendix A – Source code metrics 116
Appendix B - Code example 117
Appendix C - Java grammar 118

Tables Index

Table 1.1 Tools access information………………………………………………………………………… 20

Table 1.2 Tools characteristics………………………………………………………………………………… 21

Table 2.1 Issues found with current software metrics extraction mechanisms………………. 29

Table 3.1 BMs for WMC……………………………………………………………………………………………… 43

Table 3.2 Context-free grammar example……………………………………………………………… 44

Table 3.3 Derivations examples…………………………………………………………………………………… 45

Table 3.4 Example queries……………………………………………………………………………………………… 46

Table 3.5 BMs to calculate the NOAAM metric…………………………………………………………. 49

Table 3.6 Patterns for recursive structures…………………………………………………………………… 57

Table 3.7 Creation of an inferred grammar for similar instructions. Example 1………………. 58

Table 3.8 Creation of an inferred grammar for similar instructions. Example 2………………. 58

Table 3.9 Lexical definition…………………………………………………………………………………………… 60

Table 3.10 Tokens examples…………………………………………………………………………………………… 60

Table 3.11 Lexical definition for a Java subset………………………………………………………………… 60

Table 3.12 Lexical and syntactical definition…………………………………………………………………… 61

Table 3.13 Expressions finding example………………………………………………………………………… 65

Table 3.14 Expressions finding first context details……………………………………………………… 66

Table 3.15 Expressions finding second context details…………………………………………………… 66

Table 3.16 Expressions reduction………………………………………………………………………………… 68

Table 3.17 Context free grammar creation…………………………………………………………………… 70

Table 3.18 Expressions reduction example………………………………………………………………….. 73

Table 3.19 Context free grammar creation…………………………………………………………………… 75

Table 4.1 BMs and MQL queries for the NCLASS and NOM metrics……………………………… 79

Table 4.2 MQL queries for the NCLASS and NOM metrics………………………………………….. 80

Table 4.3 Direct metrics and MQL queries for the object oriented model……………………. 80

Table 4.4 BMs and MQL queries for the cognitive weight metric…………………………………… 83

Table 4.5 Object oriented systems………………………………………………………………………………….. 86

Table 4.6 Aspect oriented systems………………………………………………………………………………….. 86

Table 4.7 Software metrics tools………………………………………………………………………………….. 88

Table 4.8 Selected metrics per tool………………………………………………………………………………….. 89

Table 4.9 Metrics computation method………………………………………………………………………….. 91

Table 4.10 Match percentage per metric………………………………………………………………………….. 91

Table 4.11 Methods defined in the DefaultDrawingView class………………………………………….. 92

Table 4.12 Metric values per tool……………………………………………………………………………………… 93

Table 4.13 JHotDraw classes with its corresponding children………………………………………..... 94

Table 4.14 Miscalculated metrics per tool………………………………………………………………………… 96

Table 4.15 Grammatical inference examples…………………………………………………………………….. 98

Figures Index

Figure 2.1 (a) Source code snippet, (b) Extracted metrics values…………………………………… 26

Figure 2.2 Common architecture of a software metrics tool……………………………………… 27

Figure 3.1 Metrics extraction architecture using source code's intermediate representation……………………………………………………………………………………… 39

Figure 3.2 Simplified architecture using dynamic models……………………………………… 39

Figure 3.3 Proposed optimal architecture……………………………………………………… 40

Figure 3.4 Top-down insertion…………………………………………………………………… 46

Figure 3.5 Tree construction for Query 1……………………………………………………… 47

Figure 3.6 Tree construction for Query 2……………………………………………………… 47

Figure 3.7 Input to calculate the metric NOAAM……………………………………………… 48

Figure 3.8 Result trees for BM1, BM2 and BM3………………………………………………… 49

Figure 3.9 Merging process……………………………………………………………………… 49

Figure 3.10 Final merged tree…………………………………………………………………… 50

Figure 3.11 MQL gramar………………………………………………………………………… 52

Figure 4.1 Result tree…………………………………………………………………………… 81

Figure 4.2 Populated result tree………………………………………………………………… 81

Figure 4.3 Input example……………………………………………………………………… 83

Figure 4.4 Code nodes identification…………………………………………………………… 84

Figure 4.5 Result tree of each BM……………………………………………………………… 84

Figure 4.6 Merged tree………………………………………………………………………… 84

Introduction

Software metrics are an essential aid to the software measurement process. Software measurement is a task that is carried out during all the phases of the software development process (Tahir and Jafar, 2011). During this process, many intermediate or final software products are developed and measured by software product metrics (Gómez et al., 2008). One of those products is the program's source code, which is part of the final software system and is measured to assess its quality. Source code metrics are a type of product metrics that focus on measuring the source code of a system (Scotto et al., 2006). This measurement is achieved by mapping a particular attribute of the code to a numerical value (Lanza and Marinescu, 2006), taking in consideration that for some cases that value can be also categorical or ordinal.

Source code metrics such as Lines of Code (LOC), McCabe Cyclomatic Complexity (Mccabe, 1976) and the Halstead Software Science Metrics (Halstead, 1977), are widely known and have been studied for many years. It is usually mentioned that LOC was first used in the

late 1960's (Fenton and Neil, 1999), and the McCabe and Halstead metrics in the late 1970's. Those metrics were the most studied during a period of time that spanned from the 1970's to the beginning of the 1990's, when object oriented metrics became popular thanks to the research and metrics sets proposed by Chidamber and Kemerer (1994) and Li and Henry (1993). Source code metrics are a very important component for the software measurement process, and are commonly used to measure and improve the quality of the source code itself. Additionally, these metrics are being used in a wide variety of applications and experiments related to source code in general (i.e. fault prediction, testing, refactoring) to assess the overall quality of the software (Eski and Buzluca, 2011, Al Dallal, 2012, Radjenovic et al., 2013, Bavota et al., 2015, Almugrin et al., 2016). Moreover, source code metrics are gaining importance due to their applications in software product lines and programming concerns (Nuñez-Varela et al., 2017). In order to accomplish the different measurement needs, many source code metrics, designed from different programming paradigms, have been proposed, studied and validated throughout the years (Kulkarni et al., 2010; Misra et al., 2012; Sharma et al., 2012), with new metrics (Malhotra and Khanna, 2014, Parthipan et al., 2014) and studies being proposed constantly.

Source code metrics are obtained from the source code by a process known as metrics extraction. It is a complex process because of factors such as the current size of the software source code (which can consist of millions of lines of code), the amount of data required to compute the metrics, and the amount of programming languages in which the code can be written. These factors make it almost impossible to perform a manual metrics

extraction process (Etzkorn and Delugach, 2000), so specialized software for metrics extraction, able to handle any number of code files, has to be used for the extraction process. Many methodologies and metrics tools have been proposed that effectively help to achieve an automatic metrics definition and extraction process, but still they present some important issues. Two main problematics are found: 1) current methodologies and tools are dependent of the metrics and languages they support and do not provide mechanisms to define new metrics and languages, and 2) grammatical inference can help to the incorporation of new languages by automatically inferring the context-free grammar of the programming language, but it is a complex and still unresolved problem. In the following paragraphs both problematics are described.

Software metrics tools have been studied (Lincke et al., 2008, Nunez-Varela et al., 2016, Mshelia et al., 2017) and researchers have found issues regarding their use. The most important issues reported are that metrics results are different across tools, and the dependency on the metrics and languages the tool support. Research methodologies for metrics extraction try to provide common or formal extraction mechanisms, usually by means of imperative or declarative languages, which are not tied to a specific language or set of metrics, however, they present certain restrictions. The most important restriction is not allowing the definition of new metrics effectively (Budimac et al., 2012; Mshelia et al., 2017). This is because they are based on source code representations, such as models, meta-models or graphs, from which the metrics are defined and extracted, and these representations might not contain enough data to define a desired metric, thus limiting the range of metrics that can be defined. Also, these representations, especially those in

the form of meta-models, are bounded to a certain programming paradigm. In fact, the methodologies found during the research process of this thesis are exclusively designed for the object oriented paradigm, neglecting other important paradigms such as the aspect oriented paradigm and the feature oriented paradigm. These three paradigms, along with the procedural paradigm, were found to be the most used paradigms during the research process of this thesis (Nuñez-Varela et al., 2017). The object oriented paradigm is the most common nowadays, it can be used to develop almost any type of system and most of the research is based on it. The procedural paradigm is mainly used in legacy systems in industry research (Misirli et al., 2011, Arpaia et al., 2010, Durisic et al., 2013, Kwiatkowski and Verhoef, 2013, Yamashita and Moonen, 2013), and the aspect oriented paradigm has been gaining importance due to its applications in programming concerns (Figueiredo et al., 2011). Finally, the feature oriented paradigm is a future trend with the current relevance of software product lines and big scale software (Nuñez-Varela et al., 2017).

In order to be able to incorporate new languages and support multiple programing paradigms, the context-free grammar of the languages can be used as a common ground for their definition. This grammar might be complex to write and can consists on hundreds of productions. To further aid the metrics extraction process, the grammar can be inferred automatically by a process known as grammatical inference. To this day, and to the best of our knowledge, context-free grammar inference is still a complex and unresolved problem, and can be considered as a NP problem (Stevenson and Cordy, 2014), given that a language can be defined by any number of grammars and, a grammar can accept an

infinite number of valid strings. Attempts have been made to solve the problem by using incomplete grammars, or using the help of a human expert in the inference process, but this represents considerable restrictions in the process.

This thesis presents a methodology for aiding the code metrics extraction process and a methodology for context-free grammar inference, which can further aid the extraction process. The ISO/IEC 24765 standard defines a methodology as a “specification of the process to follow together with the work products to be used and generated, plus the consideration of the tools involved”. The methodologies presented here specify the processes to follow and the products that can be generated.

The methodology for aiding the metrics extraction allows the definition of new languages by providing the context-free grammar of such language, and the definition of new metrics by querying that language. This methodology uses an intermediate representation of the source code in the form of a model, as many other methodologies, but with the difference that this model is created dynamically with each query. An intermediate representation of the source code is any structure able of representing the source code for further processing, for example models, XML files, Syntax Trees, etc. These intermediate representations are usually easier to handle and are able to represent a unique structure across several languages.

Although these source code representations are easier to handle, their construction require specialized parsers and are created statically, that is, if the source code changes the representation has to be reconstructed. Since the model used as an intermediate

representation in this thesis is recreated with each query, it is always complete and contains all the needed information, additionally, since the model is created from the language grammar, all the source code data can be obtained. Furthermore, a preliminary attempt to infer the context-free grammar from the source code is presented, which provides a simplification of the process since the language grammar will no longer be needed as an input. Grammatical inference for context-free grammars is an unresolved problem, and in this thesis only the source code structures relevant for metrics extraction are inferred, including classes, member variables and methods definitions. Although no attempt is made to infer the complete language grammar, the proposed grammatical inference methodology might be powerful enough to infer all other source code structures and construct a complete context-free grammar, but other considerations must be taken, along with several tests, that are not part of the scope of this thesis.

Thesis protocol

This Chapter describes the thesis protocol, including the definition of the research questions, the objective, and the specific objectives of the thesis. Also, the research contributions are listed, and the research methodology followed during the development of the thesis is presented.

Research questions

1. How to create a methodology for source code metrics extraction without being dependent on the code metrics and programming languages it supports?

2. Which mechanisms can be created to define new code metrics and programming languages?
3. How to create a source code intermediate representation with all data needed to calculate the code metrics?
4. Which programming structures are needed to cover the source code measurement needs?
5. How to infer the programming language context-free grammar or parts of it?

Objective

To create a methodology for aiding the source code metrics extraction process by allowing the definition and computation of code metrics, and the addition of new programming languages, according to the measurement needs. This will be achieved by querying the language context-free grammar. The process is further aided by inferring the context-free grammar of source code structures relevant to the metrics extraction process.

Specific objectives

- To create a methodology, and its related mechanisms, that allows the extraction of source code metrics data.
- To allow the definition and extraction of user defined metrics and the incorporation of new languages.
- To create an intermediate representation of the source code in the form of a dynamic data model, containing all the needed data to compute the desired metrics.

- To identify the key programming language structures from the object oriented paradigm that are the most relevant for code metrics extraction.
- To create a grammatical inference process capable to generate a context-free grammar for the key programming language structures.

Research contributions

The main contributions of this thesis are:

1. A methodology that allows the user to define and extract metrics according to their needs, and the definition of the MQL language that formalizes the methodology. Based on MQL, the metrics tool MQLMetrics is introduced as the implemented software solution for code metrics extraction.
2. A methodology that allows inferring the context-free grammar of certain object oriented programming language structures.

Grammatical inference is still an unsolved problem, and no methodology or algorithm, to the best of our knowledge, has been proposed that can infer a complete language grammar. The proposed methodology infers the grammar for the type of language structures we are interested in, but can be extended and tested to infer the complete language grammar. This inference methodology can be seen as the main contribution of this thesis.

Research methodology

A research methodology with a circular loop was followed throughout the development of this thesis. It consists on a continuous analysis and development of the thesis, always

reviewing and taking in consideration the state of the art on the subjects of interest. It consists of four main activities: state of the art revision, analysis, development, and implementation and validation. Each activity is described below:

- State of the art revision: in this activity a deep bibliographic analysis was made in order to found relevant studies useful for this thesis. These studies include similar works, studies describing the source code metrics extraction problematic, and overall studies that are helpful for constructing the theoretical framework for this thesis. This activity was carried out during the entire development of this thesis.
- Analysis: this activity consisted on the comprehension and study of every aspect relevant to the development of the thesis. The first aspect that was carefully analyzed was the current issues with source code metrics extraction. From this analysis, the two methodologies described in this thesis were proposed. Both methodologies were also deeply analyzed in order to find solutions that were both correct and not very complex.
- Development: in this activity, the theoretical methodologies and their solutions that were obtained from the analysis activity are implemented as a software metrics tool. This tool contains all the necessary mechanisms for the metrics definition and extraction, and the incorporation of new programming languages. Also, a grammatical inference software was developed to further aid the metrics extraction process.
- Implementation and validation: in this activity, the developed tool and grammatical inference software were validated by hand initially. To provide a

scientific validation, a case study was proposed in which the tool extracts metrics values as defined by other commercial tools, this in order to compare the results and verify the proper functioning of the tool.

Through this research methodology, the thesis presented in this document and the proposed objectives were achieved. The overall structure of the thesis document is described below.

Thesis structure

This thesis is divided in four main chapters, conclusions, references and three appendices. Chapter 1 presents related work on current code metrics extraction methodologies, Chapter 2 presents a theoretical framework on source code metrics and their extraction, Chapter 3 presents the proposed methodology for metrics extraction and the proposed methodology for context-free grammar inference, and Chapter 4 presents the results of this thesis. Finally conclusions about the overall thesis and the knowledge gained are presented, along with the future work and references.

Three appendices are also part of this thesis document: Appendix A lists the source code metrics that are supported by the developed tool, Appendix B shows a source code used in Chapter 4.1.1, and finally Appendix C shows the complete Java grammar used in the examples and case study from Chapter 4.

Chapter 1

1. Related work on source code metrics extraction methodologies

In this Chapter we present related work on methodologies for source code metrics extraction. Relevant papers proposing tools for metrics extraction that include a description of the used methodology for the actual extraction process are also included. We are mainly interested in the structure, functioning, and the mechanisms used to define and extract metrics. A comparison and discussion is made with the works considered to be the most similar to what is proposed in this thesis (Alexan et al., 2016, Alshayeb et al., 2018)

A brief overview and description of currently available software metrics tools (free or commercial) is also presented, including a comparison with the tool that is most similar to MQLMetrics.

1.1 Source code metrics extraction methodologies

A generalized structural model is defined using graphs as an abstract representation of the source code by Cogan and Shalfeeva, 2002. This model covers all the combinations of the functional and informational components of the program and their relationships, providing a uniform definition of well-known models in the literature. From this model, software metrics can be defined and extracted from the graph characteristics (i.e. nodes and edges). The generalized model proposed by the authors does provide a common structure for metrics definition and extraction; however, several models have to be created in order to compute a wider range of code metrics. No information on how the models are created is presented, and only one metric (cyclomatic complexity) is presented as an example.

Marinescu et al., 2005 creates an object oriented meta-model in which relevant entities (i.e. classes and methods) and relations (i.e. inheritance) are defined. The metric extraction is performed by constructing collections for the meta-model entities obtained by the code analysis, and then using the meta-model by means of four elementary mechanisms: navigation, selection, arithmetic and filtering. The meta-model can be manipulated in three ways: structured based, in which the manipulation can be made via a programming language, repository based, in which the meta-model data is stored in a database and manipulated using Structured Query Language (SQL) queries, and using SAIL, a query language based on SQL that incorporates data structures to hold and manipulate the data. The authors present a comparative evaluation between the three ways of manipulation; however, examples of metrics definitions, queries or computation are not

presented. The number of metrics that can be extracted is restricted by the information on the meta-model. No details are presented on how the meta-model is created or extracted.

Alikacem and Sahraoui, 2006 present a Unified Modeling Language (UML) based meta-model that is used as a source code representation. It can be manipulated via a language, named Metric Description Language, for metrics extraction. The language is composed by predefined calls, operations to access object properties, arithmetic operators and iterators. The authors presented a more detailed version of their work (Alikacem and Sahraoui, 2009) and named their language PatOIS, a declarative language that shares a resemblance with the Object Constraint Language (OCL). Even though the language provides means to define and extract metrics, it is complicated to understand and use at first glance. The authors list a total of twenty three metrics that have been already implemented, but the PatOIS definition is only presented for eight metrics, and as with the works discussed above, the metrics that can be defined are restricted by the data available in the meta-model. No further details on the overall implementation of the methodology are presented.

Baroni and e Abreu, 2003, presented a Formal Library for Aiding Metrics Extraction (FLAME) as a base to formalize metrics definitions using OCL upon a UML meta-model. The paper presents the UML meta-model and several formal metrics definitions using OCL. These definitions are based on a group of functions that calculate data sets from which the metrics are computed. The metrics that can be defined are restricted by the

data in the meta-model, and by the data that can be manipulated via OCL. No results or real life examples are provided in the paper.

Núñez-Varela et al., 2013, presented a methodology for metrics extraction by querying an abstract representation of the language grammar. A tree is used as the abstract representation, and contains non-terminals symbols of the grammar arranged according to their parent-children relations. The data extracted from the source code is assigned to the corresponding non-terminal in the tree. Each path of the tree, that can be generated starting from the root to any non-terminal child node, could then be queried to extract the corresponding data. The methodology could effectively extract certain sections of the code to compute count metrics mainly; however, since a tree cannot contain loops, any given branch could not contain the same symbol more than once, limiting the data that could be queried and the number of metrics that could be created. The definition of a metric consists on providing the entire sequence of symbols in a given path, which could be difficult for big grammars.

Rakic et al., 2017, presented an enriched Concrete Syntax Tree (eCST), a source code intermediate representation useful for static quality code analysis. The idea is to enrich the source code syntax trees with universal nodes. These nodes represent the necessary source code elements and are unified for all programming languages. For instance, a LOOP_STATEMENT node is presented as an example that contains statements such as loop, for, while, etc. These nodes are created by observation of the common characteristics of the programming languages.

Alshayeb et al., 2018, presented a metrics extraction framework based on the Software Product Metric Definition Language (SPMDL). SPMDL allows defining metrics as an XML-based description language, and uses meta-models as a source code representation for metrics extraction. Several built-in metrics are supported as part of their framework, where variables and queries are used to extract metrics from a DMM meta-model, but it is specified that the XML metrics definitions are not linked to any specific type of implementation.

Other works (Allier et al., 2010, Aral and Ovatman, 2013, Tempero and Ralph, 2017) were found that extract metrics using graphs, but they are structured specifically for coupling and cohesion metrics only. Below, papers presenting metrics tools are analyzed that presents the methodology used for the actual extraction process.

1.2 Extraction methodologies in software metric tools

The architecture and characteristics of TAC++, a tool that allows metric management for C++, is presented by Fioravanti and Nesi, 2000. The tool allows the extraction of code metrics in two groups: low level metrics (counting metrics and code features) and high level metrics, which are metrics that can be user defined. These high level metrics can be defined based on the low level metrics predefined data, by assigning new weights and sums for calculations. The architecture and overall information of the tool are described; however, no information on how the metrics are defined and computed is presented. Even though they allow the definition of metrics, they are based on already predefined metrics, limiting the number of metrics that can be effectively defined.

Four tools, WebMetrics (Scotto et al., 2004), OOMeter (Alghamdi et al., 2005), OOMT (E, 2009) and EMBER (Harmer and Wilkie, 2002), present similar approaches in their structure and overall implementation. A parser extracts relevant data (object oriented entities and relations) from the source code and is stored in a database, the metrics are then extracted by querying that database (Webmetrics, OOMeter and EMBER use SQL for this purpose). The EMBER tool also exemplifies the post-processing of the result data by using the C language to calculate a metric. OOMT is built for Java, but the other tools are able to support new languages by incorporating new parsers. The papers do not present details on how the models are constructed or the data they contain. The metrics they can support are limited by the data in each model.

The tools SMIILE (Budimac et al., 2012) and MASU (Higo et al., 2011), are based on the use of syntax trees as intermediate representations of the source code. These trees manage the differences between programming languages, creating a common structure for metrics extraction. This is achieved by creating common nodes that represent the same language structures (i.e. loops, conditionals). The metrics are extracted from the data in the trees, and they can support several languages by providing independent syntax tree builders for each language. These tools do not allow the definition of new metrics, and the number of metrics they can extract is limited by the data that the trees contain.

The tools QScope (Eichberg et al., 2006) and DM Crawler (El-Wakil et al., 2005), use a source code representation in the form of Extensible Markup Language (XML) and XML Metadata Interchange (XMI) files respectively. From these code representations, the XQuery language is used to define and extract the metrics, limited according to the data

that the representation contains. As with the tools described above, no details are presented on how the code representations are created or what data they contain.

An open source extendible software metrics tool is presented by Alexan et al., 2016. The tool supports seven code metrics and it can be extended by allowing the user to add new code metrics by defining them in Python code, but the paper only presents examples of simple count metrics, it is not clear if more complicated metrics such as LCOM can be calculated. Also, the authors propose as future work to calculate metrics not depending on the file content, but rather in the relations among files, this is an indication that metrics such as CBO cannot currently be calculated, and represents a very important limitation in the tool.

The tool MetricAttitude++ is presented by Francese et al., 2017. The tool is mainly presented as a visualization tool, it supports object oriented size metrics, and uses information retrieval techniques to find relations and similitudes among the code files. The tool uses a third party Java parser that allows the navigation of the code attributes, from which relations and metrics values are calculated, thus limiting the tool to only support Java and extract the predefined metrics.

From the papers presented above, the works by Alexan et al., 2016 and Alshayeb et al., 2018 are the most similar to this thesis. The extendible open source tool by Alexan et al., 2016 has the objective to allow the definition of new metrics by plugging them into the tool. This is achieved by adding Python code files containing the metric definition into the

tool. This approach is similar to ours in which an infrastructure is presented and the metrics computation is made with a common programming language (Java in our case). One of the strengths of the tool is the ability to calculate churn metrics and to measure bugs, although they are not considered as source code metrics, but making heavy use of git and databases. In fact the tool is only able to analyze git repositories greatly limiting its functionality. The authors validate the tool by reproducing the results from other studies, which is similar to our approach on evaluating MQLMetrics. The main disadvantage of the tool is the fact that it does not present mechanisms for the incorporation of new programming languages, and that the tool itself is written and has to be modified in Python, on the other hand, the extraction methodology presented in this thesis allows the incorporation of new languages, and since it presents the theoretical methodology instead of simply presenting a tool, it can be written in any language.

The metrics extraction framework proposed by Alshayeb et al., 2018, consists on a well defined, and very complete infrastructure of modules for metrics definition and extraction including a parser, the meta-model database, the metrics definition, and the metrics definition API. All those modules have a similar component in the extraction methodology presented in this thesis.

The authors state that a parser is needed for each supported programming language, and that represents a major disadvantage. On the other hand, the parser used in this thesis is unique and is able to handle any language by manipulating its context-free grammar.

The meta-model is stored and manipulated in a database which adds a layer of complexity and a third party tool, in our case it is directly accessed and created with each query and since it is dynamically created it does not need to be stored.

The metrics definition mechanism is one of the strongest contributions of the paper by using an XML-based Software Product Metrics Definition Language (SPMDL). This language can effectively define a metric, along with additional information (i.e. name, acronym, authors, etc.), and be computed from that definition. In our case, the proposed Metrics Query Language (MQL) is not able to define the metric structure, it defines the data needed to compute the metric, and the computation is made in any other language (i.e. Java, C#, etc.). This choice was made because defining a metric in such a definition language is complex to write and read, also a lot functionality has to be incorporated into the language in order to compute the metric. In fact SPMDL uses XMLMatch, a third party XML language in order to compute the metrics. It is not necessary to incorporate such complexity when the computation can be made with any common programming language.

Finally the metrics definition API defined by the authors, provide the means to read the metrics definitions and query the model. The computation is based on queries and variables. It is very similar to our approach, with the main difference that in SPMDL the queries and variables are a built-in set, and in our approach the queries are written by the user as defined by MQL. The main advantage of our approach is that by allowing the user to write the desired queries, all the data is available for the metrics computation, and while the metrics extraction framework and SPMDL comprise a very complete and robust

mechanism for metrics definition and extraction, the data is still limited by the meta-model created and the queries that can be made.

Finally, we acknowledge the existence of many available software metrics tools (commercial or free). The most common tools according to our research (Nuñez-Varela et al., 2017) are: Understand, JHawk, CKJM and Metrics 1.3.6 for the object oriented programming paradigm, AOP metrics for the aspect oriented paradigm, and Featureous for the feature oriented paradigm. Understand and JHawk are black box commercial tools whose functioning is unknown, on the other hand, CKJM, Metrics 1.3.6, AOP metrics and Featureous are open source tools. CKJM extracts metrics from Java bytecode and Metrics 1.3.6 is a plugin that provides metrics visualization from the Eclipse IDE. Except for Understand, all the other tools only support Java, and only Understand and JHawk allows the definition of new metrics, but these new metrics can only be defined from the predefined metric's data, so no real mechanisms to create new metrics definitions are presented. No further discussion on their extraction mechanisms can be done for these tools since no documentation on their implementation is available. Tables 1.1 and 1.2 present information about the tools and their more relevant characteristics.

Table 1.1. Tools access information

Tool	Website	Last version
Understand	https://scitools.com/	2018
CKJM	http://www.spinellis.gr/sw/ckjm/	2012
Metrics 1.3.6	http://metrics.sourceforge.net/	2013
JHawk	http://www.virtualmachinery.com/jhawkprod.htm	2017
AOPMetrics	http://aopmetrics.stage.tigris.org/	2006
Featureous	http://featureous.org/	2016

Table 1.2. Tools characteristics

Tool	**Language(s)**	**New languages**	**Metrics**	**New metrics**	**Paradigm**
Understand	C, Ada, Basic, Pascal, C++, C# and Java	No	100+	Yes, from predefined data	PP, OOP
CKJM	Java bytecode	No	7	No	OOP
Metrics 1.3.6	Java	No	17	No	OOP
JHawk	Java	No	100+	Yes, from predefined data	OOP
AOPMetrics	AspectJ	No	18	No	AOP
Featureous	Java	No	-	No	FOP

The objective of the presented metrics extraction methodology is to allow the definition of new metrics and the incorporation of new languages. Based on that objective, the available tool that resembles more to what we are trying to achieve is Understand. It supports a wide variety of programming languages for the procedural and the object oriented paradigms, but lacks support for the aspect and feature oriented paradigms. It does not provide mechanisms to incorporate new programming languages. On the other hand, it supports a large number of metrics and allows the definition of new metrics, but only from the already predefined data (i.e. the tool calculates LOC per method, so a new metric could be average LOC in methods by using the LOC per method data), so no new metrics definitions is really supported. The MQLMetrics tool created from the presented methodology, effectively allows the incorporation of new languages and the definition or modification of existing metrics, so it is metrics and language independent.

1.3 Chapter summary

In this Chapter an in deep analysis of related work was presented, along with a brief description of available software metrics tools. Also, a comparison and discussion between the works by Alexan et al., 2016 and Alshayeb et al., 2018 and this thesis is presented. It was found that the lack of mechanisms for defining incorporating new programming languages and new metrics definitions is still an issue with all current methodologies, this provide evidence that new technology on metrics extraction is needed and that the contribution of this thesis are relevant. In the following Chapter a theoretical framework on source code metrics extraction is presented.

Chapter 2

2. Theoretical framework on source code metrics and their extraction

In this Chapter general information about source code metrics theory is presented, including definitions and examples, in order to provide the necessary information to the reader. After the general information, the code metrics extraction process is discussed, and more importantly the problematic with current tools and methodologies for metrics extraction is presented as a set of issues found in the literature. Also, a brief description of context-free grammar inference and its problematic is presented.

2.1 Source code metrics

In Software Engineering, source code metrics are part of the software measurement process, which is carried out during all the phases of the software life cycle (Tahir and

Jafar, 2011). The software is mainly measured for quality evaluation purposes. This is achieved by obtaining relevant data from which conclusions about certain quality attributes can be reached using software metrics. A software metric can be seen as a function whose input is the software, or any part of this software, and the output is a numerical value (IEEE Std. 610.12-1990) that can be interpreted as the degree in which the software contains a specific quality attribute. It is important to note that, although most of the times the output of the metric is a numerical value (Lanza and Marinescu, 2006), it can be also text or a category.

Given that metrics can be used in any part of the software life cycle, they are divided in three types: project, process and product metrics (Scotto et al., 2006). Each type of metrics has been designed and used to measure exclusively a software project, a software process or a software product.

In this thesis, a type of software product metrics known as source code metrics is studied. Source code metrics are designed to measure the source code of the software, which is one of the main software products that can be obtained during the software life cycle (ISO/IEC 24765). These metrics are used to measure source code quality attributes and design characteristics, such as complexity, maintainability, fault proneness, size, coupling and cohesion (Nuñez-Varela et al., 2017). Many other characteristics that can be measured through code metrics are linked to a specific programming paradigm, such as the number of classes or inheritance attributes for the object oriented paradigm for example. In fact, current research in source code metrics is heavily focused on the object oriented paradigm, followed by the aspect oriented paradigm and the feature oriented

paradigm, with few studies still measuring the procedural paradigm (Nuñez-Varela et al., 2017). Still, metrics are not exclusive, since some code metrics designed for a specific paradigm can be used to measure another paradigm.

The most common code metrics are LOC and the McCabe Cyclomatic Complexity (Mccabe, 1976). Those metrics were designed for the procedural paradigm and have been widely used over the years to this day (Fenton and Neil, 1999), but given that the object oriented paradigm has been the dominant paradigm for years now, new code metrics specifically designed for the paradigm have been proposed. The most important metrics for the object oriented paradigm are the set usually known as the CK metrics, proposed by Chidamber and Kemerer, 1994. These metrics are some of the most common and widely used object oriented metrics to this day.

Each code metric has a definition that allows extracting certain source code attributes mainly as numerical values. For example, LOC extracts the total number of code lines. Figure 2.1 presents a source code snippet with some metrics values.

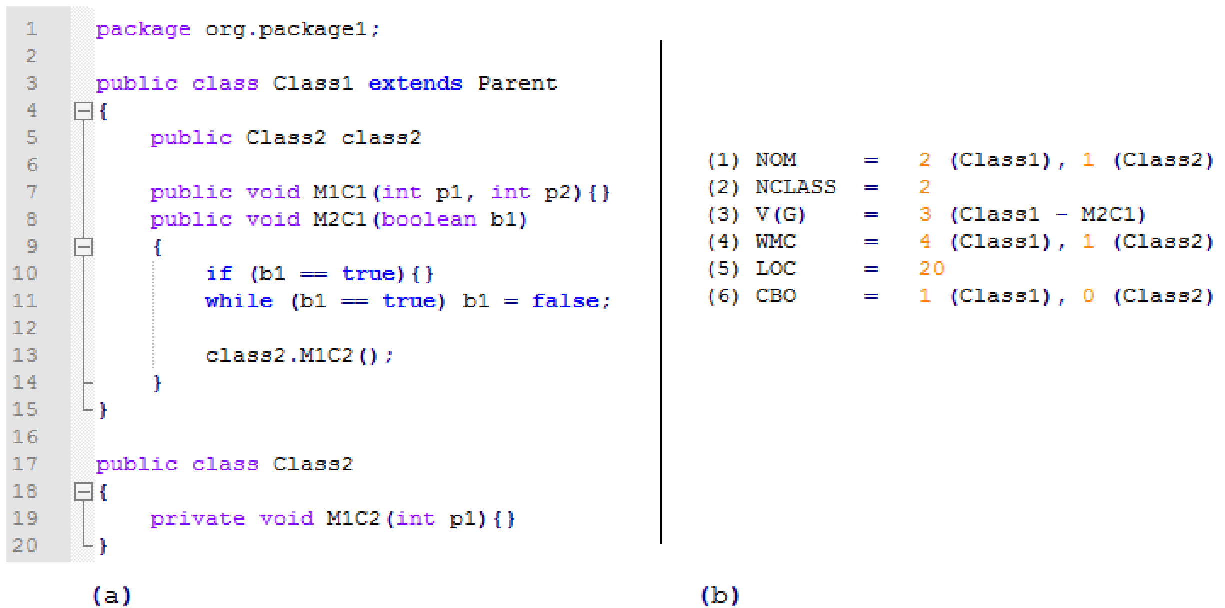

Figure 2.1 - (a) Source code snippet, (b) Extracted metrics values

The metrics in Figure 2.1 (b) are defined as follows (Mccabe, 1976, Li and Henry, 1993, Chidamber and Kemerer, 1994):

1. **Number of methods (NOM)**: counts the number of methods in each class.
2. **Number of classes (NCLASS)**: counts the total number of classes in the system.
3. **Cyclomatic complexity (V(G))**: counts the number of paths the code can have.
4. **Weighted methods per class (WMC)**: sums the cyclomatic complexity of each method per class.
5. **Lines of code (LOC)**: counts the total number of code lines.
6. **Coupling between object (CBO)**: counts the number of different classes a class has relationships with.

Most of the metrics, as indicated in Figure 2.1 and in their descriptions above, are extracted according to a scope. This scope is an enclosing context in which the values are

valid. In the example, the metrics LOC and NCLASS do not have scope, they are extracted from the whole system, but the metrics NOM, WMC and CBO are extracted at class level. Furthermore, V(G) is extracted per method per class, providing a more granular scope.

The metrics values can then be used to evaluate and obtain conclusions about the overall quality of the source code, with applications to fault prediction, bugs detection, code smells, clone detection, maintainability, code changes, testing, reusability, evolution, usability, refactoring, cohesion, coupling, inheritance, understandability, etc. (Nuñez-Varela et al., 2017).

2.2 Source code metrics extraction

Source code metrics are obtained from the source code by a process commonly known as metrics extraction. It is a complex process since it usually involves the use of dedicated parsers and algorithms in order to support a set of metrics for a given programming language. The software systems that are able to extract code metrics are usually known as software metrics tools. A common architecture of a metrics tool is depicted in Figure 2.2.

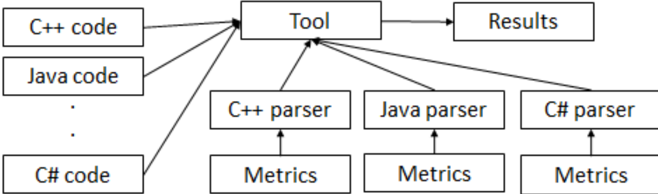

Figure 2.2 - Common architecture of a software metrics tool

A system architecture is defined by the ISO/IEC 24765 as a “fundamental organization of a system embodied in its components, their relationships to each other, and to the

environment, and the principles guiding its design and evolution". That organization and relationships are depicted in Figure 2.2. In the left side, the programming languages it supports are depicted. In this common architecture, for each language a dedicated parser has to be created and managed by the tool, so the incorporation of new languages is not an easy task. For each parser a set metrics, that may or may not differ across languages, is supported by each parser independently. This tool architecture becomes more complicated according to the number of languages and metrics it can support, also limiting the functionality and scalability of the tool.

Research has been made in order to simplify the creation of metrics tools, especially by using source code intermediate representations, models or query languages, but still several issues are found with the current extraction methodologies and available tools. The issues found in current metrics extraction methodologies are presented below.

2.2.1 Issues found in current extraction methodologies

Source code metrics extraction has been widely studied over the years, but still many issues arise. According to our research, one of the first studies reporting inconsistences across metrics tools is the study by Lincke et al., 2008. The results of that study indicate that different tools provide different values for the same metric, but it is not the only issue found, Raki et al., 2010 identifies several issues with metrics tools such as the dependency of the set of metrics and programming languages the tools accept, and recent studies (Mshelia et al., 2017) still found the same issues with recent tools. According to the research carried out in this thesis, and also according to the source code metrics related

studies we have conducted (Nunez-Varela et al., 2016, Nuñez-Varela et al., 2017), the issues found regarding software metrics tools and extraction methodologies are presented in Table 2.1.

Table 2.1. Issues found with current software metrics extraction mechanisms

ID	Issue
I1	Dependent of the set of metrics they accept
I2	Dependent of the set of languages they accept
I3	Dependent of the set of programming paradigms they accept
I4	Cannot be extended to accept new metrics or languages
I5	Updates of the tool must be made according to languages or metrics changes
I6	Tools are not updated or supported anymore
I7	Different and inconsistent results across tools
I8	The use of multiple tools for a single project
I9	Necessity to develop their own tools
I10	Lack of information
I11	Metrics miscalculations
I12	Use of third party components

Issues I1 to I4 in Table 2.1 have been reported by Cogan and Shalfeeva, 2002, Darcy and Kemerer, 2005, Alikacem and Sahraoui, 2009, Raki et al., 2010, Budimac et al., 2012, Núñez-Varela et al., 2016 and Mshelia et al., 2017. They limit the operation of the extraction methodology by allowing only the extraction of the predefined set of metrics from the source code written in the programming language it was designed for. It is important to provide mechanisms to define new metrics since hundreds of source code metrics have been defined throughout the years (Nuñez-Varela et al., 2017), and new metrics are being proposed by researchers (Mo et al., 2016, Moshtari and Sami, 2016) for different programing paradigms. Also, new programming languages, modifications to existing ones or dialects can be proposed, and even new programming paradigms (Schaefer et al., 2010). This makes it difficult to keep up to date the current metrics extraction tools capabilities.

Issue I5 in Table 2.1 is a direct consequence of issues I1 to I4 and has been reported by (Scotto et al., 2004; Sillitti et al., 2006). Issue I6 in Table 2.1 is a mayor issue regarding tools updates, as it can bee seen in Table 1.1 most of the tools do not have recent versions, and tools like AOPMetrics, which is the most common for aspect oriented metrics extraction, is no longer supported.

Issue I7 in Table 2.1 is arguably the most relevant problem to consider. It has been reported by Fioravanti and Nesi, 2000, Emanuelsson and Nilsson, 2008, Alikacem and Sahraoui, 2009, Budimac et al., 2012 and Núñez-Varela et al., 2016, and studied by Lincke et al., 2008, Novak, 2011 and Srivastava, 2014. The authors from those studies reach to the conclusion that the extracted numerical metrics values are indeed different according to the tool. Since there is not an established standard for metrics definition, and authors proposing new metrics usually provide informal definitions, that definition and its related computation can vary according to the understanding of each person, even for common metrics such as LOC or the CK metrics. Also, some metrics have multiple versions (i.e. coupling between objects and lack of cohesion), and furthermore, the result can be affected by an aggregation method (sum, percentage, etc.), or by the encapsulating scope (class, method, etc.).

Issues I8 and I9 in Table 2.1 are given as a consequence of the above issues. These issues are supported by our previous study (Nuñez-Varela et al., 2017), where it was found that from 114 studies using software metrics tools, 33 studies indicated that a tool was specifically developed, and 13 studies used multiple tools to achieve their goals, representing the 40% of the total. This represents a major issue since the development of

a metrics extraction tool is complex, but researchers do not find the current tools or methodologies suitable.

Issue I10 in Table 2.1 is present in tools and extraction methodologies using intermediate source code representations. From these representations, the metrics are defined and extracted, but using intermediate representations add an extra layer of complexity that can affect the performance and maintenance of the method, and furthermore, an extra tool might be necessary to create them. Structured models conformed by predefined entities and relations are the most common choice of intermediate representations, but the metrics that can be extracted are limited by the completeness of the model and the data it contains. Alikacem and Sahraoui, 2009, acknowledge using an incomplete model for their validations because the tool that creates such model did not extract all the necessary attributes. UML models are also used, but they might not reflect specific characteristics from a language, and might be only useful for the object oriented paradigm.

Issue I11 in Table 2.1 was found as part of this thesis. When analyzing the metrics results and definitions from the selected tools from which our results are compared to, it was found that for some code files the metrics results were miscalculated. Even the authors of the CKJM tool acknowledge that results might not correspond to the Java source code since they measure the bytecode.

Issue I12 in Table 2.1 is present in current tools that depend on third party components in order to work as expected. For example, many tools, such as Metrics 1.3.6, run as a plugin

for a certain IDE, making them totally dependent of the characteristics provided by that third party component. Also, in the described methodologies for metrics extraction in Chapter 1, the mechanisms used to consult the model are also dependent of third party components. Most of the works use a database as a requisite to manipulate the information, with SQL being used for that manipulation, thus having the necessity of a database engine to fulfill the objectives. OCL and XQuery are also used, which makes the methodology dependent on an engine, library, runtime, or application that supports those languages. Eichberg et al., 2006, acknowledge that half of the queries they ran with XQuery for certain tests had unacceptable runtime performance, and could be only used for small projects because of the engine they used. The integration of third party components adds an extra layer of complexity to the extraction process.

Finally, it is important to note that most of the tools and methodologies discussed in Chapter 1 do not present details on how the intermediate representation was created, or what data it contains, leaving to speculation what metrics can or cannot be defined. Also, none provides real life validations of their methodologies by extracting metrics from large or benchmark systems commonly used in research papers. Only one work (Alikacem and Sahraoui, 2009) presents information about their validation, but the systems measured are "developed internally" and no further information on the results is presented.

The methodologies for aiding the metrics extraction presented in this thesis will help solve issues I1 to I4 since the user can incorporate new metrics and languages, but given the relationships among the issues, I5 and I6 are also solved since the user does not need to wait for updates for new metrics or languages incorporation. Issues I7 to I9 are partly

solved since the user has control of what he is trying to measure, and a tool constructed over this methodology provides all the necessary mechanisms to not require other tools.

The field of software metrics is constantly changing, researchers propose new metrics and modifications to existing ones, and new programming languages or paradigms can suddenly gain major interest. It is a real challenge to keep up with the research advances, thus, generic mechanisms to aid the metric extraction process are necessary.

2.3 Grammatical inference

Grammatical inference is the process of automatically finding a grammar that accepts a given unknown language. It is a complex problem, and given that it is a combinatory problem it could be classified as an NP problem (Stevenson and Cordy, 2014):

- A non-deterministic polynomial time (NP) problem is that from whose solutions can be verified in polynomial time, but takes exponential time to solve (Hardesty, 2009).

- In turn a polynomial time problem (P) is that problem that can be efficiently solved by any current computer (Hardesty, 2009).

In our case, the solutions of a context-free grammar, that is, the inputs it accepts, can be verified very fast in polynomial time through the use of compilers (Aho et al., 2007), but the inferring of the context-free grammar takes exponential time to solve.

In order to provide a clear example of a P-NP problem an analogy with a labyrinth is presented:

- A labyrinth, with one entry, can have many paths, but all of them lead to the same exit (P).

- A labyrinth, with one entry, can have many paths, but each path can lead to a different exit, furthermore, there can be dead ends. If a dead end is reached, one must backtrack and go through all the possible paths until an exit is reached. Furthermore, one or more exits can be valid (NP).

For context-free grammars a language can be defined with any number of grammars and a grammar accepts an infinite number of strings, so it cannot be directly tested or proven that a given inferred grammar is a correct one and useful for all the language strings, also, an inferred grammar can be correct but not useful.

The resolution of NP problems can be approached in several ways. One of the most common, but time consuming, is the brute force approach. For grammatical inference, brute force approaches has been proposed using incomplete grammars (Crepinsek et al., 2005). Approximation and domain algorithms are more common and useful for the resolution of NP problems.

For this thesis an approximation approach is not viable since the grammar must always accepts the inputs, but a common solution for a specific domain will be useful since we are only interested in a specific domain: source code structures of object oriented Java-type languages relevant to metrics extraction, including classes, member variables and methods declarations.

To the best of our knowledge, no papers have been published with a methodology able to infer a full (or incomplete) context-free grammar for a common programming language, it is still an unresolved problem.

2.4 Chapter summary

In this Chapter general information about source code metrics, source code metrics extraction and grammatical inference is presented. This information is complemented with an example of some common code metrics definitions and their extracted values.

The most common and basic architecture of software metrics tools is depicted in order to show the complexity of such a tool, and the number of elements it must contain. This complexity has led to the study of several issues with current code metrics extraction tools. Twelve extraction issues were discussed with the most important being: dependent of the set of metrics, programming languages and programming paradigms they accept, they cannot be extended to accept new metrics or languages, the necessity of third party components, and metrics values miscalculations. Some issues have been already studied by researchers, but new issues were found during the development of this thesis, such as the necessity to develop their own tools and metrics values miscalculations.

This Chapter ends with general information about grammatical inference, and since it could be classified as a NP problem, a description of the P-NP problem is presented along with an analogy for better understanding.

Int the following Chapter, the proposed methodologies for aiding the source code metrics extraction are presented.

Chapter 3

3. Extraction methodology through the creation of dynamic data models and grammatical inference

In this Chapter the main contribution of this thesis is introduced represented by the two methodologies for aiding the source code metrics extraction. The first methodology consists on creating dynamic data models as source code representations by querying the context-free grammar of the language, from which metrics can be computed. The extraction is further aided by the second methodology which automatically infers the context-free grammar for relevant object oriented source code structures from which code metrics are defined. These structures are: class definitions with inheritance, member variables definitions and methods definitions. Finally, a language named Metrics Query Language (MQL) is presented which formalizes the first methodology, and is the base of the MQLMetrics tool.

3.1 Methodologies for code metrics extraction and grammatical inference

In this Chapter the two methodologies are presented for aiding the source code metrics extraction. The first methodology, described in Chapter 3.4, consists on creating an intermediate representation of the source code in the form of a data model. This model is represented as a tree structure given the hierarchical structure of the object oriented source code. Models are commonly using in the metrics extraction literature as intermediate representations because one model can represent any number of programming languages, so by manipulating the model several languages can be supported at the same time. Even though, the model for each language needs to be created with a specialized parser per language and are static, so the definition of new metrics are bounded to the data in the model and for defining a new language a new model has to be incorporated. The proposed methodology aims to solve those issues by creating dynamic data models, that is, the model is created according to the metric to extract and because of this, the necessary information to extract that metric is always present. These dynamic data models are created querying the context-free grammar of the language, this avoid the need of using specialized parsers and the incorporation of new languages by providing the context-free grammar of the language.

The second methodology aims to further aid the code metrics extraction process by inferring the context-free grammar of the programming language by a process known as grammatical inference, as described in Chapter 3.5. This is important given that context-free grammars are hard to write, can be very extensive and sometimes are not available. Although grammatical inference could solve the described issues, it still and unresolved

problem, especially for context-free grammars. It is unresolved given its complexity for being a combinatory problem, in which any number of grammars can define any number of languages, and a grammar can accept an infinite number of valid strings. It is common to solve these types of problems by limiting the domain of the problem and creating rules that facilitate the process. In the proposed methodology the domain is limited to infer object-oriented structures relevant for metrics extractions for Java-like languages. Rules are also established for helping the inference process. These rules are patterns found in the source code that will ultimately be transformed in the production of the inferred grammar.

The following Chapter discusses the common architecture of current metrics tools in order to provide a general idea on where the above presented methodologies are needed.

3.2 Current tools architecture

These methodologies aim to simplify the overall architecture of a software metrics tool. In the simplest form, a metrics tool presents the architecture as depicted in Figure 2.2, where each language has an independent parser with the metrics being extracted from that parser.

Since it is a complex and limiting architecture, better extraction methodologies tend to use source code intermediate representations with the idea of providing a common base for the metrics extraction, avoiding the need to modify each parser if a metric is modified or added (Higo et al., 2011, Budimac et al., 2012, Rakic et al., 2017) as depicted in Figure 3.1.

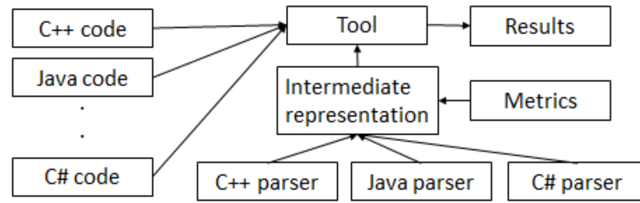

Figure 3.1 - Metrics extraction architecture using source code's intermediate representation

Still, using an intermediate representation presents many issues as discussed in Chapter 2.2.1. The first methodology presented in this Chapter simplifies the above architecture by querying the language grammars directly. From these queries the metrics are defined and processed. Although a model is created as an intermediate representation, it is dynamically created and disposed according to the query. Figure 3.2 depicts the architecture.

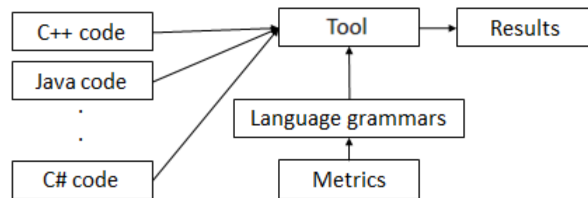

Figure 3.2 - Simplified architecture using dynamic models

Finally, an optimal architecture is presented in Figure 3.3, in which the languages grammars are not needed as an input to the tool since they are inferred from the source code. In this thesis we present an approximation to this type of architecture by providing a grammatical inference process capable of inferring the grammar for certain source code structures.

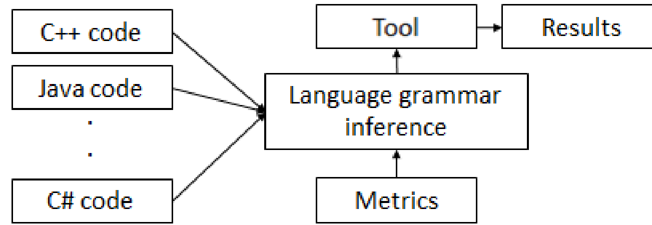

Figure 3.3 - Proposed optimal architecture

The optimal architecture can be reached by using the two methodologies presented in this Chapter:

- A methodology that allows the extraction of relevant substrings from the source code, useful for code metrics computation, arranged in a hierarchical data model. A tree structure will be used as the data model. The extraction is achieved by querying the non-terminal symbols found in the body of the productions of a programming language context-free grammar.
- A methodology for grammatical inference for certain code structures from which metrics can be queried.

In the following Chapter the used terminology is presented.

3.3 Terminology

The terms used throughout this thesis are defined as follows.

A language context-free grammar is defined as the tuple $[N, T, P, S]$ where N is the set of non-terminals, T is the set of terminals, P is a set of productions and S is the start symbol of the grammar (Aho et al., 2007).

In parsing time, a derivation is defined as a non-terminals sequence of replacements obtained by rewriting a non-terminal by the body of one of its productions, beginning from S, until a specific non-terminal (N_i) in the body of a production is reached (derivation of N_i from S). A derivation will be represented as $S \Rightarrow N_i$ or $S \rightarrow N_0 \rightarrow N_1 \ldots \rightarrow N_i$, where $\Rightarrow$ means derives in one or more steps and $\rightarrow$ derives in one step. For this thesis, we are mainly interested in the derived non-terminal symbol N_i, and since it can be reached beginning from any arbitrary non-terminal symbol X, it will be represented simply as $\Rightarrow N$ (meaning $X \Rightarrow N_i$ or $X \rightarrow N_0 \rightarrow N_1 \ldots \rightarrow N_i$).

Whenever a reduction step is performed, a substring from the input matches the body of a production; this substring is called a handle (Aho et al., 2007), which will be linked and saved as the result of the current derivation. These substrings will represent the data from which metrics are going to be computed. For this thesis a handle will be expanded and defined as the tuple [STR, I, E], where STR is the matched substring from the input, and I and E are the start and end positions (indices in the input treated as an array of characters) of the matched substring.

3.4 Source code metrics extraction methodology

A large number of source code metrics have been defined and studied, especially for the object oriented, aspect oriented and feature oriented programming paradigms (Nuñez-Varela et al., 2017). According to those metrics definitions and source code structure, we define a source code metric as the set of three components: 1) the scope which represents an enclosing context for the metric value (i.e. file, class, method), 2) the

computation method which defines how the metric's data is processed, and 3) the attribute to measure which is the metric value and is defined as "a measurable physical or abstract property or an entity" (ISO/IEC 24765).

The source code of a program contains many attributes of interest, and source code metrics are used to measure those attributes.

To illustrate the components of a code metric, the following example is presented. We want to extract the number of variables declared in an object oriented source code; the attribute, then, is the number of variables identifiers defined in the code. Variables can be declared in different parts of the code, for example, at *file*, *class* or *method* level in the object oriented paradigm, thus, we need to define the scope from which the attribute, the variables, must be extracted. Finally, since we are interested in the number of variables, the method of computation is a simple count function. The result of this metric is a single number indicating the number of variables inside the selected scope. A metric that only depends on a single set of components to be computed will be called a Base Metric (BM).

A single set of components do not always provide enough information to compute a metric, most of the current metrics are complex and require data from several code attributes. These metrics will be defined as a Composite Metric (CM), which are defined as the union of two or more BMs. For example, the metric *Weighted Methods per Class* (WMC) presents a more complex approach. This metric sums the cyclomatic complexities, which is a metric by itself, of each method in a class, and that sum is presented as the

weight of the class. The decomposition as BMs for this code metric is presented in Table 3.1.

Table 3.1. BMs for WMC

BM	Scope	Attribute	Computation method
1	Method	Control statements	Count (plus one)
2	Class	Class identifiers	Sum all values in BM1 for all methods in a class, and assign the resulting value to each corresponding class identifier

From the three components that are part of the code metric definition, the scope and attribute components can be extracted from the source code and are going to be represented by a derivation. The computation method is independent and can be achieved by post processing the results. The necessary steps to define and perform the extraction are presented below.

3.4.1 Creation of dynamic data models from source code

The proposed methodology works by querying the non-terminal symbols found in the body of the productions of a programming language grammar. The substrings that are reduced by the queried non-terminal are the result of the query, and are useful for metrics calculations. A query will be defined as a BM with the scope and attribute components. Each component will be represented as a derivation. The result of those derivations (a set of handles) will provide the information to compute the metric. The overall process is as follows:

- In parsing time, all handles tuples are created and saved when the derivations are occurring. This process depends on the type of parser or tool being used and will not be covered in this thesis.

- The scope and attribute derivations that represent a given query will be matched against the produced derivations in the step above, and the linked handle tuples are saved for both derivations.
- From the tuples obtained in the last step, a tree is constructed that will organize the strings in a hierarchical way in order to present the final result.

To illustrate the above steps, we define in Table 3.2 a language grammar that accepts the declaration of a set of classes with member variables.

Table 3.2. Context-free grammar example

<table>
<tr><th colspan="3">Grammar</th></tr>
<tr><td>language</td><td>→</td><td>entry</td></tr>
<tr><td>entry</td><td>→</td><td>classDec entry | classDec</td></tr>
<tr><td>classDec</td><td>→</td><td>modifiers ‘class’ id ‘{‘ classBody ‘}’</td></tr>
<tr><td>classBody</td><td>→</td><td>statement classBody | statement | nothing</td></tr>
<tr><td>statement</td><td>→</td><td>attributes | classDec</td></tr>
<tr><td>attributes</td><td>→</td><td>attributeDec attributes | attributeDec</td></tr>
<tr><td>attributeDec</td><td>→</td><td>modifiers ‘var’ id</td></tr>
<tr><td>modifiers</td><td>→</td><td>modifier modifiers | modifier</td></tr>
<tr><td>modifier</td><td>→</td><td>‘public’ | ‘private’</td></tr>
<tr><td>id</td><td>→</td><td>[a-z]+</td></tr>
</table>

Given the input string “`public class Class1 { private var a } public class Class2 { private var b }`”, with character positions 0 to 67, Table 3.3 presents a set of derivations with their corresponding results that can be obtained in parsing time.

Table 3.3. Derivations examples

	Derivation	Result set [S, E, STR]
1	*=>entry*	[0, 68, public class Class1 { private var a } public class Class2 { private var b }]
2	*entry→classDec*	[0, 34, public class Class1 { private var a }], [35, 68, public class Class2 { private var b }]
3	*entry→classDec→id* or *=>classDec→id*	[14, 19, Class1], [48, 53, Class2]
4	*entry→classDec→classBody→statement→ attributes→attributesDec→id* or *=>attributeDec→id*	[33, 33, a], [67, 67, b]
5	*=>id*	[14, 19, Class1], [33, 33, a], [48, 53, Class1], [67, 67, b]
6	*entry→classDec→classBody→statement→ attributes→attributesDec* or *=>attributes* or *=>attributesDec*	[20, 32, private var a], [54, 66, private var b]

Since we query through derivations, all the parsed source code attributes can be accessed, but special attention must be made in the symbol the derivation points to (the symbol to query), since undesired results can be obtained because that symbol can be part of several production bodies. For example, in derivation 5 all *id's* are queried, if only a set of determined *id's* are wanted, as in derivations 3 and 4, the correct derivation must be provided. Since a metric value might be textual, numerical or categorical, it is important to keep all the data available, including the number of items in the results set.

The data obtained from the results sets is then organized in order to reach correct results, and to correctly apply the scope. This will be achieved by replicating the nest hierarchy of programming structures (i.e. methods are written inside classes, loops or conditionals can be nested within each other) in a data model using a tree structure. The tree will be created in two steps according to the position each handle tuple takes in the input: with

the handles data obtained by the scope derivation the tree nodes are created, and with the handles data obtained from the attribute derivation the tree is populated.

For the first step, a node will be created for each tuple handle in the result set and inserted in a top-down insertion as a child of the deepest node in which the following condition is met: the start and end positions of the node to insert are within the start and end positions of the current visited node. The root node of the tree is represented by the handle tuple with the start and end positions of the whole input.

For the second step, the nodes are labeled with the substrings contained in the handles of the attribute derivation following a similar top-down insertion process. Figure 3.4 shows a code snippet for the insertion process.

```
boolean InsertNode(handle1, handle2)
    if (handle2.Start >= handle1.Start && handle2.End <= handle1.End)
        for each child in handle1
            if (InsertNode(child, handle2) == true) return true
        Insert handle2 as child of handle1
        return true
    return false
```

Figure 3.4 - Top-down insertion

To exemplify the tree creation, two queries are presented in Table 3.4, and in Figures 3.5 and 3.6 the tree construction is depicted.

Table 3.4. Example queries

Query 1 (depicted in Figure 3.5)		
Component	**Value**	**Derivation path**
Scope	Declaration of the class	*=>classDec*
Attribute	Class name	*=>classDec→id*
Query 2 (depicted in Figure 3.6)		
Component	**Value**	**Derivation path**
Scope	Declaration of the class	*=>classDec*
Attribute	Class modifier	*=>classDec→modifiers→modifier*

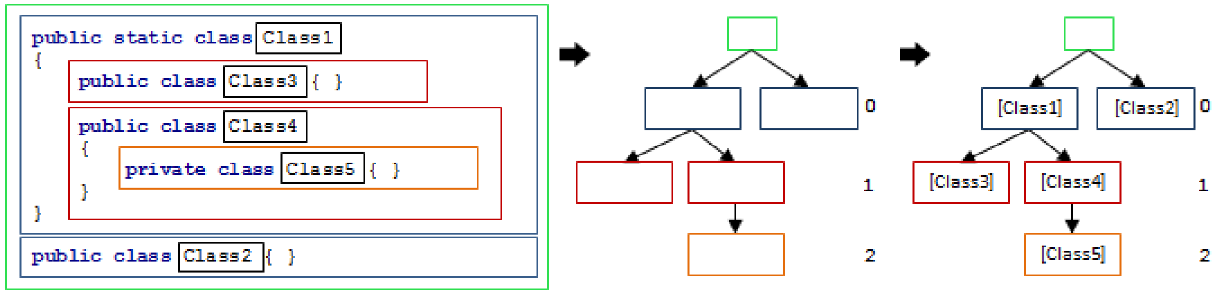

Figure 3.5 - Tree construction for Query 1

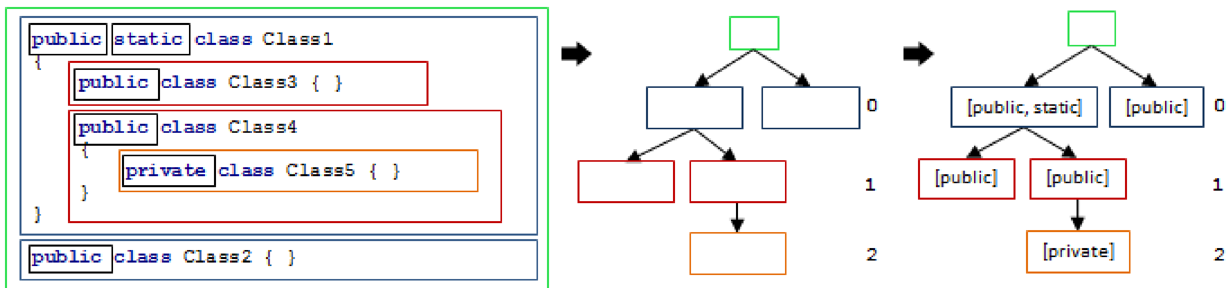

Figure 3.6 - Tree construction for Query 2

The non-terminal to query has a major impact on the result. In Figure 3.6 a node was populated with the set of handles [`public, static`] (two elements) with the derivation path *=>classDec→modifiers→modifier*. If the derivation path changes to *=>classDec→modifiers*, the node will be populated with the set [`public static`] (one element), since the handles reduced from each production are different. It is also important to note that the depth of each node is saved since it is useful for the calculation of nesting metrics. The depth is depicted in Figures 3.5 and 3.6 as the number next to the nodes. An example of metrics extraction using the depth value is presented in Chapter 4.1.2.

The methodology, as presented to this point, is applied to BMs, which might not provide enough data to compute more complex metrics. A CM was defined as the union of two or

more BMs. To query a CM, it must be decomposed into BMs and query them independently. The result of a CM is a single merged tree containing all the trees generated from the BMs. It is merged as follows:

For a CM consisting on *n* ordered BMs (BM_1, BM_2… BM_n), each BM will be queried and its result tree calculated. Each one of the *n* trees will be decomposed into a simple ordered node list (L_1, L_2… L_n), ordered according to the start position each handle takes in the input, and a new tree will be created from these lists. Applying the same algorithm in Figure 3.4, starting from L_n, all nodes in each list will be inserted as child nodes into the contiguous list to its left (L_n into L_{n-1} to L_2 into L_1).

To exemplify the process above, using the grammar in Table 3.2 and the input presented in Figure 3.7, we want to extract the *number of attributes per class with their associated modifiers* (NOAAM). Table 3.5 defines this metric as three BMs.

```
public static class Class1
{
    public static var v1c1
    public class Class3 { public var v1c3 }
}
private class Class2 { private var v1c2 }
```

Figure 3.7 - Input to calculate the metric NOAAM

Table 3.5. BMs to calculate the NOAAM metric

BM_1	Component	Value	Derivation path
	Scope	Declaration of the class	*=>classDec*
	Attribute	Class identifier	*=>classDec→id*
BM_2	**Component**	**Value**	**Derivation path**
	Scope	Member attributes declaration	*=>attributeDec*
	Attribute	Member attributes modifiers	*=>attributeDec→modifiers→modifier*
BM_3	**Component**	**Value**	**Derivation path**
	Scope	Member attributes declaration	*=>attributeDec*
	Attribute	Member attributes identifier	*=>attributeDec→id*

Figure 3.8 presents the respective result tree for each BM defined in Table 3.5. Using the trees in Figure 3.8, the merging process is depicted in Figure 3.9, and the final merged tree is presented in Figure 3.10. The depth of each node is preserved in the merge process (presented as the number after the comma in each node).

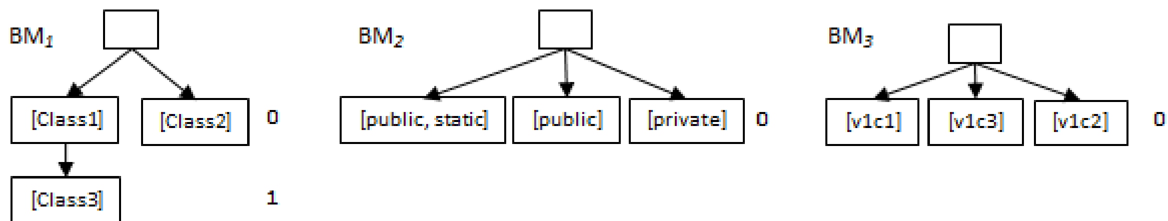

Figure 3.8 - Result trees for BM*1*, BM*2* and BM*3*

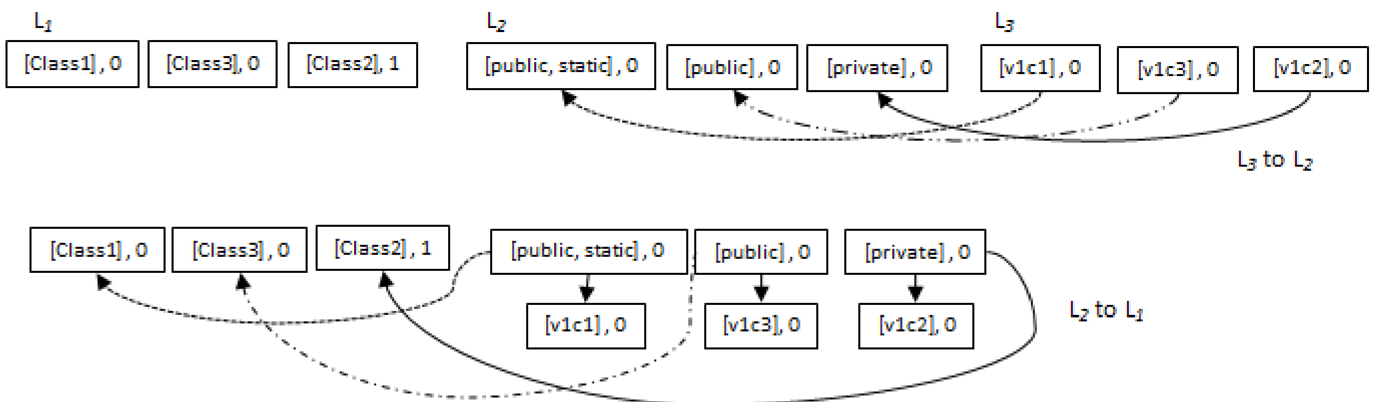

Figure 3.9 - Merging process

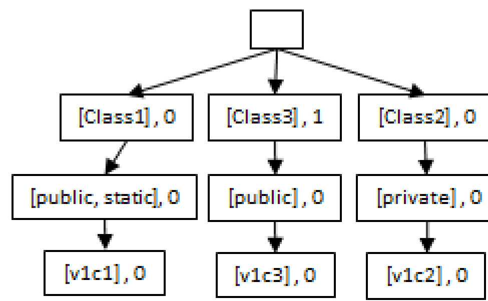

Figure 3.10 - Final merged tree

The merged tree is presented as the result of a CM. The tree can represent the metric value by itself, or the data can be post-processed in order to obtain the desired value. In the next Chapter examples and a case study are presented in order to further exemplify the methodology.

3.4.2 Context-free grammars special considerations

There is a case of ambiguity in which several instances of a queried non-terminal can be found in the body of a production. For a production such as:

$$S \rightarrow \alpha\ E_1\ \beta\ E_2\ \ldots\ E_n\ \delta\ |\ \alpha\ E_m\ \beta$$

The non-terminal *E* is found several times in the body of the production. A derivation path such as *=>S→E*, do not specify which *E* symbol is being queried, so undesired results can be presented because the data from all the *E* non-terminals is extracted. This can be easily fixed by introducing a new production and make it produce the desired *E* symbol:

$$S \rightarrow \alpha\ E_1\ \beta\ A\ \ldots\ E_n\ \delta\ |\ \alpha\ E_m\ \beta$$

$$A \rightarrow E$$

Now, the derivation path *=>A→E* (or *S→A→E*) will point to the correct symbol. It is a common case presented in language grammars, for example when declaring a class definition with inheritance:

```
classp → 'class' id 'inherits' id
```

It is important to distinguish between the two `id` symbols, since the first `id` identifies the class identifier and the other the base class identifier, the grammar can be rewritten as:

```
classp → 'class' id 'inherits' id
base   → id
```

Where *=>classp→id* and *=>base→id* point to the correct symbol. For a production such as:

$S \rightarrow E_1\ \beta\ E_2\ \ldots\ E_n\ \delta\ |\ E_m\ \beta$

Trying to change to:

$S \rightarrow A\ \beta\ E_2\ \ldots\ E_n\ \delta\ |\ E_m\ \beta$

$A \rightarrow E$

will incur in left recursion, which some parsers cannot handle. The grammar can be rewritten to solve the issue as:

$S \rightarrow E_1\ A\ \beta\ E_2\ \ldots\ E_n\ \delta\ |\ E_m\ \beta$

$A \rightarrow \varepsilon$

A rule is established that, whenever a non-terminal in the body of a production is an empty production, the production values from its left side symbol are copied to it. Now

the derivation path $S \rightarrow A$, will point to E_1. This mechanism can be also useful for creating aliases for non-terminals in general.

3.4.3 Metric Query Language

In order to formalize the above methodology, a declarative language named Metric Query Language (MQL) is proposed. The grammar of the language is presented in Figure 3.11.

```
query           →  path
query           →  path  '&'  query
path            →  addPath  'in'  addPath  excludePath
excludePath     →  'exclude'  addPath
excludePath     →  ε
addPath         →  derivationPath
addPath         →  derivationPath  '+'  addPath
derivationPath  →  relation  id
derivationPath  →  relation  id  derivationPath
relation        →  '.'
relation        →  '!'
id              →  [a-zA-Z]+
```

Figure 3.11 - MQL grammar

The language contains two reserved words: *in* and *exclude*, and the following reserved symbols '&' (*ampersand*), '+' (*plus*), '.' (*dot*) and '!' (*exclamation*).

It is a very simple language that allows us to create queries based on the methodology described. Derivation paths are written as defined in Chapter 3.3, but the symbols → and => are replaced with the *exclamation* and *dot* symbols respectively for simplicity. The derivation path can contain derive in two or more steps relations in order to facilitate the writing.

A query is defined by two derivations (attribute and scope components), the reserved word *in* will be used to link both paths into a single query in the form of: *attribute* ***in***

scope. For example, the Query 1 in Table 3.4 is written in MQL as $\Rightarrow classDec \rightarrow id$ $\boldsymbol{in}$ $\Rightarrow classDec$. The keyword $exclude$ is a feature introduced by the language, it allows the inclusion of a third derivation path that defines exclusion scopes in the form of: *attribute* ***in*** *scope* ***exclude*** scope. Any handle that is within those scopes is automatically rejected when populating the result tree. Another feature of the language is the ability to conjunct derivation paths that define the same metric element with the symbol *plus*. This is useful for extracting different data from a single query.

Finally, a CM is created from a set of BMs. The symbol *ampersand* allows the conjunction of BMs into a CM in the form of: BM_1 & BM_2 & … & BM_n.

3.5 Grammatical inference methodology

Grammatical inference is the process of automatically finding a grammar that accepts a given unknown language (Stevenson and Cordy, 2014). For this thesis, we are interested in finding a context-free grammar from the provided source code. To the best of our knowledge, no papers have been published with a methodology able to infer a full context-free grammar for a common programming language. Even though, attempts have been made by trying to infer only missing productions, or by helping the inference process by providing initial settings, using specific heuristics for a certain problem, which might or might not be useful to solve other similar problems, or even by using a person, called an oracle, which supervises the inference process and accepts or rejects the proposed

productions that are being generated. A list of methods can be found in (Stevenson and Cordy, 2014).

Grammatical inference is a complicated process since a language can be defined by any number of grammars and one or more solutions (grammars) can be obtained, but not all grammars will suit the desired results. For example, from the code:

```
public void method1() { }
public void method2() { a = 3 }
```

the following grammar can be inferred:

```
P  → 'public' 'void' id '(' ')' '{' P1
P1 → '}' | id '=' NUM '}'
```

Although the grammar is correct, it might not be optimal for certain domains. A more accurate grammar will include the '}' symbol as part of the production P given that is part of the construction it defines:

```
P  → 'public' 'void' id '(' ')' '{' P1 '}'
P1 → ε | id '=' NUM
```

Furthermore, a grammar can accept an infinite number of strings, so proving that for any given grammar all valid strings are accepted by that grammar might not possible. This also affects the fact that an input not always contains all the structures the language can accept, so we are working with incomplete data. For example, from the following code:

```
private int method1()
private int method2(int a)
```

it cannot be inferred that multiple parameters can be accepted or that the `private` reserved word is optional.

The proposed methodology automatically infers a context-free grammar from an input string written in any programming language with a Java like syntax, since it was designed for the object oriented paradigm with that type of languages in particular, for metrics related structures including classes, member variables and methods definitions. As discussed earlier, grammatical inference methodologies are usually based on certain specific domain rules or heuristics that help to reduce the complexity of the problem. For this methodology, the lexical definition of the language should be provided, along with syntax rules in the form of binary operator's precedence, in order to correctly create the context free grammar productions for our domain. Based on the provided information, a generic parser analyzes each source code instruction and creates the final grammar.

3.5.1 Patterns

The parser is a stack based algorithm that creates expressions every time a language instruction is parsed. From these expressions, the grammar productions are created, in fact, an expression can be defined as a temporary inferred complete or incomplete production that can be part of the final grammar. These expressions are created based on source code pattern detection, and will represent the actual patterns. Even though the methodology tries to automatically find the patterns in parsing time, an original set of patterns are defined by the operators' precedence. The operators' precedence is important to our domain because of the interest in specific parts of the source code that

are relevant for metrics extraction, also because finding the binary operators' patterns automatically can be inaccurate. For example, from the following instruction:

```
if ( a + b == 5 || c.d == 8 )
```

several patterns can be found, including `a + b`, `b == 5` and `5 || c`, from which only the first one is correct for our domain, and the others, even if they can produce a correct grammar, are not useful for our domain. For example a grammar can be inferred as:

```
S0 → 'if' '(' S1 ')'
S1 → S1 S3 S2 | S2
S2 → id | num
S3 → '+' | '.' | '==' | '||' | '.'
```

which is syntactically correct, but not semantically, mainly because the symbols belong to different categories, i.e. mathematical, boolean, etc. For example, `c.d` is a qualified name that can have a meaning by itself, and is not identified in the grammar.

Two types of patterns are going to be identified: α for $|\alpha|>0$, and $\alpha\ \gamma\ \beta$, where α and β are any sequences of symbols or patterns, and γ a set of operators. With these types of patterns a grammar for the above example can be inferred as:

```
S0 → 'if' '(' S1 ')'
S1 → S1 '.' S2 | S2
S2 → S2 '+' S3 | S3
S3 → S3 '==' S4 | S4
S4 → S4 '||' S5 | S5
S5 → id | num
```

The above grammar is syntactical and semantically correct, and also provides enough information to query the different relevant parts. The definition of binary operators and

their correspondent patterns also allows the creation of productions for common strings in programming languages such as var_1, var_2 … var_n, by recursively replacing the patterns as shown in Table 3.6:

Table 3.6. Patterns for recursive structures

Input	Pattern	Pattern Id
var_1	id	p1
var_1, var_2	p1 ‘,’ p1	p2
var_1, var_2, var_3	p2 ‘,’ id	p3
var_1, var_2 … var_n	No new patterns, all are p3 ‘,’ id	-

The proposed methodology finds and matches patterns with each token read and parsed from the input, but the matching is not always accurate given the lack of information and order of the instructions. For example, for the following code:

```
public int a;
public int b,c;
```

a pattern p1 defined as {modifier type id} is found given that it is the first instruction and there are no other patterns to compare to, so the whole instruction becomes the pattern. When matching the second instruction, a new pattern can be found as {p1 ‘,’ id}, which is incorrect. If the instructions are parsed in different order (the second first), a pattern *p1* is found as {id ‘,’ id}, pattern p2 as {‘modifier’ ‘type’ p2}, and pattern p3 as {modifier type id}, resulting in a correct pattern matching. This behavior also affects the overall inference process when trying to create the final inferred grammar, as presented in the Tables 3.7 and 3.8:

Table 3.7. Creation of an inferred grammar for similar instructions. Example 1

Order	Input	Grammar	Notes
1	`public int a`	`S0 → 'public' 'int' S1` `S1 → id`	Creates Grammar 1

2	`public int a, b`	`S0 → 'public' 'int' S1`
		`S1 → S2 ',' S2`
		`S2 → id`
			Grammar 1 is not useful, a new set of productions are created as Grammar 2

3	`private int a, b, c`	`S0 → 'private' 'int' S1`
		`S1 → S1 ',' S2 | S2`
		`S2 → id`
			Grammars 1 and 2 are not useful, a new set of productions are created as Grammar 3

Table 3.8. Creation of an inferred grammar for similar instructions. Example 2

Order	Input	Grammar	Notes

1	`public int a, b, c`	`S0 → 'public' 'int' S1`
		`S1 → S1 ',' S2 | S2`
		`S2 → id`
			Creates Grammar 1

2	`public int a, b`		Can be parsed with Grammar 1
3	`private int a`	`S0 → 'private' 'int' S1` `S1 → id`	Grammar 1 is not useful, a new set of productions are created as Grammar 2

Because of the issues discussed above, the following is considered:

- Independently on the order in which the instructions are parsed, the patterns are first matched against the patterns found in the instruction currently being parsed, if no matching pattern is found, all other patterns are used.
- A pattern starts or ends when a symbol (i.e. operators, commas, etc.) is found. This is because the symbols are part of the syntax rules, which are patterns by themselves.

Even if the above rules are followed, the methodology does not ensure the accuracy of the pattern matching. The accuracy is heavily influenced by the available information and instructions already parsed.

3.5.2 Lexical definition

The lexical definition of a programming language is usually defined as part of the context-free grammar, which defines the language syntax. Since not much work has been made regarding grammatical inference for programming languages context-free grammars, it is unclear if the lexical definition can be inferred as a separate process or in conjunction with the syntax definition. In general, the lexical inference of a programming language would consist on extracting the following components: regular expressions, reserved words, and symbols. According to our research, and to the best of our knowledge, lexical inference is not currently being applied to programming languages as a separate process, although regular expression inference has been studied (Bartoli et al., 2016, Cetinkaya, 2007).

In this thesis, no lexical inference process is attempted, the components of the lexical definition are provided as an input, but the final grammar will contain only the lexical components found in the provided source code. Also, since we are dealing with an inference process, it is unknown the meaning a token can have according to its position. The most representative example of this issue is when an identifier is used as a type, for example, the names of the classes are used as types. Because of this issue, the inference process will allow the tokens to have multiple classifications. Four types of classifications are defined: reserved words, regular expressions, symbols and groups. The regular expressions define the patterns read from the input, while the groups define common meanings for a set of regular expressions or reserved words. Table 3.9 presents an example of a lexical definition:

Table 3.9. Lexical definition

Classification	Tokens
Reserved	int, float, void, boolean, public, private, class
Symbols	= , - ( )
Regular expressions	*id* = [_a-zA-Z]+[_a-zA-Z0-9]* *num* = [0-9]+
Groups	*modifiers* = public | private *type* = int | float | void | id | boolean | id

According to Table 3.9, `id` is extracted as the defined regular expression, but also will be classified as *type*. A token then, will be written as the set of all the possible classifications in which it is defined with its elements enclosed in brackets. Table 3.10 presents tokens examples extracted for their given inputs using the lexical definition from Table 3.9.

Table 3.10. Tokens examples

Input	Tokens
`public class MyClass`	`[public mod] [class] [id type]`
`int a`	`[int type] [id type]`
`a=3`	`[id type] [=] [num]`

A token will equal another token as long as one of their classifications matches. For example `[id type]` equals `[int type]`.

An example of a lexical definition for a Java subset, that is going to be used throughout this thesis, is presented in Table 3.11 below.

Table 3.11. Lexical definition for a Java subset

Type	Values
Symbols	. [ ] { } ( ) + - * / % ! = , & | @ : ? " ' ; < > # $ \ && || != <= >= ==
Reserved	abstract assert boolean break byte case catch char class const continue default do double else enum extends false final finally float for package private protected public return short static strictfp string super switch goto if implements import instanceof int interface long native new null synchronized this throw throws transient true try void volatile while
Regular expressions	*id* = [_a-zA-Z]+[_a-zA-Z0-9]* *num* = [0-9]+
Groups	*mod* = public | private | protected *type* = int | float | void | id | boolean | char | double | byte | short | string

3.5.3 Syntactical definition

The inference process also needs an initial set of syntax rules in order to generate the grammar with the desired types of productions. These rules include the binary operators' precedence, symbols that act as instructions delimiters, and pairs of context symbols. The last rule defines pairs of delimiters symbols defining a context for certain instructions, for example parenthesis and braces. Table 3.12 expands the lexical definition with the syntax rules.

Table 3.12. Lexical and syntactical definition

<table>
<tr><th>Type</th><th>Values</th></tr>
<tr><td>Lexical definition</td><td></td></tr>
<tr><td>Symbols</td><td>. [ ] { } ( ) + - * / % ! = , & | @ : ? " ' ; < > # $ \ && || != <= >= ==</td></tr>
<tr><td>Reserved</td><td>abstract assert boolean break byte case catch char class const continue default do double else enum extends false final finally float for package private protected public return short static strictfp string super switch goto if implements import instanceof int interface long native new null synchronized this throw throws transient true try void volatile while</td></tr>
<tr><td>Regular expressions</td><td>id = [_a-zA-Z]+[_a-zA-Z0-9]*
num = [0-9]+</td></tr>
<tr><td>Groups</td><td>mod = public | private | protected
type = int | float | void | id | boolean | char | double | byte | short | string</td></tr>
<tr><td>Syntactical definition</td><td></td></tr>
<tr><td>Delimiters</td><td>{ ;</td></tr>
<tr><td>Precedence</td><td>+ - * / % & | = :
< > == != <= >= ,
|| &&
;</td></tr>
<tr><td>Context symbols</td><td>( ), [ ], { }</td></tr>
</table>

With the lexical and syntax definitions, the parser has all the information needed to infer the grammar.

3.5.4 Inference process

The inference process consists of four main steps: expressions finding, expressions reduction, expressions to grammar conversion and grammar reduction. The parser is only needed for the first step, all the other steps are based on the manipulation of the created

expressions. This represents a great advantage because according to how the expressions are manipulated, the final grammar is generated, thus the overall process can be easily modified to achieve better or different results, without the need to modify the parser. Each one of the steps is described below.

Expressions finding

The expressions are created from the input by parsing instruction by instruction. The parser uses a stack and pushes tokens until a syntax rule (see Table 3.12) is found and relevant tokens in the stack are extracted, from which the expressions are created. Additionally, a second stack is used in order to keep track of the starting indices according to their position in the tokens stack.

In general, two types of expressions are created:

```
expr : α
expr : α δ β
```

Where α and β are a set of symbols or expressions names, and δ is a set of binary operators with the same precedence. Since each expression represents a pattern, the final set of expressions cannot contain duplicates.

Once a syntax rule is found, the instruction is analyzed right to left in order to match the correct patterns. It follows that order because it provides a better chance to match the correct patterns, and this is especially important when parsing binary symbols. For example, in the following code:

```
public float x , e , f ;
```

if the analysis is made left to right, it would be complicated to identify the expected correct patterns. The parser could identify "`public float x`" as a pattern since no further information is provided, and then the `id` pattern. By creating that pattern, the inferred grammar would incorrectly be written to accept strings like "`public float a, public float b, d`". On the other hand, by analyzing the instruction right to left, the `id` pattern is founded first and `x` is matched as `id`, creating a correct grammar. It also affects if patterns have been already found, for example in the next code:

```
public int Method( int a , float b , int c );
public float x , e , f ;
```

a pattern `{type id}` is found when analyzing the method parameters. If the second instruction is analyzed left to right, the pattern "`float x`" would be found first, which is not correct for this instruction. It is important to consider that while the parser tries to match the correct pattern, it might be not always possible. From the above example, if the parser is not able to identify `x` as a pattern in the second instruction, "`float x`" would be the pattern, leading to an incorrect grammar. Usually the rightmost pattern is easier to find since it is delimited by symbols or the end of instruction.

The following code shows the stack based parser, highlighting the code lines where the expressions are created.

```
Infer Expressions
      for each token in the input
            Create expression with token
            if token is an instruction delimiter
                  Pop index
                  Parse instruction
                  Push current index
            else
                  if token is a context symbol
                        Parse context
                  else
                        Push token
                        Create expression expr_x : a where a = token
      expr_y = Parse the remaining of the stack
      Create expr_x : expr_y
```

```
Parse context
      Extract the tokens from the stack between the context symbols as the
current instruction
      expr_y = contextsymbol + Parse instruction + contextsymbol
```

```
Parse instruction
      Repeat until no binary symbol is found
            Find operator with lower precedence in the instruction
            Find operands patterns
            Create expression expr_x : op1 op op2

      Parse remaining of the instruction
```

```
Find operands
      Find first operand as pattern and create expr : op1
      Find second operand as pattern and create expr : op2
```

In order to illustrate the inference process, Table 3.13 presents the behavior of the parser with the following input code:

```
class MyClass
{
    private int a;
    public void Method(int a, float b);
}
```

Table 3.13. Expressions finding example

Input	Actions	Expressions created	Tokens stack	Indices stack
class	Push token	`expr_0 : ‘class’`	`[class]`	[0]
MyClass	Push token	`expr_1 : [id type]`	`[class], [id type]`	[0]
{	Push token Push current index	`expr_2 : ‘{‘`	`[class], [id type], [{]`	[0,2]
private	Push token	`expr_3 : [private mod]`	`[class], [id type], [{], [private mod]`	[0, 2]
int	Push token	`Same as expr_1`	`[class], [id type], [{], [private mod], [int type]`	[0, 2]
a	Push token	`Same as expr_1`	`[class], [id type], [{], [private mod], [int type], [id type]`	[0, 2]
;	Pop index (2) Parse instruction Push index (4)	`expr_4 : ‘;’` `expr_5 : [private mod] [int type] [id type]`	`[class], [id type], [{], [expr_5], [;]`	[0, 4]
public	Push token	`Same as expr_3`	`[class], [id type], [{], [expr_5], [;], [public mod]`	[0, 4]
void	Push token	`Same as expr_1`	`[class], [id type], [{], [expr_5], [;], [public mod], [void type]`	[0, 4]
method	Push token	`Same as expr_1`	`[class], [id type], [{], [expr_5], [;], [public mod], [void type], [id type]`	[0, 4]
(	Push token	`expr_6 : ‘(‘`	`[class], [id type], [{], [expr_5], [;], [public mod], [void type], [id type], [(]`	[0, 4]
int	Push token	`Same as expr1`	`[class], [id type], [{], [expr_5], [;], [public mod], [void type], [id type], [(], [int type]`	[0, 4]
a	Push token	`Same as expr1`	`[class], [id type], [{], [expr_5], [;], [public mod], [void type], [id type], [(], [int type], [id type]`	[0, 4]
,	Push token	`expr7 : ‘,’`	`[class], [id type], [{], [expr_5], [;], [public mod], [void type], [id type], [(], [int type], [id type], [,]`	[0, 4]
float	Push token	`Same as expr1`	`[class], [id type], [{], [expr_5], [;], [public mod], [void type], [id type], [(], [int type], [id type], [,], [float type]`	[0, 4]

Table 3.13. (continued)

Input	Actions	Expressions created	Tokens stack	Indices stack
b	Push token	Same as expr1	[class], [id type], [{], [expr_5], [;], [public mod], [void type], [id type], [(], [int type], [id type], [,], [float type], [id type]	[0, 4]
)	Parse context See Table 3.14 for details	expr_8 : ‘)’ expr_9 : [float type] [id type] expr_10 : expr_9 ‘,’ expr_9 expr_11 : ‘(‘ expr_10 ‘)’	[class], [id type], [{], [expr_5], [;], [public mod], [void type], [id type], [expr_11]	[0, 4]
;	Pop index (4) Parse instruction Push index (6)	Same as expr4 expr_12 : [public mod] [void type] [id type] expr_11	[class], [id type], [{], [expr_5], [;], [expr_12], [;]	[0, 6]
}	Pop index (6) See Table 3.15 for details	expr_13 : '}' expr_14 : nothing expr_15 : expr_12 expr_16 : expr_15 ';' expr_14 expr_17 : expr_16 expr_18 : expr_5 expr_19 : expr_18 ';' expr_17 expr_20 : '{' expr_19 '}'	[class], [id type], [expr_20]	0
EOF	Pop 0 Parse instruction	expr_21 : 'class' [id type] expr_20	Empty	empty

Table 3.14. Expressions finding first context details

Expressions	Current stack to analyze	Symbol	Left op	Right op
expr_9 : [float type] [id type] expr_10 : expr_9 ‘,’ expr_9	[int type], [id type], [,], [float type], [id type]	,	[int type] [id type]	[float type] [id type]

Table 3.15. Expressions finding second context details

Expressions	Current stack to analyze	Symbol	Left op	Right op
expr_14 : nothing expr_15 : expr_12 expr_16 : expr_15 ';' expr_14	[expr_5], [;], [expr_12], [;]	;	Expr12	nothing
expr_17 : expr_16 expr_18 : expr_5 expr_19 : expr_18 ';' expr_17	[expr_5], [;], [expr_16]	;	Expr5	Expr16

Once the set of expressions is created a parser is no longer needed, the final grammar is created from manipulating the expressions as presented in the following Chapters.

Expressions reduction

To this point, all instructions are parsed and a set of expressions is created that are logically grouped according to the instruction that created them. In order to create a common set of expressions for all the language instructions, from which the grammar can be inferred, the following general steps are followed:

1. Remove all expressions that replace in two steps in order to replace in one step if possible. This is achieved by replacing all expressions `expr_x` with `expr_y` for all `expr_x : expr_y` and `expr_y :` α. This will reduce additional productions in the final grammar.
2. For all expressions matching the type `expr_x :` $\alpha\ \delta\ \beta$ where δ is any operator, create a new expression `expr_y :` $\alpha \mid \beta$ and `expr_z : expr_y` δ `expr_y`. Remove all `expr_x` found.
3. As a complement of the last step all, for all expressions of the type $expr_x_n$: $expr_y_n\ \delta\ expr_y_n$ where δ is any operator with the same precedence, create a single $expr_x_0$: $expr_y_0\ \gamma\ expr_y_0$ where γ is the set of all operators found, and add all $expr_y_n$ to $expr_y_0$.

The example expressions from the Chapter above are reduced as depicted in Table 3.16.

Table 3.16. Expressions reduction

Step	Expressions	Notes
-	expr_0 : 'class' expr_1 : [id type] expr_2 : '{' expr_3 : [private mod] expr_4 : ';' expr_5 : [private mod] [int type] [id types] expr_6 : '(' expr_7 : ',' expr_8 : ')' expr_9 : [float type] [id type] expr_10 : expr_9 ',' expr_9 expr_11 : '(' expr_10 ')' expr_12 : [public mod] [void type] [id type] expr_11 expr_13 : '}' expr_14 : nothing expr_15 : expr_12 expr_16 : expr_15 ';' expr_14 expr_17 : expr_16 expr_18 : expr_5 expr_19 : expr_18 ';' expr_17 expr_20 : '{' expr_19 '}' expr_21 : 'class' [id type] expr_20	Original expressions
1	expr_0 : 'class' expr_1 : [id type] expr_2 : '{' expr_3 : [private mod] expr_4 : ';' expr_5 : [private mod] [int type] [id types] expr_6 : '(' expr_7 : ',' expr_8 : ')' expr_9 : [float type] [id type] expr_10 : expr_9 ',' expr_9 expr_11 : '(' expr_10 ')' expr_12 : [public mod] [void type] [id type] expr_11 expr_13 : '}' expr_14 : nothing expr_16 : expr_12 ';' expr_14 expr_19 : expr_5 ';' expr_16 expr_20 : '{' expr_19 '}' expr_21 : 'class' [id type] expr_20	Expression 15 is replace by expression 12 Expression 17 is replace by expression 16 Expression 18 is replace by expression 5

Table 3.16. (continued)

Step	Expressions	Notes
2	expr_0 : 'class' expr_1 : [id type] expr_2 : '{' expr_3 : [private mod expr_4 : ';' expr_5 : [private mod] [int type] [id type] expr_6 : '(' expr_7 : ',' expr_8 : ')' expr_9 : [float type] [id type] expr_11 : '(' expr_23 ')' expr_12 : [public mod] [void type] [id type] expr_11 expr_13 : '}' expr_14 : nothing expr_20 : '{' expr_25 '}' expr_21 : 'class' [id type] expr_20 expr_22 : expr_9 expr_23 : expr_22 ',' expr_22 expr_24 : expr_12 | expr_14 | expr_5 | expr_25 expr_25 : expr_24 ';' expr_24	Expression 22 and 23 are created from expression 10 with operator [,]. Expression 10 is removed. Expressions 24 and 25 are created from expressions 16 and 19 with operator [;]. Expressions 16 and 19 are removed.
3	expr_0 : 'class' expr_1 : [id type] expr_2 : '{' expr_3 : [private mod] expr_4 : ';' expr_5 : [private mod] [int type] [id type] expr_6 : '(' expr_7 : ',' expr_8 : ')' expr_9 : [float type] [id type] expr_11 : '(' expr_23 ')' expr_12 : [public mod] [void type] [ID type] expr_11 expr_13 : '}' expr_14 : 'nothing' expr_20 : '{' expr_25 '}' expr_21 : 'class' [id type] expr_20 expr_22 : expr_9 expr_23 : expr_22 ',' expr_22 expr_24 : expr_12 | expr_14 | expr_5 | expr_25 expr_25 : expr_24 ';' expr_24	No actions

Conversion to grammar

At this point the set of expressions are common for the language input and a general context-free grammar can be created as follows:

1. Create a production $S \rightarrow S_x$ as the initial production, where x is the last expression created before the reduction steps.

2. For all expressions of the type `expr_x : α`, where α is any set of symbols or expressions but contains at least one symbol, create the production $S_x \rightarrow \alpha$.

3. For all expressions of the type `expr_x : expr_y δ expr_y`, create the productions $S_n \rightarrow S_{n+1}\ \delta\ S_{n+1}$ and $S_{n+1} \rightarrow A_1 \mid A_2 \ldots \mid A_n$, where A is the set of expressions that can be reach in one or more steps from `expr_y`. If A contains `expr_x`, create $S_n \rightarrow S_n\ \delta\ S_{n+1} \mid S_{n+1}$ instead.

4. All other expressions are ignored.

The grammar is created from the expressions from the Chapter above as depicted in Table 3.17.

Table 3.17. Context free grammar creation

Step	Grammar	
1	S → S21	Expression 21 is set as the first production
2	S → S21 S0 → 'class' S1 → [id type] S2 → '{' S3 → [private mod] S4 → ';' S5 → [private mod] [int type] [id type] S6 → '(' S7 → ',' S8 → ')' S9 → [float type] [id type] S11 → '(' S23 ')' S12 → [public mod] [void type] [id type] S11 S13 → '}' S14 → nothing S20 → '{' S25 '}' S21 → 'class' [id type] S20	Productions 0 to 21 are created from their respective expressions numbers

Table 3.17. (continued)

Step	Grammar	
3	S → S21 S0 → 'class' S1 → [id type] S2 → '{' S3 → [private mod] S4 → ';' S5 → [private mod] [int type] [id type] S6 → '(' S7 → ',' S8 → ')' S9 → [float type] [id type] S11 → '(' S23 ')' S12 → [public mod] [void type] [id type] S11 S13 → '}' S14 → nothing S20 → '{' S25 '}' S21 → 'class' [id type] S20 S22 → S9 S23 → S26 ',' S26 S26 → S9 S25 → S25 ';' S27 | S27 S27 → S12 | S14 | S5	From expression 23, productions S23 and S26 are created. From expression 25, productions S25 and S27 are created.

Grammar reduction

As a final step, the inferred grammar can be simplified by:

1. Remove unused and unreferenced productions.
2. Remove duplicate productions
3. Group by edit distance. For this thesis only an edit distance of one with a substitution operation is considered.

Unused and duplicated productions are generated given that some productions were created from expressions of the type `expr_x : α`, but not all expressions are part of the final grammar, or unless not in their original form.

The concept of distance from step three is the same as in approximate string matching. For example, given a specific edit distance number, two strings are the same if the edit

distance is lower or equal the number of operations (insertion, deletion or substitution) applied to match the strings. For example *house* and *houses* are the same word with an edit distance of one (using a deletion operation by removing the second letter *s* from *houses*). For our process, a production is considered a string and each symbol is considered as a character, for example, the following productions are equal with an edit distance of one:

```
Sx → ‘public’ ‘void’ id
Sy → ‘public’ ‘int’ id
```

For this thesis only and edit distance of one with a substitution operation is considered.

To this point a grammar can be inferred that recognizes simple inputs, especially for the language structures we are interested in. Applying the rules above, the final inferred grammar for our example is presented below:

```
S   → S21
S5  → [private mod] [int type] [id type]
S9  → [float type] [id types]
S11 → '(' S23 ')'
S12 → [public mod] [void type] [id type] S11
S14 → nothing
S20 → '{' S25 '}'
S21 → 'class' [id type] S20
S23 → S26 ',' S26
S26 → S9
S25 → S25 ';' S27 | S27
S27 → S12 | S14 | S5
```

3.5.5 Context-free grammar inference example

The following input example is presented to illustrate the operator’s precedence syntax rule:

```
if (a == 8 && b.c + 3 != 9) { }
```

Table 3.18 shows the expressions created and the reduction steps, and Table 3.19 shows the grammar creation from the reduced set of expressions.

Table 3.18. Expressions reduction example

Step	Expressions	Notes
-	expr_0 : 'if' expr_1 : '(' expr_2 : [id type] expr_3 : '==' expr_4 : [num] expr_5 : '&&' expr_6 : '.' expr_7 : '+' expr_8 : '!=' expr_9 : ')' expr_10 : expr_2 '.' expr_2 expr_11 : expr_10 expr_12 : expr_11 '+' expr_4 expr_13 : expr_12 expr_14 : expr_13 '!=' expr_4 expr_15 : expr_2 '==' expr_4 expr_16 : expr_14 expr_17 : expr_15 expr_18 : expr_17 '&&' expr_16 expr_19 : '(' expr_18 ')' expr_20 : '{' expr_21 : '}' expr_22 : '{' '}' expr_23 : 'if' expr_19 expr_22	Original expressions
1	expr_0 : 'if' expr_1 : '(' expr_2 : [id type] expr_3 : '==' expr_4 : [num] expr_5 : '&&' expr_6 : '.' expr_7 : '+' expr_8 : '!=' expr_9 : ')' expr_10 : expr_2 '.' expr_2 expr_12 : expr_10 '+' expr_4 expr_14 : expr_12 '!=' expr_4 expr_15 : expr_2 '==' expr_4 expr_18 : expr_15 '&&' expr_14 expr_19 : '(' expr_18 ')' expr_20 : '{' expr_21 : '}' expr_22 : '{' '}' expr_23 : 'if' expr_19 expr_22	Expression 11 is replace by expression 10 Expression 13 is replace by expression 12 Expression 16 is replace by expression 14 Expression 17 is replace by expression 15

Table 3.18. (continued)

<table>
<tr><th>Step</th><th>Expressions</th><th>Notes</th></tr>
<tr><td>2</td><td>expr_0 : 'if'
expr_1 : '('
expr_2 : [id type]
expr_3 : '=='
expr_4 : [num]
expr_5 : '&&'
expr_6 : '.'
expr_7 : '+'
expr_8 : '!='
expr_9 : ')'
expr_19 : '(' expr_33 ')'
expr_20 : '{'
expr_21 : '}'
expr_22 : '{' '}'
expr_23 : 'if' expr_19 expr_22
expr_24 : expr_2
expr_25 : expr_24 '.' expr_24
expr_26 : expr_25 | expr_4
expr_27 : expr_26 '+' expr_26
expr_28 : expr_2 | expr_4
expr_29 : expr_28 '==' expr_28
expr_30 : expr_27 | expr_4
expr_31 : expr_30 '!=' expr_30
expr_32 : expr_29 | expr_31
expr_33 : expr_32 '&&' expr_32</td><td>Expressions 24 and 25 are created from expression 10 with operator [.]. Expression 10 is removed.

Expressions 26 and 27 are created from expression 12 with operator [+]. Expression 12 is removed.

Expressions 28 and 29 are created from expression 15 with operator [==]. Expression 15 is removed.

Expressions 30 and 31 are created from expression 14 with operator [!=]. Expression 14 is removed.

Expressions 32 and 33 are created from expression 18 with operator [&&]. Expression 18 is removed.</td></tr>
<tr><td>3</td><td>expr_0 : 'if'
expr_1 : '('
expr_2 : [id type]
expr_3 : '=='
expr_4 : [num]
expr_5 : '&&'
expr_6 : '.'
expr_7 : '+'
expr_8 : '!='
expr_9 : ')'
expr_19 : '(' expr_33 ')'
expr_20 : '{'
expr_21 : '}'
expr_22 : '{' '}'
expr_23 : 'if' expr_19 expr_22
expr_24 : expr_2
expr_25 : expr_24 '.' expr_24
expr_26 : expr_25 | expr_4
expr_27 : expr_26 '+' expr_26
expr_28 : expr_2 | expr_4 | expr_30
expr_29 : expr_28 '== !=' expr_28
expr_30 : expr_27 | expr_4
expr_32 : expr_29 | expr_29
expr_33 : expr_32 '&&' expr_32</td><td>Expression 29 is merged with expression 31 with symbols [==, !=]. Expression 31 is removed. Expression 28 is modified to contain expression 30, since is the operand from the merged expression 31.</td></tr>
</table>

Table 3.19. Context free grammar creation

<table>
<tr><th>Step</th><th>Grammar</th><th>Notes</th></tr>
<tr><td>1</td><td>S → S23</td><td>Expressions 23 is set as the first production</td></tr>
<tr><td>2</td><td>S → S23
S0 → 'if'
S1 → '('
S2 → [id type]
S3 → '=='
S4 → [num]
S5 → '&&'
S6 → '.'
S7 → '+'
S8 → '!='
S9 → ')'
S19 → '(' S33 ')'
S20 → '{'
S21 → '}'
S22 → '{' '}'
S23 → 'if' S19 S22</td><td>Productions 0 to 23 are created from their respective expressions numbers</td></tr>
<tr><td>3</td><td>S → S23
S0 → 'if'
S1 → '('
S2 → [id type]
S3 → '=='
S4 → [num]
S5 → '&&'
S6 → '.'
S7 → '+'
S8 → '!='
S9 → ')'
S19 → '(' S33 ')'
S20 → '{'
S21 → '}'
S22 → '{' '}'
S23 → 'if' S19 S22
S24 → S2
S25 → S34 '.' S34
S34 → S2
S27 → S35 '+' S35
S35 → S25 | S4
S29 → S36 '== !=' S36
S36 → S2 | S4 | S27
S33 → S37 '&&' S37
S37 → S29</td><td>From expression 25, productions S25 and S34 are created.

From expression 27, productions S27 and S35 are created.

From expression 29, productions S29 and S36 are created.

From expression 23, productions S23 and S37 are created.</td></tr>
</table>

Following the grammar reduction steps, the inferred grammar is presented below:

```
S   → S23
S2  → [id type]
S4  → [num]
S19 → '(' S33 ')'
S22 → '{' '}'
S23 → 'if' S19 S22
S25 → S34 '.' S34
S34 → S2
S27 → S35 '+' S35
S35 → S25 | S4
S29 → S36 '== !=' S36
S36 → S2 | S4 | S27
S33 → S37 '&&' S37
S37 → S29
```

3.6 Chapter summary

In this Chapter the two methodologies for aiding the code metrics extraction process were fully described. The first methodology allows querying the programming language context-free grammar in order to extract relevant substring from the source code. These strings are arranged on a tree structure in order to represent the hierarchical structure of the source code. The tree is described as source code representation in the form of a data model. This model is created dynamically with each query, so all the necessary information is always present. The MQL language is described in terms of its context-free grammar and functioning. This language formalizes the proposed methodology.

The grammatical inference methodology is also described. It is explained how it works based on finding source code patterns. These patterns compose syntax rules that drive the discovery of expressions. These expressions, full or modified, are transformed into productions which might be part of the final inferred grammar. Both the expressions and

the inferred productions must go through a series of transformations in order to infer the correct grammar. Examples are presented in order to depict the functioning of the methodology.

In the next Chapter the results obtained by applying these methodologies are presented.

Chapter 4

4. Results and comparisons with other tools

In this Chapter the results from the proposed methodology are presented. Based on the described methodology, a metrics tool named MQLMetrics is constructed based on MQL described in Chapter 3.4.2. Generalities about the tool are presented in Chapter 4.2, and a case study in which the tool is used to match metrics results obtained from other tools is presented in Chapter 4.3. Results from the grammatical inference process are presented in Chapter 4.4.

4.1 MQLMetrics description

A metrics tool named MQLMetrics is created to formalize and demonstrate the use of the extraction methodology presented in Chapter 3 based on the MQL language. In order to demonstrate the use of the queries using these tool two examples for metrics extraction

from Java source code are presented below. The complete Java grammar can be found in Appendix C.

4.1.1 A query for common object oriented metrics example

A query for creating a data model from which common metrics for the object oriented paradigm can be extracted and computed is presented. Table 4.1 presents two BMs useful to extract the *Number of Classes* (NCLASS) and *Number of Methods* (NOM) metrics.

Table 4.1. BMs and MQL queries for the NCLASS and NOM metrics

BM1	**All classes identifiers**	
	Value	**Query**
Scope	Class declaration	$.classDeclaration + .anonymousClassBody$
Attribute	Classes identifiers	$.classDeclaration!\,identifier$
MQL query	$.classDeclaration!\,identifier\ in\ .classDeclaration + .anonymousClassBody$	
BM2	**All method identifiers**	
	Value	**Query**
Scope	Method declaration	$.methodDeclaration$
Attribute	Methods identifiers	$.methodDeclaration!\,identifier$
MQL query	$.methodDeclaration!\,identifier\ in\ .methodDeclaration$	

The examples are for the Java language, which allows the definition of inner classes and anonymous classes. In the scope element of the BM1 we aim to capture all types of class definitions, and since anonymous classes are not defined as regular classes, we conjunct all the scopes created from normal classes and anonymous definitions into a single derivation path with the *plus* symbol. With these two BMs we can then define the queries for the NCLASS and NOM metrics as shown in Table 4.2.

Table 4.2. MQL queries for the NCLASS and NOM metrics

Metric	MQL Query
NCLASS	$.classDeclaration!identifier\ in\ .classDeclaration + .anonymousClassBody$
NOM	$.classDeclaration!identifier\ in\ .classDeclaration + .anonymousClassBody\ \&$ $methodDeclaration!identifier\ in\ .methodDeclaration$

NCLASS is defined as a BM, but NOM is defined as a CM because the number of methods must be reported per class. We can extend the example to create a bigger result tree that could be considered a data model for metrics extraction. In Java, classes are defined inside packages, and methods are defined inside classes. Each method is defined by its modifier, method identifier, and the parameters identifiers. From that data, many metrics can be defined. Table 4.3 complements the information in Table 4.1, and shows the necessary BMs to write the query that will allows us to extract the desired data.

Table 4.3. Direct metrics and MQL queries for the object oriented model

BM3	**Packages names**	
	Value	**Query**
Scope	File	$.compilationUnit$
Attribute	Package name	$.packageDeclaration!qualifiedName$
MQL query	$.packageDeclaration!qualifiedName\ in\ .compilationUnit$	
BM4	**Methods modifiers**	
	Value	**Query**
Scope	Member methods declaration	$.memberMethodsDeclaration$
Attribute	Method modifier identifier	$.memberMethodsDeclaration!modifiers!modifier$
MQL query	$.memberMethodsDeclaration!modifiers!modifier\ in\ .memberMethodsDeclaration$	
BM5	**Parameters identifiers**	
	Value	**Query**
Scope	Method declaration	$.methodDeclaration$
Attribute	Parameters identifiers	$.formalParameter!variableDeclaratorId!identifier$
MQL query	$.formalParameter!variableDeclaratorId!identifier\ in\ .methodDeclaration$	

Note that the selected scopes are within each other according to what we are trying to achieve, for example the package name (BM3) has *file* as the scope, since it must contain

all other results. The CM is written as a single query conjucting all BMs with the symbol *ampersand* in the desired order:

.packageDeclaration!qualifiedName ***in*** *.compilationUnit* ***&***
.classDeclaration!identifier ***in*** *.classDeclaration+.anonymousClassBody* ***&***
.memberMethodsDeclaration!modifiers!modifier ***in*** *.memberMethodsDeclaration* ***&***
.methodDeclaration!identifier ***in*** *.methodDeclaration* ***&***
.formalParameter!variableDeclaratorId!identifier ***in*** *.methodDeclaration*

This query will provide a result tree as depicted in Figure 4.1.

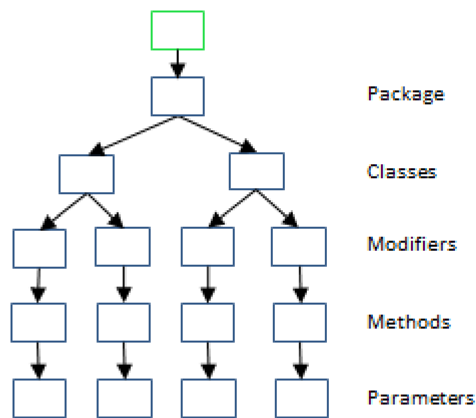

Figure 4.1 - Result tree

Using the source code in Appendix B as an input, the result from the extraction is presented in Figure 4.2.

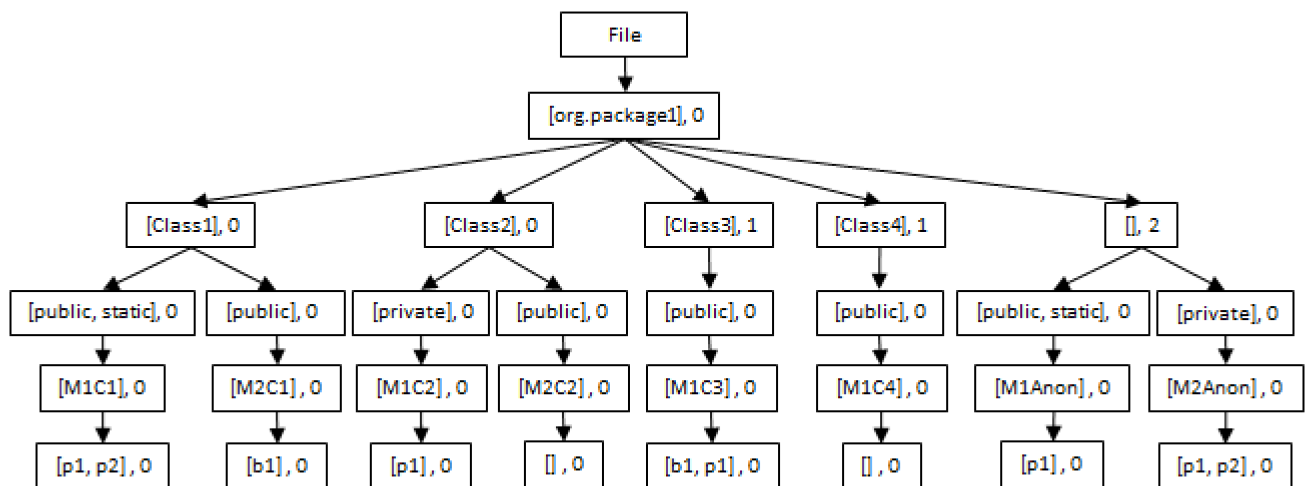

Figure 4.2 - Populated result tree

From the result tree, several metrics can be obtained along with additional information that can be obtained by the node depth value. For this case, all classes with a value of depth more than zero, represent inner or anonymous classes. The example presented in the following Chapter uses the depth value as part of the metric computation.

4.1.2 Cognitive weight example

The cognitive weight is a complicated metric to extract, and was chosen as an example of the capacity of the methodology for aiding the source code metric extraction beyond the common object oriented metrics. The cognitive weight of a program is defined as “the degree of difficulty or relative time and effort required for comprehending a given piece of software modeled by a number of basic control structures (sequential, branch, iteration, function call)” and is the base for the calculation of the cognitive complexity (Shao and Wang, 2003), which measures the complexity of a program considering its internal structure.

In order to compute it, each control structure is given a cognitive value. If the control structures are in a linear layout, the weights are summed, if they are embedded, the weights of the inner control structures are multiplied with the weight of the external structures. The necessary data to extract the cognitive weight for each method is defined by the BMs presented in Table 4.4.

Table 4.4. BMs and MQL queries for the cognitive weight metric

BM1	**Control statements (if, while, switch…)**	
	Value	**Query**
Scope	Method declaration and control statements	$.methodDeclaration + .controlStatement$
Attribute	Control statements	$.controlStatement$
MQL query	$.controlStatement\ in\ .methodDeclaration + .controlStatement$	
BM2	**Method calls**	
	Value	**Query**
Scope	Method calls	$.methodCall$
Attribute	Method calls	$.methodCall$
MQL query	$.methodCall\ in\ .methodCall$	

BM1 extracts all control statements and BM2 extracts all method calls. The *methodDeclaration* non-terminal is added to the derivation in the first BM scope in order to provide a placeholder for the main linear layout needed to compute the metric, it also establish itself as the node with the lowest depth in the tree, which is necessary to create the needed depth hierarchy. Figure 4.3 presents the input we are using for this example.

```
public void m()
{
    if (a == true) Call1();
    if (b == true)
    {
        Call2();
        while (c == true)
        {
            Call3();
            if (c == true) Call4();
        }
    }
    Call5();
}
```

Figure 4.3 - Input example

By linking both BMs definitions, we aim to create a tree consisting on the control statements, including the method declaration that is a sequential control structure by

itself, with their corresponding depths and method calls embedded in each statement. Figures 4.4, 4.5 and 4.6 present the construction of the result tree.

```
public void m()
{
    if (a == true) Call1();
    if (b == true)
    {
        Call2();
        while (c == true)
        {
            Call3();
            if (c == true) Call4();
        }
    }
    Call5();
}
```

Figure 4.4 - Code nodes identification

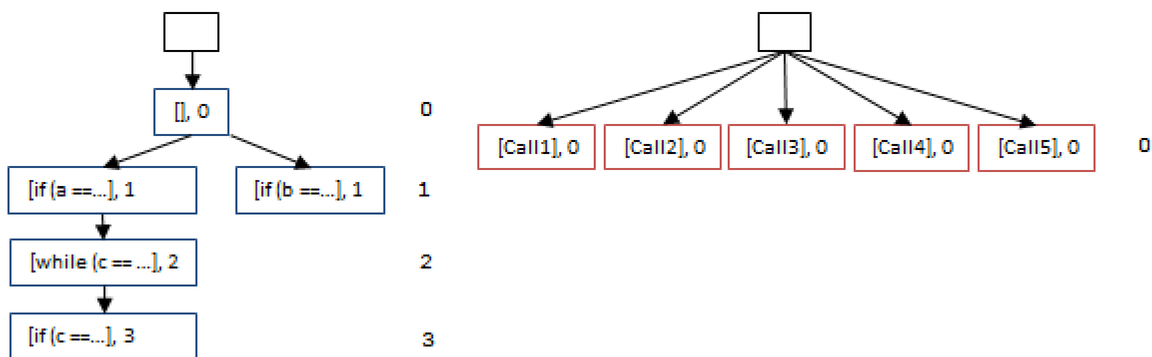

Figure 4.5 - Result tree of each BM

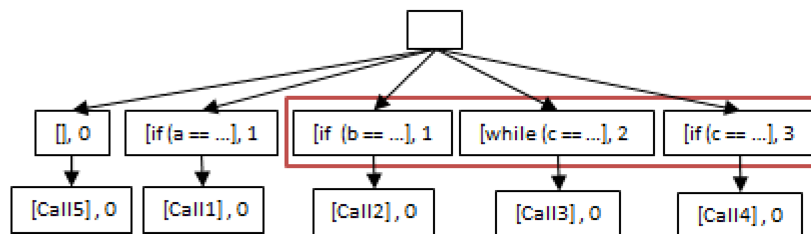

Figure 4.6 - Merged tree

The tree contains all data necessary to compute the metric. We can traverse the tree and calculate the metric via a recursive function (the red portion in Figure 4.6 indicates that

the tree preserves the nest relation between siblings) applying the corresponding weights, and summing and multiplying according to the depth information.

With this example, we want to demonstrate that complex metrics can be computed, along with the flexibility and different uses the presented methodology can have.

4.2 MQLMetrics generalized inputs support

MQLMetrics is written in Java. It is able to generate metrics datasets in XML format and supports:

- Object oriented metrics for source code written in Java (.java files)
- Aspect oriented metrics for source code written in AspectJ (.aj file)
- XML metrics (.xml files)

We chose the options above to showcase that the methodology is useful for measuring different programming paradigms, and also, that it is not restricted to extract metrics exclusively from programming languages. The supported code metrics cover a variety of design and quality attributes, such as size, complexity, inheritance, cohesion and coupling. The metrics are also extracted at different scopes, such as file, system, class and method levels.

Example datasets were created for several benchmark systems. The lists of measured systems for the object and aspect oriented paradigms are presented in Table 4.5 and Table 4.6 respectively. These benchmark systems are commonly used in code metrics research and were selected in order to provide real life measurements. The MobileMedia

system is a benchmark system for both paradigms and we chose to measure the eight versions of it. The XML metrics are extracted from the XML files generated as the datasets. The full list of source code metrics that are supported by the tool is presented in Appendix A, in total, 77 metrics can be extracted.

Table 4.5. Object oriented systems

System	Version	Files (.java)	Total lines	Classes
AJHotDraw	0.4	290	38224	291
ANT	1.9.7	1216	246485	1551
ArgoUML	0.34	1922	326155	2096
JHotDraw	5.4	484	66648	496
JUnit	4.12	426	39620	1046
MobileMedia	01	15	1720	15
MobileMedia	02	23	2002	23
MobileMedia	03	24	2194	24
MobileMedia	04	24	2205	24
MobileMedia	05	29	2490	29
MobileMedia	06	36	3198	36
MobileMedia	07	45	3662	45
MobileMedia	08	52	4052	52
Totals		4586	738655	5728

Table 4.6. Aspect oriented systems

System	Version	Files (.aj)	Total lines	Aspects
AJHotDraw	0.4	31	2579	31
MobileMedia	02	4	405	4
MobileMedia	03	5	576	5
MobileMedia	04	8	797	8
MobileMedia	05	9	871	9
MobileMedia	06	10	1068	10
MobileMedia	07	14	1652	14
MobileMedia	08	23	2273	23
Totals		104	10221	104

The ANTLR framework (http://www.antlr.org/) was used to build the application. Using ANTLR provides advantages and disadvantages. The main disadvantage is that a parser must be created for every programming language we want to extract metrics from. The advantages are that it is a very common library, has been highly used and tested, and

because of that, many complete tested grammars for all common languages are already available. On the other hand, by writing a custom compiler, the methodology could be integrated directly into the parser, and that single parser would be able to analyze and generate the production values for any grammar, however, the effort and time required for constructing such compiler could prove to be too big.

4.3 Case study

A case study is presented in which the tool will reproduce the metrics results obtained by other tools. It is a complex task given that it has been studied that metrics tools usually provide different results for the same metric, so special considerations must be made in order to find the metrics definitions and computation methods. The selection of three components is essential to perform the study: the software metric tools to compare to, the set of metrics to extract, and the programming language and system to measure.

4.3.1 Metrics tools selection

Many extraction tools and methodologies were found, as presented in Chapter 1, but only four works provided a download link to test their tool, and from those four, only two were available. Also, according to our results (Nuñez-Varela et al., 2017), the commercial tools Understand and Metrics 1.3.6 are highly used in code metrics research and will be considered for this studio. Table 4.7 presents a summary of the considered tools.

Table 4.7. Software metrics tools

Available	Tool	Website	Version	Notes	Source code available
Yes	CKJM	http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/	CKJM Extended	The new version of the original tool (https://www.spinellis.gr/sw/ckjm/) will be used	Yes
Yes	MASU	https://sourceforge.net/projects/masu/	26/November/2010 release		Yes
No	OOMeter	www.ccse.kfupm.edu.sa/~oometer/oometer	-	Link to documentation only	-
No	QSCOPE	http://www.st.informatik.tu-darmstadt.de/Magellan	-	Page not found	-
Yes	Understand	https://scitools.com/	5		No
Yes	Metrics 1.3.6	http://metrics2.sourceforge.net/	1.3.8	The new version of the original tool (http://metrics.sourceforge.net/) will be used	No

The four available tools CKJM, MASU, Understand, and Metrics 1.3.8 were tested and their metrics results analyzed. It was decided to exclude CKJM from the study since many inconsistencies were found when analyzing its metrics results, especially because those results did not match with metrics results extracted manually from the source code. The authors of the tool acknowledge in the website's FAQ section (http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/faq.html) that, since the tool calculates metrics from compiled bytecode, some code elements are not taken in consideration for the final calculations, which represents an important issue.

4.3.2 Source code metrics selection

Object oriented metrics are the most widely studied metrics and are supported by the three selected tools, with Metrics 1.3.8 and MASU extracting exclusively object oriented metrics. From those metrics, *Depth of Inheritance Tree* (DIT), *Number of Children* (NOC), *Weighted Methods per Class* (WMC), *Response for a Class* (RFC), *Lack of Cohesion* (LCOM),

Coupling Between Objects (CBO), *Number of Methods* (NOM), and *Number of Attributes* (NOA) are the most used (Nuñez-Varela et al., 2017) and are chosen as the initial set of candidate metrics. Since WMC is based on the *Cyclomatic Complexity* (V(G)) of each method, V(G) is analyzed as part of the WMC metric. As a second step, the support and definition of each candidate metric is verified for each tool, and a metric is selected as part of the study if it is calculated and supported at least by one tool as the common definition of the metric. It is also considered if there is enough information available (i.e. documentation and source code) on how the metric is calculated. Table 4.8 presents the selected metrics depicted as gray cells, and the reason why a metric was not selected for a certain tool.

Table 4.8. Selected metrics per tool

	Understand	**Metrics 1.3.8**	**MASU**
DIT		Not selected since it counts the classes in the JDK+ inheritance tree, providing different results	
NOM			Not supported
NOC			
NOA		Not supported	Not supported
WMC/V(G)			No information available on how it is calculated
LCOM			
CBO	Not enough documentation	Not supported	Not enough documentation
RFC	A different definition, it counts the number of methods	Not supported	Not enough documentation

The metrics CBO and RFC will not be part of the study because not enough information on how they are calculated was available, and their definitions are very different across the tools.

4.3.3 System and programming language selection

Even though the Understand tool supports a variety of programming languages, Metrics 1.3.8 and MASU only support Java. For this reason, Java is chosen as our language. We chose the benchmark Java system JHotDraw version 7.0.6 (https://sourceforge.net/projects/jhotdraw/files/JHotDraw/JHotDraw%207.6/) as the system to measure. It is used in a wide variety of research papers and is highly accepted as a well-designed system for empirical studies on code metrics (Nuñez-Varela et al., 2017).

4.3.4 Results

Unless otherwise noted, the tools were used "out of the box" with their default settings and configurations. The metrics will be extracted from the main Java classes, that is, the classes defined by the file name thus excluding anonymous classes, inner classes and interfaces. A total of 272 main classes are a part of the JHotDraw system.

Table 4.9 presents how each metric is calculated for each tool. The metrics definitions and computation methods were obtained in the documentation of the tool, by looking at the source code, or by testing the tool and manually inferring the computation method. All metrics are calculated per class, so 272 values are obtained for each metric. Table 4.10 presents the match percentage obtained when trying to reproduce the values obtained by the selected tools.

Table 4.9. Metrics computation method

<table>
<tr><th>Metric/Tool</th><th>Understand</th><th>Metrics 1.3.8</th><th>MASU</th></tr>
<tr><td>DIT</td><td>Counts the number of nodes from the class node to the root of the inheritance tree. It adds one if the last parent class is not found (i.e. the parent derives from object).</td><td>-</td><td>Counts the number of nodes from the class node to the root of the inheritance tree. It adds one except if the last class is not found.</td></tr>
<tr><td>NOM</td><td>Counts the defined methods including constructors.</td><td>Counts the defined methods including constructors, but excludes static methods.</td><td></td></tr>
<tr><td>NOC</td><td>Counts the number of subclasses in the inheritance tree from a given class node.</td><td>Counts the number of subclasses in the inheritance tree from a given class node.</td><td>Counts the number of subclasses in the inheritance tree from a given class node.</td></tr>
<tr><td>NOA</td><td>Counts the number of static class variables.</td><td>-</td><td>-</td></tr>
<tr><td>WMC/V(G)</td><td>It counts the number of control instructions (if, switch, while, etc.), including the simplified if structure, plus one, per method. Then sums all the values.</td><td>It counts the number of && and || symbols, plus one, per method. The sums all the values.</td><td>-</td></tr>
<tr><td>LCOM</td><td>For each instance variable, divide the number of methods that use it by the total of methods, then average all the obtained values and subtract from 1. Multiply per 100 to obtain a final percentage.</td><td>Calculate the average of methods accessing each attribute, subtract the number of methods (m) and divide by 1-m.</td><td>Counts the number of different pairs of methods that do not share class variables.</td></tr>
</table>

Table 4.10. Match percentage per metric

Metrics values match between MQLMetrics and the other tools (272 values)											
	MQLMetrics as Understand	**Understand**	**Match**		**MQLMetrics as Metrics 1.3.8**	**Metrics 1.3.8**	**Match**		**MQLMetrics as MASU**	**MASU**	**Match**
DIT	272	272	100%		-	-			272	272	100%
NOM	271	272	99.6%		272	272	100%		-	-	
NOC	262	272	96.3%		262	272	96.3%		262	272	96.3%
NOA	272	272	100%		-	-	-		-	-	
WMC/ V(G)	179	272	65.8%		169	272	62.1%		-	-	
LCOM	246	272	90.4%		99	272	36.4%		94	272	34.5%

As discussed above, it might be hard to reproduce the values obtained by the existing software metrics tools. Regardless, we were able to match a high percentage of values. A 100% results match was obtained for DIT, NOA, and NOM. For NOM, one mismatched

value is found when extracting the values as calculated by Understand for the class *DefaultDrawingView*. Understand found 65 methods and MQLMetrics 66. The metric value was manually extracted corroborating that the result obtained by our tool is correct. Table 4.11 shows the 66 methods as defined in the class.

Table 4.11. Methods defined in the *DefaultDrawingView* class

#	Method	#	Method
1	public DefaultDrawingView()	34	public Collection<Handle> getCompatibleHandles(Handle master)
2	private void initComponents()	35	public Figure findFigure(Point p)
3	public Drawing getDrawing()	36	public Collection<Figure> findFigures(Rectangle r)
4	public java.util.Set getTools()	37	public Collection<Figure> findFiguresWithin(Rectangle r)
5	public void setEmptyDrawingMessage(String newValue)	38	public void addFigureSelectionListener(FigureSelectionListener fsl)
6	public String getEmptyDrawingMessage()	39	public void removeFigureSelectionListener(FigureSelectionListener fsl)
7	public void paintComponent(Graphics gr)	40	protected void fireSelectionChanged()
8	protected void drawBackground(Graphics2D g)	41	public void handleRequestRemove(HandleEvent e)
9	protected void drawGrid(Graphics2D g)	42	protected void invalidateDimension()
10	protected void drawDrawing(Graphics2D gr)	43	public Constrainer getConstrainer()
11	protected void drawHandles(java.awt.Graphics2D g)	44	public void setConstrainer(Constrainer newValue)
12	protected void drawTool(Graphics2D g)	45	public Dimension getPreferredSize()
13	public void setDrawing(Drawing d)	46	public Point drawingToView(Point2D.Double p)
14	protected void repaint(Rectangle2D.Double r)	47	public Point2D.Double viewToDrawing(Point p)
15	public void areaInvalidated(DrawingEvent evt)	48	public Rectangle drawingToView(Rectangle2D.Double r)
16	public void areaInvalidated(HandleEvent evt)	49	public Rectangle2D.Double viewToDrawing(Rectangle r)
17	public void figureAdded(DrawingEvent evt)	50	public Container getContainer()
18	public void figureRemoved(DrawingEvent evt)	51	public double getScaleFactor()
19	public void invalidate()	52	public void setScaleFactor(double newValue)
20	public void addToSelection(Figure figure)	53	protected void fireViewTransformChanged()
21	public void addToSelection(Collection<Figure> figures)	54	public void setHandleDetailLevel(int newValue)
22	public void removeFromSelection(Figure figure)	55	public int getHandleDetailLevel()
23	public void toggleSelection(Figure figure)	56	public void handleRequestSecondaryHandles(HandleEvent e)
24	public void selectAll()	57	public AffineTransform getDrawingToViewTransform()
25	public void clearSelection()	58	public void setDOMFactory(DOMFactory newValue)
26	public boolean isFigureSelected(Figure checkFigure)	59	public DOMFactory getDOMFactory()
27	public Collection<Figure> getSelectedFigures()	60	public void copy()
28	public int getSelectionCount()	61	public void cut()
29	private java.util.List<Handle> getSelectionHandles()	62	public void delete()
30	private java.util.List<Handle> getSecondaryHandles()	63	public void paste()
31	private void invalidateHandles()	64	public void duplicate()
32	private void validateHandles()	65	public void removeNotify(DrawingEditor editor)
33	public Handle findHandle(Point p)	66	public void addNotify(DrawingEditor editor)

As it can be observed from Table 4.11, there are 66 different methods (including the constructor), and no overridden or overloaded methods can be found which could affect the result in some way. Also the modifiers have no effect on the final calculation, so it is unclear why Understand removed one of the methods.

For NOC, although the three tools provide the same definition, different results were obtained for ten classes across tools (*Worker, SwingWorker, AlignAction, LocatorHandle, MoveAction, CompositeEdit, ZoomAction, ZoomEditorAction, AbstractFigureListener, PlacardScrollPaneLayout*). We were not able to identify the main reason for these differences. The values for those ten metrics were extracted manually and it was verified that the results extracted by our tool were correct. In fact from the ten classes, only LocationHandler has children, all other classes must have a value of zero for this metric. Table 4.12 shows the values extracted by each tool, and Table 4.13 shows each class and its children. Only classes inheriting from JHotDraw classes are shown.

Table 4.12. Metric values per tool

Name	Metrics 1.3.8	MASU	Understand	MQLMetrics as:		
				Metrics 1.3.8	MASU	Understand
org.jhotdraw.util.Worker	20	10	10	0	0	0
org.jhotdraw.draw.action.SwingWorker	11	6	6	0	0	0
org.jhotdraw.draw.action.AlignAction	6	6	6	0	0	0
org.jhotdraw.draw.LocatorHandle	6	6	6	5	5	5
org.jhotdraw.draw.action.MoveAction	4	4	4	0	0	0
org.jhotdraw.undo.CompositeEdit	4	2	2	0	0	0
org.jhotdraw.draw.action.ZoomAction	2	1	1	0	0	0
org.jhotdraw.draw.action.ZoomEditorAction	2	1	1	0	0	0
org.jhotdraw.draw.AbstractFigureListener	2	2	2	0	0	0
org.jhotdraw.gui.PlacardScrollPaneLayout	1	1	1	0	0	0

Table 4.13. JHotDraw classes with its corresponding children

Class	Children	Class	Children
AbstractApplication	DefaultAppletApplication, DefaultMDIApplication, DefaultOSXApplication, DefaultSDIApplication	ChopBoxConnector	ChopBezierConnector, ChopDiamondConnector, ChopEllipseConnector, ChopRoundRectConnector, ChopTriangleConnector, StickyChopConnector
AbstractApplicationAction	AboutAction, ClearRecentFilesAction, ExitAction, NewAction, OpenAction, OpenRecentAction, OSXDropOnDockAction	CreationTool	TextAreaTool, TextTool
AbstractAttributedComposit-eFigure	SVGPath	DefaultApplicationMo del	DrawApplicationModel, NetApplicationModel, PertApplicationModel, SVGApplicationModel
AbstractBean	AbstractApplication, DefaultApplicationModel, AbstractDrawing, DefaultDrawingEditor	DefaultDOMFactory	DrawFigureFactory, NetFactory, PertFactory, SVGFigureFactory
AbstractCompositeFigure	AbstractAttributedCompositeFigure, GraphicalCompositeFigure, GroupFigure	DefaultDrawing	SVGDrawing
AbstractConnector	ChopBoxConnector, LocatorConnector	DoubleStroke	GrowStroke
AbstractDrawing	DefaultDrawing, QuadTreeDrawing	EllipseFigure	SVGEllipse
AbstractEditorAction	ZoomEditorAction	GraphicalCompositeFi gure	ListFigure, TaskFigure
AbstractFigure	AbstractCompositeFigure, AttributedFigure	GroupAction	CombineAction
AbstractHandle	BezierControlPointHandle, BezierNodeHandle, BezierScaleHandle, ChangeConnectionHandle, LocatorHandle, RotateHandle, RoundRectRadiusHandle, TriangleRotationHandler	GroupFigure	SVGGroup
AbstractLayouter	HorizontalLayouter, VerticalLayouter	LineConnectionFigure	LabeledLineConnectionFigure, DependencyFigure
AbstractLineDecoration	ArrowTip, GeneralPathTip	LineFigure	LineConnectionFigure, SeparatorLineFigure, SVGLine
AbstractLocator	BezierPointLocator, RelativeLocator	***LocatorHandle***	CloseHandle, ConnectionHandle, FontSizeHandle, MoveHandle, NullHandle
AbstractProject	DrawProject, NetProject, PertProject, SVGProject	RectangleFigure	BorderRectangleFigure
AbstractProjectAction	ExportAction, MaximizeAction, MinimizeAction, ProjectPropertyAction, RedoAction, SaveAction, SaveBeforeAction, ToggleProjectPropertyAction, UndoAction	RelativeLocator	RelativeDecoratorLocator
AbstractSelectedAction	AlignAction, ApplyAttributesAction, AttributeAction, ColorChooserAction, DefaultAttributeAction, GroupAction, MoveAction, MoveToBackAction, MoveToFrontAction, PickAttributesAction, SelectSameAction, UngroupAction	RoundRectangleFigure	SVGRect
AbstractTool	BezierTool, BidirectionalConnectionTool, ConnectionTool, CreationTool, DragTracker, HandleTracker, SelectAreaTracker, SelectionTool	SaveAction	SaveAsAction
AbstractViewAction	ToggleGridAction, ZoomAction	SaveBeforeAction	ClearAction, CloseAction, LoadAction, LoadRecentAction
AttributedFigure	BezierFigure, DiamondFigure, EllipseFigure, RectangleFigure, RoundRectangleFigure, TextAreaFigure, TextFigure, TriangleFigure, SVGImage	SelectionTool	DelegationSelectionTool
BezierFigure	LineFigure	TextFigure	LabelFigure, NodeFigure, SVGText
BezierTool	PathTool	UngroupAction	SplitAction
ChangeConnectionHandle	ChangeConnectionEndHandle, ChangeConnectionStartHandle	XMLException	XMLParseException, XMLValidationException

As stated above, only LocationHandler has children and the correct value is 5, it is unclear how the tools calculated the values for these ten metrics. It can be argued that the tools are taking into consideration interfaces, inline classes, or Java specific constructs, such as anonymous classes, when constructing the inheritance tree, but no evidence that supports that was found.

For the metrics WMC (including V(G)) and LCOM, lower match percentages were obtained. Although the definitions on how those metrics are calculated are available, or can be inferred, the maximum match percentage is 90.4% when extracting the LCOM values as the Understand tool. As with the other metrics, an analysis and manual extraction for a sample of the non-matched metrics was performed and several inconsistencies were found across the results from each tool.

For V(G) (which is the base to calculate WMC) the main problem remains in the definition of the metric itself. The documentation of the tools is very vague and very general and does not provide full information. Analyzing the results and the source code, it was discovered that the try and catch structure was also considered, along with how the switch structure is handled. For Metrics 1.3.8 simplified if's used as parameters are also count, but structures such as `return a || b` are not counted in the final value although it is stated in the documentation that the && and || symbols are counted.

The most relevant inconsistency was with the LCOM results from Metrics 1.3.8, where over 50% of the values for the metric are zero. According to their calculation method, a

zero can only be obtained if every class variable is accessed in each method in the class, which is not the case for the classes with a value of zero. Two examples of miscalculated metrics by Understand and Metrics 1.3.8 are presented in Table 4.14. The definitions of the metrics are taken from Table 4.9.

Table 4.14. Miscalculated metrics per tool

Understand	Metrics 1.3.8
Class: Worker Class variables: 1 Class methods: 6 Number of methods accessing the variable: 2 Divide the number of methods that use a variable by the total of methods: 2 / 6 = 0.33333 Average the total of obtained values, the substract from 1 and multiply by 100: 0.33 / 1 = 0.33333 1 – 0.333333 = 0.66667 0.66667 * 100 = **66.667** **Value calculated by the tool: 50**	Class: Worker Class variables: 1 Class methods: 6 Number of methods accessing the variable: 2 Average of methods accessing each attribute: 2 / 1 = 2 subtract the number of methods and divide by 1-m 2 – 6 = -4 -4 / (1 – 6) = **0.8** **Value calculated by the tool: 0**

As mentioned above, results inconsistences across software metrics tools have been studied, and that issue, along with incorrect calculations and the lack of complete information on how the metrics are computed, makes it very difficult to obtain 100% matches for most of the metrics.

The analysis, discussions and results in this Chapter present evidence that new technology for code metrics extraction is needed. New methodologies and tools must provide the necessary means to define and extract metrics in a reliable way, as the methodology

presented in this thesis and its corresponding MQLMetrics tool, which proves to be more efficient that the current metrics tools .

In this study object oriented metrics were extracted, but the proposed methodology can be used for any other language given that the grammar is queried, so no additional constraints are presented.

4.4 Grammatical inference

The inference methodology was created and tested based on common current object oriented programming languages such as Java and C#. In addition to those two languages, the Objective-C and Swift languages were also chosen since their syntax is similar to Java, but with clear syntactical differences.

According to our tests, the inference process is able to infer productions for most of the instructions from a Java type language independently, but creating the overall context-free grammar proves to be complicated. It is important to note that it is not the objective of this thesis to present a methodology able to infer the complete grammar of any given language, we are mainly interested in instructions that are relevant for code metrics extractions, focusing especially in three types of structures: class declaration with inheritance, member variables declarations and methods declaration, including parameters, modifiers and data types.

Table 4.15 presents three code fragments from which their context-free grammar was inferred.

Table 4.15. Grammatical inference examples

<table>
<tr><th>Language</th><th>Code</th><th></th></tr>
<tr><td>Java/C# like</td><td>public class MyClass : Parent
{
private int a,b;
private int a;
private float a,b,c;

public void M1();
public int M2(int a, float b, int c);
private float M3(int a);
}</td><td>S → S45
S2 → [id type]
S8 → [private mod] [int type] S52
S15 → '(' ')'
S16 → [public mod] [void type] [id types] S53
S17 → [int type] [id type]
S21 → '(' S47 ')'
S23 → '(' [int type] [id type] ')'
S26 → nothing
S44 → '{' S49 '}'
S45 → [public mod] 'class' [id type] ':' [id type] S44
S47 → S47 ',' S50 | S50
S50 → S2 | S17
S49 → S49 ';' S51 | S51
S51 → S16 | S26 | S8
S52 → S47 | [id type]
S53 → S15 | S21 | S23</td></tr>
<tr><td>Swift like</td><td>class MyClass
{
var a, b : int;
var a : int;
var a, b, c : float;

M1() -> void;
M2(a : int, b : float, int : c) -> int;
M3(a : int) -> float;
}</td><td>S → S52
S1 → [id type]
S8 → 'var' S62
S20 → '(' ')'
S22 → [id type] S20 '->' [void type]
S28 → '(' S56 ')'
S29 → [id type] S28 '->' [int type]
S33 → nothing
S51 → '{' S58 '}'
S52 → 'class' [id type] S51
S54 → S54 ',' S59 | S59
S59 → S1
S56 → S56 ':' S60 | S60
S60 → S8 | S1 | S54
S58 → S58 ';' S61 | S61
S61 → S29 | S33 | S22 | S56 | S8
S62 → S54 | S56</td></tr>
<tr><td>Objective-C like</td><td>interface MyClass
{
property int a;
property int b;

(void)M1;
(int)M2:(int)a, B:(int)b, C:(int)c;
(float)M3:(int)a;
}</td><td>S → S36
S1 → [id types]
S5 → 'property' [int type] [id types]
S8 → '(' [void type] ')'
S9 → S8 S46
S21 → nothing
S35 → '{' S42 '}'
S36 → 'interface' [id type] S35
S38 → S43 ':' S43
S43 → S1 | S9
S40 → S40 ',' S44 | S44
S44 → S38
S42 → S42 ';' S45 | S45
S45 → S38 | S21 | S9 | S5
S46 → [id type] | S40</td></tr>
</table>

4.5 Research questions answers

Five research questions were proposed as part of the thesis protocol, and now the answer for each question is presented.

RQ1. How to create a methodology for source code metrics extraction without being dependent on the code metrics and programming languages it supports?

In order to achieve the desired independence of code metrics and programming languages, an intermediate representation of the source code has to be used, but strictly speaking any kind of intermediate representation is dependent of the programming language. A common structure for all programming languages that can be manipulated has to be the basis of the methodology. This structure is the context-free grammar that defines most, if not all, current languages. Furthermore, the context-free grammar can be automatically inferred to avoid the need of writing it.

By manipulating and creating links between the context-free grammar and the source code, the relevant data for the metrics calculation can be extracted. This is described overall in Chapter 3.

RQ2. Which mechanisms can be created to define new code metrics and programming languages?

Since it was stated in RQ1 that the context-free grammar of the languages is going to be used, the definition and incorporation of new programming languages is automatically supported by accepting new grammars. On the other hand, the definition of new code metrics will be achieved by querying the non-terminal symbols of the context-free

grammar of the language. The results of these queries are relevant substrings extracted from the source code. These substrings will be arranged hierarchically in a tree structure order to reproduce the structure of the object oriented code. This is described in Chapter 3.4.1.

The tree can be seen as an intermediate representation of the source code from which the metrics can be calculated.

RQ3. How to create a source code intermediate representation with all data needed to calculate the code metrics?

As mentioned in RQ2, a tree will be used as a source code intermediate representation containing the data needed to calculate the code metrics. This tree will be created by querying the context-free grammar. Each query is composed by two derivations (scope and attribute) created from the context-free grammar that are resolved in parsing time. Each derivation can be seen as a pointer to a specific non-terminal symbol in the body of a production. The scope derivation creates valid contexts in the form of placeholder nodes in the tree, and those nodes will be populated by the data of the attribute derivation. Two or more queries can be merged in order to create more complex data trees. This is described in Chapter 3.4.1.

Since every code metric might need different data from the source code in order to be calculated, the data model is created dynamically every time a query is executed, so all the necessary data is always present for the desired code metric.

RQ4. Which programming structures are needed to cover the source code measurement needs?

The needed programming structures can be narrowed down by analyzing the most important and most used object oriented metrics (Nuñez-Varela et al., 2017). The code metrics themselves are not as important to answer this question as the quality or design attributes they measure. Size, inheritance, coupling and cohesion are current widely studied object oriented attributes (Nuñez-Varela et al., 2017), and for those attributes the necessary programming structures are: classes definitions (with inheritance), member variables definitions and methods definitions. It is important to note that those programming structures contain other structures such as modifiers, initializers, parameters definitions, which are also taken in consideration. This is described in the Introduction and in Chapter 2 as part of the research process.

RQ5. How to infer the programming language context-free grammar or parts of it?

In order to infer a context-free grammar, or parts of it, a stack based parser is used. The parser finds patterns in each source code instruction. These patterns are represented by expressions, which are complete or incomplete productions that can be part of the final inferred grammar.

Several transformations are applied to the full set expressions and are transformed into an initial context-free grammar. After manipulating the initial grammar and matching similar productions, a final inferred grammar is created. This is described in Chapter 3.5.

4.6 Threats to validity

In this Chapter the threats of validity of the results are presented, including an analysis of the complexity of the methodologies.

4.6.1 Source code extraction methodology

It is stated that the proposed extraction methodology can access all data in the source code, given that the queries are created from the language grammar. This does not means that a grammar written in any form can access certain data as expected, but it can be re-written in order to access the required data. The complexity of writing a query is directly related to how the grammar is written. In the tool presented, the grammars for Java, AspectJ and XML were obtained from an online grammar repository, and overall, no more than ten productions had to be modified in order to extract and compute the supported metrics.

The code metrics presented in Appendix A are extracted following common definitions used in research papers (Mccabe, 1976, Li and Henry, 1993, Chidamber and Kemerer, 1994), which might not be the definition proposed by the original author for a certain metric. Our intention is to showcase the functionality of the methodology presented here and the overall metric definition process, not to focus on individual metrics. Also, even if only a set of representative metrics are supported (see Appendix A), given the functionality of the methodology new metrics can be defined or already supported metrics can be modified (i.e. the metric NOM counts the number of methods per class,

but it can include or exclude constructors or static methods, so this metric can be modified according to the measurement needs).

4.6.2 Grammatical inference

One of the main issues with grammatical inference is the distinction of positive and negative samples, where a positive sample must be accepted by the grammar and a negative do not. For this thesis we are considering all inputs to be positive samples, and it is possible for the grammar to accept negative samples as well since the proposed inference methodology creates expressions and generalize them (steps two and three in the expressions reduction process). Productions involving operators can accept negative examples since both left and right operands are merged during the inference process into single productions, so it is possible to accept inputs such as `a=3` and `3=a`. This can be solved by creating contexts in which certain symbols are valid but it is out of the scope of this thesis, as mentioned above, in this thesis all inputs are considered positive.

Another important issue is the fact that only binary operators are defined, so unary operators are not always parsed correctly since no rules are defined for them. Also, some symbols have different meaning according to the instruction, for example, the same symbol can act as a unary or binary operator.

The inference methodology was designed for object programming languages, especially for languages with a Java like syntax, and no further tests were made for other type of languages, but it is possible for the inference process to work as expected in untested languages.

Finally, since context-free grammars can accept an infinite number of strings generated from the language it defines, it cannot be claimed that a grammar, or the same grammar, can be generated for all inputs from the same language.

4.6.3 Complexity

Source code metrics extraction is complex by nature because it involves the use of custom parsers linked to a specific programming language, and the heavy use of context-free grammars and intermediate representations. Any attempt to generalize the extraction process through certain mechanisms will result in an even more complex solution, given that the proposed mechanisms focus on the manipulation of the language grammar, models or the parsers themselves. If these mechanisms were hidden from the end-user, in the form a software metrics tool for example, this complexity would be handled by researchers and developers only, but this is not the case. Since the end-user is the one who will define metrics and/or incorporate new languages, it is assumed that this user has intermediate to advance programming skills and is very knowledgeable in programming languages and context-free grammars.

Research efforts should be made in order to not only provide better extraction mechanisms, but also provide the means to make the process easier for the end-user.

4.7 Chapter summary

A software metrics tool named MQLMetrics was developed in order to obtain the results needed. The tool is based on MQL, and two real life examples of common metrics

extraction are presented using MQL queries. The characteristics of the tool are then described, including the code metrics and programming languages it supports out of the box. In order to test the correct functioning of the tool, the supported metrics were extracted from a set of benchmark Java systems.

A case study is presented in which the MQLMetrics tool extracts a set of metrics as defined by the tools Understand, Metrics 1.3.8 and MASU from the benchmark system JHotDraw. How the set of metrics, tools and system to measure were chosen are described in detail. The results obtained from the MQLMetrics tool are then compared against the other tools results, and even though it was not possible to achieve 100% matches for all metrics for each tool, the results are good as an expected. A discussion on why 100% matches were not achieved is presented.

Results from a grammatical inference case study are also presented. The context-free grammars from three different languages were inferred. Even though these languages are all for the object oriented paradigm, they present several notorious syntactical differences.

Finally, the answers to our research questions are presented, along with the threats of validity.

Conclusions

The area of source code metrics provides a large body of research for various Computer Science fields. Source code metrics were initially conceived for Software Engineering quality purposes, but nowadays they are used in fields such as Artificial Intelligence, Computer Security and Computer Programming in general. Code metrics research has had a continuous growth over the years and it looks to be a tendency for the forthcoming years, especially in areas such as concerns identification, big scale software and software product lines. This continuous growth, and the use of code metrics in new applications, is reflected in the creation and modification of code metrics by researchers.

While the object oriented paradigm is the most common measured programming paradigm, and most of the metrics tools and research is focused on that paradigm, we were able to identify software product lines as a current and future trend, and also that the procedural paradigm is still in use and is being measured. This programming paradigm gap, along with the current issues in code metrics extraction, represents several limitations and researchers opt to create their own tools in order to fulfill their measurement needs. The most relevant limitations are that current code metrics tools are limited by the number of metrics and languages it supports, and by the lack of mechanisms for metrics definition. Current methodologies for metrics extraction solve these issues halfway by creating models form the source code that, even if they allow the extraction and definition of metrics, are still limited by the information they contain and are bounded to a certain language. This thesis solves those issues by dynamically creating a data model with each query, so all the necessary information to compute the desired

metric is always present. Also, by querying the language grammar directly, a new language can be incorporated by providing its grammar, making it is also programming paradigm independent. It was tested defining a large quantity of diverse code metrics by querying complete programming language grammars. By querying the grammar we can access to all available data in the source code.

In this thesis we aim to further aid the metrics extraction process by partially inferring the context-free grammar of the input language. Grammatical inference is still an area in development, even though it has already been studied for years, given the complexity of the objectives it tries to achieve. Lately, attempts have been made in order to use grammatical inference for Software Engineering, where programming languages are very relevant, but context-free grammar inference for programming languages is still an unresolved problem. Even tough, researchers have been trying to solve this problem by limiting the domain of the problem, but it is still a very complex and time consuming process.

The inference methodology presented in this thesis also limits the domain of the problem by finding specific code patterns in object oriented structures by means of binary operators' analysis. But it was found that these types of patterns are common for all other code structures, and the solution provided could be escalated in order to infer a complete context-free grammar for a given language, but more considerations and tests have to be made. The proposed methodology represents a step forward in programming language inference given that it is fully automatic and infers the whole grammar, while current

inference techniques work on incomplete grammars or with the support of a human expert.

Future work

Software metrics tools, both commercial and derived from research are widely used. Although MQLMetrics fulfill the objectives of a metrics tools, it can be greatly improved by providing a graphical user interface, documentation and help files. It is also important to provide navigators and visualizer to help the user in what he is trying to achieve. For example, if the context-free grammar is presented to the user and he can navigate through the symbols, a query can be created automatically for that symbol, and a preview of the results can be presented immediately. With this kind of help, there will be a better understanding on how to query the desired data. This new MQLMetrics version could be distributed as an open source project, which will be very useful for researchers and practitioners.

It will be very important to include in the new version of the tool described above a grammatical inference module. This will help the user considerably since the grammar will no longer be a required input in order to accept a new language. For that, the grammatical inference methodology presented in this thesis has to be extended and thoroughly tested so it can effectively infer a full context-free grammar. As a first step we are aiming to infer the grammar of complete small source code, such as the code for bubblesort or quicksort. We can work then with medium size codes until being able to infer the complete

grammar. It is a complex and time consuming task, but it is something that can be achieved.

References

Aho, A., Sethi, R., Lam, M.S., Ullman, J.D., 2007. Compilers: principles, techniques, and tools, Reading, MA,.

Al Dallal, J., 2012. Fault prediction and the discriminative powers of connectivity-based object-oriented class cohesion metrics. Inf. Softw. Technol. 54, 396–416. doi:10.1016/j.infsof.2011.11.007

Alexan, N., Garem, R. El, Othman, H., 2016. An extendible open source tool measuring software metrics for indicating software quality, in: 2016 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA). IEEE, pp. 172–176. doi:10.1109/SPA.2016.7763607

Alghamdi, J.S., Rufai, R.A., Khan, S.M., 2005. OOMeter: A Software Quality Assurance Tool, in: Ninth European Conference on Software Maintenance and Reengineering. IEEE, pp. 190–191. doi:10.1109/CSMR.2005.44

Alikacem, E.H., Sahraoui, H. a., 2009. A Metric Extraction Framework Based on a High-Level Description Language, in: 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation. IEEE, pp. 159–167. doi:10.1109/SCAM.2009.27

Alikacem, E.H., Sahraoui, H.A., 2006. Generic Metric Extraction Framework, in: IWSM/MetriKon 2006.

Allier, S., Vaucher, S., Dufour, B., Sahraoui, H., 2010. Deriving coupling metrics from call graphs. Proc. - 10th IEEE Int. Work. Conf. Source Code Anal. Manip. SCAM 2010 43–52. doi:10.1109/SCAM.2010.25

Almugrin, S., Albattah, W., Melton, A., 2016. Using indirect coupling metrics to predict package maintainability and testability. J. Syst. Softw. 0, 1–13. doi:10.1016/j.jss.2016.02.024

Alshayeb, M., Shaaban, Y., Al-Ghamdi, J., 2018. SPMDL: Software ProductMetrics Definition Language. J. Data Inf. Qual. 9, 1–30. doi:10.1145/3185049

Aral, A., Ovatman, T., 2013. Utilization of Method Graphs to Measure Cohesion in Object Oriented Software, in: 2013 IEEE 37th Annual Computer Software and Applications Conference Workshops. IEEE, pp. 505–510. doi:10.1109/COMPSACW.2013.115

Arpaia, P., Bernardi, M.L., Di Lucca, G., Inglese, V., Spiezia, G., 2010. An Aspect-Oriented Programming-based approach to software development for fault detection in measurement systems. Comput. Stand. Interfaces 32, 141–152. doi:10.1016/j.csi.2009.11.009

Baroni, A.L., e Abreu, F.B., 2003. A Formal Library for Aiding Metrics Extraction. Int. Work. Object-Oriented Re-Engineering.

Bartoli, A., De Lorenzo, A., Medvet, E., Tarlao, F., 2016. Inference of Regular Expressions for Text Extraction from Examples. IEEE Trans. Knowl. Data Eng. 28, 1217–1230. doi:10.1109/TKDE.2016.2515587

Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., Palomba, F., 2015. An experimental investigation on the innate relationship between quality and refactoring. J. Syst. Softw. 107, 1–14. doi:10.1016/j.jss.2015.05.024

Budimac, Z., Rakic, G., Hericko, M., Gerlec, C., 2012. Towards the Better Software Metrics Tool, in: 2012 16th European Conference on Software Maintenance and Reengineering. IEEE, pp. 491–494. doi:10.1109/CSMR.2012.64

Cetinkaya, A., 2007. Regular expression generation through grammatical evolution, in: Proceedings of the 2007 GECCO Conference Companion on Genetic and Evolutionary Computation - GECCO '07. ACM Press, New York, New York, USA, p. 2643. doi:10.1145/1274000.1274089

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20, 476–493. doi:10.1109/32.295895

Cogan, B., Shalfeeva, E., 2002. A Generalized Structural Model of Structured Programs for Software Metrics Definition. Softw. Qual. J. 10, 149–167. doi:10.1023/A:1020575907785

Darcy, D.P., Kemerer, C.F., 2005. OO Metrics in Practice. IEEE Softw. 22, 17–19. doi:10.1109/MS.2005.160

Durisic, D., Nilsson, M., Staron, M., Hansson, J., 2013. Measuring the impact of changes to the complexity and coupling properties of automotive software systems. J. Syst. Softw. 86, 1275–1293. doi:10.1016/j.jss.2012.12.021

E, D., 2009. Analysis and Implementation of Software Metric for Object-Oriented, in: 2009 International Conference on Computational Intelligence and Software Engineering. IEEE, pp. 1–4. doi:10.1109/CISE.2009.5363987

Eichberg, M., Germanus, D., Mezini, M., Mrokon, L., Schafer, T., 2006. QScope: an open, extensible framework for measuring software projects, in: Conference on Software Maintenance and Reengineering (CSMR'06). IEEE, p. 10 pp.-pp.122. doi:10.1109/CSMR.2006.42

El-Wakil, M., El-Bastawisi, a, Riad, M., Fahmy, a, 2005. A novel approach to formalize object-oriented design metrics. Eval. Assess. Softw. Eng. 11–12.

Emanuelsson, P., Nilsson, U., 2008. A Comparative Study of Industrial Static Analysis Tools. Electron. Notes Theor. Comput. Sci. 217, 5–21. doi:10.1016/j.entcs.2008.06.039

Eski, S., Buzluca, F., 2011. An Empirical Study on Object-Oriented Metrics and Software Evolution in Order to Reduce Testing Costs by Predicting Change-Prone Classes, in: 2011 IEEE Fourth

International Conference on Software Testing, Verification and Validation Workshops. IEEE, pp. 566–571. doi:10.1109/ICSTW.2011.43

Etzkorn, L., Delugach, H., 2000. Towards a semantic metrics suite for object-oriented design. Proceedings. 34th Int. Conf. Technol. Object-Oriented Lang. Syst. - TOOLS 34 71–80. doi:10.1109/TOOLS.2000.868960

Fenton, N.E., Neil, M., 1999. Software metrics: successes, failures and new directions. J. Syst. Softw. 47, 149–157. doi:10.1016/S0164-1212(99)00035-7

Figueiredo, E., Garcia, A., Maia, M., Ferreira, G., Nunes, C., Whittle, J., 2011. On the impact of crosscutting concern projection on code measurement, in: Proceedings of the Tenth International Conference on Aspect-Oriented Software Development - AOSD '11. ACM Press, New York, New York, USA, pp. 81–92. doi:10.1145/1960275.1960287

Fioravanti, F., Nesi, P., 2000. A method and tool for assessing object-oriented projects and metrics management. J. Syst. Softw. 53, 111–136. doi:10.1016/S0164-1212(00)00050-9

Francese, R., Risi, M., Tortora, G., 2017. MetricAttitude++: Enhancing Polymetric Views with Information Retrieval, in: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). IEEE, pp. 368–371. doi:10.1109/ICPC.2017.15

Gómez, O., Oktaba, H., Piattini, M., García, F., 2008. A Systematic Review Measurement in Software Engineering: State-of-the-Art in Measures, in: International Conference on Software and Data Technologies (ICSOFT 2006). pp. 165–176. doi:10.1007/978-3-540-70621-2

Halstead, M.H., 1997. Elements of Software Science. Elsevier Science Inc., New York, NY, USA.

Hardesty, L., 2009. http://news.mit.edu/2009/explainer-pnp. Last accessed: November 2018.

Harmer, T.J., Wilkie, F.G., 2002. An extensible metrics extraction environment for object-oriented programming languages, in: Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation. IEEE Comput. Soc, pp. 26–35. doi:10.1109/SCAM.2002.1134102

Higo, Y., Saitoh, A., Yamada, G., Miyake, T., Kusumoto, S., Inoue, K., 2011. A Pluggable Tool for Measuring Software Metrics from Source Code, in: 2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement. IEEE, pp. 3–12. doi:10.1109/IWSM-MENSURA.2011.43

IEEE Std 610.12-1990, IEEE standard glossary of software engineering terminology. https://standards.ieee.org/findstds/standard/610.12-1990.html

ISO/IEC/IEEE 24765:2010 Systems and software engineering - Vocabulary. 2010.

Kwiatkowski, M., Verhoef, C., 2013. Recovering management information from source code. Sci. Comput. Program. 78, 1368–1406. doi:10.1016/j.scico.2012.07.016

Kulkarni, U.L., Kalshetty, Y.R., Arde, V.G., 2010. Validation of CK Metrics for Object Oriented Design Measurement, in: 2010 3rd International Conference on Emerging Trends in Engineering and Technology. IEEE, pp. 646–651. doi:10.1109/ICETET.2010.159

Lanza, M., Marinescu, R., 2006. Object-Oriented Metrics in Practice. Springer Berlin Heidelberg, Berlin, Heidelberg. doi:10.1007/3-540-39538-5

Li, W., Henry, S., 1993. Object-oriented metrics that predict maintainability. J. Syst. Softw. 23, 111–122. doi:10.1016/0164-1212(93)90077-B

Lincke, R., Lundberg, J., Löwe, W., 2008. Comparing software metrics tools, in: Proceedings of the 2008 International Symposium on Software Testing and Analysis - ISSTA '08. ACM Press, New York, New York, USA, pp. 131–142. doi:10.1145/1390630.1390648

Malhotra, R., Khanna, M., 2014. A new metric for predicting software change using gene expression programming, in: Proceedings of the 5th International Workshop on Emerging Trends in Software Metrics - WETSoM 2014. ACM Press, New York, New York, USA, pp. 8–14. doi:10.1145/2593868.2593870

Marinescu, C., Marinescu, R., Girba, T., 2005. Towards a Simplified Implementation of Object-Oriented Design Metrics, in: 11th IEEE International Software Metrics Symposium (METRICS'05). IEEE, pp. 11–11. doi:10.1109/METRICS.2005.48

McCabe, T.J., 1976. A Complexity Measure. IEEE Trans. Softw. Eng. 2, 308–320.

Misirli, A.T., Çağlayan, B., Miranskyy, A. V., Bener, A., Ruffolo, N., 2011. Different strokes for different folks, in: Proceeding of the 2nd International Workshop on Emerging Trends in Software Metrics - WETSoM '11. ACM Press, New York, New York, USA, pp. 45–51. doi:10.1145/1985374.1985386

Misra, S., Akman, I., Colomo-Palacios, R., 2012. Framework for evaluation and validation of software complexity measures. IET Softw. 6, 323–334. doi:10.1049/iet-sen.2011.0206

Mo, R., Cai, Y., Kazman, R., Xiao, L., Feng, Q., 2016. Decoupling Level: A New Metric for Architectural Maintenance Complexity, in: Proceedings of the 38th International Conference on Software Engineering - ICSE '16. ACM Press, New York, New York, USA, pp. 499–510. doi:10.1145/2884781.2884825

Moshtari, S., Sami, A., 2016. Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction, in: Proceedings of the 31st Annual ACM Symposium on Applied Computing - SAC '16. ACM Press, New York, New York, USA, pp. 1415–1421. doi:10.1145/2851613.2851777

Mshelia, Y.U., Apeh, S.T., Edoghogho, O., 2017. A comparative assessment of software metrics tools, in: 2017 International Conference on Computing Networking and Informatics (ICCNI). IEEE, pp. 1–9. doi:10.1109/ICCNI.2017.8123809

Novak, J., 2011. Comparison of Software Metrics Tools for .Net, in: 13th International Multiconference Information Society (IS), Collaboration, Software And Services In Information Society (CSS). pp. 231–234.

Nunez-Varela, A.S., Perez-Gonzalez, H.G., Martinez-Perez, F.E., Cuevas-Tello, J., 2016. Building a User Oriented Application for Generic Source Code Metrics Extraction from a Metrics Framework, in: 2016 4th International Conference in Software Engineering Research and Innovation (CONISOFT). IEEE, pp. 27–32. doi:10.1109/CONISOFT.2016.13

Núñez-Varela, A., Perez-Gonzalez, H.G., Cuevas-Tello, J.C., Soubervielle-Montalvo, C., 2013. A Methodology for Obtaining Universal Software Code Metrics. Procedia Technol. 7, 336–343. doi:10.1016/j.protcy.2013.04.042

Nuñez-Varela, A.S., Pérez-Gonzalez, H.G., Martínez-Perez, F.E., Soubervielle-Montalvo, C., 2017. Source code metrics: A systematic mapping study. J. Syst. Softw. 128, 164–197. doi:10.1016/j.jss.2017.03.044

Parthipan S, Senthil Velan S, Babu, C., 2014. Design level metrics to measure the complexity across versions of AO software, in: 2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies. IEEE, pp. 1708–1714. doi:10.1109/ICACCCT.2014.7019400

Radjenovic, D., Hericko, M., Torkar, R., Zivkovic, A., 2013. Software fault prediction metrics: A systematic literature review. Inf. Softw. Technol. 55, 1397–1418. doi:10.1016/j.infsof.2013.02.009

Raki, G., Budimac, Z., Bothe, K., 2010. Towards a Universal Software Metrics Tool Motivation, Process and a Prototype, in: ICSOFT 2010 - Proceedings of the Fifth International Conference on Software and Data Technologies. pp. 263–266.

Rakic, N., Rakic, G., Sukur, N., Budimac, Z., 2017. eCST to source code generation - An idea and perspectives, in: 2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). IEEE, pp. 570–575. doi:10.23919/MIPRO.2017.7973490

Schaefer, I., Bettini, L., Bono, V., Damiani, F., Tanzarella, N., 2010. Delta-Oriented Programming of Software Product Lines, in: Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp. 77–91. doi:10.1007/978-3-642-15579-6_6

Scotto, M., Sillitti, A., Succi, G., Vernazza, T., 2004. A relational approach to software metrics, in: Proceedings of the 2004 ACM Symposium on Applied Computing - SAC ’04. ACM Press, New York, New York, USA, p. 1536. doi:10.1145/967900.968207

Scotto, M., Sillitti, A., Succi, G., Vernazza, T., 2006. A non-invasive approach to product metrics collection. J. Syst. Archit. 52, 668–675. doi:10.1016/j.sysarc.2006.06.010

Sharma, M., Gill, N.S., Sikka, S., 2012. Survey of object-oriented metrics. ACM SIGSOFT Softw. Eng. Notes 37, 1–5. doi:10.1145/2382756.2382770

Shao, J., Wang, ingxu, 2003. A new measure of software complexity based on cognitive weights. Can. J. Electr. Comput. Eng. 28, 69–74. doi:10.1109/CJECE.2003.1532511

Sillitti, A., Succi, G., De Panfilis, S., 2006. Managing non-invasive measurement tools. J. Syst. Archit. 52, 676–683. doi:10.1016/j.sysarc.2006.06.011

Srivastava, V., 2014. Integration of Metric Tools for Software Testing: Including Maintainability Index Measurement. Int. J. Res. 1, 136–139.

Stevenson, A., Cordy, J.R., 2014. A survey of grammatical inference in software engineering. Sci. Comput. Program. 96, 444–459. doi:10.1016/j.scico.2014.05.008

Tahir, T., Jafar, A., 2011. A Systematic Review on Software Measurement Programs, in: 2011 Frontiers of Information Technology. IEEE, pp. 39–44. doi:10.1109/FIT.2011.15

Tempero, E., Ralph, P., 2017. A model for defining coupling metrics. Proc. - Asia-Pacific Softw. Eng. Conf. APSEC 145–152. doi:10.1109/APSEC.2016.030

Yamashita, A., Moonen, L., 2013. To what extent can maintenance problems be predicted by code smell detection? - An empirical study. Inf. Softw. Technol. 55, 2223–2242. doi:10.1016/j.infsof.2013.08.002

Appendix A – Source code metrics

Metric	Level	Type	Paradigm
File Package, Number of Imports	File	Size	OOP
Number of Classes (NCLASS), Number of Interfaces (NINT)	File, System, Package	Size	OOP
Number of Anonymous Classes, Number of Total Lines (TLOC), Number of Blank Lines (BLOC), Number of Single Comment Lines (LCOMM), Number of Multiple Comment Lines Blocks, Number of Total Multiple Comment Lines (LCOMM)	File, System	Size	OOP
Number of Packages in the System, Abstractness (A)	System	Size	OOP
Number of Independent Classes (NIC), Number of Hierarchies (NOH), Average Number of Ancestors (ANA), Number of Descendants (NOD), Number of Ancestors (NOAC), Reuse Ratio (RR), Specialization ratio (SR)	System	Inheritance	OOP
Number of Methods (NOM), Number of Attributes (NOA), Number of Public Attributes (NOPA), Number of Private Attributes (NOPRA), Number of Public Methods (NOPM), Number of Private Methods (NOPRM), Number of Static Attributes (NSF), Number of Static Methods (NSM), Number of Attributes and Methods (Size2), Data Access Metric (DAM), Number of Constructors (NCONS)	Class	Size	OOP
Depth of Inheritance Tree (DIT), Class to Leaf Depth (CLD), Number of Children (NOC), Number of Parents (NOP)	Class	Inheritance	OOP
Coupling Between Objects (CBO), Response for a Class (RFC)	Class	Coupling	OOP
Weighted Methods per Class (WMC)	Class	Complexity	OOP
Lack of Cohesion in Methods (LCOM), Tight Class Cohesion (TCC), Lack of Cohesion in Methods 1 (LCOM1), Lack of Cohesion in Methods 2 (LCOM2), Lack of Cohesion in Methods 5 (LCOM5)	Class	Cohesion	OOP
Cohesion (Coh)	Class	Cohesion	OOP
Number of Parameters (NPAR), Number of Local Variables (LVAR)	Method	Size	OOP
Cyclomatic Complexity (V(G)), Cognitive Weight	Method	Complexity	OOP
Number of Total Lines (TLOC), Number of Blank Lines (BLOC), Number of Single Comment Lines (LCOMM), Number of Multiple Comment Lines Blocks, Number of Total Multiple Comment Lines (LCOMM)	System	Size	AOP
Number of Aspects (NA), Number of Pointcuts (NP), Number of Advices (NADV), Number of Methods (NOM), Number of InterTypes Declarations (ITD)	System, Package	Size	AOP
Coupling Between Components (CBC), Crosscutting Degree of an Aspect (CDA)	Aspect	Coupling	AOP
Weighted Operations in Module (WOM)	Aspect	Complexity	AOP
Number of Primitives (NPP)	Pointcut	Size	AOP
Number of Tags, Number of Attributes, Average Number of Attributes, Number of Tags with Text Element, Maximum Nest	File	Size	XML

Appendix B - Code example

```
package org.package1;

public class Class1
{
      public static void M1C1(int p1, int p2){}
      public void M2C1(boolean b1)
      {
            private class Class4
            {
                        public void M1C4() {}
              }
      }

      class Class3
      {
            public void M1C3(boolean b1, int p1)
            {
                  new anonClass()
                  {
                        public static void M1Anon(int p1){}
                        private void M2Anon(int p1, int p2){}

                  };
            }
      }
}

public class Class2
{
      private void M1C2(int p1){}
      public void M2C2(){}
}
```

Appendix C - Java grammar

```
compilationUnit
   :  packageDeclaration? importDeclaration* typeDeclaration* EOF
   ;

packageDeclaration
   :  annotation* 'package' qualifiedName ';'
   ;

importqualifiedName
   : qualifiedName ('.' '*')?
   ;

importDeclaration
   :  'import' 'static'? importqualifiedName ';'
   ;

typeClassDeclaration
   :  classOrInterfaceModifier* classDeclaration
   ;

typeDeclaration
   :  typeClassDeclaration  //classOrInterfaceModifier* classDeclaration
   |  classOrInterfaceModifier* enumDeclaration
   |  classOrInterfaceModifier* interfaceDeclaration
   |  classOrInterfaceModifier* annotationTypeDeclaration
   |  ';'
   ;

modifier
   :  classOrInterfaceModifier
   |  (  'native'
      |  'synchronized'
      |  'transient'
      |  'volatile'
      )
   ;

classOrInterfaceModifier
   :  annotation
   |  (  'public'
      |  'protected'
      |  'private'
      |  'static'
      |  'abstract'
      |  'final'
      |  'strictfp'
      )
   ;

variableModifier
   :  'final'
   |  annotation
   ;

classDeclaration
   :  'class' identifier typeParameters?  ('extends' type)? ('implements' typeList)? classBody
```

```
   ;

typeParameters
   :  '<' typeParameter (',' typeParameter)* '>'
   ;

typeParameter
   :  Identifier ('extends' typeBound)?
   ;

typeBound
   :  type ('&' type)*
   ;

enumDeclaration
   :  ENUM Identifier ('implements' typeList)? '{' enumConstants? ','? enumBodyDeclarations? '}'
   ;

enumConstants
   :  enumConstant (',' enumConstant)*
   ;

enumConstant
   :  annotation* Identifier arguments? classBody?
   ;

enumBodyDeclarations
   :  ';' classBodyDeclaration*
   ;

interfaceDeclaration
   :  'interface' identifier typeParameters? ('extends' typeList)? interfaceBody
   ;

typeList
   :  type (',' type)*
   ;

classBody
   :  '{' classBodyDeclaration* '}'
   ;

interfaceBody
   :  '{' interfaceBodyDeclaration* '}'
   ;

classBodyDeclaration
   :  ';'
   |   'static'? block
   |    memberDeclaration
   ;

modifiers
   :  modifier*
   ;

memberVarsDeclaration
   :  modifiers fieldDeclaration
   ;
```

```
memberMethodsDeclaration
   :  modifiers methodDeclaration
   ;

memberClassDeclaration
   :  modifiers classDeclaration
   ;

memberDeclaration
   :  memberMethodsDeclaration
   |   modifier* genericMethodDeclaration
   |   memberVarsDeclaration
   |   modifier* constructorDeclaration
   |   modifier* genericConstructorDeclaration
   |   modifier* interfaceDeclaration
   |   modifier* annotationTypeDeclaration
   |   memberClassDeclaration //modifier* classDeclaration
   |   modifier* enumDeclaration
   ;

methodDeclarationReturnType
   :  (type | 'void')
   ;

methodDeclaration
   :  methodDeclarationReturnType identifier formalParameters ('[' ']')*
      ('throws' qualifiedNameList)?
      (   methodBody
      |   ';'
      )
   ;

identifier
   : Identifier
   ;

genericMethodDeclaration
   :  typeParameters methodDeclaration
   ;

constructorDeclaration
   :  identifier formalParameters ('throws' qualifiedNameList)?
      constructorBody
   ;

genericConstructorDeclaration
   :  typeParameters constructorDeclaration
   ;

fieldDeclaration
   :  type variableDeclarators ';'
   ;

interfaceBodyDeclaration
   :  modifier* interfaceMemberDeclaration
   |   ';'
   ;
```

```
interfaceMemberDeclaration
    :   constDeclaration
    |   interfaceMethodDeclaration
    |   genericInterfaceMethodDeclaration
    |   interfaceDeclaration
    |   annotationTypeDeclaration
    |   classDeclaration
    |   enumDeclaration
    ;

constDeclaration
    :   type constantDeclarator (',' constantDeclarator)* ';'
    ;

constantDeclarator
    :   Identifier ('[' ']')* '=' variableInitializer
    ;

interfaceMethodDeclaration
    :   (type|'void') Identifier formalParameters ('[' ']')*
        ('throws' qualifiedNameList)?
        ';'
    ;

genericInterfaceMethodDeclaration
    :   typeParameters interfaceMethodDeclaration
    ;

variableDeclarators
    :   variableDeclarator (',' variableDeclarator)*
    ;

variableDeclarator
    :   variableDeclaratorId ('=' variableInitializer)?
    ;

variableDeclaratorId
    :   identifier ('[' ']')*
    ;

variableInitializer
    :   arrayInitializer
    |   expression
    ;

arrayInitializer
    :   '{' (variableInitializer (',' variableInitializer)* (',')? )? '}'
    ;

enumConstantName
    :   Identifier
    ;

type
    :   classOrInterfaceType ('[' ']')*
    |   primitiveType ('[' ']')*
    ;

classOrInterfaceType
```

```
    :   identifier typeArguments? ('.' identifier typeArguments? )*
    ;

primitiveType
    :   'boolean'
    |   'char'
    |   'byte'
    |   'short'
    |   'int'
    |   'long'
    |   'float'
    |   'double'
    ;

typeArguments
    :   '<' typeArgument (',' typeArgument)* '>'
    ;

typeArgument
    :   type
    |   '?' (('extends' | 'super') type)?
    ;

qualifiedNameList
    :   qualifiedName (',' qualifiedName)*
    ;

formalParameters
    :   '(' formalParameterList? ')'
    ;

formalParameterList
    :   formalParameter (',' formalParameter)* (',' lastFormalParameter)?
    |   lastFormalParameter
    ;

formalParameter
    :   variableModifier* type variableDeclaratorId
    ;

lastFormalParameter
    :   variableModifier* type '...' variableDeclaratorId
    ;

methodBody
    :   block
    ;

constructorBody
    :   block
    ;

qualifiedName
    :   Identifier ('.' Identifier)*
    ;

literal
    :   IntegerLiteral
    |   FloatingPointLiteral
```

```
    |   CharacterLiteral
    |   StringLiteral
    |   BooleanLiteral
    |   'null'
    ;

annotation
    :   '@' annotationName ( '(' ( elementValuePairs | elementValue )? ')' )?
    ;

annotationName
    : qualifiedName
    ;

elementValuePairs
    :   elementValuePair (',' elementValuePair)*
    ;

elementValuePair
    :   Identifier '=' elementValue
    ;

elementValue
    :   expression
    |   annotation
    |   elementValueArrayInitializer
    ;

elementValueArrayInitializer
    :   '{' (elementValue (',' elementValue)*)? (',')? '}'
    ;

annotationTypeDeclaration
    :   '@' 'interface' Identifier annotationTypeBody
    ;

annotationTypeBody
    :   '{' (annotationTypeElementDeclaration)* '}'
    ;

annotationTypeElementDeclaration
    :   modifier* annotationTypeElementRest
    |   ';'
    ;

annotationTypeElementRest
    :   type annotationMethodOrConstantRest ';'
    |   classDeclaration ';'?
    |   interfaceDeclaration ';'?
    |   enumDeclaration ';'?
    |   annotationTypeDeclaration ';'?
    ;

annotationMethodOrConstantRest
    :   annotationMethodRest
    |   annotationConstantRest
    ;

annotationMethodRest
```

```
    :  Identifier '(' ')' defaultValue?
    ;

annotationConstantRest
    :  variableDeclarators
    ;

defaultValue
    :  'default' elementValue
    ;

block
    :  '{' blockStatement* '}'
    ;

blockStatement
    :  localVariableDeclarationStatement
    |  statement
    |  typeDeclaration
    ;

localVariableDeclarationStatement
    :  localVariableDeclaration ';'
    ;

localVariableDeclaration
    :  variableModifier* type variableDeclarators
    ;

controlStatement
    :  'if' parExpression statement ('else' statement)?
    |  'for' '(' forControl ')' statement
    |  'while' parExpression statement
    |  'do' statement 'while' parExpression ';'
    |  'switch' parExpression '{' switchBlockStatementGroup* switchLabel* '}'
    ;

statement
    :  block
    |  ASSERT expression (':' expression)? ';'
    |  controlStatement
    |  'try' block (catchClause+ finallyBlock? | finallyBlock)
    |  'try' resourceSpecification block catchClause* finallyBlock?
    |  'synchronized' parExpression block
    |  'return' expression? ';'
    |  'throw' expression ';'
    |  'break' Identifier? ';'
    |  'continue' Identifier? ';'
    |  ';'
    |  statementExpression ';'
    |  Identifier ':' statement
    ;

catchClause
    :  'catch' '(' variableModifier* catchType Identifier ')' block
    ;

catchType
    :  qualifiedName ('|' qualifiedName)*
```

```
    ;

finallyBlock
    :   'finally' block
    ;

resourceSpecification
    :   '(' resources ';'? ')'
    ;

resources
    :   resource (';' resource)*
    ;

resource
    :   variableModifier* classOrInterfaceType variableDeclaratorId '=' expression
    ;

switchBlockStatementGroup
    :   switchLabel+ blockStatement+
    ;

switchLabel
    :   'case' constantExpression ':'
    |   'case' enumConstantName ':'
    |   'default' ':'
    ;

forControl
    :   enhancedForControl
    |   forInit? ';' expression? ';' forUpdate?
    ;

forInit
    :   localVariableDeclaration
    |   expressionList
    ;

enhancedForControl
    :   variableModifier* type variableDeclaratorId ':' expression
    ;

forUpdate
    :   expressionList
    ;

parExpression
    :   '(' expression ')'
    ;

expressionList
    :   expression (',' expression)*
    ;

statementExpression
    :   expression
    ;

constantExpression
```

```
   :  expression
   ;

methodCall
   :
   ;

cons
   :
   ;

expression
   :  primary
   |  expression '.' identifier
   |  expression '.' 'this'
   |  expression '.' 'new' nonWildcardTypeArguments? innerCreator
   |  expression '.' 'super' superSuffix
   |  expression '.' explicitGenericInvocation
   |  expression cons '[' expression ']'
   |  expression methodCall '(' expressionList? ')'
   |  'new' creator
   |  '(' type ')' expression
   |  expression ('++' | '--')
   |  ('+'|'-'|'++'|'--') expression
   |  ('~'|'!') expression
   |  expression ('*'|'/'|'%') expression
   |  expression ('+'|'-') expression
   |  expression ('<' '<' | '>' '>' '>' | '>' '>') expression
   |  expression ('<=' | '>=' | '>' | '<') expression
   |  expression 'instanceof' type
   |  expression ('==' | '!=') expression
   |  expression '&' expression
   |  expression '^' expression
   |  expression '|' expression
   |  expression ('&&' | '||') condOps expression
  // |   expression '||' expression
   |  expression '?' expression ':' expression simpIf
   |  <assoc=right> expression
      (   '='
      |   '+='
      |   '-='
      |   '*='
      |   '/='
      |   '&='
      |   '|='
      |   '^='
      |   '>>='
      |   '>>>='
      |   '<<='
      |   '%='
      )
      expression
   ;

simpIf
   :
   ;

condOps
```

```
    :
    ;

primary
    :   '(' expression ')'
    |   'this'
    |   'super'
    |   literal
    |   identifier
    |   type '.' 'class'
    |   'void' '.' 'class'
    |   nonWildcardTypeArguments (explicitGenericInvocationSuffix | 'this' arguments)
    ;

creator
    :   nonWildcardTypeArguments createdName classCreatorRest
    |   createdName (arrayCreatorRest | classCreatorRest)
    ;

createdName
    :   Identifier typeArgumentsOrDiamond? ('.' Identifier typeArgumentsOrDiamond?)*
    |   primitiveType
    ;

innerCreator
    :   Identifier nonWildcardTypeArgumentsOrDiamond? classCreatorRest
    ;

arrayCreatorRest
    :   '['
        (   ']' ('[' ']')* arrayInitializer
        |   expression ']' ('[' expression ']')* ('[' ']')*
        )
    ;

anonymousClassBody
    : classBody
    ;

classCreatorRest
    :   arguments anonymousClassBody?
    ;

explicitGenericInvocation
    :   nonWildcardTypeArguments explicitGenericInvocationSuffix
    ;

nonWildcardTypeArguments
    :   '<' typeList '>'
    ;

typeArgumentsOrDiamond
    :   '<' '>'
    |   typeArguments
    ;

nonWildcardTypeArgumentsOrDiamond
    :   '<' '>'
    |   nonWildcardTypeArguments
```

```
    ;

superSuffix
    :   arguments
    |   '.' Identifier arguments?
    ;

explicitGenericInvocationSuffix
    :   'super' superSuffix
    |   Identifier arguments
    ;

arguments
    :   '(' expressionList? ')'
    ;

ABSTRACT      : 'abstract';
ASSERT        : 'assert';
BOOLEAN       : 'boolean';
BREAK         : 'break';
BYTE          : 'byte';
CASE          : 'case';
CATCH         : 'catch';
CHAR          : 'char';
CLASS         : 'class';
CONST         : 'const';
CONTINUE      : 'continue';
DEFAULT       : 'default';
DO            : 'do';
DOUBLE        : 'double';
ELSE          : 'else';
ENUM          : 'enum';
EXTENDS       : 'extends';
FINAL         : 'final';
FINALLY       : 'finally';
FLOAT         : 'float';
FOR           : 'for';
IF            : 'if';
GOTO          : 'goto';
IMPLEMENTS    : 'implements';
IMPORT        : 'import';
INSTANCEOF    : 'instanceof';
INT           : 'int';
INTERFACE     : 'interface';
LONG          : 'long';
NATIVE        : 'native';
NEW           : 'new';
PACKAGE       : 'package';
PRIVATE       : 'private';
PROTECTED     : 'protected';
PUBLIC        : 'public';
RETURN        : 'return';
SHORT         : 'short';
STATIC        : 'static';
STRICTFP      : 'strictfp';
SUPER         : 'super';
SWITCH        : 'switch';
SYNCHRONIZED  : 'synchronized';
THIS          : 'this';
```

```
THROW        : 'throw';
THROWS       : 'throws';
TRANSIENT    : 'transient';
TRY          : 'try';
VOID         : 'void';
VOLATILE     : 'volatile';
WHILE        : 'while';

IntegerLiteral
   :  DecimalIntegerLiteral
   |  HexIntegerLiteral
   |  OctalIntegerLiteral
   |  BinaryIntegerLiteral
   ;

fragment
DecimalIntegerLiteral
   :  DecimalNumeral IntegerTypeSuffix?
   ;

fragment
HexIntegerLiteral
   :  HexNumeral IntegerTypeSuffix?
   ;

fragment
OctalIntegerLiteral
   :  OctalNumeral IntegerTypeSuffix?
   ;

fragment
BinaryIntegerLiteral
   :  BinaryNumeral IntegerTypeSuffix?
   ;

fragment
IntegerTypeSuffix
   :  [lL]
   ;

fragment
DecimalNumeral
   :  '0'
   |  NonZeroDigit (Digits? | Underscores Digits)
   ;

fragment
Digits
   :  Digit (DigitOrUnderscore* Digit)?
   ;

fragment
Digit
   :  '0'
   |  NonZeroDigit
   ;

fragment
NonZeroDigit
```

```
    :  [1-9]
    ;

fragment
DigitOrUnderscore
    :  Digit
    |  '_'
    ;

fragment
Underscores
    :  '_'+
    ;

fragment
HexNumeral
    :  '0' [xX] HexDigits
    ;

fragment
HexDigits
    :  HexDigit (HexDigitOrUnderscore* HexDigit)?
    ;

fragment
HexDigit
    :  [0-9a-fA-F]
    ;

fragment
HexDigitOrUnderscore
    :  HexDigit
    |  '_'
    ;

fragment
OctalNumeral
    :  '0' Underscores? OctalDigits
    ;

fragment
OctalDigits
    :  OctalDigit (OctalDigitOrUnderscore* OctalDigit)?
    ;

fragment
OctalDigit
    :  [0-7]
    ;

fragment
OctalDigitOrUnderscore
    :  OctalDigit
    |  '_'
    ;

fragment
BinaryNumeral
    :  '0' [bB] BinaryDigits
```

```
   ;

fragment
BinaryDigits
   :  BinaryDigit (BinaryDigitOrUnderscore* BinaryDigit)?
   ;

fragment
BinaryDigit
   :  [01]
   ;

fragment
BinaryDigitOrUnderscore
   :  BinaryDigit
   |  '_'
   ;

FloatingPointLiteral
   :  DecimalFloatingPointLiteral
   |  HexadecimalFloatingPointLiteral
   ;

fragment
DecimalFloatingPointLiteral
   :  Digits '.' Digits? ExponentPart? FloatTypeSuffix?
   |  '.' Digits ExponentPart? FloatTypeSuffix?
   |  Digits ExponentPart FloatTypeSuffix?
   |  Digits FloatTypeSuffix
   ;

fragment
ExponentPart
   :  ExponentIndicator SignedInteger
   ;

fragment
ExponentIndicator
   :  [eE]
   ;

fragment
SignedInteger
   :  Sign? Digits
   ;

fragment
Sign
   :  [+-]
   ;

fragment
FloatTypeSuffix
   :  [fFdD]
   ;

fragment
HexadecimalFloatingPointLiteral
   :  HexSignificand BinaryExponent FloatTypeSuffix?
```

```
    ;

fragment
HexSignificand
    :   HexNumeral '.'?
    |   '0' [xX] HexDigits? '.' HexDigits
    ;

fragment
BinaryExponent
    :   BinaryExponentIndicator SignedInteger
    ;

fragment
BinaryExponentIndicator
    :   [pP]
    ;

BooleanLiteral
    :   'true'
    |   'false'
    ;

CharacterLiteral
    :   '\'' SingleCharacter '\''
    |   '\'' EscapeSequence '\''
    ;

fragment
SingleCharacter
    :   ~['\\]
    ;

StringLiteral
    :   '"' StringCharacters? '"'
    ;

fragment
StringCharacters
    :   StringCharacter+
    ;

fragment
StringCharacter
    :   ~["\\]
    |   EscapeSequence
    ;

fragment
EscapeSequence
    :   '\\' [btnfr"'\\]
    |   OctalEscape
    |   UnicodeEscape
    ;

fragment
OctalEscape
    :   '\\' OctalDigit
    |   '\\' OctalDigit OctalDigit
```

```
    |   '\\' ZeroToThree OctalDigit OctalDigit
    ;

fragment
UnicodeEscape
    :   '\\' 'u' HexDigit HexDigit HexDigit HexDigit
    ;

fragment
ZeroToThree
    :   [0-3]
    ;

NullLiteral
    :   'null'
    ;

LPAREN          : '(';
RPAREN          : ')';
LBRACE          : '{';
RBRACE          : '}';
LBRACK          : '[';
RBRACK          : ']';
SEMI            : ';';
COMMA           : ',';
DOT             : '.';

ASSIGN          : '=';
GT              : '>';
LT              : '<';
BANG            : '!';
TILDE           : '~';
QUESTION        : '?';
COLON           : ':';
EQUAL           : '==';
LE              : '<=';
GE              : '>=';
NOTEQUAL        : '!=';
AND             : '&&';
OR              : '||';
INC             : '++';
DEC             : '--';
ADD             : '+';
SUB             : '-';
MUL             : '*';
DIV             : '/';
BITAND          : '&';
BITOR           : '|';
CARET           : '^';
MOD             : '%';

ADD_ASSIGN      : '+=';
SUB_ASSIGN      : '-=';
MUL_ASSIGN      : '*=';
DIV_ASSIGN      : '/=';
AND_ASSIGN      : '&=';
OR_ASSIGN       : '|=';
XOR_ASSIGN      : '^=';
MOD_ASSIGN      : '%=';
```

```
LSHIFT_ASSIGN   : '<<=';
RSHIFT_ASSIGN   : '>>=';
URSHIFT_ASSIGN  : '>>>=';

Identifier
   :  JavaLetter JavaLetterOrDigit*
   ;

fragment
JavaLetter
   :  [a-zA-Z$_]
   |  ~[\u0000-\u00FF\uD800-\uDBFF]
      {Character.isJavaIdentifierStart(_input.LA(-1))}?
   |  [\uD800-\uDBFF] [\uDC00-\uDFFF]
      {Character.isJavaIdentifierStart(Character.toCodePoint((char)_input.LA(-2), (char)_input.LA(-1)))}?
   ;

fragment
JavaLetterOrDigit
   :  [a-zA-Z0-9$_]
   |  ~[\u0000-\u00FF\uD800-\uDBFF]
      {Character.isJavaIdentifierPart(_input.LA(-1))}?
   |  [\uD800-\uDBFF] [\uDC00-\uDFFF]
      {Character.isJavaIdentifierPart(Character.toCodePoint((char)_input.LA(-2), (char)_input.LA(-1)))}?
   ;

AT
   : '@'
   ;

ELLIPSIS
   : '...'
   ;

WS
   :  [ \t\r\n\u000C]+ -> skip
   ;

COMMENT
   :  '/*' .*? '*/' -> skip
   ;

LINE_COMMENT
   :  '//' ~[\r\n]* -> skip
   ;
```