Casa abierta al tiempo
UNIVERSIDAD AUTÓNOMA METROPOLITANA

PC y TI

UNIVERSIDAD AUTÓNOMA METROPOLITANA – IZTAPALAPA
DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍA

COMPARING OBJECT-ORIENTED DESIGN BETWEEN JAVA AND C++ USING CHIDAMBER AND KEMERER METRICS

Thesis presented by:
Ramón Urias Morales
To obtain the degree in:
Master in Sciences and Information Technologies

Advisors:	Dr. Humberto Cervantes Maceda
	Dr. Apostolis Ampatzoglou

[illegible]

Jury qualifier:	
President:	MSc. Wolfgang Engelen Mora.
Secretary:	Dr. René MacKinney Romero
Vocal:	Dr. Humberto Cervantes Maceda

[illegible] [illegible]

México, D.F. March 2015

Acknowledgements

Special thanks to my advisors Dr. Humberto Cervantes and Dr. Apostolis Ampatzoglou for their patience, time and all their assistance provided during this thesis project.

Special thanks to Dr. Paris Avgeriou, who was the person that opened me the doors to this project.

Thanks to my reviewers of thesis M.Sc. Wolfgang Engelen and Dr. René MacKinney for their time and valuable observations during the last stage of this thesis project.

Thanks to all my schoolfellows of the Master in Sciences and Information Technologies of the Autonomous Metropolitan University who shared with me a lot of experiences and knowledge.

Thanks to all my professors of the Autonomous Metropolitan University who shared with me their valuable knowledge in the field of the information technologies.

Thanks to the National Council of Science and Technology (CONACYT).

Thanks to the Autonomous Metropolitan University (UAM-I).

Thanks to the University of Groningen.

Thanks to my weightlifting couch Oscar Gutiérrez, who always help me and teach me to be disciplined.

Thanks to all my friends of the weightlifting team from UAM-I.

Special thanks to my mother Hortensia Morales due to all the support given during my entire academic life.

Abstract

Software quality can be measured using models such as the ones proposed by McCall, Boehm or ISO. These models decompose quality in a set of quality attributes that are of interest to the users or to the developers. Among the quality attributes that are interesting to developers, we find maintainability which is understood in this thesis as a quality attribute that is associated to making changes in the system's source code. Measuring maintainability can be achieved through the use of design quality metrics such as the Chidamber and Kemerer metrics set. In this thesis we want to understand if the use of different object oriented languages (in our case Java and C++) has an impact on maintainability. To answer this question, we build Maya C++, a tool that calculates Chidamber and Kemerer metrics set for C++ programs. With the use of this tool we obtained metrics data from a project sample and this data was then used as a basis for comparison with data from Java projects. An analysis on the data reveals that there is not a significant difference in the use of the languages even though the data slightly favors Java.

.

Keywords: Software quality, object-oriented design, software metrics.

Resúmen

La calidad del software puede medirse utilizando modelos como los propuestos por McCall, Boehm o ISO. Estos modelos descomponen la calidad en un conjunto de atributos de calidad que son de interés para los usuarios o para los desarrolladores. Entre los atributos de calidad que son de interés para los desarrolladores, encontramos la mantenibilidad, que se entiende en esta tesis como un atributo de calidad que está asociado a hacer cambios en el código fuente de un sistema. La medición de la Mantenibilidad se puede lograr mediante el uso de métricas de calidad de diseño, como el conjunto de métricas de Chidamber y Kemerer. En esta tesis queremos comprender si el uso de diferentes lenguajes orientados a objetos (en nuestro caso, Java y C ++) tiene un impacto en la mantenibilidad. Para responder a esta pregunta, desarrollamos Maya C ++, una herramienta que calcula el conjunto de métricas de Chidamber y Kemerer para los programas en C ++. Con el uso de esta herramienta se obtuvieron datos de dichas métricas sobre una muestra de proyectos C++, posteriormente estos datos se utilizaron como base para la comparación con datos de proyectos Java. Un análisis de los datos revela que no hay una diferencia significativa en el uso de estos lenguajes, aunque estos datos favorecen ligeramente a Java.

Palabras clave: Calidad de software, diseño orientado a objetos, métricas de software.

Content

Contents

Chapter 1 Introduction 7

1.1 Introduction 7

1.2 Objectives 9

1.3 Methodology 10

1.4 Organization of work 11

Chapter 2 Background 12

2.1 Introduction 12

2.2 Software Quality and Quality Models 12

2.2.1 McCall Quality Model 14

2.2.2 Boehm Quality Model 16

2.2.3 ISO 9126 17

2.3 Maintainability related metrics 19

2.3.1 Measuring design properties to evaluate maintainability 20

2.3.2 Chidamber and Kemerer metrics 21

2.4 Software tools to evaluate software code 29

2.4.1 NDepend 30

2.4.2 VisualCppDepend 31

2.4.3 Understand for C++ 1.4 31

2.4.4 SourceMonitor 32

2.4.5 CCCC 33

2.4.6 Analysis on the software tools that evaluate software code 34

2.5 Summary 35

Chapter 3. Maya C++ tool requirements and design 37

3.1 Introduction 37

3.2 Vision and scope 37

3.3 Software requirements 38

3.3.1 Primary Use Case 38

3.3.2 Quality attributes scenario 40

3.3.3 Constraints 40

3.4 System design 40
3.4.1 Logical view 41
3.4.2 Dynamic view 43
3.4.3 User interface 44
3.5 Summary 45
Chapter 4. Evaluation 46
4.1 Introduction 46
4.1.1 Objectives 46
4.1.2 Research questions 46
- What programming language gives projects with better design quality with respect to Chidamber and Kemerer metrics? 46
- What metrics favor Java? 46
- What metrics favor C++? 47
- What programming language is more complied with our suggested ranges for the metrics? 47
4.2.1 Project sample 47
4.2.2 Data collection and analysis 47
4.3 Results 48
4.3.1 Results according to percentage of classes whose metrics are outside the suggested ranges 49
4.3.2 Results according to average of values of classes for each metric 56
4.4 General analysis 64
4.5 Summary 66
Chapter 5. Conclusion 67
5.1 Introduction 67
5.2 Conclusions 67
5.3 Future work 68
References 70

Chapter 1 Introduction

1.1 Introduction

In today's world, software quality has become increasingly relevant in the software industry. Software quality appears as a subjective concept that can be observed from different points of view, so what we consider as being of good quality can vary depending on the point of view. These different points of view are categorized in the five perspectives of quality proposed by David Garvin, a professor of Business Administration at the Harvard Business School: The transcendental view, the user view, the manufacturing view, the product view and the value-based view [1].

The perspectives such as the user view and the manufacturing view examine software quality from an external perspective; however, software quality seen from the product view is examined from an internal perspective [2]. Users tend to be more concerned by the external aspects, or quality attributes, such as performance or usability. On the other hand, developers tend to be more concerned by the internal aspects or quality attributes such as maintainability or testability. In this project, our interest is on one of these internal quality attributes: maintainability. The reason is because higher quality code, anticipating changes, better tuning to user needs and less code (which are all related to maintainability) result in less maintenance [3]. Furthermore, talking about total costs of a software system over its lifetime, maintenance alone consumes 50--75% of these costs [3].

In the context of this thesis, maintainability is understood as a quality attribute that is associated with making changes in the system's source code. Maintainability of software code is important because this attribute is inversely related to the costs of maintaining the software [4].

The international standard ISO 9126 [5], define sets of quality attributes (called characteristics), which represent product quality from an external or an internal point of view [6]. Since these attributes are not directly measurable, authors like McCall and Boehm have proposed hierarchical models [7] that relate these software attributes to software metrics. Relating software metrics to software quality attributes allows software quality to be evaluated quantitatively, and thus transform this apparently subjective concept into something that is objective. According to some software quality models [7, 8], maintainability is considered as a high level quality attribute, which is not directly measurable but it can be related to software metrics that are quantitatively measurable, and evaluate the design of software source code.

Chidamber and Kemerer metrics are one of the most recognized set of metrics created to evaluate the quality of design by analyzing source code of object oriented software [9, 10]. Since good quality design tends to make code more maintainable, these metrics can be used for measuring maintainability from the product point of view.

In this project we are interested in using the the Chidamber and Kemerer metrics set [11] to evaluate code design quality using different object oriented languages. More specifically, we are interested in understanding if the use of Java or C++, two popular programming languages, makes a difference with respect to the Chidamber and Kemerer metrics set. We expect to see projects written in one of these languages with better design quality than the other one, maybe it could be Java because it is a more recent language. This knowledge is important for the developers in the case they are starting a new project where they can choose the programming language.

We selected the Chidamber and Kemerer metrics set because data for these metrics is available for a huge number of Java projects in the website percerons.com [12, 13]. This data provides a starting point for our analysis. Unfortunately, there is no equivalent for C++ projects, so this data must be obtained in order to test our hypothesis. Since Chidamber and Kemerer metrics are calculated by analyzing source code, the manual calculation of this data is not possible. For this reason, it is necessary to use tools such as “Understand for C++” [14] or “CKJM” [15] which automates the calculation of these metrics for C++.

An initial analysis of the tools (see section 2.4) revealed that there are no open source tools that support the complete Chidamber and Kemerer metrics set. As a consequence, testing our hypothesis requires the development of a tool that calculates Chidamber and Kemerer metrics on C++ code.

1.2 Objectives

Based on the previous discussion, a general objective for this thesis is the following:

- To investigate if the use of Java or C++ contributes in the maintainability of programs as measured using Chidamber and Kemerer Metrics.

This general objective can be refined into a set of specific objectives which are the following:

- To understand the general concept of software quality and how the Chidamber and Kemerer metrics suite can be used to support the measurement of maintainability.
- To conduct a study of existing software tools that support the automatic calculation of Chidamber and Kemerer metrics for C++.
- To develop a tool that calculates the Chidamber and Kemerer metrics for C++ programs.
- To perform an analysis of a sample of Java and C++ programs with respect to the Chidamber and Kemerer metrics.

1.3 Methodology

To attain the specific and general objectives, the following methodology was established. First of all, we surveyed the literature and identified several quality models which were studied and compared. We then studied the Chidamber and Kemerer metrics suite in more detail.

After that, we conducted a study of available tools to evaluate if any of the available tools could be used for our study. We were particularly interested in studying tools that covered the complete Chidamber and Kemerer metrics set and whose code was available to facilitate their modification, if needed.

Since no tool was found to be suitable, we proceeded to develop our own tool which we plan to release as open source software in the future.

Once the development of the new tool was finished, we conducted a case study based on [32]. We used our tool to gather data for C++ programs. This involved making an extensive search of open source project across different domains to find a suitable sample. Once the projects were identified, we collected and analyzed the data. For the collection of data, we used data generated with our tool and data already available for Java projects, doing a normalized analysis and non-normalized analysis.

Finally, we proceeded to draw conclusions on the basis of our analysis.

1.4 Organization of work

This thesis is divided into 5 chapters. Following this introduction, Chapter 2 discusses general aspects of software quality to introduce the subject. It also discusses code design metrics and presents details about the Chidamber and Kemerer set of metrics. In this chapter, we also present a study of existing tools to calculate Chidamber and Kemerer metrics for C++. Chapter 3 discusses the requirements and design of the software tool that was developed as part of this thesis. In Chapter 4, we present the experimentation that was conducted using the tool that was developed. The data that was gathered is analyzed. Finally in the fifth and final chapter, we find the conclusions and a brief speculation on future work from this thesis.

Chapter 2 Background

2.1 Introduction

This chapter describes the concepts that are used as background for the objectives described in this thesis. These concepts include software quality, quality models and maintainability metrics. This chapter also includes an evaluation of several tools that are used for analyzing source code with respect to metrics associated with maintainability.

2.2 Software Quality and Quality Models

There are several definitions of quality, such as:

Quality is defined as the difference between the level or nature of service or product that the customer expects and the level or nature that the customer perceives [16].

Another definition for quality is given by the ISO 8402 model, which defines quality as:

The totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs [9].

Quality can be understood as a subjective term, different people can have different opinions on the quality of a certain product or service. Their opinion will depend on what characteristics from the product or service they take into account. This means that for measuring quality on any product or service we need to figure out the characteristics that constitute it, and then select metrics that allow each of these characteristics to be measured. The result of this decomposition is a quality model that is structured as a hierarchy of quality characteristics (or quality attributes) which are further decomposed into a set of metrics. Quality models allow turning something that is subjective, which is quality, into something objective and measurable.

Figure 2.1 shows a simple example of a hierarchical quality model. We will see concrete examples of such models in the following sections.

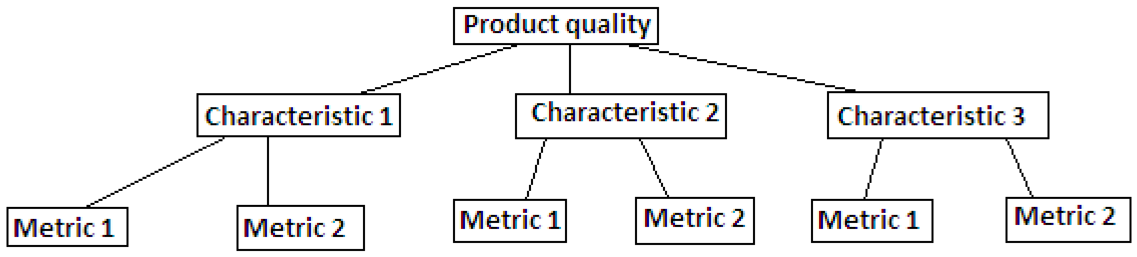


Figure 2.1 Example of a hierarchical model of product quality

David Gamin studied how quality is perceived in various domains and concluded that “quality is a complex and multifaceted concept” that can be described from five different perspectives [2]:

- The transcendental view, which sees quality as something that can be recognized but not defined.
- The user’s view, which sees quality according to the product characteristics that meet the user needs.
- The manufacturing view, which sees quality according to the manufacturing approach (ISO 9001 and CMM), focusing on the product quality before the production and after the delivery, which means that this perspective advocates conformance to process rather than to the specification.
- The value-based view, which deals with matters related to money. Purchasers of the product compare product potential benefits with the product cost.

- The product view, which examines the quality according to some properties like those proposed by the standard ISO/IEC 9126 (see section 2.2.3).

The initial attempts of development of methods to evaluate software quality seem to have appeared in 1968, with Rubey and Hartwick [17], who defined a method consisting of attributes and their metrics. Some examples of attributes they established include “mathematical operations that are correctly performed”, “the program is intelligible”, and “the program is easy to modify”. Such attributes were further analyzed to define attributes capable of being directly measurable in a scale between 0 and 100.

Several models that link quality attributes and their metrics to evaluate software quality were defined later. Examples of these models are McCall Quality Model [18], Boehm Quality Model [18], and ISO 9126 [5]. These models are discussed in the following sections.

2.2.1 McCall Quality Model

Jim McCall proposed a model in 1977 at the request of the US Air Force, with the intention of bridging the gap between users and developers, mapping the user view with the priorities of the developers [7].

The quality model proposed by McCall (Figure 2.2) divides the software product into 3 perspectives: product operations, product revision, and product transition [19].

- Product operations perspective groups together quality attributes that have to do with the degree in which the software fulfills its specifications.
- Product revision perspective groups together quality attributes that have to do with supporting changes in the product.

- Product transition perspective groups together quality attributes that have to do with influencing the ease of adaptation of the software to new environments.

These three perspectives in turn are divided into quality attributes, and in the same way, attributes are divided into metrics (which can be measured). In total McCall identified 11 quality attributes for the 3 perspectives of the software product. In this way it is possible to have an overall quality assessment by evaluating the metrics for each attribute.

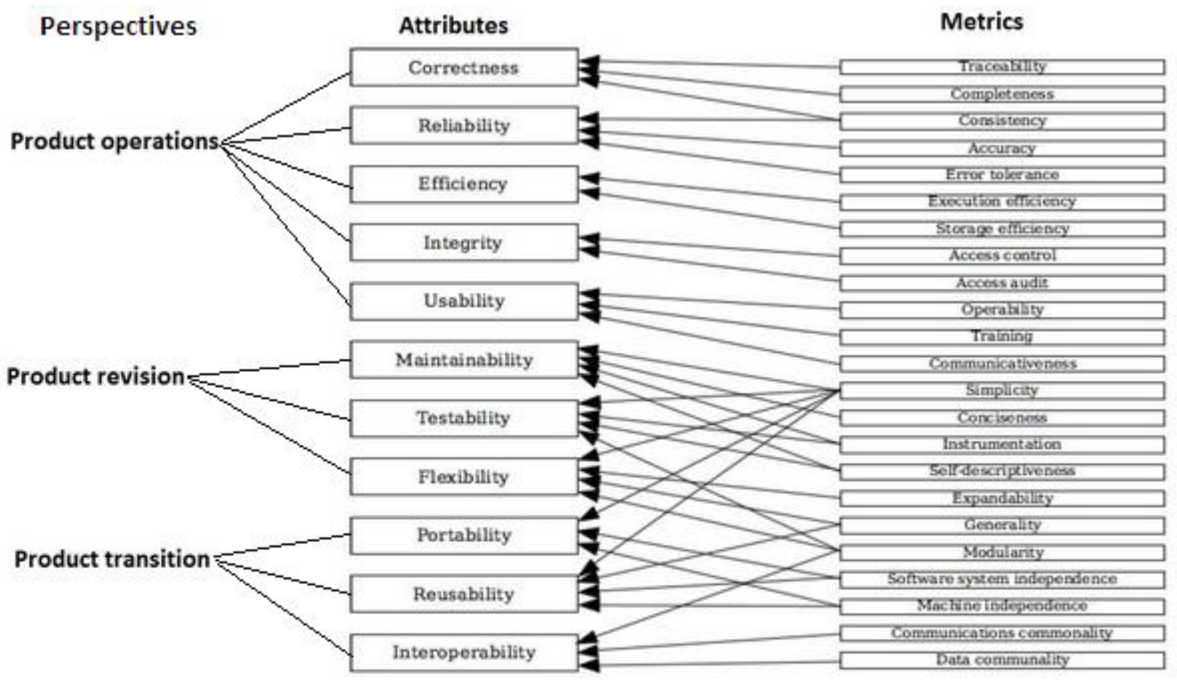


Figure 2.2 Jim McCall's quality model

If we match names for levels of this model with names for levels of our general quality model shown in Figure 2.1, attributes correspond to characteristics and metrics correspond to the level of the same name.

2.2.2 Boehm Quality Model

Another quality model proposed to evaluate the quality of software products is one defined by Barry W. Boehm in 1978 [18]. This model, like the McCall model, is composed by three levels in a hierarchy: primitive constructs (lowest level), quality factors (middle level), and primary uses (top level).

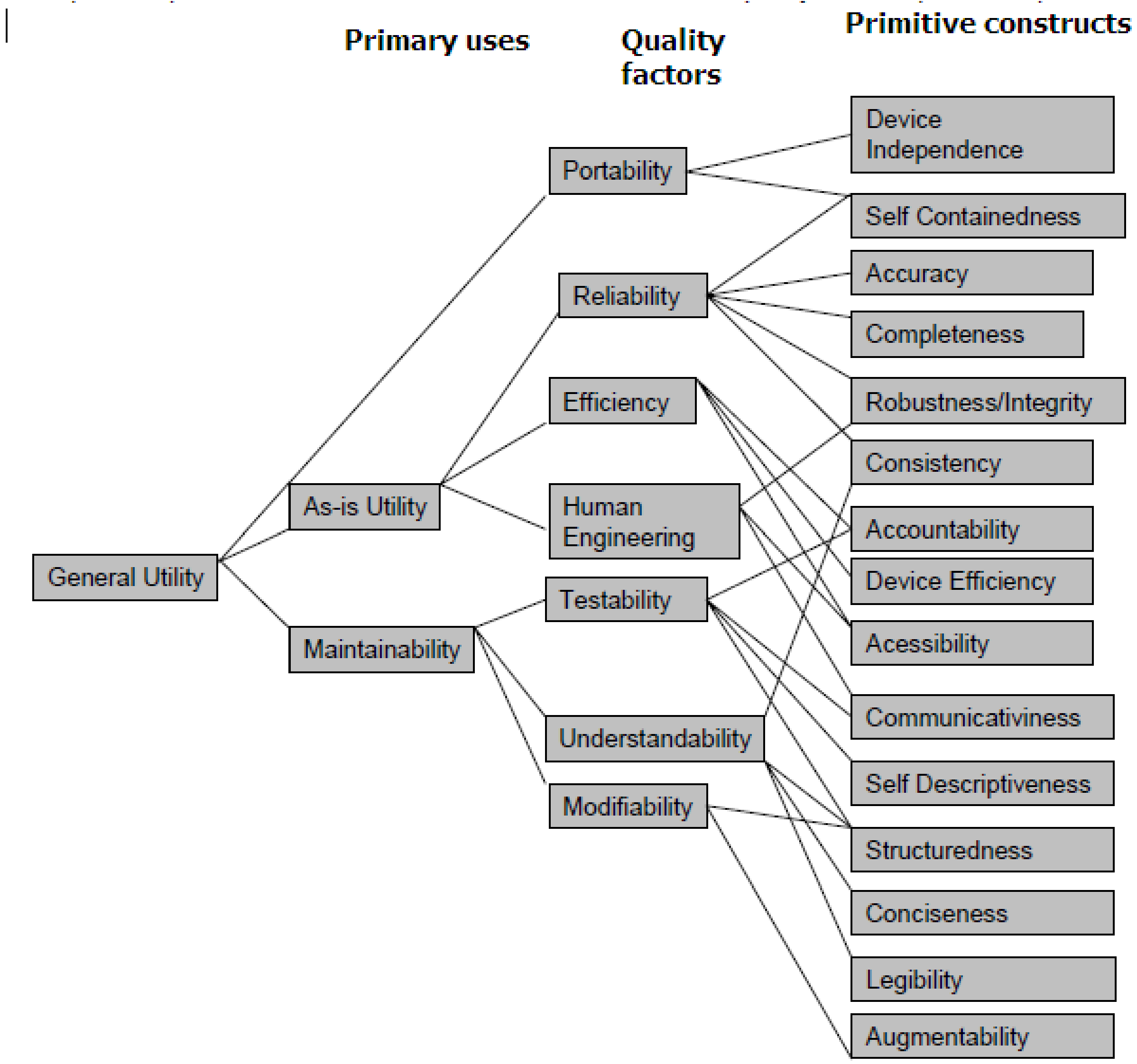


Figure 2.3 Boehm's quality model

Figure 2.3 shows the Boehm quality model, in which there are a set of quality factors and primitive constructs to quantitatively evaluate software quality.

Boehm defined three primary uses at the top level of the model (Portability, As-is utility, Maintainability) and 7 quality factors (Portability, Reliability, Efficiency, Usability (Human Engineering), Testability, Understability, Flexibility). Furthermore, these quality factors are divided into 15 metrics [7]. It must be noted that Portability is considered both a Primary Use and a Quality Factor.

Matching names for levels of this model with our model shown in Figure 2.1, quality factors correspond to characteristics and primitive constructs correspond to metrics.

2.2.3 ISO 9126

In 1993, the International Organization for Standardization (ISO) [20] created a standard for evaluating the quality of software products which is similar to the models of McCall and Boehm. This standard is composed by 6 characteristics, which in turn are broken down into 27 sub-characteristics (see Table 2.1). Each sub-characteristic is further divided in entities that can be verified or measured in the software product (metrics) but are not included in the standard due to the fact that it considers that they can change between different software products.

The ISO 9126 standard describes a set of software characteristics that allow software quality to be assessed according to the user view (external quality). Its quality characteristics are the following:

- Functionality. A set of attributes that bear on the existence of a set of functions and their specific properties. The functions are those that satisfy stated or implied needs.
- Reliability. A set of attributes that bear on the capability of software to maintain its performance level under stated conditions for a stated period of time.

- Efficiency. A set of attributes that bear on the relationship between the software's performance and the amount of resources used under stated conditions.
- Maintainability. A set of attributes that bear on the effort needed to make specific modifications which may include corrections, improvements, or adaptations of software to environmental changes, and changes in the requirements and functional specifications.
- Portability. A set of attributes that bear on the ability of software to be transferred from one environment to another. This includes the organizational, hardware or software environment.

Characteristics	**Sub-characteristics**
Functionality	Suitability, accuracy, interoperability, security, functionality compliance
Reliability	Maturity, fault tolerance, recoverability, reliability compliance
Usability	Understandability, learnability, operability, attractiveness, usability compliance
Efficiency	Time behavior, resource utilization, efficiency compliance
Maintainability	Analyzability, changeability, stability, testability, maintainability compliance
Portability	Adaptability, installability, replaceability, coexistence, portability compliance

Table 2.1 ISO/IEC 9126 Characteristics and Sub-Characteristics

In ISO/IEC 9126, "satisfaction" implies "the capability of the software product to satisfy users in a specified context of use." Satisfaction in that sense refers to the user's response to his/her interaction with the product [21].

Using any of these quality models previously described for derivation of system requirements brings clarity to definition of purpose and capability of operation [5].

	McCall's quality model	Boehm's quality model	ISO 9126
Is divided into:	Perspectives, attributes, metrics	Primary uses, quality factors, primitive constructs	Characteristics, subcharacteristics, entities
Created in:	1977	1978	1993
Quality attributes in common:	Portability, Maintainability, Reliability, Efficiency		

Table 2.2 Summary of the 3 quality models

Table 2.2 provides a brief summary of the 3 quality models that were discussed: McCall's model, Boehm's model, and ISO 9126. We can observe that the three models share a similar structure and similar purpose, which is quantitatively to evaluate the quality of a software product through the breakdown of quality attributes into quality metrics. The 3 models are composed by a hierarchy of 3 levels and, at the bottom of the hierarchy all of the models include entities that can be measured. It is interesting to observe that these 3 models share in common 4 quality attributes: Portability, Maintainability, Reliability and Efficiency. Maintainability is the quality attribute that is the focus of this thesis so we will now discuss metrics associated with it.

2.3 Maintainability related metrics

When studying software engineering quality, it is important to consider the existence of several programming paradigms: structured programming (SP), component-based programming, service oriented paradigm and object-oriented paradigm (OOP). These paradigms define a pattern that serves as a "school of thought" for programming of computers [22] and, as a consequence, each one will have different measures associated with maintainability. In the context of this thesis, we are interested in measuring maintainability for object oriented programs.

2.3.1 Measuring design properties to evaluate maintainability

OOP is characterized by properties such as inheritance, encapsulation, class hierarchies, polymorphism, etc [8]. Other important properties in the approach of OOP are cohesion (degree of consistence between parts of an object) and coupling (degree of interdependency between parts of an object) [11].

Bansiya and Davis defined a hierarchical model to evaluate software quality in OOP [8]. This model consists in product properties (that influence the quality of the product), a set of quality attributes and means to link both elements.

For the design of the model, from top to bottom, the quality attributes of the model were selected based on the attributes of ISO-9126. The model is also made of a link between quality attributes and design metrics, composed of object-oriented design properties. Design properties such as abstraction, encapsulation, complexity and design size are frequently used to measure design quality both in the object-oriented and structural paradigms. Design properties such as messaging, composition, inheritance and polymorphism are new properties introduced to measure quality in object-oriented design.

The results of this work showed the capability of the model to estimate overall design quality from design information, i.e., a significant correlation of estimated assessment by this model and the assessment of overall quality characteristics of some projects determined by independent evaluators, resulting an effective way for monitoring software product quality.

In object-oriented design, sets of metrics [11, 23] focusing on evaluating maintainability have been defined to evaluate this kind of programming paradigm. These metrics evaluate the design of code taking into account concepts that are specific to object-oriented design, such as inheritance, object, classes, message passing, etc.

As a consequence, we can conclude that measuring design properties is a good way to measure quality in terms of maintainability. The Chidamber and Kemerer set of metrics that is discussed in the following section allows design properties to be measured.

It’s to worth mentioning that we chose this set of metrics because, according to [10], Chidamber and Kemerer metrics appeared the most frequently in object-oriented studies and they perform better than other sets of metrics. Two decades after their initial publication, metrics from the C&K set are still the most used or investigated object-oriented metrics [10].

2.3.2 Chidamber and Kemerer metrics

There exist at least two sets of well known metrics for measuring design properties: The Li and Henry set [23] and the Chidamber and Kemerer set [11], which have some metrics in common (they have some overlap). In the context of this thesis, we are interested in using the Chidamber and Kemerer set as the data that will be used for evaluation purposes was collected using that particular set because data using this set is available for an extensive number of Java projects.

The Chidamber and Kemerer metrics set was established in 1994. This set of metrics was proposed due to the lack of metrics to support the evaluation of object-oriented programs [4]. Following Wand and Weber, the theoretical base chosen for these metrics was the ontology of Bunge [24]. The validation of these metrics proves that tracking them, these metrics will provide information to the project managers and developers to help them monitor the evolution of software developed under the object-oriented design paradigm. The Chidamber and Kemerer set is composed of six metrics:

- WMC: Weighted Method per Class
- DIT: Depth of Inheritance Tree
- NOC: Number of Children

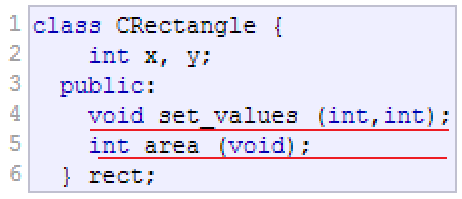
- CBO: Coupling Between Object classes
- RFC: Response for a Class
- LCOM: Lack of Cohesion of Methods

In the following sub-sections, we explain how these metrics are calculated.

2.3.2.1 WMC (Weighted Method per Class)

Weighted Method per Class is the metric that makes references to the sum of complexities of the methods of a class. An arbitrary value can be assigned to each method of a class, to simplify the calculation of this metric, a value of 1 is assigned to each method, which means this metric is simply the number of methods in a class [25].

What is a good WMC? Different limits have been defined. One way is to limit the number of methods in a class between 20 and 50, considering that classes with a huge number of methods are harder to modify and maintain. Another way is to specify that a maximum of 10% of classes can have more than 24 methods. In this way we could get larger classes, but the other 90% of classes should be small [25]. For our study, we limit the number of methods of a class to 24.



```
class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void);
  } rect;
```

Figure 2.4 Example class code

Figure 2.4 shows an example of how to calculate this metric. We can observe two underlined methods, so this metric is simply the count of methods for the class; the WMC for this small example is 2.

2.3.2.2 DIT (Depth of Inheritance Tree)

The Depth of Inheritance Tree metric provides for each class a measure of the inheritance levels from the top of the object hierarchy [25]. In the Figure 2.5, we can see an UML diagram that represents the inheritance relationships for the class “Shape”. To calculate the DIT metric for a class, we count the number of classes that we have to traverse to get to the leaf in the deepest inheritance branch. For the class “Square” the DIT metric is 3.

The deeper a class is in the hierarchy, the more methods and variables it is likely to inherit, making it more complex, which means deeper classes have a greater complexity design. A recommended DIT is 5 or less. The Visual Studio .NET documentation recommends a DIT of 5 or less because excessively deep class hierarchies are complex to develop and thus harder to maintain and modify [25].

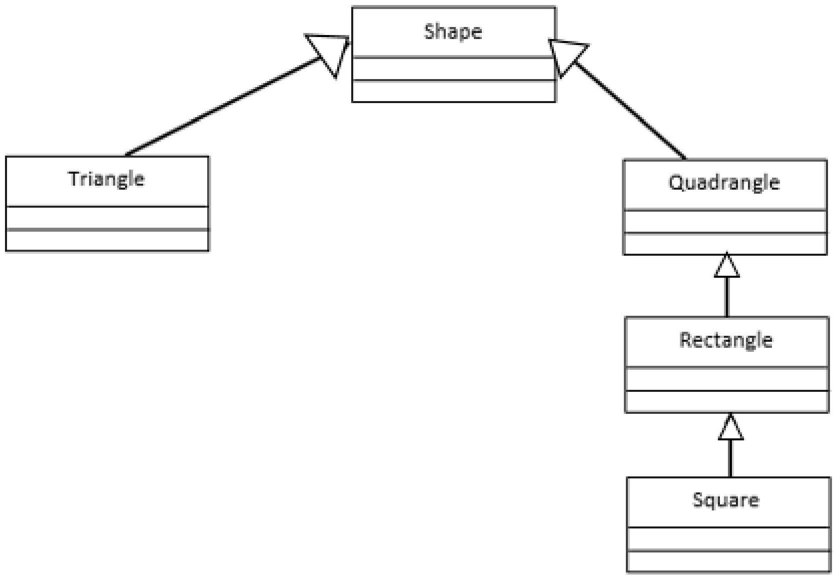


Figure 2.5 UML diagram for class Shape. DIT for this class is 3.

2.3.2.3 NOC (Number of Children)

The Number of Children metric counts the number of immediate descendants classes for a base class [25]. Figure 2.6 shows an UML diagram for the class "Shape". In this example, the calculation for the NOC metric is 3, since the class Shape has 3 subclasses.

A high NOC, a large number of child classes, can indicate several things [25]:

- High reuse of the base class since inheritance is a form of reuse.
- The base class may require more testing.
- Improper abstraction of the base class.
- Misuse of sub-classing. In such a case, it may be necessary to group related classes and introduce another level of inheritance.

The suggested value for NOC is 5 or less. This has been found to indicate fewer faults [25].

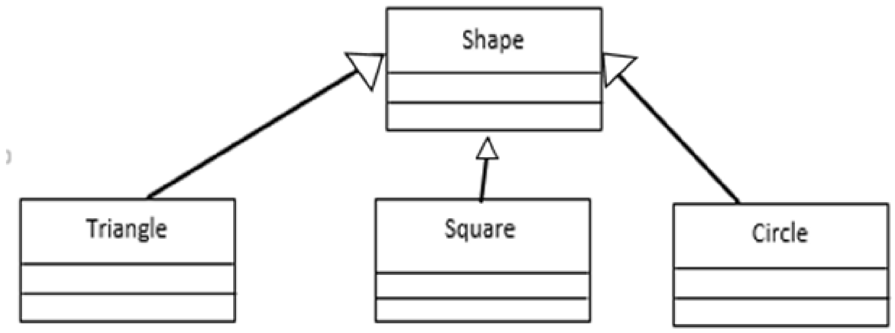


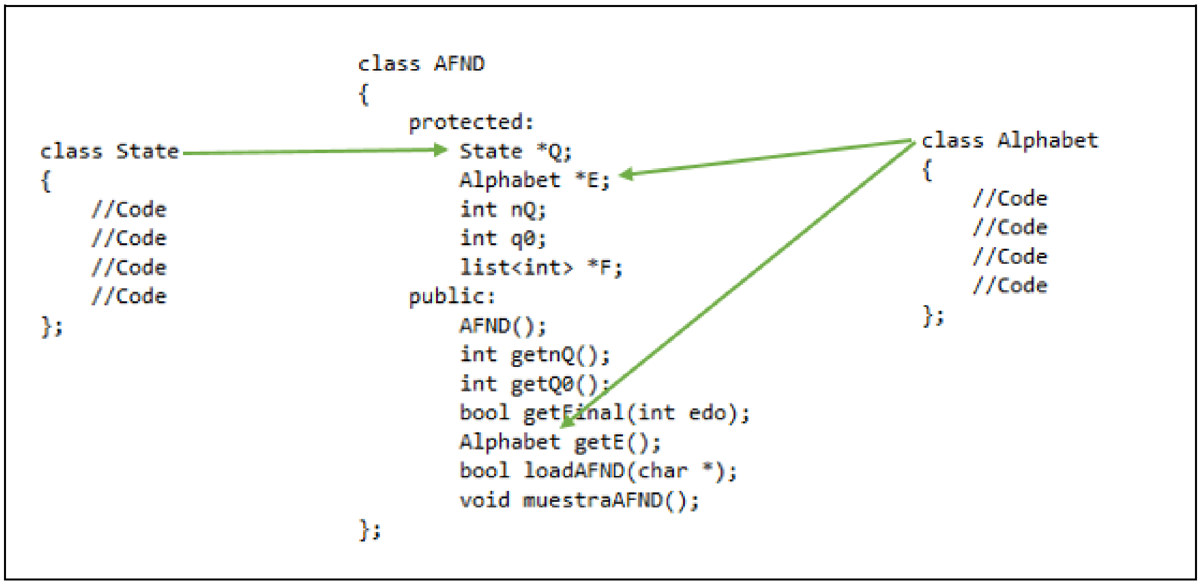
Figure 2.6 UML diagram for class Shape. NOC for this class is 3.

2.3.2.4 CBO (Coupling Between Object classes)

The CBO metric represents the number of classes coupled to a given class. This coupling can occur through method calls, attribute accesses, inheritance, arguments, return types, and exceptions [25].

High CBO is undesirable. The more independent a class is, the easier it is to reuse it in another application. The larger the number of couples, the higher the sensitivity to changes in other parts of the design of a class, and therefore maintenance of such class is more difficult. As such, excessive CBO neither is good for a modular design nor for reuse of code.

A high coupling has been found to indicate software to be faulty. Rigorous testing is thus needed. According to Sahraoui, Godin & Miceli [25], a CBO>14 is too high.



```
class State
{
    //Code
    //Code
    //Code
    //Code
};
```

```
class AFND
{
    protected:
        State *Q;
        Alphabet *E;
        int nQ;
        int q0;
        list<int> *F;
    public:
        AFND();
        int getnQ();
        int getQ0();
        bool getFinal(int edo);
        Alphabet getE();
        bool loadAFND(char *);
        void muestraAFND();
};
```

```
class Alphabet
{
    //Code
    //Code
    //Code
    //Code
};
```

Figure 2.7 Code example where CBO for class AFND is 2.

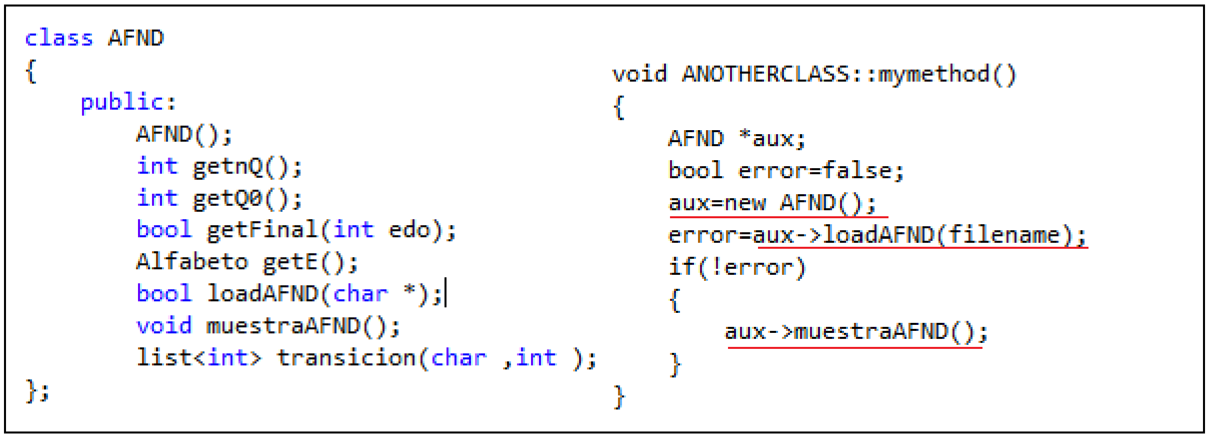
Figure 2.7 shows an example of code of class “AFND”. This figure illustrates how class “AFND” is coupled to two other classes, in this case, we find in class “AFND” a declaration of attributes of types “State” and “Alphabet” and a declaration of a method with a return value of type “Alphabet”, which represent a coupling to these two classes. We note that there are two couplings in class “AFND” to class “Alphabet”, however, multiple couplings to one class are counted as 1. In this small example, the count of CBO for class “AFND” is 2.

2.3.2.5 RFC (Response for a Class)

The metric RFC measures the number of different methods and constructors that can be invoked as a result of a method invocation on an object. The methods can be invoked in either the object that receives the message or on other objects from other classes [25].

A large RFC has been found to indicate more faults. The suggested maximum value for this metric is 24 [25]. Classes with a high RFC value are more complex and harder to understand (and thus to modify and maintain). Testing and debugging is complicated when we have a class with a high RFC [25].

Figure 2.8 shows an example of how we measure RFC: suppose we have a class called "AFND" and, from another class, we make use of the "AFND" class by creating an instance of it and invoking methods on this object. As we can see, what we count is underlined in red, so the count of the RFC metric for this example is 3. We count invocation to the constructor and to the methods from "AFND" in response to a call to the method `mymethod()`. It is worth noting that in this example we are supposing that the method `mymethod()` is the only method of "ANOTHERCLASS". If the current class that we are calculating RFC for had more methods, we would need to keep searching and calculating the number of method invocations to calculate the overall RFC.



```
class AFND
{
    public:
        AFND();
        int getnQ();
        int getQ0();
        bool getFinal(int edo);
        Alfabeto getE();
        bool loadAFND(char *);
        void muestraAFND();
        list<int> transicion(char ,int );
};
```

```
void ANOTHERCLASS::mymethod()
{
    AFND *aux;
    bool error=false;
    aux=new AFND();
    error=aux->loadAFND(filename);
    if(!error)
    {
        aux->muestraAFND();
    }
}
```

Figure 2.8 Example class code and its implementation (Example RFC)

2.3.2.6 LCOM (Lack of Cohesion of Methods)

A class's LCOM metric measures the sets of methods in a class that are not related through the sharing of some of the class's attributes [25]. The lack of cohesion in methods is then calculated by subtracting the number of method pairs that share an attribute from the number of method pairs that do not share an attribute:

$$P = \{(I_i, I_j) \mid I_i \cap I_j = \varphi\} \text{ and } Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \varphi\} \longrightarrow \text{LCOM} = |P| - |Q|.$$

Where P is the number of sets of methods that don’t share an attribute in common, Q is the number of sets of methods that share at least one attribute of the class in common, and I are the sets of methods. If the result is negative, LCOM is 0.

Class “Hello” has 3 methods {m1, m2, m3} and the attributes {a, b ,c ,d ,e ,x ,y ,z }
m1 uses the attributes {a, b, c, d, e}, m2 uses {a, b, c}, m3 uses {x, y, z}
m1 ∩ m2 = Not null, m1 ∩ m3=Null, m2 ∩ m3=Null.
2 Null –1 Not null = 1.LCOM for this class is then 1

Figure 2.9 Example of calculation LCOM

An example of how is calculated this metric is shown in figure 2.9. As we can see, we suppose we have a class named “Hello”. Suppose this class has 3 methods (method `m1()`, method `m2()` and method `m3()`), and each method uses some of the class attributes. Method `m1()` uses the attributes a, b, c, d, e; `m2()` uses the attributes a, b, c; `m3()` uses the attributes x, y, z. We then calculate the intersections between all possible method combinations. In this example, method `m1()` and method `m2()` are sharing at least one attribute in common, both have the attributes a, b and c, therefore, this intersection is not null. However, intersection between method `m1()` and method `m3()`, as well as intersection between method `m2()`and method `m3()` are null (no attributes in common). In this way we have 2 null intersections (groups or sets of methods that don't share an attribute in common) and 1 not null intersection (groups or sets of methods that share at least one attribute in common). As a result, LCOM for this example is 1.

LCOM = 0 indicates a cohesive class, so this is the recommended value.

LCOM > 0 indicates that the class needs or can be splitted into two or more classes, since its variables belong to disjoint sets.

Classes with a high LCOM have been found to be fault-prone [25]. Furthermore, high cohesion is an important design directive which is important to facilitate changes and maintain the source code of the system.

As a summary of the metrics, the 6 metrics and their descriptions are listed in Table 2.3. The recommended value for each metric is based on [25]. Given that we didn't find a threshold for some of the metrics, the threshold indicated was based on the threshold of a similar metric. For example, threshold for NOC is <5, as DIT also has to do with inheritance, threshold chosen for DIT is <5. We use thresholds only as references, as there are not established thresholds, results obtained in this work could be useful to establish a threshold for each metric.

Metric	**Description**	**Recommended value**
Weighted Method per Class (WMC)	If all method complexities are considered to be unity, then WMC = the number of methods.	<=24
Depth of Inheritance Tree (DIT)	The DIT refers to the maximum length from the root of the inheritance tree to the leaf in the deepest branch.	<=5
Number of Children (NOC)	Number of immediate subclasses subordinated to a class in a class hierarchy.	<=5
Coupling Between Object Classes (CBO)	CBO for a class is a count of the number of other classes to which it is coupled.	<=14
Response for a Class (RFC)	The response set of a class is a set of its methods that can potentially be executed in response to a message received by an object of that class.	<=24
Lack of Cohesion in Methods (LCOM)	The LCOM is a count of the number of method pairs which don't share a common attribute, minus the count of method pairs which share at least one common attribute. The larger the number of similar methods, the more cohesive the class.	0

Table 2.3 Summary of Chidamber and Kemerer metrics and their recommended values.

There exist several tools that automate the calculation of metrics to evaluate software quality. Some of these tools are commercial and other are open-source. These tools use some of the metrics proposed by Chidamber and Kemerer, or the proposed ones by Li and Henry. The following section discusses several of these tools in terms of their features and their level of support of the Chidamber and Kemerer metrics (we only mention the name of this set because the approach of this thesis is on Chidamber and Kemerer set).

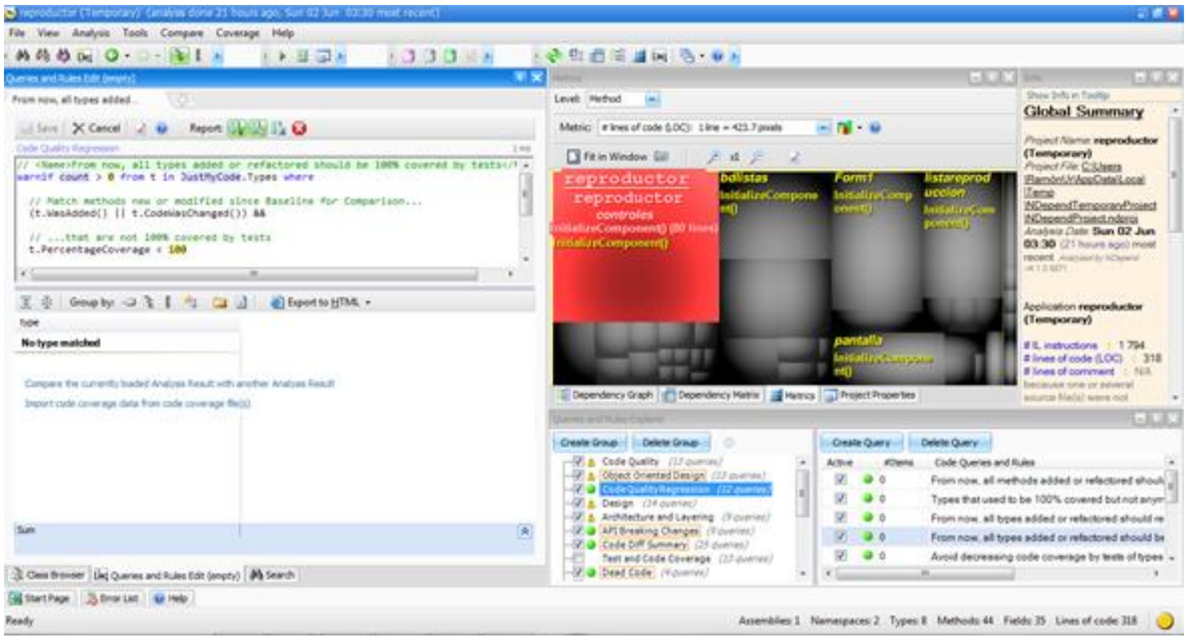
2.4 Software tools to evaluate software code

In this section we review 5 different tools that can be used to collect data in the Chidamber and Kemerer set of metrics for C++.

2.4.1 NDepend

NDepend [26] is a proprietary metrics tool for the analysis of .net managed code, including Managed C++. It provides a friendly user interface and displays detailed reports about the metrics it calculates. NDepend also provides a summary of rules that are violated, a treemap metrics review, a dependency graph, and other visualizations. Furthermore, NDepend shows warnings (suggestions), code queries and rules to improve the analyzed code. It can also export the code list elements matched by the code query to documents in HTML, Excel, XML, and export it to a graph or to a matrix. Besides its visual interface, NDepend also provides a command line version.

Within the full set of metrics this tool calculates, it supports 5 of the 6 Chidamber and Kemerer metrics and only RFC is not supported. Figure 2.10 shows a screen capture of the NDepend user interface.



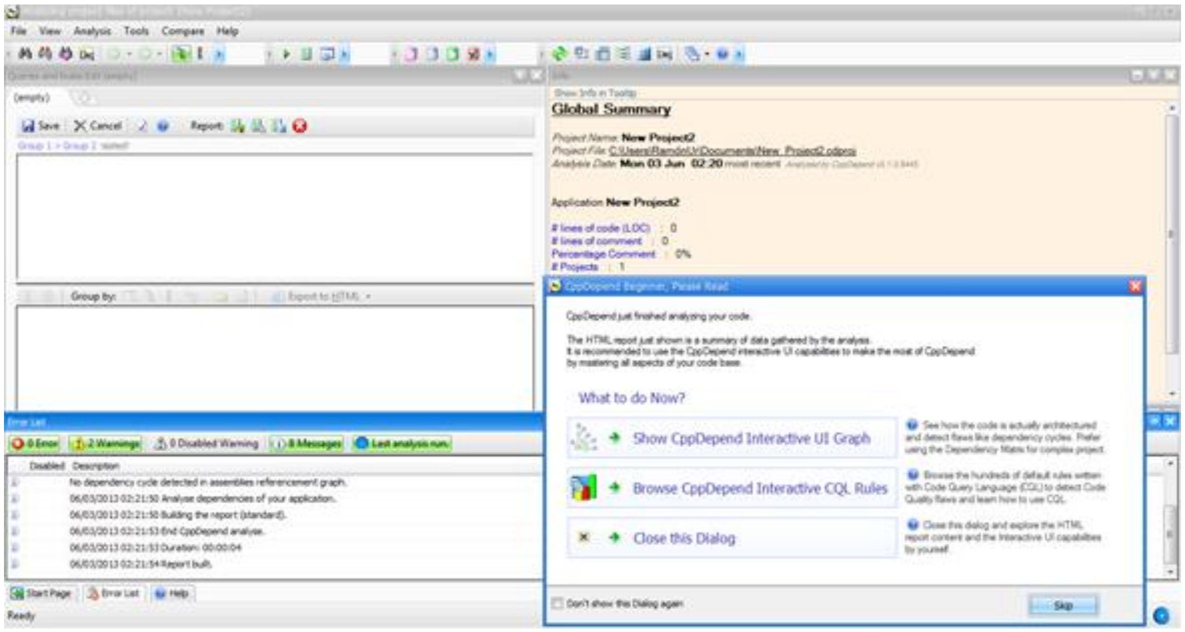
```
// <Name>From now, all types added or refactored should be 100% covered by tests</
warnif count > 0 from t in JustMyCode.Types where

  // Match methods new or modified since Baseline for Comparison...
  (t.WasAdded() || t.CodeWasChanged()) &&

  // ...that are not 100% covered by tests
  t.PercentageCoverage < 100
```

Figure 2.10 Screen capture of NDepend.

2.4.2 VisualCppDepend

VisualCppDepend [27] it is a proprietary tool that supports the analysis of C++ code. It is very similar to NDepend: Its features and the user interface are almost identical to NDepend. Unlike NDepend, however, VisualCppDepend provides a calculation for a smaller number of metrics and it also doesn’t provide code coverage data. Another difference to NDepend is that VisualCppDepend doesn't provide statistics like Standard Deviation or Average on some metrics. VisualCppDepend is also called CppDepend (when used in command line mode).

Regarding the Chidamber and Kemerer metrics, the coverage of this tool for such metrics is the same as NDepend (5 of 6 metrics, except RFC). Figure 2.11 shows a screen capture of the VisualCppDepend user interface.



Global Summary

Project Name: New Project2
Project File: C:\Users\Ramdn\Ur\Documents\New_Project2.cdproj
Analysis Date: Mon 03 Jun 02:20 most recent

Application New Project2

lines of code (LOC) : 0
lines of comment : 0
Percentage Comment : 0%
Projects : 1

CppDepend just finished analyzing your code.

The HTML report just shown is a summary of data gathered by the analysis.
It is recommended to use the CppDepend interactive UI capabilities to make the most of CppDepend by mastering all aspects of your code base.

See how the code is actually architectured and detect flaws like dependency cycles. Prefer using the Dependency Matrix for complex project.

Browse the hundreds of default rules written with Code Query Language (CQL) to detect Code Quality flaws and learn how to use CQL.

Close this dialog and explore the HTML report content and the interactive UI capabilities by yourself.

```
Disabled  Description
          No dependency cycle detected in assemblies referencement graph.
          06/03/2013 02:21:50 Analyse dependencies of your application.
          06/03/2013 02:21:50 Building the report (standard).
          06/03/2013 02:21:53 End CppDepend analyse.
          06/03/2013 02:21:53 Duration: 00:00:04
          06/03/2013 02:21:54 Report built.
```

Figure 2.11 Screen capture of VisualCppDepend.

2.4.3 Understand for C++ 1.4

Understand for C++ [28] is a proprietary tool that supports the analysis of C++ code. The user interface of this tool is a bit similar to NDepend and

VisualCppDepend but (according to our user experience) it is a little more difficult to get involved into the tool, maybe this is because it provides a more extensive menu. Understand for C++ contains a wide range of statistics metrics (average, max value, count) which, once they have been calculated on a C++ file, can be exported as a CSV file. Like other tools, Understand for C++ also provides to the user some graphs and diagrams for presenting the results of the analyzed code, the tool can even produce UML class diagrams.

Regarding the Chidamber and Kemerer metrics, this tool provides full coverage. Figure 2.12 shows a screen capture of the Understand for C++ user interface.

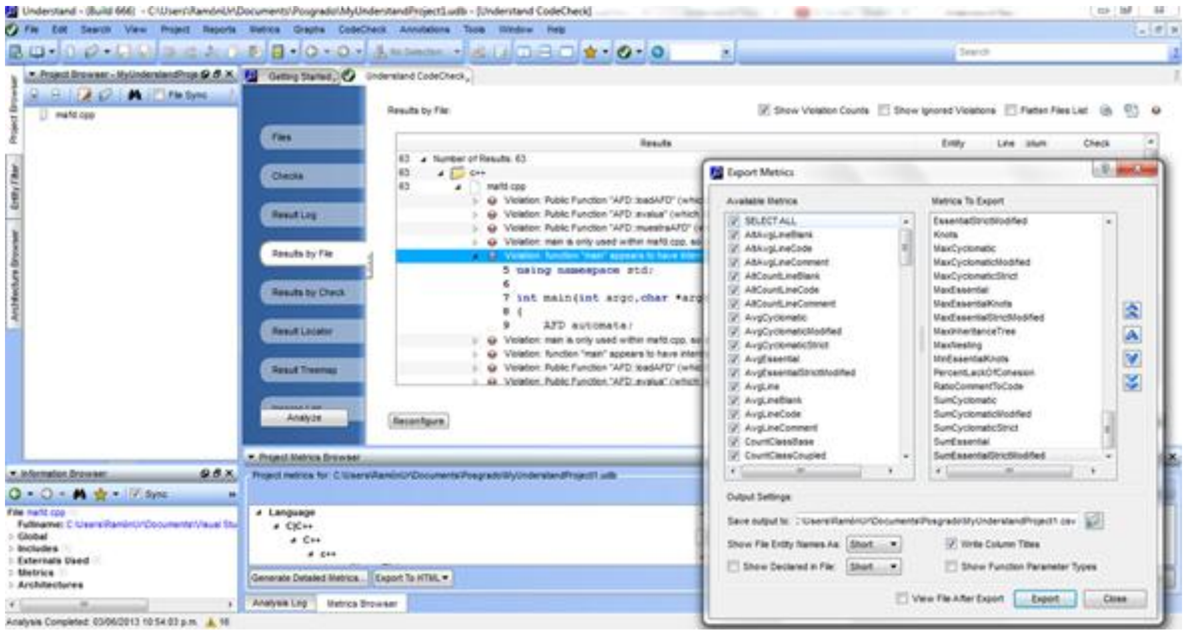


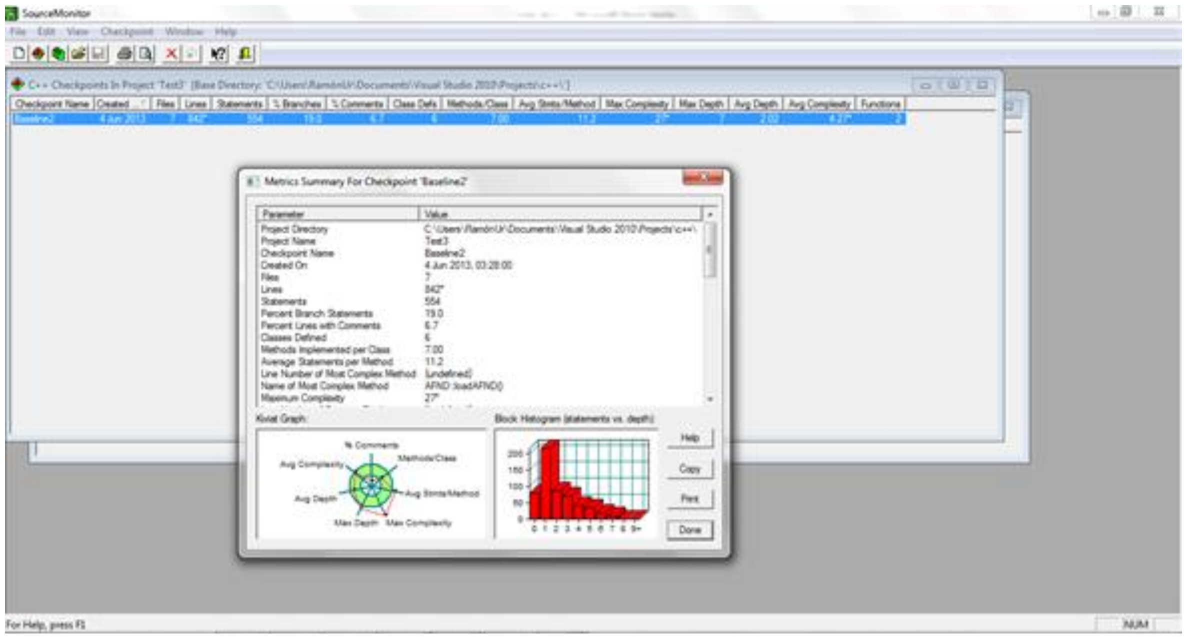
Figure 2.12 Screen capture of Understand for C++.

2.4.4 SourceMonitor

SourceMonitor [29] is another proprietary tool that supports the analysis of C++ code. This is a free program; however, it is not open-source and the source code is not available because it contains proprietary source code files from third party vendors.

Unlike the previous tools, SourceMonitor is a little easier to use as it does not provide as many functions. The only important functions include calculating a few metrics like methods per class, average statements per method, average depth, etc. This tool also provides a Kiviat graph measuring the calculated metrics and also provides a histogram block comparing the statements in depth. The main result is shown in a single table; so its interface is somewhat austere. However, as other similar tools, after calculating the metrics on the different files of a project, this tool can export the tables with the calculated metrics to a CSV or XML files.

Regarding the Chidamber and Kemerer metrics, this tool only supports three of the six metrics: DIT, NOC and RFC. Figure 2.13 shows a screen capture of the SourceMonitor user interface.


SourceMonitor
File Edit View Checkpoint Window Help
C++ Checkpoints In Project 'Test3' [Base Directory: 'C:\Users\RamónUr\Documents\Visual Studio 2010\Projects\c++\']
Metrics Summary For Checkpoint 'Baseline2'
Kiviat Graph:
% Comments
Methods/Class
Avg Stmts/Method
Max Complexity
Max Depth
Avg Depth
Avg Complexity
Block Histogram (statements vs. depth):
Help
Copy
Print
Done
For Help, press F1
NUM

Checkpoint Name	Created	Files	Lines	Statements	% Branches	% Comments	Class Defs	Methods/Class	Avg Stmts/Method	Max Complexity	Max Depth	Avg Depth	Avg Complexity	Functions
Baseline2	4 Jun 2013	7	842*	554	19.0	6.7	6	7.00	11.2	27*	7	2.02	4.27*	2

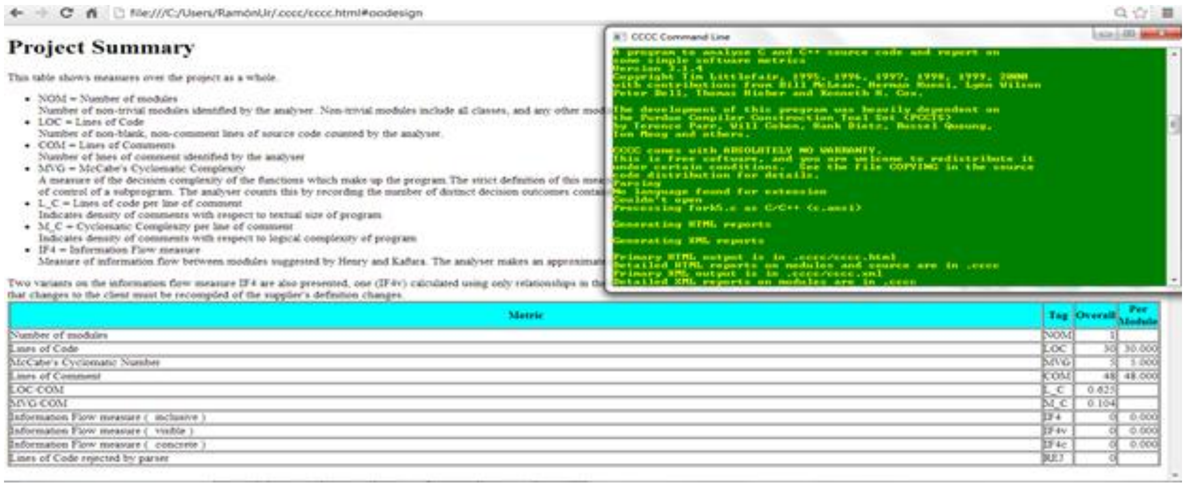
Parameter	Value
Project Directory	C:\Users\RamónUr\Documents\Visual Studio 2010\Projects\c++\
Project Name	Test3
Checkpoint Name	Baseline2
Created On	4 Jun 2013, 03:28:00
Files	7
Lines	842*
Statements	554
Percent Branch Statements	19.0
Percent Lines with Comments	6.7
Classes Defined	6
Methods Implemented per Class	7.00
Average Statements per Method	11.2
Line Number of Most Complex Method	{undefined}
Name of Most Complex Method	AFND::loadAFND()
Maximum Complexity	27*

Figure 2.13 Screen capture of SourceMonitor.

2.4.5 CCCC

CCCC [30] is an open source tool that supports the analysis of C++ code. It was developed by Tim Littlefair, as part of a PhD research project. This tool has no visual user interface; the user can only analyze C++ projects and generate reports in HTML or XML files from the command line. The generated report shows a project summary, a procedural metrics summary and a structural metrics summary. From the set of analyzed tools, CCCC seems to be the tool that has the smallest functionality.

Regarding the coverage of the Chidamber and Kemerer metrics, CCCC only supports three out of the six metrics: WMC, DIT and NOC, while it lacks CBO, RFC and LCOM. Figure 2.14 shows this tool working from the command line and a report generated by this tool in HTML format.



Project Summary

This table shows measures over the project as a whole.

- NOM = Number of modules
  Number of non-trivial modules identified by the analyser. Non-trivial modules include all classes, and any other mod
- LOC = Lines of Code
  Number of non-blank, non-comment lines of source code counted by the analyser.
- COM = Lines of Comments
  Number of lines of comment identified by the analyser
- MVG = McCabe's Cyclomatic Complexity
  A measure of the decision complexity of the functions which make up the program.The strict definition of this mea
  of control of a subprogram. The analyser counts this by recording the number of distinct decision outcomes conta
- L_C = Lines of code per line of comment
  Indicates density of comments with respect to textual size of program
- M_C = Cyclomatic Complexity per line of comment
  Indicates density of comments with respect to logical complexity of program
- IF4 = Information Flow measure
  Measure of information flow between modules suggested by Henry and Kafura. The analyser makes an approximat

Two variants on the information flow measure IF4 are also presented, one (IF4v) calculated using only relationships in th
that changes to the client must be recompiled of the supplier's definition changes.

Metric	Tag	Overall	Per Module
Number of modules	NOM	1	
Lines of Code	LOC	30	30.000
McCabe's Cyclomatic Number	MVG	3	3.000
Lines of Comment	COM	48	48.000
LOC/COM	L_C	0.625	
MVG/COM	M_C	0.104	
Information Flow measure (inclusive)	IF4	0	0.000
Information Flow measure (visible)	IF4v	0	0.000
Information Flow measure (concrete)	IF4c	0	0.000
Lines of Code rejected by parser	REJ	0	

```
A program to analyse C and C++ source code and report on
some simple software metrics
Version 3.1.4
Copyright Tim Littlefair, 1995, 1996, 1997, 1998, 1999, 2000
with contributions from Bill McLean, Herman Hueni, Lynn Wilson
Peter Bell, Thomas Hieber and Kenneth H. Cox.

The development of this program was heavily dependent on
the Purdue Compiler Construction Tool Set (PCCTS)
by Terence Parr, Will Cohen, Hank Dietz, Russel Quong,
Tom Moog and others.

CCCC comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions.  See the file COPYING in the source
code distribution for details.
Parsing
No language found for extension
Couldn't open
Processing fork5.c as C/C++ (c.ansi)

Generating HTML reports

Generating XML reports

Primary HTML output is in .cccc/cccc.html
Detailed HTML reports on modules and source are in .cccc
Primary XML output is in .cccc/cccc.xml
Detailed XML reports on modules are in .cccc
```

Figure 2.14 Command line and HTML report of CCCC.

2.4.6 Analysis on the software tools that evaluate software code

Table 2.4 summarizes the coverage of the previously discussed C++ code analysis tools in terms of the Chidamber and Kemerer metrics and it also shows whether the tool is open source or not.

	WMC	DIT	NOC	CBO	RFC	LCOM	Open Source
Understand for C++ 1.4	✓	✓	✓	✓	✓	✓	✗
CCCC	✓	✓	✓	✗	✗	✗	✓
VisualCppDepend	✓	✓	✓	✓	✗	✓	✗
Source Monitor	✗	✓	✓	✗	✓	✗	✗
Ndepend	✓	✓	✓	✓	✗	✓	✗

Table 2.4 Comparative Chidamber and Kemerer metrics with software tools

The comparison in Table 2.4 shows that there is only one tool, Understand for C++, that covers all of the 6 metrics of the Chidamber and Kemerer set, however, this tool is not open-source, which is a desired characteristic to meet the objectives pursued in this thesis. The only tool from ones in Table 2.4 that is open-source is CCCC. However it only covers half of the Chidamber and Kemerer set of metrics.

2.5 Summary

This chapter started with a discussion on quality and, more specifically, on software quality. Software quality can be seen as something subjective and a bit complicated to evaluate directly. Fortunately, a series of models have been created to establish software quality as something that can be measurable and objective. These models establish software quality as a hierarchy that defines a series of characteristics (or quality attributes) which, in turn, are refined into a set of metrics. Within these characteristics, we find maintainability which is the quality attribute that is of interest in this thesis.

Metrics that evaluate the design of object oriented systems can be used to measure maintainability. Among these metrics, the Chidamber and Kemerer set is widely used. To automate the calculation of such metrics on different object oriented languages like Java, C# or C++, several code analysis tools have been developed. We performed a comparison of 5 tools which revealed that only one proprietary tool supported the complete metrics set for C++ programs.

Chapter 3. Maya C++ tool requirements and design

3.1 Introduction

The evaluation of tools that was presented in the previous chapter revealed that there is a lack of open-source C++ code analysis tools that provide coverage of the complete Chidamber and Kemerer metrics set. Modifying the CCCC tool was considered as an option, but we found that its code is not well documented. For these reasons, we decided to create a new tool which was named Maya C++, in honor to one of the most important antique civilizations in Mexico. In the following sections we discuss the requirements and the design of the Maya C++ tool.

3.2 Vision and scope

Figure 3.1 shows a context diagram which is helpful in understanding the vision for the Maya C++ tool. Maya C++ an interactive tool that receives, as its input, the location of a project composed of C++ files (that is, files with the ".h" and ".cpp" extension). The tool analyzes the code from the files and then proceeds to calculate the Chidamber and Kemerer metrics. The results are displayed visually and can be also exported to a file using the XML of CSV format.

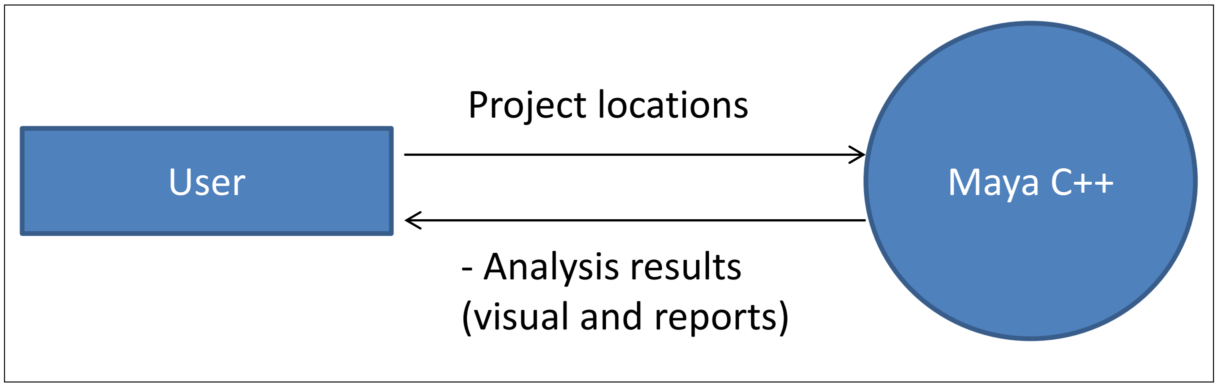


Figure 3.1 Context diagram of Maya C++

Maya C++ has a relatively limited set of features (high level requirements) but these features are similar to the tools that we reviewed in the previous chapter including data analysis and reporting. The features for the tool are presented considering the three categories that make up architectural drivers: functional requirements, quality attributes and constraints [31].

Primary functional features	• Parse project files, calculate metrics and render them graphically, and export them in CSV and XML files.
Quality attributes	• Performance, the tool should provide fast feedback to the user even if a large project is used.
Constraints	• The development of the tool must be finished in 6 months. • The software development should be open-source.

For the scope of the project, we only consider features that are necessary in order to achieve the general objective of this thesis.

3.3 Software requirements

Once the Vision and Scope for Maya C++ has been established, the features that were discussed previously are refined as requirements in the following sections.

3.3.1 Primary Use Case

Maya C++ features only one primary use case which is detailed in table 3.1.

Template for use case documentation

1. ID and name

UC-1 Calculate project metrics

2. Description

This use case allows the set of Chidamber and Kemerer metrics to be calculated on all of the classes of a C++ project. The results are displayed graphically and can be saved in CSV and XML files.

3. Actors

System User.

4. Pre-conditions

The application has been launched

5. Post-conditions

The system has calculated the metrics for the project and is awaiting user interaction.

6. Main flow

1. The user presses the “Select Project” button.
2. The system asks the user to select a directory with a C++ project.
3. The user selects a directory
4. The system analyzes the project files and displays a message informing that the project has been successfully loaded.
5. The User presses the “Extract classes and calculate metrics button”
6. The system then calculates the metrics and displays the results, including the list of classes.
7. The user selects a particular class from the list
8. The system displays the metrics for the selected class

7. Alternative Flows

AF-1: In step 7:

1. The user selects the “Save Project” option to export the data in either CSV or XML form
2. The system requests the file name and location
3. The user inputs the requested information
4. The system generates the file, exports the data and displays a message
5. The use case continues in step 7

.

8. Exceptional Flows

EF-1: In step 4, if the selected directory does not contain C++ files

1. The system notifies the user of the problem

Table 3.1 Use case “Calculate the six metrics of Chidamber and Kemerer on a C++ project.

3.3.2 Quality attributes scenario

The main quality attribute for this system concerns Performance. It is specified in Table 3.2 using the Scenario technique from the book “Software Architecture and Principles Practice” [32].

Quality attributes relevant:	Performance / Usability
Stimulus:	Provides the location of a large project and triggers the calculation of the metrics
Source of stimulus:	A user.
Environment:	The system is awaiting the user interaction.
Artefact:	The system.
Response:	Classes in the selected directory are extracted and the results for one class are displayed
Measurement of response:	In less than one second

Table 3.2 Quality scenario

It’s worth to mentioning that the system can be used to analyze either complete projects or only a few classes at a time. This scenario focuses on the latter situation as it is necessary to ensure the user can begin interacting with the results even if the project is large.

3.3.3 Constraints

Constraints are not refined from what was discussed in the previous section.

3.4 System design

In this section we discuss the design of the Maya C++ tool using software architecture principles including the concept of documenting its architecture using (partial) logical and dynamic views. Since this is a standalone system, no physical view is included.

3.4.1 Logical view

The architecture of Maya C++ is based on a Layered architectural style which is commonly used for interactive systems. Three main layers were defined:

- A presentation layer which contains modules that support user interaction
- A business layer which contains modules that perform the calculations
- A data layer which contains modules that perform data input and output.

Figure 3.2 illustrates the layers that compose the architecture and several modules that are located within the layers. Table 3.2 describes the responsibilities of the modules located inside the layers.

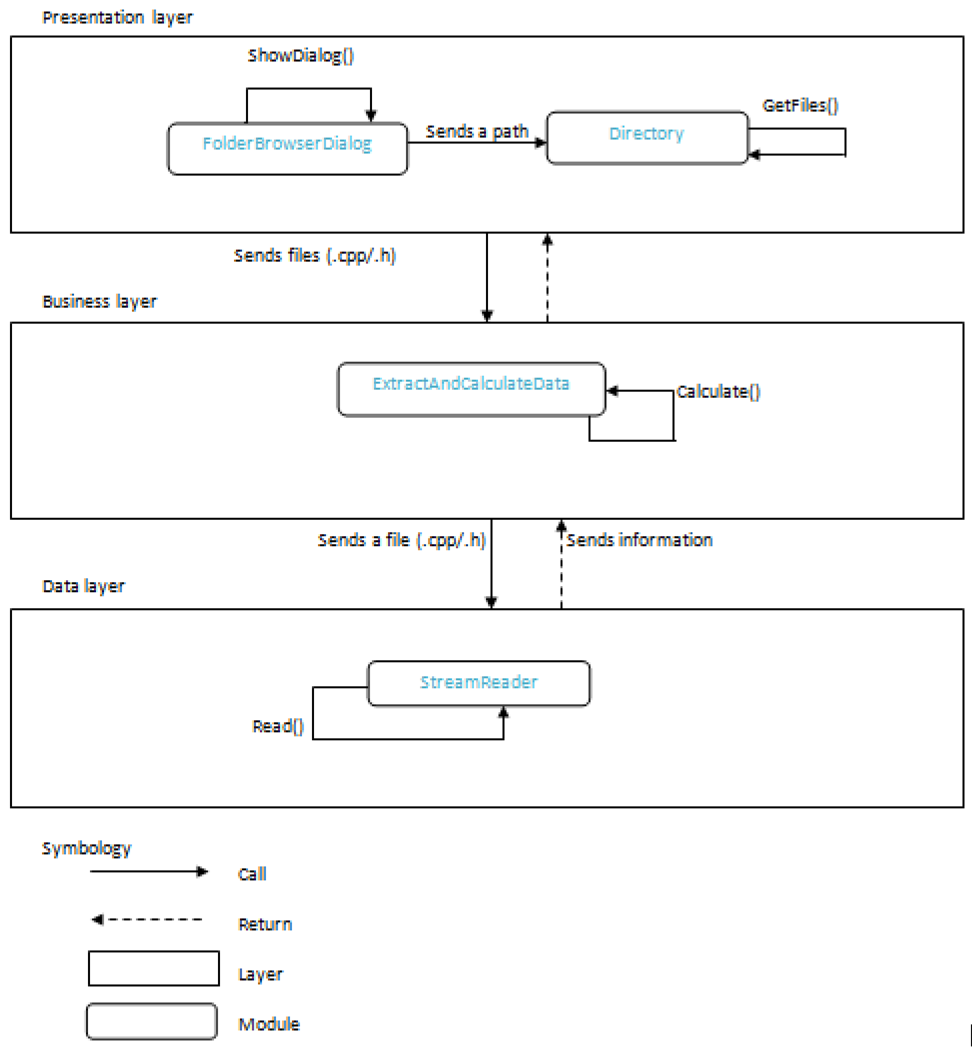


Figure 3.2

Logical view of the architecture of Maya C++

Module	Responsibility	Interface and implementation information
`FolderBrowserDialog`	This module is responsible of providing a window to select a directory.	The method ShowDialog() requests the user to select a directory where a project is located. This module is obtained from the System.Windows.Forms library.
`Directory`	This module is responsible for moving through directories and subdirectories.	The method GetFiles() obtains only the requested files for the user (.cpp & .h) in a selected directory. This module is obtained from the System.IO library.
`ExtractAndCalculateData`	This is the core module which is responsible for extracting and processing the data. It is also responsible for calculating the metrics on the data.	The method Calculate() uses the StreamReader class to read a file, extracts the information, and then calculates the metrics.
`StreamReader`	This module is responsible for reading of the files contained in a selected directory.	The method Read() reads each and all of the characters in a file until the file ends. This module is obtained from the System.IO library.

Table 3.3 Responsibilities of the modules

The design decisions relative to the selection of technology are:

- The tool was developed using the Microsoft .NET C# language. This programming language was chosen because the author was familiar with it.
- The.NET 4.0 Framework was selected because the author has skills with it and because it provides support for working with a visual editor.

- Once the project location is provided, the system loads the list of files but only performs calculations for one class in order to provide rapid (less than 1 second) feedback to the user and satisfy the performance quality scenario.

3.4.2 Dynamic view

Figure 3.4 presents a UML sequence diagram which illustrates a scenario for the primary use case where the metrics for a class in the project are calculated and the results are displayed to the user.

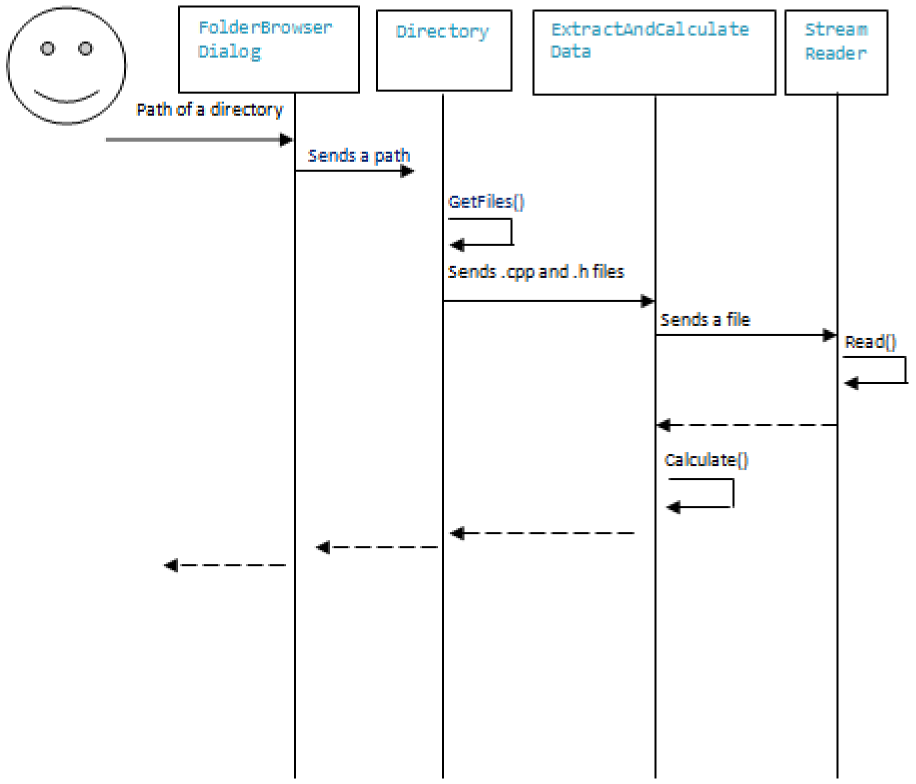
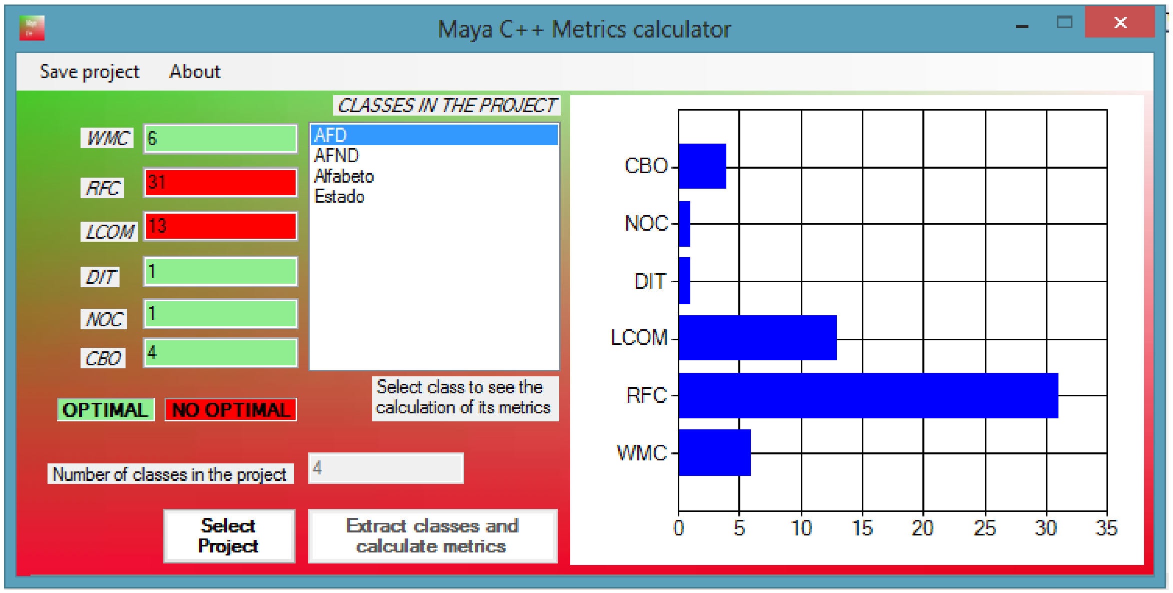


Figure 3.4 Sequence diagram for a scenario of UC-1

The diagram shows that the user gives a path (directory) as input to the FolderBrowserDialog instance, this instance sends this path to the Directory instance, which in turn gets all the .cpp and .h files in that directory. The files are then passed to an instance of the ExtractingAndCalculateData module which sends each file to a StreamReader instance which reads those files and returns the characters contained in the files. The ExtractingAndCalculateData instance parses the characters and matches it with respect to a set of tokens and then performs the metrics calculations.

3.4.3 User interface

Another important aspect of Maya C++ is its user interface which is shown in Figure 3.5. The interface of Maya C++ allows the user to analyze each class individually and to export results for the complete project as a file. The interface also warns the user whether the values of the metrics for a particular class are within the suggested ranges


Maya C++ Metrics calculator
Save project
About
CLASSES IN THE PROJECT
WMC
RFC
LCOM
DIT
NOC
CBO
AFD
AFND
Alfabeto
Estado
Select class to see the calculation of its metrics
OPTIMAL
NO OPTIMAL
Number of classes in the project
Select Project
Extract classes and calculate metrics
CBO
NOC
DIT
LCOM
RFC
WMC
0
5
10
15
20
25
30
35

WMC	6
RFC	31
LCOM	13
DIT	1
NOC	1
CBO	4
Number of classes in the project	4

Figure 3.5 Maya C++ user interface

The interaction between user and system starts when the user presses the “Select Project” button. At that moment the system displays a folder browser dialog and then the user selects a directory that contains a C++ project. Once this is done, the system displays a message that the selected project is successfully loaded; otherwise, the system displays a message that the system could not find C++ files in the directory. After a project is successfully loaded, the user selects the

“Extract classes and calculate metrics”. As a result, the system displays the classes of the selected project in the list labeled “CLASSES IN THE PROJECT”. At this point, the user is able to select classes from the list. Once the user selects a class, the system displays the values of the metrics as numbers and also as a graph. When the number for a particular metric is outside the suggested range (see section 2.3.2), it is highlighted in red, otherwise it is shown in green.

When the user selects the option “Save project” and then “As CSV file” or “As XML file”, the system asks for a location and name for the desired file. Afterwards, the system displays a message that the creation with the full calculation of the project is in process. At the end, the system displays another message informing that the creation of the CSV or XML file was successful.

3.5 Summary

This chapter has discussed the requirements and design of the Maya C++ tool. As part of the requirements, we have presented both functional and non-functional (quality attribute) requirements and, to some extent, the constraints. The design was presented using architectural concepts including the use of two types of views: logical and physical. Finally, we have also described the user interface of the tool. Maya C++ provides simple functionality in comparison to the tools that were described in section 2.4 but it provides coverage for the 6 Chidamber and Kemerer metrics.

The next chapter presents the results of the experimentation that was performed using Maya C++.

Chapter 4. Evaluation

4.1 Introduction

To verify that Maya C++ was working correctly, we did some tests by evaluating some projects taken from the sample projects analyzed in Chapter 4 with our tool, and did the same evaluations with a commercial tool (Understand for C++). We compared given results by these both tools and observed that the results were the same.

Once the Maya C++ was tested we had the necessary elements to conduct a brief case of study according to [33]. The case study includes objectives, research questions, preparation for data collection (Project sample), collecting evidence (Data collection and analysis), analysis of collected data (Data collection and analysis), and reporting (Analysis of the results). This is discussed in this chapter.

4.1.1 Objectives

- To figure out if C++ projects or Java projects have better results with respect to Chidamber and Kemerer metrics.
- In case there's no a clear programming language with better design quality, to figure out what metrics favor Java and what metrics favor C++.
- To figure out if C++ or Java complies with all of our suggested ranges for each metric.

4.1.2 Research questions

- What programming language gives projects with better design quality with respect to Chidamber and Kemerer metrics?
- What metrics favor Java?

- What metrics favor C++?
- What programming language is more complied with our suggested ranges for the metrics?

4.2.1 Project sample

We performed a comparison of 48 object oriented projects written in C++ and Java. This comparison was made with respect to the Chidamber and Kemerer metrics: WMC, DIT, NOC, CBO, RFC, LCOM. We selected for our case study a sample of 24 open-source projects in C++ from different domains. The source code for the projects was obtained from the site of http://sourceforge.net. Projects were selected randomly, and 3 projects were chosen for each different business domain, including Business and enterprise, Audio and Video, Communications, Development, Game, Graphics, Home and Education, Science and Engineering. The data from these projects was compared with data obtained from 24 evaluations of similar projects in Java available from the site http://percerons.com/.

4.2.2 Data collection and analysis

For the data collection we used two categories of methods: indirect and independent [33].

The indirect method does not require direct interaction with people and instead we used “Maya C++”, the tool that we developed, to obtain data for the 24 C++ open-source projects. Results were generated in CSV files. These files were further loaded into Excel to allow for data manipulation. The independent method for data collection was used for Java projects as the data was already available in the site percerons.com. The data was also imported into Excel tables to allow for data manipulation.

For the analysis, we did a project by project comparison across metrics, in order to study which projects have a higher percentage of classes whose metrics are outside the suggested ranges for each one of the Chidamber and Kemerer metrics. Suggested ranges of values for the different metrics were taken from [25] (see also 2.3.2) and are summarized here:

- WMC: <=24
- DIT: <=5
- NOC: <=5
- CBO: <=14
- RFC: <=24
- LCOM: 0

Since we measure how many classes have values outside the suggested ranges, higher percentages are undesirable while 0 is better.

This first analysis is normalized as it is independent of the project size. However, we also performed a second analysis to understand how much the actual metric value averages deviate from the suggested values. This second analysis is not independent of project size, but it provides additional valuable information in order to answer our research questions.

4.3 Results

We present the results using graphs and analysis of the data. In each graph we illustrate the results that were obtained with respect to the percentage of the total number of classes with non-suggested values for the six Chidamber and Kemerer metrics. Each graph shows the 48 projects that were evaluated. The 24 Java projects and the 24 C++ projects are separated in the graph by a blank. Java projects are shown on the left and range from “A Java Library for R. & W. Excel” to “DBMantain”, and C++ projects are shown on the right and range from “Avogadro” to “VVV”.

4.3.1 Results according to percentage of classes whose metrics are outside the suggested ranges

4.3.1.1 Results for WMC (Weighted Method per Class)

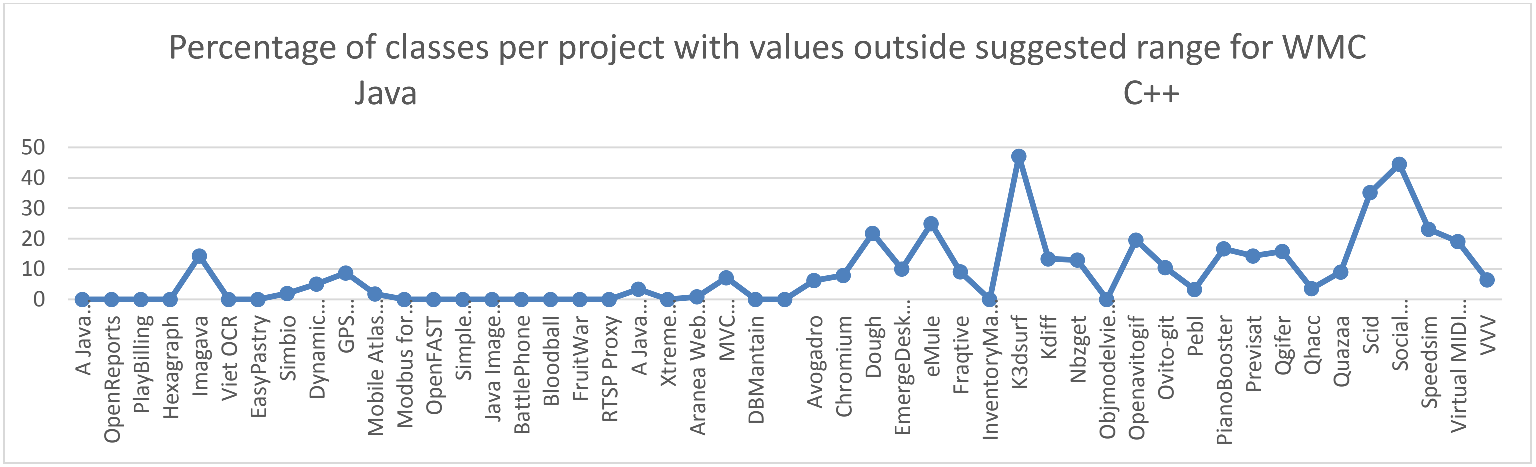


Figure 4.1 WMC results across the projects

The results for the WMC metric are shown in this graphic. The average of the percentage of classes per project with values outside suggested range for Java projects (shown on the left) is 1.80 while the average for C++ projects (shown on the right) is 15.58. Furthermore, out of 24 Java projects, 16 of them (66%) had no classes with values outside the suggested range for WMC compared with only 2 (8.3%) for C++. In conclusion, the results for WMC were better for Java. There is a marked difference between the projects in the two languages and we can conclude that classes in C++ projects are more complex (since WMC measures the number of methods). This result may be explained by considering the result on the analysis in the next metric (DIT): the analysis on DIT seems to favor C++, so possibly the approach in Java is to design classes with fewer methods and to inherit more methods from other classes.

4.3.1.2 Results for DIT (Depth of Inheritance Tree)

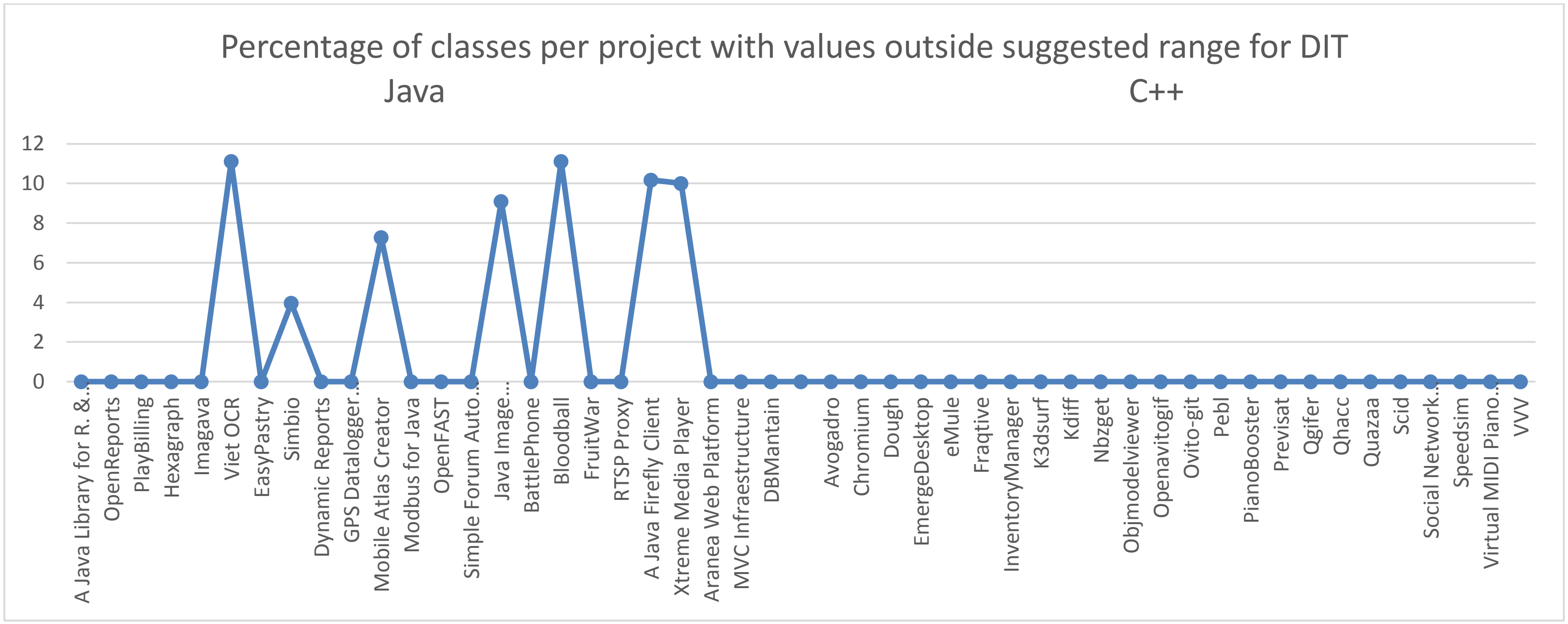


Figure 4.2 DIT results across the projects

The results for the DIT metric are shown in this graphic. The average of the percentage of classes per project with values outside suggested range for Java projects (shown on the left) is 2.61 while the average for C++ projects (shown on the right) is 0. Furthermore 17 out of 24 (70%) Java projects have no classes outside the suggested range, compared to 100% with Java. So, we can conclude that DIT is better for C++ projects; this may be explained because C++ allows multiple inheritance, which means that a class can inherit from more than one class, avoiding the need for creating deeper inheritance hierarchies.

4.3.1.3 Results for NOC (Number of Children)

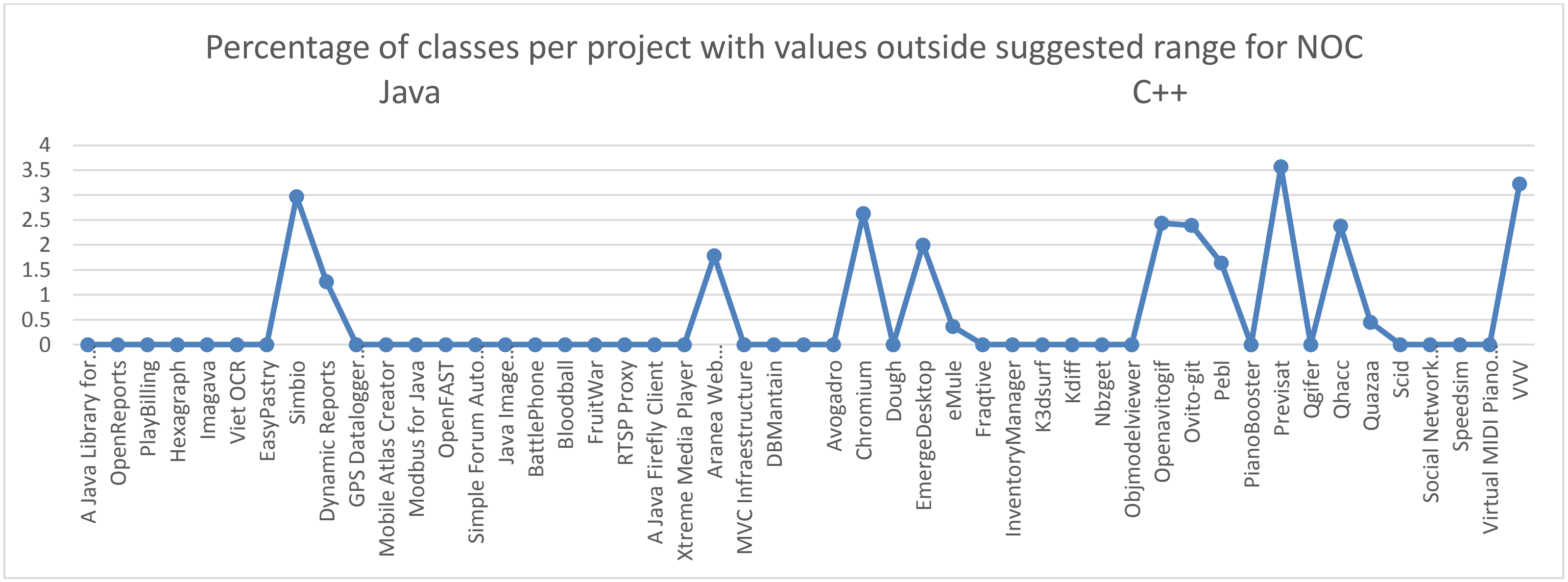


Figure 4.3 NOC results across the projects

The results for the NOC metric are shown in this graphic. The average of the percentage of classes per project with values outside suggested range for Java projects (shown on the left) is 0.25 while the average for C++ projects (shown on the right) is 0.88. Furthermore, out of 24 Java projects, 21 of them (87%) had no classes with values outside the suggested range for NOC compared with 14 (58.3%) for C++. In conclusion, the results for NOC were better for Java. This result may be explained by considering that since C++ supports multiple inheritance, more classes have more children. This is related with the result we obtained for the DIT metric; given that the DIT metric has a better result in C++ it seems that C++ programs tend to favor breadth instead of depth in the inheritance hierarchies.

4.3.1.4 Results for CBO (Coupling Between Object Classes)

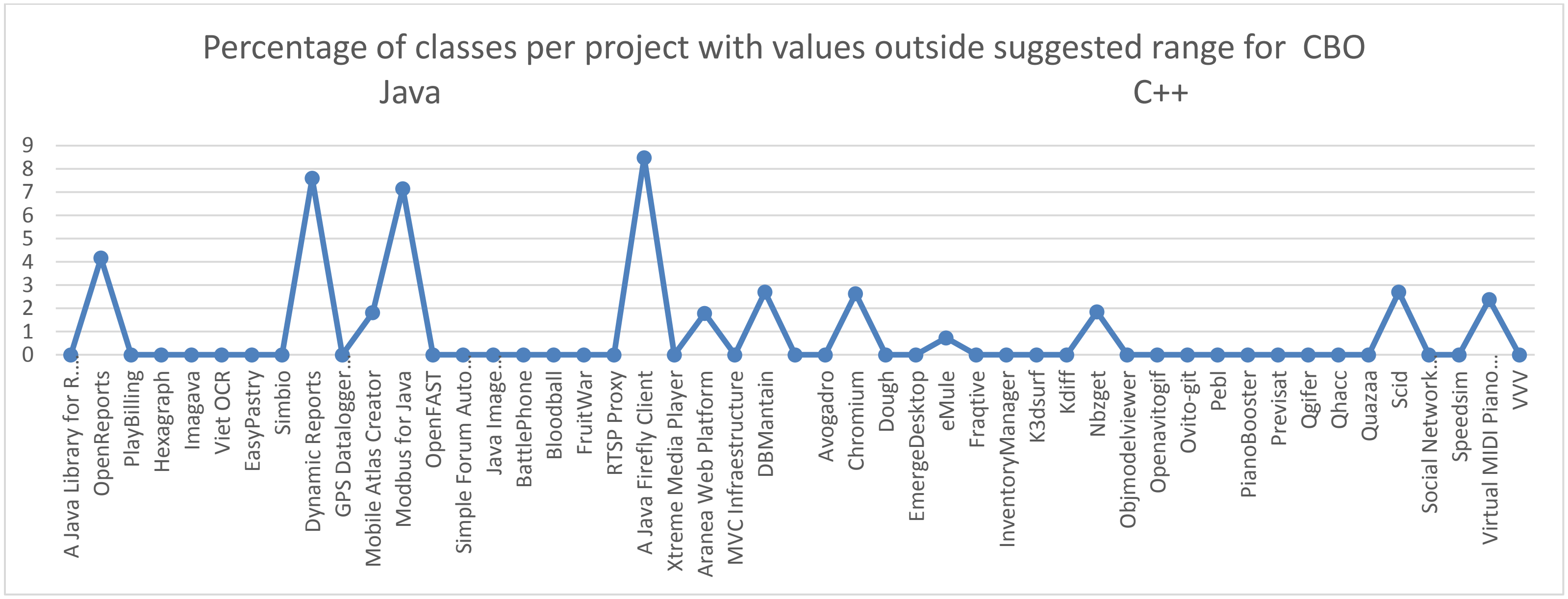


Figure 4.4 CBO results across the projects

The results for the CBO metric are shown in this graph. The average of the percentage of classes per project with values outside suggested range for Java projects (shown on the left) is 1.40 while the average for C++ projects (shown on the right) is 0.43. Furthermore, out of 24 Java projects, 17 of them (70.83%) had no classes with values outside the suggestedrange for CBO compared with 19 (79.16%) for C++. In conclusion, the results for CBO were better for C++. Higher CBO means that there exist more dependencies between classes. One possible explanation for the result is that Java programs usually make a strong use of the extensive Java API so the higher number dependencies may be the result of this.

4.3.1.5 Results for RFC (Response for a Class)

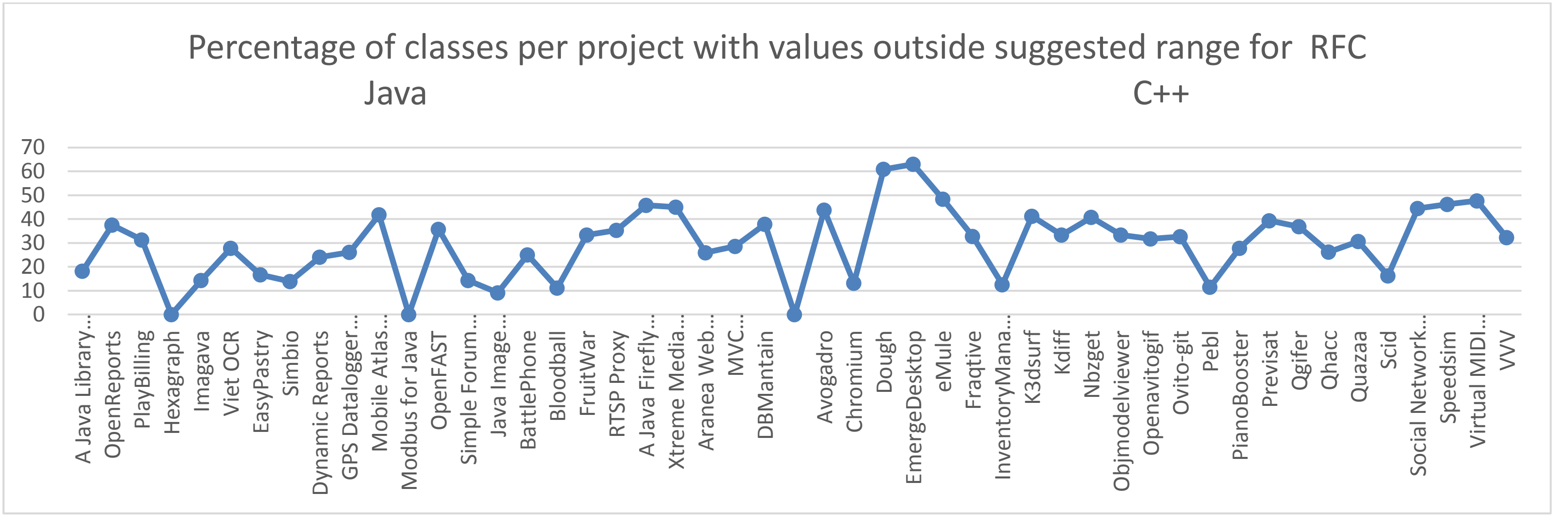


Figure 4.5 RFC results across the projects

The results for the RFC metric are shown in this graphic. The average of the percentage of classes per project with values outside suggested range for Java projects (shown on the left) is 24.93 while the average for C++ projects (shown on the right) is 35.25. Furthermore, out of 24 Java projects, only 2 of them (8.33%) had no classes with values outside the suggested range for RFC compared with 0 (0%) for C++. In conclusion, the results for RFC were better for Java. Although the results for Java were better in comparison to C++, the overall results for RFC are not good for either of the languages since most projects are outside suggested values and the averages are relatively high. The recommended value for RFC is at most 24 but maybe this number is difficult to maintain for complex projects, like the ones that are created today. However, as we mentioned before, the threshold for some metrics was selected based on similar metrics of this set. Therefore, these results can be used to adjust such thresholds.

4.3.1.6 Results for LCOM (Lack of Cohesion of Methods)

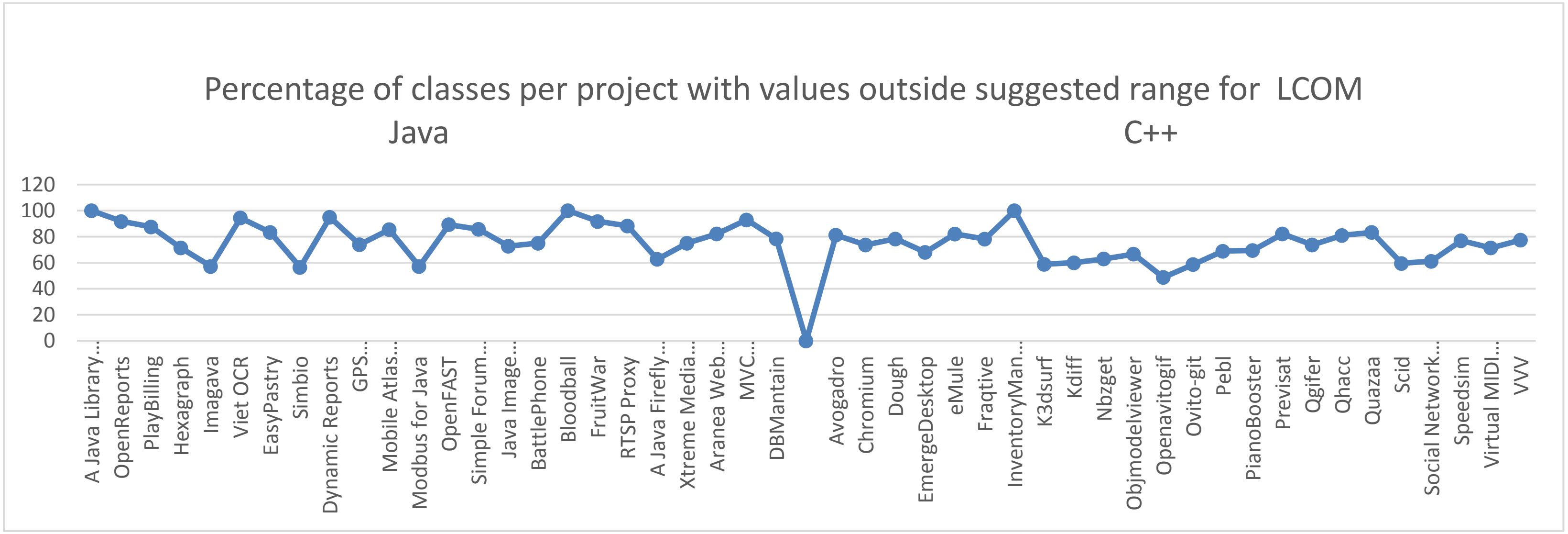


Figure 4.6 LCOM results across the projects

The results for the LCOM metric are shown in this graphic. The average of the percentage of classes per project with values outside suggested range for Java projects (shown on the left) is 81.13 while the average for C++ projects (shown on the right) is 71.75. Furthermore, none of the studied projects both in Java and C++ had classes outside the suggested range for LCOM. Considering the slightly lower average for C++, the results for LCOM were better for C++. Although the results for C++ were better in comparison to Java, the overall results for LCOM are not good for either of the languages since all of the projects are outside suggested values and the averages are high, which means that most of the classes are outside suggested values. It is difficult to explain this metric in terms of language characteristics so it seems that this is more a problem that developers introduce when designing their classes.

4.3.1.7 Analysis of the results for the first study

In the following tables, we summarize and show the results that were previously described. The first table summarizes results for Java while the second table summarizes results for C++. In each table, we highlight in green the best value compared to the other language and in red the worst language compared to the other language.

Average percentage of classes per project with values outside suggested ranges (lower is better)

	WMC	DIT	NOC	CBO	RFC	LCOM
Java	1.80	2.61	0.25	1.40	24.93	81.13
C++	15.58	0	0.88	0.43	35.25	71.75

Table 4.1 Average percentage of classes per project with values outside suggested ranges

Percentage of projects with values within suggested ranges

(Higher is better)	WMC	DIT	NOC	CBO	RFC	LCOM
Java	66%	70%	87%	70%	8.33%	0%
C++	8.3%	100%	58.3%	79.16%	0%	0%

Table 4.2 Percentage of projects with values within suggested ranges

In terms of the average percentage of classes per project with values outside suggested range, there is a match, since each language is better than the other in exactly 3 out of the 6 categories.

In terms of the percentage of projects with values within suggested range, we can see that there is a direct relationship to the average: metrics with good (i.e. low) averages also have the highest percentage of projects with values within suggested range. This is valid for all of the metrics except LCOM where there were no projects within the suggested values. We see that in the second row of the tables that Java is better in 3 out of the 6 metrics while C++ is better in 2 out the 6 categories.

In general, the difference in the values of the averages is not considerable, except for WMC. This means that, at least in the studied projects, C++ classes had considerably more methods than their Java counterparts.

The high average percentages in RFC and LCOM for both Java and C++ and WMC for C++ show that these are the metrics that are more difficult to maintain within an appropriate range.

4.3.2 Results according to average of values of classes for each metric

We now perform an analysis of the values of each metric to understand how much they deviate with respect to the suggested values.

4.3.2.1 Results for WMC (Weighted Method per Class)

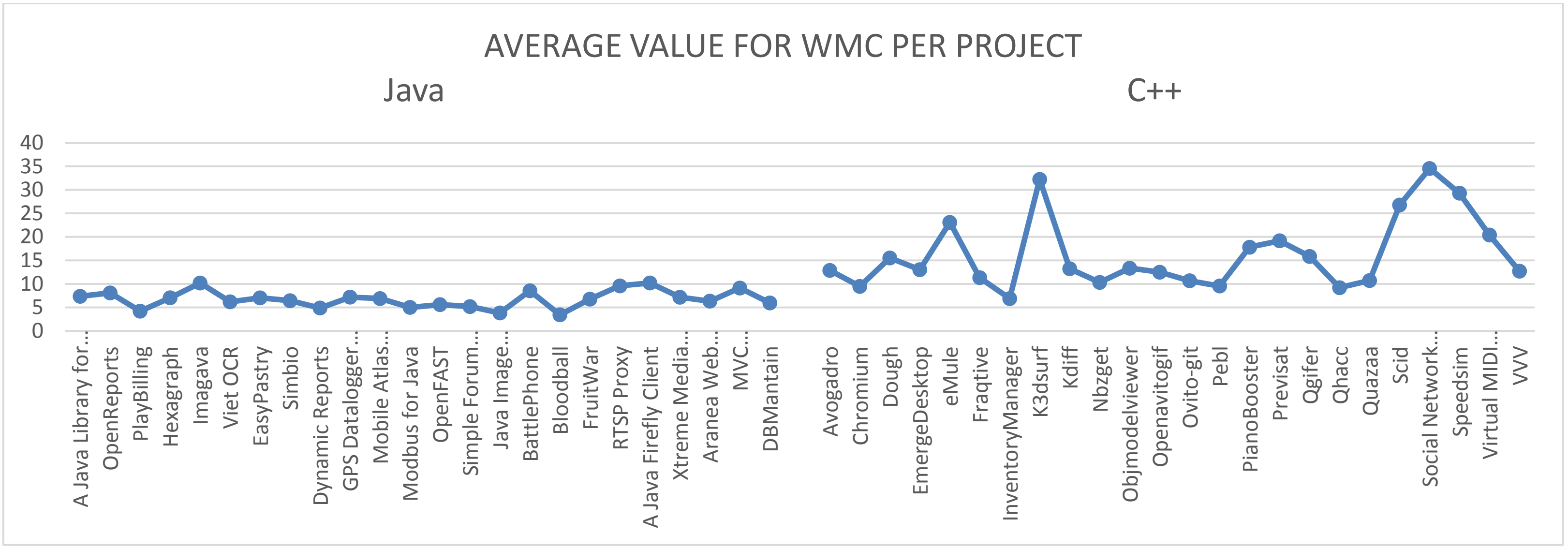


Figure 4.7 WMC results across the projects (second group of results)

In this graph we can observe average WMC values obtained for Java projects (show on the left) and for C++ projects (shown on the right). The average for the values obtained in Java projects is 6.75 while the average of these values for C++ projects is 16.27, so there is a considerable difference between the languages. Since the suggested value range for WMC is <= 24, we see that in general projects are within the range. The difference in the averages between C++ and Java confirms that in general C++ classes are more complex than their Java counterparts in terms of the number of methods. In general, we can conclude that WMC is a metric that is not so well respected in our C++ project sample, with many projects having a significant percentage of classes with values outside the ranges.

4.3.2.2 Results for DIT (Depth of Inheritance Tree)

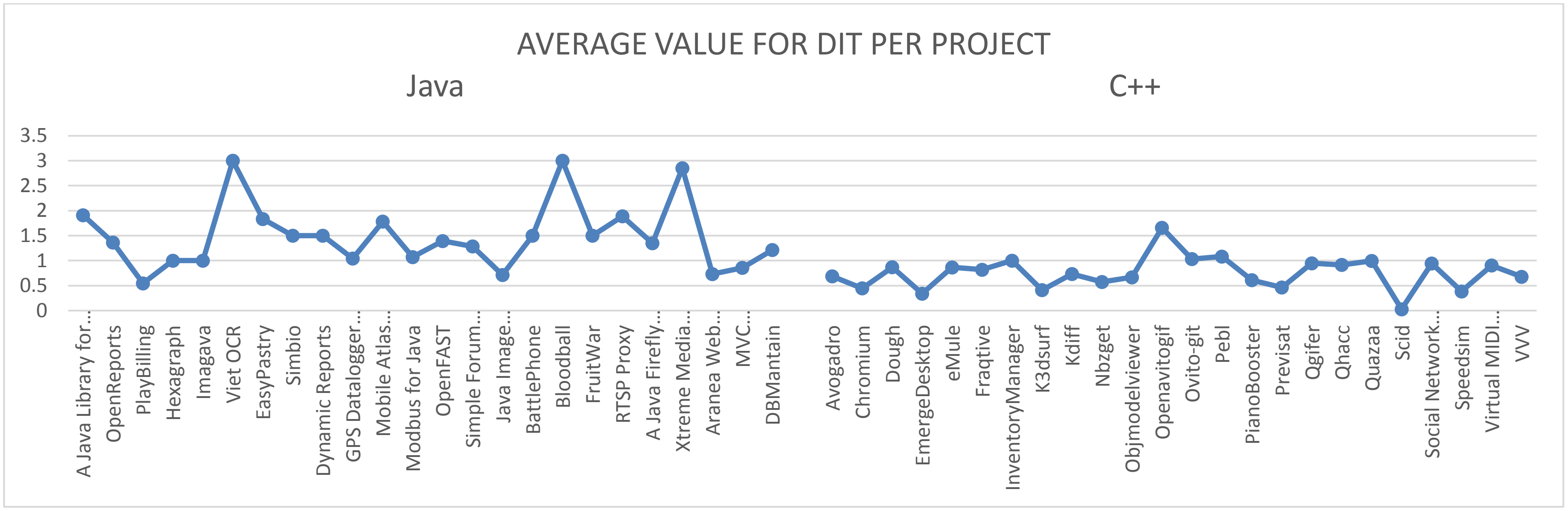


Figure 4.8 DIT results across the projects (second group of results)

In this graph we can observe average DIT values obtained for Java projects (show on the left) and for C++ projects (shown on the right). The average for the values obtained in Java projects is 1.49 while the average of these values for C++ projects is 0.75, so there is no so much difference in the average obtained for each one of these languages. Since the suggested value range for DIT is <= 5, we see that in general projects are within the suggested range even though C++ fares better. In general, we can conclude that DIT is a metric that is very well complied in our C++ project sample, with no projects having any class with values outside suggested range. For Java projects, the metric is less well respected, since a few projects had a small percentage of classes with value outside suggested range but in on average all projects had values within suggested range.

4.3.2.3 Results for NOC (Number of Children)

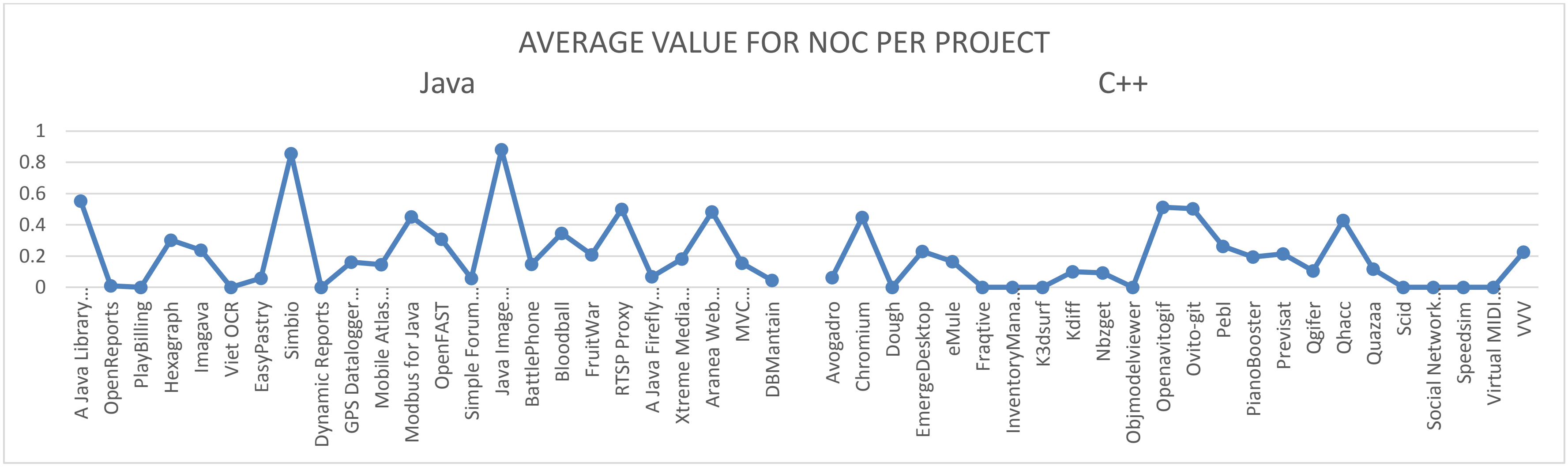


Figure 4.9 NOC results across the projects (second group of results)

In this graph we can observe average NOC values obtained for Java projects (show on the left) and for C++ projects (shown on the right). The average for the values obtained in Java projects is 0.25 while the average of these values for C++ projects is 0.15, so there is no so much difference in the average obtained for each one of these languages. Since the suggested value range for NOC is <= 5, we see that in general projects are within the suggested range even though C++ fares better. Although more C++ projects had values outside suggested range (see 4.3.1.3), on average C++ projects have a better NOC value. In general, NOC is a well respected metric in our project sample since the percentage of total number of classes with values outside suggested value per project did not exceed 4% and the averages for all projects were within value range.

4.3.2.4 Results for CBO (Coupling Between Object Classes)

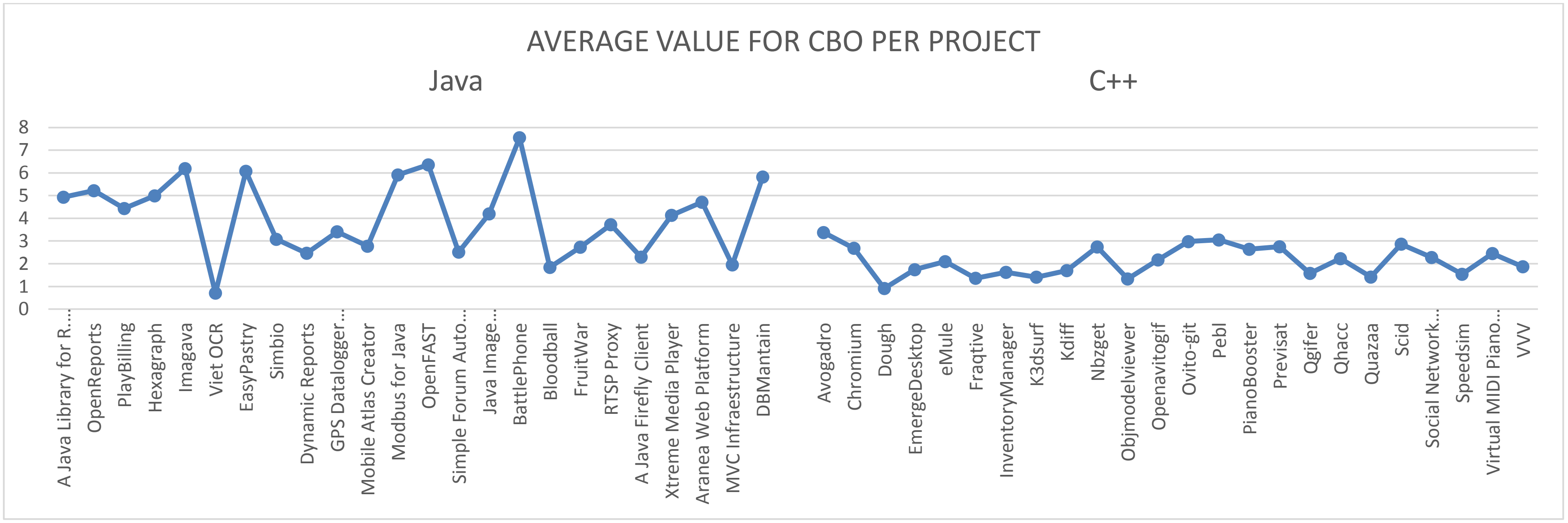


Figure 4.10 CBO results across the projects (second group of results)

In this graph we can observe average CBO values obtained for Java projects (show on the left) and for C++ projects (shown on the right). The average for the values obtained in Java projects is 4.08 while the average of these values for C++ projects is 2.11, so there is no so much difference in the average obtained for each one of these languages. Since the suggested value range for CBO is <= 14, we see that in general projects are within the suggested range even though C++ fares better. In general, CBO is a well respected metric in our project sample since the percentage of total number of classes with values outside suggested value per project did not exceed 8% and the averages for all projects were within value range.

4.3.2.5 Results for RFC (Response for a Class)

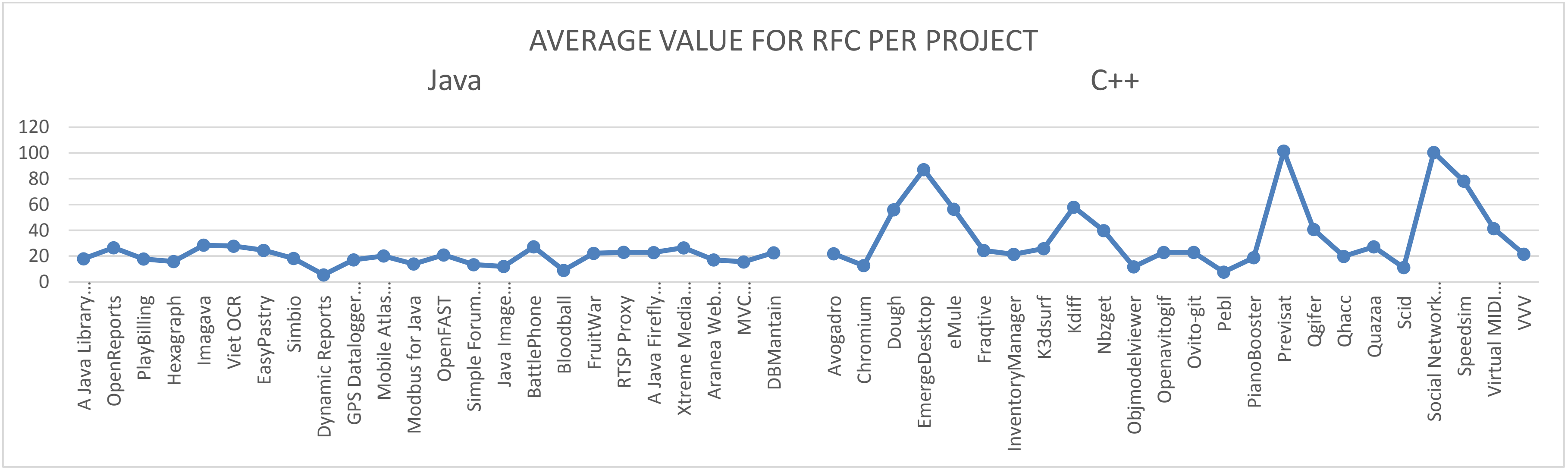


Figure 4.11 RFC results across the projects (second group of results)

In this graph we can observe average RFC values obtained for Java projects (shown on the left) and for C++ projects (shown on the right). The average for the values obtained in Java projects is 19.35 while the average of these values for C++ projects is 38.67, so there is a significant difference in the average obtained for each one of these languages. Since the suggested value range for CBO is <= 24, we see that only Java projects are within the suggested range. In general, RFC is a not a well complied metric in our C++ project sample since there were no projects with any classes with values outside the suggested range (see 4.3.1.5) and, furthermore, the average RFC values for C++ is outside the suggested range. In the case of Java, only 2 projects had no classes with values outside suggested range, but the rest of the projects had an average within the suggested range.

4.3.2.6 Results for LCOM (Lack of Cohesion of Methods)

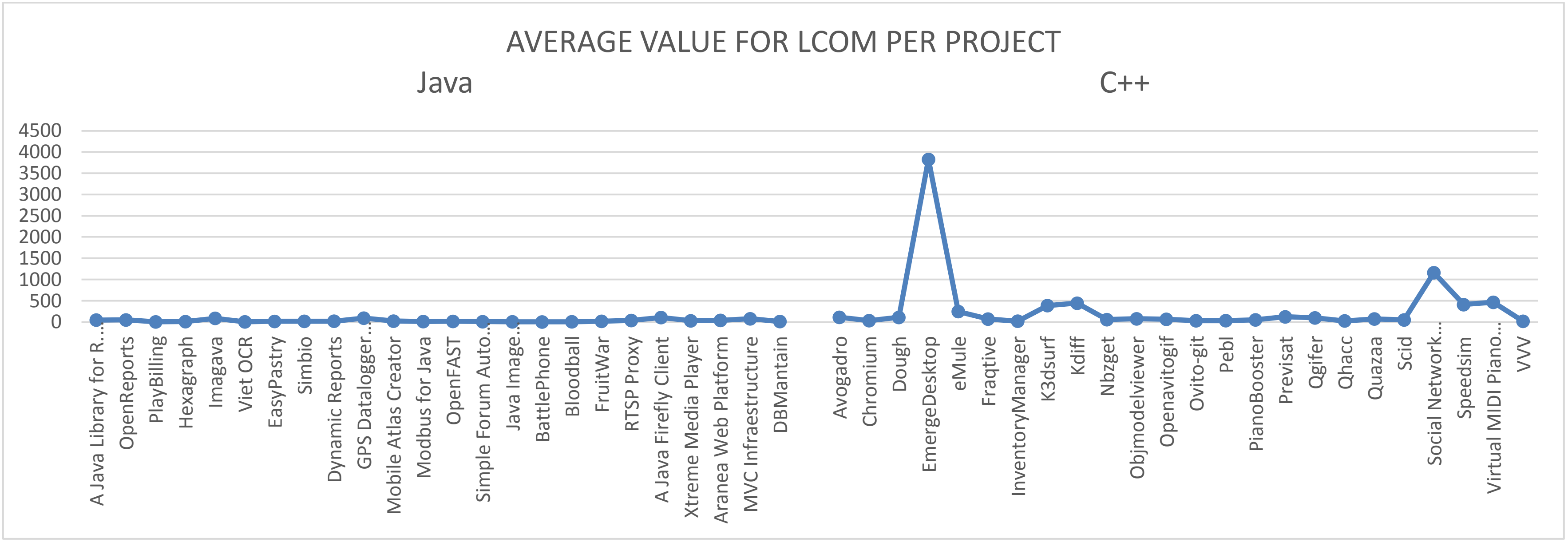


Figure 4.12 LCOM results across the projects (second group of results)

In this graph we can observe average LCOM values obtained for Java projects (shown on the left) and for C++ projects (shown on the right). The average for the values obtained in Java projects is 30.90 while the average of these values for C++ projects is 332.78, so there is a significant difference in the average obtained for each one of these languages. This difference can be explained by a few projects with excessively high LCOM values. Since the suggested value range for LCOM is <= 0, we can see that in our project sample LCOM is respected at all since all projects had a very important percentage of classes with values outside suggested range (see 4.3.1.6) and the average values of LCOM were significantly above the suggested range.

4.3.2.7 Analysis of the results for the second study

In the following tables, we summarize and show the results that were previously described. The first table summarizes results for Java while the second table summarizes results for C++. In each table, we highlight in green the best value compared to the other language and in red the worst language compared to the other language. The second row displays if the average is within the suggested range of values.

Java

	WMC	DIT	NOC	CBO	RFC	LCOM
General averages	🟩 6.757	🟥 1.493	🟥 0.256	🟥 4.084	🟩 19.354	🟩 30.903
Suggested range	<=24	<=5	<=5	<=14	<=24	0
Average as percentage of range limit	28.1%	29.8%	5.12%	29.1%	80.6%	--
Average within the range of suggested values?	✓	✓	✓	✓	✓	✗

Table 4.3 Second general results for Java

C++

	WMC	DIT	NOC	CBO	RFC	LCOM
General averages	🟥 16.272	🟩 0.752	🟩 0.153	🟩 2.116	🟥 38.671	🟥 332.780
Suggested range	<= 24	<=5	<=5	<=14	<=24	0
Average as percentage of range limit	67.7%	15%	3%	15.1%	161%	--
Average within the range of suggested values?	✓	✓	✓	✓	✗	✗

Table 4.4 Second general results for C++

In terms of average of absolute values of classes per project, there is a match, since each language is better than the other in exactly 3 out of the 6 categories. It is interesting to observe that there is not a direct relationship with the results shown in section 4.3.1.7. For example, for the NOC metric Java obtained better results in terms of the percentage of projects with values within the suggested ranges and the average percentage of classes per project with values outside of them, however in terms of the general average, C++ obtained a better result.

In terms of average per project for each metric, we see in Java that the averages for 5 out of the 6 metrics maintain a value within the range of suggested values. By contrast, we have in C++ that the averages for 4 out of the 6 metrics maintain a value within the range of suggested values.

If we compare these averages as percentages of range limit, we can see that there are considerable differences in the languages except for NOC. For WMC, DIT, CBO and RFC, the highest percentage is almost twice the lowest. LCOM cannot be compared as a percentage and there is a huge difference between C++ and Java. This difference, however, is due to a few (6) C++ projects that had excessively bad LCOM values.

4.4 General analysis

Our projects sample shows that both languages seem to favor half of the Chidamber and Kemerer metrics each. From this perspective, it is difficult to conclude if any of the two languages is “better” than the other and results in higher quality programs in terms of maintainability. However, if we take into account the other evaluations in the tables of the two analysis (“Projects with values within suggested range” and “Number of general averages for each metric within the suggested values”), we see that Java has better results than C++, and also if we take into account the metrics where the differences are considerable (WMC, DIT, CBO, RFC and LCOM), we realize that Java is favored in more of those metrics (3). Therefore, if we look into detail on the results, we could say that Java obtains a slightly better result in terms of the quality in object-oriented design. This probably results in programs that are simpler to maintain and modify. The result we present here is valid for the sample that we selected; however, a study with a larger sample is necessary in order to conclude with more certainty.

An interesting result observation is that LCOM is not a well complied metric all of the projects. As we mentioned previously, we believe this metric has more to do with class design than language characteristics.

It is worth mentioning that our results are very similar to the results found in [34] by Michael English and Patrick McCreanor. These authors concluded that classes in C++ systems tend to have more methods, which coincides with our result that WMC is higher in C++ systems than their counterpart in Java. With respect to DIT, they found that average DIT is larger in Java systems than their counterpart in C++ systems, which also coincides with our result. With respect to NOC, these authors found no-significant difference for average NOC in C++ systems and their counterpart in Java systems; however, our result favors this metric in Java (in a normalized way), although of our result, this metric is the one with a tighter result in its average between Java and C++. For CBO metric, their result also shows no-significant difference in the average for C++ systems and their counterpart in Java systems; however, our result favors this metric in C++ (in a normalized way), although of our result, this metric is the one with a tighter result in its “Percentage of projects with values within suggested range” between Java and C++. Finally, for RFC and LCOM, the final results of these metrics in the mentioned work are the same to our results; RFC and LCOM are much higher in C++ systems than their counterpart in Java systems.

4.5 Summary

In this section we have presented our experiment where we used Maya C++ to gather Chidamber and Kemerer metrics data for a set of C++ projects. The results were contrasted with data obtained from a previous study. We performed two different analyses: the first one is independent of the project size while the second is not. We also presented analysis for the data in each of the studies and a general analysis. The results slightly favor Java over C++, however there is not enough evidence to declare that Java is better than C++, we could say that there is a technical draw in terms of quality of object-oriented design.. The following chapter provides the conclusion to this thesis.

Chapter 5. Conclusion

5.1 Introduction

This chapter presents the threats to validity, the general conclusions with respect to the objectives that were established in the introduction of this thesis and also discusses contributions and possible future work.

5.2 Conclusions

The general objective of this project that we presented in the introduction was “to establish if there is a noticeable difference in terms of maintainability as a consequence of the use of a particular programming language”. This main objective was refined into a set of specific objectives for which we now discuss whether they were achieved or not.

The first specific objective was “to understand the general concept of software quality and how the Chidamber and Kemerer metrics suite can be used to support the measurement of maintainability”. We can conclude that we met this objective, given that we conducted a search on papers and websites of interest about quality, software quality and software quality in object-orientation. With this task we extracted information to get into the aforementioned concepts and establish a good base to support our research. We realized how quality (in our case: software quality) can be understood as a subjective term, but we can evaluate it through the construction of a quality model composed by attributes and metrics.

The second specific objective was: “to conduct a study of existing software tools that support the automatic calculation of Chidamber and Kemerer metrics for C++.” This objective was met and the conclusion was that we found no open-source tool providing coverage of the 6 Chidamber and Kemerer metrics to evaluate C++ projects.

This third specific objective was “to develop a tool that calculates the Chidamber and Kemerer for C++ programs”. This objective was met through the development of

Maya C++ a tool whose requirements and architectural design were described in chapter 3.

The fourth specific objective was “to perform an analysis of a sample of Java and C++ programs with respect to the Chidamber and Kemerer metrics”. This objective was met successfully.

In summary, we met all of the specific objectives that were established for the thesis. In the last chapter we also provided a conclusion to our analysis which established that for the project sample that we used, there seems to be a slight difference in favor of Java over C++. The main objective was met; however, the sample that was used needs to be extended to provide a more conclusive answer.

5.3 Future work

As with any research project, new directions appear as the project is performed. In the context of this thesis, these are some areas that can be explored in the future:

- Extend the project sample. As we mentioned in the previous chapter, the project sample was limited. Ideally we need a more extensive project sample. In terms of Java, this is no problem since the percerons website provides an extensive amount of data. In the case of C++, more work is required in order to find the potential candidates.
- Release the tool as an open source project. As we saw in the study of tools that was conducted as part of this project, there are no open source tools that calculate the whole Chidamber and Kemerer set for C++. Releasing Maya C++ as an open source tool does not require much effort, only selecting a particular license and hosting site.
- It would be interesting to conduct a similar study using additional metrics, for example by complementing the Chidamber and Kemerer metrics set with the Li and Henry set in order to observe the differences. This can be complicated by

the fact that the data in the percerons site does not include the Li and Henry metrics and it would also require modifications to the Maya C++ tool.

- It would be interesting to extend the comparison to other object oriented languages. This, of course, would require more extensive work but would be interesting nonetheless.

References

1: D. Garvin, "What Does "Product Quality" Really Mean?" Sloan Management Review, pp. 25-45, Fall 1984

2: B. Kitchenham; S. L. Pfleeger, "Software quality: the elusive target", Software, IEEE Computer Society, 13 (1), pp.12-21, January 1996

3: Hans van Vliet, Software Engineering: Principles and Practice, (c) Wiley, 2007

4: C. Van Koten and A.R. Gray, "An Application of Bayesian Network for Predicting Object-Oriented Software Maintainability", Inform Software Tech, pp. 59–67, Jan. 2006

5: http://www.sqa.net/iso9126.html - ISO 9126 Observations - July 2013

6: Ho-Won Jung; Seung-Gweon Kim; Chang-shin Chung, "Measuring software product quality: a survey of ISO/IEC 9126" Software, IEEE, vol.21, no.5, pp.88-92, Sept.-Oct. 2004

7:www.bth.se/com/besq.nsf/(WebFiles)/CF1C3230DB425EDCC125706900317C44/$FILE/chapter_1.pdf - Software Quality Models and Philosophies - June 2013

8: J. Bansiya; C. G. Davis, "A hierarchical model for object-oriented design quality assessment," Software Engineering, IEEE Transactions on, vol.28, no.1, pp.417, Jan 2002

9: B. Kitchenham, What's up with software metrics? – A preliminary mapping study, Journal of Systems and Software, January 2010

10: R. Jabangwe, J. Börstler, D. Smite and C. Wohlin, "Empirical Evidence on the Link between Object-Oriented Measures and External Quality Attributes: A Systematic Literature Review", Accepted for publication in Empirical Software Engineering: An International Journal, published online March 8, 2014

11: S.R. Chidamber; C.F.Kemerer, A metrics suite for object oriented design, Journal IEEE Transactions on Software Engineering, pp. 476-493, 1994

12: A. Ampatzoglou, A. Gortzis, I. Deligiannis and I. Stamelos, "A methodology on extracting reusable software components from Open Source Games", ACM Proceedings of the 2012 MindTREK Conference (MindTREK 2012), 3-5 October 2012

13: A. Ampatzoglou, O. Michou and I. Stamelos, "Building and Mining a Repository of Design Pattern Instances: Practical and Research Benefits", Entertanment Computing, Elsevier 2013

14: https://scitools.com/ - Understand for C++ - August 2014

15: http://www.spinellis.gr/sw/ckjm/ - CKJM - August - 2014

16: Bob Glushko, Models and measure of quality, http://courses.ischool.berkeley.edu/i290-1/f08/lectures/ISSD-20081119.pdf , November 19 2008

17: Rubey, R. J.; R. D. Hartwick, "Quantitative Measurement of Program Quality," Proceedings, ACM National Conference, pp. 671-677, 1968

18:http://www.sqa.net/softwarequalityattributes.html - Software Quality Attributes - July 2013

19: J.A. McCall, P.K. Richards and G.F. Walters, "Factors in Software Quality", Nat',l Tech. Information Service, Springfield, Va., 1977

20: http://www.iso.org/iso/home.html - ISO standards - August 2014

21: Ho-Won Jung; Seung-Gweon Kim; Chang-shin Chung, "Measuring software product quality: a survey of ISO/IEC 9126," Software, IEEE, pp.88-92, Sept.-Oct. 2004

22: http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigms.html - Programming Paradigms - July 2014

23: W.Li; S. Henry, Object Oriented Metrics which predict maintainability, Journal of Systems and Software - Special issue on object-oriented software, pp. 111-122, 1993

24: M. Bunge, Treatise on Basic Philosophy: Ontology I: The Furniture of the World. Boston: Riedel, 1977

25: http://www.aivosto.com/project/help/pm-oo-ck.html - Chidamber & Kemerer object-oriented metrics suite - July 2013

26: http://www.ndepend.com - NDepend - July 2013

27: http://www.cppdepend.com - CppDepend - July 2013

28: http://www.scitools.com - scitools Understand - July 2013

29: http://www.campwoodsw.com/sourcemonitor.html - Campwood Software - July 2013

30: http://cccc.sourceforge.net/ - CCCC C and C++ Code Counter - July 2013

31: http://www.sei.cmu.edu/architecture/start/glossary/ - Glossary SEI - March 2015

32: Software Architecture: Principles & Practices - ©2014 Carnegie Mellon University

33: P. Runeson, Martin Höst; Guidelines for conducting and reporting case study research in software engineering; Empir Software Eng (2009) 14:131–164

34: M. English; P. McCreanor, Exploring the Differing Usages of Programming Language Features in Systems Developed in C++ and Java, Limerick-Ireland, 2009