Computing and Informatics, Vol. 35, 2016, 497–527

ASSESSING COGNITIVE COMPLEXITY IN JAVA-BASED OBJECT-ORIENTED SYSTEMS: METRICS AND TOOL SUPPORT

Marco Crasso

IBM Research
Mar del Plata, Buenos Aires, Argentina
e-mail: `mcrasso@gmail.com`

Cristian Mateos, Alejandro Zunino

ISISTAN – CONICET, UNICEN University
Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina
e-mail: {`cmateos, azunino`}`@isistan.unicen.edu.ar`

Sanjay Misra

Atilim University, Ankara, Turkey
e-mail: `ssopam@gmail.com`

Pablo Polvorín

ProcessOne, Paris, France
e-mail: `pablo.polvorin@gmail.com`

Abstract. Software cognitive complexity refers to how demanding the mental process of performing tasks such as coding, testing, debugging, or modifying source code is. Achieving low levels of cognitive complexity is crucial for ensuring high levels of software maintainability, which is one of the most rewardful software quality attributes. Therefore, in order to control and ensure software maintainability, it is first necessary to accurately quantify software cognitive complexity. In this line, this paper presents a software metric to assess cognitive complexity in Object-Oriented (OO) systems, and particularly those developed in the Java language, which is very popular among OO programming languages. The proposed metric is based on a characterization of basic control structures present in Java systems. Several algorithms to compute the metric and their materialization in the Eclipse IDE are also introduced. Finally, a theoretical validation of the metric against a framework specially designed to validate software complexity metrics is presented, and the applicability of the tool is shown by illustrating the metric in the context of ten real world Java projects and relevant metrics from the well-known Chidamber-Kemerer metric suite.

Keywords: Software cognitive complexity, basic control structures, Java, CK metric suite

1 INTRODUCTION

IEEE refers to software quality as "the degree to which software possesses a desired combination of attributes (e.g. reliability, interoperability)" [1]. In an ideal world, it would be desirable to optimize all attributes, but in practice attributes may conflict among themselves. This situation is commonly known as a *trade-off*. Thus, it is necessary to assess and somehow to adjust the values these attributes take to deal with such trade-offs. The main goal of software metrics is to allow software engineers to assess software attributes and in turn to make modifications on the software based on metrics values (i.e. measures). For example, in [28] some metrics for preventing software defects were proposed, specifically those reported by customers when using a software and those identified during customer acceptance testing.

In the early 80s, the software maintainability attribute motivated radical changes in the way software was modeled, and concretely in the way source code elements were organized to represent a computational solution to a set of requirements. The widespread acceptance of the Object-Oriented (OO) programming paradigm backs up the previous statement. Due to the popularity of OO notions, many OO programming languages have been developed, such as Smalltalk, C++, and Java. Accordingly, many software projects have been developed using such languages. Since it is important to assess the maintainability of these software projects, several OO metrics that cover some inherent modeling features of the OO paradigm can be found in the literature, which are listed in [17].

The complexity of OO source code is an attribute related to software maintainability [13]. Particularly, Kearney and his colleagues [14] state that complexity relates to "*the difficulty of performing tasks such as coding, debugging, testing, or modifying a software*". Under this definition, complexity directly impacts on software maintainability. Then, "complexity" should be understood as *cognitive com-*

plexity, but for readability from now on we will use "complexity" and "cognitive complexity" interchangeably.

Source code complexity has been commonly assessed by thirty-year old software metrics such as Halstead's complexity [10], McCabe's cyclomatic complexity [19], or variants such as Hansen's measure of complexity [11]. Although all these metrics were designed for procedural programming and actually appeared before OO languages, they can also be used to characterize OO source code and are still very popular among software engineers. McCabe's cyclomatic complexity metric bases on common procedural control structures (e.g. IF-THEN-ELSE, WHILE, etc.) that are also present in OO source code. Halstead's metric bases on the number of operators (method names, arithmetical operators) and the number of operands (variables, and numeric/string constants), which also apply to object orientation. Lastly, Hansen developed a paired metric that combines the McCabe's complexity metric with operators similar to that of Halstead's.

Nowadays, however, many researchers claim that these metrics are not enough to cope with today's software. Consequently, some new metrics for measuring OO software complexity have been proposed [3]. Particularly, [22] introduced a metric for measuring the cognitive effort required to understand an OO software system. The metric considers not only the basic control structures in source codes, but also the inheritance relationships among classes and method calls. With regard to basic control structures, [22] exploits the work of [24] – which describes a characterization of ten basic control structures – to assess the effort introduced by the program statements that control the flow of a program. The metric considers inheritance relationships and intra/inter class method calls. Besides, a formulation to combine the metric values of isolated control structures by taking into account inheritance and method calls was presented, along with a preliminary evaluation involving a small system developed in C++.

Moreover, this paper proposes a metric for assessing the complexity of OO programs written in Java, which is based on an extension of [24]. Concretely, we present six new basic control structures, four of which have been designed to consider special operators of the OO paradigm, while another two consider the Java language operators for handling exceptions. This metric takes a more integral approach to complexity measurement compared to similar recent metrics [3], which as stated by their own authors are still under development (i.e. [4]) or are not designed to capture all the aspects involved in class design and implementation (i.e. [20]). Besides, because of the huge amount of hours per man manually calculating the proposed complexity metric would require, and because manually computing metrics even from small projects is an error prone task, this paper also presents a tool for automatically gathering basic control structures from Java projects and calculating their complexity. Therefore, the main contributions of this paper are:

- a complete characterization of basic control structures for OO/Java programs,
- a complexity metric that builds upon these basic control structures,
- an open source tool based on Eclipse for automatically computing the metric.

The rest of the paper is organized as follows. The next section discusses the most relevant related work. Section 3 presents the basic control structures for OO and Java programs, and describes the proposed complexity metric for OO systems. Section 4 describes a software tool, implemented as an Eclipse plug-in, for automatically computing this metric from Java projects. Section 5 validates the metric using the validation framework in [21], and illustrates the feasibility of the tool with ten real world Java projects from the Apache Software Foundation. The section also compares our metric with relevant metrics from the well-known Chidamber and Kemerer's metric suite. Future research directions are described in Section 6. Lastly, concluding remarks are presented in Section 7.

2 BACKGROUND AND RELATED WORK

This section starts by revisiting some concepts of the OO paradigm and Java. Then, we present a conceptualization and examples of basic control structures for Java.

2.1 Particularities of the OO Paradigm in the Context of Java

By definition, an object is a class instance. A class is a "blueprint" defining the attributes and operations (methods from now on) for instances of that class. An OO system consists of one or more classes plus objects communicating among them via message exchange (i.e. method calls). New classes (a.k.a. the children) may be defined by inheriting attributes and method definitions from an existing class (a.k.a. the base class or parent class), thus conforming a class hierarchy. Then, inherited attributes and methods can be accessed from children classes. For example, if a child class C inherits the definition of a method s from a parent class S, then s may be called from methods defined in C.

Inheritance is a mechanism commonly combined with *abstract* methods. Contrarily to concrete methods, an abstract method lacks implementation. When a class has at least one abstract method, it is called *abstract* and cannot be employed for instantiating objects. Only children classes providing implementations for the definitions of all inherited abstract methods can be employed for creating objects. When a child method provides an implementation for an abstract method, it is said that the former *overrides* the latter. Although abstract classes cannot be employed for the instantiation of objects, they can be used in method signatures and variable/-attribute definitions. At run-time, these elements must be instantiated using one of the available children inheriting the abstract classes.

The Java language complements the OO paradigm with exception handling, a mechanism that allows developers to handle situations that fall outside the expected or normal execution flow of program. In Java, the exception handling mechanism consists of a try block, a catch block, and optionally a finally block. A try block is the set of statements whose execution must be monitored for exception occurrences. A catch block, which is associated with each try block, is a sequence of catch clauses. Each catch clause defines the type of exception it is able to handle, and contains a block of code that is executed when an exception of that type is caught. Lastly, each try statement can have a finally block, which is always executed irrespective of an exception has occurred or not.

Another interesting feature of the Java language are *anonymous classes*. An anonymous class is a local class without a name, which can be included as part of a larger expression such as a method call. Then, anonymous classes are defined and instantiated in a single succinct expression using the new operator. Moreover, developers declare an anonymous class exactly in the statement where the class is needed.

For the purposes of this paper, it is worth describing the Java *package arrangement* mechanism. This mechanism allows developers to include many classes in an individual file, commonly a zipped .jar file (Java ARchive), which is useful for distributing and reusing such classes. A jar file may contain both Java source code files and Java bytecode files, i.e. compiled source codes. Then, one or more jar files can be imported from any Java program and, in turn, class definitions present in the jar file can be employed to instantiate objects or to act as the base class for new classes. In the rest of the paper we will refer to those classes in a jar file as "library classes". When a jar file only contains Java bytecode – but not source code – we will refer to it as a "closed library".

2.2 Software Engineering Metrics for Measuring Java and OO Systems

In [18] the authors present a new approach for detecting non-cohesive classes in Java. They state that if two different clients of a class have very different client views of the class they are using, it can be assumed that the class does not implement a maximally cohesive set of features. To detect this kind of lack of cohesion, the authors propose the LCIC (Lack of Coherence In Clients) metric that measures how coherently the clients use a given class. The metric bases on the hypothesis that a class is cohesive when all clients use the same features of the class, i.e. clients have a similar view of the class. Otherwise, the class might have unnecessary features, or it might implement several different abstractions [18].

The work also presents a prototype for automatically gathering the metric from Java source files. The prototype parses Java source files, and builds a model representing class attributes and methods. Built models are persisted using an XML database, which is then queried during metric computation [18]. This prototype has been employed for assessing the LCIC metric from three open source projects. The experiments show preliminary results about the relationship between the LCIC metric and possible code refactorings that are known to improve cohesion. Concretely, the authors analyze the LCIC metric for several classes before and after performing well-known refactorings over them.

Beyond the Java arena, one of the most recognized efforts to assess complexity in OO systems is the metric suite developed by Chidamber and Kemerer [7], which is popularly known as the "CK metric suite". The suite consists of six individual metrics: Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Object classes (CBO), Response For a Class (RFC), and Lack of COhesion in Methods (LCOM). The WMC metric counts the methods of a class. The DIT metric is defined as the maximum inheritance path from a class to the uppermost class in the hierarchy. The NOC metric counts the number of classes in a class hierarchy. The CBO metric counts how many methods or attributes defined by other classes are accessed by a given class. The RFC metric counts the methods that can potentially be executed in response to a message received by an object of a given class. The LCOM metric measures how well the methods of a class are semantically related to each other. Interestingly, all the metrics in the CK metric suite are straightforward and simple to compute.

On the other hand, the CK metric suite does not cover the aggregated complexity of a class due to all the measured factors, namely number of methods, classes hierarchical depth and cardinality, coupling and cohesion. The internal architecture of a class in terms of object orientation and the relationships due to inheritance in the class hierarchy, along with the involved complexities, are also ignored. Lastly, CK metrics counts only the number of methods coupled with other classes, ignoring their complexities.

The lack of the above features in the CK metric suite motivated the work presented in [22], which introduces a new metric suite that complements the CK metric suite. The core idea of the former metric suite is to look inside method statements for inferring the control flow of a block of code and hence better assess complexity. To do this, method statements are categorized under the concept of basic control structures. The next section focuses on describing this concept in detail.

Other similar metrics in line with the suite presented in [22] are Extended Weighted Class Complexity (EWCC) and Average Complexity of a program due to Inheritance (ACI). EWCC [4] is an indicator of class complexity due to inheritance that is very close to [22], but as the authors of the EWCC metric assert, the metric bases upon formulae that are not well-defined yet [3]. This makes the metric immature from a practical standpoint. Moreover, ACI [20] has been also proposed recently. To determine class complexity, ACI considers the complexities of the classes inherited by a class together with the complexities of the methods defined by this latter. Method complexity is computed by using classical metrics (e.g. CK) and does not consider the kind of method being called (e.g. local, third-party library, polymorphic, etc.). Like the core formalisms underpinning the metric presented in this paper, the ACI metric has been validated by means of Weyuker's properties [30], a theoretical validation framework for complexity metrics that defines desirable mathematical properties to which a software engineering metric should adhere.

Another work close to ours is [9], in which the authors propose a class complexity metric based on three submetrics: Total Method Call Complexity (TMCC), Total Data Call Complexity (TDCC) and Control Flow Complexity (CFC). TMCC considers the complexity due to non-polymorphic (i.e. static methods) and polymorphic (i.e. object) method calls. At the same time, the depth of inheritance trees is considered. Similarly, TDCC computes the complexity due to accessing static and instance data members, which also takes into account member hierarchical relationships. Lastly, CFC is self-descriptive and is computed using the McCabe's cyclomatic complexity metric. A major difference between [9] and our work is that we define a more integral approach to compute CFC, and besides we propose a way of calculating complexity at the system-level.

In a different direction, as discussed in [5], several authors conceive that the use of a single metric suite is not enough to assess a certain software quality attribute, and claim that quality should be derived from aggregating several related metrics and metric suites. For this purpose, software quality prediction models have been extensively used, which exploit classic machine learning techniques (e.g. C4.5 trees) and other artificial intelligence algorithms (e.g. genetic algorithms and ant colony optimization). Rather than proposing new metrics, the efforts in this area aim to improve the accuracy of these models. Building accurate prediction models is in general difficult due to lack of data in the software engineering domain. Therefore, prediction models built for a particular software do not accurately perform when used with metric values from another software [5].

From these models, the work in [29] uses a genetic algorithm to improve a model accuracy in identifying components whose maintainability could be compromised. The approach relies on the coupling, cohesion, inheritance, complexity and size metrics from the CK suite that are more likely to improve the performance of predictive models according to the author's experiments. Notice that since we propose an alternative way of measuring complexity, our work – as well as other works in that direction – complements [29] since our work represents another metric upon models can be built.

2.3 Basic Control Structures

Category	Basic Control Structure(s)	Weight
Sequence	Sequence	1
Branch	If-then-[else]	2
Branch	Case	3
Iteration	For-do, Repeat-until, While-do	3
Embedded component	Function call	2
Embedded component	Recursion	3
Concurrency	Parallel, Interrupt	4

Table 1. Basic control structures and weights presented in [24]

When analyzing source code complexity it is important to consider individual class-level metrics such as number of methods, number of messages, and number of different classes. The internal complexities of these classes are equally impor-

Listing 1. Basic control structure based weight: Example

```
1 short a=b;
2 short c=d;
3 if (a<c) {
4           d=a;
5           this.f();
6 } else {
7     a=c;
8 }
```

tant [10]. Shao and Wang [24] proposed a software metric that associates an individual weight to every basic control structure in a system. As defined by the authors, "Basic control structures are a set of essential flow control mechanisms that are used for building logical software architectures" [24], and basic control structures can be employed to characterize internal control flows in a source code. Then, the authors establish a weight for each basic control structure so that the higher the associated weight the higher its complexity. Basically, the idea is that a set of program statements arranged in a sequence is more understandable than those that, for example, are branched in two or more control flows. Table 1 summarizes the basic control structures identified by the authors and their associated weights [24]. Moreover, as explained in [2], the values for these individual discrete weights were obtained from empirical studies on cognitive informatics. These studies allowed the author to establish an ordered ranking of the identified basic control structures based on a measure (i.e. weights) of the human effort necessary to capture the structure of different basic control structures. All weights are relative to the Sequence basic control structure, which is assumed to be the less effort-demanding basic control structures.

For the sake of exemplification, we will analyze the weight W for the code shown in Listing 1, which is equal to $1 + 2 * [2] = 5$, where 1 is for the sequence (lines 1–2, which are conceived as a single block) and 2 is for the if branch structure (lines 3–7). There is a call to a local method inside the if body (line 5), adding 2 to the calculation. Notice that since there are nested structures, we multiply the weights instead of summing them up. This procedure is repeated for each individual system module (e.g. functions, procedures, and methods) and in turn summed, in order to assess the complexity of the entire system.

As shown in Table 1 the basic control structures proposed in [24] covers a wide range of control structures present in programs developed in various paradigms and languages. However, some basic control structures that are commonly used in OO source code have not been included in the previous set and deserve to be analyzed. For example, some loose points that this paper discusses is whether the cognitive effort of a call to an abstract method and a call to a concrete method are the equal, and whether a call to inherited methods should be treated different than calling methods from other class hierarchies, or as proposed in [22] whether class attributes should be considered when computing class complexity. Therefore, we argue that the basic control structures from [24] are not enough for assessing the complexity of Java-based OO software systems.

3 NEW BASIC CONTROL STRUCTURES AND METRIC SUITE FOR ASSESSING THE COMPLEXITY OF JAVA SYSTEMS

In our view, the cognitive complexity of an entire OO system is calculated by aggregating the individual complexities of classes [22]. In the same line, method complexities are calculated by analyzing the complexity of each code statement in terms of basic control structures, while computing the complexity of a class consists mainly on calculating method complexities along with the class inheritance relationships.

In accordance with the basic control structures categorization presented in [24], in an OO method there are control structures belonging to the following categories: Sequence, Branch, Iteration, Embedded component and Concurrency. Particularly, we propose to extend the Embedded component category for differentiating specific conditions under which a method can be called. We established weight factors for methods calls according to whether the method is *local* or *non-local.* When a called method definition is owned by the same class of the calling object or by one of its parent/child classes, the call is on a *local* method. Instead, we refer to non-locally defined methods to those methods whose definition belongs to another class in a different hierarchy than that of the calling class. Indeed, since sometimes classes are packaged into libraries for reuse purposes, and the source code of third-party libraries may not be available, we also discriminate called methods belonging to closed library classes. We will refer to these methods as *external.*

Lastly, abstract methods can be also called and the cognitive effort associated with calling them is somehow determined by the complexities of the available implementations of those methods. Concretely, an abstract method m from an abstract class S having C concrete children, means that there are C alternative implementations for sending the message m to a variable of type S. Then, the effort required to understand, modify, or test that method call is different from the one required for calling a local method and as such it should be differentiated.

To sum up, our extension to the Embedded component category of [24] (see row 8 of Table 1) bases on differentiating and determining the cognitive effort required to understand a call to a local method (i.e., when it is defined by the same class or class hierarchy than the caller method), a call to a non-local method (i.e. methods defined in a different hierarchy than that of the calling class), a call to an external method, and a call to an abstract method of a polymorphic object. In summary, our first extension to [24] focuses on the Embedded components category and covers the next basic control structures:

1. calling local methods;
2. calling non-local methods;

3. calling external methods (i.e. those present in closed libraries); and
4. calling abstract methods.

Second, we propose that the Sequence and Branch categories of basic control structures shown in [24] should be extended to be applicable to programs in Java, for accounting the effects of exception-handling constructs and exception occurrences. The rationale of this decision is that throwing an exception disrupts the normal control flow of a program, and thus increases the complexity of the code. Assuming that no catch block is declared, a try block, which must be accompanied with a finally block, is conceptually equivalent to a common sequence control structure. This is because assuming that no exception occurs, by default neither a try block nor a finally one alter the normal control flow. Despite this similitude between try/finally control structures and sequences, the former deserves having a different cognitive weight since understanding try/finally structures requires at least to be familiar with the semantics of the Java exception handling mechanism. Besides, flow disruption may occur. On the other hand, one or more catch blocks added to the try block mean that a program flow may be further branched due to potential exceptions. Therefore, catch*[catch] blocks should be given a different weight.

Category	Basic Control Structure	Weight
Embedded component	Call to a local (including super) method	2
	Call to a non-local method m	$2 + W_m$
	Call to a method m of an external library	3
	Call to an abstract method a, for which $a_0, \ldots, a_n$ override a	$I + f(W_{a_x})$
Sequence	Try-finally (no catch block)	2
Branch	catch *[catch]	3

Table 2. OO and Java basic control structures introduced in this paper

Table 2 shows the six new basic control structures introduced in this paper, which have been characterized using the basic control structure taxonomy defined in [24]. The third column of the Table presents the associated weighs for each new basic control structure. As the reader can see, two weights have been defined in terms of symbolic variables.

The weight of calling local methods is 2, which is derived from the weight given to the Function call basic control structure in [24]. As explained, this encompasses methods defined in the calling class or in parent classes of the former. On the other hand, the weight of non-local methods is defined as $2 + W_m$. This is because in [24] Function calls structures have a weight of 2 (because of the call itself), and non-local methods complexity (W_m, Equation (1)) is calculated and then summed [22]. Note that the weight of non-local methods is in fact $2 + W_m$ with $W_m = 0$. This is since the weight of locally defined methods is omitted [22], and the cognitive effort of the target method is not accounted directly in its weight but in the metrics explained below, which take into account hierarchical relationships. Furthermore, the weight of calling a method of an external closed library is 3, since it is a Function call, plus a symbolic weight of 1 due to the effort of understanding the signature of the method being called. Recall that for closed libraries the source code is not available, and thus W_m cannot be computed as we do for non-local methods.

The weight associated with calling an abstract method is defined in terms of those methods that override it. The associated weight is the sum of the weights of all overriding methods ($SUM(W_{a_x})$), their average ($AVG(W_{a_x})$), the maximum weight ($MAX(W_{a_x})$) or the minimum weight ($MIN(W_{a_x})$). The software engineer who analyzes the complexity of a system should select among these alternatives to determine the weight of the basic control structure. For example, the second alternative (i.e. $AVG(W_{a_x})$) could be selected provided the dispersion of complexities among overriding methods is not large, otherwise the average would not be not significant enough from a probabilistic standpoint.

The weight associated with a try block and optionally a finally block (without considering potential catch blocks) is defined as 2, and a weight of 3 is given to the list of catch blocks associated to a try-[finally] block. Even when try/finally has in essence a sequential structure, it allows the code to abruptly jump to the finally block in case of exceptions, so we weight similarly to the If-then-[else] basic control structure. On the other hand, a list of catch blocks can be viewed as a Case basic control structure: depending on the type of the exception, a different code block is executed.

Cognitive weights are used to measure complexities of methods once their basic control structures have been identified. Equation (2) formally defines the complexity of a single method (MC). We consider that any method comprises a Sequential basic control structure that represents its entire body and at the same time q linear basic control structures contained in the Sequential basic control structure (*methodsBCSs*). Then, the complexity of a method is defined as two plus the sum of cognitive weights of a list of arbitrary basic control structures recursively, since each basic control structure may in turn consist of layers of nested basic control structures. At each layer, again, the linear basic control structure weights are summed. In Equation (1), $W(BCS_j)$ is the cognitive weight of the involved basic control structures as established in Tables 2 and 1.

$$MC'(BCSs) = \sum_{j=1}^{q} \begin{cases} W(BCS_j) & BCS_j = \text{branch, iteration} \\ \quad * MC'(\text{innerBCSs}(BCS_j)) & \text{try/finally catch*[catch]}, \\ W(BCS_j) & \text{otherwise}, \end{cases} \quad (1)$$

$$MC = \log_2(1 + [1 + MC'(\text{methodBCSs})]). \quad (2)$$

The Weighted Class Complexity (WCC) metric goes one step beyond and computes the complexity of an entire class. The following equations calculate the

Weighted Class Complexity (WCC) of a class with m methods:

$$WCC' = AC + \sum_{i=1}^{m} MC'_i, \tag{3}$$

$$WCC = \log_2(2 + WCC'). \tag{4}$$

WCC' is the sum of all the method complexities of a class, and the Attribute Complexity (AC) metric that reflects the complexity due to attributes. Class complexity due to data members is the total number of attributes of the class. Attributes are not local to one method but to every instance of a class, and can be accessed from several methods. Accordingly, the attribute complexity of a class (AC) is given by $AC = a$, where a is the total number of attributes. Finally, for calculating the complexity of the entire system, we have to consider not only the complexity of all the classes, but also the relationships among them. That is, we are giving emphasis on the inheritance property because classes may be either parent or children classes of others. In the case of a child class, it inherits the features from its parent class. By considering this OO property, we propose to calculate the code complexity of an entire system as follows. If the classes belong to disjoint hierarchies or are in the same hierarchical level of a class hierarchy, their individual WCC values are summed. On the other hand, the weights of classes belonging to the same class hierarchy are multiplied. Formally, if there are d levels of hierarchy depth in the OO code and level j has c classes, then the Code Complexity (CC) of the system is given by:

$$CC = \log_2\left(1 + \prod_{j=1}^{d}\left[\sum_{k=1}^{c} WCC'_{jk}\right]\right). \tag{5}$$

In general, it is known that class hierarchies are a potential source of complexity in OO systems and therefore many software metrics based on the structure of class hierarchies have been proposed [25]. Intuitively, the deeper a class is in a hierarchy, the more the potential code reuse. However, this makes the cognitive effort to reason about class behavior from the code much higher. From the perspective of complexity, similarly to recent related metrics [20], we basically penalize depth in class trees.

The unit of CC, Code Complexity Unit (CCU), is defined as the cognitive weight of the simplest system, i.e. a system having a single class with no attributes nor inheritance, including a single method having only a sequential structure (sequential structure in basic control structure terminology). In other words, the MC' of this method is 1, the WCC' of the mentioned class is 1, hence $CC = \log_2(1 + 1) = 1$. In the same line, MCU (Method Complexity Unit) and WCCU (Weighted Class Complexity Unit) are also defined to measure and compare individual method and class complexities.

Four more metrics can be derived by averaging the values of the above presented metrics. At the class-level, dividing the sum of complexities of all the methods of a class by the total number of methods in that class results in the Average

Method Complexity of a class. The Average Method Complexity at a system-level is computed summing all MC values and dividing the result by the total number of methods. Similarly, the Average Number of Attributes (i.e. Average AC) per class at a system-level is computed summing all AC values and diving the result by the total number of classes. Lastly, Average WCC represents the average weighted complexity of classes, which is derived from summing individual WCC values and dividing the sum by the total number of classes.

The next section describes a tool for automatically computing CC for Java programs.

4 AN ECLIPSE PLUG-IN FOR AUTOMATICALLY ASSESSING THE COMPLEXITY OF JAVA SYSTEMS

We have designed an Eclipse plug-in called *ccm4j*[1] (Code Complexity Metric for Java). ccm4j is organized around a pipeline-like architecture, resembling the one commonly found in compilers. The main components of ccm4j are shown in Figure 1. The ccm4j architecture imposes a strict relationship order by which a filter (or phase) runs only after the previous one has completely finished its task. Moreover, there is a shared data structure among the three phases, which will be explained below. As shown in Figure 1, ccm4j has three phases, namely Parsing, Complexity Calculation, and Consolidation. Below, the most relevant details of each phase are explained.

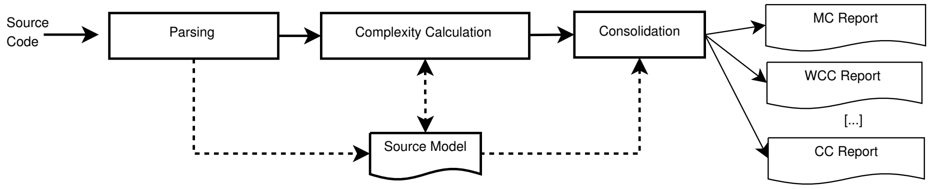


Figure 1. Architecture of ccm4j

4.1 Parsing Phase

During this phase, ccm4j parses the source code of the Java project being analyzed. The goal of the parsing phase is to build a representational model from the source code, which is suitable for later processing. To do this, ccm4j relies on the Eclipse Java Development Tools (JDT) framework, which provides APIs to access and manipulate Java source code. One of the provided APIs is called the Abstract Syntax Tree (AST), which is conceptually similar to the XML Document Object Model (DOM). DOM defines a standard way for representing and manipulating XML documents. A DOM is a tree-structured representation in memory of an XML document.

[1] ccm4j home page, `https://code.google.com/p/ccm4j/`

For parsing, ccm4j takes advantage of the ASTParser provided by the Eclipse JDT framework. The ASTParser builds an AST tree in memory capable of resolving variable bindings for each method statement while providing type information for them.

The ccm4j plug-in identifies declared classes and anonymous ones. With respect to method bodies, for each method ccm4j supplies the ASTParser with theirs statements, and creates a data structure indexed by method signatures. Basically, this structure contains a reference to each method AST, in which not only the implementation of that method can be found, but also information needed by the Complexity Calculation phase to compute method complexity. Then, this initial structure is used as input to the next two phases.

4.2 Complexity Calculation Phase

At this phase, ccm4j traverses the source model constructed in the previous phase, and calculates the weights of the basic control structures of each method, i.e. the MC' of each available method. This is the most resource demanding phase, since for calculating method complexities the tool needs to completely analyze the AST of each method. Besides processing the structure of methods, this phase also needs to handle recursion and to determine external invocations. This is because, as described in Section 3, the weight of any given method depends on the weights of the external methods called. To avoid processing the same method multiple times, the tool consults the source model built in the Parsing phase from a cache. Then, the first time a method is processed, the method entry in the model structure is associated with its complexity value, a.k.a. weight.

To compute the complexity of a particular method the ccm4j tool relies on the well-known Visitor OO pattern. The Eclipse framework provides an implementation of the Visitor Pattern to inspect the AST of a code block statement by statement. This support provides default implementations for processing all types of source code fragments and automatically step in nested structures such as code blocks and conditional statements, allowing us to redefine only those framework hooks for which a special action needs to be taken. This greatly simplified the implementation of the ccm4j tool.

The ccm4j tool handles two different types of recursive methods calls. The most simple recursive functions are those that involve only one method, calling itself in loop. We refer to this type of recursive call as a "local recursive call". For example, a method to sum the first n elements of an array, as shown in Listing 2, has a local recursive call. During the computation of the complexity of the sum method, ccm4j accounts an If-then-[else] basic control structure (according to Table 1 its weight is 2) and a Recursion basic control structure (its weight is 3). To solve this case ccm4j identifies whether the callee method is the same that the caller one or not, i.e. is a local call, before computing the complexity for the callee method.

Recursion can also be found in the form of cyclic calls involving N methods, for example if A.a() calls B.b(), B.b() calls C.c(), and C.c() calls A.a(). We refer to this

Listing 2. Simple recursive method

```
int sum(int arr[], int n) {
    if (n == 0)
        return 0;
    else
        return arr[n-1] + sum(arr, n - 1);
}
```

type of recursive calls as "complex recursive calls". For calculating the weight of complex recursive calls, ccm4j uses an algorithm that receives a yet-non-computed method as input. The algorithm relies on a stack for keeping account of methods for which complexity has been already computed. The main steps of the algorithm are:

1. Push the method being analyzed onto a stack.
2. For each basic control structure present in the body of the method:
   (a) If it is a function call, check if the target method is already on the stack.
      i If the target method is not in the stack, go to step 1 using it as input.
      ii If the target method is already on the stack, meaning that there is a loop, assign to it the constant weight associated with recursion, and go to step 2.
3. Pop the method from the stack.

The algorithm analyzes each basic control structure of a given method until a method call is detected. At step 2(a), the algorithm checks whether the complexity of the called method has been computed or not. The algorithm uses a stack to determine whether it is not necessary to go forward and process the called method, or it is necessary to process it. Once all the basic control structures of the given method have been analyzed, the method is removed from the stack.

The algorithm shown below exemplifies calculating the complexity of the method is_even, which has been deliberately defined in terms of another recursive method is_odd to introduce complex recursive calls (see Listing 3). Figure 2 shows from top to bottom the steps the algorithm performs to compute the complexity of the is_even method, and the state of the stack during the execution at each step. Note that we are assuming that is_even and is_odd are defined in different classes.

This kind of complex recursive calls are often found in OO systems, and even some OO design patterns encourage them. An example of complex recursive calls is found in the well-known Composite OO design pattern. The Composite design pattern allows developers to treat a group of objects as a whole, since these expose the same interface that a single instance would expose. In Figure 3 a materialization of this design pattern is shown. The CompositeGraphic.draw() method leads to

Listing 3. Mutually recursive methods: A canonical example

```
boolean is_even(int n) {
    if  (n == 0)
        return true;
    else
        return is_odd(n-1);
}

boolean is_odd(int n) {
    if (n == 0)
        return false;
    else
        return is_even(n-1);
}
```

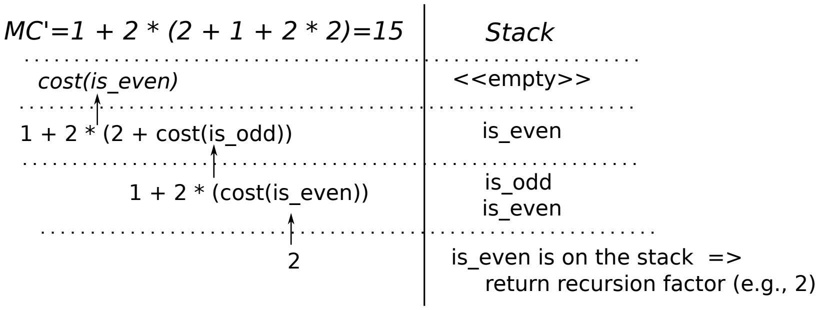
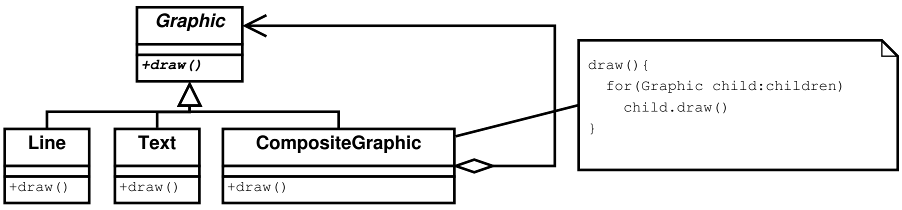

MC'=1 + 2 * (2 + 1 + 2 * 2)=15
Stack
cost(is_even)
<<empty>>
1 + 2 * (2 + cost(is_odd))
is_even
1 + 2 * (cost(is_even))
is_odd
is_even
2
is_even is on the stack =>
return recursion factor (e.g., 2)

Figure 2. Complexity of mutually recursive methods

a situation where one implementation of an abstract method recursively calls the same method.

In order to exemplify how ccm4j deals with this situation, let us assume that the complexity of both Line.draw() and Text.draw() methods is 1, and the composite implementation is the one shown in the text box on the right side of Figure 3. Then, by using the average strategy for weighting abstract methods (see Table 2 row 5) the



```
draw(){
  for(Graphic child:children)
    child.draw()
}
```

Figure 3. Mutual recursion in a hierarchy tree. The Composite Pattern example

complexity (MC') of Graphic.draw() is $(1+1+(1+3*3))/3 = 4$. Since Graphic.draw() is already on the method stack when processing CompositeGraphic.draw(), the associated cost of that call is the recursion factor, e.g. 3. Once the tool has computed the method complexity for all available methods, during the next phase the computed complexity values are aggregated to consolidate the MC, WCC and CC values.

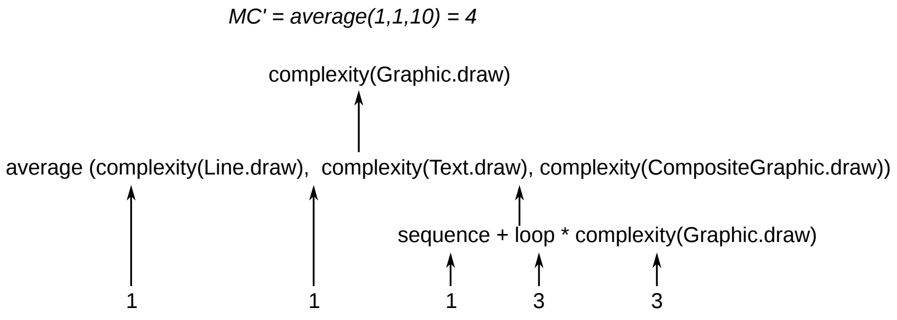

MC' = average(1,1,10) = 4
complexity(Graphic.draw)
average (complexity(Line.draw), complexity(Text.draw), complexity(CompositeGraphic.draw))
sequence + loop * complexity(Graphic.draw)
1
1
1
3
3

Figure 4. Cost of recursion in the Composite pattern

4.3 Consolidation Phase

This last phase uses the method complexities obtained in the Complexity Calculation phase to calculate the metrics. To do this, these complexity values are augmented with the class hierarchy relationships and the cost introduced by each disjoint hierarchy to calculate WCC. Besides, ccm4j aggregates data to obtain the total Code Complexity (CC) of the Java project under analysis. Finally, during this phase, the ccm4j tool produces a report as three Comma-Separated Values (CSV) files. Each CSV file stores tabular data in plain-text format with the MC, WCC, and CC values.

The MC output file has a row for each available method and five columns. The first column, "id", provides a unique identifier for each row. The second column, "method", contains the method signature. The third column, called "weight", contains the MC value for that method according to Equation (2). The column "weight expression" shows the weights summed or multiplied that produce the MC' value. The last column, called "external calls" indicates the number of external methods that are called from that method. For example, Table 3 shows the output for a method from the Apache Lenya project.

id	method	MC	complexity expression	external calls
1	WorkflowUtil#invoke	4.524	1 + (6) + 3 * [+2 + [3]]	4

Table 3. MC output example

The WCC output file consists of rows representing analyzed classes and six columns. The "id" column provides a unique identifier for each row. The second column is the class name, whereas the third column represents the parent class of that class. The other three columns contain class metrics, namely the number of methods, the number of attributes, and the resulting WCC (Equation (4)), respectively.

Finally, the CC output file has one row for each class hierarchy and six columns. As in the other files, the first column is the identifier. The second column is called "hierarchy" and contains the name of the class at the top of each separate class hierarchy. The third column is the CC value (Equation (5)). The fourth column contains the number of classes while the fifth one represents the depth of the hierarchy, respectively. Finally, the sixth column contains the CC expression. Table 4 presents an example of this output file.

Id	hierarchy	CC	number of classes	depth	complexity expression
15	org.apache.lenya.cms.rc.RCMLEntry	7.139	3	2	$10*(+9+5)$

Table 4. CC output example

5 PROPOSED METRICS AND TOOL: VALIDATION AND PRACTICAL EVALUATION

Next, we describe the validation of the metrics, and we illustrate and evaluate the ccm4j tool. The former task (Section 5.1) was performed using a recent validation framework specifically designed for evaluating software complexity metrics [21]. The aim was to complement the validation of the proposed metrics already done via Weyuker's properties [3]. We also evaluated the computational resources consumed by ccm4j on real Java projects, and illustrated the usage of the metrics for assessing complexity in classes from these projects (Section 5.2). The aim is to assess whether ccm4j is viable from a practical perspective and to show how to interpret the underlying metrics and related metrics.

5.1 Metric Validation

The framework in [21] was recently proposed to fulfill the need of validating software complexity metrics practically. Given that most validation frameworks up to now are based on measurement theory, it has been argued that other aspects should be considered when validating complexity metrics [21, 15]. Particularly, the framework prescribes a set of guidelines to assess the usefulness of a software complexity metric. Since our metrics are backed up with a tool, usefulness in practice is of utmost importance to our goals. Moreover, in this section we focus on the CC metric, since the framework is designed to validate system-level complexity metrics.

These guidelines come in the form of nine desirable properties which metrics should adhere to. To better explain the guidelines in the context the CC metric, we include other classical complexity metrics, namely statement count, Halstead's complexity [10] and McCabe's complexity [19]. The desirable properties analysis is shown below:

Property 1: The measure should be simple, which states that computing the metric should not involve complex mathematical functions, i.e. it should be computing resource-friendly. First, statement count clearly satisfies this property. McCabe's complexity does not heavily depend on the size of the system, and it is calculated from the graph that represents the control flow of the system. Halstead's uses simple formulas to compute system length, volume and effort. Finally, the calculation of CC is simple as well, since counting the variables used in its formulation is not difficult. In fact, resources consumption when computing CC on real Java projects is acceptable (see next subsection).

Property 2: The measure should be language independent. Statement count and Halstead's complexity depend on the verbosity and the operators of the programming language, respectively, and hence these metrics do not satisfy the property. Moreover, McCabe's complexity and CC are based on control flow structures and basic control structures present in many languages. Even when CC is conceptually designed for OO languages, assuming no inheritance (i.e. depth = 1) is analogous to computing the metric on a number of procedural modules as McCabe's complexity does.

Property 3: The metric should be developed on a proper scale. This means that there is always a need for a scale upon which comparison of two measures of the same metric is done. For statement count and Halstead's metric, there is not precise information on this issue [21]. For McCabe's complexity, an empirical discretization of the metric has been derived, which classifies systems into four categories (simple, less complex, complex, and highly complex). On the other hand, CC is based on a logarithmic scale, which allows different CC values to be compared.

Property 4: Metrics in metrics/measures should be consistent. Often one metric is not sufficient to consider the goals of the software project. Besides, when several metrics are proposed, lower or higher values should consistently have the same meaning across all metrics in the suite. Statement count, McCabe's complexity and CC do not propose a suite, and hence they should be used in conjunction with other metrics. Halstead's effort, on the other hand, takes a more high-level view of software complexity, and consistently considers other well-known aspects of systems, namely length, size, volume and effort [6].

Property 5: The metric should have a foundation that can be explained, which means that the reliability of a metric is poor if its values are not relative to some fundamental unit. For the case of statement count, this unit is program size (1 line is the smallest possible size). Halstead's complexity estimates number of bugs and effort to fix/develop the software. McCabe's complexity measures control flow complexity w.r.t. the simplest control flow possible. Finally, CC takes into account cognitive effort in CCU units (i.e. 1 CCU is the least possible complex system cognitively speaking).

Property 6: The metric should output positive numbers. Statement count, Halstead's complexity and McCabe's complexity give positive numbers [21]. The CC metric is computed by feeding a logarithmic function with a positive complexity value, which is obtained from multiplying and adding class complexity values. These values are based on the number of attributes and method complexities. Method complexities are computed by multiplying and adding basic control structure complexity values, which are positive numbers. Therefore, CC always gives positive numbers.

Property 7: The metric should differentiate between the complexities of the basic program constructs. Statement count and Halstead's complexity cannot clearly make this distinction. McCabe's metric computes complexity basing on the control flow of the system and not on the structure of the available constructs. CC computes complexity by considering the logical structure of program building blocks.

Property 8: The metric should differentiate between a sequence of the same construct and a nesting of them or an equivalent construct. Note that this property is not satisfied either by statement count, Halstead's complexity or McCabe's complexity, because they cannot distinguish among different constructs. Under CC, each basic control structure has its own weight. Having the same or equivalent basic control structures in sequence results in a *sum* of cognitive weights, but having so in nested structures implies *multiplying* cognitive weights.

Property 9: The metric should consider the modular complexity by a) the complexity should be affected by the addition, deletion and replacement of a module, and b) the complexity should reflect the interaction among the modules. With respect to a), it is clear that statement count and Halstead's complexity satisfy the property: the effort changes upon adding, deleting or replacing a module. However, this is not always the case with McCabe's complexity: adding a module with the same cyclomatic complexity as a module already present in the system does not affect the overall complexity. With CC, however, adding/deleting or replacing a class directly affects the associated complexity factor in the CC formula, and thus the property is satisfied. On the other hand, b) is satisfied by all metrics, since interactions between modules account for more statements, effort, cyclomatic entry points, and method weights in the respective metrics.

Table 5 summarizes the properties satisfied by the analyzed metrics. CC is the one that satisfies the most. Then, according to the guidelines in [21], CC qualifies as a useful complexity metric in practice.

Property	Statement Count	Halstead's Complexity	McCabe's Complexity	CC
1	yes	yes	yes	yes
2	no	no	yes	yes
3	no	no	yes	yes
4	no	yes	no	no
5	yes	yes	yes	yes
6	yes	yes	yes	yes
7	no	no	no	yes
8	no	no	no	yes
9	yes	yes	no	yes
# of satisfied properties	4/9	5/9	5/9	**8/9**

Table 5. Assessment of complexity metrics against the framework in [21]

5.2 Tool Evaluation and Illustration

We have evaluated ccm4j via ten open source Java projects from the Apache Software Foundation. Then, we computed the MC, WCC, and CC metrics, and assessed the demanded computational resources. The main goal of the performed experiment was to verify that ccm4j was able to process all the basic control structures and characteristics found on large, real life software projects, such as anonymous and inner classes, complex recursion patterns, multiple top level class declaration per source file, closed libraries usage, and exception handling mechanisms. In addition, we wanted to verify whether the ccm4j tool was feasible to be employed in an ordinary workstation in terms of resource consumption. In other words, we want to assess the amount of memory and CPU time needed to run ccm4j on real-life projects, and if these values are between manageable bounds. Finally, once the metrics were computed for each employed project, we used the results for identifying complex methods and classes, and comparing it with CK metrics.

The next subsection describes the experiments performed. Subsection 5.2.2 presents the performance results. Subsection 5.2.3 discusses the achieved complexity results.

5.2.1 Experimental Data and Settings

For the experiment ten projects from the Apache Software Foundation were employed. All the projects were downloaded from their official Web sites. Once a project was downloaded, all its dependencies were satisfied until the project was fully compiled. In this context, a dependency is a `jar` file required by the project to compile.

Table 6 presents a structural description of the projects, including their CC values. The employed projects have more than 100 classes. Regarding the hierarchical

Project	CC	Lines of code	# of class hierarchies	Avg. # of classes per hierarchy	Avg. hierarchy depth	# of types	# of methods
Commons-collections	33.605	23 713	200	1.915	1.260	417	3 461
Commons-math	75.861	81 803	507	1.553	1.666	924	7 084
CXF	33.087	47 222	444	1.341	1.115	709	5 153
Httpclient	33.840	17 814	178	1.315	1.146	314	1 751
Jgroups	56.306	73 156	500	1.456	1.098	800	7 836
Lenya	46.431	8 548	106	1.217	1.123	214	1 216
Log4j	50.968	20 631	202	1.431	1.134	308	2 090
Struts	55.115	23 198	165	2.370	1.157	424	2 854
Wicket	88.977	74 531	528	1.890	1.199	1 169	7 431
Xerces-J	72.305	71 027	182	2.231	1.214	570	5 788

Table 6. Project description

arrangement of their classes, all projects have an average Depth of Inheritance Tree (DIP) above 1. The number of methods of each project is quite varied, ranging from 1 216 to 7 836. At the same time, as described in the previous paragraph, the employed projects are heterogeneous not only from the functional and structural perspectives, but also from a technological one. For example, Lenya runs as a Web application on top of a Java Servlets engine, whereas Log4j and Xerces-J are libraries used by other projects that need logging facilities or processing XML files, respectively. In the same line, CXF and JGroups are used in distributed applications.

The two tables in the Web data at `http://tinyurl.com/ngu3adj` show the top 5 most complex classes and top 5 most complex methods in terms of WCC and MC, respectively. There were not significant differences between WCC values within the same project, with the exception of Wicket, JGroups and particularly Lenya. In this latter case, the complexity of the PageEnvelopeModule class at least doubles the complexity of any class in the project. Its most complex method, getAttribute(String, Configuration, Map), accounts for a big percentage of the total complexity of the class. Then, we looked inside this method code and observed that the cause for it being so complex is that it is basically a very deep chain of nested if-else blocks. On the other hand, in Httpclient, Jgroups and Xerces-J more than one method from the same class are in the top 5 method list.

The laptop used for the tests included a Core 2 Duo SU7300 processor running at 1.3 GHz with 4 GB of RAM. Note that the employed hardware represents the low end of the hardware usually available on the marketplace for software development. Lastly, the Java Virtual Machine (JVM) started to launch Eclipse was set with the parameters -XX:MaxPermSize=1024m -Xms256m -Xmx1024m.

5.2.2 Resource Consumption Analysis

We measured the runtime and memory consumption of the ccm4j tool when analyzing three of the projects from Table 6. Basically, we took a small (Lenya), a medium-sized (Log4j) and a large (Xerces-J) project in terms of lines of code to ensure representativeness. First, we assessed the run time and memory demanded by ccm4j when it was fresh started. In this context, the fresh start is used to indicate that the tool is executed just after starting an Eclipse session. The performance results obtained by averaging 10 fresh start executions of the tool on each selected project was 11.83 seconds and [110–130] MB RAM (Lenya), 17.73 seconds and [110–130] MB RAM (Log4j), and 38.32 seconds and [400–450] MB RAM (Xerces-J). As the reader can see, the run time and memory demanded seems to be influenced by the input project structural characteristics. The standard deviations for the 10 averaged executions of each project was negligible. Xerces-J, the biggest project, was the one that demanded significant resources.

Note that the normal workflow of a developer who is employing the tool includes:

1. calculating the metrics,
2. identifying complex methods or classes, and
3. modifying them to reduce their complexity.

We run the ccm4j tool with Xerces-J and, in turn, analyzed the MC metric results to identify the most complex method. The identified method was scanContent(org.apache.xerces.utils.QName) from the UCSReader class (see the abovementioned Web data). Then, we commented the entire body of the method, and we reproduced the experiment. We observed that the runtime for this second execution moves downward from 42 seconds to 13.316 seconds. To study the reasons behind this fall, we performed a second experiment. One possibility was that the fall occurred because the identified method demanded a lot of resources for computing the MC metric.

Another possibility was that the ccm4j tool experienced a "warm" effect produced by both the disk cache and the Just In Time (JIT) compiler of the JVM. To test this possibility we executed 10 times the ccm4j tool for each project, but during the same Eclipse session. The results are shown in Figure 5. As the reader can see, the demanded run time has a peak at the fresh start execution of all projects, but then it falls and stabilizes around a lower value. Specifically, by averaging the execution times from the third to the last run, Lenya run time was 2.86 seconds, Log4j 3.98 seconds, and Xerces-J 11.804 seconds. It is worth noting that the projects were not modified at all between each run.

5.2.3 Complexity Results Analysis

In order to compare our complexity metrics with the well-known CK metrics, we designed an experiment consisting on measuring the effectiveness of spotting the

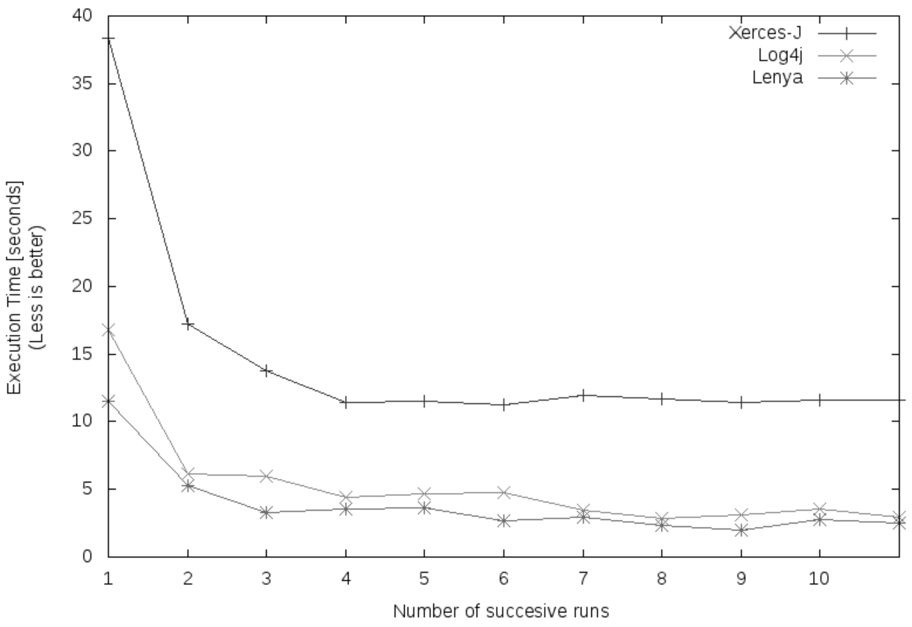


Figure 5. Effects of JIT and disc cache on tool execution time

most complex classes in each project with our plug-in and at the same time using CK metrics. For gathering CK metrics, we used ckjm (`http://www.spinellis.gr/sw/ckjm`).

As suggested earlier, the typical use of our plug-in includes spotting complex methods or classes first. Finding complex classes also involves identifying their most complex methods afterwards. Thus, in this experiment we took the opposite direction: we first spotted the top 5 most complex methods of each project and then we took the ordered list of classes (i.e. L) in which they were declared. To ensure better system coverage, for Httpclient, Jgroups and Xerces-J more methods were considered so the resulting 5-element lists contained no duplicate class names. For example, from the most complex methods in `http://tinyurl.com/ngu3adj`, it follows that L_{CXF} = *[ClientImpl$2, ClientImpl$2$1, ClientImpl, OneWayProcessorInterceptor, AttachmentDeserializer]*.

Then, for each project p, we obtained five lists containing *all* the classes sorted by WMC, DIT, RFC, CBO and WCC in decreasing order, i.e. $LWMC_p$, $LDIT_p$, $LRFC_p$, $LCBO_p$ and $LWCC_p$. Then, for each class name in L_p the index (or position) in the lists was separately computed. These results are shown in the tables at `http://tinyurl.com/l2adzaw` as $Index_*$ columns with cell values in the form [start_position-end_position], meaning that all classes in this range have the same metric value. For example, the ClientImpl$2 class of the CXF project, when sorting classes by decreasing WMC, is in the [453–558] position according to its $Index_{WMC}$. In other words, 106 classes in the project have WMC=3. As a corollary, start_position=end_position means that only one class has a particular metric value (see for example ClientImpl and $Index_{WMC}$). Finally, $LWCC_p$ has discrete values, since no duplicate values for WCC were found when considering several decimal places. Below we provide a per-metric analysis of these results.

As seen in the CK and WMC tables, using WMC to spot the potentially complex classes was in general ineffective. Basically, WMC assumes that the weight of each method is one, while in MC, and indirectly in WCC, the internal structure of methods is considered a central factor. For example, in CXF, the highest ranked class from L_p according to WMC is ClientImpl, whereas the rest appear much latter in $LWMC_p$. Moreover, although the WMC metrics may calculate the same value for two similar codes, WCC obtains different results since it considers method structures in more detail.

DIT represents the maximum length in the class hierarchy tree from the node associated to a class to the root of the tree. Hence, DIT values vary from class to class depending on the level of classes in the hierarchy. Most DIT values range from 0 to 2, and therefore they provide very limited information about the complexity of classes. Then, it is difficult to assess class complexity by just considering DIT.

Moreover, the RFC metric is defined as the total number of methods that can be potentially executed in response to a message sent to a class. This includes all the methods available in the class hierarchy. RFC is important since as it increases, the effort required for testing increases as well [23]. RFC tended to spot more classes (see for example Xerces-J) because it is a more ellaborated metric compared to WMC and DIT. Still, RFC only calculates the number of methods in response to a message but our metrics also consider the complexity of the called methods. We believe that considering not only the number of methods, but also the whole structure of methods takes into account valuable information about the complexity of the code.

Furthermore, CBO quantifies the interactions between objects by taking into account the number of other classes to which a class is coupled. Broadly, CBO has a significant impact on complexity in terms of modularity, maintenance, and testing of a system. In the analyzed projects, the effectiveness of CBO for spotting classes was slightly better than that of RFC. Note that one class might have CBO = 1 meaning that it interacts with only one class, but the former might include a high number of messages to that class, which makes the code intuitively more complex. On the other hand, our metrics consider the number of message calls to other classes plus the weight of the called methods.

All in all, the complexity factors evaluated by our metrics can be to some extent also evaluated by WMC, DIT, RFC and CBO, but it is clear that none of these metrics is designed to consider of all these factors simultaneously. This shows that our metrics do not directly compete with the analyzed CK metrics, but we aim at offering another approach to quantify complexity in OO code by aggregating several factors that are known to affect overall complexity. Another interesting result is that there is a high correspondence between MC and WCC values in the analyzed projects, which is ideal in some cases (i.e. Httpclient, JGroups and Wicket), whereas for other projects all classes appear within the top 10 most complex classes. This means that ranking system methods by decreasing MC quickly leads to the most complex classes, and vice versa, which accelerates the task of system-wide identification of the most complex code pieces to refactor or document.

6 FUTURE WORK

We are planning to extend our set of basic control structure in order to account for Java concurrency constructs (e.g. synchronized statements) and concurrency facilities such as the Thread, Executor and Callable classes from the Java concurrency framework. As for the WCC metric itself, we will study how to better deal with class attributes in terms of complexity. One starting point is to take into account data-type abstractness in the current metric to reflect polymorphic variables. Intuitively, the cognitive effort necessary to understand primitive data-types, and more concrete versus abstract non-primitive data-types should be weighted differently. In principle, this will require considering and weighting several factors, namely composition and inheritance relationships between data-types, cardinalities, Java generics, and so on. Particularly, it is known that the problem of determining whether abstract modeling (like OO data-types) leads to more cognitive effort than concrete modeling concepts or vice versa, at least in the engineering domain, depends on the learning style of users [8]. Lastly, another element from the Java language to consider are annotations. Upon compiling a project, annotations are processed by certain compiler plug-ins called annotation processors, which can create additional Java source files/resources or even modify the annotated Java code itself. Therefore, this mechanism might alter the program code, which impacts the WCC metric.

We will also do a research on weighting those classes for which the developer has no source code. A call to a library method whose implementation is unknown is associated with a weight that, while configurable, is the same for all external libraries calls regardless of the signature of the called method. We believe that there is room for refinement on this weighting schema, which is worth to be investigated. As discussed in [12], an interface by itself requires developers' effort to be used, so that simpler libraries interfaces are preferable over complex ones to reduce the overall system complexity. Then, we can combine the weight employed for calling closed external libraries with interface complexity metrics like those presented in [27]. Though these interface complexity metrics are specially designed for interfaces described in the Web Services Description Language (WSDL), their underpinnings may be applied to interfaces written in Java as well.

Moreover, we are extending the ccm4j tool by following the approach to visualize multiple metrics presented in [16], to provide a graphical layout of classes complexities as a complement of the current spreadsheet based output. This visual approach is called *Polymetric views* [16]. The polymetric views approach represents a lightweight software visualization technique enriched with software metrics information, whose goal is to help developers to understand the structure and detect problems of a software system related to quality attributes. We are adapting the polymetric views approach to our complexity metrics to increase the overall comprehensibility of a system for both maintenance and new developments activities, while easing the identification of more complex methods and classes by rapidly spotting the most complex methods/classes. An appropriate combination of metrics and relationships visually shape methods and classes so that it is easier to identify which ones require more effort to understand, test, and maintain.

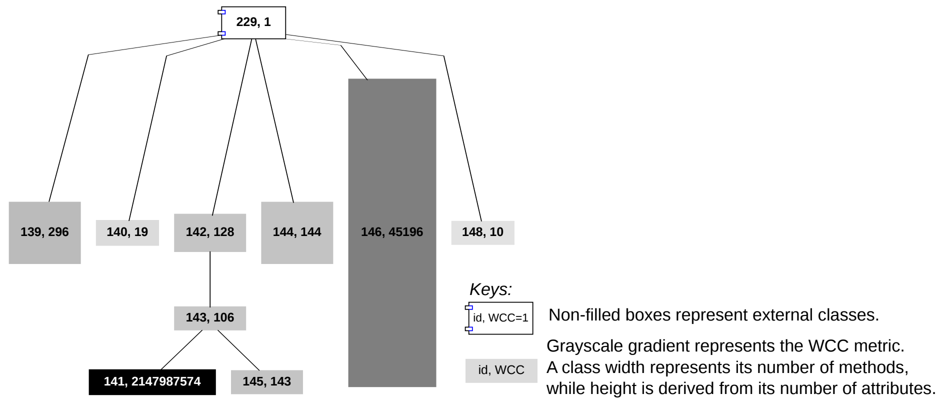

229, 1
139, 296
140, 19
142, 128
144, 144
146, 45196
148, 10
143, 106
141, 2147987574
145, 143
Keys:
id, WCC=1
Non-filled boxes represent external classes.
id, WCC
Grayscale gradient represents the WCC metric.
A class width represents its number of methods,
while height is derived from its number of attributes.

Figure 6. Using polymetric views for depicting the WCC metric of a class hierarchy

We have developed a proof of concept to realize our idea and demonstrate its feasibility. Figure 6 shows the result of displaying a class hierarchy from the Lenya project. Gray-scale boxes represent project classes. The color of these boxes depends on the normalized WCC metric results among all the classes of the project. Light gray is used for denoting classes with low WCC metric, whereas the darker a box the higher the WCC of the class it represents. The area of a box is determined by the number of attributes and methods of a class. As the reader can see, this type of diagram allows the developer to quickly figure out that classes 141 and 146 are the more complex of the hierarchy, while the complexity for class 141 is concentrated in fewer methods because of the dimensions of its respective box. In fact, the class 141 is PageEnvelopeModule, which, as mentioned earlier, has the getAttribute(String, Configuration, Map) method whose complexity is much higher than any of the others methods in the same project. This means that if a developer wants to reduce the overall hierarchical complexity she/he may choose to modify the smaller class in order to introduce as fewer changes as possible.

Finally, we plan to study MC, WCC, and CC metrics for predicting software quality attributes. Different software engineering metrics can be employed for predicting situations such as whether bugs will be reopened or not, or whether a class will have a propensity for bugs or not (e.g. [26]). In principle, we will analyze the statistical correlation among the metrics and different aspect of classes.

7 CONCLUSIONS

There is an incessant necessity for measuring software quality attributes in OO systems so as to ensure software maintainability and properly focus efforts in projects. This paper presented a set of software metrics that measure software cognitive complexity based on a set of basic control structures for characterizing Java-based OO source code. Additionally, this paper described an automatic tool for gathering these metrics, which is implemented as a plug-in for Eclipse. As a complement to a previous theoretical validation of the underpinnings of the metrics, we show that the system-level metric of the proposed set (i.e. the CC metric) satisfies eight out of nine of the practical properties for software complexity metrics identified in [21]. This shows the practical validity of the metric and, together with a theoretical evaluation performed via Weyuker's properties, suggests that the CC metric can be considered in real development scenarios.

The identified set of basic control structures to materialize the MC and indirectly WCC and CC covers a wide range of programming constructs for Java, one of the most popular OO programming languages. Complementary experimental evaluation reported in this paper provides further evidence on the feasibility of using the proposed approach to assess real-world Java projects. These experiments show that our tool can be executed using a low-end personal computer in the order of seconds. Also, based on real projects, we have shown that our metrics can be useful in practice to assess complexity in a more ellaborated way since our metrics aggregate factors that are known to individually affect complexity. In fact, the experiments have also shown that such factors are not simultaneously captured by popular CK metrics such as WMC, DIT, RFC and CBO.

Acknowledgements

This paper has been partially supported by grant PICT-2012-0045 of ANPCyT. We also thank the reviewers for their helpful comments to improve the paper.

REFERENCES

[1] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990.

[2] Wang, Y.: On Cognitive Informatics. Brain and Mind, Vol. 4, 2003, No. 2, pp. 151–167.

[3] Aloysius, A.—Arockiam, L.: A Survey on Metric of Software Cognitive Complexity for OO Design. World Academy of Science, Engineering and Technology, Vol. 58, 2011, pp. 765–769.

[4] Arockiam, L.—Geetha, K.—Aloysius, A.: On Validating Class Level Cognitive Complexity Metrics. CiiT International Journal of Software Engineering and Technology, Vol. 2, 2010, No. 3, pp. 152–157.

[5] Azar, D.—Vybihal, J.: An Ant Colony Optimization Algorithm to Improve Software Quality Prediction Models: Case of Class Stability. Information and Software Technology, Vol. 53, 2011, No. 4, pp. 388–393.

[6] Briand, L.—Morasca, S.—Basili, V.: Property-Based Software Engineering Measurement. IEEE Transactions on Software Engineering, Vol. 22, 1996, No. 1, pp. 68–86.

[7] Chidamber, S.—Kemerer, C.: A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, Vol. 20, 1994, No. 6, pp. 476–493.

[8] Felder, R.—Silverman, L.: Learning and Teaching Styles in Engineering Education. Engineering Education, Vol. 78, 1988, No. 7, pp. 674–681.

[9] Gill, N.—Sikka, S.: New Complexity Model for Classes in Object Oriented System. SIGSOFT Software Engineering Notes, Vol. 35, 2010, No. 5, pp. 1–7.

[10] Halstead, M.: Elements of Software Science (Operating and Programming Systems Series). Elsevier Science Inc., New York, NY, USA, 1977.

[11] Hansen, W.: Measurement of Program Complexity by the Pair: Cyclomatic Number, Operator Count. ACM SIGPLAN Notices, Vol. 13, 1978, No. 3, pp. 29–33.

[12] Henning, M.: API Design Matters. Communications of the ACM, Vol. 52, 2009, No. 5, pp. 46–56.

[13] Kafura, D.—Reddy, G.: The Use of Software Complexity Metrics in Software Maintenance. IEEE Transactions on Software Engineering, Vol. 13, 1987, No. 3, pp. 335–343.

[14] Kearney, J.—Sedlmeyer, R.—Thompson, W.—Gray, M.—Adler, M.: Software Complexity Measurement. Communications of the ACM, Vol. 29, 1986, No. 11, pp. 1044–1050.

[15] Kitchenham, B.—Pfleeger, S.—Fenton, N.: Towards a Framework for Software Measurement Validation. IEEE Transactions on Software Engineering, Vol. 21, 1995, No. 12, pp. 929–944.

[16] Lanza, M.—Ducasse, S.: Polymetric Views – a Lightweight Visual Approach to Reverse Engineering. IEEE Transactions on Software Engineering, Vol. 29, 2003, No. 9, pp. 782–795.

[17] Lanza, M.—Marinescu, R.: Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer Berlin Heidelberg, 2006.

[18] Mäkelä, S.—Leppänen, V.: Client-Based Cohesion Metrics for Java Programs. Science of Computer Programming, Vol. 74, 2009, pp. 355–378.

[19] McCabe, T.: A Complexity Measure. IEEE Transactions on Software Engineering, Vol. 2, 1976, No. 4, pp. 308–320.

[20] Mishra, D.: New Inheritance Complexity Metrics for Object-Oriented Software Systems: An Evaluation with Weyuker's Properties. Computing and Informatics, Vol. 30, 2011, No. 2, pp. 267–293.

[21] Misra, S.—Akman, I.—Palacios, R.: Framework for Evaluation and Validation of Software Complexity Measures. IET Software, Vol. 6, 2012, No. 4, pp. 323–334.

[22] Misra, S.—Koyuncu, M.—Crasso, M.—Mateos, C.—Zunino, A.: A Suite of Cognitive Complexity Metrics. Proceedings of the 12th International Conference on Computational Science and Its Applications (ICCSA 2012), Lecture Notes in Computer Science, Vol. 7336, 2012, pp. 234–247.

[23] Pressman, R.: Software Engineering: A Practitioner's Approach. 6th Edition. McGraw-Hill, Inc., New York, NY, USA, 2005.

[24] Shao, J.—Wang, Y.: A New Measure of Software Complexity Based on Cognitive Weights. Canadian Journal of Electrical and Computer Engineering, Vol. 28, 2003, No. 2, pp. 69–74.

[25] Sheldon, F.—Jerath, K.—Chung, H.: Metrics for Maintainability of Class Inheritance Hierarchies. Journal of Software Maintenance: Research and Practice, Vol. 14, 2002, No. 3, pp. 147–160.

[26] Shihab, E.—Ihara, A.—Kamei, Y.—Ibrahim, W.—Ohira, M.—Adams, B.—Hassan, A.—Matsumoto, K.: Studying Re-Opened Bugs in Open Source Software. Empirical Software Engineering, Vol. 18, 2013, No. 5, pp. 1005–1042.

[27] Sneed, H.: Measuring Web Service Interfaces. 12th IEEE International Symposium on Web Systems Evolution (WSE), 2010, pp. 111–115.

[28] Subramanyam, R.—Krishnan, M.: Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. IEEE Transactions on Software Engineering, Vol. 29, 2003, No. 4, pp. 297–310.

[29] Vivanco, R.—Jin, D.: Improving Predictive Models of Cognitive Complexity Using an Evolutionary Computational Approach: A Case Study. 2007 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '07), 2007, pp. 109–123.

[30] Weyuker, E.: Evaluating Software Complexity Measures. IEEE Transactions on Software Engineering, Vol. 14, 1988, No. 9, pp. 1357–1365.

Marco Crasso received his Ph.D. degree in computer science from the UNICEN in 2010. He was a member of the ISISTAN and the CONICET. Now he works at IBM Research.

Cristian Mateos received his Ph.D. degree in computer science from the UNICEN in 2008. He is a full time Teacher Assistant at the UNICEN and member of ISISTAN-CONICET. His main research interests are parallel/distributed programming, grid middlewares and service-oriented computing.

Alejandro Zunino received his Ph.D. degree in computer science from the UNICEN in 2003. He is Full Adjunct Professor at UNICEN and member of the ISISTAN-CONICET. His research areas are grid computing, service-oriented computing, web services and mobile agents.

Sanjay Misra works at the Covenant University in Ota, Nigeria. He is mainly interested in software engineering, with a special emphasis on software metrics.

Pablo Polvorín is pursuing B.Sc. in systems engineering from the UNICEN. His involvement in this project came from the interest in OO metrics and software quality. He works at Process One.