TOOLS FOR METRICS USED TO EVALUATE
SOFTWARE QUALITY: A SURVEY

by

Dima M. Moubayed

Submitted in partial fulfillment of the requirements
for the Degree of Master of Sciences

Project Advisor
Dr. Danielle Azar

Computer Science and Mathematics Division
LEBANESE AMERICAN UNIVERSITY
June 2006
LEBANESE AMERICAN UNIVERSITY

GRADUATE STUDIES

We hereby approve the project of

Dima M. Moubayed

candidate for the Master of Science degree*.

(signed) Dr. Danielle Azar

Dr. Haidar Harmanani

Dr. Chadi Nour

Date 19-06-06

* We also certify that written approval has been obtained for any proprietary material contained therein.
I grant to the LEBANESE AMERICAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University’s own purpose without cost to the University or to its students, agents and employees. I further agree that the University may reproduce and provide single copies of the work, in any format other than in or from microforms, to the public for the cost of reproduction.
TOOLS FOR METRICS USED TO EVALUATE SOFTWARE QUALITY: A SURVEY

ABSTRACT

by

DIMA M. MOUBAYED

Software Quality relies on attributes such as: Complexity, efficiency, maintainability, reusability and others. For evaluating such quality attributes, Metrics have been proposed, e.g.: the weighted methods per class, the depth of inheritance tree, the number of children of a class, etc...

The process of controlling quality of a particular computer system through such metrics computation requires the use of some related tools that serve as software quality evaluators; their use has a major impact on the quality of the software to be developed, as they can provide guidelines for future designs.

Throughout this paper, we will first define what software quality and its attributes are, point out their importance in an Object-Oriented paradigm and how they could be estimated and evaluated. We will then present a survey of some available tools, used to interpret software systems quality. While many of these tools in the market seem to be interesting to use, we will only describe in details the functionality and specifications of the ones we can possibly reach.
To my parents

To my beloved Sami and Leyla
ACKNOWLEDGMENTS

I would like to thank Dr. Danielle Azar for her significant support and sincere guidance.
I would also like to thank my parents and friends for their precious support and encouragement throughout my studies.
# TABLE OF CONTENTS

## Chapter 1: Introduction

1.1 Software Quality 1  
1.2 Software Quality Characteristics 2

## Chapter 2: Metrics for evaluating quality characteristics

2.1 Definition of metrics and their relationship with quality characteristics 5  
2.2 Importance and utility of metrics 5  
2.3 Classification of Metrics 5

## Chapter 3: Evaluating metrics: The tools

3.1 The Jstyle from Man Machine System 12  
3.2 Understand for Java from SIT 14  
3.3 The Jmetric 16  
3.4 Resource Standard Metrics from M Squared Technologies 19

## Chapter 4: Discussion of Results

23

## Chapter 5: Conclusion

31

## Bibliography

38
List of Figures

Figure 1 Method, class and project Metrics .................................................. 6
Figure 2 Generating the report .................................................................. 13
Figure 3 Example of an output text report .................................................. 14
Figure 4 Loading the input files .................................................................. 15
Figure 5 Generating Metrics in Understand for Java ..................................... 15
Figure 6 Example of Generating Metric Charts ........................................... 16
Figure 7 Chart reporting some class metrics ................................................. 18
Figure 8 XY Scatter reporting some class metrics ....................................... 18
Figure 9 Text output format reporting some class metrics ............................ 19
Figure 10 Command prompt ....................................................................... 20
Figure 11 RAM Wizard main window ......................................................... 20
Figure 12 Report generated by RSM ............................................................ 21
Figure 13 *Object-Oriented Metrics generated in an HTML report* ............... 22
List of Tables

Table 1. Quality characteristics evaluated by metrics 32
Table 2. Metrics evaluated at all levels 34
Table 3. Comparison between the four tools 36
Table 4. Tools Computing Metrics 37
Chapter 1

Introduction

1.1 Software Quality

Software quality is evaluated in terms of characteristics such as maintainability, portability, reliability, efficiency, etc.

The software quality life cycle begins when the system is created, and ends when it is no longer in use. It can be subdivided into two phases according to the ISO/IEC 9126 model [12]: The internal and external quality defined at the software product model as the quality characteristics (introduced in the next section), and the quality in use defined at the product implementation (i.e. the performance after the system's execution). To evaluate the quality in use that most interests the engineers using the Java programming Language, we should assess the internal and external quality of the system. So we will only study the internal software quality attributes, such as complexity, coupling and cohesion, manifested externally in the system by the mean of quality characteristics such as maintainability, reusability, understandability and efficiency. For this purpose, some units of measurement—metrics—are defined and used to give values that describe the quality of the software product evaluated. Examples of such metrics are number of children of the object class being evaluated, depth of inheritance, method size, etc. In our project, we are doing a survey of the tools that allow measuring these metrics.

Creating some software is not a complicated task. Guaranteeing the good quality of a system is the biggest challenge. When creating a software system, several costs should be taken into account: financial costs, human effort and time required for the process to be accomplished. Designers have a very important role in predicting the quality of their software and checking its functionality before it is ready to get launched in the market. Issues that might be considered in the system are then examined at the design phase. This has the advantage of increasing productivity in the human effort, and reducing time and financial costs of the system.

Let us consider some ready-to-use code, for which quality is already assessed. A software system having the characteristic of being re-used without the necessity of redeveloping the entire code from scratch is preferable to one that is lacking this characteristic. This shows that the impact of evaluating quality at the design stage brings many advantages to the software costs.

We will consider object-oriented Java framework, knowing that a lot of related work has been done in this area, and we will rely on prior experience learned and earned from previous work as well as our personal testing, in order to collect some evaluation criteria
for the software quality. A “well-written” software system is generally characterized by its good readability, easiness of maintenance, testing, debugging, fixing, modification, portability, low complexity, low resources consumption, and small number of compilation warning [10. We are interested in what defines such characteristics and, most importantly, how they are measured.

Next, we will introduce and describe the basic software quality factors, also called attributes or characteristics, and explain how they affect the system and influence on Object-Oriented designs. We will also show the dimensions that must be mainly considered in the evaluation of quality of the system components.

1.2. Software quality characteristics

Quality characteristics, also known as quality factors or attributes, define the property of a software product. Next, we will focus on some characteristics that define quality in a software system.

1. Maintainability

Maintainability qualifies a software product that has the ability of being modified, facilitating possible updates necessary to satisfy new requirements. When a system is reconfigured and re-adapted, a very important step is to perform a testing of the object-oriented design after being changed, to ensure its performance in spite of any requirements evolution and through different versions. So a maintainable system should easily support evaluation of its performance, no matter what corrections, improvements or adaptations affect it.

2. Efficiency

Efficiency qualifies a system that does not waste any of its resources (e.g. time, memory) during its performance. An Object-Oriented software system is efficient if its components (e.g. methods) are favorable in terms of speed and memory usage. The main sub-characteristics that can measure the efficiency of a software system are time behavior and resource behavior. Notice that the lower resource consumption is, the more efficient the system.
3. Understandability

Software products are generally promoted for being simple, easy-to-learn and practical to programmers using them and aiming to easily handle their source code. Understandability is mainly measured in terms of learnability and readability that require the code to be well-organized and clearly written in a block-structured language for the programmers using the software system. Programmers are more likely to accomplish their work in a much simpler manner than when dealing with a non-convenient software design, especially when some manuals or help are provided, realizing the operability and the limitations of the system concerned.

4. Reliability

Reliability of a software product is defined as its ability to perform as intended, in accordance with its original standards, under specific conditions over a period of time [10]. In other terms, it is expected to accomplish correct tasks in a period of time that does not exceed the promised time. This favors the software to be robust in all environments. Two basic sub-characteristics represent a reliable software; the software has to be fault-tolerant (i.e. able to get over possible errors), and able to recover from unexpected errors that could not be avoided.

5. Portability

When carried out of their original environment, many software systems fail because they were not designed to fit in all configurations. Portability is the ability of the system to be easily installed and to operate even when transferred, independently from any library routines specific to its original installation. The system should be suited to adapt to any processor and multiple configurations.

6. Functionality

A software system is said to be functional when all the required functions are available in it. It provides the awaited services, gives accurate results, and ensures a high level of security, by preventing its code and data from unauthorized access.

7. Reusability

Reusability is defined as the possibility of a particular source code to be reworked to fit other usage for the system. According to Basili, Lionel, Briand and Melo, “Reuse is assumed to be an effective strategy for building high-quality software” [13].
Reusability shows a strong impact on productivity in the context of Object-Oriented systems. By reducing time and effort due to re-use, we increase the productivity, and in this way, this is likely to lead to higher quality and consequently better software results.

8. Complexity

Complexity is the order of growth of the source code. In a program, it is measured by the number of linearly independent paths through a program. For example, for a simple function that has no conditionals, only one path exists. It influences all other quality characteristics such as maintainability of the system, its degree of portability, its reusability, efficiency, etc. As a result, measurement of all quality attributes can give an idea of the degree of complexity of the entire system, and the level of complexity is a basic dimension of quality assessment.

Maintainable, efficient, understandable, reliable, portable, functional and reusable software should basically have high quality measures. There are other factors that affect the quality of a software product, but we limit our discussion to these as they are the most widely considered.

The rest of this report is organized as follows. Section 2 presents the principal units to measure quality factors - the metrics. Section 3 introduces a series of tools chosen to be good examples for illustrating software evaluation and Section 4 describes functionalities and compares results. Some concluding comments about the tools are presented in the Section 5.
Chapter 2

Metrics for evaluating quality attributes

2.1 Definition of metrics and their relationship with quality characteristics

In the context of software quality, a metric is defined as a unit of measurement of quality attributes, and is mainly computed by specific tools that calculate its values through some output reports [13].

A large number of metrics has been proposed in Object-Oriented systems to evaluate quality of components like methods and classes, or quality attributes like coupling and inheritance (in Java).

Each of the metrics studied in this paper basically measures one or several quality factors – Table 1. For instance, the number of methods metric can explain how a component-based software system is not only maintainable, but also understandable. The metric used to calculate the total number of physical lines, including comments, blank lines and statements in a method is the Line of Code Metric, and is shown to be needed for assessing the understandability quality factor.

2.2 Importance and utility of metrics

Tom DeMarco states in his book “Controlling Software Projects: Management and estimation” [13]: “You cannot control what you cannot measure”. Metrics values help to control systems. These metrics are useful for the internal Object-Oriented quality attributes previously defined (e.g. cohesion and coupling between classes), as well as external measures of the interconnections of the source code entities (classes, methods, etc.).

2.3 Classification of Metrics

A very large number of metrics have been proposed for the evaluation of software quality factors. To simplify the study of the tools that calculate such metrics, and to make a more homogeneous comparison between results of the different outputs computed, we will mainly introduce and comment on the metrics that are most common to the tools collected.
This paper distinguishes between three categories of metrics: method, class and project metrics. It presents some traditional metrics that concern the physical structure of the methods inside the code, such as the lines of code, the number of statements, the number of executable lines, the percentage of comment, the blank lines and the Cyclomatic complexity. It also distinguishes the class category of metrics, which includes the lines of code inside the class, the number of statements, the percentage of comment lines, the executable lines, the blank lines, the number of children, the number of parents, the depth of inheritance, the number of inner classes, the fan-in, the fan-out, the response of a class, the weighted method number or complexity, the coupling between object classes and the lack of cohesion of methods. The last category of metrics concerns the project, and is characterized by the total number of lines of code, the blank lines, the comment lines, the executable lines of code, the number of statements, the number of methods, the total number classes and inner classes, the average depth of inheritance tree and the number of files/modules in the project. Below is a chart—Figure 1—that shows how metrics are going to be classified in this paper.

Following is a list with a brief description of the metrics that are of concern in this document.

1. Lines of code (LOC)

This metric is concerned with the physical count of the source code lines and is usually computed by counting all the lines of the code in a method, class, or project. This number affects maintainability, understandability and reusability.

![Diagram showing the breakdown of project metrics into method and class metrics.]

Figure 1. Method, class and project Metrics

2. Blank lines

This metric counts the number of blank or empty lines in the code. It affects maintainability, understandability and reusability.
3. Cyclomatic Complexity (CC)

The cyclomatic complexity, also known as McCabe CC after its originator, is evaluated by the number of different paths in a method. McCabe defines this metric as a measure of the complexity of each method, calculated by the count of the Boolean conditions in the code such as if, while, and other types of loops [14]. The lower its value, the simpler and less complex the method appears to be. Besides complexity, this metric affects many other quality attributes, such as maintainability, efficiency, reliability and reusability.

4. Percentage of comment lines

This is the total number of lines that have comment on them and start with either /*, // or */. It affects maintainability, understandability and reusability.

5. Number of executable code lines

This is the total number of lines that have executable code on them. It affects maintainability, understandability and reusability.

6. Number of statements

This metric measures the actual number of statements in the code (very close to executable statements). Very often in the tools used to evaluate such metrics, the number of executable code lines and the number of statements are similar. So they both evaluate the same quality characteristics: maintainability, understandability and reusability.

7. Weighted method complexity (WMC)

This is usually computed by adding up the cyclomatic complexities of all the class local methods. In other terms, WMC is the cyclomatic complexity of a class, calculated by adding the cyclomatic complexities of all methods in the class, which is equal to the number of methods if cyclomatic complexities of all methods are ones. We notice that the more methods, the larger the value of this metric, and consequently, the more complex the system. WMC evaluates complexity, maintainability and understandability.
8. Number of methods (NOM)/ Weighted method per class

This metric is calculated by counting all the local methods defined in the class. WMC affects complexity, as well as reusability, maintainability and understandability.

9. Number of files/modules

This is the total number of source files or modules included in the project. This metric affects maintainability, understandability and reusability.

10. Number of classes

It is the total number of classes defined in a module or the entire project. It includes top-level public, non-public, local and inner classes. Not included in the measurement of this metric are the classes that are referenced but whose definitions are not available in the project. The number of classes metric mainly affects maintainability.

11. Number of inner classes

This is the total number of non-static inner (public and non-public) and nested classes in the entire project. This metric affects understandability, reusability, and complexity.

12. Response for a class (RFC)

When an object of a certain class is instantiated and a message is passed by this object, RFC represents the cardinality of the set of methods invoked in response to this execution. This set includes both local methods of a certain class X, and methods from other classes called by local methods of X, due to relationships between X and those classes.
We can realize how hard it is to understand what methods are invoked if the RFC measured is too high, which makes it difficult to test and find out where an error has occurred; beside complexity, this metric evaluates maintainability and understandability.
13. Lack of cohesion of methods (LCOM) \(^{iv}\)

By definition, cohesion between methods in a class describes how closely the local methods of a class are related to the local instance variables and attributes in the same class [2]. It is high if most method pairs' use at least one common variable of the class.

LCOM measures the number of disjoint sets— that are non-intersecting, of local methods in a class, where each method references an instance variable, knowing that similar methods that operate on the same attributes belong to the same set. It is calculated by subtracting the number of method pairs using the same variables from the number of method pairs using distinct variables. The LCOM value is high when the number of method pairs not using the same instance variables is high [3].

A highly cohesive class is easier to maintain than a class with low cohesion of its methods, because such a class design is well subdivided and its complexity is simplified by the clear partitioning of methods.

LCOM affects quality factors like complexity, reliability, efficiency, maintainability and reusability.

14. Coupling between object classes (CBO)

Coupling is defined as the relationship between different components of a system. We observe three kinds of coupling: coupling through message passing among classes—when several objects require information from each others; coupling through data abstraction, where some class A might include a declaration of another class B within its variables, resulting in a coupling between the two classes A and B; and coupling through inheritance, which measure will be evaluated through some other metrics: the depth of inheritance, the number of children, the number of parents and the Fan-in and Fan-out defined in the next two sections.

Classes are coupled when they use methods or instance variables defined in each other. CBO of a class is measured by counting the number of classes coupled to this class. A high CBO value indicates a complicated, practically impossible way of reusing the component system and therefore a difficulty in maintaining it. Besides, it makes the system more complex and yet less efficient, and the components harder to understand.

This metric evaluates complexity, efficiency, understandability, maintainability, reusability, reliability and portability.

15. Fan-in

The Fan-in metric of a particular class indicates the total number of classes directly dependent on this class (i.e. classes that derive from it). For example, if two classes X
and Y use a class Z, the Fan-in of the class Z is 2. The more the Fan-in is high, the more the product is reusable. Fan-in mostly affects reusability.

16. Fan-out

The Fan-out metric of a particular class indicates the total number of classes that are used by this class, plus the number of inherited classes (children classes). A class X using three classes W, Y and Z has the Fan-out of 4, because the object class is also counted as a class used by X. This metric affects the coupling attribute - previously defined, and therefore the quality characteristics complexity, efficiency, understandability, maintainability, reusability and reliability.

17. Depth of inheritance tree (DIT)

In the class hierarchy, DIT represents the position of the class starting from the root of the tree. It counts the number of all ancestor classes to the class in the inheritance tree, which explains that the higher the level of inheritance, the more the number of possible components to be implicated, and consequently the complexity to maintain, test and understand. A good example of illustrating the measure of DIT is the value 0 for a root class, because there are no classes above its position in the tree. DIT mainly evaluates complexity, understandability, efficiency and maintainability, but also brings great influence on reusability, portability and functionality.

18. Number of children (NOC)

This metric is measured by counting the number of classes directly derived from the class. In a software product, a high number of children make the tracking for possible errors longer to test and the performance harder to describe. But at the same time, a high number of children make the system more interesting to reuse. Beside complexity, this metric measurement evaluates maintainability, efficiency, reusability and portability.

19. Average of inheritance depth (AID)

The average inheritance depth (AID) is the DIT computed for the entire project and is only computed at the project level. AID evaluates all the quality characteristics defined in this report.
Many other metrics are defined and might be computed by tools available in the market, but in this paper, we only selected some metrics that are common to most of the tools studied and tested in this paper. These are the most interesting to discuss. Some other metrics are: McCabe Design Complexity Metric (which is the amount of interaction between modules in a system), Percent Public Data (which is the percentage of Public and Protected data within a class, Message Passing Coupling, Coupling through inheritance, Data Abstraction Coupling, etc. We refer the interesting reader to [3], [4], [5], [8] and [9].

For a more homogeneous evaluation of these metrics by different tools, we classified them according to three levels: the project level, the class level, and the method level, all summarized in Table 2.
Chapter 3

Evaluating metrics: The tools

Many tools used for evaluating software quality by computing metrics are published and implemented, but the main problem that I faced during the process of choosing some of these tools is that not all are available online. Some of them have to be purchased directly from the point of sale (not through the internet). Examples of such tools are CMT Java from Testwell [15], IQ from Mc Cabe [14], and ISO 9126 from QA Systems [12]. In the case where a trial version is not online, I tried to contact many of the companies asking for their tool, but my request was never replied (e.g. for the McCabe IQ and the CMT Java).

A detailed description of the studied and tested tools is provided next.

3.1 The Jstyle from Man Machine System

The first metric analyzer tool studied in this paper is the Jstyle from Man Machine Systems, produced in India. It has several functionalities such as collecting metrics for a Java source code and revealing source deficiencies by reviewing and testing components, based on some coding guidelines developed for Java. Jstyle serves as a source code analyzer, and is characterized by its ability to handle very large projects quickly and produce output, generating comments categories of rules and guidelines that can be customized and renewed. It is not meant to fix problems of the system being evaluated, but it is only limited to reporting its deficiencies. It eventually computes some metrics like CC, WMC, RFC, LCOM and DIT.

Jstyle can be downloaded for either a free fifteen-day trial version of the product, or it could be purchased from the following site:

http://www.mmsindia.com/jstyle.html

The price for the Single User Node-Locked License is $ 595 (which is a machine-specific license locked to one particular computer), and for the Single User Floating License is $ 895.

This tool can be used on all Windows operating systems with a minimum of 128 MB of RAM on an Intel Pentium or higher processor, and needs at least of 50 MB of free memory space for the installation on the hard disk.
The input required to use Jstyle is usually an actual Java source file in the case where we want to generate comments and/or review metrics through statistics. We simply need a compiled form of the file code otherwise (e.g. in the other functionalities of the tool).

The different steps of producing reports of metrics are the following: The user should first create a project in which the Java file of classes is added. The next step consists of generating comments on the source code -Figure 2. Then, metrics are generated and displayed. This tool has the major benefit of present detailed information and description illustrated by examples about all the metrics evaluated in it.

![Figure 2. Generating the report](image)

The output format is usually a table with a grid view of all components tested, with values computed for all metrics. The user can choose any format for the output: in a table or in a chart view (e.g. TXT, RTF, HTML, DBF...). According to the type of the metrics required, he can either choose to display summary for projects, files, classes or methods -Figure 3. Our concern is about metrics computed in the project, class and method-level reports.

In the following output report, the project summary is displayed. We could first notice the name of the project name (math), followed by the date and time report generation. Some of the project metrics are shown below (e.g. Number of packages, number of classes, number of inner classes, etc.).
3.2 Understand for Java from SIT

The second metric tool is the Understand for Java, from the Scientific Toolworks Incorporation (STI) in the USA. Besides computing metrics, this software serves as a Java code navigation, exploring, and understanding. It is particularly designed to help maintain large projects.

This tool is not only specific for Java language, but we notice for example that there are understand for C++, with the same features and functionalities. The main characteristic present in this tool is that it does not require the code to be complete. Unlike many other tools, Millions of lines of code can be integrated to this software system, which is capable of navigating the entire code file and compute all required metrics.

This tool can be used on all Windows 95/98/ME/NT/2000/XP, Linux, Solaris, HP-UX, and SGI IRIX operating systems. It needs around 24 MB of free memory space for the installation on the hard disk. The single License can be purchased for $495\textsuperscript{21}. The tool can also be downloaded as a free trial version for a period of fifteen days, through the following link:

http://www.scitools.com/uj.html

To navigate into the code, a project should first be created, and the Java classes should be added into the project package -Figure 4. The comment results shown below illustrate how the modules are being loaded and analyzed before the metric computation.
Metrics are then computed and exported (Project/Reports generate) and (Project/Metrics export), and we can either read the output reports inside the tool as html reports, or just open them outside the tool. Reports can include all the metrics or just a few from the list of metrics offered by this tool, by clicking next to the category requested (i.e. we can choose only to generate Object-Oriented metrics instead of having them all computed - Figure 5. Below is the window where metrics needed in the output report should be checked for selection.

Figure 5. Generating Metrics in Understand for Java
The output that is of our concern has the format of either html or text reports that include all types of metrics (Project metrics, class metrics, file/module metrics, methods metrics, Object-oriented metrics, etc). A brief description of the metrics computed is available in the help, and can be found with more details when using the web.

3.3 The Jmetric

The third metric and analysis tool is the Jmetric, distributed under the terms of the GPL\textsuperscript{vii}. It is produced in Australia and is characterized by its very friendly graphical user interface. It is only used for computing and reporting metrics for Java source code. This software system works on the following Windows operating systems: 95, 98, Me, NT, 2000 and XP). It requires at least 128 Mb of RAM and 1.3 Mb of the disk space. It requires one other software systems to be installed, namely the Java JDK 1.1x, Java JDK 1.2 or the Swing 1.1.

The main features of Jmetric are its ability to handle projects up to 10,000 lines of code, export text files and provide colored table and chart reports with the possibility of printing. It collects not only metrics that reflect quality characteristics of the given input program, but also analyses the size of the system's components (e.g. modules and classes). It can be downloaded for free from the following site:


As soon as the file code is loaded into the tool, the analyzer software automatically provides statistical information about the project, packages, classes, methods and variables in a panel, where the project tree view is shown to make the structure of the code more understandable to the user —Figure 6 to the left. The $P_r$ in the tree refers to the new project, the $P$ refers to the package that include all the files, the $C$ refers to the classes, the $M$ refers to the methods, and the $V$ refers to the variables inside the classes. The component highlighted in yellow is the one for which metrics are being displayed.

In the chart below, metrics are computed on the method level (MathContext method). We could observe the McCabe Cyclomatic Complexity displayed and calculated for this method —value 1, and the number of lines of code is 2. Note that the blue line refers to the values of metrics calculated for this method.
Figure 6. Example of Generating Metric Charts

The input is basically a Java source code file loaded after defining a certain project in which the output documents will be saved. Values for the computed metrics are reported in several output formats according to the choice made by the user: charts (i.e. XY scatter, star, lines –Figures 7, 8), reports, tables, tree view, that could all be exported to other format (i.e. xml, txt –Figure 9).

The basic metrics evaluated by this tool are specific for each component. For instance, at the method level, the metrics reflect the number of local and used variables, the lines of code (LOC), a statement count, and the cyclomatic complexity are the different kinds of metrics. At the class level, the metrics of concern are the number methods and variable, the lines of code (LOC), a statement count, the lack of cohesion of methods (LCOM), the coupling between object classes (defined in this tool as collaborators), the number of children and the depth of inheritance are the metrics related to the class component. Some other metric, referred to as the collaborators count, is computed, but the tool does not give any clear definition or representation about it.
Figure 7. Chart reporting some class metrics

Figure 8. XY Scatter reporting some class metrics
Figure 10. Command prompt

Figure 11. RAM Wizard main window
RSM reviews compiled source codes of many programming languages such as ANSI C/C++, C#, and Java. Incomplete code will not give correct output reports. It can integrate IDE's such as Visual Studio, .NET, JBuilder and Eclipse, and has no limitations over the number of files or the length of files -i.e. over 10,000 files and 12,000,000 lines of code.

This tool as well as the wizard do not require any specific Operating System platform, and it can work across Windows 9x, NT/2000/XP, UNIX, Linux and Mac OS X. It requires some memory in the disk space for storing output reports depending on the size of the processed file. The RSM command line program needs about 2 Mb of disk space for the installation, while the RSM Wizard needs 8 Mb.

The output is a report which includes methods, classes, files and project metrics -Figure 12. It is generated through multiple formats (e.g. HTML with hyperlinks to the code, CSV for MS Excel, or simply text documents).

RSM basically computes statistics over many kinds of metrics: it computes lines counting metrics (LOC), methods complexity metrics (Cyclomatic Complexity), number of methods and percentage of comment lines, as well as Object-Oriented metrics such as DIT, NOC, and NOP -Figure 13, but unfortunately not RFC, LCOM or CBO. RSM Wizard offers the possibility to easily choose the types of metrics to be evaluated.

---

**Figure 12. Report generated by RSM**
A brief comparison between the four tool features is provided in a table —Table 3. In the following section, some java source codes are used over the four tools, and several output reports are generated. We are next concerned in comparing computed metrics, and verifying that they are equal in all generated reports.
Chapter 4

Discussion of Results

We have tested all four tools and validated their results by computing manually the value for each metric. We did this on two software systems with Windows XP Operating System. In this section, we highlight the shortcomings and strengths that differentiate between the tools. Four output results of metrics values computed by the different tools for the same input files are generated, and the resulting numbers are compared—Table 4.

1- DIT

For a class that has one ancestor, we observe that Jstyle and Understand for Java compute the DIT and get the value 2 for some given input code, Jmetric and RSM get the value 1. This is because the latter do not count the object class as part of the inheritance depth.

2- NOC

As for the number of children metric (NOC), all tools always compute the same values for any class. Below is an example of a class having one child, where NOC is evaluated.

→ Output from RSM:

```
  + java.math.MutableBigInteger [D0,C1]
```

Where C is the Number of direct child classes. Here, D0 means that DIT for the "MutableBigInteger" class is 0, and its NOC metric is 1.

→ Output from Jmetric:

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Child Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>MutableBigInteger</td>
<td>1</td>
</tr>
</tbody>
</table>
3- AID

The average inheritance depth (AID) is only computed by Jstyle.

4- CBO

Coupling between Object Classes (CBO) is only evaluated in Understand for Java.
5- RFC

RFC is computed by Jstyle and Understand for Java only. However these two tools do not calculate the same value for this metric. We think that this is due to the fact that the response set of a class in Jstyle consists of all the methods called by local methods, including local methods and methods called from other classes, while the RFC evaluated in Understand for Java only reflects the number of local methods of the class, and discards the methods from outside classes.

6- NOM

This RFC metric just described and obtained through Understand for Java represents the NOM metric. When evaluating the number of methods or the weighted method per class (WMC), the same output is reported by all tools, which makes sense because the number of methods in this class is fixed and there is no more than one way to calculate the number of methods in it.

7- McCabe Cyclomatic Complexity

For a specific method, McCabe Cyclomatic Complexity is computed by Jmetric, Jstyle, RSM and Understand for Java. They all get the same measure as shown in the next four figures. While the first three tools report the value for the metrics at the method level with a detailed information and numbers concerning every method in a class, the last tool does it in the Program Unit Complexity report.

<table>
<thead>
<tr>
<th>CLASS Name</th>
<th>McCabe Cyclomatic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitStore</td>
<td></td>
</tr>
<tr>
<td>Constructor</td>
<td>2</td>
</tr>
<tr>
<td>set (int)</td>
<td>1</td>
</tr>
<tr>
<td>set (set)</td>
<td></td>
</tr>
<tr>
<td>niceSearch: (ri, ri)</td>
<td>4</td>
</tr>
</tbody>
</table>

McCabe Cyclomatic Complexity measured by Jmetric
Method: sieveSearch [private]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Statements</td>
<td>0</td>
</tr>
<tr>
<td>Number of Exceptions thrown</td>
<td>0</td>
</tr>
<tr>
<td>Cyclomatic Number</td>
<td>4</td>
</tr>
<tr>
<td>Program Length</td>
<td>34.6292</td>
</tr>
<tr>
<td>Actual Halstead Length</td>
<td>38</td>
</tr>
<tr>
<td>Program's Vocabulary</td>
<td>17</td>
</tr>
<tr>
<td>Program Volume</td>
<td>155.324</td>
</tr>
<tr>
<td>Program Level</td>
<td>0.0641026</td>
</tr>
<tr>
<td>Program Difficulty</td>
<td>15.6</td>
</tr>
<tr>
<td>Development Effort</td>
<td>2.24356 (mins)</td>
</tr>
<tr>
<td>Bug Predicted</td>
<td>0.0517745</td>
</tr>
</tbody>
</table>

McCabe Cyclomatic Complexity measured by Jstyle

<table>
<thead>
<tr>
<th>Function: java.math.BigInteger.sieveSearch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
</tr>
<tr>
<td>LOC 11</td>
</tr>
<tr>
<td>Function Points</td>
</tr>
<tr>
<td>Lines</td>
</tr>
</tbody>
</table>

McCabe Cyclomatic Complexity measured by RMS

Program Unit Complexity Report

<table>
<thead>
<tr>
<th>Class</th>
<th>Cyclomatic</th>
<th>McMc</th>
<th>Bug</th>
<th>Format</th>
<th>Names</th>
<th>Parts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

McCabe Cyclomatic Complexity measured by Understand for Java

26
8- WMC

The Weighted method complexity of a class is the sum of the cyclomatic complexities of the local methods in the class. All tools evaluate this metric equally. In RSM and Jstyle, under the class level, the metric is reported as WMC, while in the Jmetric and the Understand for Java, we get the cyclomatic complexities by methods in the Project Report Metrics, and hence to get the WMC value of the class, we need to add all the Cyclomatic Complexities of the local methods together to get the final result.

9- Number of classes

Jmetric does not calculate the total number of classes, but it only displays a list of the classes’ names, and the user has to count them to have the total number of classes in the Project. As for Jstyle, it includes the top-level (public and nonpublic), local and inner classes of the Java package. Understand and RSM count the totality of the classes, and display the same output results.

10- Number of Inner classes

Both Jmetric and Jstyle display for every class the number of inner classes correspondent to it, while the two other tools do not evaluate such metric.

11- Number of modules/packages

RSM, Jstyle and Jmetric calculate the number of modules/packages inside the project. But the Understand for Java tool does not have the ability to compute the modules number. Instead, it calculates the total number of Java files loaded in the program.

12- Number of Blank lines

Jmetric does not evaluate blank lines metrics like the three other tools. RSM labels this metric as eLoc (i.e. empty lines of code). The only difference between the three results obtained is that Understand for Java does not count the last empty line in the file. So instead of getting a total empty lines of 32 for example, its output is always the "actual" number minus 1 (i.e. 31).
13- Number of statements

The number of statements is a count of the source code lines except the blank and the comment lines. Both Jstyle and Jmetric include the loop statement "do....while" and count it twice (i.e. the "do" as the first statement, and the "while" as the second). The main difference between the two tools is that unlike Jstyle, Jmetric counts the method definition as one statement - e.g. the method "SieveSearch( )" shown below. This might sometimes brings some differences in the number of statements between the two results. The two other tools do not evaluate this metric.

```java
private int SieveSearch(int limit, int start)
{
    if (start >= limit)
        return -1;
    int index = start;
    do {
        if (!get(index))
            return index;
        index++;
    } while (index < limit - 1);
    return -1;
}
```

Example of a method (SieveSearch) definition in a Java package

<table>
<thead>
<tr>
<th>heading</th>
<th>Class Name</th>
<th>Method Name</th>
<th>LOC</th>
<th>Statement Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>BitSieve</td>
<td>Constructor( )</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>Method</td>
<td>BitSieve</td>
<td>bit(int)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Method</td>
<td>BitSieve</td>
<td>get(int)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Method</td>
<td>BitSieve</td>
<td>set(int)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>method</td>
<td>BitSieve</td>
<td>sieveSearch(int, int)</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>Method</td>
<td>BitSieve</td>
<td>sieveSingle(int, int, int)</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Method</td>
<td>BitSieve</td>
<td>unitIndex(int)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Method</td>
<td>BitSieve</td>
<td>Constructor(BigInteger, int)</td>
<td>19</td>
<td>13</td>
</tr>
<tr>
<td>Method</td>
<td>BitSieve</td>
<td>retrieve(BigInteger, int)</td>
<td>14</td>
<td>8</td>
</tr>
</tbody>
</table>

Number of statements measured by Jmetric

```
Method: SieveSearch [private]
----------------------------------
Number of Statements : 9
Number of Exceptions thrown : 0
```
Cyclomatic Number : 4
Program Length : 54.6292
Actual Halstead Length : 38
Program's Vocabulary : 17
Program Volume : 155.324
Program Level : 0.0641026
Program Difficulty : 15.6
Development Effort : 2.24356 (mins)
Bug Predicted : 0.0517745

Number of statements measured by Jstyle

14- Number of executable lines

The number of executable lines can only be measured through Understand for Java, and is referred to as "lines exec" in the method report metrics. It does not count the "if" statements (the line where we have "if" only, i.e. the first line of the condition), considering that they might not be executed, depending whether the condition is fulfilled or not.

15- Number of Comment lines

The number of comment lines metric in a Java source code class is measured and equally obtained by all the tools studied in this paper, except the Jmetric. In Understand for Java, it is identified in the File Metrics Report. The Jstyle computes both the number of comment lines and the percentage of comment lines over the overall source code lines.

16- Number of Lines of Code (LOC)

For the Lines of Code Metric, we obtain the same number resulting from RSM and understand for Java. The Jstyle always finds this value incremented by one, adding up the header class name with the rest of the lines. All of these tools take the open brackets to be an additional line (i.e. they do not count both brackets: opened and closed, but every pair is count once). The main difference between these tools and Jmetric is that this last one does not consider brackets as lines of code. So this number of LOC is always less than the numbers in the previous tools.

17- Fan-in and Fan-out

Jstyle is the only tool among the four studied that evaluates Fan-in and Fan-out metrics. Fan-in of a class is the number of classes that derive from it, and Fan-out of a
class X is the number of classes used by X, including inherited classes. For example, when two classes A and B extend some other class C, the Fan-in of C is two. When a class A uses some other classes B and C, the Fan-out of A is two. Understand for Java evaluates some "IFANIN" Metric to be the number of base classes, meaning that it counts the object class with the number of classes dependent to. But these results are not what we can manually compute. This metric should have some other representation, which is not clear in the Help of the tool.

18- LCOM

For the Lack of Cohesion between Methods Metric, we notice that results from Jmetric and Understand for Java are very much alike. I do not insure that they are exactly the same, because the value for this metric is a percentage, which is calculated using some functions and it is not a straightforward computation. So when the LCOM computed by Jmetric is 44% for example, the value for the same metric computed by Understand for Java is 40%. When manually computed, we also get 40%, which confirm the tools output. Jstyle evaluates three kinds of LCOM. We are interested in the one according to Li and Henry, to the exception that it gives back the number of lack of cohesion instead of the percentage. To get the right output value and compare it with the two from the other tools, we have to count the total number of methods, and do the computation to get the percentage: Out of 10 methods and data variable declarations inside a specific class A, and when 4 is the number of LCOM, the percentage is then 40%, similar to the one computed by Understand for Java. Note that RSM does not compute the LCOM Metric.

After the comparison of the metric measurements of the four tools -Jstyle, Understand for Java, Jmetric and Resource Standard Metrics, we have found that all are similar from the metric computation and representation point of view. Some further concern would be whether these tools always provide exact and accurate metric computation or not. Their quality could be examined to verify the correctness of the results provided.
Chapter 5

Conclusion

In this paper, we offer a survey of four tools used to extract metrics used to evaluate software quality. These are Jstyle, Jmetric, RSM and Understand for Java. We have concentrated on the most widely used metrics that measure software quality characteristics such as maintainability, reusability, efficiency, etc., (e.g. Lines of Code, Depth of inheritance, Coupling between Object Classes, etc.). After a detailed study and testing of the four tools, we recommend buying the Jstyle because, compared to other tools, it is easy to use, it provides a very large number of metrics and generates a diversity of the output reports.

The major benefits of this survey are two-folded: 1. The metrics extracted provide guidelines for software engineers to follow in order to build software of a desired quality. 2. These metrics can also be used to evaluate already existing software prior to investing in it. This document helps the interested individual save effort and time looking for the tool that most suits his needs. It also facilitates the interpretation of the metrics measured by each of the tools.

This survey is by no means exhaustive. It focuses on the most widely used metrics in the literature. Of course, a wider study of those metrics would be a welcome addition to this survey. Also, more tools are available on the market; we only covered what we could easily access. The rest of the tools were either accessible from the point of sale only or were lacking an online trial version.
<table>
<thead>
<tr>
<th>Quality Characteristics</th>
<th>Metrics</th>
</tr>
</thead>
</table>
| **Maintainability**     | Lines of code (LOC)  
                          | Blank lines  
                          | Cyclomatic Complexity (CC)  
                          | Percentage of comment lines  
                          | Number of executable code lines  
                          | Number of statements  
                          | Weighted method complexity (WMC)  
                          | Number of methods (NOM)  
                          | Number of files/modules  
                          | Number of classes  
                          | Response for a class (RFC)  
                          | Lack of cohesion of methods (LCOM)  
                          | Coupling between object classes (CBO)  
                          | Fan-out  
                          | Number of children (NOC)  
                          | Average of inheritance depth (AID) |
| **Efficiency**          | Cyclomatic Complexity (CC)  
                          | Lack of cohesion of methods (LCOM)  
                          | Coupling between object classes (CBO)  
                          | Fan-out  
                          | Number of children (NOC)  
                          | Average of inheritance depth (AID) |
| **Understandability**   | Lines of code (LOC)  
                          | Blank lines  
                          | Percentage of comment lines  
                          | Number of executable code lines  
                          | Number of statements  
                          | Weighted method complexity (WMC)  
                          | Number of methods (NOM)  
                          | Number of files/modules  
                          | Number of inner classes  
                          | Response for a class (RFC)  
                          | Coupling between object classes (CBO)  
                          | Fan-out  
                          | Average of inheritance depth (AID) |
| **Reliability**         | Cyclomatic Complexity (CC)  
                          | Lack of cohesion of methods (LCOM)  
                          | Coupling between object classes (CBO)  
                          | Fan-out  
                          | Average of inheritance depth (AID) |
| **Portability**         | Coupling between object classes (CBO)  
                          | Depth of inheritance tree (DIT)  
                          | Number of children (NOC)  
<pre><code>                      | Average of inheritance depth (AID) |
</code></pre>
<table>
<thead>
<tr>
<th>Functionality</th>
<th>Depth of inheritance tree (DIT)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average of inheritance depth (AID)</td>
</tr>
<tr>
<td>Reusability</td>
<td>Lines of code (LOC)</td>
</tr>
<tr>
<td></td>
<td>Blank lines</td>
</tr>
<tr>
<td></td>
<td>Average of inheritance depth (AID)</td>
</tr>
<tr>
<td></td>
<td>Cyclomatic Complexity (CC)</td>
</tr>
<tr>
<td></td>
<td>Percentage of comment lines</td>
</tr>
<tr>
<td></td>
<td>Number of executable code lines</td>
</tr>
<tr>
<td></td>
<td>Number of statements</td>
</tr>
<tr>
<td></td>
<td>Number of methods (NOM)</td>
</tr>
<tr>
<td></td>
<td>Number of files/modules</td>
</tr>
<tr>
<td></td>
<td>Number of children (NOC)</td>
</tr>
<tr>
<td></td>
<td>Number of inner classes</td>
</tr>
<tr>
<td></td>
<td>Lack of cohesion of methods (LCOM)</td>
</tr>
<tr>
<td></td>
<td>Coupling between object classes (CBO)</td>
</tr>
<tr>
<td></td>
<td>Fan-in</td>
</tr>
<tr>
<td></td>
<td>Fan-out</td>
</tr>
<tr>
<td></td>
<td>Depth of inheritance tree (DIT)</td>
</tr>
<tr>
<td>Complexity</td>
<td>Cyclomatic Complexity (CC)</td>
</tr>
<tr>
<td></td>
<td>Weighted method complexity (WMC)</td>
</tr>
<tr>
<td></td>
<td>Number of methods (NOM)</td>
</tr>
<tr>
<td></td>
<td>Number of inner classes</td>
</tr>
<tr>
<td></td>
<td>Response for a class (RFC)</td>
</tr>
<tr>
<td></td>
<td>Lack of cohesion of methods (LCOM)</td>
</tr>
<tr>
<td></td>
<td>Coupling between object classes (CBO)</td>
</tr>
<tr>
<td></td>
<td>Fan-out</td>
</tr>
<tr>
<td></td>
<td>Average of inheritance depth (AID)</td>
</tr>
<tr>
<td>Metric name</td>
<td>Project</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Lines of code (LOC)</td>
<td>Total number of lines in all files, including blank, comment, and executable lines</td>
</tr>
<tr>
<td>Blank lines</td>
<td>Empty lines in all files</td>
</tr>
<tr>
<td>Mc Cabe Cyclomatic Complexity (CC)</td>
<td>X</td>
</tr>
<tr>
<td>Percentage of comment</td>
<td>Total number of comment lines in all files</td>
</tr>
<tr>
<td>Executable lines of code</td>
<td>Total number of executable lines in all the files</td>
</tr>
<tr>
<td>Number of statements</td>
<td>Total number of statements in all files</td>
</tr>
<tr>
<td>Weighted method complexity (WMC)</td>
<td>X</td>
</tr>
<tr>
<td>Number of methods (NOM)</td>
<td>Sum of NOM of all classes</td>
</tr>
<tr>
<td>Number of files/Modules</td>
<td>Total number of source files</td>
</tr>
<tr>
<td>Number of classes</td>
<td>Total number of classes defined somewhere within the project</td>
</tr>
<tr>
<td>Number of inner classes</td>
<td>Total number of classes nested in any class inside the project</td>
</tr>
<tr>
<td>Response for a class (RFC)</td>
<td>X</td>
</tr>
<tr>
<td>Lack of cohesion of methods (LCOM)</td>
<td>X</td>
</tr>
<tr>
<td>Coupling between object classes (CBO)</td>
<td>X</td>
</tr>
<tr>
<td>Fan-in</td>
<td>X</td>
</tr>
<tr>
<td>Fan-out</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>inherited classes)</td>
</tr>
<tr>
<td>------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td><strong>Depth of inheritance (DIT)</strong></td>
<td>(AID)</td>
</tr>
<tr>
<td></td>
<td>Number of ancestors to class</td>
</tr>
<tr>
<td><strong>Number of children (NOC)</strong></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>Number of classes derived from a class</td>
</tr>
<tr>
<td><strong>Average of inheritance depth (AID)</strong></td>
<td>Average inheritance depth (AID) of DIT for all classes</td>
</tr>
<tr>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Features</td>
<td>Jstyle</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>Source</td>
<td>India</td>
</tr>
<tr>
<td>Functionalities</td>
<td>-Computes Metrics -Reports deficiencies by reviewing/testing code components</td>
</tr>
<tr>
<td>Input form</td>
<td>-Complete Java source code -Java compiled file code</td>
</tr>
<tr>
<td>Maximum Input size</td>
<td>Very large projects</td>
</tr>
<tr>
<td>Operating System</td>
<td>All MS Windows</td>
</tr>
<tr>
<td>Memory required</td>
<td>128 MB of RAM</td>
</tr>
<tr>
<td>Memory Disk Space needed</td>
<td>50 MB</td>
</tr>
<tr>
<td>Prices</td>
<td>-Single User Node-Locked $595 or Floating License $895</td>
</tr>
<tr>
<td>Metrics</td>
<td>Computed Manually</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>NOC</td>
<td>N*</td>
</tr>
<tr>
<td>McCabe CC</td>
<td>N</td>
</tr>
<tr>
<td>Weighted Method</td>
<td>N</td>
</tr>
<tr>
<td>Complexity</td>
<td></td>
</tr>
<tr>
<td>DIT</td>
<td>N</td>
</tr>
<tr>
<td>AID</td>
<td>N</td>
</tr>
<tr>
<td>CBO</td>
<td>N</td>
</tr>
<tr>
<td>LCOM</td>
<td>N</td>
</tr>
<tr>
<td>NOM</td>
<td>N</td>
</tr>
<tr>
<td>RFC</td>
<td>N=LOCAL AND OTHER CLASSES METHODS</td>
</tr>
<tr>
<td>Number of inner classes</td>
<td>N</td>
</tr>
<tr>
<td>Number of comment lines</td>
<td>N</td>
</tr>
<tr>
<td>Number of modules/packages</td>
<td>N</td>
</tr>
<tr>
<td>Blank lines</td>
<td>N</td>
</tr>
<tr>
<td>Number of statements</td>
<td>N</td>
</tr>
<tr>
<td>LOC</td>
<td>N</td>
</tr>
<tr>
<td>Fan-in / Fan-out</td>
<td>N</td>
</tr>
<tr>
<td>Number of classes</td>
<td>N</td>
</tr>
<tr>
<td>Executable lines</td>
<td>N</td>
</tr>
</tbody>
</table>
Bibliography


\(^i\) Depending on the tool, the WMC is either the weighted method complexity (e.g. the sum of complexities of all local methods in the class), or the weighted method per class (e.g. the number of methods).

\(^ii\) In general, a module is a file that includes one or several Java classes.

\(^iii\) Except the Jstyle tool, all the tools in this paper counts the Public and the Final classes in the Project.

\(^iv\) LCOM is defined and computed by three different ways according to Chidamber and Kemerer [3], Li and Henry [2] and Henderson and Sellers [11]. For all the tools, we will look at the LCOM measured according to Li and Henry.

\(^v\) A method pair is defined as the combination of all methods in the class two-by-two. For example, a class having 4 methods will have 6 method pairs.

\(^vi\) Packs of 5, 10 and 100 Licenses can also be purchased for $1975, $2950 and $14990 respectively. Some additional fees should be added (maintenance, shipping, Manual and Upgrades).

\(^vii\) The GNU General Public License (GPL)

\(^viii\) Check the site for updated and more detailed prices list

\(^ix\) Graphical user Interface

\(^x\) N is a positive number

\(^xi\) This tool does not compute such Metric