CORE

Metadata, citation and similar papers at core

Provided by idUS. Depósito de Investigación Universidad de Sevilla

Open source tools for measuring the Internal Quality of Java software products. A survey

P. Tomas, M.J. Escalona , M. Mejias

Department of Computer and Systems, ETS Ingenieria Informatica, University of Seville, Av. Reina Mercedes S/N, 41012 Seville,Spain

A B S T R A C T

Keywords:
Software product
Tools
Open source
Metrics
Internal Quality
Automation
Static analysis
Source code
Java

Collecting metrics and indicators to assess objectively the different products resulting during the lifecycle of a software project is a research area that encompasses many different aspects, apart from being highly demanded by companies and software development teams.
Focusing on software products, one of the most used methods by development teams for measuring Internal Quality is the static analysis of the source code. This paper works in this line and presents a study of the state-of-the-art open source software tools that automate the collection of these metrics, particularly for developments in Java. These tools have been compared according to certain criteria defined in this study.

Contents

1. Introduction

Metrics constitute a good way to understand, monitor, control, predict and test software development and maintenance projects. Establishing systems for measuring software product quality is a key issue within the software quality control, especially regarding the current trend towards outsourcing developments, which is often performed by external teams or external software factories. Moreover, a software product quality measurement system is efficient if it is highly automated and can be used on a frequent basis without excessive waste of time [1].

The fact of having tools to automate both the acquisition and presentation of the values of metrics offers some important advantages [2] such as:

- It allows obtaining values from the analyzed metrics doing the least possible effort. The hardest work will have to be done at the beginning and it will require the effort of setting the tool used to obtain the metrics.
- It reduces metric calculation errors, achieving greater accuracy in their values.
- It allows focusing on the analysis of measurement results rather than acquisition results.
- It defines minimum and maximum thresholds, beyond which the values for certain metrics are directly presented as a warning or error.

In the area for measuring the Internal Quality of Software Products, some relevant standards (ISO and IEC) offer initial frameworks to support it. Thus, ISO/EC 13598 [3], which is composed of six parts, offers a suitable framework to value the quality of each kind of software product and defines requirements that have to be supported by measure methods and processes.

Another important reference, *ISO/IEC 9126* [4], which is divided into four parts, defines some specific aspects in software products (part 1) as well as Internal and External metrics and metrics in use (parts 2, 3 and 4). This standard describes three different types of quality: Internal and External quality and quality in use.

Currently, the subcommittee SC7 (Software Engineering and System) of the Technical Committee ISO/IEC JTC1 (Information Technology) is reviewing both standards to define a new one that may cover both ISO/IEC 9126 and ISO/IEC 14598. This new approach is called *ISO/IEC 25000* [5], frequently known as SQuaRE (Software Quality Requirements and Evaluation). It mainly focuses on providing guidelines on software product development by supporting right specification and evaluation of quality requirements. Thus, SQuaRE defines a set of criteria to specify quality requirements for software products as well as their metrics and evaluation. SQuaRe consists of five divisions and a set of extensions and technical reports. Thus, *ISO/IEC 25000* substitutes ISO/IEC 9126—Part 1.

This paper does not aim to present a global study on both quality in use and quality of the product (Internal and External). Thus, it will be restricted to the Internal Quality of the products and the remaining will be proposed as future work.

According to [6], the static analysis of the code would be defined as a set of analysis techniques where the software studied is not executed (in contrast to the dynamic analysis), but analyzed. For this reason, this type of analysis will allow obtaining Internal Quality metrics, as it does not require a software in use to be measured.

This paper focuses on Internal Quality metrics of a Software Product and software tools of static code analysis that automate measuring these metrics. Keeping in mind the variety of technologies in the present software, studying measurement tools for all of them would exceed a single paper. For this reason and due to its great popularity, this paper only focuses on source code or compiled programs of Java technology, thus covering a large part of the market products accounting for about 20% of software products. This fact is showed in the index TIOBE [7] of March 2011, in which Java stands as the most used language with a share of 19.711%. However, while this study focuses on Java technology, other technologies are proposed in Section 6 for future work.

The comparative analysis of tools in this paper has been carried out by means of the proposed guide for Systematic Literature Review (SLR) included in [8]. This process consists in three main activities:

1. *Planning*. In this phase, the survey is planned and delimited and a specific protocol to set it up must be defined. Both the aims of the survey and the environment and sources to identify the approach of study must be clearly and completely defined.
2. *Conducting the review*. In this phase, after the initial constraints have already been specified, the subject of study must be identified and reviewed through different sources. Then, the set of relevant approaches to be analyzed is identified. SEG (Software Engineering Group) proposes to establish a common characterization criterion to define each approach before executing the comparative study. It enables to obtain a uniform definition for each approach that may facilitate such comparative study.
3. *Reporting the review*. Finally, once the approaches have been assessed and the research situation has already been analyzed, the process concludes with a report on the results of the review. In fact, this paper represents the result of our review. Previous phases were executed before writing this paper, as referenced.

Despite that the number of tools found in the revision is not as large as the number of paper that is frequently included in a SLR, which is helpful in case of thousands of papers that should be manually analyzed without an established methodology, it is considered to offer a well-defined and useful process to compare these tools.

Thus, to present the results, this study is structured as follows: Section 2 introduces other related studies on static analysis tools for

Java source code while Section 3 defines the scope of the study setting out the requirements that a tool has to meet to be included in it. To follow, Section 3.1 introduces a characterization scheme to standardize the information of each of the approaches found. This section involves the first and second phases of SEG (Software Engineering Group) [8]. Section 4 offers the state-of-the-art tool analysis, which draws the information indicated in the characterization scheme obtained in Section 3 for each tool. Section 5 compares the tools assessed according to the data obtained in Section 4, and discusses in detail how the tools cover one of the features present in that scheme (the Internal Quality model). Concluding the report is Section 6 by summarizing the study and comparison of tools, and pointing out several lines of research for future work.

2. Related work

There are at least three previous studies related to static analysis tools of Java code:

A. In [8] Rutar et al. compare five bug detection tools using static analysis of Java code. The conclusion of this study is that although there are some overlaps among the types of errors detected, most of them are different. They also state that using these tools is a difficult task due to the number of results they generate. The present work examines three of these tools (FindBugs, PMD and Jlint) as they obtain the *code smells* metric. Code smells define certain structures in the code that suggest the possibility of refactoring because of potential bugs.
B. In [9] Lamas compares two tools, FindBugs and PMD, which are also analyzed both in the previous and in the present study. The conclusion is that the tools are complementary in terms of the bugs detected, as there are few overlaps among them. It also concludes that looking for bugs implies more than one static analysis tool.
C. In [10] Ayewah et al. also analyze the FindBugs tool, the kind of warnings generated and their classification into *false positives* (warnings that aren't really defects), *trivial bugs* (true defects with minimal impact) and *serious bugs* (defects with significant impact).
D. In [11] van Emden et al. present an approach for the automatic detection and visualization of *code smells* with jCOSMO, and discuss how this approach can be used in the design of a software inspection tool.
E. In [12] Artho et al. analyze how incorrect thread synchronization produces faults that are hard to find in multi-threaded programs. Jlint1's model is extended to include synchronizations creating a new version Jlint2.
F. In [13] Hovemeyer et al. describe a static analysis using FindBugs to find null pointer bugs in Java programs, and some of the simple analysis techniques used to make it more accurate.
G. In [14] Ruthruff et al. study how to help address two challenges complicating the adoption of static analysis tools for the detection of software defects: spurious false positive warnings and legitimate warnings that are not acted on. Sampling from a base of tens of thousands of static analysis warnings from Google, they have built models that predict whether FindBug warnings are false positives and, if they reveal real defects, whether these defects would be acted on by developers ("actionable warnings") or ignored despite their legitimacy ("trivial warnings").
H. In [15] Cole et al. also analyze the FindBug tool, which is based on the concept of bug patterns. A bug pattern is a code idiom that is often an error.

Previous studies found focus on tools that explore the code from a single point of view, that is, bugs or code smells. This study extends the analysis to a total of eight metrics. It also compares the typical functionality of metric tools and Quality models that they implement.

3. Planning and conducting the review

This section defines the scope of approaches (tools) that are relevant to this study. One problem is how to present each approach in a homogeneous way to compare them. According to [8], the characteristics that approaches should fulfill must be consistent with the thesis of this survey: *open source software tools that automate the acquisition of Internal Quality metrics of Software Products using static analysis of Java source code*. Following the terminology described in [12] to concretely plan and develop our review, the concepts indicated below are defined:

Context: Establishing metrics for measuring software product quality is a basic piece for controlling software quality. A system for measuring the software product quality is efficient, if it has a high level of automation and enables its use on a frequent basis without excessive waste of time. The thesis of this study focuses on automating the production of Internal Quality metrics by means of software tools that perform static analysis of Java source code.
According to [2], static analysis tools are those that perform the analysis without running the software under study. This type of analysis can be performed on the source code or the bytecode (compiled code).
Objectives: The main objectives of this study are as follows:

- Study the state-of-the-art software tools that can support Internal Quality metrics.
- Establish a framework to enable a comparison by means of cataloging these tools.
- Explore whether any of these tools is able to establish a relationship between models and Internal Quality metrics. It means, if there are any tools that can implement some of any Internal Quality models using metrics and, in this case, how they can carry it out.
- Draw conclusions and propose future work.

Methods: The following sources of information will be used for searching tools:

- Google Scholar.
- IEEE/ACM digital libraries.
- Java Power Tools book [14]. Compendium of 30 open source tools designed to improve development practices in Java.
- ISO/IEC 25000 portal [2]. Portal of ISO/IEC 25000 (evolution of the ISO/IEC 9126) which contains a small but significant section of measurement tools.
- Ohloh.net portal [15]. Social network of developers, projects and forums of open source software, which references more than 500,000 projects.

These criteria for inclusion, exclusion and grouping tools have been applied when searching in these sources of information:

- Open source tools that perform automatic static analysis of Java code have been included, in the form of either source code or compiled code (bytecode), which allows measuring, analyzing or presenting Internal Quality.
- In other words, dynamic analysis tools, tools that analyze code in languages other than Java, tools that are not open source (proprietary), tools for manual code review or tools that do not measure, analyze or present Internal Quality, have been excluded.
- Those tools that, in spite of being documented or referenced in the literature, are not available for download as they do not meet the open source software requirement of free redistribution have been excluded.
- Tools whose operation could not be tested with a sample project have been excluded.
- Tools that only differ in their execution, like those that can be run from the command line or as a plugin of an integrated development environment (IDE), have been grouped.

A specific search has been conducted for each source of information:

Manual searches
- Tools in the book *Java Power Tools*, chapter "Quality metrics tools".
- Tools in ISO/IEC 25000 portal, section "Open Source Measurement Tools".

Automatic searches
- Google Scholar was previously consulted to find out related work following these criteria: *quality metrics*, "*static analysis*", "*open source*" *tools Java*. Among the most relevant results, four articles directly associated with the tools assessed in this study have been found [16–19].
- The search tool in Ohloh.net has been used to find projects dealing with these criteria: *tag:java tag:code_analysis-tag: coverage.-tag:coverage* indicates that the results exclude the code coverage tools, a very common type of dynamic analysis tools: code coverage points out the amount of code that is subject to unit tests.

Results: To select the tools that will be studied, the search process for each source has been applied taking into account inclusion, exclusion and clustering criteria. The result of the selection process is available in Table 1. In this table, the column "Year" refers to the year of the latest version of the main tool, "Group of tools" refers to the tools that have been grouped under the name of the main tool, indicating with a "+" each of the other tools that have been grouped with the main tool. "Reference" refers to the URL where the main tool is available.

Table 1 includes 3 tools (shaded) that do not fulfill the characteristics of this study. They appear at the end of Section 3 together with the reasons why they are not included in this study.

3.1. A characterization schema

According to the characterization schema definition, the next step consists in identifying a common template to describe each approach. This scheme allows storing the information of the approaches in a common template in order to more easily compare them. The characterization scheme must answer questions about the characteristics of the tools that are consistent with the thesis of the study: *open source software tools that automate the acquisition of Internal Quality metrics of Software Products using static analysis of Java source code*. These questions are:

- Q1. Which are the Internal Quality models implemented by the tools? Software product conceptual models of Internal Quality detail which are the intrinsic characteristics from which such quality can be measured. These models can be research reports, norms and standards that enable defining a common vocabulary that supports the industry and represents a reference framework.
- Q2. Which are the Internal Quality metrics measured by the tools? Standards are conceptual and abstract models that cannot be tied to specific technologies, so tools calculate specific Internal Quality metrics of Java code. From these metrics, tools raise the level of abstraction up to the conceptual model by using intrinsic characteristics as intermediate level, thus following a hierarchical structure.
- Q3. Which are the functional features covered by the tools? The tools under study, like any software, are designed and built to meet functional requirements.
- Q4. Which is the degree of maturity of these tools? When comparing these tools, the models implemented, the metrics measured or the functions covered are not only relevant, but also their evolution level as a symptom of more reliable and efficient software.
- Q5. Is the maintenance of these tools still alive? As in any software, we would like to check that these tools are still being developed and maintained since they are tools that have been accepted in the developer community and have not been discontinued.

The information extraction template for cataloging and comparing tools, which answers the questions raised above, is provided in Table 2 and subsequent paragraphs. Selecting this specific characterization schema was not an easy task. It focused on answering the research questions previously formulated. Possibilities, such as being aligned with standards, such as ISO 25000, should be also considered to develop this scheme. However, as we aimed to get a specific answer to our questions we decided to define our own table.

Internal Quality models supported. This attribute will allow us to investigate the possible relationship between metrics and Quality models. It is used to answer Q1. After performing a preliminary study of the models implemented in tools in Table 1, these below have been obtained:

- *ISO/IEC 9126-1:2001 "Part 1: Quality model"* [13]. It is based on McCall [39] and Boehm [40] models. Like them, it has a hierarchical structure of quality characteristics. In the present study the External and Internal Quality model of the two models of ISO 9126-1 are only considered (External and Internal Quality, and quality in use). The External and Internal Quality model structures software quality attributes in 6 characteristics or factors: *functionality*, *reliability*, *usability*, *efficiency*, *maintainability* and *portability*, which in turn are divided into 27 sub-characteristics or criteria. Although sub-characteristics can be measured by Internal and External metrics, this study only considers internal metrics.
- *ISO/IEC 25010 "System and software Quality models"* [41]. It is an updating of ISO/IEC 9126-1 that combines the Internal and External Quality to define the quality of a product. The quality of a product

Table 1
Selection of tools.

Year	Group of tools	Reference
2002	jCosmo	[20]
2005	Jdepend (+JDepend plugin for Eclipse)	[21]
2005	JCSC	[22]
2006	QALab	[23]
2007	CKJM	[24]
2007	Panopticode	[25]
2007	Same	[26]
2009	FindBugs	[27]
2009	JavaNCSS	[28]
2009	PMD/CPD	[29]
2009	Xradar	[30]
2010	JCCD	[31]
2011	Checkstyle (+Eclipse Checkstyle Plugin)	[32]
2011	Sonar (+Sonar IDE)	[33]
2011	Classycle	[34]
2011	CodeCrawler/Moose	[35]
2011	Jlint	[36]
2011	Sonar Plugins	[37]
2011	Squale (Software QUALity Enhancement)	[38]

Table 2
Characterization scheme for the description of tools.

Attributes	Dominion
Internal Quality models supported	{ISO 9126, ISO 25010, SQUALE, SIG}
Metrics implemented	{Complexity, CK, code size, comment size, coding convention violations, code smells, duplicated code, dependencies}
Functional features covered	{Data acquisition, analysis of measures, data presentation}
Year of first version	Year
Year of last version	Year

structures the quality properties of software systems and products in 8 characteristics: *functional suitability*, *performance efficiency*, *compatibility*, *usability*, *reliability*, *security*, *maintainability* and *portability*. Similarly to ISO/IEC 9126-1, each characteristic is divided into sub-characteristics.

- *SQUALE (Software QUALity Enhancement)* [42]. In the same way of that of ISO 9126 model, this model is based on McCall model [39]. However, ISO 9126 uses a 3-tier model, whereas SQUALE adds a new level of practices between metrics and criteria, which is obtained by applying formulas based on the metrics for each element of the code. The three upper levels of SQUALE (factors, criteria and practices) are evaluated according to a 0–3 range: between 0 and 1, it represents a goal not achieved; between 1 and 2, it represents a goal achieved, but with reservations, and between 2 and 3, it represents a goal achieved. The criteria and factors of SQUALE are adaptations of ISO 9126 levels. It defines 15 criteria and 6 factors, being the factors: *functionality*, *architecture*, *maintainability*, *evolutivity*, *reuse capacity* and *reliability*. Practices are evaluated through code metrics, UML model metrics, rule checking metrics, dynamic analysis metrics and human analysis metrics, both manual and automatic, where applicable. The present study only takes into account the metrics that can be obtained automatically from static code analysis.
- *SIG Maintainability Model* [43]: It is a model that maps source properties in the four sub-characteristics of *maintainability* of ISO 9126: *analyzability*, *changeability*, *stability* and *testability*. It uses the following properties for this mapping: *volume* (total), *complexity per unit* (method), *duplication* (total), *unit size* (method) and *unit testing* (method). The latter refers to dynamic analysis and therefore, it is not included in this study. The scale ++/+/0/−/−−, being ++ the best and − the worst, is used to evaluate properties.

3.1.1. Metrics implemented

The more metrics a tool implements, the more complete it will be and the better it will meet our needs. It is only applicable for tools covering data acquisition otherwise it will be empty. It is used to answer Q2. After carrying out a preliminary study of the implemented metrics in tools represented in Table 1, the following results have been obtained:

- *Complexity metrics*. They are derived from McCabe's Cyclomatic Complexity Number (CCN) [44], which is based on the number of independent paths.
- *CK metrics*. Chidamber and Kemerer [45] proposed six design metrics in object-oriented classes, which later became what is commonly known as the CK metric suite: Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Object classes (CBO, also known as Efferent Couplings, Ce), Response For a Class (RFC), and Lack of Cohesion in Methods (LCOM). Normally, this category also includes Ca (Afferent Couplings).
- *Code size metrics*. Metrics of *number of packages*, *number of classes*, *Number of Methods* (NOM), *LOC* (Lines of Code) and *NCLOC* (Non-Comment Lines of Code), and *NCSS* (Non-Commenting Source Statements) are often used in the Java community as code size metrics. NCSS was defined as part of JavaNCSS [28].
- *Comment size metrics*. The *number of single-line comments*, *number of block comments*, and *number of javadocs*, among others, can be measured in order to quantify documentation. However, there are more significant metrics such as *density of comment lines* or *density of javadocs* [46].
- *Number of coding convention violations*. The number of coding convention violations is frequently used as a quality metric for readability. Sun Code Conventions for Java [47] are among the most widespread coding conventions.
- *Number of code smells*. Code smells metaphor was coined by Martin Fowler and Kent Beck [48] to define certain structures in the code that suggest the possibility of refactoring because of potential bugs. There are several ways to recognize these code smells, such as bad coding practices or bug patterns.
- *Amount of duplicated code*. A measure of the amount of duplicated code, for example in the form of *duplicated lines*, *duplicated blocks* or *duplicated tokens*, is an indicator of maintainability and reliability. Thus, if the duplicated code is changed, then this change will have to be executed to all duplicates. They are areas of high risk due to potential errors, since a defect that occurs in a particular code may occur in the same way in all the duplicates.
- *Dependency metrics*. Robert C. Martin [49] proposed three principles dealing with the relationships among components which described metrics to measure and characterize the dependency structure of the design:
    - The Acyclic Dependencies Principle (ADP): “Allows no cycles in the component dependency graph”. The metric *number of cycles of dependencies* among components is defined.
    - The Stable-Dependencies Principle (SDP): “Depends in the direction of stability”. The metric *Instability* (I) is defined, thus the I value of a component must be higher than the I value of the components it depends on. It is calculated from *Ca* (Afferent Couplings) and *Ce* (Efferent Couplings).
    - The Stable-Abstractions Principle (SAP): “A component should be as abstract as stable”. To measure the balance between Abstraction (A) and Stability (I), the metric *Distance from the main sequence* (D) is defined.

3.1.2. Functional features covered

It indicates whether it is a comprehensive tool that fully covers all features and needs at all levels, or by the contrary, it is a tool that partially covers needs and requires some others to complement it. It is used to answer Q3. According to Giles and Daich [45,50], the three main tasks that metric tools must perform are:

- *Data acquisition*. It includes a set of methods and techniques for obtaining necessary data for measurement.
- *Analysis of the measures*. It includes the ability to store, retrieve, manipulate and perform data analysis.
- *Data presentation*. It provides formats to generate the obtained documentation. There are some examples of possible representation such as tables and graphs or exporting files to other applications.

The process of measuring quality could be implemented following this order: first, data acquisition; second, analysis of the measures; finally, data presentation. The data acquisition phase represents the lowest level of abstraction, since it achieves a large set of numerical data. Then, the analysis of measure phase summarizes and interprets the data. Finally, the data presentation phase makes them understandable, becoming the highest level of abstraction.

3.1.3. Year of first version and year of last version

They are used to answer Q4 and Q5. By means of the difference between the first year and the latest year, the years of development indicating the maturity of the tool are obtained. The year of last version reveals if the tool (maintenance) development is still alive today.

4. Characterization of tools

This section represents the study of the state-of-the-art tools responsible for measuring Internal Quality of the Software Product.

Table 3
Characterization scheme for Jdepend.

Attributes	Dominion
Internal Quality models supported	None
Metrics implemented	Dependencies
Functional features covered	Data acquisition
Year of first version	2001
Year of last version	2005

Tools are displayed in chronological order by the years of last version, and the information shown in the template of Section 3.1 above is extracted for each tool.

4.1. Jdepend

Jdepend [21] is a tool that analyzes directories of compiled Java files (.class or *bytecode*) and generates design quality metrics for each of the packages. It produces metrics of dependencies among packages: Afferent Couplings (Ca), Efferent Couplings (Ce), Abstraction (A), Instability (I), and Distance from the main sequence (D). It also reports whether a packet is involved in a cycle. Table 3 represents the characterization scheme for Jdepend.

4.2. JCSC

JCSC [22] checks source code against a highly definable coding standard, which by default is "Sun Code Conventions" [47]. It also checks potential bad code, such as empty catch/finally block, switch without default, throwing of type 'Exception' and slow code. It is rule-based operating. It also collects NCSS and CCN metrics. Table 4 represents the characterization scheme for JCSC.

4.3. QALab

QALab [23] collects and consolidates data from several QA tools and keeps track of them overtime. Among other tools, it collects data from the following ones: Checkstyle, PMD/CPD and FindBugs. QALab generates two types of reports: Charts and Movers. Charts track the evolution of data from the beginning of the project or from QALab installation. Movers let users see at a glance what has changed since the last execution of QALab. Table 5 represents the characterization scheme for QALab.

4.4. CKJM

CKJM [24] calculates Chidamber and Kemerer object-oriented metrics (WMC, DIT, NOC, CBO, RFC and LCOM), and some others which do not belong to CK suite (Afferent Couplings, Ca, and number of public methods or NPM), by processing the bytecode of compiled Java files and displaying the results for each class.

Table 6 represents the characterization scheme for CKJM.

4.5. Panopticode

Panopticode [25] provides a standardized format for describing the structure of software projects as well as integrates metrics from several tools into that format through a set of Ant files (Ant [54] is a software tool for automating software build processes which uses XML files to describe the build process and its dependencies). It also provides a set of open source tools (plugins) for gathering, correlating, and displaying code metrics. Although the documentation indicates that it collects metrics from different tools, the unique version available only integrates metrics from Jdepend (*dependencies*) and JavaNCSS (*complexity*, *code size*, *comment size*) to be displayed in a treemap report [51]. [51] represents the characterization scheme for Panopticode (Table 7).

Table 4
Characterization scheme for JCSC.

Attributes	Dominion
Internal Quality models supported	None
Metrics implemented	Complexity, code size, coding convention violations, code smells
Functional features covered	Data acquisition
Year of first version	2002
Year of last version	2005

Table 5
Characterization scheme for QALab.

Attributes	Dominion
Internal Quality models supported	None
Metrics implemented	None
Functional features covered	Data presentation
Year of first version	2005
Year of last version	2006

4.6. Same

Same [26] detects duplicated code chunks within a set of Java files. It normalizes Java code and is able to find duplicated code chunks even when the formatting is radically different, when the variable name has changed, and even when constants have changed. Table 8 represents the characterization scheme for Same.

4.7. FindBugs

FindBugs [27] looks for bugs in Java programs by means of a static analysis of the *bytecode* (compiled class files). It is based on the concept of *bug patterns*. A bug pattern is a code idiom that is often an error. Bug patterns arise for a variety of reasons: difficult language features, misunderstood API methods or misunderstood invariants when the code is modified during maintenance, among others. Since its analysis is sometimes imprecise, FindBugs can report false warnings, which are warnings that do not indicate real errors. In practice, the rate of false warnings reported by FindBugs is less than 50%. FindBugs supports a plugin architecture allowing anyone to add new bug *detectors*. FindBugs uses *data flow* and *syntactic* analysis to detect bugs [9]. Table 9 represents the characterization scheme for FindBugs.

4.8. JavaNCSS

JavaNCSS [28] is a tool that globally, by package, class and method measures the code size (number of packets, number of classes, Number of Methods and NCSS), comment size (javadocs) and complexity (CCN) on the Java source code. Table 10 represents the characterization scheme for JavaNCSS.

4.9. PMD and CPD

PMD [29] is a powerful static analysis tool including a comprehensive set of rules which can be configured. PMD scans Java source code and looks for potential problems like possible bugs, dead code, suboptimal code and overcomplicated expressions, for instance. It is based on sets of validation rules or *rulesets*. Each ruleset comprises a set of rules, and every rule corresponds to a code checking. Most of the rules of PMD look for *bad coding practices* to avoid potential errors resulting from the experience of a team of expert programmers in Java. PMD uses *syntactic* analysis to detect bugs [9].

PMD also includes a module known as CPD "Copy Paste Detector", which can detect the duplicated code existing in the program, and therefore measure the number of blocks, lines and duplicated tokens. Table 11 represents the characterization scheme for PMD and CPD.

Table 6
Characterization scheme for CKJM.

Attributes	Dominion
Internal Quality models supported	None
Metrics implemented	CK
Functional features covered	Data acquisition
Year of first version	2005
Year of last version	2007

Table 7
Characterization scheme for Panopticode.

Attributes	Dominion
Internal Quality models supported	None
Metrics implemented	Complexity, code size, comment size, dependencies
Functional features covered	Data acquisition, data presentation
Year of first version	2007
Year of last version	2007

Table 9
Characterization scheme for FindBugs.

Attributes	Dominion
Internal Quality models supported	None
Metrics implemented	Code smells
Functional features covered	Data acquisition
Year of first version	2003
Year of last version	2009

4.10. Xradar

Xradar [30] is a code report tool currently supporting Java based systems. It produces HTML/SVG reports presented in tables and graphs. It gets results from several plugins: Checkstyle, CKJM, FindBugs, JavaNCSS, JDepend and PMD/CPD. It measures complexity metrics (*CCN*), CK (*WMC*, *DIT*, *CBO*, *RFC*, *LCOM*, *Ce*, *Ca*), code size (*NCSS*, *NOM*, *number of classes*), comment size (*number of javadocs*, *number of single-line comments*, *number of block comments*), coding convention violations, code smells, duplicated code (*duplicated lines*, *duplicated blocks*, *duplicated tokens*) and dependencies (*number of cycles*, *I*, *D*). The most important report in Xradar is a spider graph (or radar graph) representing the values of 12 metrics which are calculated by this tool, within 4 domains (Architecture (ARCH), Design (DES), Code Quality (CODE) and Test Suite (TS)), as part of the calculation of *Total Quality* (TQ). This overall measure inspired the Total Quality Sonar Plugin [37]. Table 12 represents the characterization scheme for Xradar.

4.11. Checkstyle

Checkstyle [32] automates the process of checking Java code that adheres to a coding standard such as Sun Code Conventions [47]. It is highly configurable and can be made to support almost any coding standard. Its operation is based on validation rules, which are equivalent in most cases to coding conventions, so *rule violations* allow measuring coding conventions violations. Even though this is its main functionality, since version 3 it can identify class design problems, duplicated code, or bug patterns. Table 13 represents the characterization scheme for Checkstyle.

4.12. Sonar

Sonar [33] is an open platform to manage Code Quality in a continuous way. It mainly consists of two executable components: a maven plugin [52] that performs static analysis and a web application [33] that stores metrics in a database and presents them. To perform static analysis it invokes tools or plugins like Checkstyle, PMD/CPD, FindBugs and Squid (a rewritten version of JavaNCSS specifically for Sonar). By using these plugins, it is able to gather metrics in all categories: code size (*Lines of Code*, *Classes* and some others related), comment size (*Density of comment lines* and some other related), duplicated code (*Density of duplicated lines* and some others related), complexity (*Average complexity by method*, *Average complexity by class*, *Average complexity by* file and some others related), coding convention violations and code smells (*Violations* and some others related), dependencies (*Package tangle index*, *Package cycles*, *Package dependencies to cut*, *File dependencies to cut* and some others related), and CK metrics (*LCOM and RFC*). It shows a radar graph indicating the extent to which 5 out of the 6 categories or characteristics of ISO 9126 are followed [13], in terms of analysis and presentation of data, since Functionality does not appear. Table 14 represents the characterization scheme for Sonar.

4.13. Classycle

Classycle [34] analyzes the static class and package dependencies. It is similar to JDepend which does also a dependency analysis but only on the package level. Besides, it can measure the number of classes and packages dependency cycles. For each analyzed class/package it can measure the number of classes/packages directly referring to this class/package, and the number of internal classes/packages directly used by this class/package. In addition, Classycle can search for unwanted dependencies and group classes/packages into layers. It also analyzes the compiled class files or *bytecode*. Table 15 represents the characterizations scheme for Classycle.

4.14. Jlint

Jlint [36] analyzes compiled Java code (*bytecode*) to find bugs, inconsistencies and synchronization problems by doing data flow analysis and building a lock graph. It uses *syntactic* and *data flow* analysis [9] and identifies 3 types of problems: synchronization (due to the use of threads, such as deadlocks), class inheritance (mismatch of methods profiles or components shadowing, among others) and data flow (value ranges of expressions and local variables). Table 16 represents the characterizations scheme for Jlint.

4.15. Sonar Plugins

Sonar [33] can be extended by means of plugins. Although there is a variety of plugins [37], the open source plugins that are directly related to the Internal Quality of Java code are shown below:

- *Security rules*. It groups the number of security rules violations of FindBugs and PMD (*code smells*).
- *Useless Code*. It reports on the number of lines that can be reduced, either for being a *duplicated code* or a dead code, using for the latter unused code rules of PMD (*code smells*).
- *Quality Index*. It combines a global measure (quality index) of quality with a measure of the method complexity (complexity factor). The Quality Index uses 4 metrics: *code smells* from PMD, distribution function of the method *complexity*, test coverage (which refers to

Table 8
Characterization scheme for Same.

Attributes	Dominion
Internal Quality models supported	None
Metrics implemented	Duplicated code
Functional features covered	Data acquisition
Year of first version	2007
Year of last version	2007

Table 10
Characterization scheme for JavaNCSS.

Attributes	Dominion
Internal Quality models supported	None
Metrics implemented	Complexity, code size, comment size
Functional features covered	Data acquisition
Year of first version	1997
Year of last version	2009

Table 11
Characterization scheme for PMD and CPD.

Attributes	Dominion
Internal Quality models supported	None
Metrics implemented	Code smells, duplicated code
Functional features covered	Data acquisition
Year of first version	2002
Year of last version	2009

Table 13
Characterization scheme for Checkstyle.

Attributes	Dominion
Internal Quality models supported	None
Metrics implemented	Coding convention violations, code smells
Functional features covered	Data acquisition
Year of first version	2001
Year of last version	2011

the External Quality), and *coding conventions violations* from Checkstyle. It uses a formula to give a global note (QI) between 0 and 10 to the project. The complexity factor measures the complexity density through the distribution function of the methods *complexity*, obtaining a summary value (CF) by means of a formula.

- *SIG Maintainability Model*. It implements the Software Improvement Group Maintainability Model [43]. Among the metrics proposed in the model for each code attribute, it selects the following: global LOC (*size*) for volume, CCN per method for unit *complexity*, *duplicated code* density for duplication, LOC per method for unit *size* and test coverage (which refers to External Quality). It shows the sub-characteristics (analyzability, changeability, stability and testability) on a graph.
- *Technical Debt*. It calculates the technical debt (cost to amend the bad quality of software) from the following 6 metrics: *duplicated code*, *coding conventions violations and code smells*, *complexity*, *test coverage* (which refers to the External Quality), *comment size* and *dependencies between files*. It calculates, by means of formulas, the effort to correct all defects in man days, the cost of correcting them given in dollars and the percentage of the technical debt versus the total possible debt of the project (called TPD, being this the technical debt when metrics have the worst value for quality).
- *Total Quality*. It combines four domains measures in order to calculate a global and unified project quality health (TQ): Architecture (ARCH), Design (DES), Code Quality (CODE) and Test Suite (TS). The metrics used in each domain are: Architecture domain, *dependency metrics*; Design domain, *CK metrics*; Code Quality domain, metrics of *comment size*, *coding convention violations*, *code smells* and *duplicated code*; Test Suite domain, test coverage metrics (External Quality). TQ metric is similar to that obtained by Xradar [30], as the main formula is the same:

$$TQ = 0.25{*}ARCH + 0.25{*}DES + 0.25{*}CODE + 0.25{*}TS.$$

Nevertheless, they differ in the formulas of each of the domains:

In Sonar:

- $ARCH = 100 - TI$
- $DES = 0.15 * NOM + 0.15 * LCOM + 0.25 * RFC + 0.25 * CBO + 0.20 * DIT$
- $CODE = 0.15 * DOC + 0.45 * RULES + 0.40 * DRYNESS$
- $TS = 0.80 * COV + 0.20 * SUC.$

In Xradar:

- $ARCH = 0.4 * MOD + 0.6 * COH$
- $DES = 0.20 * NOM + 0.30 * RFC + 0.30 * CBO + 0.20 * DIT$
- $CODE = 0.15 * DOC + 0.4 * DRY + 0.3 * FRE + 0.15 * STY$
- $TS = 0.5 * TSC + 0.5 * TMR.$

Table 17 describes the metrics used in the Total Quality plugin and Xradar.

All these plugins analyze the measurements and present the data by delegating the data acquisition of metrics to the platform provided by Sonar. Table 18 represents the characterization scheme for Sonar Plugins.

4.16. Squale (Software QUALity Enhancement)

Squale [38] consists of two executable components: a web application that presents metrics (SqualeWeb) and a batch process developed in Java that performs the analysis of source code (Squalix) by means of a database that stores the metrics. Squalix invokes the following plugins for static analysis: Checkstyle, JavaNCSS, CKJM, PMD/CPD and Jdepend. It can gather metrics in all categories by using these plugins: complexity (*CCN and summation of CCN per class*), CK metrics (*DIT*, *LCOM*, *Ca*, *RFC*, *Ce*), code size (*NCSS*, *NOM and number of classes*) and comment size (*number of comment lines*), coding convention violations and code smells, duplicated code (*number of duplicated lines*) and dependencies (*number of dependency cycles* and *Distance from the main sequence (D)*). In terms of analysis and presentation of data, it shows 3 out of 6 factors of SQUALE Quality Model: maintainability, evolutivity and reuse capacity, discarding analysis, functionality, architecture and reliability. Table 19 represents the characterization scheme for Squale.

4.17. Other related tools

There are several tools in Table 1 that have not been discussed in previous sections for the following reasons:

- JCCD [31] is an API to build detectors of duplicated code, but it is not a tool for detecting and measuring the duplicated code itself.
- JCosmo [20] is a tool for displaying source code models as a graph with information about *code smells*, which helps in refactoring code regions. However, it is not a tool to obtain code smells metrics.
- Moose [35] is a language-independent environment which represents software models in a meta-model called FAMIX. It supports tools to visualize, analyze and manipulate the source code for complex software

Table 12
Characterization scheme for Xradar.

Attributes	Dominion
Internal Quality models supported	None
Metrics implemented	Complexity, CK, code size, comment size, coding convention violations, code smells, duplicated code, dependencies
Functional features covered	Data acquisition, analysis of the measures, data presentation
Year of first version	2007
Year of last version	2009

Table 14
Characterization scheme for Sonar.

Attributes	Dominion
Internal Quality models supported	ISO 9126
Metrics implemented	Complexity, CK, code size, comment size, coding convention violations, code smells, duplicated code, dependencies
Functional features covered	Data acquisition, analysis of the measures, data presentation
Year of first version	2007
Year of last version	2011

Table 15
Characterization scheme for Classycle.

Attributes	Dominion
Internal Quality models supported	None
Metrics implemented	Dependencies
Functional features covered	Data acquisition, analysis of the measures, data presentation
Year of first version	2003
Year of last version	2011

systems. CodeCrawler (which later was renamed Mondrian) is one of the visualization tools. Nevertheless, this platform does not provide the user with the code metrics itself therefore, it is necessary to develop tools to display the calculated metrics.

5. Analysis

This section compares the analyzed tools based on the data obtained in Section 4, after applying the characterization scheme in Section 3.1. This comparative analysis is divided into each attribute of the scheme to be easily represented:

Metrics implemented. As Table 20 shows, *code smells* metrics are the most covered by the tools (7 tools), closely followed by *complexity* and *code size* metrics (6 tools). Most tools only implement a small set of metrics (since they are highly specialized), except for Sonar and Squale tools that cover all categories and become the most complete tools in relation to this feature.

Functional features covered. With regard to functionality, as observed in Table 21, *data acquisition* is the main feature of the tools. Most of them only cover one feature (again for being highly specialized), except for *Xradar*, *Sonar and Squale*, which are comprehensive tools that cover all functionalities. This means that, except for *Xradar*, *Sonar and Squale*, tools are incomplete and have to be complemented so as to cover the necessary features to interpret the measurement values.

Year of first version and year of last version. Regarding the year of the versions, Table 22 shows that there is a group of 6 tools (*Checkstyle*, *Sonar*, *Classycle*, *Jlint*, *Sonar Plugins*, *Squale*) that has an active development (2011) and a group of 4 tools (*FindBugs*, *JavaNCSS*, *PMD/CPD*, *Xradar*) with fairly recent development (2009). The 6 remaining tools were developed 4 years ago, which may indicate that they have been discarded in terms of evolution and maintenance. In addition, it can be realized that this group of 10 newly developed tools are mature enough (development period between 6 and 12 years), with the exception of *Xradar*, *Sonar*, *Sonar Plugins* and *Squale*. This fact can be justified by considering that they are heavily based on tools with more mature development (*Checkstyle*, *FindBugs*, *JavaNCSS*, *PMD/CPD*), which are used in the data acquisition phase, and they only evolve; thanks to their maturity level and gradual emergence.

Internal Quality models supported. In relation to Quality models, Table 23 shows the little coverage given by the analyzed tools to Internal Quality models, as evidence has only been found in *Sonar* (which refers to ISO 9126), *Sonar Plugins* (which refers to SIG) and *Squale* (which refers to SQUALE).

Table 16
Characterization scheme for Jlint.

Attributes	Dominion
Internal Quality models supported	None
Metrics implemented	Code smells
Functional features covered	Data acquisition, analysis of the measures, data presentation
Year of first version	2004
Year of last version	2011

Table 17
Description of TQ and Xradar metrics.

Metric	Description
TI	Tangle index
NOM	Number of Methods
LCOM	Lack of cohesion of methods
RFC	Response for class
CBO	Coupling between objects
DIT	Depth of Inheritance Tree
DOC	Documentation
RULES	Rules Compliance Index
DRY	DRYness (DRY: don't repeat yourself)
COV	Code coverage
SUC	Unit test success density
MOD	Modularization
COH	Cohesion
FRE	Freshness
STY	Stylishness
TSC	Statement test coverage
TMR	Method test reference

5.1. Implementation of models by tools

An analysis on how Internal Quality models are implemented by tools and therefore, the possible relationship between the acquired metrics and the implemented Internal Quality models is carried out.

5.1.1. Implementation of ISO 9126 model by Sonar

Before analyzing the implementation in Sonar, a review on how ISO 9126 handles internal metrics must be performed. As mentioned in [53], ISO 9126 suggests metrics that are not based on direct observation of the software product and it does not provide a guide to weight or collect metrics so as to reflect their importance in quality factors. Therefore, a methodology is needed for evaluating the static behavior as proposed in [53], which selects the quality characteristics of *functionality*, *efficiency*, *maintainability* and *portability*, and their respective sub-characteristics, rejecting *reliability* and *usability* because they are related to the dynamic behavior.

Then the implementation of the model by Sonar is analyzed. Sonar calculates a new metric called *RCI* (*Rules Compliance Index*) from coding convention violations and code smells metrics. It indicates the ratio of *Weighted Violations* and the number of *Lines of Code*. To understand this metric, it must be kept in mind that during data acquisition, Sonar collects information from 3 rule engines: Checkstyle, PMD and FindBugs. Then Sonar, based on the configuration of each rule (in a quality profile), aggregates the number of times that the rules have been violated in the project. It then calculates the RCI by the following formula:

$$RCI = 100 - \frac{\text{weighted\_violations}}{\text{ncloc}} * 100$$

ncloc: number of Lines of Code.

Table 18
Characterization scheme for Sonar Plugins.

Attributes	Dominion
Internal Quality models supported	SIG
Metrics implemented	None
Functional features covered	Analysis of the measures, data presentation
Year of first version	2010
Year of last version	2011

Table 19
Characterization scheme for Squale.

Attributes	Dominion
Internal Quality models supported	SQUALE
Metrics implemented	Complexity, CK, code size, comment size, coding convention violations, code smells, duplicated code, dependencies
Functional features covered	Data acquisition, analysis of the measures, data presentation
Year of first version	2008
Year of last version	2011

weighted_violations: Weighted Violations, which is calculated as

$$\text{weighted\_violations} = \sum \text{violations of priority } i * \text{weight of priority } i.$$

Being priority i, *blocker*, *critical*, *major*, *minor*, and *info*. The priority of each rule is specified in the quality profile. The weight of each priority is a configuration parameter of Sonar, with default values INFO = 0, MINOR = 1, MAJOR = 3, CRITICAL = 5, and BLOCKER = 10.

Sonar shows a radar-like graph that indicates the extent to which 5 out of the 6 categories or characteristics of ISO 9126 [13] are respected: Efficiency, Maintainability, Portability, Reliability and Usability (Functionality is not shown). It uses the same formula as RCI to calculate the degree of compliance of each category, but filtering violations by such category. For example, in case of Efficiency:

$$\text{Efficiency} = 100 - \frac{\text{weighted\_violations\_efficiency}}{\text{ncloc}} * 100$$

where:

$$\text{weighted\_violations\_efficiency} = \sum \text{violations of priority } i \text{ and efficiency category} * \text{weight of priority } i.$$

The category of each rule is defined in the quality profile.

Sonar does not completely support ISO 9126 due to the following reasons:

- If it is analyzed from the point of view of the total Internal Quality model, it covers 5 out of the 6 Internal Quality characteristics, discarding functionality. If it is analyzed from the point of view of the assessment methodology static behavior [53], it covers 3 out of the 4 features, discarding functionality again.
- It considers neither the sub-characteristics of ISO 9126 nor those of the methodology.
- It does not use all the metrics it can gather to calculate the characteristics; it only uses the number of coding convention violations and the number of code smells. Therefore, it considers neither the metrics nor the weighting proposed by the methodology [53].

Table 21
Functional features covered by tools.

	Data acquisition	Analysis of measures	Data presentation	Total
Jdepend	X			1
JCSC	X			1
QALab			X	1
CKJM	X			1
Panopticode	X		X	2
Same	X			1
FindBugs	X			1
JavaNCSS	X			1
PMD/CPD	X			1
Xradar	X	X	X	3
Checkstyle	X			1
Sonar	X	X	X	3
Classycle	X			1
Jlint	X			1
Sonar Plugins		X	X	2
Squale	X	X	X	3
Total	14	4	6	

5.1.2. Implementation of SIG model by Sonar Plugins

SIG Maintainability Model plugin is a complete implementation of the SIG model [43] since it covers all the sub-characteristics of maintainability (analyzability, changeability, stability and testability). Firstly, it calculates the metrics of Table 24 to obtain the code properties of the model. It also uses unit test coverage, although this metric refers to the External Quality and it is not included in the present study.

Secondly, after obtaining the numerical values of the properties, they are evaluated according to the scale ++/+/0/−/−, using the same evaluation schemes of the model. Finally, the same mapping between sub-characteristics and properties is used and the result is obtained by applying an average of the evaluations involved in a sub-characteristic.

5.1.3. Implementation of SQUALE model by Squale

SQUALE model proposes 4 levels: factors, criteria, practices and metrics. Practices are evaluated by means of automatic and manual code metrics (*metric analysis*), UML model metrics (*model analysis*), rule checking metrics (*rule checking analysis*), dynamic analysis metrics (*dynamic analysis*) and human analysis metrics (*human analysis*), where applicable. The present study only works with metrics that can be obtained automatically from static code analysis, and therefore it only considers practices, criteria and factors that may arise from them. According to this argument, practices of *metric analysis* (13 practices) and *rule checking analysis* (8 practices) are only selected in the practice

Table 20
Metrics implemented by tools.

	Complexity	CK	Code size	Comment size	Coding convention violations	Code smells	Duplicated code	Dependencies	Total
Jdepend								X	1
JCSC	X		X		X	X			4
QALab									0
CKJM		X							1
Panopticode	X		X	X				X	4
Same							X		1
FindBugs						X			1
JavaNCSS	X		X	X					3
PMD/CPD						X	X		2
Xradar	X	X	X	X					4
Checkstyle					X	X			2
Sonar	X	X	X	X	X	X	X	X	8
Classycle								X	1
Jlint						X			1
Sonar Plugins									0
Squale	X	X	X	X	X	X	X	X	8
TOTAL	6	4	6	5	4	7	4	5	

Table 22
Year of versions of tools.

	Year of first version	Year of last version	Years
Jdepend	2001	2005	4
JCSC	2002	2005	3
QALab	2005	2006	1
CKJM	2005	2007	2
Panopticode	2007	2007	0
Same	2007	2007	0
FindBugs	2003	2009	6
JavaNCSS	1997	2009	12
PMD/CPD	2002	2009	7
Xradar	2007	2009	2
Checkstyle	2001	2011	10
Sonar	2007	2011	4
Classycle	2003	2011	8
Jlint	2004	2011	7
Sonar Plugins	2010	2011	1
Squale	2008	2011	3

level. Table 25 shows that the coverage of these practices in Squale is 76%. The metrics and formulas that the model proposes to evaluate these practices are the same as those used by the tool.

It has been checked that the tool presents a comprehensive coverage of factors and criteria that can be derived from the practices listed above, as shown in the tree of factors and criteria below:

1. Maintainability
   1.1. Comprehension
   1.2. Homogeneity
   1.3. Integration capacity
   1.4. Simplicity
2. Evolutionarity
   2.1. Comprehension
   2.2. Homogeneity
   2.3. Modularity
3. Reuse capacity
   3.1. Comprehension
   3.2. Exploitability
   3.3. Integration capacity.

A weighted average of the criteria is performed, by following the model, in order to assess each factor. Similarly, a weighted average of practices is carried out for the evaluation of each criterion, thus completing the model.

Table 23
Internal Quality models supported by tools.

	Quality models
Jdepend	
JCSC	
QALab	
CKJM	
Panopticode	
Same	
FindBugs	
JavaNCSS	
PMD/CPD	
Xradar	
Checkstyle	
Sonar	ISO 9126
Classycle	
Jlint	
Sonar Plugins	SIG
Squale	SQUALE

Table 24
Relationship between metrics and properties of SIG.

Metric	Property
Total LOC	Volume
CCN per method	Complexity per unit
Density of duplicated code	Duplications
LOC per method	Unit size

6. Conclusions and future work

This paper presents an overview of open source software tools that automate the collection of Internal Quality metrics of Software Products from the point of view of static analysis of Java source code. It begins by describing the procedure defined by SEG for guiding a study of this type [8]. It applies the terminology proposed by Brereton et al. [12] to specify the strategy used in this study.

Sixteen tools are analyzed along the corresponding sections. Most of them automate the calculation of Internal Quality metrics (data acquisition), being code smells, complexity and code size the most common ones. Sonar and Squale are capable of gathering data for all categories of metrics, while the other tools are more specialized in a limited set of metrics. There are 3 complete tools (Xradar, Sonar and Squale), which perform the data acquisition, analysis and presentation. These tools are relatively new and are based on more mature tools for metric acquisition.

As a further conclusion, it would be stated that Sonar (including its plugins) and Squale give support to a greater or lesser extent to ISO 9126, SIG and SQUALE Quality models, establishing a relationship between these models and the metrics they collect. Therefore, and to conclude, it may be pointed out that although there are many tools that automate the calculation of Internal Quality metrics of Software Product, very few have shown evidences of the relationship between metrics and Quality models.

We consider that this paper offers a global review of the situation, focusing on the features (metrics) supported by each tool, without providing an empirical validation and comparison of the features exposed by each tool to assess their real quality. It constitutes the base to detect the most suitable tools, planning further studies of them and analyze them in detail in order to discover computing bugs or errors in the metrics provided. This paper has offered us the possibility of offering a first view and it is the begging for future detailed analysis. Thus, as a result of

Table 25
Coverage of metric analysis and rule checking analysis by Squale.

Practice	Coverage
Inheritance depth	X
Source comments rate	
Number of Methods	X
Method size	X
Swiss army knife	X
Class cohesion	X
Efferent Coupling	X
Afferent Coupling	X
Spaghetti code	X
Copy paste	X
Stability and abstractness level	X
Class specialization	
Dependency cycle	X
Layer respect	
Documentation standard	X
Formatting standard	X
Naming standard	X
Tracing standard	X
Security standard	
Portability standard	
Programming standard	X
Total	16

the study, several lines of research for future work have been identified in this paper:

Static analysis of other technologies
This study would focus on static analysis of Java source code, but it is applicable to other general purpose programming languages such as C, C ++ or the programming languages of .NET platform (C# and Visual Basic.NET), some database programming languages, such as PL/SQL, or even more specific contexts, such as web application development, a category where the quality of code developed for frameworks like Struts, JSF or ASP.NET should be studied.

Static analysis of other products
This line of research would study the automated measurement of Internal Quality in early stages, prior to code generation, such as analysis and design documentation. A case study would be the static analysis of UML diagrams that are generated in these phases.

Supporting tools for Internal Quality models
One of the shortages identified in the analyzed open source tools deals with the little coverage given by Internal Quality models, so this line of research would try to find tools in other areas such as that of commercial tools or propose the design of tools that fulfill this need, if any tools providing sufficient coverage to these models are found.

Dynamic analysis, External Quality and quality in use
The dynamic analysis is performed at later stages than the static analysis, since it requires running the software allowing measuring either External Quality, if performed in a test environment, or quality in use, if carried out in a production environment. For example, this research would analyze tools that measure characteristics associated with unit testing or resource consumption.

Finally, another important idea resulted from this paper. We were considering the possibility of changing our characterization scheme in order to align it with the standard ISO 25000. Despite that this idea is not particularly oriented towards answering the research questions, it could be very interesting to define how each approach focuses on the standard and offers a suitable mechanism to evaluate the relevance of each tool in this line.

Acknowledgments

This research has been supported by the project Tempros project (TIN2010-20057-C03-02) of the Ministerio de Ciencia e Innovación, Spain and the NDTQ-Framework project of the Junta de Andalucía, Spain (TIC-5789).

References

[1] Moisés Rodríguez, Marcela Genero, Javier Garzás, Mario Piattini, KEMIS: Entorno para la medición de la Calidad del Producto Software, 2007.
[2] *ISO/IEC 25000 portal*, Last Accessed September 2012.
[3] International Organization for Standardization, ISO/IEC 14598-1:1999 Information Technology — Software Product Evaluation — Part 1: General Overview, 1999.
[4] C.K.S.R. Chidamber, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (6) (June 1994) 476–493.
[5] International Organization for Standardization, ISO/IEC 25000:2005 Software Engineering — Software Product Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE, 2005.
[6] [En línea]. Available: http://www.iso25000.com/.
[7] *TIOBE Index*, Last Accessed September 2012.
[8] SEG (Software Enginnering Group), Guidelines for Performing Systematic Literature Reviews in Software Engineering Version 2.3, 2007.
[9] N. Rutar, C.B. Almazan, J.S. Foster, A Comparison of Bug Finding Tools for Java, 2004.
[10] I. Lamas Codesido, Comparación de analizadores estáticos para código java, 2011.
[11] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, YuQian Zhou, Evaluating Static Analysis Defect Warnings on Production Software, 2007.
[12] P. Brereton, B.A. Kitchenham, D. Budgen, M. Turner, Khalil, Lessons From Applying the Systematic Literature Review Process Within the Software Engineering Domain, 2007.
[13] International Organization for Standardization, ISO/IEC 9126-1:2001 Software Engineering — Product Quality — Part 1: Quality Model, 2001.
[14] J.F. Smart, Java Power Tools, 2009.
[15] Ohloh.net portal, Last Accessed September 2012.
[16] E.Van Emden, L.Moonen, Java Quality Assurance by Detecting Code Smells, 2002.
[17] J.Robbins, Adopting Open Source Software Engineering (OSSE) Practices by Adopting OSSE Tools, , 2002.
[18] D. Spinellis, G. Gousios, V. Karakoidas, P. Louridas, P.J. Adams, I. Samoladas, I. Stamelos, Evaluating the Quality of Open Source Software, 2009.
[19] S.Wagner, J.Jürjens, C.Koller, P.Trischberger, Comparing Bug Finding Tools With Reviews and Tests, 2005.
[20] JCosmo, Last Accessed September 2012.
[21] *Jdepend*, Last Accessed September 2012.
[22] *JCSC*, Last Accessed September 2012.
[23] *QALab*, Last Accessed September 2012.
[24] *CKJM*, Last Accessed September 2012.
[25] *Panopticode*, Last Accessed September 2012.
[26] *Same*, Last Accessed September 2012.
[27] *Findbugs*, Last Accessed September 2012.
[28] *JavaNCSS*, Last Accessed September 2012.
[29] *PMD*, Last Accessed September 2012.
[30] *Xradar*, Last Accessed September 2012.
[31] *JCCD*, Last Accessed September 2012.
[32] *Checkstyle*, Last Accessed September 2012.
[33] *Sonar*, Last Accessed September 2012.
[34] *Classycle*, Last Accessed September 2012.
[35] *Moose*, Last Accessed September 2012.
[36] *Jlint*, Last Accessed September 2012.
[37] *Sonar Plugins*, Last Accessed September 2012.
[38] *Squale*, Last Accessed September 2012.
[39] Jim A.McCall, Paul K.Richards, Gene F.Walters, Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality, 1977.
[40] B.W.Boehm, J.R.Brown, H.Kaspar, M.Lipow, G.McLeod, M.Merritt, Characteristics of Software Quality, 1978.
[41] I. O. f. Standardization, ISO/IEC 25010:2011 Systems And Software Engineering — Systems And Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models, 2011.
[42] F.Balmas, F.Bellingard, S.Denier, S.Ducasse, B.Franchet, J.Laval, K.Mordal-Manet, P.Vaillergues, The Squale Quality Model, 2010.
[43] IljaHeitlager, TobiasKuipers, JoostVisser, A Practical Model for Measuring Maintainability, 2007.
[44] T.McCabe, A Complexity Measure, 1976.
[45] S.R.Chidamber, C.F. Kemerer, A Metrics Suite for Object Oriented Design, 1994.
[46] M.-A. Sicilia, *Métricas de Mantenibilidad Orientadas al Producto*.
[47] *Sun Code Conventions*, Last Accessed September 2012.
[48] Martin Fowler, Kent Beck, John Bran, William Opdyke, Don Roberts, Refactoring: Improving the Design of Existing Code, 1999.
[49] R.C.Martin, Agile Software Development, Principles, Patterns, and Practices, 2002.
[50] Alan E. Giles, Gregory T. Daich, Metrics Tools. Crosstalk, 1995.
[51] B.Shneiderman, Discovering Business Intelligence Using Treemap Visualizations, 2006.
[52] *Sonar Maven Plugin*, Last Accessed September 2012.
[53] Yiannis Kanellopoulos, Panos Antonellis, Dimitris Antoniou, Christos Makris, Evangelos Theodoridis, Christos Tjortjis, Nikos Tsirakis, Code Quality Evaluation Methodology Using the ISO/IEC 9126 Standard, 2010.
[54] Apache Software Foundation. The Apache Ant Project. Available http://ant.apache.org/.