Towards a Software Quality Assessment Model Based on Open-Source Statical Code Analyzers

Ciprian-Bogdan Chirila*, Dana Juratoni*, Dacian Tudor*, Vladimir Crețu*
*University Politehnica of Timisoara, Romania
Faculty of Automation and Computer Science
Email: {chirila,dacian,veretu}@cs.upt.ro, dana.juratoni@gmail.com

Abstract—In the context of software engineering, quality assessment is not straightforward. Generally, quality assessment is important since it can cut costs in the product life-cycle. A software quality assessment model based on open-source analyzers and quality factors facilitates quality measurement. Our quality assessment model is based on three principles: mapping rules to quality factors, computing scores based on the percentage of rule violating entities (classes, methods, instructions), assessing quality as a weighted mean of the rule attached scores.

I. INTRODUCTION

Generally, engineering is looking for quality. Software engineering in not an exception in this sense [10]. Quality can be considered to be a measure of how a software system conforms to a set of specific factors and criteria. Gerald Weinberg considers quality as a subjective value depending on the user [15]. Software quality was standardized by the ISO/IEC organizations and software attributes are categorized into six main factors. Quality is important since it can lead to significant savings in the costs of software life-cycle [2].

Quality assessment in software engineering is not straightforward. Most project managers use in the development of their projects statical code analyzers to automatically detect code issues and to store them as reports. According to the size of the project the generated reports are often large and difficult to summarize and to present especially in high management meetings. When two or more statical analyzers are used the problem is even more severe, since involves manually compiling the results of all generated reports.

In this paper we will present a quality assessment model based on quality factors inspired from ISO/IEC 9126 standard [12] and the open-source statical analyzer rules: PMD [5], FindBugs [13], Metrics [9]. In figure 1 we present the main idea of our approach. We start from the source code of a Java [6] software project of which quality we want to assess. On the source code we run the statical analyzers producing a list of issues saved as reports. The information from the reports is used by the assessment model in order to quantify quality as a real number on a scale from 0.0 to 10.0. Thus, using our approach the quality of a software project can be measured using numbers, a perfect project with no penalties will get a score of 10.0 out of 10.0.

The paper is structured as follows. In section II we present the selected quality factors used in our approach and the motivation behind the selection. In section III we describe briefly some of the most popular Java open-source statical analysis tools. Section IV presents our quality model design and the quality assessment methodology. Section V tackles the mapping rationale between analyzers rules and quality factors and lists the metrics needed in our proposed quality assessment model. In section VI we present the XML representation structure of our quality assessment model. Section VII measures the number of rules covering each quality factor from our model. Section VIII presents quality related works and other assessment models in the context of software engineering. Section IX concludes and sets the future work.

II. SOFTWARE QUALITY FACTORS

Software product quality is defined in the ISO/IEC 9126 standard [12] and consists in four parts: (1) the quality model, (2) external metrics, (3) internal metrics and (4) quality in use metrics. In our approach we consider that source code is the most important artifact in the software development process, thus we focus on the internal quality. We selected 11 quality factors, part of them from ISO/IEC 9126 quality standard, which we consider the most relevant to describe the source code attributes. The selected factors are: understandability, completeness, conciseness, portability, consistency, maintainability, testability, usability, reliability, efficiency. Understandability refers to aspects related to design and user documentation which must be clearly written and easily understandable. Variable names must be self descriptive according to the functional property they represent. The completeness factor refers to aspects dealing with: the presence of all constituent parts of the software, the availability of all necessary components and resources, error handling on all potential pathways. Conciseness quality factor deals with the minimization of: redundant information - minimizing memory use, size of the code - minimizing processing bandwidth, reachability of all code instructions, code redundancy. The portability factor is related to the software capability of running on multiple: hardware configurations, operating systems,
rules. For example, the rules EmptyCatchBlock and JumbledIncrementer from the basic rule set will map the first to reliability factor while the letter to the understandability factor.

*FindBugs*\textsuperscript{TM}[13] is a static code analysis tool which aims to detect software defects and code idioms which are possible errors, due to complex language features, misunderstood API methods, invalid invariants when code is reviewed, different mistakes like typing errors or wrong operator usage. *FindBugs* is executed on the byte-code and does not require the availability of the source code in order to conduct the analysis. There are numerous successful reports on *FindBugs* on both research and commercial projects [1].

The *Metrics* [9] source code analyzer computes a set of metrics on the analyzed project like: NSM - number of static methods, TLOC - total lines of code, VG - McCabe cyclomatic complexity [8], PAR - number of parameters etc. The values computed by the metrics must be in a predefined interval, otherwise they are reported as issues for the analyzed project. These metrics could be used better as base metrics in our model.

### IV. The Design of the Quality Assessment Model

Our quality assessment model is based on three principles: i) mapping validation rules to quality factors; ii) computing a score for each rule relying on the number of the rule violating entities and on the number of all entities of the same kind from the project; iii) computing quality as a weighted mean of factor scores.

Firstly, taking into account the rule nature we can associate a quality factor on which the rule has the greater impact. For example the *EmptyCatchBlock* rule listed in figure 2 is mapped to the reliability quality factor because the rule detects untreated exceptions. Some analyzers have rule sets which can be mapped directly to a quality factor implying that all the rules from the set are mapped to that factor, but this is not possible in general.

Secondly, the penalty induced by a rule is proportional with the number of erroneous entities over the number of all entities in the project. Here, by entity we refer to: packages, classes, methods, fields, *while* instructions, for instructions etc. For example, the *ExcessiveMethodLength* rule will be mapped to the conciseness quality factor and the score will be computed using the following formula:

$$10 \times (1 - \frac{\text{NoOfMethodsWithExcessiveLength}}{\text{NoOfMethods}}), \quad (1)$$

where: \text{NoOfMethodsWithExcessiveLength} is the number of methods with excessive length, \text{NoOfMethods} is the number of all methods from all project classes. In a software system with 10,000 methods having 1,000 excessive length methods, the computed penalty fraction will be 1,000/10,000 = 0.1 and the score according to the formula will be 10 \times (1 - 0.1) = 9.00. Actually, in assessing quality through the criteria of excessive length methods we need two metrics: i) \text{NoOfMethodsWithExcessiveLength} which is
equal to the number of rule violations, which generally is detected by the source code analyzers, in this case is PMD, and must be counted by a third party tool which we intend to build as future work; ii) \( \text{NoOfMethods} \) is a general base metric whose values are computed by statical code analyzers like Metrics.

Thirdly, the quality model has a tree structure where the root is the quality node, its children are the factors and the terminal nodes are the rules. Node scores are computed in a bottom-up manner using a weighted mean of its sub-nodes. The weight factors are freely configurable but we consider that they should be proportional with the number of their children.

Our model is capable of integrating any other new source code analyzer which detects issues because it is always possible to map particular rules denoting programming or design principles to the selected general quality factors.

The model is flexible to specific projects and companies needs because of the configurable weight factors. They can be tuned according to the organizations internal policies.

If the set of default quality factors is not satisfactory then new factors can be easily added or removed from the model.

The model is represented using the XML technology, which is human readable and accessible to programmers for further processing.

Our approach does not take into account information like: code authors and the source code repository where several old versions of the system can be found. We do not intend to measure the code quality for each programmer separately in order to create work conflicts in organizations. Of course that our quality assessment methodology can be applied on in order to create work conflicts in organizations. Of course that our quality assessment methodology can be applied on

The basic JSF rule (single in the set) affects the execution of the web application, so we took the decision of mapping it to the reliability factor.

The majority of the basic JSP rules affect the conciseness quality factor. The rules detect the presence of long scripts, scriptlets, inline style information which are forbidden. Some of them affect reliability: it is not allowed to do a forward from within a JSP file. Another rule affects efficiency: the presence of HTML comments increases the traffic between the server and the client.

The basic rules verify that the analyzed code follows good practices. Most of the rules affect firstly reliability and secondly conciseness. Empty instruction blocks from statements like: if, while, switch may be a sign of missing implementation. On the other hand if they are not really needed and can be deleted without affecting the code functionality then they compromise the conciseness of the software product. The penalty fraction is computed by dividing the number of empty blocked ifs, whiles, switches over the number of all statements of the same type.

Braces rules were mapped to the understandability factor. We consider that this is a natural decision since misplaced braces affect the readability of the code, decreasing the understandability of the project. The rules from the set detect statements like: if, else, for, while which are used without curly braces. We propose to compute a score for each rule in the set using the model from formula 2.

\[
10 \times \left(1 - \frac{\text{NoOfIfBracesIssues}}{\text{NoOfIfs}} \right), \quad (2)
\]

where \( \text{NoOfIfBracesIssues} \) is the number of if brace issues and \( \text{NoOfIfs} \) is the total number of if statements. For example, if a project has 100 if statements affected by brace issues and 1000 if statements totally, then the score computed with the proposed formula for the brace rule set will be 9.00.

Clone implementation rules check several constraints regarding proper use of cloning. All the rules from the set are mapped to the reliability quality factor. A software system with improper clone method implementation is a suspect of incorrect functionality. The penalty fraction can be computed by dividing the number of certain type clone issues over the number of all clone() methods within the project.

Code size rules were mapped to conciseness quality factor. They locate program entities like classes, methods, parameter lists which have excessive sizes. The motivation for mapping this rule set to the conciseness factor is related to the fact that the smaller the program the higher the conciseness. For this rule set the formulas like 3, 1 and 4 will be used. These formulas fully respect our scoring principle.

\[
10 \times \left(1 - \frac{\text{NoOfClassesWithExcessiveLength}}{\text{NoOfClasses}} \right), \quad (3)
\]
The computed penalty for the rule measuring classes with excessive length is proportional with the ratio between number of classes violating the rule and the total number of classes in the project. We apply the same principle to methods and parameter lists. It is worth mentioning that the rule thresholds are set in the PMD configuration file and usually they conform to the organization internal conventions and policies. We consider this formula as reasonable and fair for measuring quality in this way.

Controversial rules surprise several aspects of different kinds affecting factors like: reliability, conciseness, understandability, efficiency.

Coupling rules detects high coupling objects or packages. Objects with a high number of members denotes a high coupling degree. The penalty fraction can be computed by dividing the number of classes with high number of members over the total number of classes.

Design rules are mapped to different factors. Rules detecting simplifiable boolean returns are mapped to the efficiency factor. The rule mining for switch statements missing their default section is mapped to the completeness factor. The rule which detects deeply nested conditional statements is mapped to the understandability factor.

Finalizer rules were mapped to the completeness quality factor. Finalizer rules detect different issues related to finalizers like: empty finalizers, finalizer calling only super and having no other statements, finalizer methods having parameters etc. The proposed metric for these rules is listed in formula 5.

\[ 10 \times (1 - \frac{\text{NoOfFinalizerIssues}}{\text{NoOfFinalizers}}) \]  

Import statement rules are just five and they were mapped to the conciseness quality factor. The base metric for these rules is the total number of all import statements from the project. J2EE rules and naming rules are mapped to different quality factors mostly to consistency and reliability. Rules dealing with naming conventions were mapped to the consistency quality factor. Rules dealing with calling System.exit() and using threads are mapped to the reliability quality factor.

Java Bean rule set contains only two rules which were mapped to reliability factor. One is dealing with detecting non-serializable bean members and the other with detecting bean classes with no serial version identifier field.

JUnit rules, Jakarta commons logging rules, Java logging rules are all mapped to the testability factor. Migration rules, Migration13, Migration14, Migration15 and MigratingToJava4 are all mapped to the portability quality factor for obvious reasons. The basic metrics for these rules are usually related to the number of method calls.

Naming rules were generally mapped to consistency quality factor. For these rules a natural base metric is the count of identifiers from the project.

Optimization rules were generally mapped to efficiency quality factor.

Strict exception rules are rules which detect bad practices in the context of Java exceptions. This is why we mapped them to the reliability and testability quality factors.

String and string buffer rules were generally mapped to efficiency quality factor. They refer to entities like duplicated literals, string instantiations, concatenations, comparisons, indexing.

Security code guidelines rules are just two and they were mapped obviously to the security quality factor. These rules refer to entities like internal arrays, methods which return internal arrays.

Unused code rules detect entities like: unused private fields, unused local variables, unused private methods and unused formal parameters. The base metrics for these rules are: total number of private fields, total number of local variables, total number of private methods, total number of formal parameters.

B. Language Specific Metrics

In order to implement our model we analyzed all the rules and we conclude that the following metrics are needed for the quality assessment: NoOfCalls - number of calls; NoOfCasts - number of casts; NoOfCatches - number of catch instructions; NoOfClasses - number of classes; NoOfComparisons - number of comparisons; NoOfComparisons - total number of checks; NoOfConstants - number of constants; NoOfConversions - number of conversions to String; NoOfEqualCalls - number of equals method calls; NoOfFields - number of fields; NoOfFinalizer - number of finalizer() methods; NoOfFinalities - number of finally instructions; NoOfFors - number of for instructions; NoOfFrames - number of iframes; NoOfIdentifiers - number of identifiers; NoOfIfs - number of if instructions; NoOfInitializers - number of initializers; NoOfInstantiations - number of instantiations; NoOfInstructions - number of instructions; NoOfJSFs - number of JSFs; NoOfJSPs - number of JSPs; NoOfJarFileEntry - number of created jar file entries; NoOfLoops - number of loop (for, while, do while) instructions; NoOfMethodOverride - number of method overrides; NoOfMethods - number of methods; NoOfModifiers - number of modifiers; NoOfPackages - number of packages; NoOfReturns - total number of returns; NoOfScriptAttributes - number of HTML attributes used in scripts; NoOfStaticInitializers - number of static initializers; NoOfStreams - number of streams; NoOfSwitches - number of switch instructions; NoOfSynchronizedBlocks - number of synchronized blocks; NoOfTries - number of try instructions; NoOfWhiles - number of while instructions; NoOfZipFileEntry - number of created zip file entries; ScriptLOC - script lines of code. Some of the previously listed metrics are provided by the Metrics statical analyzer and the remaining ones must be clearly defined and implemented in a software tool.

VI. MODEL REPRESENTATION

Our quality model is represented as a hierarchical set of XML files. On the top of the hierarchy is the "quality factors.xml" file which contains meta information like: model
XML representation of rules to factors mappings. The main element of the XML file is the `<tool>` element. This element has a name property. The first child of this element is a `<description>` element containing a descriptive text. Next, all the `<ruleset>` elements are listed having properties like name, weight factor and computed score. Each rule set contains a list of `<rule>` elements. The rule element is the most complex element having the following attributes. Name denotes the name of the rule. Factor1 denotes the most affected quality factor when the rule is violated. Factor2 denotes the secondly most affected quality factor when the rule is violated. Profile is an attribute which may get the following values: i) "all" - stands for the fact that the current rule is suitable to all kind of Java projects, ii) "embedded" - means that the current rule fits to Java embedded systems; iii) "web" - specifies that the rule is dedicated to Java web projects; iv) "enterprise" - denotes that the current rule is used for Java enterprise applications. The weight factor and computed score attributes are useful in the numerical quality assessment process.

VIII. QUALITY MODEL FACTORS COVERAGE

In this section we will analyze how the quality factors are covered by the analyzer rule sets. We are interested how much the quality factors are covered by each source code analyzers separately and then globally. In table I we present the number of rules from each statical code analyzer mapped to each quality factor and also the coverage percentages. We notice that each tool covers in an acceptable manner the quality factors. The most covered quality factor is reliability. This factor is covered by almost 50% of the rules for both considered source code analyzers. The least covered quality factor is usability: by one rule in PMD and by no rule in FindBugs. This is normal since source code analysis can not tell much about the programs usability. Great differences between factor coverages provided by the two source code analyzers resulted for factors like: completeness, security and conciseness.

### TABLE I

<table>
<thead>
<tr>
<th>Factor/Analyzer</th>
<th>PMD</th>
<th>FB</th>
<th>Total</th>
<th>PMD%</th>
<th>FB%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reliability</td>
<td>60</td>
<td>83</td>
<td>143</td>
<td>41.96</td>
</tr>
<tr>
<td>2</td>
<td>Completeness</td>
<td>4</td>
<td>111</td>
<td>115</td>
<td>3.48</td>
</tr>
<tr>
<td>3</td>
<td>Efficiency</td>
<td>48</td>
<td>42</td>
<td>90</td>
<td>53.13</td>
</tr>
<tr>
<td>4</td>
<td>Consistency</td>
<td>27</td>
<td>39</td>
<td>66</td>
<td>40.91</td>
</tr>
<tr>
<td>5</td>
<td>Understandability</td>
<td>30</td>
<td>17</td>
<td>47</td>
<td>63.83</td>
</tr>
<tr>
<td>6</td>
<td>Conciseness</td>
<td>35</td>
<td>8</td>
<td>43</td>
<td>81.40</td>
</tr>
<tr>
<td>7</td>
<td>Testability</td>
<td>28</td>
<td>13</td>
<td>41</td>
<td>68.29</td>
</tr>
<tr>
<td>8</td>
<td>Security</td>
<td>2</td>
<td>32</td>
<td>34</td>
<td>3.88</td>
</tr>
<tr>
<td>9</td>
<td>Maintainability</td>
<td>8</td>
<td>17</td>
<td>25</td>
<td>32.00</td>
</tr>
<tr>
<td>10</td>
<td>Portability</td>
<td>8</td>
<td>7</td>
<td>15</td>
<td>53.33</td>
</tr>
<tr>
<td>11</td>
<td>Usability</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>100.00</td>
</tr>
<tr>
<td>Total</td>
<td>251</td>
<td>369</td>
<td>620</td>
<td>40.48</td>
<td>59.52</td>
</tr>
</tbody>
</table>

Our quality assessment approach is highly based on software metrics. A software metric is a method which quantifies the number of rules from each statical code analyzer mapped to each quality factor and also the coverage percentages. Quality factors, their descriptions and concrete project information like: project name, weight factors and computed scores. In figure 3 we present the structure of the XML representation of the proposed quality assessment model. The main element of the XML file is `<quality>` which contains a `<description>` child element and a list of child `<factor>` elements. Each `<factor>` element has several attributes like name, weight factor and computed score. Each factor element has its own `<description>` child element embedding a descriptive text. Next, for each analyzer we have an XML file where the mapping between the rules and factors is set: "pmd-rules.xml", "fb-rules.xml". In figure 4 we present a fragment from the XML representation of rules to factors mappings. The main element of the XML file is the `<tool>` element. This element has a name property. The first child of this element is a `<description>` element containing a descriptive text. Next, all the `<ruleset>` elements are listed having properties like name, weight factor and computed score. Each rule set contains a list of `<rule>` elements. The rule element is the most complex element having the following attributes. Name denotes the name of the rule. Factor1 denotes the most affected quality factor when the rule is violated. Factor2 denotes the secondly most affected quality factor when the rule is violated. Profile is an attribute which may get the following values: i) "all" - stands for the fact that the current rule is suitable to all kind of Java projects, ii) "embedded" - means that the current rule fits to Java embedded systems; iii) "web" - specifies that the rule is dedicated to Java web projects; iv) "enterprise" - denotes that the current rule is used for Java enterprise applications. The weight factor and computed score attributes are useful in the numerical quality assessment process.
software product attributes into six characteristics, which are quality model for software products. The model categorizes our approach but they are embedded in the statical analyzers methods (LCOM). These metric suites are not used directly in our approach but they are embedded in the statical analyzers detecting code issues.

The international standard ISO/IEC 9126 [12] defines a quality model for software products. The model categorizes software product attributes into six characteristics, which are further subdivided into 27 sub-characteristics. From this standard we used 5 quality factors and we added another 6 which we consider relevant.

In other approaches like [7], [4] quality is assessed as a mathematical expression of entities count violating good object-oriented design principles quantified by metrics and statistical operators combined using set operators. In our approach quality is computed as a weighted mean of Java specific flaw density. Our analyzed flaws are not limited only to object-oriented design.

The STREW (Software Testing and Reliability Early Warning) model [11] measures the thoroughness of the testing effort for an object-oriented system. They measure the testing intensity ratio. We adhere to this approach but we test the fault intensity ratio and compute a score in a predefined range.

The concept of defect density is present in the work of Sherriff [14]. It is estimated the defect density using persistent records of validation and verification practices which are automatically recorded and maintained with the code. Our approach is similar to this but instead of using validation and verification records we use the list of issues generated by statical code analyzers.

IX. CONCLUSIONS AND FUTURE WORK

We conclude that a quality model can be built relying on static analyzer reports and that the factors are covered in a satisfiable degree. A reasonable mapping between rules and quality factors can always be found. Generally, the predefined rule sets can not be mapped directly to a single quality factor.

In order to compute a weighted mean of violations density we need several metrics. The nominator metrics can automatically be obtained by counting the number of violations for a certain rule. On the other hand for the denominator metrics we must use Java language specific metrics, which must be computed by a tool we intend to implement as future work.

One advantage of using our quality model is that the programmer does not have to configure the statical code analyzers since they are already used in the project for bug detection and correction. The programmer has to specify only one time the mapping rules and then use it for all the assessments.

The weight factors allow organizations to configure factor importance according to their internal rules or according to the nature of the project. Our model allows configuration of the quality model according to the nature of the project: standalone graphical user interface applications, embedded applications, enterprise and web applications.

The main future work is to automate the approach in software tool named ProSTARTER (Project Static Analyzer) capable of loading the quality model, importing the analyzers reports, computing the language specific base metrics, assessing the quality of the analyzed software system and generating a detailed presentation report for the high management.

Another future work is to experiment the tool on small, medium and large Java object-oriented systems and on several versions from their development cycles.

REFERENCES