A Tool for Software Quality

Chidung Lac, Jean-Luc Raffy
Institut National des Télécommunications
9, rue Charles Fourier, 91011 Evry Cedex, France
Tel: 33 - 1 - 60 76 47 08
E-Mail: LAC@FRINT51.BITNET

Abstract:

We present a theoretical study of a tool used to improve the quality of delivered software. Common approaches to software quality focus on the quantitative aspects of the code and neglect the qualitative sides of the utilization though these last characteristics are far from being worthless for the end-user. Starting from the work of Boehm and McCall concerning factors, criteria and metrics, we have constructed a model which serves to put together the user's needs and the programmer's possibilities. The model incorporates a quantitative assessment of both addresses. The algorithms of the tool using this model are also described.

Keywords: software quality, metrics, modelling, measurement tool.

I. Introduction:

What is software quality? Some will define it as a software delivered bug-free, while others will require in addition that the software meet the final user's requirements. From the second point of view, the classical approach is to pay special attention during the specification phase and to check that the final delivered software respects these specifications. This form of dialogue between the end-user and the software team is limited to the technical and quantitative aspects of the needs and ignores (or cannot deal with) the qualitative requirements of daily utilization.

A way to treat of this problem is to collect the qualitative needs of the end-user and to compare, through a quantitative process, with the characteristics of the software being developed.

The work described in this paper is the construction of a model used to link these qualitative needs with the software characteristics obtained through a static analysis. The model, based on the work of Boehm et al. [1] and McCall et al. [2], is extended to include a quantitative assessment of software allowing the development people to compare the results of their product with the user's requirements. Once the differences are located, the development team can then improve the source code to get closer to the user's wishes.

II. Factors, criteria and metrics:

The path from the end-user to the developer is composed of three levels: factors represent the user's needs, each of them is decomposed into criteria which are in turn combinations of metrics.

Factors and criteria are qualitative characteristics of the software, the first being viewed from the user's standpoint and the second treated by the developer, i.e., including constraints unknown from the end-user (for instance a software easy to maintain for the user means for the developer modular programs easy to test). Another way to define factors and criteria is to state that the first is an external view of the software while the second represent its internal view.

Metrics are, for their part, obtained by the use of a static analyzer and constitute the real and 'physical' measurement one can get from the source code (for instance the total number of operators contained in the program).

Three main reasons have guided us in the choice from the huge list of factors and criteria proposed in the literature:
- factors need to express the user's requirements in term of utilization, modification and transfer;
- factors have to be easy to split into measurable criteria;
- finally, the selected terms should be exhaustive enough to be able to cover the broad range of software types, i.e., scientific applications as well as accounting ones.

Figure 1 shows the list of factors linked with criteria; the first five factors are connected to utilization, the two following to modification and the last one to transfer [3]. The terms used will not be defined since they are self-explanatory or can be found in the cited references, except for the following criteria:
- testability: the property of a program being easy to test in order to improve its reliability and maintainability;
- clarity: the property of a program being understandable through its code or its documentation;
- context independence: the property of a program being software system independent and machine independent;
- exploitability: the property of a program to be easy to operate and to use.

III. Quantitative assignments:

III.1 Values assignment:

The eight factors we have selected are now a linear combination of twelve criteria, and the weight used will be the same for each of them, e.g.,

\[
\text{RELIABILITY} = \frac{\text{TOLERANCE} + \text{SIMPLICITY} + \text{MODULARITY} + \text{CONSISTENCY} + \text{TESTABILITY}}{5}. \]

![Fig. 1. Factors and criteria.](image-url)
The criteria have to be transformed into measurable values, with these final metrics of two kinds:
- the previous values are not sufficient, so we have to add some qualitative measures obtained from the end-user, i.e., answers to questions found in pre-defined checklists.

To compare the user's needs (factors), the programmer's possibilities (criteria) and the values extracted from the measures (metrics), we use a discrete quantification procedure by defining three values: 0, 1 and 2 meaning respectively bad (or rejected), medium (or critical) and good (or satisfactory).

For terminal objects such as sub-metrics or programming norms (see below), the thresholds given depend on the object.

For nonterminal ones (factors F, criteria C and metrics M), two values will be used as thresholds: 0.8 and 1.2.

\[ F, C \text{ or } M = \begin{cases} 0 & \text{if } \text{value} \in [0, 0.8] \\ 1 & \text{if } \text{value} \in [0.8, 1.2] \\ 2 & \text{if } \text{value} \in [1.2, 2] \end{cases} \]

To avoid the cases where a factor is given the value 2 when most of criteria composing it are null, we've added the following constraints:
- if the factor has less than four criteria and at least one of them is null, then the factor will be put to 1;
- if the factor has four criteria or more and at least two criteria are null, then the factor is put to 1.

For checklists composed of questions, the answers will be used to get the value of the metric by the following way: each answer can be 0 (no), 0.5 (maybe or partially) or 1 (yes). The corresponding metric \( M' \) will then be:

\[ M' = \frac{\sum (\text{value of the answer})}{\text{total number of questions}} \]

Threshold adjustment will be made in this case as follows:

\[ M' = \begin{cases} 0 & \text{if } M' \in [0, 0.4] \\ 1 & \text{if } M' \in [0.4, 0.8] \\ 2 & \text{if } M' \in [0.8, 2] \end{cases} \]

Seven programming norms were used in this work:

- \( N_1: \text{identifier's length, i.e., number of characters.} \)
  \[ N_1 = \begin{cases} 2 & \text{if } \text{length} \in [6, 16] \\ 0 & \text{else} \end{cases} \]

- \( N_2: \text{global identifier's name; it has to start with three characters recalling the name of the compilation unit (component) exporting it.} \)
  \[ N_2 = \begin{cases} 2 & \text{if the condition is satisfied} \\ 0 & \text{else} \end{cases} \]

- \( N_3: \text{literal constants are only allowed in declarative parts of symbolic constants (excepted for 0, 1 and -1).} \)
  \[ N_3 = \begin{cases} 2 & \text{if the condition is satisfied} \\ 0 & \text{else} \end{cases} \]

- \( N_4: \text{sub-program's cyclomatic number is limited.} \)
  \[ N_4 = \begin{cases} 2 & \text{if } V(G) < 15 \\ 1 & \text{if } V(G) \in [15, 18] \\ 0 & \text{if } V(G) > 18 \end{cases} \]

- \( N_5: \text{the interleaving of control structure in subprograms is limited.} \)
  \[ N_5 = \begin{cases} 2 & \text{if the value} < 5 \\ 1 & \text{if the value} \in [5, 7] \\ 0 & \text{if the value} > 7 \end{cases} \]

- \( N_6: \text{the use of GOTO instructions is restricted.} \)
  \[ N_6 = \begin{cases} 0 & \text{if the number of GOTO} \neq 0 \\ 2 & \text{else} \end{cases} \]

- \( N_7: \text{all multiple conditional branching instructions (CASE, SWITCH) need a default branch.} \)
  \[ N_7 = \begin{cases} 2 & \text{if the condition is satisfied} \\ 0 & \text{else} \end{cases} \]

III.2 Expression of criteria:

- Tolerance: \( \text{TO} = TO_1 + N_7 \) with \( TO_1, \text{the coding strength} \) being defined with the use of checklists*, i.e., questions such as "Are pre-conditions systematically used at the beginning of components?"

- Simplicity:
  \[ S = \frac{1}{4} \left( 2S_1 + \frac{1}{2} (N_6 + N_7) + S_2 \right) \]
  with \( S_1, \text{the logical complexity} \) defined using four sub-metrics:

  \( \text{\* } S_{11}, \text{representing the cyclomatic number, is defined using } \max_1, \text{the maximum of all modules' cyclomatic number.} \)

* in order to limit the length of this paper, we've not given the list of questions of the checklists; we will be pleased to provide these details to any reader who is interested.
\[ S_{11} = \begin{cases} 2 & \text{if } \max_1 < 15 \\ 1 & \text{if } \max_1 \in [15, 18] \\ 0 & \text{if } \max_1 > 18 \end{cases} \]

- **S\(_{12}\)**, representing the **pending nodes**, i.e., terminal nodes minus one, is defined using \( \max_2 \), the maximum of all modules' node number.

\[ S_{12} = \begin{cases} 2 & \text{if } \max_2 = 0 \\ 0 & \text{else} \end{cases} \]

- **S\(_{13}\)**, representing the **interleaving**, is defined using \( \max_3 \), the maximum interleaving of control structure.

\[ S_{13} = \begin{cases} 2 & \text{if } \max_3 < 5 \\ 1 & \text{if } \max_3 \in [5, 7] \\ 0 & \text{if } \max_3 > 7 \end{cases} \]

- **S\(_{14}\)**, representing the **number of degrees**, is defined using \( \max_4 \), the maximum number of edges connected to a node of the graph.

\[ S_{14} = \begin{cases} 2 & \text{if } \max_4 < 8 \\ 1 & \text{if } \max_4 \in [8, 10] \\ 0 & \text{if } \max_4 > 10 \end{cases} \]

then:

\[ S_1 = \begin{cases} 0 & \text{if more than one sub-metric} = 0 \\ 1 & \text{if one sub-metric} = 0 \text{ and more than one} = 1 \\ 2 & \text{else} \end{cases} \]

and **S\(_2\)**, the **textual complexity**, defined using \( D \), the program difficulty, i.e., the inverse of program level:

\[ D = \frac{\eta_1 N_2}{2 \eta_2} \]

where \( \eta_1 \) is the number of distinct operators, \( \eta_2 \) the number of distinct operands and \( \eta_2 \) the total usage of all the operands.

then:

\[ S_2 = \begin{cases} 2 & \text{if } D < 30 \\ 1 & \text{if } D \in [30, 35] \\ 0 & \text{if } D > 35 \end{cases} \]

- **Modularity**:

\[ M = \frac{1}{9} \left( 2 (M_1 + M_2 + M_3) + 3 M_4 \right) \]

with **M\(_1\)**, the **hierarchical complexity** of the call graph defined using \( h \), the mean number of modules per level:

\[ M_1 = \begin{cases} 0 & \text{if } h < 4 \\ 2 & \text{if } h \in [4, 8] \\ 0 & \text{if } h > 8 \end{cases} \]

**M\(_2\)**, the **structural complexity** defined using \( s \), the mean number of calls per module:

\[ M_2 = \begin{cases} 0 & \text{if } s < 1 \\ 2 & \text{if } s \in [1, 2.5] \\ 0 & \text{if } s > 2.5 \end{cases} \]

**M\(_3\)**, the **mean number of paths** per module defined using \( p \), the ratio of the number of different calling paths (between the root and the terminal modules) to the number of terminal modules:

\[ M_3 = \begin{cases} 0 & \text{if } p < 1 \\ 2 & \text{if } p \in [1, 2] \\ 0 & \text{if } p > 2 \end{cases} \]

and **M\(_4\)**, the **code modularity** defined using \( m \), the number of code lines per module:

\[ M_4 = \begin{cases} 0 & \text{if } m < 30 \\ 1 & \text{if } m \in [30, 50] \\ 2 & \text{if } m \in [50, 80] \\ 1 & \text{if } m \in [80, 100] \\ 0 & \text{if } m > 100 \end{cases} \]

- **Consistency**:

\[ CO = \frac{1}{3} (N_1 + N_2 + N_3) \]

- **Testability**:

\[ TE = \frac{1}{4} (M_3 + N_4 + N_5 + N_6) \]

with **M\(_3\)** being defined as above.

- **Clarity**:

\[ C = \frac{1}{9} \left( 5C_1 + \frac{1}{2} (C_2 + C_3) + 2C_4 + \frac{1}{4} (N_1 + N_2) + \frac{1}{2} N_6 \right) \]

with **C\(_1\)**, Halstead's **effort** defined using the mental effort \( e \):

\[ C_1 = \begin{cases} 2 & \text{if } e < 20000 \\ 1 & \text{if } e \in [20000, 25000] \\ 0 & \text{if } e > 25000 \end{cases} \]

**C\(_2\)**, expansion rate of the source code defined using the ratio \( cp \) of code lines to the number of pseudo code lines:
\[ C_2 = \begin{cases} 1 & \text{if } cp < 3 \\ 2 & \text{if } cp \in \{3, 5\} \\ 1 & \text{if } cp \in \{5, 7\} \\ 0 & \text{if } cp > 7 \end{cases} \]

C_3, the comment rate defined using the ratio \( r \) of comments to the number of instructions:

\[ C_3 = \begin{cases} 2 & \text{if } r \in [0.3, 0.5] \\ 1 & \text{if } r \in [0.0, 0.3] \text{ or } r \in [0.5, 1] \\ 0 & \text{if } r > 1 \end{cases} \]

and C_4, the language clarity defined as follow:

\[ C_4 = \begin{cases} 2 & \text{with ADA, LTR3, PASCAL} \\ 1 & \text{with FORTRAN, C, COBOL} \\ 0 & \text{with APL, assembly languages} \end{cases} \]

- Context independence: \( CI = CI_1 \) with \( CI_1 \), independence during the coding phase, defined with the use of checklists.

- Exploitability:
  \[ XPL = \frac{1}{4} (XPL_1 + XPL_2 + XPL_3 + XPL_4) \]

The four metrics, defined with the use of checklists, assess the capability of a software:

* to be easy to employ for different experience-level users (XPL_1);
* to be easy to modify for experienced users (XPL_2);
* to allow backtracking (XPL_3);
* to manage errors committed by users (XPL_4).

- Training:
  \[ T = \frac{1}{5} (2T_1 + T_2 + XPL_3 + XPL_4) \]

with \( T_1 \), the ability to be memorized, \( T_2 \), the ability to propose an on-line help, defined with the use of checklists and XPL_3, XPL_4 being defined as above.

- Access control: \( AC = AC_1 \)

This metric, assessing the capability of a software to be protected against its own execution, is defined with the use of checklists.

- Access audit: \( AA = AA_1 \)

This metric, assessing the capability of a software to be protected against any violation, is also defined with the use of checklists.

- Expandability: \( XPA = N_3 \)

IV. Functioning of the tool:

IV.1 Input data:

Two types of data must be provided to the tool: the first one is composed of user’s needs and the second one is the set of metrics taken from the code with the use of a static analyzer and the answers to the questions of the checklists. The thresholds for the different values of metrics or sub-metrics explained earlier can be adjusted by the user.

To get the end-user’s wishes in term of quality factors, an interface will propose a set of application types, each of it being linked to some default values of the factors. These values can be changed by the user according to specific needs. The application types we have selected are as follow:

- demonstration model (DM);
- real time (RT);
- critical (C);
- management (M);
- scientific computing (SC);
- security oriented (SO);
- entertainment (E);
- basic (B).

Figure 2 shows the default values of the factors set for each application type.

<table>
<thead>
<tr>
<th>Application type</th>
<th>DM</th>
<th>RT</th>
<th>C</th>
<th>M</th>
<th>SC</th>
<th>SO</th>
<th>E</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORRECTNESS</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RELIABILITY</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>USABILITY</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>INTEROPERABILITY</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>INTEGRITY</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MAINTAINABILITY</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>FLEXIBILITY</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>PORTABILITY</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Fig. 2. Default values of factors.
IV.2 Forward and backward propagation:

The analysis proceeds in two steps:

- comparison of factors needed and measured, the first category being defined through the user-interface (see preceding section), and the second one after the measurement of the code with a static analyzer and the answers to the checklists' questions. A backward propagation procedure is used to get the second category of factors: starting from Halstead's and McCabe's measures and checklists' answers, values for sub-metrics and metrics are found. With the use of heuristics defined before, values for criteria and for factors are obtained, allowing then a quantitative comparison.

- the second step, following a forward propagation procedure, is intended to identify the details of the code which need correction in order to get the measured factors close to the user's needs.

To accomplish this second step, some rules have been defined for the identification of the different objects of the model:

- factors: if the measured value of a factor has a value greater than or equal to the one needed, the factor is left uncorrected. Factors to be corrected are sorted according to the values needed in a decreasing order and for each value, according to the value measured in an increasing order.

- criteria: two rules are defined in this case. The first rule states that for each criterion, given the requested value, say F_i, of the factors including it, the ratio of the sum of F_i to the total number of factors is computed. These values are used to sort the criteria in a decreasing order and to put them in a priority list. The second rule states that given a such list, one obtains for each factor a restricted list by suppressing all the criteria whose value is 2 and the criteria which are not part of the concerned factor. Recursivity is then used to determine the criteria to be improved to reach the desired value of the factor.

- metrics: for each criterion, a list of metrics priority is produced using the decreasing value of the weight in the heuristics. Following this list, one determines by recursion one or more metrics to be improved to achieve the requested value of the criterion.

Finally, an array shows the computed values of the sub-metrics and the requested ones for each module. After the improvement at the code level, the whole procedure has to be started again to verify quantitatively the results.

Figure 3, showing the logical scheme of the analysis, summarizes the theoretical functioning of the tool.

V. Future work:

The study described in this paper, although not exhaustive, will serve as the main background for the development of a tool aiming to improve the quality of developed software. Discussions are on the way with vendors of static analyzers to build an interface to their package which will be used to get the metrics needed for the measurement. The thresholds given for the heuristics of the criteria were found analytically and need to be confirmed by experience.

Acknowledgment:

The authors are deeply indebted to E. Despreux and O. Magnon who provided valuable help on an earlier version of this paper.

References:

Input of user's needs

Application type?

Measures from static analyzer

Comparison of factors needed and measured

Heuristics on metrics and sub-metrics → criteria

Heuristics on criteria → factors

Answers to checklists

Backward propagation

Classification of criteria according to user's wishes

Forward propagation

Determination of criteria to modify

Results print out

Determination of metrics to modify

Validation of factors values
Threshold adjustment

Fig. 3. Functioning logical path.