On Issues with Software Quality Models

2 authors, including:

Yann-Gaël Guéhéneuc
Concordia University Montreal
268 PUBLICATIONS **5,592** CITATIONS

SEE PROFILE

On Issues with Software Quality Models

Khashayar Khosravi and Yann-Gaël Guéhéneuc

GEODES - Group of Open and Distributed
Systems, Experimental Software Engineering

Department of Informatics and Operations Research
University of Montreal, Quebec, Canada
{khosravk,guehene}@iro.umontreal.ca

Abstract. Software metrics and quality models play a pivotal role in measurement of software quality. A number of well-known quality models and software metrics are used to build quality software both in industry and in academia. However, during our research on measuring software quality using design patterns, we faced many issues related to existing software metrics and quality models. In this position paper, we discuss some of these issues and present our approach to software quality assessment

1 Introduction

As software becomes more and more pervasive, there has been a growing concern in the academic community and in the public about software quality. This concern arises from the acknowledgment that the main objective of software industries in non-military sectors is to balance price and quality to stay ahead of competitors.

In the past 15 years, the software industry has created many new different markets, such as open source software and commerce over the Internet. With these new markets, customers of programs now have very high expectations on quality and use quality as a major drive in choosing programs.

Some organisations, such as ISO and IEEE, try to standardise software quality by defining models combining and relating software quality characteristics and sub-characteristics. Meanwhile, researchers propose software metrics as tools to measure programs source code, architecture, and performances. However, there is a not yet clear and consensual relation among software quality models and between models and metrics. Moreover the process of software quality assessment remains an open issue with many models.

In this position paper, we state some open issues related to software quality, and extend the classical view of software quality models and software metrics. We propose some possible solutions to the mentioned open issues: Modifying software quality models so that characteristics and sub-characteristics are more meaningful to their users; Considering the dynamic behaviour of the software during its execution; Using design patterns as high level building blocks to asses

the software design. All these solutions can bring significant improvements in the assessment of the quality of programs.

In the first part of this paper, we review software assessment tools, some problems, and solutions. In the second part, we introduce a practical example of software quality assessment based on our modified assessment method.

2 Software Quality

"Walter Shewhart was a statistician at AT&T Bell Labratories in 1920s and is regarded as the funder of statistical quality improvement, and modern process improvement is based on the concept of process control developed by Shewhart"[1].

Since 1920, progress in the field of software quality has been time-consuming and ambiguous. Software metrics and software quality models are known as major reference in the field of software quality assessment but there is still no clear methods for assessing software quality via software metrics.

2.1 Missing Facts on Quality

In a book on software development and reality construction [2], the authors acknowledge the dichotomy—in traditional science—between observer and observation and the view of human needs as being *outside* of the realm of scientific enquiry. They regret this view and state emphasises theirs:

> An important aspect of computer science is that it deals with *creating reality*: The technical reality of the programs executed on the computer, and the condition for the human reality which unfolds around the computer in use. Therefore, the conceptual categories "true" and "false" [that computer science] relies on are not sufficient in themselves. We have to go beyond them by finding categories for expressing *the felicity of our choices*, for distinguishing "more or less suitable" as we proceed in making distinctions and decisions in communicative design processes. This is essential for dealing with quality in software development and use[2].

For example, one observer could record, over a period of a year, the Sun rises. Then, she could state a law generalising these events with the following "Everyday, the Sun rises". Finally, she could develop a theory of the Earth spinning and orbiting around the Sun, thus explaining the laws and the observations.

In other fields of science, such as physics, the discovery of relationships among artifacts follows the scientific method of observations, laws, and theories. [3, 4]. This method consists in recording observations on some facts, then stating laws generalising the observations, finally developing a theory able to explain the laws and thus the observations.

The scientific method begins with facts and observations. We believe that the field of software quality is so far limited by the lack of concrete and consensual

facts on the quality of software. Facts in software engineering, in particular about quality, remain too complex because of the many interactions between software artifacts and of the discreet nature of software.

We must find facts in software engineering pertaining to software quality. We believe that design patterns, in particular their design motifs (solutions), could be such facts. Indeed, design motifs are concrete artifacts in programs and consensual in the quality characteristics they bring.

3 Open Issues

The following is the result of our idea about modification and improvement of existing tools for software quality assessment.

3.1 Human Estimation

A person can look at source code or software products and judge their performance and their quality. Human evaluation is the best source of measurement of software quality because at the it is a person who will deal with quality of software.

The Problem:

- *Different taste, different value:* Often, software evaluation of one person can not be expanded as acceptable evaluation for other people because different people have different view on quality. For example, just listen to other people advising for choosing an operating system or a wordprocessor...
- *Assessing the quality of software by your own is not practical:* It is impossible that everybody have the knowledge and the ability to evaluating the software performance and quality. In addition it is a very hard and time consuming task.

The solution: Birds of a feather flock together: We must categorize the people who deal with software at different level by considering their need for software quality, and then we can create tailored models for each group, or range of values which are acceptable for similar people. For example, end users mostly have similar ideas about quality of software, but these ideas maybe different from those of people who deal with the maintenance of the same software.

3.2 Software Metrics

To our best knowledge, instead of using human estimation, software metrics are the only mechanized tools for assessing the value of internal attributes [5].

Software metrics are defined as "standard of measurement, used to judge the attributes of something being measured, such as quality or complexity, in an objective manner" [6], but subjective measurement of quality comes from human estimation.

The Problem:

- *Evaluation of software code is not enough:* We believe that considering a source code with no regard for its execution is the same as considering the body without its sprit. Well-known metrics are just computing size, filiation, cohesion, coupling, and complexity. These internal attributes are related to code but the quality of a software does not depend on its code only: Acceptable quality in code evaluation does not guarantee performance and quality of software in execution with respect to the user's expectation.
- *Acceptable value for metrics evaluation:* With different views of quality, it is hard to find a numerical value for quality which could be acceptable by all the people. Also, having different views affects software categorization in certain classification by considering the numerical value as the only parameter on software evaluation.

The Solution: Code without value of execution is not valuable: The value of software metrics must be modified by runtime values for better results. Also, using a good structure and patterns (such as design patterns [7]) in the software design and resulting architecture could increase the software quality. Thus, we want to consider the architectural quality of software by considering the use of design patterns (or lack thereof) in the software architecture.

3.3 Quality Model

a quality model is a schema to better explain of our view of quality. Some existing quality models can predict fault-proneness with reasonable accuracy in certain contexts. Other quality models attempt at evaluating several quality characteristics but fail at providing reasonable accuracy, from lack of data mainly.

We believe that quality models must evaluate high-level quality characteristics with great accuracy in terms well-known to software engineers to help maintainers in assessing programs and thus in predicting maintenance effort.

Such quality models can also help developers in building better quality programs by exposing the relationships between internal attributes and external quality characteristics clearly.

We take a less "quantitative" approach than quality models counting, for example, numbers of errors per classes and linking these numbers with internal attributes. We favour a more "qualitative" approach linking quality characteristics related to the maintainers' perception and work directly.

The Problem:

- *Are all sub-characteristics equal in affecting software characteristics:* In the literature, quality models define the relation between quality characteristics and sub-characteristics. However, the impacts of quality sub-characteristics on characteristics are not equivalent. For example: Adaptability and Instalability are two sub-characteristics related to Portability, the question is: If we assess the value of Adaptability as A and the value of Instalability as B, then is the value of Protability equals to A+B or $\frac{2}{3}A + \frac{1}{3}B$ or ...

– *Main concepts of quality are missing:* In 384 BCE, Aristotle, as a scientist, knew all about medicine, philosophy... In 2005 AD, the concept of quality is the same as science in the age of Aristotle: Quality does not distribute in specific part, when we talk about software quality, we talk about assessing entire items which are part of the concept of quality.

The Solution:

– *Coefficient:* Quality as an objective value is dependent on sets of software attributes and customer's requirements. These attributes are explain as different level of characteristics and sub-characteristics in models of quality, but the relation and impact of each characteristic and sub-characteristic should be distinguished. Models can be made more meaningful for different persons by using coefficients which relate characteristic and sub-characteristic. For example: ISO/IEC TR 9126-2:2003(E) [8] define Maintainability as Analyzability, Changeability, Stability, Testability, and Compliance, but what is the impact of Changeability or Maintainability? 20%? 30%? ...
– *Jack of all trades and master of none:* Assessing all the attributes related to software quality represent lots of work. We extend quality models by defining a subject (super-characteristic) to focus on as base concept in quality. Also, the super-characteristic describe the context of the model.

4 Our approach to Software Quality Evaluation

To highlight some solutions of the above mentioned problems, we deal with the 9 steps needed to apply our approach to software quality evaluation, which solves some of the open issues.

4.1 Step by Step:

The following steps highlight the main ideas to implement software quality assessment while considering human requirements.

Step1: Choosing Category of People. We must choose at least a person from the category of people which our software evaluation will be implement for, for example: Programmers, End-user ...

Step2: Identifying Sample Program. We must choose a simple programs ($\mathcal{BP}$) to be considered as sample evaluation set of our model.

Step3: Building a Quality Model. The process of building a quality model decomposes in two main tasks generally:

– Choosing a super-characteristic.
– Choosing and organising characteristics related to super-characteristic.

In our case study, we consider design patterns especially as bridge between internal attributes of programs, external quality characteristics, and software engineers.

Step4: Human Evaluation. The small group, or at least one person from the group, must look in the program or product $\mathcal{BP}$ and evaluate the quality charateristics we defined in our quality model, the evaluation could be in form of numerical value or different levels on aLickert scale.

Step5: Computing Software Metrics over $\mathcal{BP}$. By using software metrics we evaluate $\mathcal{BP}$ numerical values related to software internal attributes.

Step6: Machine Learning Tools. JRip [9] as machine learning algorithm generate the relation between human evaluation of software quality (result from Step4) and value of software metrics (result from Step5). The WEKA's out put consider as set of $\mathbb{RULE}$ (an example is shown in Table1) to be used for other software evaluation .

$$\begin{array}{rl} if & (LCOM5 \leq 1.1) \wedge (NOA \leq 33.25) \\ then & (Learnability = Good) \\ else & (Learnability = Fair) \end{array}$$

Table 1. $\mathbb{RULE}$ for learnability by assessment the quality value of $\mathcal{BP}$

Step7: Computing Software Metrics over $\mathcal{EP}$. Software metrics are used to assess the values of internal attributes over the $\mathcal{EP}$ in the same way as they were for the evaluation of $\mathcal{BP}$

Step8: Adapting Metric. By using ratio over the values from Step7 and Step5, we can related the numerical values of Step7 with those of Step5. The following method will be used for relation evaluation:

Phase1. Finding the Max and Min value of each metrics in $\mathcal{EP}$.

Phase2. Finding the Max and Min value of same metrics we were compute on *Phase1* over the $\mathcal{BP}$.

Phase3. Analyzing the ratio for the values from *Phase1* plus values we have in $\mathbb{RULE}$, we build a new $\mathbb{RULE}$ compatible with $\mathcal{EP}$. For example: consider the $\mathbb{RULE}$ in Table1, considering that upper range and lower range of metrics NOA in $\mathcal{BP}$ is $U^{\mathcal{BP}}_{NOA}$ and $L^{\mathcal{BP}}_{NOA}$, then the new $\mathbb{RULE}$ for Learnability is presented in Table2.

Step9: Software Evaluation. Now, we can evaluate other programs ($\mathcal{EP}$) by applying the of adjusted $\mathbb{RULE}$ (from Step8) and software metrics evaluation over the $\mathcal{EP}$.

$$
\begin{array}{rc}
if & (LCOM5 \leq \frac{(U^{\mathcal{EP}}_{LCOM5} - L^{\mathcal{EP}}_{LCOM5})(1.1 - L^{\mathcal{BP}}_{LCOM5})}{U^{\mathcal{BP}}_{LCOM5} - L^{\mathcal{BP}}_{LCOM5}}) \\
 & \wedge \\
 & (NOA \leq \frac{(U^{\mathcal{EP}}_{NOA} - L^{\mathcal{EP}}_{NOA})(33.25 - L^{\mathcal{BP}}_{NOA})}{U^{\mathcal{BP}}_{NOA} - L^{\mathcal{BP}}_{NOA}}) \\
then & (Learnability = Good) \\
else & (Learnability = Fair)
\end{array}
$$

Table 2. Adjustable learnability $\mathbb{RULE}$ for $\mathcal{EP}$

4.2 Conclusion

To our best knowledge, the method we presented is new but still we were using the classical tools of software engineering. The only modification we respect from our modification in Section3 is using the super-characteristic for building our software quality model.

5 Case Study

We perform a case study to apply the previous approach to building and to applying a quality model considering program architectures.

5.1 General Information

The following general information offer a synthetic view on our quality evaluation method.

Dependent Variables. The dependent variables in our quality model are the quality characteristics related to a super-characteristics.

Independent Variables. The independent variables in our quality model are the internal attributes which can be measured by software metrics. These internal attributes are similar to those in other quality models from the literature: Size, filiation, cohesion, coupling, and complexity.

Analysis Technique. We use a propositional rule learner algorithm, JRip. JRip is Weka—an open-source program collecting machine learning algorithms for data mining tasks [9]—implementation of the Ripper rule learner. It is a fast algorithm for learning "If–Then" rules. Like decision trees, rule learning algorithms are popular because the knowledge representation is easy to interpret.

Design Patterns. We use design patterns as a basis to build a quality model. We choose design patterns because they are now well-known constructs and have been studied extensively.

Design patterns provide *good* solutions to architectural design problems, which maintainers can use in the assessment of the quality characteristics of program architectures naturally. Indeed, "[a]ll well-structured object-oriented architectures are full of patterns" [7, page xiii]. Also, design patterns provide a basis for choosing and for organising external quality characteristics related to the maintenance effort.

5.2 Implementation of our Approach to Software Quality Assessment

We perform the following tasks to build a quality assessment method considering program architectures based on design patterns.

Choose Category of People. We consider for our experience group of university students who know enough about programming and are familiar with the basic concepts of software engineering.

Building a Quality Model. Paragraph *Defining a Super-characteristic.*

By considering software reusability as super-characteristic in our quality model, we focus on reusability, understandability, flexibility, and modularity [7]. So, we add these quality characteristics to our quality model.

Also, through our past experience, we add robustness and scalability (which define together software elegancy [10]) to our quality model.

Software elegancy is defined as maximizing the information delivered through the simplest possible interface. Issues of elegance in software are reflected to robustness, scalability, flexibility, and usability [11].

Thus, the structure of our model starts with the following quality characteristics:

- Flexibility.
- Reusability.
- Robustness.
- Scalability.
- Usability.

Organising the Quality Characteristics. We consider a hierarchical model, because it is more understandable and also most of the standard models are hierarchical [12]. To define attributes and metrics for our model, we start with standard definitions from IEEE and ISO/IEC and, if we do not find a match for the characteristics we are looking for, we try to match them with other models.

- Usability: ISO/IEC defines the usability as part of quality characteristics related with the following attributes:

- Understandability.
- Learnability.
- Operability.

For assistance of this definition, McCall's model defines the usability as:

- Operability.
- Training.
- Communicativeness.

To cover the understandability's attributes, Boehm's model define the understandability as a characteristic that is related to:

- Structuredness.
- Conciseness.
- Legibility.

– Reusability McCall's model defines the reusability as a characteristic that is related with following attributes:

- Software system independence.
- Machine independence.
- Generality.
- Modularity.

– Flexibility
McCall's model defines the flexibility as a characteristic that is related with following attributes:

- Self Descriptiveness.
- Expendability
- Generality.
- Modularity.

– Scalability
Smith and Williams are defined the scalability as "the ability of a system to continue to meet its response time or throughput objectives as the demand for the software functions increases" [13], but with considering the vertical definition, the Scalability would be increase by levels of processing power and application performance" [14].
– Robustness Donald G. Firesmith in his Technical Note [15] define the Robustness as a characteristic that is related to:

- Environmental tolerance.
- Error tolerance.
- Failure tolerance.

Tus, we organise the quality characteristics and decompose these in sub-characteristics using definitions from IEEE, ISO/IEC, and several other models, such as McCall's, Boehm's, Firesmith's [13, 15, 16, 14]. Figure 1 presents our quality model to evaluate software quality related to software maintenance based on design patterns.

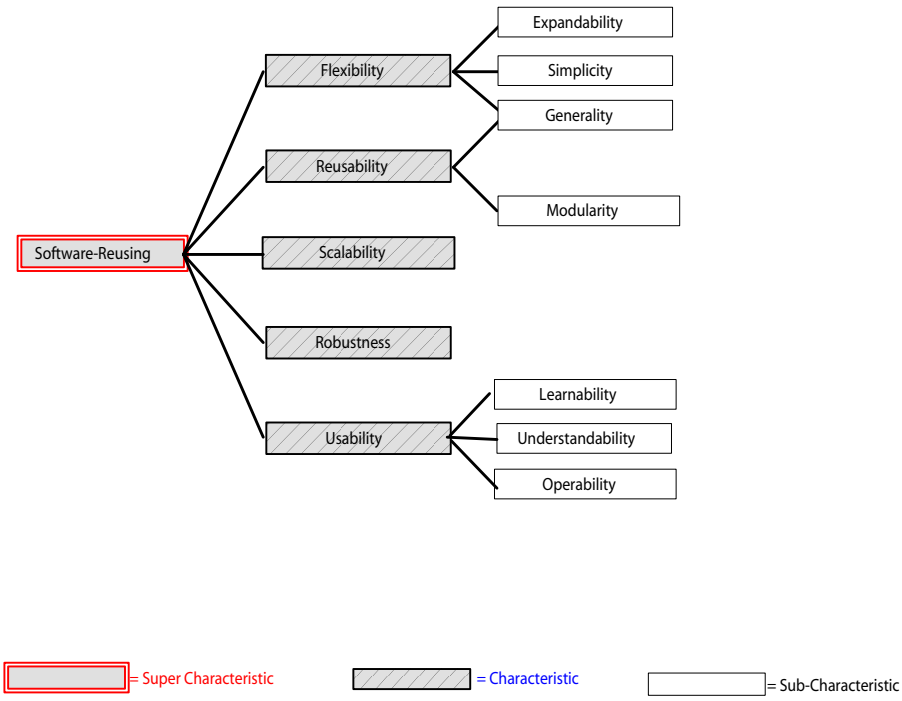

Fig. 1. A quality model based on software reusing

Software Metrics. We choose size, filiation, coupling, cohesion, and complexity as internal attributes. We use the metrics from Chidamber and Kemerer's study [17] mainly to measure these internal attributes, with additions from other metrics by Briand *et al.* [18], by Hitz and Montazeri [19], by Lorenz and Kidd [6], and by Tegarden *et al.* [20].

The complete list of metrics used to measure internal attributes is: ACAIC, ACMIC, AID, CBO, CLD, cohesionAttributes, connectivity, DCAEC, DCMEC, DIT, ICHClass, LCOM1, LCOM2, LCOM5, NCM, NMA, NMI, NMO, NOA, NOC, NOD, NOP, SIX, and WMC.

Identifying Base Program ($\mathcal{BP}$). We use the set of programs implementing design patterns from Kuchana's book as base programs [21]. Each program of this set implements design patterns from Gamma *et al.*'s book [7]. This set of programs forms our base programs $\mathcal{BP}$.

Human Evaluation. We assess the quality characteristics of design patterns manually, using our quality model and the set $\mathcal{BP}$. Table 3 summaries our evaluation of the quality characteristics of the twenty-three design patterns.

Computing Metrics. The metrics we chose for evaluation of $\mathcal{BP}$ are used to measure the internal attributes of programs are all class-based metrics.

	Quality Sub-characteristics and Characteristics								
Design Patterns	Expendability	Simplicity	Generality	Modularity	Learnability	Understandability	Operability	Scalability	Robustness
Abs. Fact.	E	E	G	G	G	G	G	G	G
Builder	G	G	F	F	F	G	F	G	G
Fact. Met.	P	P	F	G	G	G	G	G	G
Prototype	E	G	F	G	F	G	F	E	G
Singleton	P	B	F	E	F	F	F	G	G
Adapter	F	F	P	G	G	F	F	G	F
Bridge	G	F	G	G	F	F	G	G	G
Composite	F	F	F	F	F	G	F	F	G
Decotator	E	E	G	F	G	G	G	G	F
Façade	G	G	G	G	F	G	F	F	F
Flyweight	P	P	F	G	G	P	F	G	G
Proxy	G	P	F	G	F	P	G	G	F
Chain of Res.	G	G	G	P	F	F	G	P	F
Command	G	P	F	F	P	B	G	G	G
Interpreter	G	F	G	F	F	F	G	G	F
Iterator	E	E	G	F	G	F	F	G	G
Mediator	G	F	G	G	F	F	G	G	F
Memento	G	F	F	B	P	F	G	F	P
Observer	E	G	E	F	F	G	G	G	G
State	G	G	F	P	F	B	G	G	F
Strategy	G	F	P	F	P	P	F	P	F
Tem. Met.	E	G	F	F	G	G	G	G	G
Visitor	E	G	G	F	G	P	F	G	F

Table 3. Design patterns quality characteristics in $\mathcal{BP}$ (E = Excellent, G = Good, F = Fair, P = Poor, and B = Bad)

We analyse the programs and their micro-architectures using PADL, a meta-model to represent programs. Then, we apply POM, a framework for metrics definition and computation based on PADL [22], on the program models to compute the metric values.

Linking Internal Attributes and Quality Characteristics. We use the JRip algorithm to find the relation between quality characteristics and values of the metrics. The rule in Table 1 is the rule associated with the learnability quality characteristics, when applying JRip ion the metric values of the base programs. It shows that the learnability quality characteristics is related to the NOA and LCOM5 metrics more than to any other metric.

We do not introduce here all the rules found for the different quality sub-characteristics and characteristics in our model for lack of space. The rules are specific to the current case study but help in illustrating the advantages and limitations of our approach.

Adapting our $\mathbb{RULE}$. We apply the quality model built in the previous Subsection to JHotDraw (we only apply our model on a subset of the micro-architectures for lack of space), JUnit, and Lexi programs. For lack of space, we only applying $\mathbb{RULE}$ in Table1.

We adapt the metric values in the rule in Table1 by computing the ratio between the minimum and maximin values of the LCOM5 and NOA metrics for the base programs on the one hand, and each micro-architecture on the other hand. Table 4 also displays the adapted rules for all the micro-architectures.

Upper lower range of NOA and LCOM5 for $\mathcal{BP}$ are computed as:

$min_{LCOM5} = 0.75$, $max_{LCOM5} = 1.82$, $min_{NOA} = 1.00$, and $max_{NOA} = 86.00$.

Micro-Architectures	Design Patterns	$LCOM5$	min_{LCOM5}	max_{LCOM5}	NOA	min_{NOA}	max_{NOA}	Rule for learnability
Subset of the micro-architectures in JHotDraw								
MA74	Command	1.07	0.50	1.63	29.35	1.00	164.00	$(LCOM5 \leq 1.16) \wedge (NOA \leq 62.30) \Rightarrow Good$
MA85	Singleton	0.67	0.67	0.67	1.00	1.00	1.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00) \Rightarrow Fair$
MA91	Strategy	0.95	0.80	1.0	553.88	221.00	792.00	$(LCOM5 \leq 0.21) \wedge (NOA \leq 218.23) \Rightarrow Fair$
JUnit								
MA65	Composite	0.65	0.25	0.95	70.10	4.00	148.00	$(LCOM5 \leq 0.72) \wedge (NOA \leq 56.33) \Rightarrow Fair$
MA66	Decorator	0.65	0.25	0.90	135.41	49.00	176.00	$(LCOM5 \leq 0.67) \wedge (NOA \leq 49.68) \Rightarrow Fair$
MA67	Iterator	0.92	0.83	0.99	30.67	1.00	48.00	$(LCOM5 \leq 0.17) \wedge (NOA \leq 18.38) \Rightarrow Fair$
MA68	Observer	0.90	0.66	1.03	112.43	1.00	191.00	$(LCOM5 \leq 0.38) \wedge (NOA \leq 74.32) \Rightarrow Fair$
MA69	Observer	0.83	0.83	0.83	1.00	1.00	1.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00) \Rightarrow Fair$
MA70	Observer	0.83	0.83	0.83	11.00	11.00	11.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00) \Rightarrow Fair$
MA71	Singleton	0.00	0.00	0.00	1.00	1.00	1.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00) \Rightarrow Fair$
MA72	Singleton	0.00	0.00	0.00	1.00	1.00	1.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00) \Rightarrow Fair$
Lexi								
MA8	Builder	0.95	0.93	0.97	7.75	1.00	12.00	$(LCOM5 \leq 0.03) \wedge (NOA \leq 4.30 \Rightarrow Fair$
MA9	Observer	0.95	0.94	0.97	9.50	1.00	18.00	$(LCOM5 \leq 0.02) \wedge (NOA \leq 6.65) \Rightarrow Fair$
MA10	Observer	0.95	0.94	0.97	61.67	35.00	94.00	$(LCOM5 \leq 0.02) \wedge (NOA \leq 23.08) \Rightarrow Fair$
MA11	Singleton	1.01	1.01	1.01	1.00	1.00	1.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00) \Rightarrow Fair$
MA12	Singleton	0.99	0.99	0.99	2.00	2.00	2.00	$(LCOM5 \leq 0.00) \wedge (NOA \leq 0.00) \Rightarrow Fair$

Table 4. Data and rules when applying the quality model to a subset of JHotDraw, JUnit, and Lexi

Applying the Rules. We compare the expected metric values in the adapted rules with the metric values computed for each micro-architecture and we update the $\mathbb{RULE}$ related to our software evaluation. The results are shown in Table4.

6 Conclusion

In this position paper, we reported our experience in using software metrics and quality models to assess software. Our conclusion are:

- Software quality models must state clearly their target user's and define an supplementary layer of characteristics ("super"-characteristics) to be more useful and comparable.
  item Software quality models must take into account other aspects of software such as their performance, runtime adequacy, and architecture (for example, through the assessment of design patterns).
- More experience is needed to describe out advantage and disadvantage of our quality assessment method.

We would like to discuss with the participants the identified open issued and their proposed solutions and opportunities of improving software quality models.

References

1. O'Regan, G.: A Practical Approach to Software Quality. 1^{st} edn. Springer (2002)
2. Floyd, C., Budde, R., Zullighoven, H.: 1. In: Human Questions in Computer Science. Springer Verlag (1992) 15–27
3. Basili, V.R.: The experimental paradigm in software engineering. In Rombach, H.D., Basili, V.R., Selby, R.W., eds.: proceedings of the international workshop on Experimental Software Engineering Issues: Critical Assessment and Future Directions, Springer Verlag (1992) 3–12
4. Glass, R.L.: The software-research crisis. IEEE Software **11** (1994) 42–47
5. ISO: ISO/IEC 14598-1. International Standard **Information technology software product evaluation** (1999)
6. Lorenz, M., Kidd, J.: Object-Oriented Software Metrics. 1^{st} edn. Prentice Hall (1994)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. 1^{st} edn. Addison-Wesley (1994)
8. Standard, I.: Iso/iec 9126-1. Institute of Electrical and Electronics Engineers **Part 1,2,3: Quality model** (2001)
9. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. 1^{st} edn. Morgan Kaufmann (1999)
10. for Software Engineering, C.: OO Analysis and Design: Modeling, Integration, Abstraction. (2002) http://sunset.usc.edu/classes/ cs577b_2002/EC/03/EC-03.ppt.
11. Erich Gamma, Richard Helm, R.J., Vlissides, J.: Design Patterns Elements of Reusable Object-Oriented Software. Addison-Wesley Pub Co (1995)

12. Fenton, N.E., Pfleeger, S.L.: Software Metrics A Rigorous and Practical Approach. 2^{nd} edn. PWS Publishing Company (1997)
13. Connie U. Smith, L.G.W.: 1. In: Introduction to Software Performance Engineering. Addison Wesley (2001) http://www.awprofessional.com/articles/article.asp?p=24009.
14. Online, C.: Scalability from the edge. Computer Business review Online, CBR Online (2002)
15. Firesmith, D.G.: Common concepts underlying safety, security, and survivability engineering. Carnegie Mellon Software Engineering Institute - Technical note CMU/SEI-2003-TN-033 (2003)
16. Khosravi, K., Guéhéneuc, Y.G.: A quality model for design patterns. Technical Report 1249, Université de Montréal (2004)
17. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object-oriented design. Technical Report E53-315, MIT Sloan School of Management (1993)
18. Briand, L., Devanbu, P., Melo, W.: An investigation into coupling measures for C++. In Adrion, W.R., ed.: proceedings of the 19^{th} International Conference on Software Engineering, ACM Press (1997) 412–421
19. Hitz, M., Montazeri, B.: Measuring coupling and cohesion in object-oriented systems. In: proceedings of the 3^{rd} International Symposium on Applied Corporate Computing, Texas A & M University (1995) 25–27
20. Tegarden, D.P., Sheetz, S.D., Monarchi, D.E.: A software complexity model of object-oriented systems. Decision Support Systems **13** (1995) 241–262
21. Kuchana, P.: Software Architecture Design Patterns in Java. 1^{st} edn. Auerbach Publications (2004)
22. Guéhéneuc, Y.G., Sahraoui, H., Zaidi, F.: Fingerprinting design patterns. In Stroulia, E., de Lucia, A., eds.: proceedings of the 11^{th} Working Conference on Reverse Engineering, IEEE Computer Society Press (2004) 172–181