The « SQALE » Analysis Model

An analysis model compliant with the representation condition for assessing the Quality of Software Source Code

Jean-Louis Letouzey
DNV IT Global Services
Arcueil, France
jean-louis.letouzey@dnv.com

Thierry Coq
DNV IT Global Services
Arcueil, France
thierry.coq@dnv.com

Abstract — **This paper presents the analysis model of the assessment method of software source code SQALE (Software Quality Assessment Based on Lifecycle Expectations). We explain what brought us to develop consolidation rules based in remediation indices. We describe how the analysis model can be implemented in practice.**

Keywords – quality ; source code ; quality model ; analysis model ; SQALE.

I. Introduction

We estimate that the analysis model and more precisely, the aggregation rules of base measures is a key success factor for the effective implementation of qualimetry within an organisation. This paper details the analysis model of the SQALE (Software Quality Assessment Based on Lifecycle Expectations) method, defined by DNV to assess the quality of software and in particular software source code. An overall view of the SQALE method has been presented in our white paper [1]. Also the quality model of SQALE and its application to real-time or embedded software has been described in [2]. We will explain in this paper what brought us to develop our own analysis model, explain the objectives we were striving to reach for this model and how they were concretely applied within the SQALE method.

II. A Known Need, Not Covered by Existing Standards

Use cases of qualimetry are numerous. For example, verifying a piece of software delivered by a supplier, benchmarking open source applications, and the comparing of the work performed by two different teams.

Taking in account these needs, one would wish to have a system, a method to assess the quality of software, which, as any efficient measurement system should be, would be objective, precise and sensitive: For a given measured software, a precise rating should be obtained and after some refactoring has been performed resulting in an increase of the rating, a concrete increase in perceived quality should also be achieved.

Qualimetry is a well-known discipline. It has been supported and promoted by standards such as ISO 9126 [3]. This standard indeed describes the characteristics and sub-characteristics one should expect from software. If one focuses on internal qualities of the software, particularly the software source code, this standard helps to identify the measurement control points that may be useful to characterize the software's quality. A source code quality mol;del can be created which will link base or derived measures with characteristics such as maintainability or reliability. For example, the comment ratio or the locally excessive complexity of an application may be measured to assess its maintainability

However, the ISO 9126 standard and other preceding quality models such as Boehm's quality model [4] don't give a method nor precise enough directives to compute the aggregate indices derived from the proposed measures to rate the quality characteristics they have identified.

The aggregation issue can be broken down in two steps, as represented in Figure 1. A first step is to bring each measure performed back to a common scale. Measures may be structural such as cyclomatic complexity [5], object-oriented such as the RFC (Response For a Class) [6] or the result of the count of coding rules violations, all need to be transformed to obtain final measures on a common scale.

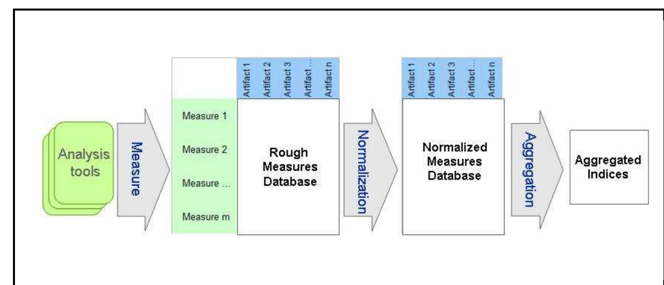


Figure 1. From rough measure to usable index

In a second step, since all measures are now on the same scale, it will now become possible to perform aggregating operations. The subject of this paper is not the normalization function, which issues are well known and will not be dealt with further here. For the aggregation, the issue is more complex since it may be applied to two different hierarchies. Indeed, the need for aggregation may be summarized as follows: the software product's quality may be characterized

based on the hierarchy of the source code artefacts. The software is a breakdown of software components themselves broken down further into sub-components. These are further broken down into classes and operations (for object-oriented software), and so on.

The second hierarchy is based on the quality model which breaks down the overall quality into a set of quality characteristics themselves broken down into sub-characteristics, and so on until the last layer of base measures directly linked to the source code is reached.

As shown in Figure 2, operations are needed to compute the quality index of a couple Artefact, Quality (sub-) characteristic {A, Q}, using either the quality indices of the lower level of the artefact hierarchy, or the quality indices of the lower level of the quality characteristic hierarchy. Using the lowest quality couples {A_0, Q_0} and applying aggregation operation, the whole application may be rated in all its quality aspects.

The lack of standard ruling the aggregation operations has led the user to create their own aggregation method.

Based on available literature and specialized tools, the most common chosen method is based on a system of weighted averages.

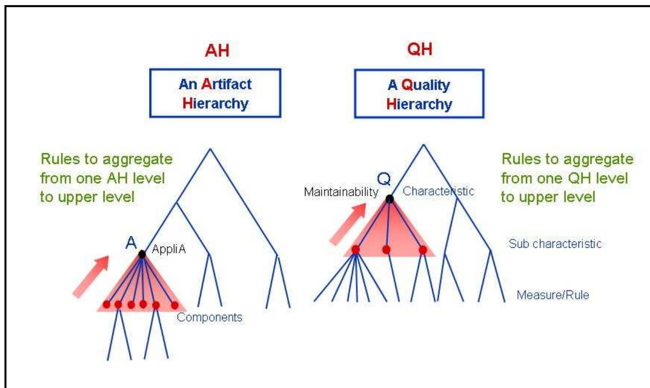


Figure 2. The aggregation needs within two different hierarchies

This approach is advantageous in the sense it is simple to implement and to detect in some cases the risky components of an application. However aggregating measures in this manner is applying a rather questionable principle if the measurement system must be reliable. This is what this paper demonstrates in the following.

III. THE GOAL

Based on our experience in code auditing and our need of a method to aggregate base measurements as precise enough as possible for the SQALE method, we have defined a specific aggregation system.

We have applied the measurement theory to software qualimetry, in particular the representation condition, to guide our choices for the implementation of our method.

This representation condition (see Figure 3) is critical, since any system which would violate this condition would provide measures which may not be representative, in other words establishing wrong findings. A non-representative assessment system is not reliable, the users will either not trust, or will spend many hours checking its diagnostic. Since most qualimetry users try to systematize, automate and make the results of code assessment reproducible, an analysis produced in a few minutes (or a few hours depending on the application size) should not need many hours of navigation and inspection to verify and validate its findings.

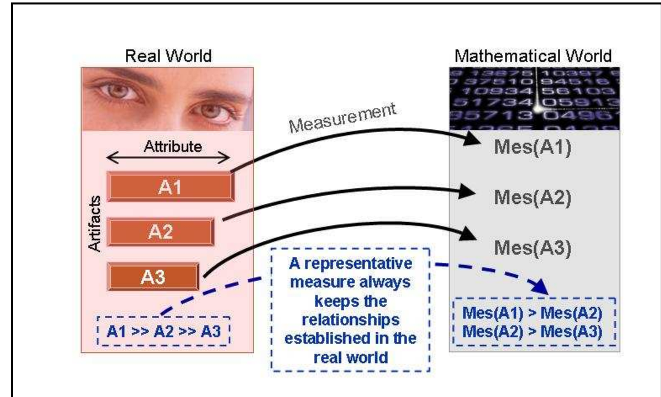


Figure 3. An illustration of the representation condition

Studying the measurement principles shows that the allowable aggregation operations depend on the type of measure used. More precisely they only depend on the type of scale of the measurement used.

Five types of scales are generally defined [7]. They are presented in the Table 1. The valid aggregation operations are described in this table, which allow a limited number of combinations for the {Scale type, operation} couple. The table shows that if a measure has been normalized using an ordinal scale (for example rating from 'A' to 'E' based on the results of the measures on the code), the allowable aggregations are the 'min', the 'max' and the 'median' operations.

Scale	Valid Transformation	Main Valid Agregation
Nominal	1 to 1 mapping	None
Ordinal	Monotonic increasing function	Min, Max, Median
Interval	M' = aM + B (a>0)	Min, Max, Median, Average
Ratio	M' = aM (a>0)	Min, Max, Median, Mean, Average, Sum, Distance (Euclidian or other)
Absolute	M' = M	All

Table 1: Scale types, allowable transformations and aggregations

The systematic analysis of the various possible combinations has allowed us to identify situations where the representation condition is not satisfied. We have identified three categories of such situations: masking, compensation and threshold effects. By removing from the possible couples

{Scale type, Operation} all that exhibit these effects, the allowable couples are in the end very limited.

A. *The masking effect*

The masking effect appears when the aggregate value is not sensitive to the variation of one of the base values.

Table 2 illustrates the phenomenon. A first version of the software Va is composed of 5 files which measures are provided on an ordinal scale from A to E. Aggregates such as min, max and median are provided. A second version of the same software Vb is nearly identical except for one file which measure changes from E to D, which is an improvement. The case shows there is no impact on the aggregate values which therefore demonstrates the representation condition is violated by each of the aggregate operations.

	File 1	File 2	File 3	File 4	File 5	Min	Max	Median
MyAppli Va	A	A	C	E	E	A	E	C
MyAppli Vb	A	A	C	D	E	A	E	C

Table 2: Aggregation by Min, Max and Median on 2 different artefacts

B. *The compensation effect*

The compensation effect appears on some aggregation functions such as: mean, weighted mean, median. For example, these functions are used in the Mi3 and Mi4 [8] indices to aggregate the comment ratio. An acceptable comment ratio average may result from compensating one file's few comments with other files' over-abundantly commented. On average, the aggregate comment ratio may be judged as acceptable. Yet it is quite certain the surplus of comments in one file will not alleviate the lack of comments in another. This observation can be generalized to each aggregate operation using averages, weighted or not. In conclusion, averages allow compensation effects which result in not diagnosing some weaknesses of software applications.

C. *The threshold effect*

The last effect mentioned is the threshold effect. It appears in some cases where the normalization is performed on a limited interval, for example [0, 100], and where the normalization function uses a threshold.

In some cases, such as those illustrated in the Figure 4, different base measures will lead to the same normalized value. This violates the representation condition.

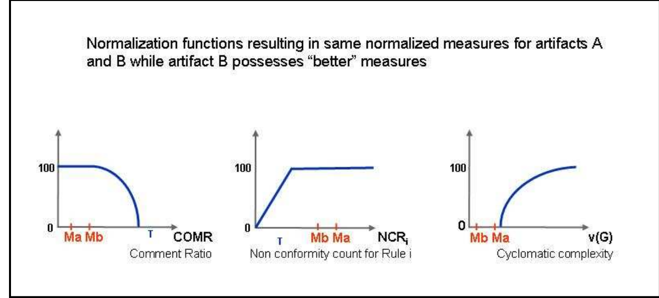


Figure 4. Threshold effect due to some normalization functions

In the end, and taking into account the Table 1, the effects shown previously lead us to remove all combinations using an ordinal or interval scale type and all min, max, median and average operations on ratio scales.

Taking into account our main criteria:

- satisfying the representation condition,
- adhering to measurement theory and allowed operations for a given scale type,

As Table 3 synthesizes, the only allowable aggregation rules are the addition and distance operations on a ratio or absolute scale type. The SQALE method's implementation of the above is presented in the next section.

Scale	Min, Max, median	Average, Weighted average	Sum, Distance
Nominal	Not allowed	Not allowed	Not allowed
Ordinal	Potentially not representative	Not allowed	Not allowed
Interval	Potentially not representative	Potentially not representative	Not allowed
Ratio	Potentially not representative	Potentially not representative	Representative
Absolute	Potentially not representative	Potentially not representative	Representative

Table 3: Synthesis of the various combinations {Scale type, aggregation operation} study

IV. Applying the SQALE Analysis Model

Before describing in detail the implementation of the SQALE analysis model, which complies with the above observations, the quality model of the SQALE method is briefly presented.

The quality model of the SQALE method is a model which translates the requirements applicable to the software and its source code over its life cycle. In the same manner that the activities linked to making the software and in particular its source code, follow a clear chronology the requirements applicable to the source code appear in a same order. The approach and the structure of the SQALE quality model has been detailed elsewhere [1] and are summarized

in Figures 5 and 6. The generic SQALE model is derived according to the implementation technologies (design and source code languages) and the tailoring needs of the project.

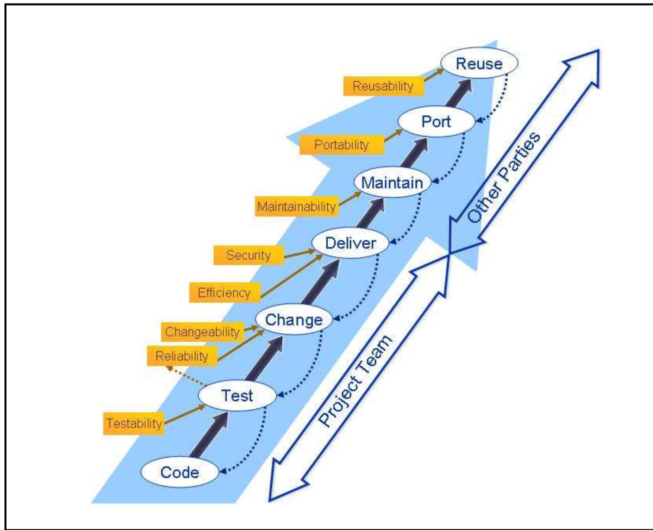


Figure 5. The “File” Lifecycle used for the SQALE Quality Model. Dependencies between Activities and Quality Characteristics

As stated previously, the quality model is a requirements model. As written by Ph. Crosby [9], assessing a software source code is therefore similar to measuring the distance which separates it from its quality target. To measure this distance, the concept of remediation index has been defined and implemented in the SQALE analysis model. A quality index is associated to each component of the software source code (for example, a file, a module or a class). The index represents the remediation effort which would be necessary to correct the non compliances detected in the component, versus the model requirements. Since the remediation index represents a work effort, the consolidation of the indices is a simple addition of uniform information, which is compliant with the representation condition and a major advantage of the model. As illustrated in Figure 7, a component index is computed by the simple addition of the indices of its elements.

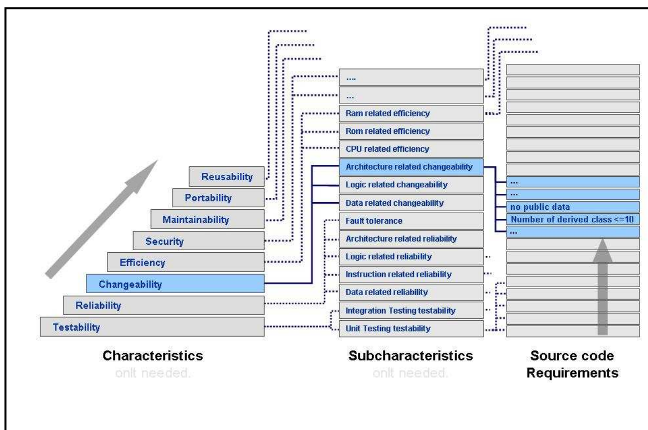
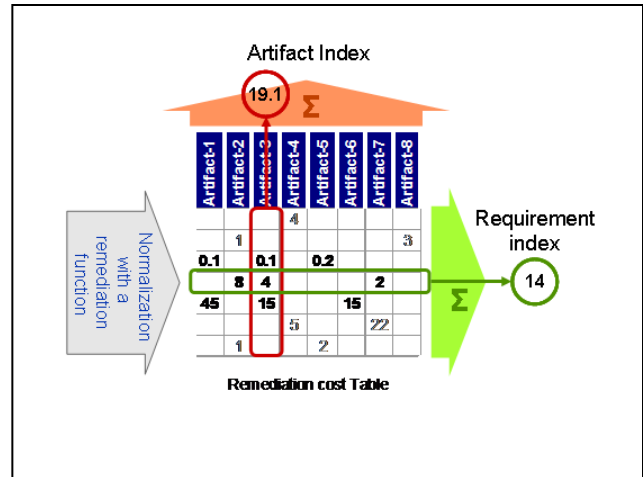


Figure 6. Some details of Level 2 and 3 of the SQALE Quality Model

A characteristic index is computed by the addition of the base indices of its sub-characteristics. A sub-characteristic index is computed by the addition of the indices of its control points.



Base indices are computed by applying the following rules:

- A base index takes into account the unit remediation effort to correct the non-compliance. In practice, this effort mostly depends on the type of the non-compliance. For example correcting a presentation defect (bad indentation, dead code) does not have the same unit effort cost as correcting an active code defect (which implies the creation and execution of new unit tests, possible new integration and regression tests).
- A base index also considers the number of non-compliances. For example, a file which has three methods which need to be broken down into smaller methods because of complexity will have an index three times as high as a file which has only one complex method, all other things being equal.

In the end, the remediation indices provide a means to measure and compare non compliances of very different origins and very different types. Coding rule violation non-compliances, threshold violations for a metric or the presence of an antipattern non-compliance can be compared using their relative impact on the index.

The standard measure set of SQALE has more than 30 different control points. A few examples of base measures are explained in more detail, to show how each complies with the conditions explained above:

- A well-known indicator of reduced testability is an excessive cyclomatic complexity for a given operation (procedure or function) in the code. The default threshold for excessive complexity in SQALE is 15. Any operation having a V(G) over 15 will be counted as one violation, and the count is cumulative

per class and file in order to apply the representation condition. This measure is mostly independent of the programming language.
- Another testability indicator is the presence of duplicated code. Indeed, every piece of duplicated code increases the unit test effort.
- A reliability indicator is the absence of dynamic memory allocation (for real-time software) or a balanced use of allocation and deallocation instructions (malloc and freemem). Each violation of this rule increments the count by one, again for each class and file.
- An understandability (a sub-characteristic of maintainability) indicator is the file comment ratio. If it is below SQALE's default threshold of 25%, a violation is counted.
- Finally, the presence of commented-out code is also counted as a readability (maintainability) violation. Indeed commented-out code increases the reading efforts for no added value.

Of course, the various examples presented above have different remediation function. Generic parameters for these functions are available in the SQALE model, which can be tailored to the organisation. The parameters of the remediation functions can be calibrated based on historical data from past projects, if available.

The SQALE quality and analysis models have been used to perform many assessments of software source code, of various sizes and in different application domains. The same layered and generic quality model has been used to assess COBOL, Java, embedded Ada, C or C++ software source code. For Java, C++ and Ada, the quality model contains, of course, object-oriented metrics to assess Testability, Changeability and Reusability. The quality model also provides control points to detect the absence of antipatterns such as those identified by Brown [10].

To summarize even further, ratings in five levels (from "poor" to "excellent") are established for the application managers by setting thresholds on the index densities.

Several graphs and diagrams are used to efficiently visualize the strengths and weaknesses of the assessed source code:

- Which software pieces contribute the most to the risks,
- Which quality criteria are the most impacted,
- What is the impact of the identified non-compliances on the project or its users.

Some of the indicators we use in our assessments and that provide in our experience a good visibility and good information for taking decisions are presented here. (Figures 8 and 9).

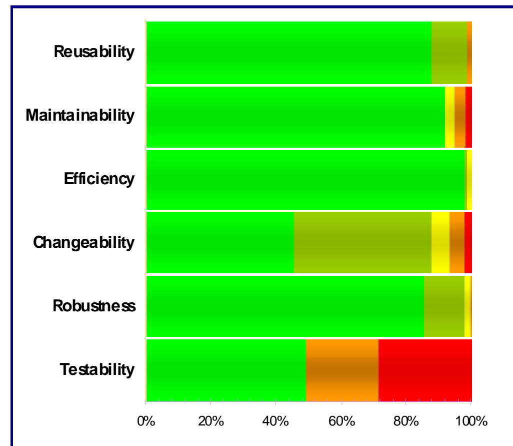


Figure 8. Example of file rating distribution for a sample application

In the case of the application shown in this example, the Testability aggregate index is high. Therefore, due to the chronological dependencies (see Figure 5), the other external characteristics cannot be satisfied (even partially), even in the presence of a rather good compliance to the control points for the higher characteristics. The diagram presents a finding which may seem paradoxical. Indeed, to markedly improve the Maintainability of the application, as perceived by the owner, it is preferable and more efficient to improve the Testability than to add comments.

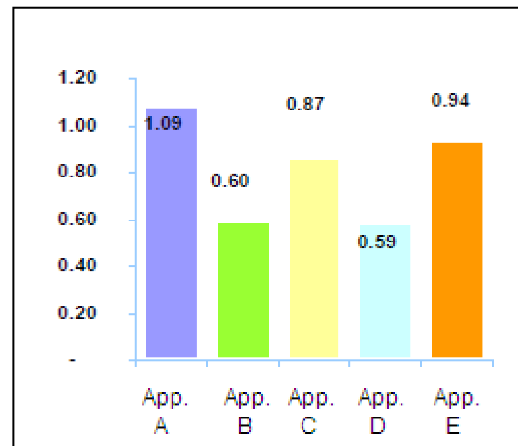


Figure 9: Testability index density broken down per application

The analysis indicators mentioned have allowed us to localize precisely the high priority modifications (quick-wins) to apply to increase the Testability, and therefore, the overall quality of the application. The analysis of the index density as shown in Figure 9 for example (the density is the ratio of the index to the size of the piece of software) has also enabled us to compare the non compliance distribution between different parts of the overall application. An

additional effort is needed to improve the testability of parts A, C and E by re-factoring.

V. CONCLUSION

This paper presents an original analysis model based on remediation indices. To aggregate theses indices, the addition operation is simply applied to the lower level indices. These aggregation operations do not exhibit any of the defects that may be encountered using other aggregation operations. The absence of any masking, compensating or threshold effects provides aggregates and high level indices completely compatible with the representation clause. This is true, of course, only if base attributes and measures are selected that comply with the representation clause.

Both the quality and the analysis models of the SQALE model are easy to implement and to automate, provided the base measures can be implemented.

REFERENCES

[1] J-L Letouzey and Th. Coq, The SQALE Models for assessing the quality of software source code, DNV Paris, white paper, September 2009

[2] J.-L. Letouzey and Th. Coq, The SQALE Models for Assessing the Quality of Real Time Source Code, ERTSS 2010, Toulouse, September 2010

[3] ISO, International Organization for Standardization, 9126-1:2001, Software engineering – Product quality, Part 1: Quality model, 2001

[4] B. W. Boehm, J. R Brown, H. Kaspar, M. Lipow, G. McLeod, and M. Merrit, Characteristics of Software Quality, North Holland, 1978

[5] Th. McCabe and A. H. Watson, Structured Testing: A Testing Methodology using the Cyclomatic Complexity Metric, National Institute of Standards and Technology, Special Publication 500-235, 1996

[6] S. R. Chidamber and C.F. Kemerer, A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, Vol 20, N° 6, PP 476-493, June 1994

[7] N.E. Fenton and S. L. Pfleeger, Software Metrics: A rigourous & Practical Approach, second edition, ISBN 053495425-1,PWS Publishing Company, Boston, 1997

[8] D. Coleman, D. Ash, B. Lowther, and P. W. Oman, Using Metrics to Evaluate Software System Maintainability, IEEE Computer, 1994, 27(8), pp. 44-49

[9] P.B. Crosby, Quality is free : the art of making quality certain, ISBN 0-07-014512-1, McGraw-Hill, New-York, 1979

[10] W.J. Brown, R.C. Malveau, H.W. McCormick, and T.J. Mowbray, John Wiley, New-York, 1998