The SQALE Method for Evaluating Technical Debt

Jean-Louis Letouzey
inspearit
Arcueil, France
jean-louis.letouzey@inspearit.com

***Abstract*–This paper presents the SQALE (Software Quality Assessment Based on Lifecycle Expectations) method. We describe its Quality Model and Analysis Model which is used to estimate the Quality and the Technical Debt of an application source code. We provide recommendations and guidelines for using the SQALE indicators in order to analyse the structure and the impact of the Technical Debt.**

***Keywords*-Technical Debt; quality ; source code ; quality model ; analysis model ; SQALE.**

I. INTRODUCTION

Technical Debt is a powerful metaphor which is used more and more often at all levels of an IT organization. The concept has even been defined and strongly supported by the agile community, but it applies to all kind of developments and all types of languages. Whatever the unit you use for measuring the debt (dollars, hours, work units etc.), these figures are easy to understand and to handle. They allow:

- To provide a common language for all teams
- To monitor trends
- To report comparative data
- To proactively manage application assets.

Being able to estimate the Technical Debt of an application is a good point. With historical data, it allows to monitor the debt and detect progress or regression. But developers and managers need more detail. They expect to be able to understand and analyze the debt. They need information that will help them evaluate risks and set priorities to be able to reduce this debt and furthermore avoid paying the interests of the debt.

The sample graphs which illustrate this paper have been produced with the SQALE plugin of the Sonar [1] tool.

II. THE TECHNICAL DEBT METAPHOR

As a reminder, Technical Debt is a metaphor that was used for the first time in 1992 by Ward Cunningham [2].

To summarize this definition, let's quote him:

- "Neglecting the design is like borrowing money
- Refactoring, it's like paying off the principal debt
- Developing slower because of this debt is like paying interest on the loan.
- Every minute spent on not-quite-right code counts as interest on that debt."

There are numerous discussions, publications [3], [4], [5] and a book [6] about the concept and about the roots of the debt including questions like:

- Does unintentional debt count?
- Do process issues count?

For SQALE and for the rest of this paper:

- The debt evaluated with SQALE is the internal debt associated to the source code of an application. This excludes process related debt.
- The issues may come intentionally (as a way to achieve the objective of a sprint) or unintentionally. In both cases, there will be a negative impact (which means interest to pay), so this should be counted as debt by the method.

III. HISTORY AND OVERVIEW OF THE SQALE METHOD

There has always been a need for a standardized method to objectively evaluate the quality of an application's source code. Standards such as ISO 9126 [7] or ISO / IEC 15939

TABLE 1: SAMPLE LIST OF REQUIREMENTS AND THEIR MAPPING INTO A SQALE QUALITY MODEL.

Characteristic	Sub-Characteristic	Requirement
Maintainability	Readability	There is no commented out block of instruction
Changeability	Architecture related Changeability	There is no cyclic dependency between packages
Reliability	Exception handling Reliability	Exception handling shall not catch NPE
Reliability	Instruction related Reliability	Code shall override both equals and hashcode
Reliability	Data related Reliability	There is no comparison between floating point
Reliability	Coverage related Reliability	All files have unit testing with at least 70% code coverage
Testability	Unit test level Testability	There is no method with a cyclomatic complexity over 12
Testability	Unit test level Testability	There is no cloned parts of 100 tokens or more

[8] provide some guidance but unfortunately are incomplete and do not offer a practical and reliable method to calculate a quality score for source code.
It is in response to this need that inspearit (formerly DNV ITGS) developed the SQALE method (Software Quality Assessment based on Life Cycle Expectations).

The goal was to design a method as objective as possible and with the least amount of "false positives". This is why the method is based on the fundamental principles of the measurement theory including the representation condition [9]. We will see that for SQALE, measuring the quality of an application means measuring its Technical Debt.

The complete method (Version 0.9) is defined in a definition document [10] available, with other justification papers, on the method's website.

The method defines 4 key concepts:

1. **The Quality Model:** The SQALE Quality Model is used for formulating and organising the non-functional requirements that relate to code quality. It is organised in three hierarchical levels. The first level is composed of characteristics, the second of sub-characteristics. The third level is composed of requirements that relate to the source code's internal attributes. These requirements usually depend on the software context and language.
2. **The Analysis Model:** The SQALE Analysis Model contains on the one hand the rules that are used for normalising the measures and the violations relating to the code, and on the other hand the rules for aggregating the normalised values.
3. **The Indices:** All indices represent costs. As we will see later, one represents the Technical Debt of the application. Others are used to analyse that debt.
4. **The Indicators:** The method defines three summarized indicators. They have been defined to provide a visual representation of the Technical Debt.

In addition to these concepts, the method defines 8 principles with the aim of obtaining a precise estimation of the Quality and the Technical Debt.

We could summarize these principles as follows:

- Quality means conformance to requirements, therefore those requirements should first be defined. They should be: atomic, unambiguous, non-redundant, justifiable, acceptable, implementable and verifiable.
- The method assesses the distance to requirement conformity by considering the necessary remediation cost to bring the source code to conformity.
- The method adds up remediation costs to calculate quality indicators.
- The SQALE Quality Model is orthogonal meaning that a quality requirement appears once and only once in the Quality Model.

The method will be detailed in the following paragraphs.

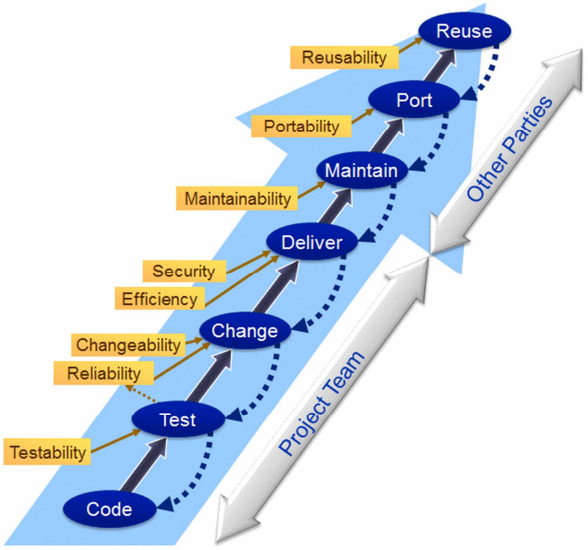

Figure 1. The "File" Lifecycle used for the SQALE Quality Model. Dependencies between Activities and Quality Characteristics.

IV. THE SQALE QUALITY MODEL

The Quality Model of the SQALE method contains the internal properties expected from the code in the context of the evaluation. In other words, SQALE ask projects or organizations to establish their own concrete definition of "right code". This will clearly define what in their context and in their mind generates debt and what does not.

The definition of "right code" will typically include implementation level, naming and presentation requirements. As stated in [11], it should also include architectural and structural requirements. Most agile projects will refer to this definition in their definition of "Done".

Compared to other software Quality Models as [12], this model has four important properties:

- As stated before, it is a requirement model and not a set of best practices to implement.

Figure 2. The first level of the generic SQALE Quality Model.

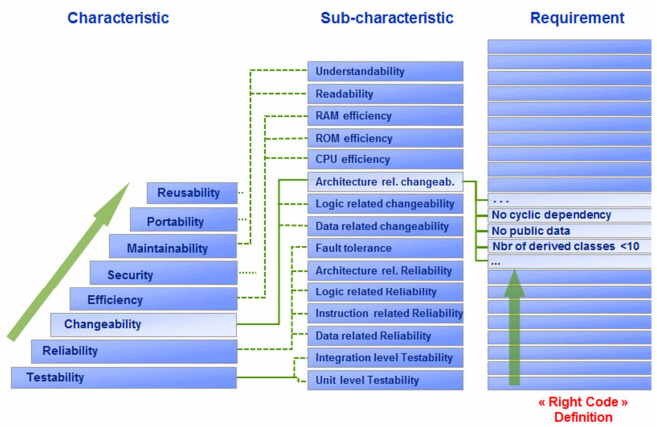

Figure 3. Some details of Level 2 and 3 of the SQALE Quality Model.

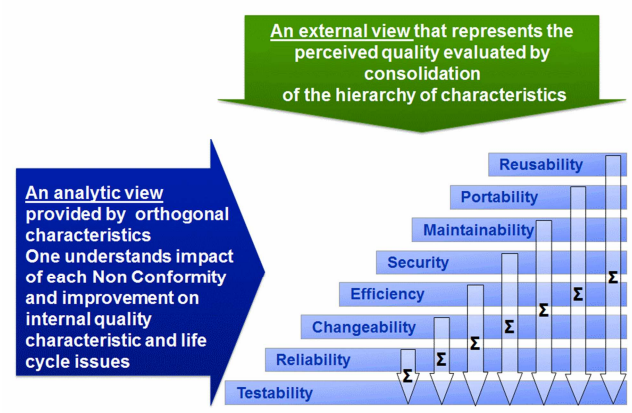

Figure 4. The two viewpoints of the SQALE Quality Model.

- It has been built in a systematic manner that takes into account the lifecycle of source code.
- It is structured in hierarchical layers.
- It takes into account both the developer's point of view and the software user's point of view.

To define the basic quality characteristics of our model, an approach based on the typical lifecycle of a source code file was used.

This approach organizes the requirements and needs for source code quality sequentially as represented in Fig. 1.

The layered model presented in Fig. 2 is the end result of this approach. This basic model can be adapted by taking into account the expected lifecycle for the source code to be assessed.

In this structure, a sub-characteristic or a requirement used in the model is never duplicated in any higher level. Using a requirement and a sub-characteristic only once is an essential rule of the SQALE Quality Model, which must be followed at all times. This rule applies to whatever requirement you have identified in your definition of "right code". Fig. 3 shows how a typical requirement will be mapped in a SQALE Quality Model.

In existing models such as ISO 9126, Testability and Changeability are sub-characteristics of Maintainability. In SQALE, because we consider that they are needed earlier in the lifecycle of a source code file, they are moved to the first level and become characteristics.

The Table 1 provides a sample of "right code" requirements and their mapping into a SQALE Quality Model.

The last property of the SQALE Quality Model is its capacity to support two different points of view, the developer point of view and the owner/user point of view, as illustrated in Fig. 4.

By reading the model horizontally, non-conformities can be analysed by originating characteristic or sub-characteristic to understand the impact on the development activities (testing, implementing changes ...).

By reading the model vertically, the consequences for the owner of operating the software with non-compliances can easily be understood.

V. THE SQALE ANALYIS MODEL

As written previously, with the SQALE method, assessing a software source code is similar to measuring the distance between its current state and its quality target. This distance allows normalizing all internal measures on a common scale. To measure this, the SQALE Analysis Model uses a remediation index, associated to each component of the software source code (for example, a module, a file or a class). The remediation index represents the remediation cost of actions needed to correct the non compliances detected in the component (versus the model requirements). Since the remediation index represents a cost, 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 Fig. 5, a component index is computed by the simple addition of the indices of its elements.

Remediation costs are calculated by associating a

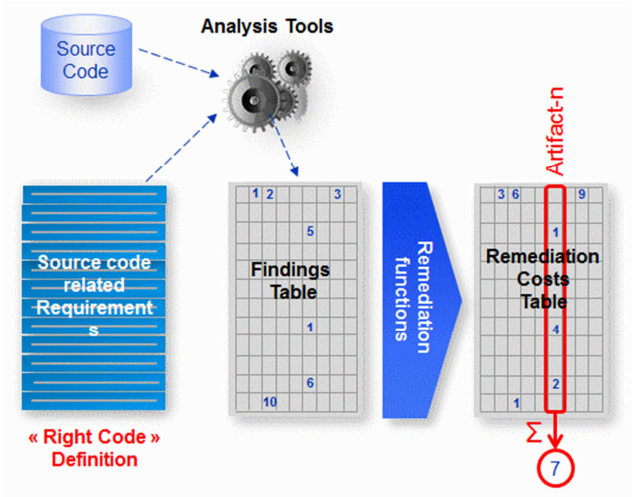

Figure 5. Calculation of a remediation index.

TABLE 2: SAMPLE LIST OF REMEDIATION FUNCTIONS

Requirement	Remediation Details	Remediation function
There is no commented out block of instruction	Remove, no impact on compiled code	2 mn per occurrence
Code indentation shall follow a consistent rule	Fix with help of an IDE feature	2 mn per file, whatever the number of violations
Code shall override both equals and hashcode	Write code and associated test	1h per occurrence
All files have at least 70% code coverage	Write additional tests	20 mn per uncovered line to achieve 70%
There is no cloned parts of 100 tokens or more	Refactor with IDE and write tests	20 mn per occurrence

Remediation Function to all requirements of the Quality Model. Remediation Functions must be established and calibrated by the project or the organization. They depend mainly on the sequence of activities needed to fix a non-compliance. For example correcting a presentation defect (bad indentation, dead code) does not have the same effort cost as correcting a structural code defect (which implies the creation and execution of new unit tests, possible new integration and regression tests). Table 2 provides some examples of Remediation Functions. The complete set of Remediation Functions is a Technical Debt estimation model.

In the end, the remediation indices provide a way to measure and compare non-compliances of very different origins and very different types. Naming convention violation, threshold violations for a metric or the presence of an antipattern [13] can be compared using their relative impact on the index.

VI. THE SQALE INDICES

The most important index defined by the method is a global quality index: For any element of the hierarchy of the source code artefacts, the remediation index relating to all the characteristics of the Quality Model can be estimated by adding all remediation indices linked to all the requirements of the Quality Model. This is called the SQALE Quality Index (SQI). This represents the Technical Debt of the assessed source code. The precision of the result depends on the level of detail of the Remediation Functions established by the project or the organization.

The remediation indices and the Technical Debt can be calculated in work units, in time units or in monetary units.

There are additional indices defined in the method.

For any element of the hierarchy of the source code artefacts, the remediation cost relating to a given characteristic can be estimated by adding all remediation indices for this characteristic's requirements. The indices of SQALE characteristics are the following:

SQALE Testability Index : STI
SQALE Reliability Index : SRI
SQALE Changeability Index : SCI
SQALE Efficiency Index : SEI
SQALE Security Index : SSI
SQALE Maintainability Index : SMI
SQALE Portability Index : SPI
SQALE Reusability Index : SRuI

Also, the method defines index densities like the debt density (SQID: SQALE Quality Density Index) which allows comparison of Technical Debt for different size products.

VII. THE SQALE INDICATORS

The method defines some indicators which will be useful for analysis and for building dashboards. We will describe two of them.

A. Rating

It is a very high level indicator based on the ratio between Technical Debt and development cost. The method requires defining a Rating grid for this ratio (Fig. 6 provides an example for a 5 level rating).

This indicator suggested by Israel Gat [14] is a powerful way to visualize a complete portfolio and highlight applications at high risk.

B. SQALE Pyramid

This graph is used to visualize the distribution of the Technical Debt over the different characteristics used in the

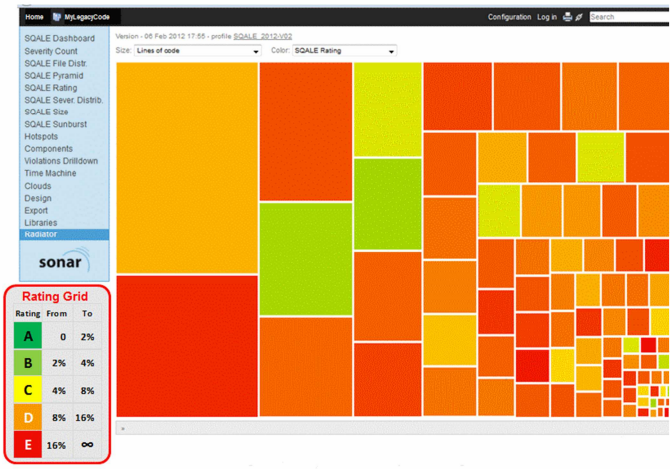

Figure 6. Rating Grid and usage sample for Portfolio Management.

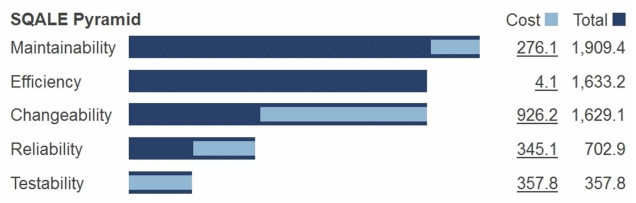

Figure 7. A SQALE pyramid generated with Sonar.

Quality Model. It also shows the consolidated indices needed to support the owner/user perspective described previously.
Fig. 7 provides an example of such a Pyramid using a 5 characteristic Quality Model.

What does this graph tell us?
It tells us that the part of the debt related to Testability is 357 days and the part of the debt related to Reliability is 345 days. From an analytic point of view 345 days will be needed to fix issues related to Reliability. This may cover issues such as insufficient test coverage, dangerous cast, wrong exception management or potential endless loop …But in fact, this is probably not enough to achieve Reliability as perceived by users. To evaluate the external Reliability of the application, the code should also be testable and some additional issues should also be fixed, for example methods that are too complex (which have too many test paths) or functions with a very large amount of parameters. So the total of remediation costs for Reliability and Testability (i.e. 702 days) is a more realistic representation of the amount of work needed to achieve a good external Reliability.
For the same reason, the consolidated Maintainability index, in this case 1,909 days is the amount of work needed to achieve a good Maintainability from an owner point of view.

The graph also provides us remediation priorities. As all other quality characteristics rely on a first layer which is Testability, the remediation activities should start by fixing issues at the bottom. If you don't follow the natural order of remediation provided by the structure of the SQALE Quality Model, you will spend hours fixing issues like memory, exception management or lack of comments within files that you will need to remove or restructure completely because they have a Testability debt. Perhaps you will need to cut some over complex method or even get rid of duplicate code (because of Copy and Paste issues) wasting the effort you spent to improve their Reliability or their Maintainability.

This graph also tells us "how agile" an application is. Out of a total Technical Debt of 1,904 days, 1,629 days (85%) are associated to Testability, Reliability and Changeability. The velocity of the project is directly impacted by these three characteristics as they are needed to support the constant stream of changes needed in an agile project.

With some experience, it is possible to make an accurate diagnostic of the "agility" of applications by looking at their Technical Debt and the associated pyramid profile.

VIII. Method Implementation

As the method is published with an open source license and is free of royalty, static analysis tool editors are now supporting the method. It is not an easy goal as the analysis tool should be able to perform verifications on the entire spectrum of requirements that may have been selected in the definition of "right code".

The list of available tools is published and updated on the SQALE web site. As of today, there are solutions for at least the following languages: COBOL, Aba, Ada, C, C++, Java, C#, PHP, Visual Basic.

Some organizations have performed their own implementation of the method and use a set of static analysis tool and a Business Intelligence database to build navigable dashboards on their application portfolio.

IX. Managing Technical Debt on a Daily Basis

In the last two years, we have coached some large organizations in their implementation and deployment of the SQALE method. Based on this field experience, we provide the following basic recommendations.

1. You need to establish rules and processes for Technical Debt management within your organization. These should address questions such as:
   - Is there a common definition of "right code" within the organization?
   - Who is allowed to add or remove requirements from this definition?
   - What is the acceptable level of Technical Debt for a new project?
   - How do we deal with the Technical Debt of all our "legacy code"?
2. Use a pilot project to develop, validate and adjust your Quality Model, calibrate your Remediation Functions to your context. Also use this period to calibrate your SQALE Rating grid. If you use the default values delivered in the available tools, all your projects might end up with the same rating, which won't help you identify priorities in your portfolio.
3. Define a standard dashboard that will be used to report, analyse and make remediation decisions on the Technical Debt of a given perimeter. This standard dashboard (see example in Fig. 8) should be used whatever the size (small application or a complete portfolio of millions of line of code). This will help communicate at all levels of your IT organization.
4. Automate as much as you can. Indicators and navigable dashboards should be updated and available for each build.
5. Focus specific attention on the deployment phase. As the Technical Debt concept will be used at all hierarchical levels, a large number of people will be impacted. You will need to organize enough awareness and coaching

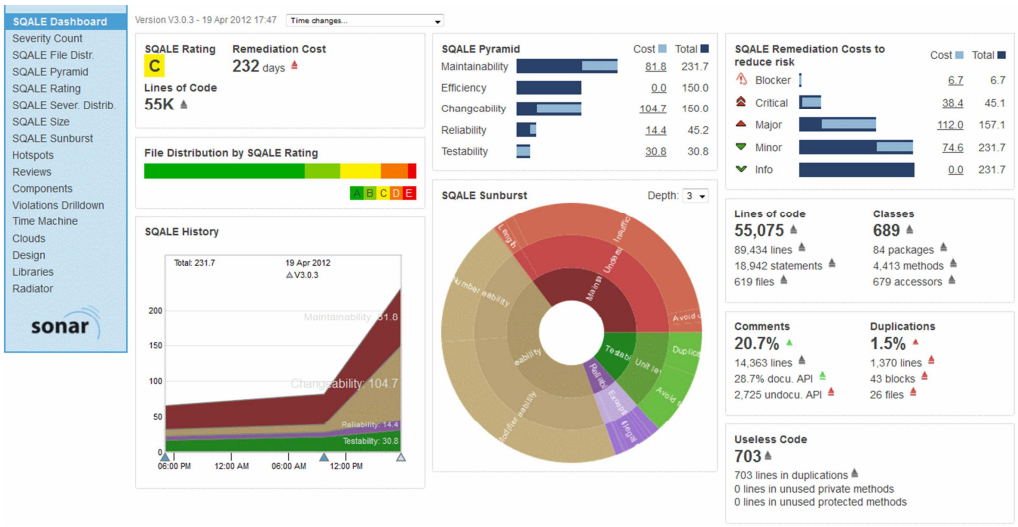

Figure 8. Example of a SQALE dashboard.

sessions, to make sure you institutionalize the technical solution, your own process and your policy for managing the Technical Debt.

X. Future Work

Developers and organizations using the method have provided feedback. In addition, a thesis [15] has been published which establishes the correlation between SQALE ratings and team perceptions. This thesis also contains valuable suggestions for improvements.

All the collected input could be summarized in two main points:

The first one is that no SQALE index takes into account the relative importance of non-conformities for business or operations. Let's take an example: a "non-initialized variable" will have a relatively low remediation cost while representing a large potential damage. In the Technical Debt it may have less impact on the figures than missing comments for an API, while the business impact of the issue may be a lot more important than missing comments.

The new version (Version 1.0) of the method will take this aspect into account and will introduce specific indices and indicators for more analysis capabilities and better priority management.

The second remark is not really a SQALE issue, but users are expecting a standardized consensus on the definition of "right code". This is work in progress with the ongoing CISQ initiative [16].

XI. Conclusion

This paper presents a formal method or a framework for estimating and analyzing the Technical Debt.

Both the Quality and the Analysis Models of the SQALE method are configurable, easy to implement and to automate, thus allowing continuous monitoring and analyzing of the Technical Debt.

References

[1] SonarSource, the Sonar company, SQALE plugin overview. http://www.sonarsource.com/plugins/plugin-sqale/overview/

[2] W. Cunningham, "The WyCash portfolio management system", ACM SIGPLAN OOPS Messenger, vol. 4(2), pp. 29–30, 1993.

[3] M. Fowler, "Technical Debt Quadrant, 2009. http://martinfowler.com/bliki/TechnicalDebtQuadrant.html

[4] S. McConnell, "What is Technical Debt? Two Basic Kinds". http://forums.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx

[5] I. Gat, "Technical Debt", IT Journal, vol. 23, October 2010.

[6] C. Sterling, "Managing Software Debt"; Addison-Wesley, 2010.

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

[8] ISO/IEC, International Standard, «ISO/IEC 15939: 2007 (E), "Systems and software engineering –Measurement process", 2007.

[9] J.-L. Letouzey, Th. Coq, "The SQALE Analysis Model - An analysis model compliant with the representation condition for assessing the Quality of Software Source Code", VALID 2010, Nice, August 2010.

[10] J.-L. Letouzey, The SQALE Method – Definition Document, Version 1.0, January 2012. http://www.sqale.org/

[11] N. Brown, M. Gonzalez, Ph. Kruchten, R. Nord, I. Ozkaya, "Managing Structural Technical Debt", OOPSLA 2011, Portland, 2011.

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

[13] W.J. Brown, R.C. Malveau, H.W. McCormick, and T.J. Mowbray, "Antipattern – Refactoring Software, Architectures and Projects in Crisis", John Wiley, New-York, 1998.

[14] I. Gat, "Revolution in Software: Using Technical Debt Techniques to Govern the Software Development Process", Agile Product and Project Management, Cutter Consortium Executive Report, Vol. 11, N°. 4, 2010.

[15] J. H. Hegeman, 'On the Quality of Quality Models", Master thesis, University of Twente, July 2011.

[16] CISQ, Consortium for IT Software Quality Consortium for IT Software Quality. http://it-cisq.org/