Practices in the Squale Quality Model (Squale Deliverable 1.3)

Françoise Balmas, Fabrice Bellingard, Simon Denier, Stéphane Ducasse, Bertrand Franchet, Jannik Laval, Karine Mordal-Manet, Philippe Vaillergues

▶ **To cite this version:**

Françoise Balmas, Fabrice Bellingard, Simon Denier, Stéphane Ducasse, Bertrand Franchet, et al.. Practices in the Squale Quality Model (Squale Deliverable 1.3). [Research Report] 2010, pp.60. inria-00533654

HAL Id: inria-00533654
https://inria.hal.science/inria-00533654

Submitted on 8 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The Squale Quality Model

Modèle enrichi d'agrégation des pratiques pour Java et C++

Workpackage: 1.3

November 8, 2010

This deliverable is available as a free download.

Copyright © 2010, 2008 by F. Balmas, F. Bellingard, S. Denier, S. Ducasse, B. Franchet, J. Laval, K. Mordal-Manet, P. Vaillergues.

The contents of this deliverable are protected under Creative Commons Attribution-Noncommercial-ShareAlike 3.0 Unported license.
You are free:

to Share — to copy, distribute and transmit the work

to Remix — to adapt the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Noncommercial. You may not use this work for commercial purposes.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page: `creativecommons.org/licenses/by-sa/3.0/`
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.

Your fair dealing and other rights are in no way affected by the above. This is a human-readable summary of the Legal Code (the full license):
`http://creativecommons.org/licenses/by-nc-sa/3.0/legalcode`

Second Edition, October, 2010.

Workpackage: 1.3

Version: 1.0

Authors: F. Balmas, F. Bellingard, S. Denier, S. Ducasse, B. Franchet, J. Laval, K. Mordal-Manet, P. Vaillergues

Planning

- Delivery Date: October 1, 2010
- First Version: July 3, 2009

Contents

Chapter1. Introduction

This document presents the Squale Software Quality Model as defined by Qualixo. It first reviews existing quality models and presents the Squale model with its particularity, namely a *practice layer*. Then it reviews in details an instance of this Squale Model with its Factors, Criteria and Practices, giving precise definitions and description[1]. Finally, it discusses possible future enhancements of this model like new practices or its agreement with the program life-cycle and the change of needs during this life cycle.

[1] The Measure layer has been described in [BBD+09]

Chapter2. State of the Art

Software quality is primarily seen as the set of processes and methods enabling to produce software without defects that fully satisfies customers[1] [ABDT04]. The NASA organization, for examples, established well structured procedures, work instructions and checklists to help ensure that every step of the whole development process is performed in the correct way[2] [EBM06].

McCall Factors Criteria Metrics (FCM).

The objective of McCall [MRW76] is to create a software quality model and to measure the level of quality in a software. FCM is composed by two layers on top of metrics: Criteria and Factors. McCall identified 50 factors and selected the 11 most important ones which should represent the external vision of the quality, as viewed by the users. These factors are characterized by 23 criteria which represent the internal vision of the quality: the programmer's point of view. These 11 quality factors have been distributed in 3 perspectives: product revision, product transition and product operations. Product revision perspective identify the ability to change the software product. Product transition identify the ability to adapt the software to new environments. Product operation perspective identify the software fulfillment with its specification.

This model is complete but very difficult to apply because of the 300 metrics needed to compute it. It is implemented in several commercial tools but the correspondence between metrics and criteria is not clearly defined as already reported by Marinescu and Ratiu [MR04]. An important weakness is the lack of connexion between a criterium and the potential problem it reflects. When a criterium has a poor mark we don't know exactly what is the cause of the problem. Even if the criterium is computed with a single metric, it does not give the solution to improve the quality. And when the criterium is computed with several metrics, it becomes very difficult to determine how to remedy to the problem. The Squale model, inspired by the ISO 9126 and the McCall models, keep the advantage of the overall view of the quality but brings a new dimension to this kind of model which allows to keep all the details: practices give information on the quality of the project that can be interpreted in order to improve this quality.

ISO 9126. ISO 9126 is an international standard for the evaluation of software quality. It is the normalization of several previous attempts. It presents a set of six general characteristics to give an overview of software quality: functionality, reliability, usability, efficiency, maintainability, portability. Each characteristic is divided in subcharacteristics to review. ISO 9126 offers a top-down look at software quality and targets end-users as well as project managers. As a consequence, not all characteristics can be reviewed automatically. Subcharacteristics such as conformance and compliance rely on laws and external standard; learnability and operability can not be assessed automatically.

The Squale model draws some inspiration from the characteristics division in ISO 9126 but targets computable characteristics. Table 2.1 shows a comparison. Squale

[1] http://www.swebok.org

[2] http://sw-assurance.gsfc.nasa.gov/disciplines/quality/index.php

is more focused and more detailed on the characteristics of a project which can be assessed from its concrete resources (code source, documentation). For example, it presents an architecture factor which is not in the ISO 9126 model. It should be noted that several concepts are different in Squale than in the ISO 9126. For example, the stability concept in ISO 9126 refers to sensitivity to system changes as a maintainability subcharacteristic, whereas in Squale it refers to runtime robustness in the reliability factor.

Goal Question Metrics (GQM). Goal Question Metrics is an approach to software quality that has been promoted by Victor Basili [BCR94]. It defines a measurement model on three levels: Conceptual level — the Goal Level—, Operational level — The Question Level — and Quantitative Level — The Metrics Level. This method aims to determine which metrics must be used to measure if the goal of the enterprise is reached. The first work determine the aims of the organization: this is the Goal Level. Then, for each objective the organization must ask the good questions related to the goal: this is the Question Level. Finally, for each question, metrics must be defined to answer the questions: this is the Metrics Level.

QMOOD. The Quality Model for Object-Oriented Design (QMOOD) model has lower-level design metrics defined in terms of design characteristics, and quality is assessed as an aggregation of the model's individual high-level quality attributes. These high-level attributes are assessed using a set of empirically identified and weighted object-oriented design properties [BD02]. QMOOD is based on ISO 9126 but has been transformed so that higher-level quality attributes always rely on computable lower-level metrics.

QMOOD involves four levels (L_1 to L_4), and three mappings (L_{12}, L_{23}, L_{34}) used to connect the four levels. While defining the levels involves identifying design quality attributes, quality carrying design properties, object-oriented design metrics, and object-oriented design components, defining the mapping involves connecting adjacent levels by relating a lower level to the next higher level.

Factor-Strategy. Marinescu and Ratiu [MR04] raised the following question *How should we deal with measurement results?* After pinpointing a few limitations in Factor-Criteria-Metric models (e.g., obscure mapping of quality criteria onto metrics, poor capacity to map quality problems to causes), they introduce detection strategies as a generic mechanism for analyzing a source code model using metrics. The use of metrics in the detection strategies is based on mechanisms for filtering and composition. A filtering operation is characterized with thresholds and extremities. Composition operators are and, or, butnotin.

Based on the detection strategy mechanism, a new quality model is proposed, called *Factor-Strategy*. This model uses a decompositional approach, but after decomposing quality in factors, these factors are not anymore associated directly with metrics numbers. Instead, quality factors are now expressed and evaluated in terms of detection strategies, which are the quantified expressions of the good-style design rules for the object-oriented paradigm.

Each factor or strategy receives a score, which is computed with the help of a matrix of ranks: given a raw data or score, the matrix will give a normalized quality score to be used in other formula. However, the discrete nature of the matrix implies that this

approach is still sensitive to staircase effects.

Assessment Methodologies for free/open source software have started to emerge: OSMM, OpenBBR, QSOS, QUALOSS. Those methodologies are based on models such as ISO 9126 and deal with the specificity of free/open source projects and as such broaden the scope of their model to include community-related attributes.

The Pyramid

The overview Pyramid has been proposed in [LM06]. It propose a pyramid composed with 3 aspects : size and complexity, coupling, inheritance.

The Figure 2.1 show a screenshot of a pyramid generated by iPlasma, a software which import java code and C++ code. This screenshot is the import of source code of ejb3, for the example.

The Pyramid is composed of three parts: the size and complexity aspect in yellow, the coupling aspect in purple and the inheritance aspect in green.

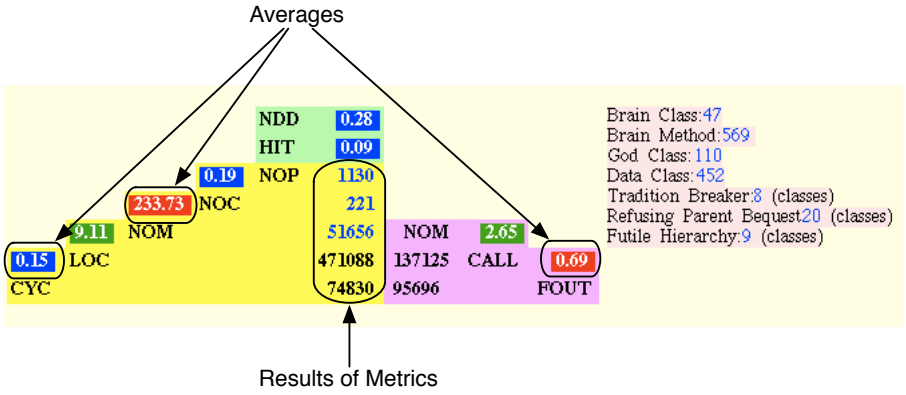


Figure 2.1: The overview pyramid.

The size and complexity aspect (in yellow) shows three kinds of information:

- the text is the name of metrics.
  - CYC: Cyclomatic complexity
  - LOC: Lines of Code
  - NOM: Number of methods
  - NOC: Number of classes
  - NOP: Number of packages
- numbers in the center of the pyramid represent results of these metrics. For example the number 74830 in the Figure 2.1 represent the cyclomatic complexity.

ISO 9126	Squale	Factor	Functionality			Architecture			Maintainability				Evolutivity				Reuse Capacity			Reliability			
Features	Criteria		Task Aptitude	Modeling	Acceptance Tests	Architecture Relevance	Architecture Modularity	Architecture Respect	Homogeneity	Comprehension	Simplicity	Integration Capacity	Homogeneity	Comprehension	Modeling	Modularity	Comprehension	Exploitability	Integration Capacity	Technical Tests	Stability	Simplicity	Technical Tests	Security
	Sub-characteristics	1	0	1	0	0	0	0	1	1	2	0	1	0	1	1	3	1	1	4	1	1	1
Functionality	Suitability	2	Z	Z																			
	Accuracy	0																					
	Interoperability	1															Z						
	Functionality compliance	0																					
	Security	1																					X
Reliability	Maturity	0																		Z			
	Fault Tolerance	1																		Z			
	Recoverability	1																		Z			
	Reliability compliance	0																					
Usability	Understandability	3								X				X			X						
	Learnability	0																					
	Operability	0																					
	Attractiveness	0																					
	Usability Compliance	0																					
Efficiency	Time behavior	1																		Z			
	Resource utilization	1																		Z			
	Efficiency compliance	0																		Z			
Maintainability	Analyzability	2									Z										Z		
	Changeability	0																					
	Stability	1										Z											
	Testability	2																	Z			Z	
	Maintainability compliance	0																	Z			Z	
Portability	Adaptability	1															Z						
	Installability	1															Z						
	Portability compliance	0																					
	Co-Existence	0																					
	Replacebility	3										Z			Z			Z					

Table 2.1: ISO 9126 vs Squale Comparison: X shows perfect match, Z partial match.

- numbers in the left of the pyramid represent averages between the metric at its right and metric at its bottom. For example the number 0.15 is the average of cyclomatic complexity by line of code. From left to right, they represent Intrinsic operation complexity (CYCLO/LOC), Operation structuring (LOC/NOM), Class structuring (NOM/NOC) and high level structuring (NOC/NOP).

The coupling aspect (in purple) has the same structure

- the text is the name of metrics.
  - NOM: Number of methods
  - CALL: number of operation calls
  - FOUT: Fan out, number of called classes
- numbers in the center of the pyramid represent results of these metrics. For example the number 95696 in the Figure 2.1 represent the Fan out.
- numbers in the right of the pyramid represent averages between the metric at its left and metric at its bottom. For example the number 0.69 is the average of FanOut by Call. From left to right, they represent Coupling intensity (CALLS/NOM) and Coupling dispersion (FOUT/CALL)

The inheritance aspect (in green) give information about the average of Height of the inheritance tree (HIT) and the average of children (NDD).

Each average have a color: red means means high, green means normal and blue means low.

As this last example shows, there is a clear need to put metric values in perspective in order to make them easily understandable. For this reason as next Section will show, the Squale Model introduces the concept of *practices*, which proposes aggregated metric values in such a way that developers or managers can know what to do to enhance code quality.

Chapter3. The Squale quality model

The Squale model is inspired by the factors-criteria-metrics model (FCM) of McCall [MRW76]. However, while McCall defined a top-down model to express the quality of a system, the Squale model promotes a bottom-up approach, aggregating low-level measures into more abstract quality elements. This approach ensures that the computation of top-level quality assessments is always grounded by concrete repeatable measures or audit on actual project components.

The Squale model introduces the new level of *practices* between criteria and metrics. Practices are the key elements which bridge the gap between the low-level measures, based on metrics, rule checkers or human audits, and the top-level quality assessments —expressed through criteria and factors. Thus the Squale model is composed of four levels (see Figure 3.1): factors, criteria, practices, and measures.

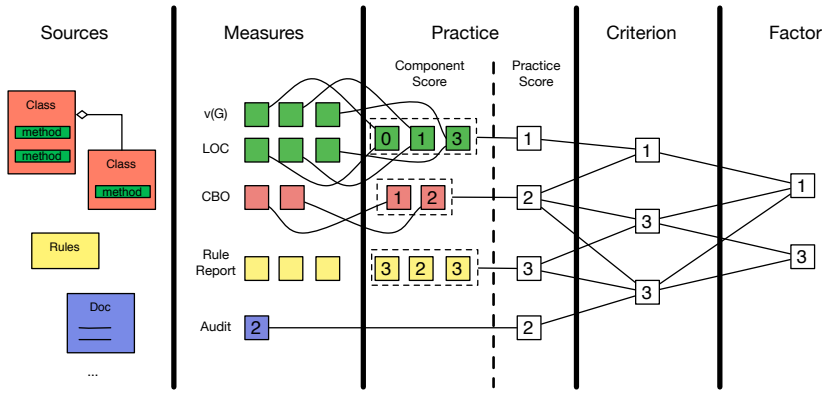


Figure 3.1: Data sources and levels of the Squale model.

The three top levels of Squale use the standard mark system defined by the ISO 9126 standard. All quality marks take their value in the range $[0; 3]$, as shown in Figure 3.1, to support an uniform interpretation and comparison:

- between 0 and 1, the goal is not achieved;
- between 1 and 2, the goal is achieved but with some reservations;
- between 2 and 3, the goal is achieved.

3.1 The four levels of the Squale model

The following subsections briefly present the four levels of the Squale model, from the bottom measures to the top factors. Figure 3.2 shows how measures are aggregated into practices, then into criteria and factors. Section 3.2 explains how marks are computed in the different levels. Exact formula to compute measure aggregation into practices are given in Section 4.3.

3.1.1 Measures

A *measure* is a raw information extracted from the project data.

The Squale model takes into account different kinds of measure to assess the quality of a software project: automatically computable measures that can be computed easily and as often as needed, and manual measures which have a predefined life time and must be updated mainly after major changes to the software.

The automatically computable measures are divided into three groups. The first group is composed of metrics [FP96, Mar97, BDW98] like Number of Lines of Code [CK94], Depth of Inheritance Tree [LK94], or cyclomatic complexity [McC76]. A preliminary analysis selected only relevant metrics [BBD+09][1]. However, Squale is able to adapt to a wide range of metrics provided by external tools. The second group is composed of rules checking analysis like syntactic rules or naming rules, which verify that programming conventions are enforced in the source code and allow one to correct some bugs. These rules are defined before starting the project and must be known by developers. The third group is composed of measures which qualify the quality of tests applied to the project such as test coverage. This group may also contain security vulnerability analysis results.

The manual measures express the analysis made by human expertise during audits. These measures qualify the documentation needed for a project, such as specification documents or quality assurance plan. They verify also that the implementation of the project respects the documented constraints.

A measure is computed with respect to its scoping entity in the project data: method, class, package, or the project itself for an object-oriented software.

Between 50 and 200 different measures are used in various instances of the Squale model. Usable measures depend on the available tools, the current stage in the project life-cycle, and the requirements of the company.

3.1.2 Practices

A *practice* assesses the respect of a technical principle in the project (such as *complex classes should be more documented than trivial ones*). It is directly addressed to the developer in terms of good or bad property with respect to the project quality. Good practices should be fulfilled while bad practices should be avoided. The overall set of practices expresses rules to achieve optimum software quality from a developer's point of view. Around 50 practices have been defined based on Air France quality standards. However, the list of practices is not closed and such practices can be adjusted.

A practice combines and weights different measures to assess the fulfillment of technical principles. A practice mark is computed for an individual element of the source code. A global mark for the practice adjusts the variations of the individual marks. We detail this aspect in Section 3.2.1.

For example, the *comment rate* practice combines the *comment rate per method LOC* and *cyclomatic complexity* of a method to relate the number of comments in the source code with the complexity of the method: the more complex the method, the

[1] http://www.squale.org/quality-models-site/

more comments it should have.

3.1.3 Criteria

A *criterion* assesses one principle of software quality (*safety, simplicity*, or *modularity* for example). It is addressed to managers as a detailed level to understand more finely project quality. The criteria used in the Squale model are adapted to face the special needs of Air France and PSA. In particular, they are tailored for the assessment of quality in *information systems*.

A criterion aggregates a set of practices. A criterion mark is computed as the weighted average of the composed practice marks. Currently around 15 criteria are defined.

For example, the following practices:

- Inheritance depth
- Documentation standard (rule checking of documenting conventions)
- Documentation quality (human audit with respect to project requirements)
- Class specialization
- Source comments rate (per method with respect to cyclomatic complexity)

define the *comprehension* criterion.

3.1.4 Factors

A *factor* represents the highest quality assessment to provide an overview of project health (*Functional capacity* or *reliability* for example). It is addressed to non-technical persons. A factor aggregates a set of criteria. A factor mark is computed as the weighted average of the composed criteria marks.

The six factors used in the Squale model are inspired by the ISO 9126 factors and refined based on the experience and needs of engineers from PSA, Air France, and Qualixo.

For example, the following criteria :

- Comprehension
- Homogeneity
- Integration Capacity
- Simplicity

define the *maintainability* factor. This means that a system should be easier to correct when it is homogeneous (respect of architectural layers and of programming conventions), simple to understand and modify (good documentation, manageable size), and conveniently coupled.

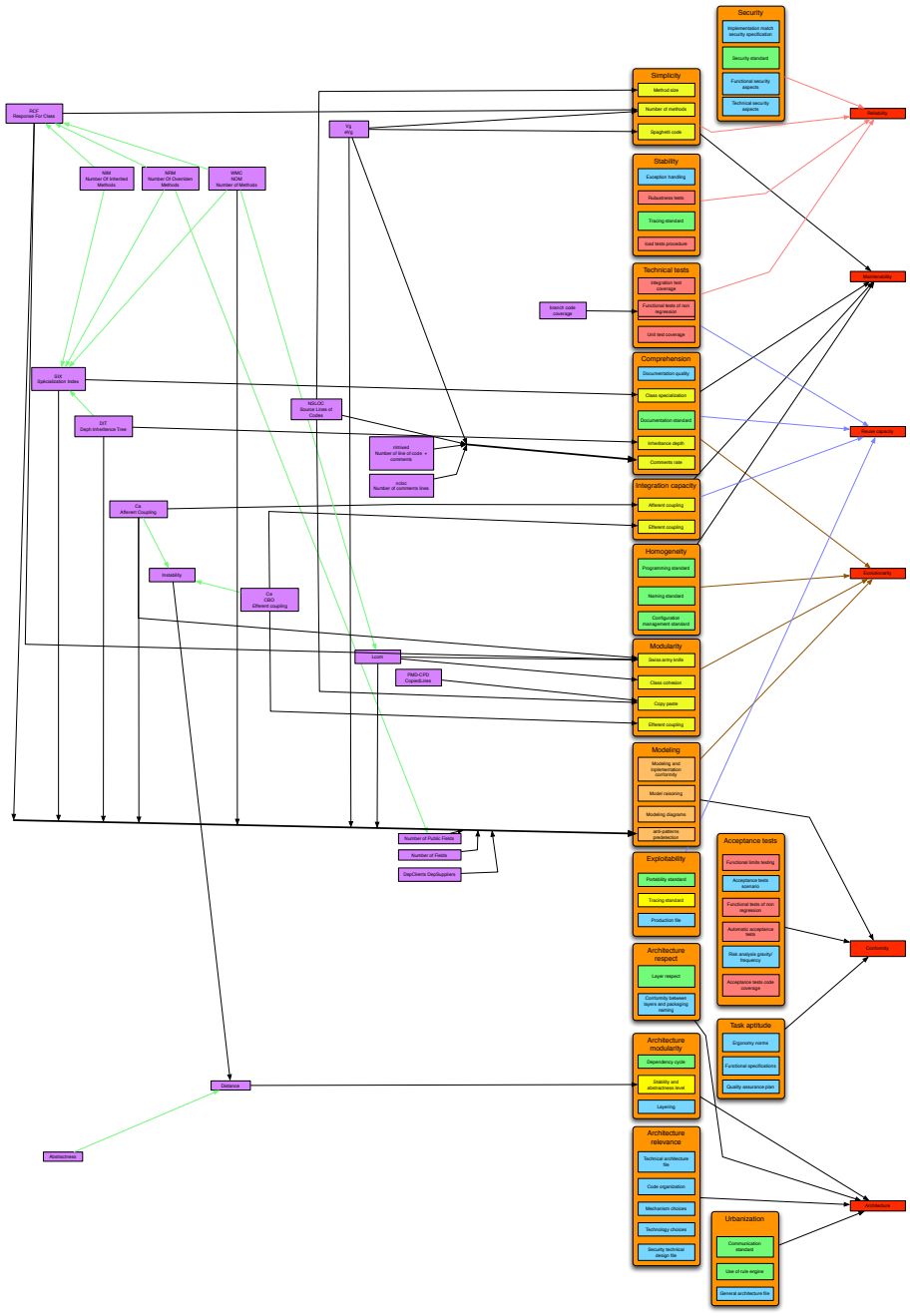


Figure 3.2: The Four Level Squale Model

3.2 The marks of the Squale model

This Section shows how marks are computed a the different levels.

3.2.1 Practice marks

Practice global marks for a project are computed, in most cases, in two steps:

Individual mark Each element (method, class, or package in object-oriented programs) targeted by a practice is given a mark with respect to its measures. For example, the two metrics composing the *comment rate* practice, *cyclomatic complexity* and *source line of code*, are defined at the method level; thus a *comment rate* mark is computed for each method.

Global mark A global mark for the practice is computed using a weighted average of the previous individual marks.

Note that when the scope of the measures is the project itself, the global mark can be directly computed.

The different formulae also normalize practice marks to enable comparison between practices on a common scale.

Individual mark

Two kinds of formulae exist for computing individual marks, namely discrete and continuous formulae. An individual mark is computed from measures in multiple ranges into a single mark in the range $[0; 3]$.

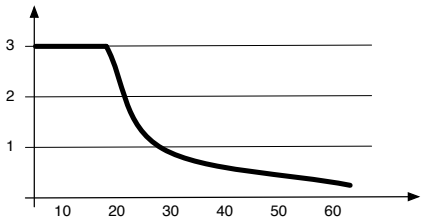


Figure 3.3: Sample graph for a practice mark based on one measure.

A discrete marking system is simple to implement and easy to read. It is well adapted to manual measures such as audits. For example, the practice for *functional specifications* is given a mark in a discrete range. If there is no functional specification, the mark 0 is given. If functional specifications are consistent with the client requirements, the mark 3 is given. The two intermediate marks are used to qualify existing yet incorrect functional specifications. Thus this mark assesses two information: the existence of functional specifications and their consistency. While the practice can only be evaluated by an expert, the discrete range limits the subjectivity of the given mark.

Discrete marking is not adapted to all practices. For metrics-based practices, the discrete formula introduces staircase values and threshold effects, which smoothes detailed information and triggers wrong interpretation. When surveying the evolution of quality, it hides slight fluctuations—progression or regression—of an individual element.

A continuous formula is used to avoid this phenomenon when it is possible. It better translates the variations of metric values on the mark scale. Indeed, such formulae are first built around a couple of measure-mark binding, agreed upon by the experts. Then, the formula is defined as a linear or non-linear equation which best approximate those special values and allows one to interpolate marks for any value.

Figure 3.3 shows an example using a continuous equation of correspondence between a single measure (x axis) and its given mark (y axis). First there is a threshold of 20 below which the mark is automatically 3 (the continuous equation is clipped). It is the maximal value which allows one to achieve the goal. Above this threshold, the individual mark decreases following an exponential curve: the individual mark tends quickly towards zero.

Global mark

The global practice mark is obtained from the individual marks through a weighted average. The weighting function allows one to adjust individual marks for the given practice in order to stress or loosen tolerance for bad marks:

- a hard weighting is applied when there is a really low tolerance for bad individual marks in this practice. It accentuates the effect of poor marks in the computation of the practice mark. The global mark falls in the range $[0; 1]$ as soon there is a few low individual marks.
- a medium weighting is applied when there is a medium tolerance for bad individual marks. The global mark falls in the range $[0; 1]$ only when there is an average number of low individual marks.
- a soft weighting is applied when there is a large tolerance for bad individual marks. The global mark falls in the range $[0; 1]$ only when there is a large number of low individual marks.

Weighting is chosen to highlight critical practices: hard weighting leads to a low practice mark much faster than soft weighting.

The computation of the practice mark is a two-step process. First a weighting function is applied to each individual mark:

$$g(IM) = \lambda^{-IM}$$

where IM is the individual mark and λ the constant defining the hard, medium, or soft weighting, λ being greater for a hard weighting and smaller for a soft one. This formula translates individual marks into a new space where low marks have significantly more weight than others. The average of the weighted marks will reflect the more important weight of the low marks. Then the inverse function:

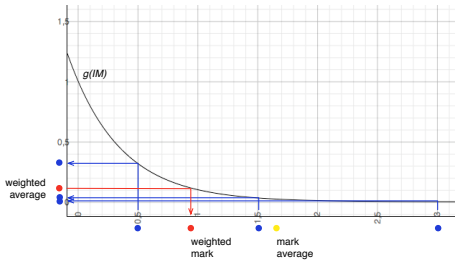


Figure 3.4: Principle of weighting: individual marks are lowered when translated in the weighted space.

$$g^{-1}(averageWeightedIMs) = -log_{\lambda}(averageWeightedIMs)$$

is applied on the average to come back in the range $[0; 3]$.

Thus the global mark for a practice is:

$$mark = -log_{\lambda}\left(\frac{\sum_1^n \lambda^{-IM_n}}{n}\right)$$

where λ varies to give a hard, medium, or soft weighting.

Figure 3.4 illustrates how the $g(IM)$ function and its inverse work to reflect low individual marks in the practice mark. Here, $\lambda = 9$, which is a medium weighting. There are three individual marks (blue dots on the x axis) at 0.5, 1.5, and 3. This series gives a normal average around 1.67 (yellow dot). Instead, the marks are translated in the weighted space (blue arrows) where the 0.5 mark is significantly higher than the two other marks. The weighted average (red dot on y axis) is then translated back in the mark range(red arrow) with the value of 0.93. The lower weighted mark for the practice, compared to the normal average, is a clear indication that something is wrong, despite the high mark of 3.

3.2.2 Criterion marks

Marks for criteria are computed from the corresponding practice marks as a standard weighted average. This means that each practice can be given a weight that reflects its importance inside a criterion. For example, in the criterion *comprehension*, practices *documentation standard*, *class specialization* and *inheritance depth* can be given the weight 1, and practices *source comments rate* and *documentation quality* the weight 2, because the latter are recognized as more accurate, reliable and valuable for the criterion than the former.

Note that in the current Squale model, all practices are weigthed with 1, meaning that criterion marks are just a simple average of the different practice marks. However, the squale tool was designed to allow practice weighting.

3.2.3 Factor marks

Marks for factors are computed from the corresponding criterion marks with the same standard weighted average than criterion marks (see above Section 3.2.2).

Chapter4. Current Squale Model Instance

The Squale project allowed to develop a first Squale Model Instance. This Instance have been made for Air France-KLM, PSA Peugeot-Citroen, so it is tailored to information systems.

This chapter describe this particular Model with its factors, criteria and practices.

4.1 Factors analysis

We describe the six factors implemented in this Squale Model instance. Three factors are the same as in the ISO 9126 model and the other have been customized to better match the enterprise requirements.

Functionality: Qualification of the adequacy between software functionality and customer needs. This factor is decomposed into the following criteria:

- Acceptance Tests
- Modeling
- Task Aptitude

Reliability: Capability of the software product to maintain a specified level of performance, stability and services when used under specified conditions and during a given period. This factor is decomposed into the following criteria:

- Security
- Simplicity
- Stability
- Technical Tests

Maintainability: Capability of software to assist in locating and correction of bugs. This includes only corrections of defects because of non-compliance with the specifications. This factor is decomposed into the following criteria:

- Comprehension
- Homogeneity
- Integration Capacity
- Simplicity

Architecture: Quality level of the technical architecture project, not only because of the choices of the technical architecture but also to qualify the respect of this architecture. This factor is decomposed into the following criteria:

- Architecture Modularity
- Architecture Relevance

- Architecture Respect

Evolutivity: Facility to add new functionality of the software. This factor is the complement of the maintainability Factor but it is no more constant functional perimeter. This factor is decomposed into the following criteria:

- Comprehension
- Homogeneity
- Modeling
- Modularity

Reuse Capacity: Facility to reuse all or part of the software in another project, in any technical or functional environment. This factor is decomposed into the following criteria:

- Comprehension
- Exploitability
- Integration Capacity
- Technical Tests

4.2 Criteria analysis

We describe the 15 criteria defined in this squale model instance. The figure 4.1 represent the criteria dispatching into the six factors describe in the precedent section.

Acceptance tests: Quality assessment for acceptance tests scenarios and their results. This criterion is decomposed into the following practices:

- Functional tests of non regression
- Functional limits testing
- Acceptance tests code coverage
- Risk analysis gravity/frequency
- Automatic acceptance tests
- Acceptance tests scenario

Architecture Modularity: Describes the relevance of architectural choices to obtain a good decomposition of the project in the perspective of project evolution and reuse. This criterion is decomposed into the following practices:

- Dependency cycle
- Layering
- Stability and abstractness level

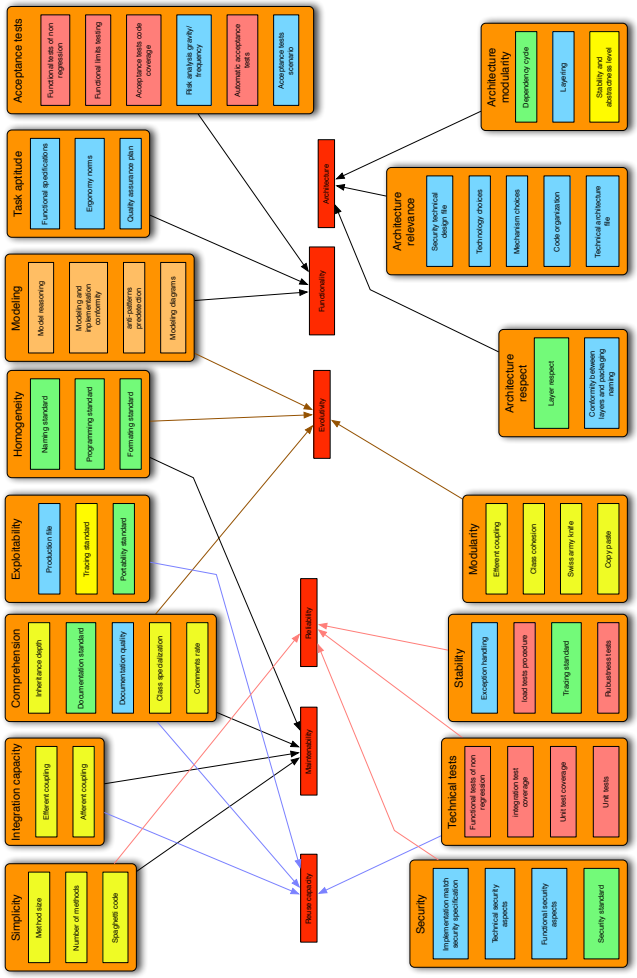


Figure 4.1: Criteria, Factors and Practices in the Squale Model

Architecture Relevance: Qualify the architectural choices and their particular relevance with respect to the technologies chosen, through documents and the implementation of given principles. This criterion is decomposed into the following practices:

- Security technical design file
- Technology choices
- Mechanism choices
- Code organization
- Technical architecture file

Architecture Respect: Qualify the conformity level between the architectural choices and their application. This criterion is decomposed into the following practices:

- Layer respect
- Conformity between layers and packaging naming

Comprehension: The facility for a developer to quickly understand the overall logic of the source code through technical documentation and javadoc. This criterion is decomposed into the following practices:

- Inheritance depth
- Documentation standard
- Documentation quality
- Class specialization
- Source comments rate

Exploitability: Describes the packaging, the deployment ease and the portability in another system for the software product. This criterion describes the level of exploitability for the software through documentation and respect of production norms. This criterion is decomposed into the following practices:

- Production file
- Tracing file
- Portability standard

Homogeneity: Qualify the homogeneity of the source code and how this code is documented. This criterion describes the capability of the software product to adhere to standards or conventions relating to maintainability. This criterion is decomposed into the following practices:

- Naming standard
- Programming standard
- Formatting standard

Integration capacity: Qualify the level of dependency between classes inside a project. Too high dependency level between elements results in high coupling: any change in one place may have further unintended consequences. This criterion is decomposed into the following practices:

- Efferent coupling
- Afferent coupling

Modeling: Quality assessment of the modeling project and conformity between the model and the software. It ensures the control and use of modeling techniques for the project. This criterion is decomposed into the following practices:

- Model reasoning
- Modeling and implementation conformity
- Anti-patterns predetection
- Modeling diagrams

Modularity: Describes the software decomposition in multiple limited size elements. This criterion is decomposed into the following practices:

- Efferent coupling
- Class cohesion
- Swiss army knife
- Copy paste

Security: Qualify the security level of the application for the data level and the source code vulnerability. This criterion is decomposed into the following practices:

- Implementation match security specification
- Technical security aspects
- Functional security aspects
- Security standard

Simplicity: Assessment of source code readability and ease to diagnostic regardless documentation. This criterion is decomposed into the following practices:

- Method size
- Number of methods
- Spaghetti code

Stability: Qualify the capability of the software to respond to potential failures. This criterion is decomposed into the following practices:

- Exception handling
- Load tests procedure

- Tracing standard
- Robustness tests

Task Aptitude: The capability of the software product to provide the right or agreed results or effects with the needed degree of precision. This criterion is decomposed into the following practices:

- Functional specifications
- Ergonomic norms
- Quality assurance plan

Technical Tests: Qualify the quality of test production and their results analysis. This criterion is decomposed into the following practices:

- Functional tests of non regression
- Integration test coverage
- Unit test coverage
- Unit tests

4.3 Practice analysis

To take into account all aspects of the quality of a project, different kinds of analyses must be made. Practices which compose the Squale model come from the following analyses:

- metric analysis based on computed metrics.
- model analysis based on model analysis.
- rules checking analysis based on programming rules.
- human analysis based on human expertise.
- dynamic analysis based on tests analysis.

For almost all practices, the following formula is used to compute a weighted global practice mark as this is explained in the section 3.2.1:

$$mark = -log_{\lambda}\left(\frac{\sum_{1}^{n}\lambda^{-IM_n}}{n}\right)$$

With :

IM for Individual Mark

λ for the constant defining the hard, medium or soft weighting.

In the practice description, this formula is called *Weighted global average*. When the formulae used to compute global mark is different, we explicitly describe it in the practice description.

As mentioned in 3.2.1, practices which have a project scope are given only global marks.

4.3.1 Practices derived from metrics

These practices are calculated with the metrics which have been discussed and tested in deliverable 1.1. They qualify the quality of code and design with metrics. For each measure, the result is aggregated to calculate an overall mark assigned to a practice.

Name **Inheritance Depth**

Criteria **Comprehension**

Scope class

Metrics DIT (depth inheritance tree)

Definition Highlight classes with a high inheritance depth. A class with a too hight inheritance depth is much more difficult to maintain and to understand. The threshold is set to take into account the capability of an human to easily understand the source code and the different calls made into this code. It doesn't take into account the framework inheritance depth: a class can be a subclass in a framework, hence has a large DIT but in the context of the application a small DIT value.

Mark Individual mark:

- 0 if $dit > 7$
- 1 if $7 \geq dit > 6$
- 2 if $6 \geq dit > 5$
- 3 if $dit \leq 5$

Global practice mark: weighted global average.

Weight soft

Name **Source comments rate**

Criteria **Comprehension**

Scope methods

Metrics NCLOC (number of lines of comments), SLOC (sources lines of code), v(G) (cyclomatic complexity)

Definition Qualifies the comment rate in the lines of code. The Javadoc is not included to compute this practice. This practice verify that the more complex a method is the more comments it has. The number of comments line depends not only to the method number of lines of code but also to its complexity. The appropriate

threshold depends to the complexity of the method. Mark are not computed if metrics are below some threshold to filter out methods like encapsulation methods(getters and setters in java). The main idea is to verify that methods include at least 30% of comments.

Mark Individual mark:

If $v(G) >= 5$ or $sloc > 30$:

then :

$IM = (ncloc) * 9/(ncloc + sloc)/(1 - 10^{(-v(G)/15)})$

Main values for individual marks:

Practice value	Metric v(G) values	Comment rate value
< 0.5	>= 5	1%
< 1	>= 5	5%
>= 1 and <= 2	<= 15	10%
< 1	> 15	
3	<= 14	30%
2.8 or 2.9	<= 26	
2.7	>= 27	
3	any	50%

Global practice mark: weighted global average.

Weight soft

Name **Number of methods**

Criteria **Simplicity**

Scope class

Metrics V(g) (cyclomatic complexity), NOM (number of local methods)

Definition Qualifies the number of methods for each class of project. This practice detects the classes with too much methods. Theses classes centralize too much functionalities and may be difficult to maintain or modify.

Mark Individual mark:

- if $\sum v(G) \geq 80$ then $IM = 2^{(30-NOM)/10}$
- if $\sum v(G) \geq 50$ and $NOM \geq 15$ then $IM = 2 + (20 - NOM)/30$
- if $\sum v(G) \geq 30$ then $IM = 3 + (15 - NOM)/15$

– else $IM = 3$

Main values for individual marks:

Practice value	Metric nom value	Sum of v(G) values
3	<= 14	>= 80
2	20	
1	30	
0	>= 74	
2.2	15	>= 50
1.5	35	
1	50	
0	>= 79	
3	<= 15	>= 30
2	30	
1	45	
0	>= 60	

Global practice mark: weighted global average.

Weight medium

Name **Method Size**

Criteria **Simplicity**

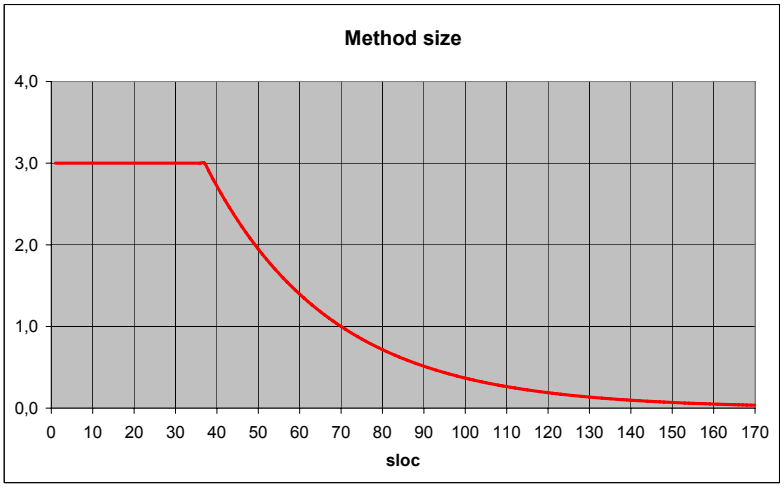
Scope method

Metrics SLOC (number of source lines of code)

Definition Computes the method size to highlight methods which are too long, hence too difficult to maintain and understand. In the formula below, the threshold is at 70 SLOC.

Mark Individual mark:

$IM = 2^{(70-sloc)/21}$



Main values for individual marks:

Pratice value	Metric sloc value
3	<= 37
2.5	42
2	49
1.5	58
1	70
0.5	91
0	>= 162

Global practice mark: weighted global average.

Weight medium

Name **Swiss army knife**

Criteria **Modularity**

Scope class

Metrics LCOM2 (lack of cohesion in methods), Ca (afferent coupling), NOM (number of local methods)

Definition This practice searches for the "utility" classes which are often very difficult to maintain. These classes are generally without child or parent with few attributes but very many methods. They are called for too many different cases. The practice mark should be improved because it only detect utility classes without qualifying classes which are not full utility classes but neither not utility classes at all.

Mark Individual mark:

- 0 if $ca > 20$ and $lcom2 > 50$ and $rfc > 30$
- 3 if $ca \leq 20$ or $lcom2 \leq 50$ or $rfc \leq 30$

Global practice mark: weighted global average.

Weight soft

Name **Class cohesion**

Criteria **modularity**

Scope class

Metrics LCOM2 (lack of cohesion in methods)

Definition Qualifies the relations between methods in a class. The metric used to compute this practice is not really adequate but this practice is really important to compute in given used technologies. This practice should be improved, in particular the metric should be replaced by LCOM5 and the formula adapted to this new metric.

Mark Individual mark:

- 0 if $lcom2 > 100$
- 1 if $lcom2 > 50$
- 2 if $lcom2 > 0$
- 3 if $lcom2 \leq 0$

Global practice mark: weighted global average.

Weight soft

Name **Efferent Coupling**

Criteria **Modularity, integration capacity**

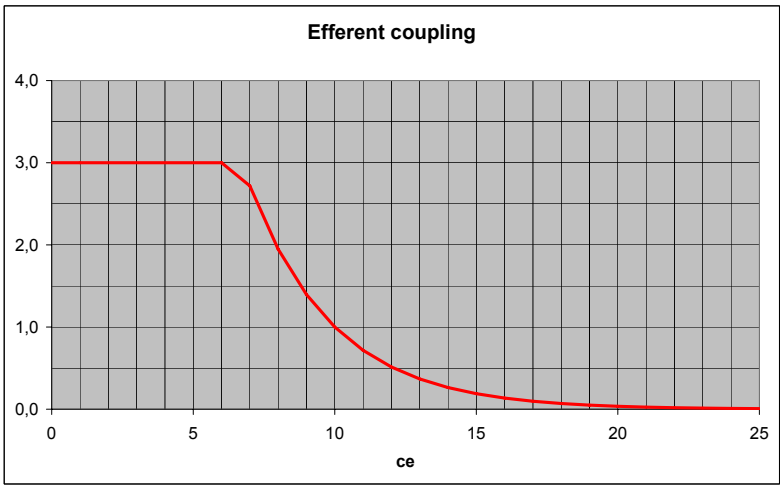
Scope class

Metrics Ce (efferent coupling)

Definition Qualifies the efferent coupling for a class and analyzes the dependence between one class and the other classes as well as the public data of the project. A class which uses many other classes should be potentially more affected by any other class modification. This practice highlights the most dependent classes.

Mark Individual mark:

$IM = 2^{(10-ce)/2}$



Main values for individual marks:

Practice value	Metric ce value
3	<= 6
2.8	7
2	8
1.4	9
1	10
0.5	12
0	>= 19

Global practice mark: weighted global average.

Weight soft

Name **Afferent Coupling**

Criteria **Integration Capacity**

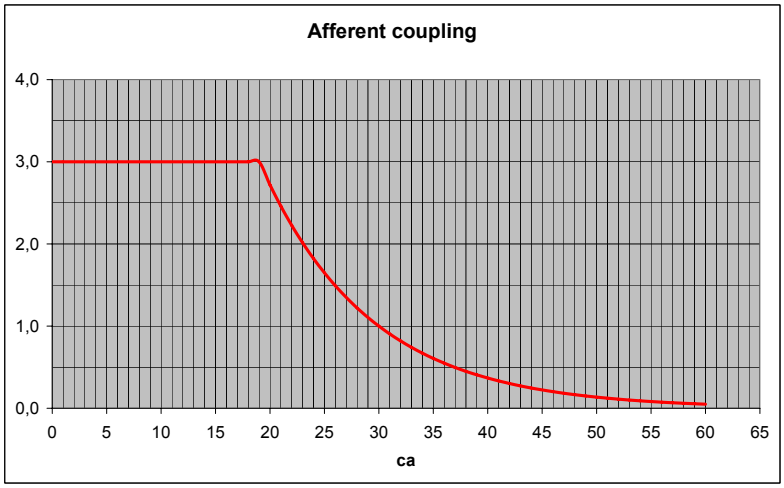
Scope class

Metrics Ca (afferent coupling)

Definition This practice complements the *efferent coupling* practice. It analyzes the dependences between all the classes and one studied class : it is the number of classes which depend on the studied class. This practice highlights the classes which have too much responsibilities in the project. Any modification in this kind of class may have an important impact in the whole project. This practice is particular to interpret: classes should be called, but the more they are called, the less they can change without consequences.

Mark Individual mark:

$IM = 2^{(30-ca)/7}$



Main values for individual marks:

Practice value	Metric ca value
3	<= 19
2.7	20
2	23
1.5	26
1	30
0.5	37
0	>= 60

Global practice mark: weighted global average.

Weight medium

Name **Spaghetti Code**

Criteria **Simplicity**

Scope method

Metrics ev(G) (cyclomatic complexity)

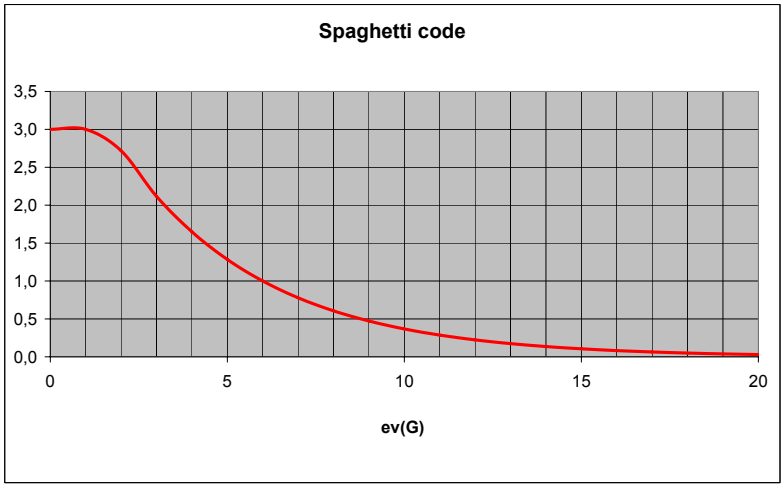
Definition Qualifies the complexity and the structure of the code in order to highlight those parts of the code which are particularly complex, distorted. This practice is associated with SLOC to eliminate the short methods from the scope of investigation.

Mark Individual mark:

If $sloc > 30$:

then :

$IM = 2^{(6-ev(g))/3}$



Main values for individual marks:

Practice value	Metric ev(G) value
3	<= 1
2.5	2
2	3
1.5	4
1	6
0.5	9
0	>= 20

Global practice mark: weighted global average.

Weight medium

Name **Copy Paste**

Criteria **Modularity**

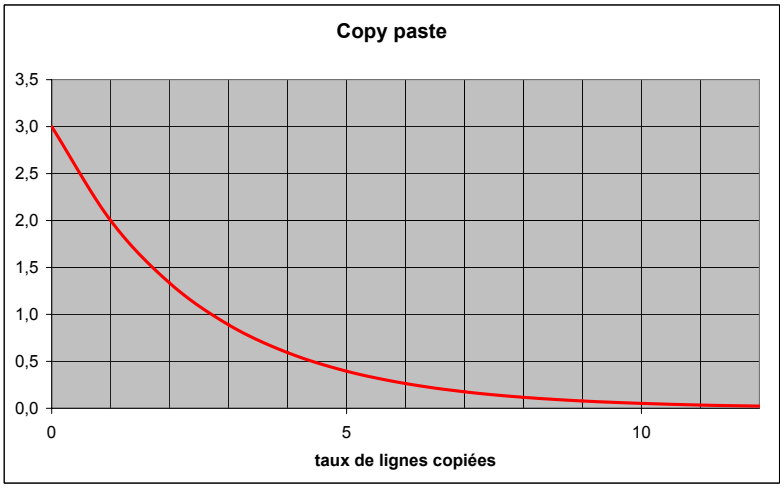
Scope project

Metrics SLOC (number of source lines of code)

Definition This practice highlights the lines of code which are duplicated. This practice heavily depends on the tools used to detect duplicated code, which give all somewhat different results. So the formula given to compute the practice mark must be adapted for each tool results. In any case, it should be severe to detect any duplicated blocs of lines of code : in object oriented in particular, there should never be any copied lines of code. Instead of this, developer should use inheritance. The tool currently used in Squale to compute this practice identifies copied blocks of code copied by searching for blocks of min. 100 tokens. It returns the percentage of copied lines.

Mark Global practice mark:

$$mark = 3 * \frac{2}{3}^{\frac{100 * Number\_of\_copiedlines}{sloc}}$$



Main values :

Practice value	taux de lignes copiees
3	0
2	1
1.3	2
0.9	3
0.4	5
0	>= 11

Weight soft

Name **Stability and abstractness level**

Criteria **Architecture Modularity**

Scope package

Metrics Distance (Abstractness and instability distance)

Definition Determines the respect of the separation between interface and implementation. An abstract package must have a poor efferent coupling while a concrete package must have a poor afferent coupling to ensure this separation.

Mark Individual mark:

$IM = 3 + 2 \times \frac{25 - Distance}{25}$

Main values for individual marks:

Practice value	Metric distance value
3	<= 25
2.5	31
2	38
1.5	55
1	50
0.5	56
0	>= 62

Global practice mark: weighted global average.

Weight soft

Name **Class specialization**

Criteria **comprehension**

Scope class

Metrics SIX (specialization index)

Definition Qualifies the class specialization to highlight a potentially inappropriate use of inheritance. As it is mentioned in [BBD+09], the metric SIX is unreliable, so this practice should be deprecated.

Mark Individual mark:

- 0 if SIX >= 0,5

- 3 if SIX < 0,5

Global practice mark: weighted global average.

Weight soft

Name **Dependency cycle**

Criteria **Architecture Modularity**

Scope package

Metrics jdepend.cycle

Definition This practice detects the package cycles to highlight a bad packaging or a poor design. This practice, based on the JDepend metric tool, should be improved to compute a quality mark in addition to make a simple cycle detection: JDepend detects only cyclic dependency in the import graph. Also the scope is not relevant to detect cycle, it should be adapted to the granularity of modifications. This practice is interesting although it is perfectible.

Mark The individual mark is calculated with a rate of transgressions:

Individual mark :

- 0 if there is a cycle
- 3 if there is no cycle

Global practice mark: simple global average of the different individual marks.

Weight soft

4.3.2 Practices from models

The objective of the Squale Model is to qualify the project as soon as possible and to help developers to improve the quality of their project. So when an U.M.L. model is made, the Squale quality model helps analyzing the relevance of the modeling project. This allows to detect as soon as possible a wrong design, even before the implementation of the project. Theses analyses verify also if the implementation matches the modeling project.

Name **Modeling diagrams**

Criteria **Modeling**

Scope project

Definition Verifies the completeness and validation of components of modeling diagrams (class diagrams and data models)

Mark Manual global practice mark:

- 0 if there is no completeness
- 1 or 2 intermediary value according to human expertise
- 3 if the modeling diagram is validated and match the requirements.

Name **Antipattern predetection**

Criteria **Modeling**

Scope project

Definition If there is automatic generation of code, this practice qualifies and predetects antipatterns in the UML model.

This Practice is divided in 8 sub-practices which detect the following antipatterns:

- Inheritance depth
- Swiss army knife classes
- Number of methods
- Class specialization
- Encapsulation
- Classes without method:
- Classes without attribute
- Isolated classes

The sub-practices Inheritance depth, Swiss army knife, Number of methods and Class specialization are defined in 4.3.1 while the other are defined below.

Mark Computed global practice mark: simple average which each sub-practice is weighted at 1/8.

Name **Encapsulation**

Criteria **practice Antipattern predetection**

Scope class

Metrics number of publics fields

Definition Qualifies the number of publics fields for a class. Publics fields should be avoided. It is recommended to use accessors instead.

Mark Computed Individual mark:

- 0 if there are public fields
- 3 if there are no public fields

Global practice mark: weighted global average.

Name **Classes without method**

Criteria **practice Antipattern predetection**

Scope class

Metrics NOM (number of methods)

Definition Qualifies the number of methods for a class. A class should have at least accessors methods to access attributes and methods to perform its own treatments.

Mark Computed Individual mark:

- 0 if there is no method
- 3 if there are methods

Global practice mark: weighted global average.

Name **Classes without attribute**

Criteria **practice Antipattern predetection**

Scope class

Metrics number of fields

Definition qualifies the number of fields for a class. A class should have at least one field to describe here instance.

Mark Computed Individual mark:

- 0 if there is no attribute

- 3 if there are attributes

Global practice mark: weighted global average.

Name **Isolated classes**

Criteria **practice Antipattern predetection**

Scope class

Metrics DepClients, DepSuppliers

Definition Qualifies the relation between this class and the rest of the project. A class must be in relation with another to ensure its existence in the model.

Mark Computed Individual mark:

- 0 if Depclient and DepSupplier = 0
- 3 if Depclient or DepSupplier != 0

Global practice mark: weighted global average.

Name **Model Reasoning**

Criteria **Modeling**

Scope project

Definition Evaluates the level of model reasoning in case of correction, modification or addition of features to the project.

Mark Manual global practice mark:

- 0 if the model does not meet the requirements for modification or addition of features to the project.
- 1 or 2 intermediary value according to human expertise.
- 3 if the model is correct.

Name **Modeling and implementation conformity**

Criteria **Modeling**

Scope project

Definition Qualifies the coherence between modeling and implementation. Verifies the coherence controls and the passages between each of these models.

Mark Manual global practice mark:

- 0 if implementation is not coherent with model.
- 1 or 2 intermediary value according to human expertise.
- 3 if implementation is coherent with model.

4.3.3 Practices from Rules Checking

These practices determine the quality of program development. They verify that the rules of programming are respected in the lines of code. These rules are defined before starting the project and must be known by the team developers. These practices depend on the rules defined by the enterprise. These rules are:

- syntactic and formatting rules: define the structure of the code.
- naming rules: define the naming convention for data, methods, classes, packages.
- programming rules: look for some practice which are known to be bad practice or to potentially introduce bugs.
- documentation rules: must be respected to allow javadoc to generate automatically the documentation from the source code.
- architecture rules: define the respect of the laying architecture and the use of design patterns.

For these practices, the marks score the transgressions of the rules. There are three kinds of transgressions :

- errors which are most strongly weighted (W_1)
- warning which are moderately weighted (W_2)
- informations which are lightly weighted (W_3)

The weight applied to the transgressions reflects the importance of the transgression. It corresponds to one transgression tolerated per number of items (lines of code, classes).

Scope (as well as parameters) of a rule checking practice depends on the tool used. If the tool reports only files, the practice can't easily be mapped back to source code entities: then it is reported at project level. Checkstyle or PMD tools —actually used to perform theses practices— give only reports without specify which entity is concerned. Rule checking practice mark are dependent of theses tools therefore they are computed directly for the project.

Name **Layer respect**

Criteria **Architecture Respect**

Scope project

Metrics Number of classes

Definition Determines the level of layer respect compared to the initial project. This measure calculates the level of transgression.

Mark Global practice mark:

$$mark = 3 * \frac{2}{3}^{\frac{W_1 * ErrorNumber + W_2 * WarningNumber + W_3 * InfoNumber}{Number\_of\_Classes}}$$

Weight hard

Name **Documentation standard**

Criteria **comprehension**

Scope project

Metrics sloc (number of source lines of code)

Definition Determines if extractable documentation (like JavaDoc) exists for the project. Qualifies the API documentation but not the lines of code documentation (see "Source comments rate" practice).

Mark Global practice mark:

$$mark = 3 * \frac{2}{3}^{\frac{W_1 * ErrorNumber + W_2 * WarningNumber + W_3 * InfoNumber}{sloc}}$$

Weight soft

Name **Formatting standard**

Criteria **Homogeneity**

Scope project

Metrics sloc (number of source lines of code)

Definition Determines if the formatting rules for source code are respected. Verifies the homogeneity of source code.

Mark Global practice mark:

$$mark = 3 * \frac{2}{3}^{\frac{W_1 * ErrorNumber + W_2 * WarningNumber + W_3 * InfoNumber}{sloc}}$$

Weight soft

Name **Naming standard**

Criteria **Homogeneity**

Scope project

Metrics sloc (number of source lines of code)

Definition Determines the level of compliance for naming rules for the project.

Mark Global practice mark:

$$mark = 3 * \frac{2}{3}^{\frac{W_1 * ErrorNumber + W_2 * WarningNumber + W_3 * InfoNumber}{sloc}}$$

Weight soft

Name **Tracing standard**

Criteria **Exploitability, Stability**

Scope project

Metrics sloc (number of source lines of code)

Definition Qualifies tracing elements for automatic generation of log files.

Mark Global practice mark:

$$mark = 3 * \frac{2}{3}^{\frac{W_1 * ErrorNumber + W_2 * WarningNumber + W_3 * InfoNumber}{sloc}}$$

Weight hard

Name **Security standards**

Criteria **Security**

Scope project

Metrics sloc (number of source lines of code)

Definition Qualifies the respect of security rules for the source lines of code.

Mark Global practice mark:

$$mark = 3 * \frac{2}{3}^{\frac{W_1 * ErrorNumber + W_2 * WarningNumber + W_3 * InfoNumber}{sloc}}$$

Weight hard

Name **Portability standard**

Criteria **exploitability**

Scope project

Metrics sloc (number of source lines of code)

Definition Determines the portability of the application. Verifies that there is no material or software dependency.

Mark Global practice mark:

$$mark = 3 * \frac{2}{3}^{\frac{W_1 * ErrorNumber + W_2 * WarningNumber + W_3 * InfoNumber}{sloc}}$$

Weight hard

Name **Programming standard**

Criteria **Homogeneity**

Scope project

Metrics sloc (number of lines of code)

Definition Determines the level of compliance for programing rules for the project.

Mark Global practice mark:

$$mark = 3 * \frac{2}{3}^{\frac{W_1 * ErrorNumber + W_2 * WarningNumber + W_3 * InfoNumber}{sloc}}$$

Weight soft

4.3.4 Practices from Human Analysis

A project must include some documents like functional specifications or documentation files. The quality of a project depend on the quality of these documents. The goal of this analysis is to verify that all the documentation needed for a good quality project exists and to qualify the quality of these documents. These analysis require an human expertise and can not be automated.

Since human computed, these practice marks are not computed as often as the metric based practice marks. They have thus a limited life time and must be re-determined after a given time.

Name **Quality Assurance Plan**

Criteria **Task Aptitude**

Scope project

Definition Verifies that there is a Quality Assurance Plan accorded to the methodology of the enterprise.

Mark Global practice mark:

- 0 If there is no Quality Assurance Plan.
- 1 If there is a Quality Assurance Plan but not conform.
- 3 If there is a Quality Assurance Plan.

Name **Ergonomy norms**

Criteria **Task Aptitude**

Scope project

Definition Verifies that there are ergonomy norms for the project.

Mark Global practice mark:

- 0 if there aren't any ergonomy norms

- 1 if there are ergonomy norms but not totally respected.
- 3 If there are ergonomy norms conforming to the graphic chart of the enterprise.

Name **Functional specification**

Criteria **Task Aptitude**

Scope project

Definition Verifies that there is a functional specification for the project.

Mark Global practice mark:

- 0 if there is no functional specification.
- 1 or 2 if there is a functional specification but not entirely correct.
- 3 if the functional specification is present and correct.

Name **Functional security aspects**

Criteria **Security**

Scope project

Definition Verifies that the functional security aspects described in the functional specification file are applied.

Mark Global practice mark:

- 0 if the functional security aspect are not applied.
- 1 or 2 if the functional security aspect are not totally applied.
- 3 if the functional security aspect are correctly applied.

Name **Technical security aspects**

Criteria **Security**

Scope project

Definition Verifies that the technical security aspects described in the technical specification file are applied.

Mark Global practice mark:

- 0 if the technical security aspects are not applied.
- 1 or 2 if the technical security aspects are not totally applied.
- 3 if the technical security aspects are exactly applied.

Name **Implementation match security specifications**

Criteria **Security**

Scope project

Definition Verifies that the security specifications described in the functional specification file and the technical specification file are applied in the implementation of project.

Mark Global practice mark:

- 0 if security aspects are not implemented.
- 1 or 2 if security aspects are not totally implemented.
- 3 if security aspects are exactly implemented.

Name **Production file**

Criteria **Exploitability**

Scope project

Definition Verifies that there is a Production File and that it is relevant.

Mark Global practice mark:

- 0 if there is no Production File.
- 1 or 2 if Production File is not totally relevant.
- 3 If there is a complete and relevant Production File and if there is a deployment procedure updated.

Name **Exception handling**

Criteria **Stability**

Scope project

Definition Verifies that there is an Exception handling in a written document and that this Exception handling is applied in the code.

Mark Global practice mark:

- 0 if there is no Exception handling.
- 1 or 2 if Exception handling is not applied or not totally applied.
- 3 if Exception handling is relevant and applied in the source code.

Name **Documentation quality**

Criteria **comprehension**

Scope project

Definition Qualifies the technical documentation of the code according to the enterprise methodology. This documentation allows a programmer to understand quickly the code.

Mark Global practice mark:

- 0 if there is no technical documentation.
- 1 or 2 if technical documentation does not follow the enterprise methodology.
- 3 if the technical documentation is relevant and written according to the enterprise methodology.

Name **Risk analysis gravity/frequency**

Criteria **acceptance test**

Scope project

Definition Verifies that the Strategy of Acceptance test scenarios is based on functional and technical risk assessment.

Mark Global practice mark:

- 0 If there is no Risk Analysis.
- 1 or 2 if the strategy of acceptance tests scenarios is not totally based on risk assessment.

- 3 If there is a complete Risk Analysis.

Name **Acceptance test scenario**

Criteria **Acceptance tests**

Scope project

Definition Verifies that there are Acceptance test scenarios to measure the quality of the features expressed in the functional specifications for the project. This specifications must have been directed by the Business Technology consultant and validated.

Mark Global practice mark:

- 0 If there are no Acceptance test scenarios.
- 1 or 2 If there are Acceptance test scenarios but not for 100% of exigencies.
- 3 If there are Acceptance test scenarios for 100% of exigencies.

Name **Automatic acceptance test**

Criteria **Acceptance tests**

Scope project

Definition Verifies that there are automatic scenarios of acceptance tests and the maturity of these.

Mark Global practice mark:

- 0 if there is no automatic scenarios.
- 1 and 2 if automatic scenarios are not totally relevant.
- 3 if there are satisfactory automatic scenarios.

Name **Layering**

Criteria **Architecture Modularity**

Scope project

Definition Verifies that there is a validated Technical Specification File, with a clearly description for architecture and with a clearly detailed layering.

Mark Global practice mark:

- 0 if there is no layering.
- 1 or 2 if the layering is not enough detailed or not satisfying.
- 3 if the layering is correct.

Name **Conformity between layers and package naming**

Criteria **Architecture Respect**

Scope project

Definition Verifies that package naming is in conformity with layers defined in the Technical Specification File.

Mark Global practice mark:

- 0 if there is no conformity.
- 1 or 2 if the conformity is not totally respected.
- 3 if there is a total conformity.

Name **Code organization**

Criteria **Architecture Relevance**

Scope project

Definition Qualifies the general code organization: the coherence between packages, the sharing of common elements, the management of libraries, the dead code.

Mark Global practice mark:

- 0 if the code organization is really bad.
- 1 or 2 if the code organization could be better.
- 3 if the code organization is correct.

Name **Mechanism choices**

Criteria **Architecture Relevance**

Scope project

Definition Examines the mechanisms linking the kinematics of the application : design-pattern, their implementation and their relevance.

Mark Global practice mark:

- 0 if the design-patterns are "bad" or the implementation is not correct.
- 1 or 2 if the design-patterns are not totally correct or not really relevant or consistent.
- 3 if the design-patterns are correct and relevant and if they are consistent with the project.

Name **Security technical design file**

Criteria **Architecture Relevance**

Scope project

Definition Verifies that the Security principles which are described in the technical design file are applied.

Mark Global practice mark:

- 0 if there is no Security principles in the technical specification file.
- 1 or 2 if the security principles are not totally applied.
- 3 if the Security principles are applied.

Name **Technical architecture file**

Criteria **Architecture Relevance**

Scope project

Definition Verifies that there is a Technical specification file and qualify its consistency with respect to the functional and technical constraints.

Mark Global practice mark:

- 0 if there is no Technical specification file.
- 1 or 2 if there is a Technical specification file but not totally consistent with the constraints.
- 3 If there is a Technical specification file with 100% of fulfilled requirements.

Name **Technology choices**

Criteria **Architecture Relevance**

Scope project

Definition Verifies the technology choices and checks that they are compliant with the project.

Mark Global practice mark:

- 0 if the technology choices are inappropriate to requirements or if the technologies are not under control.
- 1 or 2 if the technology choices are not totally correct or not totally mastered.
- 3 if the technology choices are appropriate to the requirements and the technologies are under control.

4.3.5 Practices from Dynamic Analysis

One of the more important phase of the development of a project is the testing of the code. The Squale Model include the analysis of the code coverage. This analysis aims to qualify the behavior of the software in exploitation.

These practices are either computed or manually obtained.

Name **Acceptance test code coverage**

Criteria **acceptance test**

Scope project

Metrics code coverage per branch

Definition Determines the level of acceptance test code coverage.

Mark Computed global practice mark:

$$3 * (code\_coverage/100)$$

Name **Functional limits testing**

Criteria **acceptance test**

Scope project

Definition Verifies that Functional limits are tested and qualify the results.

Mark Manual global practice mark:

- 0 if there is no Functional limits testing.
- 3 if there are satisfactory functional limits testing.
- 1 and 2 are intermediary level.

Name **Functional tests of non regression**

Criteria **acceptance test, technical tests**

Scope project

Definition Verifies that the non regression is tested and qualify the results.

Mark Manual global practice mark:

- 0 if there is no Functional test of non regression.
- 3 if Functional tests of non regression are performed on 80% of the code with 100% success rate.
- 1 and 2 are intermediary level.

Name **Unit test**

Criteria **Technical tests**

Scope project

Definition verify if there are unit tests and qualify the results.

Mark Manual global practice mark:

- 0 if there is no unit tests.
- 3 if there is a 100% success rate.
- 1 and 2 are intermediary level.

Name **Unit test coverage**

Criteria **Technical tests**

Scope project

Metrics code coverage, branch code coverage

Definition Qualifies the level of unit test code coverage.

Mark Computed global practice mark:

$$3 * (branch\_code\_coverage + code\_coverage)/200$$

Name **Integration test coverage**

Criteria **Technical tests**

Scope project

Definition Verifies that there are integration tests and qualify the results of these tests.

Mark Manual global practice mark:

- 0 if there is no integration tests.
- 3 if there is a 100% success rate.
- 1 and 2 are intermediary level.

Name **Robustness tests**

Criteria **Stability**

Scope project

Definition Verifies that there are robustness tests and qualify these tests.

Mark Manual global practice mark:

- 0 if there is no robustness test.
- 3 if robustness tests are performed on 80% of the code with 100% success rate.
- 1 and 2 are intermediary level.

Name **Load tests procedure**

Criteria **Stability**

Scope project

Definition Verifies that there are load tests procedure and qualify these tests.

Mark Manual global practice mark:

- 0 if there is no load tests procedure.
- 3 if tests are performed on 80% of the code with 100% success rate.
- 1 and 2 are intermediary level.

Chapter5. Perspectives

5.1 Squale adaptability

The Squale quality model is defined to be adapted to different technologies as well as standards of enterprise through choices of alternative practices, criteria and weights in formulae.

Currently we performed one squale model instance as it is described in 4. This model instance is based on industrial needs and tailored to information system in object-oriented programs. Indeed, some practices are technology dependent. For example, the *inheritance depth* practice is only relevant for object-oriented programs. Practices defined for modeling analysis are based on UML but other modeling methodology such as Merise can be considered. Some practices have alternatives depending on the technology. *coupling* practices are defined for object-oriented programs but are equivalent to *fan-in* and *fan-out* practices for procedural programs.

In future work, other Squale models instances will be defined to deal with different kinds of software systems, like for example embedded or real time programs.

Different technologies will be also taken into account and practices will be defined independently of technologies and metric tools used.

5.2 Practices for packages

We found that packages are not well assessed by current practices. Packages embody program organization and are the unit of modularity, release, and reuse, at the program level. Their relationships represent the dependencies in the architecture of a program. We believe that based on Martin's design principles we should be able to define some new practices.

Martin discusses principles of architecture and package design, addressing package cohesion and package coupling [Mar00]. Package cohesion links to granularity while package coupling links to stability.

Package cohesion primarily defines a package as a granule of release. A package is cohesive if its classes work together, are reused together, or change together during subsequent releases of the package. The package cohesion principles are:

Reuse/Release Equivalency Principle (REP): *the granule of reuse is the granule of release. The granule is the package.* Packages are reused by clients as basic OO libraries. Then, each package should be tracked and released in consistent state. Clients should only reuse packages which have been released, so that they are updated against consistent changes.

Common Reuse Principle (CRP): *the classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.* At the architecture level, a dependency upon one class in the package is not different from a dependency upon everything within the package. Then, grouping classes that are reused together in one package will limit the number of dependencies clients have to declare.

Common Closure Principle (CCP): *a change that affects a package affects all the classes in that package.* To minimize the number of packages that are changed in any given release cycle, it is better to group classes that change together into the same package. This way, a class change is more likely to impact classes in the same package, thus limiting the impact and propagation on other packages.

Coupling is generally defined as: *if changing one package in a program requires changing another package, then coupling between these two packages exists* [BDW98, Fow01]. Martin's package coupling principles are:

Acyclic Dependencies Principle (ADP): *there must be no cycles in the dependency structure.* Changes in package propagate to clients of the package and furthermore. A cycle in package dependency makes all packages in the cycle dependent on the others and their dependencies. Then, a package becomes dependent on numerous packages it does not use directly or indirectly. Any change in the cycle and its dependencies require a full build, breaking modularity.

Stable Dependencies Principle (SDP): *a package should only depend upon packages that are more stable that it is.* package stability is concerned with the amount of work required by a change in it: not only its internals change, but also packages which depend on it can change. The more incoming dependencies a package has, the more responsible it is towards its client packages because a change can impact them. Thus, the more stable it should be. On the other hand, a package with no incoming dependency is not responsible in front of other packages and can be very unstable.

Stable Abstractions Principle (SAP): *packages that are maximally stable should be maximally abstract.* Packages with concrete implementation are likely to change often because of all implementation details. Then, they can not be fully stable and depended upon. Abstract packages are less likely to change if they can hide such details. Then, they are more stable and useful as core dependencies. Stable packages should be highly abstract while concrete packages should be unstable. Thus, to improve the flexibility of applications, architects can compose unstable packages that are easy to change, and stable packages that are easy to extend.

Analysis.

A common guideline behind Martin's principles is that good packages are designed to limit the impact of changes. Each principle teaches a particular lesson:

REP	The package is the unit of change at the project level. From Martin [Mar00]: *package dependency diagrams are a map of how to build the application.*
CRP	Reusing classes together in a single package limits the number of dependencies.
CCP	Classes changing together in a single package only affects this package and its dependencies.
ADP	Acyclic dependencies limit the propagation of changes.
SDP	The more responsibilities a package has, the less it should change.
SAP	The more abstract a package is, the less it changes.

Cohesion and coupling are among the most used metrics during perfective maintenance, because they help identify which packages should be restructured [MT07, AG01, RC92, BDW99, ABF04, LM06]. In general, good packages should group classes that are needed for the same task [PN06], and they should have a few clear dependencies to other packages: they should be highly cohesive and lightly coupled. However, cohesion and coupling alone do not help maintainers understand the structure, roles, or relationships of packages. In particular, they do *not* indicate whether, why, or how a package respects Martin's cohesion/coupling principles, nor do they help decide what to do. A practice may help in that direction.

Class practices. Some new practices at the class scope can complement good package design. For example, a practice covering the *Law of Demeter* is interesting to assess coupling between classes. Practices based on the *Interface Segregation Principle* [Mar00], or the guideline *"program to an interface, not an implementation"* can help to assess the stability of a design. We will work on the definition of a set of practices using the metrics identified in the workspace 1.1 to address this lack.

5.3 Practices in the life cycle

A quality model should be able to monitor the evolution of quality over the whole software life cycle.

Considering the software life cycle, not all data are available to assess all practices during the whole course of the project. At the beginning, only specifications are available while the different measures become available only toward the end. This dependency of practices over the life time of the project must be reflected in the quality model. Thus the model should integrate the stage of the project in order to highlight useful practices for which data are available and to give a meaningful interpretation of them depending of the project maturity.

The software life cycle should also give an indication to decide which manual practices should be determined at a given moment. For example, the practices which qualify the specifications must be evaluated only if the project is at the beginning.

Taking into account the life cycle of the project in the Squale Model will also allow to use the model as a tool to validate the different phases of project development: if a criterion does not have the required mark, the project doesn't access to the next stage for example.

When practice marks are not good, the Squale tool is able to provide to users a remediation plan which tries to establish priorities within the problem listed [DDB10]. This plan should also take into account the stage of the project to be pertinent. For example, specifications are important to manage a project and if they are not defined, related practices will show bad marks. But spending time to correct these practices is judicious only at the beginning of a project. So the Squale model should reflect this aspect within the different practice definitions to highlight only those problems that can be taken into account in an efficient way.

5.4 Weighting criterion and factor marks

As explained in Section 3.2.2, the Squale model computes criterion (resp. factor) marks with a standard weighted average. Nevertheless, in the current implementation each practice (resp. criterion) is weigthed as 1, meaning the the mark computation is just a simple average. Effectively exploiting this mechanism is thus an interesting perspective to explore in order to handle differently practices (resp. criteria), for example, at different stages of the system life cycle.

Alternatively, introducing at the criterion (resp. factor) level a mark computation similar to the one used for the practice level (see Section 3.2.1) would also be interesting. This would avoid cases of good criterion (resp. factor) ranks where a few bad marked pratices (resp. criteria) are hidden by a large number of good marked ones. Depending of the choosen weighting, such bad practices (resp. criteria) could thus be either stressed or ignored.

Bibliography

[ABDT04] A. Abran, P. Bourque, R. Dupuis, and L.L. Tripp. Guide to the software engineering body of knowledge (ironman version). Technical report, IEEE Computer Society, 2004.

[ABF04] Erik Arisholm, Lionel C. Briand, and Audun Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.

[AG01] Fernando Britoe Abreu and Miguel Goulao. Coupling and cohesion as modularization drivers: are we being over-persuaded? In *Fifth European Conference on Software Maintenance and Reengineering*, pages 47–57, March 2001.

[BBD+09] Francoise Balmas, Alexandre Bergel, Simon Denier, Stéphane Ducasse, Jannik Laval, Karine Mordal-Manet, H. Abdeen, and Fabrice Bellinguard. Software metrics for java and cpp practices, v1, http://www.squale.org/quality-models-site/, 2009.

[BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley, 1994.

[BD02] Jagdish Bansiya and Carl Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, January 2002.

[BDW98] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering: An International Journal*, 3(1):65–117, 1998.

[BDW99] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[DDB10] Simon Denier, Stéphane Ducasse, and Fabrice Bellinguard. Technical and economical model, 2010.

[EBM06] Software Quality Esmond B. Marvray, NASA Goddard Space Flight Center. Procedure for developing and implementing software quality programs, last update 2006.

[Fow01] Martin Fowler. Reducing coupling. *IEEE Software*, 2001.

[FP96] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1996.

[LK94] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[Mar97] Robert C. Martin. Stability, 1997. www.objectmentor.com.

[Mar00] Robert C. Martin. Design principles and design patterns, 2000. www.objectmentor.com.

[McC76] T.J. McCabe. A measure of complexity. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[MR04] Radu Marinescu and Daniel Raţiu. Quantifying the quality of object-oriented design: the factor-strategy model. In *Proceedings 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 192–201, Los Alamitos CA, 2004. IEEE Computer Society Press.

[MRW76] Jim McCall, Paul Richards, and Gene Walters. *Factors in Software Quality*. NTIS Springfield, 1976.

[MT07] Hayden Melton and Ewan Tempero. The crss metric for package design quality. In *ACSC '07: Proceedings of the Australian Computer Science Conference*, 2007.

[PN06] Laura Ponisio and Oscar Nierstrasz. Using context information to re-architect a system. In *Proceedings of the 3rd Software Measurement European Forum 2006 (SMEF'06)*, pages 91–103, 2006.

[RC92] Linda Rising and Frank W. Calliss. Problems with determining package cohesion and coupling. *Software - Practice and Experience*, 22(7):553–571, 1992.