Practices in the Squale Quality Model

Workpackage: 1.3

July 3, 2009

First Edition, June, 2009.

Workpackage: 1.3

Version: 1.0

Authors: F. Balmas, F. Belingard, S. Denier, S. Ducasse, J. Laval, K. Mordal-Manet

Planning

- Delivery Date: June 29, 2009
- First Version: April 15, 2008

Contents

1 Introduction **5**

2 State of the Art **6**

3 The Squale quality model **11**
3.1 Measures 11
3.2 Practices 12
3.3 Criteria 13
3.4 Factors 13

4 Practices in Details **15**
4.1 Individual mark 15
4.2 Practice mark 16
4.3 Practice analysis 17

5 Perspectives **42**
5.1 Squale adaptability 42
5.2 Practices for packages 42
5.3 Practices in the life cycle 44

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 the different existing pratices, giving precise definitions and description. Finally, it discusses possible future enhancements of this model.

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][NASA Goddard Space Flight Center, Software Quality, last update 2006].

McCall Factors Criteria Metrics (FCM).

The objective of FCM 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. Factors represent characteristics of the software, criteria represent subcharacteristics.

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 on 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 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 deals with in the ISO 9126 model. It should be noted that the concept of stability is different in Squale than in the ISO 9126. 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.

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 through 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,

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

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

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.

Others

Swat4j `http://www.codeswat.com` is a tool offering a source code auditing for Java. It comes with about 30+ Metrics and about 100+ Industry Standard Best Practice Rules. Swat4j seems to be based on the principles of ISO 9126-1 (Quality Model) and ISO 9126-3 (Software Product Quality, Internal Metrics). It is not clear what is their quality model.

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 wich import java code and C++ code. This screenshot is the import of source code of ejb3, for the exemple.

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.

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

- the text is the name of metrics.
  - CYC: Cyclomatic complexity

ISO 9126	Squale	Facteur	Capacité fonctionnelle			Architecture			Maintenabilité				Évolutivité				Capacité de réutilisation				Fiabilité			
Caractéristiques		Critère	Aptitude à la tâche	Modélisation	Recette fonctionnelle	Pertinence de l'architecture	Modularité de l'architecture	Respect de l'architecture	Homogénéité	Compréhension	Simplicité	Niveau d'interdépendance	Homogénéité	Compréhension	Modélisation	Modularité	Compréhension	Exploitabilité du code	Niveau d'interdépendance	Tests techniques	Stabilité	Simplicité	Tests techniques	Sécurité
	Sous-caractéristiques		1	0	1	0	0	0	0	1	1	2	0	1	0	1	1	3	1	1	4	1	1	1
Capacité fonctionnelle	Aptitude	2	Z		Z																			
	Exactitude	0																						
	Interopérabilité	1																Z						
	Conformité réglementaire	0																						
	Sécurité	1																						X
Fiabilité	Maturité	0																						
	Tolérance aux fautes	1																			Z			
	Possibilité de récupération	1																			Z			
Facilité d'utilisation	Facilité de compréhension	3								X				X			X							
	Facilité d'apprentissage	0																						
	Facilité d'exploitation	0																						
Rendement	Comportement vis à vis du temps	1																			Z			
	Comportement vis à vis des ressources	1																			Z			
Maintenabilité	Facilité d'analyse	2									Z											Z		
	Facilité de modification	0																						
	Stabilité	1										Z												
	Facilité de test	2																		Z			Z	
Portabilité	Facilité d'adaptation	1																Z						
	Facilité à l'installation	1																Z						
	Conformité relative aux règles de portabilité	0																						
	Interchangeabilité	3										Z				Z			Z					

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

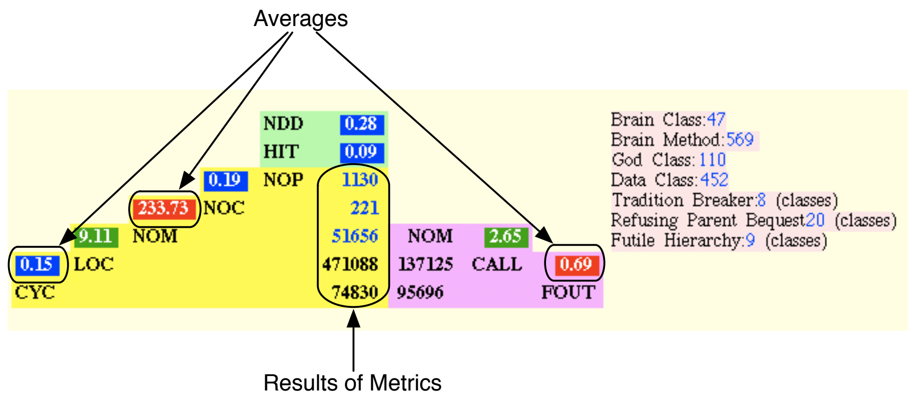

Figure 2.1: The overview pyramid.

- 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.
- 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 right to left, 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 a fourth layer, *the 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.

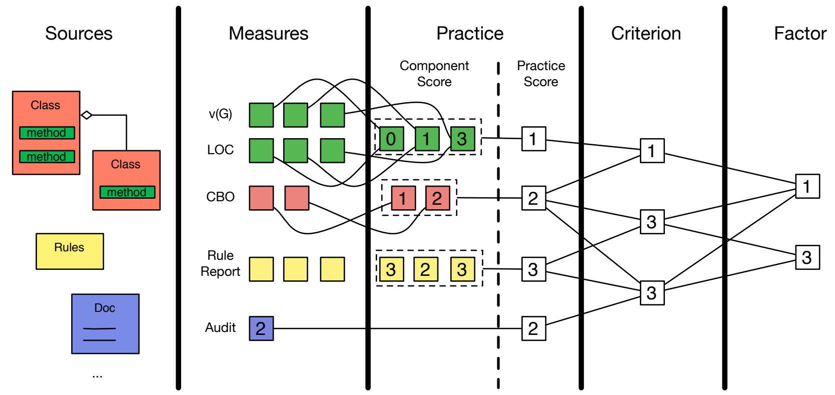

Figure 3.1: Data sources and levels of the Squale model.

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.

The following subsections briefly present the four levels of the Squale model, from the bottom measures to the top factors.

3.1 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], Hierarchy Nesting Level or 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.

Around 50 to 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.

3.2 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 can be 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 4.1.

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 more comments it should have.

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

3.3 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 marks is computed as the weighted average of the composed practice marks. Currently around 15 criteria are defined.

For example, the following practices:

- comment rate (per method with respect to cyclomatic complexity)
- inheritance depth
- documentation achievement (human audit with respect to project requirements)
- documentation quality (rule checking of programming conventions)

define the *comprehension* criterion.

3.4 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 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 :

- Homogeneity
- Comprehension
- Simplicity
- Integration Capacity

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

Figure 3.2 shows how measures are aggregated into practices, then into criteria and factors. The next Section will give exact formula to compute measure aggregation into practices. Formula for criteria and factors will be given in a future document.

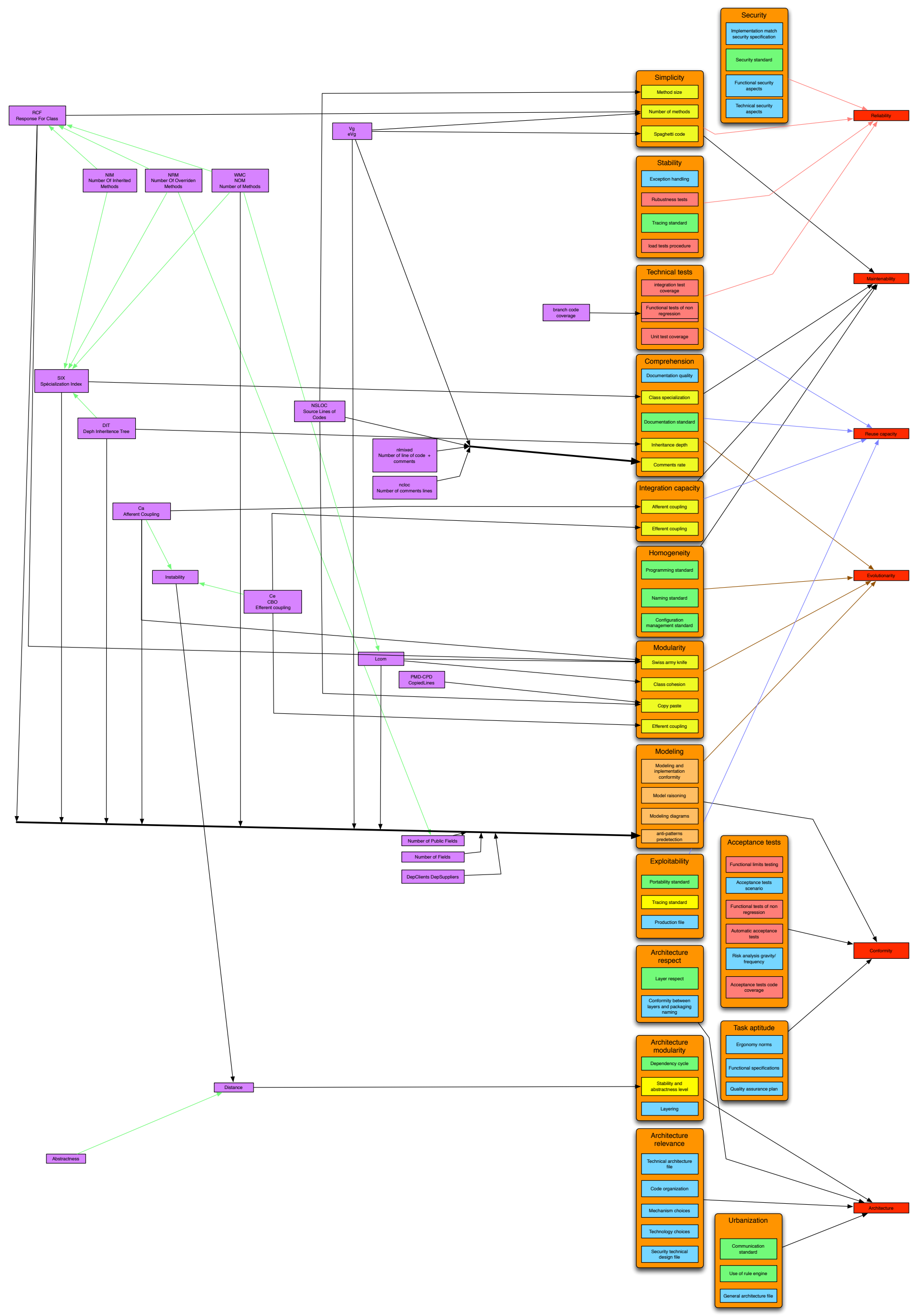

Figure 3.2: The Four Level Squale Model

Chapter4. Practices in Details

We now present in detail the practice layer and its specification as it defines the backbone of the Squale model.

A global mark for a practice is computed 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 can be computed for each method.

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

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

4.1 Individual mark

The formulae for computing individual marks come as two kinds, discrete or continuous. An individual mark is computed from measures in multiple ranges into a single mark in the range $[0; 3]$.

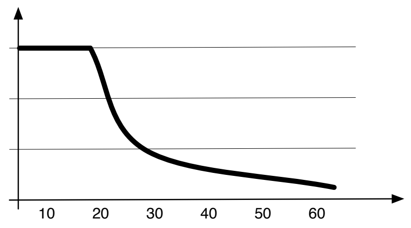

Figure 4.1: Sample graph for a practice mark based on one measure.

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 4.1 shows a mixed example using discrete and continuous equations 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.

4.2 Practice 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. 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 $g^{-1}(IM) = -log_{\lambda}(IM)$ is applied on the average to come back in the range $[0; 3]$.

Thus the global mark for a practice is:

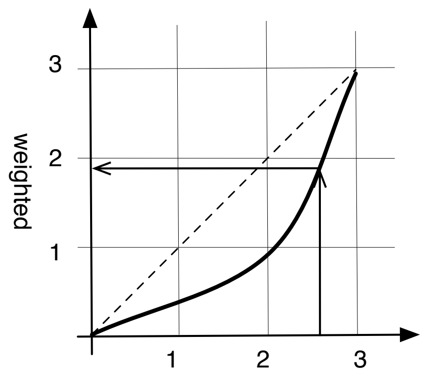

Figure 4.2: Principle of weighting: individual marks are lowered when translated in the weighted space.

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

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

Figure 4.2 illustrates how the $g(IM)$ function and its inverse works to reflect low individual marks in the practice mark. 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, above two of the marks. Instead, the marks are translated in the weighted space (red 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 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.

4.3 Practice analysis

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

- metrics analysis
- model analysis
- rules checking analysis
- documentary analysis
- testing analysis

For all the formulae we applied these abbreviations:

IM Individual Mark

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

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 score assigned to a practice.

Name **Inheritance Depth**

Criteria **Comprehension**

Scope class

Metrics DIT (depth inheritance tree)

Definition Determines the Inheritance depth for a class.

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 :

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

Name **Comments rate**

Criteria **Comprehension**

Scope methods

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

Definition Qualifies the comment rate in the lines of code. The appropriate threshold depends on the complexity of the method.

Mark Individual_mark

If $v(G) < 5$ and $sloc < 30$:

then the individual component is not marked

else:

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

Global practice mark :

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

Name **Number of methods**

Criteria **Simplicity**

Scope class

Metrics RFC (response for a class), V(g) (cyclomatic complexity), NOM (number of local and inherited methods), WMC (weigthed methods per class)

Definition Qualifies the number of methods for each class of project

Mark Individual Mark:

- $IM = \exp^{(30-wmc)/15}$ if $\sum v(G) >= 80$
- $IM = 2 + (20 - rfc)/30$ if $\sum v(G) \geq 50$ and $rfc \geq 15$
- $IM = 3 + (15 - rfc)/15$ if $\sum v(G) >= 30$
- $IM = 3$ if $\sum v(G) < 30$and $rfc < 15$

Global practice mark :

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

Name **Method Size**

Criteria **Simplicity**

Scope method

Metrics sloc (number of source lines of code)

Definition Qualifies the method size.

Mark individual_mark

$IM = \exp^{(70-sloc)/30}$

Global practice mark :

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

Name **Swiss army knife**

Criteria **Modularity**

Scope class

Metrics LCOM2 (lack of cohesion in methods), Ca (afferent coupling), RFC (response for a class)

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.

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 :

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

Name **Class cohesion**

Criteria **modularity**

Scope class

Metrics lcom2 (lack of cohesion in methods)

Definition Qualifies the relations between the methods of a class.

Mark individual mark:

- 0 if $lcom2 > 100$
- 1 if $lcom2 > 50$

- 2 if $lcom2 > 0$
- 3 if $lcom2 \leq 0$

Global practice mark :

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

Name	**Efferent Coupling**
Criteria	**Modularity, integration capacity**
Scope	class
Metrics	CBO (coupling between object classes)
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.
Mark	Individual mark

$IM = \exp^{(10-cbo)/3}$

Global practice mark :

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

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 the classes and one class : it's the number of classes which depend on the studied class.
Mark	Individual mark

$IM = \exp^{(30-ca)}/10$

Global practice mark :

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

Name **Spaghetti Code**

Criteria **Simplicity**

Scope method or project

Metrics v(G) or ev(G) (cyclomatic complexity) and sloc (number of source lines of code)

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

Mark Individual mark

$IM = e^{(6-ev(g))/4}$

Global practice mark :

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

Name **Copy Paste**

Criteria **Modularity**

Scope project

Metrics SLOC

Definition This practice highlights the lines of code which are duplicated. This number of copied lines is compared with the percentage of similarity between the methods.

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

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:

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

Global practice mark :

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

Name **Class specialization**

Criteria **comprehension**

Scope class

Metrics SIX (specialization index)

Definition Qualifies the class specialization.

Mark Individual mark:

- 0 if SIX >= 0,5
- 3 if SIX < 0,5

Global practice mark :

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

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 ameliorate the quality of their project. So when an U.M.L. model is made, the quality model is able to analyze the relevance of the modeling project. This allows to detect as soon as possible a wrong design and even before the implementation of these project. Theses analyzes verify also if the implementation is matching with the modeling project.

Except if explicitly mentioned, these practices are scored with the same discrete formula.

Individual mark:

- 0 (not done)
- 1 or 2 (intermediary value according to customer exigencies)
- 3 (done)

Global practice mark : weighted average of individual marks.

Name **Modeling diagrams**

Criteria **Modeling**

Scope project

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

Mark

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 wich detect the following antipatterns:

- Inheritance Depth
- Swiss army knife classes
- Number of methods
- public fields
- classes without method
- classes without attribute
- isoled classes
- class specialization

The sub-practices Inheritance Depth, Swiss army knife, Number of methods and class specialization are defined in 4.3.1 and the other sub-practices are defined here.

Mark 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.

Mark Discrete: Individual mark:

- 0 if there are public fields
- 3 if there aren't any public fields

Name **Classes without method**

Criteria **practice Antipattern predetection**

Scope class

Metrics NOM (number of methods)

Definition Qualifies the number of methods for a class.

Mark Discrete: Individual mark:

- 0 if there isn't any method
- 3 if there are methods

Global practice mark : weighted average of individual marks

Name **Classes without attribute**

Criteria **practice Antipattern predetection**

Scope class

Metrics number of fields

Definition qualifies the number of fields for a class.

Mark Discrete: Individual mark:

- 0 if there isn't any attributes
- 3 if there are attributes

Global practice mark : weighted average of individual marks orangeModel

---

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 Discrete: Individual mark:

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

Global practice mark : weighted average of individual marks

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

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

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. Theses rules are defined before starting the project and must be known by the team developers. Theses rules are:

- the syntactic rules: define the formatting code.
- the naming rules: define the naming convention for data, methods, classes, packages.
- the programming rules: look for some practice which are reputed as bad practice or potentially bring bugs.
- the documentation rules: are needed to generate an automatic documentation from source code.
- the 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 the number of lines tolerated per transgression.

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.

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: $\frac{\sum_1^n IM_n}{n}$

Name	**Layer respect**
Criteria	**Architecture Respect**
Scope	project, SLOC
Metrics	number of classes
Definition	Determines the level of layer respect compared to the initial project. This measure calculates the level of transgression.
Mark	$$mark = 3 * \frac{2}{3}^{\frac{W_1 * ErrorNumber + W_2 * WarningNumber + W_3 * InfoNumber}{number\_of\_classes}}$$

Name	**Documentation standard**
Criteria	**comprehension**
Scope	project
Metrics	sloc, checkstyle("documentationstandard")
Definition	Determines if the Javadoc exists for all the projects.
Mark	$$mark = 3 * \frac{2}{3}^{\frac{W_1 * ErrorNumber + W_2 * WarningNumber + W_3 * InfoNumber}{number\_of\_classes}}$$

Name	**Formating standard**
Criteria	**Homogeneity**
Scope	project
Metrics	sloc (number of source lines of code), checkstyle("formatingstandard")
Definition	Determines if the formatting rules for source code are respected. Verifies the homogeneity of source code.
Mark	$$mark = 3 * \frac{2}{3}^{\frac{W_1 * ErrorNumber + W_2 * WarningNumber + W_3 * InfoNumber}{sloc}}$$

Name **Naming standard**

Criteria **Homogeneity**

Scope project

Metrics sloc (number of source lines of code), checkstyle("namingstandard")

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

Mark

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

Name **Tracing standard**

Criteria **Exploitability, Stability, Performance**

Scope project

Metrics Number of lines of code, chekstyle.("tracingstandard")

Definition Qualifies tracing elements for automatic generation of log files.

Mark

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

Name **Security standards**

Criteria **Security**

Scope project

Metrics sloc (number of source lines of code), chekstyle("securitystandard")

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

Mark

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

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

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

Name **Programming standard**

Criteria **Homogeneity**

Scope project

Metrics sloc (number of lines of code), checkstyle("programmingstandard")

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

Mark

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

Name **Communication standard**

Criteria **urbanization**

Scope project

Metrics sloc (number of source line of code)

Definition Verifies that the project uses the pivot model for the communication with external systems and the EAI and EDI norms.

Mark

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

Name **Use of rule engine**

Criteria **urbanization**

Scope project

Metrics sloc (number of source lines of code)

Definition Verifies that the project uses the rules engine.

Mark

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

Name **Configuration management standard**

Criteria **Homogeneity**

Scope project

Metrics sloc

Definition Verifies that the project uses the configuration management standards: use and management of branches and labels

Mark

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

4.3.4 Practices from Documentation 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 quality project exists and to qualify the quality of these documents. These analysis require an human expertise and can not be automatic.

Except if explicitly mentioned, these practices are scored with the same discrete formula.

- 0 (not done)
- 1 or 2 (intermediary value according to customer exigencies)
- 3 (done)

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
- 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

Name **Functional specification**

Criteria **Task Aptitude**

Scope project

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

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

Name **Technical security aspects**

Criteria **Security**

Scope project

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

Mark

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

Name **Production file**

Criteria **Exploitability**

Scope project

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

Mark

- 0 if there is no Production File
- 1 or 2
- 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 writing document and that this Exception handling is applied in the code

Mark

Name **Documentation quality**

Criteria **comprehension**

Scope project

Definition Qualify the technical documentation according to the enterprise methodology. This documentation allows a programmer to understand quickly the code. This practice looks for comments in code and detects the lines of code in comments.

Mark

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
- 0 If there is a Risk Analysis.
- 3 If there is a complete Risk Analysis.

Name **Acceptance test scenario**

Criteria **acceptance test**

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
- 0 If there aren't Acceptance test scenarios
- 1 or 2 If there are Acceptance test scenarios but not for 100% of exigences.

- 3 If there are Acceptance test scenarios for 100% of exigences.

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
- 0 If there is no layering
- 1 or 2 If the Technical Specification File is not clear or not detailed enough.
- 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
- 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
- 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 Examine the mechanisms linking the kinematics of the application : design-pattern, their implementation and their relevance.

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
- 0 If there is not Security 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
- 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 fullfilled requirements.

Name **Technology choices**

Criteria **Architecture Relevance**

Scope project

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

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 of not totally mastered.
- 3 If the technology choices are appropriate to the requirements and the technologies are under control.

Name **General Architecture File**

Criteria **Urbanization**

Scope project

Definition Verifies that there is a General Architecture File and that it is compliant with urbanization constraints

Mark
- 0 If there is no General Architecture File
- 1 or 2 If there is a General Architecture File but not totally relevant.
- 3 If there is a General Architecture File in compliance with urbanization constraints.

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 aim to qualify the behavior of the software in exploitation.

These practices are either calculated or manually obtained.

Name **Automatic acceptance test**

Criteria **Acceptance tests**

Scope project

Metrics manual

Definition verify if there are automatics scenarios of acceptance tests and the maturity of these.

Mark discrete obtained manually :

- 0 if there is not automatics scenarios.
- 3 if there are satisfactory automatics scenarios.
- 1 and 2 are intermediaries level.

Name **Acceptance test code coverage**

Criteria **acceptance test**

Scope project

Metrics code coverage per branch

Definition determine the level of acceptance test code coverage.

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

Name **Functional limits testing**

Criteria **acceptance test**

Scope project

Metrics manual

Definition verify if Functional limits are tested and qualify the results

Mark discrete obtained manually :

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

Name **Functional tests of non regression**

Criteria **acceptance test, technical tests**

Scope project

Metrics manual

Definition verify if the non regression is tested and qualify the results

Mark discrete obtained manually :

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

Name **Unit test**

Criteria **Technical tests**

Scope project

Metrics manual

Definition verify if there are unit tests and qualify the results

Mark discrete obtained manually :

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

Name **Unit test coverage**

Criteria **Technical tests**

Scope project

Metrics code coverage, branch code coverage

Definition qualify the level of unit test code coverage

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

Name **Integration test coverage**

Criteria **Technical tests**

Scope project

Metrics manual

Definition verify if there is integration tests and qualify the results of these tests.

Mark discrete obtained manually :

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

Name **Robustness tests**

Criteria **Stability**

Scope project

Metrics manual

Definition verify if there is robustness tests and qualify these tests.

Mark discrete obtained manually :

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

Name **Load tests procedure**

Criteria **Stability**

Scope project

Metrics manual

Definition verify if there is load tests procedure and qualify these tests.

Mark discrete obtained manually :

- 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 intermediaries level.

Name	**Performance tests**
Criteria	**performance**
Scope	project
Metrics	manual
Definition	verify if there is performance tests and qualify these tests.
Mark	discrete obtained manually :

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

Chapter5. Perspectives

5.1 Squale adaptability

Each Squale model is customized on needed basis through choices of alternative practices and weights in formulae, taking into account different technologies as well as standards of enterprises. 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 a procedural programs.

Currently, the Squale model exists in several customized versions but without any common meta-model. A Squale meta-model should be defined to smooth the particularities of the different models and to give a generic model reference adaptable to the requirements. Practices should then 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 readily 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 know what is the stage of the project in order to select meaningful practices for which data are available.

To reflect the relative importance of practices in relation to each other and to modulate the importance of some practices in relation with the life cycle of the project, we will introduce weighting functions to aggregate practices in criteria. The weighting value could be changed according to the level of maturity of the project.

For example, specifications are important to conduct a project and if they are not defined, linked 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 within the different practice formulae to highlight only what can effectively be taken into account.

Weighting a practice should also take into account the deterioration of a practice: some practices are easy to respect at the beginning of a project but they tend to deteriorate over the time. For example, the Code Organization Practice controls that the general organization of code is respected. This practice tends to deteriorate gradually as the time life of the project, particularly because of external constraints such as planning. So it is very important to focus on this practice in relation with the maturity of the project.

Therefore, we plan to define a set of enhanced formulae for practices, criteria and factors during the second part of this work package.

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.

[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.

[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.