Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2013; **25**:1117–1135
Published online 13 August 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1558

Software quality metrics aggregation in industry

Karine Mordal[1], Nicolas Anquetil[2,*,†], Jannik Laval[4], Alexander Serebrenik[3], Bogdan Vasilescu[3] and Stéphane Ducasse[2]

[1] *LIASD, University of Paris 8, France*
[2] *RMoD Team, INRIA, Lille, France*
[3] *Technische Universiteit Eindhoven, The Netherlands*
[4] *LaBRI, Université de Bordeaux, France*

SUMMARY

With the growing need for quality assessment of entire software systems in the industry, new issues are emerging. First, because most software quality metrics are defined at the level of individual software components, there is a need for aggregation methods to summarize the results at the system level. Second, because a software evaluation requires the use of different metrics, with possibly widely varying output ranges, there is a need to combine these results into a unified quality assessment. In this paper we derive, from our experience on real industrial cases and from the scientific literature, requirements for an aggregation method. We then present a solution through the Squale model for metric aggregation, a model specifically designed to address the needs of practitioners. We empirically validate the adequacy of Squale through experiments on ECLIPSE. Additionally, we compare the Squale model to both traditional aggregation techniques (e.g., the arithmetic mean), and to econometric inequality indices (e.g., the Gini or the Theil indices), recently applied to aggregation of software metrics. Copyright © 2012 John Wiley & Sons, Ltd.

Received 27 May 2011; Revised 6 April 2012; Accepted 11 April 2012

KEY WORDS: software metrics; software quality; aggregation; inequality indices

1. INTRODUCTION

Software metrics are becoming part of the software development fabric, essential to understanding whether the quality of the software we are building corresponds to our expectations [1]. As a consequence, many different metrics have been proposed, and a plethora of tools to compute them and perform quality assessments. Considering the different stakeholders participating in software projects (e.g., developers, managers, users), quality needs to be evaluated at different levels of detail. Practical application of software metrics is, however, challenged by (i) the need to combine different metrics as recommended by quality-model design methods such as factor-criteria-metric (FCM) [2], or goal-question-metric [3]; and (ii) the need to obtain insights in the quality of the entire system based on the metric values obtained for low-level system elements such as classes and methods. We detail each challenge separately.

First, a meaningful quality assessment needs to combine the results of various methods to answer specific questions as suggested by quality-model design methods. For example, cyclomatic complexity might be combined with test coverage metrics to stress the importance of covering complex methods rather than accessors. However, integration of different metrics might be hindered by the different result ranges: for example, Martin's instability [4] ranges over $[0, 1]$, while the inheritance depth should be inferior to 10 in practice, the number of methods could go up to 100, and the number of lines of code can be expected not to exceed 1000.

*Correspondence to: Nicolas Anquetil, INRIA Team RMod, Parc Scientifique de la Haute Borne, 40, avenue Halley. Bat. A, Park Plaza, 59650 Villeneuve d'Ascq, France.
†E-mail: Nicolas.Anquetil@inria.fr

Second, most of the existing metrics are defined at the level of individual software components (classes, methods). However, for understanding larger software artifacts, such as components and systems, insights must be derived from these low-level results. A typical solution consists in averaging the results of a metric for all software components. This approach has an undesirable smoothing effect, potentially diluting bad results in the overall acceptable quality [5, 6]. Recently, there is a trend in applying econometric inequality indices to aggregation of software metrics [7–9]. Even though their applicability has been discussed [5, 6], their use for quality assessments under industrial considerations has not been evaluated yet.

In [10] we have proposed Squale[1], an empirical model for continuous and weighted metric aggregation, to address the aforementioned two challenges. In this paper, we further discuss the various issues arising when trying to assess the quality of software projects in an industrial setting. On the basis of these challenges, and current research trends in aggregation of software metrics, we distill requirements for software quality models. Additionally, we perform both a theoretical, and an empirical comparative evaluation of Squale with some of the existing techniques, and highlight their relative strengths and shortcomings.

The main contributions of this paper are threefold: (i) we identify requirements for software quality assessments in practice; (ii) through the Squale model, a quality aggregation solution defined empirically on industrial projects and evaluated more formally in this research, we present solutions to meet these requirements; and, (iii) we compare this model theoretically and empirically to econometric inequality indices, the most recent trend in software metrics aggregation [5–9, 11] and determine if they both fulfill requirements identified.

The remainder of this paper is organized as follows: In Section 2 we review existing techniques for software quality assessment, including a recent trend that involves econometric inequality indices, and we explain problems that may arise with such techniques in a real industrial context. In Section 3 we identify requirements for a meaningful quality assessment method. These requirements are derived from experience with quality evaluation in industry using the Squale model, and scientific literature on aggregation techniques for software metrics. In Section 4, we present the Squale model, a quality assessment method that was defined empirically on real-world projects to attend to the expectations of developers and managers. We consider how well Squale satisfies the requirements identified previously. In Section 5, we compare theoretically and empirically the Squale model to the econometric inequality indices. Finally, Section 6 discusses related work before concluding.

2. SOFTWARE QUALITY ASSESSMENT

Software project quality assessment raises two problems. First, software quality metrics, for example as proposed in the ISO 9126 standard [12], are often defined for individual software components (i.e., methods, classes, etc.) and cannot be easily transposed to higher abstraction levels (i.e., packages or entire systems). To evaluate a project, one needs to aggregate these metrics' results. Second, quality characteristics should be computed as a combination of several metrics. For example Changeability in part I of ISO 9126 is defined as 'the capability of the software product to enable a specified modification to be implemented' [12]. This subcharacteristic may be associated with several metrics, such as number of source lines of code (SLOC), cyclomatic complexity, number of methods per class, and inheritance depth (DIT).

Thus, combining the low-level metric values of all the individual components of a project can be understood in two ways. First, for a given component, one needs to compose the results of all the individual quality metrics considered, for example, SLOC and cyclomatic complexity. Second, for a given quality characteristic, be it an individual metric or a composed characteristic as Changeability, one needs to aggregate the results of all components into one high level value. Both operations result in information loss to gain a more abstract understanding: either individual metrics values are

[1]Since 2011, the Squale model is developped in the "Squash" research project. This project is supported and labelled by the "Systematic - PARIS Region" competitive Cluster, and partially funded by Paris region and the DGE ("Direction Générale des Entreprises") in the context of the French Inter-ministerial R&D project 2011–2013 ("Projet R&D du Fonds Unique Interministériel").

DOI: 10.1002/smr

Table I. A discrete mapping example of the SLOC metric to the [0, 3] interval.

SLOC	≤ 35	$]35, 70]$	$]70, 160]$	> 160
Normalized value	3	2	1	0
Interpretation	Good	Acceptable	Problems	Bad

lost in the composed results, or the quality evaluation of individual components is lost in the evaluation of the aggregation.

Although there is no predefined ordering of the two combination steps, in practice it is more meaningful to compose metrics before aggregating the results at a higher level. Metric composition is a semantic operation that may depend on the meaning and interplay of the metrics composed. For example, a quality evaluation of the comment rate of a component could be based on the composition of cyclomatic complexity and CLOC (commented lines of code) to allow assessing the fact that a complex method must be more commented than a simple one. On the other hand, aggregating results of different components is more statistical. If one were to compose already aggregated metrics results, one could lose this specific meaning. For example, the comment rate quality evaluation would already be less meaningful at the level of a class than at the level of individual methods: a class could have a very complex, poorly commented method and a very simple, overdocumented one, resulting in globally normal cyclomatic complexity and CLOC. Moreover, composing metrics at low levels and aggregating the results of this composition at higher level may provide a quality assessment of the evaluated characteristic for both the overall project and each of its components. Such an approach allows one to compare individual components and determine more easily which component should be addressed to improve the quality characteristic measured.

Another issue with quality evaluation in industry is linked to Wiegers' warning that using metrics to motivate rather than understand is a common trap: 'Metrics data is intrinsically neither virtuous nor evil, simply informative. Using metrics to motivate rather than to learn has the potential of leading to dysfunctional behaviour, in which the results obtained are not consistent with the goals intended by the motivator' [13]. However, in practice, and in any human activity, it is difficult to conceive any quality model that will not tend to become a goal of its own. To be accepted in practice, a quality model should not be solely an assessment model but also be usable as a guideline to increase quality. A manager should know if the project has quality problems, but a developer should know what component must be corrected. This implies that the composition and/or aggregation techniques also allow for a fine-grained analysis of the results.

In the remainder of this section we further discuss issues with composition and aggregation of metrics when applied in real industrial settings.

2.1. *Composition of software metrics*

Metrics composition involves taking into account the ranges of the metrics and raises two difficulties. First, the ranges may be very different, for example in the case of the changeability characteristic and its associated metrics (SLOC, cyclomatic complexity, number of methods per class, DIT), one sees that DIT can take its values in a different interval than SLOC. In this case, one must ensure not to dilute the results of one metric into the other. Second, metrics may have very different meanings, which imposes dealing with them in very different ways, for example, by using specific composition methods for each characteristic based on any given two (or more) metrics.

To be able to compose these metrics in a unified result, one can normalize them into a given interval of values. It is important that the interval be continuous (see below) as opposed to discrete values, for example, as in a Likert scale [14], and it is preferable that the interval have a finite bound on both sides to ease comparison.

Considering the normalization for SLOC measured per method (illustrated in Table I[2]), a discrete mapping would have the following drawbacks:

[2]Here, as well as throughout the rest of this paper, we use 'reversed brackets' interval notation [54], that is, $]a, b]$ is the set of all numbers x satisfying $a < x \leq b$.

DOI: 10.1002/smr

- *Hide modifications*. Discrete mapping of metric results introduces staircase and threshold effects that may hide detailed information and trigger wrong interpretation. Slight fluctuations — progression or regression — of individual elements might not appear if they remain in the same interval. For example, following the mapping proposed in Table I, a method with SLOC = 150 would be mapped to a normalized value of 1. If developers reduce the size of this method by half (SLOC = 75), the quality evaluation of the project does not reflect this change because the method is still mapped to the same normalized value.

- *Badly influence reengineering decisions*. A corollary of modifications within the same interval being hidden is that working on components close to a quality threshold value would exhibit more benefit on the overall quality than working on components whose values are far from a threshold. Therefore, engineers can use this mapping behaviour to improve the perceived quality at the cost of not fixing more serious problems. We saw this practice in one company, where developers selected their tasks to maximize their impact on the quality assessment.

2.2. Aggregation of software metrics

We now present the most common techniques employed in industrial settings for aggregation of software metrics and we highlight some of their drawbacks. We also discuss the state of the art of aggregation techniques in scientific literature.

2.2.1. Aggregation by simple averaging. Computing the arithmetic mean of individual metric results might not be representative enough because it does not convey the standard deviation of the population and may dilute unwanted values in the generally acceptable results, as illustrated in Table II (note that this is an already well known characteristic of the arithmetic mean). Table II presents the SLOC of four methods (denoted A to D) in two different projects. Assuming that lower SLOC values are more desirable for methods, Project 2 scores better than Project 1 when looking at the average SLOC values. However, this hides the fact that method *A* is an outlier hence, while the mark is better, the quality of the project might actually be lower. The average, because it smooths results, does not always represent reality [5].

For example, in one of our customers a method of 300 lines of code cannot be accepted. The simple average could easily fail to highlight this kind of problems, and even worse, it may hide the presence of very low-quality components. To have a quality model that highlights low-quality components, one could use a weighted average instead. This solution is discussed next.

2.2.2. Aggregation by weighted averaging. To highlight a low quality component or a critical component in the aggregation method, a possible solution is to increase the weight of the metric or the component in the average. However, this solution introduces problems of its own.

Table III shows an example of two versions of a project with weighted average of SLOC. The weights used in Table III were used in an initial version of the Squale quality model. In this example, the weighted average of Version 1 is 222.75. In Version 2, despite the reduction of the sizes of methods A, B, and C, the weighted average increases to 259.53. Hence, the aggregated value increased, suggesting a decrease of the software quality, while the code actually improved. A quality model should reflect all improvements as closely as possible.

Table II. Number of source lines of code for four methods in two projects.

Method	Project 1	Project 2
A	24	71
B	25	9
C	27	10
D	24	8
Average	25.0	24.5

DOI: 10.1002/smr

Table III. Two versions of a project's methods with weighted average (*wa*) of SLOC. The weights are: $[0, 35] \to \times 1$; $]35, 70] \to \times 3$; $]70, 160] \to \times 9$; $]160, +\infty[\to \times 27$.

	Version 1			Version 2		
Methods	SLOC	weight	w·SLOC	SLOC	weight	w·SLOC
A	30	1	30	25	1	25
B	50	3	150	30	1	30
C	70	9	630	50	3	150
D	300	27	8100	300	27	8100
		Σ=40	*wa*=222.75		Σ=32	*wa*=259.53

2.2.3. Other statistical aggregation techniques. In addition to the simple and weighted averages discussed above, in scientific literature aggregation of software metrics is realized using such functions as median or standard deviation [15–17]. However, the interpretation of central tendency measures (mean, median), becomes unreliable in the presence of highly-skewed distributions, common in software engineering [18]. In turn, this also compromises the interpretation reliability of aggregation functions based on the central tendency measures, such as the standard deviation, which is based on the mean.

An alternative is offered by distribution fitting [18–20], which consists of manually selecting a known family of distributions (e.g., log-normal or exponential) and fitting its parameters to approximate the metric values observed. The fitted parameters can then be seen as aggregating these values. However, the fitting process should be repeated with each new metric considered, and, moreover, it is still a matter of controversy whether, for example, software size is distributed log-normally [19] or double Pareto [21].

2.3. New trend in software metrics aggregation

As a response to these challenges (i.e., reliability under highly-skewed distributions, and simple application procedures), there is an emerging trend in using more advanced aggregation techniques borrowed from econometrics (inequality indices), where they are used to study inequality of income or welfare distributions [22–24]. Because data distribution in econometric is similar to data distribution in software engineering (highly-skewed distributions), and because these indices summarize a large quantity of data, their use has been recently proposed as aggregation techniques for software engineering quality metrics. This use does present some difficulties, an important one being that they are indicators of inequality and as such will give good grade to a population of all equally bad quality evaluations. We will come back to this issue in the experimental evaluation of these indices.

In this paper we consider the Gini [25], Theil and mean logarithmic deviation [26], Atkinson [27], Hoover [28] (also known as the Ricci–Schutz coefficient, or the Robin Hood index), and Kolm [29] income inequality indices. Table IV lists the definitions of the inequality indices considered when applied to values $x_1, \ldots, x_n$. We further use $\bar{x}$ to denote the mean of $x_1, \ldots, x_n$ and $|x|$ to denote the absolute value of x.

2.3.1. Mathematical properties of the inequality indices. Econometric inequality indices are based on a number of assumptions valid for economic values such as income or welfare, but not necessarily so for software metrics. For example, inequality indices cannot discriminate between all values being equally low and all values being equally high [24]. Such a fact is damageable for software metrics,

Table IV. Definitions of the inequality indices.

Index	Definition	Index	Definition
I_{Gini}	$\frac{1}{2n^2\bar{x}} \sum_{i=1}^{n} \sum_{j=1}^{n} \lvert x_i - x_j \rvert$	$I^{\alpha}_{\text{Atkinson}}$	$1 - \frac{1}{\bar{x}} \left(\frac{1}{n} \sum_{i=1}^{n} x_i^{1-\alpha} \right)^{\frac{1}{1-\alpha}}$
I_{Theil}	$\frac{1}{n} \sum_{i=1}^{n} \left(\frac{x_i}{\bar{x}} \log \frac{x_i}{\bar{x}} \right)$	I_{Hoover}	$\frac{1}{2n\bar{x}} \sum_{i=1}^{n} \lvert x_i - \bar{x} \rvert$
I_{MLD}	$\frac{1}{n} \sum_{i=1}^{n} \left(\log \frac{\bar{x}}{x_i} \right)$	I^{β}_{Kolm}	$\frac{1}{\beta} \log \left[\frac{1}{n} \sum_{i=1}^{n} e^{\beta(\bar{x} - x_i)} \right]$

DOI: 10.1002/smr

because a system with all files being equally complex should be considered more alarming than one in which all files are equally simple.

A number of properties of inequality indices are relevant for their application to aggregation of software metrics [6], including:

- *Domain and range*. Different inequality indices have different domains and ranges, not necessarily compatible with the ranges of the metrics aggregated, or among themselves. Recall, that the domain of a binary relation $R \subseteq X \times Y$ is the set of all $x \in X$ such that $(x, y) \in R$ for some $y \in Y$. Similarly, the range of R is the set of all $y \in Y$ such that $(x, y) \in R$ for some $x \in X$ [30]. To simplify the notation of domains and ranges in Table IV, we write $\varphi(x_1, \ldots, x_n)$ to indicate that $x_i \geq 0$ for all i, $1 \leq i \leq n$, and that there exists j, $1 \leq j \leq n$, such that $x_j > 0$. Similarly, we write $\mathbb{R}^n_\varphi$ to denote $\{(x_1, \ldots, x_n) | (x_1, \ldots, x_n) \in \mathbb{R}^n \wedge \varphi(x_1, \ldots, x_n)\}$. For example, the domain of I_{Theil}, I_{MLD}, and $I^\alpha_{\text{Atkinson}}$ is $\mathbb{R}^n_\varphi$, that is, these inequality indices cannot be applied to metrics with negative values such as the maintainability index [31]. Moreover, the range of I_{Theil} is $[0, logn]$, that is, the maximal possible value depends on the number of values being aggregated. Hence, if I_{Theil} is used to compare software systems of very different sizes, one should consider normalization of the aggregated values, for example, by dividing them by $\log n$ [7].
- *Invariance*. Invariance with respect to addition means that if one adds a constant to all individual values, this does not change the aggregated result; similarly, invariance with respect to multiplication means that if one multiplies all individual values by a constant factor, this does not change the aggregated result [24]. Both are mutually exclusive, of course.
- *Translatability*. As opposed to invariance with respect to addition, translatability means that adding a constant to all individual results increases the aggregated result by the same value. Translatability and invariance with regard to addition are mutually exclusive.
- *Symmetry* [32] or *impartiality* [29]. The property ensures that the aggregated result does not depend on the order of the elements being aggregated.
- *Decomposability* [33]. Decomposability enables measuring the extent to which the aggregated result can be attributed to differences between system subcomponents, a task often required when interpreting system-level results [8]. We further discuss decomposability in the next section.

Table V summarizes information about domain, range, invariance (with regard to addition or multiplication), symmetry and decomposability for the inequality indices considered.

2.3.2. Decomposability of inequality indices. One of the use cases for decomposability in an industrial software engineering setting is, for example, measuring the inequality of size (SLOC) between the classes in a software system, which is organized into packages. In this sense, an important question in interpreting the inequality value aggregated on a system level pertains to the extent to which the result can be attributed to differences between system subcomponents. This

Table V. Mathematical properties of the inequality indices.

Index	Domain	Range	Invariance	Symmetry	Decomposability
I_{Gini}	$\mathbb{R}^n_{\bar{x}\neq 0}$	$\mathbb{R}$ $\left[0, 1-\frac{1}{n}\right]$, if $\varphi(x_1, \ldots, x_n)$	*	Y	N
I_{Theil}	$\mathbb{R}^n_\varphi$	$[0, \log n]$	*	Y	Y
I_{MLD}	$\mathbb{R}^n_\varphi$	$\mathbb{R}^{\geq 0}$	*	Y	Y
$I^\alpha_{\text{Atkinson}}$	$\mathbb{R}^n_\varphi$	$\left[0, 1-\frac{1}{n}\right]$	*	Y	N
I_{Hoover}	$\mathbb{R}^n_{\bar{x}\neq 0}$	$\mathbb{R}$ $[0, 1]$, if $\varphi(x_1, \ldots, x_n)$	*	Y	N
I^β_{Kolm}	$\mathbb{R}^n$	$\mathbb{R}^{\geq 0}$	+	Y	Y

DOI: 10.1002/smr

allows to compare different partitioning of the population and see which one better explains the inequality in the measure, for example, is it the programming languages, the subsystems, the outsourced developers? As an example, using R (the ratio of the inequality between the groups and the total amount of inequality), and I_{Theil}, expenditure in Indonesian households [34] has been shown to be better explained by the education level of the head of the household than by the province of residence or by the gender of the household's head. Similarly, it has been observed that inequality in file sizes (SLOC) of the Linux Debian *Lenny* distribution can be better explained by the distribution package these files belong to, rather than the implementation language, or the distribution package maintainer [7]. This suggests that if one would like to reduce this inequality, that is, distribute functionality across the units in a more egalitarian way, one should focus on establishing cross-package size guidelines first.

Different approaches to decomposability [22, 33, 35–37] can be found in the scientific literature. Decomposability is typically accomplished by expressing the aggregation result computed at a system level as the sum of a non-negative 'within-group' term and a non-negative 'between-group' term, that is, $I = I^{\text{between}} + I^{\text{within}}$ given a decomposable inequality index I and a mutually exclusive and completely exhaustive (MECE) partitioning $\mathcal{G} = \{G_1, \ldots, G_m\}$. The 'within-group' contribution I^{within} is itself a weighted sum of applying I at the subcomponent level, such that the sum of the weighting coefficients is 1, that is, $I^{\text{within}} = \sum_{i=1}^{m} w_i I(G_i)$, $\sum_{i=1}^{m} w_i = 1$.

The 'between-group' term can be used to measure to what extent the aggregated value at the system level can be explained by a specific partitioning of the system into subsystems [7, 24], using the R index [22]. For I and $\mathcal{G}$ as above, the R index is defined as the ratio of the inequality between the groups and the total amount of inequality, that is, $R(\mathcal{G}) = \frac{I^{\text{between}}(\mathcal{G})}{I(x_1, x_2, \ldots, x_n)}$.

R indicates what share of the inequality can be explained by the partitioning into $\{G_1, \ldots, G_m\}$, and it ranges between 0 and 1. $R = 0$ in case of a trivial partition of the population into one group, that is, inequality is completely attributed to inequality within the group. $R = 1$ corresponds to the case when the partition is 'complete', that is, every element of the population is considered a group in itself.

It should be noted that although decompositions of I_{Gini} and $I^{\alpha}_{\text{Atkinson}}$ have been proposed in the literature [38], these do not adhere to the definitions above, hence are not recorded in Table V.

3. REQUIREMENTS FOR SOFTWARE QUALITY ASSESSMENT

As noticed by Rosenberg [39], when metrics are used to evaluate projects, there is no guideline to interpret their results. Often qualifying the result is based on common sense and experience. Determining what is an acceptable value depends on enterprise requirements and developer experience. For example, some companies require that depth of inheritance does not exceed a given threshold, while others focus on the general architecture or on the use of naming standards.

Therefore, we stress that a quality model must take into account organization-specific practices and requirements. Moreover, it should try to give a useful measure of quality that managers and developers can use to take corrective actions.

We now identify requirements for a successful aggregation technique, based on the Squale experience in industry and the issues raised in the previous sections. These requirements will be categorized as 'must', 'should', and 'could' to illustrate their varying importance (cf. [40]).

'Must' requirements are imposed by our perception of low-level metric values' combination as a sequence of two steps, *composition* and *aggregation*; 'should' and 'could' requirements were based on properties of aggregation techniques found in the literature and our experience with using the Squale model in industry.

Must:

- *Aggregation*: Must aggregate low level quality results (from the level of individual software components like classes or methods) at a higher level (e.g., a subsystem or an entire project) to evaluate the quality of an entire project, as discussed in Section 2.2;

DOI: 10.1002/smr

– *Composition*: Must compose different metric values with different ranges to a single quality interval, as explained in Section 2.1;

– *Composition/Aggregation Range and Domain*: Whether composition occurs before aggregation (as recommended in Section 2), or the opposite, the range (output) of the first must be compatible with the domain (input) of the second. For example, if the aggregation formula contains a logarithm, the composition method must have strictly positive range;

Should:

– *Highlight problems*: Should be more sensitive to problematic values to pinpoint them, and also to provide a stronger positive feedback when problems are corrected, as discussed in Section 2;

– *Do not hide progress*: Improvement in quality should never result in a worsening of the evaluation (e.g., Sections 2.2.1 and 2.2.2). As a counter example, it is known that econometric inequality indices will worsen when going from an 'all equally-bad' situation to a situation where all are equally bad except one;

– *Decomposability*: Should be decomposable (as discussed in Section 2.3.1) to measure to what extent the aggregated value at the system level can be explained by a specific partitioning of the system into subsystems [7, 24];

– *Composition before Aggregation*: Composition should be performed at the level of individual components to retain the intended semantic of the composition (see discussion in the beginning of Section 2);

– *Aggregation range*: Should be in a continuous scale, preferably bounded (i.e., left and right-bounded) (see Section 2.1);

– *Symmetry*: The final result should not be dependent on the order of the elements being aggregated (see Section 2.3.1). This requirement is typically not applicable for composition, because, for example, one can hardly expect a composition function f defined on size s and cyclomatic complexity v to satisfy $f(s, v) = f(v, s)$;

Could:

– *Evaluation normalization*: Could normalize all results (metrics, combination, aggregation) to allow unified interpretation at all levels (see Section 2.1);

– *Invariance* and *translatability*: Both invariance and translatability are interesting, for example, for SLOC, if the same header (containing licensing information) is added to all classes (invariance w.r.t. addition and translatability), or if percentages of the total SLOC are considered rather than the number itself (invariance w.r.t. multiplication).

4. THE SQUALE MODEL

We now introduce the Squale model, a software quality model developed empirically with the collaboration of large companies to answer the requirements set in Section 3. Squale is a quality model targeting both developers and managers. To give a coherent answer to the different needs and audience, the Squale model is inspired from the FCM model [2].

Currently, 100 projects are being monitored by Squale at Air France-KLM, 20 of which are actively using Squale to improve the quality of their source code. Overall, Squale monitors about seven Million lines of code (MLOC). Squale has also been used at PSA Peugeot-Citroen for the last 2 years. In the first year, it monitored about 0.9 MLOC distributed over 10 Java applications. Currently, it realizes around 640 audits and monitors about 10 MLOC dispatched in 90 Java applications with 350 modules. Each team sets its own quality requirements that are translated into composition formulas for the practices it chooses.

DOI: 10.1002/smr

4.1. *Definitions*

The Squale Model considers two groups of marks: (i) low-level (i.e., measures) and (ii) high-level (i.e., practices, criteria and factors) [41]. Each computed low-level mark gives a result in its own range, while high-level marks are all normalized to $[0, 3]$. To ease interpretation, it is generally assumed that in Squale, $[0, 1[$ maps to 'goal not achieved'; $[1, 2[$ maps to 'goal mostly achieved'; and $[2, 3]$ maps to 'goal achieved'. As opposed to FCM (or goal-question-metric), transforming individual results into global marks involves a new level between criteria and metrics introduced by the Squale model and called *practices*. Practices are the level in the model where low-level metric results are transformed into normalized marks (composition), and aggregated over multiple components.

- Low-level marks:
  - A *measure* is a raw piece of information extracted from the project data. It comes from human expertise (manual measures) or from different tools (raw metrics, e.g., code metrics, rule checking metrics, or test metrics). Currently, the Squale model uses a number of raw metrics (ranging from 50 to 100), depending on the project being analyzed, the development stage, and the manual audits performed.
- High-level marks:
  - A *practice* assesses whether a technical principle in the project is respected.[3] Composition and aggregation of low-level marks occur at this level. It is addressed to developers, in terms of good or bad properties with respect to the project quality. Practices are primarily computed at the level of the entire system (aggregation), but one can also look at them at a lower abstraction level (e.g., class) to dig out the causes of an unsatisfying quality assessment. There are around 50 practices already defined based on Air France-KLM quality standards, but the list of practices remains open [42].
  - A *criterion* assesses one main component of software quality (e.g., the criterion *Simplicity* assesses the source code readability and the ease to diagnostic regardless of documentation). A criterion is addressed to managers, at a more fine-grained detail level than factors. The criteria used in the Squale model are adapted to face the special needs of Air France-KLM and PSA Peugeot-Citroen. In particular, they are tailored for the assessment of quality in information systems.
  - A *factor* represents the highest-level quality assessment, used to provide an overview of a project's health. It is addressed to nontechnical persons. Factors correspond roughly to the characteristics of the ISO 9126.

Note that factor and criterion are not further detailed in this paper. The current implementation (i.e., in the companies using Squale) is based on a simple average of practices for criterion and a simple average of criteria for factor. This is not a carved in stone, only Squale's clients did not express the need for more elaborate composition techniques[4] at these levels of abstraction. Because composition and aggregation of metrics occur at the practices level, the remainder of this section is dedicated to it.

4.2. *Composition/aggregation of metrics*

The Squale model uses low-level marks to compute high-level marks at the practice level. This process is conducted in two steps (composition + aggregation). We distinguish between *Individual Marks* (IM) computed from raw metrics (at the level of components), and *Global Marks* computed from individual

[3]Practices as combinations of metrics are similar to *detection strategies* [55]. Detection strategies aim, however, at identification of problematic code fragments, that is, values of detection strategies are binary: either the code fragment is problematic or it is not. Practices generalize detection strategies by extending the range of possible values to $[0, 1]$. Moreover, practices are not limited to filtering and composition as defined for detection strategies.

[4]Factors and criteria are composed of, respectively, criteria and practices.

DOI: 10.1002/smr

marks (for the entire project) [41]. Manual measures, whenever used, are directly expressed as global marks.

- *Composition*: Metrics used to assess a practice can be composed, for example, by:
    - Simple or weighted averaging of the different values of the metrics. This is only possible when the different metrics have similar range and semantic;
    - Thresholding on one metric such as *cyclomatic complexity* to consider or not the other metrics, for example, when cyclomatic complexity is more than 50, one could decide to divide the number of lines of comment by some value to highlight the fact that overly complex methods need to be overly commented;
    - Interpolating, given example components by the developers and their perceived evaluation of quality (e.g., one method with 50 LOC would be perceived of quality 2.5 — on an interval of $[0, 3]$— and another example with 100 LOC would be perceived of quality 1.5), one can interpolate a function to convert other values;
    - A combination of these methods. For instance, the 'Number of methods' practice [42] relates complexity of the class $CC(C)$, defined as the sum of the cyclomatic complexities of the class methods, to the number of class methods $NOM(C)$:

$$IM(C) = \begin{cases} 2^{\frac{30-NOM(C)}{10}} & \text{if } CC(C) \geq 80 \\ 2 + \dfrac{20-NOM(C)}{30} & \text{if } 50 \leq CC(C) < 80 \text{ and } NOM \geq 15 \\ 3 + \dfrac{15-NOM(C)}{15} & \text{if } 30 \leq CC(C) < 50 \text{ and } NOM \geq 15 \\ 3 & \text{otherwise} \end{cases}$$

The result of the composition of metrics values for a practice is called IM. Individual marks for a practice are computed from raw metrics with multiple ranges, and constitute single marks in the range $[0, 3]$. The raw metrics composed may have multiple ranges.

- *Aggregation*: Aggregation of IMs for a practice requires several steps (illustrated with an example in Figure 1; the dark dots on the *x*-axis are the IMs to be aggregated — 0.5, 1.5, and 3):
    1. A weighting function is applied to each IM: $g(IM) = \lambda^{-IM}$ where IM is the individual mark and λ the constant defining the 'hard', 'medium', or 'soft' weighting. Hard weighting

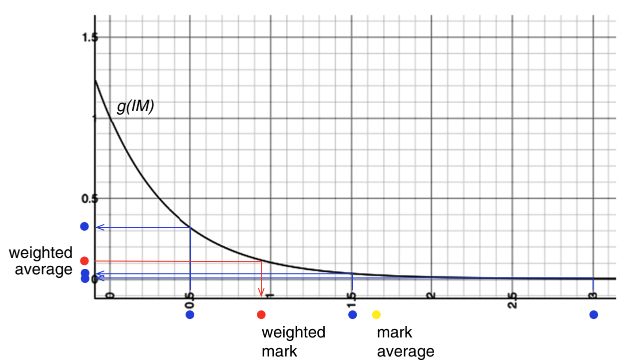

Figure 1. Computing the weighted average in Squale (here $\boldsymbol{\lambda = 9}$). First, points 0.5, 1.5, and 3 (darker dots on *X*-axis) are weighted (darker dots on *Y*-axis), then these are averaged (lighter dot on *Y*-axis), and this average is converted back to the $\mathbf{[0, 3]}$ interval (lighter dot on *X*-axis, close to 1). The very light grey dot on *X*-axis, past 1.5, shows the arithmetic mean of the three initial values.

gives more weight to bad results than soft weighting. λ is greater for a hard weighting and smaller for a soft one.[5] This formula translates individual marks into a new space where low marks may have significantly more weight than others. In Figure 1, weighted IMs are the dark dots on the Y axis, assuming a medium weighting ($\lambda = 9$);

2. Second, we average the weighted marks. The result thus reflects the greater weight of the low marks (lighter dot on the Y axis, slightly above 0.1);

3. Third, we compute the inverse function $g^{-1}(Wavg(IMs)) = -\,log_{\lambda}(Wavg(IMs))$ on the average, to return to the range $[0, 3]$ (lighter dot on the X axis, at 0.93). The $Wavg(IMs)$ is the weighted average of the IMs. For comparison, the arithmetic average of the initial values is given in very light grey (at 1.67).

Therefore, the global mark of a practice (for n components) is computed as $GM^{\lambda} = -log_{\lambda}\left(\frac{1}{n}\sum_{i=1}^{n} \lambda^{-IM_i}\right)$, where λ varies between hard ($\lambda = 30$), medium ($\lambda = 9$), and soft ($\lambda = 3$) weights.

For comparison, the global mark for the three IMs considered here (0.5, 1.5, 3) computed with arithmetic mean, soft, medium and hard weights are 1.67, 1.19, 0.93, and 0.81, respectively. The figure suggests that the aggregated values can never be lower than the smallest of the IMs (0.5) and can never exceed the arithmetic mean (1.67). The following theorem proves that this is indeed the case, that is, the global mark is never less sensitive to the undesirable low values than the arithmetic mean. For consistency with the inequality indices discussed in Section 2.3, we will denote the Squale aggregation function (GM^{λ}) as $I^{\lambda}_{\text{Squale}}$.

Theorem 1
Let $x_1, \ldots, x_n$ be real numbers and let $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$.

Then for $\lambda > 1$

$$\min(x_1, \ldots, x_n) \le I^{\lambda}_{\text{Squale}}(x_1, \ldots, x_n) \le \bar{x}.$$

Proof
Because $\min(x_1, \ldots, x_n) \le x_i$ for all $1 \le i \le n$, then it also holds that $-x_i \le -\min(x_1, \ldots, x_n)$. Because $\lambda > 1$ it holds that $\lambda^{-\bar{x}_i} \le \lambda^{-\min(x_1,\ldots,x_n)}$ for all i.

Therefore,
$$\sum_{i=1}^{n} \lambda^{-x_i} \le n\lambda^{-\min(x_1,\ldots,x_n)} \equiv \frac{1}{n}\sum_{i=1}^{n} \lambda^{-x_i} \le \lambda^{-\min(x_1,\ldots,x_n)} \equiv$$
$$\log_{\lambda}\left(\frac{1}{n}\sum_{i=1}^{n} \lambda^{-x_i}\right) \le -\min(x_1, \ldots, x_n) \equiv \min(x_1, \ldots, x_n) \le I^{\lambda}_{\text{Squale}}(x_1, \ldots, x_n)$$

Now, the geometric mean never exceeds the arithmetic mean, that is, $\sqrt[n]{\prod_{i=1}^{n} \lambda^{-x_i}} \le \frac{1}{n}\sum_{i=1}^{n} \lambda^{-x_i}$.
However, $\sqrt[n]{\prod_{i=1}^{n} \lambda^{-x_i}} = \lambda^{-\frac{1}{n}\sum_{i=1}^{n} x_i} = \lambda^{-x}$. Hence, $\lambda^{-\bar{x}} \le \frac{1}{n}\sum_{i=1}^{n} \lambda^{-x_i}$
Because $\lambda > 1$, $-\bar{x} \le \log_{\lambda}\left(\frac{1}{n}\sum_{i=1}^{n} \lambda^{-x_i}\right) \equiv -\log_{\lambda}\left(\frac{1}{n}\sum_{i=1}^{n} \lambda^{-x_i}\right) \le \bar{x} \equiv I^{\lambda}_{\text{Squale}}(x_1, \ldots, x_n) \le \bar{x}$

4.3. Properties of the Squale model

Next, we discuss the properties of the Squale model, given the requirements in Section 3.

- *Aggregation*: This requirement is satisfied by the computation of the global marks;
- *Composition*: This requirement is satisfied by the computation of the individual marks;

[5]We typically use the values: hard $\lambda = 30$, medium $\lambda = 9$, and soft $\lambda = 3$.

DOI: 10.1002/smr

- *Highlight problems*: For calculation of the individual marks, satisfaction of this requirement depends on the function used to determine the IMs. For calculation of the global mark we refer back to showing that $I^{\lambda}_{\text{Squale}}$ gives more weight to low individual marks than the arithmetic mean for all three weighting coefficients above. In Section 5 we reconsider this requirement by means of an experiment;
- *Do not hide progress*: We prove in Section 5.1 (Theorem 4) that Squale satisfies this requirement.
- *Composition before Aggregation*: Squale applies aggregation on the result of the composition;
- *Composition range*: The IMs' range is $[0, 3]$, which is compatible with the definition of the aggregation function $I^{\lambda}_{\text{Squale}}$;
- *Aggregation range*: The aggregation range is defined in Squale to be $[0, 3]$;
- *Symmetry*: $I^{\lambda}_{\text{Squale}}$ satisfies this requirement;
- *Evaluation normalization*: The set of all possible IMs and GMs is defined to be $[0, 3]$.
- *Invariance* and *translatability*: Theorem 2 shows that $I^{\lambda}_{\text{Squale}}$ is translatable for any $\lambda \in \mathbb{R}$, $\lambda \geq 0, \lambda \neq 1$. Therefore $I^{\lambda}_{\text{Squale}}$ is neither additively nor multiplicatively invariant.

Theorem 2
Let $x_1, \ldots, x_n$ be real numbers. Then for any $\lambda \in \mathbb{R}, \lambda \geq 0, \lambda \neq 1$ we have $I^{\lambda}_{\text{Squale}}(x_1 + c, \ldots, x_n + c) = I^{\lambda}_{\text{Squale}}(x_1, \ldots, x_n) + c$.

Proof
To see that the theorem holds, observe that

$$\begin{aligned} I^{\lambda}_{\text{Squale}}(x_1 + c, \ldots, x_n + c) &= -\log_{\lambda}\left(\frac{1}{n}\sum\nolimits_{i=1}^{n} \lambda^{-(x_i+c)}\right) = -\log_{\lambda}\left(\frac{1}{n}\sum\nolimits_{i=1}^{n} \left(\lambda^{-x_i}\lambda^{-c}\right)\right) \\ &= -\log_{\lambda}\left(\frac{\lambda^{-c}}{n}\sum\nolimits_{i=1}^{n} \lambda^{-x_i}\right) = -\left(\log_{\lambda}\left(\frac{1}{n}\sum\nolimits_{i=1}^{n} \lambda^{-x_i}\right) + \log_{\lambda}\lambda^{-c}\right) \\ &= -\left(\log_{\lambda}\left(\frac{1}{n}\sum\nolimits_{i=1}^{n} \lambda^{-x_i}\right) + (-c)\right) = -\log_{\lambda}\left(\frac{1}{n}\sum\nolimits_{i=1}^{n} \lambda^{-x_i}\right) + c \\ &= I^{\lambda}_{\text{Squale}}(x_1, \ldots, x_n) + c \end{aligned}$$

- *Decomposability*: Theorem 3 shows that $I^{\lambda}_{\text{Squale}}$ is not decomposable.

Theorem 3
Theorem 3. $I^{\lambda}_{\text{Squale}}$ is not decomposable according to Section 2.3.2.

Proof
Recall that for an aggregation to technique I to be decomposable according to Section 2.3.2 for any collection of real numbers $x_1, x_2, \ldots, x_n$ and any MECE partitioning $= \{G_1, G_2, \ldots, G_m\}$ it should satisfy

1. $I = I^{\text{between},\mathcal{G}} + I^{\text{within},\mathcal{G}}$
2. $I^{\text{between},\mathcal{G}} \geq 0$
3. $I^{\text{within},\mathcal{G}} = \sum_{i=1}^{m} w_i I(G_i)$
4. $\sum_{i=1}^{m} w_i = 1$

Assume for the sake of contradiction that I_{Squale} is decomposable. Then, I_{Squale} is decomposable for a collection $\mathcal{X}$ consisting of n equal numbers x, with $x > 0$, and the MECE partitioning $\mathcal{G}$ that places

DOI: 10.1002/smr

each number in its own group. Recall from Section 2.3.2 that $R = 1$ for partitions that consider every element of the population as a group in itself. Hence, $R^{\mathcal{X},\mathcal{G}} = 1$. By definition of R, $R = I^{\text{between}}_{\text{Squale}}\left(\frac{\mathcal{G}}{I_{\text{Squale}}(\mathcal{X})}\right)$ and $R = 1 - \frac{I^{\text{within}}_{\text{Squale}}(\mathcal{G})}{I_{\text{Squale}}(\mathcal{X})}$ because $I = I^{\text{between}} + I^{\text{within}}$. Thus $\frac{I^{\text{within}}_{\text{Squale}}(\mathcal{G})}{I_{\text{Squale}}(\mathcal{X})} = 0$ and, because $I_{\text{Squale}}(\mathcal{X}) \geq x > 0$ (from Theorem 1), $I^{\text{within}}_{\text{Squale}}(\mathcal{G}) = 0$.

However, $I^{\text{within}}_{\text{Squale}} = \sum_{i=1}^{n} w_i I_{\text{Squale}}(\{x\}) = \sum_{i=1}^{n} w_i x$ (by (3)). Because $x > 0$ it follows that $w_i = 0$ for all $1 \leq i \leq n$, and hence, $\sum_{i=1}^{m} w_i = 0$, contradicting (4).

Therefore, our assumption was incorrect and I_{Squale} is not decomposable according to Section 2.3.2.

5. EVALUATION

In this section, we compare Squale theoretically and empirically to a popular aggregation technique, the arithmetic mean [43], and to econometric inequality indices, the most recent trend in aggregation of software metrics [5–9, 11]. We perform our evaluation along two lines. First, we exploit a close theoretical relation between Squale and I_{Kolm} in Section 5.1, and infer an additional mathematical property of Squale. Later, we empirically compare the sensitivities of Squale, the arithmetic mean and the inequality indices to bad values, in Section 5.2.

5.1. Theoretical comparison

The relation between $I^{\lambda}_{\text{Squale}}$ and the arithmetic mean has been established in Theorem 1. Next we show that Squale is closely related to I_{Kolm} [29] in Lemma 1.

Lemma 1
$I^{\log\lambda}_{\text{Kolm}}(x_1, \ldots, x_n) + I^{\lambda}_{\text{Squale}}(x_1, \ldots, x_n) = \bar{x}$

Proof
To see that the theorem holds, observe that

$$\begin{aligned}
I^{\log\lambda}_{\text{Kolm}}(x_1, \ldots, x_n) + I^{\lambda}_{\text{Squale}}(x_1, \ldots, x_n) &= \frac{1}{\log\lambda}\log\left(\frac{1}{n}\sum_{i=1}^{n} e^{\log\lambda(\bar{x}-x_i)}\right) - \log_{\lambda}\left(\frac{1}{n}\sum_{i=1}^{n} \lambda^{-x_i}\right) \\
&= \frac{1}{\log\lambda}\log\left(\frac{1}{n}\sum_{i=1}^{n} \lambda^{\bar{x}-x_i}\right) - \frac{1}{\log\lambda}\log\left(\frac{1}{n}\sum_{i=1}^{n} \lambda^{-x_i}\right) \\
&= \frac{1}{\log\lambda}\left(\log\left(\frac{1}{n}\lambda^{\bar{x}}\sum_{i=1}^{n} \lambda^{-x_i}\right) - \log\left(\frac{1}{n}\sum_{i=1}^{n} \lambda^{-x_i}\right)\right) \\
&= \frac{1}{\log\lambda}\left(\log\frac{\frac{1}{n}\lambda^{\bar{x}}\sum_{i=1}^{n} \lambda^{-x_i}}{\frac{1}{n}\sum_{i=1}^{n} \lambda^{-x_i}}\right) = \frac{1}{\log\lambda}\left(\log\lambda^{\bar{x}}\right) = \bar{x}
\end{aligned}$$

In addition to establishing a relation between I_{Kolm} and I_{Squale}, it allows us to prove the following important property of Squale. On the basis of Lemma 1, Theorem 4 proves that Squale guarantees that subsequent improvements in quality are reflected in the aggregated quality assessment result. For instance, if source lines of code values measured per method are considered undesirable when greater than 36 [10], then a decrease in SLOC of 20 for one method with SLOC 60 at the cost of an equivalent increase in SLOC for another method with SLOC 20 would result in an increase of quality as measured by I_{Squale}. We denote this property of I_{Squale} the 'anti-transfers principle', in analogy to the 'transfers principle' [29] satisfied by various inequality indices [24] including I_{Kolm}.

Theorem 5

Let $x_i < x_j$ and let $\delta > 0$ be such that $x_i + \delta \le x_j - \delta$. Then, $I^{\lambda}_{\mathrm{Squale}}$ satisfies the 'anti-transfers principle', that is, $I^{\lambda}_{\mathrm{Squale}}(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n) < I^{\lambda}_{\mathrm{Squale}}(x_1, \ldots, x_i + \delta, \ldots, x_j - \delta, \ldots, x_n)$.

Proof

I_{Kolm} is known to satisfy the transfers principle [29], that is, for any β it holds that $I^{\beta}_{\mathrm{Kolm}}(x_1, \ldots, x_i, \ldots, x_j, x_n) > I^{\beta}_{\mathrm{Kolm}}(x_1, \ldots, x_i + \delta, \ldots, x_j - \delta, \ldots, x_n)$, for x_i, x_j, δ as above.

From Lemma 1 we have $I^{log\lambda}_{\mathrm{Kolm}}(x_1, \ldots, x_n) = mean(x_1, \ldots, x_n) - I^{\lambda}_{\mathrm{Squale}}(x_1, \ldots, x_n)$, and $I^{log\lambda}_{\mathrm{Kolm}}(x_1, \ldots, x_i + \delta, \ldots, x_j - \delta, \ldots, x_n) = mean(x_1, \ldots, x_i + \delta, \ldots, x_j - \delta, \ldots, x_n) -$ $I^{\lambda}_{\mathrm{Squale}}(x_1, \ldots, x_i + \delta, \ldots, x_j - \delta, \ldots, x_n) = mean(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n) -$ $I^{\lambda}_{\mathrm{Squale}}(x_1, \ldots, x_i + \delta, \ldots, x_j - \delta, \ldots, x_n)$. The claim follows. □

Proving Theorem 4 allows us to summarize the requirements reach by the econometric inequality indices. First, one must remember that inequality indices do not constitute full quality models, as opposed to Squale, and as such were not designed with these requirements in mind. Thus, if they have already been used as aggregation technique, they are not intended to be composition techniques although they clearly can be applied both to individual metrics and to practices obtained after the composition step. They may also hide progress but only in extreme situations. Indeed they may decrease when switching from an 'all equally-bad' situation to an 'one good, all others equally-bad'. We will consider in more detail how well they can highlight problems in the experimental evaluation. Some of them (see Section 2.3) do satisfy the decomposability requirement, which is not the case for Squale. Other requirements such as Symmetry, Invariance or Translatability were already discussed (see Table V).

5.2. Experimental evaluation

As mentioned in Section 3, a successful software quality model must aggregate metrics in a normalized range and highlight bad components to warn the software engineers in case of potential problems. We already explained in Section 4.3 that Squale does attend to this requirement, but we wish to understand better its sensitivity to problems and how it compares to other aggregation techniques.

5.2.1. Experimental setup. To better evaluate how sensitive Squale and other aggregation techniques are to problems, we compare their reactions in the presence of an increasingly larger amount of problems. We use a controlled experiment where the amount of problem may be quantitative or qualitative, and we consider two independent variables:

- Quantity of problems in a known quantity of good results;
- Quality of the problems (badness degree) in a known quantity of perfect results.

The dependent variable is the final result of the aggregation technique. The treatments are the different aggregation techniques: $I^{\lambda}_{\mathrm{Squale}}$ with $\lambda = 3$, 9, and 30, I_{Theil}, I_{MLD}, I_{Gini}, $I^{\alpha}_{\mathrm{Atkinson}}$, $I^{\beta}_{\mathrm{Kolm}}$, I_{Hoover}. We assume standard instantiations [44] for $I^{\alpha}_{\mathrm{Akinson}}$ and $I^{\beta}_{\mathrm{Kolm}}$ for $\alpha = 0.5$ and $\beta = 1$, respectively.

One problem with such experiment is to find a suitable case study. Another issue is to find a system that can provide the needed variation in quantity and degree of bad results. For convenience, we chose to use ECLIPSE 2.0[6] as the test bed for the experiment. We will aggregate the individual marks of the *method size* practic, which is based on the sole SLOC metric. We chose a practice based on only one metric to enable comparison with econometric aggregation techniques that do not offer composition mechanisms by default. Finally, we normalize the raw results of the metric to the $[0, 3]$ interval as defined in Squale, even though the econometric indices do not require this step. The normalization function will be the one defined for Air France-KLM (given in Table I).

The exact set-up of the experiment is the following: The system has a total of 8612 methods, from which 8093 have a mark of 3. The base, 'perfect', case consists of these 8093 methods. Actually for

[6]Ref: http://eclipse.org

this perfect case, the number of components is irrelevant because they all have the same evaluation. For the 'quantity of problems' independent variable, we work with 8612 methods containing a given proportion of imperfect methods. This proportion will vary from 10% to 100% in steps of 10%. For example, for the test with 10% imperfect methods, we have a random selection of 7751 perfect methods and 861 imperfect ones. When we need more imperfect methods than the system actually contains, we allow selecting the same ones several times. For the 'quality of problems' independent variable, we choose components with IMs in the intervals: $[2,3[; [1,2[; [0.5,1[; [0.1,0.5[$; and $[0,0.1[$. These intervals were chosen to have a fine-grained understanding of what happens with bad results. For each treatment, the experiment will consist of the Cartesian product of all values for the two independent variables. Furthermore, because each experiment involves randomly selecting the imperfect components, we repeat it 10 times and present the mean of the 10 results.

5.2.2. Results. Figure 2 presents the results for all the aggregation methods. The first graph (top left), gives the results for the arithmetic mean. It shows that even with 30% very bad marks (imperfect methods in $[0,0.1[$), the aggregated result is still ≥ 2, which would still indicate a good quality.

The results of this first graph are repeated in all other graphs in the form of a grey triangle in the background, to ease comparing all other aggregation techniques to the upper bound and lower bound of the results for arithmetic mean.

The results for I_{Squale} show that it behaves as expected, with the gradation of the different weights (from soft $\lambda = 3$ to hard $\lambda = 30$). In particular, hard weighting does give a low aggregated result even for a small quantity (10%) of bad marks. For medium and hard weighting, after a sudden drop (10% bad marks) the curves show a milder slope, suggesting that Squale is less sensitive to 20% or more bad marks than to the first 10%. This could be a problem because one cannot know beforehand whether there are only a few or many problems. Moreover, although it does not break the 'Do not hide progresses' requirement, going from 90% to 30% bad marks shows very little improvement in the final mark. In this sense, a more linear aggregation technique like a softer weighting or the simple arithmetic mean might be preferable in extreme situations when the quality of the system is completely unknown (first run), or when the system has very low quality. One must, therefore, strike a balance between the requirements 'Highlight problems' and 'Do not hide progress'.

For the econometric indices, one must remember that they are inequality measures, and therefore would normally give low results for aggregated values all equal (e.g., base case). This characteristic is the opposite of what we are looking for. Therefore, to ease comparison with Squale, we inverted the *Y*-axis of their results (on the left of the graphs; the right *Y*-axis is for the grey triangle referring to arithmetic means).

I_{Theil} was described as being biased toward 'rich people' (higher values) [24], that is, I_{Theil} should be more sensitive going from, for example, 90% to 80% perfect marks than from 30% to 20% perfect marks. However, our experiments suggest that I_{Theil} is the aggregation technique that least highlights bad results ('poor people'), even less than the arithmetic mean.

In contrast, I_{Kolm} is the inequality index that best behaves as required with respect to highlighting bad results, as long as there are not too many of them (up to 30% or 40%). It can be observed that when the proportion of bad result increases, there is less inequality and therefore I_{Kolm} decreases (curve going up on our inverted axis). However, it not a disadvantageous characteristic, especially because software assessed in an industrial context almost never have components exceed 40 % imperfect marks for the same measure. However, more worrying for I_{Kolm} is the fact that an improvement of the quality (for example from 60% to 50% imperfect marks) will also result in an augmentation of inequality (from a majority of imperfect methods to less) and, therefore, a worsening of the aggregated value. Some work would be needed to improve this aspect, but one must not forget that we are considering here artificial data with only a limited range of imperfect marks (e.g.,$[0.5,1[$), whereas on real projects they would be more spread out. One must also remember that the aggregation is performed here on normalized SLOC results into $[0,3]$, which limits the possible inequalities therefore confining the possible values for the inequality indexes.

5.2.3. Threats to validity. We identified the following threats to validity:

- The experiment was conducted on a single software system with a single metric. However, because the aggregation results are based only on the numerical values of the metrics for this

DOI: 10.1002/smr

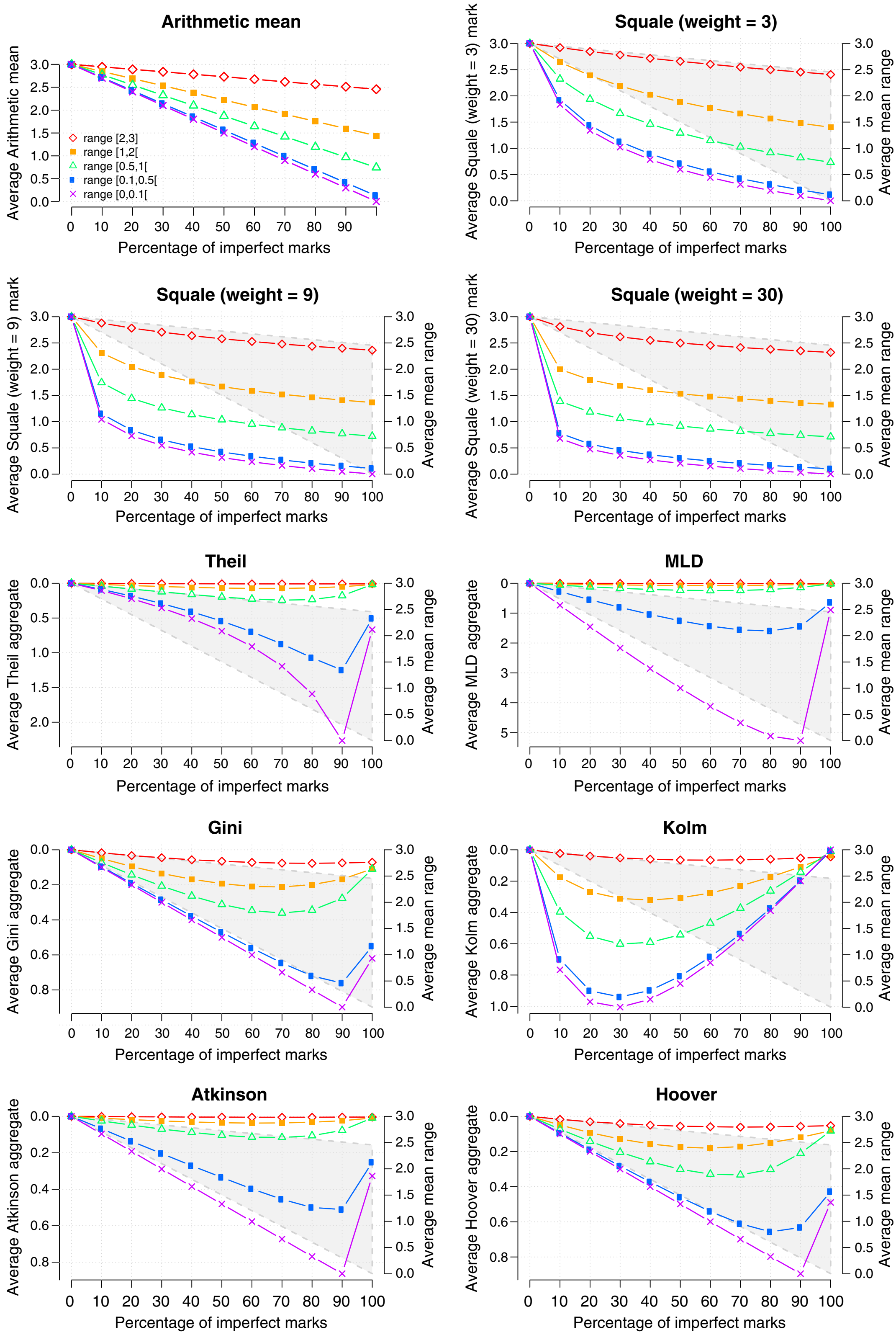

Figure 2. Results of experiments for all aggregation indexes (see text for explanation). The topmost left figure displays the common legend.

DOI: 10.1002/smr

system's components, this fact has little bearing and can be ignored. Our experimentation validates only aggregation method, not composition equations.

- The data are artificial and do not represent a real case where different quantities of problems with varying quality would be found. However, this setup was necessary to finely analyze the response of each aggregation technique to a varying amount of problems. This issue is inherent to controlled experiments.
- We used only one metric (SLOC), and its results were normalized to the $[0, 3]$ interval. These two restrictions were required to be able to compare on the same ground the Squale model and the different econometric inequality indices. With real values, having a larger range, econometric inequality indices could have performed better because they would have reacted more strongly to larger differences. However, we already argued that in a real evaluation context it would usually make more sense to compose metrics before aggregating them, and composition will often result in some normalization of the metrics' values to smoothen the differences in ranges. It is therefore not an unrealistic setting.

6. RELATED WORK

Software metrics are essential to understanding whether the quality of the software we are building corresponds to our expectations [1]. Not surprisingly, the scientific community has amassed a huge literature on software metrics, including such works as [1, 45–47]. In the following we focus solely on the studies of metrics aggregation and/or composition.

Composition of different metrics has been used to assess software maintainability in metrics such as the maintainability index [31] or modularization quality [48]. Moreover, composition of different metrics is common in applications of software quality models such as [49–51]. Both [50] and [51] aim at predicting software defects with regression formulas based on Chidamber and Kemerer's metrics [52]. One of the problems with the approaches based on linear regression is related to the linear character of the dependency between the dependent and independent variables, that is, increasing of DIT with 3 always has according to the formula of [51] the same effect on the number of defects irrespectively of the original value of the metrics. This contradicts an intuitive expectation that DIT of a given class increasing from 4 to 7 should have a more adverse effect on the number of defects than increasing it from 1 to 3. Software quality models such as in [49] are frequently threshold based, and hence, frequently suffer from the staircase effect discussed in Section 2.1. Squale addresses both shortcomings by introducing nonlinear relations between independent (metrics) and dependent (marks) variables.

Aggregation of metrics values obtained for the same metric and different artifacts constitutes the second step in the application of Squale. The need to aggregate information from smaller elements (functions or methods) to larger elements (packages) has been recognized early on. Traditional approaches [16] use the arithmetic mean. Another popular approach [19,53] consists in selecting a known family of distributions and fitting its parameters to approximate the metric values observed. They were both discussed in Section 2.

7. DISCUSSION AND CONCLUSION

Measuring the quality of their software projects is important for organizations that want to keep control of their systems. If there are numerous software quality metrics available to measure the varying aspect of the quality of software, these metrics are defined at a low level of individual components: functions, methods, classes, whereas developers need a global view at the level of an entire system. In this paper we identified practical issues with the existing aggregation methods when used on real projects, including: the need for composing metrics with different ranges (e.g., DIT $\in [0, 10]$ and SLOC $\in [0, 1000]$); the need to aggregate quality assessment of many components; or, the need to highlight bad results that need be corrected. We then presented Squale, a quality model defined empirically on concrete projects in large companies (Air France-KLM, PSA Peugeot-Citroen) to answer these requirements. We also discussed the possible use of econometric indexes to aggregate individual quality results as proposed in recent literature [7,8]. After discussing the theoretic properties of the different aggregation methods proposed, we experimented their ability to highlight bad results on ECLIPSE.

The results are that Squale satisfies most of the requirements identified with decomposability being the notable exception. For example, the experiments show that it does answer the requirement of highlighting bad results even if there is a small proportion of them.

The econometric indexes also answer most of the requirements. I_{Kolm} gives the most interesting results in the experiment, even if we identified some issues with the fact that it is an inequality measure, which means it can give good results when all low level quality assessments are bad because there is no inequality between them. However, this should not be an issue in practice because it is unlikely to occur. Because there is an important literature on econometric indexes, it might be interesting to continue studying them and see how they can be adapted to the needs of quality assessment. We suggest one area of research, noticing that the experiment we performed are artificial in the sense that the distribution of quality results for individual components is limited to two small intervals whereas in real life they could be much more spread out.

ACKNOWLEDGEMENT

Supported by the Dutch Science Foundation project ‘Multi-Language Systems: Analysis and Visualization of Evolution—Analysis’ (612.001.020).

REFERENCES

1. Pfleeger SL. Software metrics: Progress after 25 years? *Software, IEEE* 2008; **25**(6):32–34.
2. McCall J, Richards P, Walters G. *Factors in Software Quality*. NTIS: Springfield, 1976.
3. Basili VR. Software modeling and measurement: the goal/question/metric paradigm. *Technical Report*, College Park, MD, USA, 1992.
4. Martin RC. OO design quality metrics: An analysis of dependencies, October 1994. Available from: http://condor.depaul.edu/dmumaugh/OOT/Design-Principles/oodmetrc.pdf Consulted on January 11, 2009.
5. Vasilescu B, Serebrenik A, van den Brand MGJ. Comparative study of software metrics’ aggregation techniques. In *9th Belgian-Netherlands Softw. Evolution Seminar*, Ducasse S, Duchien L, Seinturier L (eds.). University of Lille-1: Lille, France, 2010; 1–5.
6. Vasilescu B, Serebrenik A, van den Brand MGJ. By no means: A study on aggregating software metrics. In *2nd International Workshop on Emerging Trends in Software Metrics*, Concas G, Di Penta M, Tempero E, Zhang H (eds.). ACM Press: New York, NY, USA, 2011.
7. Serebrenik A, van den Brand MGJ. Theil index for aggregation of software metrics values. In *Int. Conf. on Software Maintenance*. IEEE, 2010; 1–9.
8. Vasa R, Lumpe M, Branch P, Nierstrasz OM. Comparative analysis of evolving software systems using the Gini coefficient. In *Int. Conf. on Software Maintenance*. IEEE, 2009; 179–188.
9. Goeminne M, Mens T. Evidence for the Pareto principle in Open Source Software Activity. In *Proc. Int’l Workshop SQM 2011*. CEUR-WS workshop proceedings, 2011.
10. Mordal-Manet K, Laval J, Ducasse S, Anquetil N, Balmas F, Bellingard F, Bouhier L, Vaillergues P, McCabe T. An empirical model for continuous and weighted metric aggregation. In *15th Eur. Conf. Soft. Maintenance and Reeng*. IEEE, 2011; 141–150.
11. Vasilescu B, Serebrenik A, van den Brand MGJ. You can’t control the unfamiliar: A study on the relations between aggregation techniques for software metrics. In *Int. Conf. on Software Maintenance*. IEEE, 2011.
12. ISO/IEC. ISO/IEC 9126 software engineering –product quality–, 2003.
13. Wiegers KE. Software process improvement: Ten traps to avoid. *Software Development* 1996; **4**:51–58.
14. Likert R. A technique for measurement of attitudes. *Archives of Psychology* 1932; **140**:5–53.
15. Perepletchikov M, Ryan C, Frampton K, Tari Z. Coupling metrics for predicting maintainability in service-oriented designs. In *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*. IEEE Computer Society: Los Alamitos, California, USA, 2007; 329–340.
16. Lanza M, Marinescu R. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Verlag: Berlin Heidelberg, 2006.
17. Barkmann H, Lincke R, Löwe W. Quantitative evaluation of software quality metrics in open-source projects. In *Advanced Information Networking and Applications (WAINA’09). International Conference on*, IEEE, 2009; 1067–1072.
18. Turnu I, Concas G, Marchesi M, Pinna S, Tonelli R. A modified Yule process to model the evolution of some object-oriented system properties. *Information Sciences* February 2011; **181**:883–902.
19. Concas G, Marchesi M, Pinna S, Serra N. Power-laws in a large object-oriented software system. *IEEE Transactions Software Engineering* 2007; **33**(10):687–708.
20. Serebrenik A, Roubtsov S, van den Brand MGJ. D_n-based architecture assessment of Java open source software systems. In *ICPC ’09: Proc. 17th Int. Conf. on Program Comprehension, 2009*, IEEE, 2009; 198–207.

DOI: 10.1002/smr

21. Herraiz I. A statistical examination of the evolution and properties of libre software. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 2009; 439–442.
22. Cowell FA, Jenkins SP. How much inequality can we explain? a methodology and an application to the United States. *The Economic Journal* March 1995; **105**(429):421–30.
23. Cowell FA, Kuga K. Inequality measurement: An axiomatic approach. *European Economic Review* March 1981; **15**(3):287–305.
24. Cowell FA. Measurement of inequality. In *Handbook of Income Distribution*, volume **1**, Atkinson AB, Bourguignon F (eds.). Elsevier Science: Amsterdam, The Netherlands, 2000; 87–166.
25. Gini C. Measurement of inequality of incomes. *The Economic Journal* 1921; **31**:124–126.
26. Theil H. *Economics and Information Theory*. North-Holland Publishing Company: Amsterdam, The Netherlands, 1967.
27. Atkinson AB. On the measurement of inequality. *Journal of Economic Theory* 1970; **2**(3):244–263.
28. Hoover EM. The measurement of industrial localization. *The Review of Economic Statistics* 1936; **18**(4):162–171.
29. Kolm S-C. Unequal inequalities I. *Journal of Economic Theory* 1976; **12**(3):416–442.
30. Johnsonbaugh R. *Discrete mathematics*. Pearson Education, an imprint of Prentice Hall: Harlow, Essex, UK, 2001.
31. Oman P, Hagemeister J. Construction and testing of polynomials predicting software maintainability. *Journal of Systems and Software* 1994; **24**(3):251–266.
32. Foster JE. An axiomatic characterization of the theil measure of income inequality. *Journal of Economic Theory* October 1983; **31**(1):105–121.
33. Shorrocks AF. The class of additively decomposable inequality measures. *Econometrica* April 1980; **48**(3):613–625.
34. Akita T, Lukman RA, Yamada Y. Inequality in the distribution of household expenditures in Indonesia: A Theil decomposition analysis. *The Developing Economies* June 1999; **XXXVII**(2):197–221.
35. Parker SC. The inequality of employment and self-employment incomes: a decomposition analysis for the U.K. *Review of Income and Wealth* 1999; **45**(2):263–274.
36. Blackorby C, Donaldson D, Auersperg M. A new procedure for the measurement of inequality within and among population subgroups. *The Canadian Journal of Economics/Revue canadienne d'Economique* 1981; **14**(4):665–685.
37. Bourguignon F. Decomposable income inequality measures. *Econometrica* July 1979; **47**(4):901–20.
38. Lambert PJ, Aronson JR. Inequality decomposition analysis and the Gini coefficient revisited. *The Economic Journal* September 1993; **103**(420):1221–27.
39. Rosenberg LH. Applying and interpreting object oriented metrics. Software Technology Conference, Utah, April 1998.
40. Stapleton J. *DSDM Dynamic Systems Development Method : The Method in Practice*. Pearson Education Limited: Harlow, Essex, UK, 1997.
41. Mordal-Manet K, Balmas F, Denier S, Ducasse S, Wertz H, Laval J, Bellingard F, Vaillergues P. The Squale model—a practice-based industrial quality model. In *ICSM '09*. IEEE Computer Society: Los Alamitos, California, USA, 2009; 94–103.
42. Balmas F, Bellingard F, Denier S, Ducasse S, Franchet B, Laval J, Mordal-Manet K, Vaillergues P. The Squale quality model. modèle enrichi d'agrégation des pratiques pour java et c++ (Squale deliverable 1.3). *Technical Report*, INRIA, 2010.
43. Lanza M, Marinescu R. *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer-Verlag: Berlin Heidelberg, 2006.
44. Zeileis A. Package 'ineq' for R. *Technical Report*, CRAN, 2009.
45. Boehm B, Brown W. Value-based software metrics. *IEE Seminar Digests* 2004; **2004**(909):4–6.
46. Bucci G, Fioravanti F, Nesi P, Perlini S. Metrics and tool for system assessment. In *Proceedings of IEEE Conference on Complex Computer Systems. USA: IEEE Publ*, 1998; 36–46.
47. Kitchenham BA. What's up with software metrics? - a preliminary mapping study. *Journal of Systems and Software* 2010; **83**(1):37–51.
48. Mancoridis S, Mitchell BS, Chen Y, Gansner ER. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of the IEEE International Conference on Software Maintenance*. ICSM'99, IEEE Computer Society: Washington, DC, USA, 1999; 50–62.
49. Heitlager I, Kuipers T, Visser J. A practical model for measuring maintainability. In *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*. IEEE Computer Society: Washington, DC, USA, 2007; 30–39.
50. Subramanyam R, Krishnan MS. Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. *Software Engineering, IEEE Transactions on* april 2003; **29**(4):297–310.
51. Yu P, Systa T, Muller H. Predicting fault-proneness using oo metrics. an industrial case study. In *Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on*. IEEE Computer Society: Los Alamitos, California, USA, 2002; 99–107.
52. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on* jun 1994; **20**(6):476–493.
53. Tamai T, Nakatani T. Statistical modelling of software evolution processes. In *Software Evolution and Feedback. Theory and Practice*, Madhavji NH, Fernández-Ramil JC, Perry DE (eds.). John Wiley & Sons Ltd: Chichester, England, 2006; 143–160.
54. ISO. ISO 31–11 mathematical signs and symbols for use in physical sciences and technology. ISO: Geneva, Switzerland, 1992.
55. Marinescu R. Detection strategies: Metrics-based rules for detecting design flaws. In *ICSM*. IEEE Computer Society: Los Alamitos, California, USA, 2004; 350–359.

DOI: 10.1002/smr