Quantifying the Analyzability of Software Architectures

Eric Bouwers*‡, José Pedro Correia*, Arie van Deursen‡ and Joost Visser*

* Software Improvement Group, Amsterdam, The Netherlands
E-mail {e.bouwers,j.p.correia,j.visser}@sig.eu
‡ Delft University of Technology, Delft, The Netherlands
E-mail {Arie.vanDeursen,E.M.Bouwers}@tudelft.nl

***Abstract*—The decomposition of a software system into components is a major decision in any software architecture, having a strong influence on many of its quality aspects. A system's analyzability, in particular, is influenced by its decomposition into components. But into how many components should a system be decomposed to achieve optimal analyzability? And how should the elements of the system be distributed over those components?**

In this paper, we set out to find answers to these questions with the support of a large repository of industrial and open source software systems. Based on our findings, we designed a metric which we call *Component Balance*. In a case study we show that the metric provides pertinent results in various evaluation scenarios. In addition, we report on an empirical study that demonstrates that the metric is strongly correlated with ratings for analyzability as given by experts.

I. Introduction

Software architecture is loosely defined as the organizational structure of a software system including components, connections, constraints, and rationale [18]. Choosing the right architecture for a system is important, since *"Architectures allow or preclude nearly all of the system's quality attributes"* [9]. Fortunately, there is a wide range of software architecture evaluation methods available to assist in designing an initial architecture (for overviews see [1], [11]).

After the initial design, it is important to regularly evaluate whether the architecture of the software system is still in line with the requirements of the stakeholders [24]. However, a complete re-evaluation of a software architecture involves the interaction of various stakeholders and experts, which makes this a time-consuming and expensive process. An alternative is to use more lean methodologies that support a high-frequency evaluation, such as those proposed in [6], [7], [13].

Any evaluation process, the high-frequency ones in particular, greatly benefits from the use of software metrics to support it. Advantages include reducing the effort and time needed to perform the evaluation, as well as making the evaluation more objective and repeatable. Furthermore, metrics can enable continuous monitoring and thus early detection of deviations in quality.

The work on metrics for software architectures has traditionally been focussed on the way components depend on eachother and how components are internally structured (coupling and cohesion [23], [26]). Two related aspects of a software architecture have, however, received relatively little attention when it comes to metrics: the decomposition of the system in terms of the *number of components* and their *relative sizes*.

Both of these aspects have a strong influence on how easy it is to locate the parts of the system that need to be changed, i.e., the system's *analyzability*. Having only one component (or one large component combined with several small ones) does not offer much discriminative power to locate specific functionality. In contrast, having a large number of (equally sized) components can overwhelm a software engineer with too many choices.

In earlier work, efforts have been made to quantify the relative sizes of components [21] and there have been references to an "ideal" number of components for a system [4]. However, there has been no effort to quantify these concerns and capture them in a single metric, such that they can be evaluated in a condensed manner.

In this paper, we present a metric called *Component Balance* which takes into account both the number of components and their relative sizes. In order to define this metric, we determined what a "reasonable" number of components can be by studying the decomposition of over 80 systems.

To investigate whether the proposed metric accurately reflects the analyzability of a system, we performed a quantitative experiment in which we test the correlation of the values of the metric with the judgement of experts. In addition, we performed a qualitative case study on an open-source project to show that the metric is usable in an evaluation setting.

In short, this paper makes the following contributions:

- We describe an empirical exploration of how systems are decomposed into top-level components;
- We define a metric to measure the balance of components which is usable across all life-cycle phases of a project;
- We show how the metric is correlated with the opinion of experts about the analyzability of a software system.

II. Problem statement

We are looking for a metric which characterizes a system's analyzability by evaluating the decomposition of a system into components. In order to define more clearly the problem at hand, we need to define our notion of *component*, as well as its connection to *analyzability*.

A. Definition of component

We define component by adopting the definition of software modules by Clements et al., i.e., a component is considered to be *an implementation unit of software that provides a coherent unit of functionality* [8]. Within a system, such coherent units of functionality exist on multiple levels (e.g., class, package). To ensure a consistent use of the term, we define a component to be a module at *the first level of decomposition in a system*.

For example, the components could be the top-level packages in a Java system, or the collection of files which together form a working project in the IDE of a developer. A component can further be divided into modules (e.g., classes in Java or files in a working project), which are in turn decomposed into units (e.g., methods in Java or functions in C). Note that, in this definition, there is no assumption of the type of functionality which is implemented in a component. A decomposition can be based on a technical point of view (e.g., having components for file-access, network connections and the GUI) or a business point of view (e.g., containing components for savings, accounts and stocks).

B. Analyzability and component decomposition

The ISO/9126 [17] standard for software quality defines analyzability, a sub-characteristic of maintainability, as: *"the capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified"*[17]. A common strategy to find those parts that need to be modified is by following the control-flow of a program. However, before this strategy can be applied, a software engineer needs to identify where to start following the control-flow for a specific feature. From a cognitive perspective, this first step is influenced by how easy it is for a software engineer to split up the overall software system into meaningful chunks of functionality [5], without being overwhelmed by too many choices.

To illustrate this problem, consider Figure 1 which shows several examples of how a system can be decomposed. Figure 1(a) shows the simplest case in which a system is "decomposed" into a single component. Such a decomposition hinders analyzability, as the structure of the code does not provide any hints as to where functionality is implemented. On the other hand, a division as shown in Figure 1(b), in which the system is decomposed into many small components, does not provide a software engineer with sufficient clues as to which component should be chosen to inspect.

However, inspecting only the number of components does not suffice to conclude about the analyzability of a system. As illustrated in Figure 1(c), there can still be a situation in which a system has a reasonable number of components, but where one component contains almost all the code of the system. Similar to having only a single component, this decomposition provides only limited clues as to where which functionality is implemented. To provide maximal discriminative power to a software engineer, a system should be decomposed into a limited number of components of roughly the same size, as illustrated in Figure 1(d).

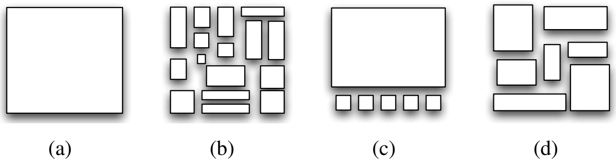


Fig. 1. Decomposition of three systems which are hard to analyze (a), (b), (c) and one which is considered to be good (d)

These observations lead us to conclude that a metric which measures the analyzability of a software system must quantify whether a software system is decomposed into a *reasonable number* of components with *a low variation in size*.

III. Requirements

To guide our search and evaluation of a metric that fits our problem, let us establish some basic requirements. The first follows naturally from the discussion in the previous section:

R1: The metric should provide an indication of the analyzability of a software system in terms of its structural decomposition.

The fulfillment of this requirement ensures that the metric is usable during the evaluation of a software system in a single moment of time.

Apart from such a one-off assessment it is desirable to use the metric to track the evolution of the analyzability over time. To be able to compare the results of the metric over a longer time-period, we should ensure that the values of the metric are equally meaningful in every stage of the development of a software system, which leads to the second requirement:

R2: The metric should provide relevant results during all stages in the life-cycle of a software system.

Lastly, in order to ensure that the metric can be used in a wide range of systems we require that the metric is not restricted to a specific programming language or programming paradigm. This leads us to the last requirement:

R3: The metric should be technology-independent.

With these requirements in mind, let us explore existing metrics in the literature.

IV. Related Work

One of the seven design principles of Sarkar et al. [21] is the uniformity of component size, i.e., component should be roughly equal in size, an attribute also mentioned by several other researchers [14], [16], [5]. Apart from mentioning the design principle, Sarkar et al. define a metric to measure the uniformity of the component size called the Module Size Uniformity Index (MSUI) [21]. The MSUI is defined as the division of the average component size of a system by the sum of this average component size and the standard deviation. In a later paper, Sarkar et al. specialized this metric towards object-oriented systems [20].

Unfortunately, the proposed metrics do not deal with all situations outlined in Figure 1. In particular, the situation in Figure 1(a) would receive the highest possible score of 1, while even the original authors of this metric state that having only a single component in a system is considered to be a bad decomposition [21].

In addition, both the original MSUI and its object-oriented variant are considered to be supporting metrics and are only briefly mentioned in their evaluations. In the first paper [21] the authors explain that the change in values is related to the way in which their clustering algorithm works, while in their second paper [20] the authors explain that the evaluation is not geared towards these metrics.

A second metric which can be used to measure the uniformity of the sizes of components is the Gini coefficient [12]. This metric was proposed by the statistician Corrado Gini in 1921 to measure the inequality of the distribution of income of a given population. This measurement has recently been applied in the field of software metrics (e.g., [25]) with interesting results. Unfortunately, this metric has the same problem as MSUI when it comes to dealing with a single component, and there has been no validation of this metrics with respect to measuring the size distribution of components in software systems.

Unfortunately, papers proposing metrics to address the number of components of a system are scarce. There is evidence in the literature that a single component is considered to be a bad decomposition [21], and that breaking up a system into many small pieces may harm reliability [14]. In addition, Blundell et al. conclude that there is an optimal number of components for a given software system [4]. However, we are unaware of papers which define metrics to quantify the number of components of a system with respect to this optimum or either of the extremes.

The only quantification which comes close to specifying an optimal number of components can be found in the work of cognitive psychologist Miller [19]. This work shows that humans can cope with around 7±2 pieces of information at the same time through their short-term memory. Given this theory, one could suggest that around 7 would be the ideal number of components for a software system. However, we are not aware of any existing work which validates this hypothesis in the context of software components.

V. Counting components

Not having found in the literature a suitable metric for our purposes, we set out to investigate what a "reasonable" number of components might be. For that, we took an empirical approach and created a repository of different software systems. This allowed us to investigate how systems are typically decomposed into components, in particular into how many.

This section describes the general criteria for creating such a repository, the composition of our particular instance and discusses some of our observations.

A. Repository composition criteria

Establishing a repository of systems amounts to sampling individuals from a population. To ensure generalizability and reliability of the results, general considerations for sample taking should be taken to heart, such as maximizing size and representativeness, consistent data collection, and outlier inspection and removal.

Consistent data collection requires that a clear definition of *component* exists and is applied consistently across systems. Furthermore, the measurement of the volume of the various components should be done according to common guidelines or with a single tool.

The level of quality of the architecture of systems should not play a role in their selection. Otherwise a bias will be introduced in the sample which could compromise the generalizability of the results. On the other hand, the degree of stability of the architecture of candidate systems should be taken into account. Systems that are in the initial stages of development or in a phase of rapid architectural churn are best excluded from the sample, since the architecture at the time of measurement could be not representative of the architecture of the system during a substantial phase of its life.

B. Repository instantiation

Because we are interested in today's state-of-the-art, the population of software systems from which we wish to take a sample are modern, object-oriented systems that support corporations or large user communities. This excludes, for instance, old legacy systems or research prototypes. Within this group, we want systems to be represented of different sizes and development contexts (industrial and open-source).

We created a repository by gathering software systems previously analyzed by the Software Improvement Group (SIG), an independent advisory firm that employs a standardized process for evaluating software systems of their clients [2]. These industry systems were supplemented by open source systems previously analyzed by SIG's research department. Since, in the experience of SIG, the overwhelming majority of modern industrial systems are developed in C-like programming languages, we restricted our selection to Java, .NET (C#, VB.NET), and C/C++ systems.

The following table characterizes the repository in terms of number of systems per technology and development context[1]:

	Java	.NET	C/C++	Total
Industry	35	19	5	59
Open source	17	4	6	27
Total	52	23	11	86

Thus, the repository contains a total of 86 systems. Almost 70% were developed in an industrial context. About 60% were developed on the Java platform, 27% on .NET and the remaining 13% are C/C++ systems. The selected systems offer functionality in a broad spectrum of domains (e.g. public administration, finance, developer tools, system control) with a size ranging between 1 thousand and 3 million Lines of Code.

C. Component breakdown

For the industrial systems, the component breakdown was determined by the technical analysts of SIG. They

[1] An online appendix with detailed experiment data is available at www.sig.eu/en/component-balance

work according to standard guidelines that start with elicitation of component information from the system's development/maintenance team and ends by validating the defined breakdown with that team.

For the open-source systems, the breakdown was determined also by a technical analyst and/or researcher of SIG, based on available documentation. In cases where the documentation was insufficient, the directory structure of the source-code was used to guide the component breakdown. In line with our definition of component in Section II, we used only the *first major decomposition*, leaving any deeper hierarchical decomposition out of consideration. We took as leading the breakdown for the main programming language in each system, also when a deviating breakdown was present for an auxiliary language.

D. Observations

Figure 2 shows the distribution of the number of components per system based on the created repository. As it can be observed, the number of components metric is distributed in a non-symmetric way. In fact, a Shapiro-Wilk test [22] yielded a p-value of 0.0014, thus allowing us to reject the hypothesis that the data is normally distributed.

A second observation is that a large portion (57%) of the systems in our repository tends to have between 5 and 10 components, which is roughly in-line with the 7 ± 2 rule of Miller [19]. However, if we inspect the central tendency of the repository, using the *median* in order to be robust against the asymmetry, we observe that this is valued at 8. Thus apparently, the most common number of components for a system is close to this number.

A question which this repository could answer is whether a large system more often consists of a high number of components, simply because there is more functionality to be implemented. If this is the case, a high correlation between the size of the system and the number of components in the system should be observed. When measuring the size of the systems in the repository by their Lines of Code, we observed no strong positive or negative correlation between the sizes and the number of components of a system (Spearman rank correlation shows 0.27 with a significant p-value).

We believe the reason for this is that whenever a system grows in terms of the number of components, there comes a certain point at which the current components of the system are grouped together within a new level of abstraction. In other words, when the system grows, new levels of abstraction are added to ensure that the first level of decomposition in the system remains manageable.

VI. Metric definition

With more empirical data on the number of components available, we can use this knowledge to define a general metric called *Component Balance (CB)*. We define this metric as a combination of two other metrics *System Breakdown (SB)*, which is designed to measure whether a system is decomposed into a reasonable number of components and *Component Size Uniformity (CSU)*, which aims to capture whether the components are all reasonably sized. We define the metrics in general, non software specific terms first, after which they are instantiated for the domain of components in a software system in Section VI-F.

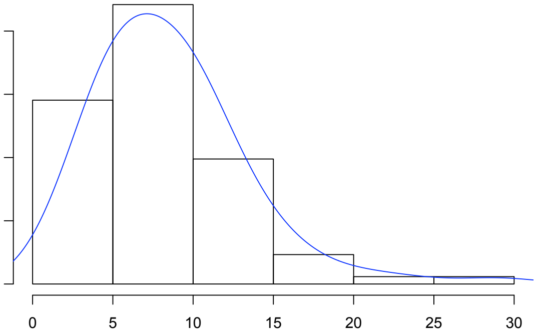


Fig. 2. Distribution of number of components per system (histogram and estimated density function).

A. Terminology

Let $S = \langle M, C \rangle$ be a system, consisting of a set of modules M and a set of components C. Each module is assigned to a component and none of the components overlap. More formally, the set $C \subseteq \mathcal{P}(M)$ is a partition of M, i.e.,

- $\forall c_1, c_2 \in C : c_1 \neq c_2 \Rightarrow c_1 \cap c_2 = \emptyset$.
- $\bigcup_{c \in C} = M$

Furthermore, each module has a given size (measured by, for example, the Lines of Code or Function Points), which is captured by a function $size : M \rightarrow \mathbb{N}$. The volume of a component $c \in C$ is defined simply as the sum of the size of its modules, thus:

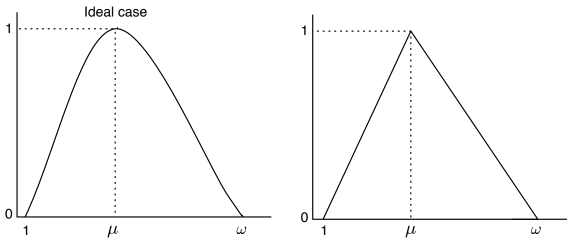
$$volume(c) = \sum_{m \in c} size(m)$$

B. System Breakdown

The basis for measuring the System Breakdown (*SB*) metric is the number of components $|C|$, which is an unbounded positive number. However, a higher number of components does not imply better analyzability. As discussed in Section II, both a high number of components and a low number of components hinders analyzability. To capture this in a metric, we want to map the number of components to a fixed range of numbers in which the highest number denotes a better analyzability, thus $\mathit{SB} : \mathbb{N}^+ \rightarrow [0, 1]$.

Since the number of components has a lower bound of 1, we define $\mathit{SB}(1) = 0$, thereby assigning the lowest value of the metric to the minimum number of components. On the other end, there is no theoretical upper bound for the number of components so we define an artificial upper limit $\omega > 1$ for which $\mathit{SB}(n) = 0$ when $n \geq \omega$. Between 1 and ω there is a number of components which depicts the best analyzability, let us consider $\mu < \omega$ to represent this number and define $\mathit{SB}(\mu) = 1$, thus assigning the highest value of the metric to the optimal number of components.

The values of the function between 1 and μ and between μ and ω are still undefined. Ideally, one would like the function to behave as represented in Figure 3(a) (closely resembling



(a) (b)

Fig. 3. An ideal component deviation function (a) and an example plot of the *SB* function (b).

the shape of the distribution of our repository), where being slightly off the optimal case still warrants a high score, but being farther away would warrant progressively lower scores. However, to keep the metric definition as simple as possible, we opted to define the intermediate values by linear interpolation between the aforementioned points, thereby obtaining the following metric definition:

$$\mathit{SB}(n) = \begin{cases} \frac{n-1}{\mu-1} & \text{if } n \leq \mu \\ 1 - \frac{n-\mu}{\omega-\mu} & \text{if } \mu < n < \omega \\ 0 & \text{if } n \geq \omega \end{cases}$$

which behaves as shown in Figure 3(b).

The result of this function is thus a number in the range $[0, 1]$, where a higher value denotes less deviation from the "optimal number of components", and thus a better decomposition of the system.

C. Component Size Uniformity

The goal of the Component Size Uniformity (*CSU*) metric is to measure how uniformly the volume of the system is distributed over its components. A way to measure this is by using the Gini coefficient [12], [25]. The value of the coefficient is a number in the range $[0, 1]$, where a low value denotes a more balanced distribution in the population, and a higher value means more inequality, i.e., a small part of the population has a significantly higher value than the rest.

The coefficient is directly applicable to the problem at hand, except that we would like a lower value to represent a less well-distributed decomposition, thus we define *CSU* as:

$$\mathit{CSU}(C) = 1 - \mathit{Gini}(\{\mathit{volume}(c) : c \in C\})$$

The result of this function is a number in the range $[0, 1]$, where a higher value denotes a more balanced distribution of the volume of the system into its components.

As discussed in Section IV, another option would have been to use the MSUI metric of Sarkar et al. [21]. We evaluate this option and compare it to the *CSU* in Section VIII.

D. Component Balance

The values of *SB* and *CSU* need to be combined to give intuitive quantifications to the scenarios listed in Section II. Since there are various ways to combine the two metrics (e.g., minimum, maximum, sum, product, average), we need to choose an aggregation function based on desired properties. The main property we require is that the function is *conjunctive*, i.e., that it does not allow for compensation [3].

To illustrate, consider the extreme case where a system is decomposed into a single component. This results in a low score on *SB* (0), but a perfect score of 1 on *CSU*. This last score is due to the fact that when a single component holds the complete size of the system, the size is trivially evenly distributed. When using, for example, the average, which is a disjunctive function, the resulting score would be $\frac{1+0}{2} = 0.5$, thus assigning a value in the middle of the range for what is considered a very bad component balance.

The simplest examples of conjunctive aggregation functions are the minimum and the product. The minimum, however, reduces the discriminative power of the metric. For example, a system for which $\mathit{SB} = 0.1$ (few components) and $\mathit{CSU} = 0.1$ (badly balanced) would not be distinguishable from another system with the same number of components but for which, for example, $\mathit{CSU} = 1$ (perfectly balanced). For that reason we chose the product as the aggregation function, thus:

$$\mathit{CB}(S) = \mathit{SB}(|C|) \times \mathit{CSU}(C)$$

The result of this function is a number in the range [0,1]. Higher values represent better component decomposition.

E. Properties

The definition of *CB*, as outlined above, exhibits several desirable properties. First of all, the definitions of the metrics are not tailored towards any programming language or methodology, which ensures that requirement **R3** is satisfied.

Secondly, the metrics are not influenced by the volume of the system. As shown before, the number of components is not correlated with the size of the system. Additionally, the Gini-coefficient has specifically been designed to be agnostic to population size. This enables the comparison of the values for a system over time, even when the system grows.

More generally, even though the metric is explained in terms of the components of the software system, it is not necessarily limited to this particular situation. In fact, the definition is generic enough to apply to any situation in which an entity is decomposed into distinct parts with a given size.

Regarding limitations of the metric, we emphasize that it can currently only be applied on *a single level of decomposition*. Therefore the metric cannot be directly used to quantify the analyzability of a multi-layered architecture. However, the metric can be applied to each layer of such a decomposition in isolation, which reduces the impact of this restriction.

F. Metric instantiation

To instantiate the metric for a specific domain two actions need to be taken. First, to be able to calculate the metric, the free variables of *SB* must be instantiated. In our current situation, we should instantiate μ with the "optimal" number of components for a system, which is impossible to determine. For practical reasons, we will use "the most common number of components" as an approximation of the ideal number of components. By using the central tendency of the repository described in Section V, we can define $\mu = 8$. The value of ω should be one of the higher values of the metric observed in the repository. However, to not be overly sensitive to extreme values we decided to take the 95th percentile of the repository which is valued at 16. We thus have $\omega = 16$.

Secondly, in most domains a small variation in the size of components is allowed and expected, which means that a value of 1 for *CB* is only achieved under artificial circumstances. Therefore, the distribution of the values of *CB* for real-world cases should be analyzed to determine which values can be considered indicative of a certain level of analyzability, instead of using the theoretical range of the metric.

VII. Evaluation Design

While the definition of the metric inherently satisfies requirement **R3**, both requirement **R1** and **R2** call for a more extensive evaluation. First of all, we want to evaluate whether *CB* satisfies requirement **R1**, thus we have the following goal:

G1: Evaluate if *CB* can be used as an indicator for the analyzability of a software system.

Whether or not this goal is achieved, it could be the case that taking an alternative choice in the design of the metric would lead to better results, thus it would be interesting to evaluate the alternatives:

G2: Evaluate if the choices made in the design of the metric are appropriate.

In order to achieve these two goals, we designed a quantitative experiment where the metric and some alternatives were compared to a "Gold standard", which consisted of ratings provided by experts. This is presented in Section VIII.

To satisfy requirement **R2** we need to determine whether *CB* provides relevant results during all stages of the life-cycle of a software system. Thus, we would like to address two goals, namely:

G3: Understand how the value of *CB* can help during the assessment of a software architecture.

G4: Understand how the value of *CB* evolves over time.

To accomplish these goals, we performed a case study where we assess a software system in terms of its structural decomposition. We used the metric as a basis for discussion of the current decomposition, as well as to investigate its evolution through time. This is presented in Section IX.

The integrated evaluation findings are presented in Section X, along with an assessment of the threats to validity covered in Section XI.

VIII. Quantitative evaluation of metric performance

A. Experiment design and execution

The first step is to create the "Gold standard" to compare the metric to. To do this, we conducted interviews with eight experts working at SIG (see Section V-B) in the field of software quality assessment, who were asked to rate the analyzability of a given software system in terms of its structural decomposition into components. They were requested to use a 5-point Likert scale, but in certain cases they did not find the scale detailed enough and chose to award half-point ratings.

System	Language	$\|C\|$	KLOC	Rating
A	Java	8	53	2
B	Java	10	153	3
C	VB.NET	2	87	2.25
D	C#	11	22	2
E	C#	9	82	2
F	Java	5	273	3
G	Java	5	64	2.5
H	Java	51	333	1
I	Java	5	35	3.5
J	Java	5	25	3
K	Java	11	145	2
L	Java	14	512	2
M	C#	16	125	2
N	Java	9	197	5

TABLE I
Systems used for the evaluation.

All experts are experienced in evaluating the technical quality (focussed on maintainability) of industrial systems. For each expert, we selected 1–3 systems for which they had conducted regular, monthly assessments, during at least three months. This time period was chosen to ensure that the expert was familiar with the systems under evaluation. This resulted in 15 different systems of which 10 are implemented in Java, 4 in C# and 1 in VB.NET.

For 6 out of the 15 systems we asked two experts to provide us with a rating, for the other 9 we could only interview a single expert due to resource constraints. In 4 out of the 6 double ratings the experts agreed with each other by giving the same rating. In one case they disagreed only by half a point, so we decided to include the data point using the average of their ratings (2.25). In another case, one of the experts gave a rating of 1 while the other expert gave a rating of 3. Because of this disagreement we excluded those ratings from the results, resulting in 14 data-points.

The details of the systems used, as well as the ratings assigned by the experts can be found in Table I. As can be seen, the systems range over different sizes and numbers of components. Furthermore, the experts tend to provide a rating below the average rating of three, while distributing the ratings over the full possible range.

B. Experiment details

In order to ensure that the experts all had the same understanding of the terms used they received an explanation of the overall goal of the Component Balance metric at the start of the interview. It was explained that when the code is evenly distributed over the components on a similar level of abstraction, it is easier to find out where changes in a system need to be made and that, therefore, this metric is related to the analyzability of the system.

To lower the risk that the experts use pre-existing knowledge about metrics related to Component Balance to guess the desired outcome, the experts were made aware of the fact that using this type of knowledge would mean that we would measure their ability to guess a metric. It was stressed that we are only interested in their expert opinion. To emphasize this point and to make the evaluation as realistic as possible, the experts were asked to imagine that the customer behind the system is asking for such a rating.

To evaluate the technical quality of systems, all experts are experienced users of the SIG Quality Model [15]. This model provides a rating for the maintainability of a software system on a scale of 1 to 5. The scale deviates in the extremes, splitting a set of systems into 5/30/30/30/5-percent of the systems. This means that assigning a rating of 1 to a system puts this system within the worst 5 percent of systems, and assigning a score of 5 places a system amongst the top 5 percent of systems [2], [10]. Because the experts are familiar with the usage of such a scale, we have chosen to adopt the same scale for the interviews.

During the rating-phase, the experts had access to all the data that they usually have while evaluating these systems. This data includes (among other things) size, coupling, complexity and duplication metrics. In addition, dependency graphs between components and the evaluation of these dependencies over time were available. In connection to the metric, the experts do have access to the number of components and the different sizes of the components, but they do not have access to the Gini-values for component sizes, nor do they have access to any contextual information provided by a repository such as defined in Section V.

Apart from the actual rating, the experts were also asked to provide a motivation for it, which was used to a) cross-check ratings between experts which rated the same systems, b) see whether our initial intuition is shared by the experts, and c) determine which metrics the experts used in their evaluation.

C. Results

In order to evaluate whether the values of *CB* can serve as an indicator for the opinion of the experts, we calculated a Spearman rank correlation coefficient (ρ) between the ratings given by them and the Component Balance metric values. This non-parametric rank correlation test was chosen because no assumptions can be made as to how the values extracted from either the experts or the value of *CB* are distributed.

Furthermore, we use the ranking to evaluate some of the choices made in the design of the metric. For example, we combine *SB* and *CSU* by multiplication rather than a different aggregation function. In addition, we have chosen to use the Gini coefficient as a means to calculate *CSU* instead of the *MSUI* metric proposed by Sarkar et al. [21]. To validate whether these decisions are justified, we calculate the Spearman correlation scores for two alternative aggregation functions as well as the replacement of *CSU* with *MSUI*. Lastly, to ensure that combining the two parts of the metric is actually needed we also calculate the correlation scores for *CSU*, *SB* and *MSUI* in isolation. The results are summarized in Table II. The Spearman rank correlation between the opinion of the experts and the values obtained from *CB* is 0.80 (with a p-value of 6.4×10^{-4}), which indicates a strong, significant correlation between both rankings.

The table shows that almost all of the alternative metrics can be used as an indicator for the opinion of experts, but that the definition for *CB* as given in Section VI yields a significantly higher correlation score than most of the alternative metrics. Only using the minimum as an aggregation function has a similar performance, but as explained in Section VI-D, using multiplication is preferable to taking the minimum, because of the added discriminative power.

Metric	ρ	p-value
CB ($SB \times CSU$)	0.80	0.00064
CB ($min(SB, CSU)$)	0.79	0.00084
CSU	0.73	0.0031
CB ($\frac{SB+CSU}{2}$)	0.70	0.0057
$MSUI$	0.68	0.0071
CB ($SB \times MSUI$)	0.62	0.019
SB	N/S	0.43

TABLE II
CORRELATIONS SCORES BETWEEN EXPERTS RANKING AND DIFFERENT DEFINITIONS OF *CB*

The only alternative metric for which there is no significant correlation score is using *SB* in isolation. We believe that using only this metric to assess the quality of a component decomposition is not sufficient, but the lack of statistically significant evidence leaves this question open. An experiment involving more subjects should be conducted to investigate this hypothesis more thoroughly.

IX. CASE STUDY

A. Subject system

The subject of the case study is Checkstyle[2], an open-source Java-library that checks for coding violations. This project has been chosen because it is mature (having a history of over 10 years), widely used in both industry and open-source (and therefore well-known), yet small enough to be evaluated and understood in a reasonable amount of time. In addition, the open-source nature of the project allows for easy replication.

In this evaluation, we considered the major releases from 1.0 (January 2001) until 5.1 (February 2010). We only considered the Java-code in the main `src` directory of the Checkstyle project, excluding the separate source-tree under `contrib`.

B. Architecture assessment

To understand how the value of *CB* can be used in a discussion about the top-level decomposition of a software system, we perform an in-depth evaluation for release 5.1 of Checkstyle, which has a *CB* value of 0.29. By evaluating this value in the context of Checkstyle, we hope to determine actions to be taken to increase the analyzability of the project.

To put the value of 0.29 in perspective, we compare it to the *CB*-values of other systems. In particular, we can compare it to the *CB*-values of the repository established in Section V. It turns out most systems (90%) have a *CB*-value between 0 and 0.53. Checkstyle scores better than about 55% of the systems in the repository, meaning that the quality of its decomposition is slightly above average.

If we breakdown the metric into its two parts, we have $SB = 0.86$ and $CSU = 0.34$. In the context of the repository, the value of *SB* is higher than 84% of the systems, but *CSU* scores in the bottom 22%. This indicates that the system is

[2] http://checkstyle.sourceforge.net/

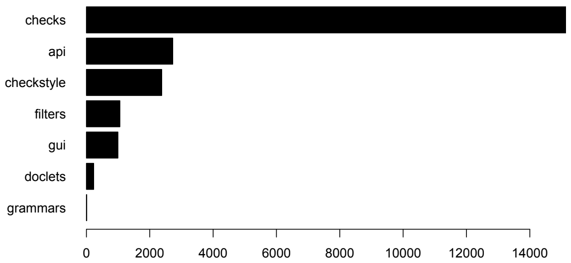


Fig. 4. Sizes of the top-level packages of Checkstyle 5.1 measured in Lines of Code. $SB = 0.86$, $CSU = 0.34$ and $CB = 0.29$.

decomposed into a reasonable number of components, but that there is a large variation in the sizes of those components. The plot of the relative sizes of the components of Checkstyle shown in Figure 4 confirms this.

The biggest component is `checks`, containing almost 70% of the code of the system. Given that the core-functionality of the library is to check for coding-standard violations, it is not surprising to notice that most of the code of the project is indeed dedicated to the implementation of the various checks. Given this distribution of size, our intuition is that most of the changes take place within the `checks` component. A manual inspection of the release notes of the project reveals that in the last three releases, most of the features and bug-fixes have indeed been related to different checks.

Based on the fact that checkstyle is a plug-in based architecture, a recommendation could be to split up the project into a "framework"-project and a "plugins"-projects. Benefits of this approach are, amongst others, a more strict separation of concerns on the architecture level and less code to analyze for developers working on checks. In addition, such a break-up would have benefits on the project-management level, possibly resulting in more focussed project-teams and separate release cycles. In an industry setting, this would also lead to a new allocation of budget and other resources.

To evaluate the result of such a break-up we calculated the values for *CB* for both the hypothetical "framework" project (all top-level packages minus the `checks` packages) and the hypothetical "plugins" project (the top-level `checks` package which is decomposed into sub-packages). For the framework project, the value of *CB* would rise to 0.39, placing it in the higher regions of our repository. In the other hand, the score for the plugins project would be valued at $CB = 0$. This low score is due to the fact that $SB = 0$, which in turn is caused by the fact that the `checks`-package is subdivided into 16 different sub-packages.

A closer inspection of the naming of these sub-packages reveals that not all of them are on the same level of abstraction. There are both sub-packages with generic names such as `design`, `coding` and `metrics`, as well as sub-packages with a more specific purpose such as `modifier`, `header` and `annotation`. This last sub-package, together with the sub-package `blocks`, could probably be merged into the more generic sub-package `code`. In addition, the grouping of the checks in the different sub-packages is not consistent.

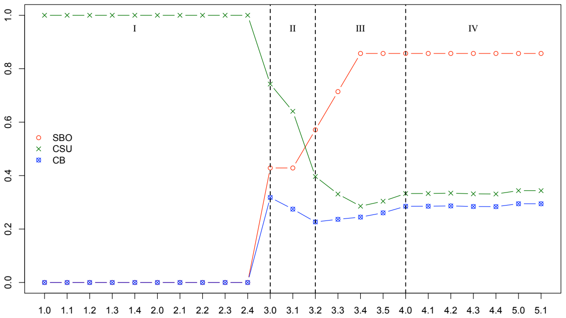


Fig. 5. Metric Trends for Checkstyle

For example, there is a sub-package called `whitespace`, but the `NewlineAtEndOfFileCheck` is placed in the root of the `checks`-package, right along abstract types for checking formatting and options. Again an example of placing concepts from different levels of abstraction side-by-side.

These observations strengthen the evaluation by *CB* that within this part of checkstyle it is indeed hard to determine where certain functionality is located. A reorganization of the checks into clearly defined concepts on a similar level of abstraction would help new developers to understand more easily where to look for specific checks.

C. System evolution

To understand how the value of *CB* evolves over time we plotted the values of *CB*, *SB* and *CSU* for 23 releases of Checkstyle in Figure 5. In this plot, four distinct time periods (marked as *I*, *II*, *III* and *IV*) can be distinguished. For each of these periods, we relate the changes in the values of the metrics to specific events in the development of the project and their impact on the analyzability.

In the first time period *I*, corresponding to the releases 1.0 to 3.0, one can observe that the metric is valued at 0. In fact, initially Checkstyle had no major component decomposition. At the end of the period, we see a large positive jump. The explanation for this jump can be found in the release notes of release 3.0, which state: *"Completely new architecture based around pluggable modules."* This release decomposed the system into three distinct components; the core, the API and the checks, thus making it easier to distinguish between these three types of functionality.

The second time period *II* shows a sharp decrease in *CB*-value between releases 3.0 and 3.2. This decrease is mostly due to the sharp decline in the values of *CSU* which suggest that a large amount of code was added to a single component. An inspection shows that this is indeed the case, since the largest component `checks` almost doubled in size between both releases 3.0 and 3.1 and between release 3.1 and 3.2, while the size of the other components changed only a little.

Within the third time period *III*, spanning releases 3.2 until 4.0, the value of *CB* slowly increases. In the first two releases of this period, this is due to the increasing value of *SB*, which is explained by the addition of the components `grammar` and `doclets`. In the later two releases, the number of components is stable, thus the increase in value can only be explained by an increase in *CSU*. Inspection of the code shows that between these releases several checks have been retired to a separate source tree which results in almost a 20 percent decrease of code in the `checks` component. Although removing this code requires a software engineer to examine less code in the largest component, the overall analyzability only increases a little.

Since release 4.0, the value of *CB* has not been changing significantly, indicating the architecture of the project has stabilized in time period *IV*. Inspecting the release notes confirms this, since after release 4.0 no mention to architectural changes on the level of components can be found.

X. Discussion

This section discusses the results of the two phases of the evaluation with respect to the goals as outlines in Section VII.

In Section VIII-C we showed that the value of *CB* strongly correlates with a ranking of the analyzability of a software system as given by experts. Because of this, we conclude that *CB* can be used as a proxy for the analyzability of a system and thus **G1** is satisfied. In addition, the results also show that alternative implementations of the metric do not outperform the definition as given in Section VI, thus satisfying **G2**.

In order to evaluate *why CB* shows this strong correlation, we took a closer look at the systems which where assigned similar rankings by both the experts and *CB*. The system which is ranked lowest by both the *CB*-metric and the experts consists of over 50 components which is the result of combining both a technical and functional decomposition on a single level. This results in components which are either dedicated to the implementation of a single business functionality, or contain the complete GUI of the system. On the high end of the scale there is the system which is ranked the highest by both the experts, as well as the metric. According to the expert who evaluated this system, the naming of the components suggest a similar level of abstraction.

There are two cases in which *CB* gave a significantly higher ranking than the experts. One of these systems was decomposed into eight components, of which two contained almost all of the code. According to the experts, the architecture aims to be a Model-View-Controller architecture, in which the representation of the model of the application is separated from the implementation of the manipulation and display functionalities. However, in the implementation of this system, both the model and the functionality to manipulate this model are placed in a single component, which leads to a highly skewed implementation. The situation was similar for the other system which has nine components, two of them containing too much functionality. In both cases, the score for *CSU* is low, but the high score for *SB* places the systems on a higher position than the expert(s).

For none of the systems the ranking of *CB* was significantly higher than the ranking given by the experts. Given the previous example, it could be the case that a high score for *SB* has a too strong influence on the value of *CB*. This might be because reaching an optimal value for *SB* is relatively easy, while reaching an optimal value for *CSU* is very difficult under realistic scenarios. To compensate for this, we could make the optimal value for *SB* harder to reach or choose a more complex aggregation function. Both solutions might lead to better results at the cost of a more complicated definition of the metric. Evaluation of this trade-off is future work.

With respect to both **G3** and **G4** several observations can be made. First of all, the case-study shows that the metric provides a solid quantified basis for discussing the decomposition of the subject system. In addition, the values of *CB* were show to indicate problems in the code which are related to the analyzability of the system.

A second observation is that the metric must be combined with metrics related to the dependencies between the compoments to come to a well-balanced conclusion. In our recently introduced architecture evaluation checklist [6], the metric is used to answer only one of 17 questions related to the structural decomposition, thus ensuring that other aspects are also taken into account. We plan to further investigate which coupling and cohesion metrics best complement *CB*.

As described in Section VI, the definition of the metric is currently limited to a single level of decomposition. Nevertheless, the metric could be extended to take into account hierarchical decompositions by, for example, using the inverse of the *CB* of each component as weights for the Gini coefficient on a higher level. The reasoning behind this is that the influence of the size of a component to the overall inequality in size distribution should be lower the better it is itself properly decomposed into sub-components. An adequate exploration of this alternative remains as future work.

XI. Threats to Validity

To put the results presented in Section X in perspective, here we address several questions related to the evaluation design.

A first threat that needs to be addressed is the presence and influence of ties in the expert ratings in Section VIII. To evaluate its impact, we compared the tied expert ratings and the metric rankings. This revealed that, for the two groups of ties, the corresponding *CB* ranks are contiguous with the exception of two cases. To quantify the impact of these ties we examined two scenarios of breaking up the ties. The first one corresponds to a worst-case scenario in which the opinion of the experts is the opposite of the ranking of *CB*, while the second scenario illustrates the best case in which the expert opinion gives the same ranking. This results in a significant correlation score between 0.6 for the worst case, and 0.9 in the best case scenario. Since both scenario's show a strong correlation, we believe that the influence of these ties is limited and does not invalidate our conclusions.

An important question related to the scope of the results is how far the results of the validation can be generalized (external validity). The projects used in the quantitative evaluation are all industry systems written in modern object-oriented languages, but differ in size, application domain and number of components. In principle, the generalizability of the results is limited to these types of systems. Nevertheless, since the metric is not specific for any language nor has any preference towards industry or open-source systems, we believe that the generalization is possible. One of the area's of our future work aims to validate this claim more formally.

Apart from the set of systems used in the evaluation, the generalizability of the results is also limited by the choices made in the instantiation of the repository (Section V-B). For example, using legacy systems, rather than modern ones, might lead to a different instantiation of *SB*, and thus lead to different results. A preliminary examination of our metric for some COBOL and Pascal systems revealed that they do not stand out as outliers with respect to the systems in our repository. However, more empirical work is needed to determine the impact of taking into account legacy systems.

Concerning the repository, we only examined software systems from the point of view of their main technology. However, today's software systems usually consist of multiple languages. To evaluate these hybrid systems one can either evaluate the decomposition of the system per language, or evaluate the system by combining all the languages. Experimenting with both scenarios to determine which one performs best is also part of our future work.

Lastly, a fact that influences the generalizability of the results is the use of industry experts working at a single company. To evaluate whether the same results can be obtained with different experts, the experiment described in Section VIII should be replicated using a more heterogeneous set of experts. However, the exact replication of the study is not possible due to the confidential nature of the industry systems used. This threat could have been countered by asking the experts to evaluate freely available open-source systems, but that would reduce the value of their opinion due to less familiarity with the systems. To counter the reduced repeatability, we have explicitly chosen an open-source system as the subject of the case-study and mixed industry and open-source systems in the repository used to instantiate the metric.

XII. Conclusions

In this paper, we make the following contributions:

- We describe an empirical exploration of how systems are decomposed into top-level components;
- We define a metric to measure the balance of components which is usable across all life-cycle phases of a project;
- We show how the metric is correlated with the opinion of experts about the analyzability of a software system.

Although the metric was developed in the context of the evaluation of software architecture, its definition is not constrained to a particular language, programming paradigm or level of abstraction. Actually, the metric can in theory be applied to any entity which is decomposed into a discrete set of components with a given size. This allows for the application of the metric on, for example, the requirements documentation of a software system, or to measure the quality of the section breakdown of a scientific publication. Investigation of these applications of the metric are also on our roadmap for future work.

References

[1] M. Babar, L. Zhu, and D. R. Jeffery. A framework for classifying and comparing software architecture evaluation methods. In *ASWEC '04: Proceedings of the 2004 Australian Software Engineering Conference*, page 309. IEEE Computer Society, 2004.

[2] R. Baggen, K. Schill, and J. Visser. Standardized code quality benchmarking for improving software maintainability. In *4th International Workshop on Software Quality and Maintainability (SQM 2010)*, 2010.

[3] G. Beliakov, A. Pradera, and T. Calvo. *Aggregation Functions: A Guide for Practitioners*. Springer Publishing Company, Incorporated, 2008.

[4] J. Blundell, M. Hines, and J. Stach. The measurement of software design quality. *Annals of Software Engineering*, 4(1022-7091):235–255, 1997.

[5] E. Bouwers, C. Lilienthal, J. Visser, and A. van Deursen. A cognitive model for software architecture complexity. In *Proc. 18th Int. Conf. on Program Comprehension*. IEEE Computer Society, 2010.

[6] E. Bouwers and A. van Deursen. A lightweight sanity check for implemented architectures. *IEEE Software*, 27(4), 2010.

[7] H. B. Christensen, K. M. Hansen, and B. Lindstrøm. Lightweight and continuous architectural software quality assurance using the asqa technique. In *Proceedings of the 4th European conference on Software architecture*, ECSA'10, pages 118–132. Springer-Verlag, 2010.

[8] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, Boston, MA, 2003.

[9] P. Clements, R. Kazman, and M. Klein. *Evaluating software architectures*. Addison-Wesley, 2005.

[10] J. Correia and J. Visser. Certification of technical quality of software products. In *Proc. 2nd Int. Workshop on Foundations and Techniques for Open Source Software Certification*, pages 35–51, 2008.

[11] L. Dobrica and E. Niemelä. A survey on software architecture analysis methods. *IEEE Trans. Softw. Eng.*, 28(7):638–653, 2002.

[12] C. Gini. Measurement of inequality of income. *Economic Journal*, 31:22–43, 1921.

[13] N. Harrison and P. Avgeriou. Pattern-based architecture reviews. *Software, IEEE*, PP(99):1, 2010.

[14] L. Hatton. Reexamining the fault density-component size connection. *IEEE Software*, 14(2):89–97, 1997.

[15] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *QUATIC '07: Proc. 6th Int. Conf. on Quality of Information and Communications Technology*, pages 30–39. IEEE Computer Society, 2007.

[16] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Trans. on Software Engineering*, 11(8):749–757, 1985.

[17] International Organization for Standardization. ISO/IEC 9126-1: Software engineering - product quality - part 1: Quality model, 2001.

[18] P. Kogut and P. Clements. The software architecture renaissance. *Crosstalk - The Journal of Defense Software Engineering*, 7:20–24, 1994.

[19] G. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63(2):81–97, 1956.

[20] S. Sarkar, A. C. Kak, and G. M. Rama. Metrics for measuring the quality of modularization of large-scale object-oriented software. *IEEE Transactions on Software Engineering*, 34:700–720, 2008.

[21] S. Sarkar, G. M. Rama, and A. C. Kak. API-based and information-theoretic metrics for measuring the quality of software modularization. *IEEE Transactions of Software Engineering*, 33(1):14–32, 2007.

[22] S. Shapiro and M. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3-4):591–611, 1965.

[23] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Syst. J.*, 13(2):115–139, 1974.

[24] M. Svahnberg. *Supporting Software Architecture Evolution*. PhD thesis, Blekinge Institute of Technology, 2003.

[25] R. Vasa, M. Lumpe, P. Branch, and O. Nierstrasz. Comparative analysis of evolving software systems using the gini coefficient. In *Proc. 25th Int. Conf. on Sw. Maintenance (ICSM)*, pages 179–188. IEEE, 2009.

[26] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., 1979.