Making Code Structural Quality Metrics More Relevant and Useful in a Project Context

Annapurna H.C. and Jagadish S.
Wipro Technologies
53/1, Hosur Road, Madivala
Bangalore - 560079
annapurna.hosahalli@wipro.com & jagadish.shri@wipro.com

Abstract: **Several experimental studies have shown that code structural complexity metrics have a high influence on the quality of software.**

Generally, code structural metrics for software are measured at each applicable unit level (method, class etc) by static analysis tools. Some tools also aggregate them in various ways to get a system level indicator for that metric. There is also an attempt to show a single health indicator for a piece of software combining multiple metrics in different dimensions by certain tools.

Our research attempt has been to establish a comprehensive structural analysis methodology to assess a piece of software from structural perspective in multiple dimensions which we believe is not much prevalent in industry. The objective of this paper is to bring out inadequacies of existing approaches and limitations of tools which we discovered during this journey. Further we also share a new comprehensive approach which was found to be more meaningful and useful in the project context. This approach has been validated by applying it in 40+ real-life projects covering 15 million lines of code in Java and C++ technologies.

I. INTRODUCTION

The ease and cost of maintenance of any software is a direct function of software quality. Quality of the software developed is greatly influenced by intrinsic quality, structure and organization of code [1]. In this regard, several studies and experiments have been conducted and several code complexity metrics have been established in the literature and by tool vendors at various levels of granularity like method, function, class and file. McCabe's Cyclomatic Complexity [2], C&K Metrics [3], Maintainability Index [4], traditional metrics like method length, comment to code ratio are some examples. The control of these metrics has proven to improve the maintainability of the software significantly as evidenced by the literature [5][6] and in our own study.

Most of the research studies and experiments so far have been aimed to prove that structural metrics have an impact on software maintenance effort and cost. These have been based on relative study like considering different versions of the same piece of software over a period or on experiments of crafted code bases and using selected metrics [5][6][7][8]. However not much thought has been given on how comprehensive structural analysis can be defined or used to assess a piece of software covering all dimensions.

Hence, our research goal was to check and explore if a comprehensive methodology can be defined to validate if a piece of software in a project scenario is structurally acceptable in all dimensions/levels. The following sections explain the primary aspects of establishing such a methodology.

II. SELECTION OF METRICS AND NORMS

As structure of a piece of software has multiple dimensions and can be assessed at different levels of granularity, the metrics/analysis/discovery applicable at each level is different. The typical levels of code organization are Statement level, Function level/Method level, Class level, Package/Module level and System level. The following table provides sample metrics that are applicable at each level.

TABLE 1: SAMPLE METRICS AT DIFFERENT LEVELS

Level of Code Organization	Metrics
Statement level	Syntax and coding standards adherence
Function/Method level	Method length, Cyclomatic complexity, Maintainability Index
Class level	C&K metrics (Number of methods per class, Depth of inheritance, Coupling between objects)
Package/System	Stability, Robert Martin's metrics, Anti- patterns

It is meaningless to apply a metric in isolation without a norm or standard against which measurement can be compared. An acceptable level of a measure i.e. norm must be specified for each of the selected metrics, to judge whether or not a code base is overly complex or unstructured. For some of structural metrics, there are several indicative recommendations of norms by various studies and tool vendors though the norms are not standardized across the industry in a formal way.

Also, there are a multitude of metrics defined by various researchers [2] and it can be quite confusing for a project team to select the right set of metrics and apply at right level.

So based on a set of criteria, we arrived at a shortlist of recommended metrics covering all dimensions/levels of code organization based on our experience of using these metrics in live projects with large code bases. A few of the primary criteria were coverage across various dimensions, proven in industry, non-redundant, ease of measurement, interpretation, impact on various quality parameters and availability of reliable norms. This marked the first step in making the structural metrics applicable and useful in a project context.

As the next step, we established norms for all the selected metrics based on the various researches, industry recommendations and our own expertise and experiments.

In general, most of the researches that define metrics at unit level [2][3][4][9] recommend the norms on relative scale. Even many of the static analysis tools (ex: PMD, CheckStyle, Rational application developer, and Rational software Analyzer, RSM metrics etc) provide option to define norms for these metrics and raise a violation when a particular unit does not meet the norms.

III. NEED FOR ADHERENCE INDICATOR/ AGGREGATE METRIC

From the above context, it is evident that software can be certified to be structurally good, if all units at each level adhere to the recommended norms specified for the structural metrics chosen. However, in the real project scenario, it is not feasible for each unit to adhere to the norms specified and have zero violations due to several reasons as listed below:

- Design constraints.
- Tradeoffs among different quality parameters.
- Tradeoffs between alternative design/architecture approaches.
- Tradeoffs between the different metrics on various dimensions.

Furthermore, if there is a system which is already developed or under maintenance, there is a very slim chance that it would adhere to the structural norms completely, especially if such a thought or mandate was not given during the development.

As a result we encounter a sizeable number of violations with any real life project.

The above situations demand for an adherence indicator at system level to assess the structural quality of software corresponding to each unit level metric. The unit level measurement values have to be aggregated in such a way to make the aggregation more useful. The metric should be a leading indicator towards the outlier units and hence aid in taking corrective actions.

IV. CURRENT ADHERENCE INDICATOR/AGGREGATION METRICS PREVALENT IN INDUSTRY

Given the need for a system level adherence indicator metric for a structural metric, we started exploring the available metrics prevalent in the industry (please note that adherence indicator and aggregate metric is used interchangeably in this paper context).

There are no research studies available attempting to define such aggregate indicator metric. Several research studies while doing relative structural metric impact [6] deploy sum, absolute number of outliers as an aggregate.

Most of the static analysis tools do not provide any type of such aggregate indicators (Ex: PMD, CheckStyle, Metric eclipse plugins, etc). Another set of tools (Ex: RAD, JTest, RSAR, RSMmetrics, JTest,

CMT++, Understand2.0) employ average, maximum, minimum or sum values to provide a system level indicator.

There are a few tools which attempt to provide a single indicator at system level (example: QAC/QAC++, CAST, Enerjy, Structure101). However they do not provide the information on how a specific metric is impacting this overall metric. As a result it is not clear how corrective action needs to be taken to influence the system level metric.

V. Shortcomings Of The Current Aggregate/ Adherence Indicator Mechanisms

As indicated earlier, common methods of aggregation for a metric at system level are average, maximum, minimum and sum values to provide a system level indicator.

However, these commonly used aggregate metrics suffer from fundamental drawbacks as follows:

- In general Object Oriented (OO) systems, the complexity and size follow the power law distribution. The power law distribution is commonly called as 80% - 20% rule. For example, typically about 80% of a code base falls into low complexity category whereas rest 20% falls into high complexity category. Hence the average complexity and size will be invariably low. This is illustrated by a specific example later.
- Normally, the getter and setter methods in Java (C++ and other OO) outnumber other methods and have complexity 1, which lowers the average numbers and masks problematic outliers.
- Maximum or minimum value indicators just show the degree of the variation, but not the number/magnitude of such severe violators.
- These aggregate metrics do not provide direct insight to any corrective actions to be taken as violating methods do not get highlighted in this measurement.
- Aggregate metrics based on “sum” values depend upon the size of module, file or system. The norms or conclusions cannot be applied, unless normalized against the size. If normalized, they suffer from same drawback as of average aggregates. Hence, they are not good aggregate measures.
- The maintainability issues follow 80-20% rule in general. 80% of the maintainability issues come from the 20% of a code base. Hence, even if 80% of a code base adheres to the norms and only 20% violates, the impact on the maintainability effort and cost is quite significant. However, the above aggregate metrics do not reveal this degree of violation.

To demonstrate/explain the above points, here are the metrics taken from a code base. The code base is of size 490 KLOC and has around 11769 methods. The distribution of the methods based on Method length and Cyclomatic complexity (CC) is shown below in Fig. 1 and Fig. 2.

In Fig. 1, we can notice that around 68% of the methods have Cyclomatic complexity of 1, owing to getter, setter, or simple constructor methods. These methods have lowered down the average Cyclomatic complexity, which turns out to be 3.99, which is well within the norm (the recommended norm for Cyclomatic complexity is < 10). However, there are around 7% of the total methods which violate Cyclomatic complexity on various intensities. If this happens to be part of 20% of code from which 80% of the maintenance problems arise, then we have a significant effect on the maintenance, although the average and percentage of violating methods looks very small.

Similarly, in Fig. 2, we notice around 58% of the methods have method length < 5 Logical Lines (LL) and about 13% of methods have Method length > 50 LL (which is the recommended norm for Method length) and the average Method length is 25. The large number of methods with Method length < 5 LL has lowered the average.

Both the above examples support the observation that the average metrics are skewed and cannot be the basis for evaluating a system.

VI. Definition Of New Adherence Indicator/ Aggregate Metrics

As average and cumulative measures are poor indicators, the alternative is to quantify the percentage of code (method/class/package) that does not meet the norm, thereby focusing clearly on the problematic outliers. However, this can also be reduced to an insignificant percentage if there are a huge number of methods/classes with very low complexity. To overcome this, we have defined a set of aggregate metrics using modified version of ‘percentage of violations’.

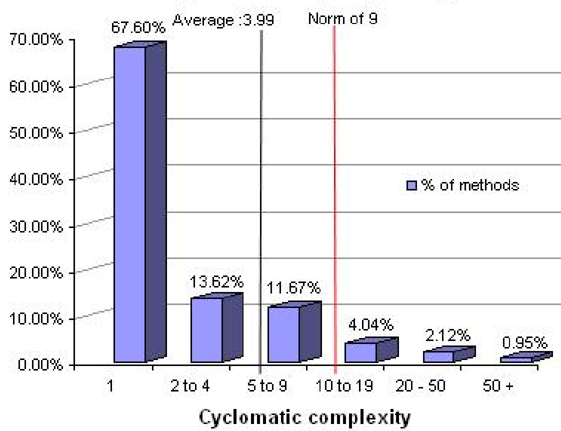

Fig. 1. Distribution of Cyclomatic complexity among Methods

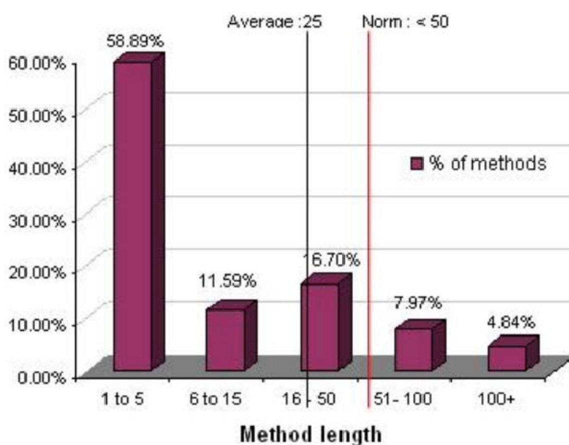

Fig. 2. Distribution of Method length among Methods

We have defined our new aggregate/indicator metric as 'Percentage of violations with respect to the norm for a metric after excluding a set of units based on specific criterion'. We have defined multiple aggregate indicators for a metric based on the degree of violation. For example, for Cyclomatic complexity (CC), the modified version of aggregate metric is the percentage (%) of violations calculated by excluding a set of methods, which fall below a minimum length criteria (for example methods with length < 5 LL are excluded), so that the results do not get skewed. We have 3 aggregate metrics for Cyclomatic complexity: Medium Cyclomatic complexity violation index, High Cyclomatic complexity violation index and Non maintainable Cyclomatic complexity violation index, corresponding to the degree of violation.

To illustrate the difference with new aggregate metric, let us reconsider the same example given in the previous section (Fig. 1). Without excluding the methods falling below the minimum length criteria, the violating methods fall into the categories as given in Table 2 adding up to a total of 7.11%.

TABLE 2: PERCENTAGE OF OUTLIERS WITHOUT EXCLUSION OF SHORT METHODS

Violation Category	Percentage of outliers
Methods with CC in the range 10 to 19	4.04%
Methods with CC in the range 20 to 49	2.12%
Methods with CC in the range 50 and above	0.95%

In the modified aggregate metric, we exclude the methods which are < 5 LL (there is very less chance that Cyclomatic complexity or nesting depth for a method is higher than norm, when logical length < 5 LL, hence probability of excluding a violator is very less). So, if we recalculate the aggregate metric after exclusion, it gives the violation distribution as depicted in Table 3, which add up to a total of 17.31%.

TABLE 3: NEW AGGREGATE METRICS FOR CYCLOMATIC COMPLEXITY (PERCENTAGE OF OUTLIERS WITH EXCLUSION OF SHORT METHODS)

Violation Category	Percentage of outliers
Medium CC violation Index (CC in the range 10 to 19)	9.84%
High CC violation index (CC in the in the range 20 to 49)	5.15%
Non maintainable CC violation index(CC in the range 50 and above)	2.32%

It is interesting to note that if we have used this exclusion of methods (methods < 5LL) while calculating the averages, still the average Cyclomatic complexity in the above case would be 7.8, which is still well within the norm and masks the outliers.

The new definition of aggregate metrics is applicable broadly at a method and class level. At package level aggregate metrics is not applied currently, and it needs further investigation.

These defined metrics will have meaning only if they are compared against a norm or standard to qualify if it is acceptable or not. So the acceptable level of adherence to

structural metrics is defined by norms of the aggregate metrics based on the experience in real projects.

VII. Results With New Adherence Indicator Metrics

We applied these new aggregate metrics and violation norms in 40+ projects comprising of 15 Million lines.

The project teams so far were not giving much importance to the structural metrics though metrics were measured. It was mainly due to the inherent inability to adhere to norm for every unit for every metric. Hence Structural metrics were felt irrelevant. Furthermore they were misguided by overall average of each structural metric which is generally well within the norms. There were no enablers for prioritizing or focusing on the problematic outliers. This new methodology of structural analysis provided relevant and meaningful assessment, which when used to control the quality accordingly would reap benefits in controlling the total cost of ownership. The initial results show that these metrics along with norms have shown better correlation with the quality of code derived out of subjective evaluation.

The merit of this methodology is demonstrated below by taking an example from real life projects.

Two code bases of same technology, domain and customer were taken for the analysis after the completion and delivery of the projects. Both were development projects executed using the iterative software development lifecycle model. Both were evaluated for their structural quality based on the prevalent aggregation methodology i.e. average and the new methodology. To provide a fair comparison, the averages were considered after excluding units with the minimum length criterion as in case of new aggregate metrics. The comparative results are depicted in Table 4. Table 5 presents other operational parameters for each project.

TABLE 4: Comparison Of Structural Code Quality Adherence Of Projects In Two Methodologies

Project Name	Adherence by using 'averages' Methodology (after exclusion principle)	Adherence by our new defined aggregation methodology
Project 1	All aggregate metrics adhered to the norms.	All aggregate metrics adhered to the norms.
Project 2	All aggregate metrics adhered to the norms.	All aggregate metrics violated the norms.

TABLE 5: Comparison Of Operational Parameters Of 2 Projects

Operational Parameters	Project 1	Project 2
Customer feedback/ rating	Rated as high quality code by client and rated 5/5 on customer satisfaction	-
Delivery adherence	On time	Re-estimation required.
Team feedback	Comfortable and in control	Challenges encountered
System testing defects	1	34
Post delivery defects	0	0.37/KLOC
First time Test pass	100%	90%

As it can be seen that the project 2 which had poor operational parameters was shown as a violator in the new methodology, however, found to be in the acceptable category as per earlier methodology of aggregation using 'averages'.

The example proves that the new method of aggregation is superior and has better correlation with operational parameters. It would have created enhanced impact in controlling the quality and operational parameters, if this new methodology was adopted earlier and corrective actions were taken.

VIII. Feasibility Of single Structural Quality Indicator

As indicated earlier, software Structural metrics are along the various dimensions (statement, method, class/package, system and file). One more aspect covered was the need for aggregate metrics and norms for a particular metric. Considering the multitude of dimensions, metrics and aggregates a question which would naturally arise is – *"is there one single indicator for the overall structural health of the software system?"* This would naturally be a further extension to the aggregation method introduced previously.

There are arguments both for and against having a single indicator. A few of them are:

For:

- It would be good to have a single indicator especially from an organization level dash-boarding perspective (x% of projects are okay based on the single indicator)

- In a maintenance project, this could be a product quality trend indicator release on release.

Against:

- The overall indicator cannot trigger any corrective action, by itself. One needs to look at the specific dimensions of violation, for taking corrective actions.
- A single metric might miss out the richness of information contained in the multiple dimensions.

Rather than being limited by arguments, we decided to explore this further by evaluating various single metric options:

1. Sum of values of all aggregate metrics
2. Weighted sum of values of all aggregate metrics
3. Weighted sum of values of selected[1] aggregate metrics
4. Weighted sum of values of rating of violations (The Violation ranges were rated on a scale 1-7)

In each of above cases, the values were normalized against the values of allowed violations. For example, in the case of second option, the single indicator may be defined as below:

$$\text{Structural Quality Indicator} = \frac{\text{Weighted sum of values of aggregate metrics of a codebase}}{\text{Weighted sum of values of norms of aggregate metrics}}$$

As per the definition above, if the value of Structural Quality Indicator is < 1, then the code base is within acceptable structural health range. If the value is 0, then it is excellent in terms of structural quality. However, if the value >1 then the quality is not acceptable and the structure of the code base needs improvement.

Nevertheless, all the above options suffer from a fundamental flaw as explained further. A code base may score good values for a few metrics but score unacceptable values for a few other metrics. With the computation of a single indicator, there is a possibility of one severe violation being masked by the cumulative favorable effect of other scores. We can illustrate this with an example. Table 6 gives the sample details of violation norms and values of actual violations for a real project - “project A”.

TABLE 6: SAMPLE AGGREGATE METRIC VALUES OF PROJECT A AND NORMS

Sl No.	Aggregate level violations	Norms	Project A
1	Medium Cyclomatic complexity violation index	5.00%	0.00
2	High Cyclomatic complexity violation index	1.00%	0
3	Non maintainable Cyclomatic complexity violation index	0.00%	0.00%
…	…	5.00%	0.00%
…	…	15.00%	7.58%
…	…	20.00%	0.00%
…	…	…	…
23	Low Comment to code ratio violation index	20.00%	**85.53%**
24	High Comment to code ratio violation index	10.00%	1.32%
	Sum of all violations	**280.30%**	**212.84%**

The Structural Quality Indicator for *Project A* is 0.76(=212.84/277.30), hence Project *A would* get rated as very good by Structural analysis. However, it can be seen that the ‘Low comment to code ratio violation index’ (shown in white colored font) is violated severely, but fails to show up in the overall Structural Quality Indicator due the masked effect caused by the favorable contribution from other metrics’ values.

Thus, this example illustrates that if a single overall indicator is used, there is always a chance of misjudgment because of the minority behavior getting masked by the majority behavior. A severe violation may go unnoticed and left unattended.

So, the above proposal of having a single structural indicator was discarded. Instead, we could go by the indication, that the software is good if there is no violation on any of the aggregate measures and in all other cases there is a need for improvement (the number

[1] We selected the metrics out of the recommended list in such way that the impact is not double counted. For example: at method level, metric *Maintainability index without comments* includes *Cyclomatic complexity* and *Method length* in its computation, hence maintainability index was included whereas the *Cyclomatic complexity* and method length were excluded.

of dimensions on which there is a violation is an indicator of the degree of 'structural weakness' of the software) and action needs to be taken.

This is well supported by the medical diagnostics analogy in that the one single indicator of health of a person is the absence of abnormalities on each key dimension (like blood pressure, blood sugar, ECG etc). An abnormality on any single dimension is a cause for concern and needs to be addressed. It would not make sense for example to combine a bad blood pressure metric value and a very good blood sugar metric value to come up with an "average" overall health indicator which does not indicate anything.

IX. CONCLUSION

Above study proves that structural metrics are meaningful, effective and useful to control the quality of the software only with suitable aggregate metrics in addition to unit level metrics. Inappropriate or lack of aggregate metrics will not provide sufficient information to use these unit level metrics effectively.

In addition to this, using of single overall health indicator for a system (combining metrics in all dimensions) is discouraged as it will lead to misjudgment as demonstrated by the example taken in this paper.

ACKNOWLEDGMENT

We would like to express our thanks to all project teams who participated in the analysis which helped us to validate and refine this methodology.

REFERENCES

[1] ISO/IEC 9126/1 -Software Product quality model.

[2] McCabe, Thomas J., A Complexity Measure, IEEE Transactions on Software Engineering, SE-2 No. 4, pp. 308-320, December 1976

[3] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design, IEEE Transactions on Software Engineering, 20(6):476–493, 1994.

[4] Edmond VanDoren, Maintainability Index Technique for Measuring Program Maintainability, SEI STR report, Mar 2002

[5] D.Kafura and G.R.Reddy, The use of Software complexity Metrics in Software maintenance, IEEE transactions on software engineering, Vol.SE 13, No.3, March 1987

[6] R.K.Bandi, V.K.Vaishnavi and Daniel.E.Turk, Predicting Maintenance Performance using object oriented design complexity measures, IEEE transactions on Software Engineering, vol 29, No.1. January 2003

[7] Rajiv D. Banker & Srikant. M. Datar, Chris F. Kemer and Dani Zweig, Software complexity and maintenance costs, Communications of the ACM, Vol 32, November 1993

[8] Andreas Epping & Christopher M. Lott, 19th Annual Software Engineering Workshop, 30 Nov–1 Dec 1994.

[9] Seyyed Mohsen Jamali, Object oriented metrics, Survey paper, Sharif University of Technology, Tehran Iran, Jan 2006.