A Framework for Source Code metrics

Neli Maneva, Nikolay Grozev, Delyan Lilov

Abstract: *The paper presents our approach to the systematic and tool-supported source code measurement for quality analysis. First, we describe the results of the performed thorough study about the use of some static measurement tools. We classify them as reporting and combining and manage to elucidate the main requirements for a feasible framework, supporting a set of such metrics. Next the details about the design and implementation principles of the framework are given. In conclusion we share some ideas about future scientific and practice-oriented work in this area.*

Key words: *Software Metrics, Source code, Static analysis, Framework, Program quality.*

INTRODUCTION

Software measurement has been introduced as a promising approach but now "*it is a professional embarrassment as of 2009 and urgently needs improvement in both the quantity and quality of measures*" [2]. In practice, the majority of software companies measure poorly and do not keep track of historical data. Thus they are unable to monitor and assess efficiently software development so as to improve the process and product quality. This premises the great need for tools that help in the process of measuring and analyzing measurements values.

Different software engineering artefacts can be automatically measured, but we focus our research on source code measurement only. This is a reasonable choice because the source code is one of the few mandatory and one of the most important ingredients of a software system, since it is the primary way to implement functionality. It can be argued that there exist software systems created by only configuring already existing components to work together, but this is a rare case. Even in systems that rely solely on such external components there is the so called ***glue code***, which helps the interaction among the different components. Unfortunately, this glue code is often a source of defects.

The program code is therefore a natural target for a diversity of tools for discovering and predicting vulnerabilities and different problems. Monitoring and analyzing different quantitative aspects (e.g. metrics) of the code has long been seen as a way for targeting these goals. It has been decades since the first metrics for source code have been developed, and since then many others have also emerged. Hundreds of theoretical and experimental research projects have been carried out to evaluate the applicability of the different metrics. This gives a solid ground for the development and incorporation in the software lifecycle of tools that automate the computation of metrics values and aid the team members in interpreting these values in order to achieve better understanding of the code structure and other quality characteristics.

BACKGROUND AND ANALYSIS

Many of the above mentioned tools exist, and we have reviewed and experimented with several of the available ones. We have discovered common features in their functionalities. Therefore we propose to classify them as ***reporting tools*** and ***combining tools***.

The tools that we call ***reporting tools*** have the functionality to compute a predefined set of metrics and produce reports of some kind (GUI form, pdf files, html files etc.).

Most of these tools have predefined thresholds and interpretation intervals for each domain of metric values. These are used to put emphasis on inappropriate values of metrics in the generated reports. In many tools these thresholds and intervals can be modified by the user in order to achieve more useful reports.

The main disadvantage of the ***reporting tools*** is that the user must be aware of the definitions and peculiarities of all the metrics represented in the report in order to reason about the quality of the code. A software team member may find a significant use in a reporting tool by acquiring knowledge about a few of the well known for their utility metrics (i.e. lines of code, cyclomatic complexity). Much more can be achieved by employing a wider "arsenal" of metrics. However, it is not realistic to expect that in every team there is a member, who has deep insight into the peculiarities of such a wide range of metrics.

The tools that we call ***combining tools*** try to tackle this specific problem. Typically a ***combining tool*** has most of the features of a ***reporting tool***, but in addition it can combine the values of the metrics in order to extract specific knowledge about the code being analyzed. This knowledge usually is in the form of a new measure, or a highlight of a specific design problem (anti-pattern). This "digested" knowledge actually represents an interpretation of the values of the metrics and allows the users to ignore the details of the concrete metrics.

In our opinion (supported by several research projects e.g. [1], [4]) the best choice of combinations of metrics relies heavily on the ***context*** of the analyzed system. By ***context*** we mean all factors that influence the quality of the code and the specific requirements for the quality of the code. Examples of such factors are the used programming languages, programming frameworks and libraries and the application area. Unfortunately, the majority of the ***combining tools*** do not have the functionality to easily adapt their combinations to the current context. Even more, many of these tools employ combinations, extracted by statistical and data mining techniques over a huge set of source code files. Such combinations are very hard to be changed in accordance with a specific context, because this would include defining a new training set and reapplying the statistical and data mining algorithms over it.

Another major problem that is observed in both ***reporting*** and ***combining tools*** is that the metric computation is not context-dependent. To illustrate this problem, we give an example with the ***Coupling Between Objects (CBO)*** metric, which is defined as "*number of classes to which a class is coupled*". It is quite natural to skip certain classes that do not increase the complexity of a class from this count (i.e. some languages have standard classes like String, Logger etc). Some tools have the functionality to skip such classes which lessens the "noise" in this metric value. In some cases it is better to extend the set of skipped classes, for example when using a standard and stable library. A typical example is when some GUI components are developed. In this case a class representing the GUI form usually depends on a lot of classes, representing all kinds of controls (buttons, labels etc.). It is often better not to include these dependencies when computing this metric for classes representing GUI forms. However, such dependencies must be included when computing the metric value for classes implementing the business logic of a system. This comes to illustrate how important it is that the computation of the metrics is context-dependent. We are currently unaware of any tools that provide adequate means to input context information and compute and combine metrics values in accordance with this information.

OUR APPROACH TO SOURCE CODE MEASUREMENT THROUGH A FRAMEWORK

1. Framework requirements

Based on the previous analysis we have identified and will outline the properties

which we consider as essential ones for a source code metrics framework. Naturally such a framework must have functionality to compute and interpret a set of metrics. The computation of these metrics should be done in such a way, so that it is possible for a user to tune the computation of the metrics in accordance with the context. Besides computing metric values the framework should also have the functionality to combine these values so as to produce specific knowledge about the code. These combinations should also be context-dependent and a user should have the ability to change them or to define new combinations that are more suitable for his or her purposes.

These configurations for determining the context, however, may turn out to be tedious and lengthy to be done by a user. Besides, the users may not have the needed knowledge to determine these settings. In this respect, the framework should employ a "convention over configuration" policy and provide sensible "out of the box" configurations for often recurring contexts.

2. Design and structure of the framework

Among the key ideas of the proposed framework is that a ***base set of metrics*** must be defined. This set of metrics comprises all metrics whose values are computed by the framework. The algorithms for the computation of these metrics values are built in the framework and the user is not given the ability to define new metrics (except for metrics combining measures from the ***base set of metrics***). For each of the base-set metrics there should be an interpretation for its values.

It is important to point out that the structure of the framework is not dependent on the specific metrics in this set. This set must however be constructed in such a way that a wide range of combinations for a wide range of contexts is possible. The other key idea is that the computation of the measures and their combinations are distributed into distinct modules. To simplify things we represent these modules as functions. The framework comprises the following types of functions:

- **Metric functions** – specialized in the extraction of the value of a single metric from the ***base set of metrics*** for a given code unit. Thus for every metric in the ***base set of metrics*** there is a corresponding metric function. Each metric function takes as parameters the source code artefacts needed to produce the corresponding metric values.
- **Preprocessor functions** – used to prepare the parameters for the metric functions. Each metric function has a single preprocessor function assigned to it. Typically the preprocessor functions implement filtering of the artefacts, used for the computation of a metric value from elements that cause this value to be inaccurate in some context.
- **Evaluation functions** – used to combine the values of the metrics from the ***base set of metrics*** to a value of a custom metric or a specific design problem (i.e. an anti-pattern). Typically these functions are used to combine measures into a meaningful evaluation of a code quality.

The process of evaluating a source code unit starts with the extraction of the code artefacts (control flow graphs, dependency graphs etc.) that are to be used by the metric functions to compute the values of the metrics in the ***base set of metrics***. This extraction is out of the scope of this paper. After the artefacts are extracted, the preprocessor functions are used to refine the parameters for the corresponding metric functions. After the refinement of the parameters, the metric functions are used for producing the values of the metrics. These values are then used by the evaluation functions to produce different evaluations of the code. *Fig.1* illustrates how the metric, the preprocessor functions and an evaluation function can be used in combination to achieve an evaluation of the code:

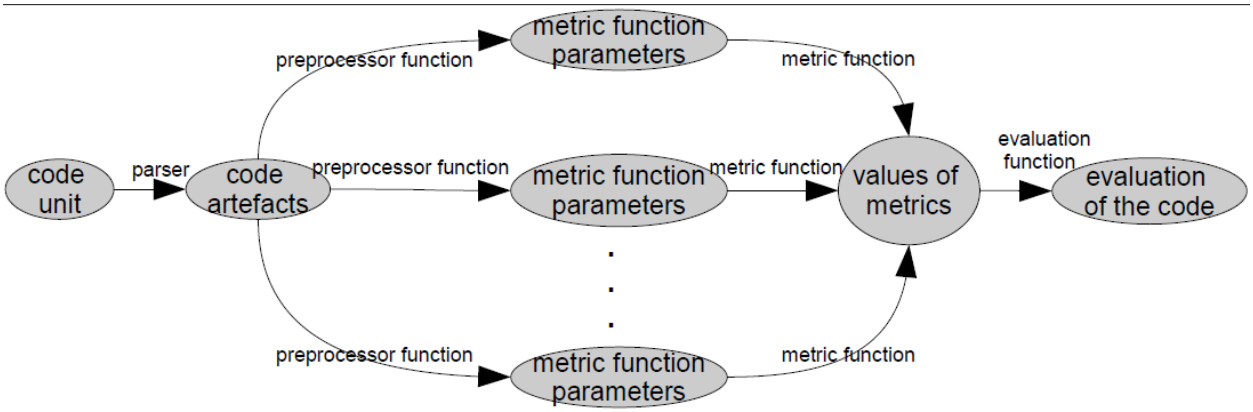

Fig.1 Illustration of the evaluation process

3. Evaluation and preprocessor functions in detail

In the computational schema described above there are two major extension points, where the contextual tuning of the metrics and their combination can occur – the evaluation and the preprocessor functions. In this section we explain these functions and their purposes in detail and provide some examples.

3.1. Preprocessor functions

Informally, the purpose of a preprocessor function is to determine which artefacts are relevant for the computation of a metric. Usually these functions only filter some artefacts irrelevant to a metric function input. Thus a typical implementation of such a function would be additionally parameterized with values, determining which artefacts are relevant for the code unit being evaluated. The user has to specify these values. The mechanism for this parameter passing from the user to the framework is not in the scope of this paper, even though it is important for the usability of the framework.

To illustrate the interaction between the preprocessor and the metric functions we will give an example about the computation of the above mentioned ***CBO*** metric value. The metric function for the ***CBO*** metric takes as input the dependency graph of the classes in an application. Thus the corresponding preprocessor function produces a dependency graph, which is a subgraph of the original graph and includes only the nodes representing classes that should be taken into consideration when computing the metric value. This new graph is then passed to the metric function.

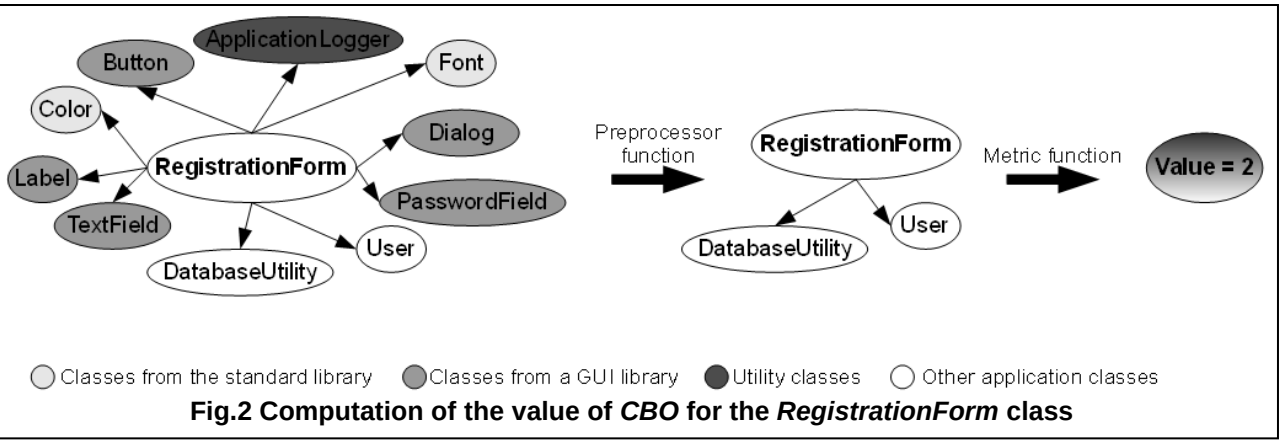

Fig.2 Computation of the value of *CBO* for the *RegistrationForm* class

Fig. 2 illustrates the computation of the CBO value for a class *RegistrationForm*, which represents a standard GUI form for registering a user in a system. *RegistrationForm* uses standard GUI components and communicates with classes, implementing the business logic. Assuming that the user has specified that *RegistrationForm* is a custom GUI component whose complexity is not increased from depending on standard GUI classes and classes implementing common functionality (i.e. Loggers, Utility classes etc), the preprocessor function has to remove these classes from the dependency graph.

3.2. Evaluation functions

Evaluation functions are also a way to specify context-dependent behaviour of the framework. Since they are used to combine the metrics values, any "noise" in these values should be minimized (by the proper implementations of the preprocessor functions).

Typically there are two types of evaluation functions – functions that ***highlight*** a particular design problem in the code and functions that ***combine*** the values of the base metrics into an evaluation of some sort of a quality of a code unit (i.e. an evaluation of the maintainability of a class in the range 1-10). As to the former type of evaluation functions, there are already promising researches into the area of OOP design problem recognition using metrics [3]. However, these approaches do not use context-dependent computation of metrics values. We believe that the context-dependent computation of metrics values in our framework, implemented through the preprocessor functions, can lessen the "noise" in the values of the metrics and can improve the success rate of these approaches.

The evaluation functions from the latter type combine the values of a subset of the ***base set of metrics*** in order to produce a new value, describing a target quality of a code unit. We call this subset of metrics an ***evaluation set***. The ***evaluation set*** typically consists of all the metrics which can be used to evaluate the target source code quality for some kind of code units (i.e. all metrics, whose value indicate the maintainability of a function). A natural approach for implementing such evaluation functions is to define their results as a linear combination of the values of the metrics from the corresponding ***evaluation set***. Another approach is to use some sort of guided machine learning techniques to determine useful combinations. Following this approach the user has to input evaluations of code units (based on his or her expert knowledge). These evaluations would later be used to determine the ***evaluation set*** and a function (parameterized with this set) that approximates the evaluations, provided by the user.

The approach using machine learning techniques does not require the user to have previous knowledge about metrics and thus has a clear advantage over the approach using linear combinations. Evaluation functions that use explicit linear combinations may be used by users with expertise in metrics or as default implementations.

Both of these approaches for defining evaluation functions, however, observe a common problem. It is that when combining the metrics values inappropriate ones may be "covered" from the values of other metrics. A value of a metric is considered inappropriate if according to the metric interpretation, this value implies very low quality of the measured element. That way a combination may turn out to be deceiving and to mislead the user in certain cases. To illustrate the problem let's consider an evaluation function **f**, which has an evaluation set of metrics - $\mathbf{m_1}$, $\mathbf{m_2}$ … $\mathbf{m_n}$. Its value is defined as $\mathbf{f(v_1, v_2, \ldots v_n) = a_1v_1 + a_2v_2 + \ldots + a_nv_n}$, where $\mathbf{v_1}$... $\mathbf{v_n}$ are values of the metrics from the evaluation set for some source code unit and $\mathbf{a_1}$... $\mathbf{a_n}$ are real-valued coefficients. If a value $\mathbf{v_i}$ of a metric $\mathbf{m_i}$ is inappropriate, then it is possible that this is "covered" by appropriate values for the other metrics from the evaluation set and the coefficients $\mathbf{a_1}$... $\mathbf{a_n}$. For some metrics, essential to the evaluated quality, this is not an acceptable behaviour. The problem occurs mostly for evaluation functions that are defined as explicit linear combinations, but may also occur in functions defined through some machine learning algorithms.

To solve this problem we introduce a new set of metrics called **critical domain**. The critical domain is usually a subset of the evaluation set and consists of metrics, whose values are essential to the target quality being evaluated. Thus if a critical domain is specified, the evaluation function checks whether the values of the metrics in the critical domain are appropriate. If any of them is not appropriate, the evaluation function returns an inappropriate value itself. If all values in the critical domain are appropriate, then the computation proceeds by applying the corresponding combination.

Informally, the critical domain may be thought of as a security net. It is a way for the users to specify that they would like to observe a specific combination of metrics that does not allow certain inappropriate values.

Since the framework should be easy to use with minimal efforts for settings, there should be default implementations for the preprocessor and evaluation functions. These default functions are however very much dependent on the programming languages and technologies that are used and we will not describe them here.

CONCLUSION

Our study about the practical use of the described framework validates the feasibility and usefulness of the chosen approach to source code measurement. We demonstrated that the framework fulfils all of the defined requirements and that it is flexible enough to be used successfully in a variety of contexts.

Our future research activities will be focused mainly on the further development of specific methods for metrics preprocessing and combinations. Besides, future research into appropriate adaption of machine learning and visualization techniques can be beneficial to the usability of the proposed framework. Last but not least, further research is needed to determine the best way to incorporate the framework throughout the software lifecycle.

ACKNOWLEDGEMENTS

This work was partially supported by the National Innovative Fund attached to the Bulgarian Ministry of Economy and Energy (project № 5ИФ-02-3 / 03.12.08).

REFERENCES

[1] G. Denaro and M. Pezze, "An Empirical Evaluation of Fault-Proneness Models", *Proceedings of the 24th International Conference on Software Engineering*, 2002, pp. 241-251

[2] C. Jones, *Software Engineering Best Practices*, Mc Grow Hill, 2010

[3] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*, first edition, Berlin: Springer Verlag, 2006.

[4] N. Nagappan, T. Ball and A. Zeller, "Mining Metrics to Predict Component Failures", *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 452-461

ABOUT THE AUTHORS

Assoc.Prof. PhD Neli Maneva, Software Engineering Department, Institute of Mathematics and Informatics - BAS, "Acad. G. Bonchev" str. bl.8, 1113 Sofia, Bulgaria, tel. (02) 979 28-75, e-mail: neli.maneva@gmail.com

Nikolay Grozev, Musala Soft Ltd. 36 Dragan Tsankov blvd. Sofia, Bulgaria +359 2 969 58 21, nikolay.grozev@musala.com

Delyan Lilov, Musala Soft Ltd. 36 Dragan Tsankov blvd. Sofia, Bulgaria +359 2 969 58 21, delyan.lilov@musala.com