Indonesian Journal of Data and Science

Volume 4 Issue 2
ISSN 2715-9936
https://doi.org/10.56705/ijodas.v4i2.90

Research Article

M2SmallLint: software health monitoring tool

Hayatou Qumarou [1,*], Nurul Rismayanti [2]

[1] Department of Computer Science, HTTC, University of Maroua, Cameroon, Oumarou.hayatou@univ-maroua.cm
[2] Universitas Muslim Indonesia, Makassar, Indonesia, nurulrismayanti.labfik@umi.ac.id

Correspondence should be addressed to Hayatou Qumarou: Oumarou.hayatou@univ-maroua.cm

Received 12 June 2023; Accepted 28 June 2023; Published 31 July 2023

© Authors 2023. CC BY-NC 4.0 (non-commercial use with attribution, indicate changes).
License: https://creativecommons.org/licenses/by-nc/4.0/ — Published by Indonesian Journal of Data and Science.

Abstract:

Many studies have shown that appropriate visualisation of software artefacts can significantly reduce the effort required to understand and maintain the system. In recent years, several studies have focused on the visualisation of software quality using various techniques and tools. The authors agree on the use of the absence of bugs and programming rule violations as the main measures of software quality. In addition, metrics such as the Lack of Cohesion in Methods (LCOM) and the Weighted Methods per Class (WMC) of the Chidamber and Kemerer (CK) suite are used to measure software quality. In this paper, we propose a model for visualising software quality using several recognised quality metrics. We have implemented this model in Moose, an open-source software for data analysis, and Pharo, a pure object-oriented programming language and powerful environment focused on simplicity and immediate feedback. The result is a tool for visualising the health of software. It has two main advantages: (1) it allows navigation through the properties of the source code to locate the entities that present quality problems, and (2) it gives an overall view of the quality state of the software. The expert evaluations of the tools highlighted the usefulness and usability of the tool.

Keywords: Software data visualization, Software quality, Software metrics, Software tools.
Dataset link: -

1. Introduction

Several tools have been developed to detect programming rule violations (alerts) and potential bugs in software systems. These tools (Rule Checker) are based on several techniques [1], [2], including theorem provers (eg ESC/Java), model checking (eg Bandera), adhoc methods (eg ISA), syntactic analysis (eg PMD) and dataflow analysis (eg JLint).

On the one hand, studies have shown that these tools generate a very high number of alerts. For example, in the authors present a large-scale experiment using 11 repair tools based on a Java test suite and 5 bug tests. According to the terminology proposed by [3] an alert will be called actionable (true) if it has been fixed by the developers. Whereas an alert that has not been corrected is said to be inactionable (false). So, an actionable alert detected by a tool will be called true positive, if it is not detected it will be called true negative. An inactionable alert will be said to be false positive if it is detected by the tool, whereas it will be said to be false negative if it is not. Kremenek [4] has shown that the false positive rate in alerts generated varies from 30% to 100%. The very large number of alerts generated by these tools and the high false positive rate discourage engineers from examining them and correcting the true positives.

On the other hand, despite the growing number of source code defect analysis tools, they do not provide any real mechanisms for assessing correlations between alerts generated by rule checkers and the internal quality of the software system. To our knowledge, no specific study has been carried out in this area. More specifically, no tool has been designed to highlight the causal relationships between source code properties and alerts generated by rule checkers.

In this article we propose and describe M2SmallLint, a tool for viewing and navigating through source code properties in relation to alerts generated by Rule Checkers. Specifically, this tool allows us to:

- generate alerts (of a given type) for a given package;
- view alerts generated in this way by method;
- navigate through the properties of the source code in relation to rule violations.

This will help the developer to answer the following questions, in order to improve the quality of their source code:

- What type of alerts are most frequently detected?
- Which method/class/package has the most/least alerts?
- Which method/class/package is reliable in terms of the number of alerts detected?

Our article will be organised as follows. Section 2 presents the work relating to our field. Section 3 presents M2SmallLint. Section 4 presents the perspectives of our work and we conclude in section 5.

Hatari [5] is a tool that locates code at risk. The tool gathers data from a version management system (eg CVS), and a problem tracking system (eg BugZilla) in order to locate the piece of code whose modification introduced the bug. The location thus found is marked as being at risk by a set of coloured bars. Following the same logic, BugMaps [6] has been developed to visualise the evolution of bugs in a system over time. BugMaps-Granger [7] an extension of BugMaps, shows the causal relationships between bugs and certain metrics in the object-oriented paradigm. Other tools such as Vis-aVis [8] VisioTM [9] MossaiCode [10] Rigi [11] etc. are used to visualise the structure of systems. M2SmallLint takes its inspiration from HATARI and BugMaps-Granger to locate in the source code the locations representing alerts generated by a Rule Checker that have a high probability of being real bugs.

Method:

M2SmallLint

The M2SmallLint tool is based on SmallLint and Moose. The following sub-sections present SmallLint, Moose and M2SmallLint respectively. M2SmallLint consists of two modules. The graphics module and the navigation module. **Figure 1** shows its structure.

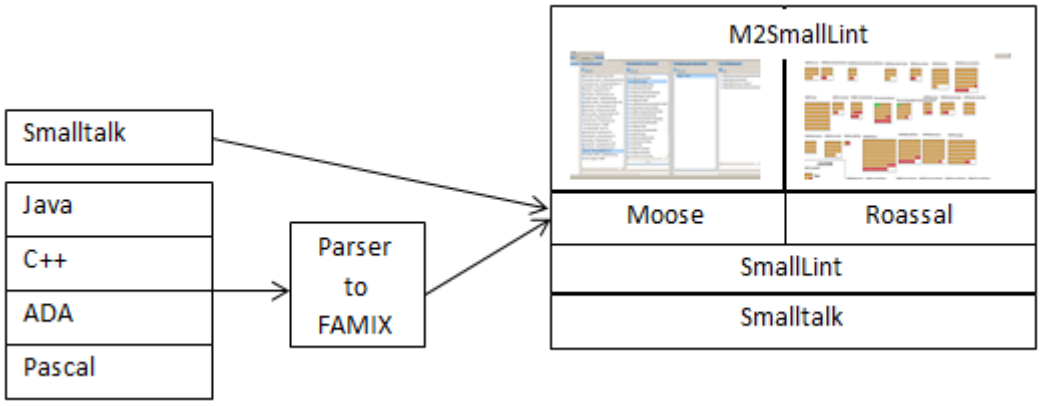

Figure 1. Structure of M2SmallLint

SmallLint is a code analysis tool included in Pharo [12] to detect bugs or possible errors. It seeks to identify a hundred or so possible problems, such as methods that are too long, possible bugs, unnecessary code, etc. Moose [13] is a Smalltalk implementation of the FAMIX meta model [14] which allows the concepts of the object-oriented paradigm to be presented independently of the language. Roassal is a tool developed to visualise and act with arbitrary data, defined in terms of objects and their relationships. Roassal is generally used to produce interactive visualisations.

The Graphics Module

This module receives as input the alerts generated by SmallLint and some metrics (eg number of lines of code) from the source code derived from the object-oriented paradigm. Based on this information, the module offers several visualisations. Each visualization is a two-dimensional representation, based on four metrics, as shown in the following **Figure 2**:

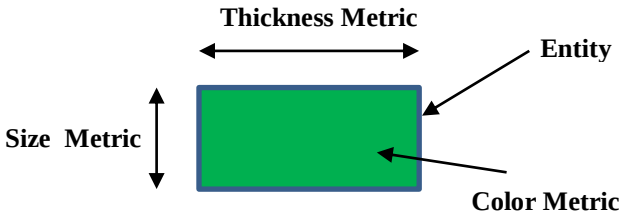

Figure 2. M2SmallLint visualisation metric

Thickness metric: we use the instruction number

Size metric: we use the size of the vocabulary. This is equal to the sum of the number of distinct operators and the number of distinct operands.

Colour metric: depending on the Entity Color (EC) score, colours range from light green to dark red.

The formula is given by :

$$\mathbf{EC}= 1 - \frac{e^{(WMC)}}{LCOM + e^{(WMC)}} \quad (1)$$

Calculation of Cyclomatic Class Complexity (WMC)

The software engineering community has its own way of characterising software quality. With regard to object-oriented software, several metric suites have been proposed to assess its quality, such as the CK [15] metric suite, the MOOD [16] etc. Thus, the aim of Object-Oriented software metrics is to provide a quantitative view of how these mechanisms have been applied to achieve these objectives. For our work, we use the WMC. These are defined to characterise the complexity of a class. The WMC is defined as:

C_K represents the complexity of each method the class

$$WMC = \sum_{k=1}^{m} c_k \quad (2)$$

WMC is a measure of the complexity of a class. The higher the number, the more likely it is to be complexity-specific, limiting the possibility of reuse. In software development practice, it is recommended that WMC is kept low.

Cohesion metric (LCOM)

A very important feature of object-oriented programmes is cohesion. A class must represent a single logical concept and not be a collection of diverse functionalities. The purpose of a cohesion value is to say to what extent a class fulfils this design principle. "Lack of Cohesion Metric (LCOM) [17] is a metric that informs us about the level of cohesion of a class. But for our work we will use the following formula:

$$\text{LCOM}= 1 - \frac{\sum \text{MF}}{\text{M x F}} \quad (3)$$

- M is the number of methods in the class (includes static and instance methods, constructors, properties, etc.).
- F is the number of instance fields.
- MF is the number of methods in the class calling a given field.
- Σ (MF) is therefore the sum total of the value obtained by MF on all the fields in the class.

Entity: as an entity we distinguish methods, classes and packages

Relationship: we essentially have the method/class/package membership relationship

3. Results and Discussion

Case Study

We applied M2SmallLint to the *Famix-Core* package of Pharo 9.0 and obtained the following **Figure 3**.

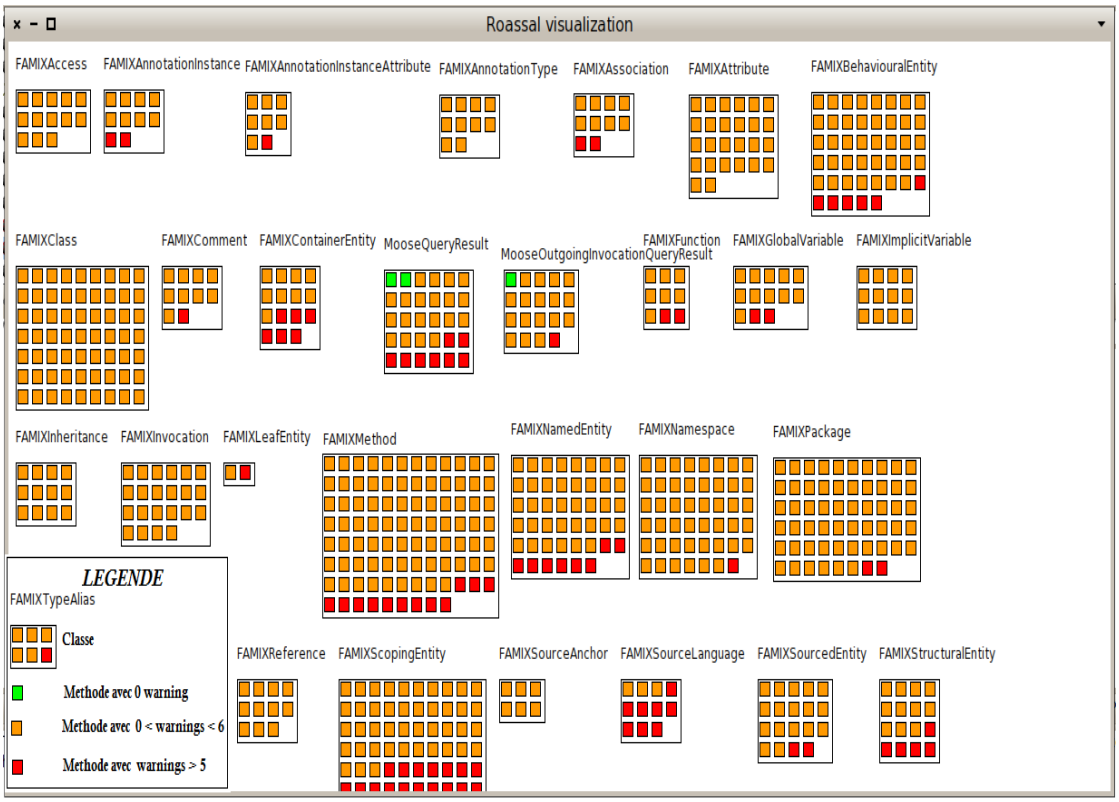

Figure 3. Viewing the Famix-Core Package with SmallLintMaps

This module receives as input the alerts generated by SmallLint, which it integrates into the FAMIX meta model. It can then be used to navigate through the properties of the source code. Among other things, it can be used to answer the following questions:

Which method/class has the highest number of alerts?

What are the properties of such methods/classes?

What rules are broken in a package?

What are the properties of such a package?

Which rule is broken the most?

Case study:

We applied M2SmallLint to the *Famix-Core* package of Pharo 2.0 and obtained the following **Figure 4**.

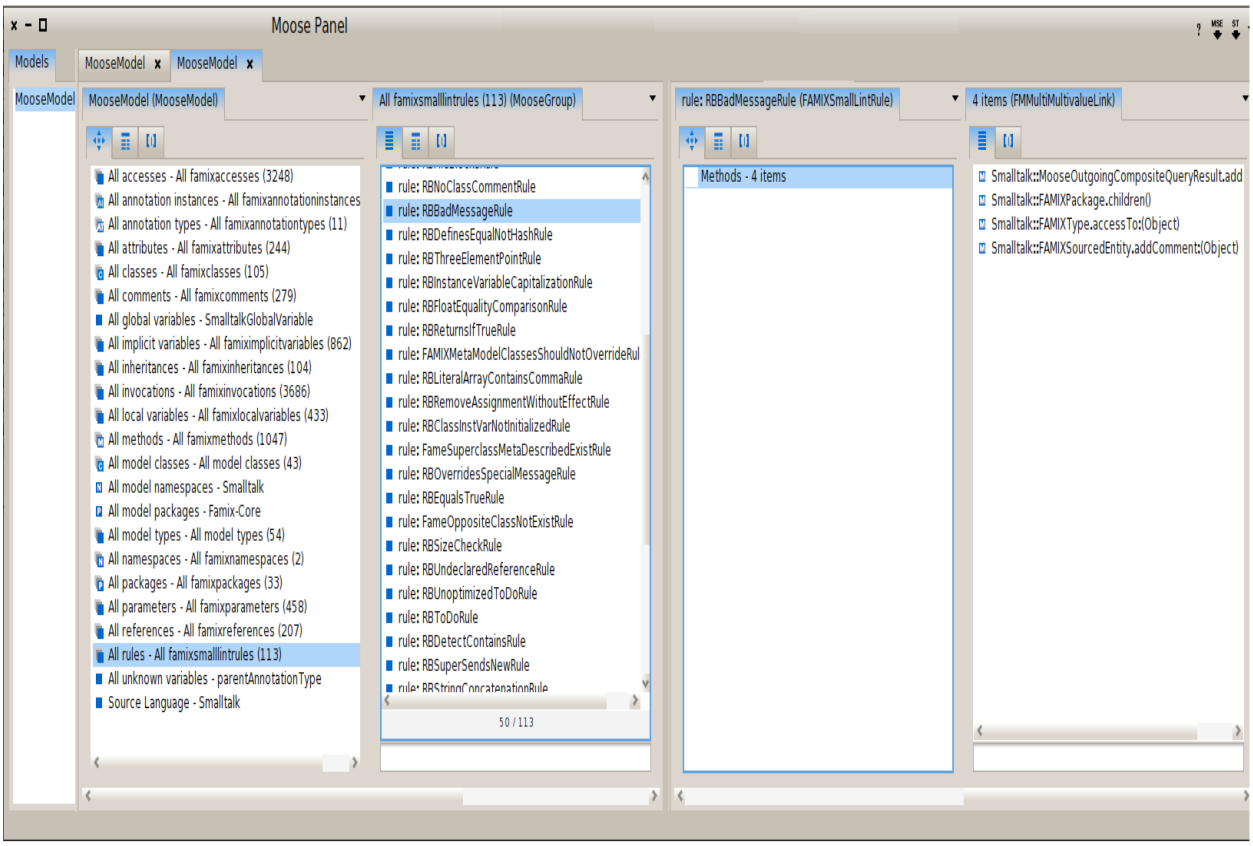

Figure 4. Navigating the rules applied to the Famix-Core Package with MoosePanel

Recommendations for Further Research:

Future studies could explore ensemble methods, combining multiple algorithms to enhance prediction accuracy. Feature engineering and domain-specific insights could also be integrated to refine the model. Additionally, comparisons with other algorithms could provide a more holistic understanding of machine learning's potential in predicting water potability.

4. Conclusion

In this paper, we propose a tool for the visual analysis of programming rule violations and the statistical exploration of source code properties related to these violations. Our objective is to identify a relationship between true positives and source code properties in order to improve the performance of Rule Checkers. Our ultimate goal is to propose a generic method for detecting true positives in alerts generated by Rule Checkers.

References:

[1] J. Sliwerski, T. Zimmermann Et A. Zeller, HATARI: raising risk awareness., *ESEC/SIGSOFT FSE*, pp. 107-110., 2005.

[2] A. Hora, N. Anquetil Et S. Ducasse, Bug maps: A tool for the visual exploration and analysis of bugs., *In : 2012 16th European Conference on Software Maintenance and Reengineering. IEEE.*, pp. 523-526, 2012.

[3] C. Couto, P. Pires, M. T. Valente Et H. N. A. Andre, Bugmaps-granger: A tool for causality analysis between source code metrics and bugs., Brazilian Conference on Software: Theory and Practice (CBSoft'13)., 2013.

[4] F. Bolte Et S. Bruckner, Vis-a-vis: visual exploration of visualization source code evolution., *IEEE Transactions on Visualization and Computer Graphics*, pp. 3153-3167., 2020.

[5] Reddivari, Sandeep Et Khan, Mohammed Salman. VisioTM: A Tool for Visualizing Source Code Based on Topic Modeling., *43rd Annual Computer Software and Applications Conference (COMPSAC) IEEE.* , pp. 932-933., 2019.

[6] J. Maletic, D. Mosora Et C. Newman, MosaiCode: visualizing large scale software: A tool demonstration., *6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT). IEEE*, pp. 1-4, 2011.

[7] H. Kienle Et H. Müller, The Rigi reverse engineering environment., *1nd International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 1).*, 2008.

[8] PharoConsortium, Welcome to Pharo, 2022. [En ligne]. Available: https://pharo.org/. [Accès le 13 juin 2022].

[9] Synectique, Moose, 2022. [En ligne]. Available: https://moosetechnology.org/. [Accès le 04 juillet 2022].

[10] S. Ducasse, N. Anquetil Et M. U. Bhatti, Mse And Famix 3.0: an interexchange format and source code model family,» 2011.

[11] S. R. C. a. C. F. Kemerer, A Metrics Suite for Object, IEEE Transaction on Software Engineering, 2020.

[12] C. a. N. Harrison, Evaluation of the MOOD Set of Object-Oriented Software Metrics, IEEE Transaction on Software, 2020.

[13] P. Mengal, Métriques Et Critères D'évaluation De La Qualité Du Code Source D'un Logiciel, Revue d'un professionnel de l'industrie du logicie, Paris, 2019.

[14] T. Durieux, F. Madeiral, M. Martinez et R. Abreu, Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts., *27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 302-313, 2019.

[15] P. Hegedűs et R. Ferenc, Static code analysis alarms filtering reloaded: A new real-world dataset and its ML-based utilization., *IEEE Access, 10,* , pp. 55090-55101. , 2022.

[16] S. Heckman Et L. Williams, On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques., *Second ACM-IEEE international symposium on Empirical software engineering and measurement* ., pp. 41-50, 2008.

[17] T. Kremenek Et D. Engler, Z-ranking: Using statistical analysis to counter the impact of static analysis approximations.,» chez *International Symposium, SAS.* , San Diego, CA, USA, 2003.