Software Quality Evaluation via Static Analysis and Static Measurement: an Industrial Experience

Luigi Lavazza

Dipartimento di Scienze Teoriche e Applicate
Università degli Studi dell'Insubria
Varese, Italy
Email: luigi.lavazza@uninsubria.it

***Abstract*—Business organizations that outsource software development need to evaluate the quality of the code delivered by suppliers. In this paper, we illustrate an experience in setting up and using a toolset for evaluating code quality for a company that outsources software development. The selected tools perform static code analysis and static measurement, and provide evidence of possible quality issues. To verify whether the issues reported by tools are associated to real problems, code inspections were carried out. The combination of automated analysis and inspections proved effective, in that several types of defects were identified. Based on our findings, the business company was able to learn what are the most frequent and dangerous types of defects that affect the acquired code: currently, this knowledge is being used to perform focused verification activities.**

Keywords–Software quality; Static analysis; Software measurement; Code clones; Code measures.

I. INTRODUCTION

Today, software is necessary for running practically any kind of business. However, poor quality software is generally expensive, because poor quality code can cause expensive failures, increased maintenance cost and security breaches. Hence, organizations that rely on software for running their business need to keep the quality of their software under control.

Many companies do not have the possibility or the will of developing the software they need. Hence, they outsource software development. In this case, an organization has no direct visibility and control of the development process; instead, they can only check the quality of the delivered product.

In this paper, we report about an experience in setting up the toolset needed for evaluating the quality of the code provided by a supplier to an organization. The software involved in the reported activities is used by the organization to run two Business-to-Consumer (B2C) portals.

The organization needed to evaluate the quality of the supplied code; specifically, they wanted to check that the code was correctly structured, cleanly organized, well programmed, and free from defects that could cause failures or could be exploited by attackers. The organization had already in place a testing environment to evaluate the quality of the code from the point of view of behavioral correctness. They wanted to complement test-based verification with checks on the internal qualities that could affect maintainability, fault-proneness and vulnerability.

To accomplish this goal, we selected a set of tools that provide useful indications based on static analysis and measurement of code. The toolset was intended to be used to evaluate two releases of the portal, and then to be set up at the company's premises, and used to evaluate the following releases.

The contributions of the paper are twofold. We provide methodological guidance on the selection and usage of a small set of tools that can provide quite useful insights on code quality. We also provide some results, which can give the reader a measure of the results that can be achieved via the proposed approach.

Because of confidentiality constraints, in this paper we shall not give the names of the involved parties, and we shall omit some non-essential details.

The paper is structured as follows. In Section II, we provide some details concerning the evaluated software and the tools used for the static analysis and measurement. Section III illustrates the methodological approach. In Section IV, the results of the evaluation are described. Section V provides some suggestions about the organization of a software development process that takes advantage of a static analysis and measurement toolset. Section VI illustrates the related work, while Section VII draws some conclusions and outlines future work.

II. THE CONTEXT

In this section, we describe the problem and the tools that were available to be employed in the problem context.

A. The Software to be Evaluated and the Aim of the Study

The evaluation addressed two B2C portals, coded almost entirely in Java. The analyses concentrated exclusively on the Java code. Table I provides a few descriptive statistics concerning the two portals (LLOC is the number of logical lines of code, i.e., the lines that contain executable code).

TABLE I. CHARACTERISTICS OF THE ANALYZED PORTALS.

	Portal 1	Portal 2
Number of files	1507	280
LLOC	100375	37467
LOC	202249	55934
Number of Classes	1158	247
Number of Methods	13351	5370

 ISBN: 978-1-61208-827-3

The aim of the study consisted in evaluating the quality of the products, highlighting weaknesses and improvement opportunities. In this sense, it was important to spot the types of the most frequently recurring issues, rather than finding *all* the actual defects and issues.

It was also required that the toolset could be transferred to the company's premises. To this end, open-source (or free to use) software was to be preferred.

Accordingly, we looked for tools that can

– Detect bad programming practices, based on the identification of specific code patterns.
– Detect bad programming practices, based on code measures (e.g., methods too long, classes excessively coupled, etc.).
– Detect duplicated code.
– Identify vulnerabilities.

After some search and evaluation, we selected the tools mentioned in Table II. These tools are described in some detail in the following sections.

TABLE II. TOOLS USED AND THEIR PURPOSE.

Purpose	Tool	Main features
Identify defects	SpotBugs	Static analysis is used to identify code patterns that are likely associated to defects.
Collect static measures	SourceMeter	Static measurement is applied at different granularity levels (class, method, etc.) to provide a variety of measures.
Detect code clones	SourceMeter	Structurally similar code blocks are identified.
Identify security issues	FindSecBugs	A plug-in for FindBugs, specifically oriented to identifying vulnerable code.

B. Tools Used

1) Static Analysis for Identifying Defects: SpotBugs (formerly known as FindBugs) is a program which uses static analysis to look for bugs in Java code [1][2]. SpotBugs looks for code patterns that are most likely associated to defects. For instance, SpotBugs is able to identify the usage of a reference that is possibly null.

SpotBugs was chosen because it is open-source and one among the best known tools of its kind. Besides, SpotBugs proved to be quite efficient: on a reasonably powerful laptop, it took less than a minute to analyze Portal 1.

SpotBugs provides "confidence" and "rank" for each of the issued warnings. Confidence indicates how much SpotBugs is confident that the issued warning is associated to a real problem; confidence is expressed in a three-level ordinal scale (high, medium, low). The rank indicates how serious the problem is believed to be. The rank ranges from 1 (highest) to 20 (lowest); SpotBugs also indicates levels: "scariest"($1 \leq$rank≤ 4), "scary" ($5 \leq$rank≤ 9), "worrying" ($10 \leq$rank≤ 14), "of concern" ($15 \leq$rank≤ 20).

2) Static Analysis for Identifying Vulnerabilities: Having selected SpotBugs as a static analyzer, it was fairly natural to equip it with FindSecBugs [3], a plug-in for SpotBugs that addresses security problems.

FindSecBugs works like FindBugs and SpotBugs, looking for code patterns that can be associated to security issues, with reference to problems reported by the Open Web Application Security Project (OWASP) [4] an the weaknesses listed in the Common Weaknesses Enumeration (CWE) [5].

3) Static Measures of Code: Several tools are available to measure the most relevant characteristics of code, including size, complexity, coupling, cohesion, etc.

SourceMeter [6] was chosen because it is free to use, efficient and provides many measures, including all the most popular and relevant.

4) Code Clone Detection: Noticeably, SourceMeter is also able to detect code clones. Specifically, SourceMeter is capable of identifying the so-called Type-2 clones, i.e. code fragments that are structurally identical, but may differ in variable names, literals, identifiers, etc.

III. The Method

Since the most interesting properties of code are undecidable, tools that perform static analysis often issue warnings concerning problems that are likely—but not certain—to occur. In practice, the issues reported by static analysis tools can be false positives. Therefore, we always inspected manually the code that had been flagged as possibly incorrect by the tools.

Similar considerations apply to static measures. For instance, consider a method that has unusually high McCabe complexity: only via manual inspection we can check whether the program was badly structured or the coded algorithm is intrinsically complex.

Problem detection was performed as described in Figure 1. The real problems identified via the process described in Figure 1 were classified according to their type, so that the company that asked for the code quality analysis could focus improvement efforts on the most frequent and serious problems.

IV. Results

Here, we describe the code quality problems that were identified.

A. Warnings issued by SpotBugs

Tables III and IV illustrate the number of warnings that SpotBugs issued for the analyzed code, respectively by confidence level and by rank. In Table III, the density indicates the number of warnings per line of code.

TABLE III. SPOTBUGS WARNINGS BY CONFIDENCE.

	Portal 1		Portal 2	
Metric	Warnings	Density	Warnings	Density
High Confidence	68	0.07%	50	0.13%
Medium Confidence	774	0.77%	502	1.34%
Low Confidence	824	0.82%	420	1.12%
Total	1666	1.66%	972	2.59%

SpotBugs also classifies warning by type (for additional information on warning types, see [7]). Table V illustrates the warnings we obtained, by type. It can be noticed that most warnings concerned security (types "Security" and "Malicious code vulnerability").

1) Results deriving from the inspection of SpotBugs warnings: The effort allocated to the project did not allow analyzing all the warnings issued by SpotBugs. Therefore, we inspected the code where SpotBugs had identified issues ranked "scary" and "scariest." Specifically, we analyzed the warnings described in Table VI.

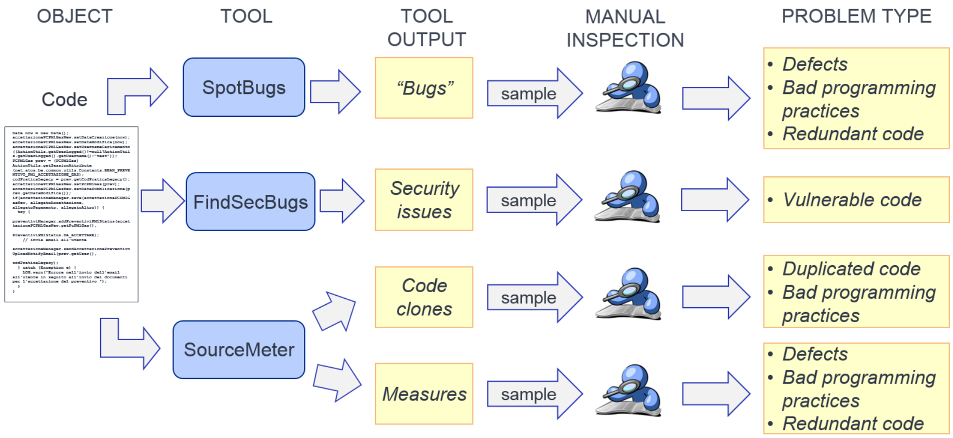

Figure 1. The evaluation process: problem detection phase.

TABLE IV. SPOTBUGS WARNINGS BY RANK

Rank	1	6	7	8	9	10	11	12	14	15	16	17	18	19	20
Portal 1	24	9	1	42		12	39	21	2	604	17	129	562	82	122
Portal 2	10			6	1	7	8	27	18	191	10	59	401	66	168

TABLE V. SPOTBUGS WARNINGS BY TYPE.

	Portal 1		Portal 2	
Warning Type	Number	Percentage	Number	Percentage
Bad practice	73	4.38%	81	8.33%
Correctness	91	5.46%	30	3.09%
Experimental	0	0.00%	1	0.10%
Internationalization	16	0.96%	39	4.01%
Malicious code vulnerability	496	29.77%	316	32.51%
Multithreaded correctness	28	1.68%	1	0.10%
Performance	74	4.44%	143	14.71%
Security	631	37.88%	215	22.12%
Dodgy code	257	15.43%	146	15.02%
Total	1666	100%	972	100%

TABLE VI. SPOTBUGS WARNINGS THAT WERE VERIFIED MANUALLY.

		Occurrences	
Rank	Type	Portal 1	Portal 2
1	Suspicious reference comparison	10	9
1	Call to `equals()` comparing different types	14	1
6	Possible null pointer dereference	8	–
8	Possible null pointer dereference	–	2
8	Method ignores return value	–	4
9	Comparison of `String` objects using == or !=	–	1

Our inspections revealed several code quality problems:

– The existence of problems matching the types of warning issued by SpotBugs was confirmed.

– Some language constructs were not used properly. For instance, class `Boolean` was incorrectly used instead of `boolean`; objects of type `String` were used instead of `boolean` values; etc.

– We found redundant code, i.e., some pieces of code were unnecessarily repeated, even where avoiding code duplication—e.g., via inheritance or even simply by creating methods that could be used in different places—would have been easy and definitely convenient.

– We found some pieces of code that were conceptually incorrect. The types of defect were not of any type that a static analyzer could find, but were quite apparent when inspecting the code.

Concerning the correctness of warnings issued by SpotBugs, we found just one false positive: the "comparison of `String` objects using == or !=" was not an error, in the examined case. We also found that the four instances of "Method ignores return value" were of little practical consequences. In summary, the great majority of warnings indicated real problems, which could cause possibly serious consequences. The remaining warning indicated situations where a better coding discipline could make the code less error prone, if applied systematically.

2) Results deriving from the inspection of FindSecBugs warnings: The great majority of the security warnings (types "Security" and "Malicious code vulnerability") were ranked by FindSecBugs as not very worrying. Specifically, no "scariest") warning was issued, and only one "scary" warning was issued. Therefore, we inspected the only "scary" warning (rank 7, see Table VII), and all the warnings at the highest rank of the level "troubling" (rank 10, see Table VII).

We found that all the warnings pointed to code that had security problems. In many cases, SpotBugs documentation provided quite straightforward ways for correcting the code.

TABLE VII. FINDSECBUGS WARNINGS THAT WERE VERIFIED MANUALLY.

		Occurrences	
Rank	Type	Portal 1	Portal 2
7	HTTP response splitting vulnerability	1	–
10	Cipher with no data integrity	4	2
10	ECB mode is insecure	4	2
10	URL Connection Server-Side Request	1	–
	Forgery and File Disclosure	–	–
10	Unvalidated Redirect	2	–
10	Request Dispatcher File Disclosure	–	1

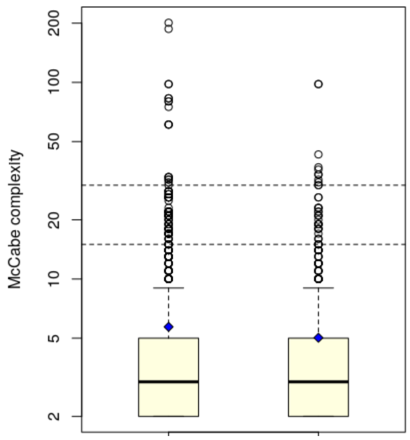

Figure 2. Boxplots illustrating the distributions of McCabe complexity in the two portals (blue diamonds indicate mean values). The scale is logarithmic.

B. Inspection of code elements having measures beyond threshold

Static measures concerning size, complexity, cohesion, coupling, among others, are expected to provide indications on the quality of code. In fact, one expects that code characterized by large size, high complexity, low cohesion, strong coupling and similar "bad" characteristics is error-prone. Accordingly, we inspected code elements having measures definitely out or the usually considered safe ranges. Specifically, we considered McCabe complexity [8], Logical Lines of Code and Response for Class (RFC) [9] as possibly correlated with problems. In fact, we also looked at Coupling Between Objects, Lack of Cohesion in Methods and Weighted Method Count, but these measures turned out to provide no additional information with respect to the aforementioned three measures, i.e., they pointed to the same classes or methods identified as possibly problematic by the aforementioned measures.

We found that several methods featured McCabe complexity well over the threshold that is generally considered safe. Figure 2 shows the distributions of McCabe complexity of methods (excluding setters and getters) together with two thresholds representing values that should and must not be exceeded, according to common knowledge [8][10][11][12]. Specifically, we found methods with McCabe complexity close to 200.

When considering size, we found several classes featuring over 1000 LLOC; the largest class contained slightly less then 6000 LLOC. When considering RFC, we found 12 classes having RFC greater than 200. Interestingly, the class with the highest RFC (709) was also the one containing the method with the greatest McCabe complexity. The biggest class contained the second most complex method. These results were not surprising, since it is known that several measures are correlated to size.

Inspections revealed that the classes and methods featuring excessively high values of LLOC, RFC and McCabe complexity were all affected by the same problem. The considered code had to deal with several types of services, which where very similar under several respects, although each one had its own specificity. The analyzed code ignored the similarities among the services to be managed, so that the code dealing with similar service aspects was duplicated in multiple methods. The code could have been organized differently using basic object-oriented features: a generic class could collect the features that are common to similar services, and a specialized class for every service type could take care of the specificity of different service types.

In conclusion, by inspecting code featuring unusual static measures, we found design problems, namely inheritance and late binding were not used where it was possible and convenient.

C. Inspection of duplicated code

SourceMeter was also used to find duplicated code. Specifically, structurally similar blocks of 10 or more lines of code where looked for. Many duplicated blocks were found. For instance, in Portal 1, 434 duplicated blocks were found. In many cases, blocks included more than one hundred lines. The largest duplicated blocks contained 205 lines. A small minority of detections concerned false positives.

We found three types of duplications:

a) Duplicates within the same file. That is, the same code was found in different parts of the same file (or the same class, often).
b) Duplicates in different files. That is, the same code fragment was found in different files (of the same portal).
c) Duplicates in different portals. That is, the same code fragment was found in files belonging to different portal.

Duplicates of type c) highlighted the existence of versioning problems: different versions of the same class were used in the two portals.

Duplicates of types a) and b) pointed to the same type of problem already identified, i.e., not using inheritance to factor code that can be shared among classes dealing with similar services. Concerning this issue, it is worth noting that static measures revealed a general problem with the design of code, but were not able to indicate precisely which parts of the code could be factorized. On the contrary, duplicated code detection was quite effective in identifying all the cases where code could be factorized, with little need of inspecting the code. In this sense, code clone detection added some value to inspections aiming at understanding the reasons for 'out of range' measures.

V. Suggestions for Improving the Development Process

Given the results described in Section IV, it seems convenient that the capabilities of static analysis and measurement tools are exploited on a regular basis. To this end, we can observe that two not exclusive approaches are possible.

1) Evaluation of code: The toolset can be used to evaluate the released code as described in Section III. However, it would be advisable that developers verify their own code via SpotBugs and SourceMeter even before releasing it: in such a way, a not negligible number of bugs would be removed even before testing and other Verification&Validation activities, thus saving time and effort. With respect to the evaluation described in Section III, where just a sample of the issues reported by the tools were inspected, in the actual development process all issues should be inspected.

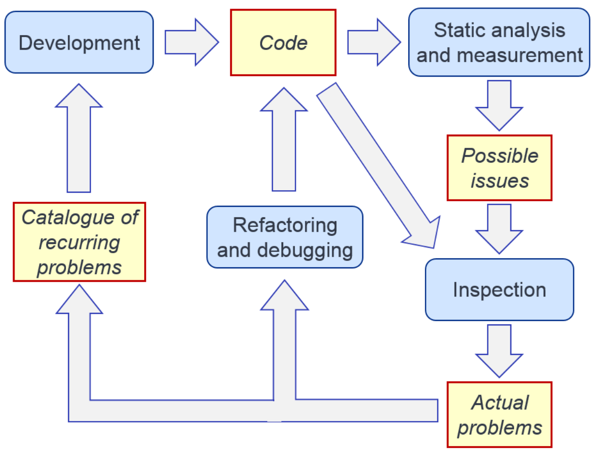

Figure 3. Suggested Development Process.

2) Prevention: The practice of issue identification and verification leads to identifying the most frequently recurring types of problems. It is therefore possible to compile a catalogue of the most frequent and dangerous problems. Accordingly, programmers could be instructed to carefully avoid such issues. This could imply teaching programmers specific techniques and good programming practices.

As a result of the considerations illustrated above, software development activities could be organized as described in Figure 3. If development is outsourced, as in the cases described in this paper, the catalogue of recurrent problems could be used as part of the contract annex that specifies the required code quality level.

Finally, it is worth noting that the proposed approach can be applied in practically any type of lifecycle. For instance, in an agile development environment, the proposed evaluation practices could be applied at the end of every sprint.

VI. Related Work

The effectiveness of using automated static analysis tools for detecting and removing bugs was documented by Zheng et al. [13]. Among other facts, they found that the cost per detected fault is of the same order of magnitude for static analysis tools and inspections, and the defect removal yield of static analysis tools is not significantly different from that of inspections.

Thung et al. performed an empirical study to evaluate to what extent could field defects be detected by FindBugs and similar tools [14]. To this end, FindBugs was applied to three open-source programs (Lucene, Rhino and AspectJ). The study by Thung et al. takes into consideration only known bugs, and is performed on open-source programs. On the contrary, we analyzed programs developed in an industrial context, and relied on manual inspection to identify actual bugs.

Habib and Pradel performed a study to determine how many of all real-world bugs do static bug detectors find [15]. They used three static bug detectors, including SpotBugs, to analyze a version of the Defects4J dataset that consisted of 15 Java projects with 594 known bugs. They found that static bug detectors find a small but non-negligible amount of all bugs.

Vetrò et al. [16] evaluated the accuracy of FindBugs. The code base used for the evaluation consisted of Java projects developed by students in the context of an object-oriented programming course. The code is equipped with acceptance tests written by teachers of the course in such a way that all functionalities are checked. To determine true positives, they used temporal and spatial coincidence: an issue was considered related to a bug when an issues disappeared at the same time as a bug get fixed (according to tests). In a later paper [17] Vetrò et al. repeated the analysis, with a larger code set and performing inspections concerning four types of issues found by FindBugs, namely the types of findings that are considered more reliable.

Tomassi [18] considered 320 Java bugs from the BugSwarm dataset, and determine which of these bugs can potentially be found by SpotBugs and another analyzer—namely, ErrorProne (https://github.com/google/error-prone)—and how many are indeed detected. He found that 40.3% of the bugs were of types that SpotBugs should detect, but only one of such bugs was actually detected by SpotBugs.

In general, the papers mentioned above have goals and use methods that are somewhat similar to ours, but are nonetheless different in important respects. A work that shares context, goals and methods with ours was reported by Steidl et al. [19]. They observed that companies often use static analyses tools, but they do not learn from results, so that they fail to improve code quality. Steidl et al. propose a continuous quality control process that combines measures, manual action, and a close cooperation between quality engineers, developers, and managers. Although there are evident differences between the work by Steidl et al. and the work reported in this paper (for instance, the situation addressed by Steidl et al. does not involve outsourcing), the suggestions for improving the development process given in Section V are conceptually coherent with the proposal by Steidl et al.

Similarly, Wagner et al. [20] performed an evaluation of the effectiveness of static analysis tools in combination with other techniques (including testing and reviews). They observed that a combination of the usage of bug finding tools together with reviews and tests is advisable if the number of false positives is low, as in fact is in the cases we analyzed (many false positives would imply that a relevant effort is wasted).

An alternative to static analyzers like SpotBugs is given by tools that detect the presence of “code smells” [21] in code. A comparison of these types of tools was performed by applying SpotBugs and JDeodorant [22][23] to a set of set of open-source applications [24]. The study showed that the considered tools can help software practitioners detect and remove defects in an effective way, to limit the amount of resources that would otherwise be spent in more cost-intensive activities, such as software inspections. Specifically, SpotBugs appeared to detect defects with good Precision, hence manual inspection of the code flagged defective by SpotBugs becomes cost-effective.

Another empirical study evaluated the persistence of SpotBugs issues in open-source software evolution [25]. This study showed that around half the issues discovered by SpotBugs are actually removed from code. This fact is interpreted as a confirmation that SpotBugs identifies situations that are considered worth correcting by developers.

 ISBN: 978-1-61208-827-3

VII. CONCLUSIONS

Evaluating the quality of software is important in general, and especially for business organization that outsource development, and do not have visibility and control of the development process. Software testing can provide some kind of quality evaluations, but to a limited extent. In fact, some aspects of code quality (e.g., whether the code is organized in a way that favors maintainability) cannot be assessed via testing.

This paper describes an approach to software quality evaluation that consists of two phases: in the first phase, tools are used to identify possible issues in the code; in the second phase, code is manually inspected to verify whether the reported issues are associated to real problems. The tools used are of two kinds: the first performs static analysis of code looking for patterns that are likely associated to problematic code; the second type yields measures of static code properties (like size, complexity, cohesion, coupling etc.), thus helping identifying software elements having excessive, hence probably problematic, characteristics.

The mentioned approach was applied to the code of the web portals used by a European company to let its customers use a set of services. The experience was successful, as tool-driven inspections uncovered several types of defects. In the process, the tools (namely SpotBugs and SourceMeter) identified problems of inherently different nature, hence it is advisable to use both types of tools.

Based on our findings, the business company was able to learn what are the most frequent and dangerous types of defects that affect the acquired code: this knowledge is being used to perform focused verification activities.

The proposed approach and toolset (possibly composed of equivalent tools) can be useful in several contexts where code quality evaluation is needed. Noticeably, the proposed approach can be used in different types of development process, including agile processes.

Among the future possible evolutions of this work, the most intriguing one concerns studying the possibility of replacing inspection via some sort of AI-based models that can discriminate false positives and true problems.

ACKNOWLEDGMENT

This work has been partially supported by the "Fondo di ricerca d'Ateneo" of the Università degli Studi dell'Insubria.

REFERENCES

[1] D. Hovemeyer and W. Pugh, "Finding bugs is easy," ACM Sigplan notices, vol. 39, no. 12, 2004, pp. 92–106.

[2] https://spotbugs.github.io/ [retrieved: August 2020].

[3] https://find-sec-bugs.github.io/ [retrieved: August 2020].

[4] "Open Web Application Security Project (OWASP)," https://www.owasp.org [retrieved: August 2020].

[5] "Common Weaknesses Enumeration," https://cwe.mitre.org [retrieved: August 2020].

[6] R. Ferenc, L. Lang, I. Siket, T. Gyimthy, and T. Bakota, "Source meter sonar qube plug-in," in 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, 2014, pp. 77–82.

[7] https://spotbugs.readthedocs.io/en/stable/bugDescriptions.html/ [retrieved: August 2020].

[8] T. J. McCabe, "A complexity measure," IEEE Transactions on software Engineering, no. 4, 1976, pp. 308–320.

[9] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," IEEE Transactions on software engineering, vol. 20, no. 6, 1994, pp. 476–493.

[10] A. H. Watson, D. R. Wallace, and T. J. McCabe, Structured testing: A testing methodology using the cyclomatic complexity metric. US Department of Commerce, Technology Administration, National Institute of , 1996, vol. 500, no. 235.

[11] M. Bray, K. Brune, D. A. Fisher, J. Foreman, and M. Gerken, "C4 software technology reference guide-a prototype." Carnegie-Mellon Univ., Pittsburgh PA, Software Engineering Inst, Tech. Rep., 1997.

[12] JPL, "JPL Institutional Coding Standard for the C Programming Language," Jet Propulsion Laboratory, Tech. Rep., 2009.

[13] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," IEEE transactions on software engineering, vol. 32, no. 4, 2006, pp. 240–253.

[14] F. Thung, L. Lucia, D. Lo, L. Jiang, P. Devanbu, and F. Rahman, "To what extent could we detect field defects? an extended empirical study of false negatives in static bug-finding tools," Automated Software Engineering, vol. 22, no. 4, 2015, pp. 561–602.

[15] A. Habib and M. Pradel, "How many of all bugs do we find? a study of static bug detectors," in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018, pp. 317–328.

[16] A. Vetrò, M. Torchiano, and M. Morisio, "Assessing the precision of FindBugs by mining java projects developed at a university," in 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). IEEE, 2010, pp. 110–113.

[17] A. Vetrò, M. Morisio, and M. Torchiano, "An empirical validation of findbugs issues related to defects," in 15th Annual Conference on Evaluation and Assessment in Software Engineering (EASE 2011). IET, 2011, pp. 144–153.

[18] D. A. Tomassi, "Bugs in the wild: examining the effectiveness of static analyzers at finding real-world bugs," in Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 980–982.

[19] D. Steidl, F. Deissenboeck, M. Poehlmann, R. Heinke, and B. Uhink-Mergenthaler, "Continuous software quality control in practice," in 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, 2014, pp. 561–564.

[20] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger, "Comparing bug finding tools with reviews and tests," in IFIP International Conference on Testing of Communicating Systems. Springer, 2005, pp. 40–55.

[21] M. Fowler, Refactoring: improving the design of existing code. Addison-Wesley Professional, 2018.

[22] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of feature envy bad smells," in 2007 IEEE International Conference on Software Maintenance. IEEE, 2007, pp. 519–520.

[23] "JDeodorant website," 2020. [Online]. Available: https://github.com/tsantalis/JDeodorant

[24] L. Lavazza, S. Morasca, and D. Tosi, "Comparing static analysis and code smells as defect predictors: an empirical study," in Empirical Software Engineering and Measurement – ESEM 2020, 2020.

[25] L. Lavazza, D. Tosi, and S. Morasca, "An empirical study on the persistence of spotbugs issues in open-source software evolution," in 13th International Conference on the Quality of Information and Communications Technology – QUATIC, 2020.