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>
<thead>
<tr>
<th>Portal 1</th>
<th>Portal 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of files</td>
<td>1507</td>
</tr>
<tr>
<td>LLOC</td>
<td>100375</td>
</tr>
<tr>
<td>LOC</td>
<td>202249</td>
</tr>
<tr>
<td>Number of Classes</td>
<td>1158</td>
</tr>
<tr>
<td>Number of Methods</td>
<td>13351</td>
</tr>
</tbody>
</table>

TABLE I. CHARACTERISTICS OF THE ANALYZED PORTALS.
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>
<thead>
<tr>
<th>Purpose</th>
<th>Tool</th>
<th>Main features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identify defects</td>
<td>SpotBugs</td>
<td>Static analysis is used to identify code patterns that are likely associated to defects.</td>
</tr>
<tr>
<td>Collect static measures</td>
<td>SourceMeter</td>
<td>Static measurement is applied at different granularity levels (class, method, etc.) to provide a variety of measures.</td>
</tr>
<tr>
<td>Detect code clones</td>
<td>SourceMeter</td>
<td>Structurally similar code blocks are identified.</td>
</tr>
<tr>
<td>Identify security issues</td>
<td>FindSecBugs</td>
<td>A plug-in for FindBugs, specifically oriented to identifying vulnerable code.</td>
</tr>
</tbody>
</table>

### 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>
<thead>
<tr>
<th>Metric</th>
<th>Portal 1 Warnings</th>
<th>Portal 1 Density</th>
<th>Portal 2 Warnings</th>
<th>Portal 2 Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>High Confidence</td>
<td>88</td>
<td>0.07%</td>
<td>30</td>
<td>0.13%</td>
</tr>
<tr>
<td>Medium Confidence</td>
<td>774</td>
<td>0.77%</td>
<td>502</td>
<td>1.34%</td>
</tr>
<tr>
<td>Low Confidence</td>
<td>824</td>
<td>0.82%</td>
<td>420</td>
<td>1.12%</td>
</tr>
<tr>
<td>Total</td>
<td>1666</td>
<td>1.60%</td>
<td>972</td>
<td>2.59%</td>
</tr>
</tbody>
</table>

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.*
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 IV. SpotBugs warnings by rank

<table>
<thead>
<tr>
<th>Rank</th>
<th>Portal 1</th>
<th>Portal 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>42</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>39</td>
<td>27</td>
</tr>
<tr>
<td>11</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>14</td>
<td>604</td>
<td>17</td>
</tr>
<tr>
<td>15</td>
<td>129</td>
<td>59</td>
</tr>
<tr>
<td>16</td>
<td>562</td>
<td>401</td>
</tr>
<tr>
<td>17</td>
<td>82</td>
<td>66</td>
</tr>
<tr>
<td>18</td>
<td>122</td>
<td>108</td>
</tr>
</tbody>
</table>

### Table V. SpotBugs warnings by type

<table>
<thead>
<tr>
<th>Warning Type</th>
<th>Portal 1</th>
<th>Portal 2</th>
<th>Number</th>
<th>Percentage</th>
<th>Number</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bad practice</td>
<td>73</td>
<td>81</td>
<td>4.38%</td>
<td>8.33%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Correctness</td>
<td>91</td>
<td>30</td>
<td>5.46%</td>
<td>3.09%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Experimental</td>
<td>0</td>
<td>1</td>
<td>0.00%</td>
<td>0.10%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internationalization</td>
<td>16</td>
<td>39</td>
<td>0.96%</td>
<td>4.01%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Malicious code vulnerability</td>
<td>406</td>
<td>316</td>
<td>29.77%</td>
<td>32.51%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multithreaded correctness</td>
<td>28</td>
<td>1</td>
<td>1.68%</td>
<td>0.10%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Performance</td>
<td>74</td>
<td>143</td>
<td>4.44%</td>
<td>14.71%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Security</td>
<td>631</td>
<td>215</td>
<td>37.88%</td>
<td>22.12%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dodgy code</td>
<td>257</td>
<td>146</td>
<td>15.43%</td>
<td>15.02%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Total                                       | 1666     | 972      | 100%   | 100%       |        |            |

### Table VI. SpotBugs warnings that were verified manually.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Type</th>
<th>Portal 1</th>
<th>Portal 2</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Suspicious reference comparison</td>
<td>10</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Call to equals() comparing different types</td>
<td>14</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Possible null pointer dereference</td>
<td>8</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>Possible null pointer dereference</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Comparison of String objects using == or !=</td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

---

### Table VII. FindSecBugs warnings that were verified manually.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Type</th>
<th>Portal 1</th>
<th>Portal 2</th>
<th>Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>HTTP response splitting vulnerability</td>
<td>1</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>Cipher with no data integrity</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>ECB mode is insecure</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>URL Connection Server-Side Request</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Forgery and File Disclosure</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Unvalidated Redirect</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Request Dispatcher File Disclosure</td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
the services to be managed, so that the code dealing with
specificity. The analyzed code ignored the similarities among
similar under several respects, although each one had its own
had to deal with several types of services, which were very
excessively high values of LLOC, RFC and McCabe complex-
correlated to size.

class contained the second most complex method. These results
method with the greatest McCabe complexity. The biggest
with the highest RFC (709) was also the one containing the
classes having RFC greater than 200. Interestingly, the class
then 6000 LLOC. When considering RFC, we found 12
over 1000 LLOC; the largest class contained slightly less
thresholds representing values that should and must not be
of methods (excluding setters and getters) together with two
safe. Figure 2 shows the distributions of McCabe complexity
McCabe complexity [8], Logical Lines of Code and Response
measures turned out to provide no additional information
of Cohesion in Methods and Weighted Method Count, but
these measures revealed a general problem with the design
of code, but were not able to indicate precisely which parts
static measures revealed a general problem with the design
factor code that can be shared among classes dealing with
factorization of many problem already identified, i.e., not using inheritance and
late binding were not used where it was possible and conve-
nient.

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 complex-
ity 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 complex-
ity were all affected by the same problem. The considered code
had to deal with several types of services, which were 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 conve-
nient.

C. Inspection of duplicated code

SourceMeter was also used to find duplicated code. Specif-
ically, 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 version-
ing 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 conve-
nient 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.
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 recurring 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. [15]. 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.
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