Uncovering Bad Practices in Junior Developer Projects Using Static Analysis and Formal Concept Analysis

Simona Motogna[a], Diana Cristea[b], Diana-Florina Şotropa[c] and Arthur-Jozsef Molnar[d]
Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania

Keywords: Static Code Analysis, Formal Concept Analysis, Bad Practices, SonarQube.

Abstract: Static code analysis tools have been widely used as a resource for early error detection in software development. This paper explores the use of SonarQube together with Formal Concept Analysis, used for detecting data clusters, in enhancing source code quality among junior developers by facilitating the early detection of various quality issues and revealing dependencies among detected issues. We analyze the distribution of bad-practice issues in junior developers' projects and show where the main problems occur, as well as the associations of bad practice issues with other types of issues. We conclude the analysis with a comparison between Python and Java projects with respect to the mentioned aspects. While focusing the analysis on issues related to bad practices in both Java and Python projects, the paper aims to to uncover challenges faced by junior developers in Java and Python projects, promoting awareness of code quality.

1 INTRODUCTION

As the complexity and size of software systems constantly increase, their quality must be an essential objective through entire life cycle. Quality of the produced source code should be a primary concern to all developers, junior and senior all together. Observing and assessing coding guidelines and best practices in junior developers' projects can be a tedious and time consuming task.

This study shows how static analysis tools can be successfully used to assess quality aspects in junior developers' projects. The main features of such tools allow early automatic detection of several types of quality issues, allowing in depth analysis of bad practices patterns. Code review tools based on static analysis have been widely adopted by industry, such as SonarQube which is used in this study, and proved to be a useful resource to improve quality (McConnell, 2004; Avgeriou and o., 2021).

Data generated by these tools can be further subject to analysis to detect programmers behavior, common mistakes or programming concepts misunderstanding. We have chosen to use Formal Concept Analysis (FCA) for data analysis given previous applications of this method to similar educational problems (Duquenne, 2007; Priss, 2013; Priss, 2020; Cristea et al., 2021).

Both Java and Python are programming languages intensively used in software development in industry. Python became more and more popular in the last years due to its versatility, efficiency and strong support for AI/ML application development, so it is desirable to also have specific quality gates.

The goal of our study is to investigate the results of static analysis related to bad practices. For this purpose, we designed and carried a case study on student projects, developed in Java and Python, and the findings can be used to detect dependencies between the issues detected by static analysis tools, respectively to observe common mistakes in programming behavior.

The methodology that we apply in our study combines quantitative observations with the strengths of FCA in order to produce a more precise understanding of the dependencies at code level.

[a] https://orcid.org/0000-0002-8208-6949
[b] https://orcid.org/0000-0003-1440-3786
[c] https://orcid.org/0000-0003-4403-9946
[d] https://orcid.org/0000-0002-4113-2953

2 THEORETICAL BACKGROUND

Static Analysis Tools. The existing static analysis tools have undergone a significant transformation by using the abstract syntax tree (AST) to perform source code analysis and compute several metrics. This has a

Motogna, S., Cristea, D., Şotropa, D. and Molnar, A.
Uncovering Bad Practices in Junior Developer Projects Using Static Analysis and Formal Concept Analysis.
DOI: 10.5220/0012739500003687
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2024)*, pages 752-759
ISBN: 978-989-758-696-5; ISSN: 2184-4895
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

Table 1: SonarQube rules tagged as bad practice for Java in our extended quality profile, as described in (SonarSource, 2024) (note that Java rule id's are prefixed with *java:* (e.g., java:S1215)).

Rule Id	**Description**	**Type**	**Severity**
S1215	Execution of the Garbage Collector should be triggered only by the JVM	Code Smell	Critical
S2077	Formatting SQL queries is security-sensitive	Security Hotspot	Major
S5976	Similar tests should be grouped in a single Parameterized test	Code Smell	Major
S2925	"Thread.sleep" should not be used in tests	Code Smell	Major
S1607	JUnit4 @Ignored and JUnit5 @Disabled should be used to disable tests and should provide a rationale	Code Smell	Major
S1181	Throwable and Error should not be caught	Code Smell	Major
S1161	"@Override" should be used on overriding and implementing methods	Code Smell	Major
S1214	Constants should not be defined in interfaces	Code Smell	Critical
S1123	Deprecated elements should have both the annotation and the Javadoc tag	Code Smell	Major
S106	Standard outputs should not be used directly to log anything	Code Smell	Major
S4838	An iteration on a Collection should be performed on the type handled by the Collection	Code Smell	Minor
S3066	"enum" fields should not be publicly mutable	Code Smell	Minor
S1319	Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList"	Code Smell	Minor
S1301	"switch" statements should have at least 3 "case" clauses	Code Smell	Minor
S1199	Nested code blocks should not be used	Code Smell	Minor
S1132	Strings literals should be placed on the left side when checking for equality	Code Smell	Minor

Table 2: SonarQube rules tagged as bad practice for Python in our extended quality profile, as described in (SonarSource, 2024) (note that Python rule id's are prefixed with *python:* (e.g., python:S5754)).

Rule Id	**Description**	**Type**	**Severity**
S5754	"SystemExit" should be re-raised	Code Smell	Critical
S5712	Some special methods should return "NotImplemented" instead of raising "NotImplementedError"	Code Smell	Critical
S2077	Formatting SQL queries is security-sensitive	Security Hotspot	Major
S5806	Builtins should not be shadowed by local variables	Code Smell	Major
S5706	Special method "__exit__" should not re-raise the provided exception	Code Smell	Major
S1607	A reason should be provided when skipping a test	Code Smell	Major

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

direct impact on the entire development time and also on the time to fix issues in code (Boehm and Papaccio, 1988; Boehm and Basili, 2001).

SonarQube is one of the most popular and widely used static analysis tools, providing continuous inspection in order to deliver clean code. One of the reasons for its extensive use is the support offered for more than 30 programming languages, and integration with different IDEs. As a functioning principle, once the code is transformed into an AST, analysis rules are applied to the tree structure. These rules encompass a set of predefined patterns, best practices, and coding standards that are used to identify potential issues classified in: code smells, bugs, vulnerabilities or security hotspots.

SonarQube also assigns a severity level to each issue: Info, Minor, Major, Critical, and Blocker. Blocker and Critical detrimentally influence the system. Blocker issues should be handled with higher priority and considered to have a greater impact. Major issues can considerably affect a developer's productivity, while Minor and Info are considered as having a reduced impact.

In this study we focus on rules tagged as "bad practice", which, according to SonarQube documentation (SonarSource, 2024), means that they incorporate a bad design decision. This category is important, especially for junior developers, since it establishes some guidelines for making the code easier to understand, update, and maintain over time.

Table 1 comprises the rules tagged with "bad practice" defined for Java language, while Table 2 those for Python, together with their category and severity level.

Formal Concept Analysis. provides a theoretical model for clustering data based on lattice theory (Ganter and Wille, 1999) . The fundamental structures of FCA are the *formal context*, i.e. a dataset containing objects, attributes and a relation between them, and *formal concepts*, i.e. maximal clusters of objects having certain attributes. The *concept lattice* provides a visual representation of all concepts of a context. In the triadic case, FCA additionally contains conditions as a third dimension (Lehmann and Wille, 1995). The triadic context does not always have a visual lattice representation. However, when projecting on one of the dimensions the data can be visualized as a dyadic lattice. The projection can then be changed

in order to continue the data exploration ((Rudolph et al., 2015b)).
Answer Set Programming for FCA. It is known that FCA has some scalability issues regarding computing concepts in larger contexts ((Khaund et al., 2023), (Slezak, 2012)). Most tools cannot compute the concepts of a very large dataset or generate the corresponding lattice. In order to deal with this scalability issue, we use Answer Set Programming (ASP) (Gebser et al., 2012), which is a declarative approach to solve NP-hard problems in a time efficient manner. For this purpose we use the ASP encoding for FCA (Rudolph et al., 2015a) and the Potsdam answer set solving collection, Potassco (Potassco, 2024).

3 PROGRAMMING PRACTICES ASSESSMENT

Our main objective is to *analyze bad practice issues from the perspective of junior developers in the context of Python and Java software projects.* We further divide the main objective in the following research questions:
RQ1. Which is the distribution of bad practice issues in the source code?
RQ2. Are bad practice issues associated with other issues?
RQ3. Is there any similarity between Python and Java in terms of issues tagged with bad practice?

In RQ1 we focus on the bad practice issues in terms of occurrence and severity, and we assess which issues labeled as bad practice are more frequent and how often they occur. Then in RQ2 we aim to identify and understand the dependencies with other issues detected by SonarQube, while in RQ3 we compare bad practice issues from Java and Python projects.

3.1 Data Collection and Processing

Our case study covers three iterations, namely fall semester of 2020, 2021 and 2022, of the *Formal Languages and Compiler Design* mandatory course, taken by last year undergratuate students of the Computer Science program at the Babeş-Bolyai University. We assimilate them as being junior developers, since they have accumulated knowledge in different programming languages and IDEs, and have spent at least six weeks at internships in software companies.

As part of laboratory activities, students had to implement two programming assignments (medium size projects, developed over two to four weeks), using a language of their choice. Requirements were exactly the same for all students implementing the same assignment. Since most students used either Python or Java, we focus our case study on the assignments submitted in these languages.

We configured SonarQube 9.9.0 to employ an extended quality profile by activating all static analysis rules for both Java and Python, with the exception of rules that enforce coding standards (e.g., location of curly braces in Java), the use of annotations or those related to tests (e.g., test method discoverability in Python). This resulted in a Sonar profile having 588 rules for Java and 205 rules for Python.

The static analysis resulted in 25,665 issues, of which 2,990 are tagged as *bad-practice* by SonarQube. The vast majority of them (2,883 issues) were found in Java code, with only 107 issues targeting Python. All issues were categorized as code smells and thus belong to the maintainability domain. With regards to severity, there were 22 critical issues, 2,131 major issues and 837 minor issues.

3.2 FCA Applied

When interested in an overview of correlations among issues in the whole dataset, the visual representation is not essential, hence using the approach described in section 2 we compute the concepts using ASP.

For an in depth analysis we have an approach that supports data exploration and is focused on particular correlations among issues. In this case, both a visual lattice representation and a triadic FCA approach can be useful. Therefore, we use FCA Tools Bundle (Kis et al., 2016), which, to the best of our knowledge, is the only tool that includes the triadic navigation paradigm described in section 2 and also uses ASP for efficiently computing the concepts. For this approach, we generate a triadic context, where dimensions are represented by objects, i.e. a collection of projects with the same requirements denoted *2020-L1*, *2021-L1*, *2022-L1*, *2020-L3*, *2021-L3*, *2022-L3*, attributes, i.e. classes from individual projects denoted *p1-c1*, *p1-c2*, etc., *p2-c1*, *p2-c2*, etc. (where *p1*, *p2*, etc. are the projects of each collection), and conditions, i.e. issues. While the analysis is on the whole dataset, each dyadic projection shows a subset of the data. In Figure 1 we can see the lattice obtained by projecting on the collection of projects *2020-L1*.

For a better readability of the lattice we use the sparse representation where each label is represented only once and we use a functionality of FCA Tools Bundle that allows hiding a type of labels. In Figure 1 we hide the object labels represented by classes of individual projects, since the names of the classes are not relevant to the analysis. When reading the

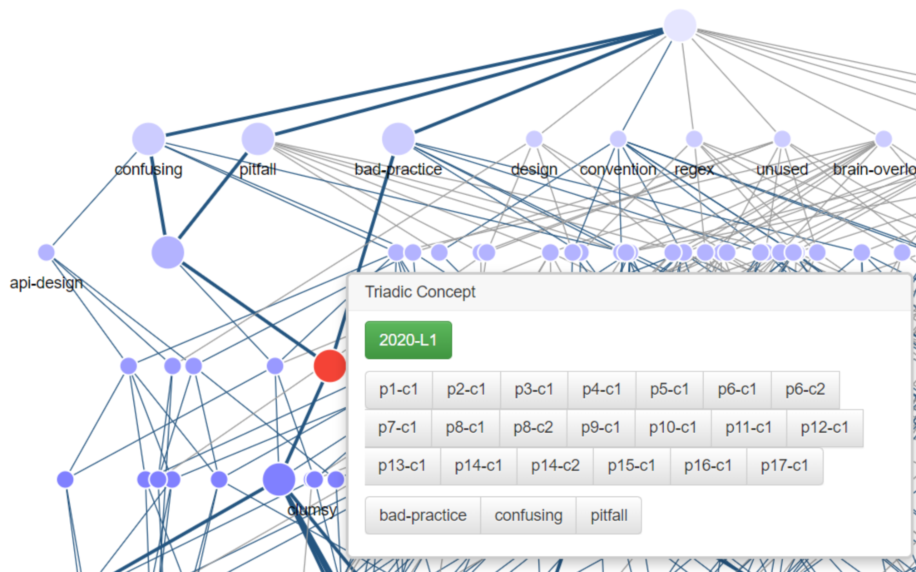

Figure 1: Fragment of dyadic lattice with classes of individual projects as objects and issues as attributes, obtained by projecting on the collection of projects 2020-L1.

Table 3: SonarQube rules generating bad-practice issues in Java projects.

Rule Id	Severity	Total number of occurrences	No. of classes in which the issues occurred/ Total no. of classes with issues
S106	major	2044	307 / 6606
S1132	minor	613	150 / 6606
S1319	minor	212	98 / 6606
S1199	minor	9	3 / 6606
S1301	minor	3	3 / 6606
S1161	major	2	1 / 6606

Table 4: SonarQube rules generating bad-practice issues in Python projects.

Rule Id	Severity	Total number of occurrences	No. of classes in which the issues occurred/ Total no. of classes with issues
S5806	major	85	77 / 1023
S5754	critical	22	18 / 1023

elements of a concept in the sparse lattice representation one must follow the lattice lines up and down as described in the following. The concept marked with red in the zoomed in Figure 1 has one dimension represented by the projection, {*2020-L1*}, the next dimension contains all the objects from the lattice reachable when going downward, {*p1-c1, p2-c1, p3-c1, etc.*}, and the third dimension contains all the attributes reachable when going upward in the lattice, {*confusing, pitfall, bad-practice*}.

3.3 Results

We analyze the source code in relation with Sonar rules that generate bad-practice issues and, based on our results, we provide answers to the formulated research questions.

RQ1: Which is the distribution of bad practice issues in the source code? We aggregated all the results of running SonarQube on individual source code, and then summarize the findings: bad-practice occurs in 122 out of the 126 Java projects, with a total of 2,883 occurrences, as shown in Table 3. In case of Python, bad-practice occurs in 75 out of the 228 Python projects, with a total of 107 occurrences, as shown in Table 4.

Bad Practice Issues in Java: As the Table 3 shows, there is an overwhelming appearance of rule S106 in the dataset. Rule S106 refers to "Standard outputs should not be used directly to log anything", and according to SonarQube documentation, it is generated by the fact that the program writes directly to the standard output, instead of abstracting away the output behind a method. Using standard output does not comply to logging requirements, such as to easily retrieve

the logs, to record the data or to log securely in case of sensitive data.

Two rules with minor severity that are worth mentioning are: i) S1132 that addresses equality method calls, recommending to replace foo.equals(”bar”) with ”bar”.equals(foo), in an attempt to prevent raising null pointer exceptions; and ii) S1319 that signals cases when a collection implementation class from java.util.* is used in different typing context of a public method, and mostly related to performance.

Starting with version 8.6, introduced in September 2020, SonarQube also provides information about the clean code attributes, as defined by (Martin, 2008). Regarding the distribution of concerns for the identified bad practice issues in Java, all of the mentioned issues have an impact on maintainability, while as clean code attributes affected, they include: adaptability (S106), intentionality (S1132), respectively consistency (S1319).

Remark 1: *Regarding Distribution of Bad Practices Issues in Java*, junior developers seem to produce clean code, with the exception of one aspect related to handling of output operations. This has a relative small impact to maintainability. Only three out of 16 rules are not respected on a meaningful basis, two of them being minor, and only one major. One important remark concerning the diffuseness of rule *java:S106* is that it appears in the top 10 most violated rules in (Baldassarre et al., 2020), in a study that considers a similar context, being the only rule in the bad practice category.

Bad Practice Issues in Python: The results of our analysis are summarized in Table 4. In terms of severity, the rule *python:S5754*, namely ”SystemExit” should be re-raised, has a meaningful appearance in the experiments. Another rule that appears quite often, of major severity, is *python:S5806*, which generates an issue when a local variable name matches the builtin name, thus builtin name becoming locally inaccessible. This might be error-prone.

Discussing the clean code attributes in relation to these bad practice issues in Python, they are all related to intentionality, defined as code being ”clear, logical, complete, and efficient”. They affect maintainability, at a high level.

Remark 2: *Regarding Distribution of Bad Practices Issues in Python*, junior developers exhibit problems when dealing with output operations. Two out of eight rules are broken, but the severity of these rules are of concern, being critical and major.

RQ2: Are Bad Practice Issues Associated with other issues? In order to answer this research question, we explore the lattices and the concepts created through FCA as explained in section 3.2. The tool we used, allowed us to detect the association of tag ”bad practice” with other tags, namely other types of issues. Another aspect that we were interested in was that the associations are relevant in terms of appearance, so we introduced a threshold of 3%.

Associations of Bad Practice Issues in Java: some remarks can be drawn from the results in Table 5. Firstly, the occurrences are below 5%, which considering the target group of junior developers is a good sign. The next aspect that we notice is that bad practice issues are mostly associated with issues tagged as ”cwe” and ”owasp” which correspond to security vulnerabilities. This was inferred from a similar situation as the one presented in Figure 1, involving 307 classes with attributes $\{$*bad-practice, cwe, owasp-a3*$\}$.

Also, ”cert”, linked to community developed standard CERT, is quite related to bad practice issue. Other associations that have a mentionable occurrence refer to: brain-overload (signaling high complexity), pitfall (notifying possible future errors) and error-handling (several code smell and bug issues related to handling errors in code).

Remark 3: *Regarding association of bad practices issues in Java*, although they have a lower occurrence ratio, some concerns are raised regarding frequent association between bad practice and security issues. Even if bad practice issues are mostly code smells (see Table 1) their association with security vulnerabilities, high complexity and standard coding practices highlights their importance.

Associations of Bad Practice Issues in Python: Table 6 summarizes the lattice exploration for Python projects, in which we also applied a 3% threshold for meaningful results. The bad practice issues and associations with other issues are, in our opinion, in some cases quite high, between 5 to 9.19%. Regarding the different types of associated issues we notice: ”pitfall” (high possibility to generate future error), ”confusing” (denoting code comprehension difficulty, and in consequence higher maintainability costs) and ”brain-overload” (high complexity indicator). We allocate a lower importance to the ”convention” tag, as it refers to coding conventions.

Remark 4: *Regarding association of bad practices issues in Python*, the projects corresponding to junior developers exhibit a significant link to issues related to high maintainability costs: pitfall, confusing, brain-overload. 4.50% of the projects expose association of bad practice tags with all three other types of issues.

RQ3: Is there any similarity between Python and Java in terms of issues tagged with bad practice? Our dataset consists of two sets of projects with the same requirements. Based on the fact that the stu-

Table 5: Frequent issues associated with bad-practice in Java projects (Number, respective percentage of classes in which the issues occurred / Total no. of classes with issues).

No.	Percentage of classes	Tag(s)
308 / 6606	4.66%	bad-practice
307 / 6606	4.65%	bad-practice, cert, owasp-a3
225 / 6606	3.41%	bad-practice, cert, owasp-a3, cwe
216 / 6606	3.27%	bad-practice, brain-overload
215 / 6606	3.25%	bad-practice, cert, owasp-a3, brain-overload
209 / 6606	3.16%	bad-practice, pitfall
208 / 6606	3.15%	bad-practice, pitfall, cert, owasp-a3
207 / 6606	3.13%	bad-practice, error-handling, cert,owasp-a3

Table 6: Frequent issues associated with bad-practice in Python projects (Number, respective percentage of classes in which the issues occurred / Total no. of classes with issues).

No.	Percentage	Tag(s)
94 / 1023	9.19%	bad-practice
78 / 1023	7.62%	bad-practice, pitfall
77 / 1023	7.53%	bad-practice, confusing,pitfall
58 / 1023	5.67%	bad-practice, brain-overload
47 / 1023	4.59%	bad-practice, brain-overload,pitfall
46 / 1023	4.50%	bad-practice, brain-overload,confusing,pitfall
33 / 1023	3.26%	bad-practice, convention

dents had to implement the same requirements, thus using the same algorithms, it allows us to explore if the type of issues that they introduce using Java and Python are related in any way.

Our first observations is that the imbalance between the number of Java and Python rules in our extended Sonar quality profile (588 and 205, respectively) is maintained after filtering for rules covering bad practices. As shown in Tables 1 and 2, our extended ruleset includes 16 rules for Java and 6 rules for Python covering bad practices. The only common rule is S2077, which refers to formatting SQL queries and which did not generate any issues in our dataset. While some of the rules are platform-specific (e.g., *java:S1181*, *java:1319*), platform-agnostic rules such as *java:S1215* or *java:S1199* do not have a Python equivalent, even when considering rules that are not tagged as *bad-practice*.

Tables 3 and 4 illustrate the prevalence of issues generated by the considered rules. We observe that only a small number of the considered rules generate issues; furthermore, most issues are generated by a small number of rules, which is in line with the findings in previous research. (Walkinshaw and Minku, 2018) found that most defects were indeed reported in a limited number of files, while the case study carried out in (Molnar and Motogna, 2020b) showed that in the case of open-source software analyzed using SonarQube, around 80% of reported issues were grouped under a third of all tags.

In our case, we find the rules generating the majority of issues across Java and Python projects are not comparable. We find this to be the result of platform differences combined with the different level of support these languages have in SonarQube itself.

Remark 5: *Regarding the similarity of bad-practice issues between the studied languages*, we believe that the sound takeaway from a teaching perspective is to combine the detected issues and adapt existing teaching techniques in order to mitigate detected bad practices. While *python:S5806* highlights an already well-known bad practice, we can assimilate *python:S5754* with *java:S1181* and highlight the platform-dependent correct way of handling special cases of exceptions and program halting.

3.4 Discussion

Understanding and adhering to clean code principles and best practices, particularly among junior developers, are essential for producing high-quality software with improved maintainability and reduced technical debt. Thus, we consider that investigating bad practices issues in junior developers code can have a significant impact on the way they will evolve and learn.

The findings of the case study reveals that bad practices vary dependending on the language (in this case Java, respectively Python), with a reasonable frequency but they can be associated with other issues in code. As remarks 3 and 4 has pointed out, we suggest that classes or modules that expose several code issues at the same time should be candidates for a code review performed by an experienced developer.

We consider our study to be an experiential report with impact in education. We encourage the adoption of static analysis tools, such as SonarQube in soft-

ware engineering discipline, such that students can be aware of the importance of the quality of the code they produce.

The practitioners community can also benefit from our findings, both in adopting SonarQube or similar tools to inspect and manage code quality from early stages in the development, but also to pay more attention to code fragments that expose several quality issues simultaneously.

From a research perspective, we show the strengths of FCA in investigating software quality issues, based on its capabilities to observe relations and dependencies among attributes and offering several perspectives (dyadic and triadic) about the data.

4 RELATED WORK

Technical Debt (TD) has raised the interest of the research community for over 10 years with a large number of significant contributions to the domain as a systematic literature review records (Murillo et al., 2023). As SonarQube is one of the most used tools in industry to detect and manage technical debt, there are studies that focus on investigating the technical debt as handled by this tool (Molnar and Motogna, 2020a; Molnar and Motogna, 2020a; Lenarduzzi et al., 2020; Baldassarre et al., 2020; Lenarduzzi et al., 2020b),

Similar approaches in which the scope of the research study is to investigate how junior developers deal with technical debt are (Lenarduzzi et al., 2020a), (Baldassarre et al., 2020) and (Plösch and Neumüller, 2020). In (Lenarduzzi et al., 2020a), the purpose was to investigate how junior developers (also last year undergraduate students as in our case) deal with TD prioritization during refactoring, concluding that they focus homogeneously when dealing with different types of TD and that they appreciate SonarQube as a tool to handle the code quality. Junior developers were also involved in the study (Baldassarre et al., 2020) focusing on diffuseness and remediation time of SonarQube issues in relation with their type. The first study (Lenarduzzi et al., 2020a) also considers same project requirements given to different students as in our study, while in the second study the junior developers were asked to address quality issues to a set of selected open source Java projects. Finally, (Plösch and Neumüller, 2020) shows that fixing SonarQube issues significantly depends on the experience level of students.

Regarding comparison dealing with coding practices and quality issues between Java and Python, (Tan et al., 2021) perform an empirical study of 44 Python projects from Apache Software Foundation ecosystem, and compare the remediation effort with a previous study on Java projects, showing similarities between fixing rates in both languages.

Techniques like FCA, which involve representing, manipulating, and categorizing data, can be employed in software engineering to offer solution to different specific problems such as software reuse (Godin et al., 1995), reverse engineering and code inspection (Dekel, 2002), or concept and fault location (Poshyvanyk and Marcus, 2007), or programming style (Cristea et al., 2021). This study shows another approach in which FCA can be used in investigating quality issues in data from SonarQube.

5 THREATS TO VALIDITY

We addressed potential threats to our study's validity by observing existing best practices for empirical (Paul Ralph (ed.), 2021) and case study research (Runeson and Höst, 2008). In order to facilitate replicating or extending our work, we created a data package that includes the SonarQube rule configuration employed together with the anonymized list of issues resulting from the analysis (Molnar, 2024).

Internal threats were addressed by carrying out a manual examination of the source code included in our study. Our reliance on SonarQube as a static analysis tool can be construed as an internal threat.

We aimed to limit **external threats** by including three course iterations in our study in order to improve data triangulation (Runeson and Höst, 2008) and help identify recurring issues or trends. We included both the complete rule set employed as well as all detected issues in our open data package (not just those tagged *bad-practice*).

Construct threats were mitigated by using SonarQube, the most widely used static analysis tool in both academia and the industry (Avgeriou and o., 2021). We took into consideration existing results arguing for the limited fault-prediction power of individual SonarQube rules (Lenarduzzi et al., 2020), and used detected issues as guidelines that will direct future teaching efforts.

6 CONCLUSIONS

Static code analysis is known to provide an important insight into the quality of software projects. In this paper, we propose an approach that uses FCA as a data mining technique on the output of SonarQube in order to analyze behavior of junior developers. The current analysis focuses on issues related to bad-practice

in Python and Java projects, since this type of issues directly impacts the maintainability and reliability of the source code.

The study shows that junior developers have difficulties handling output operations both in Java and Python. Moreover, the results showed that bad practice issues in Java are often associated to security issues, while bad practice issues in Python are often associated to issues related to high maintainability costs.

As future work, we plan to further explore FCA-based data mining and static analysis tools in software projects in order to analyze the error-prone behavior of junior developers and to find ways in which it can be improved regarding other categories of issues as well, such as ”brain-overload”.

REFERENCES

Avgeriou, P. and o. (2021). An Overview and Comparison of Technical Debt Measurement Tools. *IEEE Software*, PP.

Baldassarre, M. T., Lenarduzzi, V., Romano, S., and Saarimaki, N. (2020). On the diffuseness of technical debt items and accuracy of remediation time when using SonarQube. *IST*, 128:106377.

Boehm, B. and Basili, V. R. (2001). Software defect reduction top 10 list. *Computer*, 34(1):135–137.

Boehm, B. and Papaccio, P. N. (1988). Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477.

Cristea, D., Şotropa, D., Molnar, A.-J., and Motogna, S. (2021). On the use of FCA models in static analysis tools to detect common errors in programming. In *Proc. of ICCS*, pages 3–18.

Dekel, U. (2002). Applications of Concept Lattices to Code Inspection and Review.

Duquenne, V. (2007). What can lattices do for teaching math. and education? volume 331.

Ganter, B. and Wille, R. (1999). *Formal Concept Analysis - Mathematical Foundations*. Springer.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. (2012). *Answer Set Solving in Practice*. SLAIML.

Godin, R., Mineau, G., Missaoui, R., St-Germain, M., and Faraj, N. (1995). Applying Concept Formation Methods to Software Reuse. *IJSEKE*, 5:119–142.

Khaund, A., Sharma, A. M., Tiwari, A., Garg, S., and Kailasam, S. (2023). RD-FCA: A resilient distributed framework for formal concept analysis. *J. Parallel Distributed Comput.*, 179:104710.

Kis, L. L., Sacarea, C., and Troanca, D. (2016). FCA Tools Bundle - A Tool that Enables Dyadic and Triadic Conceptual Navigation. In *Proc. of FCA4AI*. CEUR.

Lehmann, F. and Wille, R. (1995). A triadic approach to formal concept analysis. In *Proc. of ICCS*. Springer.

Lenarduzzi, V., Lomio, F., Huttunen, H., and Taibi, D. (2020). Are SonarQube Rules Inducing Bugs? In *Proc. of SANER*, pages 501–511.

Lenarduzzi, V., Mandić, V., Katin, A., and Taibi, D. (2020a). How long do Junior Developers take to Remove Technical Debt Items? In *Proc. of ESEM*.

Lenarduzzi, V., Saarimaki, N., and Taibi, D. (2020b). Some SonarQube issues have a significant but small effect on faults and changes. A large-scale empirical study. *Journal of Systems and Software*, 170:110750.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, USA.

McConnell, S. (2004). *Code Complete, Second Edition*. Microsoft Press, USA.

Molnar, A. and Motogna, S. (2020a). Longitudinal evaluation of open-source software maintainability. In *Proc. of ENASE*, pages 120–131. INSTICC, SciTePress.

Molnar, A.-J. (2024). Open Data Package. https://doi.org/10.6084/m9.figshare.25270243.v1.

Molnar, A.-J. and Motogna, S. (2020b). Long-Term Evaluation of Technical Debt in Open-Source Software. In *ESEM 2020*. ACM.

Murillo, M. I., Lopez, G., Spanola, R., Guzman, J., Rios, N., and Pacheco, A. (2023). Identification and Management of Technical Debt: A Systematic Mapping Study Update. *JSERD*, 11(1):8:1 – 8:20.

Paul Ralph (ed.) (2021). ACM Sigsoft Empirical Standards for Software Engineering Research, version 0.2.0.

Plösch, R. and Neumüller, C. (2020). Does Static Analysis Help Software Engineering Students? In *ICEIT*.

Poshyvanyk, D. and Marcus, A. (2007). Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In *Proc. of ICPC*.

Potassco (02024). Potassco, the Potsdam Answer Set Solving Collection.

Priss, U. (2013). Using FCA to analyse how students learn to program. In *Proc. of ICFCA*, volume 7880 of *LNCS*, pages 216–227. Springer.

Priss, U. (2020). A preliminary semiotic-conceptual analysis of a learning management system. *Procedia Computer Science*, 176:3702–3709.

Rudolph, S., Sacarea, C., and Troanca, D. (2015a). Membership Constraints in Formal Concept Analysis. In *Proc. of IJCAI*, pages 3186–3192. AAAI Press.

Rudolph, S., Sacarea, C., and Troanca, D. (2015b). Towards a Navigation Paradigm for Triadic Concepts. In *Proc. of ICFCA*, volume 9113 of *LNCS*. Springer.

Runeson, P. and Höst, M. (2008). Guidelines for conducting and reporting case study research in software engineering. *ESE*, 14:131–164.

Slezak, D. (2012). Rough Sets and FCA - Scalability Challenges. In *Proc. of ICFCA*, volume 7278 of *LNCS*.

SonarSource (2024). Sonar source static code analysis.

Tan, J., Feitosa, D., Avgeriou, P., and Lungu, M. (2021). Evolution of technical debt remediation in Python: A case study on the Apache Software Ecosystem. *Journal of Software: Evolution and Process*, 33(4).

Walkinshaw, N. and Minku, L. (2018). Are 20% of files responsible for 80% of defects? In *Proc. of ESEM*.