2024 International Conference on Emerging eLearning Technologies and Applications (ICETA) | 979-8-3315-2771-6/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICETA63795.2024.10850845

Enhancing Code Quality Through Static Analysis: Optimizing the Flava Tool for Detecting Code Smells and Errors Using SonarQube Integration

Sylvia Maťašová
Department of Computers and Informatics, FEI TU of Košice, Slovak Republic
sylvia.matasova@tuke.sk

Martin Chovanec
Department of Computers and Informatics, FEI TU of Košice, Slovak Republic
martin.chovanec@tuke.sk

Renáta Rusnáková
Department of Computers and Informatics, FEI TU of Košice, Slovak Republic
renata.rusnakova@tuke.sk

Dušan Čatloch
Department of Computers and Informatics, FEI TU of Košice, Slovak Republic
dusan.catloch@tuke.sk

***Abstract*—This article focuses on the optimization and enhancement of the Flava tool, with a specific emphasis on integrating static code analysis capabilities. The primary objective is to evaluate various approaches to addressing issues related to code quality and propose effective measures to mitigate common problems such as "code smells" and errors. By expanding the static analysis features of the Flava tool, this work enables the automated detection of code defects, thereby supporting the identification and correction of problematic areas in the code. As a result, the tool provides more comprehensive support for improving the quality and maintainability of software, ultimately leading to more efficient and sustainable software development processes. This research presents significant insights into improving software development workflows, increasing code reliability, and promoting best practices in software engineering. The enhanced Flava tool will contribute to better code quality and reduce the technical debt often associated with poor coding practices. Furthermore, this research addresses the growing need for tools that can seamlessly integrate into development environments, providing real-time feedback and aiding developers in proactively identifying potential issues before they become critical problems.**

Keywords—Static code analysis, Server, Code Smells, Bugs, Detection

I. Introduction

Software projects are an essential component of modern information systems. And technology environment. Their success and quality are Often determined by code quality and development efficiency. However, software projects constantly face challenges and The success and sustainability of their endeavors can be affected by complications.

This post's goal is to extend the Flava tool functionality to detect code smells and code errors. The goal is to improve the tool's ability to identify potentially problematic pieces of code.

The issue addressed in this paper is important to modern software development, as poor code quality, can have a significant impact on the overall success of a project. Until now, there is no sufficiently accurate and efficient tool to detect these problems in the development environment, which underlines the importance of this paper.

This post is motivated by an effort to improve the quality of software projects and optimize the development process through automated code analysis. With this effort comes the expectation of increased developer productivity and ensuring greater sustainability and extensibility of software applications.

In order to provide a comprehensive solution to this issue and actively contribute to the development of the field of software engineering, this work focuses on the implementation and deployment of an improved version of the Flava tool. This new version of the tool will be able to effectively identify "code smells" and code errors, thereby contributing to improving the quality and sustainability of software projects.

II. Related Work

The literature review provides a comprehensive overview of existing research and publications related to the identification and mitigation of "code smell" and bugs in software development.

Fowler and Beck [1] introduced the concept of 'Code Smell' as subjective signs or characteristics of code indicating potential problems. Their work laid the foundation for understanding the importance of identifying and solving these problems in software projects. Their importance lies in indicating design flaws, outdated code, or other factors that negatively affect software quality. It is emphasized that eliminating these odors can significantly contribute to increasing the overall quality and sustainability of a software project.

In addition, errors in code can arise from various causes, such as syntax errors, logical errors, or algorithmic errors. These defects can have a significant impact on the functionality and security of the software product and may require considerable effort to identify and correct.

Martin's book "Clean Code: A Handbook of Agile Software Craftsmanship" [2] emphasizes the importance of clean code and careful work in detecting and fixing 'Code Smells'. Martin provides an overview of best practices for improving code quality and highlights the role of thorough code analysis in ensuring the robustness and reliability of software applications.

According to Johnson and Johnson [3], the success of software projects is directly related to their ability to respond to customer requirements and flexibly adapt to changing market needs. However, these requirements can be compromised by the presence of "code smell" and code bugs, which can lead to reduced readability, maintainability issues, and other complications in the development process. Jones and Smith [4] also state that errors in the code can lead to increased maintenance and repair costs in later stages of development.

Automated code analysis tools play a key role in effectively identifying "code smells" and code bugs. Smith and Smith [5] emphasize the importance of considering contextual factors and project-specific requirements when using these code analysis tools. They also emphasize that when identifying "code smells" and code bugs, it is important to take into account not only syntactic and formal aspects, but also the context and specific requirements of the project.

Agile software development methodologies prioritize flexibility and responsiveness to change. Martin's book [6] discusses how agile principles align with the need to identify and address "code smells" and code bugs to support iterative development and continuous improvement.

Brown et al. [7] present an overview of code quality improvement techniques, including static and dynamic code analysis, testing, and code review. Their work underscores the importance of a multifaceted approach to improving code quality, with each technique contributing to the overall reliability and maintainability of software projects.

Martin's book "Refactoring: Improving the Design of Existing Code" [8] discusses the process of code refactoring as a key strategy for improving code design and quality. Refactoring allows developers to systematically address "code smells" and improve the maintainability and extensibility of software projects.

In this day and age, it's important to have tools to identify these issues and help developers create quality software. These tools can automate the process of identifying "code smells" and code bugs, making code analysis and improvement easier. As Smith [9] pointed out, automated code analysis provides many opportunities to improve the quality and maintainability of software projects. However, there are also problems such as the correct setup and configuration of the tools and the interpretation of the analysis results.

In the article [10] the authors present the early stage of the profile of controlled analysis of the source code, where their interest is focused on the investigation of the source code to generate a knowledge profile. Such a profile represents an objective assessment of the current knowledge and skills, and individual progress compared to the past, relevant deficiencies to be addressed.

The authors of [11] deal with more complex parsers that extract more information from sources. This data can later be processed to gain better insight into the code. For a long time, they have been trying to create a system that would build a precise and knowledgeable profile of programmers based solely on code analysis. In the short term, they presented different approaches, in particular creating a custom tool for obtaining information about the use of certain language constructors or former already popular tools by adding support for code quality measurement, copy-pasted code, and design pattern detection. The mentioned system is Flava, which we also try in the article.

The article [12] deals with the generation of profiles for the C programming language. The main idea is based on the static analysis of source codes, collecting the most important data about their author. Such profiles can point to some of the skills and habits of the programmer. In the process of static analysis, methods and techniques of complexity metrics and clone detection are applied. They also present an experiment aimed at beginning programmers. The results show that knowledge programs provide an easy way to track the progress of novice programmers. It is believed that the main benefit is in the area of understanding the program.

The literature review demonstrates the range of research and practical knowledge available on the topic of "Code Smell" and improving code quality. It emphasizes the importance of identifying and solving these problems to ensure the success and sustainability of software projects.

III. METHODOLOGY

Our research focused on improving the code quality review process and identifying potential issues by integrating bug detection tools and Code Smells directly into the Flava tool. To achieve this goal, we first underwent a thorough analysis of the available options on the market.

Based on this analysis, we decided to implement a combination of SonarQube server and Sonar Scanner. SonarQube server was chosen as the central platform for code quality analysis because it offers advanced analysis capabilities and a wide range of controls that can be tailored to specific project needs. Sonar Scanner was subsequently chosen as a tool for scanning and evaluating the code, which will enable the automated detection of errors and Code Smells in the code.

This new solution aims to provide students with a tool to more quickly identify and fix potential problems in their code before assignments are submitted. The implementation of this solution should improve the overall quality of the code and contribute to more efficient development of software projects.

A. Implementation steps

Installation and configuration of the SonarQube server:The first step will be the installation and configuration of the SonarQube server.

Integration with Flava: The next step will be the integration of the SonarQube server with our Flava tool.

Configuration and start of scanning using Sonar Scanner: Subsequently, we will configure and start scanning of the source code using Sonar Scanner through the Flava tool.

Results Processing and Analysis: After the scanning process is complete, we will thoroughly review the results to identify potential bugs and Code Smells in the analyzed code.

Implementation of fixes and improvements: Based on identified issues, we will implement the necessary fixes and improvements to the code.

The implementation of this change proposal should bring the following benefits:

Faster problem identification: Automated code quality analysis will enable faster detection of potential problems and Code Smells.

Improved overall code quality: The ability to systematically remove errors and improve code quality should contribute to an overall improvement in the quality of student assignments.

Continuous review and improvement: Regular scanning and analysis of the code will allow continuous review and improvement of its quality during the study.

B. *Implementation of Code Smells and bugs detection*

Our implementation is based on the use of SonarQube server, SonarScanner, and SonarQube API to analyze Code Smells and identify errors in our projects. This set of tools allows us to get a comprehensive overview of code quality and systematically uncover potential flaws and problems in the software development process.

As part of our implementation task, we focused on expanding the functionality of the FlavaPlugin interface. Specifically, we added a new function called startSonarScanner, which accepts as a parameter an object of type File representing the project directory and returns a list of issues of type Issues. We subsequently integrated this new functionality into every existing plugin. We will address this issue in more detail in the context of the Java programming language in the JavaPlugin module.

To integrate SonarScanner into our application, we defined the sonarScannerPath variable within the FlavaPlugin, which determines the way SonarScanner is launched. It is also necessary to set a token for Sonar, which we get from the server. We make these settings directly within the FlavaPlugin, which allows our plugins for individual programming languages to access these variables without having to set them again in every part of the application. In this way, we achieved centralized control over the configuration of SonarScanner and ensured consistency of integration throughout our application.

a) Function startSonarScanner: The main activity of the startSonarScanner function is performed in the try block. In the event of an exception, the program is transferred to the corresponding catch block, where the exception is caught and its status is written using the printStackTrace() method from the Throwable class. This method provides information about the place in the code where the exception occurred and the cause of the occurrence. In addition, the current analysis is terminated and the user is presented with a failure message in the user interface.

In the code, a variable called projectPath is defined, in which the absolute path to the project directory is stored using the getAbsolutePath() method from the projectFolder object. This variable is used to dynamically determine the location of the project to be analyzed. In this way, it is possible to easily change the location of the project on which the analysis is to be performed, without the need to manually modify the configuration files or other parameters of the application.

In the code, we created an instance of the Properties class, which is used to load configuration properties from a file. Next, we used a try-with-resources block that initializes a new FileInputStream to load the sonar-project.properties file. This file is defined by projectPath. Within this block, we called the load() method of the Properties class, which loaded the properties from the file and stored them in the Properties instance. If an IOException occurred while loading the file, it was caught and its status was printed using the printStackTrace() method of the Throwable class.

In the code, we set the sonar.java.binaries property to the projectPath value using the setProperty() method of the Properties class. This property indicates the path to the directory containing the project's Java binaries. We then created a try-with-resources block in which we initialized the FileOutputStream to create a new file called sonar-project.properties. Within this block, we called the store() method, which stores the contents of the Properties instance into a file represented by FileOutputStream. If an IOException occurred while saving to a file, it was caught and its status was printed using the printStackTrace() method of the Throwable class. This code focuses on dynamically configuring the analysis process and stores the configuration properties in the sonar-project.properties file.

Before starting the for cycle, it is necessary to create an instance of the Issues. Issue type list in which we will store the results of the analysis. We will then use this list to record the results and return them outside the loop. In this way, we can effectively manage and save the analysis results for further processing or presentation.

Next, a new instance of the HttpConnector class is created using the newBuilder() method. Subsequently, the string methods (url(), token()) are called to set the target URL and add the authentication token.

The url() method establishes the URL address as localhost on port 8000, indicating that the HttpConnector will interact with the server hosted at this location.

The token() function is used to set an authentication token, which serves as an identifier to verify the connection with the target server. This authentication token is configured in the sonarToken variable defined within the FlavaPlugin.

Finally, an HttpConnector instance is created with the parameters set using the build() method. This instance will be used to establish and manage the connection to the target server.

Next, a new instance of the WsClient class is created in the code using the newClient() method on the WsClientFactories factory. This factory provides a default WsClient instance that uses the parameters defined in the httpConnector object to communicate with the server.

Subsequently, the Issue processing service (IssuesService) is obtained from the wsClient instance. This service provides methods to handle Issues on the server.

Then we created a new search request (SearchRequest). This request may contain different criteria for searching for an Issue. In our case, we used the criteria for the number of records per page, setting the key for the component and the Issue status to 'OPEN'. These settings help us get only those Issues that are relevant for our purposes.

The for loop is necessary to repeatedly run the process using the scanner and subsequent modifications. Based on our experiments, we found that repeating this process twice in a row is crucial to obtaining reliable analysis results. We found that in a single run of the process, the analysis results from Sonar were not reliable. We also experimented with adding a delay using the Thread.sleep method set to two minutes, but this modification did not produce the desired results. Therefore, we decided to run the analysis twice, which helped us achieve more reliable results.

In the next part of the code, an instance of the Process class is created by calling the exec() method on the Runtime instance. This Process instance is used to run the external command that is defined in the sonarScannerPath variable. The command contains various arguments that are separated from each other by spaces. The sonar.projectBaseDir string is set to the value of projectPath and the sonar.token string is set to some value from the sonarToken variable.

This code allows you to run an external process, which is SonarScanner, with certain settings and parameters. Since this part of the code is contained in the for loop we mentioned earlier, we decided to initialize the issueList variable as an empty list at the beginning of this loop. We have taken this measure to prevent potential contamination of existing analysis results and to ensure that we only store current results in the issueList.

The next section of code creates an object of the BufferedReader class, which is intended for reading the input stream (output) from the process process. This stream is obtained using the getInputStream() method from the process object and is subsequently 'wrapped' into an InputStreamReader, which allows reading characters from the input stream. BufferedReader is then used to read lines from the input stream one at a time. Before the start of the while loop, we created an auxiliary variable isError, to which we set the default value to false.

In the whole block, each line is sequentially read from the input stream and assigned to the line variable. The loop continues until a null line is read, indicating that we have reached the end of the input stream.

In the body of the loop, each line from the input stream is printed using the System.out.println(line) statement, which displays the line on standard output. This method allows you to monitor the output of the process that was started.

In the cycle, we also used the if check. This section of code is used to check the content of the input text string line. If the phrase 'INFO: EXECUTION FAILURE' occurs in the text, the isError variable is set to true. This mechanism is a core element of our error management strategy during analysis, enabling us to effectively identify and resolve issues within our analysis process.

In the following section of code, the object of the Buffere-dReader class is initialized again, but this time it is intended to read the error output of the process, if there is one. If there is an error output at the output of the process, we change the value of the isError variable to true, signaling that an error has been detected. Subsequently, we check the value of the isError variable, and if the value is true, we raise an IOException with its description, thereby announcing the occurrence of an error in the process. This mechanism is important for effective exception management and problem dentification within the analysis process.

The next step is to call the waitFor() method on the process object, which means that the current thread waits until the running process is finished. The return value of this method is an integer that represents the exit code of the process after its termination. This exit code is assigned to the exitCode variable.

Subsequently, the output code of the SonarScanner process is displayed using the System.out.println() command. This procedure allows you to monitor and diagnose process termination within an application by outputting to standard output.

If the variable ’i’ has the value 0, the cycle proceeds as follows. First, it initializes the variables needed to page the issues. It then runs a do-while loop that continues until the total number of problems is greater than the current page multiplied by one hundred. At each step of this cycle, the search request page is updated issues are searched and their data is retrieved. A list of issues is set up to change the status of issues to ’resolved’ en masse. Due to the limitations of SonarScanner, we have limited this bulk change to one hundred issues per page, although the documentation states that up to five hundred records can be sent, but this limit does not apply and causes errors.

Otherwise, if the variable 'i' is not 0, the following code is executed. First, two variables are initialized - 'total' and 'page', with 'total' used to store the total number of issues and 'page' to indicate the current page in the paging process. Subsequently, the size of the search page is set to 500 records using the 'setPs' method on the 'searchRequest' object.

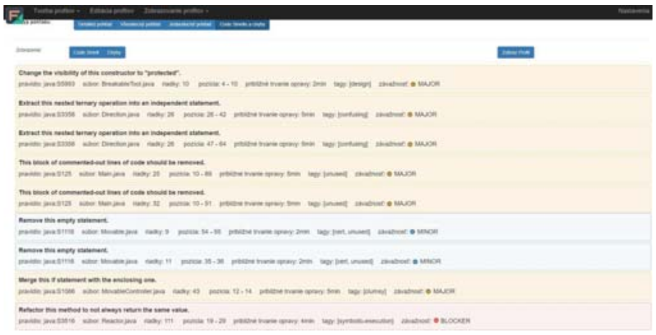
The 'do-while' loop continues to execute repeatedly until the total number of problems exceeds a multiple of the current page and 500. At each loop step, the value of the variable 'page' is incremented and the page of the search request is set to the value of the current page using the 'setP' method. Subsequently, a search for issues is performed using the 'issuesService' service, using the 'searchRequest' object.

After the problem search is performed, the paging information is retrieved from the service response, and the value of the ’total’ variable is updated according to the total number of pages in the pagination. Finally, the list of retrieved issues is added to the existing ’issueList’ list.

Finally, in the startSonarScanner function, we return the list of Issues that we obtained in the for loop. This list contains all issues (Issues) that were identified during the execution of the for loop.

In the next stages of the project, we implemented the startSonarScanner function call, which is used to start the code analysis using SonarScanner. After performing the analysis, we checked the results from this function to see if the problem meets the criteria of the ’java:S1598’ rule, which warns about a mismatch between the declared package and the expected package. The expected package is missing because there is no direct access to the files on the computer during analysis with Flava. Flava has copied the files into her folder and is working with them from there. This means that the original file path, for example,’sk/tuke/flava’, has now been changed to ’/’. We modify the file name stored in the component variable by removing the 'flava:' prefix, which represents the name of the project on the SonarQube server where the analysis results are stored. Subsequently, we sorted them according to their type, namely bugs and code smells. We then integrated these categories into an existing file containing metrics for a given project profile.

In addition, we have added the display of analysis results to the user interface of our application. Based on the severity of individual problems, we created color-coded cards with a description of the respective problems. This visual approach improves the clarity of results and allows users to quickly identify the most critical areas in their code.



Change the visibility of this constructor to "protected".
pravidlo: java:S5993 súbor: BreakableTool.java riadky: 10 pozícia: 4 - 10 približné trvanie opravy: 2min tagy: [design] závažnosť: ● MAJOR

Extract this nested ternary operation into an independent statement.
pravidlo: java:S3358 súbor: Direction.java riadky: 28 pozícia: 28 - 42 približné trvanie opravy: 5min tagy: [confusing] závažnosť: ● MAJOR

Extract this nested ternary operation into an independent statement.
pravidlo: java:S3358 súbor: Direction.java riadky: 28 pozícia: 47 - 64 približné trvanie opravy: 5min tagy: [confusing] závažnosť: ● MAJOR

This block of commented-out lines of code should be removed.
pravidlo: java:S125 súbor: Main.java riadky: 25 pozícia: 10 - 89 približné trvanie opravy: 5min tagy: [unused] závažnosť: ● MAJOR

This block of commented-out lines of code should be removed.
pravidlo: java:S125 súbor: Main.java riadky: 32 pozícia: 10 - 51 približné trvanie opravy: 5min tagy: [unused] závažnosť: ● MAJOR

Remove this empty statement.
pravidlo: java:S1116 súbor: Movable.java riadky: 9 pozícia: 54 - 55 približné trvanie opravy: 2min tagy: [cert, unused] závažnosť: ● MINOR

Remove this empty statement.
pravidlo: java:S1116 súbor: Movable.java riadky: 11 pozícia: 35 - 36 približné trvanie opravy: 2min tagy: [cert, unused] závažnosť: ● MINOR

Merge this if statement with the enclosing one.
pravidlo: java:S1066 súbor: MovableController.java riadky: 43 pozícia: 12 - 14 približné trvanie opravy: 5min tagy: [clumsy] závažnosť: ● MAJOR

Refactor this method to not always return the same value.
pravidlo: java:S3516 súbor: Reactor.java riadky: 111 pozícia: 19 - 29 približné trvanie opravy: 4min tagy: [symbolic-execution] závažnosť: ● BLOCKER

Fig. 1. View Code Smells

IV. RESULT

In the results section, we will discuss the accuracy of the implementation of our tool for detecting 'Code Smells' and code errors. This process will involve testing our tool on the samples we used in the analysis and then comparing those results to the output from the SonarQube tool. In addition, we plan to take multiple samples from a single task and compare them to each other to better understand the differences in the abilities of individual programmers. Finally, we will take multiple tasks from one student and analyze how his ability to write clean code has evolved during his studies at our school.

We marked 'Code Smells' as CS in the results tables.

A. *Comparison of the performance of our tool with SonarQube outputs*

After completing the static analysis and recording the results, we will be looking to compare the number of detected errors and 'Code Smells' between SonarQube and Flava applied to our code samples.

TABLE I.

	SonarQube		Flava		
Cause	***Bugs***	***CS***	***Bugs***	***CS***	***Time***
First sample	6	82	6	82	5m 29s
Second sample	6	51	6	51	4m 35s
Third sample	7	56	7	56	5m 13s

Fig. 2. Comparison of SonarQube and Flava

As we can see in the table above, the results between Sonar-Qube and Flava match. The analysis takes about five minutes because the problem is analyzing the given assignments to one project, on the SonarQube server, which has the same final effect, just that they are from three different students. In the given project, we cannot use the SonarQube API to delete an already-created Issue. We fixed it by changing the status of already existing Issues to 'resolve'. Subsequently, we only look for those that have the status 'open', so we consider them newly created. We perform the entire process of starting the analysis twice in a row, if we started it only once it gave the results of the previous analysis. We did not manage to figure out why, since we pulled the analysis results only after the analysis with the Sonar scanner was finished, and the SonarQube server already had the current results.

Thanks to this comparison, we can identify the rules that have been violated without the need for a detailed review. For example, we can recognize that the rule 'java:S1598' has not been violated. If a violation occurred to him, the results of the analysis would not agree with us. Although there was no violation in our case, it is important to note that such a failure could have adverse consequences on the reliability of the results. It is therefore critical to ensure that these rules are properly followed to avoid potential problems with analysis results while ensuring code quality and security.

B. *Comparison of the same assigment between different students*

In this test phase, we focused on a task from the subject Object Oriented Programming from 2022, implemented in Java. Our sample included seven randomly selected students with varied final grades.

TABLE II.

Sample	BUGS	CS	Time	Evaluation
5365	5	70	3m 42s	9% (FX)
6725	0	69	1m 13s	55% (E)
6722	0	65	0m 51s	62% (D)
5607	6	72	3m 22s	77% (C)
7019	1	59	0m 52s	85% (B)
6645	4	51	0m 59s	91% (A)
6386	11	96	3m 31s	100% (A)

Fig. 3. Comparison between different students

The chart we created provides a detailed look at the relationship between students' percentiles and their code quality, as measured by the number of bugs and smells in their programming code. For this research, we collected data from students' evaluations of the same programming task and compared them with the number of bugs and smells found in their code.

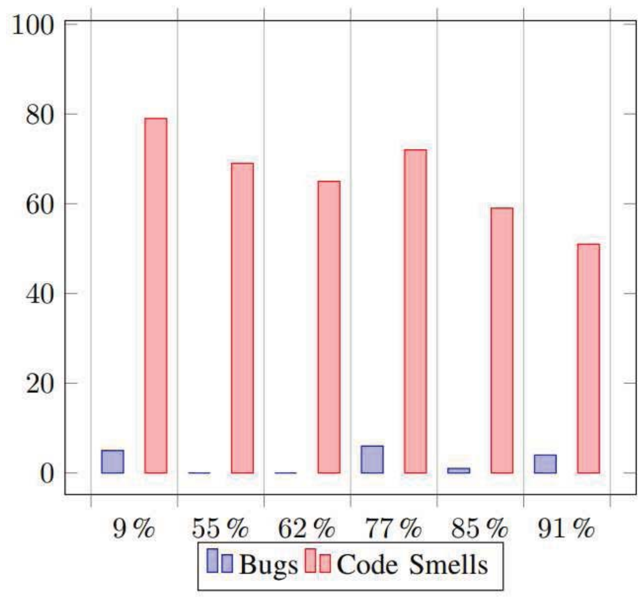
The students' percentage rating corresponds to their overall rating for the subject in which the assignment was given. This assessment is the basis for our analysis of student performance.

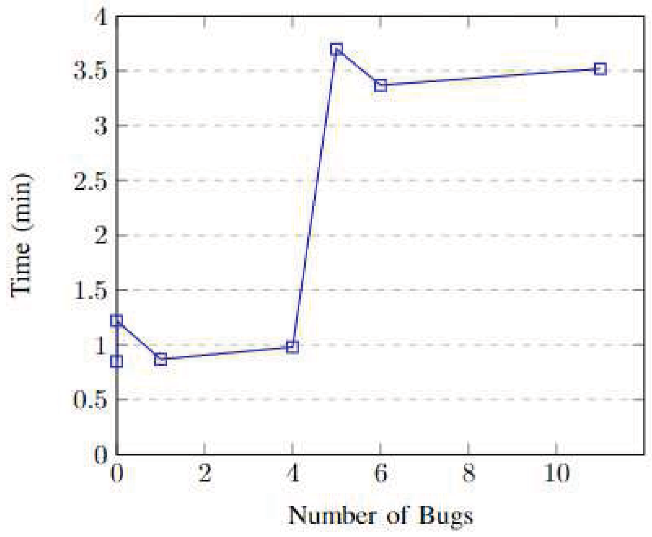
Analyzing these three factors—percentage rating, number of code errors, and number of code smells—allows us to gain a deeper understanding of the connection between code quality and student performance. A graph displays these relationships in a visual form, allowing for quick and efficient comparison of results and identification of patterns and trends.

The following graphs give us a detailed view of how the time required for analysis changes depending on the number of code smells and errors. For better clarity, we have divided these data into two separate graphs.

The data shows that the more bugs, the more time we need to analyze. We can observe this trend in all performed analyses.

Similarly, the second graph shows that as the number of code smells increases, the time required for analysis also increases. This relationship is important because it indicates that code quality affects analysis time. Identifying and removing code smells requires additional effort, resulting in a more demanding analysis.





For better orientation in the following graph, we created a sub-table for our main table where we compared students. In this table, we show the sample code (student code), the number of analyzed files and the time required to analyze the project.

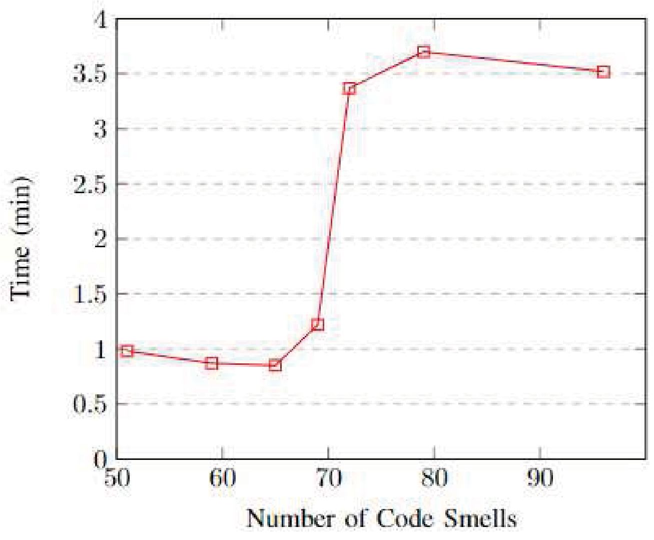
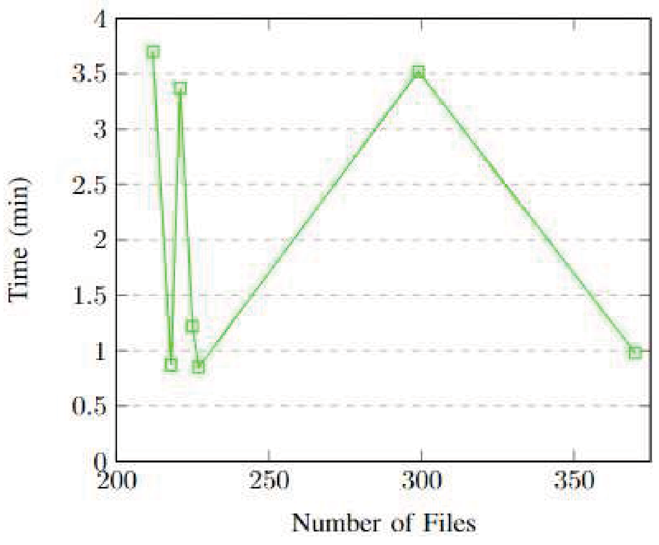


TABLE III.

Sample	Number of Files	Time
5365	212	3m 42s
6725	225	1m 13s
6722	227	0m 51s
5607	221	3m 22s
7019	218	0m 52s
6645	370	0m 59s
6386	299	3m 31s

Fig. 4. Number of files in comparison

In the following graph, we observe the relationship between the number of files and the time required for analysis. For this particular number of files, the parsing time does not appear to be significantly dependent. However, if we increased the number of files by 1000, we could observe changes in this dependence.



During this testing phase, we identified interesting relationships between the number of issues discovered and the analysis speed, as shown in the table. We observed that the fewer problems found in the code, the faster the analysis. Specifically, we found that if no error was identified in the code, the analysis time was approximately one minute. However, with five or more errors, we saw a dramatic increase to over three minutes, a significant time difference.

It is interesting to observe that the best-rated student does not show the best-processed code in the table. Paradoxically, this student achieved an excellent grade despite his code containing eleven errors and ninety-six instances of 'Code Smells'. This situation may be a result of the higher volume of work that the student has done compared to others.

C. *Comparison of the different assigments from one subject of one student*

In this part of the testing, we focused on observing the development of the student's abilities during the subject Programming in the DotNET environment. Given that this course had three submissions in 2021 plus a code addition for the exam, we were able to track how the student developed during the course.

TABLE IV.

Sample	BUGS	CS	Time	File Count
1st transmission	0	0	0m 31s	44
2nd delivery	32	510	3m 56s	377
3rd delivery	32	512	3m 55s	576
Exam	32	512	4m 25s	576

Fig. 5. Comparison of different assignments from one subject

Ultimately, we see a lack of progress. We could probably record it if students paid more attention to writing clean code, even though it's not required. As the amount of code submitted increases, and so does the number of problems in it.

D. *Comparison of the different assigments from different subjects of one student*

In the following part of the work, we will follow the gradual development of one specific student from his matriculation assignment in computer science, through the beginnings at the university to the second year of engineering studies. We will focus on his lessons learned and improvements in writing clean code. We selected the following subjects for this analysis:

- Graduation - maturity exam in the last year of high school
- OOP - Object-oriented programming (second year of Bachelor's studies - winter semester)
- WT - web technologies (second year of Bachelor's studies - summer semester)
- PDotNET - Programming in the DotNET environment (second year of Bachelor's studies - summer semester)
- ISI - Intelligent systems in computer science (third year of Bachelor's studies - winter semester)
- SMART - Application development for smart devices (third year of Bachelor's study - winter semester)
- BP - Bachelor's project (third year of Bachelor's studies - summer semester)
- BvPS - Security in computer systems (third year of Bachelor's studies - summer semester)
- IOT1 - Basics of the Internet of Things (third year of Bachelor's studies - summer semester)
- PPS - Parallel computer systems (third year of Bachelor's studies - summer semester)

TABLE V.

Subject	BUGS	CS	File Count
Graduation (Python)	0	3	42
OOP (Java)	6	62	219
WT (JS)	0	82	14
PDotNET (C#)	32	512	576
ISI (Python)	0	3	18
SMART – Compass (JS)	0	18	61
SMART – Hack (JS)	2	41	40
BP (C#)	1	0	105
BvPS (Python)	0	3	2
IoT1 (Python + C#)	0	114	850
PPS (C#)	0	0	

Fig. 6. Comparison of different assignments from different subjects

In the analysis of the results of our tests, we identified several interesting trends.

In the case of the Python programming language, we observed a relative stability in the number of Code Smells, indicating that there has been no significant improvement, but neither has the situation worsened. It looks like the student who worked on this language tried to follow clean code principles. However, it is important to mention that we identified 82 cases of Python Code Smells for the Internet of Things Fundamentals (IoT1) course. It should also be noted that several students participated in the project, not just the one whose work we are evaluating.

For the JavaScript programming language, we observed an improvement between the summer semester of the second year and the winter semester of the third year of undergraduate studies. While in the second year, we identified 82 instances of Code Smells in fourteen files, in the third year we had only 61 instances in 40 files for the Compass task. However, we saw worse results in the Hackathon entries compared to the second year, which may be related to the fact that the code was developed in a team. Specifically, we identified two bugs and 41 instances of Code Smells in 40 files.

In the case of C Sharp, we have seen some improvement. We found disastrous results at the start, analyzing 576 files. Our analysis revealed 32 bugs and 512 cases of Code Smells. In the third year of our bachelor's studies, we worked on two projects in the C# language. One of these projects was also written in Python, which we discussed above. In the case of C Sharp, we found no cases of Code Smells or errors. In another case, as part of a bachelor's thesis, we identified only one error in the code. The project focused on the subject of Parallel Computer Systems, which was part of the first year of engineering studies, we did not find any Code Smells or errors.

In the case of the Java programming language, we had only one project where we identified six code bugs and 62 Code Smells.

In this case, we can state the improvement of the programming skills during the studies of the given student. He was improving in Python, if we consider that the project in which he had the worst results was created by several students. In the case of the JavaScript language, we can also note an improvement, again with the fact that more people worked on the last assignment in this language. We can observe the greatest progress in the C# language, except for the first encounter with this language, our student wrote clean code.

V. CONCLUSION

Based on our thorough analysis and testing, we concluded that implementing static analysis using SonarQube brings obvious benefits to our projects. The [13] literature emphasizes that code quality has a significant impact on the cost and effectiveness of software maintenance. Deficiencies in the code can lead to increased time and cost needed to identify and fix bugs and 'Code Smells' in later stages of development. Also in the article [14] it is stated that agile methodologies emphasize the importance of flexibility and speed in software development. This approach also includes continuous code analysis and identification of 'Code Smells' and code errors to ensure a quick and efficient response to changes in requirements. These benefits include improving code quality, identifying potential bugs, and increasing overall code clarity and maintainability. However, despite these positives, our research has revealed several significant issues that require our attention and resolution.

One of the main problems we have identified is the inability to remove issues created using the SonarQube tool. Instead, these issues can only be marked as 'resolved', which can lead to a backlog of unresolved issues and server overload. Another aspect we noticed is a flaw in the implementation of the 'java:S1598' rule, which can cause errors or inaccuracies in the analysis results.

Another problem found is the lack of support for code analysis in C and C++ languages, which are, however, an important part of our Flava project. This shortcoming limits the effectiveness and completeness of our analysis and needs to be addressed in the future.

Additionally, we observed that implementing static analysis using SonarQube caused a slight increase in analysis time. This increase in time was due to the need to run the analysis repeatedly, as we often experienced inaccurate results obtained from the server.

A summary of our research findings points out both the advantages and disadvantages of using static analysis through the SonarQube tool. Based on this knowledge, we have concluded that it is necessary to explore in more detail on the possibilities to improve and extend this tool to better meet our needs and expectations.

We recommend further activities that should focus on optimizing the analysis and solving the identified problems. It is important to pay attention to these shortcomings in order to successfully use static analysis as an important tool for improving software quality.

In conclusion, it can be concluded that the implementation of static analysis using SonarQube represents an important step in the process of improving software quality. However, to reach its full potential and fully cover our needs and expectations, it is necessary to make further efforts and work to improve this tool.

Another suggested improvement is to consider making the Flava tool available through the school server. The current local version limits the accessibility of the tool and its use outside the student or research environment. Deploying the tool on the school server would allow a wider range of users to enjoy its benefits and conKeytribute to a better integration of the tool into the teaching process and research projects. This move would increase the availability of the tool and could encourage its wider adoption and use in the academic community.

REFERENCES

[1] M. Fowler and K. Beck, “Refactoring: Improving the design of existing code,” *Addison-Wesley Professional*, 1999.

[2] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[3] E. Johnson and S. Johnson, *Software Engineering* Best *Practices: Lessons from Successful Projects in the Top Companies*. McGraw-Hill Education, 2014.

[4] M. Jones and R. Smith, “Impact of code quality on software maintenance costs: A case study,” *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 687–701, 2017.

[5] D. Smith and J. Smith, “Automated code analysis tools: A comprehensive review,” *Journal of Software Engineering Research and Development*, vol. 22, no. 3, pp. 245–264, 2016.

[6] R. C. Martin, *Agile Principles, Patterns, and Practices in C#.* Prentice Hall, 2006.

[7] J. Brown and E. Taylor, “Improving software quality: A survey of modern techniques,” *Journal of Systems and Software*, vol. 122, pp. 315–336, 2016.

[8] R. C. Martin, “Refactoring: Improving the design of existing code,” *Addison-Wesley Professional*, 1999

[9] A. Smith and D. Johnson, “Automated code analysis: Challenges and opportunities,” in *Proceedings of the International Conference on Software Engineering*. IEEE, 2019, pp. 112–121.

[10] E. Pietriková and S. Chodarev, “Profile-driven source code exploration,” in *Federated Conference on Computer Science and Information Systems (FedCSIS),* 2015, pp. 929–934.

[11] I. Vehec and E. Pietriková, “Metrics for student source code analysis,” in *International Conference on Emerging eLearning Technologies and Applications (ICETA),* 2020, pp. 739–744.

[12] E. Pietriková, “Towards program comprehension: Knowledge profiles for c programmers,” *Acta Electrotechnica et Informatica*, vol. 21, no. 4, pp. 30–35, 2021.

[13] L. Clarke and B. Johnson, “The impact of code quality on software maintenance: A literature review,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 27, no. 9-10, pp. 1489–1518, 2017.

[14] P. Wilson and S. Thompson, “Agility in software development: A systematic literature review,” *Information and Software Technology*, vol. 80, pp. 78–91, 2016.