Taxonomic Classification of Open Source on GitHub Cephyr.

Daniel Giraldo Sanclemente[1],

[1] Departamento de Computação e Sistemas
Universidade Federal de Ouro Preto – João Monlevade, MG– Brazil

daniel.sanclemente@aluno.ufop.edu.br

Abstract. *The following document presents a code analysis conducted on Cephyr, which is a real-time operating system (RTOS). The diagnosis is performed using the SonarCloud platform, and Python along with pandas is utilized to analyze the first 500 code smells found. The analysis yields a total of 197 unique code alert messages and 52 infringed rules in code smells. All of this is classified into six categories: severity, estimated effort required to fix, clean code attribute, clean code attribute category, impacts, and security standards.*

1. Introduction

As programming has evolved over time, new needs and areas have emerged as essential requirements when developing software, or at least requirements for the best programming companies. A clear example of this is the area of quality. Nowadays, it is crucial not only for developments to be functional, but also for the software to be reliable, secure, maintainable, among other qualities. As a result, technologies have been created to assess software quality according to specific parameters that have been established after extensive study in the field.

SonarCloud is a web tool that integrates with platforms like GitHub to extract code for analysis. This technology enables static code analysis, meaning the code does not need to be executed. By doing this, defects or risky situations in the code can be addressed before moving on to dynamic testing, where the costs of fixing a defect can significantly increase. Static analysis offers benefits such as early detection and correction of defects, early warnings about suspicious aspects through metric calculations, detection of dependencies, and inconsistencies in the software model. This improves the code's maintainability and enhances code production by learning from the detected errors.

Code Smells are indicators of bad programming practices that should be avoided, as they tend to negatively affect code quality. While they are not actual errors, these practices can make the code harder to understand, maintain, and scale in the long run. The presence of Code Smells can lead to issues like logic duplication, unnecessary complexity, or a lack of clarity in the execution flow. Tools like SonarCloud are key to identifying and mitigating these problems, as they allow for the analysis of code quality in multiple programming languages. Currently, SonarCloud evaluates code based on a set of 3,972 rules, addressing various categories such as language, security, performance, and software maintainability.

2. Methology

For the development of this project, a fork was created from the following open-source repository on GitHub: https://github.com/zephyrproject-rtos/zephyr. This open-source project, widely recognized in the realm of real-time operating systems (RTOS), was selected for its relevance and complexity, allowing for an in-depth analysis of best practices and potential improvements in the code.

Subsequently, a connection was established with SonarCloud to execute a code analysis. This enabled the acquisition of detailed metrics and recommendations that facilitated the identification of potential code smells.

To enable a more detailed analysis, the SonarCloud API was utilized, allowing for the extraction of part of the diagnosis in .csv format for more efficient data handling. This connection was implemented using the Python programming language, which provided the necessary flexibility to automate the extraction. Together with the pandas library, data processing was carried out. Given the complexity of the code, the focus was set on the first 500 issues detected by the tool. After eliminating duplicates, the final analysis revealed a total of 197 code smells present in the code.

Additionally, in the same manner as previously described, an abstraction of all the rules found in SonarCloud for identifying code smells was performed. A cross-reference was made between the violated rules in the diagnosis and the existing rules, resulting in a much more detailed final diagnosis.

3. Results

After the analysis, a total of 52 violated rules were identified in the 197 reports of code smells, which are listed in Table 1. However, this count does not yet account for the repetition of each rule. Graph 1 illustrates how many times each rule is repeated according to its rule identifier.

Table1. *Rules id and name*

Rule	Name
c:S1905	Redundant casts should not be used
c:S859	A cast shall not remove any const or volatile qualification from the type of a pointer or reference
c:S5350	Pointer and reference local variables should be "const" if the corresponding object is not modified
c:S5955	Loop variables should be declared in the minimal possible scope
c:S995	Pointer and reference parameters should be "const" if the corresponding object is not modified
c:S3776	Cognitive Complexity of functions should not be too high
c:S1659	Multiple variables should not be declared on the same line
c:S1110	Redundant pairs of parentheses should be removed
c:S134	Control flow statements "if", "for", "while", "switch" and "try" should not be nested too deeply
c:S3458	Empty "case" clauses that fall through to the "default" should be omitted
c:S924	Loops should not have more than one "break" or "goto" statement

c:S3687	Local variables and member data should not be volatile
c:S125	Sections of code should not be commented out
c:S1481	Unused local variables should be removed
c:S3646	Types and variables should be declared in separate statements
c:S1135	Track uses of "TODO" tags
c:S954	#include directives in a file should only be preceded by other preprocessor directives or comments
c:S936	Function names should be used either as a call with a parameter list or with the "&" operator
c:S108	Nested blocks of code should not be left empty
c:S1854	Unused assignments should be removed
c:S1172	Unused function parameters should be removed
c:S824	Functions should not be declared at block scope
python:S1192	String literals should not be duplicated
python:S3776	Cognitive Complexity of functions should not be too high
cpp:S5421	Non-const global variables should not be used
cpp:S1172	Unused function parameters should be removed
cpp:S995	Pointer and reference parameters should be "const" if the corresponding object is not modified
cpp:S5008	"void *" should not be used in typedefs, member variables, function parameters or return type
cpp:S1116	Empty statements should be removed
cpp:S4962	"nullptr" should be used to denote the null pointer
cpp:S5945	C-style array should not be used
cpp:S5566	STL algorithms and range-based for loops should be preferred to traditional for loops
cpp:S6022	"std::byte" should be used when you need byte-oriented memory access
c:S1820	Structures should not have too many fields
c:S1301	"if" statements should be preferred over "switch" when simpler
c:S1066	Mergeable "if" statements should be combined
c:S912	The right-hand operands of && and || should not contain side effects
c:S3491	Redundant pointer operator sequences should be removed
c:S3358	Conditional operators should not be nested
c:S1134	Track uses of "FIXME" tags
c:S959	"#undef" should be used with caution
python:S3358	Conditional expressions should not be nested
cpp:S5205	Function pointers should not be used as function parameters
javascript:S2703	Variables should be declared explicitly
javascript:S2004	Functions should not be nested too deeply
c:S1103	"/*" and "//" should not be used within comments
c:S2234	Parameters should be passed in the correct order
c:S1186	Methods should not be empty
c:S1116	Empty statements should be removed
c:S886	The three expressions of a "for" statement should only be concerned with loop control
python:S112	"Exception" and "BaseException" should not be raised
python:S108	Nested blocks of code should not be left empty

Table 2. *Repetition of broken rules*

Rule	**Severity**	**Quantity**

c:S995	Minor	66
c:S1172	Major	65
c:S859	Critical	46
c:S1659	Minor	40
c:S1905	Minor	33
c:S3776	Critical	32
c:S5350	Minor	27
c:S954	Major	18
cpp:S5945	Major	15
c:S134	Critical	12
python:S112	Major	11
c:S1854	Major	10
c:S3687	Major	9
c:S1135	Info	8
c:S3358	Major	7
c:S1066	Major	6
c:S108	Major	6
c:S1110	Major	6
c:S912	Blocker	6
c:S924	Major	6
python:S3776	Critical	5
c:S1301	Minor	4
c:S3646	Minor	4
c:S5955	Minor	4
cpp:S995	Minor	4
c:S1103	Minor	3
c:S125	Major	3
c:S1481	Minor	3
cpp:S1172	Major	3
cpp:S4962	Critical	3
cpp:S5008	Critical	3
python:S1192	Critical	3
c:S1116	Minor	2
c:S1134	Major	2
c:S1186	Critical	2
c:S2234	Major	2
c:S3458	Minor	2
c:S824	Major	2
c:S936	Critical	2

c:S959	Critical	2
python:S108	Major	2
c:S1820	Major	1
c:S3491	Blocker	1
c:S886	Minor	1
cpp:S1116	Minor	1
cpp:S5205	Major	1
cpp:S5421	Critical	1
cpp:S5566	Minor	1
cpp:S6022	Major	1
javascript:S2004	Critical	1
javascript:S2703	Blocker	1
python:S3358	Major	1

As an additional process, a taxonomy proposed by SonarCloud classifies the rules into six categories: severity, estimated effort required to fix, clean code attribute, clean code attribute category, impacts, and security standards.

- **Severity**: The severity is classified as blocker, critical, major, and minor. The assignment of values considers the worst possible consequence that could occur, and the rules are classified accordingly.
- **Remaining Function Base Effort**: This metric is generally expressed in minutes and is calculated as a standard estimate of the time an experienced developer would take to fix the issue. A value is assigned based on factors such as code complexity, the number of affected lines, and the severity of the identified problems.
- **Clean Code Attribute and Clean Code Attribute Category**: These metrics relate to attributes of clean code, indicating a focus on code quality according to clean programming principles such as readability, simplicity, and maintainability.
- **Impacts**: This refers to the "impacts" that a code issue has on certain key aspects of the software. These impacts can relate to the overall code quality, maintainability, security, performance, or efficiency.
- **Security Standards**: Security standards are those that are violated when breaking the established rule. In some cases, there may not be a specific standard, while in others, there may be more than one.

For a detailed analysis and the code implemented, you can review the following link: https://github.com/Danigs2301/ProjectSoftwareQuality.git. The file containing the final analysis is `analysis.xlsx`, which shows the 197 alerts along with their classifications.

4. Discussion

The reflection on the analysis reveals a deep effort to break down and classify rule violations in the code. From the 197 reports of code smells, 52 infringing rules have been identified. However, the true value of this analysis goes beyond simply identifying how many rules have been violated. The graph showing the repetition of rules, along with the taxonomy proposed by SonarCloud, provides a more comprehensive understanding of code quality—not only in terms of the quantity of issues but also in their severity, correction time, and the impact they have on the product.

Dividing the rules into six categories allows for a richer and more nuanced perspective. Severity (blocker, critical, major, minor) emphasizes the importance of prioritizing the resolution of certain issues. Metrics regarding correction time, such as Remaining Function Base Effort, assist in planning the necessary resources for code improvement, while metrics related to clean code indicate a commitment to development practices that promote the long-term sustainability of the software.

The focus on impacts, covering aspects such as security, maintainability, and efficiency, reinforces that the violated rules not only affect the immediate functionality of the software but also its future adaptability and resilience. Additionally, considering security standards serves as a reminder that some issues may pose critical risks that transcend the code structure and affect system security.

5. Conclusions

The previous analysis is just one part of the multiple issues that the code may present. While violations and areas for improvement have been identified, it is essential to recognize that the code is already at an advanced stage of development. This further underscores the importance of development teams adopting tools like SonarCloud not just as a corrective measure, but as a continuous quality assurance strategy.

SonarCloud offers a proactive approach, allowing developers to identify and fix issues during the development cycle. By integrating with daily work environments, this technology provides real-time feedback on code deficiencies, facilitating immediate decision-making and reducing the accumulation of technical debt. Instead of waiting for problems to escalate and become costly production errors, teams can address quality violations proactively, improving efficiency and maintaining software stability.

Moreover, the constant use of these tools fosters a culture of continuous improvement within the team, where code quality is not seen as an isolated process at the end of development but as an integral component of daily activities. This promotes a preventive approach and encourages developers to write cleaner, safer, and more maintainable code from the start.

SonarCloud's ability to classify issues according to their severity, impact, and the time required to fix them allows teams to prioritize efforts based on available resources, maximizing benefits and ensuring that the final product meets high-quality standards.

Ultimately, this tool not only enhances the technical quality of the code but also contributes to the overall success of the project by reducing risks and optimizing development time.

References

SONARCLOUD. SonarCloud Web API. SonarCloud. https://sonarcloud.io/web_api. Accessed: September 8, 2024.

SONARSOURCE. SonarCloud Rules. SonarCloud Documentation. https://docs.sonarsource.com/sonarcloud/digging-deeper/rules/. Accessed: September 8, 2024.