Evaluating the design of an open-source project

Software Design and Modelling, Università della Svizzera Italiana

Luca Di Bello

Monday 4th November, 2024

1 Introduction

This assignment requires leveraging multiple specialized tools to scan a selected open-source project in order to automatically detect design flaws such as code smells and anti-patterns. The results will be compiled in a single report to provide a detailed analysis of the project's codebase and identify potential areas for improvement in code quality and maintainability.

To address this task, SonarQube and PMD static code analysis tools will be utilised, which are both tools are widely used in the industry to detect a wide range of coding flaws and bad practices.

Similar to the previous assignment, the chosen open-source project must meet the following requirements: at least 100 stars, 100 forks, 10 open issue, and at least 50'000 lines of Java code (comments included). In order to find a valid project, the GitHub search feature was used, filtering the results based on the cited requirements. To do so, the following search query was used:

```
stars:>100 forks:>100 language:java
```

Listing 1: GitHub search query

Since GitHub's search feature does not offer filters for open issues or total number of lines of code, each result was manually inspected to ensure its requirements were satisfied. To count the total lines of code (later referred as *LOC*) of a project, the web application Count LOC was used. Using this tool, we are able to easily determine the LOC count of a project by providing the URL of the GitHub repository, without the need to clone the repository locally.

1.1 Project selection

To provide a meaningful analysis and gain insights into code quality in large open-source projects, the search was focused on active projects with a large community and a good number of stars and forks. After selecting a few projects that satisfied the requirements, three were chosen for further inspection:

- apache/camel: An open-source integration framework based on known Enterprise Integration Patterns (EIPs). [1] *Apache Camel* provides the tools to connect different messaging systems and protocols, providing easy integration and routing of messages across different systems. The project has 5.5k stars, 4.5k forks, 455 open issues (refer to Jira dashboard here) and around 1.5M LOC. This project was selected as it represents a large and complex project, used in many production environments. Unfortunately, the project was later discarded due to its extreme size, which could make the analysis too complex and time-consuming.

- hibernate/hibernate-orm: The *Hibernate ORM* is one of the most popular Java Object-Relational Mapping (ORM) frameworks, allowing developers to map Java objects to database tables and vice versa, facilitating the development of database-driven applications. The project has 6k stars, 3.5k forks, 232 open issues and over 1.3M LOC (comments excluded). *Hibernate* is widely

used in the industry and has a large community of developers, making it a perfect candidate for the analysis. Similar to the previous project, the size of the codebase was considered too large for the scope of the assignment, and was therefore discarded.

- resilience4j/resilience4j: Library to improve resiliency and fault tolerance in Java projects, providing a set of modules to face common issues such as rate limiting, circuit breaking, automatic retrying and more. The project has 9.8k stars, 1.3k forks, 219 open issues and around 80k LOC (comments excluded) across 6 different modules. This project was inspired by the Netflix Hystrix fault-tolerance library. Since *Hystrix* is no longer in active development, the Netflix team advised users to migrate to *resilience4j*. [2]

  The *resilience4j* library is actively maintained by over 100 contributors and employed by many companies in their production systems. Due to its smaller size and its utility in real-world applications, this project was chosen for the analysis.

1.2 High-level overview of the project structure

The *resilience4j* library includes six main modules designed to improve application resiliency: *CircuitBreaker* to prevent cascading failures, *RateLimiter* to control request rates, *Bulkhead* to limit concurrent calls, *Retry* for automatic retries, *TimeLimiter* for operation timeouts, and *Cache* for storing results to enhance efficiency. Each module is self-contained, allowing developers to use only the necessary features without including the entire library. Each module includes its own testing suite to ensure implementation correctness and reliability.

The library also offers adapters for popular frameworks (such as *Spring Boot*, *Micronaut*, *Reactor*) and libraries (like *RxJava* and *CompletableFuture*).

1.3 Building the project

The project utilizes the Gradle build system to manage dependencies and build the project. Thanks to the Gradle Wrapper script, it is possible to automatically configure the project and download the necessary dependencies without the need to install Gradle on the local machine. The following command is used to build the project and generate the necessary artifacts:

```
./gradlew build -x test
```

Listing 2: Building the project

The additional flag `-x test` is used to skip the execution of the several test suites included in the project. Since the focus of this assignment is on the analysis of the codebase and not on the testing process, the tests were excluded as their execution could take a considerable amount of time and would not add any value to the analysis.

1.4 Initial expectations and analysis strategy

Due to its intensive use and the large community behind the project, we expect the *resilienc4j* codebase to exhibit a high level of maintainability and readability. The project is actively maintained and receives regular updates, properties hinting an up-to-date codebase that follows the latest best practices in Java development. Given the nature of *resilience4j*, a focus on performance, reliability, and robust error handling mechanisms is expected.

This theory is supported by the fact that each pull request, apart from being reviewed by the maintainers, is also automatically tested using tailored GitHub actions. The configured GitHub actions show that for some critical *pull requests*, the project employs SonarCloud to automatically check the code quality of the changes. This is a good indicator that the project maintainers care about the code

quality and are actively working to improve it. Refer to the GitHub actions configuration file for more details. Even though the project employs multiple tools and techniques to ensure code quality, is still expected to find minor issues and potential areas for improvement in the codebase.

Inside this report we will only analyze files related to the core library, leaving behind the test files provided by the project. The test suites are not relevant for the analysis of the code quality, as they do not represent the actual functionality of the library. To do so, we will instruct the scanning tools to ignore the test files and focus only on relevant ones.

1.5 Usage of scanning tools

The process of using mentioned scanning tools is straightforward; both SonarQube and PMD provide a comprehensive command-line interface that allows users to easily scan a project's codebase. The tools can be configured to use a specific *ruleset*[1], which defines the coding rules that will be used to analyze the project.

Since the codebase will be scanned two different tools, and summarize their results inside single report, was decided to opt for a generic ruleset for each tool to provide a common ground for the analysis: if the two tools will detect similar issues, it will be easier to compare the results. In this way, we can ensure that the results are consistent and that the analysis is fair.

PMD In order to run the PMD static code analysis tool, we need to define some important parameters, such as the ruleset to be used, the version of Java to analyze, the output format, and the output file. The following command can be used to run PMD on the *resilience4j* project:

```
/path/to/pmd check -d /workspaces/design-evaluation/resilience4j \
  -R rulesets/java/quickstart.xml -f csv \
  --use-version java-17 --report-file /path/to/report.csv
```

Listing 3: Command to run PDM static code analysis

For a comprehensive codebase scan, the `quickstart` ruleset was selected for its balanced set of general-purpose rules that cover a wide range of coding flaws and anti-patterns considered industry-standard. This ruleset represents a good starting point, allowing to have a general overview of the project's code quality. Refer to the PMD documentation for a complete list of rules included in the `quickstart` ruleset.

The output format was set to `csv` in order to easily preprocess the results. Refer to subsection 2.1 for more details.

SonarQube On the other hand, the SonarQube code analysis tool requires a more complex setup, as we need to install the *SonarQube Scanner CLI* and configure a local *SonarQube Server* instance where the results of the analysis will be stored. The SonarQube Scanner CLI is used to analyze the codebase of a project and send the results to the SonarQube Server, which provides a web interface to visualize the detected issues.

In order to start the SonarQube server, we can use the pre-built Docker image provided by the SonarQube team. We can start the container using the following command:

```
docker run --name sonarqube-server -p 9000:9000 sonarqube:lts-community
```

Listing 4: Starting the SonarQube server

After ensuring that the server is up and running, we can start the analysis of the *resilience4j* project using the SonarQube Scanner CLI by running the following command:

[1] Set of rules that the tool uses to detect issues in the codebase.

```
/path/to/sonar-scanner -Dsonar.projectKey=resilience4j \
  -Dsonar.sources=/path/to/resilience4j
  -Dsonar.host.url=http://localhost:9000 \
  -Dsonar.login=admin -Dsonar.password=admin \
  -Dsonar.scanner.skipJreProvisioning=true
```

Listing 5: Command to run SonarQube analysis on *resilience4j* project codebase

The default SonarQube ruleset was chosen to capture a broad range of issues thus providing a common basis for comparing the tools results. Refer to the SonarQube documentation for a complete list of rules included in the default ruleset.
Note: by default the SonarQube Server credentials are set to `admin:admin`. After the first login, the password can be changed to a more secure one.

Furthermore, since SonarQube does not provide an option to export results directly to CSV, we manually exported the issue counts in a CSV file to allow programmatic comparison with the PMD results. Refer to subsection 2.1 for more details.

2 Analysis of Results

By running the two tools on the codebase, we were able to find a large number of issues: PMD found 1074 issues, while SonarQube found over 1300 issues. Both tools already divide the issues into categories, allowing for easier analysis of the results.

2.1 Data Preprocessing and Mapping

The two tools recognize different kinds of issues, and the way they categorize them differs. As the two sets of issues are not directly comparable, a Python script was developed to read the output of both tools and extract the categories of issues detected, counting the number of issues in each category.

SonarQube found 21 different categories, providing a more detailed analysis of the issues. PMD, on the other hand, provides a more general categorization with only 7 categories. Therefore, we decided to map the SonarQube categories to the PMD categories to provide a more general overview of the issues identified by both tools. The mapping is as follows:

- **Code Style:** Mapped from SonarQube's *convention*, *unused*, *confusing*, *clumsy*, *obsolete*, *duplicate*, and *redundant*, *brain-overload* categories. Issues in this category are related to code readability and maintainability, and the misuse of Java features.
- **Best Practices:** Mapped from SonarQube's *java8*, *bad-practice*, and *serialization* categories. These issues relate to code quality and the misuse of Java features.
- **Design:** Includes SonarQube's *design* and *brain-overload* categories, focusing on architectural and design quality. The *brain-overload* category is included here as it relates to the complexity of the code, highlighting cases where a class or method is too complex (cyclomatic complexity and cognitive complexity, refer to SonarQube's documentation for more details).

  Furthermore, it was later decided to map the *brain-overload* category to the *Code Style* category as well, as it also relates to the readability and maintainability of the code.
- **Error Prone:** Mapped from SonarQube's *cert*, *pitfall*, *suspicious*, *cwe*, *error-handling*, *owasp-a3*, and *leak* categories, each addressing issues that could lead to potential bugs or security vulnerabilities.
- **Multithreading:** Directly matched with SonarQube's *multi-threading* category, dealing with concurrency-related issues (i.e., synchronization, thread safety, etc.).

- **Performance:** Mapped from SonarQube's *performance* category, highlighting areas where code efficiency could be improved.
- **Documentation:** No matching categories in SonarQube; this category remains with issues only detected by PMD. Was decided to keep this category separate, as documentation is an important aspect of code quality and can be easily overlooked.

2.2 Comparison of Results

After preprocessing the data, we were able to compare the results of the two tools. Table 1 shows the number of issues found by each tool, categorized based on the mapping described in subsection 2.1.

Category	PMD Issues	SonarQube Issues
Code Style	496	308
Best Practices	190	207
Design	154	505
Error Prone	135	702
Documentation	84	0
Multithreading	8	12
Performance	6	61
TOTAL	**1074**	**1327**

Table 1: Summary of Issues by Category for PMD and SonarQube

As illustrated in Table 1, SonarQube detected a greater number of issues overall, with particular attention in the *Design* and *Error Prone* categories. Conversely, while PMD found fewer issues in total, reported a higher count in the *Code Style* category. It is imporant to highlight that the *Documentation* category is exclusive to PMD, as SonarQube does not provide a similar category (refer to subsection 2.1 for more details). This result suggests that each tool has different strengths, with SonarQube focusing more on structural and potential bug-related issues and PMD emphasizing code readability and styling standards.

This difference in focus highlights the complementary nature of these tools. Using both tools together can yield a more comprehensive analysis, combining PMD's focus on code style and best practices with SonarQube's broader approach to design, performance, and error-prone code.

It is important to note that PMD was run using the `quickstart` rule set (refer to section 1.5), which puts particular attention on code style and the detection of best practices, while SonarQube's analysis was conducted with its default comprehensive rule set.

As already discussed, this choice was made to ensure a balanced comparison, as the default rule sets reflect typical configurations that many developers would use. These results suggest that SonarQube's default rule set is more comprehensive than PMD's `quickstart` rule set, providing a richer set of issues across different categories.

2.3 Project Quality Assessment

From the results in Table 1, it is evident that the codebase has a significant number of issues, particularly in categories associated with potential bugs, security vulnerabilities, and design improvements. These findings suggest areas for potential quality improvement, including enhanced error handling, adherence to best design practices, and modifications to address code style issues.

Addressing the issues highlighted by both tools would likely improve the overall quality of the project, contributing to better maintainability, readability, and potentially reducing future bug occurrences. This contradicts the initial hypothesis outlined in subsection 1.4, which suggested that the

project would have a low number of issues. The results indicate that the project could benefit from a more thorough review and refactoring to address the identified issues.

After a brief manual inspection of the issues, it was observed that some of the issues detected by the tools are minor and may be false positives. These issues may not necessarily indicate a problem in the codebase but are flagged due to the tools' static analysis approach. False positives can be a common occurrence in static analysis tools, especially when dealing with complex codebases or specific programming patterns that the tools may not fully understand.

SonarQube provides a scoring system that assesses the overall quality of the project based on the number and severity of issues detected per category. The project quality score is calculated from the number of issues in each category and their severity, providing a quantitative measure of the project's quality. Even though the project quality score is not the focus of this analysis, it can be a useful metric for evaluating the overall health of the codebase. subsection 2.3 shows the project quality score for the codebase.

Property name	**Score**
Reliability	C
Security	A
Maintainability	A

Table 2: SonarQube project quality score - A is the highest score, F is the lowest

From the project quality score in subsection 2.3, it can be seen that, although the project has a significant number of issues, it still scores highly in the *Security* and *Maintainability* categories. Research shows that SonarQube is known to report many issues that are not necessarily problems in the codebase, which may explain the high number of issues detected alongside a high score in the *Security* and *Maintainability* categories. This could also contribute to the high number of issues found in the *Error Prone* category in Table 1.

2.3.1 False Positives

SonarQube identified 20 possible bugs in the project, explaining the lower score in the *Reliability* category in subsection 2.3. This is a serious issue that should be addressed promptly, as it may lead to potential bugs and security vulnerabilities in the codebase. A manual inspection of each reported bug revealed that all of them arise from two common patterns in the codebase. The following listing details the results of the manual inspection:

1. *Use "remove()" instead of "set(null)"* - This issue is reported every time a method sets a variable to `null` rather than using the `remove` method (when available). For this reason, every call to `threadLocal.set(null)` is reported as a bug. The following listing shows an example code snippet that generates this issue:

```
if (threadLocal.get() != null) {
    // BUG - Use "remove()" instead of "set(null)".
    threadLocal.set(null);
    threadLocal.remove();
}
```

   This instance is a false positive, as every `set(null)` operation is followed by a `remove()` call for added safety. This ensures that the pointer is correctly dereferenced and the variable removed from the map. This pattern is common in the codebase, leading to numerous false positives.

2. *Catch "InterruptedException" when not expected* - This issue is reported when the code specifies a try-catch block to intercept `InterruptedException` exceptions, but the interruption is not expected. After carefully inspecting each reported bug for this particular issue, it was noticed that all instances are false positives triggered by the same operation pattern in the codebase. The following listing shows an example code snippet that generates this issue:

```
...

try {
  Thread.sleep(someValue);
} catch (InterruptedException e) {
    // BUG - Catch "InterruptedException" when not expected.
    ...
}

...
```

   This bug is a false positive, as `Thread.sleep()` is known to throw an `InterruptedException` when the thread is interrupted. Hence, the reported bug is a false positive, as the exception is expected and handled correctly.

Hence, it can be that the project does not contain any real bug, as all of the reported bugs are false positives. This is a common issue with static analysis tools, as they may not fully understand the context or specific patterns in the codebase, leading to false positives.

This emphasizes the importance of manual inspection and verification of the reported issues to confirm the validity and of reported issues and determine if they require attention. Due to the large number of issues reported, manually inspecting each one is not feasible. However, by identifying common patterns that generate false positives, it is possible to filter out these issues and focus on the real problems in the codebase.

On the other hand, by manually sampling and analyzing the issues reported by PMD, it was observed that all of them are true positive instances. Since the chosen rule set focuses on code style and best practices, properties that are simpler to detect than complex bugs or design issues, problems detected by this tool are less likely to be false positives. This would explain the absence of false positives in the manual inspection of PMD issues.

References

[1] Apache Camel. *What is Camel?* Accessed: 31.10.2024. Apache Software Foundation. URL: https://camel.apache.org/manual/faq/what-is-camel.html.

[2] Netflix. *Hystrix*. Accessed: 01.11.2024. URL: https://github.com/Netflix/Hystrix?tab=readme-ov-file#hystrix-status.