Evaluating the design of an open-source project

Software Design and Modelling, Università della Svizzera Italiana

Luca Di Bello

Sunday 3rd November, 2024

1 Introduction

In this assignment we are asked to leverage multiple specialized tools to scan a chosen open-source project in order to automatically detect design flaws such as code smells and anti-patterns. The results will be condensed inside a single report in order to provide a detailed analysis of the project's codebase, and to identify potential areas for improvement in terms of code quality and maintainability.

To address this task, we are going to use SonarQube and PMD static code analysis tools, which are widely used in the industry to detect a wide range of coding flaws and bad practices:

Similarly to the previous assignment, the chosen open-source project must satisfy 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

However, as the Github search feature does not provide any filtering options for the number of open issues or the 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

In order to provide a interesting analysis of a project, and also to learn more about 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). 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 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 defines its own testing suite to ensure the correctness of the implementation.

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 can be used to build the project:

```
./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 to find a well-structured codebase with a good level of maintainability and readability. The project is actively maintained and receives regular updates, which should ensure that the codebase is up-to-date and follows the latest best practices in Java development.

However, due to the size of the codebase we still expect to find some minor issues, but is still expected to find an high-quality codebase. This theory is supported by the fact that the project has a large number of contributors, and each pull request apart from being reviewed by the maintainers, is also automatically tested using GitHub actions. In fact, by the configured GitHub actions, is possible to see that for some PRs the project employes 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 here.

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 usage of the scanning tools is straightforward, as both SonarQube and PMD provide a command-line interface to analyze the codebase of a project. In the following paragraphs, we will provide an overview of how to use each tool to analyze the *resilience4j* project.

Since we are going to use two scanning tools and combine their results in a single report, we are going to use simple rulesets for each tool to avoid having too many false positives and to ensure a consistent analysis.

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

To perform a comprehensive scan of the whole codebase, was decided to use the `quickstart` ruleset, as it include a set of rules that are considered to be the most important and useful for a general-purpose analysis. This ruleset represents a good starting point to detect common coding flaws and anti-patterns, 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. The latter is necessary to store the results of the analysis and to provide 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:

```
/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

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.

As the tool does does not provide any way to export the results in a CSV format, offering only a web interface to visualize the detected issues, the different categories of issues found by SonarQube and the relative counts were manually and stored in a CSV file to allow a comparison with the PMD results in a programmatic way. 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 an 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. As the two sets of issues are not directly comparable, was developed a Python script that reads the output of both tools and extracts 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 of the issues, with only 7 categories. As such, we decided to map the SonarQube categories to the PMD categories, in order to provide a more general overview of the issues found by both tools. The mapping is as follows:

- **Code Style:** Included SonarQube's *convention*, *unused*, *confusing*, *clumsy*, *obsolete*, *duplicate*, and *redundant*, *brain-overload* categories. Issues in this category are related to the code's readability and maintainability, and the misuse of Java features.
- **Best Practices:** Mapped from SonarQube's *java8*, *bad-practice*, and *serialization* categories. These issues are related to the code's quality, and the misuse of Java features.
- **Design:** Included SonarQube's *design* and *brain-overload* categories, focused on architectural and design quality. The *brain-overload* category was included in this category as it relates to the complexity of the code, alerting when a class or method is considered too complex (cyclomatic complexity, nesting, etc.).

  Furthermore, was later decided to add map *brain-overload* category also to the *Code Style* category, as it strictly 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.
- **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.

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 by the mapping described in Section 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 is possible to see, SonarQube generally found more issues than PMD, expecially in the *Design* and *Error Prone* categories, while PMD found more issues in the *Code Style* category. The *Documentation* category is exclusive to PMD, as SonarQube does not provide a similar category. This result hinting that each tool has different strengths and weaknesses, and that the combination of both tools can provide a more comprehensive analysis of the codebase.

It is important to highlight that PMD analysis has been executed using the `quickstart` rule set (refer to section 1.5), which is more focused on code style and best practices, while SonarQube uses a more comprehensive set of default rules. This could explain the higher number of issues found by SonarQube, as it has a more detailed analysis of the codebase. This was decided to provide a more fair comparison between the two tools, as the default rule sets are the most commonly used by developers.

2.3 Project quality

From the results in Table 1, it is possible to see that the codebase has a large number of issues, especially. This indicates that the codebase has a number of potential bugs, security vulnerabilities, and design issues that could be addressed to improve the overall quality of the project.

2.4 Analysis quality and coverage

2.5 False positives

2.6 False negatives

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.