Università della Svizzera Italiana

Evaluating the Design of a Project

Pablo Landrove Pérez-Gorgoroso

November 26, 2025

1 Project selection

Apart from the original constraints of minimum number of stars, forks, etc... We would like to be able to use both PMD and SonarQube for the analysis, so we will aim to find a project which can be easily built. PMD only requires the source files to run the analysis, but SonarQube requires the compiled classes, for this reason we need to build the project first before performing the analysis.

We part from the projects we saw for the last assignment, those are:

- `Guava`: open-source Java library that with utilities to make Java development easier.
- `Arduino`: Hardware and software platform designed for building electronic projects.
- `Hadoop`: Framework for distributed storage and processing of large datasets.
- `Kafka`: Distributed event streaming platform used for real-time data pipelines.
- `Termux`: Terminal emulator and Linux environment app for Android.

Among the projects considered, the only ones that could be built easily without importing numberus dependencies were `Kafka` and `Guava`. However, since we were unable to run the SonarQube analysis on `Kafka`, we focused our study on `Guava` which allows us to use both tools.

2 Modules Analyzed

Guava is comprised of 5 modules:

- Core: Contains the common utilities, collections, primitives, hashing, I/O, etc.
- Android: Core library tailored for Android
- GWT (Google Web Development Kit)
- Tests: JUnit tests to ensure the correctness of the core library.
- Testlib: A set of Java classes for more convenient unit testing.

Because Android and GWT are adaptations of the core module, they have a high amount of replicated code, so we will skip them. We decided to analyze the Core module, because it contains the library implementations.

3 Rule selection

3.1 PMD

PMD has multiple sets of predefined rules to do the analysis. This sets are:

- Best practices: Rules which enforce generally accepted best practices.
- Codestyle: Rules which enforce a specific coding style.
- Design: Rules that help discover design issues.
- Documentation: Rules that are related to code documentation.
- Error prone: Rules that detect constructs that are broken or prone to runtime errors.
- Multithreading: Rules that flag issues when dealing with multiple threads.
- Performance: Rules that flag suboptimal code.
- Security: Rules that flag potential security flaws.

For our analysis, we decided to discard the **Security** rules, as they focus on issues such as insecure encryption or hardcoded keys, which are unrelated to Guava's functionality. The **Documentation** rules were also omitted, since our focus lies on the project's design. We excluded **Code Style** rules because many of them are subjective, for example, enforcing specific naming conventions or limits on constructors, which isn't relevant for this analysis.

Multithreading rules were skipped as well, since evaluating concurrency correctness falls outside the scope of this analysis. We also excluded the **Performance** and **Best Practices** rules, as they mainly target coding conventions rather than structural design issues. Finally, we chose to skip the **Error Prone** ruleset, as it primarily detects beginner-level mistakes that are unlikely to appear in this library.

In conclusion, since the main objective of this analysis is to identify potential design flaws in the Guava project, we selected the **Design**.

The **Design** rules are central to this goal, as they detect issues such as excessive coupling, large or overly complex classes, and violations of design principles that could affect maintainability or extensibility.

3.2 SonarQube

SonarQube distinguishes between **Bugs** (code which may produce runtime errors), **Vulnerabilities** (security issues that may be exploited by an attacker), **Code Smells** (design issues that make the code harder to maintain) and **Security Hotspots** (code sections that require manual review). For the java default quality profile there are 150 bugs, 33 vulnerabilities, 37 security hotspots and 408 code smells.

After a quick overview of the bugs, vulnerabilities and hotspots we can see that they fall out of the scope, so we will stick only to the code smells. In the code smells there are many subcategories, the ones that seem more relevant to this assignment are:

- Design: Detects structural or architectural issues that affect maintainability, readability, and extensibility.
- Brain-overload: Detects code that is mentally hard to understand due to high complexity.

- Confusing: Detects code that is ambiguous or misleading, making it easy to misinterpret.

We can create a profile that tracks only these three types of code smells.

4 Quantitative summary

4.1 PMD

After running the analysis with `PMD` with the mentioned rule set over the core module we find 1.384 flaws of 31 different types, which we can see in Table 4.1.

Flaw	Count
LawOfDemeter	463
TooManyMethods	289
CyclomaticComplexity	110
CouplingBetweenObjects	83
CognitiveComplexity	83
GodClass	59
AvoidDeeplyNestedIfStmts	53
SignatureDeclareThrowsException	37
AvoidThrowingNullPointerException	30
ExcessiveImports	30
ExcessiveParameterList	30
UselessOverridingMethod	26
SimplifyBooleanReturns	21
AvoidThrowingRawExceptionTypes	15
AbstractClassWithoutAnyMethod	8
NPathComplexity	8
ClassWithOnlyPrivateConstructorsShouldBeFinal	7
DataClass	6
CollapsibleIfStatements	6
ExcessivePublicCount	5
NcssCount	4
SwitchDensity	2
MutableStaticState	1
TooManyFields	1
AvoidThrowingNewInstanceOfSameException	1
AvoidUncheckedExceptionsInSignatures	1
LogicInversion	1
UseUtilityClass	1
ExceptionAsFlowControl	1
DoNotExtendJavaLangError	1

Table 4.1: Flaws found by PMD.

4.2 SonarQube

- Design smells (91 total): These indicate structural or architectural problems that can make the code harder to maintain or extend. Of these, 29 were classified as **critical** and 62 as **major**, suggesting several classes exhibit high complexity, excessive coupling, or poor cohesion.
- Brain-overload issues (130 total): This category captures code that is mentally difficult to understand due to complexity, deep nesting, or long methods. Among these, 69 were **critical** and 46 **major**, pointing to sections of the code that require significant cognitive effort to read and comprehend.
- Confusing code smells (90 total): These highlight code that is ambiguous, misleading, or error-prone due to naming, duplicated logic, or unexpected side effects. The analysis found 17 **blocker**, 3 **critical**, 44 **major**, and 26 **minor** issues, indicating that several areas of the code could lead to misunderstandings or mistakes during maintenance.

Overall, these results provide a quantitative overview of the codebase's maintainability, complexity, and clarity, and can help prioritize areas for refactoring.

The most relevant rule violations are summarized in Table 4.2. Notably, Cognitive Complexity appears most frequently, highlighting methods that are difficult to understand and may benefit from decomposition. Other common issues include deep inheritance trees, methods with too many arguments, and private methods accessed only by inner classes, all of which point to potential design or structural problems. Duplicate method implementations, although less frequent, can also affect maintainability and indicate opportunities for refactoring.

Flaw	Count
Cognitive Complexity	69
Inheritance Tree too deep	48
Method with too many arguments	41
Private methods called only by inner Class	21
Methods with duplicate implementations	4

Table 4.2: Flaws found by PMD.

5 Qualitative discussion

5.1 PMD

The most software flaws that were found were vilations of **Law of Demeter**, **high coupling** and **God Classes**

5.1.1 Law of Demeter

The **Law of Demeter** is a design principle that states that a method should only communicate with its immediate collaborators, its own fields, method parameters, or directly created objects. A violation occurs when code chains multiple method calls across different objects, indicating that the class depends on the internal structure of other objects.

PMD detects such violations by analyzing call chains and identifying cases where methods access members of objects returned by other methods, revealing excessive coupling and a breach of encapsulation.

5.1.2 High Coupling

High coupling occurs when a class depends on too many other classes, making the system harder to maintain and modify. Highly coupled classes are more fragile, as changes in one part of the code can easily propagate to others.

PMD detects this issue by measuring the number of unique external classes referenced or used within a given class. A high number of dependencies indicates strong coupling, suggesting that the class may be taking on too many responsibilities or lacking proper abstraction.

5.1.3 God Class

PMD flags a class as a **God Class** when it exhibits a combination of high WMC, high ATFD, and low TCC values. These metrics are defined as follows:

- WMC (**Weighted Method Count**): Measures the overall complexity of a class by summing the individual complexities of its methods.
- ATFD (**Access to Foreign Data**): Quantifies how much the class depends on or accesses data from other classes.
- TCC (**Tight Class Cohesion**): Indicates how closely related the methods of a class are, based on shared attribute usage.

These design flaws can be identified by measuring method dependencies or class complexity, at which PMD is very effective. However, other issues, such as poorly implemented patterns, Poltergeist classes, or Shotgun Surgery are more contextual and depend on design intent. These require human judgment and architectural understanding rather than purely metric-based detection.

5.2 SonarQube

While SonarQube's graphical interface makes it much easier to filter and explore the different categories of flaws and their associated labels, we observe that, similar to PMD, many of these labels are primarily metric-based. They rely on mechanical counts such as the number of methods, number of parent classes, or cognitive complexity, rather than capturing deeper contextual or architectural issues.

From a qualitative perspective, the most prominent issues identified include:

- Cognitive Complexity: Methods flagged under this category are often long and deeply nested, making them difficult to understand and maintain. Refactoring these methods into smaller, more focused units would improve readability and reduce cognitive load for developers.
- Inheritance Tree Too Deep: Classes that inherit from many levels of parent classes can be harder to understand and modify, as behavior may be spread across multiple layers. This can indicate overly complex hierarchies and a need to simplify or flatten inheritance.
- Methods with Too Many Arguments: Methods with a large number of parameters are harder to use and test, and may indicate that the method is trying to perform multiple responsibilities. Splitting such methods can modularity.

- Private Methods Called Only by Inner Classes: These situations suggest tight coupling between the outer and inner classes. While sometimes unavoidable, this pattern can make the code harder to maintain and may indicate a need to rethink class responsibilities.
- Duplicate Implementations: Duplicate methods increase maintenance overhead and the risk of inconsistencies. Consolidating duplicate code can reduce redundancy and improve maintainability.

6 False positives

In our analysis, no false positives were detected. This is unsurprising, as the rules used by both `PMD` and `SonarQube` are largely static and metric-based. Each reported issue corresponds to a measurable property of the code, such as high method count, excessive coupling, or cognitive complexity, making it very unlikely for the tools to flag incorrect or irrelevant violations.

7 Comparison of the Tools

In our analysis, `PMD` identified a larger number of flaws compared to SonarQube. One reason for this is that PMD includes several rules that capture more nuanced design issues, such as God Classes, violations of the Law of Demeter, and high coupling between objects. These rules go beyond simple counts of methods or arguments, providing deeper insight into the structure and maintainability of the code.

`SonarQube`, on the other hand, categorizes issues into clear types like code smells, bugs, and vulnerabilities. While many of its rules are metric-based, such as counting the number of method parameters or inheritance depth, they are somewhat less sophisticated in detecting higher-level design problems compared to PMD.

8 Project Assessment

Based on the analysis with both `PMD` and `SonarQube`, we observe that Guava contains a noticeable number of `God Classes` with a large number of methods. These classes often centralize a significant portion of the library's functionality, resulting in high complexity, extensive coupling, and cognitive load.

However, this pattern is not necessarily a flaw in the context of Guava. As a library designed to provide robust data structures and utility tools, it is expected that certain classes serve as comprehensive, feature-rich components. The high method count and complexity reflect the library's goal of offering a wide range of functionality in a single, cohesive API, rather than poor design practices.

Overall, while the tools highlight areas that would be concerning in typical application code, in the context of a mature, well-maintained library like Guava, these design characteristics are largely justified. The results provide useful insights into maintainability and potential refactoring points, but they should be interpreted with the understanding of the library's intended purpose and scope.