Università della Svizzera Italiana

Documenting Design Patterns

Pablo Landrove Pérez-Gorgoroso

November 7, 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.

From those projects, the only ones that could be built easily without importing many dependencies were `Kafka` and `Guava`. In the other assignment we discarded `Kafka` because the pattern detection tool failed to detect any patterns, but it may prove useful for this task.

Kafka doesn't wark with SonarQube.

After using PMD with `Guava` we obtain a very big `html` file with a table of approximately 79.000 rows, each corresponding to a specific line in a .java file.

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

With PMD we are able to get a .txt result in which each line corresponds to a specific occurrence of a flaw. It is easy to implement a Python script which counts how many occurrences were found for each of the rules. We find a total of 12.689 flaws in all of the project out of 36 types of flaws.

This amounts to 63% of the total flaws. This is partly due because we are running the analysis over all the Guava modules.

After reapeating the analysis with the same rules only over just the Core library we find 1.384 flaws of 31 different types, which we can see in Table 4.1.

4.2 SonarQube

5 False positives

6 Qualitative discussion

6.1 PMD

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

6.1.1 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.

6.1.2 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.

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
	1

Table 4.1: Flaws found by PMD.

`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.

6.1.3 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.

7 Comparison of the tools

8 Project assesment