Università della Svizzera Italiana

Documenting Design Patterns

Pablo Landrove Pérez-Gorgoroso

November 6, 2025

1. What to do

- Pdf report of at most 7 pages (including images).
- X The project selection process, (projects you discarded); choose a project, with a variety of features that can be analyzed with a flaw detection tool.
- X Short description of the project with stats (size, number of contributors, whether it's a library or application);
- Wether you analyzed all project or only some parts of it (e.g., did you analyze testing code as well?);
- Discussion of what predefined rules you selected for your project analysis and why;
- Quantitative summary of the tools' output, such as a table with how many problems have been found, of which kind, by what tool, and so on;
- Whether you found any false positives in the tools' output, that is reports of rule violations that should not actually be considered violations;
- Qualitative discussion of the kinds of problems the tools are more or less likely to report;
- Comparison of the tools' output at a high level (if you used multiple tools), or of the tool's output with different options (if you used a single tool)
- Assessment of the project's design quality based on the tools' analysis combined with your own judgement; if possible, try to relate your findings to specific characteristics of your project (e.g., it's a library, or it depends on legacy projects, ...).

2. 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.

3. PMD

From the output of PMD we found many reports on badly implemented Unit Tests, but these were ignored, since we are looking for design flaws in the implementation of the library and the tests fall out of the scope.

3.1. Rules

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 and implementation** rather than its comment quality. We excluded **Code Style** rules because many of them are subjective—for example, enforcing specific naming conventions or limits on constructors—and Guava already follows a consistent, well-established style.

Most **Multithreading** rules were skipped as well, since evaluating concurrency correctness falls outside the scope of this analysis. However, a few specific checks, such as those detecting incorrect Singleton implementations, may still be of interest. Finally, we chose to skip the **Error Prone** ruleset, as it primarily detects beginner-level mistakes that are unlikely to appear in such a **mature and well-maintained** library.

In conclusion, since the main objective of this analysis is to identify potential **design flaws** in the Guava project, we selected the **Design**, **Best Practices**, and **Performance** rulesets, along with a few carefully chosen **Error Prone** rules.

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. The **Best Practices** rules complement this by ensuring that the implementation remains clean and consistent, which helps avoid structural problems caused by poor coding habits. The **Performance** rules contribute to the analysis by revealing inefficient design choices that may impact the quality of the system's architecture.

To obtain the PMD analysis following this rulesets, we create a file named `ruleset.xml` that we include in the source code directory of guava and run the analysis. 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 81.088 flaws in all of the project out of 96 types of flaws.

What stands out the most is the huge number of flaws related to the use of JUnit, in particular we have the following flaws:

Pattern	Count
UnitTestShouldUseTestAnnotation	34862
UnitTestContainsTooManyAsserts	12504
UnitTestShouldIncludeAssert	3402
JUnit4SuitesShouldUseSuiteAnnotation	346
JUnitUseExpected	105
Total	51219

Table 3.1: Design Patterns reported by Pattern4J.

This amounts to 63% of the total flaws. This is partly due because we are running the analysis over all the Guava modules. Let's see what each of the modules do:

- 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 we mainly want to analyze the implementation of the library we will choose the Core module, we don't use Android or GWT because it mainly duplicates most of the classes changing some implementation details.

After reapeating the analysis with the same rules only over the Core library we find 1.524 flaws of 59 different types, but most of the flaws are not relevant for the current analysis, such as `AddEmptyString`, `SystemPrintln`, `AvoidUsingHardcodedIP`. So in the end we decided to also remove the **Best Practice** and **Performance** rulesets and just use the **Design**, after which we obtain the following flaws:

3.2. HashBiMap

This Class can be found in the following route: guava/guava-gwt/target/classes/com/google/common/collect/

This Class implements the interface BiMap, which is a bidirectional map, that allows to obtain both a value from its key and a key from its assigned value.

Pattern	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 3.2: Design Patterns reported by Pattern4J.

`PMD` identified several design issues in this class:

- Line 25: High coupling within the class.
- Possible *God Class* (WMC = 70, ATFD = 93, TCC = 11.111%).
- Excessive number of methods.

3.2.1. 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.

In summary, a *God Class* is typically large and complex, interacts heavily with external classes, and performs multiple unrelated responsibilities. Such characteristics make it difficult to understand, test, and maintain. In this case, the purpose of the class is not immediately clear from the code structure.

The `HashBiMap` class contains 70 methods, which is an unusually high number for a single class. This suggests that the class may be handling several different responsibilities that could be better separated into smaller, more cohesive units.

Furthermore, `PMD` has also flagged some methods in this class as having excessive cognitive complexity. This cognitive complexity is a term that was first introduced by Sonar, it takes into account metrics such as the number of times the linear flow of control is broken and nesting of flow-breaking structures. Let's look at the implementation of one of this methods (`delete(BiEntry<K, V> entry)`).

This method deletes an element from the table. This data structure is implemented as two hash tables that map key to value and value to key. In each entry of the hash table we have a bucket for all the elements which key map to that bucket. This bucket is implemented as a linked list, which we need to iterate to find the element to remove.

Because the structure of the two hash tables is similar, the code in the method is deleted to handle both deletions. This already is **code clone** which should be made into its own method.

In the first part of the method, the implementation iterates over the elements of the bucket until finding the one we are looking and deletes it from the list.

The method iterates over the linked list using a for loop which stopping condition is never met, instead, we get out of the loop by using a break instruction. This adds unnecessary cognitive complexity and could be simplified by using a while loop to iterate the element in the list and the do the deletion after the loop.

Code 1: Implementation of `delete(BiEntry<K, V> entry)`

```
for (BiEntry<K, V> bucketEntry = hashTableKToV[keyBucket];
  true;
  bucketEntry = bucketEntry.nextInKToVBucket) {
  if (bucketEntry == entry) {
    if (prevBucketEntry == null) {
      hashTableKToV[keyBucket] = entry.nextInKToVBucket;
    } else {
      prevBucketEntry.nextInKToVBucket = entry.nextInKToVBucket;
    }
    break;
  }
  prevBucketEntry = bucketEntry;
}
```

3.3. guava/android/guava/src/com/google/common/cache/LocalCache.java

This class represents another example of a potential *God Class*. It also defines a large number of methods and exhibits a high degree of coupling with other components in the project. Such design patterns indicate that the class may be performing too many roles at once, reducing modularity and increasing maintenance difficulty.

4. SonarQube

A. Code Listings