

Evaluating the Design of an Open-Source Project

Authors: Lombardo Federico, Occhipinti Gianlorenzo

Project (GitHub): Open-Refine

Preamble

This report is about the second assignment of Software Design & Modeling course. Our goal is to detect flaws and anti-patterns found in a Java project selected on GitHub and discuss the different types of flaws, the ones that can be avoided and discuss how they can affect the overall quality of the project.

Anti-patterns (also known as design smells) are features of a software design that violate design principles and thus lower the quality of the code.

Anti-patterns are divided into five categories: Complexity, Modular, Abstraction, Inheritance and Naming. As Java is an object-oriented programming language, Anti-patterns will typically show flaws in relationships and interactions between classes or objects. However, not all the anti-patterns will fit with every programming paradigm we will discuss anti-patterns that are Java specific.

To find some exploitable projects on GitHub we focused on 5 important metrics:

- Programming Language
- Stars
- Forks
- Open issues
- Lines of code.

The project was selected by querying GitHub API, limiting the project to the ones written in Java, with at least 100 stars, 100 forks and 100 open issues. Then we cloned the project and used "cloc" (<https://github.com/AlDanial/cloc>) to retrieve the physical line of Java source code.

In this assignment, we use PMD, a source code analyzer that finds common programming flaws like unused variables, empty catch blocks, and unnecessary object creation. Another tool that we are using is SonarQube, an automatic code review tool to detect bugs, vulnerabilities, and code smells.

Selection of the project

During the selection process, we discard all the projects that are tutorial/interview repositories (like <https://github.com/iluwatar/java-design-patterns> or <https://github.com/doocs/advanced-java>) since we believe that the chances of finding the desired amount of code flaws are thinner and most importantly they are neither a library nor an application.

We are discarding the android applications (<https://github.com/PhilJay/MPAndroidChart>) because the structure is too specific and we wanted to focus on general-purpose software.

We got through several interesting projects, the first one is Spark: a web framework built for Java 8 (<https://github.com/perwendel/spark-kotlin>). Unfortunately, even if PMD detects an interesting amount of flaws, the number of lines of code is not sufficient, so we are discarding it.

Another exciting project is ElasticSearch, a distributed search and analytics engine that allows index, store, and retrieve data in real-time. It supports several kinds of data (structured or unstructured text, numerical data, or geospatial data) in a scalable distributed system.

We discarded this project for the extreme complexity and the size of the project that counts 16 thousand classes and 2 million lines of code with a vast amount of pattern.

Open Refine

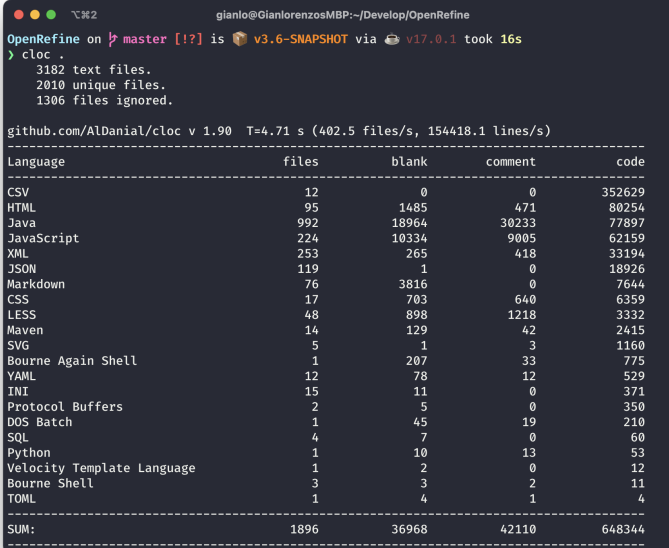
OpenRefine is an open-source desktop application for data cleanup and allows you to work with several formats, such as CSV, XML and JSON.

It is a powerful tool for working with messy data, it permits you to explore data, clean it, transform it from one format into another and even parse it from a website.

Project Metrics

The number of contributors is 206, and it has 1.5k forks and 8.4k stars and 650 open issues.

Also, the number of lines of code surpassed the threshold we defined, and the PMD detected more than 900 flaws, so we decided to analyze OpenRefine.



```
OpenRefine on master [!?] is v3.6-SNAPSHOT via v17.0.1 took 16s
> cloc .
  3182 text files.
  2010 unique files.
  1306 files ignored.

github.com/Aldanial/cloc v 1.90 T=4.71 s (402.5 files/s, 154418.1 lines/s)
-----
Language      files      blank      comment      code
-----
CSV             12           0           0       352629
HTML            95        1485         471       80254
Java           992       18964       30233       77897
JavaScript      224       10334        9005       62159
XML             253         265         418       33194
JSON            119          1           0       18926
Markdown         76        3816           0        7644
CSS              17          703         640        6359
LESS             48          898       1218        3332
Maven            14          129          42        2415
SVG               5           1           3        1160
Bourne Again Shell 1         207          33         775
YAML             12           78          12         529
INI              15           11           0         371
Protocol Buffers   2           5           0         350
DOS Batch         1           45          19         210
SQL               4           7           0          60
Python            1          10          13          53
Velocity Template Language 1           2           0          12
Bourne Shell       3           3           2          11
TOML              1           4           1           4
-----
SUM:           1896       36968       42110      648344
```

- CLOC/ELOC = 0.388
- The cyclomatic complexity of the src dir is 9.378
- Tests are on 400 classes with a coverage of 51%

These metrics suggest that the program is relatively complex and inter-winded, but it seems to be also adequately tested. Furthermore, the ratio of comments to lines of code indicates that the code is well commented. At the same time, the Cyclomatic Metric clarifies that we are in the presence of low complexity, structured and well-written classes with low effort to maintain it.

PMD

PMD is a source code analyzer that supports Java, JavaScript. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth.

PMD provides several rulesets for Java; the basic ones are `java-basic`, `java-design`, `java-unusedcode`; we decided to also use `java-codesize` to identify classes with complexity and size-related problems and `java-naming` to detect code smell like "Long Method", "Good class", etc.

Ruleset description

The rulesets that we choose are:

- `java-basic`: the Basic ruleset contains a collection of good practices which should be followed.
- `java-design`: the Design ruleset contains rules that flag suboptimal code implementations. Alternate approaches are suggested.
- `java-unusedcode`: the Unused Code ruleset contains rules that find unused or ineffective code.
- `java-codesize`: the Code Size ruleset contains rules that find problems related to code size or complexity.
- `java-naming`: has rules regarding preferred usage of names and identifiers.

Flaws

We ran the PMD analysis, and we obtained the following results

Open refine PMD analysis overview

 Name	 Tags
<u># Code Style</u>	2269
<u># of Design</u>	968
<u># of Best Practices</u>	152
<u># of Error Prone</u>	84
<u># of Multithreading</u>	26
<u># Documentation</u>	16
<u># of Performance</u>	3

PMD discovered a total of 3518 issues, which we think is a good value considering the project's total size (77897). Furthermore, if we compute the issues - LOC ratio, we obtain a value of 0,04, which is a good result considering that half of the issues are "Code Style", and thus, they probably aren't code smells.

We provided some flaws that we consider interesting and involves classes (that needs future refactoring):

PMD detected `main/src/com/google/refine/exporters/TabularSerializer.java` as an anaemic domain model; as we can see, this class has only attributes without methods.

```
static public class CellData {
    final public String columnName;
    final public Object value;
    final public String text;
    final public String link;
    public CellData(String columnName, Object value, String text, String link) {
```

```

        this.columnName = columnName;
        this.value = value;
        this.text = text;
        this.link = link;
    }
}

```

PMD detected **main/src/com/google/refine/importing/ImportingUtilities.java** as a possible god class; in fact, it has a *WMC* (the sum of complexities of all methods declared in a class) equals 160 and *ATFD* (number of external classes from which a given class accesses attributes) equals 168, and *LOC* (lines of code) equals to 1072.

Analyzing the class, PMD spotted several issues regarding the size of the methods for example `retrieveContentFromPostRequest` has 227 lines of code and 5 parameters and therefore surpass the threshold we defined.

A class with similar problems is **main/src/com/google/refine/expr/util/CalendarParser.java** with *WMC* = 301, *ATFD* = 321 and methods like `parseNumericToken` with 280 lines of code.

In total, we found 18 possible God Classes.

Sonarqube analysis

Sonarqube is an automatic code review tool to detect bugs, vulnerabilities, and code smells in our code. The project analysis can be automatized thanks to some continuous integration tools that runs after an event (such as a commit, pull request) happen. The SonarQube analysis process generates an interactive report that shows the issue on your code. Issues are generated by rules: coding standards or practice which, if they have not complied leads to bugs, vulnerabilities, security hotspots and code smells. Rules can check the quality of the code and they can be used to get an estimate of the technical debt of a project. In SonarQube, there are mainly four kinds of rules:

- **Code Smell** (Maintainability domain)
- **Bug** (Reliability domain)
- **Vulnerability** (Security domain), a security-related issue that represents a backdoor for attackers.
- **Security Hotspot** (Security domain), security-sensitive pieces of code that need to be manually reviewed.

Each rule should provide a description with some compliant and non-compliant examples that help the devs to fix and understand the problem, some tags in order to help discover them more easily and a severity. SonarQube defines 5 levels of severity and all of them depends upon the **impact** of the rule and the **likelihood** that the worst-case described by the rule can happen. Those levels are:

- **Blocker**: strong impact and high likelihood.
- **Critical**: strong impact and low likelihood.
- **Major**: weak impact and high likelihood.
- **Minor**: weak impact and low likelihood.
- **Info**: any impact at all.

Rules contribute to the quality level of a project. SonarQube marks all projects with a score depending upon some quality gates (es. no new blocker issue, code coverage higher than 80%, etc)

In order to reach the goal of this paper, we set up a SonarQube community server and we run the analysis with sonar-scanner. For the purpose of generating the most exhaustive report, we analyzed the

entire directory of OpenRefine including compiled classes (needed by the tool) and tests. We choose to include tests in our analysis because we saw that SonarQube provides some rules that are specific for tests (an example can be found in [this link](#)).

Sonarqube predefined rules

In SonarQube you can group analysis rules in profiles. Our preliminary analysis was configured with the default SonarQube profile which contains:



- **Code Smell:** 262 rules
- **Bug:** 143 rules
- **Vulnerability:** 27 rules
- **Security Hotspot:** 29 rules

for a total of 461 active rules. We saw that 162 more rules can be activated, but those rules seem to be very specific for some edge cases. So for now we preferred to analyze the project with the default set of rules that is described by SonarQube as a "good starting point for every project".

SonarQube report



The analysis ran on OpenRefine reported some very interesting results:

Copy of Open refine analysis overview

 Name	 Value
# of Bugs	1.1K
# of Code smells	7.7K
# of Security Hotspot	323
# of Vulnerabilities	2
Percentage of duplication	9.6%
Technical Debt	199 days

it's impressive to see that a project has so many code smells, it seems to be impossible to refactor all of them and probably most of them aren't a problem at all. A more interesting overview can be done inspecting the severity of the rules

Copy of Open refine analysis rule severity overview

 Name	 Value
# of Blocker	65
# of Critical	971
# of Major	3.8k
# of Minor	3.7k
# of Info	187

These metrics suggest that the program is huge and has lots of problems, however, we have to remember that we have analyzed the entire system, including also the tests. *To break it down*, those metrics are polluted by tests, which of course must follow some design rules, but they are usually written without

caring too much about some aspects (like code duplication, variable name, etc...) because they are just tests! If we inspect the main module of Open Refine, we have this situation

Copy of Open refine main module report

Aa Directory	≡ LOC	# Bugs	≡ Vulnerabilities	≡ Code Smells	≡ Security Hotpot	≡ Duplication
<u>main</u>	41K	77	0	6167	47	5.2%
<u>tests</u>	38K	461	1	1548	189	27.9%

To get a precise understanding of the quality of the code we ran an analysis on the main directory (without tests) of the main module. We have done that because we think that the main module is a core part of the system.

We ran this analysis with a subset of rules of the default quality profile for Java (called SonarWay) by including only rules with these tags:

- **bad-practice rules** (<https://rules.sonarsource.com/java/tag/bad-practice/RSPEC-2077>) that are related to general java coding bad practices (e.g. enum must not be mutable or switch must have more than 3 cases)
- **brain-overloaded** (<https://rules.sonarsource.com/java/tag/brain-overload>) **rules** regarding code readability and intelligibility (e.g. method not too complex or with too many parameters, an inner class must be small).
- **design** (<https://rules.sonarsource.com/java/tag/design>) **rules** that regard the object-oriented design (e.g. inheritance classes must not be too deep, classes without public constructor should be final)

The result is a ruleset more oriented to find code smells instead of vulnerabilities or security hotspots. As result, the number of code smells found by sonar way **drops from 7.7K to 4.1K**, which seems to be more affordable than the previous result.

However, we should point out that 2.2K of the code smells are Java S109:**Magic numbers should not be used**. Magic numbers are numbers that are used directly in statements (e.g. in a for loop), and their use leads to problems during debugging.

Another frequent occurrence (161) code smells that we found are S1541:**Methods should not be too complex** which means that the Cyclomatic Complexity is higher than ten which is the defined threshold.

The last issue that we believe can be interesting for future refactoring in S138:**Methods should not have too many lines** which define the threshold to 75 lines of code, and we have 27 methods with this issue. This kind of code smell can often be spotted in a project of this size and is even more frequent in open source projects where it is hard to enforce "good design" rules.

Our custom rule

In this section, we will start with the description of the rule that we choose to implement; next, we will see how we implemented it and finally, we will see if SonarQube will find some occurrences of our rule in OpenRefine.

Feature Envy

We chose to implement the Code Smell Feature Envy because we didn't find any rule that discovers it, and we believe that it can be a design flaw that badly affects the design of a system.

With this in mind, starting from the definition of this code smell:

Feature envy occurs when a method mainly accesses another class's data instead of its own

It is immediately apparent that it affects internal qualities; it lowers internal cohesion and increases external coupling, making the software harder to modify. As other consequences, It also leads to poor encapsulation (envious method's classes are harder to reuse), and it makes the dependencies unclear; it is more difficult to understand how classes are related and work together.

In order to find this code smell, we have to look for a class with the following characteristics:

- External attribute CALLS: **high** because the method should be "envious";
- Internal/external attribute CALLS: **low** to be "envious" it must mainly use external methods rather than its.
- Fan out: **low** because it has to be "envious" of a small number of other classes;
- Static methods: **overused**

Given that, we have to define a quantifier for "low", "high" and "overused"; we established the following thresholds:

- External attribute CALLS: $external \geq 4$
- Internal/external attribute CALLS: $\frac{internal}{external} \leq 0.3$
- Fan out: $FO \leq 4$
- Static methods: $\frac{static}{nonstatic+static} \geq 0.3$

Implementing feature envy in SonarQube

To implement a new rule in SonarQube, we had to create a new SonarQube plugin that exploits the SonarQube Semantic APY to retrieve valuable information from the class source code.

We developed it following a Test-Driven Development. As an example of that, we will provide a non-compliant class that is marked as "Feature Envy" by our custom plugin:

```
Example 3
class FeatureEnvyCheck2 {
    List<Animal> explodingKitties = new ArrayList<>();

    static giveTreats(Animal cat, int treatNumber) {
        int count = 0;
        while (count < treatNumber) {
            cat.pet(); //external 1
            cat.giveTreat(); //external 2
            count++;
        }
    }

    public void addKitten(Animal cat, int countdown) {
        explodingKitties.add(cat); //internal 1
        double seconds = Math.random() * countdown; //external 3
        cat.explodeIn(seconds); //external 4
    }

    public void removeKitten(Animal cat) {
        cat.pet(); //external 5
        explodingKitties.remove(cat);
    }

    public boolean containsKitten(Animal cat) {
```

```

        return explodingKitties.contains(cat); //internal 2
    }
} //noncompliant
/** Checks
external calls: 5 //OK
int/ext = 2/5 = 0.4 //NO
fo = 2 //OK
stat/calls = 1/4 = 0.25 //NO
*/

```

As we can see, our plugin checks if all the constraints explained above are followed and, if just only one is not respected, it will mark the class as a Feature Envy.

Feature Envy in our analysis

During the previous analysis, we have included the feature envy in our custom profile yet. The results are surprising, **we found 7 Feature Envy inside Open Refine** and we inspected manually if all of those are truly an issue. Proof of this is given by the class `JSONUtilities.java` that is marked as a Feature Envy because, as it is a utility class, it provides lots of static methods that works only with method arguments. However, our plugin isn't perfect because we identified a false positive in the `Parser.java` class and the reason for that is because we didn't implement a check of the Object type. If the method argument is a String, which is an external class, we think that it should not be considered as an object because it represents a "primitive type". To summarise, we believe that our tool will detect false positives in classes in which methods use wrapping classes like Integer, Float or classes like String, which should not be considered as external objects.

Conclusion

Keeping in mind what we saw, we can clearly state that this project has good overall quality because, even if it's a massive project (500K+ LOC), it has few bugs, smells and security hotspots. Furthermore, despite the fact the repository has more than 100 contributors, the project has lots of tests with a discrete coverage of 50%, which is reasonable considering the size of the project.

If we have to choose between SonarQube and PMD for an analysis, SonarQube could be a better choice since it provides by default a simple web UI dashboard and valuable insights about the issues and how to fix them. Anyway, for a complete analysis, we have to mention that SonarQube can import the issues found by PMD. Therefore, the best thing probably is to run an analysis using both without duplicating the rules (maybe choosing the most interesting of each tool).

We have to mention that SonarQube can import the issues found by PMD so, the best thing probably is to run an analysis using both without duplicating the rules (maybe choosing the most interesting of each tool). We think that the most reasonable analysis should be done only on the main code to avoid the metrics being polluted by tests and with a custom subset of the available rules. We remained surprised when we didn't find Feature Envy present on the SonarQube rules, in fact, we think that this is an important rule because this code smell can bring to undesirable conditions, as we stated above. Therefore we created our custom Feature Envy rule, and we remained satisfied when we stated seven occurrences of that inside OpenRefine.

To conclude, we think that OpenRefine it's not a perfect project; we noticed that it has some issues that we think are common to different open source projects because on it the design rules cannot be easily

enforced. However, we have to remember that it has half a million lines of code and more than 200 contributors; despite that, it's tested and the most important thing is that **the number of the critical or major issues is significantly irrelevant compared to its code size**. In the end, we can say that they are developing a good project.