Assginment 2 – Software Design and Modelling

Claudio Maggioni Christian Squadrito

November 9, 2022

Contents

1 Project selection process

This project focuses on evaluating the design of an open source project by using automated flaw-detection tools. We have to choose a Java-based project on GitHub that follows the following requirements:

- 100 or more stars;
- 100 or more forks;
- 10 or more open issues;
- 100,000 or more lines of code.

Additionally, we added some less strict constraints that we thought would lead to a more significant analysis:

- Repository data, documentation, and comments must be written in English. Choosing a project using another language would drastically hamper our ability to understand the code, and thus our ability to detect potential false positive the ;
- The artifact the project produces must not rely on external components and have a streamlined build process, with all code stored in a single Maven/Gradle module. This improves our ability to tinker with the project and simplifies the process of running flaw detection tools, since all the sources are located under a single directory.

Finally, we decided to query GitHub manually instead of relying on its API. Although more laborious, this improves the chance of finding a project we are familiar with or that we have used in the past.

Therefore, we choose between the following repositories:

qos-ch/logback a widely used Java logging library backed by a Swiss company. We discard this project as it has less than 60,000 lines of non-blank lines of Java code;

hibernate/hibernate-orm a de-facto implementation of the Java Persistence API (JPA) and one of the most widely used Object-Relational Mapping (ORM) libraries in Java. The project satisfies all the “hard” requirements based on repository statistics. However, the project is over 1 million lines of code and divided into several Maven modules, and we feed that it would be hard to analyze in detail;

apache/commons-lang a commonly used Java utility library part of the Apache Commons library collection. This project meets both our “hard” and “soft” selection criteria, having acceptable repository statistics and a relatively straightforward single-module Maven setup.

1.1 The Apache Commons Lang Project

The Apache Commons family of libraries is an Apache Software Foundation[1] sponsored collection of Java libraries designed to complement the standard libraries of Java. The Apache Commons Lang project focuses on classes that would have fitted in the *java.lang* package if they were included with Java. According to GitHub, the project has 179 contributors as of October 30th, 2022.

All the source and test classes are contained within in the package *org.apache.commons.lang3* or in a sub-package of that package. For the sake of brevity, this prefix is omitted from now on when mentioning packages and classes in the project.

There is a lot of variety in the functionality the library classes provide, from aiding in implementing the builder pattern (package *builder*), to alternatives and improvements of concurrent primitives (package *concurrent*), utility classes for reflection (package *reflect*) and mutable implementations of boxed primitive types (package *mutable*).

We choose to analyze version 3.12.0 of the library (i.e. the code under the *git* tag *rel/commons-lang-3.12.0*) because it is the latest stable version available at the time of writing.

After verifying that the project meets the hard requirements related to GitHub (more than 2,400 stars, more than 1,400 forks, 222 open issues on the Apache Commons JIRA instance[2]), we ensured that the project had enough lines of code by using the *cloc* tool, which provided the following output shown in Figure **??**. By looking at the results we can finally assert that the project contains 146419 non-blank lines of Java code and this satisfies all the requirements.

[1]`https://apache.org/`
[2]`https://issues.apache.org/jira/browse/`, as of 2022-11-07 (ISO 8601 date)

Language	Files	Blank	Comment	Code
Java	409	15,790	60,363	86,056
HTML	22	1,015	100	13,028
Text	30	1,858	0	12,415
XML	38	434	539	4,819
Maven	1	31	37	940
JavaScript	5	21	78	698
Markdown	3	38	0	202
CSS	4	36	66	140
Velocity Template Language	1	23	31	90
Groovy	1	12	22	81
YAML	3	12	42	55
Bourne Shell	1	0	2	2
Total	518	19,270	61,280	118,526

Figure 1: Output of the *cloc* tool for the Apache Commons Lang project at tag *rel/commons-lang-3.12.0*.

2 Analysis

In this section we discuss the techniques and tools we use to find flaws and code smells in the Apache Commons Lang project. We decide to use two automated flaw detection tools:

PMD an open-source source code analyzer able to find common programming flaws and code smells[3]. It supports many languages including Java, and it analyzes the code statically by building an abstract syntax tree of the source code that can be queried and inspected by a suite of several rules. The rule definition process is quite simple, as one can define a rule by building an XPath expression matching against the generated AST, for simple and coincise definition commands.

Sonarqube an open-source platform for ensuring code quality, security and maintainability [4]. This tool decouples the code analysis logic into a separate "scanner" module, which uploads results on a "server" module implementing a web interface for easily browsing the detection output. Additionally, this tool is able to gather additional internal quality metrics, such as test coverage and code duplication.

Given the customisable nature of PMD and the high degree of integration of Sonarqube (especially in the dashboard outputted by the "server" component), we focus on the former for flaw detection and for implementing custom detection rules, while we use the latter to double-check our analysis and to briefly broaden the scope of our analysis to the test coverage and code duplication metrics.

Finally, we considered to analyze both the library source code and its test code, as the former is surprisingly smaller than the other (roughly 25KLOC vs roughly 60KLOC as reported by the *cloc* tool when ran on the separate subdirectories).

2.1 Choice of Detection Rules

While using the PMD tool, we decide to enable almost all the predefined rule-sets since they are all useful, although with varying degrees of relevancy. The rule-sets we deem most important are:

Performance Rules that flag sub-optimal code and being a base library is supposed to run efficiently.

Multi-threading Rules that flag issues when dealing with multiple threads of execution and this library is involved in offering helper classes for thread (package reflect).

Documentation Rules that are related to code documentation and are useful for having a neat and clean documented class interfaces in order to be read easily by every Java Developer.

Error Prone Rules to detect constructs that are either broken, extremely confusing or prone to runtime errors. The same reason to enable this rule is because it's a base library and it's the solid fundamental to build upon other software layers.

Secondly, the less important but necessary to cover as many flaws as possible:

Best Practices Rules which enforce generally accepted best practices.

[3]`https://pmd.github.io/`
[4]`https://www.sonarqube.org/`

Code Style Rules which enforce a specific coding style.

Design Rules that help you discover design issues.

Finally, we decide not to enable the **Security** rule-set as Apache Commons Lang does not implement any features related to cryptography. Enabling the rule-set would likely cause false positives on code that uses pseudo-random generators as they are insecure for cryptography, ignoring the fact that such classes specify in their documentation that they are indeed unsuited for use in these settings.

2.2 Custom Detection Rule

As mentioned, we choose to implement a custom detection rule using the PMD tool. In particular, we choose to implement the rule using XPath expressions for the aforementioned ease of use benefits.

The rule we decide to implement is based on an observation we make by browsing the Apache Commons Lang source code. Many class have a name ending in *Utils*, suggesting they are "utilty classes", i.e. classes that offer only static methods in their interfaces that are meant to be used as simple "procedural" functions (this concept is analogous to "static classes" in C#). Some of these classes, however, expose an interface that is not completely congruent with how a library user is allowed to use one of such classes.

In particular, we found instances of **public constructors** and class declarations missing the **final modifier**, which make little sense as these classes are not meant to be instanced or extended. Listing **??** shows one of such flawed implementations.

Listing 1: Interface of class *CharSequenceUtils* showing a flawed implementation of a utility class.

```
public class CharSequenceUtils {
    public static CharSequence subSequence(final CharSequence cs, final int start)
    { /*...*/ }
    public static int lastIndexOf(final CharSequence cs, final int searchChar,
        int start)
    { /*...*/ }
    public static int indexOf(final CharSequence cs, final int searchChar, int start)
    { /*...*/ }
    // [further methods omitted for brevity]
    public CharSequenceUtils() {}
}
```

Our XPath rule implementation for detecting this flaw, shown in listing **??**, can be summarized as follows:

- Firstly, we check if the class ended with the suffix `Utils` to be sure we are indeed encountering a potential utility class.
- Secondly, we verify whether a **final modifier** is missing the class declaration and **public constructors** in the class interface. If any of the two is true, we have a violation.
- Finally, we do a last check on the **static methods** analyzing eventual static modifier left.

Listing 2: PMD XPath expression able to find flawed utility class implementations.

```
//ClassOrInterfaceDeclaration
[
    if (ends-with(@SimpleName,'Utils'))
        then @Final=false() or
        ClassOrInterfaceBody/ClassOrInterfaceBodyDeclaration/ConstructorDeclaration[
        @Private=false() ] or
        ClassOrInterfaceBody/ClassOrInterfaceBodyDeclaration[
        count(./MethodDeclaration [@Kind="METHOD" and not(@Static=true())]) $>0$ ]
    else ()
]
```

XPath is indeed easy to use and we building this rule is simple, at least compared to writing a piece of Java code performing the same kind of detection. We are confident this rule can give us insights about the quality of the Apache Commons Lang project, as many classes declare themselves as utility classes in their contract and documentation and checking if they are indeed what they claim to be assures that the principle of least astonishment[5] is followed in the library's design.

[5] `https://en.wikipedia.org/wiki/Principle_of_least_astonishment`

2.3 Sonarqube

As Sonarqube is quite a complex yet complete tool, we decide to run it with the default parameters. After installing the “server” component, we add a new Maven project to it and we use the following Maven command to run the “scanner” component:

Listing 3: Maven command used to run the “scanner” component. `[KEY]` to be replaced with the “server” project key and `[TOKEN]` to be replaced with the “server” project token.

```
mvn clean site sonar:sonar -Dsonar.projectKey=Apache-Commons-Lang \
    -Dsonar.host.url=[KEY] \
    -Dsonar.login=[TOKEN] \
    -Dsonar.sourceEncoding=UTF-8 \
    -Dsonar.language=java \
    -Dsonar.junit.reportsPath=target/surefire-reports \
    -Dsonar.surefire.reportsPath=target/surefire-reports \
    -Dsonar.verbose=true \
    -Dsonar.jacoco.xmlReportPaths=target/site/jacoco/jacoco.xml \
    -Dsonar.java.coveragePlugin=jacoco
```

Using this command the Apache Commons Lang project will build correctly, and the build toolchain will run the already configured JaCoCo tool to provide to Sonarqube correct test coverage data.

3 PMD Results

This section shows the results we obtain by running the analysis we design in the previous section. Figure **??** shows a summary of the violations found by the PMD tool using the existing rule-set we choose.

Rule Name	**Violations**	**Rule Name**	**Violations**
AvoidReassigningParameters	246	ClassWithOnlyPrivateConstructorsShouldBeFinal	7
UseVarargs	233	MissingBreakInSwitch	7
FieldDeclarationsShouldBeAtStartOfClass	224	CloseResource	6
UncommentedEmptyConstructor	67	SimpleDateFormatNeedsLocale	5
GodClass	56	UseLocaleWithCaseConversions	4
UseUtilityClass	48	SingleMethodSingleton	4
CompareObjectsWithEquals	47	ConfusingTernary	3
ConstructorCallsOverridableMethod	41	SingletonClassReturningNewInstance	3
UncommentedEmptyMethodBody	35	AbstractClassWithoutAbstractMethod	3
FinalFieldCouldBeStatic	26	SwitchStmtsShouldHaveDefault	2
AvoidSynchronizedAtMethodLevel	22	UnnecessaryLocalBeforeReturn	2
PreserveStackTrace	17	DataClass	2
AvoidDeeplyNestedIfStmts	13	DefaultLabelNotLastInSwitchStmt	1
InstantiationToGetClass	9	LogicInversion	1
SimplifyBooleanExpressions	8		

Figure 2: Number of violations by rule found by the PMD tool using the existing rule-sets we choose.

We notice that *UseVarargs*, *FieldDeclarationsShouldBeAtStartOfClass* and *AvoidReassigningParameters* are the rules most violated by the Apache Commons Lang sources.

3.1 AvoidReassigningParameters

This rule is also quite self-documenting: parameters declared in a method signature should not be reassigned in that method’s body, or in other words, all formal parameter variables should be *effectively final*.

PMD classifies this rule at “medium-high priority”, and we agree with this high property placement. Even though avoiding parameter re-assignment is a mere stylistic choice, it is one that brings great benefits. It considerably improves code readability and it follows the principle of least astonishment, as developers often assume parameters are not mutated.

We provide an example of poor readability in the form of Apache Commons Lang’s *ArrayUtils.shift(...)* method, correctly reported by PMD as a violation of this rule and shown in listing **??**. Understanding the algorithm in the `while` loop is made slightly more confusing by the fact that *startIndexInclusive* and *endIndexExclusive* are mutated, even if only once and for simple bound checks. The bound check is not as obvious as it can be, and a developer may think the following code is buggy as it is operating on potentially unboundedd parameters.

Listing 4: The *shift(. . .)* method in the *ArrayUtils* class shows why parameter re-assigment may be confusing when reading code.

```java
public class ArrayUtils {
    // [...]
    public static void shift(final long[] array, int startIndexInclusive, int endIndexExclusive,
         int offset) {
        // [...]
        if (startIndexInclusive < 0) {
            startIndexInclusive = 0;
        }
        if (endIndexExclusive >= array.length) {
            endIndexExclusive = array.length;
        }
        int n = endIndexExclusive - startIndexInclusive;
        // [...]
        while (n > 1 && offset > 0) {
            final int n_offset = n - offset;
            if (offset > n_offset) {
                swap(array, startIndexInclusive, startIndexInclusive + n - n_offset, n_offset);
                n = offset;
                offset -= n_offset;
            } else if (offset < n_offset) {
                swap(array, startIndexInclusive, startIndexInclusive + n_offset, offset);
                startIndexInclusive += offset;
                n = n_offset;
            } else {
                swap(array, startIndexInclusive, startIndexInclusive + n_offset, offset);
                break;
            }
        }
    }
    // [...]
}
```

3.2 UseVarargs

UseVarargs checks if methods taking arrays as parameters use the *varargs* syntax in the signature declaration to support variadic method calls. According to PMD, this warning is "medium-low critical", i.e. of low importance. There are indeed benefit in using the *varargs* syntax, however its absence is not necessarily a code smell. This could also be due to a deliberate stylistic choice, as some classes are indeed designed to work specifically on array objects and calls using the variadic syntax would make little sense. One example where this speculation may apply is the *ArraySorter* class, whose method signatures matching the rule are shown in Listing **??**. This utility class was clearly designed for sorting arrays, and although a variadic parameter list is implicitly an array, this is not immediately obvious and might pose problems in the readability of client code.

Listing 5: Some method signatures of *ArraySorter* class flagged by PMD's *UseVarargs* rule.

```java
public class ArraySorter {
    public static byte[] sort(final byte[] array) { /* ... */ }
    public static char[] sort(final char[] array) { /* ... */ }
    // [other methods for the other primitive types]
    public static <T> T[] sort(final T[] array,
        final Comparator<? super T> comparator) { /* ... */ }
}
```

3.3 FieldDeclarationsShouldBeAtStartOfClass

This rule is fairly self explanatory: field declarations should by place in the beginning of a class declaration. This is a de-facto stylistic rule widely accepted by the Java community, as placing fields and methods in separate regions of a class's source code improves the readability of the overall code. Fields are usually placed on the top by convention, one that likely has its roots in the UML class diagram notation.

This rule has "medium priority" w.r.t. the PMD output, although we think it is of little concern, as all violations would be solved by running a code formatter tool over the source code.

3.4 ConstructorCallsOverridableMethod

This rule forbids calls to methods that are not *final* in constructors of classes that may be overridden. Such method calls should be avoided since a child class may override the method and act on fields that were not yet initialized by the parent class constructor, causing a *NullPointerException*. The PMD tool documentation further mentions that this "can be difficult to debug".

PMD assign a "high" priority to these violations, and we agree due to the particularly insidious nature of bugs that such method calls may cause.

Additionally, we think this decreases the robustness of the class' interface, in particular of the constructor. We illustrate as an example the class *text.StrBuilder*, shown in listing **??**. If a child class would override the *append(...)* method with a buggy implementation, for example by accessing an index out of bounds of the *buffer* field, this will not cause runtime errors on the method itself, but on the *public StrBuilder(String)* constructor.

We finally want to mention *text.StrBuilder* is deprecated in the 3.12.0 version of the Apache Commons Lang, as all the classes in the *text* package were moved in the Apache Commons Text library. It is possible the copy of this class in the Text library does not contain this violation anymore.

Listing 6: Constructor and *append(...)* method of class *text.StrBuilder*.

```java
public class StrBuilder implements CharSequence, Appendable, Serializable, Builder<String> {
    // [...]
    protected char[] buffer;
    // [...]
    public StrBuilder(final String str) {
        if (str == null) {
            buffer = new char[CAPACITY];
        } else {
            buffer = new char[str.length() + CAPACITY];
            append(str);
        }
    }
    // [...]
    public StrBuilder append(final String str) {
        if (str == null) {
            return appendNull();
        }
        final int strLen = str.length();
        if (strLen > 0) {
            final int len = length();
            ensureCapacity(len + strLen);
            str.getChars(0, strLen, buffer, len);
            size += strLen;
        }
        return this;
    }
    // [...]
}
```

3.5 PreserveStackTrace

This rule mandates that when throwing an exception in a catch block the thrown exception should reference the stack trace of the caught exception. This is usually achieved by making the thrown exception object reference the caught exception object as a "cause" (e.g. the *RuntimeException(Throwable cause)* constructor allows this).

PMD classifies this rule with a severity of "medium", and we agree with the rather severe rating as failing to follow this rule will likely make debugging affected code harder as part of the stack trace is not reported.

As a violation example, the *builder.ReflectionDiffBuilder.appendFields(Class<?>)* method does catch but not correctly report stack traces of *IllegalAccessException* objects. The relevant class is shown in listing **??**.

Listing 7: Improper exception handling in class *builder.ReflectionDiffBuilder*.

```
public class ReflectionDiffBuilder<T> implements Builder<DiffResult<T>> {
    // [...]
    private void appendFields(final Class<?> clazz) {
        for (final Field field : FieldUtils.getAllFields(clazz)) {
            if (accept(field)) {
                try {
                    diffBuilder.append(field.getName(), readField(field, left, true),
                        readField(field, right, true));
                } catch (final IllegalAccessException ex) {
                    throw new InternalError("Unexpected IllegalAccessException: " +
                        ex.getMessage());
                }
            }
        }
    }
    // [...]
}
```

3.6 SingletonClassReturningNewInstance

Analyzing the violation of the **SingletonClassReturningNewInstance** the false positive shown in listing **??**.

Listing 8: The *CharSet* class – mistakenly detected as a flawed singleton pattern application.

```
public class CharSet implements Serializable {
    // [...]
    protected static final Map<String, CharSet> COMMON =
        Collections.synchronizedMap(new HashMap<>());
    // [...]
    static {
        COMMON.put(null, EMPTY);
        COMMON.put(StringUtils.EMPTY, EMPTY);
        COMMON.put("a-zA-Z", ASCII_ALPHA);
        COMMON.put("A-Za-z", ASCII_ALPHA);
        COMMON.put("a-z", ASCII_ALPHA_LOWER);
        COMMON.put("A-Z", ASCII_ALPHA_UPPER);
        COMMON.put("0-9", ASCII_NUMERIC);
    }
    // [...]
    public static CharSet getInstance(final String... setStrs) {
        if (setStrs == null) { return null; }
        if (setStrs.length == 1) {
            final CharSet common = COMMON.get(setStrs[0]);
            if (common != null) { return common; }
        }
        return new CharSet(setStrs);
    }
    // [...]
}
```

PMD revealed a violation of the Singleton design pattern even though the latter isn't really implemented in this class. Actually, we are in front of a *"factory" method* which, given a variadic sequence of *String* objects as input parameters, it creates an instance of *CharSet* if and only if it is not already stored into the *HashMap* static field named *COMMON*. Finally either a stored or a newly allocated *CharSet* instance is returned.

In fact from a first glance the PMD result sounds right by only looking at the method name *getInstance()* and the singleton template in the body method (...if (common != null)...).

3.7 Custom Rule – Utility Class Rule

AnnotationUtils	exception.ExceptionUtils	RegExUtils
ArchUtils	LocaleUtils	SerializationUtils
ArrayUtils	math.IEEE754rUtils	StringEscapeUtils
BooleanUtils	math.NumberUtils	StringUtils
CharSequenceUtils	ObjectUtils	SystemUtils
CharSetUtils	RandomStringUtils	text.FormattableUtils
CharUtils	RandomUtils	text.WordUtils
ClassLoaderUtils	reflect.ConstructorUtils	ThreadUtils
ClassPathUtils	reflect.FieldUtils	time.CalendarUtils
ClassUtils	reflect.InheritanceUtils	time.DateFormatUtils
compare.ComparableUtils	reflect.MemberUtils	time.DateUtils
concurrent.ConcurrentUtils	reflect.MethodUtils	time.DurationFormatUtils
EnumUtils	reflect.TypeUtils	time.DurationUtils
event.EventUtils		

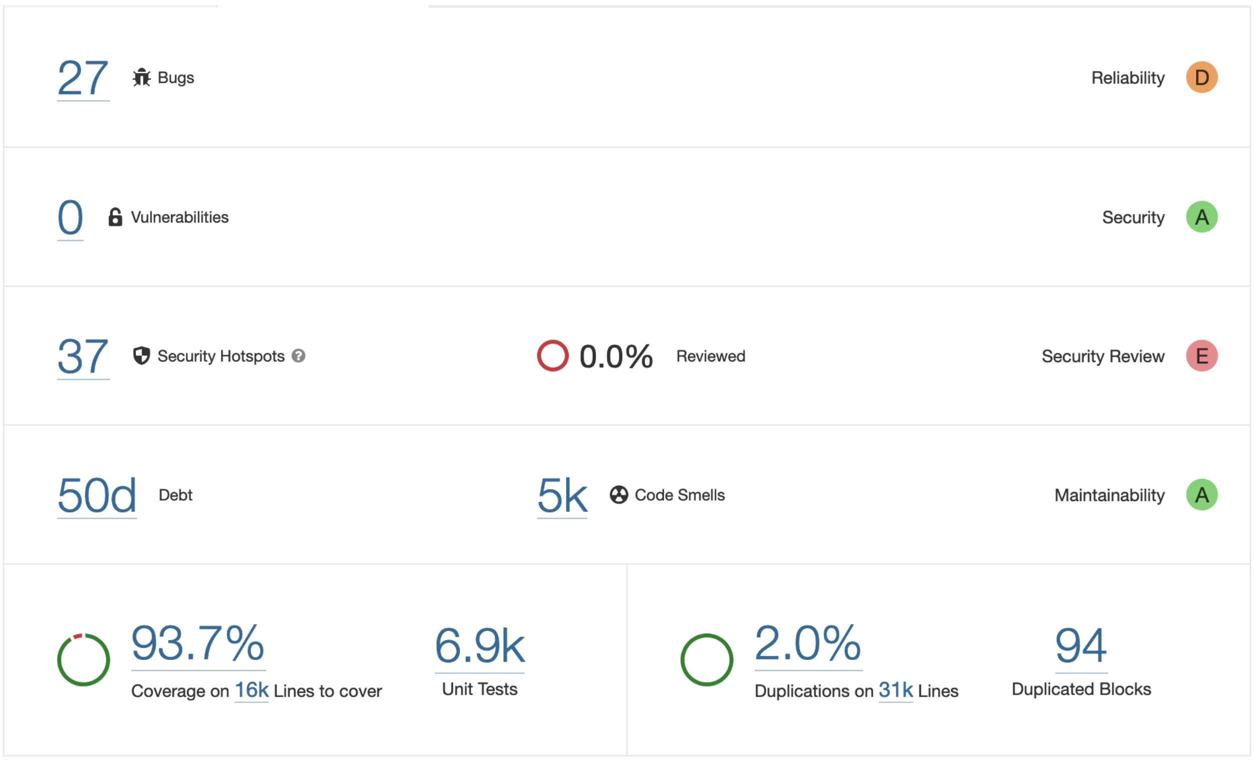
Figure 3: List of classes matching the utility class custom rule we implement.

Figure **??** shows the list of classes in Apache Commons Lang matching the utility class rule we implement in the analysis section.

Concerning the custom rule, we want to join *ClassWithOnlyPrivateConstructorsShouldBeFinal* and *UseUtilityClass* in the same rule since the library contains plenty of utility classes which do not comply with this two rules. In fact our custom rule merges them at considering the *Utils* suffix the Apache Commons suite of libaries uses.

Moreover, PMD reports that all the utility classes violate the *GodClass* rule considering them too complex, given their low cohesion, high coupling, and high line of code count. However, since the Commons Lang library is quite legacy, the Apache developers cannot modify their existing interface up to a certain point. Therefore, we could consider these violations as false positives.

4 Sonarqube results


27 Bugs Reliability D

0 Vulnerabilities Security A

37 Security Hotspots 0.0% Reviewed Security Review E

50d Debt 5k Code Smells Maintainability A

93.7% Coverage on 16k Lines to cover 6.9k Unit Tests

2.0% Duplications on 31k Lines 94 Duplicated Blocks

Figure 4: Sonarqube report summary as shown by the “server” module.

Figure ?? shows the Sonarqube report summary as shown by the web interface of the “server” module. Even if the metrics collected are quite rich and diverse, the “scanner” component took only approximately 5 minutes to generate the report as the codebase is not particularly large (especially given sources are only roughly 20KLOC).

The report shows 27 bugs found, no vulnerabilities, 37 security-related problems, and 5.000 code smells, for estimated debt of 50 man-days. Additionally, the 6.900 tests have a combined coverage of 93.7% (94.9% line coverage and 91.7% statement coverage) and a duplication figure of 2.0% across 94 duplicated code blocks.

A breakdown of the 27 bugs reported by Sonarqube follows:

- 3 critical bugs related to the use of *assertNotNull(. . .)* in tests over primitive arguments, which will always lead to true;
- 2 likely false positive critical bugs related to *assertEquals(. . .)* comparing dissimilar types, that are indeed equivalent anonymous class instances;
- 5 major bugs related to the unchecked use of potentially null variables;
- 3 false positive major bugs related to using reference equality on objects, whose use is correct since the checks reside in *Object.equals(Object)* overrides and it is there for performance reasons;
- 5 major bugs related to trivial assertions in tests that the test developers describe in a comment as `// sanity checks:`;
- 1 major bug related to an *Object.clone()* override possibly returning null;
- 1 major bug related to a reflective check of whether two objects have the same dynamic type, where Sonarqube suggests to use the method *isAssignableFrom(. . .)*;
- 3 minor bugs related to *int* expressions being stored in *long* variables;
- 2 minor bugs related to *volatile* fields being used instead of thread-safe types;
- 1 minor bug related to an iterator implementation not throwing *NoSuchElementException* when no next element is available;
- 1 minor bug related to an *Object.hashCode()* override not having a matching *Object.equals(Object)* override.

The 5.000 code smells break down in 236 blocker problems, 167 critical problems, 772 major problems, 117 minor problems and 3.700 “info”-level warnings.

All the security related problems mention the cryptographically unsafe nature of the *java.util.Random* class due to the use of pseudo-randomness and an inadequately random seed. These errors can be discarded since as mentioned in the analysis section, Apache Commons Lang does not provide by its contract any cryptographically safe feature and thus the use of pseudo-randomness is acceptable.

5 Conclusions

The Apache Commons Lang library, as shown by our analysis, has several anti-patterns and design flaws that might be avoided or mitigated. Both of the tools used show several violations according to their respective rule-sets. Given the relatively legacy nature of the project a certain number of flaws is to be expected. This is especially true given the wide use of utility classes, a practice that nowadays for new projects is frowned upon as it breaks traditional Object Oriented Programming principles.

We find both PMD and Sonarqube effective tools. PMD is highly customizable in a simple fashion, yet very powerful in the realm of static code analysis. Sonarqube is able to provide deeper insights and additional metric, at the cost of a more highly integrated design and a lesser degree of customization.

Our results likely apply to other Apache Commons libraries, given that many of them are meant to be used in a similar fashion as the Lang library. This is especially true if these libraries use the same class structure and widely exploit the utility class pattern, a likely true assumption given that some of them are direct spin-offs of packages in the Lang library (like Commons Text). However, given the peculiar “utility” role Apache Commons plays in Java development, these techniques do not directly translate for a generic Java library.