Assignment 2 – Software Analysis

Static Analysis with Infer

Claudio Maggioni

1 Project selection

Given that this assignment draws parallels with the class of Software Design and Modelling of last semester, specifically regarding static analyzers, I choose to analyze the same project I analyzed in the past with PMD and SonarQube using Infer[1] to make for an interesting comparison between static analysis paradigms.

The project I analyze is therefore **apache/commons-lang**.

1.1 The Apache Commons Lang Project

The Apache Commons family of libraries is an Apache Software Foundation[2] 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.

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 file paths and classes in the project.

I 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 same version analyzed during the SDM class.

To verify that the project satisfies the 5000 lines of code requirement, I run the *cloc* tool. Results are shown in table 1. Given the project has more than 86,000 lines of Java code, this requirement is satisfied.

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

Table 1: Output of the *cloc* tool for the Apache Commons Lang project at tag *rel/commons-lang-3.12.0* (before fixes are applied).

[1] `https://fbinfer.com/`
[2] `https://apache.org/`

2 Running the Infer tool

The relevant source code to analyze has been copied to the directory *before* in the assignment repository

usi-si-teaching/msde/2022-2023/software-analysis/maggioni/assignment-2

on *gitlab.com*. The script *docker-infer.sh* can be executed to automatically run the Infer tool using default options through the course tools docker image *bugcounting/satools:y23*.

The script executes Infer in Maven capture mode executing the *compile* and *test* targets while disabling the Apache RAT software license checker (which fails for this release) and the Animal Sniffer Maven plugin do to the failure message "This feature requires ASM7" it produces if ran. Since unit tests are executed, running the script before and after the warning guided refactoring ensures the fixes I introduce do not introduce regressions.

The analysis outputs are located in *before/infer-out/report.txt*.

3 Results

Table 2 shows the results of the analysis performed by Infer providing comments on true and false positives and the actions taken for each result.

File	**Line**	**Kind**	**True Pos.**	**Reason why flagged expression is a false positive**
AnnotationUtils.java	72	Null	Yes	
reflect/MethodUtils.java	486	Null	Yes	
concurrent/MultiBackgroundInitializer.java	160	Thread Safety	Yes	
builder/ToStringBuilder.java	223	Null	No	Infer flags the value `null` when used as a nullable method argument
builder/ReflectionToStringBuilder.java	131	Null	No	
time/DurationUtils.java	142	Null	No	The method which may return a null value returns a non-null value if its parameter is non-null, and a non-null parameter is given
CharSetUtils.java	181	Null	No	According to *java.lang* documentation, the method always returns a non-null value
reflect/FieldUtils.java	126	Null	No	
reflect/FieldUtils.java	341	Null	No	A utility method is used to guard the
reflect/FieldUtils.java	385	Null	No	dereference reported with an exception
reflect/FieldUtils.java	599	Null	No	throw
reflect/FieldUtils.java	644	Null	No	
reflect/MethodUtils.java	987	Null	No	The method which may return a null value returns a non-null value if its parameter is non-null, and a non-null parameter is always given according to the *java.lang* documentation for the inner nested method

Table 2: Results of the Infer static analysis tool execution with default options. *True Pos.* denotes whether a result is a true positive, while *Kind* denotes with *Null* and *Thread Safety* respectively null dereference warnings and thread safety violations.

In total Infer reports 13 warnings, 12 of which are null dereference warnings and 1 is a thread safety violation. Of all warnings, 3 are true positives and 10 are false positives, resulting in a precision of 23%. These values are summarized in table 3.

Total number of warnings:	13
Null dereference warnings:	12
Thread safety violations:	1
True positives:	3
False positives:	10

Table 3: Quantitative results of the static analysis performed by Infer.

3.1 False Positives

As can be deduced from table 2, Infer especially struggles at determining Vthe null safety of nested method calls or non-trivial data flow paths. This is sometimes caused by insufficient knowledge of the nullability contracts of the Java standard library (e.g. *java.lang* package).

In other cases, the flagged expression is guarded by an utility method throwing an exception if its value is indeed `null`. One of such guards is the `static Validate.notNull(Object, String, Object...)` method, which checks if its first argument is null and throws a `NullPointerException` with a "pretty" message composed from a format string and variadic arguments if so.

Additionally, Infer seems to struggle with boxed primitives and not understanding that their value is always non-null by construction. As an example I provide the warning reported in the file *time/DurationUtils.java*:

Listing 1 Method *toMillisInt(Duration)* of class *time.DurationUtils* in Apache Commons Lang 3.12.0.

```
139 public static int toMillisInt(final Duration duration) {
140     Objects.requireNonNull(duration, "duration");
141     // intValue() does not do a narrowing conversion here
142     return LONG_TO_INT_RANGE.fit(Long.valueOf(duration.toMillis())).intValue();
143 }
```

Here Infer reports that the first argument of `LONG_TO_INT_RANGE.fit(Long)` may be null. However, the return value of `Long.valueOf(long)` is always non-null since the method simply boxes its `long` argument.

Finally, some warnings are caused by Infer flagging the use of the `null` keyword as a method argument for methods that would accept a nullable argument in that position without causing null dereferences. This warning could point to a potential ambiguity is selecting the right method to call at compile time given the argument types (i.e. static dispatching), as `null` is a valid expression of all object types. For example, in the given class:

```
class C {
  m(String s) {}
  m(Object o) {}
}
```

A call to `C.m(null)` would be ambiguous as `null` is both a `String` and an `Object`, thus a cast to either type woule be required to make the code compile. However, the warnings reported by Infer do not present such ambiguity as in those cases overloaded methods have different numbers of parameters. Additionally, even introducing explicit casts for all `null` arguments does not extinguish the warning.

Therefore, I can not find a conclusive explaination on the nature of the false positive, however I can attest to these instances being indeed false positives by having manually verified that the methods in question indeed can accept a null value without causing null dereferences.

3.2 True Positives

In this section I now cover the warnings that are true positives and thus are causes for refactoring.

A copy of the entire code base where refactoring has been applied can be found in the *after* directory of the assignment's repository.

Listing 2 Method *getShortClassName(Class<?>)* of class *AnnotationUtils* in Apache Commons Lang 3.12.0.

```
71 @Override
72 protected String getShortClassName(final Class<?> cls) {
73     for (final Class<?> iface : ClassUtils.getAllInterfaces(cls)) {
74         if (Annotation.class.isAssignableFrom(iface)) {
75             return "@" + iface.getName();
76         }
77     }
78     return StringUtils.EMPTY;
79 }
```

Listing 2 shows the location of the first true positive warning. Here, `ClassUtils.getAllInterfaces(cls)` at line 73 may return a null value thus potentially causing a null dereference when iterating in the enhanced for loop. Indeed, the method's implementation shows that while for non-null `cls` values the return value is non-null, if `cls` is null `null` is returned. Since the method is protected its signature is visible to all child classes, even ones potentially outside of the Apache Commons Lang library. As the method's documentation does not specify anything about the nullability of the argument `cls`, its value may indeed be null and thus a refactor is needed to avoid the potential null dereference.

I choose to simply return `StringUtils.EMPTY` when `cls` is null. The refactored implementation can be found in listing 3.

Listing 3 Refactor of method *getShortClassName(Class<?>)* of class *AnnotationUtils*.

```
71 @Override
72 protected String getShortClassName(final Class<?> cls) {
73     final List<Class<?>> interfaces = ClassUtils.getAllInterfaces(cls);
74     if (interfaces == null) {
75         return StringUtils.EMPTY;
76     }
77
78     for (final Class<?> iface : interfaces) {
79         if (Annotation.class.isAssignableFrom(iface)) {
80             return "@" + iface.getName();
81         }
82     }
83     return StringUtils.EMPTY;
84 }
```

Next, class *reflect.MethodUtils* contains a bug due to insufficient defensiveness when accepting the

method parameters.

Listing 4 Method *getVarArgs(Object[], Class<?>[])* of class *reflect.MethodUtils* in Apache Commons Lang 3.12.0.

```
461 static Object[] getVarArgs(final Object[] args, final Class<?>[] methodParameterTypes) {
462     if (args.length == methodParameterTypes.length && (args[args.length - 1] == null ||
463             args[args.length -
                ↪  1].getClass().equals(methodParameterTypes[methodParameterTypes.length -
                ↪  1]))) {
464         // The args array is already in the canonical form for the method.
465         return args;
466     }
467 
468     // Construct a new array matching the method's declared parameter types.
469     final Object[] newArgs = new Object[methodParameterTypes.length];
470 
471     // Copy the normal (non-varargs) parameters
472     System.arraycopy(args, 0, newArgs, 0, methodParameterTypes.length - 1);
473 
474     // Construct a new array for the variadic parameters
475     final Class<?> varArgComponentType = methodParameterTypes[methodParameterTypes.length -
        ↪  1].getComponentType();
476     final int varArgLength = args.length - methodParameterTypes.length + 1;
477 
478     Object varArgsArray =
        ↪  Array.newInstance(ClassUtils.primitiveToWrapper(varArgComponentType), varArgLength);
479     // Copy the variadic arguments into the varargs array.
480     System.arraycopy(args, methodParameterTypes.length - 1, varArgsArray, 0, varArgLength);
481 
482     if (varArgComponentType.isPrimitive()) {
483         // unbox from wrapper type to primitive type
484         varArgsArray = ArrayUtils.toPrimitive(varArgsArray);
485     }
486 
487     // Store the varargs array in the last position of the array to return
488     newArgs[methodParameterTypes.length - 1] = varArgsArray;
489 
490     // Return the canonical varargs array.
491     return newArgs;
492 }
```

Line 482 of the class (listed in listing 5) is correctly flagged as a potential null dereference as `varArgComponentType` may be null. Indeed, according to the documentation of the method generating its value, the variable is null when the last value in the array `methodParameterTypes` is a class object for a non-array class. Such value would violate the method's contract, which specifies that the array should correspond to class types for the arguments of a variadic method, and the last argument in a variadic method is always an array. However, this is not checked defensively, and we can fix this by introducing a check after line 475 like this:

```
if (varArgComponentType == null) {
    throw new IllegalArgumentException("last method parameter type is not an array");
}
```

It is quite interesting Infer was able to detect this null dereference as it stems from a lack of sufficient defensiveness. Improper use of the method would trigger a *NullPointerException* without any useful message indicating the precondition violation.

Finally, the last warning involves class *concurrent.MultiBackgroundInitializer*. The relevant method source code is shown in listing **??**.

Listing 5 Method *getVarArgs(Object[], Class<?>[])* of class *reflect.MethodUtils* in Apache Commons Lang 3.12.0.

```
157 @Override
158 protected int getTaskCount() {
159     int result = 1;
160
161     for (final BackgroundInitializer<?> bi : childInitializers.values()) {
162         result += bi.getTaskCount();
163     }
164
165     return result;
166 }
```

Here, the object pointed by `childInitializers` is accessed without using any lock, while other methods of the class use the implicit lock of the instance (i.e. `synchronized(this)`). Simply adding a `synchronized` block around the for loop extinguishes the warning.

The refactoring performed for all three warnings has not produced further warnings, and it has not caused the failure of any unit test.

4 Conclusions

Given that Apache Commons Lang is a very mature project (the code still contains workarounds targeting Java 1.3) it is not surprising that Infer has detected a relatively limited number of warnings. If anything this should be a good testament to the skills and discipline of all project contributors. However, 3 true positives were found in this production-ready release version. This is a testament to the insidiousness of some programming mistakes like lack of defensiveness when dealing with possibly null values and improper use of lock constructs, a fact which applies to this project as well.

The high false positive ratio of the analysis highlights the tradeoff between soundness and completeness software analysis tecniques face. Given that the true positives were quite non-trivial, nevertheless Infer proves itself as a useful tool even with mature and production-ready software.

To draw some conclusions between this analysis and the one performed last semester during the SDM class, I hereby compare the current results with the ones formerly found.

When compared to the PMD source code analyzer and SonarQube, Infer reports issues closer to external quality factors (i.e. user satisfaction, and consequently runtime behaviour) rather than internal quality ones, such as the adherence to code style rules and presence of adequate documentation.

Some overlap is present in the scope of the analysis of both tools, namely with respect to thread safety (and null dereference with SonarQube). It is worth to note that all three tool reported warnings with distinct locations in the source code, and no overlap was found between them.

However, due to its control and data flow analysis capabilities, Infer is a superior tool to understand and check possible runtime behavior while analyzing only the source code.

We can also say both PMD and SonarQube have superior awareness regarding the behaviour of Java and its standard library. For example, they are aware of boxing and unboxing of primitive types (and SonarQube even produces warnings about misuse of the feature), where as hinted by this analysis Infer seems not to be aware of, at least while searching null dereferences.