Métricas de calidad de código

Raúl Estrada

Octubre 2020

1. Overview

In this article, we will be highlighting some of the important rules featured in code analysis tools like FindBugs, PMD and CheckStyle.

2. Cyclomatic Complexity

2.1. What Is Cyclomatic Complexity?

Code complexity is important, yet difficult metric to measure. PMD offers a solid set of rules under its Code Size Rules section, these rules are designed to detect violation regarding methods size and structure complexity.

CheckStyle is known for its ability to analyse code against coding standards, and formatting rules. However, it can also detect problems in classes/methods design by calculating some complexity metrics.

One of the most relevant complexity measurement featured in both tools is the CC (Cyclomatic Complexity).

CC value can be calculated by measuring the number of independent execution paths of a program.

For instance, the following method will yield a cyclomatic complexity of 3:

```
public void callInsurance(Vehicle vehicle) {
    if (vehicle.isValid()) {
        if (vehicle instanceof Car) {
            callCarInsurance();
        } else {
            delegateInsurance();
        }
    }
}
```

CC takes into account the nesting of conditional statements and multi-part boolean expressions.

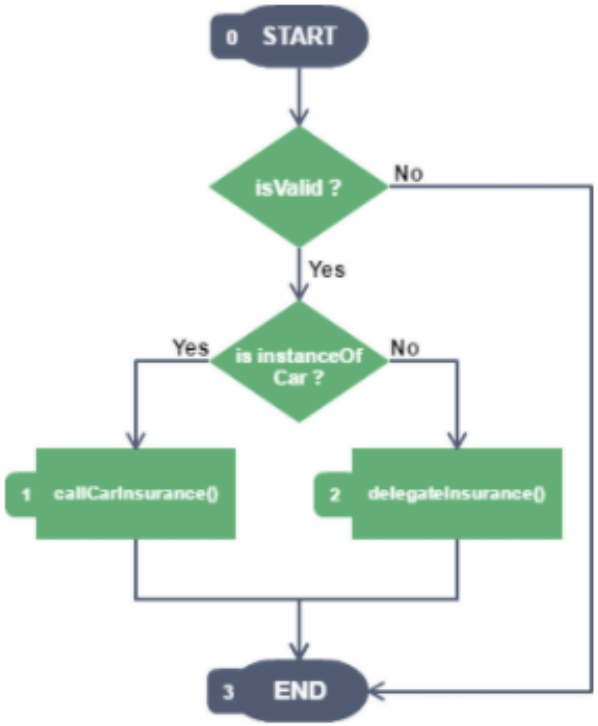
Generally speaking, a code with a value higher than 11 in terms of CC, is considered very complex, and difficult de test and maintain.

Some common values used by static analysis tools are shown below:

- 1-4: low complexity – easy to test
- 5-7: moderate complexity – tolerable
- 8-10: high complexity – refactoring should be considered to ease testing
- 11 + very high complexity – very difficult to test

The complexity level also affects the testability of the code, **the higher the CC, the higher the difficulty to implement pertinent tests**. In fact, the cyclomatic complexity value shows exactly the number of test cases needed to achieve a 100% branches coverage score.

The flow graph associated with the *callInsurance()* method is:



The possible execution paths are:

- 0 => 3
- 0 => 1 => 3
- 0 => 2 => 3

Mathematically speaking, CC can be calculated using the following simple formula:

```
CC = E - N + 2P
```

- E: Total number of edges
- N: Total number of nodes
- P: Total number of exit points

2.2. How to Reduce Cyclomatic Complexity ?

In order to write substantially less complex code, developers may tend to use different approaches, depending on the situation:

- Avoid writing lengthy *switch* statements by using design patterns, e.g. the builder and strategy patterns may be good candidates to deal with code size and complexity issues
- Write reusable and extensible methods by modularizing the code structure and implementing the **Single Responsibility Principle**
- **Following other PMD code size rules may have a direct impact on CC**, e.g. excessive method length rule, too many fields in a single class, excessive parameters list in a single method...etc

You can also consider following principles and patterns regarding code size and complexity, e.g. the **KISS (Keep It Simple and Stupid) principle**, and **DRY (Don't Repeat Yourself).**

3. Exception Handling Rules

Defects related to exceptions might be usual, but some of them are hugely underestimated and should be corrected to avoid critical dysfunctioning in production code.

PMD and FindBugs offer both a handful set of rules regarding exceptions. Here's our pick of what may be considered critical in a Java program when handling exceptions.

3.1. Do Not Throw Exception in Finally

As you may already know, the *finally{}* block in Java is generally used for closing files and releasing resources, using it for other purposes might be considered as a **code smell**.

A typical error-prone routine is throwing an exception inside the *finally{}* block:

```
String content = null;
try {
    String lowerCaseString = content.toLowerCase();
} finally {
    throw new IOException();
}
```

This method is supposed to throw a *NullPointerException*, but surprisingly it throws an *IOException*, which may mislead the calling method to handle the wrong exception.

3.2. Returning in the *finally* Block

Using the return statement inside a *finally[]* block may be nothing but confusing. The reason why this rule is so important, it's because whenever a code throws an exception, it gets discarded by the *return* statement.

For example, the following code runs with no errors whatsoever:

```
String content = null;
try {
    String lowerCaseString = content.toLowerCase();
} finally {
    return;
}
```

A *NullPointerException* was not caught, yet, still discarded by the return statement in the *finally* block.

3.3. Failing to Close Stream on Exception

Closing streams is one of the main reasons why we use a *finally* block, but it's not a trivial task as it seems to be.

The following code tries to close two streams in a *finally* block:

```
OutputStream outStream = null;
OutputStream outStream2 = null;
try {
    outStream = new FileOutputStream("test1.txt");
    outStream2  = new FileOutputStream("test2.txt");
    outStream.write(bytes);
    outStream2.write(bytes);
} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        outStream.close();
        outStream2.close();
    } catch (IOException e) {
        // Handling IOException
    }
}
```

If the *outStream.close()* instruction throws an *IOException*, the *outStream2.close()* will be skipped.

A quick fix would be to use a separate try/catch block to close the second stream:

```
finally {
    try {
        outStream.close();
    } catch (IOException e) {
        // Handling IOException
    }
    try {
        outStream2.close();
    } catch (IOException e) {
        // Handling IOException
    }
}
```

If you want a nice way to avoid consecutive *try/catch* blocks, check the IOUtils.closeQuiety method from Apache commons, it makes it simple to handle streams closing without throwing an *IOException*.

5. Bad Practices

5.1. Class Defines compareto() and Uses Object.equals()

Whenever you implement the *compareTo()* method, don't forget to do the same with the *equals()* method, otherwise, the results returned by this code may be confusing:

```
Car car = new Car();
Car car2 = new Car();
if(car.equals(car2)) {
    logger.info("They're equal");
} else {
    logger.info("They're not equal");
}
if(car.compareTo(car2) == 0) {
    logger.info("They're equal");
} else {
    logger.info("They're not equal");
}
```

Result:

```
They're not equal
They're equal
```

To clear confusions, it is recommended to make sure that *Object.equals()* is never called when implementing *Comparable*, instead, you should try to override it with something like this:

```
boolean equals(Object o) {
    return compareTo(o) == 0;
}
```

5.2. Possible Null Pointer Dereference

NullPointerException (NPE) is considered the most encountered *Exception* in Java programming, and FindBugs complains about Null PointeD dereference to avoid throwing it.

Here's the most basic example of throwing an NPE:

```
1 Car car = null;
2 car.doSomething();
```

The easiest way to avoid NPEs is to perform a null check:

```
1  Car car = null;
2  if (car != null) {
3      car.doSomething();
4  }
```

Null checks may avoid NPEs, but when used extensively, they certainly affect code readability.

So here's some technics used to avoid NPEs without null checks:

- **Avoid the keyword *null* while coding*:*** This rule is simple, avoid using the keyword *null* when initializing variables, or returning values
- **Use *@NotNull* and *@Nullable* annotations**
- **Use *java.util.Optional***
- **Implement the Null Object Pattern**

6. Conclusion

In this article, we have done an overall look on some of the critical defects detected by static analysis tools, with basic guidelines to address appropriately the detected issues.