Software Quality

- *Assignment 1 (Applying VizzMaintenance)*

Author: Sarpreet Singh Buttar
Email: sb223ce@student.lnu.se

Introduction

The purpose of this report is to analyze the quality of a Java library called JSON Web Tokens for Java and Android (JWT). It is an open source project[1] and provides functionality to create and verify JSON web tokens. JWT is actively developing since September 2014 and has 18 releases in total. In this report, we target the latest one, i.e., 0.10.6. We use VizzMaintenance[2] tool for the analyses, and Python programming language with Matplotlib[3] library to visualize the analyses results.

Project Setup

We follow following steps to analyze JWT:

1. Download Eclipse[4]
2. Download VizzMaintenance[2]
3. Download JWT from GitHub[1]
4. Import JWT in Eclipse as a Maven project
5. Enable VizzMaintenance in Eclipse
6. Apply VizzMaintenance on JWT to get analyses results

Global Statistics

JWT is a medium size project that have three modules named as API, extension, and Implementation. API module consists of 17 interfaces and 50 classes, which makes it the biggest module of JWT. Whereas, implementation module is the second biggest with 9 interfaces and 36 classes. Extension module is the smallest and only has 4 classes. JWT implements 522 methods in 9586 lines of code. Table 1 summarizes these properties.

Top level packages (subsystems)	3
Packages	24
Interfaces	26
Classes	90
Methods	522
Lines of Code	9586

Table 1: Global Statistics of JWT

[1] https://github.com/jwtk/jjwt
[2] http://arisa.se/products.php?lang=en
[3] https://matplotlib.org/index.html
[4] http://www.eclipse.org

Analysis of Large and Complex Classes

We use two metrics, i.e., Lines of Code (LOC) and Weighted Method Count (WMC), to determine the classes that are both large and complex. LOC computes the size of a class, whereas, WMC calculates the complexity of methods available in a class, see figure 1.

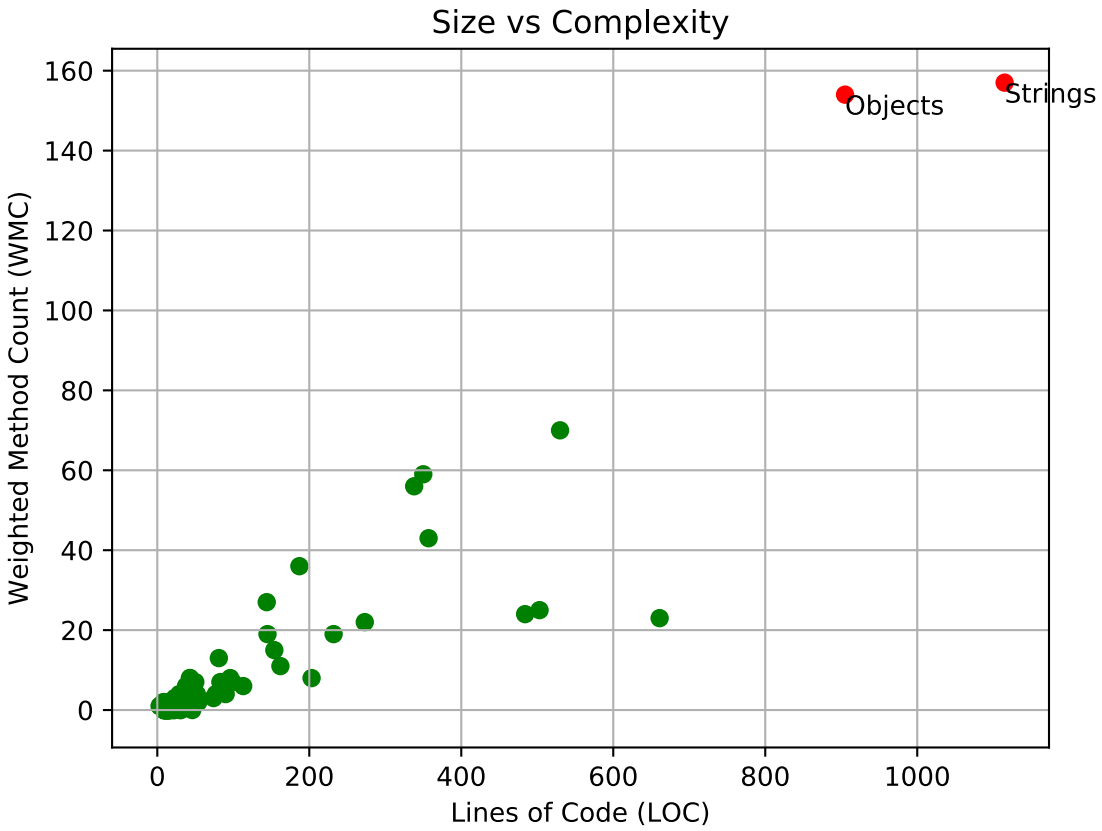

Figure 1: Size vs Complexity

The results show that JWT has two classes, i.e., Objects and Strings, that are both complex and large. Hence, it can be time consuming to understand and maintain these classes. We advise to split the functionality of these classes in order to reduce their size and complexity. In contrast, most of the other classes are close to the origin. However, there are some classes that are big in size but not complex. Therefore, we ignore them.

Analysis of Classes with Low Cohesion and High Coupling

We use Tight Package Cohesion (TCC) and Data Abstract Coupling (DAC) metrics to compute the cohesion and coupling values of the classes. Lack of Cohesion on Methods (LCOM) is an alternative to measure cohesion. It computes the difference among the methods that connected directly and indirectly. However, we consider JWT as a black box, and therefore TCC is better as it computes the class's cohesion. In general, a class is considered to be good if it has high cohesion and low coupling, see figure 2.

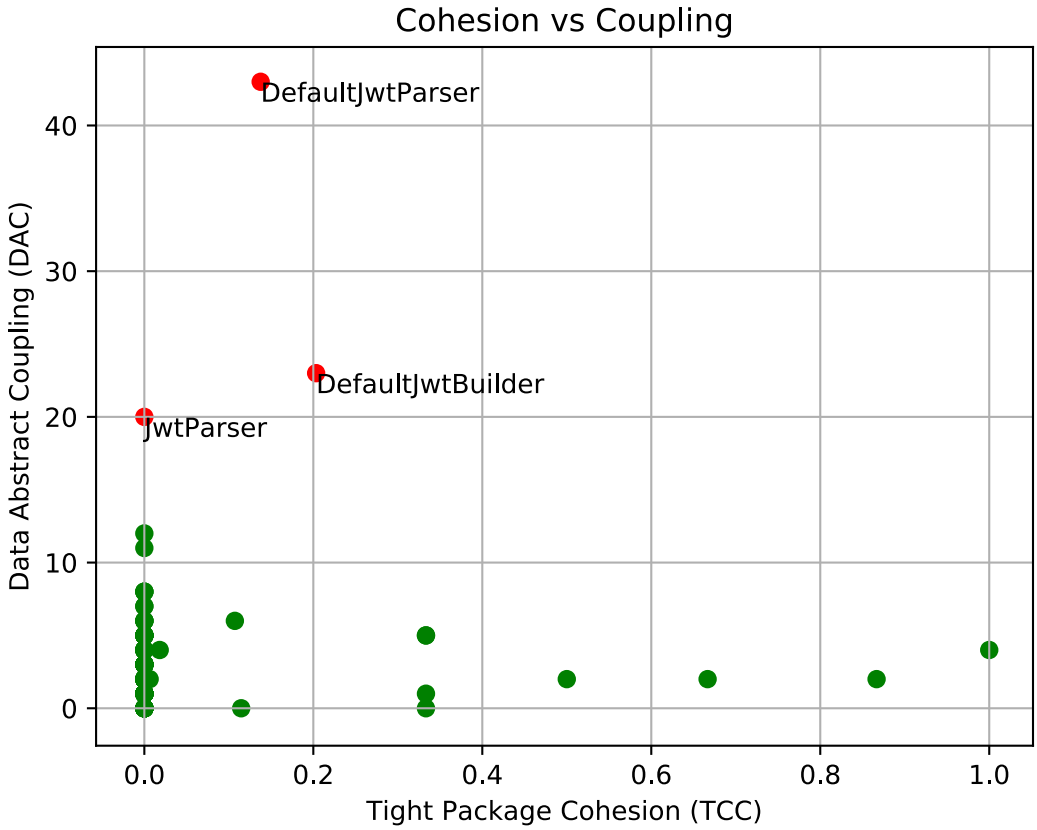

Figure 2: Cohesion vs Coupling

Figure 2 shows that JWT has three classes, i.e., DefaultJwtParser, DefaultJwtBuilder, and JwtParser, that have low cohesion and high coupling. It represents that these classes have methods that are not related to each other. Therefore, we propose to split these classes and make new ones with related methods. This will improve the cohesion which will eventually reduce the coupling.

Analysis of Classes with High Dependency

We focus on two main types of dependencies: coupling and inheritance. We use Depth in Inheritance (DIT) metric to measure the inheritance of a class. For measuring coupling, we use Coupling Between Objects (CBO) metric. The DIT value represents the number of classes that a particular class is inheriting from. Similarly, CBO value reflects the number of classes that a specific class coupled with. In both cases, low value is considered good, see figure 3.

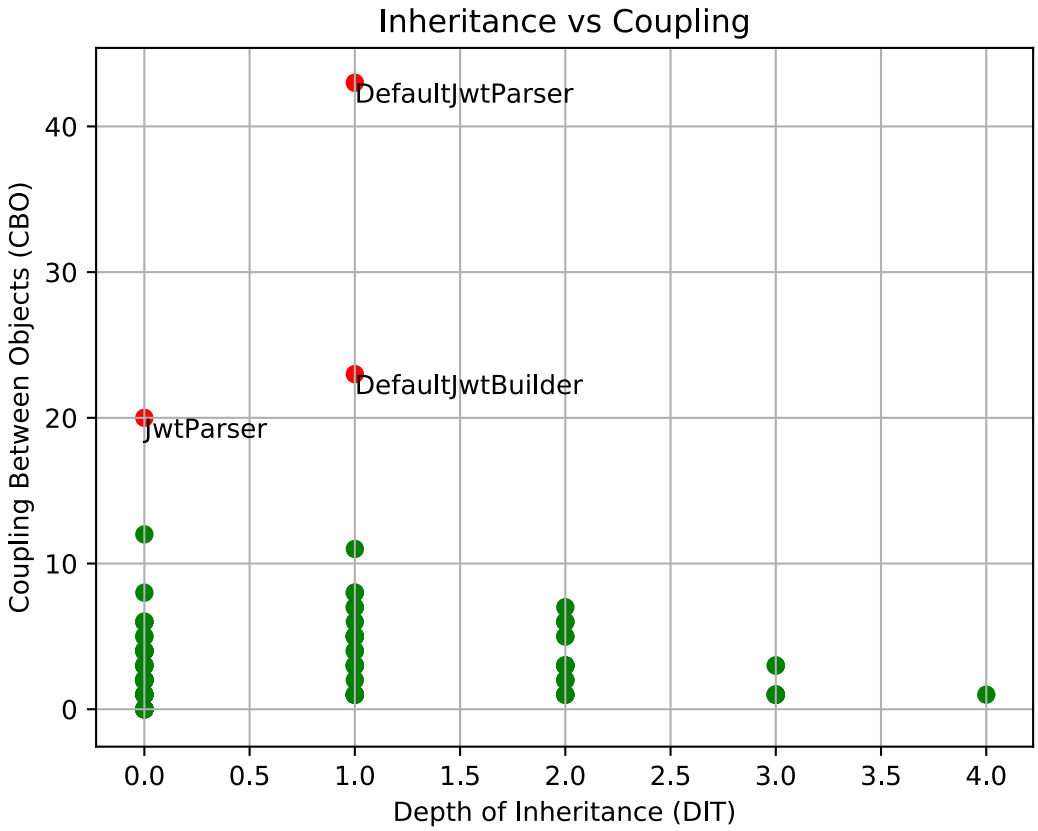

Figure 3: Inheritance vs Coupling

Figure 3 shows that JWT has three classes, DefaultJwtParser, DefaultJwtBuilder, and JwtParser, that are coupled with more than 15 classes. However, from their names, we can assume that the dependency is inverse, i.e., other classes are dependent on them. The remaining classes have good level of inheritance and coupling.

Analysis of Classes with High Complexity or Depending Classes

We use WMC metric as before to measure the complexity of a class. For finding the depending classes, we use Number of Children (NOC) metric. A good class should have lower value with both metrics, see figure 4.

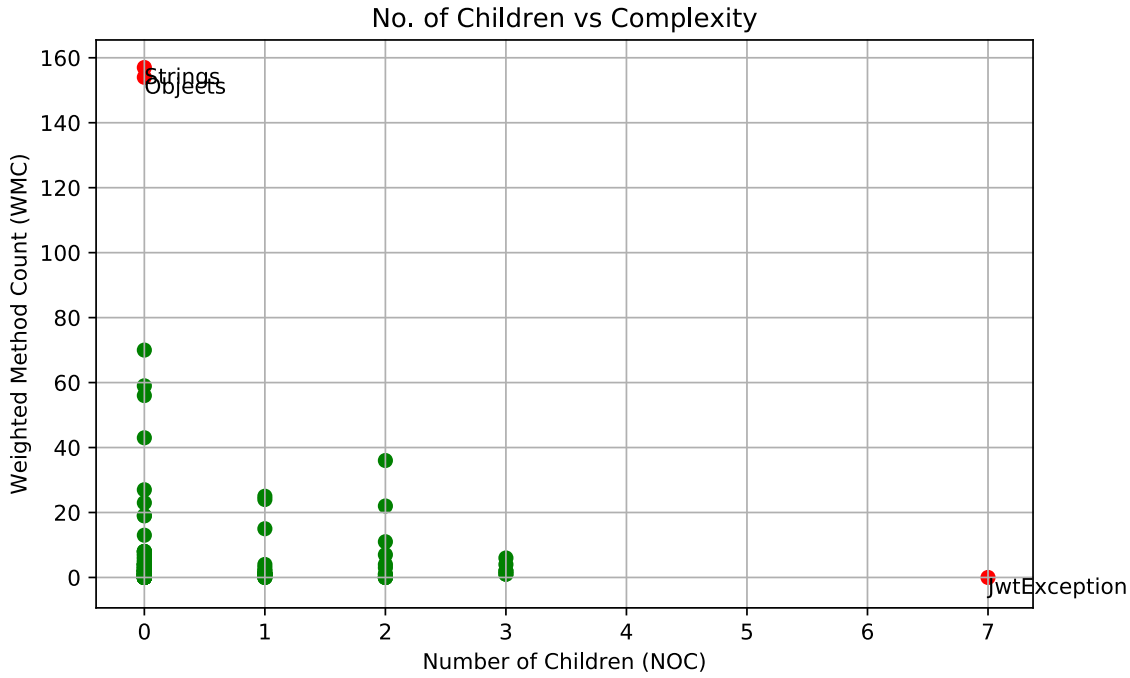

Figure 4: No. of Children vs Complexity

Figure 4 shows Strings and Objects classes are too complex, whereas, JwtException class has 7 depending classes. A minor change in these classes can have huge impact on the whole project. The remaining classes are good as they have few children and low complexity.

Conclusion

According to the analyses results, we found that JWT has few classes that are too big and complex. We advise to split them in order to improve cohesion and reduce complexity. Overall, JWT has very good design.