LOG8430E - Software Architecture and Advanced Design

Fall 2022

TP2: Design quality, bad quality and refactoring

By

Team Alpha

1949477 - Ming Xiao Yuan

1956576 - Laura Beaudoin

1957959 - Reetesh Dooleea

2168201 - Mohsen Rahmati

Submitted to: **Zoreh Sharafi**

November 8th 2022

Table of contents

Introduction

This report is an analysis of a software system Mango [1]. It's a web application for controlling and managing a set of sensors. The goal is to analyze its design quality by evaluating and measuring a set of quality metrics. It also concentrates on the anomalies of the system by using automated tools like MetricsTree and MetricsReloaded as well as the refactoring solutions to fix those problems.

1. Design Quality

This section contains the values of the different metrics calculated for the Mango application. We will explore metrics about cohesion, complexity, coupling and size. The tools used are MetricsTree and MetricsReloaded and they are specified for each metric. The standard deviation has been calculated using an external Excel sheet. It only concerns the *src* folder of the project.

1.1. Metrics

First of all, for the metrics, we analyzed some of them on a class level, and others on a package level or on a method level, depending on what was more appropriate. We mentioned it for each metric.

1.1.1. Cohesion metrics

The table 1.1 contains the metrics for the cohesion:

	Average	Maximum	Standard Deviation	Total
LCOM (MetricsTree)	2.95	41	3.27	1974
TCC (MetricsTree)	0.22	1	0.28	147

Table 1.1 Cohesion metrics for Mango

First of all, those metrics have both been calculated on a class level.

MetricTree defines the **Lack of Cohesion Of Methods (LCOM)** as the number of not connected method pairs in a class, which represents the independent parts, having no cohesion. Actually, it's the difference between the number of method pairs not having instance variables in common and the number of method pairs having common instance variables. So, the lower this value is, the better the cohesion is, and the better our application is, because it helps with the reliability, maintainability, reusability and portability. The goal is for the value to be close to 0. A value above 0.725 indicates a bad cohesion [2]. In Mango, we have a very bad average (2.95), and the maximum (41) is not acceptable.

On the other hand, when **Tight Class Cohesion (TCC)** increases, cohesion increases. It's the relative number of directly connected public methods in the class, measuring the cohesion between them. A low value can indicate the same quality issues as for LCOM. The value we obtained of this metric is coherent with what we had with LCOM, since we have an average of 0.22, showing once again that Mango is not really cohesive.

From those 2 metrics, we can conclude that classes have multiple responsibilities, violating the SRP principle. So, it is bad for the reliability, maintainability, reusability and portability of the code.

1.1.2. Complexity metrics

The table 1.2 contains the metrics relevant to the complexity of Mango:

	Average	Maximum	Standard Deviation	Total
CC (MetricsReloaded)	1.87	38	2.51	11 797
WMC (MetricsTree)	17.75	282	23.98	11 874
RFC (MetricsTree)	19.25	480	26.86	12 876

Table 1.2 Complexity metrics for Mango

First of all, the **McCabe Cyclomatic Complexity (CC)**, which has been calculated on a method level, is the number of independent paths in a method. The value can go from 1 to infinite, and the lower it is, the better. A low value of CC helps to have a good maintainability (analyzability, changeability and testability) and portability (adaptability and replaceability). This metric is low risk when it has a value between 1 and 10, but high risk for values between 21 and 50 [3]. For Mango, we have an excellent average (1.87), and the standard deviation is good, which suggests that we have a low risk complexity. But the maximum we have is too high and is not acceptable, and should be looked at, because it is high risk and will be hard to maintain and test, for example.

The **Weighted Methods per Class (WMC)** that was calculated on a class level is a weighted sum of the methods implemented in a class. For the complexity to be good (low), this metric also has to be low. Otherwise, it will affect the quality of the system the same way CC does. A common value for this metric is less than 11, casual is between 11 and 34 and uncommon is greater than 34 [2]. Our average is casual, but the maximum is uncommon. If we look at the standard deviation, we can also see that a lot of values are uncommon, so the 282 is not an exception. So, unlike for the CC metric, this one seems to say that our project's complexity is high, indicating for example a hard maintainability, understandability and testability.

Lastly, the **Response For Class (RFC)** is the number of public methods in a class and methods directly called by these. The values are integers from 0 to infinite, but the threshold value of this metric is 44. The lower the value is, the better (lower) the complexity of the code is. In Mango, the average is once again good, and the standard deviation suggests that most of the values are below 46, which is not far from the acceptable value of 44 [4]. Once again, we have to consider the fact that we have a value of 480, which is obviously not correct and definitely is too complex. But, we can still say that unlike WMC but like CC, the RFC metric suggests that the complexity is not bad, which makes it more understandable and maintainable.

In conclusion, since 2 of 3 metrics show that Mango has a good (low) complexity, it's fair to say that it's more positive than negative, but for the moment it's hard to make a firm conclusion if the understandability and maintainability of Mango are actually good or not, but it's probably safe to say that there are some violations of SRP in the code, because of high complexity in some methods or classes.

1.1.3. Coupling metrics

The table 1.3 presents the coupling metrics or the project:

	Average	Maximum	Standard Deviation	Total
CBO (MetricsTree)	10.96	194	16.85	7334
Ca (MetricsReloaded)	168.63	2259	378.53	18 718

Table 1.3 Coupling metrics for Mango

The first metric for coupling, **Coupling Between Objects (CBO)**, has been calculated on a class level and it represents the number of other classes that a class is coupled to. The threshold for it is 13, and the lower it is, the better (lower) the coupling of the source code is. When the value is low, it helps with reusability (understandability and attractiveness), with maintainability (analyzability, changeability, stability and testability) and adaptability. In our case, the average value (10.96) is not really good, especially combined with the standard deviation of 16.85, which means that a lot of values are above 13 [4]. Also, we notice the maximum of 194 which is absolutely horrible. As mentioned, this will negatively affect the reusability, the maintainability, and the adaptability of Mango.

The second metric is **Afferent Coupling (Ca)**, and it has been calculated on a package level. It is the number of external classes coupled to classes of a package. Obviously, the higher this value is, the higher the coupling is. A common value is 7 or less, a casual value is between 7 and 39 and a high value is higher than 39 [2]. We definitely have a bad coupling situation here, with our average of 168.63 and the even higher standard deviation of 378.53. The maximum of 2259 is also inacceptable. This metrics demonstrates once again that the coupling is way too

high, and it will probably result in a bad portability (replaceability) or maintainability of the application.

1.1.4. Size metrics

The last metrics for the application are about size and are listed in the table 1.4:

	Average	Maximum	Standard Deviation	Total
LOC (MetricsReleased)	84.47	2086	138.52	61 137
MLOC (MetricsReleased)	7.93	195	11.60	49 991
NOM (MetricsTree)	9.53	126	11.35	6372
SIZE2 (MetricsTree)	25.16	152	25.68	16 831

Table 1.4 Size metrics for Mango

The **Lines Of Code (LOC)** metric is simply the lines in the code, in our case in each class. In Mango, the average of 84.47 LOC for a class is pretty good, even with the standard deviation of 138.52. On the other hand, the maximum is absolutely not correct and could lead to bad understandability, maintainability (analysability, changeability, testability) and adaptability and is a sign of bad quality for that specific class. For the others, it seems acceptable for now.

The **Method Lines Of Code (MLOC)** metric is the same as LOC, but this time we calculated the number of lines in each method. Once again, the average (7.93) is good even when considering the standard deviation of 11.60. Although, the maximum is once again too high and could affect the understandability, maintainability (analysability, changeability, testability) and adaptability, but most of the time MLOC is acceptable.

The **Number of Methods (NOM)** metric represents the number of local methods in a class. We calculated it on a class level. To have a good size, the value must be 7 or lower, a casual value is between 7 and 14 and an uncommon value is 14 or more [2]. A low NOM is good for reusability (understandability, learnability and operability), maintainability (analyzability, changeability and testability) and portability (adaptability and replaceability). For Mango, the NOM is not good, because the average is 9.53, which is casual, but with the standard deviation of 11.35, a lot of values are above the threshold of 14, which means uncommon or bad value.

The **number of attributes and methods (SIZE2)** metric simply counts the number of attributes and methods in a class (NOM). It's a combination of NOM and NOA (Number Of Attributes). For NOM to be bad, it has to be greater than 14 [2] and for NOA, it has to be greater than 8. So, a simple estimation would be to say that a value above 22 for SIZE2 would be bad. Since we have an average of 25.16 for Mango, we can say that the size of classes, based on the number

of attributes and methods, is too big. This can reduce reusability (understandability, learnability, operability), maintainability (analysability, changeability, testability) and portability (adaptability, replaceability) of the system.

In conclusion, we can say that the size of Mango is not good. Even if the number of lines of code is acceptable for the majority of the methods or classes, there still are some too long methods or classes. Also, there are too many methods and attributes in the classes. The quality of the system is decreased on a reusability, maintainability and portability point of vue and it probably violates the Single Responsibility Principle many times.

1.2. Metrics for the ISO 9126 criteria

With MetricsTree, it's possible to get QMOOD Quality attributes values for the project (*src* folder). Table 1.5 presents the values calculated by MetricsTree:

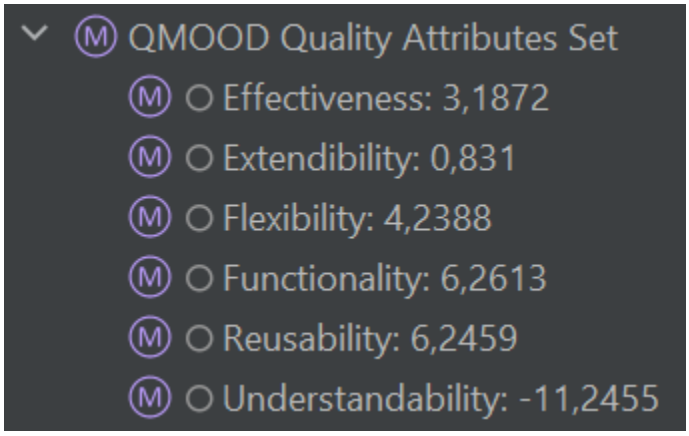

Figure 1.1 QMOOD Quality attributes values

There are no reference values to evaluate those metrics, but it is possible to suppose that a high value implies increased quality and that the comparison of those values together helps to determine which one is better than another. We will analyze and compare the 6 metrics one by one.

1.2.1. Reusability

According to MetricsTree, the reusability metric: "Reflects the presence of object-oriented design characteristics that allow a design to be reapplied to a new problem without significant effort." So, it's the capacity of a component to be used many times, by many systems. With a value of 6.2459, when compared to the values for the other criterias, we can suppose it's one one the attributes of Mango that is the less problematic. As we will see in section 1.3, this metric is related to coupling, cohesion, messaging and design size. It means that the coupling must be low (relationships between modules) and that the cohesion (relationships within modules), messaging and design size must be standard.

1.2.2. Flexibility

The metric of flexibility, according to MetricsTree, is the “Characteristics that allow the incorporation of changes in a design. The ability of a design to be adapted to provide functionally related capabilities.” Since that metric is evaluated at 4.2388, it could definitely be improved. As we will explore in the next section, it means that the encapsulation, composition and polymorphism must be acceptable and that the coupling must not be too high.

1.2.3. Understandability

The understandability of the system is described by MetricsTree as: “The properties of the design that enable it to be easily learned and comprehended. This directly relates to the complexity of the design structure”. It means that if anyone looks at the project or the code, they should not experience issues trying to understand or modify it. It leads to a better maintainability of the application. In the case of Mango, this is the worst quality, with a value of -11.2455. Therefore, we can consider it as the worst quality of Mango, which has really been neglected in the implementation. As explained in the 1.3 section, it means that abstraction, coupling, polymorphism, complexity and design size are too high and that encapsulation is probably low.

1.2.4. Functionality

To continue with the functionality metric, the definition given by MetricsTree is: “The responsibilities assigned to the classes of a design, which are made available by the classes through their public interfaces.” It results in a system providing the desired functionalities and therefore satisfying functional requirements. It has the highest metric value (6.2613) of all the qualities, but still, the value is not impressive. As we will analyze more in section 1.3, it means that cohesion, polymorphism, messaging, design size and hierarchies are probably normal.

1.2.5. Extendibility

The extensibility metric “Refers to the presence and usage of properties in an existing design that allow for the incorporation of new requirements in the design.” according to MetricsTree. In other words, it means that if that criteria is good, modules or components should be added easily to the project. With a value of 0.831, we can’t really consider this metric as good or bad. It probably means that abstraction, inheritance and polymorphism are not bad, and that coupling is not too high. That will be discussed in the next section of this report.

1.2.6. Effectiveness

Lastly, the effectiveness quality metric’s definition is “This refers to a design's ability to achieve the desired functionality and behavior using object-oriented design concepts and techniques.”, according to MetricsTree. This quality metric has a value of 3.1872 for Mango. It means that it’s normal. Therefore, the abstraction, encapsulation, composition, inheritance and polymorphism should be standard, as we will see in the next section.

1.3. Correlation between ISO criterias and metrics values

To calculate the quality attributes for the ISO 9126 criteria, we need to use the formulas presented in the table 1.5:

Quality Attribute	Index Computation Equation
Reusability	-0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * Design Size
Flexibility	0.25 * Encapsulation - 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism
Understandability	-0.33 * Abstraction + 0.33 * Encapsulation - 0.33 * Coupling + 0.33 * Cohesion - 0.33 * Polymorphism - 0.33 * Complexity - 0.33 * Design Size
Functionality	0.12 * Cohesion + 0.22 * Polymorphism + 0.22 Messaging + 0.22 * Design Size + 0.22 * Hierarchies
Extendibility	0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism
Effectiveness	0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism

Table 1.5 Formulas to calculate ISO 9126 criteria quality attributes

The metrics used in those formulas are listed in the table 1.6:

Design Property	Derived Design Metric
Design Size	Design Size in Classes (DSC)
Hierarchies	Number of Hierarchies (NOH)
Abstraction	Average Number of Ancestors (ANA)
Encapsulation	Data Access Metric (DAM)
Coupling	Direct Class Coupling (DCC)
Cohesion	Cohesion Among Methods in Class (CAM)
Composition	Measure of Aggregation (MOA)
Inheritance	Measure of Functional Abstraction (MFA)
Polymorphism	Number of Polymorphic Methods (NOP)
Messaging	Class Interface Size (CIS)
Complexity	Number of Methods (NOM)

Table 1.6 Metrics to calculate ISO 9126 criteria quality attributes

In MetricsTree, we can easily observe that some of those metrics have high or regular values:

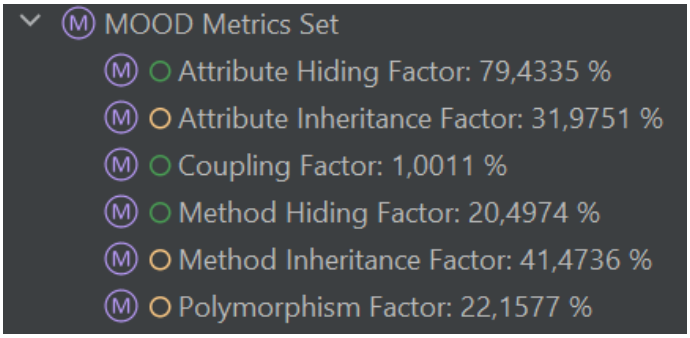

Figure 1.2 Metrics for encapsulation, coupling, inheritance, polymorphism

1.3.1. Reusability

As mentioned, reusability is related to coupling, cohesion, messaging and design size. In other words, it is affected negatively by DCC and positively by CAM, CIS and DSC.

For our first concrete example, in the case of Mango, even if the reusability value is one of the highest, it is still not that good compared to other applications, with a value of only 6.2459. We will see how the metrics we collected can affect the reusability of the system.

First of all, for the coupling, it's possible to rely on CBO and Ca to analyze it. In our program, we have high (bad) values for both of them, which respectively have averages of 10.96 and 168.63. This can explain why the value for reusability is not that good.

Also, the cohesion we got, according to LCOM (2.95) and TCC (0.22) is really not optimal, reducing once again the reusability of Mango.

With all those bad values for coupling and cohesion, since the reusability is not that bad, it can suggest that maybe messaging or design size, which have a bigger impact, are acceptable.

1.3.2. Flexibility

For the flexibility, as it is affected positively by encapsulation, composition and polymorphism and negatively by coupling. It means that DAM, MOA and NOP have a positive impact on flexibility but DCC has a negative impact. To relate the flexibility quality attribute to the metrics in section 1.1, it is important to note that high CBO and Ca also negatively affect the flexibility, because it increases coupling. Additionally, NOM also has an impact on flexibility, since it increases the polymorphism design property.

1.3.3. Understandability

On its side, a good understandability can be obtained by a low abstraction, coupling, polymorphism, complexity and design size and with a high encapsulation and cohesion. So, ANA, DCC, NOP, NOM and DSC must be low and DAM and CAM should be high to have a good understandability.

As a second concrete example, in the case of Mango, we have a negative (bad) value of -11.2455. As we analyzed earlier, it means that the understandability must not be good. Let's take a closer look at the metrics we have that affect the understandability of the system.

For the coupling, since DCC is similar to CBO and that we have a high value for CBO (average of 10.96 and standard deviation of 16.85), it is a first good sign that understandability will be affected. Also, the Ca, another metric related to coupling, is way too high, also indicating an issue with coupling and therefore understandability.

In addition, for the complexity, the NOM value that we have (average of 9.53 and standard deviation of 11.35) is too high, which once again, makes the understandability value lower because of a high complexity. Also, we have a mostly low risk CC (average of 1.87) and low RFC (average of 19.25), but the WMC (average of 17.75) value contradict that, meaning once again we have a complexity too high, leading to a bad understandability.

For a good understandability, you need good cohesion. Since we have a very high LCOM (2.95) and a low TCC (0.22), which both indicate a bad cohesion, we can deduce that the CAM value is not good and therefore, that justifies once again the bad understandability we have in Mango.

Finally, the polymorphism is affected by the NOM metric once again. Once again, with our high value, the polymorphism is increased and it reduces the understandability of the system.

1.3.4. Functionality

The functionality quality criteria is good when cohesion, polymorphism, messaging, design size and hierarchies are all high. So, CAM, NOP, CIS, DSC and NOH help have a better functionality. If we take a quick look at the metrics in section 1.1, LCOM and TCC affect functionality. The lower LCOM is and the higher TCC is, the better cohesion is and therefore the better functionality is. NOM also has an impact on functionality, since it increases the polymorphism design property.

1.3.5. Extendibility

Extendibility of a system is present when abstraction, inheritance and polymorphism are there and the coupling is low. In other words, it means that for example ANA, MFA and NOP have high values and DCC is low.

For our last concrete example of how our metrics affect the qualities of our system, let's look at the extendibility of Mango. With MetricsTree, we got a value of 0.831 for this quality, which was the second lowest, and which is really close to 0, meaning that it's not good. By using MetricsTree, we have all the metrics to calculate the 4 design properties of the system.

First, for the coupling, which decreases the extendibility, it is equivalent to CBO, which has an average value of 10.96, as seen in section 1.1, and it's relatively high, if we consider the standard deviation of 16.85. Coupling is also affected by the Ca metric, which is once again really high (168.63).

For the inheritance, that helps the code to be extendable, in MetricsTree, we can get the MIF (Method Inheritance Factor), which has a value of 41.47%, and it is labeled as a high value.

Finally, for the polymorphism, which also helps to get a good extendibility, we can get it again in MetricsTree by looking at the value of PF (Polymorphism Factor), which is 22,1577%, categorized as high, and the NOM value of 9.53 which is correct, but tending on the high side when considering the standard deviation of 11.34.

With all those bad values and also neutral values, we understand more easily why the extendibility of Mango could be improved by having especially a lower coupling and higher abstraction, since inheritance and polymorphism are already high according to MetricsTree.

1.3.6. Effectiveness

Lastly, effectiveness of the code means that abstraction, encapsulation, composition, inheritance and polymorphism are present. So, the values of ANA, DAM, MOA, MFA and NOP are relatively high. To make a correlation with the metrics in section 1.1, it means that NOM is high, resulting in higher polymorphism and then more effectiveness.

2. Anomalies

2.1. Definition of the metric threshold

It is possible to detect certain code antipatterns with metrics and conditions. In order to do that, we can define thresholds on the metrics and combine the relevant conditions and detect the antipattern. For any given metric, the thresholds are classed by Low, High and Very high. The following grid contains the metrics presented above and their thresholds. The low is calculated by subtracting the Standard deviation from the Average. If the value is negative, the Low value is 0. The High value is obtained by adding the Average and Standard deviation. Very high value is obtained through (Average+Standard deviation)*1.5.

Metric	Average	Standard deviation	Low	High	Very high
LCOM	2.95	3.27	0	6.22	9.33
TCC	0.22	0.28	0	0.5	0.75
CC	1.87	2.51	0	4.38	6.57
WMC	17.75	23.98	0	41.73	62.595
RFC	19.25	26.86	0	46.11	69.165
CBO	10.96	16.85	0	27.81	41.715
Ca	168.63	378.53	0	547.16	820.74
Ce	168.63	393.08	0	561.71	842.565
LOC	84.47	138.52	0	222.99	334.485
MLOC	7.93	11.6	0	19.53	29.295
SIZE2	25.16	25.68	0	50.84	76.26
NOM	9.53	11.35	0	20.88	31.32

Table 2.1: Metrics thresholds of Mango

In the detection of antipatterns with metrics, we also need to define conditions for each of the seven antipatterns, which are “Duplicated Code”, “Long Method”, “Large Class”, “Shotgun Surgery”, “Feature Envy”, “Refused Bequest” as well as “Long Parameter List”. The following grid contains the definitions of the threshold constants and it is taken from the “Bad Design” course material.

NONE	0
SEVERAL	2
FEW	2-5
Short Memory Cap	7
MANY	>=7

Table 2.2: Threshold definitions

2.2. Combining the metrics into rules

2.2.1. Duplicated Code

For the “Duplicated Code” antipattern, the rule is defined based on the figure 2.1 presented in the course material.

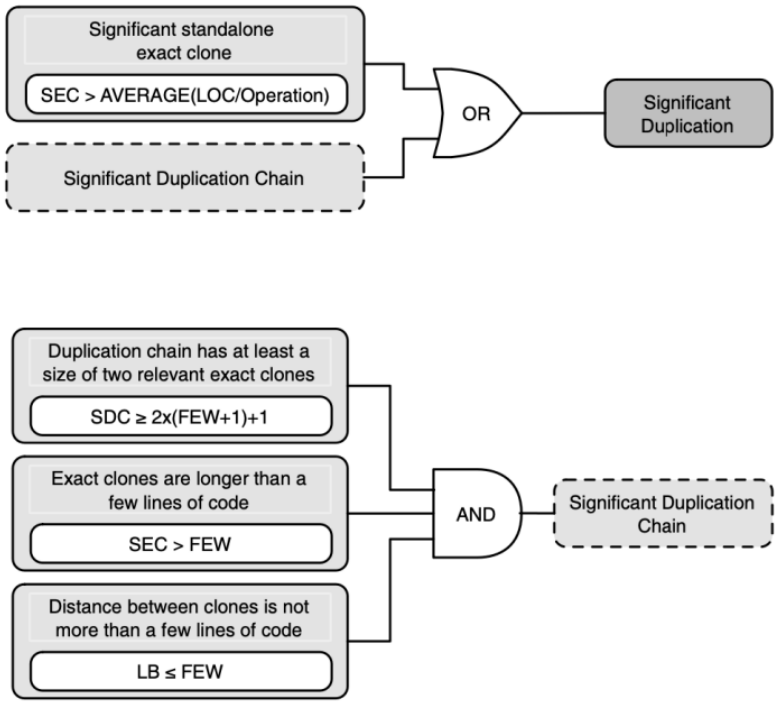

Figure 2.1: Duplicated Code

SEC means Size of exact clone, which represents the number of lines of exact code. LB means Line Bias and represents the distance between two exact clones. SDC means Size of Duplication Chain which represents the number of exact clones in a duplication chain. Based on figure 2.1, our conditions would look like

SEC > 84.47 *OR (SDC >= 7 AND SEC > 2 AND LB <= 4)*

2.2.2. Long Method

For the “Long Method” antipattern, the rule is defined based on the figure 2.2 presented in the course material.

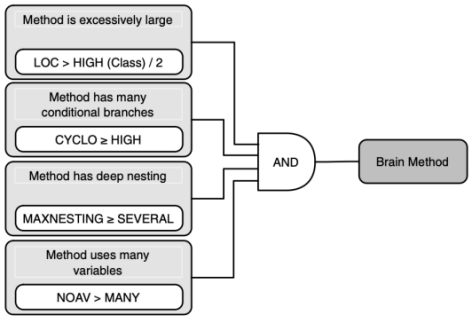

Figure 2.2: Long Method Logic

NOAV means Number of Accessed Variables and MAXNESTING means the Maximum Nesting Level. Based on figure 2.2, our conditions would look like

LOC > 9.765 AND CC >= 4.38 AND MAXNESTING >= 2 AND NOAV > 7

2.2.3. Large Class

For the “Large Class” antipattern, the rule is defined based on the figure 2.3 presented in the course material.

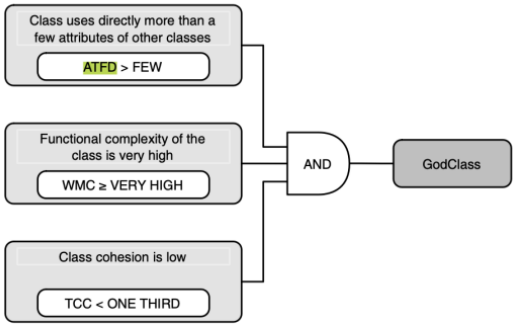

Figure 2.3: Large Class Logic

ATFD means Access to Foreign Data which represents the number of classes from which the class accesses attributes. Based on figure 2.3, our conditions would look like

ATFD > 3 AND WMC >= 62.595 *AND TCC < 1/3*

2.2.4. Shotgun Surgery

For the “Shotgun Surgery” antipattern, the rule is defined based on the figure 2.4 presented in the course material.

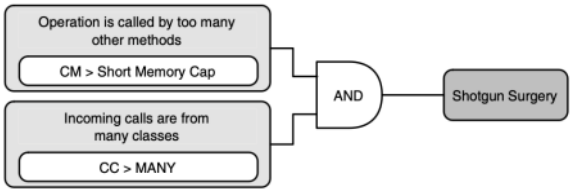

Figure 2.4: Shotgun Surgery Logic

CM means Changing Methods which represents the number of methods that call the measured method. CC means Changing Classes which represents the number of classes where the CM are defined. Based on figure 2.4, our conditions would look like

CM > 7 AND CC > 7

2.2.5. Feature Envy

For the “Feature Envy” antipattern, the rule is defined based on the figure 2.5 presented in the course material.

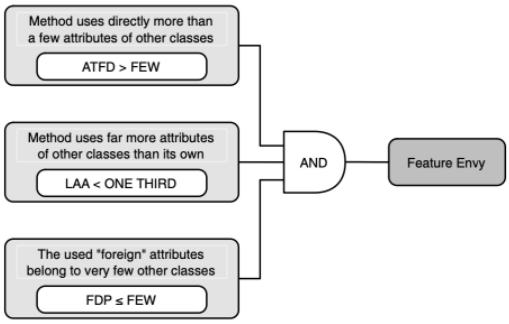

Figure 2.5: Feature Envy

LAA means Locality of Attribute Accesses which represents the number of attributes belonging to the class of the method divided by the total number of attributes accessed by the method. FDP means Foreign Data Providers which represents the number of classes where the ATFD attributes are defined. Based on figure 2.5, our conditions would look like

ATFD > 3 AND LAA < ⅓ AND FDP <=3

2.2.6. Refused Bequest

For the “Refused Bequest” antipattern, the rule is defined based on the figure 2.6 presented in the course material.

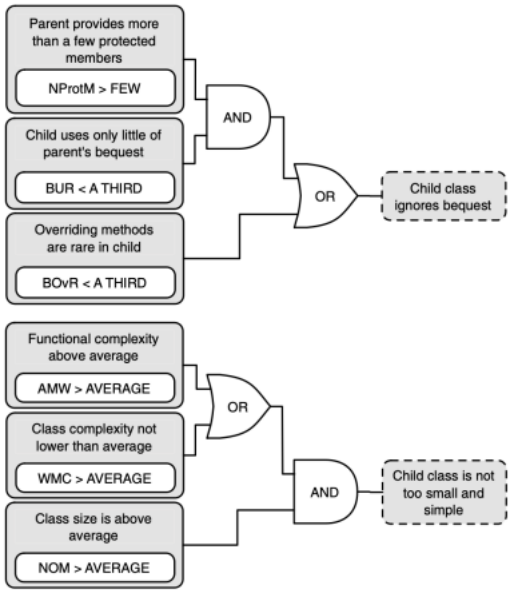

Figure 2.6: Refused Bequest Logic

NProtM means Number of Protected Members, BUR means Base-class Usage Ratio which represents the usage ratio of protected members. BOvR means Base-class Overriding Ratio which means the ratio of overridden members. AMW means Average Method Weight which means the average complexity of all methods of a class. Based on figure 2.6, our conditions based on child class ignores bequest would look like

(NProtM > 3 AND BUR < 1/3) OR BOvR < ⅓

2.2.7. Long Parameter List

For the “Long Parameter List” antipattern, intense research was performed on the web on popular software related websites such as Refactoring Guru [6] and Principles Wiki [7]. Thus, the rule to detect the Long Parameter List code smell is based on the NOPM metric which calculates the number of parameters in a class method or constructor. Normally, our research allowed us to conclude that the rule for the Long Parameter List code smell is as follows:

NOPM > FEW

2.3. Presence of anomalies

2.3.1. ViewDwr.java

The ViewDwr.java is located in *com/serotonin/mango/web/dwr* folder and it has a WMC of 110, ATFD of 24 and TCC of 0.0107. The regular range for WMC value is between 0 and 12, the regular range for ATFD is between 0 and 6 and the range for TCC is between 0.33 and 1. This class clearly surpasses all of them by a lot. If we apply it to one of our rules, we can see that it is clearly a “Large class” antipattern because it matches all three conditions. Therefore, it is safe to say we have a god class and it defeats the purpose of SRP.

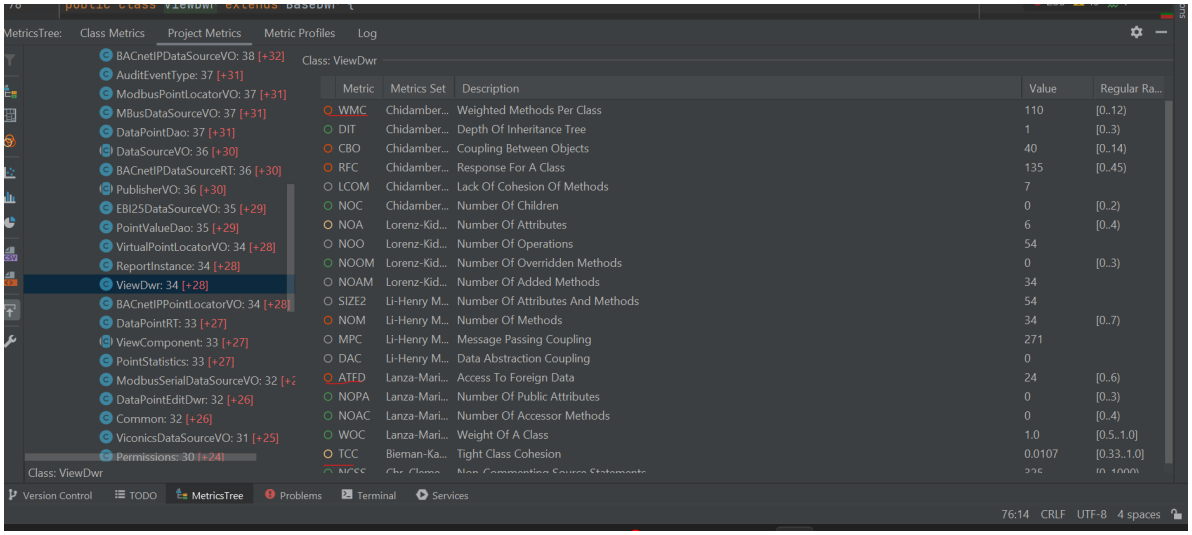

Figure 2.7: IntelliJ screenshot for ViewDwr.java

2.3.2. DataPointVO.java

The DataPointVO.java is located in *com/serotonin/mango/vo* folder and the method *jsonDeserialize* violates the Long Method antipattern. This method is clearly a Long Method because if we focus on the metrics, it has 66 for the LOC metric, which is bigger than the threshold value of 9.765. The CC metric is 18 which is far over the threshold of 4.107. The MAXNESTING of the method is equal to 2 and NOAV is way bigger than 7. These metrics tell us that this method is clearly too big and refactoring is needed.

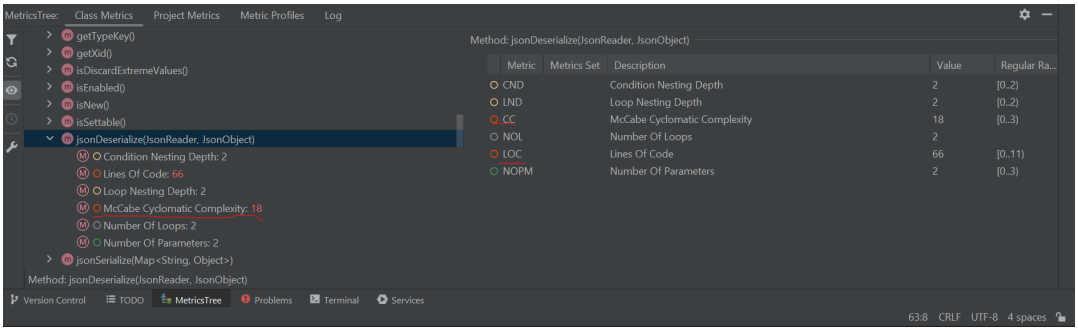

Figure 2.8: IntelliJ screenshot for DataPointVO.java

2.3.3. MaintenanceEventsDwr.java

The MaintenanceEventsDwr.java is located in *com/serotonin/mango/web/dwr* folder and the method saveMaintenanceEvent violates the Long Parameter List. The NOPM metric of this method is 21 which is way above FEW which corresponds to either 3, 4 or 5. Hence, this method definitely constitutes a Long Parameter List anomaly which makes the method hard to understand and to use as someone will most probably face difficulties to get the parameters in the right order.

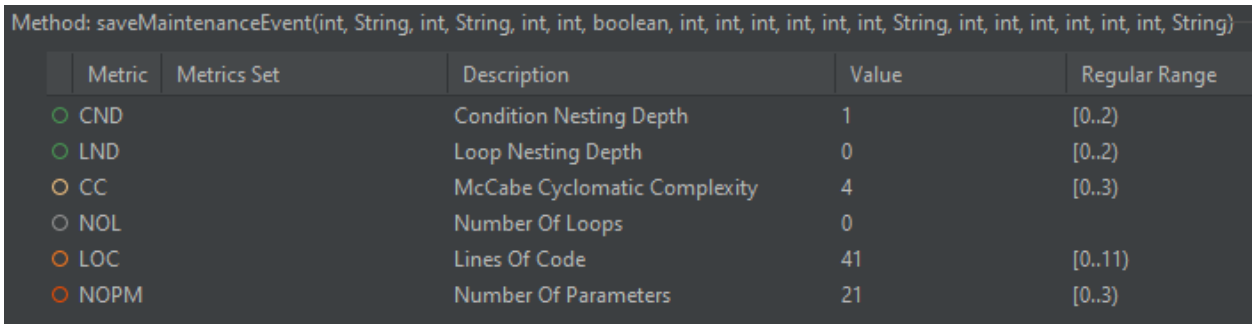

Figure 2.9: IntelliJ screenshot for MaintenanceEventsDwr.java

2.3.4. Common.java

The Common.java is located in *com/serotonin/mango* folder and one of its methods violates the Shotgun Surgery antipattern. The *getPeriod()* method has a Cylomatic complexity of 10 which is clearly bigger than the MANY threshold of 7. It also has a CM or Changing Methods metric value of 8, which is bigger than the Short Memory Cap threshold of 7. It means that this method is called way too many times my other methods. The figure 2.11 shows where the *getPeriod()* method gets called. With these violations, we can safely assume that this method has a Shotgun Surgery antipattern.

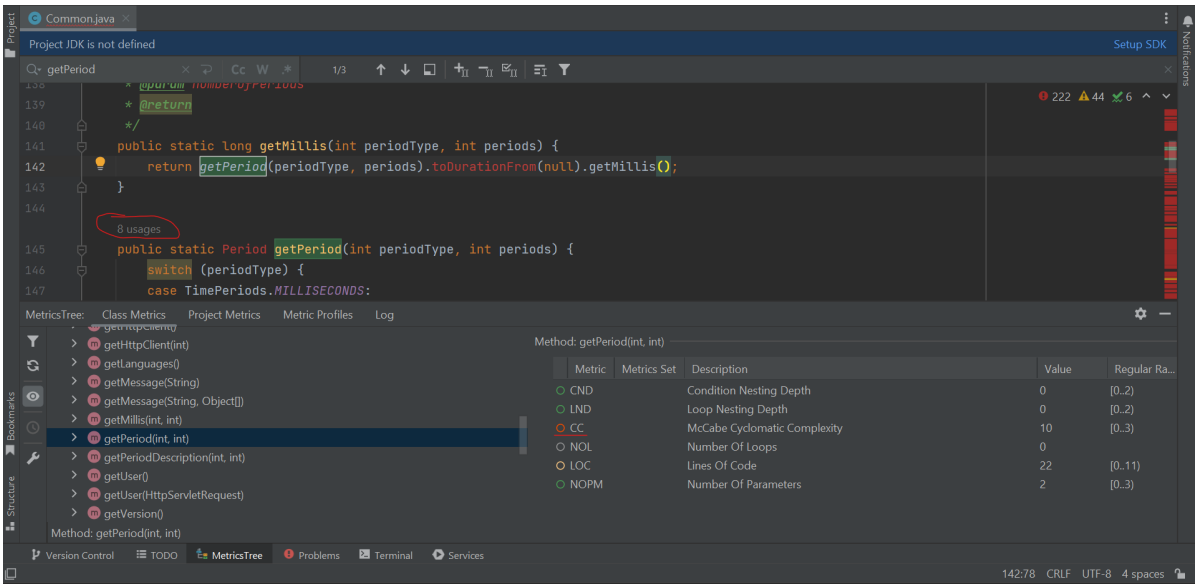

Figure 2.10: IntelliJ screenshot for Common.java

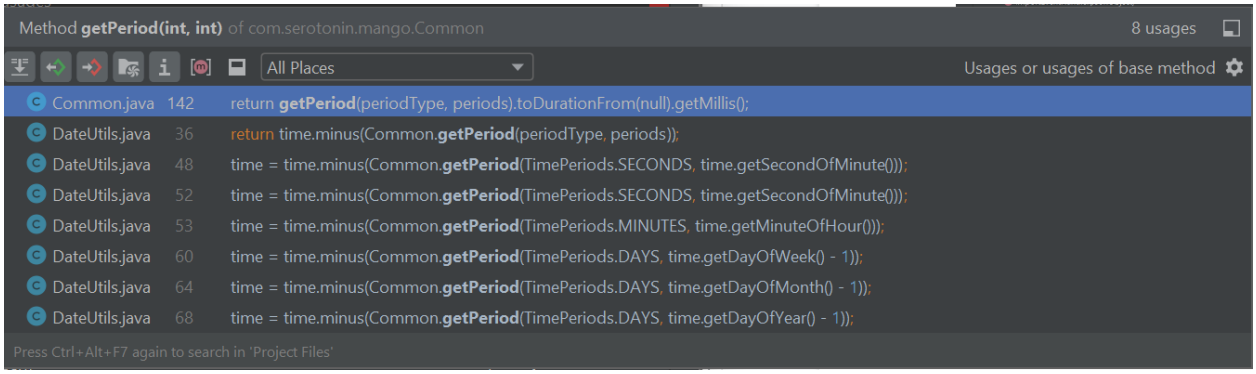

Figure 2.11: IntelliJ screenshot for usage of getPeriod()

2.3.5. EBI25InterfaceReader.java

The EBI25InterfaceReader.java is located in *com/serotonin/mango/web/dwr/beans* folder and this class violates the Feature Envy antipattern. The ATFD metric value of this class is 10 which is bigger than the threshold established in the rule of 3. The *removeOrCreatePoint()* method uses a lot of attributes of other classes than its own. This value can be evaluated with the LAA metric. The LAA metric is calculated by the usage of local attributes divided by the total attributes accessed by the method. This method uses 1 of its local attributes *dataPointDao* but it uses a lot of attributes from other classes. It has 25 access to three other classes *DataPointVO, EBI25PointLocatorVO* and *EBI25LoggerInfo.* It has respectively 6, 11 and 8 accesses to each three methods. We now have a LAA value of 1/25 which is smaller than 1/3. We can also see that the FDP metric value is equal to 3 classes. Therefore, we have a Feature Envy antipattern.

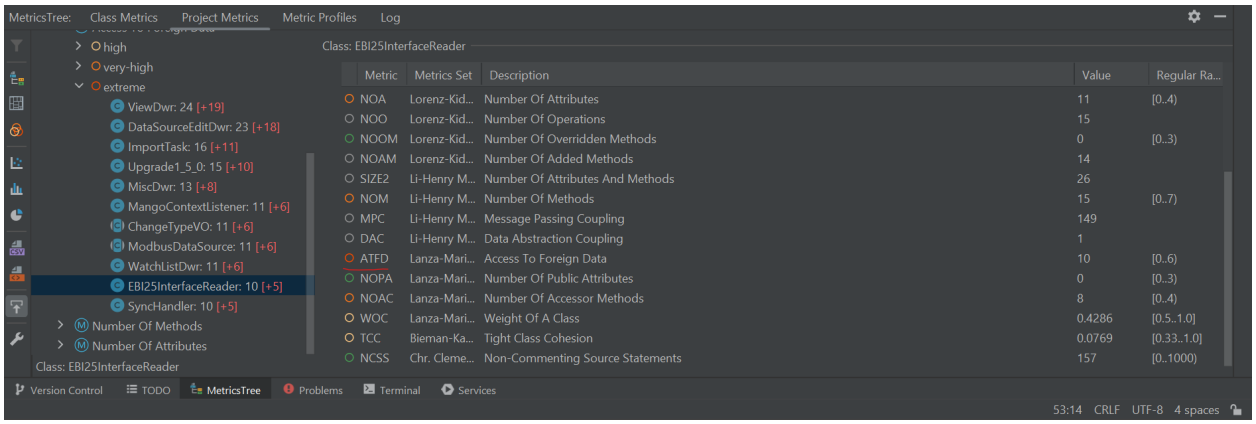

Figure 2.12: IntelliJ screenshot of EBI25InterfaceReader.java

3. Corrections

In this section, the main topic is related to refactoring of each anomaly identified in the previous section in order to reduce code smells. For each antipattern, an analysis of the refactoring will be presented in order to explain how the new changes consist of an improvement via refactoring. The latter will then be applied and detailed in order to evaluate the quality and measure the updated metrics of the Mango system again.

3.1. ViewDwr.java - Large class

3.1.1. Analysis of refactoring

This class has been identified as having an antipattern of “Large Class” which means that this class contains too many fields, methods, lines of code or simply, too many responsibilities. The strategy of the refactoring consists of extracting the ViewDwr class into classes for each distinct responsibility. Hence, by decomposing the large class into specific classes, this will reduce the WMC of the main class and orient it closer towards the Single Responsibility Principle.

3.1.2. Application of refactoring

First of all, the following figure illustrates the UML diagram of the code before performing the refactoring. As highlighted, we can see the class is overloaded with methods of different responsibilities.

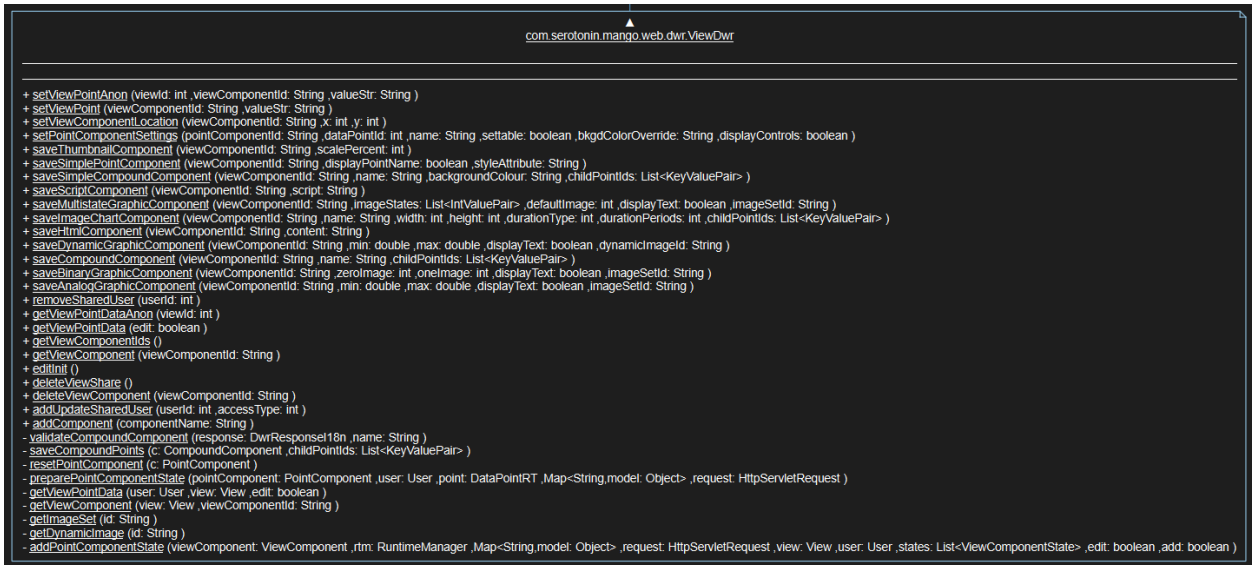

Figure 3.1: Initial implementation of the ViewDwd.java class

The above class diagram gives us a high-level overview of the class and allows us to figure out the common responsibilities. After an initial inspection of the source code, we observed that

there were several features in this class that had no cohesion amongst each other. So several subclasses were created to divide the responsibilities accordingly into unique classes.

For example, we notice that several methods concern the responsibility of *saving* a component. We identify that it is possible to save two groups of components: Graphic component and normal components. Hence, two classes are created: *GraphicSaveComponent* and *SaveBaseComponent*. Instantly, we observe that both classes have some methods in common. To eliminate duplicated code, inheritance is applied. A new abstract class named *AbstractSaveComponent* is created which is inherited by our two new classes.

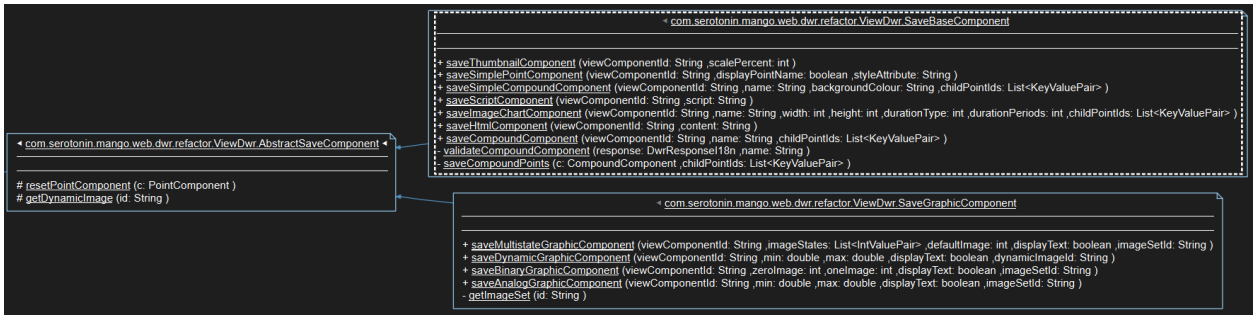

Figure 3.2: Addition of SaveBaseComponent and SaveGraphicComponent classes

Using the same logic, other responsibilities in the ViewDwr class were identified and extracted into their distinct corresponding classes such as *ViewUsers*, *ViewComponent* and *ViewPointDwr* where each class follows the Single Responsibility Principle. Finally, after the refactoring, the main ViewDwr.java class has been simplified as follows:

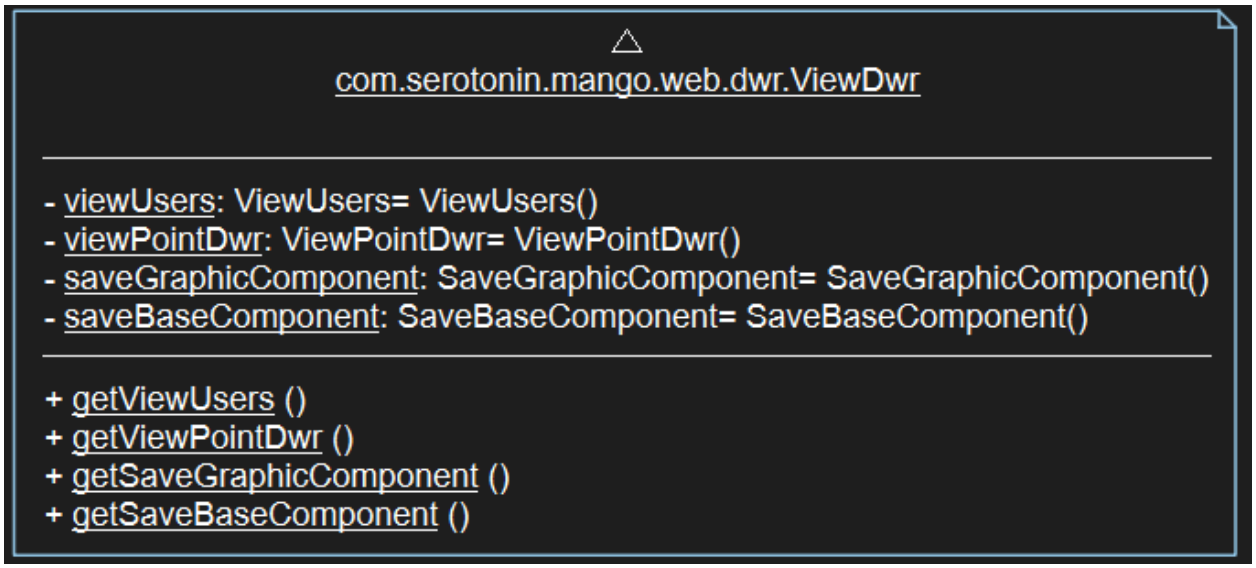

Figure 3.3: Refactoring of ViewDwr into several classes

```
public class ViewUsers {
    1 related problem
    protected List<User> getShareUsers(User excludeUser) {...}

    @MethodFilter
    public List<ShareUser> removeSharedUser(int userId) {...}

    protected String getMessage(String key) { return I18NUtils.getMessage(getResourceBundle(), key); }

    @MethodFilter
    public void deleteViewShare() {...}

    @MethodFilter
    public Map<String, Object> editInit() {...}
}
```

Figure 3.4: Example of a newly added class named ViewUsers

3.1.3. Evaluation of quality

After the refactor, the ViewDwr class no longer corresponds to a “Large class” as the metrics value has been improved.

Class: ViewDwr

M...	...	Description	Value	Regular Range
ATFD	...	Access To Foreign Data	0	[0..6)
CBO	...	Coupling Between Objects	6	[0..14)
DAC	...	Data Abstraction Coupling	4	
DIT	...	Depth Of Inheritance Tree	3	[0..3)
LCOM	...	Lack Of Cohesion Of Methods	4	
MPC	...	Message Passing Coupling	0	
NCSS	...	Non-Commenting Source Statements	4	[0..1000)
NOA	...	Number Of Attributes	10	[0..4)
NOAC	...	Number Of Accessor Methods	4	[0..4)
NOAM	...	Number Of Added Methods	4	
NOC	...	Number Of Children	0	[0..2)
NOM	...	Number Of Methods	4	[0..7)
NOO	...	Number Of Operations	42	
NOOM	...	Number Of Overridden Methods	0	[0..3)
NOPA	...	Number Of Public Attributes	0	[0..3)
RFC	...	Response For A Class	4	[0..45)
SIZE2	...	Number Of Attributes And Methods	45	
TCC	...	Tight Class Cohesion	0.0	[0.33..1.0]
WMC	...	Weighted Methods Per Class	4	[0..12)

Figure 3.5: Evaluation of the ViewDwr metrics after refactoring

As specified in section 2.2.3, a large class is based on the following conditions: *ATFD > 3 AND WMC* >= 62.595 *AND TCC* < ⅓. However, this condition no longer applies to this class since ATFD = 0 < 3. Furthermore, the WMC value of 4 also fits in the regular range. Thus, all the

metrics that forced this class to belong to a god class have been improved in order to fit the desired value range.

In parallel, the newly added classes also respect the individual metrics defined. For instance, the following figure illustrates the metrics of the *ViewUsers* class:

Class: ViewUsers

M...	Metrics Set	Description	Value	Regular Range
ATFD	Lanza-Mari...	Access To Foreign Data	5	[0..6)
CBO	Chidamber...	Coupling Between Objects	14	[0..14)
DAC	Li-Henry M...	Data Abstraction Coupling	0	
DIT	Chidamber...	Depth Of Inheritance Tree	1	[0..3)
LCOM	Chidamber...	Lack Of Cohesion Of Methods	3	
MPC	Li-Henry M...	Message Passing Coupling	34	
NCSS	Chr. Cleme...	Non-Commenting Source Statements	33	[0..1000)
NOA	Lorenz-Kid...	Number Of Attributes	0	[0..4)
NOAC	Lanza-Mari...	Number Of Accessor Methods	0	[0..4)
NOAM	Lorenz-Kid...	Number Of Added Methods	5	
NOC	Chidamber...	Number Of Children	0	[0..2)
NOM	Li-Henry M...	Number Of Methods	5	[0..7)
NOO	Lorenz-Kid...	Number Of Operations	18	
NOOM	Lorenz-Kid...	Number Of Overridden Methods	0	[0..3)
NOPA	Lanza-Mari...	Number Of Public Attributes	0	[0..3)
RFC	Chidamber...	Response For A Class	28	[0..45)
SIZE2	Li-Henry M...	Number Of Attributes And Methods	17	
TCC	Bieman-Ka...	Tight Class Cohesion	0.0	[0.33..1.0]
WMC	Chidamber...	Weighted Methods Per Class	12	[0..12)
WOC	Lanza-Mari...	Weight Of A Class	1.0	[0.5..1.0]

Figure 3.6: Evaluation of the ViewUsers class metrics

3.1.4. Measurement of metrics

This refactor has definitely improved the QMOOD Quality attributes values as proved by the following figure:

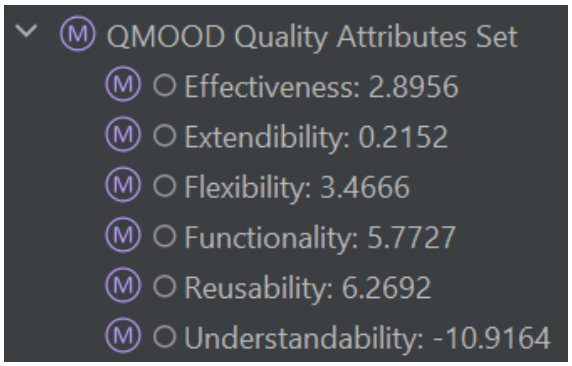

Figure 3.7: New measurement of the QMOOD Attributes set after refactor

The new individual metrics explain certainly that this class is now more **flexible** since this class was successfully extracted into several classes. The coupling is very low in this class. The fact that the new classes have unique single responsibilities, that means that they are **reusable**. The **extendibility** metrics have been improved a lot which means that new requirements or features can be easily integrated in this class. The **understandability** metric has also lowered, but it still signifies that this class followed with its members are not easily comprehended. The **functionality** and **effectiveness** have also been slightly improved.

3.2. DataPointVO.java - Long method

3.2.1. Analysis of refactoring

Firstly, the refactoring tactic chosen to fix the Long method antipattern is mainly the extraction of the method into classes and methods. The *jsonDeserialize* method in the *DataPointVO* class surpasses the threshold for LOC and CC which can be improved by making the method much smaller and less dense.

An inspection of the method allowed us to observe that the first half of the method is responsible for validating a string based on a code type. If the string is invalid, a *LocalizableJsonException* is thrown. Hence, this observation allows us to separate all the Json validation logic into its own custom class. As for the second half of the method, we notice that two tasks are being performed: one is to populate a Json object with a reader instance and another one is a simple assignment operation. Thus, to populate the JsonObject, we can shift its implementation into its own class. The assignment operation can be simply extracted to its own method. Finally, to keep the *jsonDeserialize* even neater, we can separate all these three distinct implementations into their own custom method and call them in the *jsonDeserialize* method.

3.2.2. Application of refactoring

Firstly, the following figure displays the initial long *jsonDeserialize* method before performing the refactoring:

```java
@Override
public void jsonDeserialize(JsonReader reader, JsonObject json) throws JsonException {
    String text = json.getString( name: "loggingType");
    if (text != null) {
        loggingType = LOGGING_TYPE_CODES.getId(text);
        if (loggingType == -1)
            throw new LocalizableJsonException("emport.error.invalid", "loggingType", text,
                    LOGGING_TYPE_CODES.getCodeList());
    }

    text = json.getString( name: "intervalLoggingPeriodType");
    if (text != null) {
        intervalLoggingPeriodType = Common.TIME_PERIOD_CODES.getId(text);
        if (intervalLoggingPeriodType == -1)
            throw new LocalizableJsonException("emport.error.invalid", "intervalLoggingPeriodType", text,
                    Common.TIME_PERIOD_CODES.getCodeList());
    }

    text = json.getString( name: "intervalLoggingType");
    if (text != null) {...}

    text = json.getString( name: "purgeType");
    if (text != null) {...}

    JsonObject locatorJson = json.getJsonObject("pointLocator");
    if (locatorJson != null)
        reader.populateObject(pointLocator, locatorJson);

    JsonArray pedArray = json.getJsonArray( name: "eventDetectors");
    if (pedArray != null) {
        for (JsonValue jv : pedArray.getElements()) {
            JsonObject pedObject = jv.toJsonObject();

            String pedXid = pedObject.getString( name: "xid");
            if (StringUtils.isEmpty(pedXid))
                throw new LocalizableJsonException("emport.error.ped.missingAttr", "xid");
```

Figure 3.8.1: First part of the initial jsonDeserialize method

```java
            // Use the ped xid to lookup an existing ped.
            PointEventDetectorVO ped = null;
            for (PointEventDetectorVO existing : eventDetectors) {
                if (StringUtils.isEqual(pedXid, existing.getXid())) {
                    ped = existing;
                    break;
                }
            }

            if (ped == null) {
                // Create a new one
                ped = new PointEventDetectorVO();
                ped.setId(Common.NEW_ID);
                ped.setXid(pedXid);
                ped.njbSetDataPoint(this);
                eventDetectors.add(ped);
            }

            reader.populateObject(ped, pedObject);
        }
    }

    text = json.getString( name: "engineeringUnits");
    if (text != null) {
        engineeringUnits = ENGINEERING_UNITS_CODES.getId(text);
        if (engineeringUnits == -1)
            engineeringUnits = ENGINEERING_UNITS_DEFAULT;
    }
}
```

Figure 3.8.2: Second part of the initial jsonDeserialize method

Firstly, to begin with the refactoring, as mentioned in the above section, the validation logic has been moved to a *JsonDeserializeValidation* class. However, all validation logic consists of the same similar step. Hence, a private method handles the validation of the text string passed and the exception error. The other methods are called by external clients, which, in our case, is the *jsonDeserialize* method in the *DataPointVO* class. All the methods in the new *JsonDeserializeValidation* class call the private method with the appropriate arguments.

```
public class JsonDeserializeValidation {
    private final ExportCodes loggingTypeCodes;
    private final ExportCodes intervalLoggingTypeCodes;
    private final JsonObject json;

    JsonDeserializeValidation(ExportCodes loggingTypeCodes, ExportCodes intervalLoggingTypeCodes, JsonObject json) {
        this.loggingTypeCodes = loggingTypeCodes;
        this.json = json;
        this.intervalLoggingTypeCodes = intervalLoggingTypeCodes;
    }

    private int validate(String jsonText, int type, ExportCodes code, String exception) throws LocalizableJsonException {
        String text = json.getString(jsonText);
        if (text != null) {
            type = code.getId(text);
            if (type == -1)
                throw new LocalizableJsonException("emport.error.invalid", exception, text, code.getCodeList());
        }
        return type;
    }

    int validateLoggingType(int loggingType) throws LocalizableJsonException {
        return this.validate( jsonText: "loggingType", loggingType, this.loggingTypeCodes,  exception: "loggingType");
    }

    int validateIntervalLoggingPeriodType(int intervalLoggingPeriodType) throws LocalizableJsonException {
        return this.validate( jsonText: "intervalLoggingPeriodType", intervalLoggingPeriodType, Common.TIME_PERIOD_CODES,  exception: "intervalLoggingPeriodType");
    }

    int validateIntervalLoggingType(int intervalLoggingType) throws LocalizableJsonException {
        return this.validate( jsonText: "intervalLoggingType", intervalLoggingType, this.intervalLoggingTypeCodes,  exception: "intervalLoggingType");
    }

    int validatePurge(int purgeType) throws LocalizableJsonException {
        return this.validate( jsonText: "purgeType", purgeType, Common.TIME_PERIOD_CODES,  exception: "purgeType");
    }
}
```

Figure 3.9: Implementation of the new JsonDeserializeValidation class

Secondly, another new class named *JsonPointEventDetector* was implemented. This class takes a *JsonReader* and *JsonObject* as parameters when constructing the class. It handles the logic for populating a locator and a PointEventDetector. The public methods *populateLocator* and *populatePed* are called from within the *jsonDeserialize* method.

```java
public class JsonPointEventDetector {
    private final JsonReader reader;
    private final JsonObject json;

    JsonPointEventDetector(JsonReader reader, JsonObject json) {
        this.reader = reader;
        this.json = json;
    }

    private PointEventDetectorVO getPed(List<PointEventDetectorVO> eventDetectors, DataPointVO dataPointVO, String pedXid) {
        PointEventDetectorVO ped = null;
        for (PointEventDetectorVO existing : eventDetectors) {
            if (StringUtils.isEqual(pedXid, existing.getXid())) {
                ped = existing;
                break;
            }
        }

        if (ped == null) {
            ped = createNewPed(eventDetectors, dataPointVO, pedXid);
        }
        return ped;
    }

    private PointEventDetectorVO createNewPed(List<PointEventDetectorVO> eventDetectors, DataPointVO dataPointVO, String pedXid) {
        PointEventDetectorVO ped = new PointEventDetectorVO();
        ped.setId(Common.NEW_ID);
        ped.setXid(pedXid);
        ped.njbSetDataPoint(dataPointVO);
        eventDetectors.add(ped);
        return ped;
    }

    private String getPedXid(JsonObject pedObject) throws LocalizableJsonException {
        String pedXid = pedObject.getString( name: "xid");
        if (StringUtils.isEmpty(pedXid))
            throw new LocalizableJsonException("emport.error.ped.missingAttr", "xid");
        return pedXid;
    }
```

Figure 3.10.1: First part (private methods) of the newly created JsonPointEventDetector class

```java
void populateLocator(PointLocatorVO pointLocator) throws JsonException {
    JsonObject locatorJson = json.getJsonObject("pointLocator");
    if (locatorJson != null)
        reader.populateObject(pointLocator, locatorJson);
}

void populatePed(List<PointEventDetectorVO> eventDetectors, DataPointVO dataPointVO) throws JsonException {
    JsonArray pedArray = json.getJsonArray( name: "eventDetectors");
    if (pedArray != null) {
        for (JsonValue jv : pedArray.getElements()) {
            JsonObject pedObject = jv.toJsonObject();
            PointEventDetectorVO ped = getPed(eventDetectors, dataPointVO, getPedXid(pedObject));
            reader.populateObject(ped, pedObject);
        }
    }
}
```

Figure 3.11.2: Second part (public methods) of the newly created JsonPointEventDetector class

Finally, here is how the *jsonDeserialize* method looks after decomposing the method into other classes and methods.

```
@Override
public void jsonDeserialize(JsonReader reader, JsonObject json) throws JsonException {
    JsonDeserializeValidation jsonValidation = new JsonDeserializeValidation(LOGGING_TYPE_CODES, INTERVAL_LOGGING_TYPE_CODES, json);
    loggingType = jsonValidation.validateLoggingType(loggingType);
    intervalLoggingPeriodType = jsonValidation.validateIntervalLoggingPeriodType(intervalLoggingPeriodType);
    intervalLoggingType = jsonValidation.validateIntervalLoggingType(intervalLoggingType);
    purgeType = jsonValidation.validatePurge(purgeType);

    JsonPointEventDetector jsonPED = new JsonPointEventDetector(reader, json);
    jsonPED.populateLocator(pointLocator);
    jsonPED.populatePed(eventDetectors, dataPointVO: this);

    String text = json.getString( name: "engineeringUnits");
    if (text != null) {
        engineeringUnits = ENGINEERING_UNITS_CODES.getId(text);
        if (engineeringUnits == -1)
            engineeringUnits = ENGINEERING_UNITS_DEFAULT;
    }
}
```

Figure 3.12: A peek of the jsonDeserialize method after some refactoring

However, the *jsonDeserialize* can be improved even more by extracting the three distinct logics into their own methods as illustrated by the following figure.

```
@Override
public void jsonDeserialize(JsonReader reader, JsonObject json) throws JsonException {
    this.validateJsonDeserialize(json);
    this.populateJsonPED(reader, json);
    this.updateJsonEngineeringUnits(json);
}

private void validateJsonDeserialize(JsonObject json) throws LocalizableJsonException {
    JsonDeserializeValidation jsonValidation = new JsonDeserializeValidation(LOGGING_TYPE_CODES, INTERVAL_LOGGING_TYPE_CODES, json);
    loggingType = jsonValidation.validateLoggingType(loggingType);
    intervalLoggingPeriodType = jsonValidation.validateIntervalLoggingPeriodType(intervalLoggingPeriodType);
    intervalLoggingType = jsonValidation.validateIntervalLoggingType(intervalLoggingType);
    purgeType = jsonValidation.validatePurge(purgeType);
}

private void populateJsonPED(JsonReader reader, JsonObject json) throws JsonException {
    JsonPointEventDetector jsonPED = new JsonPointEventDetector(reader, json);
    jsonPED.populateLocator(pointLocator);
    jsonPED.populatePed(eventDetectors, dataPointVO: this);
}

private void updateJsonEngineeringUnits(JsonObject json) {
    String text = json.getString( name: "engineeringUnits");
    if (text != null) {
        engineeringUnits = ENGINEERING_UNITS_CODES.getId(text);
        if (engineeringUnits == -1)
            engineeringUnits = ENGINEERING_UNITS_DEFAULT;
    }
}
```

Figure 3.13: Final look of the jsonDeserialize method after complete refactoring

3.2.3. Evaluation of quality

Visually, we can observe that the lines of code of the *jsonDeserialize* method have significantly reduced and overall, the method is much smaller and cleaner as it contains only three lines of code. Most of the associated metrics have improved drastically as well.

Method: jsonDeserialize(JsonReader, JsonObject)

M...	Metrics Set	Description	Value	Regular Range
CC		McCabe Cyclomatic Complexity	1	[0..3)
CND		Condition Nesting Depth	0	[0..2)
LND		Loop Nesting Depth	0	[0..2)
LOC		Lines Of Code	6	[0..11)
NOL		Number Of Loops	0	
NOPM		Number Of Parameters	2	[0..3)

Figure 3.14: Evaluation of the metrics of jsonDeserialize method after refactoring

As illustrated by the above figure, the new value of the LOC went from 66 to 6 which is an excellent improvement. It is indeed below our threshold value of 9.765. The CC metric is now only 1 which indicates that our refactoring was a huge success and the *jsonDeserialize* no longer consists of a long method. Similarly, the metrics of all the newly added classes and methods signify that there is no presence of code smell. Few examples are illustrated below.

Method: validateJsonDeserialize(JsonObject)

M...	Metrics Set	Description	Value	Regular Range
CC		McCabe Cyclomatic Complexity	1	[0..3)
CND		Condition Nesting Depth	0	[0..2)
LND		Loop Nesting Depth	0	[0..2)
LOC		Lines Of Code	7	[0..11)
NOL		Number Of Loops	0	
NOPM		Number Of Parameters	1	[0..3)

Figure 3.15: Evaluation of metrics of the validateJsonDeserialize method of DataPointVO class

Method: populatePed(List<PointEventDetectorVO>, DataPointVO)

M...	Metrics Set	Description	Value	Regular Range
CC		McCabe Cyclomatic Complexity	3	[0..3)
CND		Condition Nesting Depth	1	[0..2)
LND		Loop Nesting Depth	1	[0..2)
LOC		Lines Of Code	10	[0..11)
NOL		Number Of Loops	1	
NOPM		Number Of Parameters	2	[0..3)

Figure 3.16: Evaluation of metrics of the populatePed method of JsonPointEventDetector class

3.2.4. Measurement of metrics

Once again, after performing this successful refactor, the QMOOD Quality attributes were recalculated as followed:

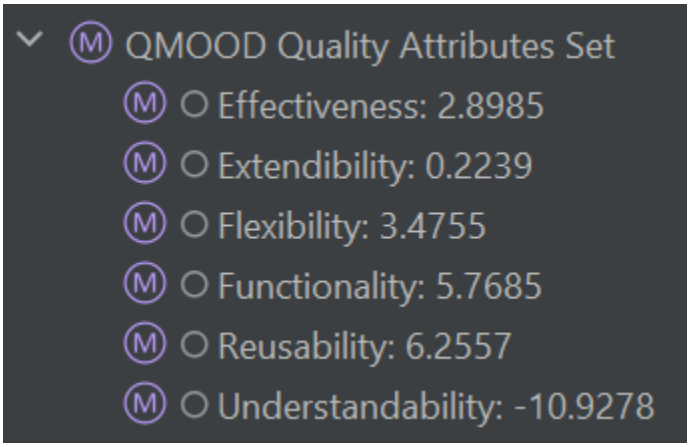

Figure 3.17: New measurement of the QMOOD Attributes set after refactor

We notice that less metrics have changed after the measurement from section 3.1.4. They are all similar to the last measurement which still means that these new changes consist of improvements when compared to those of the initial Mango system.

3.3. MaintenanceEventsDwr.java - Long Parameter List

3.3.1. Analysis of refactoring

This class consists of a violation of the Long Parameter List as the NOPM, which stands for number of parameters, of the *saveMaintenanceEvent* method is 21. In order to perform a refactor of this method, a parameter object can be introduced and applied. This strategy will significantly reduce the number of parameters as all parameters will be encapsulated in a single object. This approach will also improve the LOC metric as few lines of code will be removed since a local object in the method is currently created with the help of the passed parameters. Finally, the method will no longer consist of a violation of the Long Parameter List antipattern.

3.3.2. Application of refactoring

The following figure displays the initial *saveMaintenanceEvent* method before performing the refactoring. Visually, we can deduce that this method is overloaded with ambiguous parameters and the lines of code are exaggerated.

```java
public DwrResponseI18n saveMaintenanceEvent(int id, String xid, int dataSourceId, String alias, int alarmLevel,
        int scheduleType, boolean disabled, int activeYear, int activeMonth, int activeDay, int activeHour,
        int activeMinute, int activeSecond, String activeCron, int inactiveYear, int inactiveMonth,
        int inactiveDay, int inactiveHour, int inactiveMinute, int inactiveSecond, String inactiveCron) {
    Permissions.ensureAdmin();

    MaintenanceEventVO e = new MaintenanceEventVO();
    e.setId(id);
    e.setXid(xid);
    e.setDataSourceId(dataSourceId);
    e.setAlias(alias);
    e.setAlarmLevel(alarmLevel);
    e.setScheduleType(scheduleType);
    e.setDisabled(disabled);
    e.setActiveYear(activeYear);
    e.setActiveMonth(activeMonth);
    e.setActiveDay(activeDay);
    e.setActiveHour(activeHour);
    e.setActiveMinute(activeMinute);
    e.setActiveSecond(activeSecond);
    e.setActiveCron(activeCron);
    e.setInactiveYear(inactiveYear);
    e.setInactiveMonth(inactiveMonth);
    e.setInactiveDay(inactiveDay);
    e.setInactiveHour(inactiveHour);
    e.setInactiveMinute(inactiveMinute);
    e.setInactiveSecond(inactiveSecond);
    e.setInactiveCron(inactiveCron);

    DwrResponseI18n response = new DwrResponseI18n();
    MaintenanceEventDao maintenanceEventDao = new MaintenanceEventDao();

    if (StringUtils.isEmpty(xid))
        response.addContextualMessage( contextKey: "xid",  contextualMessageKey: "validate.required");
    else if (!maintenanceEventDao.isXidUnique(xid, id))
        response.addContextualMessage( contextKey: "xid",  contextualMessageKey: "validate.xidUsed");

    e.validate(response);

    // Save the maintenance event
    if (!response.getHasMessages()) {
        Common.ctx.getRuntimeManager().saveMaintenanceEvent(e);
        response.addData("meId", e.getId());
    }

    return response;
}
```

Figure 3.18: Initial saveMaintenanceEvent method

As mentioned, in order to perform the refactor, a parameter object can be introduced and this will replace all current parameters of the *saveMaintenanceEvent* method with a single object. This keeps the code cleaner, shorter and more comprehensive. However, we observe that a class named *MaintenanceEventVO* already exists for this purpose and is used to create an object instance in this method with all the 21 parameters. Hence, we can directly pass the *MaintenanceEventVO* object as a parameter itself rather than passing all its properties as parameters. The following figure illustrates this point:

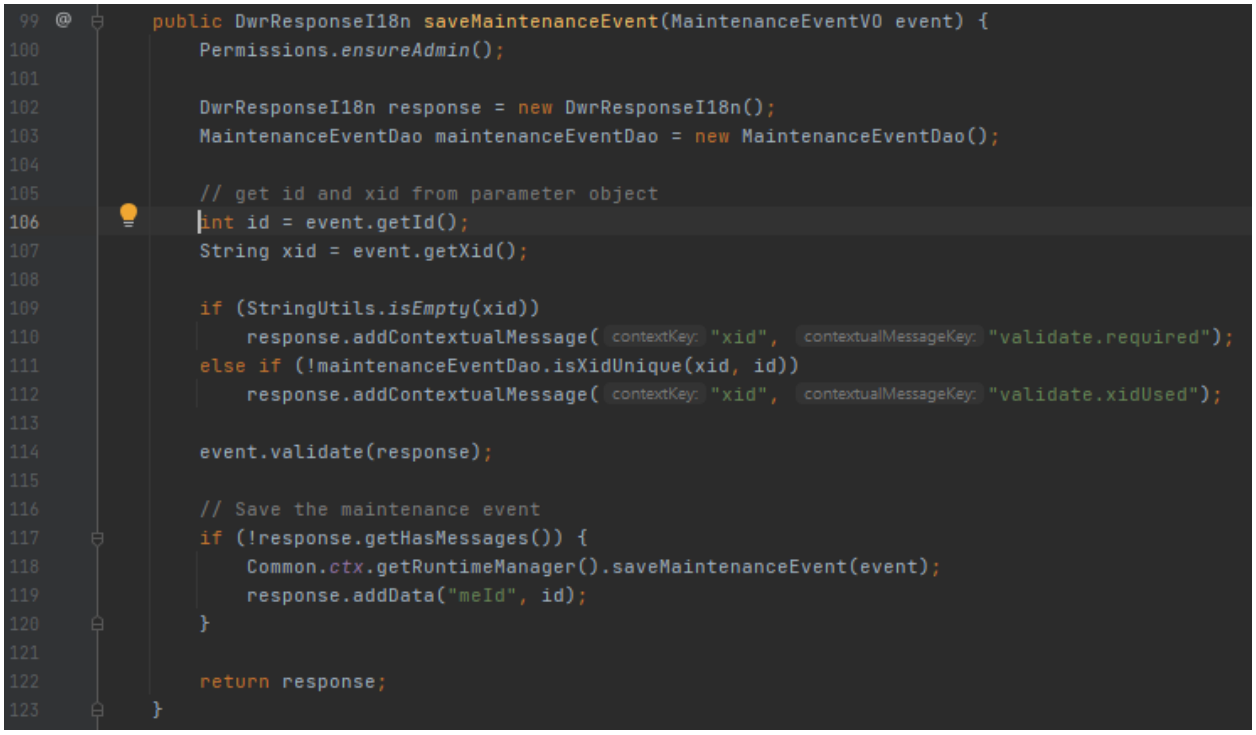

Figure 3.19: Refactoring of saveMaintenanceEvent method

As we can observe, the number of parameters decreased to only 1 with the help of the parameter object. The parameter object replaced all the numerous vague parameters. In case some properties and values of the parameter object are needed elsewhere in the method, getters can be implemented to retrieve the value of those elements as shown on line 106 and 107 of the above figure. An example of a getter of an object is illustrated below:

```
public int getId() {
    return id;
}
```

Figure 3.20: Example of a getter to retrieve the value of a private variable named 'id'

3.3.3. Evaluation of quality

The metrics of the *saveMaintenanceEvent* method have been calculated again to prove that the refactor was a success. As expected, the NOPM and LOC values have remarkably reduced:

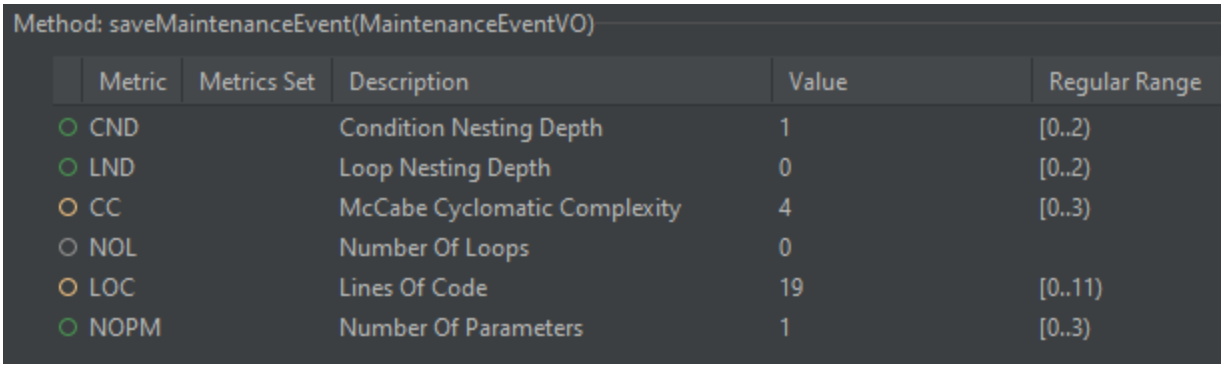

Figure 3.21: Evaluation of the metrics of the saveMaintenanceEvent method after refactoring

To conclude, the final NOPM metric value after refactoring is officially 1, as shown by the MetricsTree tool. Similarly, the LOC has improved from 41 to 19 just by introducing a parameter object and by removing the creation of a local MaintenanceEventVO object in the method with the passed parameter object. We can thus also conclude that the anomaly has been corrected and this method no longer suffers from the Long Parameter List code smell.

3.3.4. Measurement of metrics

For this section, the new QMOOD Quality attributes metrics have been recalculated after performing the refactoring as defined in the above section. We can observe that the values for the quality attributes are similar to those of the last refactor which proves that the removal of the anomaly was indeed positive. Furthermore, we can also notice that the value of Extendibility has improved a lot from 0.2239 to 0.0398. Similarly, the value of Effectiveness, Flexibility, Functionality and Reusability are better than the previous measurement from section 3.2.4 which indicates again that the quality of the Mango system has been improved.

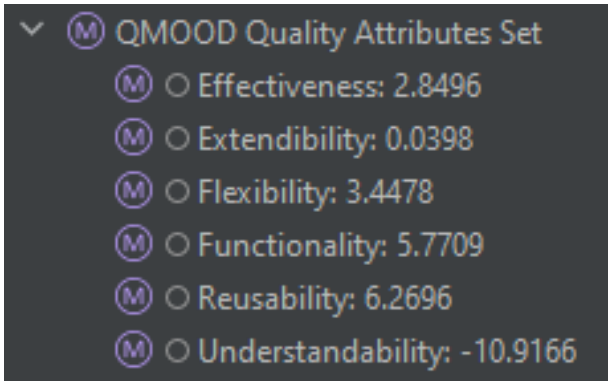

Figure 3.22: New measurement of the QMOOD Attributes set after refactor

3.4. Common.java - Shotgun Surgery

3.4.1. Analysis of refactoring

So, with refactoring, we need to decrease the value of CC (Cylomatic complexity) and the value of CM (Changing Methods). We know that the number of cases in the switch case in the *getPeriod* method affects the value of CC. So, we need to do the switch case in different functions to decrease the value of CC. Also, doing switch cases in different functions decreases the value of CM because all numbers calling getPeriod() will be divided between different functions. Hence, each new function has the least call and the value of CM.

3.4.2. Application of refactoring

This class has interfaces, functions one of which is getPeriod() and it has the situation of Shotgun Surgery antipattern because of a Cyclomatic Complexity (CC) of 10 and a Changing Methods (CM) of 8. I prepare the UML of Common class and the control flow of the getPeriod method that shows the Shotgun Surgery antipattern.

Figure 3.23: The initial UML of the Common Class

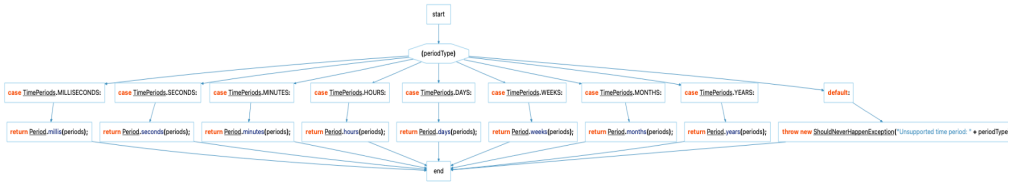

Figure 3.24: The initial control flow of the getPeriod method

In the following figures, you can see that we could divide the getPeriod function into three sub functions: getPeriod_one, getPeriod_two and getPeriod_three. This decreases the value of CC and CM as we mentioned in the previous part.

```
145     public static Period getPeriod_one(int periodType, int periods) {
146         switch (periodType) {
147             case TimePeriods.SECONDS:
148                 return Period.seconds(periods);
149             case TimePeriods.MINUTES:
150                 return Period.minutes(periods);
151             case TimePeriods.HOURS:
152                 return Period.hours(periods);
153             default:
154                 throw new ShouldNeverHappenException("Unsupported time period: " + periodType);
155         }
156     }
157         public static Period getPeriod_two(int periodType, int periods)
158         {
159             switch (periodType) {
160                 case TimePeriods.WEEKS:
161                     return Period.weeks(periods);
162                 case TimePeriods.MONTHS:
163                     return Period.months(periods);
164                 case TimePeriods.YEARS:
165                     return Period.years(periods);
166                 default:
167                     throw new ShouldNeverHappenException("Unsupported time period: " + periodType);
168             }
169         }
```

Figure 3.25: The changes of getPeriod() for solving Shotgun Surgery

```
170     public static Period getPeriod_three(int periodType, int periods)
171     {
172         switch (periodType) {
173             case TimePeriods.MILLISECONDS:
174                 return Period.millis(periods);
175             case TimePeriods.DAYS:
176                 return Period.days(periods);
177             default:
178                 throw new ShouldNeverHappenException("Unsupported time period: " + periodType);
179         }
180     }
```

Figure 3.26: The changes of getPeriod() for solving Shotgun Surgery

3.4.3. Evaluation of quality

The metrics that need to be recalculated are *CM* and *CC.* Both of them are below the threshold of 7. Because of the division of the function, it causes a decrease in calling each function. So, the final *CM* metric is below 7.

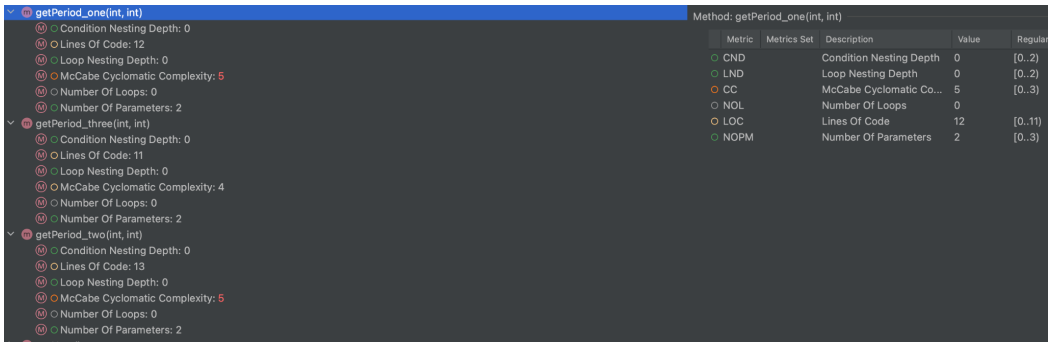

Figure 3.27: The changes of getPeriod() metrics

3.4.4. Measurement of metrics

We recalculated the QMOOD for this antipattern. All the previous refactorings from section 3.1 to 3.3 were not considered. This QMOOD is just about this refactoring of Shotgun Surgery of getPeriod(). We do observe there is a slight improvement in the Extendibility metric.

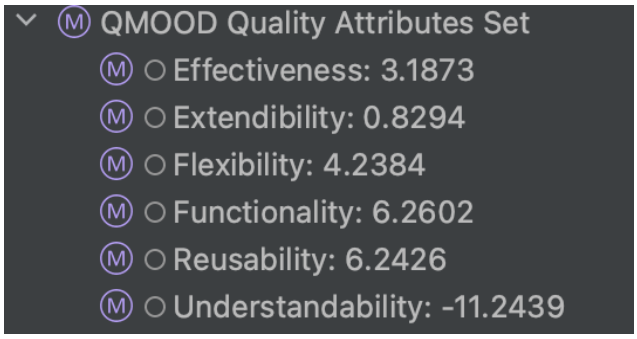

Figure 3.28: The changes of QMOOD after solving Shotgun Surgery

3.5. EBI25InterfaceReader.java - Feature Envy

3.5.1. Analysis of refactoring

The class of EBI25InterfaceReader has the ATFD metric (Access To Foreign Data) of 10 which shows the Feature Envy anti pattern. So, with the help of refactoring, we need to decrease the access to the forging classes. So, we try to remove the functionality which uses some of foreign classes to decrease ATFD.

3.5.2. Application of refactoring

At first, we mentioned the UML of the class of EBI25InterfaceReader. Also, they show the forgein accesses.

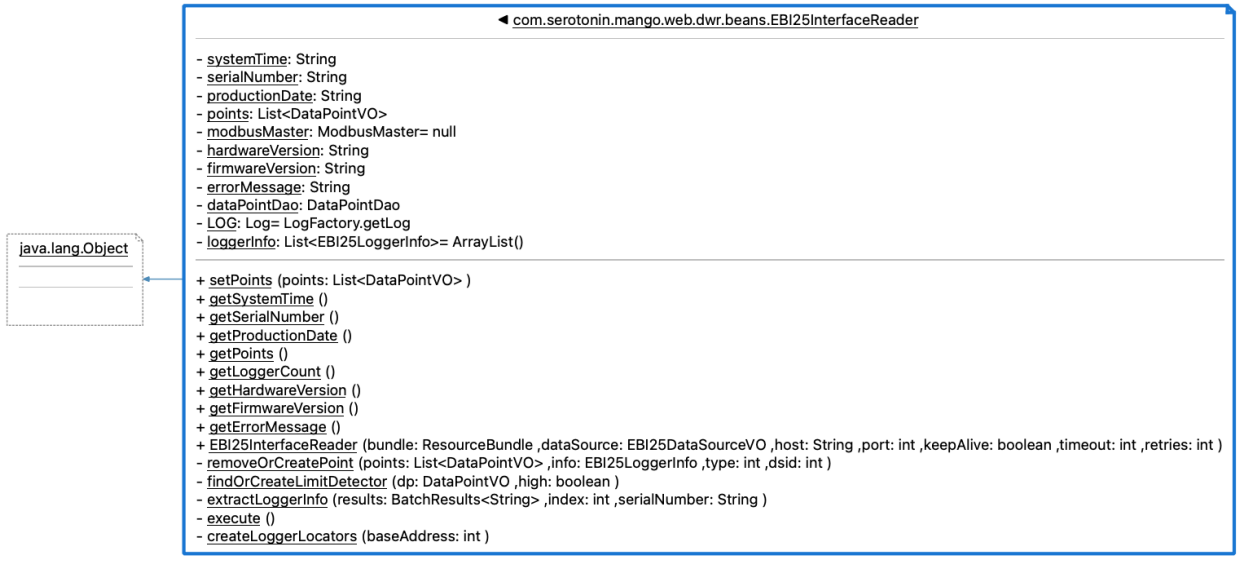

Figure 3.29: The initial UML of the EBI25InterfaceReader *Class*

Also, we mention the control flow of this class.

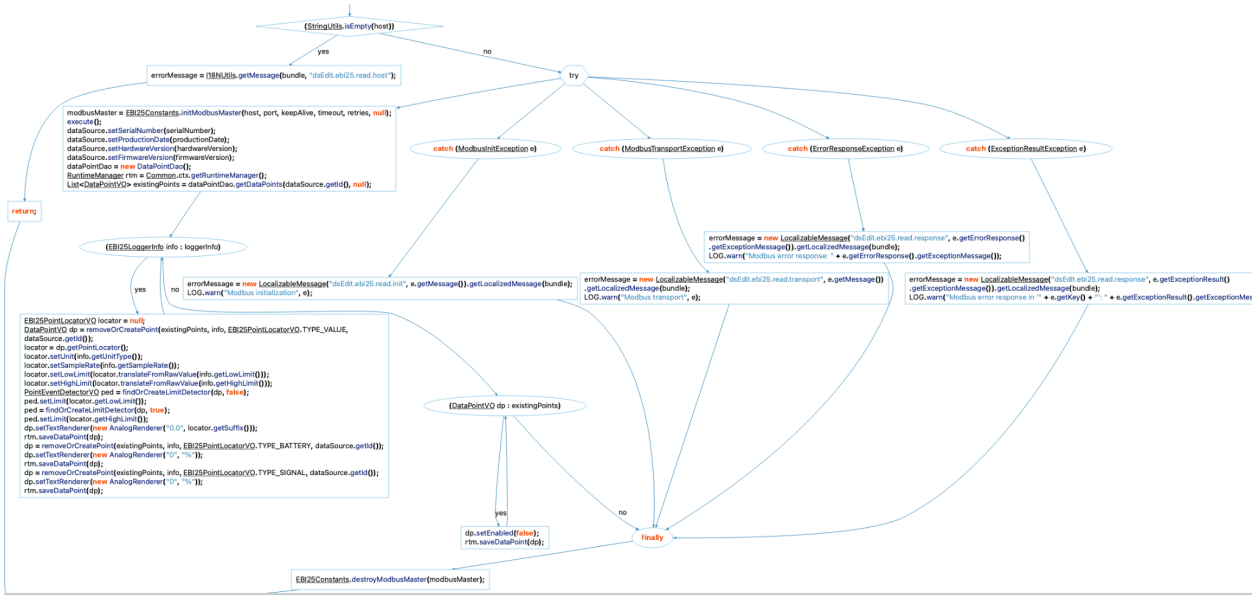

Figure 3.30: The initial Control Flow of the EBI25InterfaceReader *Class*

In this EBI25InterfaceReader class, we have three accesses from classes *DataPointVO, EBI25PointLocatorVO* and *EBI25LoggerInfo.* We try to remove the access of *EBI25LoggerInfo* to decrease the access to the forgein data (ATFD metric). This refactoring method helps to solve the problem of feature envy. Also, we will do the functionality of the access of *EBI25LoggerInfo*

in a new class. If we need the output of this new class, we just use the output one time instead of using *EBI25LoggerInfo* frequently or we need to create another class like EBI25InterfaceReader which can use *EBI25LoggerInfo* and do the removed functionality.

Except for the Feature Envy of EBI25InterfaceReader, If we want to solve the Feature Envy *removeOrCreatePoint()* method, we have 25 access from three classes including respectively 6, 11 and 8 accesses to classes *DataPointVO, EBI25PointLocatorVO* and *EBI25LoggerInfo.* So, we need to divide the functionality of EBI25InterfaceReader into nine classes and the forgein access will be 3, 3, 3, 3, 2, 3, 3, 3 and 2 (3+3+3+3+2+3+3+3+2 = 25). We could do it by divide the accesses of *DataPointVO into two classes* (3 and 3), the accesses of *EBI25PointLocatorVO into three classes* (3, 3, and 2), and the accesses of *EBI25LoggerInfo into four classes* (3, 3, 3, and 2). So, ATFD metric value for each class is 1 and LAA is 1/3 or 1/2 for each class. So we could solve the Feature Envy completely for the *removeOrCreatePoint* method.

In the following figure, you can see that we commented the part of the code which uses *EBI25LoggerInfo.* We do not prepare 6 classes because it is not possible to do it because of different dependencies. So we explained our solution with detail and also we divide accesses from *EBI25LoggerInfo.* But our solution consider all metrics and situations.

```
53  public class EBI25InterfaceReader {
        4 usages
54      private static final Log LOG = LogFactory.getLog(EBI25InterfaceReader.class);
55
        5 usages
56      private ModbusMaster modbusMaster = null;
        6 usages
57      private String errorMessage;
        3 usages
58      private String serialNumber;
        3 usages
59      private String productionDate;
        3 usages
60      private String hardwareVersion;
        3 usages
61      private String firmwareVersion;
        2 usages
62      private String systemTime;
63  //    private final List<EBI25LoggerInfo> loggerInfo = new ArrayList<EBI25LoggerInfo>();
```

Figure 3.31: The removed parts of EBI25InterfaceReader *Class for decreasing access to EBI25LoggerInfo*

```
91      //          for (EBI25LoggerInfo info : loggerInfo) {
92      //              EBI25PointLocatorVO locator = null;
93      //
94      //              //
95      //              // Create or update the logger value point.
96      //              DataPointVO dp = removeOrCreatePoint(existingPoints, info, EBI25PointLocatorVO.TYPE_VALUE,
97      //                      dataSource.getId());
98      //
99      //              locator = dp.getPointLocator();
100     //              locator.setUnit(info.getUnitType());
101     //              locator.setSampleRate(info.getSampleRate());
102     //              locator.setLowLimit(locator.translateFromRawValue(info.getLowLimit()));
103     //              locator.setHighLimit(locator.translateFromRawValue(info.getHighLimit()));
104     //
105     //              // Define low and high limit detectors.
106     //              PointEventDetectorVO ped = findOrCreateLimitDetector(dp, false);
107     //              ped.setLimit(locator.getLowLimit());
108     //
109     //              ped = findOrCreateLimitDetector(dp, true);
110     //              ped.setLimit(locator.getHighLimit());
```

Figure 3.32: The removed parts of EBI25InterfaceReader *Class for decreasing access to EBI25LoggerInfo*

```
111     //
112     //              // Update the text renderer
113     //              dp.setTextRenderer(new AnalogRenderer("0.0", locator.getSuffix()));
114     //
115     //              rtm.saveDataPoint(dp);
116     //
117     //              //
118     //              // Create or update the logger battery point.
119     //              dp = removeOrCreatePoint(existingPoints, info, EBI25PointLocatorVO.TYPE_BATTERY, dataSource.getId());
120     //              dp.setTextRenderer(new AnalogRenderer("0", "%"));
121     //              rtm.saveDataPoint(dp);
122     //
123     //              //
124     //              // Create or update the logger signal point.
125     //              dp = removeOrCreatePoint(existingPoints, info, EBI25PointLocatorVO.TYPE_SIGNAL, dataSource.getId());
126     //              dp.setTextRenderer(new AnalogRenderer("0", "%"));
127     //              rtm.saveDataPoint(dp);
128     //          }
```

Figure 3.33: The removed parts of EBI25InterfaceReader *Class for decreasing access to EBI25LoggerInfo*

```
290     //    private EBI25LoggerInfo extractLoggerInfo(BatchResults<String> results, int index, String serialNumber)
291     //            throws ExceptionResultException {
292     //        EBI25LoggerInfo info = new EBI25LoggerInfo();
293     //
294     //        info.setIndex(index);
295     //        info.setSerialNumber(serialNumber);
296     //        info.setProductionDate(EBI25Constants.getDateResult(results, "proddate"));
297     //        info.setCalibrationDate(EBI25Constants.getDateResult(results, "caldate"));
298     //        info.setHardwareVersion(EBI25Constants.getVersionResult(results, "hwver"));
299     //        info.setFirmwareVersion(EBI25Constants.getVersionResult(results, "fwver"));
300     //        info.setUnitType(EBI25Constants.getIntResult(results, "unit"));
301     //        info.setSampleRate(EBI25Constants.getIntResult(results, "rate"));
302     //        info.setLowLimit(EBI25Constants.getIntResult(results, "low"));
303     //        info.setHighLimit(EBI25Constants.getIntResult(results, "high"));
304     //
305     //        return info;
306     //    }
```

Figure 3.34: The removed parts of EBI25InterfaceReader *Class for decreasing access to EBI25LoggerInfo*

3.5.3. Evaluation of quality

As you can see in the following figures, we have EBI25InterfaceReader in the extreme part of Access to Forgein Data but after refactoring it is removed completely.

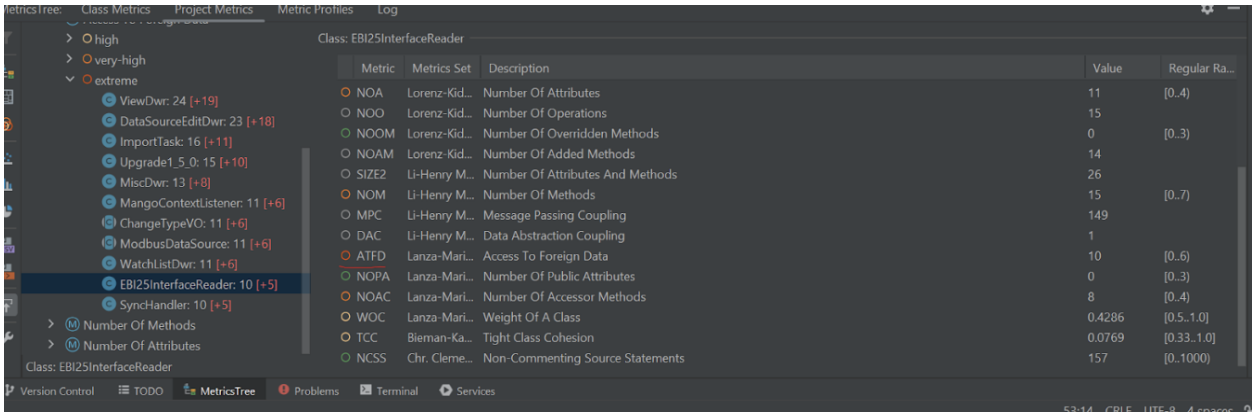

Figure 3.35: The metrics of EBI25InterfaceReader *Class before refactoring*

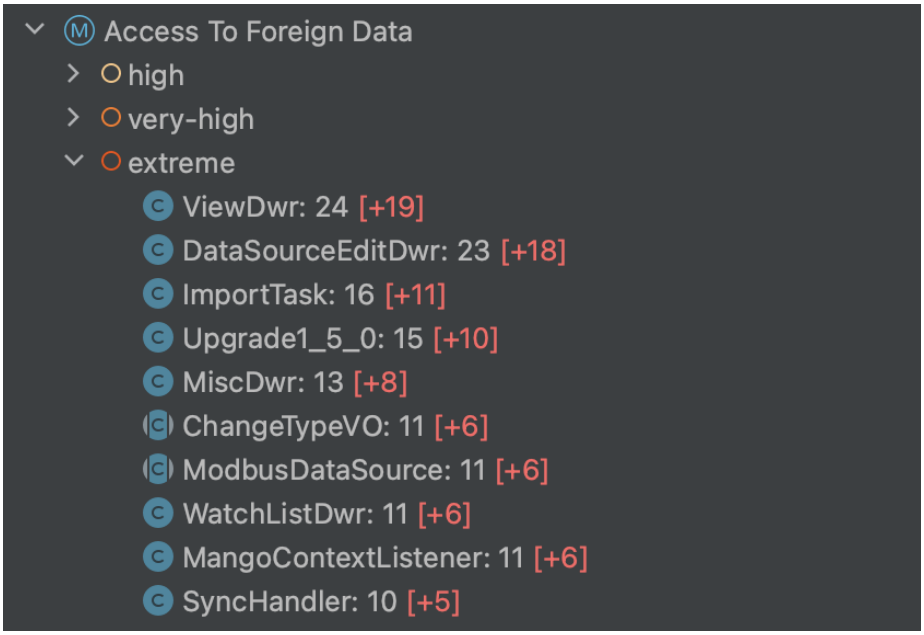

Figure 3.36: The metrics of EBI25InterfaceReader *Class after refactoring*

3.5.4. Measurement of metrics

We mentioned QMOOD after this refactoring. We do not consider the improvements from section 3.1 to 3.4. In this QMOOD, we just consider the changes about Feature Envy antipattern after refactoring by dividing the class. We could see the improvements in Effectiveness.

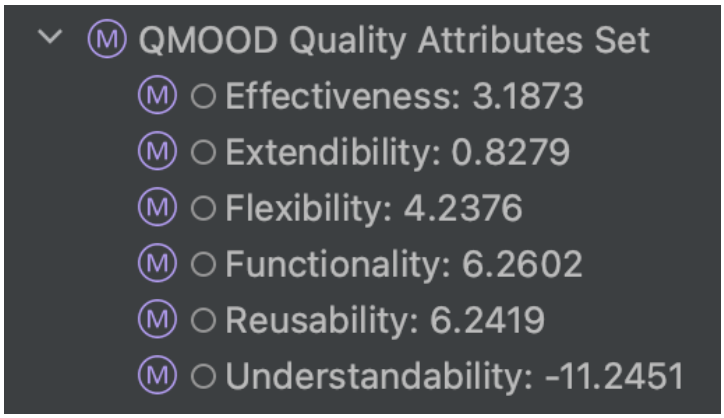

Figure 3.37: The QMOOD of EBI25InterfaceReader *Class after refactoring by dividing the class*

References

[1] Link for Mango source code:
https://drive.google.com/file/d/1sec7Ky9xzzjqyLG7VA0Jx0LBXsIEUJgh/view?usp=sharing

[2] Filó, T.G., & Bigonha, M. (2015). A Catalogue of Thresholds for Object-Oriented Software Metrics. Available:
https://www.semanticscholar.org/paper/A-Catalogue-of-Thresholds-for-Object-Oriented-Fil%C3%B3-Bigonha/1c17571ff1a0072010e11b36b83e5cdfb25ff203. [Accessed 3-Nov-2022]

[3] “Cyclomatic Complexity (McCabe) Metric,” *GMetrics*. [Online]. Available:
https://dx42.github.io/gmetrics/metrics/CyclomaticComplexityMetric.html#:~:text=Interpreting%20Cyclomatic%20Complexity%20Values,too%20complex%20(higher%20risk). [Accessed 4-Nov-2022].

[4] Shatnawi, Raed & Li, Wei & Swain, James & Newman, Tim. (2010). Finding software metrics threshold values using ROC curves. Journal of Software Maintenance. 22. 1-16. 10.1002/smr.404.

[5] “Large Class,” *Refactoring Guru.* [Online]. Available:
https://refactoring.guru/smells/large-class. [Accessed 27-Oct-2022].

[6] “Long Parameter List,” Refactoring Guru. [Online]. Available:
https://refactoring.guru/smells/long-parameter-list. [Accessed 7-Oct-2022].

[7] “Long Parameter List,” Principle Wiki. [Online]. Available:
http://principles-wiki.net/anti-patterns:long_parameter_list. [Accessed 7-Oct-2022].