Assignment #2
LOG8430e - Software Architecture and Advanced Design

Trimester: Fall 2021

Yankees

Alice Gong 1961605

Yuhan Li 1947497

Yanis Toubal 1960266

Group: 01

Submitted to :
Mohammadreza Rasolroveicy

Polytechnique Montréal
November 8th 2021

Question 1: Measure the design quality of AmazeFile

a)

Table 1. Size metrics

	Average	Maximum	Standard Deviation	Total
LOC (MetricsReloaded)	124.700	1897	222.500	40155
SIZE2 (MetricsTree)	82.410	1219	182.432	20520
NOM (MetricsTree)	7.406	75	9.875	1844

As its name indicates, size metrics focus on the size of a software. There are different ways to determine its size, whether it be counting the lines of executable code, counting the number of methods in a class, etc. Size oriented metrics' main purpose is to measure and evaluate the productivity of programmers. For this type of metric, LOC, SIZE2 and NOM were chosen to represent it. It is to be noted that the metrics in the table above were analyzed at a class level.

If we take a look at the compiled results above, we observe that LOC, or lines of code, has an average of ≈124.70 and a maximum of 1897, which is a high value in comparison. A class this large is problematic, as it decreases readability, understandability and maintainability. A solution would be to take another look at the class, and to refactor it by dividing it into smaller classes for instance.

As for the SIZE2, this metric represents the number of attributes and methods within a class. In the case of the classes of AmazeFile, we obtained an average of ≈82.41 and a maximum of 1219. In other words, the classes possess, on average, a total of ≈82 methods and attributes each, which indicates that there might be a lack of cohesion throughout the application. The maximum value of 1219 is a clear indication that the source code should be revised.

Finally, the NOM metric counts the number of methods in a class. Similarly to both previous size metrics, we observe a big gap between the average value of ≈7.4, and the maximum of 75. Again, this suggests that non relevant methods might be present inside the classes of the application.

Table 2: Cohesion metrics

	Average	Maximum	Standard Deviation	Total
LCOM (MetricsReloaded)	1.938	22	2.243	620

As seen in class, the cohesion metric determines how closely related the elements of a module are to each other. In other words, it measures how strongly the methods of a class and its data are connected. A highly cohesive class is one that fulfills a single purpose. A high cohesion is desirable, as it makes the class more understandable, reliable, reusable and modifiable. There are many ways to evaluate this type of characteristic, but for this report, solely LCOM is evaluated. It represents the lack of cohesion between methods of class. The table above displays the results from analyzing the LCOM metric of the classes within AmazeFile.

We obtained an average of ≈ 1.938, and a maximum of 22. As explained in Henderson-Sellers' definition, the value of LCOM represents a perfect cohesion when it equals 0. On the other hand, the more its value approaches or goes past 1, the less cohesive a class is. In our case, since the average value surpasses 1, we may say that the classes of AmazeFile lack cohesion, as aforementioned whilst analyzing the size oriented metrics.

Table 3: Complexity metrics

	Average	Maximum	Standard Deviation	Total
CC (MetricsReloaded)	2.830	97	4.725	6202
WMC (MetricsTree)	23.032	348	44.869	5735
RFC (MetricsTree)	33.948	524	55.948	8453

The complexity refers to a set of characteristics of code. It gives a general idea of its overall quality. Generally speaking, the complexity focuses on the methods to determine the complexity of the code. A high complexity often causes issues with testing and debugging, it makes the code harder to read and understand and, overall, makes the code hard to maintain.

By looking at the overall results of the cyclomatic complexity's metrics, it can be noted that the maximums seem pretty high in comparison to the average. This metric gives us the number of independent paths in a given method. By taking a look at the CC metric, it can be seen that the method's average is relatively small since, according to the website *guru99*, a value of 10 or less, in general, indicates that the code of the method is well written, but, as noticed by the maximum of 97, this value is way higher than the average. The method associated with this value could most likely be separated into small methods.

As for the WMC metric, it calculates the complexity of each class by doing the weighted sum of the complexity of all methods in each class which gives a rough idea of the complexity of a class. According to the Chidamber & Kemerer definition, a value between 12 and 34 (23 in average in our case) would be considered average. It would mean that some classes in the project can be considered complex based on this metric so some refactoring could be required in that case. As for the maximum value, it's way higher than the average with a value of 348 which means that a class contains one or many complex methods and it could be a great candidate for refactoring.

Finally, for the RFC metric, it indicates the number of public methods of a class plus the number of methods that these public methods are calling. According to the Chidamber & Kemerer definition, a general threshold value for that metric is 44 so everything above 44 can be considered complex and harder to understand, hard to test and debug, and can be an important source of bugs. In the case of AmazeFile, the average value of RFC is very close to 34 which is a good indication. As for the maximum, the value of 524 is extremely above the threshold which raises red flags about this class and, as previously discussed, this class could be a good candidate for refactoring.

Table 4: Coupling metrics

	Average	Maximum	Standard Deviation	Total
CBO (MetricsTree)	10.145	111	13.329	2526
Ca (MetricsReloaded)	109.515	1095	202.582	7228

The coupling metric indicates the degree of interdependence between software components. In this case, the focus is on the classes. A low degree of coupling is desirable since it makes reusability easier, improves modularity, enforces encapsulation and makes maintenance easier. On the other hand, a high degree of coupling makes the code unreusable, violates encapsulation and makes the code very sensitive to changes.

For the CBO metric, it specifies the numbers of classes that are coupled to another class. In other words, when a class calls the methods or accesses attributes of another class. According to the Chidamber & Kemerer definition, a value higher than 14 means that the level of coupling is too high for a class. In our case, since the average is 10, the coupling isn't problematic but nevertheless still close to the threshold. As for the maximum of 111, it's way above the limit of 14 and this class may cause problems in the future because of the high level of dependencies.

For the Ca metric, it indicates the afferent coupling which is the number of external packages that use methods of a given package. For this metric, a threshold value isn't defined but, by examining the value, it can be noted that packages seem to have many interdependence between them. Also, the maximum of 1095 is way above the average of ~109 which means that changes to the package with this maximum value will impact many other packages since a high level afferent coupling means that many packages depend on it. This package is most likely a central part of the whole application.

b)

Size
The size is high and demonstrates that there are many large classes within AmazeFile. This causes many undesirable consequences, such as reducing the understandability, the analyzability and the modifiability of the classes.

Cohesion
The cohesion is not optimal, which means that the classes might perform more than one function, thus violating the Single Responsibility Principle. This indicates bad design of the code source and increases its complexibility, while decreasing the maintainability and the reusability of the classes.

Complexity
The complexity is, on average, relatively low which means that the code is easy to test and debug. This low value also makes the code easy to read and understand and easier to maintain.

Coupling
The coupling is medium, which means that there are a lot of interdependencies in the code. Some parts of the code can be hard to reuse, to maintain and it might violate encapsulation.

c)

Table 5: Values for Quality Attributes

Quality Attribute	Value
Reusability	$-0.25 * 10.145 + 0.25 * 0.997 + 0.5 * 6.259 + 0.5 * 409 = 205.3425$
Flexibility	$0.25 * 0.6916 - 0.25 * 10.145 + 0.5 * 6.02 + 0.5 * 7.872 = 4.5827$
Understandability	$-0.33 * 0.8276 + 0.33 * 0.6916 - 0.33 * 10.145 + 0.33 * 0.912 - 0.33 * 7.872 - 0.33 * 7.406 - 0.33 * 409 = -143.1035$
Functionality	$0.12 * 0.912 + 0.22 * 7.872 + 0.22 * 6.259 + 0.22 * 409 + 0.22 * 58 = 105.958$
Extendibility	$0.5 * 0.8276 - 0.5 * 10.145 + 0.5 * 0.1340 + 0.5 * 7.872 = -0.6557$
Effectiveness	$0.2 * 0.8276 + 0.2 * 0.6916 + 0.2 * 6.02 + 0.2 * 0.1340 + 0.2 * 7.872 = 3.109$

Table 6: Proxy metrics values for ISO 9126 Criteria Metrics

Design Property	Metric Value
Design size	409
Hierarchies	16+42 = 58
Abstraction	48/58 = 0.8276
Encapsulation	69.16% = 0.6916
Coupling	10.145
Cohesion	1-(1.938/22) = 0.912
Composition	6.02
Inheritance	13.40% = 0.1340
Polymorphism	7.406*1.063 = 7.872
Messaging	6.2587
Complexity	7.406

d)

Reusability

This quality is an advantage provided by the Object oriented architecture. It represents how well the entire code can be comprehended and reused by a developper. The high value (highly above 0) in our case suggests that the code contains many parts that can be reapplied elsewhere in other situations. In order to reuse the code, it must be, to a certain degree, high quality, flexible and easy to understand.

Flexibility

Flexibility is a quality that represents how easy it is to add changes in the code. In our situation, the value of this quality is relatively small since it is close to zero compared to the other positive values. This means that the code isn't very adaptable and configurable since adding or enhancing functionalities could necessitate some effort.

Understandability

This attribute represents how well a developper can identify logical concepts and applicability of the entire code. In the ISO/IEC 9126 standard, it can be seen as a sub-characteristic of Usability and Reusability. In our case, the value is highly negative which means that the code's logic and its overall applicability are very hard to understand. This could bring many challenges when trying to make changes in the code. Also, it causes the code to be difficult to test, debug and adding new programmers could be hard.

Functionality

This is a quality that depicts how well our software's code provides functions that carry out their responsibilities well. Therefore, since we have a pretty high functionality value, it means that the code contains many classes that implement the interfaces according to their role in the application.

Extendibility

This quality represents how well the system can be changed without causing undesirable side effects. In our case, the relatively small value (slightly negative) means that the system is not extensible at all. Small as well as big changes would require a lot of work.

Effectiveness

This quality represents how well the system performs according to its goal-behavior. In our case, the value is highly comparable to the Extendibility quality. For this quality, it means that although the software is compatible with its goal, it still could have been done better since the value is fairly close to 0.

e)

Reusability in ImmutableEntry.Java

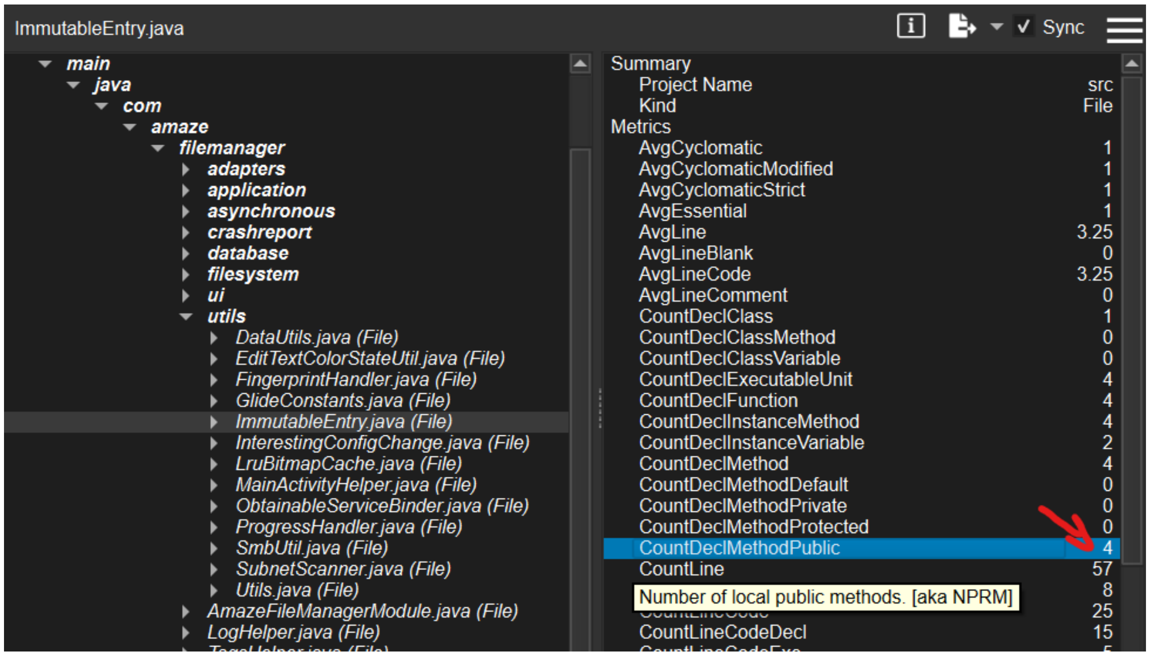
As explained in the previous question, we have a pretty high value for this quality, and therefore have many components that can be easily reused. To calculate the value for the Reusability quality, we took in consideration the Coupling, Cohesion, Messaging and Design Size properties. We notice that Coupling and Cohesion should be inversely proportional, as they respectively weigh -25% and 25%. On the other hand, the Messaging and Design Size both weigh 50% in the calculations. According to the metrics related to these properties, we identified a class that represents a highly reusable class; the ImmutableEntry.java.

For the Coupling property, we based its analysis solely on the value of the CBO (Coupling Between Objects) metric. In this case, the CBO has a value of 0 as we can see on Figure 2 below. Knowing what the entire project's average (10.145) and maximum (111) CBO are, this class has an extremely low coupling between objects. In other words, ImmutableEntry.java does not access other classes and is not being accessed by others. This low interaction with other objects is beneficial for the code's simplicity and comprehensibility, which makes it more easily reusable.

The Cohesion property is based on the LCOM (Lack of Cohesion Of Methods) metric, and specifically by the calculation: 1-(LCOM/Max(LCOM)). As we can see on Figure 2, the LCOM for ImmutableEntry is 3, which is higher than average considering that the entire project's average is 1.938 and the maximum is 22 (see Table 2 in Question 1). This means that ImmutableEntry does not have many unconnected pairs of methods in its class, or in other words does not have many uncorrelated methods. Moreover, this can be proven when we calculate the cohesion property of this class using the previously stated value. Indeed, we obtain a fairly high cohesion of 0.86 (86.36%). This is a beneficial result for the Reusability property, since a high cohesion means a better learnability and operability that helps reuse the code.

Design Size is represented by the Cp metric, which is the number of Product classes. In this case, since we are analyzing one specific class, it is not logical to consider the Cp value.

The Messaging property can be represented by the NOM (Number of Methods) metric, but only taking in consideration the public methods. The lesser it is, the more a class is simple and has a specific responsibility. In our case, as shown on Figure 1 right below, the number of local public methods in the ImmutableEntry.java class is 4. Therefore, this class has a pretty low NOM if we compare it to the software's value represented on Table 6. This means that the class represents more security, interoperability and attractiveness because of the encapsulation, which are all attributes that contribute to a reusable code.

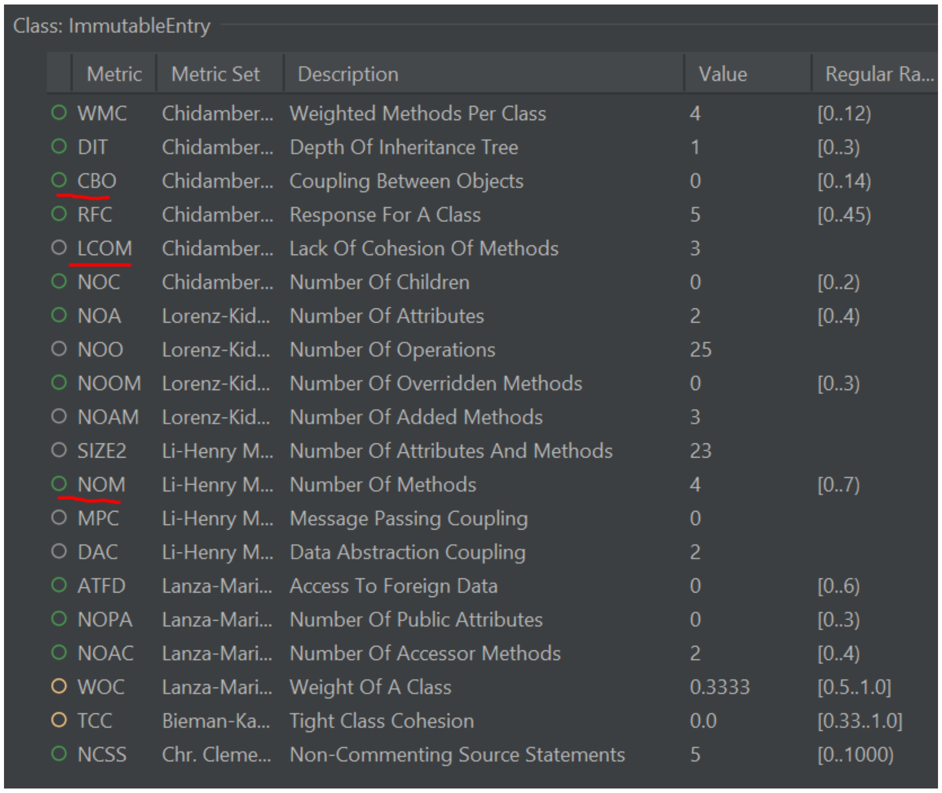


```
main
  java
    com
      amaze
        filemanager
          adapters
          application
          asynchronous
          crashreport
          database
          filesystem
          ui
          utils
            DataUtils.java (File)
            EditTextColorStateUtil.java (File)
            FingerprintHandler.java (File)
            GlideConstants.java (File)
            ImmutableEntry.java (File)
            InterestingConfigChange.java (File)
            LruBitmapCache.java (File)
            MainActivityHelper.java (File)
            ObtainableServiceBinder.java (File)
            ProgressHandler.java (File)
            SmbUtil.java (File)
            SubnetScanner.java (File)
            Utils.java (File)
          AmazeFileManagerModule.java (File)
          LogHelper.java (File)
```

Summary	
Project Name	src
Kind	File
Metrics	
AvgCyclomatic	1
AvgCyclomaticModified	1
AvgCyclomaticStrict	1
AvgEssential	1
AvgLine	3.25
AvgLineBlank	0
AvgLineCode	3.25
AvgLineComment	0
CountDeclClass	1
CountDeclClassMethod	0
CountDeclClassVariable	0
CountDeclExecutableUnit	4
CountDeclFunction	4
CountDeclInstanceMethod	4
CountDeclInstanceVariable	2
CountDeclMethod	4
CountDeclMethodDefault	0
CountDeclMethodPrivate	0
CountDeclMethodProtected	0
CountDeclMethodPublic	4
CountLine	57
	8
	25
CountLineCodeDecl	15

Number of local public methods. [aka NPRM]

Figure 1. NOM metric (public methods only) for *ImmuableEntry.java*


Class: ImmutableEntry

	Metric	Metric Set	Description	Value	Regular Ra...
○	WMC	Chidamber...	Weighted Methods Per Class	4	[0..12)
○	DIT	Chidamber...	Depth Of Inheritance Tree	1	[0..3)
○	CBO	Chidamber...	Coupling Between Objects	0	[0..14)
○	RFC	Chidamber...	Response For A Class	5	[0..45)
○	LCOM	Chidamber...	Lack Of Cohesion Of Methods	3	
○	NOC	Chidamber...	Number Of Children	0	[0..2)
○	NOA	Lorenz-Kid...	Number Of Attributes	2	[0..4)
○	NOO	Lorenz-Kid...	Number Of Operations	25	
○	NOOM	Lorenz-Kid...	Number Of Overridden Methods	0	[0..3)
○	NOAM	Lorenz-Kid...	Number Of Added Methods	3	
○	SIZE2	Li-Henry M...	Number Of Attributes And Methods	23	
○	NOM	Li-Henry M...	Number Of Methods	4	[0..7)
○	MPC	Li-Henry M...	Message Passing Coupling	0	
○	DAC	Li-Henry M...	Data Abstraction Coupling	2	
○	ATFD	Lanza-Mari...	Access To Foreign Data	0	[0..6)
○	NOPA	Lanza-Mari...	Number Of Public Attributes	0	[0..3)
○	NOAC	Lanza-Mari...	Number Of Accessor Methods	2	[0..4)
○	WOC	Lanza-Mari...	Weight Of A Class	0.3333	[0.5..1.0]
○	TCC	Bieman-Ka...	Tight Class Cohesion	0.0	[0.33..1.0]
○	NCSS	Chr. Cleme...	Non-Commenting Source Statements	5	[0..1000)

Figure 2. *ImmutableEntry.java*'s CBO, LCOM and NOM metrics

Flexibility

The design properties that define this quality criteria are Encapsulation, Coupling, Composition and Polymorphism in which coupling is negatively correlated while the others are positively correlated. The most impactful metrics for this criteria are the Composition and Polymorphism with a weight of 33,33% each (0.5/1.5) and the least impactful metrics are the Encapsulation and the Coupling with a weight of 16,66% each.

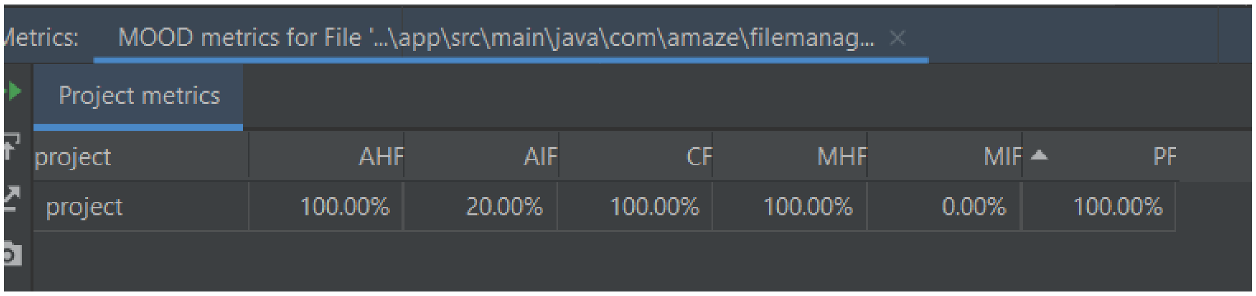
In our case the value of the flexibility was slightly positive. Let's examine a class with a slightly positive flexibility. As an example, we chose the class LoadFilesListTask.java

Encapsulation is defined by the metric AHF (Attribute hiding factor). This metric defines the percentage of the attributes in the class that are encapsulated. As we can see from the MetricsReloaded MOOD analysis in Figure 3, the AHF value is 100% which is higher than the average of 69.16% of the whole project. This means that this class is very well encapsulated which in turn increases its flexibility.

Coupling is defined by the metric CBO. This metric defines the level of coupling in the class. From the MetricsTree analysis in Figure 4, the CBO value given is 25 which is higher than the average of 10,145. This high coupling makes the class less flexible.

Composition is defined by the metric Number of dependencies. This metric defines the number of aggregation from other classes that this class has. This value is represented by the column Dcy of the MetricsReloaded analysis in Figure 5 which gives a result of 25 which is above the average of 6,02. This high level of aggregation gives a better flexibility to the class.

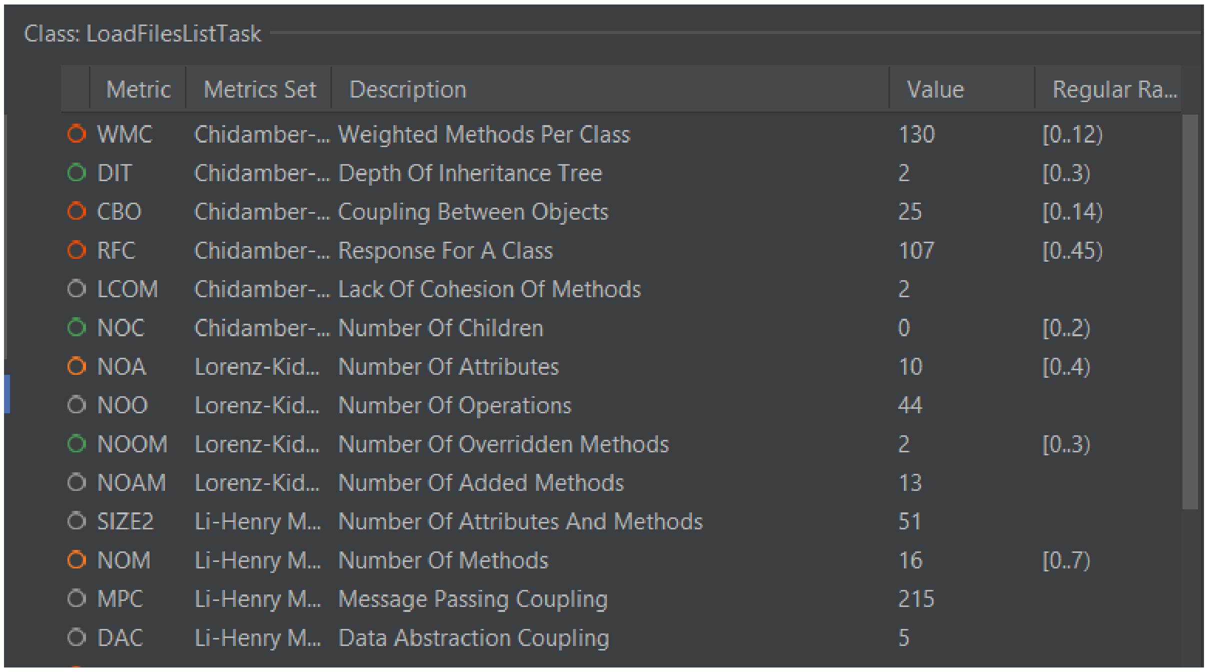
Polymorphism is defined by the metric NOM (number of methods)*PF (polymorphism factor). This metric defines the number of polymorphic methods by multiplying the number of methods (NOM) with the polymorphism factor. The value of the metric NOM given by MetricsTree in Figure 4 is 16 which is higher than the average of 7,406 and the value of the PF given by MetricsReloaded in Figure 3 is 100% which is slightly lower than the average of 106,3% of the project. Given the two values, the number of polymorphic methods is quite high which gives a better flexibility to the class.


Metrics: MOOD metrics for File '...\app\src\main\java\com\amaze\filemanag...

Project metrics

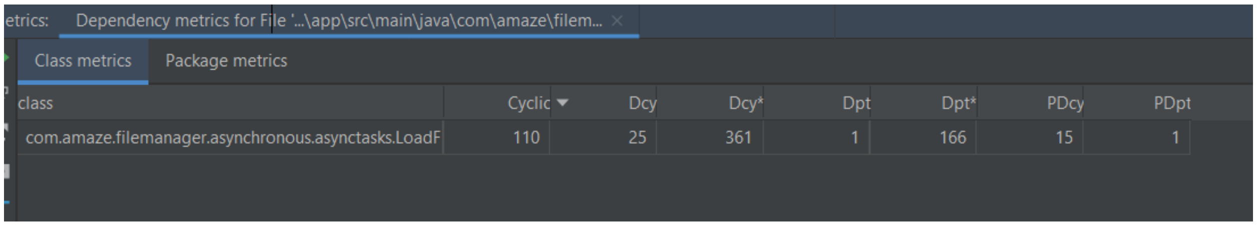
project	AHF	AIF	CF	MHF	MIF	PF
project	100.00%	20.00%	100.00%	100.00%	0.00%	100.00%

Figure 3. *LoadFilesListTask.java* MOOD metrics


Class: LoadFilesListTask

Metric	Metrics Set	Description	Value	Regular Ra...
WMC	Chidamber-...	Weighted Methods Per Class	130	[0..12)
DIT	Chidamber-...	Depth Of Inheritance Tree	2	[0..3)
CBO	Chidamber-...	Coupling Between Objects	25	[0..14)
RFC	Chidamber-...	Response For A Class	107	[0..45)
LCOM	Chidamber-...	Lack Of Cohesion Of Methods	2	
NOC	Chidamber-...	Number Of Children	0	[0..2)
NOA	Lorenz-Kid...	Number Of Attributes	10	[0..4)
NOO	Lorenz-Kid...	Number Of Operations	44	
NOOM	Lorenz-Kid...	Number Of Overridden Methods	2	[0..3)
NOAM	Lorenz-Kid...	Number Of Added Methods	13	
SIZE2	Li-Henry M...	Number Of Attributes And Methods	51	
NOM	Li-Henry M...	Number Of Methods	16	[0..7)
MPC	Li-Henry M...	Message Passing Coupling	215	
DAC	Li-Henry M...	Data Abstraction Coupling	5	

Figure 4. *LoadFilesListTask.java*'s CBO and NOM metrics


etrics: Dependency metrics for File '...\app\src\main\java\com\amaze\filem...

Class metrics | Package metrics

class	Cyclic	Dcy	Dcy*	Dpt	Dpt*	PDcy	PDpt
com.amaze.filemanager.asynchronous.asynctasks.LoadF	110	25	361	1	166	15	1

Figure 5.*LoadFilesListTask.java*'s dependency metrics

Functionality

This quality is calculated using the Cohesion, Polymorphism, Messaging, Design Size and Hierarchies properties. Cohesion weighs the least, as it is 12% while the others all weigh equally 22%. In our case, we explained previously that we had a fairly high value for the Functionality. Let's examine a class with a high functionality. As an example we chose the class FileUtils.java.

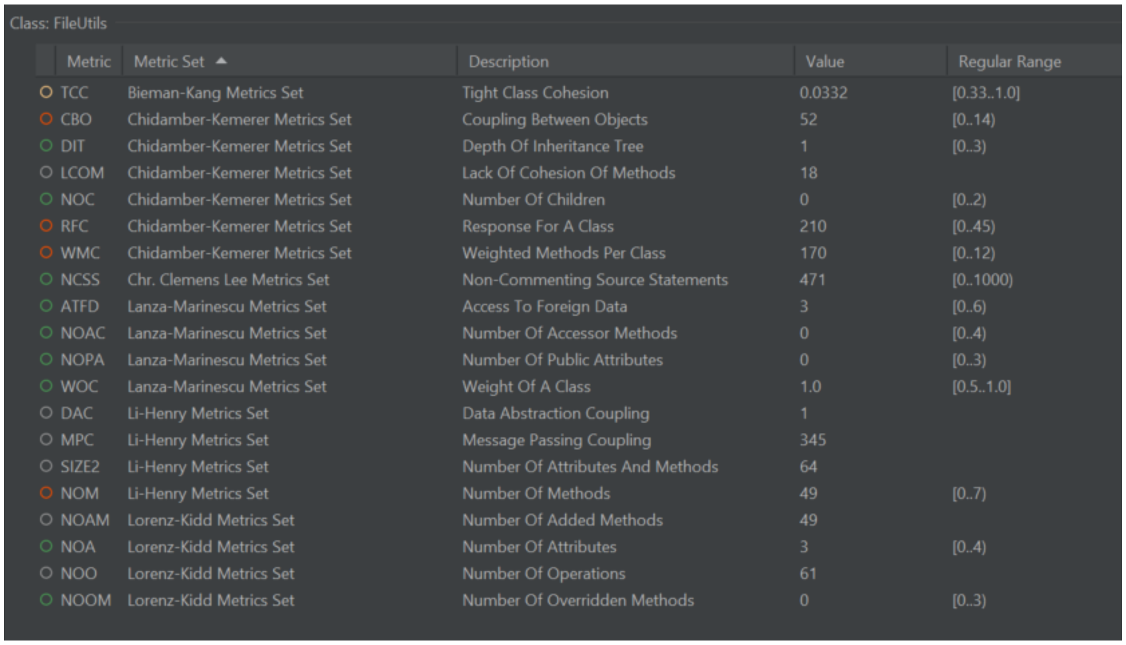
Cohesion is defined by the metric LCOM in the equation 1-(LCOM/Max(LCOM)) that represents the cohesion with the methods of the class. As we can see from the result given by MetricsTree in Figure 6, the LCOM value is quite close to the maximum LCOM (22) of the project which means that this class lacks cohesion in its methods. Using the formula mentioned previously, we get a cohesion of 0,1818 which is a relatively small value. This results in a very slight positive effect on the functionality of the class.

Polymorphism is defined by the metric NOM (number of methods)*PF (polymorphism factor). This metric defines the number of polymorphic methods by multiplying the number of methods (NOM) with the polymorphism factor. The NOM metric, which was calculated with MetricsTree in Figure 6, gives us a high value of 49 which is higher than the average of 7,406. As for the PF metric, it was calculated with MetricsReloaded (see Figure 8) and the result of 100% means that the class is highly polymorphic. Given these two values, the polymorphism of this class is 49 which is quite high. This value of polymorphism makes the class very functional.

Messaging is defined by the number of public methods of the class. This metric was calculated with Understand (see Figure 7) and the result of 46 which gave a value way higher than 6,2587. This high number of public methods makes the code more functional.

Design Size is represented by the Cp metric, which is the number of Product classes. In this case, since we are analyzing one specific class, it is not logical to consider the Cp value.

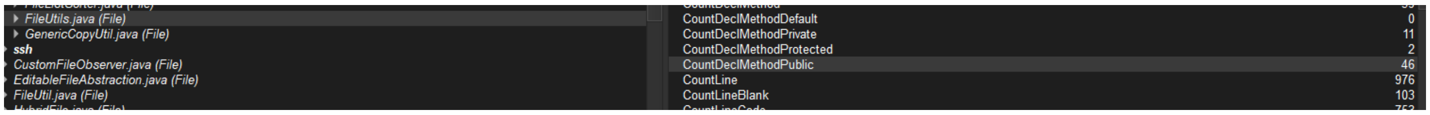
Hierarchies is defined by the number of abstract classes added with the number of interfaces. In our case, since we are analyzing one specific class, it is not logical to consider the Cp value.



Class: FileUtils

Metric	Metric Set	Description	Value	Regular Range
TCC	Bieman-Kang Metrics Set	Tight Class Cohesion	0.0332	[0.33..1.0]
CBO	Chidamber-Kemerer Metrics Set	Coupling Between Objects	52	[0..14)
DIT	Chidamber-Kemerer Metrics Set	Depth Of Inheritance Tree	1	[0..3)
LCOM	Chidamber-Kemerer Metrics Set	Lack Of Cohesion Of Methods	18	
NOC	Chidamber-Kemerer Metrics Set	Number Of Children	0	[0..2)
RFC	Chidamber-Kemerer Metrics Set	Response For A Class	210	[0..45)
WMC	Chidamber-Kemerer Metrics Set	Weighted Methods Per Class	170	[0..12)
NCSS	Chr. Clemens Lee Metrics Set	Non-Commenting Source Statements	471	[0..1000)
ATFD	Lanza-Marinescu Metrics Set	Access To Foreign Data	3	[0..6)
NOAC	Lanza-Marinescu Metrics Set	Number Of Accessor Methods	0	[0..4)
NOPA	Lanza-Marinescu Metrics Set	Number Of Public Attributes	0	[0..3)
WOC	Lanza-Marinescu Metrics Set	Weight Of A Class	1.0	[0.5..1.0]
DAC	Li-Henry Metrics Set	Data Abstraction Coupling	1	
MPC	Li-Henry Metrics Set	Message Passing Coupling	345	
SIZE2	Li-Henry Metrics Set	Number Of Attributes And Methods	64	
NOM	Li-Henry Metrics Set	Number Of Methods	49	[0..7)
NOAM	Lorenz-Kidd Metrics Set	Number Of Added Methods	49	
NOA	Lorenz-Kidd Metrics Set	Number Of Attributes	3	[0..4)
NOO	Lorenz-Kidd Metrics Set	Number Of Operations	61	
NOOM	Lorenz-Kidd Metrics Set	Number Of Overridden Methods	0	[0..3)

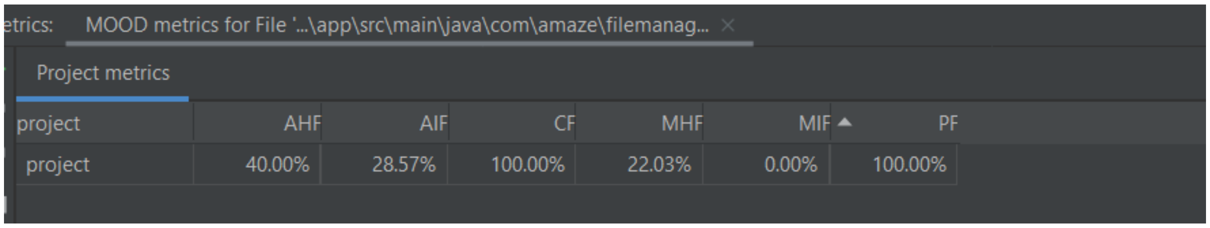
Figure 6. *FileUtils.java*'s LCOM and NOM metrics



FileUtils.java (File)
GenericCopyUtil.java (File)
ssh
CustomFileObserver.java (File)
EditableFileAbstraction.java (File)
FileUtil.java (File)

CountDeclMethodDefault	0
CountDeclMethodPrivate	11
CountDeclMethodProtected	2
CountDeclMethodPublic	46
CountLine	976
CountLineBlank	103

Figure 7. NOM metric (public methods only) for *FileUtils.java*



etrics: MOOD metrics for File '...\app\src\main\java\com\amaze\filemanag...

Project metrics

project	AHF	AIF	CF	MHF	MIF	PF
project	40.00%	28.57%	100.00%	22.03%	0.00%	100.00%

Figure 8. *FileUtils.java*'s MOOD metrics

Question 2: Find five different anomalies in AmazeFile

a)

In order to define the metric thresholds, we used the general and statistical definition from the book "Object-Oriented Metrics in Practice". In other words, the equations below were used to determine them:

$$Low = Avg - St.Dev$$
$$High = Avg + St.Dev$$
$$Very\ High = (Avg + St.Dev) * 1.5$$

$$0 = None,\ 1 = ONE,\ 2 - 5 = FEW/SEVERAL,\ 7 = Short\ Memory\ Cap,\ > 7 = MANY$$

Table 7: Metric thresholds

Metric	AVG	STD	LOW	HIGH	VERY HIGH
NOM	7.406	9.875	-2.469 (≈0)	17.281	25.9215
WMC	23.032	44.869	-21.837 (≈0)	67.901	101.8515
CC	2.83	4.725	-1.895 (≈0)	7.555	11.3325
LOC	15.93	24.176	-8.246 (≈0)	40.106	60.159

b) c)

For this question, we have decided to use *MetricsTree, MetricsReloaded* and *Understand* to determine the metrics of individual classes.

1. *length* Method (from *HybridFile.java*)

While observing the *length* method from the HybridFile class, we identified the *Brain Method* code smell. We were able to determine this anti-pattern with the help of the figure below from "Object-Oriented Metrics in Practice":

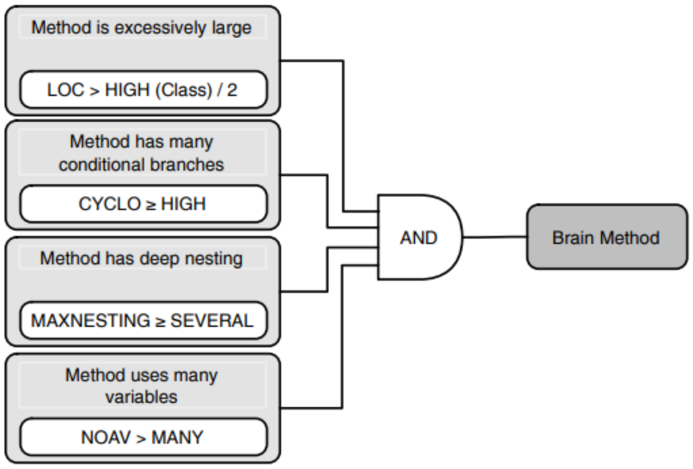
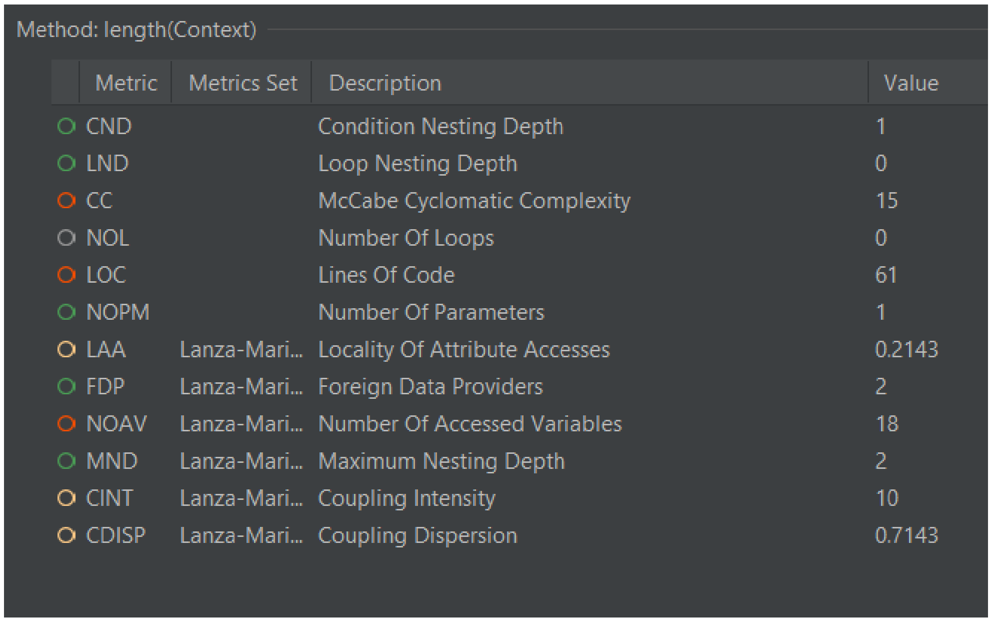


Figure 9. Metrics that influence the Brain Method antipattern

Whilst calculating its metric, we obtained a LOC value of 61 (MetricsTree), a CC of 15 (MetricsTree), a MND value of 2 and a NOAV of 18. If we compare these metrics to the figure above and the calculated thresholds in a), we can confirm that LOC is superior to HIGH/2 (LOC > 40.106/2), CC is above HIGH, MNDis equal to SEVERAL and NOAV surpasses MANY.



Method: length(Context)

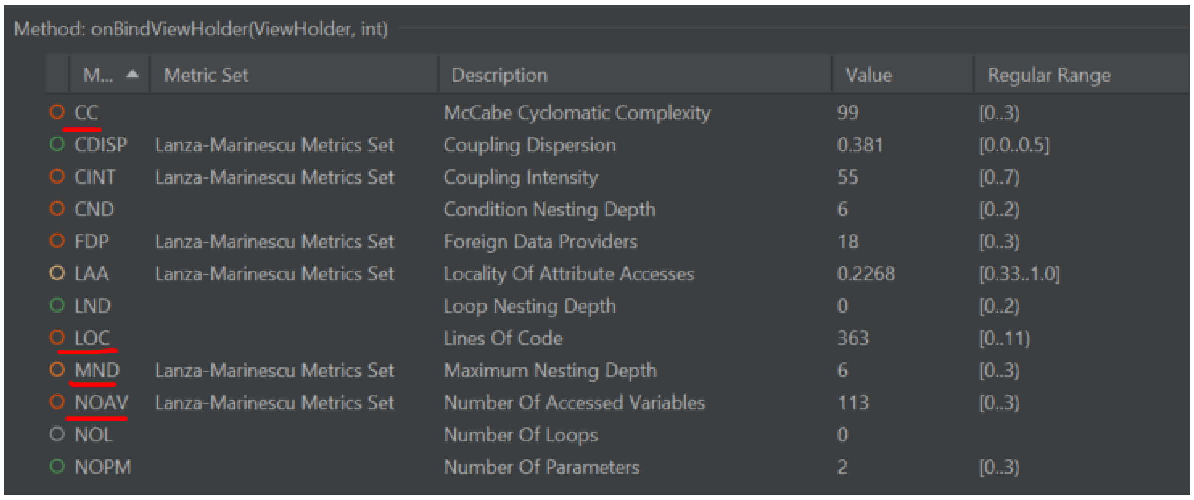
Metric	Metrics Set	Description	Value
CND		Condition Nesting Depth	1
LND		Loop Nesting Depth	0
CC		McCabe Cyclomatic Complexity	15
NOL		Number Of Loops	0
LOC		Lines Of Code	61
NOPM		Number Of Parameters	1
LAA	Lanza-Mari...	Locality Of Attribute Accesses	0.2143
FDP	Lanza-Mari...	Foreign Data Providers	2
NOAV	Lanza-Mari...	Number Of Accessed Variables	18
MND	Lanza-Mari...	Maximum Nesting Depth	2
CINT	Lanza-Mari...	Coupling Intensity	10
CDISP	Lanza-Mari...	Coupling Dispersion	0.7143

Figure 10. *length* method's (in *HybridFile.java*) LOC and CC metrics

This high LOC value indicates that the method contains a large number of lines, which causes a lack of understandability and modifiability. As for CC, it suggests that the method in question has many conditional branches, and in consequence, has a fairly high complexity. This method's MND value tells us that some nesting is present in the method, and finally, the high NOAV value indicates that this *HybridFile* method calls its parameters, local variables and global variables multiple times. In consequence, this increases the risk of introducing bugs to the code in question. In the end, these combined metrics allow us to recognize the *Brain Method* code smell. Essentially, this method contains too many lines of code and too many responsibilities.

2. <u>*onBindViewHolder* Method (from *RecyclerAdapter.java*)</u>

From a different class (RecyclerAdapter.java), we found another case of Brain Method. As we can see on the Figure 11 below, the method onBindViewHolder(ViewHolder, int) has some pretty high values for the metrics CC, LOC, MND and NOAV. These metrics, as explained in the previous code smell (1. Brain Method from HybridFile Class), are the ones contributing to the Brain Method anomaly.


Method: onBindViewHolder(ViewHolder, int)

M...	Metric Set	Description	Value	Regular Range
CC		McCabe Cyclomatic Complexity	99	[0..3)
CDISP	Lanza-Marinescu Metrics Set	Coupling Dispersion	0.381	[0.0..0.5]
CINT	Lanza-Marinescu Metrics Set	Coupling Intensity	55	[0..7)
CND		Condition Nesting Depth	6	[0..2)
FDP	Lanza-Marinescu Metrics Set	Foreign Data Providers	18	[0..3)
LAA	Lanza-Marinescu Metrics Set	Locality Of Attribute Accesses	0.2268	[0.33..1.0]
LND		Loop Nesting Depth	0	[0..2)
LOC		Lines Of Code	363	[0..11)
MND	Lanza-Marinescu Metrics Set	Maximum Nesting Depth	6	[0..3)
NOAV	Lanza-Marinescu Metrics Set	Number Of Accessed Variables	113	[0..3)
NOL		Number Of Loops	0	
NOPM		Number Of Parameters	2	[0..3)

Figure 11. *onBindViewHolder* method (in *HybridFile.java*) MND, NOAV, LOC and CC metrics

The Cyclomatic Complexity (CC) has a value of 99, which is a lot higher than the HIGH value (7.555) calculated in the threshold table. This extremely high value indicates that *onBindViewHolder* is a highly complex method, and more specifically a method that overuses conditional branches. This inevitably affects the method's length which is negative for its maintainability and readability.

The value of LOC (Lines of Code) is also extremely large. As stated on Figure 9, we have to compare it to: HIGH(Class)/2. From Table 7, we know that HIGH equals 40.106, which means that the point of comparison is 20.05. In the case of *onBindViewHolder*, the LOC is significantly higher, with a value of 363. This metric clearly shows that the method is way too long in terms of size, which makes it hard to understand, modify, test and replace.

The Maximum Nesting Depth for *onBindViewHolder* is equal to 6. Based on the Figure 9 again, we know we have to compare this value to SEVERAL (2-5). Therefore, since it's higher than 2-5, we know that this method contains a high number of nested blocks due to the overuse of control structures (conditional branches, loops, etc). Similarly to the CC metric, this large value makes a method heavier in size, which has a negative impact on the performance and clarity of the code.

Finally, the NOAV represents the number of variables used by the method. When we compare the value 113 in *onBindViewHolder* to MANY (>7), we realize that it is extremely big. Combined with the three other metrics that are also fairly high, we obtain a very complex method that fits in the Brain Method code smell category.

3. *Operations.java*

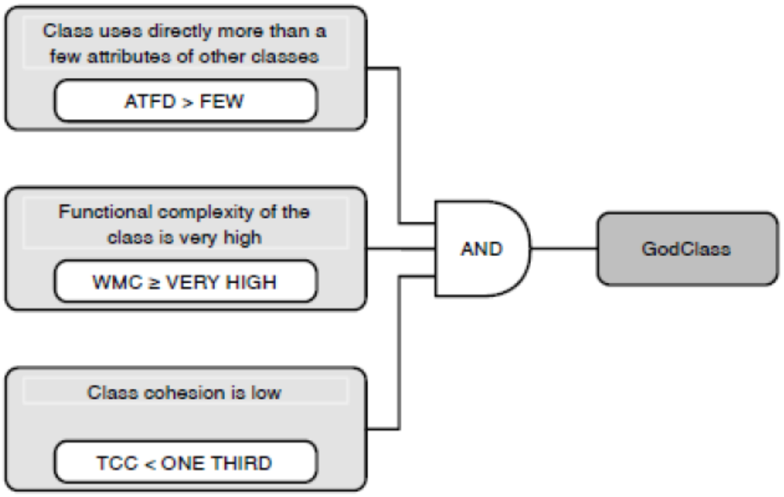
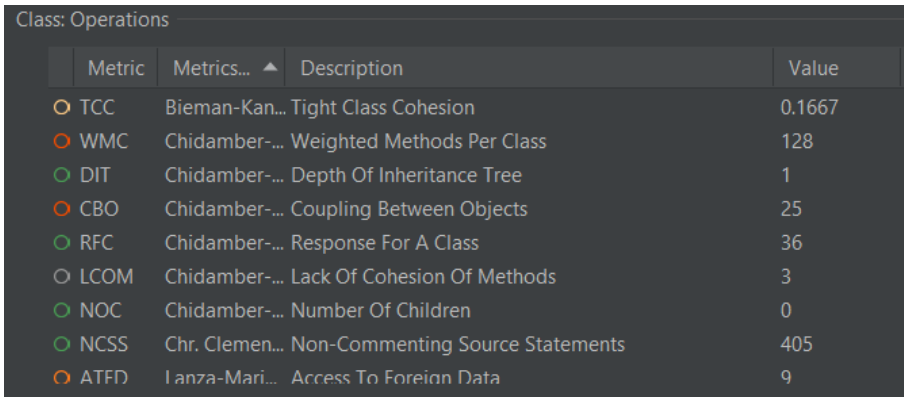

Class uses directly more than a few attributes of other classes
ATFD > FEW
Functional complexity of the class is very high
WMC ≥ VERY HIGH
Class cohesion is low
TCC < ONE THIRD
AND
GodClass

Figure 12. Metrics that influence the GodClass antipattern

While observing the Operations class, we identified the *God Class* code smell. As we can see on the Figure 12 above, this type of anomaly is based upon three metrics: ATFD, WMC and TCC. We can see on Figure 13 below that the value of the metric ATFD is 9, the value of the metric WMC is 128 and the value of TCC is 0,1667. If we compare these values to the thresholds we can see that the class has an ATFD higher than a FEW (2 to 5), the WMC higher than VERY HIGH, which is 101,8515 (see Table 7), and TCC is smaller than ⅓ .



Class: Operations

Metric	Metrics...	Description	Value
TCC	Bieman-Kan...	Tight Class Cohesion	0.1667
WMC	Chidamber-...	Weighted Methods Per Class	128
DIT	Chidamber-...	Depth Of Inheritance Tree	1
CBO	Chidamber-...	Coupling Between Objects	25
RFC	Chidamber-...	Response For A Class	36
LCOM	Chidamber-...	Lack Of Cohesion Of Methods	3
NOC	Chidamber-...	Number Of Children	0
NCSS	Chr. Clemen...	Non-Commenting Source Statements	405
ATFD	Lanza-Mari...	Access To Foreign Data	9

Figure 13. *Operations.java*'s ATFD, WMC and TCC metrics

The high value of the ATFD metric tells us that the method directly uses many attributes of other classes which makes the class very dependent on other classes. As for the high value of the WMC metric, it tells us that the methods of the class are very complex which in return makes the overall class complex. As for the small value of the metric TCC, it tells us that the public methods of the class aren't not very connected which makes the cohesion of the class low. Overall, the combination of both metrics hints us toward the God Class anomaly since the class is complex and has low cohesion.

4. *publishResults* Method (from *AbstractProgressiveService.java*)

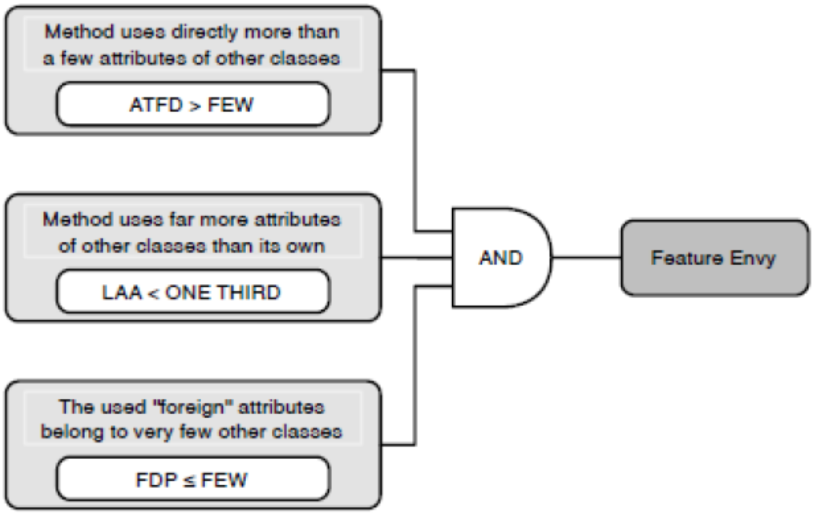
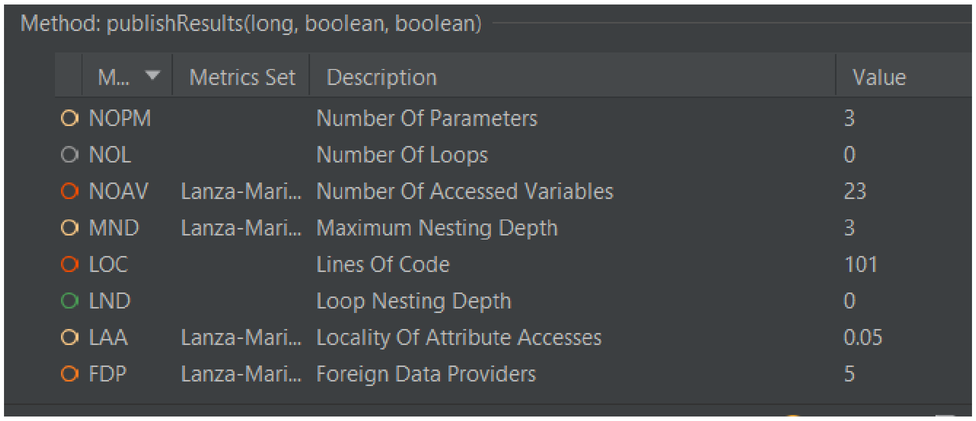

Method uses directly more than a few attributes of other classes
ATFD > FEW
Method uses far more attributes of other classes than its own
LAA < ONE THIRD
The used "foreign" attributes belong to very few other classes
FDP ≤ FEW
AND
Feature Envy

Figure 14. Metrics that influence the Feature Envy antipattern

While observing the *publishResults* method from the HybridFile class, we identified the *Feature Envy* code smell. As we can see on the Figure 14 above, this antipattern is influenced by three metrics, which are ATFD, LAA and FDP. From the Figure 15 below, we can see that the metric LAA is 0.05 and the metric FDP is 5. As for the ATFD metric, it can be estimated to the value 21.85 ((1-0.05)*23) based on the metric NOAV and LAA that are also figured on Figure 15. By examining all those values together, we can conclude that this method has a Feature Envy anomaly since ATFD is way above the value of FEW (2 to 5), LAA is considerably under the value of ⅓ and FDP is between 2 and 5 (equals to FEW).


Method: publishResults(long, boolean, boolean)

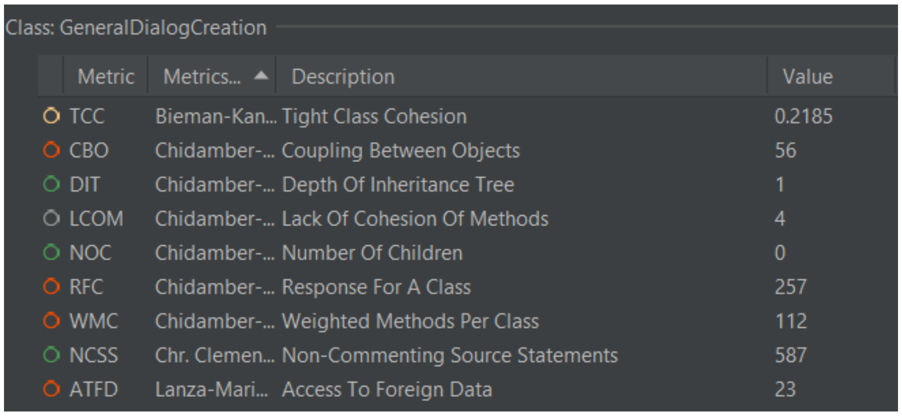
M...	Metrics Set	Description	Value
NOPM		Number Of Parameters	3
NOL		Number Of Loops	0
NOAV	Lanza-Mari...	Number Of Accessed Variables	23
MND	Lanza-Mari...	Maximum Nesting Depth	3
LOC		Lines Of Code	101
LND		Loop Nesting Depth	0
LAA	Lanza-Mari...	Locality Of Attribute Accesses	0.05
FDP	Lanza-Mari...	Foreign Data Providers	5

Figure 15. *publishResults* method's (in *AbstractProgressiveService.java*) LAA, FDP and NOAV metrics

The high value of the ATFD metric tells us that the method directly uses many attributes of other classes which makes the class very dependent on other classes. The low value of the LAA metric tells us that the method uses a lot more attributes of other classes than attributes of its own class which once again makes the class very dependent on other classes. The low value of FDP tells us that the attributes come only from a few other classes which makes the class depend on only a few classes. The combination of these metrics hints us toward the Feature Envy anomaly since the class uses mostly other attributes that come from a few other classes.

5. *GeneralDialogCreation.java*

The final code smell was identified in the class GeneralDialogCreation and it consists of a *God Class* antipattern. As explained for the first identified code smell (Operations.java), this anomaly depends on the metrics ATFD, WMC and TCC. From the Figure 16 below, we know that the metric ATFD has a value of 9, the metric WMC has a value of 128, and the metric TCC has a value of 0.1667. When we compare these values to the threshold, it is possible to see that GeneralDialogCreation has an ATFD higher than FEW (2 to 5), a WMC higher than the value of VERY HIGH (101.8515 presented on Table 7) and a TCC smaller than ⅓ .


Class: GeneralDialogCreation

Metric	Metrics...	Description	Value
TCC	Bieman-Kan...	Tight Class Cohesion	0.2185
CBO	Chidamber-...	Coupling Between Objects	56
DIT	Chidamber-...	Depth Of Inheritance Tree	1
LCOM	Chidamber-...	Lack Of Cohesion Of Methods	4
NOC	Chidamber-...	Number Of Children	0
RFC	Chidamber-...	Response For A Class	257
WMC	Chidamber-...	Weighted Methods Per Class	112
NCSS	Chr. Clemen...	Non-Commenting Source Statements	587
ATFD	Lanza-Mari...	Access To Foreign Data	23

Figure 16. *GeneralDialogCreation.java*'s ATFD, WMC and TCC metrics

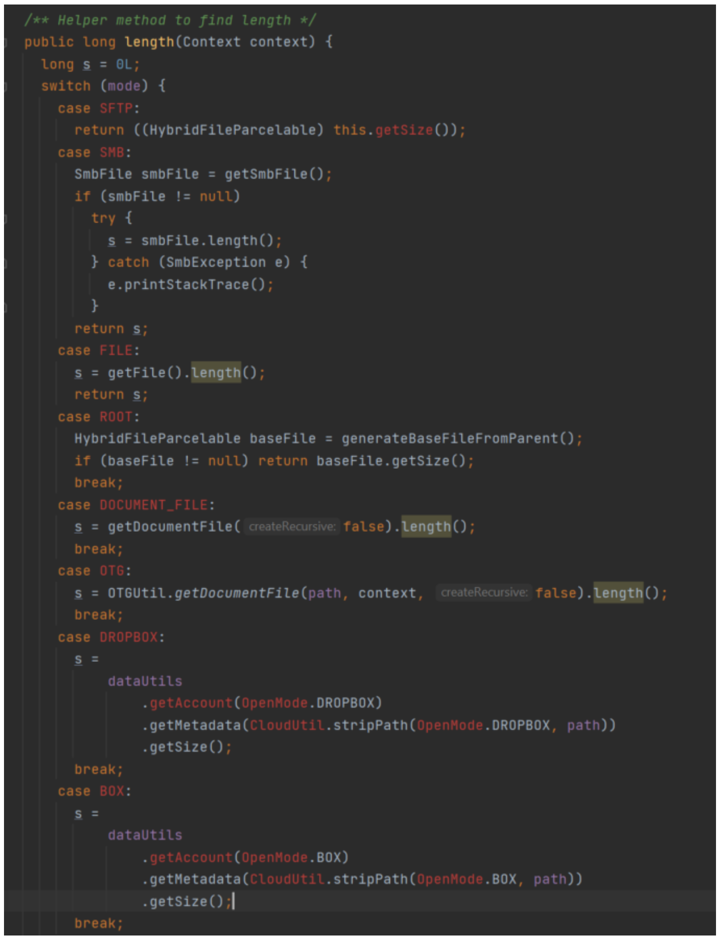
The high value of the ATFD metric tells us that the method directly uses many attributes of other classes which makes the class very dependent on other classes. As for the high value of the WMC metric, it tells us that the methods of the class are very complex which in return makes the overall class complex. As for the small value of the metric TCC, it tells us that the public methods of the class aren't not very connected which makes the cohesion of the class low. Overall, the combination of both metrics hints us toward the God Class anomaly since the class is complex and has low cohesion.

Question 3: Correct the anomalies by applying refactorings

a) b)

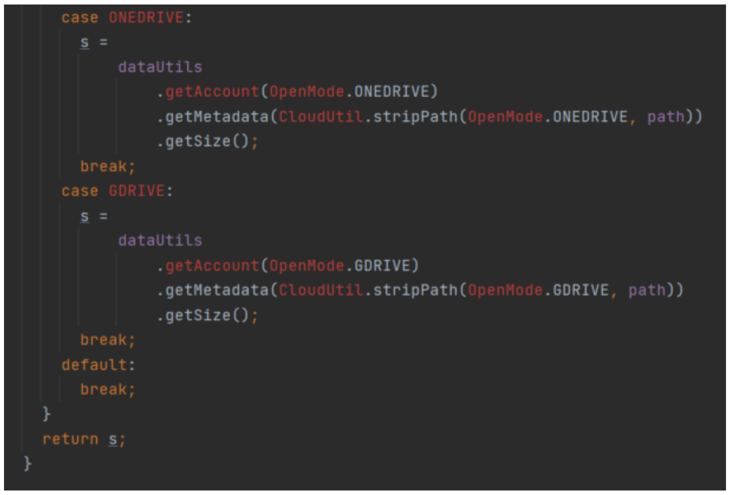
1. <u>Brain Method in *HybridFile.java*</u>

As we have seen in class, there are many ways of correcting the *Brain Method* code smell, such as extracting the method or using polymorphism. In our case, since the *length* method consisted of a switch statement, we decided to refactor using the Strategy pattern, as well as the existing *OpenMode* enum. The first step was to create an interface (*FileMode*) that is used to apply the algorithm. As shown in Figure 21, this interface solely contains the *length* method's signature, a method that is to be implemented by each of the strategies. If we were to interpret our code in terms of the Strategy pattern's structure, this newly created interface plays the role of the *Strategy* interface as it declares the method used to execute the strategy. The *Context*, on the other hand, is represented by the *HybridFile* class as it maintains a reference to the *Strategy* interface. Finally, the *OpenMode* enum contains the *Concrete Strategies* that implement the interface. In other words, this allowed us to move the initial code into *OpenMode*, thus simplifying the method by eliminating the switch statement.



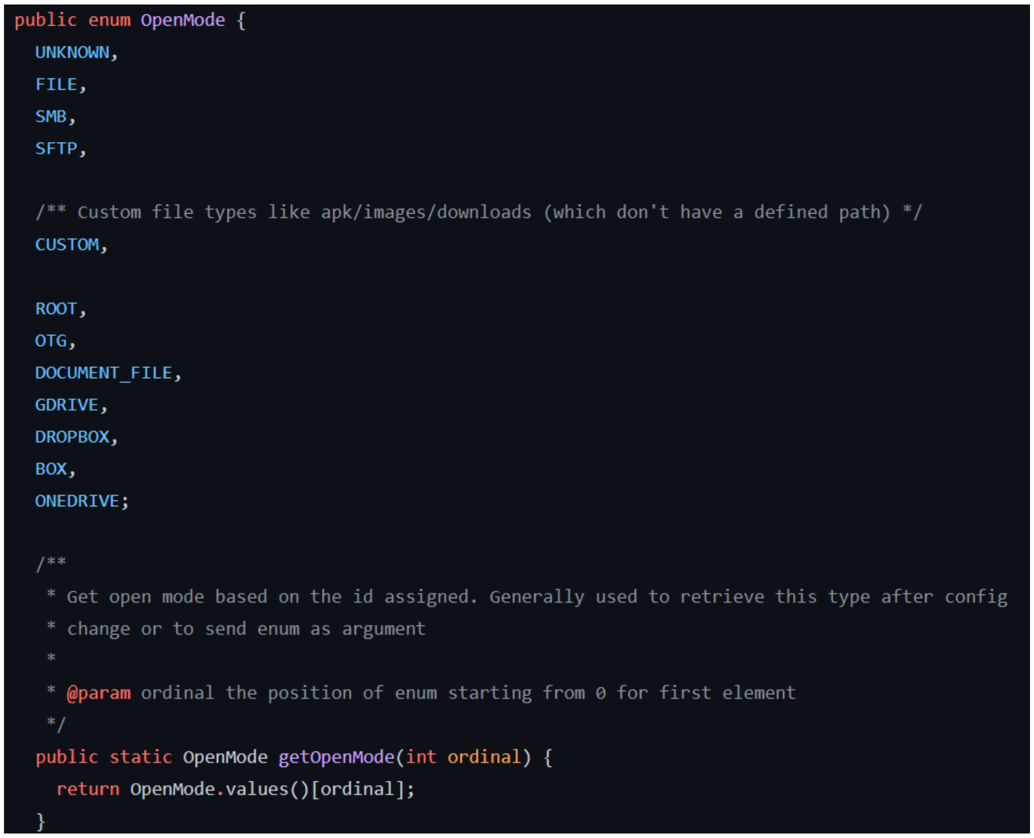
```java
/** Helper method to find length */
public long length(Context context) {
  long s = 0L;
  switch (mode) {
    case SFTP:
      return ((HybridFileParcelable) this.getSize());
    case SMB:
      SmbFile smbFile = getSmbFile();
      if (smbFile != null)
        try {
          s = smbFile.length();
        } catch (SmbException e) {
          e.printStackTrace();
        }
      return s;
    case FILE:
      s = getFile().length();
      return s;
    case ROOT:
      HybridFileParcelable baseFile = generateBaseFileFromParent();
      if (baseFile != null) return baseFile.getSize();
      break;
    case DOCUMENT_FILE:
      s = getDocumentFile( createRecursive: false).length();
      break;
    case OTG:
      s = OTGUtil.getDocumentFile(path, context,  createRecursive: false).length();
      break;
    case DROPBOX:
      s =
          dataUtils
              .getAccount(OpenMode.DROPBOX)
              .getMetadata(CloudUtil.stripPath(OpenMode.DROPBOX, path))
              .getSize();
      break;
    case BOX:
      s =
          dataUtils
              .getAccount(OpenMode.BOX)
              .getMetadata(CloudUtil.stripPath(OpenMode.BOX, path))
              .getSize();
      break;
```

Figure 17. Original method (part 1)



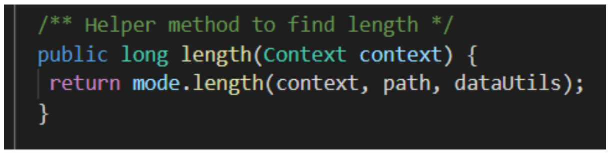
```
      case ONEDRIVE:
        s =
            dataUtils
                .getAccount(OpenMode.ONEDRIVE)
                .getMetadata(CloudUtil.stripPath(OpenMode.ONEDRIVE, path))
                .getSize();
        break;
      case GDRIVE:
        s =
            dataUtils
                .getAccount(OpenMode.GDRIVE)
                .getMetadata(CloudUtil.stripPath(OpenMode.GDRIVE, path))
                .getSize();
        break;
      default:
        break;
    }
    return s;
  }
```

Figure 18. Original method (part 2)



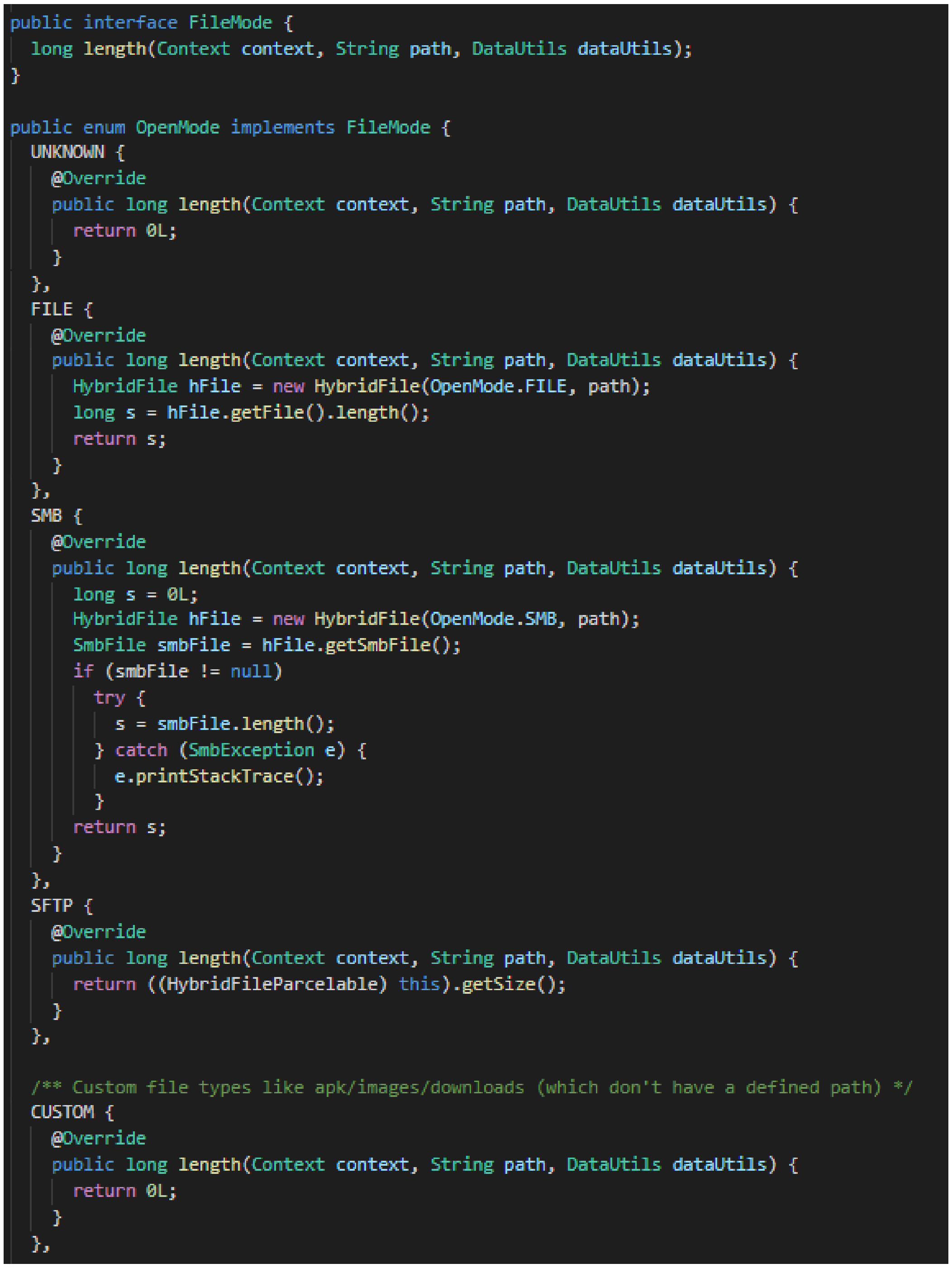
```
public enum OpenMode {
  UNKNOWN,
  FILE,
  SMB,
  SFTP,

  /** Custom file types like apk/images/downloads (which don't have a defined path) */
  CUSTOM,

  ROOT,
  OTG,
  DOCUMENT_FILE,
  GDRIVE,
  DROPBOX,
  BOX,
  ONEDRIVE;

  /**
   * Get open mode based on the id assigned. Generally used to retrieve this type after config
   * change or to send enum as argument
   *
   * @param ordinal the position of enum starting from 0 for first element
   */
  public static OpenMode getOpenMode(int ordinal) {
    return OpenMode.values()[ordinal];
  }
```

Figure 19. Original *OpenMode* enum



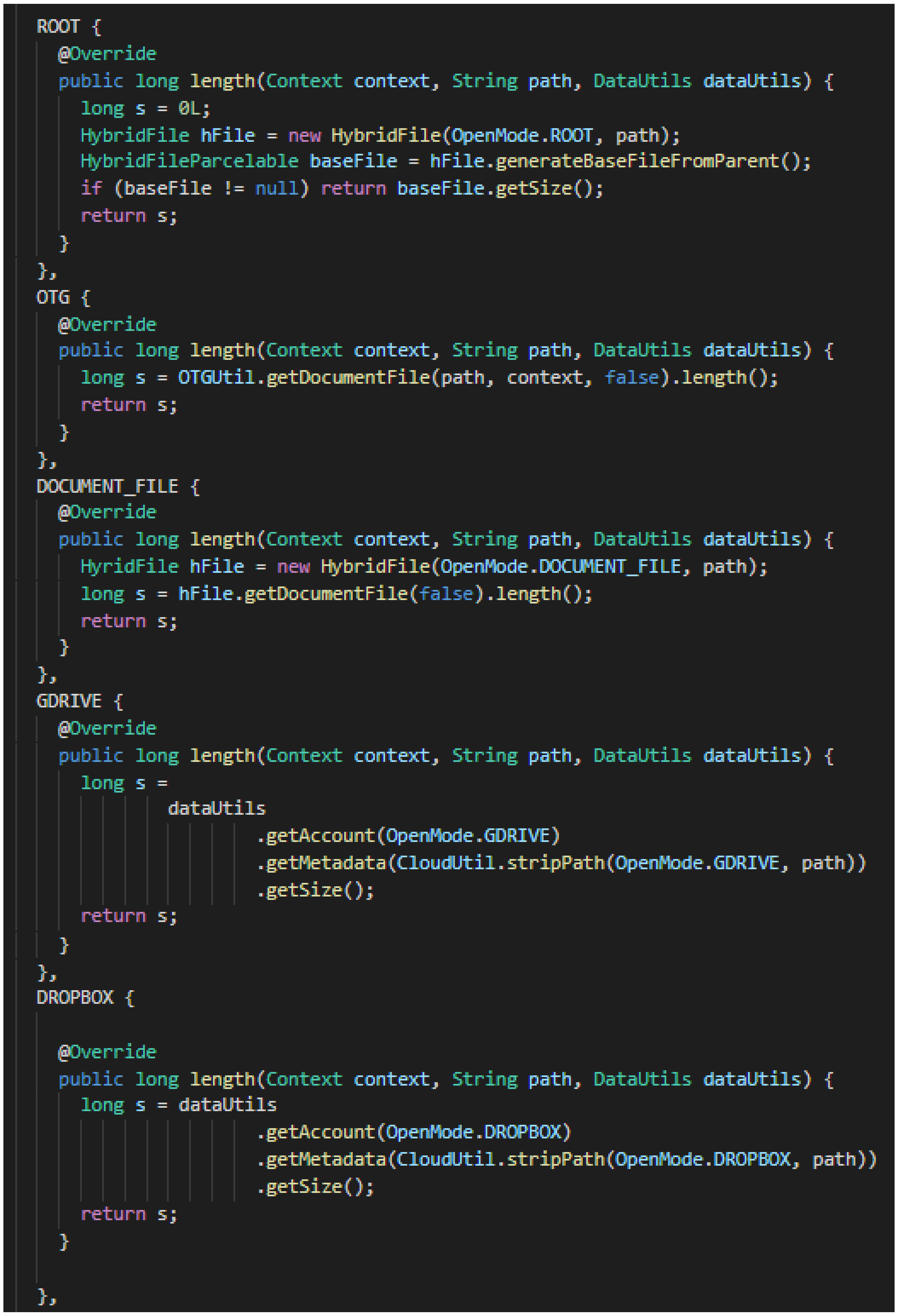
```
/** Helper method to find length */
public long length(Context context) {
  return mode.length(context, path, dataUtils);
}
```

Figure 20. Refactored method



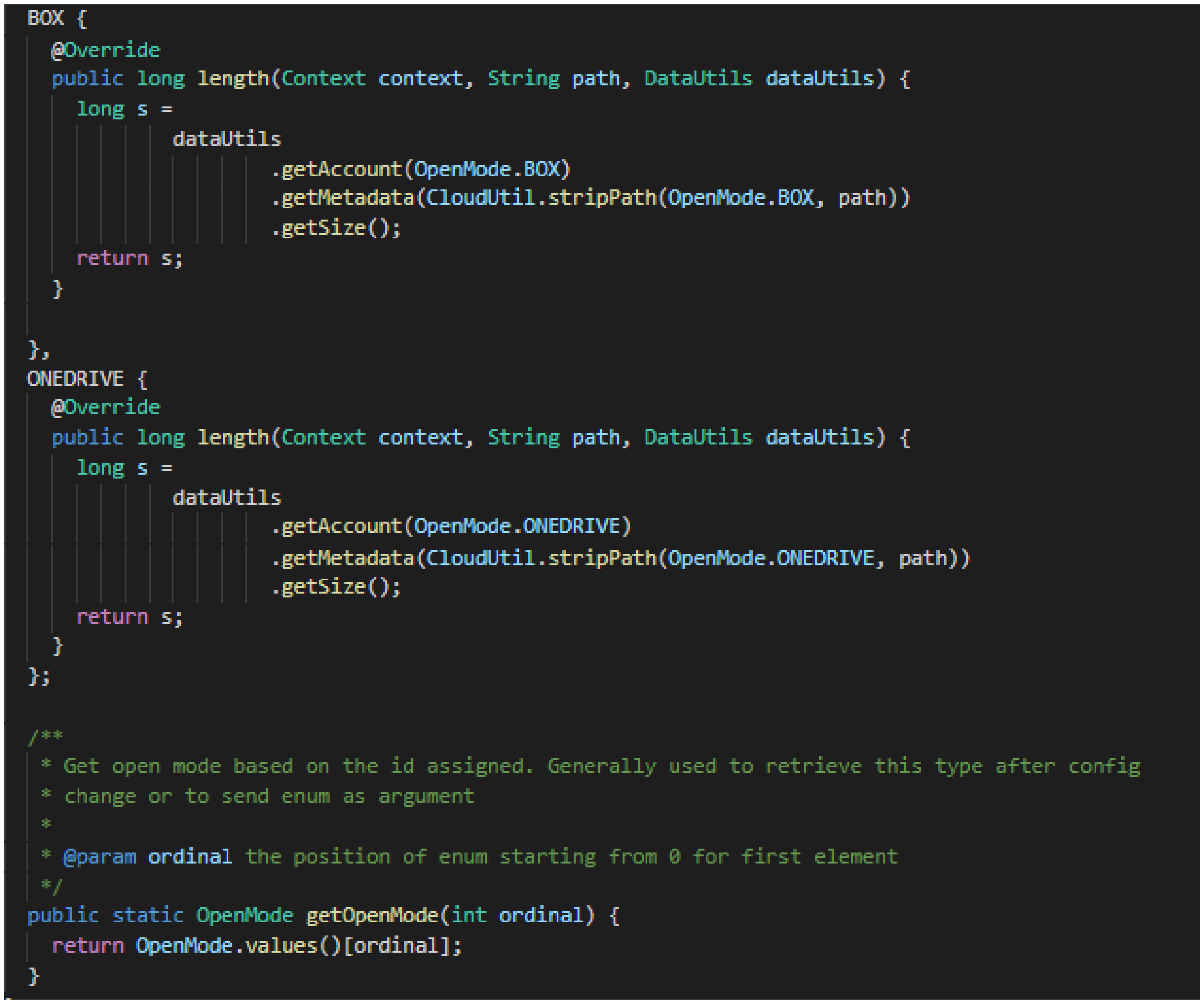
```java
public interface FileMode {
  long length(Context context, String path, DataUtils dataUtils);
}

public enum OpenMode implements FileMode {
  UNKNOWN {
    @Override
    public long length(Context context, String path, DataUtils dataUtils) {
      return 0L;
    }
  },
  FILE {
    @Override
    public long length(Context context, String path, DataUtils dataUtils) {
      HybridFile hFile = new HybridFile(OpenMode.FILE, path);
      long s = hFile.getFile().length();
      return s;
    }
  },
  SMB {
    @Override
    public long length(Context context, String path, DataUtils dataUtils) {
      long s = 0L;
      HybridFile hFile = new HybridFile(OpenMode.SMB, path);
      SmbFile smbFile = hFile.getSmbFile();
      if (smbFile != null)
        try {
          s = smbFile.length();
        } catch (SmbException e) {
          e.printStackTrace();
        }
      return s;
    }
  },
  SFTP {
    @Override
    public long length(Context context, String path, DataUtils dataUtils) {
      return ((HybridFileParcelable) this).getSize();
    }
  },

  /** Custom file types like apk/images/downloads (which don't have a defined path) */
  CUSTOM {
    @Override
    public long length(Context context, String path, DataUtils dataUtils) {
      return 0L;
    }
  },
```

Figure 21. Modified *OpenMode* enum (part 1)



```
ROOT {
  @Override
  public long length(Context context, String path, DataUtils dataUtils) {
    long s = 0L;
    HybridFile hFile = new HybridFile(OpenMode.ROOT, path);
    HybridFileParcelable baseFile = hFile.generateBaseFileFromParent();
    if (baseFile != null) return baseFile.getSize();
    return s;
  }
},
OTG {
  @Override
  public long length(Context context, String path, DataUtils dataUtils) {
    long s = OTGUtil.getDocumentFile(path, context, false).length();
    return s;
  }
},
DOCUMENT_FILE {
  @Override
  public long length(Context context, String path, DataUtils dataUtils) {
    HyridFile hFile = new HybridFile(OpenMode.DOCUMENT_FILE, path);
    long s = hFile.getDocumentFile(false).length();
    return s;
  }
},
GDRIVE {
  @Override
  public long length(Context context, String path, DataUtils dataUtils) {
    long s =
            dataUtils
                    .getAccount(OpenMode.GDRIVE)
                    .getMetadata(CloudUtil.stripPath(OpenMode.GDRIVE, path))
                    .getSize();
    return s;
  }
},
DROPBOX {

  @Override
  public long length(Context context, String path, DataUtils dataUtils) {
    long s = dataUtils
                    .getAccount(OpenMode.DROPBOX)
                    .getMetadata(CloudUtil.stripPath(OpenMode.DROPBOX, path))
                    .getSize();
    return s;
  }

},
```

Figure 22. Modified *OpenMode* enum (part 2)



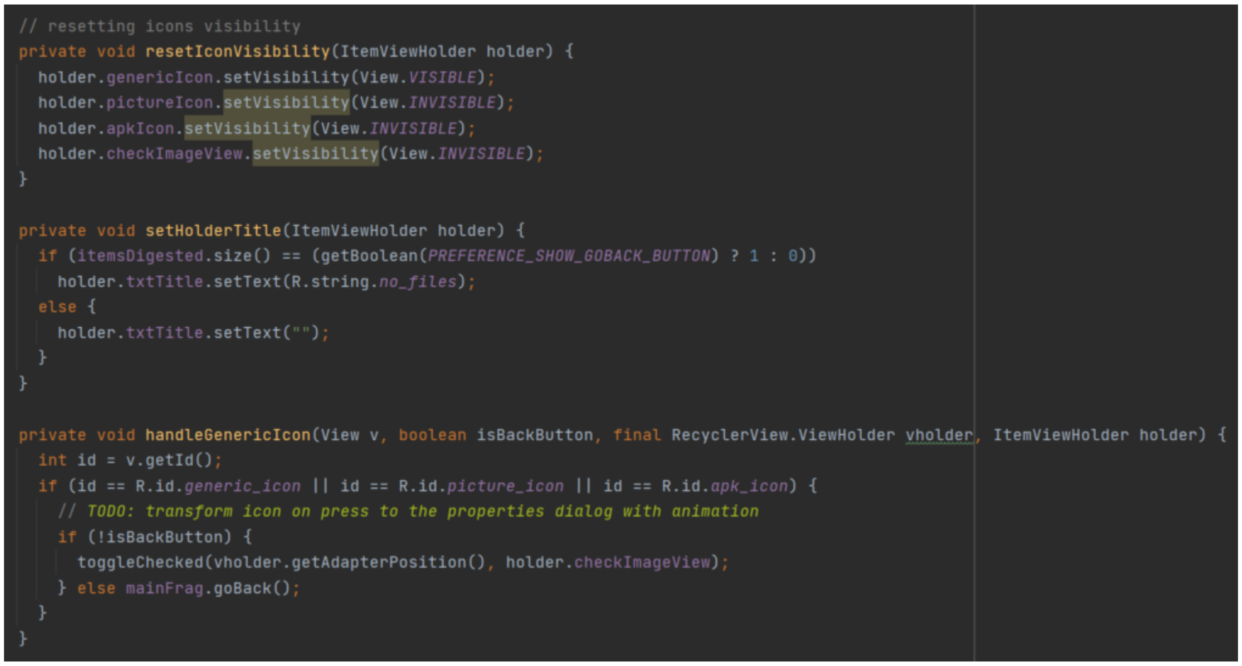
```
BOX {
  @Override
  public long length(Context context, String path, DataUtils dataUtils) {
    long s =
            dataUtils
                    .getAccount(OpenMode.BOX)
                    .getMetadata(CloudUtil.stripPath(OpenMode.BOX, path))
                    .getSize();
    return s;
  }

},
ONEDRIVE {
  @Override
  public long length(Context context, String path, DataUtils dataUtils) {
    long s =
            dataUtils
                    .getAccount(OpenMode.ONEDRIVE)
                    .getMetadata(CloudUtil.stripPath(OpenMode.ONEDRIVE, path))
                    .getSize();
    return s;
  }
};

/**
 * Get open mode based on the id assigned. Generally used to retrieve this type after config
 * change or to send enum as argument
 *
 * @param ordinal the position of enum starting from 0 for first element
 */
public static OpenMode getOpenMode(int ordinal) {
  return OpenMode.values()[ordinal];
}
```

Figure 23. Modified *OpenMode* enum (part 3)

2. Brain Method in *RecyclerAdapter.java*

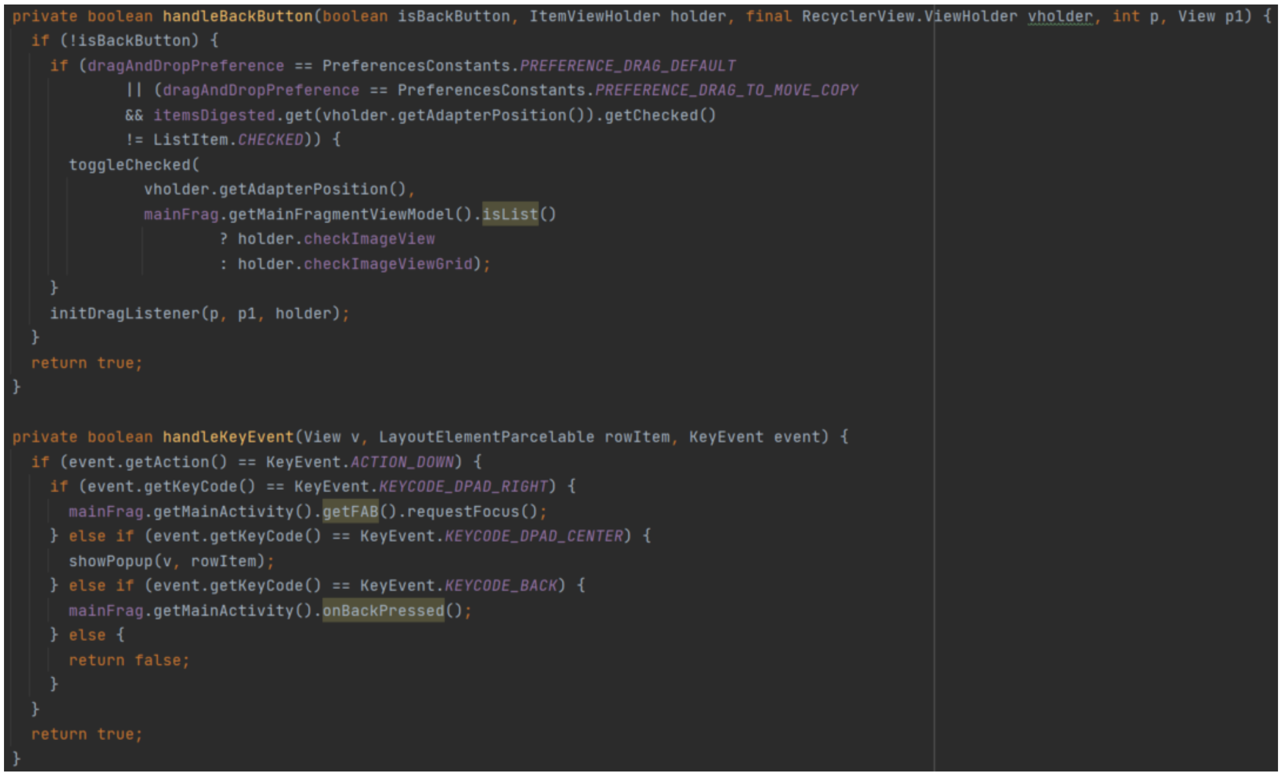
One of the fastest and most efficient ways to fix a Brain Method anomaly would be to extract or move the different methods and/or attributes used inside the Brain Method we are trying to correct. The method we were targeting was onBindViewHolder(ViewHolder, int), which was a lengthy method with 363 lines of code. Since its size was enormous, we decided to go over the code through each nested block. Indeed, since each of these blocks come from a conditional branch, we tried to identify a responsibility to each of these blocks. From there, if the code inside these blocks were too long or repetitive, we extracted them into independent private methods. Because of the size and complexity of this method, we did not do an exhaustive refactoring, meaning that we could have continued the extraction process furthermore.

The code shown in the methods of Figure 24 to Figure 26 is the code directly extracted from the original location. After the extraction, we simply replaced this code by the new private methods that we created according to the blocks' responsibilities, as we can see on Figure 27 and 28.



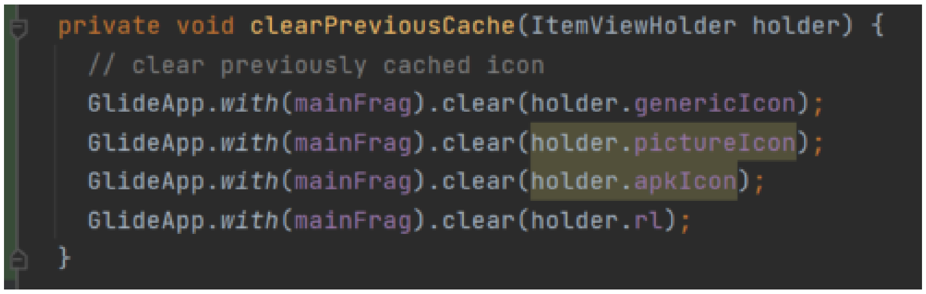
```java
// resetting icons visibility
private void resetIconVisibility(ItemViewHolder holder) {
  holder.genericIcon.setVisibility(View.VISIBLE);
  holder.pictureIcon.setVisibility(View.INVISIBLE);
  holder.apkIcon.setVisibility(View.INVISIBLE);
  holder.checkImageView.setVisibility(View.INVISIBLE);
}

private void setHolderTitle(ItemViewHolder holder) {
  if (itemsDigested.size() == (getBoolean(PREFERENCE_SHOW_GOBACK_BUTTON) ? 1 : 0))
    holder.txtTitle.setText(R.string.no_files);
  else {
    holder.txtTitle.setText("");
  }
}

private void handleGenericIcon(View v, boolean isBackButton, final RecyclerView.ViewHolder vholder, ItemViewHolder holder) {
  int id = v.getId();
  if (id == R.id.generic_icon || id == R.id.picture_icon || id == R.id.apk_icon) {
    // TODO: transform icon on press to the properties dialog with animation
    if (!isBackButton) {
      toggleChecked(vholder.getAdapterPosition(), holder.checkImageView);
    } else mainFrag.goBack();
  }
}
```

Figure 24. Extracted methods (part 1)



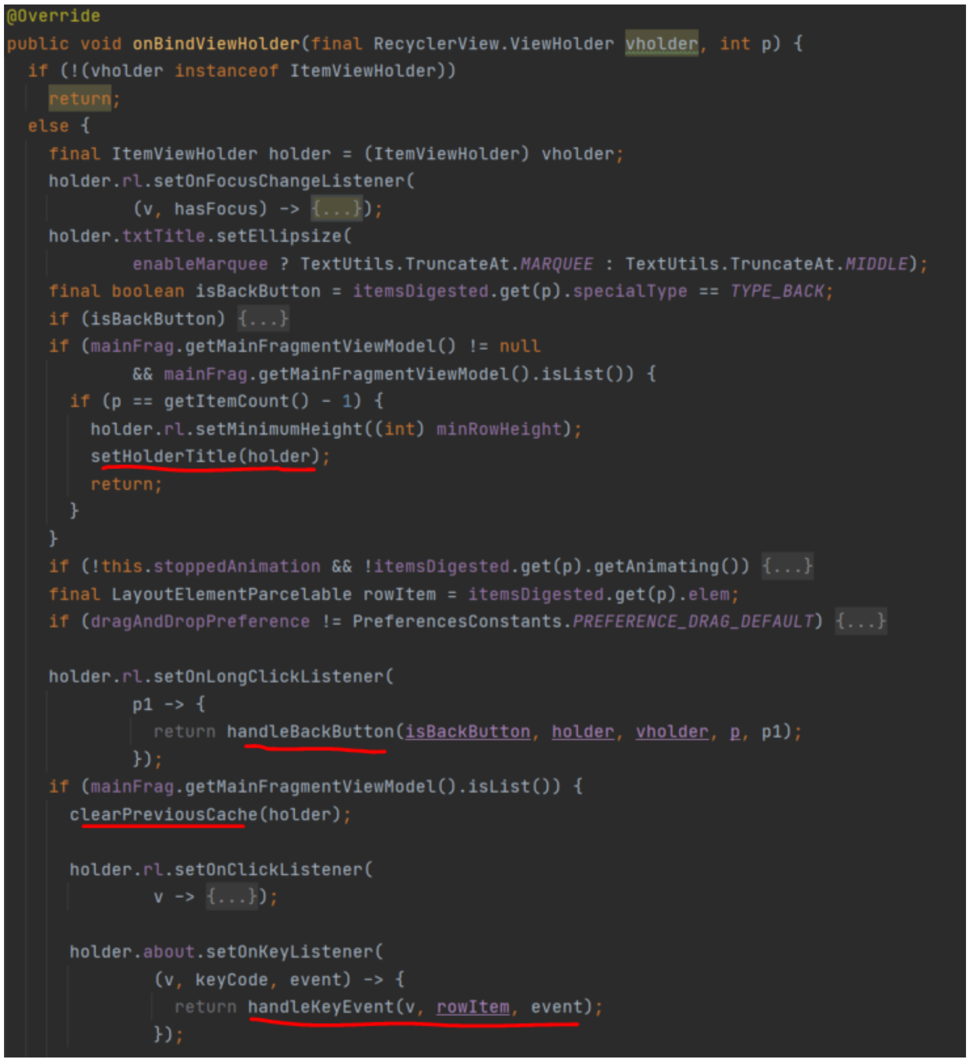
```java
private boolean handleBackButton(boolean isBackButton, ItemViewHolder holder, final RecyclerView.ViewHolder vholder, int p, View p1) {
  if (!isBackButton) {
    if (dragAndDropPreference == PreferencesConstants.PREFERENCE_DRAG_DEFAULT
            || (dragAndDropPreference == PreferencesConstants.PREFERENCE_DRAG_TO_MOVE_COPY
            && itemsDigested.get(vholder.getAdapterPosition()).getChecked()
            != ListItem.CHECKED)) {
      toggleChecked(
              vholder.getAdapterPosition(),
              mainFrag.getMainFragmentViewModel().isList()
                      ? holder.checkImageView
                      : holder.checkImageViewGrid);
    }
    initDragListener(p, p1, holder);
  }
  return true;
}

private boolean handleKeyEvent(View v, LayoutElementParcelable rowItem, KeyEvent event) {
  if (event.getAction() == KeyEvent.ACTION_DOWN) {
    if (event.getKeyCode() == KeyEvent.KEYCODE_DPAD_RIGHT) {
      mainFrag.getMainActivity().getFAB().requestFocus();
    } else if (event.getKeyCode() == KeyEvent.KEYCODE_DPAD_CENTER) {
      showPopup(v, rowItem);
    } else if (event.getKeyCode() == KeyEvent.KEYCODE_BACK) {
      mainFrag.getMainActivity().onBackPressed();
    } else {
      return false;
    }
  }
  return true;
}
```

Figure 25. Extracted methods (part 2)



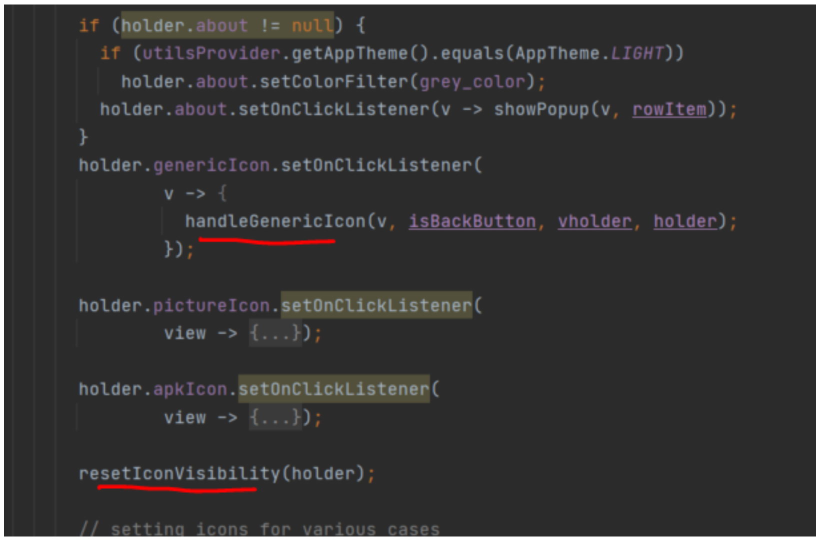
```java
private void clearPreviousCache(ItemViewHolder holder) {
    // clear previously cached icon
    GlideApp.with(mainFrag).clear(holder.genericIcon);
    GlideApp.with(mainFrag).clear(holder.pictureIcon);
    GlideApp.with(mainFrag).clear(holder.apkIcon);
    GlideApp.with(mainFrag).clear(holder.rl);
}
```

Figure 26. Extracted methods (part 3)



```java
@Override
public void onBindViewHolder(final RecyclerView.ViewHolder vholder, int p) {
  if (!(vholder instanceof ItemViewHolder))
    return;
  else {
    final ItemViewHolder holder = (ItemViewHolder) vholder;
    holder.rl.setOnFocusChangeListener(
            (v, hasFocus) -> {...});
    holder.txtTitle.setEllipsize(
            enableMarquee ? TextUtils.TruncateAt.MARQUEE : TextUtils.TruncateAt.MIDDLE);
    final boolean isBackButton = itemsDigested.get(p).specialType == TYPE_BACK;
    if (isBackButton) {...}
    if (mainFrag.getMainFragmentViewModel() != null
            && mainFrag.getMainFragmentViewModel().isList()) {
      if (p == getItemCount() - 1) {
        holder.rl.setMinimumHeight((int) minRowHeight);
        setHolderTitle(holder);
        return;
      }
    }
    if (!this.stoppedAnimation && !itemsDigested.get(p).getAnimating()) {...}
    final LayoutElementParcelable rowItem = itemsDigested.get(p).elem;
    if (dragAndDropPreference != PreferencesConstants.PREFERENCE_DRAG_DEFAULT) {...}

    holder.rl.setOnLongClickListener(
            p1 -> {
              return handleBackButton(isBackButton, holder, vholder, p, p1);
            });
    if (mainFrag.getMainFragmentViewModel().isList()) {
      clearPreviousCache(holder);

      holder.rl.setOnClickListener(
              v -> {...});

      holder.about.setOnKeyListener(
              (v, keyCode, event) -> {
                return handleKeyEvent(v, rowItem, event);
              });
```

Figure 27. Replacing the code blocks by the extracted methods (part 1)

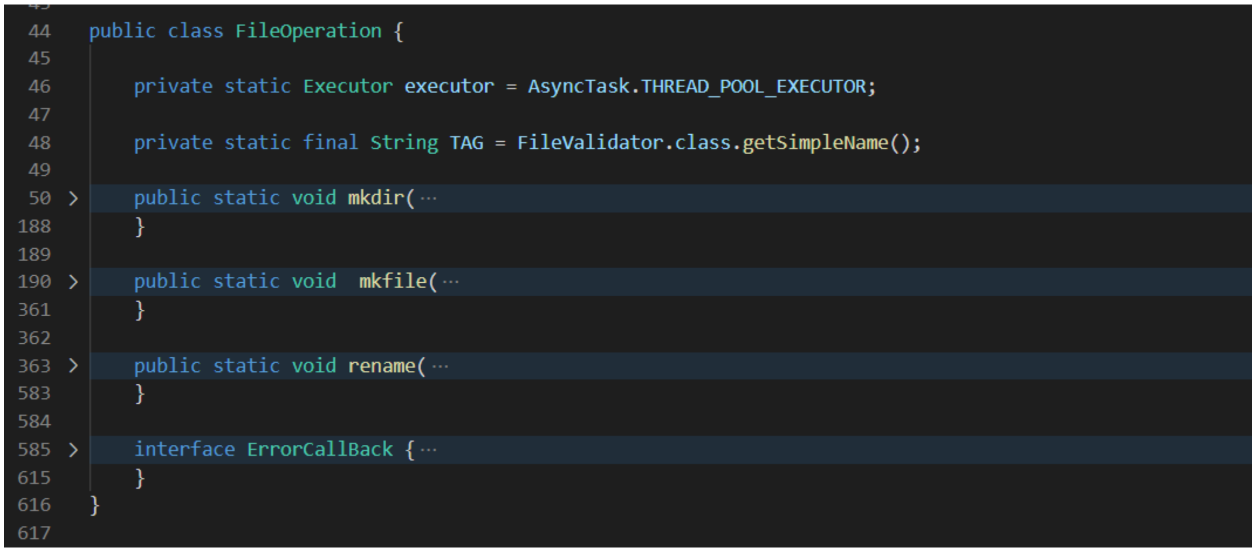


```
if (holder.about != null) {
  if (utilsProvider.getAppTheme().equals(AppTheme.LIGHT))
    holder.about.setColorFilter(grey_color);
  holder.about.setOnClickListener(v -> showPopup(v, rowItem));
}
holder.genericIcon.setOnClickListener(
        v -> {
          handleGenericIcon(v, isBackButton, vholder, holder);
        });

holder.pictureIcon.setOnClickListener(
        view -> {...});

holder.apkIcon.setOnClickListener(
        view -> {...});

resetIconVisibility(holder);

// setting icons for various cases
```

Figure 28. Replacing the code blocks by the extracted methods (part 2)

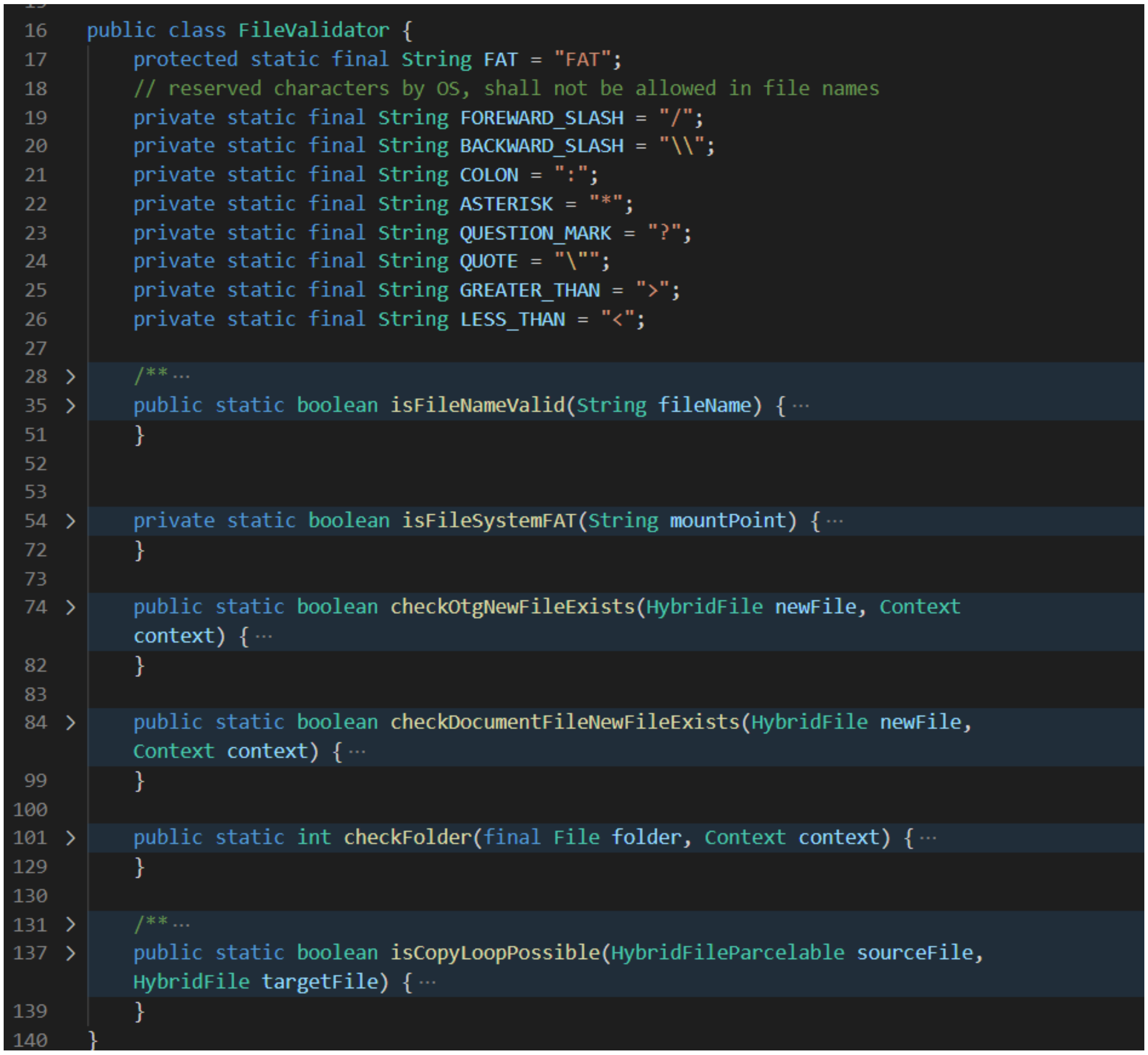
3. God Class in *Operations.java*

For this anomaly the idea was to separate the logic of the class by creating new classes. This process is called class Extraction which is commonly used to refactor a God Class. After a detailed examination of the code, we decided to divide the logic into 2 files which are FileOperator and FileValidator. As the name suggests, the former is where we put the logic of the file operations namely mkdir, mkfile and rename. As for the latter, we put the validation methods there namely isFileNameValid, isFileSystemFat, etc. These new classes completely replace the old Operations class. By separating the logic into 2 classes, the overall complexity decreases since the new classes are smaller (they contain less methods) and their content is more cohesive since they are more logically related.



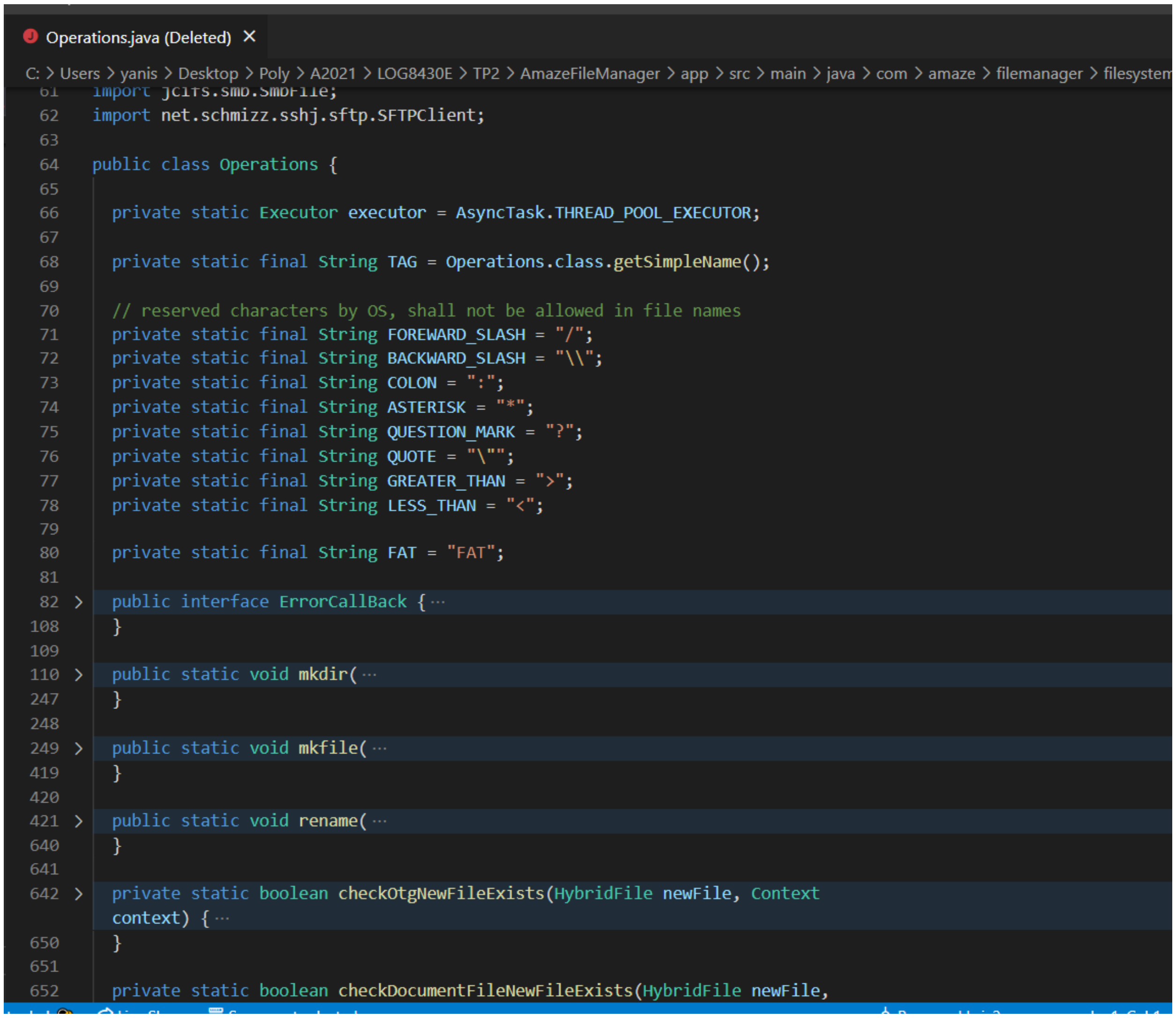
```
public class FileOperation {

    private static Executor executor = AsyncTask.THREAD_POOL_EXECUTOR;

    private static final String TAG = FileValidator.class.getSimpleName();

    public static void mkdir(···
    }

    public static void  mkfile(···
    }

    public static void rename(···
    }

    interface ErrorCallBack {···
    }
}
```

Figure 29. New class *FileOperation.java*



```java
public class FileValidator {
    protected static final String FAT = "FAT";
    // reserved characters by OS, shall not be allowed in file names
    private static final String FOREWARD_SLASH = "/";
    private static final String BACKWARD_SLASH = "\\";
    private static final String COLON = ":";
    private static final String ASTERISK = "*";
    private static final String QUESTION_MARK = "?";
    private static final String QUOTE = "\"";
    private static final String GREATER_THAN = ">";
    private static final String LESS_THAN = "<";

    /**...
    public static boolean isFileNameValid(String fileName) {...
    }


    private static boolean isFileSystemFAT(String mountPoint) {...
    }

    public static boolean checkOtgNewFileExists(HybridFile newFile, Context
    context) {...
    }

    public static boolean checkDocumentFileNewFileExists(HybridFile newFile,
    Context context) {...
    }

    public static int checkFolder(final File folder, Context context) {...
    }

    /**...
    public static boolean isCopyLoopPossible(HybridFileParcelable sourceFile,
    HybridFile targetFile) {...
    }
}
```

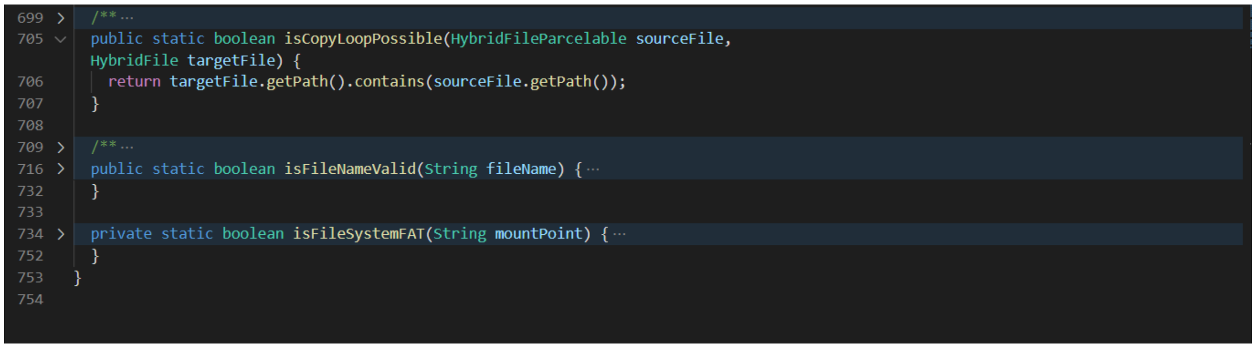
Figure 30. New class *FileValidator.java*


Operations.java (Deleted) ✕

C: > Users > yanis > Desktop > Poly > A2021 > LOG8430E > TP2 > AmazeFileManager > app > src > main > java > com > amaze > filemanager > filesystem

```java
import jcifs.smb.SmbFile;
import net.schmizz.sshj.sftp.SFTPClient;

public class Operations {

  private static Executor executor = AsyncTask.THREAD_POOL_EXECUTOR;

  private static final String TAG = Operations.class.getSimpleName();

  // reserved characters by OS, shall not be allowed in file names
  private static final String FOREWARD_SLASH = "/";
  private static final String BACKWARD_SLASH = "\\";
  private static final String COLON = ":";
  private static final String ASTERISK = "*";
  private static final String QUESTION_MARK = "?";
  private static final String QUOTE = "\"";
  private static final String GREATER_THAN = ">";
  private static final String LESS_THAN = "<";

  private static final String FAT = "FAT";

  public interface ErrorCallBack {…
  }

  public static void mkdir(…
  }

  public static void mkfile(…
  }

  public static void rename(…
  }

  private static boolean checkOtgNewFileExists(HybridFile newFile, Context
  context) {…
  }

  private static boolean checkDocumentFileNewFileExists(HybridFile newFile,
```

Figure 31. Original *Operations.java* (part 1)

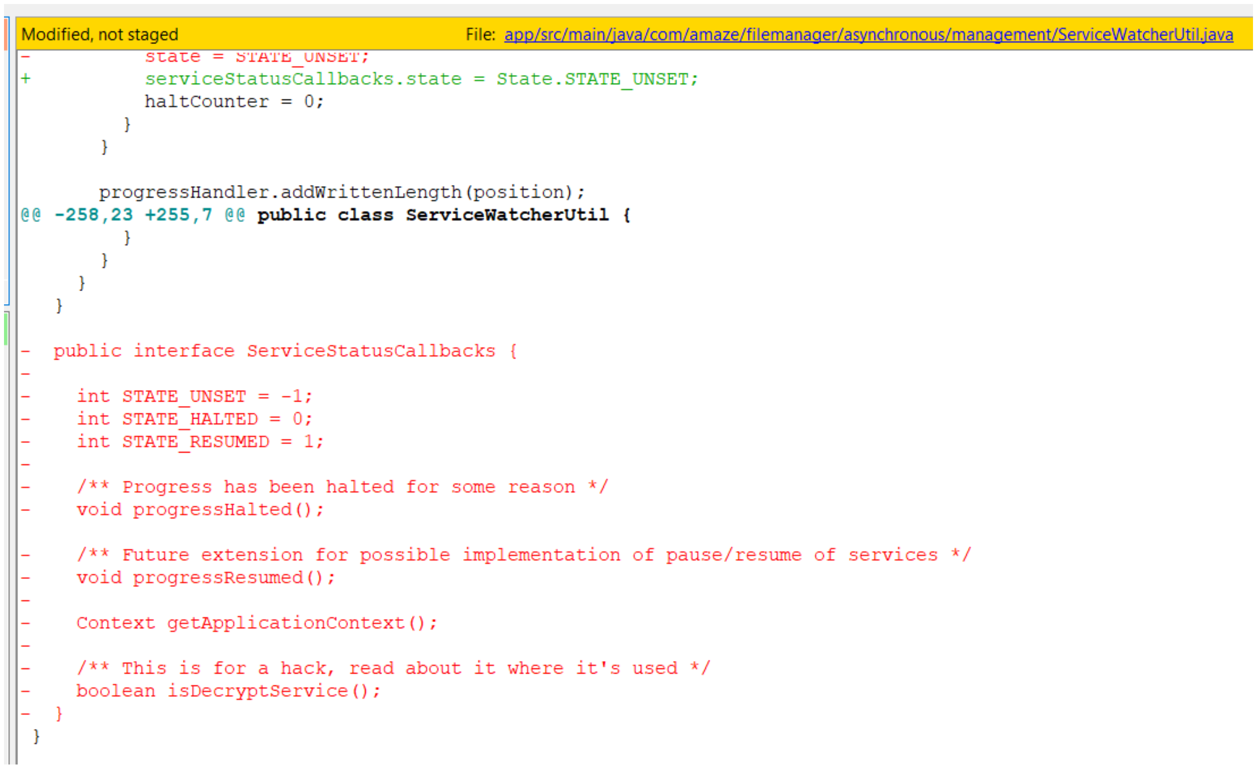


```java
  /**…
  public static boolean isCopyLoopPossible(HybridFileParcelable sourceFile,
  HybridFile targetFile) {
    return targetFile.getPath().contains(sourceFile.getPath());
  }

  /**…
  public static boolean isFileNameValid(String fileName) {…
  }

  private static boolean isFileSystemFAT(String mountPoint) {…
  }
}
```

Figure 32: Original *Operations.java* (part 2)

4. Feature Envy in *AbstractProgressiveService.java*

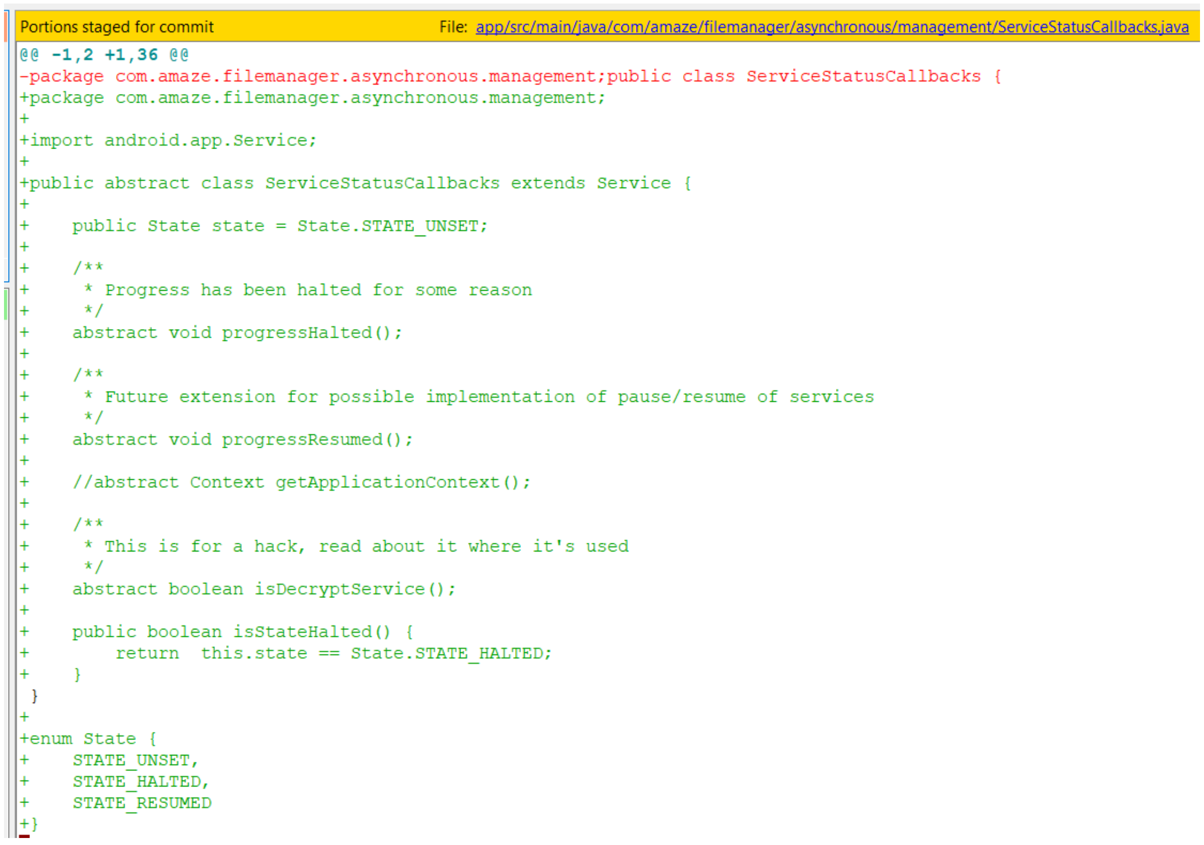
For this anomaly, the idea was to move the attributes and the methods closer to the class using it. This process is called ‘Move methods’ which is commonly used to refactor a Feature Envy. Many fixes were applied to this class. First, we moved the attribute state from the class *ServiceWatcherUtil* to the class *ServiceStatusCallbacks*, which *AbstractProgressiveService* extends. By doing so, the attribute state can be referred directly by the *AbstractProgressiveService* class instead of referencing it through *ServiceWatcherUtil*. In order to do this, we had to transform the interface *ServiceStatusCallbacks* to an abstract class, make *ServiceStatus* extend the Service class and turn the private constant states to an enum. Secondly, we moved the formatting of the timer to the *AbstractProgressiveService* class from the Util class to bring the operation in the class.



Modified, not staged File: app/src/main/java/com/amaze/filemanager/asynchronous/management/ServiceWatcherUtil.java

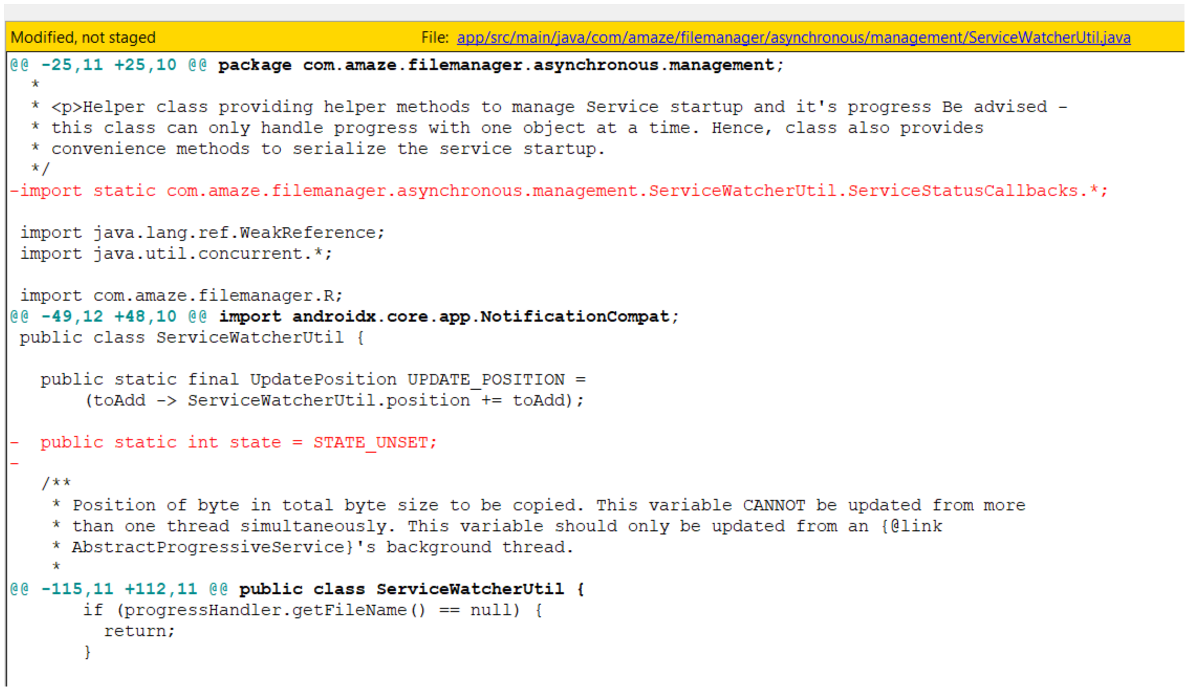
```
-           state = STATE_UNSET;
+           serviceStatusCallbacks.state = State.STATE_UNSET;
            haltCounter = 0;
          }
        }

        progressHandler.addWrittenLength(position);
@@ -258,23 +255,7 @@ public class ServiceWatcherUtil {
          }
        }
      }
    }

-   public interface ServiceStatusCallbacks {
-
-     int STATE_UNSET = -1;
-     int STATE_HALTED = 0;
-     int STATE_RESUMED = 1;
-
-     /** Progress has been halted for some reason */
-     void progressHalted();

-     /** Future extension for possible implementation of pause/resume of services */
-     void progressResumed();
-
-     Context getApplicationContext();
-
-     /** This is for a hack, read about it where it's used */
-     boolean isDecryptService();
-   }
  }
```

Figure 33. Original *ServiceStatusCallbacks.java* class



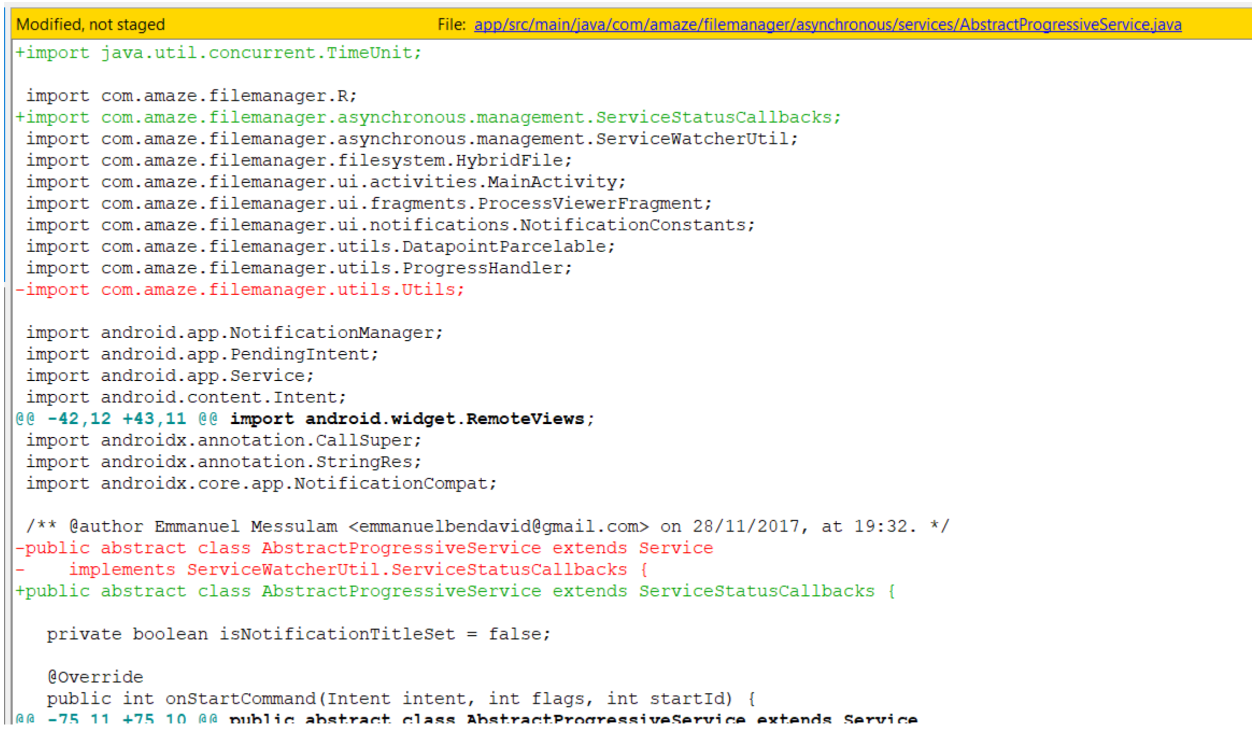
```
Portions staged for commit          File: app/src/main/java/com/amaze/filemanager/asynchronous/management/ServiceStatusCallbacks.java
@@ -1,2 +1,36 @@
-package com.amaze.filemanager.asynchronous.management;public class ServiceStatusCallbacks {
+package com.amaze.filemanager.asynchronous.management;
+
+import android.app.Service;
+
+public abstract class ServiceStatusCallbacks extends Service {
+
+    public State state = State.STATE_UNSET;
+
+    /**
+     * Progress has been halted for some reason
+     */
+    abstract void progressHalted();
+
+    /**
+     * Future extension for possible implementation of pause/resume of services
+     */
+    abstract void progressResumed();
+
+    //abstract Context getApplicationContext();
+
+    /**
+     * This is for a hack, read about it where it's used
+     */
+    abstract boolean isDecryptService();
+
+    public boolean isStateHalted() {
+        return  this.state == State.STATE_HALTED;
+    }
 }
+
+enum State {
+    STATE_UNSET,
+    STATE_HALTED,
+    STATE_RESUMED
+}
```

Figure 34. New *ServiceStatusCallbacks.java* class



```
Modified, not staged          File: app/src/main/java/com/amaze/filemanager/asynchronous/management/ServiceWatcherUtil.java
@@ -25,11 +25,10 @@ package com.amaze.filemanager.asynchronous.management;
  *
  * <p>Helper class providing helper methods to manage Service startup and it's progress Be advised -
  * this class can only handle progress with one object at a time. Hence, class also provides
  * convenience methods to serialize the service startup.
  */
-import static com.amaze.filemanager.asynchronous.management.ServiceWatcherUtil.ServiceStatusCallbacks.*;

 import java.lang.ref.WeakReference;
 import java.util.concurrent.*;

 import com.amaze.filemanager.R;
@@ -49,12 +48,10 @@ import androidx.core.app.NotificationCompat;
 public class ServiceWatcherUtil {

   public static final UpdatePosition UPDATE_POSITION =
       (toAdd -> ServiceWatcherUtil.position += toAdd);

-  public static int state = STATE_UNSET;
-
   /**
    * Position of byte in total byte size to be copied. This variable CANNOT be updated from more
    * than one thread simultaneously. This variable should only be updated from an {@link
    * AbstractProgressiveService}'s background thread.
    *
@@ -115,11 +112,11 @@ public class ServiceWatcherUtil {
       if (progressHandler.getFileName() == null) {
         return;
       }
```

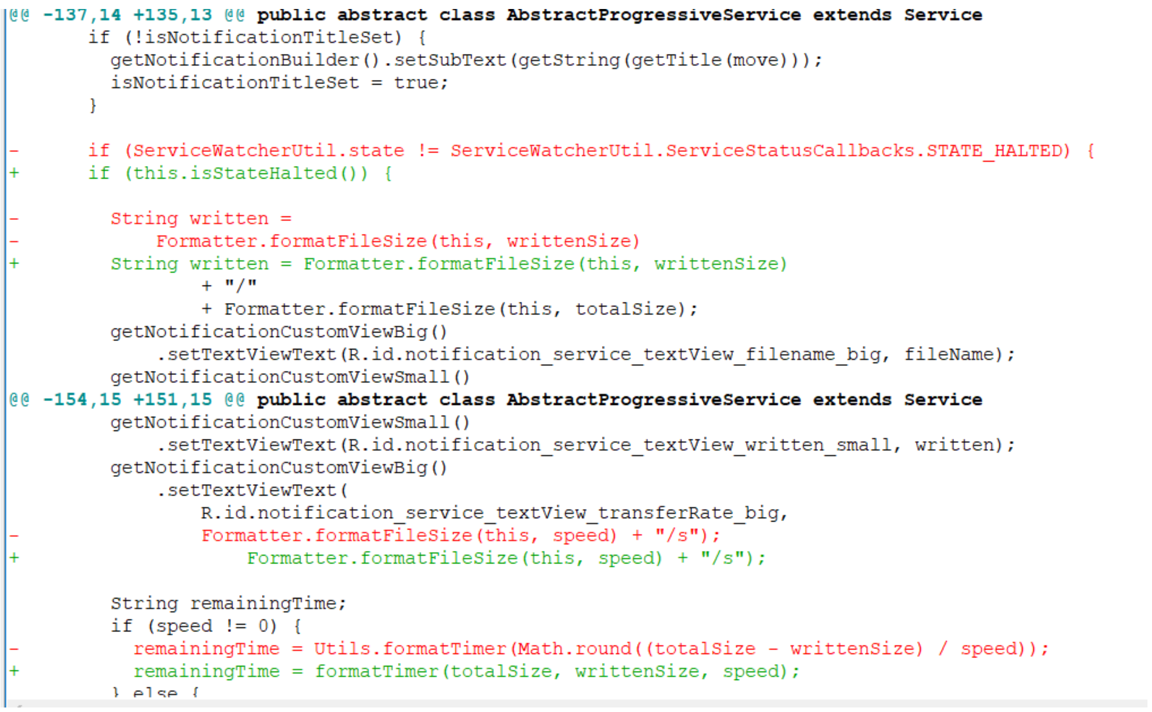
Figure 35. Old ServiceWatcherUtil.java class



Modified, not staged   File: app/src/main/java/com/amaze/filemanager/asynchronous/services/AbstractProgressiveService.java

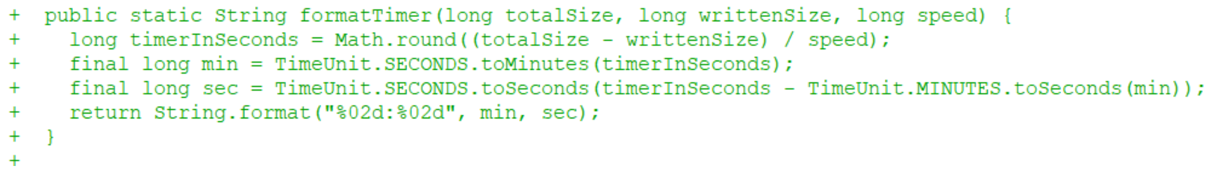
```
+import java.util.concurrent.TimeUnit;

 import com.amaze.filemanager.R;
+import com.amaze.filemanager.asynchronous.management.ServiceStatusCallbacks;
 import com.amaze.filemanager.asynchronous.management.ServiceWatcherUtil;
 import com.amaze.filemanager.filesystem.HybridFile;
 import com.amaze.filemanager.ui.activities.MainActivity;
 import com.amaze.filemanager.ui.fragments.ProcessViewerFragment;
 import com.amaze.filemanager.ui.notifications.NotificationConstants;
 import com.amaze.filemanager.utils.DatapointParcelable;
 import com.amaze.filemanager.utils.ProgressHandler;
-import com.amaze.filemanager.utils.Utils;

 import android.app.NotificationManager;
 import android.app.PendingIntent;
 import android.app.Service;
 import android.content.Intent;
@@ -42,12 +43,11 @@ import android.widget.RemoteViews;
 import androidx.annotation.CallSuper;
 import androidx.annotation.StringRes;
 import androidx.core.app.NotificationCompat;

 /** @author Emmanuel Messulam <emmanuelbendavid@gmail.com> on 28/11/2017, at 19:32. */
-public abstract class AbstractProgressiveService extends Service
-    implements ServiceWatcherUtil.ServiceStatusCallbacks {
+public abstract class AbstractProgressiveService extends ServiceStatusCallbacks {

   private boolean isNotificationTitleSet = false;

   @Override
   public int onStartCommand(Intent intent, int flags, int startId) {
@@ -75,11 +75,10 @@ public abstract class AbstractProgressiveService extends Service
```

Figure 36. Original and new *AbstractProgressiveService.java* class



```
@@ -137,14 +135,13 @@ public abstract class AbstractProgressiveService extends Service
       if (!isNotificationTitleSet) {
         getNotificationBuilder().setSubText(getString(getTitle(move)));
         isNotificationTitleSet = true;
       }

-      if (ServiceWatcherUtil.state != ServiceWatcherUtil.ServiceStatusCallbacks.STATE_HALTED) {
+      if (this.isStateHalted()) {

-        String written =
-            Formatter.formatFileSize(this, writtenSize)
+        String written = Formatter.formatFileSize(this, writtenSize)
                 + "/"
                 + Formatter.formatFileSize(this, totalSize);
         getNotificationCustomViewBig()
             .setTextViewText(R.id.notification_service_textView_filename_big, fileName);
         getNotificationCustomViewSmall()
@@ -154,15 +151,15 @@ public abstract class AbstractProgressiveService extends Service
         getNotificationCustomViewSmall()
             .setTextViewText(R.id.notification_service_textView_written_small, written);
         getNotificationCustomViewBig()
             .setTextViewText(
                 R.id.notification_service_textView_transferRate_big,
-                Formatter.formatFileSize(this, speed) + "/s");
+                    Formatter.formatFileSize(this, speed) + "/s");

         String remainingTime;
         if (speed != 0) {
-          remainingTime = Utils.formatTimer(Math.round((totalSize - writtenSize) / speed));
+          remainingTime = formatTimer(totalSize, writtenSize, speed);
         } else {
```

Figure 37. Original and new *publishResults* methods in *AbstractProgressiveService.java* class

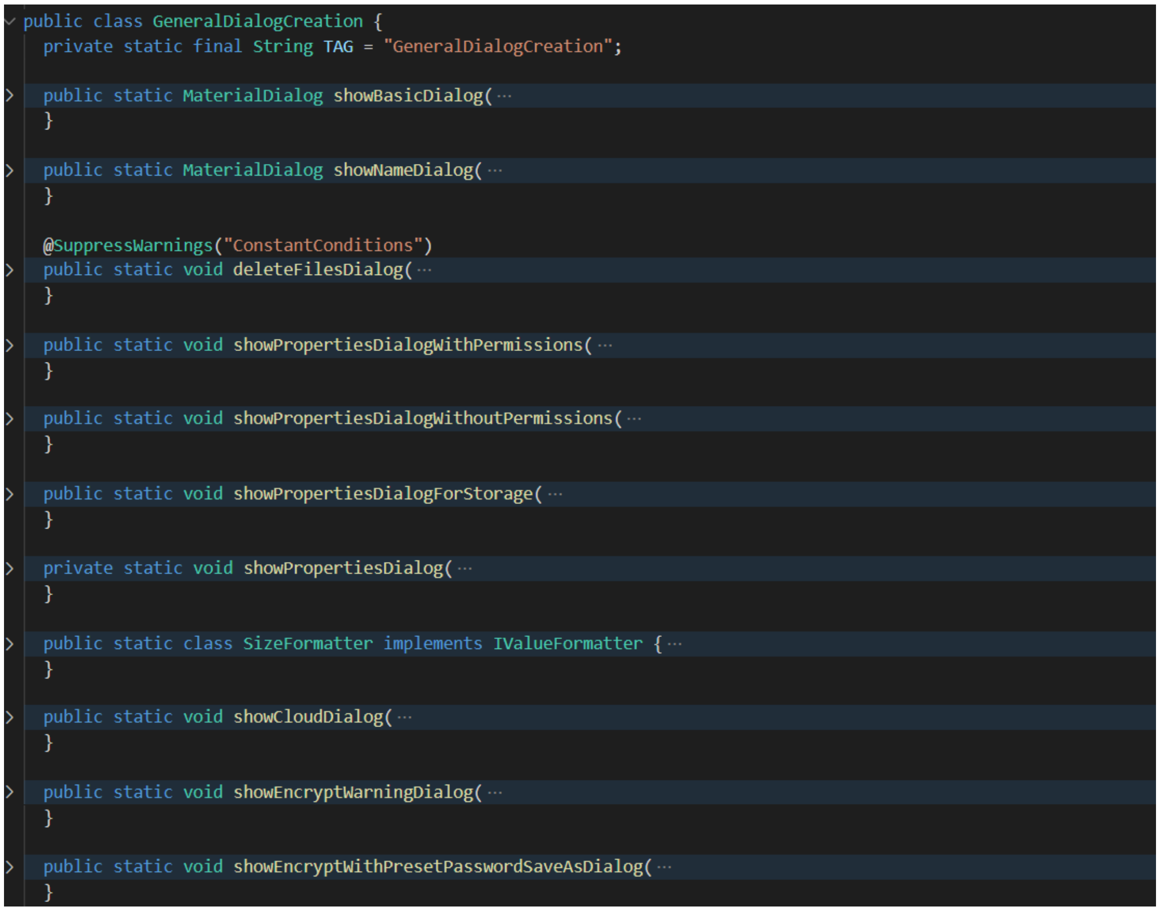


```
+  public static String formatTimer(long totalSize, long writtenSize, long speed) {
+    long timerInSeconds = Math.round((totalSize - writtenSize) / speed);
+    final long min = TimeUnit.SECONDS.toMinutes(timerInSeconds);
+    final long sec = TimeUnit.SECONDS.toSeconds(timerInSeconds - TimeUnit.MINUTES.toSeconds(min));
+    return String.format("%02d:%02d", min, sec);
+  }
+
```

Figure 38. New *formatTimer* method used in the *publishResults* method

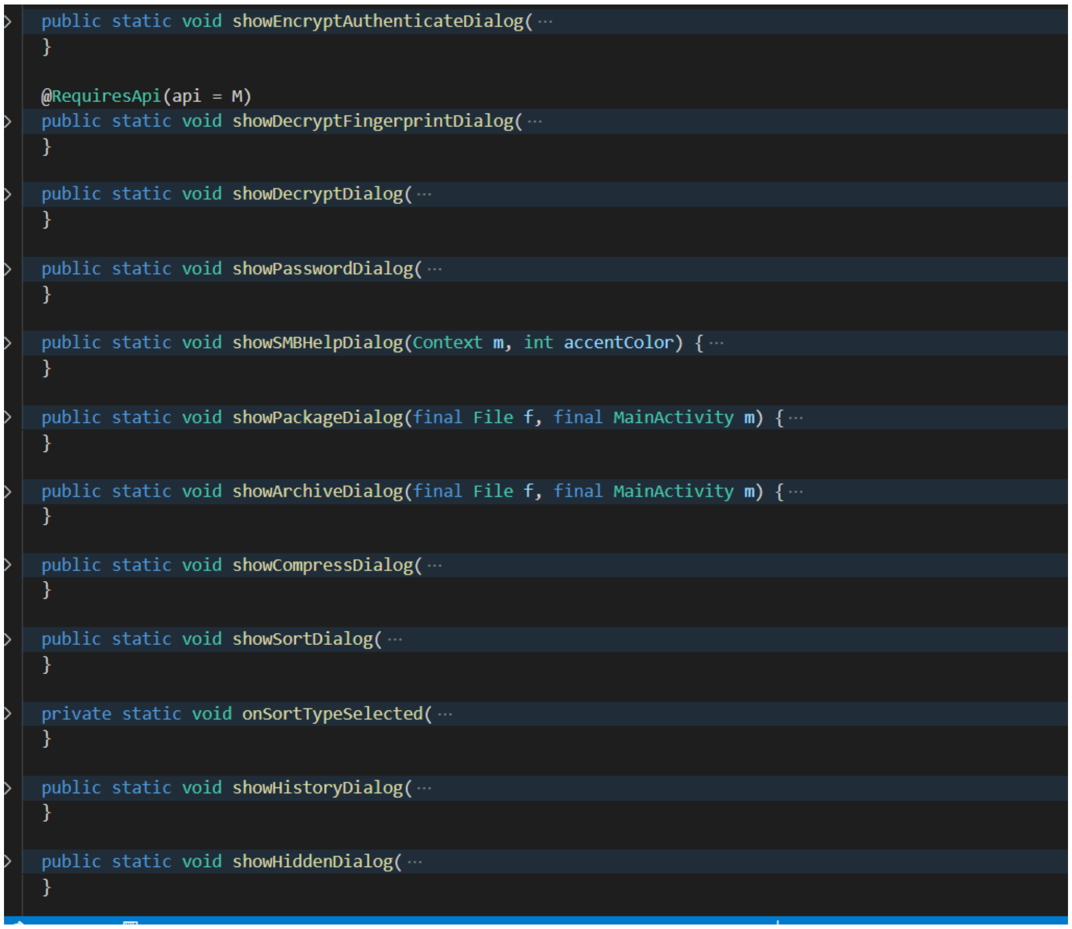
5. God Class in *GeneralDialogCreation.java*

Similarly to the code smell identified on 3. God Class in Operations.java, the GeneralDialogCreation is a God Class that can be fixed by separating the class' logic into extracted new classes. To do so, we went over the code and noticed that the class contained many methods that did a similar action: showPropertiesDialog (Figure 39), showCloudDialog (Figure 39), showDecryptDialog (Figure 40), showPasswordDialog (Figure 40), etc. Since these classes all did a similar thing but through different ways, we thought that each of them could have their own class. However, as we were limited by time, we simply extracted three classes in order to analyze the new metric values obtained through a non-exhaustive refactoring. Therefore, some old operations that showed a particular dialog were separated into: PasswordDialog.java (which replaced the *showPasswordDialog's* method logic), NameDialog.java (which replaced the *showNameDialog*'s method logic) and BasicDialog.java (which replaced the *showBasicDialog's* method logic). Again, similarly to the refactoring done on 3. God Class in Operations.java, separating the logic into three new classes is a good way to decrease GeneralDialogCreation's complexity.



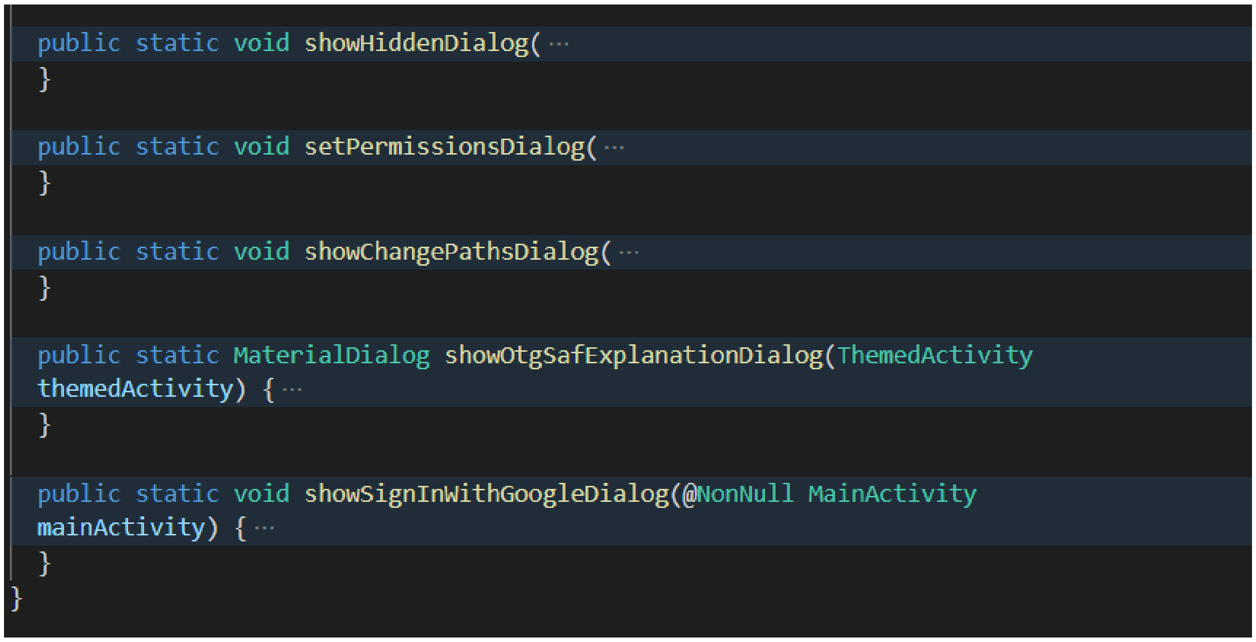
```java
public class GeneralDialogCreation {
    private static final String TAG = "GeneralDialogCreation";

    public static MaterialDialog showBasicDialog( ···
    }

    public static MaterialDialog showNameDialog( ···
    }

    @SuppressWarnings("ConstantConditions")
    public static void deleteFilesDialog( ···
    }

    public static void showPropertiesDialogWithPermissions( ···
    }

    public static void showPropertiesDialogWithoutPermissions( ···
    }

    public static void showPropertiesDialogForStorage( ···
    }

    private static void showPropertiesDialog( ···
    }

    public static class SizeFormatter implements IValueFormatter { ···
    }

    public static void showCloudDialog( ···
    }

    public static void showEncryptWarningDialog( ···
    }

    public static void showEncryptWithPresetPasswordSaveAsDialog( ···
    }
```

Figure 39. Original *GeneralDialogCreation.java* class



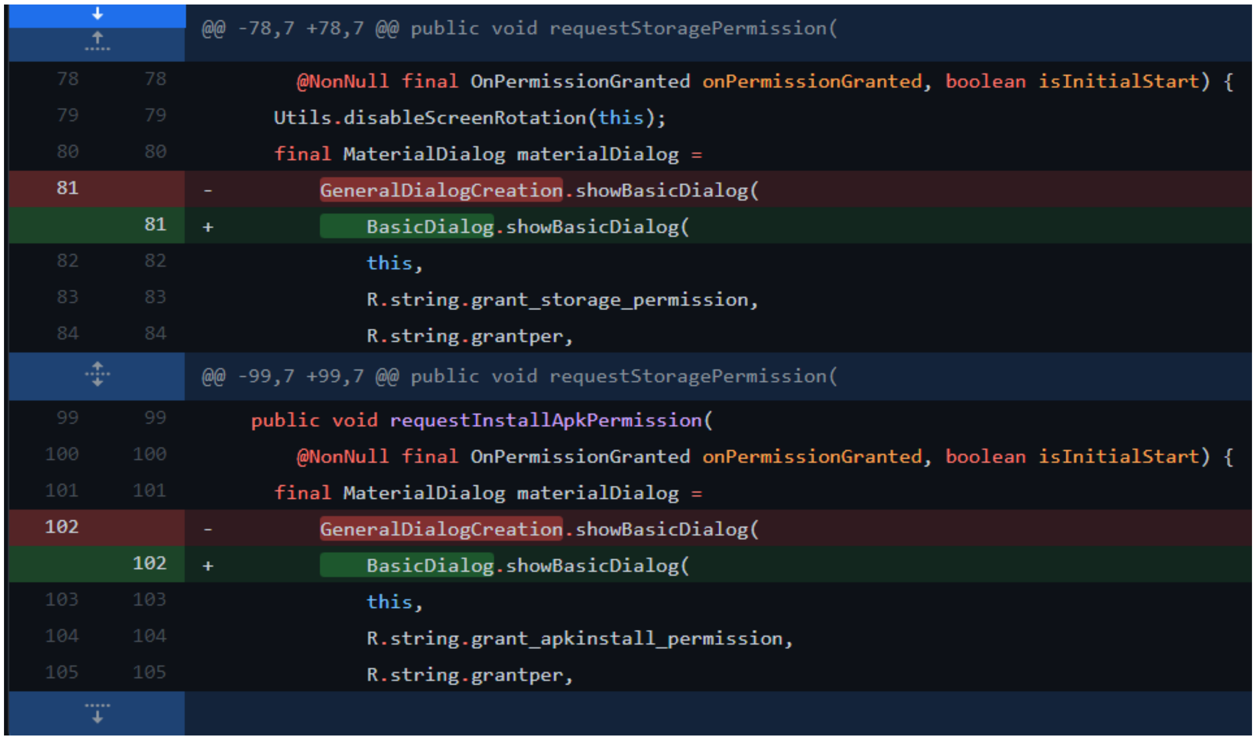
```
public static void showEncryptAuthenticateDialog( ···
}

@RequiresApi(api = M)
public static void showDecryptFingerprintDialog( ···
}

public static void showDecryptDialog( ···
}

public static void showPasswordDialog( ···
}

public static void showSMBHelpDialog(Context m, int accentColor) { ···
}

public static void showPackageDialog(final File f, final MainActivity m) { ···
}

public static void showArchiveDialog(final File f, final MainActivity m) { ···
}

public static void showCompressDialog( ···
}

public static void showSortDialog( ···
}

private static void onSortTypeSelected( ···
}

public static void showHistoryDialog( ···
}

public static void showHiddenDialog( ···
}
```

Figure 40. Original *GeneralDialogCreation.java* class



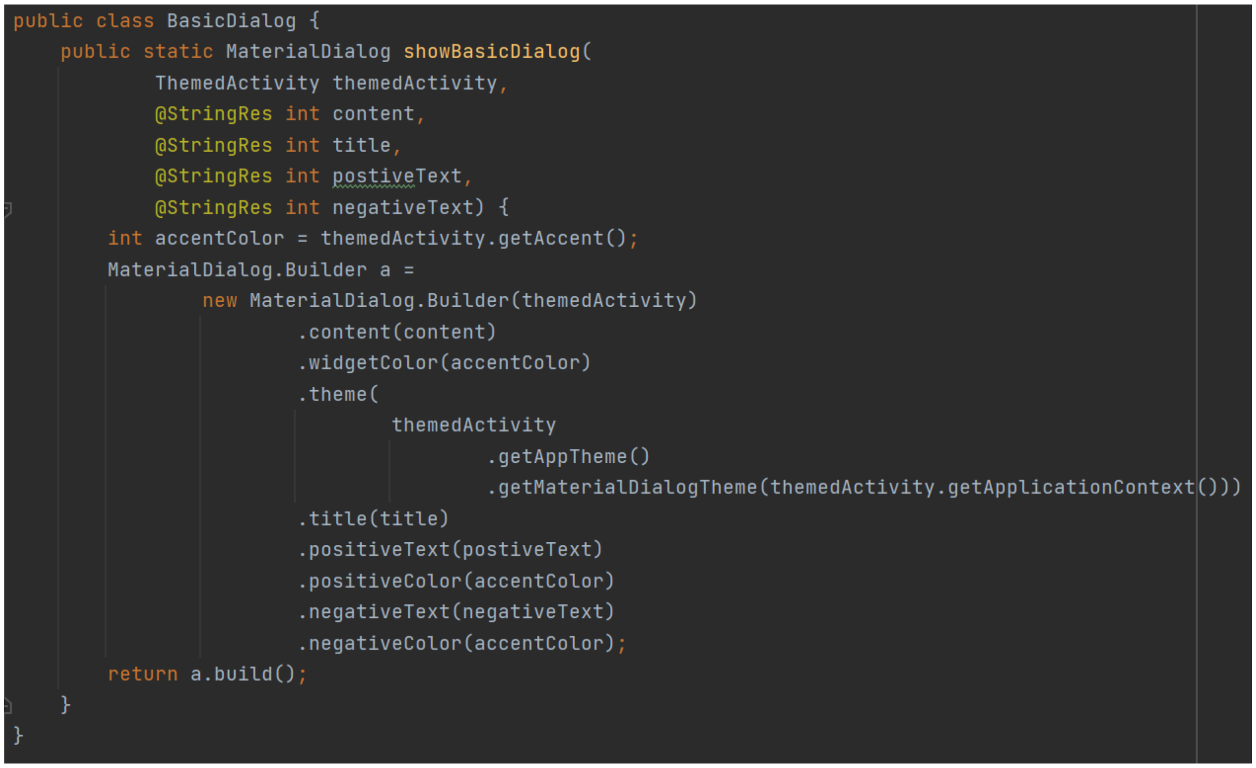
```
    public static void showHiddenDialog( ···
    }

    public static void setPermissionsDialog( ···
    }

    public static void showChangePathsDialog( ···
    }

    public static MaterialDialog showOtgSafExplanationDialog(ThemedActivity
    themedActivity) { ···
    }

    public static void showSignInWithGoogleDialog(@NonNull MainActivity
    mainActivity) { ···
    }
}
```

Figure 41. Original *GeneralDialogCreation.java* class



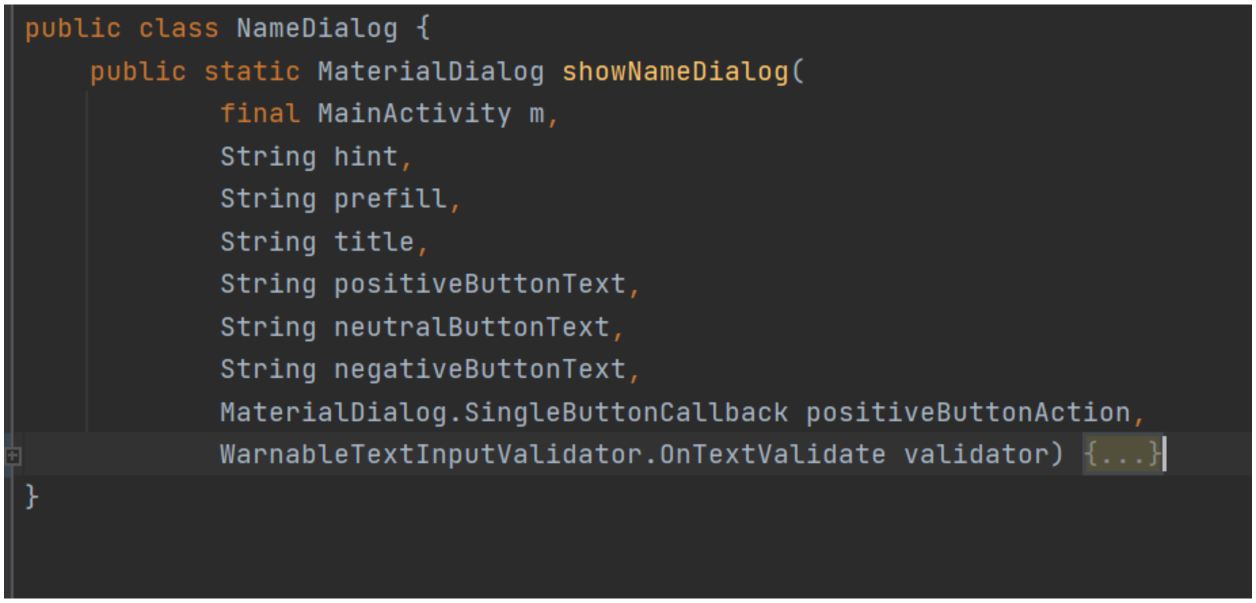
```diff
@@ -78,7 +78,7 @@ public void requestStoragePermission(
             @NonNull final OnPermissionGranted onPermissionGranted, boolean isInitialStart) {
           Utils.disableScreenRotation(this);
           final MaterialDialog materialDialog =
-              GeneralDialogCreation.showBasicDialog(
+                  BasicDialog.showBasicDialog(
                   this,
                   R.string.grant_storage_permission,
                   R.string.grantper,
@@ -99,7 +99,7 @@ public void requestStoragePermission(
         public void requestInstallApkPermission(
             @NonNull final OnPermissionGranted onPermissionGranted, boolean isInitialStart) {
           final MaterialDialog materialDialog =
-              GeneralDialogCreation.showBasicDialog(
+                  BasicDialog.showBasicDialog(
                   this,
                   R.string.grant_apkinstall_permission,
                   R.string.grantper,
```

Figure 42. Original and new *GeneralDialogCreation.java* class



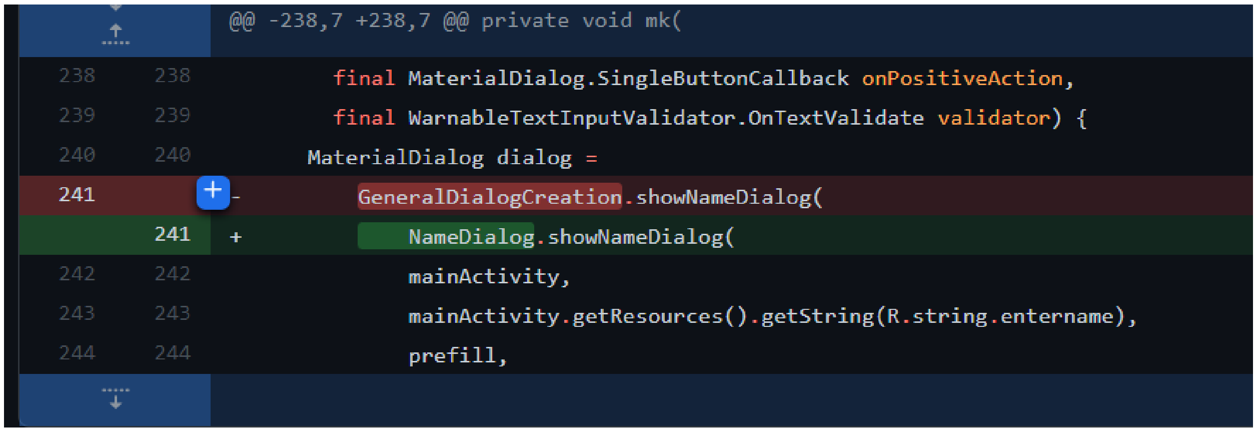
```java
public class BasicDialog {
    public static MaterialDialog showBasicDialog(
            ThemedActivity themedActivity,
            @StringRes int content,
            @StringRes int title,
            @StringRes int postiveText,
            @StringRes int negativeText) {
        int accentColor = themedActivity.getAccent();
        MaterialDialog.Builder a =
                new MaterialDialog.Builder(themedActivity)
                        .content(content)
                        .widgetColor(accentColor)
                        .theme(
                                themedActivity
                                        .getAppTheme()
                                        .getMaterialDialogTheme(themedActivity.getApplicationContext()))
                        .title(title)
                        .positiveText(postiveText)
                        .positiveColor(accentColor)
                        .negativeText(negativeText)
                        .negativeColor(accentColor);
        return a.build();
    }
}
```

Figure 43. Newly created *BasicDialog.java* class



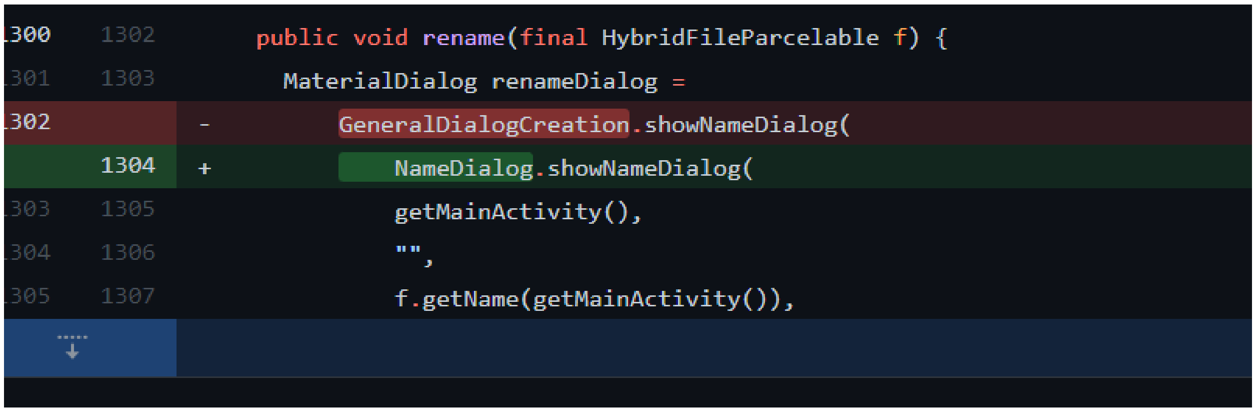
```java
public class NameDialog {
    public static MaterialDialog showNameDialog(
            final MainActivity m,
            String hint,
            String prefill,
            String title,
            String positiveButtonText,
            String neutralButtonText,
            String negativeButtonText,
            MaterialDialog.SingleButtonCallback positiveButtonAction,
            WarnableTextInputValidator.OnTextValidate validator) {...}
}
```

Figure 44. Newly created *NameDialog.java* class



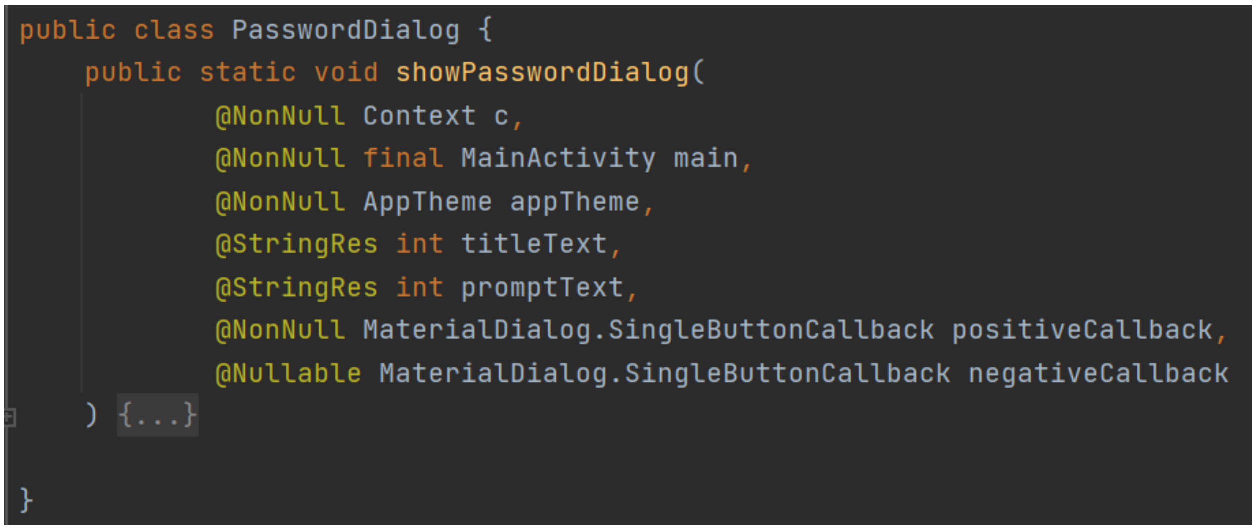
```diff
@@ -238,7 +238,7 @@ private void mk(
          final MaterialDialog.SingleButtonCallback onPositiveAction,
          final WarnableTextInputValidator.OnTextValidate validator) {
        MaterialDialog dialog =
-           GeneralDialogCreation.showNameDialog(
+           NameDialog.showNameDialog(
                mainActivity,
                mainActivity.getResources().getString(R.string.entername),
                prefill,
```

Figure 45. Example of an old code location where we used the new *NameDialog.java* class



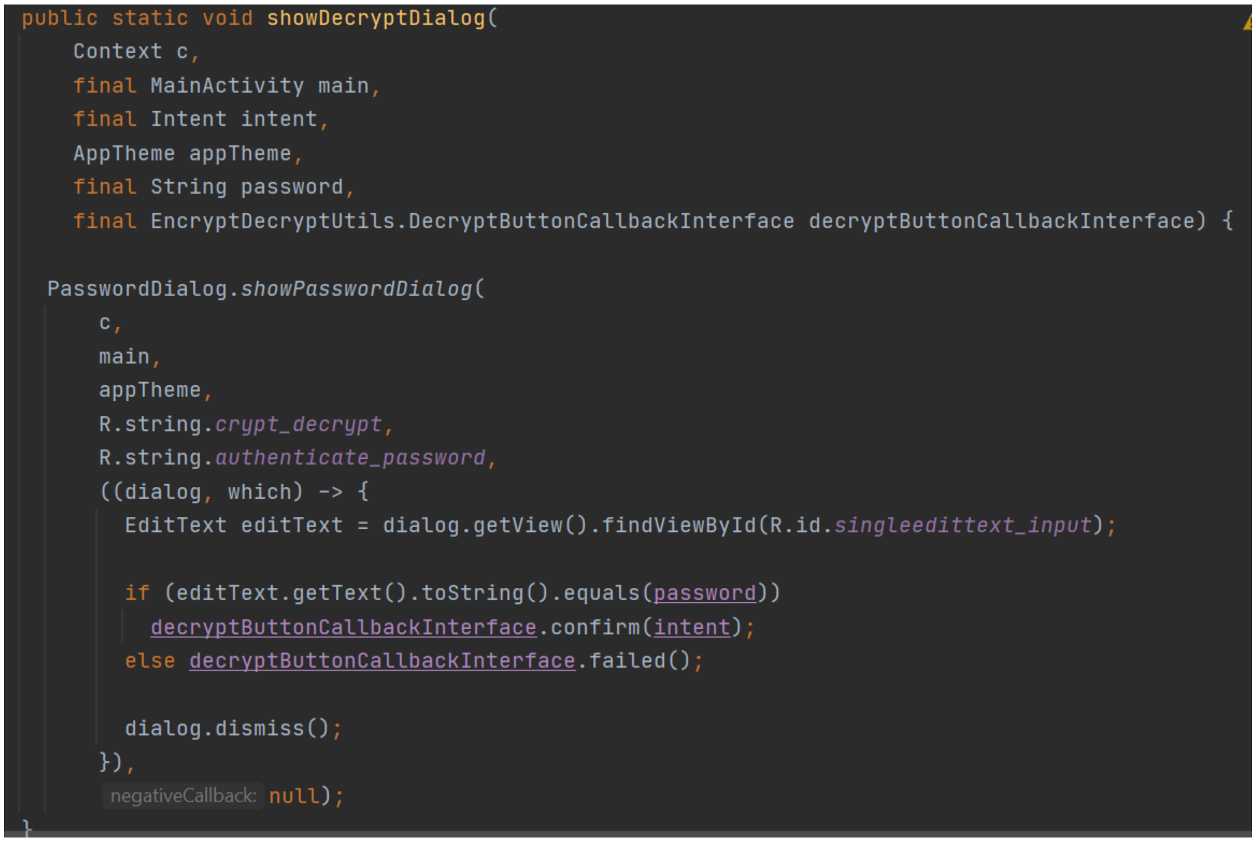
```diff
      public void rename(final HybridFileParcelable f) {
        MaterialDialog renameDialog =
-           GeneralDialogCreation.showNameDialog(
+           NameDialog.showNameDialog(
                getMainActivity(),
                "",
                f.getName(getMainActivity()),
```

Figure 46. Example of an old code location where we used the new *NameDialog.java* class



```java
public class PasswordDialog {
    public static void showPasswordDialog(
            @NonNull Context c,
            @NonNull final MainActivity main,
            @NonNull AppTheme appTheme,
            @StringRes int titleText,
            @StringRes int promptText,
            @NonNull MaterialDialog.SingleButtonCallback positiveCallback,
            @Nullable MaterialDialog.SingleButtonCallback negativeCallback
    ) {...}

}
```

Figure 47. Newly created *PasswordDialog.java* class

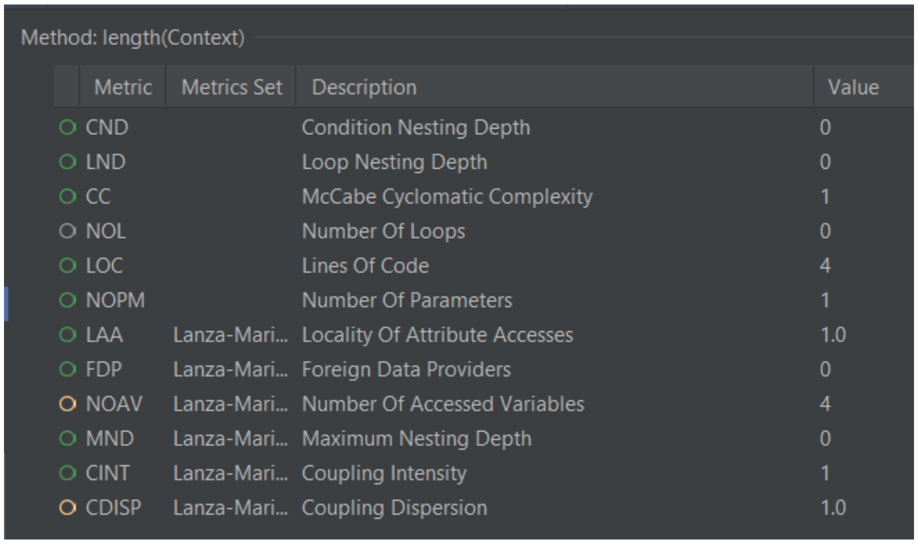


```java
public static void showDecryptDialog(
    Context c,
    final MainActivity main,
    final Intent intent,
    AppTheme appTheme,
    final String password,
    final EncryptDecryptUtils.DecryptButtonCallbackInterface decryptButtonCallbackInterface) {

  PasswordDialog.showPasswordDialog(
      c,
      main,
      appTheme,
      R.string.crypt_decrypt,
      R.string.authenticate_password,
      ((dialog, which) -> {
        EditText editText = dialog.getView().findViewById(R.id.singleedittext_input);

        if (editText.getText().toString().equals(password))
          decryptButtonCallbackInterface.confirm(intent);
        else decryptButtonCallbackInterface.failed();

        dialog.dismiss();
      }),
      negativeCallback: null);
}
```

Figure 48. Example of an old code location where we used the new *PasswordDialog.java* class

c)

1. Brain Method in *HybridFile.java*



Method: length(Context)

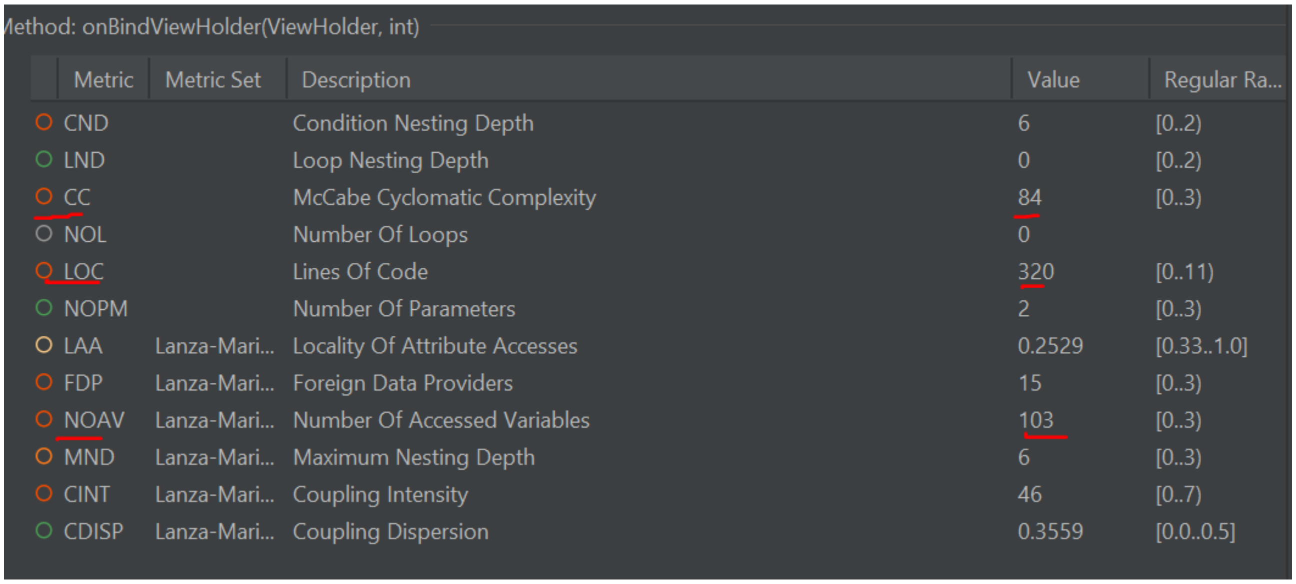
	Metric	Metrics Set	Description	Value
○	CND		Condition Nesting Depth	0
○	LND		Loop Nesting Depth	0
○	CC		McCabe Cyclomatic Complexity	1
○	NOL		Number Of Loops	0
○	LOC		Lines Of Code	4
○	NOPM		Number Of Parameters	1
○	LAA	Lanza-Mari...	Locality Of Attribute Accesses	1.0
○	FDP	Lanza-Mari...	Foreign Data Providers	0
○	NOAV	Lanza-Mari...	Number Of Accessed Variables	4
○	MND	Lanza-Mari...	Maximum Nesting Depth	0
○	CINT	Lanza-Mari...	Coupling Intensity	1
○	CDISP	Lanza-Mari...	Coupling Dispersion	1.0

Figure 49. *length* method's new metrics

As mentioned in the question 3a), by using the Strategy pattern and reusing the existing *OpenMode* interface, we were able to eliminate the switch statement, thus, reducing the nesting level, as well as the method's complexity and the number of accessed variables. Furthemore, we were able to shorten the method's definition in a substantial amount, therefore, reducing its lines of code. More precisely, if we were to compare the figure above with the method's initial metrics (see Figure 10), it is safe to say that we were able to reduce LOC from 61 to 4, which represents a 93% improvement. Moreover, NOAV also decreased from 18 to 4 (a 78% improvement) and MND, from 2 to 0.

In brief, this refactoring method was quite successful in correcting the Brain Method code smell in this *HybridFile.java*'s method.

2. Brain Method in *RecyclerAdapter.java*



lethod: onBindViewHolder(ViewHolder, int)

Metric	Metric Set	Description	Value	Regular Ra...
CND		Condition Nesting Depth	6	[0..2)
LND		Loop Nesting Depth	0	[0..2)
CC		McCabe Cyclomatic Complexity	84	[0..3)
NOL		Number Of Loops	0	
LOC		Lines Of Code	320	[0..11)
NOPM		Number Of Parameters	2	[0..3)
LAA	Lanza-Mari...	Locality Of Attribute Accesses	0.2529	[0.33..1.0]
FDP	Lanza-Mari...	Foreign Data Providers	15	[0..3)
NOAV	Lanza-Mari...	Number Of Accessed Variables	103	[0..3)
MND	Lanza-Mari...	Maximum Nesting Depth	6	[0..3)
CINT	Lanza-Mari...	Coupling Intensity	46	[0..7)
CDISP	Lanza-Mari...	Coupling Dispersion	0.3559	[0.0..0.5]

Figure 50. *onBindViewHolder* method's new metrics

As aforementioned, the refactoring for onBindViewHolder(ViewHolder, int) was not exhaustive. Therefore, the code smell was not fully fixed, as the metrics value for CC, LOC, NOAC, and MND are still very high. However, for the level of extraction we did, the value for CC, LOC and NOAV have significantly lowered. Indeed, as we can see on Figure 48 above, the Cyclomatic Complexity went from 99 to 84, which means that it got more than 15% lower. Also, the number of code lines went from 363 to 320, and the NOAV value went from 113 to 103. We can therefore say that the refactoring method we used was pretty effective. If time and energy were present, we could have fully refactored the class and corrected this code smell.

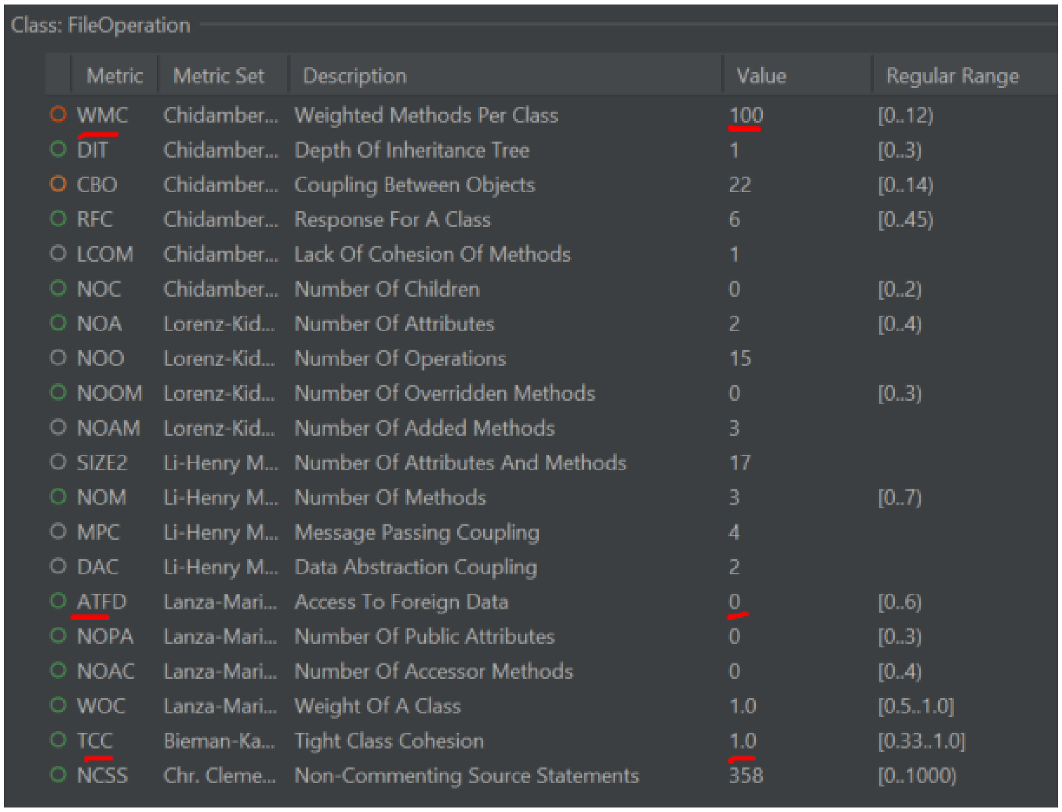
3. God Class in *Operations.java*

Since we completely removed the Operations.java class, we will analyze the metrics from the two classes (FileOperation.java and FileValidator.java) we extracted in order to prove that our refactoring worked. Again, as explained in the previous section, the metrics concerned are WMC, ATFD and TCC.

On Figure 49 for the new class FileOperation.java, we can see that WMC has a value of 100 which is still pretty high, but lower than the WMC of 128 in the old class Operations.java. The ATFD metric has a value of 0, which is now below FEW (2-5). Finally, still compared to the old class *Operations*, the TCC metric increased to 1 which is now above the value of ⅓. Since this new class does not match the three criterias for a God Class, we can say that the refactor was successful.

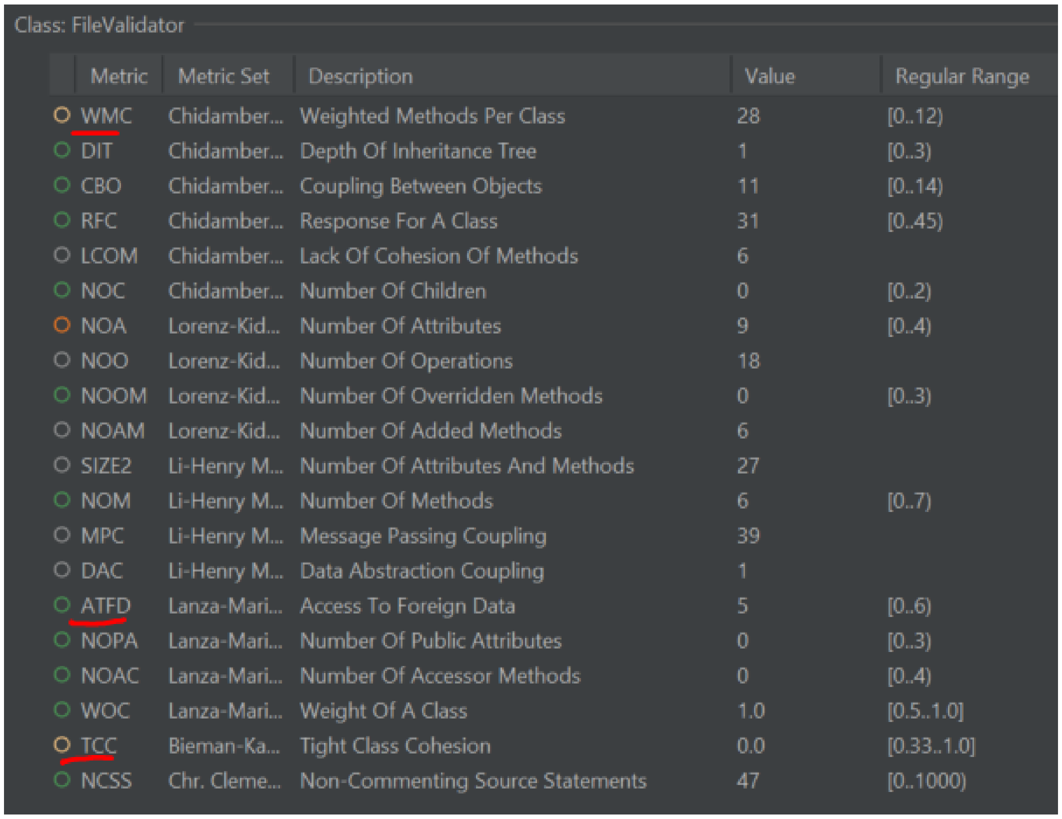
For the second class (FileValidator), we can see its metrics on Figure 50. In this case, even if it is still higher than 7 (VERY HIGH), the WMC lowered significantly to a value of 28. The TCC, with a value of 0, also corresponds to a criteria of God Class as it is below ⅓. However, the

ATFD metric got smaller and is now equal to 5, which is between the value 2-5 (= FEW). Therefore, we can also say that our refactoring worked well for this second class, since it does not contain a God Class anomaly.


Class: FileOperation

Metric	Metric Set	Description	Value	Regular Range
WMC	Chidamber...	Weighted Methods Per Class	100	[0..12)
DIT	Chidamber...	Depth Of Inheritance Tree	1	[0..3)
CBO	Chidamber...	Coupling Between Objects	22	[0..14)
RFC	Chidamber...	Response For A Class	6	[0..45)
LCOM	Chidamber...	Lack Of Cohesion Of Methods	1	
NOC	Chidamber...	Number Of Children	0	[0..2)
NOA	Lorenz-Kid...	Number Of Attributes	2	[0..4)
NOO	Lorenz-Kid...	Number Of Operations	15	
NOOM	Lorenz-Kid...	Number Of Overridden Methods	0	[0..3)
NOAM	Lorenz-Kid...	Number Of Added Methods	3	
SIZE2	Li-Henry M...	Number Of Attributes And Methods	17	
NOM	Li-Henry M...	Number Of Methods	3	[0..7)
MPC	Li-Henry M...	Message Passing Coupling	4	
DAC	Li-Henry M...	Data Abstraction Coupling	2	
ATFD	Lanza-Mari...	Access To Foreign Data	0	[0..6)
NOPA	Lanza-Mari...	Number Of Public Attributes	0	[0..3)
NOAC	Lanza-Mari...	Number Of Accessor Methods	0	[0..4)
WOC	Lanza-Mari...	Weight Of A Class	1.0	[0.5..1.0]
TCC	Bieman-Ka...	Tight Class Cohesion	1.0	[0.33..1.0]
NCSS	Chr. Cleme...	Non-Commenting Source Statements	358	[0..1000)

Figure 51. New Metrics for FileOperation Class



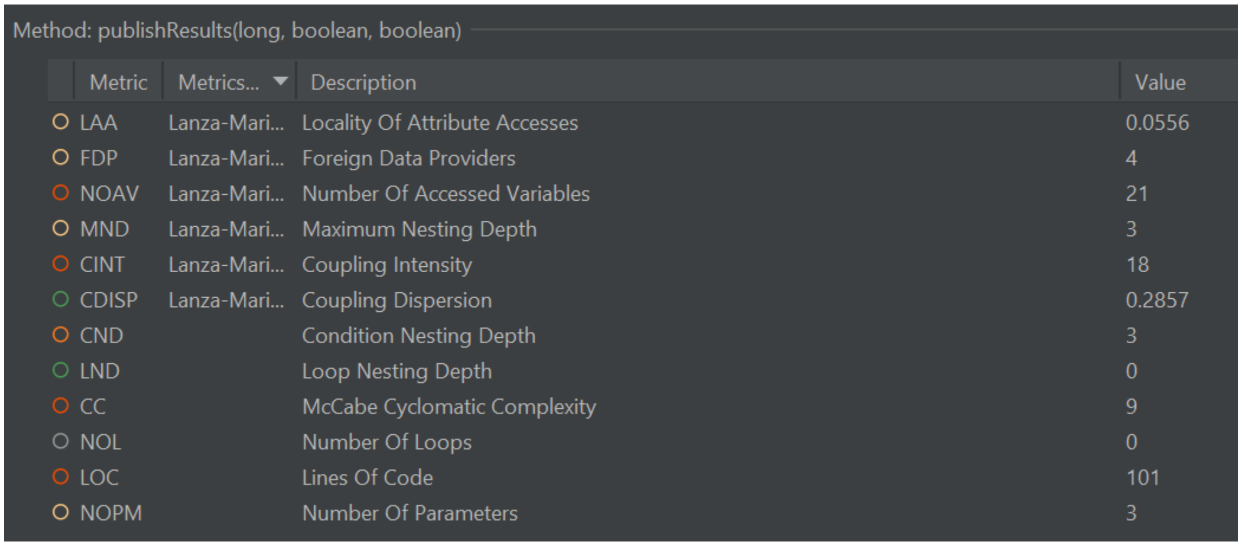
Class: FileValidator

Metric	Metric Set	Description	Value	Regular Range
WMC	Chidamber...	Weighted Methods Per Class	28	[0..12)
DIT	Chidamber...	Depth Of Inheritance Tree	1	[0..3)
CBO	Chidamber...	Coupling Between Objects	11	[0..14)
RFC	Chidamber...	Response For A Class	31	[0..45)
LCOM	Chidamber...	Lack Of Cohesion Of Methods	6	
NOC	Chidamber...	Number Of Children	0	[0..2)
NOA	Lorenz-Kid...	Number Of Attributes	9	[0..4)
NOO	Lorenz-Kid...	Number Of Operations	18	
NOOM	Lorenz-Kid...	Number Of Overridden Methods	0	[0..3)
NOAM	Lorenz-Kid...	Number Of Added Methods	6	
SIZE2	Li-Henry M...	Number Of Attributes And Methods	27	
NOM	Li-Henry M...	Number Of Methods	6	[0..7)
MPC	Li-Henry M...	Message Passing Coupling	39	
DAC	Li-Henry M...	Data Abstraction Coupling	1	
ATFD	Lanza-Mari...	Access To Foreign Data	5	[0..6)
NOPA	Lanza-Mari...	Number Of Public Attributes	0	[0..3)
NOAC	Lanza-Mari...	Number Of Accessor Methods	0	[0..4)
WOC	Lanza-Mari...	Weight Of A Class	1.0	[0.5..1.0]
TCC	Bieman-Ka...	Tight Class Cohesion	0.0	[0.33..1.0]
NCSS	Chr. Cleme...	Non-Commenting Source Statements	47	[0..1000)

Figure 52. New Metrics for FileValidator Class

4. <u>Feature Envy in *AbstractProgressiveService.java*</u>

The metrics of the refactored *publishResults* method can be seen in Figure 50.



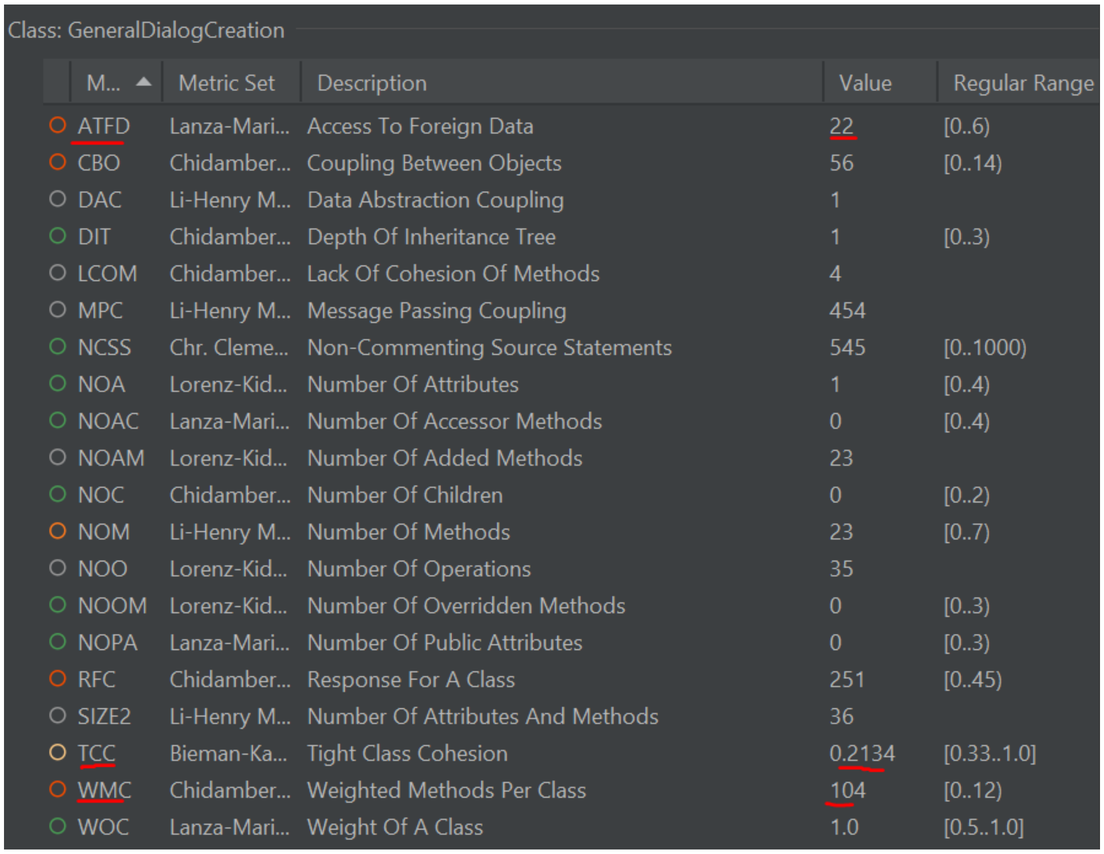
Method: publishResults(long, boolean, boolean)

Metric	Metrics...	Description	Value
LAA	Lanza-Mari...	Locality Of Attribute Accesses	0.0556
FDP	Lanza-Mari...	Foreign Data Providers	4
NOAV	Lanza-Mari...	Number Of Accessed Variables	21
MND	Lanza-Mari...	Maximum Nesting Depth	3
CINT	Lanza-Mari...	Coupling Intensity	18
CDISP	Lanza-Mari...	Coupling Dispersion	0.2857
CND		Condition Nesting Depth	3
LND		Loop Nesting Depth	0
CC		McCabe Cyclomatic Complexity	9
NOL		Number Of Loops	0
LOC		Lines Of Code	101
NOPM		Number Of Parameters	3

Figure 53. *publicResults* method's new metrics

If we compare these new metrics with its initial one (see Figure 15), we see that LAA slightly increased, with it being 0.05 initially and now, 0.0556. As for FDP, it has slightly decreased, passing from a value of 5 to 4. Finally, if we recalculate ATFD based on the NOAV and LAA metrics in Figure 51, we obtain a value of 19.832 ((1-0.0556)*21), which is lower than its original value of 21.85. Unfortunately, these newly obtained metrics are still problematic: ATFD is still superior to FEW (19.832 > [2, 5]), LAA is still significantly smaller than ⅓ (0.0556 < ⅓) and FDP is higher than FEW (4 < [2.5]). That being so, despite the modifications and the refactoring applied to the method in question, the Feature Envy code smell is still present, thus making the refactoring unsuccessful.

5. God Class in *GeneralDialogCreation.java*



Class: GeneralDialogCreation

M...	Metric Set	Description	Value	Regular Range
ATFD	Lanza-Mari...	Access To Foreign Data	22	[0..6)
CBO	Chidamber...	Coupling Between Objects	56	[0..14)
DAC	Li-Henry M...	Data Abstraction Coupling	1	
DIT	Chidamber...	Depth Of Inheritance Tree	1	[0..3)
LCOM	Chidamber...	Lack Of Cohesion Of Methods	4	
MPC	Li-Henry M...	Message Passing Coupling	454	
NCSS	Chr. Cleme...	Non-Commenting Source Statements	545	[0..1000)
NOA	Lorenz-Kid...	Number Of Attributes	1	[0..4)
NOAC	Lanza-Mari...	Number Of Accessor Methods	0	[0..4)
NOAM	Lorenz-Kid...	Number Of Added Methods	23	
NOC	Chidamber...	Number Of Children	0	[0..2)
NOM	Li-Henry M...	Number Of Methods	23	[0..7)
NOO	Lorenz-Kid...	Number Of Operations	35	
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	251	[0..45)
SIZE2	Li-Henry M...	Number Of Attributes And Methods	36	
TCC	Bieman-Ka...	Tight Class Cohesion	0.2134	[0.33..1.0]
WMC	Chidamber...	Weighted Methods Per Class	104	[0..12)
WOC	Lanza-Mari...	Weight Of A Class	1.0	[0.5..1.0]

Figure 54. *GeneralDialogCreation.java* class' new metrics

Similarly to the refactoring of the Brain Method code smell in *onBindViewHolder*, the refactoring of the God Class in GeneralDialogCreation was not exhaustive. As we were limited on time, we did not fully refactor the class. However, the metrics concerned have still lowered. If we look at Figure 52 above, we can see that ATFD went from 23 to 22, TCC went from 0.2185 to 0.2134 and WMC went from 114 to 104. These changes happened only with 3 small class extractions. If we took the time to analyze and refactor the class more in depth, the anomaly could have been dealt in a better way.

After reevaluating each metric related to the five anomalies, let’s take a look at the overall metrics of the program and see if they have changed for the better.

Table 8. New Size Metrics

	Average	Maximum	Standard Deviation	Total
LOC (MetricsReloaded)	123.500	1897	219.0149	40015
SIZE2 (MetricsTree)	118.506	1219	227.342	29982
NOM (MetricsTree)	7.352	77	9.971	1860

As we can see in table 8, LOC and NOM remain quite similar to their original values. Initially, we had obtained a LOC average of 124.7 and a NOM average of 7.406 (see Table 1), as well as a LOC maximum of 1897 and a NOM maximum of 75. The total value of LOC has also slightly decreased, passing from 40155 to 40015, whereas for NOM, its total increased, passing from 1844 to 1860. SIZE2’s values, on the other hand, have increased a lot. Initially, we obtained a SIZE2 average of 82.410, a maximum of 1219 and a total of 20520. After the refactoring, those values became 118.506, 1219 and 29982 respectively. The overall improvement of these size metrics is mainly thanks to the refactoring of the Brain Method anomalies: both refactored methods decreased in size.

Table 9: New Cohesion metrics

	Average	Maximum	Standard Deviation	Total
LCOM (MetricsTree)	1.810	18	2.109	458

When we compare the new LCOM metrics, we can see that all of them decreased. Indeed, the Average went from 1.938 to 1.81, the Maximum went from 22 to 18, the Standard Deviation went from 2.243 to 2.109 and the Total went from 620 to 458. As explained in the first question, a high cohesion is something desirable for a program. Fortunately, our LCOM values did not decrease significantly, and the values are still considered pretty high. We think that these new values were caused by our quick refactoring methods. Since the code was pretty complex and we were focused on refactoring independent code smells, we did not pay attention to every detail while extracting and modifying. These details probably impacted other parts of the program that we did not consider.

Table 10: New Complexity metrics

	Average	Maximum	Standard Deviation	Total
CC (MetricsReloaded)	2.820	82	4.588	6227
WMC (MetricsTree)	22.806	350	44.810	5770
RFC (MetricsTree)	33.814	532	55.899	8555

After refactoring the five anomalies aforementioned, the newly determined complexity metrics have not changed much compared to the metrics in Table 3. The CC average decreased from 2.830 to 2.820, the maximum decreased as well from 97 to 82 and the total increased from 6202 to 6227. As for WMC, its average went from 23.032 to 22.806, the maximum, from 524 to 532, and the total from 5735 to 5770. Finally, for RFC, the average remains quite similar, with it being 33.943 initially and 33.814 now. For its maximum, it increased slightly from 524 to 532, and the total went from 8453 to 8555 after the refactoring. In short, it is safe to say that the refactoring of the anomalies did not really affect the program's complexity.

Table 11: New Coupling metrics

	Average	Maximum	Standard Deviation	Total
CBO (MetricsTree)	10.245	114	13.482	2592
Ca (MetricsReloaded)	33.317	396	89.239	2099

The two metrics shown on Table 11 represent the overall Coupling of our code. As explained before, a low coupling is always desirable since it means a lower interdependence between components. After refactoring, we can see that CBO values have increased and that Ca values have decreased. However, the decrease of the Ca metric was more drastic than the increase of the CBO metric. For example the average for CBO increased a little bit from 10.145 to 10.245, but the one for Ca decreased pretty significantly from 109.515 to 33.317. The Total for CBO went from 2326 to 2592 whilst the one for Ca went from 7228 to 2099. Therefore, we assume that our refactoring contributed positively to the Coupling quality of our system.

d)

Table 12: Values for Quality Attributes

Quality Attribute	Value
Reusability	$-0.25 * 10.245 + 0.25 * 0.899 + 0.5 * 6.191 + 0.5 * 404 = 202.759$
Flexibility	$0.25 * 0.6933 - 0.25 * 10.245 + 0.5 * 6.08 + 0.5 * 7.013 = 4.159$
Understandability	$-0.33 * 1.186 + 0.33 * 0.6933 - 0.33 * 10.245 + 0.33 * 0.899 - 0.33 * 7.013 - 0.33 * 7.352 - 0.33 * 404 = -141.307$
Functionality	$0.12 * 0.899 + 0.22 * 7.013 + 0.22 * 6.191 + 0.22 * 404 + 0.22 * 59 = 104.873$
Extendibility	$0.5 * 1.186 - 0.5 * 10.245 + 0.5 * 0.3744 + 0.5 * 7.013 = -0.8358$
Effectiveness	$0.2 * 1.186 + 0.2 * 0.6933 + 0.2 * 6.08 + 0.2 * 0.3744 + 0.2 * 7.013 = 3.069$

*NOTE1: The calculations presented on Table 12 use the values presented on Table 13
**NOTE2: The final values presented on Table 12 are being compared to the ones on Table 5

The Reusability quality went from 205.3425 to 202.759. We explained earlier that a high value for this quality is something desirable, since it represents a system with components easier to reuse. In this case, the value dropped a little bit, but it is still highly above zero. Therefore, we conclude that the overall code is still very reusable and that our refactoring did not have a really significant impact on this quality.

Like the Reusability quality, the Flexibility is better when its value is as high as possible (above 0). In our case, the value went from 4.5827 to 4.159. This decrease is not really desirable, as the value gets even closer to zero. We think that this is due to the lack of attention to details during our refactoring process. Indeed, Flexibility is influenced by properties like Composition and Polymorphism. Our refactoring methods mostly consisted of method or class extractions, which surely did not contribute to the Polymorphism and Composition aspects. As explained in previous sections, a lot of the refactoring was non-exhaustive, meaning that a lot of details might have been left out and that better refactoring methods could have been applied at times.

Just like the two previous qualities, the Understandability quality is considered optimal the higher its value is; as the Understandability value increases, the program becomes easier to understand as well. In our case, after refactoring the five anomalies, we observe a slight increase from -143.1035 to -141.407. Even though its value is still negative, this change remains a slight improvement. We can say that the refactoring was beneficial and therefore, improved the program's Understandability quality.

Unfortunately for the Extendibility quality, this aspect of the program was worsened. With the original source code, the Extendibility was -0.6557. However, after modifying it, this value decreased to -0.8358. This means that modifying the system consists of a more difficult task, as it has more chances to cause undesirable side effects. In other words, the code has become harder to maintain after having refactored the five code smells.

The Effectiveness went from 3.109 to 3.069 which means that the value got closer to zero, and therefore decreased in effectiveness. Similarly to the Reusability quality, the calculations for this quality are based on properties like Polymorphism, Composition, Encapsulation, Inheritance and Abstraction. Since we did not have many refactoring methods that took in consideration these properties, we do not think that it is surprising that they worsened.

Table 13: Proxy metrics values for ISO 9126 Criteria Metrics

Design Property	Metric Value
Design size	404
Hierarchies	17+42 = 59
Abstraction	70/59 = 1.186
Encapsulation	69.33% = 0.6933
Coupling	10.245
Cohesion	1 - (1.810/18) = 0.899
Composition	6.08
Inheritance	37.44% = 0.3744
Polymorphism	7.406 * 0.9469 = 7.013
Messaging	6.191
Complexity	7.352

References

Aivosto. (2013). *Chidamber & Kemerer object-oriented metrics suite.* http://people.scs.carleton.ca/~jeanpier/sharedF14/T1/extra%20stuff/about%20metrics/Chidamber%20&%20Kemerer%20object-oriented%20metrics%20suite.pdf

Hamilton, T (2021). *Mccabe's Cyclomatic Complexity: Calculate with Flow Graph (Example).* Guru99. https://www.guru99.com/cyclomatic-complexity.html

Lanza, M. & Marinescu R. (2011). *Object-Oriented Metrics in Practice : Using Software Metrics to characterize, evaluate, and improve the Design of Object-Oriented Systems*. Berlin, Germany. Springer.