



**POLYTECHNIQUE  
MONTREAL**

UNIVERSITÉ  
D'INGÉNIERIE

Département de génie informatique et génie logiciel

LOG8430 : Architecture logicielle et conception avancée

TP2 : Qualité de la conception, anomalies et *refactoring*

Victor Kim: 1954607

Yasser Benmansour 2225733

Mohammed Ridha Ghoul: 2224721

Anis Youcef Dilmi: 1895231

Présenté à Nour Abiad

Le dimanche 19 mars 2023

<b>Introduction.....</b>	<b>3</b>
<b>Mesurer la qualité de conception d’OmniNotes .....</b>	<b>3</b>
<i>Métriques .....</i>	<i>3</i>
Présentation des métriques .....	3
Métriques de taille .....	4
Métrique de complexité .....	6
Métriques de complexité structurelle .....	7
Métriques d'héritage .....	9
Métriques de couplage.....	10
Métriques de cohésion.....	11
<i>Qualité globale du système .....</i>	<i>12</i>
<i>Métriques pour les critères ISO 9126 .....</i>	<i>13</i>
<i>Description de la qualité du système en fonction des critères métriques .....</i>	<i>14</i>
Réutilisabilité .....	14
Flexibilité.....	14
Compréhensibilité .....	14
Fonctionnalité .....	14
Extensibilité .....	15
Efficacité.....	15
<i>Corrélation des valeurs des métriques ISO avec les valeurs des métriques individuelles .....</i>	<i>15</i>
Réutilisabilité .....	15
Flexibilité.....	17
Fonctionnalité .....	19
<b>Identification des anomalies.....</b>	<b>20</b>
<i>Seuils des métriques .....</i>	<i>20</i>
<i>Présentation et identification des anomalies .....</i>	<i>22</i>
Long Method .....	22
Large Class .....	24
Shotgun Surgery .....	25
Long Parameter List .....	26
Switch statements .....	27
<b>Refactoring pour corriger les anomalies .....</b>	<b>27</b>
<i>Long method.....</i>	<i>27</i>
<i>Large class .....</i>	<i>32</i>
<i>Shotgun surgery.....</i>	<i>39</i>
<i>Long parameter list .....</i>	<i>39</i>
<i>Switch statement.....</i>	<i>46</i>
<i>Métriques pour les critères de qualité après le refactoring .....</i>	<i>48</i>
<b>Références : .....</b>	<b>50</b>

# Introduction

Ce rapport analyse la qualité de la conception en profondeur, identifie des anomalies à l'aide de métriques et montre la correction de celles-ci dans le système d'OmniNotes en se basant sur les meilleures pratiques d'ingénierie logicielle. Pour ce faire, il sera présenté dans un premier temps la qualité de conception incluant la présentation des métriques, la qualité globale du système, les métriques pour les critères ISO 9126, la description de la qualité du système en fonction des critères métriques et la corrélation de ses valeurs. Ensuite, il sera abordé l'identification d'anomalies dans OmniNotes, incluant les seuils des métriques. Enfin, la correction des différentes anomalies sera détaillée.

## Mesurer la qualité de conception d'OmniNotes

### Métriques

#### Présentation des métriques

Les métriques présentées dans cette section sont obtenues à travers un ou plusieurs logiciels suivants:

1. Le logiciel SciTools Understand™
2. Le plugiciel Android Studio MetricsReloaded
3. Le plugiciel Android Studio Metrics Tree

Il existe plusieurs catégories de métriques destinées à évaluer divers aspects du système informatique, tels que la taille du code, la complexité de l'interface, la structure, l'héritage, le couplage, la cohésion et la documentation. Ces métriques peuvent être évaluées à différents niveaux du système, notamment au niveau du projet, du package, de la classe ou de la méthode.

## Métriques de taille

En premier lieu, les métriques de tailles se concentrent sur la taille du système. Pour déterminer la taille de système on a utilisé les métriques suivantes:

- Le nombre total de lignes de code (TLOC) correspond au nombre total de lignes de code dans le projet, y compris les commentaires et les lignes vides.
- Les lignes de code sans commentaire (NCLOC) excluent les commentaires et les lignes vides pour mesurer les instructions de code réelles.
- Les lignes de code de méthode (MLOC) mesurent le nombre de lignes de code dans une seule méthode ou fonction.
- Les lignes de code de classe (CLOC) mesurent le nombre de lignes de code dans une seule classe.

Ces mesures sont couramment utilisées pour évaluer la taille, la complexité et la maintenabilité d'un logiciel.

Métrique	Plugiciel	Niveau de la métrique	Moyenne	Minimum	Maximum	Total
Total Lines of Code (TLOC)	Metrics Reloaded	Projet	N/A	N/A	N/A	53768
Non-Comment Lines of Code (NCLOC)	Metrics Reloaded	Projet	N/A	N/A	N/A	43970
Method Lines of Code (MLOC)	Metrics Reloaded	Méthodes	11.06	2	394	14393
Class Lines of Code (CLOC)	Metrics Reloaded	Classes	76.41	1	1748	15587

**Tableau 1** : Tableau des métriques de taille

On observe que la métrique CLOC des classes de OmniNotes a une moyenne de 76.41 et un maximum de 1748. Cela veut dire qu'il existe une classe avec 1748 lignes de code. Une telle longueur est problématique, car la compréhensibilité ainsi que la maintenabilité d'une telle classe est diminuée. Une solution à cette problématique est de refactoriser une telle classe en la divisant en des sous-classes dont chacune s'occupe d'une responsabilité unique.

## Métrique de complexité

La métrique SIZE2 mesure le nombre total d'attributs et de méthodes dans une classe. Elle est souvent utilisée pour évaluer la complexité d'une classe et sa maintenabilité. La métrique NOM mesure spécifiquement le nombre de méthodes définies dans une classe.

Une valeur élevée pour SIZE2 peut indiquer qu'une classe en fait trop et qu'elle devrait être remaniée en classes plus petites et plus ciblées. De même, une valeur élevée de NOM peut indiquer qu'une classe devient trop complexe et qu'elle devrait être divisée en composants plus petits et plus faciles à gérer.

Métrique	Plugiciel	Niveau de la métrique	Moyenne	Minimum	Maximum	Total
Number of Local Methods (NOM)	Metrics Tree	Projet	6.96	0	98	1099
Number of Attributes and Methods (SIZE2)	Metrics Tree	Projet	167.53	12	1239	26471

**Tableau 2:** Tableau des métriques de complexité

En calculant la métrique SIZE2 pour le projet OmniNotes, nous avons obtenu une moyenne de 167.53. En d'autres mots, il existe en moyenne 168 attributs et méthodes dans chaque classe, de façon générale, cela peut indiquer un manque de cohésion dans les classes. De plus, le maximum est de 1239 attributs et méthodes pour une classe, ceci indique un manque de cohésion dans une telle classe. Il faut donc la revoir et la refactoriser en la séparant en sous-classes qui exercent une responsabilité unique. En général, lorsque cette métrique est élevée, on peut aussi observer une diminution de la compréhensibilité, de la possibilité de modification et de la testabilité.

Pour la métrique NOM des classes de OmniNotes, on remarque qu'une classe a en moyenne 6.95 méthodes et la plus grosse classe possède 98 méthodes. Il existe une grande différence entre la valeur moyenne et la valeur maximale. En règle générale, une conception efficace de système réduit le nombre de méthodes dans une classe afin de respecter le principe de responsabilité unique (SRP). Par conséquent, il est préoccupant de constater des classes qui dépassent les 98 méthodes.

## Métriques de complexité structurelle

Comme abordé dans le cours, les métriques de complexité évaluent diverses caractéristiques du code pour fournir une indication générale de sa qualité. Les métriques de complexité se concentrent notamment sur les méthodes, qui déterminent largement la complexité du code. Une complexité élevée peut entraîner des problèmes lors du débogage et des tests, ainsi qu'une lecture et une compréhension plus difficiles pour les développeurs, rendant le code plus difficile à maintenir. Pour notre analyse de la complexité structurelle, nous avons utilisé des métriques telles que la complexité cyclomatique (CC), le nombre pondéré de méthodes (WMC) et la réponse pour une classe (RFC). Il est à noter que CC a été extraite au niveau des méthodes, tandis que WMC et RFC au niveau des classes.

Métrique	Plugiciel	Niveau de la métrique	Moyenne	Minimum	Maximum	Total
McCabe Cyclomatic Complexity (CC)	Metrics Reloaded	Méthodes	2.33	0	43	2573
Response for Class (RFC)	Metrics Tree	Classes	31.3	0	498	4946
Weighted Methods Count (WMC)	Metrics Tree	Classes	16.29	0	332	2575

**Tableau 3:** Tableau des métriques de complexité structurelle

En observant les résultats des métriques de la complexité cyclomatique, on remarque que les valeurs maximales sont élevées par rapport aux moyennes. Cette métrique permet de compter le nombre de chemins indépendants dans une méthode. De plus, on peut constater que la moyenne du nombre de méthodes pour la métrique CC est très faible (2,33), ce qui est généralement un signe de bonne qualité de code lorsque la moyenne est inférieure à 10 [1]. Cependant, dans notre cas, la valeur maximale pour CC est de 43, ce qui est très élevé par rapport à la moyenne. Pour remédier à cela, il est suggéré de diviser la méthode associée à cette grande valeur (43), en plusieurs méthodes plus petites.

La métrique WMC évalue la complexité de chaque classe en additionnant de manière pondérée la complexité de toutes les méthodes de cette classe, fournissant ainsi une estimation approximative de la complexité de la classe. La valeur normale pour cette métrique devrait être comprise entre 12 et 34 [2]. Dans le cas d'Omni-Notes, la moyenne de WMC est de 16.29, cela indique que les classes du projet ont un faible niveau de complexité. Cependant, la valeur maximale de cette métrique est de 332, cela suggère qu'au moins une classe contient une ou plusieurs méthodes complexes et qu'une refactorisation serait nécessaire pour améliorer la lisibilité et la maintenabilité du code.

La dernière métrique de complexité structurelle à l'étude est la Response for class (RFC), celle-ci mesure le nombre de méthodes publiques d'une classe ainsi que le nombre de méthodes appelées par celles-ci. Une valeur de seuil généralement acceptée pour cette métrique est de 44. Tout ce qui est supérieur à ce seuil peut être considéré comme complexe et plus difficile à comprendre, à tester et à déboguer [3]. Dans l'application OmniNotes, la valeur moyenne pour la RFC est de 31.3, ce qui est une bonne indication. Cependant, la valeur maximale est de 498, ce qui est bien supérieur au seuil de 44. Cela suggère qu'une refactorisation de la classe en question serait nécessaire pour améliorer la qualité du code.



## Métriques d'héritage

La profondeur de l'arbre d'héritage (DIT) est une métrique de la complexité d'une hiérarchie de classes dans la programmation orientée objet. Elle mesure la distance entre la classe la plus haute dans la hiérarchie et la classe la plus basse. Plus la DIT est élevée, plus la hiérarchie est complexe et potentiellement difficile à maintenir.

Le nombre d'enfants (NOC) est une autre métrique qui mesure le nombre direct de sous-classes qui héritent d'une classe parente dans une hiérarchie. Plus le NOC est élevé, plus une classe parente a de responsabilités à gérer et plus elle peut être difficile à maintenir.

Métrique	Plugiciel	Niveau de la métrique	Moyenne	Minimum	Maximum	Total
Depth of Inheritance Tree (DIT)	Metrics Tree	Classes	2.46	1	10	388
Number of Children (NOC)	Metrics Tree	Classes	0.12	0	6	19

**Tableau 4:** Tableau des métriques d'héritage

D'après le tableau ci-dessus on remarque que

- Dans la hiérarchie de classes, il y a une distance (DIT) moyenne de trois classes entre une classe dérivée et sa classe de base.
- Une métrique NOC moyenne de 0,121 suggère qu'il y a moins d'une classe enfant directe pour chaque classe de base dans la hiérarchie de classes.

## Métriques de couplage

Métrique	Plugiciel	Niveau de la métrique	Moyenne	Minimum	Maximum	Total
Afferent Coupling (Ca)	Metrics Reloaded	Projet	109.64	0	1046	3070
Coupling Between Objects (CBO)	Metrics Tree	Classes	8.89	0	77	1404

**Tableau 5 :** Tableau des Métriques de Couplage

La métrique CBO mesure le nombre de classes qui sont liées les unes aux autres, soit par l'appel de méthodes, soit par l'accès à des attributs. Si la valeur de CBO est comprise entre 1 et 4, cela indique un couplage [4]. Au-delà de 4, le niveau de couplage est considéré comme trop élevé. Dans le cas du projet OmniNotes, la moyenne de CBO étant d'environ 8,89, cela suggère un couplage problématique entre les classes.

La métrique Ca mesure le niveau de couplage entre les paquets externes qui utilisent les méthodes d'un paquet spécifique. Pour un projet composé de 101 à 1000 classes, une valeur de Ca comprise entre 2 et 20 est considérée comme normale [5]. Une valeur supérieure à 20 indique un couplage élevé entre les classes. D'après les résultats du tableau ci-dessus, nous constatons que les paquets d'Omni Notes présentent un niveau de couplage élevé, avec une moyenne de Ca de 109,64. Cela suggère que l'application OmniNotes est difficile à modifier. De plus, la valeur maximale de Ca est de 1064, ce qui est nettement supérieur à la moyenne. Cela signifie que des modifications apportées au paquet présentant cette valeur maximale auront un impact important sur de nombreux autres paquets, étant donné leur interdépendance.

## Métriques de cohésion

Les mesures de cohésion évaluent le degré de corrélation entre les éléments d'un module. En d'autres termes, ces mesures évaluent la connexion entre les méthodes et les attributs au sein d'une même classe en examinant combien de fois une méthode utilise un attribut d'une classe. Une classe qui présente une forte cohésion est responsable d'une unique fonctionnalité et ne nécessite qu'une seule modification. Cela est souhaitable car une telle classe est plus compréhensible, réutilisable et modifiable. Bien qu'il existe plusieurs manières d'évaluer la cohésion d'un système, la métrique couramment utilisée est la mesure de manque de cohésion entre les méthodes d'une classe (LCOM). Les résultats de l'évaluation de LCOM au niveau des classes dans le système OmniNotes sont présentés dans le tableau ci-dessous.

Métrique	Plugiciel	Niveau de la métrique	Moyenne	Minimum	Maximum	Total
Lack Of Cohesion of Methods (LCOM)	Metrics Reloaded	Classes	2.09	0	15	331

**Tableau 6:** Tableau des Métriques de cohésion

Pour la valeur de LCOM, le plugiciel Metrics Reloaded a permis d'obtenir une moyenne de 2,09 et un maximum de 15. En général, une valeur de LCOM proche de zéro est préférable, car cela indique que la classe est parfaitement cohésive selon la définition de Henderson-Sellers. En revanche, plus la valeur LCOM se rapproche ou dépasse 1, plus cela indique un manque de cohésion au sein de la classe. Dans notre cas, la valeur moyenne de LCOM dépasse 1, ce qui suggère un manque de cohésion dans les classes de Omni-Notes. Cette conclusion est cohérente avec celle tirée de l'analyse des métriques de taille.

## Qualité globale du système

En examinant les métriques de taille, nous observons qu'elles sont élevées, ce qui implique que les classes sont trop longues dans OmniNotes. Cette situation a plusieurs conséquences indésirables, telles que la diminution de la compréhensibilité, de l'analyse et de la modifiabilité des classes. De plus, cette mauvaise implémentation peut être une indication qu'il y a violation du principe responsabilité unique (SRP) ou un indice des anti-patterns Swiss Army Knife et modularisation insuffisante.

Nous avons observé une faible cohésion dans les classes de l'application. Cela indique que les méthodes n'utilisent pas les autres membres et méthodes de la même classe de façon optimale. En général, ceci peut être un signe de modularisation brisée ou de violation du SRP. Cette mauvaise conception du code source diminue sa compréhensibilité, possibilité de modification, testabilité et remplaçabilité.

En analysant la complexité du projet, on déduit que certaines classes ont une complexité élevée. Cette situation implique que le code de ces classes peut être plus difficile à tester et à déboguer, ce qui peut rendre la maintenance et l'évolution du code plus compliquées pour les développeurs. De plus, une complexité élevée peut rendre la lecture et la compréhension du code plus ardue.

Enfin, nous constatons que le couplage dans le code est élevé, ce qui implique qu'il y a de nombreuses interdépendances entre les différentes classes. Cette situation peut rendre certaines parties du code difficiles à réutiliser et à maintenir et peut également entraîner une violation de l'encapsulation. Un haut couplage est une des causes de l'antipatron Stovepipe System, c'est aussi une conséquence des anti-patterns suivants : modularisation cycliquement dépendante, hiérarchie cyclique et modularisation brisée.

## Métriques pour les critères ISO 9126

Qualité des attributs	Valeurs
Réutilisabilité	$-0.25 * \text{Coupling} + 0.25 * \text{Cohesion} + 0.5 * \text{Messaging} + 0.5 * \text{Design Size} = 4.7494$
Flexibilité	$0.25 * \text{Encapsulation} - 0.25 * \text{Coupling} + 0.5 * \text{Composition} + 0.5 * \text{Polymorphism} = 4.853$
Compréhensibilité	$-0.33 * \text{Abstraction} + 0.33 * \text{Encapsulation} - 0.33 * \text{Coupling} + 0.33 * \text{Cohesion} - 0.33 * \text{Polymorphism} - 0.33 * \text{Complexity} - 0.33 * \text{Design Size} = -8.2682$
Fonctionnalité	$0.12 * \text{Cohesion} + 0.22 * \text{Polymorphism} + 0.22 * \text{Messaging} + 0.22 * \text{Design Size} + 0.22 * \text{Hierarchies} = 4.9266$
Extensibilité	$0.5 * \text{Abstraction} - 0.5 * \text{Coupling} + 0.5 * \text{Inheritance} + 0.5 * \text{Polymorphism} = 2.4873$
Efficacité	$0.2 * \text{Abstraction} + 0.2 * \text{Encapsulation} + 0.2 * \text{Composition} + 0.2 * \text{Inheritance} + 0.2 * \text{Polymorphism} = 3.0152$

**Tableau 7:** des métriques pour les critères ISO 9126 (QMOOD)

## Description de la qualité du système en fonction des critères métriques

### Réutilisabilité

Cette propriété est un avantage de l'utilisation de la programmation orientée objet. Mesure la compréhension générale du code afin que les développeurs puissent réutiliser le code dans d'autres systèmes. Une valeur supérieure à zéro, comme dans le cas présent, signifie que le code contient plusieurs composants qui peuvent être réutilisés dans d'autres projets. Il est clair que le code doit être de haute qualité, compréhensible et flexible afin qu'il puisse être utilisé par d'autres développeurs.

### Flexibilité

La flexibilité est une propriété qui décrit la facilité avec laquelle il est possible de modifier le code. Dans ce cas, la valeur de flexibilité est de 4,853. Par conséquent, bien qu'il soit possible de modifier le code de l'application Omni-Notes, des précautions doivent être prises lors des modifications afin de ne pas interrompre les fonctionnalités existantes.

### Compréhensibilité

Cette qualité mesure la capacité du développeur à identifier les concepts logiques et leur applicabilité dans le code. Cette qualité peut être considérée comme une fonction partielle de l'utilisabilité et de la réutilisabilité. En fait, les développeurs ont besoin de comprendre les concepts logiques avant de les réutiliser dans un autre contexte. Dans notre cas, le score de compréhension est négatif. Cela signifie que la logique de votre code est généralement difficile à comprendre. Cela signifie également que le code est difficile à tester et à déboguer. De plus, ajouter de nouvelles fonctionnalités peut être difficile.

### Fonctionnalité

Cette qualité mesure à quel point le code logiciel implémente ses fonctionnalités. Dans notre cas, la valeur de la caractéristique est de 4,92, ce qui est relativement élevé. Cela

montre qu'en général les classes implémentent une interface correspondant à leur rôle dans l'application mais que d'autres classes ont des responsabilités multiples.

## Extensibilité

Cette qualité décrit dans quelle mesure un système peut être modifié sans provoquer d'effets indésirables. Dans le cas de Omni-Notes, le score d'extensibilité est légèrement supérieur à zéro, ce qui indique qu'il est généralement difficile de modifier le code de l'application pour ajouter de nouvelles fonctionnalités sans modifier d'autres composants du système.

## Efficacité

Cette qualité décrit la performance du système en fonction de son comportement attendu. Dans ce cas, le rendement est de 2,48. La valeur obtenue pour cette qualité signifie que le logiciel est adapté au but de l'application.

## Corrélation des valeurs des métriques ISO avec les valeurs des métriques individuelles

### Réutilisabilité

Comme indiqué dans la question précédente, la réutilisabilité est très souhaitable. Omni-Notes possède des composants qui peuvent être utilisés tels quel par d'autres logiciels. Pour calculer la valeur de réutilisabilité, il faut tenir compte du couplage, de la cohésion, du nombre de méthodes publiques et du nombre de classes. Dans les calculs de qualité, les poids de couplage et de cohésion sont respectivement de -25 % et 25 %. Nous pouvons donc voir qu'ils sont inversement proportionnels. En revanche, le nombre de classes et le nombre de méthodes publiques ont chacun un poids de 50%. Par conséquent, pour qu'une classe soit réutilisable, elle doit être cohérente sans trop de couplage. Cela signifie qu'une classe ne doit pas dépendre de plusieurs autres classes du le système et que les attributs de la classe seront utilisés par ses propres méthodes. On retrouve ces propriétés dans la classe PushBulletMessage.

Pour les propriétés de couplage, l'analyse s'est basée sur la valeur de la métrique Coupling Between Objects (CBO). Dans cette classe, la valeur CBO est 0, comme le montre la **Figure 1** ci-dessous. Le score CBO moyen du projet est de 8,88 et le maximum est de 77, donc cette classe a un faible couplage. C'est-à-dire que PushBulletMessage n'accède pas à d'autres classes et elle n'est pas accédée par d'autres classes. Cette interaction nulle avec d'autres classes est bénéfique car elle rend la classe plus simple et facile à comprendre et à réutiliser.

Le calcul de la propriété de cohésion est basé sur la valeur de la métrique Lack of cohesion of Methods (LCOM). Les valeurs de résistance cohésive sont calculées à l'aide de la formule suivante :  $1 - (LCOM / MAX(LCOM))$ .

La classe PushBulletMessage a une valeur LCOM de 0, comme illustré à la **Figure 1**. Ceci est inférieur à la valeur LCOM moyenne du projet de 2,09 (tableau pour la question 1). Cela signifie que cette classe n'a pas beaucoup d'attributs et de méthodes non connectés entre eux. Par conséquent, la cohésion de cette classe est élevée.

Le nombre de classes est une mesure de niveau projet car il représente le nombre total de classes dans l'ensemble du système. Il n'est donc pas logique de calculer sa valeur dans la classe PushBulletMessage.

Les propriétés de messagerie peuvent être représentées par le nombre de méthodes publiques dans une classe. Dans cet exemple, la classe PushBulletMessage a une méthode publique comme illustré à la **Figure 1**. On peut donc dire que cette classe respecte le principe de responsabilité unique. Cela améliore la compréhensibilité, la maintenabilité et la réutilisabilité.

class	CBO	DIT	LCOM	NOC	RFC	WMC
it.feio.android.omninotes.utils.ConstantsBase					0	
it.feio.android.omninotes.utils.date.SublimePickerFragment.Callback					2	
it.feio.android.omninotes.models.PushBulletMessage	0	1	0	0	1	1
it.feio.android.omninotes.models.holders.ImageAndTextItem	0	1	2	0	6	5
it.feio.android.omninotes.models.misc.PlayStoreMetadataFetcherResult	0	1	6	0	12	12
it.feio.android.omninotes.models.views.VerticalSeekBar	0	5	3	0	32	19

**Figure 1:** Les métriques LCOM et CBO de la classe PushBulletMessage



Class: PushBulletMessage

Metric	Description	Value
CHVL	Halstead Volume	23.2647
CHD	Halstead Difficulty	2.0
CHL	Halstead Length	9
CHEF	Halstead Effort	46.5293
CHVC	Halstead Vocabulary	6
CHER	Halstead Errors	0.0043
WMC	Weighted Methods Per Class	1
DIT	Depth Of Inheritance Tree	1
CBO	Coupling Between Objects	0
RFC	Response For A Class	1
LCOM	Lack Of Cohesion Of Methods	0
NOC	Number Of Children	0
NOA	Number Of Attributes	1
NOO	Number Of Operations	13
NOOM	Number Of Overridden Methods	0
NOAM	Number Of Added Methods	0
SIZE2	Number Of Attributes And Methods	14
<b>NOM</b>	<b>Number Of Methods</b>	<b>1</b>
MPC	Message Passing Coupling	0
DAC	Data Abstraction Coupling	1
ATFD	Access To Foreign Data	0
NOPA	Number Of Public Attributes	1
NOAC	Number Of Accessor Methods	0
WOC	Weight Of A Class	0.0
TCC	Tight Class Cohesion	0.0
NCSS	Non-Commenting Source Statements	1
CMI	Maintainability Index	80.0573

**Figure 2:** Nombre de méthodes publiques de la classe PushBulletMessage

## Flexibilité

Cette norme de qualité est définie par des propriétés telles que le couplage, l'encapsulation, la composition et le polymorphisme. Les mesures qui ont le plus d'impact sur la flexibilité sont la composition et le polymorphisme. Autrement dit, la composition et le polymorphisme ont chacun un poids de  $(0,5 / 1,5) * 100 = 33,33 \%$ . De plus, les métriques les moins influentes sont le couplage et l'encapsulation, chacune avec un poids de  $(0,25 / 1,5) * 100 = 16,66 \%$ .

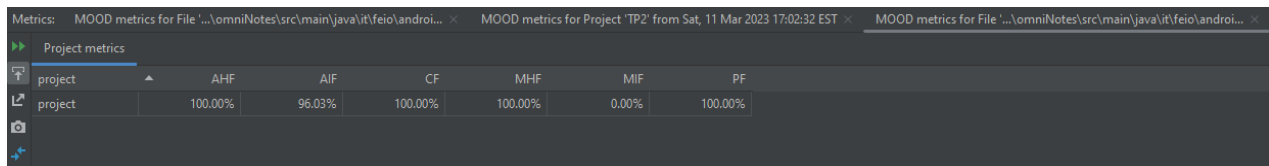
L'analyse de l'application montre que la valeur de flexibilité est positive. Pour cette raison, nous allons examiner la classe `DataBackupIntentService.java`.

Premièrement, l'encapsulation est définie par une métrique AHF qui indique le pourcentage d'attributs de classe qui sont encapsulés. En regardant l'analyse effectuée sur la métrique MOOD de la **Figure 3**, le score AHF est de 100 %. Cette valeur est supérieure à la moyenne globale du projet (63.6%). On en déduit que cette classe est bien encapsulée et possède un bon niveau flexibilité.

Le couplage est alors défini par la métrique CBO. En effet, cette métrique définit le degré de couplage entre les classes. En regardant la **Figure 4**, nous pouvons voir que la valeur CBO est de 21, ce qui est supérieur à la moyenne du projet (8.88). On peut affirmer que ce couplage élevé rend la classe moins flexible.

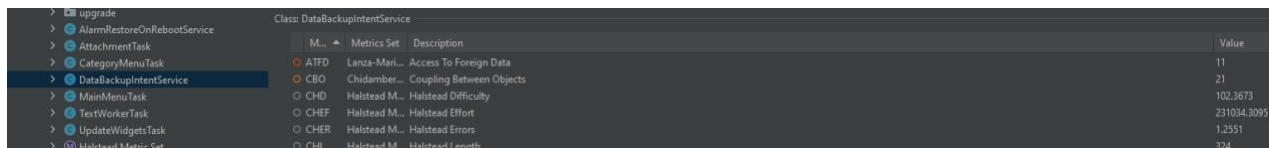
La composition est définie par le nombre de dépendances. En effet, cette métrique permet de définir combien une classe a d'agrégations. La valeur de composition représentée par la colonne Dcy de l'analyse MetricsReloaded dans la **Figure 5** renvoie un score de 20, supérieur à la moyenne du projet de 5,5. Ce haut niveau d'agrégation montre que la classe est peu flexible.

Enfin, le polymorphisme est défini par la métrique  $NOM * PF$ . C'est le nombre de méthodes multiplié par le facteur de polymorphisme. Cette métrique définit le nombre de méthodes polymorphes en multipliant le nombre de méthodes NOM par le facteur de polymorphisme PF. La métrique NOM a une valeur de 14, ce qui est supérieur à la moyenne du projet de 6.95 observée dans le **Tableau 2**. La valeur PF de la **Figure 3** est de 100%, ce qui est inférieur à la valeur PF du projet (303.03 %). En considérant ces deux valeurs, le nombre de méthodes polymorphes est grand et la flexibilité de la classe est haute.



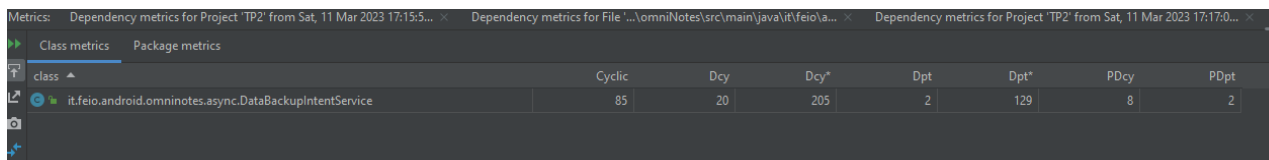
Project metrics	AHF	AIF	CF	MHF	MIF	PF
project	100.00%	96.03%	100.00%	100.00%	0.00%	100.00%

**Figure 3:** Les métriques MOOD de la classe DataBackupIntentService.java



M...	Metrics Set	Description	Value
ATFD	Lanza-Mari...	Access To Foreign Data	11
CBO	Chidamber...	Coupling Between Objects	21
CHD	Halstead M...	Halstead Difficulty	102.3673
CHF	Halstead M...	Halstead Effort	231034.3095
CHER	Halstead M...	Halstead Errors	1.2551
CHL	Halstead M...	Halstead Length	324

**Figure 4:** La métrique CBO de la classe DataBackupIntentService.java



Class metrics	Cyclic	Dcy	Dcy*	Dpt	Dpt*	PDcy	PDpt
it.feio.android.omninetes.async.DataBackupIntentService	85	20	205	2	129	8	2

**Figure 5:** La métrique Dcy de la classe DataBackupIntentService.java

Metric	Value
AlarmRestoreOnRebootService	120
AttachmentTask	0
CategoryMenuTask	13
DataBackupIntentService	98
MainMenuTask	0
TextWorkerTask	329
UpdateWidgetsTask	0
Halstead Metric Set	0
Robert C. Martin Metrics Set	65
Statistics	334
Maintainability Index	0.2692
LOC	24
NOM	98
NOA	10
NOAC	0
NOAM	0
NOC	0
NOM	98
NOD	0
NOOM	0
NOPA	0
RFC	0
SIZE2	0
TCC	0
WMC	0
WMC	0

**Figure 6:** La métrique NOM de la classe DataBackupIntentService.java

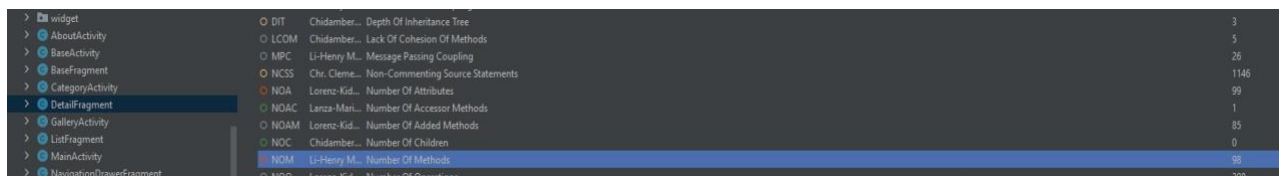
## Fonctionnalité

Cette qualité est calculée en fonction du couplage, du polymorphisme, des méthodes publiques de classe, de la taille de conception et du niveau de hiérarchie. La cohésion est pondérée à 12% et les autres éléments sont pondérés à 22%. Pour analyser la fonctionnalité, il est intéressant d'examiner la classe DetailFragment.

Le facteur de cohésion est défini à partir de la métrique LCOM. Plus spécifiquement, ce facteur est calculé à partir de l'équation  $1 - (LCOM / MAX(LCOM))$  qui représente la cohésion entre les méthodes de la classe. Comme il est montré dans les résultats données par MetricsTree dans la **Figure 6**, la valeur de LCOM associée à cette classe est 5, ce qui veut dire que cette classe ne manque pas de cohésion entre ses méthodes et ses attributs. En utilisant la formule mentionnée précédemment, on obtient une cohésion de 0,66. Par conséquent, la cohésion de cette classe est élevée et cela contribue à sa fonctionnalité.

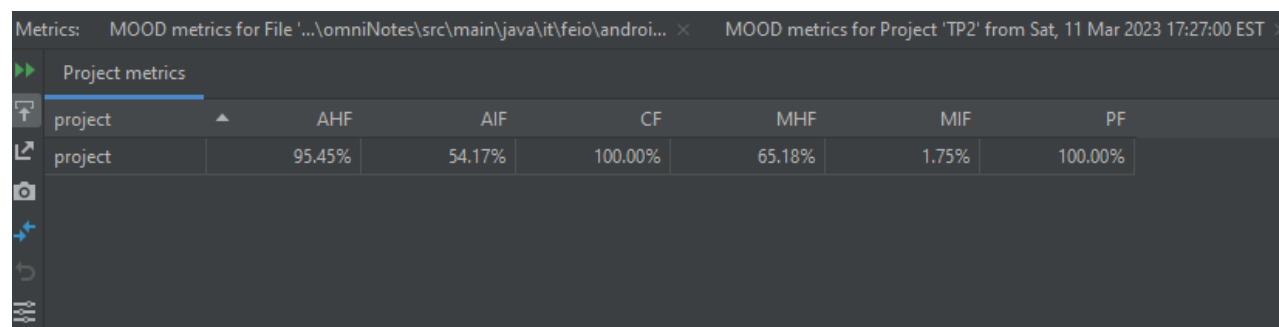
Le facteur de polymorphisme est défini par la métrique de nombre de méthodes (NOM) \* Facteur de polymorphisme (PF). La métrique définit le nombre de méthodes polymorphiques dans une classe en multipliant le facteur de polymorphisme d'une classe par son nombre de méthodes. La valeur de NOM qui a été calculée avec MetricsTree et qui est présenté dans la **Figure 6** est 98, ce qui est égal au maximum des NOM montré par le **Tableau 2**. En ce qui concerne la valeur du facteur de polymorphisme, elle a été calculée pour la classe en utilisant MetricReloaded. Cette dernière est 100%, cela signifie que cette classe est polymorphique. Considérons les deux valeurs précédente, celles-ci démontrent que le polymorphisme de la classe est élevé. Ces valeurs de polymorphisme indiquent que la classe est fonctionnelle.

La taille d'architecture est représentée par le nombre de classes du produit Cp. Dans ce cas, nous analysons une seule classe, donc cela n'a pas de sens de calculer le Cp pour cette classe. Le niveau de hiérarchie est défini par le nombre de classes abstraites additionné par le nombre d'interfaces dans le système. Dans notre cas, cela n'a pas de sens de calculer le nombre de classes abstraites ni le nombre d'interfaces pour une seule classe.



Metric	Value
DIT	3
LCOM	5
MPC	26
NCSS	1146
NOA	99
NOAC	1
NOAM	85
NOC	0
NOM	98
NOP	300

**Figure 6:** LCOM et NOM métriques extraits avec MetricsTree



Project	AHF	AIF	CF	MHF	MIF	PF
project	95.45%	54.17%	100.00%	65.18%	1.75%	100.00%

**Figure 7:** Facteur de polymorphisme de la classe DetailFragment avec MetricsReloaded.

## Identification des anomalies

### Seuils des métriques

Il est possible de détecter certains anti-patterns de code avec une stratégie de détection impliquant des métriques et des conditions et des seuils. Pour ce faire, nous pouvons utiliser des seuils à comparer aux métriques et combiner les conditions pertinentes pour détecter les anti-patterns. Les différents seuils des métriques sont définis dans le livre Object-Oriented Metrics in Practice. Les métriques utilisées dans la description de

chaque anomalie sont déterminées à l'aide des outils MetricsTree et MetricReloaded. Les quatre prochains tableaux présentent les seuils généraux, statistiques et sémantiques. Il n'est pas possible de calculer les seuils statistiques avec la formule qui utilise l'écart type et la moyenne, pour cela nous avons utilisé les seuils statistiques présentés dans le livre Object-Oriented Metrics in Practice. Ces seuils sont calculés à partir de 45 projets Java et le projet Omni-Notes utilise ce langage.

Numeric Value	Semantic Label
0	NONE
1	ONE/SHALLOW
2 – 5	TWO, THREE/FEW/ SEVERAL
7 – 8	Short Memory Capacity

Tableau 8 : Seuils généraux. [9]

	Java				C++			
Metric	Low	Ave- rage	High	Very High	Low	Ave- rage	High	Very High
WMC	5	14	31	47	4	23	72	108
AMW	1.1	2.0	3.1	4.7	1.0	2.5	4.8	7.0
LOC/Class	28	70	130	195	20	90	240	360
NOM/Class	4	7	10	15	4	9	15	23

Tableau 9 : Seuils statistiques [9]

	Java				C++			
Metric	Low	Ave- rage	High	Very High	Low	Ave- rage	High	Very High
CYCLO/Line of Code	0.16	0.20	0.24	0.36	0.20	0.25	0.30	0.45
LOC/Method	7	10	13	19.5	5	10	16	24
NOM/Class	4	7	10	15	4	9	15	22.5

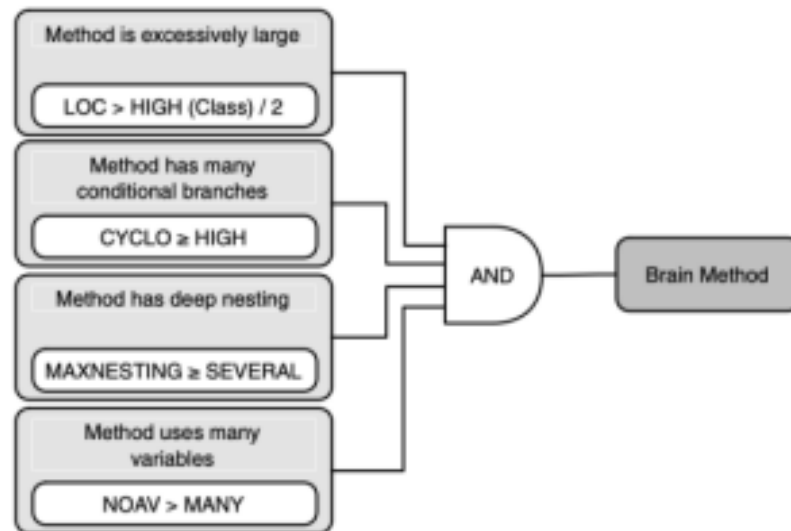
Tableau 10 : Seuils statistiques 2 [9]

Numeric Value	Semantic Label
0.25	ONE-QUARTER
0.33	ONE-THIRD
0.5	HALF
0.66	TWO-THIRDS
0.75	THREE-QUARTERS

Tableau 11 : Seuils sémantique [9]

## Présentation et identification des anomalies

### Long Method



**Figure 8:** Stratégie de détection « Long method »

On peut détecter l'anomalie « Long method » grâce au schéma de la Figure 9 tiré des notes de cours. On voit que la méthode `getNotes()` dans la classe `DbHelper` a beaucoup trop de fonctionnalités, c'est une anomalie « Long method ». Nous devons calculer les conditions suivantes pour confirmer l'anomalie :

Énoncé de condition	Condition	Résultat
Method is excessively large	$LOC (Method) > High(Class) / 2$ $88 > 65$	TRUE
Method has many conditional branches	$CC / LOC (Method) \geq HIGH$ $0.13 \not\geq 0.24$	FALSE

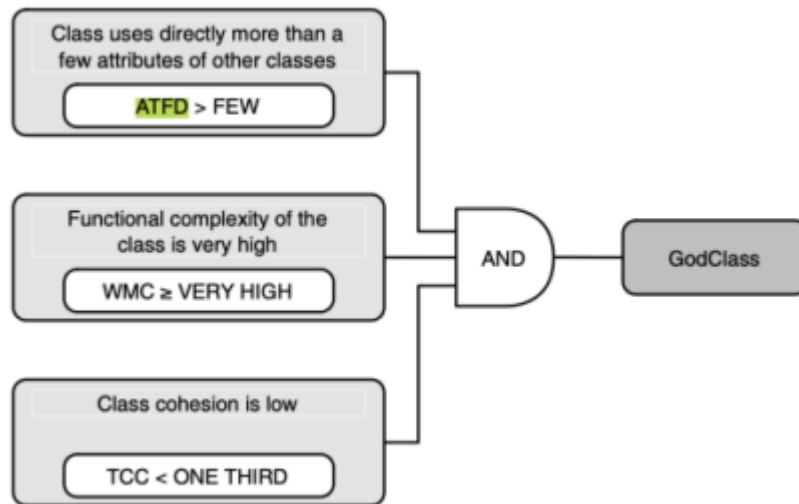
Method has deep nesting	$\text{MAXNESTING} \geq \text{SEVERAL}$ $3 \geq 2$	TRUE
Method uses many variables	$\text{NOAV} > \text{MANY}$ $137 > 7$	TRUE

Tableau 12 : Stratégie de détection « Long method »

La stratégie de détection de l'anti-patron « Long method » a permis de détecter 3 critères sur 4 qui indiquent que getNotes() est un exemple de cet anti-patron. Bien que la méthode ait peu de branches conditionnelles  $0.13 \neq 0.24$ , elle utilise un nombre excessif de variables, soit 137.

## Large Class

Pour l'antipattern « Large Class », la règle est définie à partir de la **Figure 10** tirée des notes de cours.



**Figure 9:** Stratégie de détection « Large class ».

Metric	Metrics Set	Description	Value
CHVL	Halstead M...	Halstead Volume	5172.1109
CHD	Halstead M...	Halstead Difficulty	169.5484
CHL	Halstead M...	Halstead Length	676
CHEF	Halstead M...	Halstead Effort	876923.0683
CHVC	Halstead M...	Halstead Vocabulary	201
CHER	Halstead M...	Halstead Errors	3.0539
WMC	Chidamber...	Weighted Methods Per Class	80
DIT	Chidamber...	Depth Of Inheritance Tree	1
CBO	Chidamber...	Coupling Between Objects	21
RFC	Chidamber...	Response For A Class	102
LCOM	Chidamber...	Lack Of Cohesion Of Methods	11
NOC	Chidamber...	Number Of Children	0
NOA	Lorenz-Kid...	Number Of Attributes	0
NOO	Lorenz-Kid...	Number Of Operations	43
NOOM	Lorenz-Kid...	Number Of Overridden Methods	0
NOAM	Lorenz-Kid...	Number Of Added Methods	31
SIZE2	Li-Henry M...	Number Of Attributes And Methods	43
NOM	Li-Henry M...	Number Of Methods	31
MPC	Li-Henry M...	Message Passing Coupling	132
DAC	Li-Henry M...	Data Abstraction Coupling	0
ATFD	Lanza-Mari...	Access To Foreign Data	8
NOPA	Lanza-Mari...	Number Of Public Attributes	0
NOAC	Lanza-Mari...	Number Of Accessor Methods	0
WOC	Lanza-Mari...	Weight Of A Class	1.0
TCC	Bieman-Kan...	Tight Class Cohesion	0.0065
NCSS	Chr. Clemen...	Non-Commenting Source Statements	220
CMI	Maintainabi...	Maintainability Index	17.2577

**Figure 10 :** Métrique dans la classe StorageHelper.java.



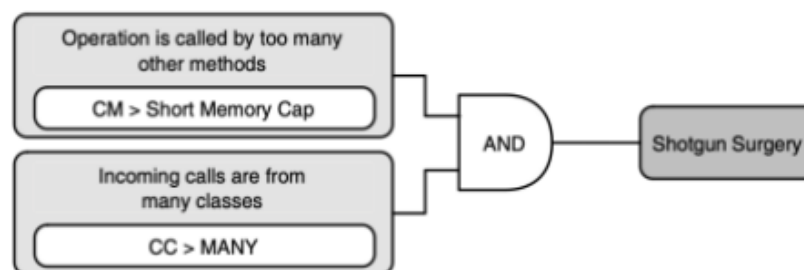
Énoncé de condition	Condition	Résultat
Class uses directly more than a few attributes of other classes	$ATFD = 8 > Few = 5$	TRUE
Functional complexity of the class is very high	$WMC = 80 > VERY\ HIGH = 47$	TRUE
Class cohesion is low	$TCC = 0.0065 < One\ Third = 0.33$	TRUE

Tableau 13 : Stratégie de détection « Large class ».

Dans le fichier StorageHelper.java, il y a l'anomalie « Large class » qu'on définit comme une classe qui contient beaucoup de membres. Dans ce cas, la classe possède énormément de membres et de responsabilité, car elle s'occupe de l'accès à la base de données de l'application. La valeur de WMC est de 80, comme le montre la **Figure 10**, ce qui est supérieur à la valeur VERY HIGH, Cela indique la présence de l'anomalie « Large class ».

## Shotgun Surgery

Pour l'anomalie « Shotgun Surgery », la règle est définie à partir de la **Figure 11** tirée des notes de cours.



**Figure 11:** Stratégie de détection « Shotgun Surgery ».

Énoncé de condition	Condition	Résultat
Operation is called by too many other methods	CM > Short Memory Cap 9 > 7	TRUE
Incoming calls are from many classes	CC > MANY 9 > 7	TRUE

Tableau 14 : Stratégie de détection « Shotgun surgery ».

La stratégie de détection de l'anti-patron « Shotgun surgery » a permis de détecter 2 critères sur 2 qui indiquent que la méthode `updateNote()` est un exemple de cet anti-patron. Le nombre de méthodes et de classes qui utilisent la méthode est bien au-dessus du seuil. Cette anomalie a plusieurs impacts comme un couplage élevé et une faible modularité car les modification a une méthode demande qu'on modifie plusieurs autres méthodes. La faible cohésion car la méthode cliente n'est pas très liée avec le reste de sa classe mais plutôt avec l'extérieur.

### Long Parameter List

Pour l'antipattern « Long Parameter List », d'intenses recherches ont été menées sur le web sur logiciels populaires liés à des sites Web tels que Refactoring Guru [6] et Principles Wiki [7]. Ainsi, la règle pour détecter l'odeur du code de la liste des paramètres longs est basée sur la métrique NOPM qui calcule le nombre de paramètres dans une méthode de classe ou constructeur. Normalement, notre recherche nous a permis de conclure que la règle pour l'odeur du code de la liste des paramètres longs est la suivante :  $NOPM > FEW$ . On voit clairement que le NOPM (nombre de paramètres) de la méthode `createNotification` égale à 5 qui est supérieur à  $FEW = 3$  ( $NOPM = 5 > FEW = 3$ ) donc l'antipatron « Long Parameter List » est vérifié.

## Switch statements

Pour l'anti-patron « Switch statements », on sait que le « Switch » permet d'exécuter différents blocs d'instructions selon la valeur d'une variable. Plusieurs chemins indépendants sont donc créés lors de la construction d'un graphe de flot de contrôle. La métrique McCabe Cyclomatic Complexity (MCC) est une métrique qui évalue le nombre de chemins indépendants, plus la valeur de MCC est élevée plus la fonction est complexe. On peut donc évaluer la métrique CYCLO/Line of code et comparer la valeur avec les valeurs dans le **Tableau 10**. L'évaluation de la métrique CYCLO/Line of Code pour la fonction onUpgradeTo476 donne une valeur de  $8/31=0.258$  qui est entre HIGH et VERY HIGH du **Tableau 10** ce qui indique que la fonction est très complexe.

## Refactoring pour corriger les anomalies

### Long method

Pour corriger une l'anomalie « Long method », il suffit de la décomposer en plusieurs plus petites méthodes qui exerce chacune une responsabilité unique. La « Long method » va simplement appeler les méthodes nouvellement créées. En d'autres termes, il s'agit de composer la « Long method » avec des appels à des méthodes plus petites et plus modulaires.

On voit que la méthode getNotes() dans la classe DbHelper a beaucoup trop de fonctionnalités, c'est une anomalie « Long method ». La méthode doit créer une variable qui indique comment trier les colonnes, une variable qui indique l'ordre de triage, créer une variable qui est la requête à exécuter et enfin exécuter la requête pour récupérer les notes.

```
public List<Note> getNotes(String whereCondition, boolean order) {
    List<Note> noteList = new ArrayList<>();

    String sortColumn = "";
    String sortOrder = "";

    // Getting sorting criteria from preferences. Reminder screen forces
    sorting.
    if (Navigation.checkNavigation(Navigation.REMINDERS)) {
        sortColumn = KEY_REMINDER;
```

```

    } else {
        sortColumn = Prefs.getString(PREF_SORTING_COLUMN, KEY_TITLE);
    }
    if (order) {
        sortOrder =
            KEY_TITLE.equals(sortColumn) || KEY_REMINDER.equals(sortColumn) ? "
ASC" : "DESC";
    }

    // In case of title sorting criteria it must be handled empty title by
    concatenating content
    sortColumn = KEY_TITLE.equals(sortColumn) ? KEY_TITLE + "||" + KEY_CONTENT
: sortColumn;

    // In case of reminder sorting criteria the empty reminder notes must be
    moved on bottom of results
    sortColumn = KEY_REMINDER.equals(sortColumn) ? "IFNULL(" + KEY_REMINDER +
" ," +
        "" + TIMESTAMP_UNIX_EPOCH + ")" : sortColumn;

    // Generic query to be specialized with conditions passed as parameter
    String query = "SELECT "
        + KEY_CREATION + " ,"
        + KEY_LAST_MODIFICATION + " ,"
        + KEY_TITLE + " ,"
        + KEY_CONTENT + " ,"
        + KEY_ARCHIVED + " ,"
        + KEY_TRASHED + " ,"
        + KEY_REMINDER + " ,"
        + KEY_REMINDER_FIRED + " ,"
        + KEY_RECURRENCE_RULE + " ,"
        + KEY_LATITUDE + " ,"
        + KEY_LONGITUDE + " ,"
        + KEY_ADDRESS + " ,"
        + KEY_LOCKED + " ,"
        + KEY_CHECKLIST + " ,"
        + KEY_CATEGORY + " ,"
        + KEY_CATEGORY_NAME + " ,"
        + KEY_CATEGORY_DESCRIPTION + " ,"
        + KEY_CATEGORY_COLOR
        + " FROM " + TABLE_NOTES
        + " LEFT JOIN " + TABLE_CATEGORY + " USING ( " + KEY_CATEGORY + " )"
        + whereCondition
        + (order ? " ORDER BY " + sortColumn + " COLLATE NOCASE " + sortOrder :
        "");

    LogDelegate.v("Query: " + query);

    try (Cursor cursor = getDatabase().rawQuery(query, null)) {
        if (cursor.moveToFirst()) {
            do {
                int i = 0;
                Note note = new Note();
                note.setCreation(cursor.getLong(i++));
                note.setLastModification(cursor.getLong(i++));
                note.setTitle(cursor.getString(i++));
            } while (cursor.moveToNext());
        }
    }

```

```

        note.setContent(cursor.getString(i++));
        note.setArchived("1".equals(cursor.getString(i++)));
        note.setTrashed("1".equals(cursor.getString(i++)));
        note.setAlarm(cursor.getString(i++));
        note.setReminderFired(cursor.getInt(i++));
        note.setRecurrenceRule(cursor.getString(i++));
        note.setLatitude(cursor.getString(i++));
        note.setLongitude(cursor.getString(i++));
        note.setAddress(cursor.getString(i++));
        note.setLocked("1".equals(cursor.getString(i++)));
        note.setChecklist("1".equals(cursor.getString(i++)));

        // Eventual decryption of content
        if (Boolean.TRUE.equals(note.isLocked())) {
            note.setContent(
                Security.decrypt(note.getContent(),
Prefs.getString(PREF_PASSWORD, ""));
        }

        // Set category
        long categoryId = cursor.getLong(i++);
        if (categoryId != 0) {
            Category category = new Category(categoryId, cursor.getString(i++),
                cursor.getString(i++), cursor.getString(i));
            note.setCategory(category);
        }

        // Add eventual attachments uri
        note.setAttachmentsList(getNoteAttachments(note));

        // Adding note to list
        noteList.add(note);
    } while (cursor.moveToNext());
}

}

LogDelegate.v("Query: Retrieval finished!");
return noteList;
}

```

**Figure 12** : Méthode getNotes avant modification.

Pour corriger ce code, on peut créer une méthode sortParams() pour les paramètres de triage sortColumn et sortOrder. On peut également créer une méthode constructQuery pour la construction de la requête à exécuter pour récupérer les données de la base de données. Avec l'extraction de ces méthodes, la méthode getNotes s'occupe seulement de récupérer les notes. Les trois prochaines figure montre les deux méthodes extraite de getNotes() puis la méthode getNotes() après le « refactoring ». Cette anomalie est

corrigée par notre « refactoring » car la « Long method » devient plus courte et mieux compréhensible à la lecture, de plus, son contenu devient plus réutilisable. Le code est maintenant plus modulaire. De plus si on recalcule les conditions en fonction des seuil on trouve le résultat suivant :

1) LOC (Method) > High(Class) / 2 51 $\nlessgtr$ 65 $\Rightarrow$ Condition is False	3) MAXNESTING $\geq$ SEVERAL 3 $\geq$ 2 $\Rightarrow$ Condition is True
2) CC / LOC (Method) $\geq$ HIGH 0.098 $\nlessgtr$ 0.24 $\Rightarrow$ Condition is False	4) NOAV > MANY 78 > 7 $\Rightarrow$ Condition is True

Tableau 15 : Stratégie de détection de « Long method » après « refactoring »

La première condition est passée à false et la quatrième condition est restée True mais NOAV a diminué d'un facteur 2.

```
1 usage new *
public List<String> sortParams(boolean order) {
    List<String> params=new ArrayList<>();

    String sortColumn = "";
    String sortOrder = "";

    if (Navigation.checkNavigation(Navigation.REMINDERS)) {
        sortColumn = KEY_REMINDER;
    } else {
        sortColumn = Prefs.getString(PREF_SORTING_COLUMN, KEY_TITLE);
    }
    if (order) {
        sortOrder =
            KEY_TITLE.equals(sortColumn) || KEY_REMINDER.equals(sortColumn) ? " ASC " : " DESC ";
    }

    // In case of title sorting criteria it must be handled empty title by concatenating content
    sortColumn = KEY_TITLE.equals(sortColumn) ? KEY_TITLE + "||" + KEY_CONTENT : sortColumn;

    // In case of reminder sorting criteria the empty reminder notes must be moved on bottom of results
    sortColumn = KEY_REMINDER.equals(sortColumn) ? "IFNULL(" + KEY_REMINDER + ", " +
        "" + TIMESTAMP_UNIX_EPOCH + ")" : sortColumn;

    params.add(sortColumn);
    params.add(sortOrder);

    return params;
}
```

Figure 13 : Méthode sortParams.

```

1 usage new *
public String constructQuery(String whereCondition,boolean order,String sortColumn,String sortOrder) {
    String query = "SELECT "
        + KEY_CREATION + ","
        + KEY_LAST_MODIFICATION + ","
        + KEY_TITLE + ","
        + KEY_CONTENT + ","
        + KEY_ARCHIVED + ","
        + KEY_TRASHED + ","
        + KEY_REMINDER + ","
        + KEY_REMINDER_FIRED + ","
        + KEY_RECURRENCE_RULE + ","
        + KEY_LATITUDE + ","
        + KEY_LONGITUDE + ","
        + KEY_ADDRESS + ","
        + KEY_LOCKED + ","
        + KEY_CHECKLIST + ","
        + KEY_CATEGORY + ","
        + KEY_CATEGORY_NAME + ","
        + KEY_CATEGORY_DESCRIPTION + ","
        + KEY_CATEGORY_COLOR
        + " FROM " + TABLE_NOTES
        + " LEFT JOIN " + TABLE_CATEGORY + " USING( " + KEY_CATEGORY + " ) "
        + whereCondition
        + (order ? " ORDER BY " + sortColumn + " COLLATE NOCASE " + sortOrder : "");

    return query;
}

```

Figure 14 : Méthode constructQuery

```

public List<Note> getNotes(String whereCondition, boolean order) {
    List<Note> noteList = new ArrayList<>();

    /** [0] sortColumn [1] sortOrder */
    List<String> params=sortParams(order);
    /** construction du query */
    String
    query=constructQuery(whereCondition,order,params.get(0),params.get(1));

    LogDelegate.v("Query: " + query);

    try (Cursor cursor = getDatabase().rawQuery(query, null)) {

        if (cursor.moveToFirst()) {
            do {
                int i = 0;
                Note note = new Note();
                note.setCreation(cursor.getLong(i++));
                note.setLastModification(cursor.getLong(i++));
                note.setTitle(cursor.getString(i++));
                note.setContent(cursor.getString(i++));
                note.setArchived("1".equals(cursor.getString(i++)));
                note.setTrashed("1".equals(cursor.getString(i++)));
                note.setAlarm(cursor.getString(i++));
            } while (cursor.moveToNext());
        }
    }

    return noteList;
}

```

```

        note.setReminderFired(cursor.getInt(i++));
        note.setRecurrenceRule(cursor.getString(i++));
        note.setLatitude(cursor.getString(i++));
        note.setLongitude(cursor.getString(i++));
        note.setAddress(cursor.getString(i++));
        note.setLocked("1".equals(cursor.getString(i++)));
        note.setChecklist("1".equals(cursor.getString(i++)));

        // Eventual decryption of content
        if (Boolean.TRUE.equals(note.isLocked())) {
            note.setContent(
                Security.decrypt(note.getContent(),
Prefs.getString(PREF_PASSWORD, ""));
        }

        // Set category
        long categoryId = cursor.getLong(i++);
        if (categoryId != 0) {
            Category category = new Category(categoryId, cursor.getString(i++),
                cursor.getString(i++), cursor.getString(i));
            note.setCategory(category);
        }

        // Add eventual attachments uri
        note.setAttachmentsList(getNoteAttachments(note));

        // Adding note to list
        noteList.add(note);
    } while (cursor.moveToNext());
}

LogDelegate.v("Query: Retrieval finished!");
return noteList;
}

```

**Figure 15** : Méthode getNotes après modification.

## Large class

Pour corriger l'anomalie « Large class », comme la classe a trop de membres et de responsabilités, on va extraire de plus petites classes pour alléger la « Large class ». Elle aura donc moins de responsabilités car celles-ci seront distribuées à plusieurs petites classes qui seront contenues dans la classe originale. Cette anomalie est corrigée car la « Large class » devient plus petite et mieux compréhensible à la lecture, de plus, son contenu devient plus réutilisable.



Étant donné que la classe StorageHelper est un exemple de « Large class », une refactorisation de type « extract class » semble être suffisante pour régler cette anomalie. En effet, nous avons décidé de remanier la classe StorageHelper dans le fichier StorageHelper.java en la divisant en plusieurs classes différentes (Storage, Fichier, Dir).

```
public static boolean checkStorage() {
    boolean mExternalStorageAvailable;
    boolean mExternalStorageWriteable;
    String state = Environment.getExternalStorageState();

    switch (state) {
        case Environment.MEDIA_MOUNTED:
            // We can read and write the media
            mExternalStorageAvailable = mExternalStorageWriteable = true;
            break;
        case Environment.MEDIA_MOUNTED_READ_ONLY:
            // We can only read the media
            mExternalStorageAvailable = true;
            mExternalStorageWriteable = false;
            break;
        default:
            // Something else is wrong. It may be one of many other states, but
            // all we need
            // to know is we can neither read nor write
            mExternalStorageAvailable = mExternalStorageWriteable = false;
            break;
    }
    return mExternalStorageAvailable && mExternalStorageWriteable;
}
```

**Figure 16 :** La méthode checkStorage avant le refactoring.

```
public static boolean checkStorage() {
    return Storage.checkStorage();
}
```

**Figure 17 :** La méthode checkStorage après le refactoring.

```
public static File getDbSyncDir(Context mContext) {
    File extFilesDir = mContext.getExternalFilesDir(null);
    File dbSyncDir = new File(extFilesDir, Constants.APP_STORAGE_DIRECTORY_SB_SYNC);
    dbSyncDir.mkdirs();
    if (dbSyncDir.exists() && dbSyncDir.isDirectory()) {
        return dbSyncDir;
    } else {
        return null;
    }
}
```

**Figure 18 :** La méthode getDbSyncDir avant le refactoring.

```
public static File getDbSyncDir(Context mContext) {
    return Storage.getDbSyncDir(mContext);
}
```

**Figure 19 :** La méthode getDbSyncDir après le refactoring.

```

public static File createExternalStoragePrivateFile(Context mContext, Uri uri, String extension) {

    if (!checkStorage()) {
        Toast.makeText(mContext, mContext.getString(R.string.storage_not_available),
            Toast.LENGTH_SHORT).show();
        return null;
    }
    File file = createNewAttachmentFile(mContext, extension);

    InputStream contentResolverInputStream = null;
    OutputStream contentResolverOutputStream = null;
    try {
        contentResolverInputStream = mContext.getContentResolver().openInputStream(uri);
        contentResolverOutputStream = new FileOutputStream(file);
        copyFile(contentResolverInputStream, contentResolverOutputStream);
    } catch (IOException e) {
        try {
            FileUtils.copyFile(new File(FileHelper.getPath(mContext, uri)), file);
            // It's a path!!
        } catch (NullPointerException e1) {
            try {
                FileUtils.copyFile(new File(uri.getPath()), file);
            } catch (IOException e2) {
                LogDelegate.e( message: "Error writing " + file, e2);
                file = null;
            }
        }
        LogDelegate.e( message: "Error writing " + file, e2);
        file = null;
    } finally {
        try {
            if (contentResolverInputStream != null) {
                contentResolverInputStream.close();
            }
            if (contentResolverOutputStream != null) {
                contentResolverOutputStream.close();
            }
        } catch (IOException e) {
            LogDelegate.e( message: "Error closing streams", e);
        }
    }
    return file;
}

```

**Figure 20 :** La méthode createExternalStoragePrivateFile avant le refactoring.

```

public static File createExternalStoragePrivateFile(Context mContext, Uri uri, String extension) {
    return Storage.createExternalStoragePrivateFile(mContext, uri, extension);
}

```

**Figure 21 :** La méthode createExternalStoragePrivateFile après le refactoring.

```

public static boolean copyFile(Context context, Uri fileUri, Uri targetUri) {
    ContentResolver content = context.getContentResolver();
    try (var is :InputStream = content.openInputStream(fileUri);
        var os :OutputStream = content.openOutputStream(targetUri)) {
        IOUtils.copy(is, os);
        return true;
    } catch (IOException e) {
        LogDelegate.e( message: "Error copying file", e);
        return false;
    }
}

```

**Figure 22 :** La méthode copyFile avant le refactoring.

```

usage  Federico Iosue *
public static boolean copyFile(Context context, Uri fileUri, Uri targetUri) {
    return Fichier.copyFile(context, fileUri, targetUri);
}

```

**Figure 23 :** La méthode copyFile après le refactoring.

```

usage  Federico Iosue
public static boolean deleteExternalStoragePrivateFile(Context mContext, String name) {
    // Checks for external storage availability
    if (!checkStorage()) {
        Toast.makeText(mContext, "Storage not available",
            Toast.LENGTH_SHORT).show();
        return false;
    }
    File file = new File(mContext.getExternalFilesDir(null), name);
    return file.delete();
}

```

**Figure 24 :** La méthode deleteExternalStoragePrivateFile avant le refactoring.

```

1 usage  Federico Iosue *
public static boolean deleteExternalStoragePrivateFile(Context mContext, String name) {
    return Fichier.deleteExternalStoragePrivateFile(mContext, name);
}

```

**Figure 25 :** La méthode deleteExternalStoragePrivateFile après le refactoring.

```

public static boolean delete(Context mContext, String path) {
    if (!checkStorage()) {
        Toast.makeText(mContext, "Storage not available",
            Toast.LENGTH_SHORT).show();
        return false;
    }
    try {
        FileUtils.forceDelete(new File(path));
    } catch (IOException e) {
        LogDelegate.e("Can't delete '" + path + "': " + e.getMessage());
        return false;
    }
    return true;
}

```

**Figure 26 :** La méthode delete avant le refactoring.

```

Federico Iosue *
public static boolean delete(Context mContext, String path) {
    return Fichier.delete(mContext, path);
}

```

**Figure 27 :** La méthode delete après le refactoring.

```

public static String getRealPathFromURI(Context mContext, Uri contentUri) {
    String[] proj = {MediaStore.Images.Media.DATA};
    Cursor cursor = mContext.getContentResolver().query(contentUri, proj, selection: null, selectionArgs: null, sortOrder: null);
    if (cursor == null) {
        return null;
    }
    int column_index = cursor.getColumnIndexOrThrow(MediaStore.Images.Media.DATA);
    cursor.moveToFirst();
    String path = cursor.getString(column_index);
    cursor.close();
    return path;
}

```

**Figure 28 :** La méthode `getRealPathFromURI` avant le refactoring.

```
△ Federico Iosue *  
public static String getRealPathFromURI(Context mContext, Uri contentUri) {  
    return Fichier.getRealPathFromURI(mContext, contentUri);  
}  
}
```

**Figure 29 :** La méthode `getRealPathFromURI` après le refactoring.

```
public static File copyToBackupDir(File backupDir, String fileName, InputStream fileInputStream) {  
    if (!checkStorage()) {  
        return null;  
    }  
    if (!backupDir.exists()) {  
        backupDir.mkdirs();  
    }  
    File destination = new File(backupDir, fileName);  
    try {  
        copyFile(fileInputStream, new FileOutputStream(destination));  
    } catch (FileNotFoundException e) {  
        e.printStackTrace();  
    }  
    return destination;  
}
```

**Figure 30 :** La méthode `copyToBackupDir` avant le refactoring.

```
△ Federico Iosue *  
public static File copyToBackupDir(File backupDir, String fileName, InputStream fileInputStream) {  
    return Dir.copyToBackupDir( backupDir, fileName, fileInputStream);  
}
```

**Figure 31 :** La méthode `copyToBackupDir` après le refactoring.

```
public static String getMimeTypeInternal(Context mContext, String mimeType) {  
    if (mimeType != null) {  
        if (mimeType.contains("image/")) {  
            mimeType = Constants.MIME_TYPE_IMAGE;  
        } else if (mimeType.contains("audio/")) {  
            mimeType = Constants.MIME_TYPE_AUDIO;  
        } else if (mimeType.contains("video/")) {  
            mimeType = Constants.MIME_TYPE_VIDEO;  
        } else {  
            mimeType = Constants.MIME_TYPE_FILES;  
        }  
    }  
    return mimeType;  
}
```

**Figure 32:** la méthode `getMimeTypeInternal` avant le refactoring.

```
public static String getMimeTypeInternal(Context mContext, String mimeType) {  
    return Dir.getMimeTypeInternal( mContext, mimeType);  
}
```

**Figure 33 :** la méthode `getMimeTypeInternal` après le refactoring.

```

3 usages  Federico Iosue
public static Attachment createAttachmentFromUri(Context mContext, Uri uri, boolean moveSource) {
    String name = FileHelper.getNameFromUri(mContext, uri);
    String extension = FileHelper.getFileExtension(FileHelper.getNameFromUri(mContext, uri))
        .toLowerCase(
            Locale.getDefault());
    File f;
    if (moveSource) {
        f = createNewAttachmentFile(mContext, extension);
        try {
            FileUtils.moveFile(new File(uri.getPath()), f);
        } catch (IOException e) {
            LogDelegate.e("Can't move file " + uri.getPath());
        }
    } else {
        f = StorageHelper.createExternalStoragePrivateFile(mContext, uri, extension);
    }
    Attachment mAttachment = null;
    if (f != null) {
        mAttachment = new Attachment(Uri.fromFile(f),
            StorageHelper.getMimeTypeInternal(mContext, uri));
        mAttachment.setName(name);
        mAttachment.setSize(f.length());
    }
    return mAttachment;
}

```

**Figure 34 :** La méthode createAttachmentFromUri avant le refactoring.

```

3 usages  Federico Iosue
public static Attachment createAttachmentFromUri(Context mContext, Uri uri, boolean moveSource) {
    return Dir.createAttachmentFromUri(mContext, uri, moveSource);
}

```

**Figure 35 :** La méthode createAttachmentFromUri après le refactoring.

Class: Storage			
Metric	Metric Set	Description	Value
CHVL	Halstead Metric Set	ad Volume	264.9721
CHD	Halstead M.L.	Halstead Difficulty	15.75
CHL	Halstead M.L.	Halstead Length	54
CHEF	Halstead M.L.	Halstead Effort	4173.3105
CHVC	Halstead M.L.	Halstead Vocabulary	30
CHER	Halstead M.L.	Halstead Errors	0.0864
WMC	Chidamber...	Weighted Methods Per Class	8
DIT	Chidamber...	Depth Of Inheritance Tree	1
RFC	Chidamber...	Response For A Class	11
LCOM	Chidamber...	Lack Of Cohesion Of Methods	4
NOC	Chidamber...	Number Of Children	0
NOA	Lorenz-Kid...	Number Of Attributes	0
NOO	Lorenz-Kid...	Number Of Operations	16
NOOM	Lorenz-Kid...	Number Of Overridden Methods	0
NOAM	Lorenz-Kid...	Number Of Added Methods	4
SIZE2	Li-Henry M.L.	Number Of Attributes And Methods	16
NOM	Li-Henry M.L.	Number Of Methods	4
MPC	Li-Henry M.L.	Message Passing Coupling	6
DAC	Li-Henry M.L.	Data Abstraction Coupling	0
NCSS	Chr. Clemen...	Non-Commenting Source Statements	20
CMI	Maintainabi...	Maintainability Index	47.8169

**Figure 36 :** Les métriques de la classe Storage.

Class: Fichier			
Metric	Metrics Set	Description	Value
CHVL	Halstead M...	Halstead Volume	2969.6469
CHD	Halstead M...	Halstead Difficulty	113.3871
CHL	Halstead M...	Halstead Length	419
CHEF	Halstead M...	Halstead Effort	336719.6439
CHVC	Halstead M...	Halstead Vocabulary	136
CHER	Halstead M...	Halstead Errors	1.6133
WMC	Chidamber...	Weighted Methods Per Class	46
DIT	Chidamber...	Depth Of Inheritance Tree	1
RFC	Chidamber...	Response For A Class	71
LCOM	Chidamber...	Lack Of Cohesion Of Methods	8
NOC	Chidamber...	Number Of Children	0
NOA	Lorenz-Kid...	Number Of Attributes	0
NOO	Lorenz-Kid...	Number Of Operations	30
NOOM	Lorenz-Kid...	Number Of Overridden Methods	0
NOAM	Lorenz-Kid...	Number Of Added Methods	18
SIZE2	Li-Henry M...	Number Of Attributes And Methods	30
NOM	Li-Henry M...	Number Of Methods	18
MPC	Li-Henry M...	Message Passing Coupling	88
DAC	Li-Henry M...	Data Abstraction Coupling	0
NCSS	Chr. Clemen...	Non-Commenting Source Statements	133
CMI	Maintainabi...	Maintainability Index	23.3671

**Figure 37** : Les métriques de la classe Fichier.

Class: Dir			
Metric	Metrics Set	Description	Value
CHVL	Halstead M...	Halstead Volume	2704.3502
CHD	Halstead M...	Halstead Difficulty	111.7119
CHL	Halstead M...	Halstead Length	381
CHEF	Halstead M...	Halstead Effort	302108.0055
CHVC	Halstead M...	Halstead Vocabulary	137
CHER	Halstead M...	Halstead Errors	1.5008
WMC	Chidamber...	Weighted Methods Per Class	44
DIT	Chidamber...	Depth Of Inheritance Tree	1
RFC	Chidamber...	Response For A Class	73
LCOM	Chidamber...	Lack Of Cohesion Of Methods	9
NOC	Chidamber...	Number Of Children	0
NOA	Lorenz-Kid...	Number Of Attributes	0
NOO	Lorenz-Kid...	Number Of Operations	31
NOOM	Lorenz-Kid...	Number Of Overridden Methods	0
NOAM	Lorenz-Kid...	Number Of Added Methods	19
SIZE2	Li-Henry M...	Number Of Attributes And Methods	31
NOM	Li-Henry M...	Number Of Methods	19
MPC	Li-Henry M...	Message Passing Coupling	76
DAC	Li-Henry M...	Data Abstraction Coupling	0
NCSS	Chr. Clemen...	Non-Commenting Source Statements	125
CMI	Maintainabi...	Maintainability Index	24.8485

**Figure 38** : Les métriques de la classe Dir.

Comme on peut le voir sur les métriques de la classe StorageHelper la valeur de WMC est passé de 80 à 43 comme on peut le voir sur Figure ci-dessous, ce qui est une bonne amélioration.

Class: StorageHelper			
Metric	Metrics Set	Description	Value
CHVL	Halstead M...	Halstead Volume	2064.2523
CHD	Halstead M...	Halstead Difficulty	74.1383
CHL	Halstead M...	Halstead Length	301
CHEF	Halstead M...	Halstead Effort	153040.1504
CHVC	Halstead M...	Halstead Vocabulary	116
CHER	Halstead M...	Halstead Errors	0.9537
WMC	Chidamber...	Weighted Methods Per Class	43
DIT	Chidamber...	Depth Of Inheritance Tree	1
CBO	Chidamber...	Coupling Between Objects	23
RFC	Chidamber...	Response For A Class	72
LCOM	Chidamber...	Lack Of Cohesion Of Methods	24
NOC	Chidamber...	Number Of Children	0
NOA	Lorenz-Kid...	Number Of Attributes	0
NOD	Lorenz-Kid...	Number Of Operations	43
NODM	Lorenz-Kid...	Number Of Overridden Methods	0
NOAM	Lorenz-Kid...	Number Of Added Methods	31
SIZE2	Li-Hervey M...	Number Of Attributes And Methods	43
NOM	Li-Hervey M...	Number Of Methods	31
MPC	Li-Hervey M...	Message Passing Coupling	52
DAC	Li-Hervey M...	Data Abstraction Coupling	0
ATFD	Lanza-Mari...	Access To Foreign Data	5
NOPA	Lanza-Mari...	Number Of Public Attributes	0
NOAC	Lanza-Mari...	Number Of Accessor Methods	0
WOC	Lanza-Mari...	Weight Of A Class	1.0
TCC	Bieman-Kan...	Tight Class Cohesion	0.0
NCSS	Chr. Clemen...	Non-Commenting Source Statements	71
CMI	Maintainabi...	Maintainability Index	26.8285

**Figure 39 :** Les métriques de la classe StorageHelper.

## Shotgun surgery

Pour corriger l'anomalie « shotgun surgery », il suffit de déplacer certaines méthodes. Une des raisons possibles de cette anomalie est que les méthodes ont été implémentées dans les mauvaises classes ce qui fait que lors d'une modification, on doit changer le code à plusieurs endroits.

## Long parameter list

Pour corriger Long parameter lists, on peut simplement regrouper les paramètres communs dans des objets et faire passer ces objets en paramètre. Le nombre de paramètres de la fonction va diminuer et elle pourrait appeler les attributs de l'objet.

On voit que la méthode createNotification a trop de paramètres ce qui est un exemple de « Long parameter list ». Pour régler ce problème on peut créer un objet qui contient ces paramètres et passer l'objet en paramètre.

```

2 usages  Federico Iosue
public NotificationsHelper createNotification(
    @NonNull NotificationChannels.NotificationChannelNames channelName, int
    smallIcon, String title, PendingIntent notifyIntent, boolean isOngoing) {
    mBuilder = new NotificationCompat.Builder(mContext,
        NotificationChannels.channels.get(channelName).id)
        .setSmallIcon(smallIcon)
        .setContentTitle(title)
        .setAutoCancel(!isOngoing)
        .setOngoing(isOngoing)
        .setColor(mContext.getResources().getColor(R.color.colorAccent))
        .setContentIntent(notifyIntent);

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
        setLargeIcon(R.drawable.logo_notification_lollipop);
    } else {
        setLargeIcon(R.mipmap.ic_launcher);
    }

    return this;
}

```

Figure 40 : Méthode createNotification avant modification.

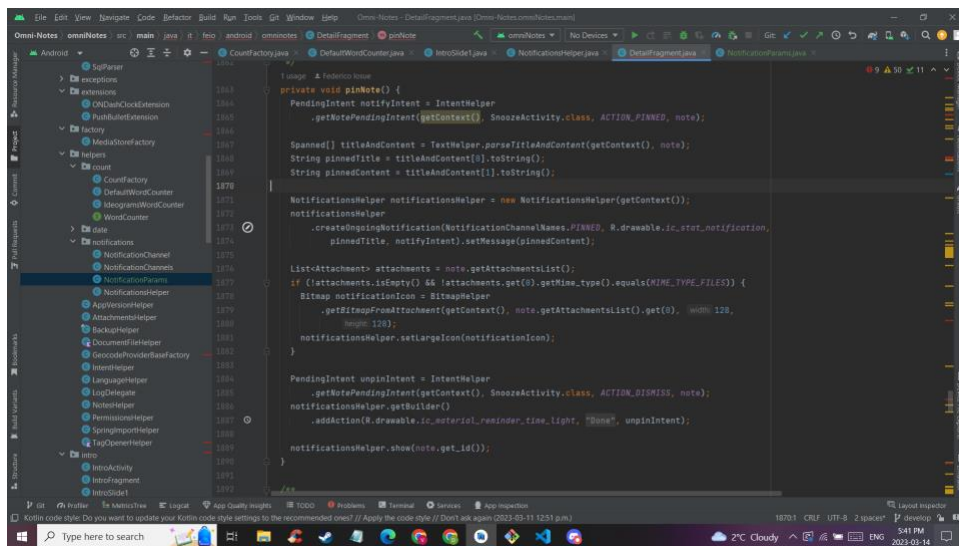


Figure 41 : Méthode pinNote de la classe DetailFragment avant modification.



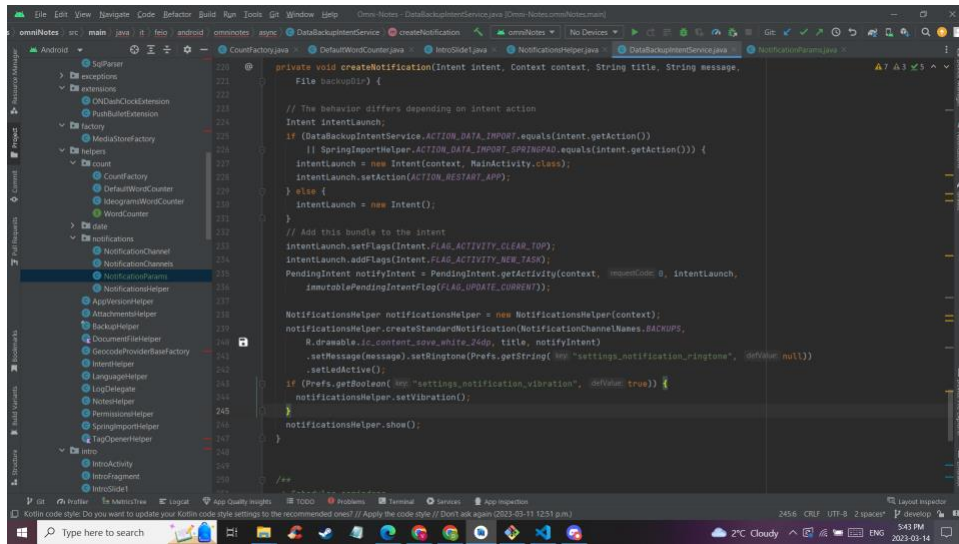


Figure 42 : Méthode `createNotification` de la classe `DataBackupIntentService` avant modification.

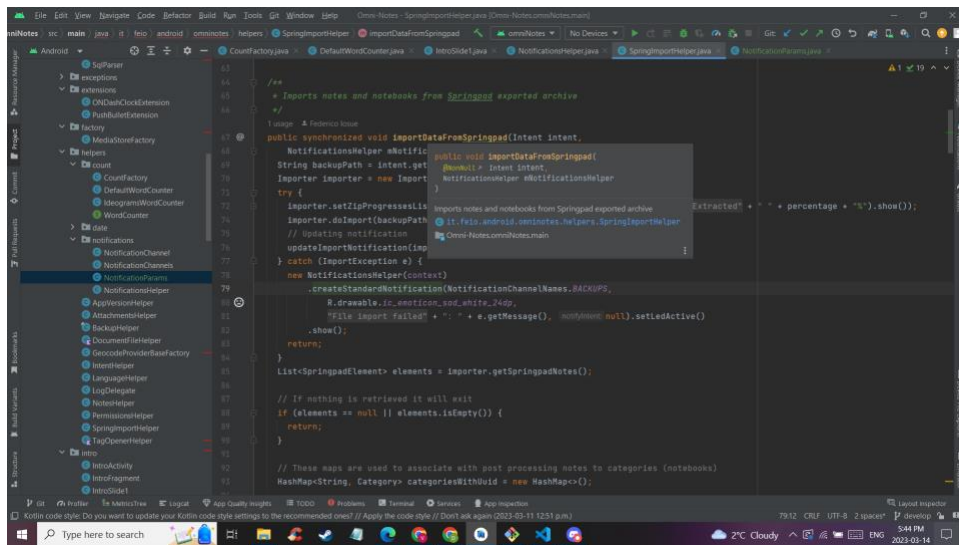
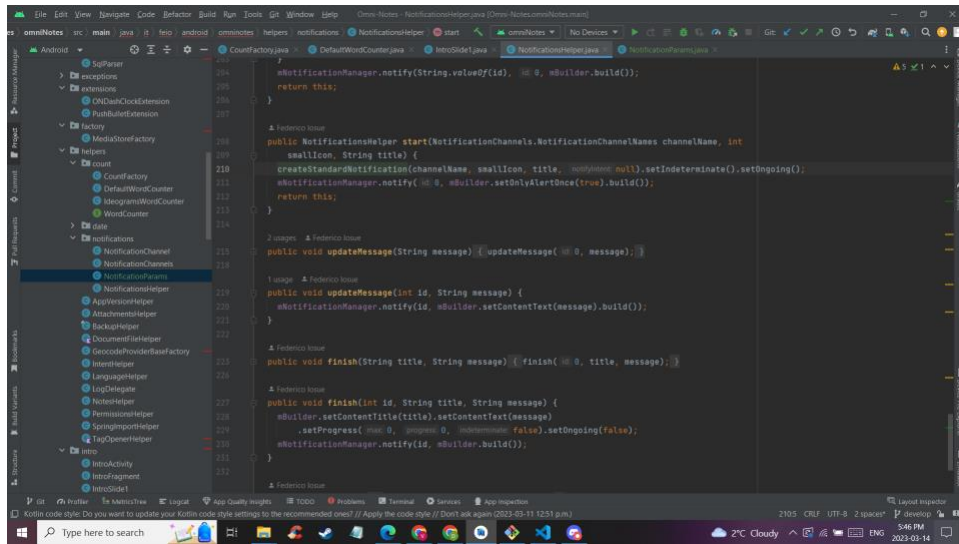
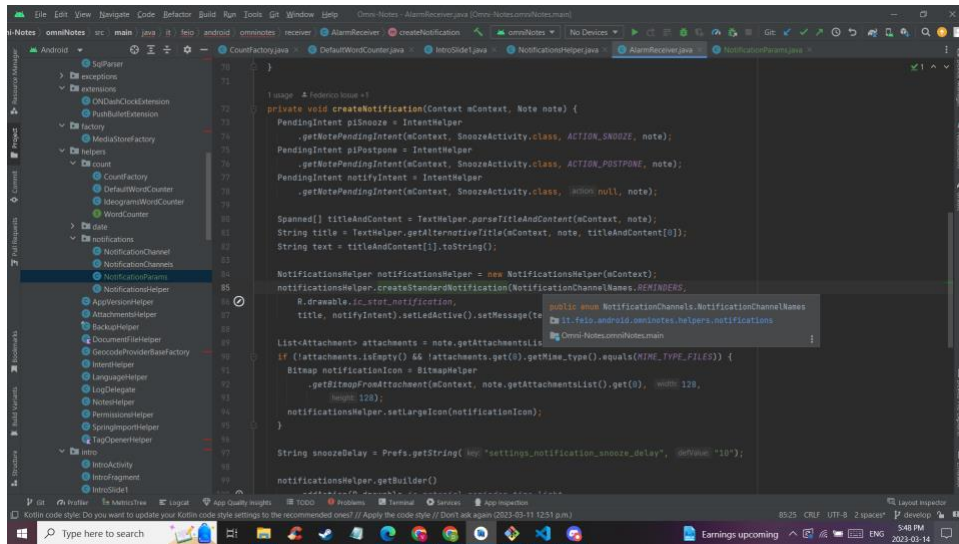


Figure 43 : Méthode `importDataFromSpringpad` de la classe `SpringImportHelper` avant modification.



**Figure 44 :** Méthode start de la classe SpringImportHelper avant modification.



**Figure 45 :** Méthode createNotification de la classe AlarmReceiver avant modification.

Pour corriger ce code, on peut créer une classe `NotificationParams` qui va contenir les paramètres qui sont passés dans une séquence d'appels.

```

2 usages  Federico Iosue *
public NotificationsHelper createNotification(
    NotificationParams params) {
    mBuilder = new NotificationCompat.Builder(mContext,
        NotificationChannels.channels.get(params.channelName).id)
        .setSmallIcon(params.smallIcon)
        .setContentTitle(params.title)
        .setAutoCancel(!params.isOngoing)
        .setOngoing(params.isOngoing)
        .setColor(mContext.getResources().getColor(R.color.colorAccent))
        .setContentIntent(params.notifyIntent);

    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
        setLargeIcon(R.drawable.logo_notification_lollipop);
    } else {
        setLargeIcon(R.mipmap.ic_launcher);
    }

    return this;
}

```

**Figure 46** : Méthode createNotification après modification.

```

4 usages  Federico Iosue *
public NotificationsHelper createStandardNotification(NotificationParams params) {
    params.setIsOngoing(false);
    return createNotification(params);
}

1 usage  Federico Iosue *
public NotificationsHelper createOngoingNotification(NotificationParams params) {
    params.setIsOngoing(true);
    return createNotification(params);
}

```

**Figure 47** : Méthode pour les 2 types de notifications après modification.

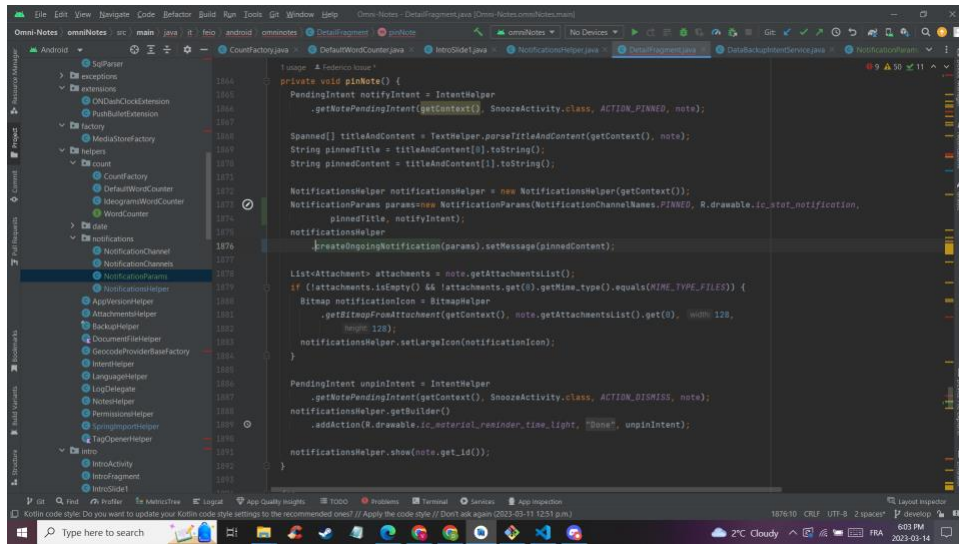


Figure 48 : Méthode pinNote de la classe DetailFragment après modification.

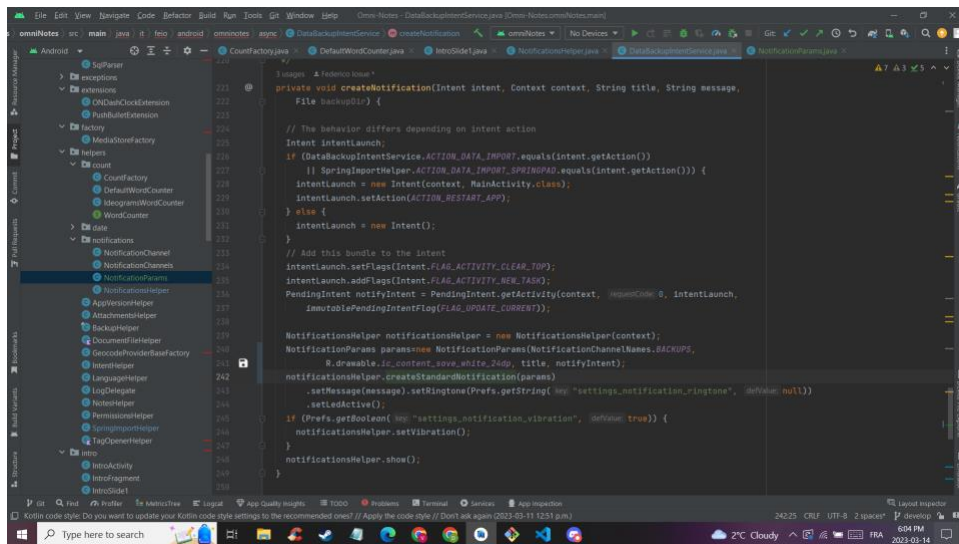
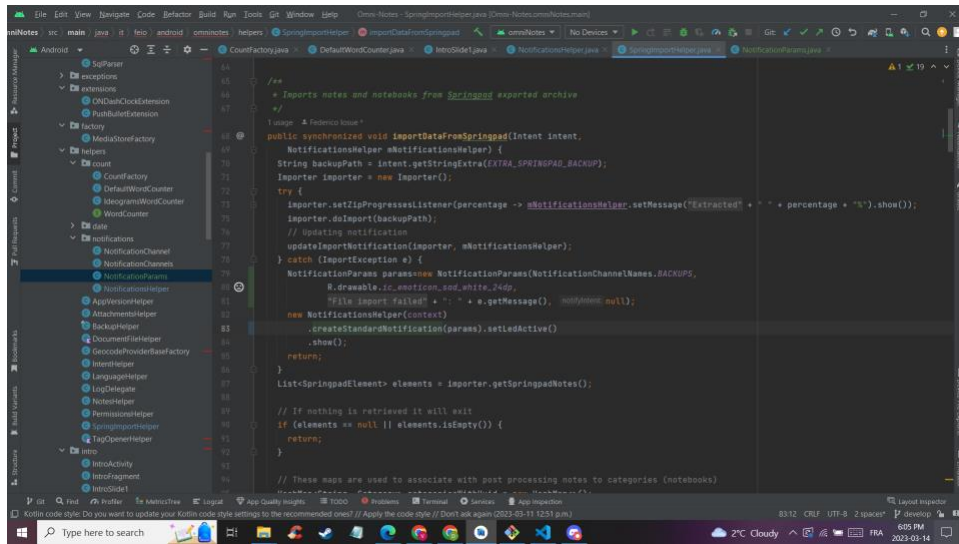
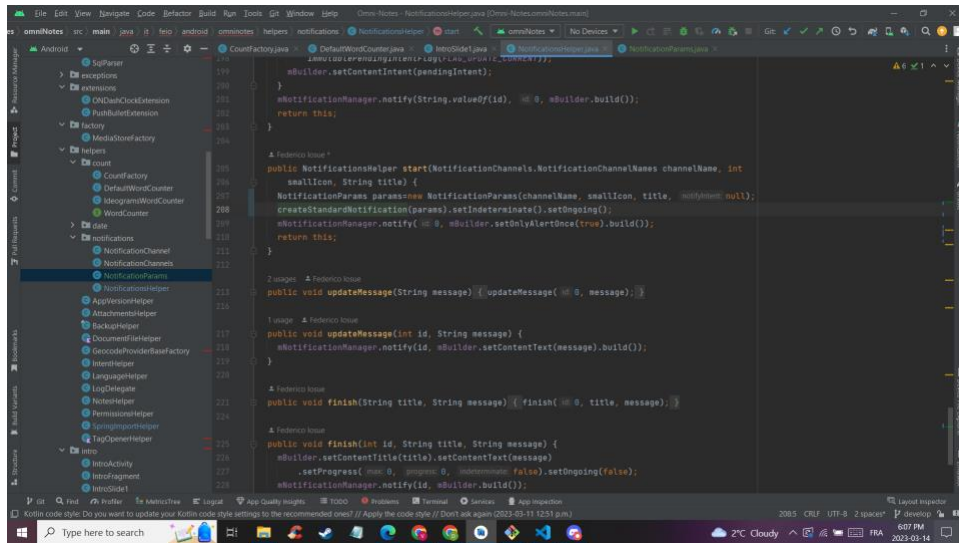


Figure 49 : Méthode createNotification de la classe DataBackupIntentService après modification.



**Figure 50 :** Méthode importDataFromSpringPad de la classe SpringImportHelper après modification.



**Figure 51 :** Méthode start de la classe SpringImportHelper après modification.



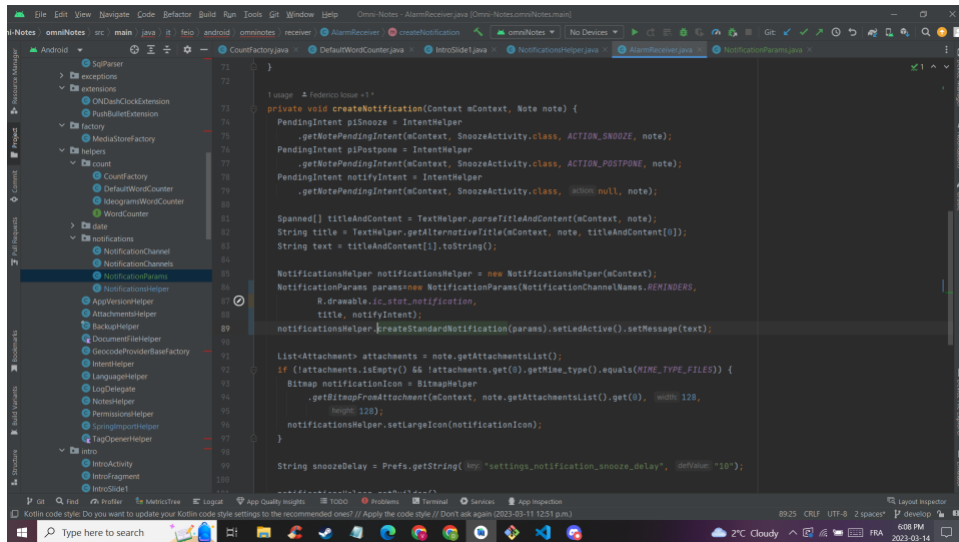


Figure 52 : Méthode createNotification de la classe AlarmReceiver après modification.

## Switch statement

Pour corriger le switch statement, on peut exploiter le concept de polymorphisme pour la détection de type et l'exécution de différentes séries d'instruction. On peut créer une classe de base pour MimeType et faire des classes dérivées pour ces types. Ce changement va alléger la complexité de la fonction car la détection du MimeType va se faire automatiquement et la maintenance de la fonction va être beaucoup plus facile.

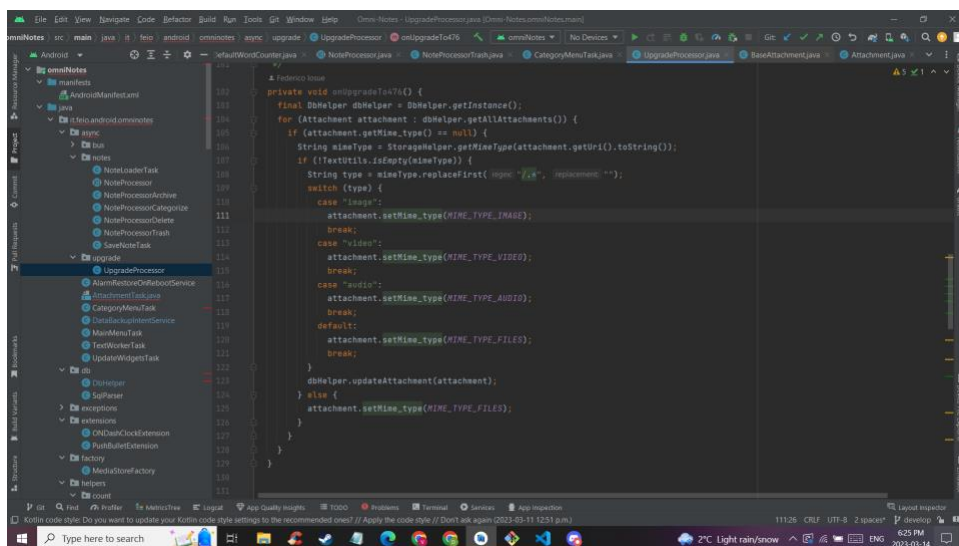
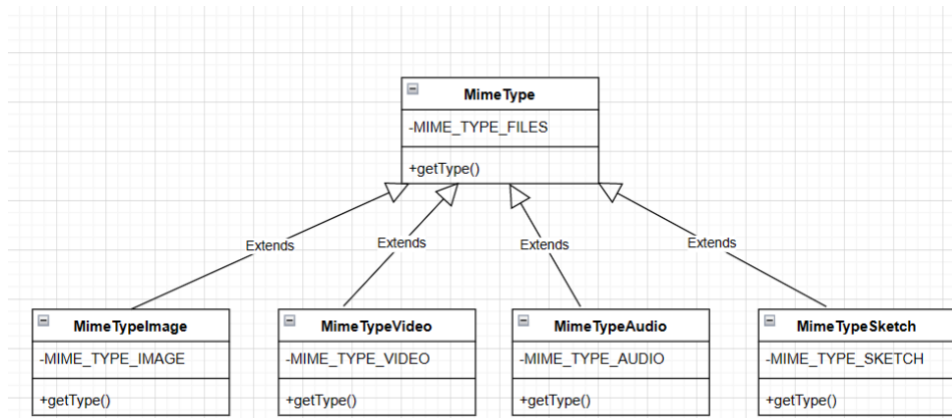


Figure 53: Méthode onUpgradeTo476 de la classe UpgradeProcessor avant modification.

Cette factorisation ne va pas être implémenté en code dû au fait qu'il y a beaucoup trop de codes à l'intérieur du code source. En dessous, on présente le UML de base nécessaire pour le changement où on dérive MimeTypeImage, MimeTypeVideo, MimeTypeAudio et MimeTypeSketch de la classe de base MimeType et pour chacun des classes dérivées on va « override » la méthode getType().

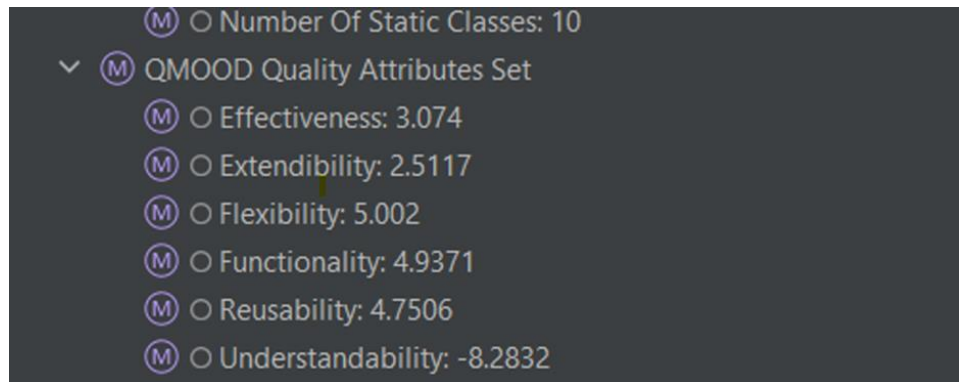


**Figure 54:** Classe de base et classes dérivées de MimeType

Comme le changement ne va pas être implémenté dans le code on ne va pas pouvoir vérifier s'il y a eu des changements dans les métriques de cette classe. Cependant, on assume que la complexité cyclomatique de la méthode va diminuer considérablement dû au fait qu'il va avoir moins de chemins indépendants si on construit un graphe de flot de contrôle et le calcul de CYCLO/Line va également diminuer. La méthode devrait être beaucoup moins complexe.

L'application de ce changement devrait améliorer la moyenne du système de la métrique de complexité cyclomatique, car on trouve plusieurs classes qui font des vérifications de type de MimeType.

## Métriques pour les critères de qualité après le refactoring



**Figure 55:** Les métriques correspondant aux critères de qualité.

Le tableau présenté ci-dessous met en évidence les différences entre les métriques associées aux critères de qualité avant et après la refonte.

Qualité des attributs	Valeurs avant la refonte	Valeurs après la refonte
Réutilisabilité	4.7494	4.7506
Flexibilité	4.853	5.002
Compréhensibilité	-8.2682	-8.2832
Fonctionnalité	4.9266	4.9371
Extensibilité	2.4873	2.5117
Efficacité	3.0152	3.074

Tableau 16 : Les métriques correspondant aux critères de qualité.

D'après le tableau, on peut voir qu'il y a une amélioration en ce qui concerne certaines de ces mesures, mais pas toutes.



Les mesures de la Réutilisabilité et de la Flexibilité ont augmenté après la refonte de l'application, ce qui peut être considéré comme une amélioration. Cela suggère que l'application peut être plus facilement réutilisée et modifiée pour répondre aux besoins futurs. Les améliorations de ces métriques peuvent être dues à l'application de bonnes pratiques de développement logiciel, telles que la modularité et la cohérence.

Cependant, la mesure de la Compréhensibilité a diminué après la refonte, ce qui suggère que l'application peut être devenue plus difficile à comprendre pour les développeurs et les utilisateurs. Cela peut être dû à des modifications importantes apportées à l'architecture de l'application, qui ont introduit des complexités supplémentaires. Une faible Compréhensibilité peut entraîner des coûts supplémentaires pour la maintenance et le support de l'application.

Les mesures de la Fonctionnalité, de l'Extensibilité et de l'Efficacité ont légèrement augmenté après la refonte, mais ces améliorations sont assez minimes. L'amélioration de la Fonctionnalité suggère que l'application fonctionne mieux qu'auparavant, tandis que l'amélioration de l'Extensibilité suggère que l'application peut être plus facilement étendue pour ajouter de nouvelles fonctionnalités. L'amélioration de l'Efficacité suggère que l'application peut être devenue légèrement plus rapide ou utilise moins de ressources qu'auparavant.

En résumé, la refonte de l'application a entraîné des améliorations dans certains aspects de la qualité du logiciel, mais aussi des régressions dans d'autres. Les améliorations peuvent être attribuées à l'application de bonnes pratiques de développement logiciel, tandis que les régressions peuvent être dues à des modifications importantes apportées à l'architecture de l'application.

## Références :

[1] Cyclomatic Complexity - GeeksforGeeks. (2018). En ligne. Disponible:

<https://www.geeksforgeeks.org/cyclomatic-complexity/>

[2] Rafa E. Al-Qutaish, International Journal of Software Engineering and Its Applications. (2014). En ligne. Disponible:

[https://www.researchgate.net/publication/260835125\\_Chidamber\\_and\\_Kemerer\\_Object-Oriented\\_Measures\\_Analysis\\_of\\_their\\_Design\\_from\\_the\\_Metrology\\_Perspective](https://www.researchgate.net/publication/260835125_Chidamber_and_Kemerer_Object-Oriented_Measures_Analysis_of_their_Design_from_the_Metrology_Perspective)

[3] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. (juin 1994). En ligne. Disponible:

<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=295895>

[4] Aivosto. (2013). *Chidamber & Kemerer object-oriented metrics suite*. En ligne. Disponible:

<http://people.scs.carleton.ca/~jeanpier/sharedF14/T1/extra%20stuff/about%20metrics/Cnidamber%20&%20Kemerer%20object-oriented%20metrics%20suite.pdf>

[5] Kecia A.M. Ferreira, Mariza A.S. Bigonha, Roberto S. Bigonha, Luiz F.O. Mendes, Heitor C. Almeida. The Journal of Systems and Software. (2012). En ligne. Disponible:

<http://llp.dcc.ufmg.br/Publications/Journal2012/JSS-journal.pdf>

[6] "Long Parameter List," Refactoring Guru. En ligne. Disponible:

<https://refactoring.guru/smells/long-parameter-list>

[7] "Long Parameter List," Principle Wiki. En ligne. Disponible: [http://principles-wiki.net/anti-patterns:long\\_parameter\\_list](http://principles-wiki.net/anti-patterns:long_parameter_list)

[8] Object-Oriented Metrics in Practice. En ligne. Disponible:

<https://link.springer.com/book/10.1007/3-540-39538-5>