



LOG8430 - Architecture logicielle et conception avancée

Travail Pratique #2 Qualité de la conception, anomalies et refactoring

Équipe: Atwater

2065803 - Vincent Boogaart
1871842 - Ben Jemaa Manel
2024717 - Henri Larocque
1621855 - Ahmed Gafsi
2051212 - Carrie Kam

Remis à George Salib

Hiver 2024

Table des matières

Introduction.....	5
1. Mesure de la qualité de conception de Timber.....	6
1.1 Mesure des métriques.....	6
1.1.1 Métriques de taille.....	6
1.1.2 Métriques de complexité.....	8
1.1.3 Métriques de complexité structurelle.....	9
1.1.4 Métriques d'héritage.....	11
1.1.5 Métriques de couplage.....	12
1.1.6 Métriques de cohésion.....	13
1.1.7 Métrique de documentation.....	15
1.2 Qualité globale.....	15
1.3 Métriques pour les critères ISO 9126.....	18
1.3.1 Réutilisabilité.....	18
1.3.2 Flexibilité.....	19
1.3.3 Compréhensibilité.....	19
1.3.4 Fonctionnalité.....	19
1.3.5 Extensibilité.....	20
1.3.6 Efficacité.....	20
1.4 Corrélation entre les métriques et les critères ISO 9126.....	21
1.4.1 Réutilisabilité.....	22
1.4.2 Flexibilité.....	22
1.4.3 Compréhensibilité.....	23
1.4.4 Fonctionnalité.....	23
1.4.5 Extensibilité.....	23
1.4.6 Efficacité.....	24
1.4.7 Exemple de l'analyse pratique de l'impact des métriques sur la qualité du système.....	25

2. Anomalies de Timber.....	27
2.1 Définition des seuils des métriques.....	28
2.2 Présentation des anomalies.....	29
2.2.1 Large Class.....	29
2.2.2 Switch Statements.....	30
2.2.3 Long Method.....	31
2.2.4 Long Parameter List.....	31
2.2.5 Feature envy.....	32
3. Refactoring.....	35
3.1 Correction des anomalies.....	35
3.1.1 Large Class.....	35
3.1.2 Switch Statements.....	40
3.1.3 Long Method.....	47
3.1.4 Long Parameter List.....	54
3.1.5 Feature envy.....	57
3.2 Métriques après refactoring.....	59
3.3 Critères du modèle ISO 9126 après refactoring.....	60
3.3.1 Réutilisabilité.....	60
3.3.2 Flexibilité.....	60
3.3.3 Compréhensibilité.....	61
3.3.4 Fonctionnalité:.....	61
3.3.5 Extensibilité.....	61
3.3.6 Efficacité.....	62
Conclusion.....	62
Références.....	64

Introduction

Ce document présente une analyse du système logiciel Timber, durant laquelle on a abordé des concepts clés en matière de qualité de conception, de détection d'anomalies et de refactoring. Ce travail se concentre sur l'analyse approfondie du projet Timber, utilisant diverses métriques pour évaluer sa qualité globale et identifier des améliorations potentielles à travers des techniques de refactoring.

L'objectif de ce travail est d'appliquer concrètement les connaissances théoriques acquises en cours, notamment celles liées aux critères de qualité ISO 9126 et aux méthodologies de détection d'anomalies. À travers un examen minutieux des différentes métriques comme la cohésion, la complexité, le couplage, la taille, et d'autres métriques spécifiques (NOM, LOC, LCOM, WMC, CC, CBO, etc.), nous avons pu évaluer la qualité du projet Timber et identifier des opportunités de refactoring pour améliorer sa conception.

Notre rapport se divise en plusieurs sections clés, débutant par une analyse de la qualité de conception et suivie par une exploration détaillée de différentes métriques. Nous examinons ensuite les anomalies spécifiques présentes dans Timber et proposons des stratégies de refactoring adaptées. Finalement, une réévaluation de la qualité du projet, post-refactoring, permet de mesurer l'impact de nos interventions. La compréhension et l'amélioration de la qualité logicielle sont essentielles pour le développement et la maintenance efficaces de systèmes logiciels. Ce travail pratique offre ainsi une occasion unique de mettre en pratique ces compétences essentielles, cruciales pour tout professionnel de l'architecture logicielle.

1. Mesure de la qualité de conception de Timber

Cette section contient les valeurs des différentes métriques calculées pour l'application Timber. Nous explorerons des métriques sur la cohésion, la complexité, le couplage et la taille. Les outils utilisés sont MetricsTree, MetricsReloaded et Understand.

1.1 Mesure des métriques

1.1.1 Métriques de taille

Pour déterminer la taille d'un système, on doit utiliser les métriques suivantes :

- **Nombre total de lignes de code (TLOC)**: Cette métrique mesure la taille globale du projet en comptant toutes les lignes de code, y compris les commentaires et les lignes vides. Cela donne une indication générale de la taille du code source.
- **Lignes de code sans commentaires (NCLOC)**: Cette métrique exclut les commentaires et les lignes vides, se concentrant uniquement sur les instructions de code réelles. Cela permet d'obtenir une mesure plus précise de la taille du code en éliminant les éléments non essentiels.
- **Lignes de code de méthode (MLOC)**: Cette métrique mesure le nombre de lignes de code dans une seule méthode ou fonction. Cela peut être utilisé pour évaluer la complexité et la taille des différentes parties du code.
- **Lignes de code de classe (CLOC)**: Cette métrique mesure le nombre de lignes de code dans une seule classe. Cela permet d'évaluer la taille et la complexité des classes individuelles dans le système.

En utilisant ces métriques, les développeurs peuvent avoir une meilleure compréhension de la taille du code, de sa structure et de sa complexité, ce qui peut aider à prendre des décisions concernant la maintenance, l'optimisation et l'évolution du logiciel.

Métrique	Plugiciel	Niveau de la métrique	Moyenne	Écart-type	Minimum	Maximum	Total
Total Lines of Code (TLOC)	Metrics Reloaded	Projet	N/A	N/A	N/A	N/A	30119
Non-Comment Lines of Code (NCLOC)	Metrics Reloaded	Projet	N/A	N/A	N/A	N/A	26714
Method Lines of Code (MLOC)	Metrics Reloaded	Méthodes	11,25	13.53	1	166	17498
Class Lines of Code (CLOC)	Metrics Reloaded	Classes	87,40	176,62	2	1983	20015

Tableau 1: Métriques de taille

Nous constatons que la métrique CLOC (Count Lines of Code) des classes de Timber a une moyenne de 87,41 et un maximum de 1983 lignes. Ainsi, une classe spécifique contient 1983 lignes de code. Une telle longueur est problématique, car elle diminue la compréhensibilité et la maintenabilité du code. Pour remédier à cette situation, il est recommandé de refactoriser cette classe en la divisant en plusieurs sous-classes, chacune ayant une responsabilité unique, conformément au principe SRP (Single Responsibility Principle).

En divisant cette classe en sous-classes distinctes, nous pouvons mieux organiser le code, le rendant plus lisible, plus facile à comprendre et à entretenir. Chaque sous-classe serait responsable d'une tâche spécifique, ce qui favoriserait la modularité et la réutilisabilité du code. De plus, cela permettrait de réduire la complexité de la classe initiale, facilitant ainsi sa gestion et son évolution future.

1.1.2 Métriques de complexité

La métrique SIZE2 évalue le nombre total d'attributs et de méthodes présents dans une classe, offrant ainsi une indication de sa complexité et de sa maintenabilité. D'autre part, la métrique NOM se concentre spécifiquement sur le nombre de méthodes définies dans une classe.

Une valeur élevée pour SIZE2 suggère qu'une classe pourrait être surchargée et nécessiterait d'être décomposée en classes plus petites et plus spécialisées. De même, une valeur élevée de NOM pourrait signaler une complexité excessive dans une classe, nécessitant sa subdivision en composants plus gérables et plus granulaires.

Métrique	Plugiciel	Niveau de la métrique	Moyenne	Écart-type	Minimum	Maximum	Total
Number of Local Methods (NOM)	Metrics Tree	Projet	6,8311	11,046	0	105	1537
Number of Attributes and Methods (SIZE2)	Metrics Tree	Projet	15,0177	23,43	1	211	3379

Tableau 2: Métriques de complexité

En calculant la métrique SIZE2 pour le projet Timber, nous avons obtenu une moyenne de 15,017 attributs et méthodes par classe. Cette moyenne suggère une potentielle absence de cohésion dans les classes, ce qui peut nuire à la clarté et à la maintenabilité du code. De plus, le maximum de 211 attributs et méthodes pour une classe soulève des préoccupations similaires quant à la cohésion de cette classe particulière. Il est donc nécessaire de réexaminer et de refactoriser cette classe en la divisant en sous-classes, chacune ayant une responsabilité unique. En général, une métrique SIZE2 élevée peut entraîner une diminution de la compréhensibilité, de la facilité de modification et de la testabilité du code.

Concernant la métrique NOM des classes de Timber, nous observons qu'en moyenne, une classe possède 6,8311 méthodes, tandis que la plus grande classe en compte 105. Cette disparité entre la valeur moyenne et le maximum suggère une potentielle violation du principe de responsabilité unique (SRP). Une conception efficace du système vise à limiter le nombre de méthodes dans une classe, ce qui favorise la clarté et la modularité du code. Ainsi, la présence de classes dépassant 105 méthodes est préoccupante et nécessite une révision et une éventuelle refonte pour améliorer la structure et la maintenabilité du code.

1.1.3 Métriques de complexité structurelle

Les métriques de complexité évaluent différentes caractéristiques du code afin de fournir une estimation globale de sa qualité. Elles mettent particulièrement l'accent sur les méthodes, qui jouent un rôle crucial dans la détermination de 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, ce qui rend le code plus ardu à maintenir. Pour notre analyse de la complexité structurelle, nous avons utilisé des métriques telles que la complexité cyclomatique (CC), le nombre de méthodes pondérées (WMC) et la réponse pour une classe (RFC). La complexité cyclomatique (CC) a été calculée au niveau des méthodes, tandis que le WMC et le RFC ont été évalués au niveau des classes.

Métrique	Plugiciel	Niveau de la métrique	Moyenne	Écart-type	Minimum	Maximum	Total
McCabe Cyclomatic Complexity (CC)	Metrics Reloaded	Méthodes	2,41	3,132	1	54	3815
Response for Class (RFC)	Metrics Reloaded	Classes	9,003	14,149	0	150	2323
Weighted Methods Count (WMC)	Metrics Reloaded	Classes	15,79	34,172	0	404	3617

Tableau 3: Métriques de complexité structurelle

En observant les résultats des métriques de complexité cyclomatique, on remarque que les valeurs maximales sont nettement supérieures aux moyennes. Cette métrique quantifie le nombre de chemins indépendants dans une méthode. En outre, la moyenne du nombre de méthodes pour la métrique CC est très faible (2,41), 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 atteint 54, ce qui est considérablement élevé par rapport à la moyenne. Pour résoudre ce problème, il est recommandé de subdiviser la méthode associée à cette valeur élevée (54) en plusieurs méthodes plus petites.

La métrique Weighted Methods per Class (WMC) évalue la complexité de chaque classe en sommant de manière pondérée la complexité de toutes ses méthodes, offrant ainsi une estimation de la complexité de la classe. La plage normale pour cette métrique devrait se situer entre 12 et 34[2]. Dans le cas de Timber, la moyenne de WMC est de 15,79, ce qui suggère un faible niveau de complexité pour les classes du projet. Cependant, la valeur maximale de cette métrique est de 404, ce qui indique qu'au moins une classe contient une ou plusieurs méthodes complexes nécessitant une refactorisation pour améliorer la lisibilité et la maintenabilité du code.

En ce qui concerne la métrique 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. Un seuil généralement accepté pour cette métrique est de 44[3]. Toute valeur dépassant ce seuil peut être considérée comme complexe et difficile à comprendre, à tester et à déboguer. Dans l'application Timber, la valeur moyenne pour le RFC est de 9,003, ce qui est encourageant. Cependant, la valeur maximale atteint 404, bien au-dessus du seuil de 44. Cela souligne la nécessité de refactoriser la classe en question pour améliorer la qualité du code.

En résumé, en identifiant les méthodes et les classes avec des valeurs de métriques élevées et en proposant des solutions de refactorisation spécifiques, nous pourrons améliorer la qualité et la maintenabilité du code de l'application Timber.

1.1.4 Métriques d'héritage

La classe racine est à la base, et les classes héritées s'étendent vers le haut comme des branches. La profondeur de l'arbre d'héritage (DIT) mesure la distance parcourue par une classe pour atteindre son ancêtre le plus éloigné. Plus la DIT est faible, plus la hiérarchie est simple et claire. Les classes héritent de fonctionnalités de base et se spécialisent légèrement. En revanche, une DIT élevée indique une complexité accrue. Naviguer et maintenir une telle structure devient difficile.

Le nombre d'enfants (NOC) s'intéresse à une classe spécifique et compte ses sous-classes directes. Comme le nombre d'enfants dans une famille, un NOC faible signifie une responsabilité bien définie pour la classe parente. Un NOC élevé peut créer un engorgement. La classe parente devient un point central complexe, gérant de nombreuses responsabilités et sous-classes.

Métrique	Plugin	Niveau de la métrique	Moyenne	Écart-type	Minimum	Maximum	Total
Depth of Inheritance Tree (DIT)	Metrics Tree	Classes	0.164	0.3833	0	2	37
Number of Children (NOC)	Metrics Tree	Classes	0.2	1.10	0	11	45

Tableau 4: Métriques d'héritage

On observe que, dans la hiérarchie de classes, il y a une distance moyenne d'une classe entre une classe dérivée et sa classe de base.

Pour la métrique NOC moyenne de 0,2 suggère qu'il y a moins d'une classe enfant directe pour chaque classe de base dans la hiérarchie de classes.

1.1.5 Métriques de couplage

La métrique CBO, ou Coupling Between Objects, évalue le degré de couplage entre les classes dans un système logiciel. Elle mesure le nombre de classes qui sont liées entre elles par des appels de méthodes ou par l'accès à des attributs. Un faible couplage entre les classes est généralement préférable, car il rend le système plus modulaire, flexible et facile à maintenir.

Lorsque la valeur de CBO est comprise entre 1 et 4, cela indique un faible niveau de couplage, ce qui est bénéfique pour la qualité du logiciel. Un faible couplage signifie que les classes sont moins dépendantes les unes des autres. Ce qui facilite la réutilisation du code, la modification et l'extension du système sans perturber les autres morceaux du code. En revanche, un couplage élevé peut rendre le code plus rigide, plus complexe et plus difficile à maintenir.

Métrique	Plugiciel	Niveau de la métrique	Moyenne	Écart-type	Minimum	Maximum	Total
Afferent Coupling (Ca)	Metrics Reloaded	Projet	90,92	186,85	0	886	2455
Coupling Between Objects (CBO)	Metrics Tree	Classes	6,091	7,1411	0	45	1395

Tableau 5: Métriques de couplage

La métrique CBO mesure le nombre de classes liées les unes aux autres, que ce soit par l'appel de méthodes ou l'accès à des attributs. Un niveau de couplage est considéré comme problématique lorsque la valeur de CBO dépasse 4[4]. Dans le cas du projet Timber, avec une moyenne de CBO d'environ 6,091, il est clair que le couplage entre les classes est trop élevé, ce qui peut rendre le code difficile à maintenir et à modifier.

Pour la métrique Ca, elle évalue le couplage entre les paquets externes qui utilisent les méthodes d'un paquet spécifique. Pour un projet comportant entre 101 et 1000 classes, une valeur de Ca comprise entre 2 et 20 est considérée comme normale[5]. Une valeur de Ca dépasse 20 montre un couplage élevé entre les classes. Dans le cas de Timber, on a une moyenne de Ca de 90,92, il est clair que les paquets présentent un niveau de couplage élevé, rendant l'application difficile à modifier.

De plus, la valeur maximale de Ca est 2455 qui est alarmante, car elle indique que des modifications apportées au paquet associé auront un impact significatif sur de nombreux autres paquets, en raison de leur interdépendance. Cette situation souligne la nécessité de réduire le couplage entre les classes et les paquets afin d'améliorer la flexibilité et la maintenabilité de l'application Timber.

1.1.6 Métriques de cohésion

Les mesures de cohésion évaluent la relation entre les différents éléments au sein d'un module logiciel, tels que les méthodes et les attributs d'une classe. Elles examinent comment ces éléments interagissent entre eux. Une cohésion élevée signifie qu'il existe une forte corrélation entre les méthodes et les attributs, ce qui indique que la classe est responsable d'une seule fonctionnalité spécifique et ne nécessite qu'une modification limitée.

Une classe présentant une forte cohésion est considérée comme souhaitable, car elle est plus compréhensible, réutilisable et modifiable. Cela signifie qu'elle est mieux organisée et plus facile à maintenir. L'évaluation de la cohésion peut se faire de différentes manières, mais la mesure la plus couramment utilisée est la mesure de manque de cohésion entre les méthodes d'une classe (LCOM).

Dans le système Timber, les résultats de l'évaluation de la LCOM au niveau des classes sont présentés dans un tableau, fournissant ainsi une indication de la cohésion des différentes classes du système. Ces résultats permettent aux développeurs d'identifier les classes qui pourraient nécessiter une refonte pour améliorer leur cohésion et leur maintenabilité.

Métrique	Plugiciel	Niveau de la métrique	Moyenne	Écart-type	Minimum	Maximum	Total
Lack Of Cohesion of Methods (LCOM)	Metrics Reloaded	Classes	2,104	2,378	0	21	482
Tight Class Cohesion (TCC)	Metrics Reloaded	Classes	0,173	0,266	0	1	39,11

Tableau 6: Métriques de cohésion

L'analyse du LCOM (Lack of Cohesion of Methods) permet d'évaluer la cohésion des méthodes dans une classe. L'outil Metrics Reloaded a permis d'obtenir une moyenne de LCOM de 2,104 et un maximum de 21. Le LCOM mesure le nombre de méthodes dans une classe qui n'utilisent pas d'attributs d'instance communs. En général, une valeur de LCOM proche de zéro indique une forte cohésion, tandis qu'une valeur supérieure à 1 indique une cohésion plus faible. Dans notre cas, la valeur moyenne de LCOM est supérieure à 1, ce qui suggère un manque de cohésion dans les classes de Timber. Ce manque de cohésion peut rendre les classes plus difficiles à comprendre et à maintenir. Cette conclusion est cohérente avec celle tirée de l'analyse des métriques de taille, qui indiquait une complexité élevée.

L'analyse du TCC (Tight Class Cohesion) vise à évaluer le nombre de paires de méthodes qui accèdent aux attributs directement dans une classe. Avec l'outil Metrics Reloaded, nous obtenons une moyenne de 0,173. Dans le cas de la métrique TCC, plus que le nombre s'approche de 1, plus que nous avons une cohésion maximale. Cependant, puisque la moyenne se rapproche davantage de 0 que de 1, cela révèle une faible cohésion au sein des classes, comme le démontre dans l'analyse du LCOM.

1.1.7 Métrique de documentation

Les mesures de documentation évaluent la proportion de documentation au sein d'un module logiciel. Une documentation élevée signifie que de l'effort a été mis afin de présenter le contexte, les métadonnées et le fonctionnement du code. Cela aide généralement la compréhension du code et de ce fait même la maintenabilité du système.

Métrique	Plugiciel	Niveau de la métrique	Moyenne	Écart-type	Minimum	Maximum	Total
Comment Density (CD)	cloc	Projet	N/A	N/A	N/A	N/A	0.128

Tableau 7: Métrique de documentation

La métrique CD permet de comparer la quantité de commentaires avec la quantité de ligne de code. Ici, une valeur de 0,128 indique que pour chaque 100 lignes de code, il y aura environ 12,8 lignes de commentaires. En considérant les bonnes pratiques de codage tel que l'écriture de docstring pour les classes et méthodes, ce montant semble assez faible. Cette métrique indique donc un manquement au niveau de la documentation. Ce manquement indique la présence de code non documenté au sein de Timber. Cette réalité nuit à la compréhension du code et de ce fait à sa maintenabilité.

1.2 Qualité globale

Afin de jauger l'impact de la qualité du système sur le développement et la maintenance du système, il faut d'abord déterminer les critères permettant de l'évaluer. Sous la perspective de la maintenance, plusieurs facteurs influencent la qualité du système: la testabilité, sa facilité d'analyse et sa capacité de modification du système [2].

La testabilité du système détermine la facilité de celui-ci à être testé afin d'identifier et de résoudre des défauts. Cette caractéristique est intrinsèquement liée à la complexité du code formant le système et l'interdépendance des modules. L'habileté d'analyse du système fait référence à la capacité de celui-ci d'être examiné. Celle-ci dépend de la complexité du système et de la documentation du système. Finalement, la modifiabilité du système se décrit par le degré de difficulté associé à la modification ou l'extension d'une fonctionnalité du système. L'évaluation de ce facteur est associée à la complexité du système.

Ainsi, une évaluation de la qualité du système pour la maintenance peut se simplifier à une évaluation des métriques liées aux trois critères: la complexité, l'interdépendance des modules (cohésion + couplage) et la documentation.

Dans la section de la mesure des métriques, nous avons fait une analyse approfondie des métriques de complexité. Résumons leurs impacts pour la maintenabilité du système. Selon la métrique NOM, la majorité des classes possèdent un nombre raisonnable de méthodes. Cependant, il existe aussi des classes ayant un nombre absurde (105) méthodes. Cela indique une violation du principe SRP et insinue la présence d'une complexité au sein du système. Puis, la métrique SIZE2 démontre la même réalité que la présence de classes très complexes avec beaucoup d'attribut et méthodes. Les autres métriques de complexités telles que CC, RFC et WMC démontrent la même chose. En moyenne, la complexité du système est acceptable. Néanmoins, il existe des cas extrêmes au sein du système qui est très complexe. L'existence de ces cas extrêmes nuit à la maintenabilité du système puisque la compréhension, la testabilité et la facilité de modification du système diminuent

proportionnellement à la complexité du système. Dans un autre ordre d'idée, les métriques de tailles sont également liées à la complexité du système, puisque les grosses classes et modules sont généralement plus complexes que des petites classes et modules. De ce fait, l'analyse des métriques MLOC et CLOC pourrait aussi être utilisée afin d'estimer la complexité du système. Or, L'analyse de ces métriques montrent la même réalité que les métriques NOM et SIZE2, soit une moyenne acceptable pour la majorité des classes, mais il existe quelques classes extrêmes très grosses et de ce fait complexes.

Dans le même ordre d'idée, dans la section de la mesure des métriques, nous avons également fait une étude approfondie des métriques de couplage et cohésion. L'analyse des métriques CA et CBO possède des valeurs élevées. Timber possède donc un haut niveau de couplage entre ces classes et ces paquetages. Un haut taux de couplage peut avoir plusieurs conséquences sur la maintenance. Tout d'abord, le système sera plus difficile à tester à cause des dépendances externes. Puis, les modifications à un module seront plus difficiles puisqu'il a le potentiel d'impacter le fonctionnement de plusieurs autres modules. Puis, toujours tirées de la section de la mesure des métriques, les valeurs des métriques de cohésion LCOM et TCC indiquent un manque de cohésion général pour les classes de Timber. Un manque de cohésion nuit à la compréhensibilité du système, augmente généralement la complexité de celui-ci et rend la modification d'une classe plus difficile. Tous ces impacts nuisent directement à la maintenabilité du système.

Par après, toujours dans la section de la mesure des métriques, nous avons également analysé la métrique CD qui indique un manquement dans la documentation du code avec un taux de 0,128 ligne de commentaires par ligne de code. Ce manque de documentation peut avoir des impacts néfastes au niveau de la maintenabilité. Notamment, il sera plus difficile à comprendre lorsqu'on voudra éventuellement y retoucher, rendant plus difficiles la compréhension et la modification du système.

1.3 Métriques pour les critères ISO 9126

Les critères ISO 9126 présentent différents attributs (réutilisabilité, flexibilité, compréhensibilité, fonctionnalité, extensibilité, efficacité). Pour le projet Timber, les valeurs pour les différentes qualités d'attributs calculées par les plugins MetricsTree et MetricsReloaded sont :

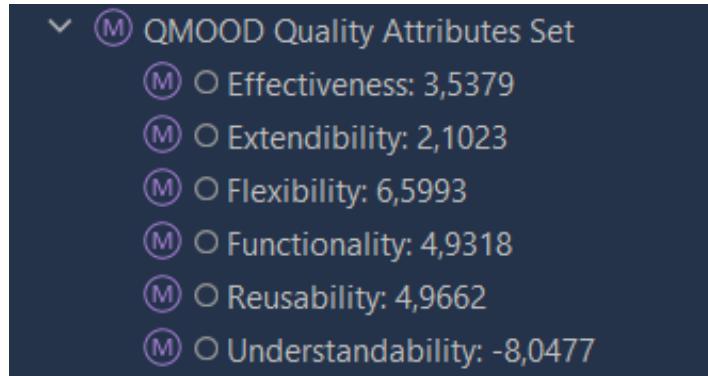


Figure 1: Valeurs des attributs de qualité QMOOD

1.3.1 Réutilisabilité

La réutilisabilité indique une adaptation aisée de la conception de nouveaux problèmes grâce à des caractéristiques de conception orientée objet. La valeur attribuée à la réutilisabilité dans le projet est de 4,9662.

Cette valeur est assez élevée au-dessus de zéro ce qui signifie donc que le système est globalement assez réutilisable. On peut ainsi supposer que ses classes et ses méthodes peuvent être utilisées afin de développer d'autres fonctionnalités ou d'autres projets, sans devoir réécrire beaucoup de code. Cela est dû entre autres à une forte cohésion et un faible couplage, ainsi qu'à la taille du projet. La qualité du projet est en conséquence élevée du point de vue de la réutilisabilité.

1.3.2 Flexibilité

La flexibilité mesure la capacité d'intégration des changements dans une conception et sa faculté à fournir des fonctionnalités connexes. Dans le projet, la valeur de la flexibilité est de 6,5993.

Cette valeur est également plutôt élevée, grâce au faible couplage, ainsi qu'à l'utilisation de l'encapsulation et du polymorphisme dans le projet. Cela implique donc que le système est flexible et peut être modifié ou agrandi sans trop d'efforts.

1.3.3 Compréhensibilité

La compréhensibilité évalue la capacité à apprendre et à comprendre la conception. Cette métrique est liée à la complexité de la structure de la conception. Dans le projet, la valeur relevée était de -8,0477.

Bien que cette valeur semble très basse, il est normal d'observer des valeurs de compréhensibilité négatives (on pourra constater cela dans la formule de calcul plus loin), surtout dans de larges projets comme Timber. En effet, la taille du projet, ainsi que sa complexité, l'utilisation du polymorphisme et l'utilisation d'abstractions contribuent toutes à diminuer la valeur de la compréhensibilité. Cette valeur pourrait être améliorée pour le projet Timber en réduisant certaines des métriques mentionnées et augmentant la cohésion et l'utilisation de l'encapsulation dans le projet.

1.3.4 Fonctionnalité

La fonctionnalité regarde les responsabilités attribuées aux classes d'une conception, qui sont rendues disponibles par les classes à travers leurs interfaces publiques. Dans le projet, la valeur de la fonctionnalité est de 4,9318.

C'est une valeur relativement élevée qui est liée à la taille du projet, aux tailles des interfaces, à l'utilisation du polymorphisme ainsi qu'à la forte cohésion. On peut donc dire que la qualité de Timber est élevée en termes de fonctionnalité et que ses classes ont généralement des responsabilités bien définies.

1.3.5 Extensibilité

L'extensibilité se réfère à l'aptitude d'une conception à intégrer de nouvelles exigences grâce à des attributs déjà présents. Dans le projet, la valeur de l'extensibilité est de 2,1023.

Cette valeur semble un peu basse, surtout quand on la compare aux autres, ce qui pourrait être lié à du couplage élevé ou bien à un manque d'utilisation d'héritage, d'abstraction et de polymorphisme. Le système est donc peu extensible.

1.3.6 Efficacité

L'efficacité mesure la capacité d'une conception à atteindre les fonctionnalités et les comportements souhaités grâce à des concepts et techniques de conception orientée objet. Dans le projet, la valeur de l'efficacité est de 3,5379.

C'est une valeur quand même élevée, mais pas dans les plus hautes pour ce projet. Elle révèle donc une utilisation adéquate de techniques orientées-objet dans l'application Timber.

En gros, la qualité du système est assez élevée, avec une bonne réutilisabilité, flexibilité, fonctionnalité et efficacité. Sa compréhensibilité et son extensibilité pourraient cependant être améliorées afin d'augmenter la qualité globale de la conception de Timber.

1.4 Corrélation entre les métriques et les critères ISO 9126

Dans cette section, nous explorons la corrélation entre diverses métriques et les critères de qualité définis par la norme ISO 9126, en se concentrant sur l'évaluation de la qualité d'ensemble du système. Nous présenterons les formules de calcul et détaillerons leurs différents composants. Ces formules proviennent du tableau ci-joint, extrait des notes de cours:

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

Figure 2: Formules pour calculer les attributs de qualité[7]

De plus, les métriques appliquées dans les formules sont détaillées dans le tableau ci-dessous:

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

Figure 3: Métriques de conception pour les propriétés de conception[7]

1.4.1 Réutilisabilité

Formule:

$$- 0.25 \times \text{Couplage} + 0.25 \times \text{Cohésion} + 0.5 \times \text{Messagerie} + 0.5 \times \text{Taille de conception}$$

La réutilisabilité est cruciale pour maximiser l'efficacité du développement. Un système à couplage faible (DCC) avec une forte cohésion (CAM), une communication efficace (CIS) et une taille de conception optimale (DSC) se caractérise par une modularité qui facilite la réutilisation des composants dans différents contextes, réduisant ainsi le temps et les coûts de développement.

1.4.2 Flexibilité

Formule:

$$0.25 \times \text{Encapsulation} - 0.25 \times \text{Couplage} + 0.5 \times \text{Composition} + 0.5 \times \text{Polymorphisme}$$

La flexibilité d'un système est essentielle pour s'adapter rapidement aux changements de besoins. L'encapsulation (DAM) assure la séparation des préoccupations, la composition (MOA) permet une combinaison flexible des composants et le polymorphisme (NOP) offre une adaptabilité dans les interactions. Le couplage (DCC) réduit cette flexibilité en liant étroitement les composants, entravant les modifications.

1.4.3 Compréhensibilité

Formule:

$$\begin{aligned} & - 0.33 \times \text{Abstraction} + 0.33 \times \text{Encapsulation} - 0.33 \times \text{Couplage} + 0.33 \times \text{Cohésion} \\ & - 0.33 \times \text{Polymorphisme} - 0.33 \times \text{Complexité} - 0.33 \times \text{Taille de Conception} \end{aligned}$$

Un système compréhensible est plus facile à maintenir et à étendre. L'abstraction (ANA) excessive peut rendre le système conceptuellement dense, tandis que le couplage élevé (DCC) et la complexité (NOM) le rendent difficile à déchiffrer. La cohésion (CAM), l'encapsulation (DAM) et une taille de conception maîtrisée (DSC) clarifient la structure et la logique du système.

1.4.4 Fonctionnalité

Formule:

$$0.12 \times \text{Cohésion} + 0.22 \times \text{Polymorphisme} + 0.22 \times \text{Messagerie} + 0.22 \times \text{Taille de Conception} + 0.22 \times \text{Hiérarchies}$$

La fonctionnalité est améliorée par une structure bien organisée. Une cohésion élevée (CAM) signifie que les composants sont bien définis et ciblés. Le polymorphisme (NOP) permet une flexibilité fonctionnelle, tandis que la communication efficace (CIS) et une hiérarchie bien conçue (NOH) facilitent l'intégration des fonctionnalités. La taille de conception (DSC) équilibrée assure la maniabilité des fonctionnalités.

1.4.5 Extensibilité

Formule:

$$0.5 \times \text{Abstraction} - 0.5 \times \text{Couplage} + 0.5 \times \text{Héritage} + 0.5 \times \text{Polymorphisme}$$

L'extensibilité est la capacité d'un système à évoluer avec de nouvelles exigences. L'abstraction élevée (ANA) et l'héritage (MFA) favorisent l'ajout de nouvelles fonctionnalités sans perturber le système existant. Le polymorphisme (NOP) permet l'extension des fonctionnalités existantes. Toutefois, un couplage élevé (DCC) peut limiter cette extensibilité en rendant difficile l'intégration de nouveaux modules.

1.4.6 Efficacité

Formule:

$$0.2 \times Abstraction + 0.2 \times Encapsulation + 0.2 \times Composition + 0.2 \times Héritage \\ + 0.2 \times Polymorphisme$$

L'efficacité se traduit par la capacité du système à accomplir ses fonctions avec un minimum de gaspillage de ressources. Chaque aspect de cette formule contribue à une utilisation efficace des ressources : l'abstraction (ANA) pour la généralisation, l'encapsulation (DAM) pour la protection des données, la composition (MOA) pour l'assemblage flexible, l'héritage (MFA) pour la réutilisation du code et le polymorphisme (NOP) pour la flexibilité d'interaction.

Ces analyses montrent que l'équilibre et l'interaction entre différentes propriétés de conception sont fondamentaux pour atteindre un niveau de qualité élevé dans un système informatique. En ajustant ces métriques, on peut optimiser de façon considérable les caractéristiques globales du système.

1.4.7 Exemple de l'analyse pratique de l'impact des métriques sur la qualité du système

Premier exemple: compréhensibilité et nombre de méthodes (NOM)

Dans ce premier cas, nous examinons l'influence du Nombre de Méthodes (NOM) dans une classe sur la compréhensibilité. Prenons l'exemple de la classe *MusicService.java* qui compte initialement 105 méthodes. Avec cette complexité, la compréhensibilité du système est faible (-8.0477). Cette faible compréhensibilité peut être attribuée à la complexité des classes, souvent évaluée par leur nombre de méthodes. En supprimant une méthode (*scrobble*) de 8 lignes de code, la compréhensibilité passe de -8.0477 à -8.036. Bien que minime, cette modification illustre l'effet du NOM sur la compréhensibilité du système.

Class: MusicService				
Metric	Metrics Set	Description	Value	Regular Ra...
○ NOM	Li-Henry M...	Number Of Methods	105	[0..7)

Figure 4: Nombre de méthode de la classe MusicService

```
void scrobble() {
    if (LastfmUserSession.getSession(this).isLogedin()) {
        Log.d("Scrobble", "to LastFM");
        String trackname = getTrackName();
        if (trackname != null)
            LastFmClient.getInstance(this).Scrobble(new ScrobbleQuery(getArtistName(), trackname, timestamp: System.currentTimeMillis() / 1000));
    }
}
```

Figure 5: Code de la méthode Scrobble

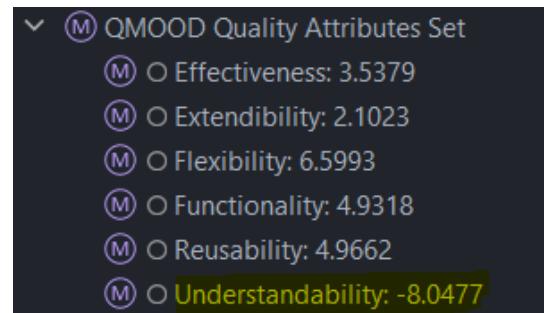


Figure 6: Valeur de l'Understandability sans la méthode Scrobble

Class: MusicService				
Metric	Metrics Set	Description	Value	Regular Ra...
○ NOM	Li-Henry M...	Number Of Methods	104	[0..7)

Figure 7: Nombre de méthode de la classe MusicService après la suppression de la méthode Scrobble

Deuxième exemple: couplage et extensibilité

Ensuite, nous analysons l'impact du couplage (mesuré par la métrique CBO) sur différents critères de qualité. En particulier, un couplage élevé nuit à l'Extensibilité, comme illustré par les classes *MusicService.java* et *MainActivity.java* avec des valeurs CBO de 31 et 21, respectivement (au-dessus de la plage idéale de 0-15). Cette situation contribue à une faible valeur d'extensibilité dans le système, confirmant l'effet négatif du couplage sur ce critère.

Troisième exemple: polymorphisme et qualité globale

Enfin, nous étudions la métrique NOP (Nombre de Méthodes Polymorphiques), calculée par $NOP = NOM \times PF$ (Facteur de Polymorphisme). Pour notre système, avec un PF de 25,1736 %, et 1538 méthodes totales, nous obtenons environ 387 méthodes polymorphiques. Le polymorphisme a un impact positif sur la Flexibilité, la Fonctionnalité, l'Extensibilité et l'Efficacité. Malgré des valeurs pas exceptionnellement élevées pour ces critères, elles sont positives, ce qui est encourageant. La Flexibilité, influencée à 50 % par le polymorphisme, montre une valeur appréciable (6.5993), attribuable en grande partie à la nature polymorphe de nombreuses méthodes.

Project: main				
M...	Metrics Set	Description	Value	Regular Range
○ MIF	MOOD Metrics Set	Method Inheritance Factor	14.3580%	[60.9000%..84.4000%)
○ PF	MOOD Metrics Set	Polymorphism Factor	25.1736%	[1.7000%..15.1000%)

Figure 8: Facteur de Polymorphisme du système Timber

Cependant, un PF élevé peut poser des problèmes de compréhensibilité, comme le montre le critère de Compréhensibilité bas du système. En effet, un excès de polymorphisme, malgré ses avantages, peut complexifier la compréhension du système, corroborant l'effet négatif du polymorphisme sur la Compréhensibilité selon l'équation associée.

2. Anomalies de Timber

Cette section présente et catégorise cinq anomalies détectées dans le code de Timber, en les alignant sur divers critères. Chaque anomalie est analysée et justifiée en se basant sur un ensemble de métriques spécifiques, puis qualifiée en termes de code smells. Ainsi, cette section fusionne les réponses aux questions 2 et 3, traitant simultanément de l'identification et de la classification des anomalies pour chacune d'entre elles.

2.1 Définition des seuils des métriques

Selon le livre *Object Oriented Metrics in Practice* [6], il est envisageable de repérer certains anti-patrons de code en se référant aux métriques. Pour ce faire, il suffit de définir des seuils sur ces métriques en associant des conditions appropriées, ce qui permettra de détecter les anti-patrons. Dans notre analyse, ces seuils sont classés en trois catégories : faible (**LOW**), élevé (**HIGH**), et très élevé (**VERY HIGH**). Pour établir ces seuils, dans le cadre des statistiques, deux variables sont utilisées : la moyenne (AVG) et l'écart-type (STDEV). Les formules suivantes sont employées pour établir l'intervalle de la métrique :

$$\text{LOW} = \text{AVG} - \text{STDEV}$$

$$\text{HIGH} = \text{AVG} + \text{STDEV}$$

$$\text{VERY HIGH} = (\text{AVG} + \text{STDEV}) * 1.5$$

Pour le seuil faible, si le résultat est négatif, on retient la valeur 0. Dans le cas d'un écart extrême, les valeurs seront considérées comme très élevées si elles dépassent de 50 % le seuil élevé.

Voici les valeurs calculées pour les seuils des métriques:

Métrique	Moyenne	Écart-type	LOW	HIGH	VERY HIGH
LCOM	2,104	2,378	0	4,482	6,723
TCC	0,173	0,266	0	0,439	0,6585
WMC	15,79	34,172	0	49,962	74,943
CC (méthode)	2,41	3,132	0	5,542	8,313
RFC	9,003	14,149	0	23,152	34,728
CBO	6,091	7,1411	0	77,502	116,253
Ca	90,92	186,65	0	277,57	416,355
MLOC	11,25	13,53	0	24,78	37,17
NOM	6,8311	11,046	57,265	79,357	119,0355
SIZE2	15,0177	23,43	126,747	173,607	260,4105

Tableau 8: Seuils des métriques pour Timber

Dans les cas relatifs, on utilise ces seuils:

NONE = 0

ONE = 1

FEW = 2 - 5

Short Memory Cap = 7

MANY > 7

2.2 Présentation des anomalies

2.2.1 Large Class

Fichier: MusicService.java

La première anomalie identifiée est classée comme *Large Class* selon la formule suivante:

$$((ATFD > FEW) \text{ AND } (WMC \geq \text{VERY HIGH}) \text{ AND } (TCC < \text{ONE THIRD}))$$

En appliquant cette formule à notre classe, nous obtenons :

$$(6 > 5) \text{ AND } (467 \geq 74,943) \text{ AND } (0.2042 < 0.33333)$$

Les trois conditions sont remplies, indiquant que le fichier *MusicService.java* contient une anomalie due à une classe trop volumineuse. Le nombre de classes auxquelles *MusicService* accède (ATFD) est supérieur à la constante FEW. En outre, la somme pondérée des méthodes de la classe (WMC) dépasse largement le seuil *VERY HIGH* spécifié. Enfin, la cohésion entre les méthodes et les responsabilités de la classe (TCC) est de 0.2042, ce qui est considéré comme faible et inférieur au seuil de 0.33333.

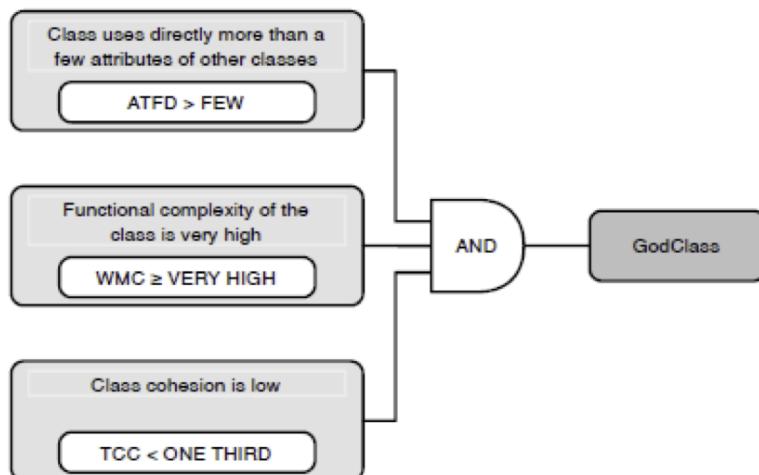


Figure 9: Stratégie de détection “Large Class”[7]

2.2.2 Switch Statements

Méthode: onTouchEvent(MotionEvent event)

Fichier: CircularSeekBar.java

L'analyse de la méthode *onTouchEvent* révèle deux problèmes potentiels: l'utilisation excessive d'instructions *switch*, ainsi une complexité élevée.

L'utilisation excessive d'instructions *switch* peut rendre le code difficile à lire, à maintenir et à étendre. Dans ce cas, la méthode *onTouchEvent* utilise une instruction *switch* pour gérer différents états de l'événement *MotionEvent*. Cette utilisation d'instructions *switch* pourrait être simplifiée et rendue plus flexible en utilisant une approche basée sur des classes et des méthodes polymorphiques, ou en utilisant des patterns comme le pattern State.

La métrique *McCabe Cyclomatic Complexity* (MCC) évalue le nombre de chemins indépendants dans une fonction. Plus la valeur de MCC est élevée, plus la fonction est complexe. La valeur CYCLO/Line of Code pour la méthode *onTouchEvent* est de 168, et une complexité cyclomatique de 35 qui se situe entre les niveaux VERY HIGH du tableau 8. Cela indique que la fonction est effectivement très complexe.

2.2.3 Long Method

Méthode: processTouchEvent()

Fichier: ViewDragHelper.java.

La troisième anomalie identifiée correspond à une méthode trop longue (*Long Method*).

Pour la classifier dans cette catégorie, nous utilisons la formule suivante:

$(LOC > (HIGH (Class) / 2)) \text{ ET } (CYCLO \geq HIGH) \text{ ET } (MAXNESTING} \geq SEVERAL \\ \text{ET } (NOAV} \geq MANY)$

En appliquant cette formule avec les valeurs spécifiques à notre classe, cela donne :

$$(134 > (24.2906386 / 2)) ET (27 \geq 1.1860605) ET (2 \geq 2) ET (47 > 7)$$

Dans le fichier ViewDragHelper.java, la méthode processTouchEvent() remplit toutes les conditions de cette formule, ce qui indique clairement que cette méthode est excessivement longue. Elle compte 134 lignes (LOC). Son niveau d'imbrication maximal (MAXNESTING) est de 2. Sa complexité cyclomatique de McCabe, qui évalue le nombre de chemins indépendants dans la fonction, est de 27. Enfin, elle accède à 27 variables différentes au sein de la méthode (NOAV).

2.2.4 Long Parameter List

Méthode: playAll()

Fichier: BaseSongAdapter.java.

Cette anomalie est classée sous le type « Longue Liste de Paramètres » (*Long Parameter List*). Pour la classifier ainsi, nous nous référerons à la figure correspondante:

Method: playAll(Activity, long[], int, long, IdType, boolean, Song, boolean)				
Metric	Metrics Set	Description	Value	Regular Ran...
○ CND		Condition Nesting Depth	2	[0..2)
○ LND		Loop Nesting Depth	0	[0..2)
○ CC		McCabe Cyclomatic Complexity	4	[0..3)
○ NOL		Number Of Loops	0	
○ LOC		Lines Of Code	18	[0..11)
○ NOPM		Number Of Parameters	8	[0..3)
○ HVL	Halstead ...	Halstead Volume	235.022	
○ HD	Halstead ...	Halstead Difficulty	15.6818	
○ HL	Halstead ...	Halstead Length	50	
○ HEF	Halstead ...	Halstead Effort	3685.5721	
○ HVC	Halstead ...	Halstead Vocabulary	26	
○ HER	Halstead ...	Halstead Errors	0.0795	
○ MMI	Maintainability Index	Maintainability Index	55.8288	[0.0..19.0]

Figure 10: Métriques pour la méthode playAll()

Dans *BaseSongAdapter.java*, la fonction *playAll()* a huit paramètres, comme indiqué par sa valeur NOPM (Number Of Parameters Metric) de huit. Parmi ces huit paramètres, trois ne sont pas utilisés. La norme généralement acceptée est de ne pas dépasser trois paramètres par fonction, or cette fonction dépasse presque le triple de cette limite.

2.2.5 Feature envy

Méthode: *getControlPointsFor()*

Fichier: *NumberUtils.java*

La dernière anomalie relevée dans l'application est de type *Feature Envy*, ce qui signifie une méthode qui utilise davantage de membres d'une autre classe que de la sienne.

Elle peut être repérée en utilisant la formule suivante:

$(ATFD \geq FEW) \text{ ET } (FDP \geq FEW) \text{ ET } (LAA < ONE\ THIRD)$

Ce qui donne pour nous:

$(ATFD \geq 5) \text{ ET } (FDP \geq 5) \text{ ET } (LAA < 0,33)$

Metric	Value	Excess
Access To Foreign Data	11	+6
Coupling Between Objects	13	-
Data Abstraction Coupling	0	-
Depth Of Inheritance Tree	1	-
Halstead Difficulty	12,0	-
Halstead Effort	4250,9...	-
Halstead Errors	0,0875	-
Halstead Length	68	-
Halstead Vocabulary	27	-

Figure 11: Les métriques de la classe *NumberUtils*

Dans la méthode `getControlPointsFor()` de la classe `NumberUtils.java`, on peut observer les valeurs suivantes pour les métriques **ATFD**, **FDP** et **LAA**.

$$\mathbf{ATFD} = 11$$

$$\mathbf{FDP} = 11$$

$$\mathbf{LAA} = 1.0$$

En effet, 11 propriétés venant de 11 autres classes sont utilisées dans cette méthode, tandis qu'aucune propriété de sa propre classe (`NumberUtils`) n'est utilisée. Le facteur LAA se calcule donc comme cela:

$$\frac{\text{Nombre d'attributs de la classe de la méthode utilisée}}{\text{Nombre d'attributs utilisés au total}} \\ = \frac{11}{11} \\ = 1$$

Une méthode qui utilise trop de membres d'autres classes peut être problématique, car cela signifie qu'elle est soit définie dans la mauvaise classe, ou bien qu'elle est trop complexe et qu'elle sera difficile à modifier. Il est donc préférable de la déplacer ou d'en extraire certaines parties afin de la simplifier.

3. Refactoring

3.1 Correction des anomalies

3.1.1 Large Class

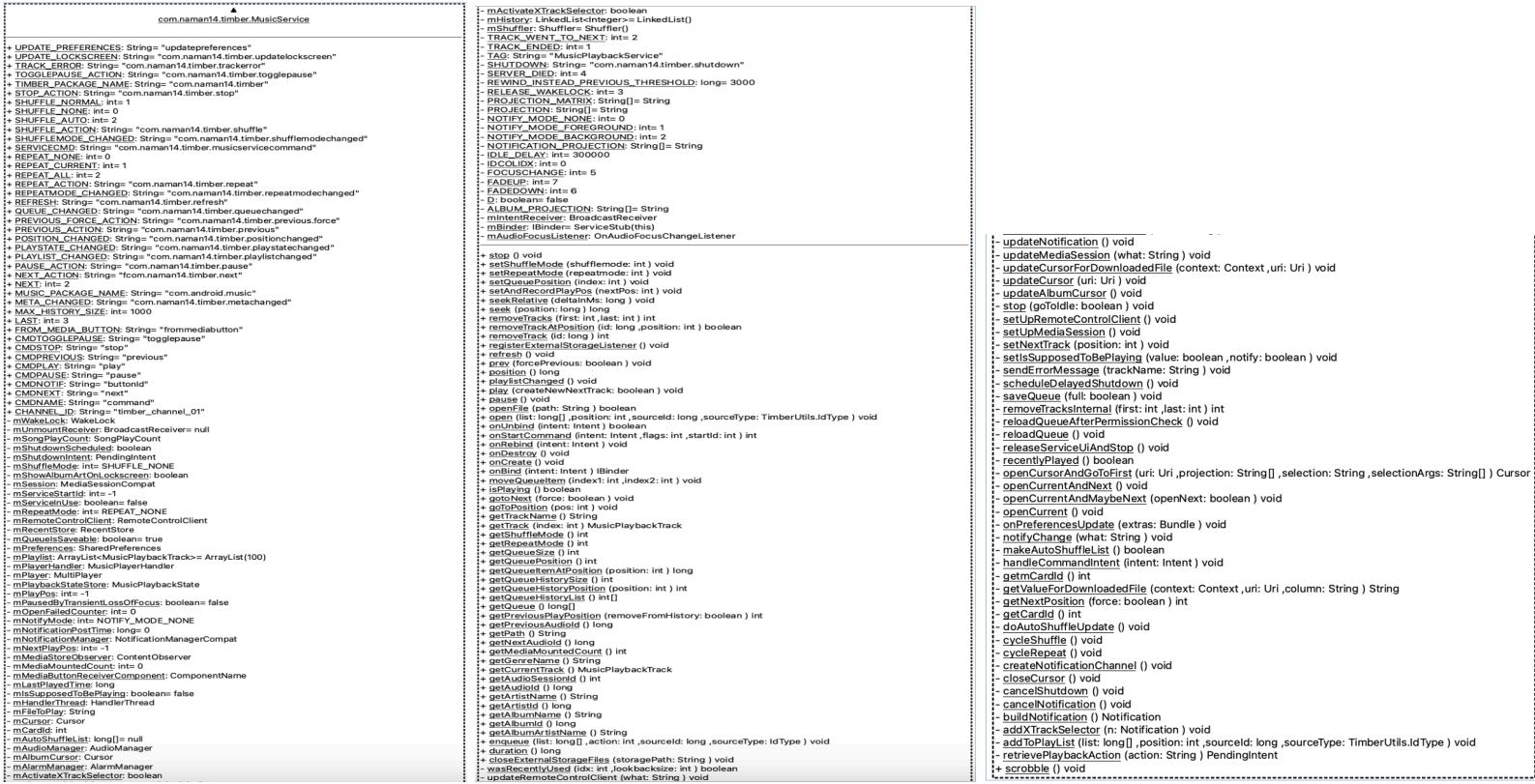


Figure 12: Diagramme de classe de la classe MusicService (étalé horizontalement, car de très grande taille)

D'après les images précédentes, la classe MusicService est particulièrement volumineuse, avec 2864 lignes, ce qui en fait une anomalie de type Large Class. Pour corriger cela, il est conseillé de procéder à un refactoring en employant diverses techniques visant à diviser le code en sous-composants plus petits et gérables. La première étape consistera à créer une interface qui regroupe les différentes constantes, comme illustrées dans la figure.

```

interface Constants {
    static final String PLAYSTATE_CHANGED = "com.naman14.timber.playstatechanged";
    static final String POSITION_CHANGED = "com.naman14.timber.positionchanged";
    static final String META_CHANGED = "com.naman14.timber.metachanged";
    static final String QUEUE_CHANGED = "com.naman14.timber.queuechanged";
    static final String PLAYLIST_CHANGED = "com.naman14.timber.playlistchanged";
    static final String REPEATMODE_CHANGED = "com.naman14.timber.repeatmodechanged";
    static final String SHUFFLEMODE_CHANGED = "com.naman14.timber.shufflemodechanged";
    static final String TRACK_ERROR = "com.naman14.timber.trackerror";
    static final String TIMBER_PACKAGE_NAME = "com.naman14.timber";
    static final String MUSIC_PACKAGE_NAME = "com.android.music";
    static final String SERVICECMD = "com.naman14.timber.musicservicecommand";
    static final String TOGGLEPAUSE_ACTION = "com.naman14.timber.togglepause";
    static final String PAUSE_ACTION = "com.naman14.timber.pause";
    static final String STOP_ACTION = "com.naman14.timber.stop";
    static final String PREVIOUS_ACTION = "com.naman14.timber.previous";
    static final String PREVIOUS_FORCE_ACTION = "com.naman14.timber.previous.force";
    static final String NEXT_ACTION = "com.naman14.timber.next";
    static final String REPEAT_ACTION = "com.naman14.timber.repeat";
    static final String SHUFFLE_ACTION = "com.naman14.timber.shuffle";
    static final String FROM_MEDIA_BUTTON = "frommediabutton";
    static final String REFRESH = "com.naman14.timber.refresh";
    static final String UPDATE_LOCKSCREEN = "com.naman14.timber.updatelockscreen";
    static final String CMDNAME = "command";
    static final String CMDTOGGLEPAUSE = "togglepause";
    static final String CMDSTOP = "stop";
    static final String CMDPAUSE = "pause";
    static final String CMDPLAY = "play";
    static final String CMDPREVIOUS = "previous";
    static final String CMDNEXT = "next";
    static final String CMDNOTIE = "buttonId";
    static final String UPDATE_PREFERENCES = "updatepreferences";
    static final String CHANNEL_ID = "timber_channel_01";
    static final int NEXT = 2;
    static final int LAST = 3;
}

```

Figure 13: L'interface Constante

Cette interface regroupe toutes les variables constantes. Non seulement cette approche réduira la taille de la classe MusicService, mais elle améliorera également sa maintenabilité. Il sera plus aisément d'ajouter, de modifier ou de retirer une ou plusieurs constantes.

La seconde technique implique l'extraction des méthodes de MusicService vers de nouvelles classes plus petites. De plus, les sous-classes existant au sein de MusicService seront également extraites, contribuant ainsi à une réduction supplémentaire de sa complexité.

```

public class MusicServiceGetter {
    2 related problems
    public int getQueueHistorySize() {...}

    2 related problems
    public int getQueueHistoryPosition(int position) {...}

    2 related problems
    public int[] getQueueHistoryList() {...}

    4 related problems
    public String getPath() {...}

    2 related problems
    public String getAlbumName() {...}

    2 related problems
    public String getTrackName() {...}

    public String getGenreName() {...}

    2 related problems
    public String getArtistName() {...}

    public String getAlbumArtistName() {...}

    1 related problem
    public long getAlbumId() {...}

    1 related problem
    public long getArtistId() {...}

    3 related problems
    public long getAudioId() {...}

    2 related problems
    public MusicPlaybackTrack getCurrentTrack() { return getTrack(mPlayPos); }

    2 related problems
    public synchronized MusicPlaybackTrack getTrack(int index) {...}
}

```

Figure 14: La classe MusicServiceGetter

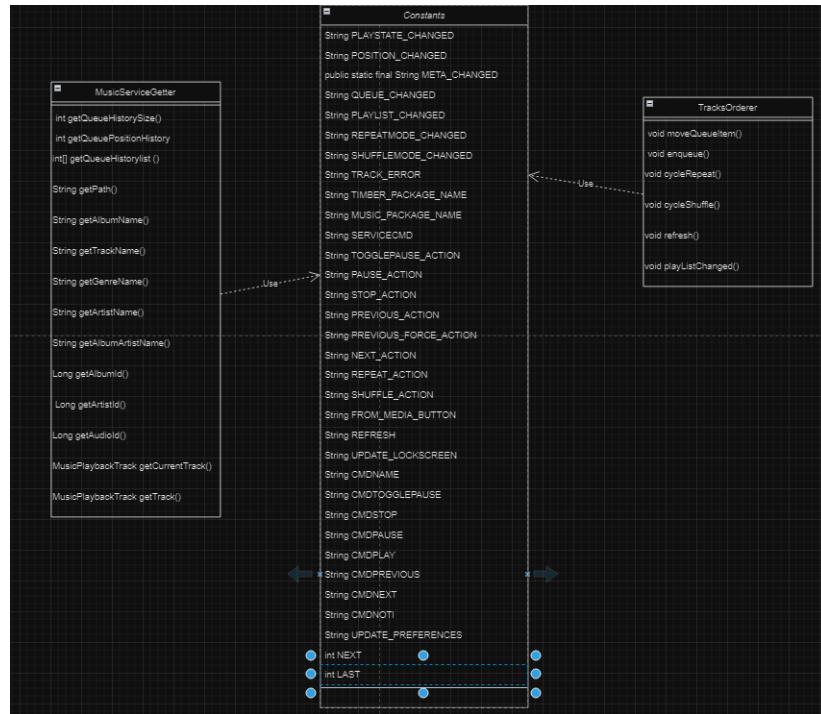


Figure 15: Diagramme UML après le refactoring

```
public class MusicServiceGetter {...}
public class MusicPlayerHandler extends Handler {...}
public class MediaStoreObserver extends ContentObserver implements Runnable {...}
public class Shuffler {...}
public class Player() {...}
public class TrackErrorInfo {...}
interface Constants {...}
1 related problem
interface TrackErrorExtra {...}
public class MultiPlayer implements MediaPlayer.OnErrorListener,
    MediaPlayer.OnCompletionListener {...}
public class ServiceStub extends ITimberService.Stub {...}
public class CursorHandler {...}
public class TracksOrderer {
    1 related problem
    public void moveQueueItem(int index1, int index2) {...} private boolean mIsSupposedToBeP
    2 related problems
    public void enqueue(final long[] list, final int action) {...}

    private void cycleRepeat() {...}

    private void cycleShuffle() {...}

    1 related problem
    public void refresh() { notifyChange(REFRESH); }

    public void playlistChanged() { notifyChange(PLAYLIST_CHANGED); }
}
```

Figure 16: Les classes créées après le refactoring de la classe MusicService

Le résultat du refactoring est un ensemble de classes qui adhèrent au principe de responsabilité unique. Auparavant, cette conformité au principe n'était pas observée.

3.1.2 Switch Statements

L'utilisation d'un switch statement dans la méthode onTouchEvent() présente plusieurs inconvénients. Cette pratique centralise la logique de plusieurs actions tactiles dans un seul bloc, ce qui rend le code difficile à lire, à maintenir et à étendre. De plus, elle viole le principe d'ouverture-fermeture.

Pour remédier à ces problèmes, il est possible de recourir au polymorphisme en utilisant des stratégies pour gérer les différents types d'événements tactiles. Cette

approche permet de séparer les différentes actions en classes distinctes, ce qui rend le code plus modulaire, extensible et facile à comprendre.

Cette factorisation ne va pas être implémentée 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.

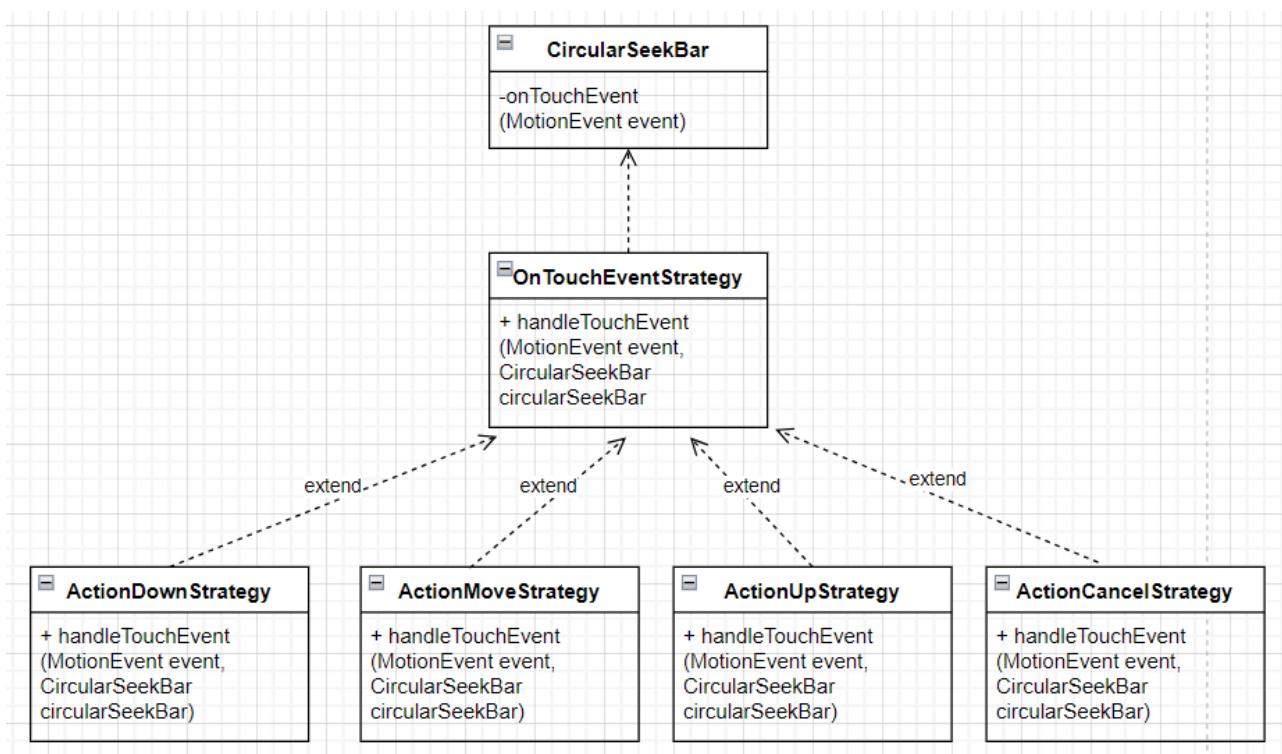


Figure 17: Diagramme UML après la refactorisation de `onTouchEvent`

```

switch (event.getAction()) {
    case MotionEvent.ACTION_DOWN:
        // These are only used for ACTION_DOWN for handling if the pointer was the part that was touched
        float pointerRadiusDegrees = (float) ((mPointerRadius * 180) / (Math.PI * Math.max(mCircleHeight, mCircleWidth)));
        cwDistanceFromPointer = touchAngle - mPointerPosition;
        cwDistanceFromPointer = (cwDistanceFromPointer < 0 ? 360f + cwDistanceFromPointer : cwDistanceFromPointer);
        ccwDistanceFromPointer = 360f - cwDistanceFromPointer;
        // This is for if the first touch is on the actual pointer.
        if (((touchEventRadius >= innerRadius) && (touchEventRadius <= outerRadius)) && ((cwDistanceFromPointer <= pointerRadiusDegrees) ||
            (ccwDistanceFromPointer <= pointerRadiusDegrees))) {
            setProgressBasedOnAngle(mPointerPosition);
            lastCWDistanceFromStart = cwDistanceFromStart;
            mIsMovingCW = true;
            mPointerHaloPaint.setAlpha(mPointerAlphaOnTouch);
            mPointerHaloPaint.setColor(mPointerHaloColorOnTouch);
            recalculateAll();
            invalidate();
            if (mOnCircularSeekBarChangeListener != null) {...}
            mUserIsMovingPointer = true;
            lockAtEnd = false;
            lockAtStart = false;
        } else if (cwDistanceFromStart > mTotalCircleDegrees) {...} else if ((touchEventRadius >= innerRadius) && (touchEventRadius <= outerRadius)) {
            setProgressBasedOnAngle(touchAngle);
            lastCWDistanceFromStart = cwDistanceFromStart;
            mIsMovingCW = true;
            mPointerHaloPaint.setAlpha(mPointerAlphaOnTouch);
            mPointerHaloPaint.setColor(mPointerHaloColorOnTouch);

            recalculateAll();
            invalidate();
            if (mOnCircularSeekBarChangeListener != null) {...}
            mUserIsMovingPointer = true;
            lockAtEnd = false;
            lockAtStart = false;
        } else {...}
        break;
    case MotionEvent.ACTION_MOVE:
        if (mUserIsMovingPointer) {...} else {...}
        break;
    case MotionEvent.ACTION_UP:
        mPointerHaloPaint.setAlpha(mPointerAlpha);
        mPointerHaloPaint.setColor(mPointerHaloColor);
        if (mUserIsMovingPointer) {...} else {...}
        break;
    case MotionEvent.ACTION_CANCEL: // Used when the parent view intercepts touches for things like scrolling
        mPointerHaloPaint.setAlpha(mPointerAlpha);
        mPointerHaloPaint.setColor(mPointerHaloColor);
        mUserIsMovingPointer = false;
        invalidate();
        break;
}
}

```

Figure 18: Switch case de la méthode onTouchEvent()

```
no usages new
public interface OnTouchEventStrategy {
    no usages new *
    void ActionDownStrategy(float x, float y);
    no usages new *
    void ActionMoveStrategy(float x, float y);
    no usages new *
    void ActionUpStrategy();
    no usages new *
    void ActionCancelStrategy();
}
```

Figure 19: Interface OnTouchEventStrategy

```
public class CircularTouchHandler implements OnTouchEventStrategy {
    no usages
    @java.lang.Override
    public void ActionDownStrategy(float x, float y) {

    }
    no usages
    @java.lang.Override
    public void ActionMoveStrategy(float x, float y) {
    }
    no usages
    @java.lang.Override
    public void ActionUpStrategy() {
    }
    no usages
    @java.lang.Override
    public void ActionCancelStrategy() {
    }
}
```

Figure 20: La classe CircularTouchHandler qui implémente l'interface
onTouchEventStrategy

```
public class RectangularTouchHandler implements OnTouchEventStrategy {
    no usages
    @java.lang.Override
    public void ActionDownStrategy(float x, float y) {
    }
    no usages
    @java.lang.Override
    public void ActionMoveStrategy(float x, float y) {
    }
    no usages
    @java.lang.Override
    public void ActionUpStrategy() {
    }
    no usages
    @java.lang.Override
    public void ActionCancelStrategy() {
    }
}
```

Figure 21: La classe `RectangularTouchHandler` qui implémente l'interface `OnTouchEventStrategy`

```
private OnTouchEventStrategy touchHandler;
no usages new *
public void setTouchHandler(OnTouchEventStrategy touchHandler) {
    this.touchHandler = touchHandler;
}
no usages new * 3 related problems
@Override
public boolean onTouchEvent(MotionEvent event) {
    if (touchHandler != null) {
        switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN:
                touchHandler.handleTouchDown(event.getX(), event.getY());
                break;
            case MotionEvent.ACTION_MOVE:
                touchHandler.handleTouchMove(event.getX(), event.getY());
                break;
            case MotionEvent.ACTION_UP:
                touchHandler.handleTouchUp();
                break;
            case MotionEvent.ACTION_CANCEL:
                touchHandler.handleTouchCancel();
                break;
        }
        return true;
    } else {
        return super.onTouchEvent(event);
    }
}
```

Figure 22: Switch case de la méthode `onTouchEvent()` après le refactoring

CircularSeekBar est la classe qui contient la méthode *onTouchEvent*. *OnTouchEventStrategy* est l'interface commune que toutes les stratégies d'événement tactile implémentent. *ActionDownStrategy*, *ActionMoveStrategy*, *ActionUpStrategy* et *ActionCancelStrategy* sont des classes qui implémentent l'interface. *OnTouchEventStrategy* et contiennent la logique pour gérer respectivement les événements ACTION_DOWN, ACTION_MOVE, ACTION_UP et ACTION_CANCEL.

Chaque classe de stratégie a une méthode *handleTouchEvent* qui prend un événement *MotionEvent* et une instance de *CircularSeekBar* en tant que paramètres, et renvoie un booléen indiquant si l'événement a été traité avec succès.

Il est possible d'évaluer l'impact potentiel sur les métriques du code.

CC (Complexity Cyclomatic) :

- Avant le refactoring, la complexité cyclomatique de la méthode *onTouchEvent()* était élevée (35) en raison de la logique complexe regroupée dans le switch statement.
- Après le refactoring, la complexité cyclomatique devrait baisser, car la logique est répartie entre plusieurs classes de stratégies, réduisant ainsi la complexité de chaque classe individuelle.

WMC (Weighted Methods per Class) :

- Après le refactoring, le WMC devrait baisser, car la logique est répartie entre plusieurs classes de stratégies, répartissant ainsi la complexité sur plusieurs classes.

CBO (Coupling Between Object classes) :

- Après le refactoring, le couplage devrait diminuer, car la classe *CircularSeekBar* n'est plus directement dépendante des différentes actions tactiles, mais utilise des interfaces pour communiquer avec les stratégies.

En résumé, le refactoring devrait améliorer les métriques individuelles en réduisant la complexité, en augmentant la cohésion et en réduisant le couplage. Cependant, il est important de noter que l'impact exact dépendra de la manière dont le refactoring a été réalisé et de la structure spécifique du code.

3.1.3 Long Method

La méthode `processTouchEvent()` est affectée par l'anomalie d'une « méthode longue » (Long Method). Avec 150 lignes, sa longueur est excessive, ce qui nuit considérablement à sa lisibilité et à sa maintenance. Il est donc nécessaire de procéder à un refactoring pour en réduire la taille.

```

public void processTouchEvent(MotionEvent ev) {
    final int action = MotionEventCompat.getActionMasked(ev);
    final int actionBarIndex = MotionEventCompat.getActionIndex(ev);

    if (action == MotionEvent.ACTION_DOWN) {
        // Reset things for a new event stream, just in case we didn't get
        // the whole previous stream.
        cancel();
    }

    if (mVelocityTracker == null) {
        mVelocityTracker = VelocityTracker.obtain();
    }
    mVelocityTracker.addMovement(ev);

    switch (action) {
        case MotionEvent.ACTION_DOWN: {
            final float x = ev.getX();
            final float y = ev.getY();
            final int pointerId = MotionEventCompat.getPointerId(ev, 0);
            final View toCapture = findTopChildUnder((int) x, (int) y);

            saveInitialMotion(x, y, pointerId);

            // Since the parent is already directly processing this touch event,
            // there is no reason to delay for a stop before dragging.
            // Start immediately if possible.
            tryCaptureViewForDrag(toCapture, pointerId);

            final int edgesTouched = mInitialEdgesTouched[pointerId];
        }
    }

    case MotionEvent.ACTION_MOVE: {
        if (mDragState == STATE_DRAGGING) {
            final int index = MotionEventCompat.findPointerIndex(ev, mActivePointerId);
            final float x = MotionEventCompat.getX(ev, index);
            final float y = MotionEventCompat.getY(ev, index);
            final int idx = (int) (x - mLastMotionX[mActivePointerId]);
            final int idy = (int) (y - mLastMotionY[mActivePointerId]);

            dragTo(left: mCapturedView.getLeft() + idx, top: mCapturedView.getTop() + idy, idx, idy);

            saveLastMotion(ev);
        } else {
            // Check to see if any pointer is now over a draggableView.
            final int pointerCount = MotionEventCompat.getPointerCount(ev);
            for (int i = 0; i < pointerCount; i++) {
                final int pointerId = MotionEventCompat.getPointerId(ev, i);
                final float x = MotionEventCompat.getX(ev, i);
                final float y = MotionEventCompat.getY(ev, i);
                final float dx = x - mInitialMotion[pointerId];
                final float dy = y - mInitialMotion[pointerId];

                reportNewEdgeDrags(dx, dy, pointerId);
            }
            if (mDragState == STATE_DRAGGING) {
                // Callback might have started an edge drag.
                break;
            }
        }
        final View toCapture = findTopChildUnder((int) x, (int) y);
    }

    if ((edgesTouched & mTrackingEdges) != 0) {
        mCallback.onEdgeTouched(edgeFlags.edgesTouched & mTrackingEdges, pointerId);
    }
    break;
}

case MotionEventCompat.ACTION_POINTER_DOWN: {
    final int pointerId = MotionEventCompat.getPointerId(ev, actionBarIndex);
    final float x = MotionEventCompat.getX(ev, actionBarIndex);
    final float y = MotionEventCompat.getY(ev, actionBarIndex);

    saveInitialMotion(x, y, pointerId);

    // A ViewDragHelper can only manipulate one view at a time.
    if (mDragState == STATE_IDLE) {
        // If we're idle we can do anything! Treat it like a normal down event.

        final View toCapture = findTopChildUnder((int) x, (int) y);
        tryCaptureViewForDrag(toCapture, pointerId);

        final int edgesTouched = mInitialEdgesTouched[pointerId];
        if ((edgesTouched & mTrackingEdges) != 0) {
            mCallback.onEdgeTouched(edgeFlags.edgesTouched & mTrackingEdges, pointerId);
        }
        else if (isCapturedViewUnder((int) x, (int) y)) {
            // We're still tracking a captured view. If the same view is under this
            // point, we'll swap to controlling it with this pointer instead.
            // (This will still work if we're "catching" a settling view.)

            tryCaptureViewForDrag(mCapturedView, pointerId);
        }
    }
    break;
}

case MotionEventCompat.ACTION_POINTER_UP: {
    final int pointerId = MotionEventCompat.getPointerId(ev, actionBarIndex);
    if (mDragState == STATE_DRAGGING & pointerId == mActivePointerId) {
        // Try to find another pointer that's still holding on to the captured view.
        int newActivePointer = INVALID_POINTER;
        final int pointerCount = MotionEventCompat.getPointerCount(ev);
        for (int i = 0; i < pointerCount; i++) {
            final int id = MotionEventCompat.getPointerId(ev, i);
            if (id == mActivePointerId) {
                // This one's going away, skip.
                continue;
            }

            final float x = MotionEventCompat.getX(ev, i);
            final float y = MotionEventCompat.getY(ev, i);
            if (findTopChildUnder((int) x, (int) y) == mCapturedView &&
                tryCaptureViewForDrag(mCapturedView, id)) {
                newActivePointer = mActivePointerId;
            }
        }
    }
    break;
}
}

```

```
        if (newActivePointer == INVALID_POINTER) {
            // We didn't find another pointer still touching the view, release it.
            releaseViewForPointerUp();
        }
    }
    clearMotionHistory(pointerId);
    break;
}

case MotionEvent.ACTION_UP: {
    if (mDragState == STATE_DRAGGING) {
        releaseViewForPointerUp();
    }
    cancel();
    break;
}

case MotionEvent.ACTION_CANCEL: {
    if (mDragState == STATE_DRAGGING) {
        dispatchViewReleased(xvel: 0, yvel: 0);
    }
    cancel();
    break;
}
}
```

Figure 23: La méthode processTouchEvent()

Pour alléger la méthode *processTouchEvent()*, nous nous sommes concentrés sur la structure du *switch-case*. D'après les figures ci-dessus, il est évident que c'est cette section qui occupe le plus de lignes. L'analyse du code a révélé qu'il était possible de la refondre en utilisant plusieurs méthodes privées plutôt que de maintenir l'intégralité du code dans une seule méthode. Cette approche améliore la lisibilité en segmentant les fonctionnalités. Les figures ci-dessous illustrent la portion simplifiée de la méthode et présentent les nouvelles méthodes ajoutées.

Method: processTouchEvent(MotionEvent)					
	Metric	Metrics Set	Description	Value	Regular Ran...
<input type="radio"/>	CND		Condition Nesting Depth	2	[0..2)
<input type="radio"/>	LND		Loop Nesting Depth	1	[0..2)
<input type="radio"/>	CC		McCabe Cyclomatic Complexity	27	[0..3)
<input type="radio"/>	NOL		Number Of Loops	2	
<input type="radio"/>	LOC		Lines Of Code	134	[0..11)
<input type="radio"/>	NOPM		Number Of Parameters	1	[0..3)
<input type="radio"/>	HVL	Halstead ...	Halstead Volume	1867.7178	
<input type="radio"/>	HD	Halstead ...	Halstead Difficulty	72.0	
<input type="radio"/>	HL	Halstead ...	Halstead Length	309	
<input type="radio"/>	HEF	Halstead ...	Halstead Effort	134475.6...	
<input type="radio"/>	HVC	Halstead ...	Halstead Vocabulary	66	
<input type="radio"/>	HER	Halstead ...	Halstead Errors	0.8749	
<input type="radio"/>	MMI	Maintaina...	Maintainability Index	30.2515	[0.0..19.0]

Figure 24: Les métriques de la classe `processTouchEvent()` après les changements

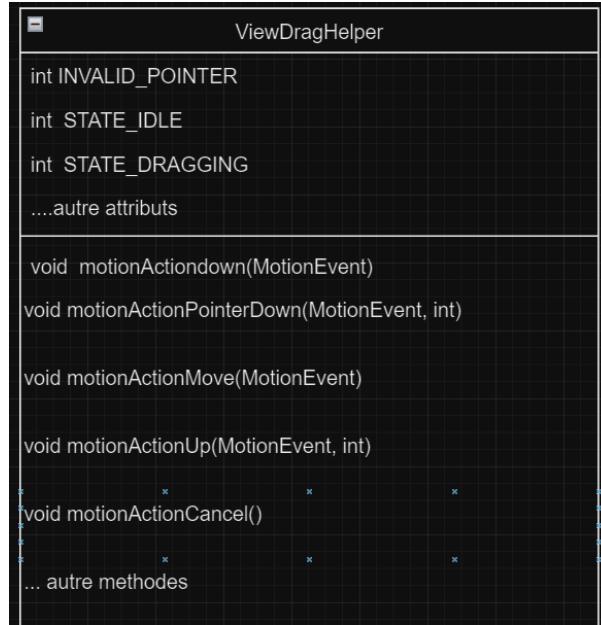


Figure 25: Diagramme UML de la classe `ViewDragHelper` après le refactoring

A screenshot of the Android Studio code editor showing the refactoring process. The code is being modified in the `onTouchEvent` method of a class. The original code is:

```

if (mVelocityTracker == null) {
    mVelocityTracker = VelocityTracker.obtain();
}
mVelocityTracker.addMovement(ev);

switch (action) {
    case MotionEvent.ACTION_DOWN: {
        motionActionDown(ev);
        break;
    }

    case MotionEventCompat.ACTION_POINTER_DOWN: {
        motionActionPointerDown(ev, actionIndex);
        break;
    }

    case MotionEvent.ACTION_MOVE: {
        motionActionMove(ev);
        break;
    }

    case MotionEventCompat.ACTION_POINTER_UP: {
        motionActionPointerUp(ev, actionIndex);
        break;
    }

    case MotionEvent.ACTION_UP: {
        motionActionUp();
        break;
    }

    case MotionEvent.ACTION_CANCEL: {
        motionActionCancel();
        break;
    }
}

```

The cursor is currently positioned on the closing brace of the `case MotionEvent.ACTION_CANCEL:` block. The status bar at the bottom shows 'Sync' and 'Build Output'.

Figure 26: La création des nouvelles méthodes dans le switch case

```

private void motionActionDown(MotionEvent ev) { Complexity is 5 Everything is cool!
    final float x = ev.getX();
    final float y = ev.getY();
    final int pointerId = MotionEventCompat.getPointerId(ev, 0);
    final View toCapture = findTopChildUnder((int) x, (int) y);

    saveInitialMotion(x, y, pointerId);

    // Since the parent is already directly processing this touch event,
    // there is no reason to delay for a slop before dragging.
    // Start immediately if possible.
    tryCaptureViewForDrag(toCapture, pointerId);

    final int edgesTouched = mInitialEdgesTouched[pointerId];
    if ((edgesTouched & mTrackingEdges) != 0) {
        mCallback.onEdgeTouched(edgeFlags, edgesTouched & mTrackingEdges, pointerId);
    }
}

```

```

private void motionActionPointerDown(MotionEvent ev, int actionIndex) { Complexity is 8 It's time to do
    final int pointerId = MotionEventCompat.getPointerId(ev, actionIndex);
    final float x = MotionEventCompat.get(ev, actionIndex);
    final float y = MotionEventCompat.get(ev, actionIndex);

    saveInitialMotion(x, y, pointerId);

    // A ViewDragHelper can only manipulate one view at a time.
    if (mDragState == STATE_IDLE) {
        // If we're idle we can do anything! Treat it like a normal down event.

        final View toCapture = findTopChildUnder((int) x, (int) y);
        tryCaptureViewForDrag(toCapture, pointerId);

        final int edgesTouched = mInitialEdgesTouched[pointerId];
        if ((edgesTouched & mTrackingEdges) != 0) {
            mCallback.onEdgeTouched(edgeFlags, edgesTouched & mTrackingEdges, pointerId);
        }
    } else if (isCapturedViewUnder((int) x, (int) y)) {
        // We're still tracking a captured view. If the same view is under this
        // point, we'll swap to controlling it with this pointer instead.
        // (This will still work if we're "catching" a settling view.)

        tryCaptureViewForDrag(mCapturedView, pointerId);
    }
}

```

```

private void motionActionMove(MotionEvent ev) {
    if (mDragState == STATE_DRAGGING) {
        final int index = MotionEventCompat.findPointerIndex(ev, mActivePointerId);
        final float x = MotionEventCompat.getX(ev, index);
        final float y = MotionEventCompat.getY(ev, index);
        final int idx = (int) (x - mLastMotionX[mActivePointerId]);
        final int idy = (int) (y - mLastMotionY[mActivePointerId]);

        dragTo( left: mCapturedView.getLeft() + idx, top: mCapturedView.getTop() + idy, idx, idy);

        saveLastMotion(ev);
    } else {
        // Check to see if any pointer is now over a draggable view.
        final int pointerCount = MotionEventCompat.getPointerCount(ev);
        for (int i = 0; i < pointerCount; i++) {
            final int pointerId = MotionEventCompat.getPointerId(ev, i);
            final float x = MotionEventCompat.getX(ev, i);
            final float y = MotionEventCompat.getY(ev, i);
            final float dx = x - mInitialMotionX[pointerId];
            final float dy = y - mInitialMotionY[pointerId];

            reportNewEdgeDrags(dx, dy, pointerId);
            if (mDragState == STATE_DRAGGING) {
                // Callback might have started an edge drag.
                return;
            }
        }
        final View toCapture = findTopChildUnder((int) x, (int) y);
        if (checkTouchSlop(toCapture, dx, dy) &&
            tryCaptureViewForDrag(toCapture, pointerId)) {
            return;
        }
    }
    saveLastMotion(ev);
}

```

```

private void motionActionPointerUp(MotionEvent ev, int actionIndex) {
    final int pointerId = MotionEventCompat.getPointerId(ev, actionIndex);
    if (mDragState == STATE_DRAGGING & pointerId == mActivePointerId) {
        // Try to find another pointer that's still holding on to the captured view.
        int newActivePointer = INVALID_POINTER;
        final int pointerCount = MotionEventCompat.getPointerCount(ev);
        for (int i = 0; i < pointerCount; i++) {
            final int id = MotionEventCompat.getPointerId(ev, i);
            if (id == mActivePointerId) {
                // This one's going away, skip.
                continue;
            }

            final float x = MotionEventCompat.getX(ev, i);
            final float y = MotionEventCompat.getY(ev, i);
            if (findTopChildUnder((int) x, (int) y) == mCapturedView &&
                tryCaptureViewForDrag(mCapturedView, id)) {
                newActivePointer = mActivePointerId;
                break;
            }
        }

        if (newActivePointer == INVALID_POINTER) {
            // We didn't find another pointer still touching the view, release it.
            releaseViewForPointerUp();
        }
    }
    clearMotionHistory(pointerId);
}

```

```

private void motionActionUp() {
    if (mDragState == STATE_DRAGGING) {
        releaseViewForPointerUp();
    }
    cancel();
}

private void motionActionCancel() {
    if (mDragState == STATE_DRAGGING) {
        dispatchViewReleased(xvel: 0, yvel: 0);
    }
    cancel();
}

```

Figure 27: Les nouvelles méthodes motionActionDown(), motionActionPointerDown(), motionActionMove(), motionActionPointerUp(), motionActionUp, motionActionCancel()

La création de ces méthodes a donné des résultats nettement plus compréhensibles et faciles à maintenir.

3.1.4 Long Parameter List

La méthode playAll() dans la classe BaseSongAdapter souffre d'une anomalie de type Long Parameter List. Selon la figure 8, cette méthode à huit paramètres, un nombre qu'il est nécessaire de réduire à trois ou moins pour résoudre l'anomalie. Une analyse révèle que certains de ces paramètres ne sont pas utilisés et peuvent donc être éliminés. Ensuite, nous allons passer un objet en argument, ce qui permet de regrouper plusieurs valeurs et de réduire considérablement le nombre de paramètres. Le résultat de cette modification est illustré dans les figures suivantes.

```
public void playAll(final Activity context, final long[] list, int position,
                    final long sourceId, final TimberUtils.IdType sourceType,
                    final boolean forceShuffle, final Song currentSong, boolean navigateNowPlaying) {

    if (context instanceof BaseActivity) {
        CastSession castSession = ((BaseActivity) context).getCastSession();
        if (castSession != null) {
            navigateNowPlaying = false;
            TimberCastHelper.startCasting(castSession, currentSong);
        } else {
            MusicPlayer.playAll(context, list, position, sourceId: -1, TimberUtils.IdType.NA, forceShuffle: false);
        }
    } else {
        MusicPlayer.playAll(context, list, position, sourceId: -1, TimberUtils.IdType.NA, forceShuffle: false);
    }

    if (navigateNowPlaying) {
        NavigationUtils.navigateToNowplaying(context, withAnimations: true);
    }
}
```

Figure 28: La méthode playAll

```
public void playAll(final Activity context, PlayAllParams params, boolean navigateNowPlaying) {  
    if (context instanceof BaseActivity) {  
        CastSession castSession = ((BaseActivity) context).getCastSession();  
        if (castSession != null) {  
            navigateNowPlaying = false;  
            TimberCastHelper.startCasting(castSession, params.currentSong);  
        } else {  
            MusicPlayer.playAll(context, params.list, params.position, [sourcedId: -1, TimberUtils.IdType.NA, forceShuffle: false]);  
        }  
    } else {  
        MusicPlayer.playAll(context, params.list, params.position, [sourcedId: -1, TimberUtils.IdType.NA, forceShuffle: false]);  
    }  
  
    if (navigateNowPlaying) {  
        NavigationUtils.navigateToNowplaying(context, [withAnimations: true]);  
    }  
}
```

Figure 29: La méthode `playAll()` après l'ajout de la classe `PlayAllParams`

```
public class PlayAllParams {  
    private final long[] list;  
    private final int position;  
    private final Song currentSong;  
  
    public PlayAllParams(long[] list, int position, Song currentSong) {  
        this.list = list;  
        this.position = position;  
        this.currentSong = currentSong;  
    }  
  
    public long[] getList() {  
        return list;  
    }  
  
    7 related problems  
    public int getPosition() {  
        return position;  
    }  
  
    public Song getCurrentSong() {  
        return currentSong;  
    }  
}
```

Figure 30: La classe `PlayAllParams`

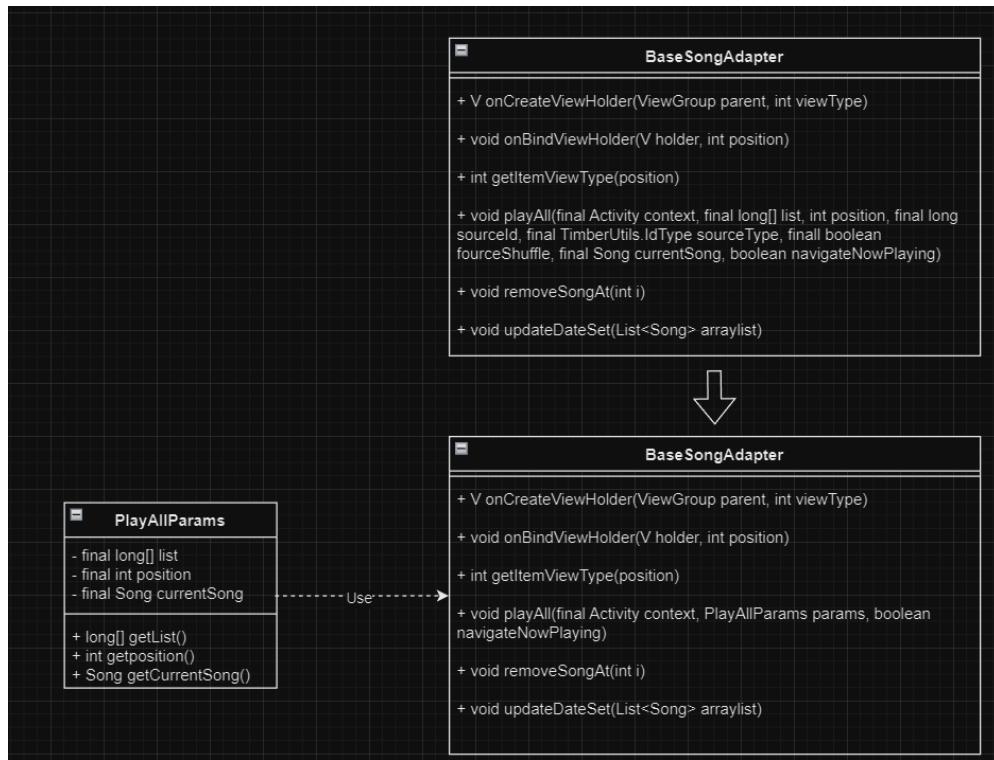


Figure 31: Diagramme UML de l'avant et après de la classe `BaseSongAdapter`

3.1.5 Feature envy

```
public class NumberUtils {

    public static float[][] getControlPointsFor(int start) {
        switch (start) {
            case -1:
                return Null.getInstance().getControlPoints();
            case 0:
                return Zero.getInstance().getControlPoints();
            case 1:
                return One.getInstance().getControlPoints();
            case 2:
                return Two.getInstance().getControlPoints();
            case 3:
                return Three.getInstance().getControlPoints();
            case 4:
                return Four.getInstance().getControlPoints();
            case 5:
                return Five.getInstance().getControlPoints();
            case 6:
                return Six.getInstance().getControlPoints();
            case 7:
                return Seven.getInstance().getControlPoints();
            case 8:
                return Eight.getInstance().getControlPoints();
            case 9:
                return Nine.getInstance().getControlPoints();
            default:
                throw new InvalidParameterException("Unsupported number requested");
        }
    }
}
```

Figure 32: La classe *NumberUtils*

Afin de réduire le nombre d'accès à des attributs de classes externes pour la méthode *getControlPointsFor* de la classe *NumberUtils*, on peut simplement extraire les valeurs des attributs des classes externes dans un tableau qui sera une propriété de la classe *NumberUtils*. Celà simplifiera beaucoup la méthode *getControlPointsFor*, car au lieu d'avoir un switch statement à 12 cas, on a réduit la fonction à 3 cas possible: accéder aux points de contrôles Zero to Nine à l'aide du tableau, accéder au point de contrôle *Null* (cas spécial *start == -1*) ou sinon retourner une erreur lorsque n'importe quelle autre valeur est fournie.

Les valeurs de ATFD et FDP deviennent alors 1 et la valeur de LAA devient 0.5, donc les conditions ne sont plus valides pour avoir une anomalie de type *Feature Envy*.

```

import com.naman14.timber.timely.model.number.*;
import java.security.InvalidParameterException;

public class NumberUtils {

    private static final float[][][] controlPoints = {
        Zero.getInstance().getControlPoints(),
        One.getInstance().getControlPoints(),
        Two.getInstance().getControlPoints(),
        Three.getInstance().getControlPoints(),
        Four.getInstance().getControlPoints(),
        Five.getInstance().getControlPoints(),
        Six.getInstance().getControlPoints(),
        Seven.getInstance().getControlPoints(),
        Eight.getInstance().getControlPoints(),
        Nine.getInstance().getControlPoints()
    };

    public static float[][][] getControlPointsFor(int start) {
        if (start == -1) return Null.getInstance().getControlPoints();
        if (start < controlPoints.length && start >= 0) return controlPoints[start];

        throw new InvalidParameterException("Unsupported number requested");
    }
}

```

Figure 34: La classe NumberUtils après refactoring

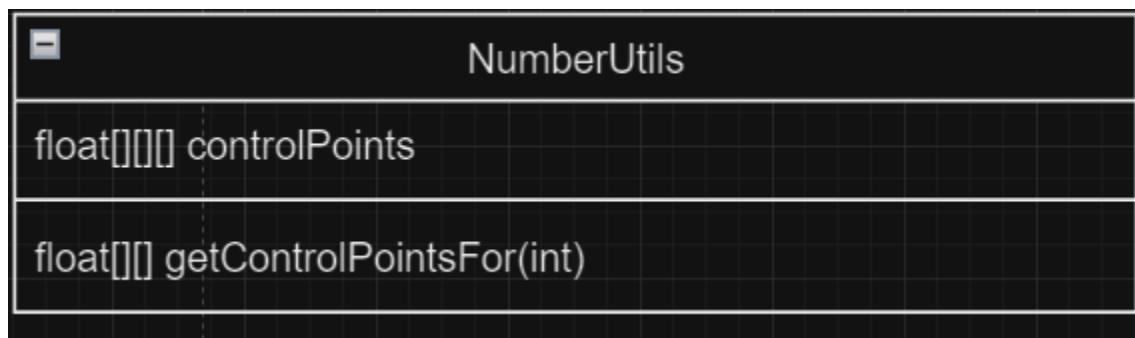


Figure 35: Diagramme UML de la classe NumberUtils après refactoring

3.2 Métriques après le refactoring

Après les récentes opérations de refactoring, nous avons recalculé les métriques pour les projets individuels, les classes et les méthodes, aboutissant au tableau suivant:

Métrique	Impact	Moyenne	Écart-type	Min	Max
TLOC	Minime	10.8	13.1	0	168
NOM	Minime	6,49	9,36	0	62
SIZE2	Grande	24,13	14,89	13	118
CC	Minime	2.4	3.1	1	54
RFC	Moyen	11	15	0	127
WMC	Grande	15,78	31,91	0	289
DIT	Minime	1,06	0.24	1	2
NOC	Minime	0,17	0,98	0	11
CA	Moyen	12,222	19,0367	0	86
CBO	Moyen	5,45	6,733	0	44
LCOM	Moyen	2,211	2,828	0	25

Tableau 9 : Métriques après le refactoring

Cette mise à jour montre des améliorations notables après la correction des anomalies, en particulier dans les métriques SIZE2 et WMC, qui indiquent une évolution positive. Cependant, ces changements ne sont pas suffisamment significatifs pour considérer que l'impact global sur le projet est majeur. En effet, la plupart des métriques présentent peu de variations, et ces légères modifications ont eu un impact limité sur l'envergure du projet. En conclusion, bien que le refactoring ait apporté des améliorations positives, offrant une meilleure flexibilité et maintenabilité, il reste encore de nombreux aspects à améliorer dans le projet.

3.3 Critères du modèle ISO 9126 après le refactoring

Les fruits de nos efforts pour améliorer le code, les nouvelles évaluations basées sur les critères ISO 9126 sont présentés dans la figure ci-dessous :



Figure 35: Évaluation des critères ISO 9126 après le refactoring

Ces métriques ont été calculées grâce aux formules que nous avons présentées dans la figure 2.

3.3.1 Réutilisabilité

En comparant les valeurs de métriques de réutilisabilité, on constate qu'elle a diminué, passant d'une valeur de 4,9662 à 3,3775. Cela indique que le réusinage du code effectué n'a pas permis l'augmentation de sa réutilisabilité. Afin d'améliorer la valeur de cette métrique, il faudrait corriger des anomalies influençant directement la réutilisabilité du code. C'est-à-dire des corrections qui permettent de diminuer le couplage, augmenter la cohésion, etc.

3.3.2 Flexibilité

La flexibilité a été particulièrement améliorée, presque doublée, grâce notamment à l'introduction d'interfaces favorisant le polymorphisme. La métrique NOP a joué un rôle clé dans cette amélioration. Sa nouvelle valeur est de 11,3973, soit une nette augmentation face à son ancienne valeur de 6,6115.

3.3.3 Compréhensibilité

Le recours accru au polymorphisme peut réduire la compréhensibilité générale du code. La nouvelle valeur de la Compréhensibilité, -9,6952, est inférieure à celle d'avant le refactoring -8,0583. Cela s'explique par l'augmentation du nombre de méthodes polymorphiques, rendant le système globalement moins compréhensible.

Par ailleurs, nous avons réduit le nombre de lignes de code dans la classe MusicService, diminuant ainsi la valeur de la métrique NOM pour cette classe, dans le but d'améliorer la Compréhensibilité. Toutefois, cet effet est éclipsé par l'augmentation de la métrique NOP, ce qui explique pourquoi l'amélioration de la Compréhensibilité n'est pas plus marquée malgré la simplification notable de la classe MusicService.

3.3.4 Fonctionnalité:

En observant la valeur des métriques de fonctionnalité, on constate une légère augmentation du critère, soit le fait qu'elle passe d'une valeur de 4,9392 à 5,3444. Cette augmentation indique que le réusinage du code a été utile pour l'augmentation de cette métrique. Notamment, l'ajout de fonction polymorphe et le gain de cohésion reçu en divisant une grosse classe pourraient justifier cet écart.

3.3.5 Extensibilité

Pour l'extensibilité, on note une forte augmentation de la métrique, elle passe de 2,1076 à 5,8279. Cette forte augmentation indique que le réusinage effectué du code a grandement contribué à la capacité du système à intégrer de nouvelles exigences grâce à des attributs déjà présents. L'ajout de méthodes polymorphiques aurait influencé cette métrique.

3.3.6 Efficacité

Similairement à l'extensibilité, la métrique de l'efficacité a également reçu une augmentation, passant de 3,532 8 à 5,344 4. Cela est signe d'une utilisation plus efficace des concepts et techniques de conception orientée objet afin d'accomplir ses objectifs. L'ajout de fonction polymorphe pourrait expliquer l'augmentation de cette métrique.

Globalement, en comparant ces résultats avec ceux de la figure 1, on observe une amélioration notable de la qualité globale du système. Les critères ayant affiché des scores plus élevés incluent: Efficacité, Extensibilité, Flexibilité et Fonctionnalité. Voici une comparaison des scores totaux avant et après les modifications :

Avant refactoring, la somme des valeurs des critères était de:

$$3.5319 + 2.1145 + 6.6015 + 4.6223 + 4.9914 - 8.0186 = 13.843$$

Après refactoring, cette somme s'élève à:

$$5.3444 + 5.8279 + 11.3973 + 3.377 - 9.6952 = 16.2514$$

Le gain total de valeur est donc de 2,408 4, ce qui confirme que le refactoring a eu un impact positif sur le système.

Conclusion

En conclusion, ce projet a permis d'examiner en profondeur la qualité de conception du système logiciel Timber à travers une série de métriques et d'analyses détaillées. Nous avons appliqué des principes et des techniques avancés d'architecture logicielle pour évaluer, identifier et corriger des anomalies dans le code, en adhérant aux critères de qualité ISO 9126.

L'évaluation de différentes métriques telles que la cohésion, la complexité, le couplage et la taille a révélé des aspects cruciaux de la qualité du logiciel. Notre analyse a permis d'identifier des "code smells" et des anomalies, conduisant à des stratégies de refactoring ciblées. Ces interventions ont amélioré significativement la maintenabilité, la compréhensibilité, et d'autres aspects de la qualité du code.

Le travail sur Timber a démontré l'importance d'une conception et d'une architecture logicielle rigoureuses pour la réussite et la pérennité d'un système logiciel. Les compétences acquises et appliquées dans ce projet sont inestimables pour tout professionnel de l'informatique, mettant en lumière l'impact direct de l'architecture logicielle sur la qualité et la performance d'un système.

Donc ce travail pratique a permis d'améliorer le système Timber d'une part, mais aussi d'enrichir notre compréhension et nos compétences en architecture logicielle, en conception avancée, et en pratiques de refactoring d'autre part. C'est une étape importante dans notre parcours académique et professionnel, offrant une expérience précieuse dans l'application des théories et des concepts à des problèmes logiciels concrets.

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
- [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/Chidamber%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] Lanza, M., & Marinescu, R. (2011). *Object-oriented metrics in practice: Using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer.
- [7] M. Fokaefs et F. Guibault. (2024). LOG8430 : Qualité de Conception [PDF]. Non publié.https://moodle.polymtl.ca/pluginfile.php/1217417/mod_label/intro/03.QualiteDeConception-H24.pdf.