LOG8430

Architecture logicielle et conception avancée

Automne 2018

TP2

Qualité de la conception, mauvaise conception et refactoring

Présenté par :

Alexandre Clark (1803508)

David Tremblay (1748125)

Félix Agagnier (1795792)

12 octobre 2018

École Polytechnique de Montréal

Département de Génie Informatique et Logiciel

Question 1

Mesurez la qualité de la conception de JFreeChart

A. Métriques

Tableau 1 : Complexité

	Maximum	Moyenne	Total	Std. Dev.
Complexité cyclomatique de McCabe (*Metrics3*) [CC]	57	2.188	N/A	3.411
Nombre de méthodes (Metrics3) [NOM]	230	13.331	8452	21.699
Méthodes pondérées par classe (Metrics3) [WMC]	658	30.511	19 344	53.761
Réponses pour classe (MetricsReloaded) [RFC]	N/A	33.30	N/A	N/D

En ce qui a trait à la complexité cyclomatique, le code source de JfreeChart obtient un score de 2.188 en moyenne. Une bonne convention de codage suggère d'avoir une complexité cyclomatique d'au plus 10. Par contre, on peut observer que la classe ayant la plus grande complexité a un score de 57, ce qui est très élevé. Une classe ayant une grande complexité cyclomatique demande beaucoup plus de tests puisqu'elle contient un grand nombre de chemins logiques que le code peut parcourir. Les classes ayant un grand niveau de complexité cyclomatique sont aussi souvent peu cohésives puisque les nombreux points de décisions suggèrent que la classe fait plus qu'une tâche bien définie et ne respecte donc pas le principe SOLID de responsabilité unique. Le même raisonnement peut être appliqué pour la métrique NOM. En effet, les classes possèdent en moyenne 13 méthodes mais la classe ayant le plus de méthodes en possède 230. Une classe qui possède autant de méthodes est responsable de beaucoup trop de choses et ne respecte pas, encore une fois, le principe de responsabilité unique. Dans le même ordre d'idée, on observe une valeur moyenne de 33 pour la métrique RFC qui indique le nombre de méthodes pouvant être appelées par un objet d'une classe. Une valeur élevée pour cette métrique indique un code qui est difficile à maintenir et à tester puisque les objets peuvent appeler un très grand nombre de méthodes et un niveau de compréhension élevé est

nécessaire pour bien maîtriser le code. En bref, la qualité générale de JFreeChart pour la métrique de complexité est assez bonne en moyenne mais on constate que certaines classes devraient être refactorisées et probablement séparées en plusieurs sous-classes.

Tableau 2 : Taille

	Maximum	**Moyenne**	**Total**	**Std. Dev**
Nombre de lignes de code (par méthode) (*Metrics3*) [LOC]	311	8.816	77942	16.338
Nombre d'attributs et méthodes d'une classe (*MetricsReloaded*) [SIZE2]	N/D	121.96	66 225	N/D
Nombre de méthodes déclarées (*Metrics3*) [NOM]	230	13.331	8452	21.699

Pour la métrique de taille, on remarque que les méthodes ont en moyenne ~13 lignes de code, ce qui est très bien. Par contre, on remarque une méthode de taille maximale dépassant les 300 lignes. Une telle méthode pourrait très certainement être découpée en méthodes plus petites. Pour le nombre d'attributs par classe, on obtient un score de 121.96 en moyenne. Cette valeur est très élevée et suggère une faible cohésion dans le code source de JFreeChart en général. Pour le nombre de méthodes par classes, une moyenne de 10 méthodes est excellente et suggère que les classes ont en général une responsabilité unique respectant le principe SOLID mais nous pouvons observer un maximum de 230 méthodes dans une classe qui devrait très probablement être refactorisée en plusieurs sous-méthodes. En résumé, pour la métrique de taille, on remarque que les classes ont trop d'attributs et méthodes indiquant une violation du principe SOLID SRP et que les méthodes sont en moyenne assez courtes mais que de la refactorisation est nécessaire sur quelques classes.

Tableau 3 : Couplage

	Maximum	Moyenne	Total	Std. Dev.
Couplage entre objet (*MetricsReloaded*) [CBO]	N/D	13.50	N/A	N/D
Couplage afférent (Metrics3) [Ca]	221	48.622	N/D	67.062

Pour la métrique de couplage, nous obtenons une valeur de 13.50. Plus cette valeur est petite, meilleur est le code. Une valeur de 13.50 est donc une bonne valeur qui indique un faible niveau de couplage entre les modules du code source de JFreeChart. Un faible couplage permet de faciliter la maintenance du code puisqu'un changement dans une classe aura un faible impact sur les autres classes. Par contre, pour ce qui est du couplage afférent, on remarque une moyenne de 49 et une valeur maximale de 221. Le couplage afférent étant une métrique indiquant le nombre de classes qui dépendent d'une classe, on constate donc que les classes sont très dépendantes les unes des autres et qu'il semble une avoir une classe dont un très grand nombre de classes dépendent (probablement une des classes de base du projet JFreeChart).

Tableau 4 : Cohésion

	Maximum	Moyenne	Total	Std. Dev.
Manque de cohésion des méthodes (*MetricsReloaded*) [LCOM]	1.333	0.267	N/A	N/D

On obtient à l'aide de l'outil *MetricsReloaded* une valeur de 0.267 en moyenne pour la métrique LCOM. Une valeur de 0 indique une parfaite cohésion et une valeur de 1 indique la pire cohésion pour cette métrique. Cela semble donc suggérer une assez bonne cohésion en moyenne pour le code source de JFreeChart. Une bonne cohésion est un facteur important pour respecter les principes SOLID puisque cela suggère que les classes ont une responsabilité unique.

B. Critères ISO 9126

Nous ne sommes pas arrivés à obtenir les métriques du critère ISO 9126 à l'aide de l'outil Metrics3. Nous obtenons des valeurs pour l'outil ptidej, mais les valeurs obtenues sont celles de chacune des classes du projet JFreeChart plutôt que de l'ensemble du projet. Nous avons donc observé l'ensemble des valeurs et avons commenté nos résultats sur ce que nous observons en moyenne (négatif, positif, etc.) et avons rapporté les valeurs minimales et maximales observées dans le tableau ci-dessous.

Tableau 5 : Valeurs des métriques ISO 9126

	MIN	**MAX**	**General**
Flexibility	-188.00	1.25	Très negatif
Effectiveness	0.447	2.381	Légèrement positif
Extensibility	-436.215	1.118	Négatif
Functionality	343.3	351.9	Très positif
Reusability	545.0	794.5	Très positif
Understandability	-817.525	-514.217	Très négatif

Tout d'abord, nous observons une valeur grandement négative pour la métrique *Flexibility*. Cela suggère donc que le code est peu propice aux changements faciles. C'est-à-dire qu'un changement dans le code risque d'être ardu et qu'il est difficile de modifier un module. De plus, on observe une valeur faible pour la métrique *Extensibility.* Cela démontre donc qu'il est difficile d'ajouter des éléments aux modules du projet.Par contre, pour la métrique *Reusability*, on observe des valeurs qui sont élevées. Cela suggère un code orienté objet. En observant le code source du projet, on voit que c'est le cas dû au très grand nombre de classes et chaque classe représentant une entité. Donc, bien que les modules soit difficiles à modifier et qu'ile st difficile d'y ajouter des fonctionnalités, les modules sont bien séparés les uns des autres et il est facile de les réutiliser par la suite. Bien que la variable *Effectiveness* soit positive, sa valeur est basse. Il semble donc que le projet ait de la difficulté à atteindre le comportement désiré à l'aide de l'architecture orientée objet. Finalement, on constate que le code de JFreeChart est très difficile à comprendre à l'aide de la métrique *Understandability*. Un code difficile à lire est souvent dur à maintenir et à tester aussi.

C. Impact des métriques sur la qualité du projet

On constate en a) des problèmes de complexité cyclomatique et des classes beaucoup trop grosses. Cela concorde avec la métrique *Understandability*. En effet, un projet peu compréhensible est souvent causé par une complexité élevée, une faible cohésion et des classes ayant trop de responsabilités. Par contre, un code réutilisable est habituellement cohésif et nos métriques en a) se contredisent sur ce point.

1. Le nombre de lignes de codes par méthode qui atteint les 311 lignes de code dans la classe XYDifferenceRenderer cause bel et bien des problèmes de compréhension due à la longueur et la complexité d'une telle méthode.
2. Le nombre de méthodes déclarées de 230 dans la classe XYPlot effectue beaucoup trop de responsabilités et dénote notamment une mauvaise odeur de type *God Class* et atteint la réutilisabilité et la compréhension du code.
3. Une haute cohérence dans une classe augmente aussi drastiquement la compréhension et la réutilisation du code, puisque les classes sont concises et effectuent le moins de d'opérations possibles pour les besoins de la solution. Cela augmente aussi la réutilisabilité du code.

Question 2

Trouver le seuil des métriques

Tableau 5 : Seuil des métriques

	AVG	ST.DEV	LOW (AVG - ST.DEV)	HIGH (AVG + ST.DEV)	VERY_HIGH ((AVG + ST.DEV) * 1.5)
CC	2.188	3.411	-1.223 = ~ 0	5.599	8.3985
NOM	13.331	21.699	-8.368 = ~ 0	35.03	52.545
WMC	30.511	53.761	-23.25 = ~ 0	84.272	126.408
LOC	8.816	16.338	-7.522 = ~ 0	25.154	37.731
Ca	48.622	67.062	-18.44 = ~ 0	115.684	173.526

En analysant la question 1 et le tableau ci-dessus, on peut s'attendre à trouver des anomalies de type *God Class* dans le projet JFreeChart. En effet, notre seuil indique une

valeur supérieure à 52 pour la métrique NOM sera alarmant. Nous avons déjà observé des classes ayant beaucoup plus de méthodes dans les tableaux 1 à 4 et avons remarqué que les classes semblent avoir trop de responsabilités dans le projet JFreeChart. Nous avons aussi observé une faible cohésion et un couplage élevé dans le projet. Deux signes de l'anti-patron Feature Envy. Nous avons donc porté une attention particulière à ses métriques pour détecter des valeurs qui ne respectent pas nos seuils définis ci-haut.

Trouver cinq anomalies différentes dans JFreeChart

1.StandardChartTheme.class

Justification

Selon l'outil STAN, le critère de complexité FAT atteint un niveau critique, avec la valeur de 827. La figure suivante représente une échelle de comparaison entre les valeurs optimales, acceptables et mauvaises pour le critère FAT. On peut observer que la métrique obtenue est plus de six fois supérieure à ce qui est considéré comme une trop grande complexité pour la classe et treize fois la valeur optimale.

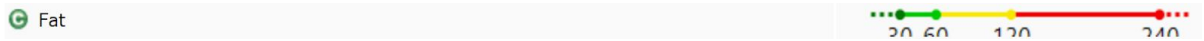


Figure 1: Seuil de valeur pour le critère FAT

De plus, on peut remarquer que la métrique CBO (couplage entre objets) est haute, avec une valeur de 41. La figure suivante démontre l'échelle de comparaison pour la métrique et on peut observer que la valeur pour la classe actuelle dépasse de 64% la valeur optimale pour une classe.

Figure 2: Seuil de valeur pour le critère CBO

Un couplage aussi élevé combiné à des métriques de complexité de la classe complètement absurdes démontre la présence sans équivoque d'une anomalie dans la conception de cette classe.

Anomalie

Cette anomalie est une représentation de la mauvaise odeur “Feature Envy”. Cette mauvaise odeur dicte qu'une classe utilise trop les méthodes d’une autre classe. Le couplage élevé ainsi que la faible cohésion causée par la trop grande complexité des méthodes sont des symptômes clé de cette mauvaise odeur.

2. JFreeChart

Justification

Cette classe possède beaucoup de métriques anormales. Le graphique suivant montre le rapport complet selon l’outil STAN pour cette classe.

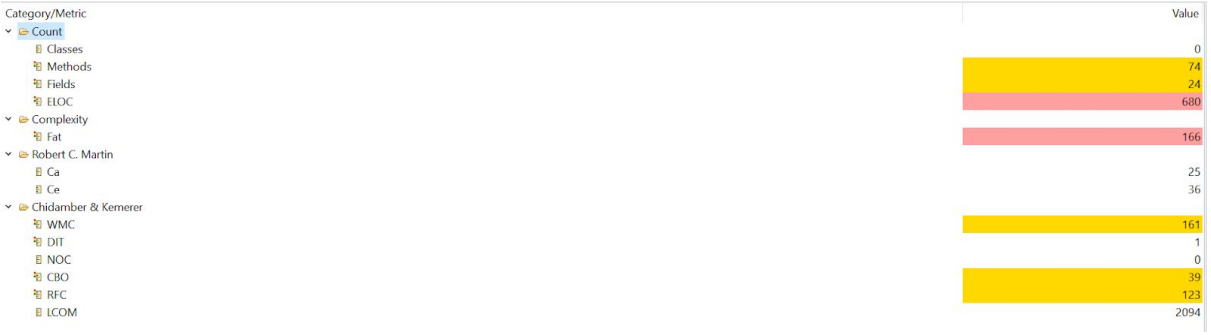


Figure 3: Rapport STAN pour la classe JFreeChart

Comme on peut voir, cinq métriques obtenues dépassent leurs seuils acceptables respectifs, soit la classe possède trop de méthodes (74), trop d’attributs (24), un couplage élevé (CBO de 39 et RFC de 123). De plus, deux métriques sont dangereusement élevés, soit la longueur de la classe (ELOC) ainsi que la métrique de complexité FAT. Un rapport comportant autant de métriques alarmantes pour une même classe pointe vers la présence d’une anomalie.

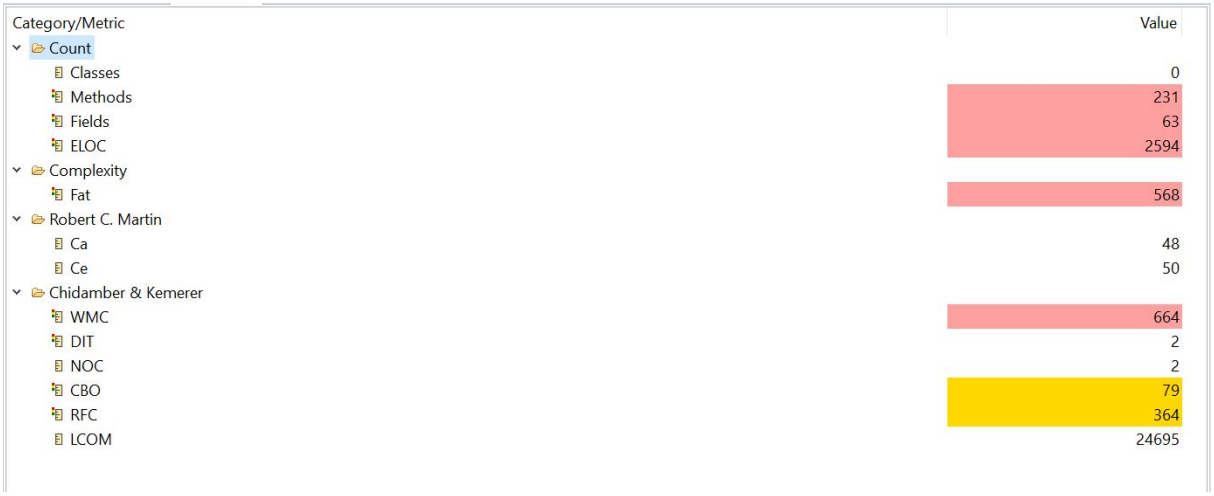
Anomalie

Ces métriques nous démontrent bien que cette classe est beaucoup trop longue et tente de faire trop de choses. Ces symptômes sont synonymes de la mauvaise odeur “God class”.

3. XYPlot

Justification

Cette classe comporte plus de cinq métriques dangereusement élevées comme le démontre le rapport effectué par l'outil STAN dans la figure suivante.



Category/Metric	Value
Count	
Classes	0
Methods	231
Fields	63
ELOC	2594
Complexity	
Fat	568
Robert C. Martin	
Ca	48
Ce	50
Chidamber & Kemerer	
WMC	664
DIT	2
NOC	2
CBO	79
RFC	364
LCOM	24695

Figure 4: Rapport STAN de la classe XYPlot

Effectivement, pratiquement toutes les métriques énonçant la complexité de la classe dépassent largement les valeurs acceptables, que ce soit longueur de la classe (2594 lignes de codes), le nombre de méthodes (231) ou le nombre d'attributs (63). Une telle complexité combinée à un fort couplage démontre un défaut dans la conception de cette classe.

Anomalie

L'anomalie dans cette classe est ainsi une *God Class*, c'est-à-dire la classe encapsule beaucoup trop de logique et contrôle beaucoup trop d'objets pour une seule classe.

4. Axis

Justification

Cette classe comporte trois métriques dépassant les seuils convenables selon l'outil STAN, comme le démontre la figure suivante.

Category/Metric	Value
Count	
Classes	0
Methods	78
Fields	42
ELOC	724
Complexity	
Fat	195
Robert C. Martin	
Ca	32
Ce	20
Chidamber & Kemerer	
WMC	156
DIT	1
NOC	2
CBO	30
RFC	111
LCOM	2208

Figure 5: Rapport STAN pour la classe Axis

Effectivement, la métrique du nombre d'attributs est beaucoup trop élevée (42) ainsi que le nombre de lignes de code exécutables (724), ce qui démontre une anomalie dans la classe lorsque combinée à un couplage élevé (CBO de 30) et une complexité à risque (RFC de 11).

Anomalie

Selon les métriques observées et de l'outil JDeodorant, on peut observer la présence de l'anomalie *Feature Envy*.

`5.TextTitle`

Justification

Comme le démontre la figure suivante, la très grande complexité de cette classe dénote une anomalie dans la conception de la classe.

Category/Metric	Value
Count	
Classes	0
Methods	37
Fields	13
ELOC	438
Complexity	
Fat	88
Robert C. Martin	
Ca	8
Ce	24
Chidamber & Kemerer	
WMC	112
DIT	3
NOC	2
CBO	20
RFC	78
LCOM	336

Figure 6: Rapport STAN pour la classe TextTitle

Anomalie

Grâce à l'outil JDeodorant et des métriques de complexité élevés, il est possible d'observer une anomalie de type *GodClass*, où la classe en question accomplie beaucoup trop pour une seule classe et que celle-ci devrait être subdivisée.

Question 3

1.

a) La mauvaise odeur "Feature Envy" sera corrigée par une refactorisation de type "Move method" par l'outil JDeodorant. Cette refactorisation aura pour conséquence de déplacer les méthodes comme applyToAbstractRenderer de la classe ***StandardChartTheme*** vers les classes comportant les données de ceux-ci, comme ***AbstractRenderer***. La conséquence de ce refactoring est une baisse du couplage, une hausse de la cohésion et la diminution de la complexité. Effectivement, des méthodes seront déplacées hors de la classe si nécessaire pour que la classe effectue un nombre réduit d'opérations sur d'autres objets.

b)

Avant:

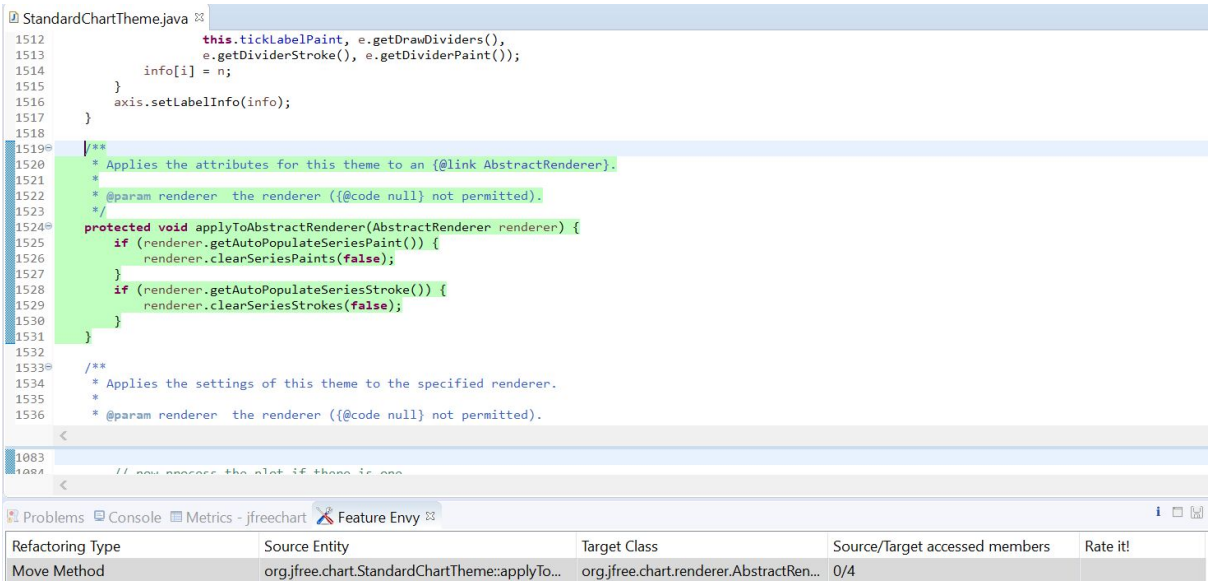


Figure : Classe StandardChartTheme initale

Après:

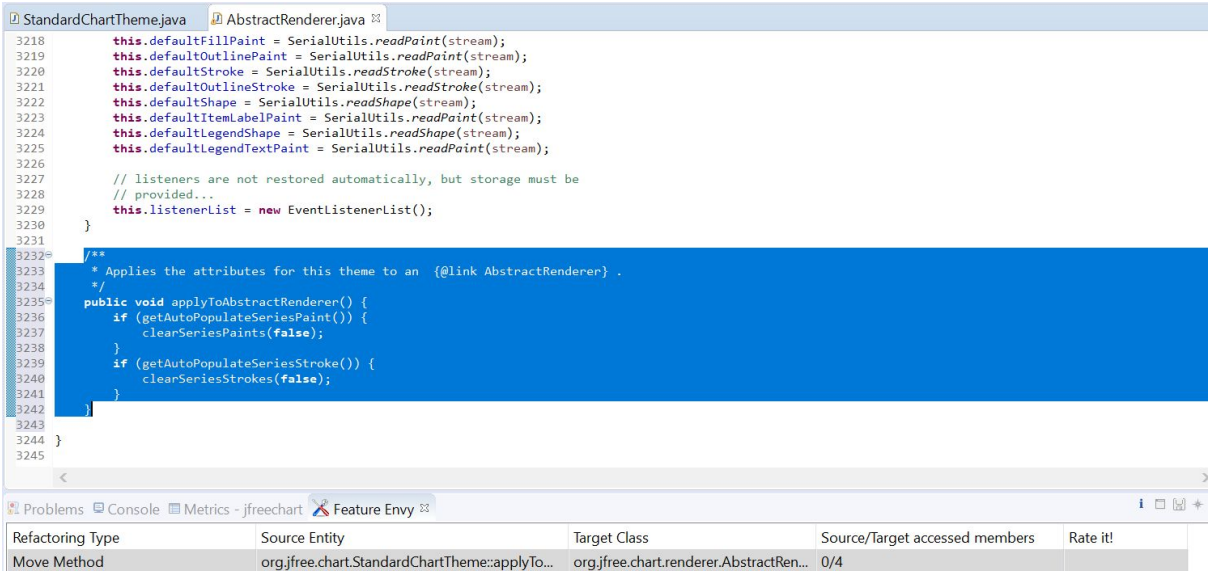


Figure : Classe StandardChartTheme après correction

c) Suite au refactoring, le CBO reste à 41, mais le FAT est désormais 285. Ce dernier est toujours critique selon l'échelle. On peut, ainsi, déterminer qu'il existe d'autres mauvaises odeurs pour cette classe.

Category/Metric	Value
Count	
Classes	0
Methods	90
Fields	32
ELOC	817
Complexity	
Fat	284
Robert C. Martin	
Ca	1
Ce	54
Chidamber & Kemerer	
WMC	194
DIT	1
NOC	1
CBO	41
RFC	219
LCOM	2727

2.

a) La mauvaise odeurl "God class" sera corrigé par une refactorisation de type "Extract Class" par l'outil JDeodorant pour ainsi diviser la logique trop importante dans la classe actuelle. Ainsi, en créant plusieurs classes hautement cohésives à partir de la classe initiale, la complexité diminuera.

b)

Avant:

	org.jfree.chart.JFreeChart		0/16
	[set]		
Extract Class		[set]	0/16
Extract Class		[listen]	0/3
Extract Class		[chang, listen]	0/1
Extract Class		[set]	1/17
Extract Class		[set]	1/13
Extract Class		[set, background]	1/9
Extract Class		[subtitl]	1/4
Extract Class		[border, paint]	1/3
Extract Class		[notifi]	1/2
Extract Class		[background, imag]	2/5
Extract Class		[set, background, imag]	2/4
Extract Class		[set]	2/3
	[listen]		
Extract Class		[listen]	0/1

Figure : Classe JFreeChart initale

Après:

Suite au refactoring par Jdeodorant, 2 classes ont été extraites: JFreeChartProduct.class et JFreeChartEventHandler.class. JFreeChartProduct rassemble plusieurs attributs tels que la bordure, l'espacement, l'arrière-plan, le titre, le sous-titre, ainsi que quelques méthodes qui modifient ces attributs. JFreeChartEventHandler, de son côté, contient les méthodes qui gèrent tous les évènements que JFreeChart comportait.

C) Voici les métriques selon STAN suite au refactoring:

Figure : Rapport STAN pour la classe JFreeChart suite aux corrections

Comme on peut le remarquer sur la figure ci-dessus, le nombre d'attributs à chuter grandement et est maintenant optimal pour la classe. De plus, le critère *Fat* est maintenant rendu raisonnable, ce qui indique que la complexité a bel et bien diminué pour la classe.

3.

La mauvaise odeur "God class" sera corrigé par une autre refactorisation de type "Extract Class" grâce à l'outil JDeodorant pour ainsi diviser la logique trop importante dans la classe actuelle. Ainsi, en créant plusieurs classes hautement cohésives à partir de la classe initiale, la complexité devrait diminuer.

Avant:

Figure : Classe XYPlot initiale

Après:

On extrait les classes XYPlotCrosshair, XYPlotCrosshairLock, XYPlotDomainAxis, XYPlotRangeAxis, XYPlotDomainGridlines, XYPlotRangeGridlines, XYPlotMarker de la classe initiale.

c)

Figure :Rapport STAN pour la classe XYPlot après corrections

En comparant les métriques obtenues à la question 2 et celles ci-dessus, on peut voir le refactoring avec les outils automatisés n'a pas été suffisant pour corriger les anomalies dans la classe. La taille de la classe étant de 2400 lignes de code, elle est très difficilement corrigée par un simple refactor. Pour corriger entièrement l'anomalie, il faudrait extraire et séparer manuellement encore plus de classes de la classe principale. Bien que très peu efficace, on peut quand même observer des améliorations sur les différentes métriques obtenues.

4- Axis

a) Le *code smell* Feature Envy sera corrigé par une refactorisation de type “Move method” par l’outil JDeodorant. Cette refactorisation aura pour conséquence de déplacer les méthodes comme labelLocation de la classe Axis vers les classes comportant les données de ceux-ci, comme AxisLabelLocation. On verra ainsi une baisse du couplage et une hausse de la cohésion. Effectivement, en faisant l’extraction de méthode de cette façon, une partie de la grande complexité de la classe devrait ainsi quitter la classe initiale.

b)

Avant:

Axis.java

```
1273        return result;
1274    }
1275
1276    protected double labelLocationX(AxisLabelLocation location,
1277            Rectangle2D dataArea) {
1278        if (location.equals(AxisLabelLocation.HIGH_END)) {
1279            return dataArea.getMaxX();
1280        }
1281        if (location.equals(AxisLabelLocation.MIDDLE)) {
1282            return dataArea.getCenterX();
1283        }
1284        if (location.equals(AxisLabelLocation.LOW_END)) {
1285            return dataArea.getMinX();
1286        }
1287        throw new RuntimeException("Unexpected AxisLabelLocation: " + location);
1288    }
1289
1290    protected double labelLocationY(AxisLabelLocation location,
1291            Rectangle2D dataArea) {
1292        if (location.equals(AxisLabelLocation.HIGH_END)) {
1293            return dataArea.getMinY();
1294        }
1295        if (location.equals(AxisLabelLocation.MIDDLE)) {
1296            return dataArea.getCenterY();
```

Problems | Console | Feature Envy | Type Checking

Refactoring Type	Source Entity	Target Class	Source/Target a...	Rate it!
Move Method	org.jfree.chart.axis.Axis::labelLocationX(org.jfree.chart.axis.Axi...	org.jfree.chart.axis.AxisLabelLocation	0/1	
Move Method	org.jfree.chart.axis.Axis::labelLocationY(org.jfree.chart.axis.Axi...	org.jfree.chart.axis.AxisLabelLocation	0/1	
Move Method	org.jfree.chart.axis.Axis::labelAnchorH(org.jfree.chart.axis.Axis...	org.jfree.chart.axis.AxisLabelLocation	0/1	
Move Method	org.jfree.chart.axis.Axis::labelAnchorV(org.jfree.chart.axis.Axis...	org.jfree.chart.axis.AxisLabelLocation	0/1	

Figure : Classe Axis iniale

Apres:

Axis.java AxisLabelLocation.java

```
144
145         return null;
146     }
147
148     public double labelLocationX(Rectangle2D dataArea) {
149         if (equals(AxisLabelLocation.HIGH_END)) {
150             return dataArea.getMaxX();
151         }
152         if (equals(AxisLabelLocation.MIDDLE)) {
153             return dataArea.getCenterX();
154         }
155         if (equals(AxisLabelLocation.LOW_END)) {
156             return dataArea.getMinX();
157         }
158         throw new RuntimeException("Unexpected AxisLabelLocation: " + this);
159     }
160
161     public double labelLocationY(Rectangle2D dataArea) {
162         if (equals(AxisLabelLocation.HIGH_END)) {
163             return dataArea.getMinY();
164         }
165         if (equals(AxisLabelLocation.MIDDLE)) {
166             return dataArea.getCenterY();
167         }
```

Figure : Classe Axis après corrections

c)

Category/Metric	Value
Count	
Classes	0
Methods	74
Fields	42
ELOC	670
Complexity	
Fat	187
Robert C. Martin	
Ca	32
Ce	20
Chidamber & Kemerer	
WMC	140
DIT	1
NOC	2
CBO	30
RFC	111
LCOM	1926

Figure : Rapport STAN pour la classe Axis après corrections

Encore une fois, le refactoring a amélioré les résultats des métriques correspondants à l'anomalie, sans toutefois les corrigées complètement. On peut donc déduire qu'il faudrait continuer le travail de refactoring et chercher pour d'autres anomalies potentielles dans cette classe.

5-

a) La mauvaise odeur "God class" a été corrigée par une refactorisation de type "Extract Class" à l"aide de l'outil *JDeodorant*. Comme pour les autres anomalies de ce type corrigées précédemment, la classe ayant l'anomalie a été séparée en plusieurs sous-classes afin de diminuer les responsabilités et la taille de la classe

afin d'y augmenter sa cohésion. Certaines parties de la classe TextTitle ont donc été extraites dans deux nouvelles classes: TextTitleSettings et TextTitleAttachedContent. Avec cette modification, la taille des classes est diminuée. De plus, chaque classe se retrouve avec des responsabilités réduites et sont donc plus cohésives. L'anomalie devrait donc s'en trouver réglée. Par contre, même après les corrections, la classe reste encore trop longue (métrique ELOC) et la complexité est supérieure à la normale. Il faudrait peut-être tenter de la séparer manuellement davantage puisque *JDeodorant* ne propose pas plus de solutions ou chercher la présence d'un deuxième anti patron.

b)

Avant:

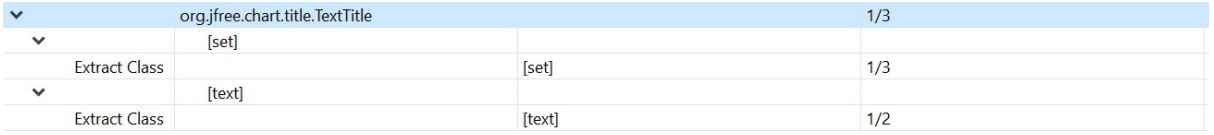


Figure : Refactorisation de la classe TextTitle avec JDeodorant

Apres

c)

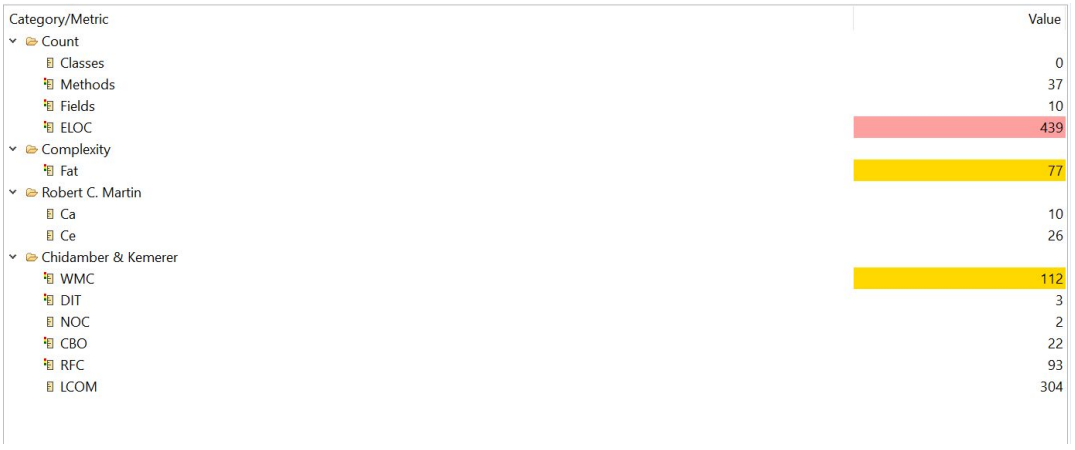


Figure : Rapport STAN pour la classe TextTitle après corrections

d)

Tableau 6 : Attributs de qualités selon l'anomalie détectée

	Avant	**Apres**	**Avant**	**Apres**	**Avant**	**Apres**
	1	1	2	2	3	3
Effectiveness	0.8556	0.4472	0.4472	0.4472	0.4472	0.4472
Extendibility	-58.86	-58.86	1.118	1.118	1.118	1.118
Flexibility	-29.25	-29.25	0	0	0	0
Functionality	357.62	357.62	342.32	342.32	342.32	342.32
Reusability	782.75	782.75	778.0	778.0	778.0	778.0
Understandability	-583.52	-583.52	-675.13	-514.22	-701.68	-514.22

1: StandardChartTheme, 2 : JFreeChart, 3 : XYPlot

	Avant	**Apres**	**Avant**	**Apres**
	4	4	5	5
Effectiveness	0.4472	0.4472	0.4472	0.4472
Extendibility	1.118	1.118	1.118	1.118
Flexibility	0	0	0	0
Functionality	342.32	342.32	342.32	342.32
Reusability	778.0	778.0	778.0	778.0
Understandability	-589.91	-514.22	-514.22	-514.22

4: Axis, 5 : TextTitle

En général, les métriques correspondantes aux critères de qualité, obtenues par l'outil ptidej, ont restées sensiblement les mêmes avec quelques améliorations au niveau du critère *understandability*. Ce phénomène s'explique principalement par les faibles changements dans les métriques individuelles suite aux refactoring. De plus, bien que la complexité de plusieurs classes a diminuée, les attributs de qualité dépendent de plusieurs métriques non affectées directement par nos refactoring, par exemple le polymorphisme, l'abstraction, la composition, la taille du *design,* etc. Effectivement, comme le démontre

l'article A Hierarchical Model for Object-Oriented Design Quality Assessment[1], la complexité et le couplage n'ont pas d'effet direct sur les attributs de qualité et ainsi nos refactorisations de type *extract class* et *move method* ont raison de ne pas influencer les valeurs des métriques de qualités.

[1]J.Bansiya, C.G.Davis, (Janvier 2002), Hierarchical Model for Object-Oriented Design Quality Assessment, vol.28, no.1.