LOG8430

Architecture logicielle et conception avancée

Automne 2018

TP1

Principes et patrons de conception

Présenté par :

Alexandre Clark (1803508)

David Tremblay (1748125)

Félix Agagnier (1795792)

21 septembre 2018

École Polytechnique de Montréal

Département de Génie Informatique et Logiciel

Question 1

Trouvez trois (3) instances différentes des patrons de conception dans JFreeChart

1. Patron Singleton

Diagramme

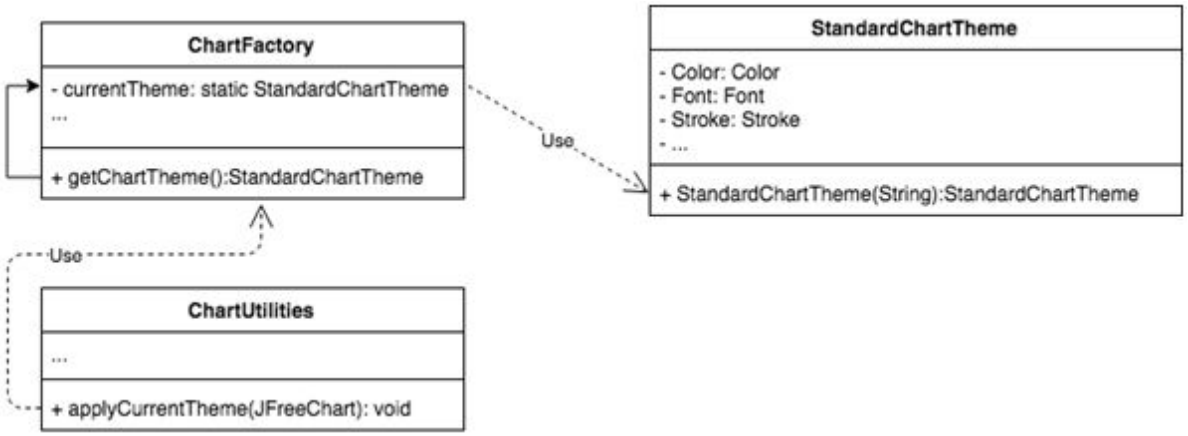

Figure 1: Patron Singleton dans JFreeChart

Fonctionnalité

Le patron singleton est utilisé dans le cas suivant pour s'assurer de posséder une seule et unique instance du thème ***StandardChartTheme***. Le choix de ce patron est logique puisque le but principal d'un thème est de définir des propriétés qui seront partagées et utilisées à travers la solution entière. En ayant une seule instance du thème, on s'assure de l'unicité du thème.

La classe ***ChartFactory*** initialise une instance privée et statique de la classe ***StandardChartTheme***, puis chaque fois que le thème est nécessaire, on lui accède à l'aide de la méthode *getChartTheme( )* exposée publiquement par la factory.

Explication

Tout d'abord, un objet ***ChartTheme*** est instancié de manière privée et statique en tant qu'attribut de la classe. Cette variable est instanciée à l'aide du constructeur de l'objet ***StandardChartTheme*** avec comme paramètre la valeur

"*JFree*" qui correspond au nom du thème par défaut dans JFreeChart. Cette variable peut être ensuite obtenue à l'aide de la méthode publique *getChartTheme*(). Cette méthode est utilisée dans la méthode *applyCurrentTheme()* qui se trouve dans la classe ***ChartUtilities***. De cette manière, on s'assure que le thème retourné de ***ChartFactory*** est toujours unique. Un exemple d'utilisation de ce patron serait tout simplement de faire appelle à *applyCurrentTheme()* sur un graphique afin d'y appliquer le thème courant dans ***ChartFactory***.

Il est à noter que le patron Singleton utilisé dans cette classe est différent du patron classique. Une amélioration pertinente à apporter au code serait de ne pas instancier la variable statique *currentTheme* au départ et de plutôt toujours passer par la méthode *getChartTheme*() qui vérifierait si la variable *currentTheme* serait de valeur nulle et dans ce cas, elle en créerait une instance.

2- Patron Factory

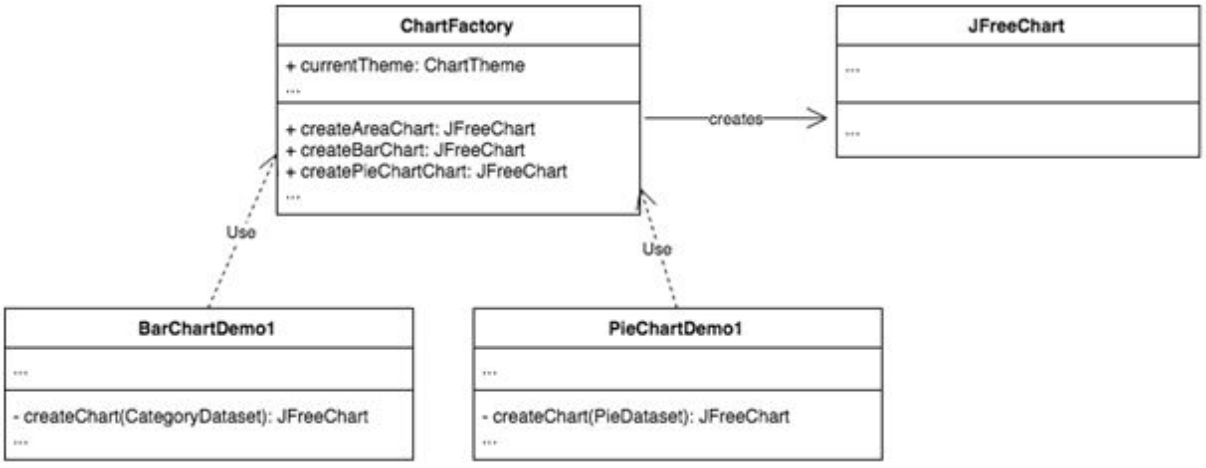

Figure 2: Patron Factory dans JFreeChart

Fonctionnalité

Le patron factory fournit une interface pour créer des objets ***JFreeChart***, en laissant la décision du type d'objet à créer aux sous-classes utilisant la factory. Ainsi, les sous-classes peuvent créer divers types de diagrammes, notamment des *AreaChart, BoxChart et PieChart*. Le but d'utiliser ce patron est d'encapsuler toute

la logique de création des diagrammes dans une classe *factory* qui évite de devoir implémenter la création de chaque type de diagrammes dans chacun des modules qui crée des diagrammes. On réduit ainsi le couplage entre les différents types de diagrammes et les classes les créant, facilitant ainsi l'ajout de types de diagramme.

Ainsi, les classes ***BarChartDemo1*** et ***PieChartDemo1*** utilisent respectivement les fonctions *createBarChart* et *createPieChart* en accédant à la classe ***ChartFactory***, créant ainsi les bons diagrammes ***JFreeChart***.

Explication

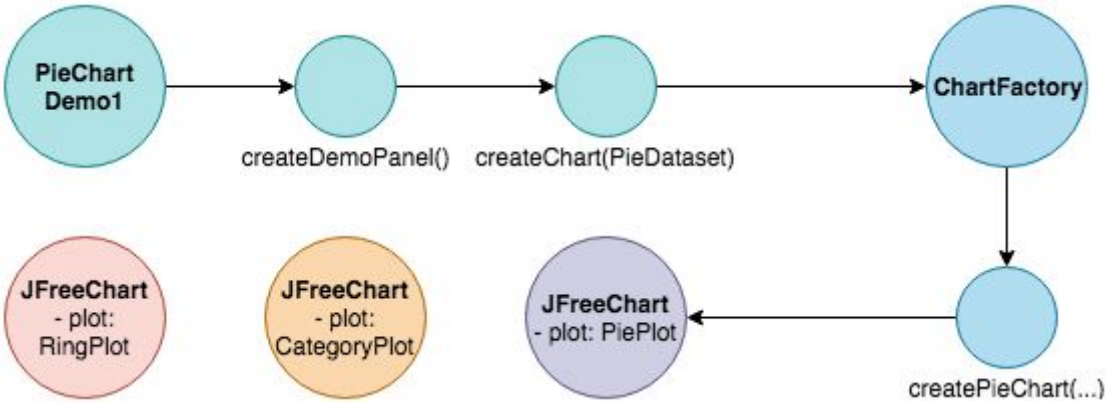

Figure 3: Exemple de cas d'utilisation du patron Strategy

Comme le démontre la figure ci-dessus, différentes démos comme la classe ***PieChartDemo1*** initialisent leurs diagrammes grâce à la factory plutôt que d'accéder directement à un constructeur de la classe ***JFreeChart***. Ainsi, dépendamment du créateur et de son type de graphique, différents diagrammes peuvent être créés avec des graphiques (*plot*) différents.

3- Strategy

Diagramme

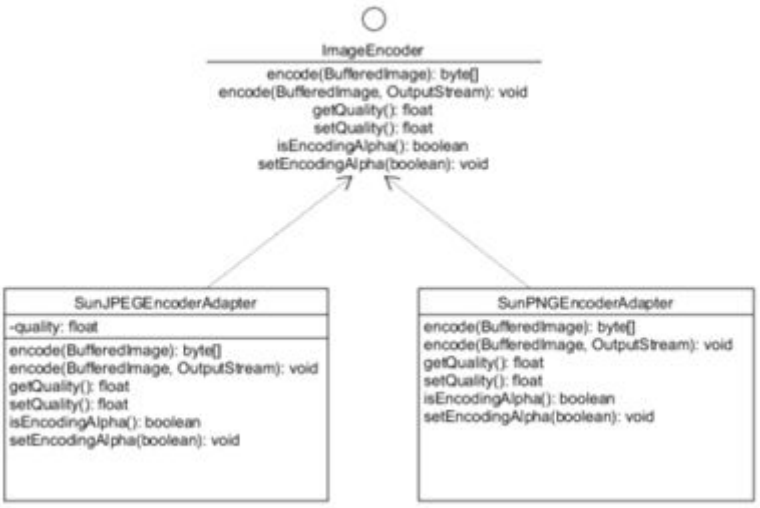

Figure 4: Patron Strategy dans JFreeChart

Fonctionnalité

Le but du patron strategy est d'encapsuler les différentes implémentations des fonctions d'encodage dans différentes classes et ainsi lorsqu'une image a besoin d'être encodée, on utilise la bonne stratégie pour encoder selon le bon type d'image. Ce patron est très utile dans la situation actuelle, puisque cela nous permet de découpler les différentes implémentations d'encodage et de facilement retirer ou ajouter un encodeur. Cela permet d'autant plus de diminuer la complexité cyclomatique des différentes méthodes d'encodage et d'améliorer la maintenabilité du code.

Dans la situation actuelle, les classes ***SunJPEGEncoderAdapter*** et ***SunPNGEncoderAdapter*** implémentent les méthodes de la classe ***ImageEncoder***. La classe ***ImageEncoderFactory*** permet ensuite d'instancier le bon encodeur pour un type donné dans des sous-classes telles ***EncoderUtil***.

Explication

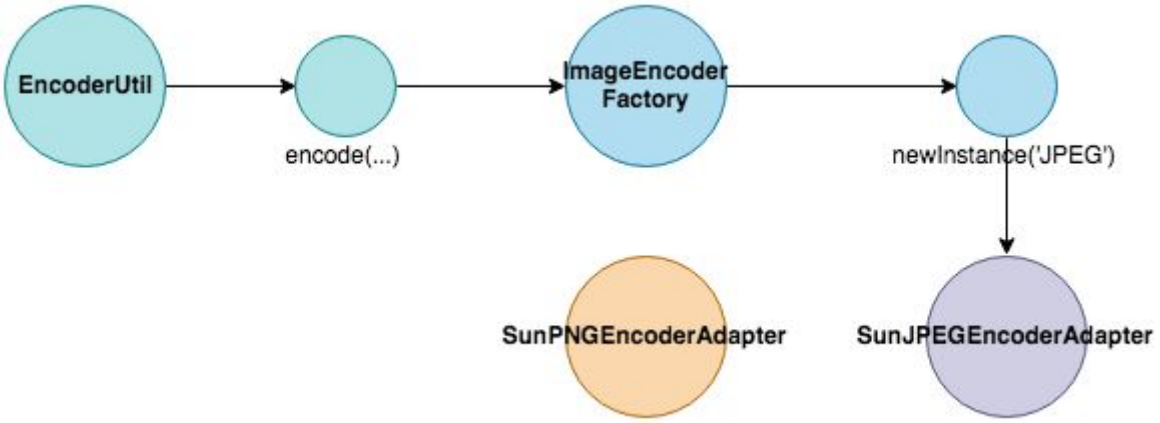

Figure 5: Cas d'utilisation du patron Strategy

Comme le démontre la figure ci-dessus, en utilisant la classe utilitaire ***EncoderUtil***, lors de l'encodage d'une image, un objet ***ImageEncoder*** est instancié et les méthodes nécessaires pour l'encodage du type de fichier sont explosées. Dans l'exemple ci-contre, puisque le type du fichier est '*JPEG*' l'objet instancié sera un ***SunJPEGEncoder*** et le bon encodage sera fait par l'utilitaire.

Question 2

Trouvez trois (3) instances différentes des principes SOLID dans JFreeChart

1- Interface Segregation Principle

Diagramme

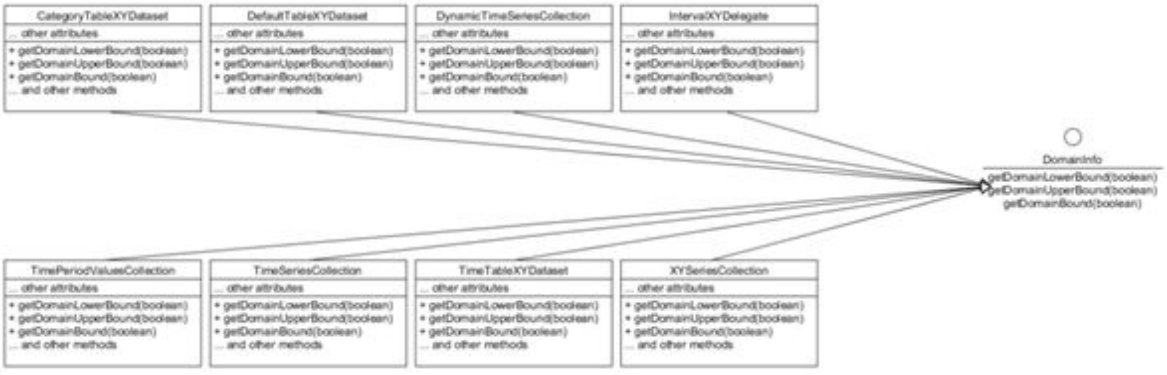

Figure 6: Principe ISP dans JFreeChart

Importance

Le principe de ségrégation des interfaces dicte qu'avoir plusieurs interfaces est supérieur à avoir une seule interface. En pratique ce principe, suggère qu'une classe qui implémente cette interface doit avoir besoin de toutes les méthodes de l'interface et que toute méthode non utilisée est une violation du principe.

Dans ce cas-ci, on peut remarquer que ***DomainInfo*** est une petite interface de 3 méthodes. De plus, on peut remarquer que chaque implémentation utilise chaque méthode et que chaque implémentation est une utilisation correcte de la méthode, c'est-à-dire, les méthodes, par exemple, ne lancent pas d'exception du genre "cette méthode ne devrait pas être appelée".

Rôles

Domain: Interface qui expose 3 méthodes: *getDomainLowerBound*, *getDomainUpperBound*, *getDomainBounds*.
DynamicTimeSeriesCollection: Classe qui implémente ***DomainInfo*** et qui implémente les méthodes de celle-ci.
TimePeriodValuesCollection: Classe qui implémente ***DomainInfo*** et qui implémente les méthodes de celle-ci.
TimeSeriesCollection: Classe qui implémente ***DomainInfo*** et qui implémente les méthodes de celle-ci.
TimeTableXYDataset: Classe qui implémente ***DomainInfo*** et qui implémente les méthodes de celle-ci.
CategoryTableXYDataset: Classe qui implémente ***DomainInfo*** et qui implémente les méthodes de celle-ci.
DefaultTableXYDataset: Classe qui implémente ***DomainInfo*** et qui implémente les méthodes de celle-ci.
IntervalXYDelegate: Classe qui implémente ***DomainInfo*** et qui implémente les méthodes de celle-ci.

XYSeriesCollection: Classe qui implémente ***DomainInfo*** et qui implémente les méthodes de celle-ci.

2- Liskov Substitution Principle

Diagramme

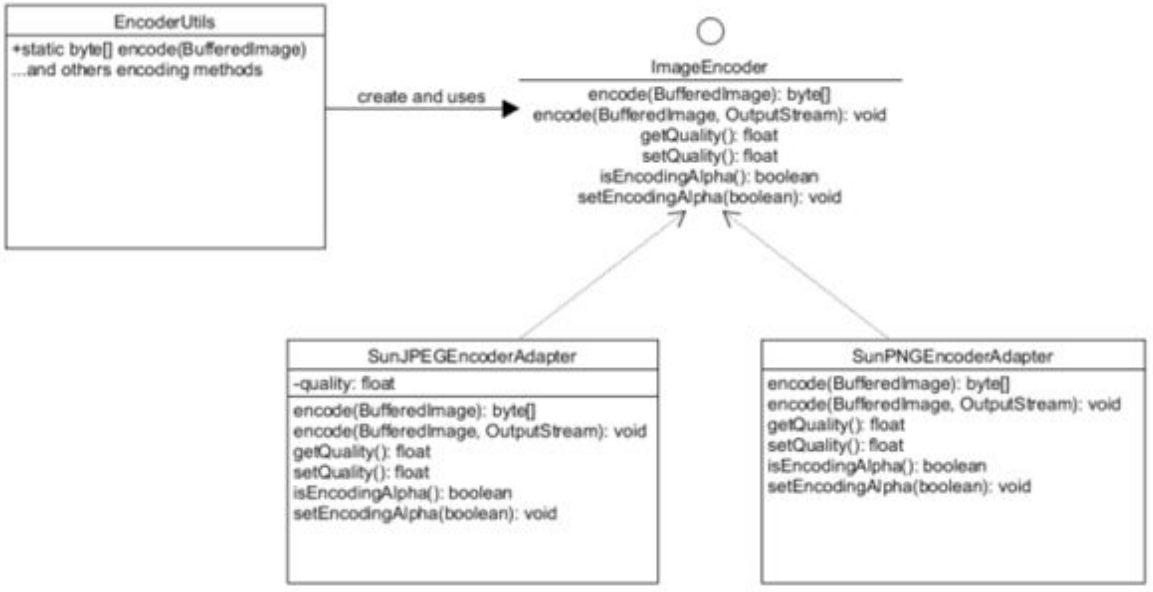

Figure 7: Principe Substitution dans JFreeChart

Importance

Le principe de substitution de Liskov dicte qu'on doit pouvoir utiliser les méthodes d'un objet de base sur les classes dérivées et s'attendre à un comportement similaire et ce, sans que l'objet appelant voie la différence.

Ainsi, dans ce cas-ci, peu importe l'implémentation de ***ImageEncoder*** et pour toutes les méthodes on s'attend à un comportement similaire. Par exemple, autant pour ***SunJPEGEncoderAdapter*** et ***SunPNGEncoderAdapter***, on s'attend à ce que la méthode encode prenne un objet *BufferedImage*, écrive celle-ci dans un *OutputStream* et retourne les octets de ce dernier.

Rôles

ImageEncoder: classe de base, interface dans ce cas.
SunJPEGEncoderAdapter: classe dérivée de ***ImageEncoder***
SunPNGEncoderAdapter: classe dérivée de ***ImageEncoder***
EncoderUtils: crée une instance de ***ImageEncoder*** et appelle les méthodes pour encoder une image. Cette classe ne sait pas si l'image est un PNG ou un JPEG.

3- Single Responsibility Principle

Diagramme

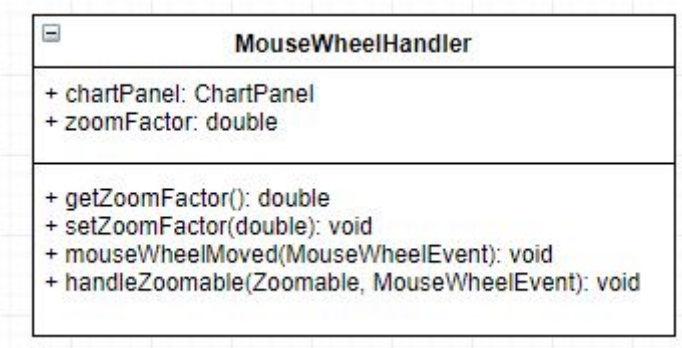

Figure 8: Principe SRP dans JFreeChart

Importance

Le principe SOLID de responsabilité unique stipule qu'une classe ne devrait avoir qu'une seule responsabilité ou en d'autres mots, une seule raison de changer. Une classe qui respecte ce principe possède donc forcément une haute cohésion ainsi qu'un faible couplage.

Ce principe est important dans une architecture logicielle telle que JFreeChart puisqu'il assure la robustesse des classes. En effet, imaginons un instant que cette classe soit aussi responsable des évènements du clavier. Un changement dans les méthodes du clavier pourrait affecter les évènements de la roulette de souris puisque les deux responsabilités seraient groupées dans la même

classe. Le principe de responsabilité unique limite donc l'effet d'un changement sur le reste du code.

Rôles

Tout d'abord, la classe **MouseWheelHandler** n'utilise que 4 autres classes (*PiePlot, Plot, PlotRenderingInfo et Zoomable*) ainsi que 2 attributs (*chartPanel* et *zoomFactor*). Elle est donc fortement cohésive et possède un couplage faible. Maintenant, à ce qui a trait aux méthodes de la classe, il y a une méthode pour obtenir et modifier le zoom, une méthode qui réagit aux mouvements de la roulette de souris ainsi qu'une méthode qui gère les types de graphiques qui possèdent l'interface Zoomable. Étant donné que la roulette de souris sert au zoom dans JFreeChart, on peut dire que toutes ces méthodes respectent le principe de responsabilité unique qui est de gérer les évènements liés à la roulette de souris comme le nom de la classe le suggère.

Question 3

Trouvez une violation des principes SOLID dans JFreeChart

Diagramme

Figure 9: Mauvaise odeur Large class dans StandardChartTheme

La classe **StandardChartTheme** obtient une détection de 100% avec l'outil *ptidej* pour la détection de l'anti-patron "Large Class".

Explication

La classe possède près de 1800 lignes de codes. Il est fort probable que celle-ci puisse être décomposée en plusieurs sous-classes. Ainsi, la lisibilité et la maintenance du code s'en trouveraient améliorée. Une classe aussi grande ne respecte pas le premier principe SOLID qui stipule qu'une classe ne doit avoir qu'une seule responsabilité. On remarque que la classe ***StandardChartTheme*** est toutefois en charge de créer les thèmes (*createDarknessTheme*() par exemple),

d'appliquer ces thèmes aux différents types de graphiques (*applyToPiePlot*() par exemple) et d'obtenir ou de modifier les différents éléments d'un graphique (*setTitlePaint*() par exemple). De plus, toutes les méthodes *applyTo*[..]() brise le principe *Open/Close*, car la structure interne de l'objet passé en paramètre est exposée au thème. Ceci est facile à détecter, puisqu'on peut remarquer qu'il y a beaucoup de *setters* qui sont utilisés. Finalement, on constate qu'il y a 66 classes utilisées dans cette classe ainsi que 33 attributs indiquant un haut couplage et une faible cohésion.

Correction

Un moyen de corriger cette classe est de séparer des groupes d'attributs, par exemple largeFont, smallFont, regularFont en classes, par exemple dans ce cas-ci FontTheme. Les méthodes applyTo[...]([...]) devrait être "inversés" de façon à avoir, par exemple *plot.applyTheme*(thème) au lieu de *applyToPlot*(plot). Ceci permet de tirer avantage du polymorphisme et d'éviter tous les instanceof qu'on retrouve dans les méthodes, ainsi que d'éviter de devoir créer une nouvelle méthode à chaque fois qu'on ajoute une nouvelle classe qui utilise le thème. De plus les méthodes *create*[...]Theme, peuvent être extraites dans une factory et les valeurs serait passé en paramètre dans le constructeur. En appliquant ces principes, tous ce qui va rester dans ***StandardChartTheme*** sont les attributs, les getters et les setters, les méthodes *equals(), clone(), writeObject()* et *readObject()*.