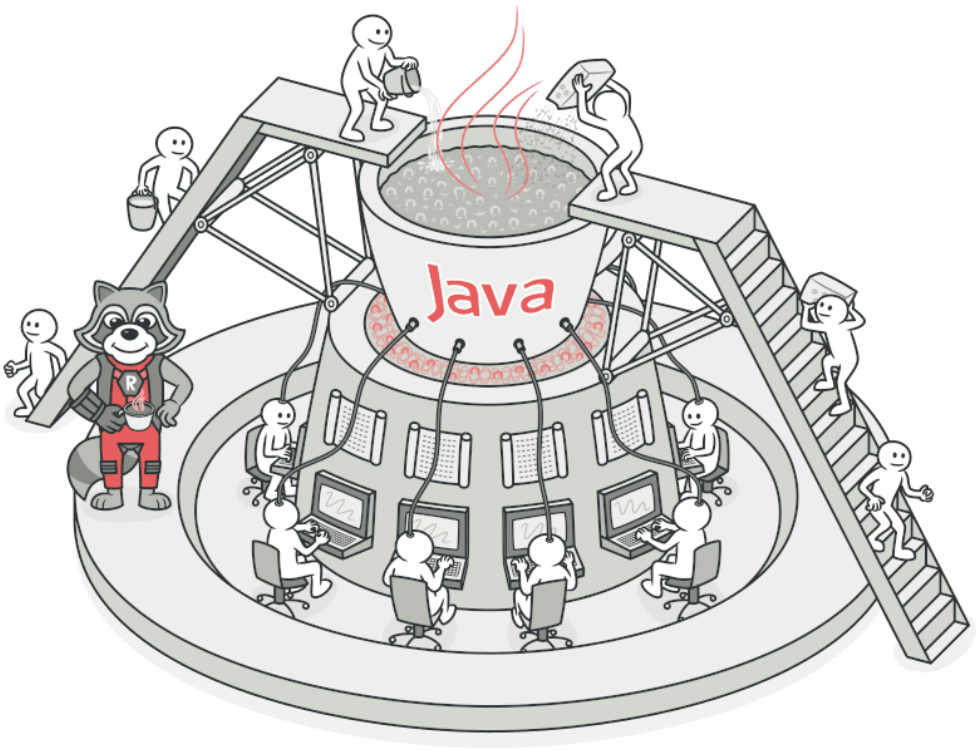


Rapport de Design Pattern

UGE Paint

Léo Barroux

Sommaire

1. Introduction

Ce rapport entre dans le contexte du cours “Design Pattern”, étudié en 3ème année à l’ESIPE. J’ai ainsi eu l’occasion de développer une application graphique, répondant aux problématiques SOLID de la conception de code.

SOLID est un ensemble de bonnes pratiques que chaque bon développeur se doit de respecter s’il souhaite que son application soit efficace viable dans le temps. Ci-dessous un bref rappel des principes SOLID :

- **S**ingle responsibility principle : Une méthode ne doit pas faire 2 actions différentes, une classe ne doit pas répondre à 2 problématiques différentes. Une classe ne doit pas être impactée par deux aspects différents de la spécification du programme. On applique ici le principe de composition et délégation, je stocke un objet et je lui laisse faire le travail.

- **O**pen/closed principle : Une classe doit être fermée aux modifications mais ouverte aux extensions. Les extensions se font en ajoutant des classes et surtout pas en modifiant le code existant. On doit dans notre cas par exemple assurer la possibilité de rajouter une librairie graphique en y ajoutant un adaptateur, sans avoir à modifier les classes déjà existantes. On parle alors de *Design Pattern Strategy*.

- **L**iskov substitution principle : Ce principe parle en majeure partie de l'héritage, mais de également de tous les sous-types. Lorsque l’on fait de l'héritage, on doit garantir un certain nombre de propriétés. Tout ce qui était vrai avec un type A doit être vrai avec un sous-type B.

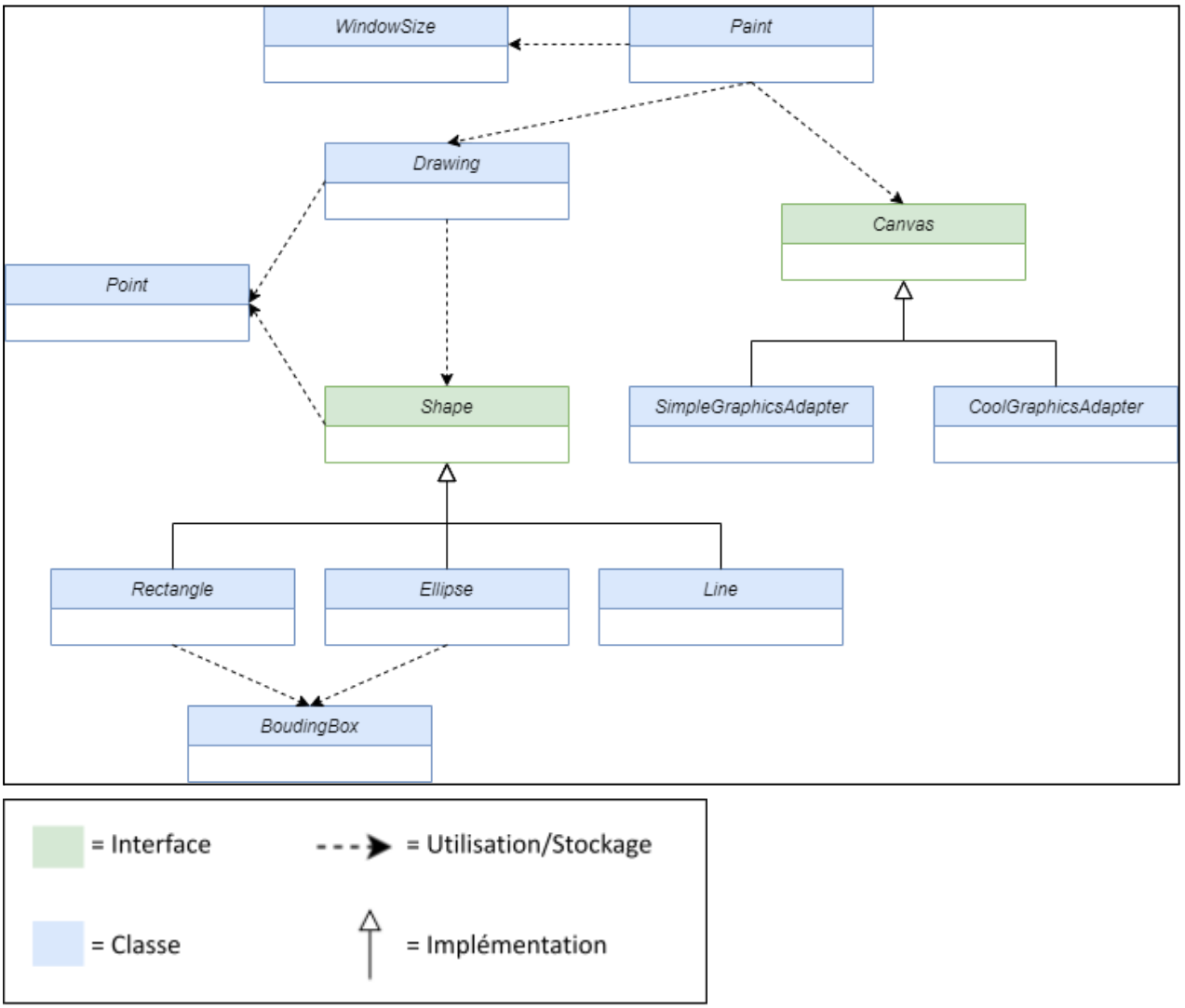
- **I**nterface segregation principle : Ne pas faire d'interface fourre-tout, qui ferait tout et n’importe quoi. Une interface est avant tout un contrat, qui ne doit s’occuper que d’une tâche.

- **D**ependency inversion principle : Les classes de haut niveau ne devraient pas avoir à être modifiées lorsqu'une classe de bas niveau est modifiée. Les classes de haut niveau doivent définir une abstraction à laquelle se conforme la classe de bas niveau.

Je présenterai dans un premier temps l’architecture de mon code, séparée non pas par package mais par parties que j’aurais moi-même définies afin d’expliquer de manière plus efficace mon code. Je mettrai ensuite en parallèle les choix techniques de mon architecture et les principes SOLID vus précédemment. J’essaierai ensuite d’apporter une solution aux problématiques qui nous ont été données. Je finirai par présenter la solution que j’ai mise en place pour l’exercice 9 du sujet.

2. Architecture du code

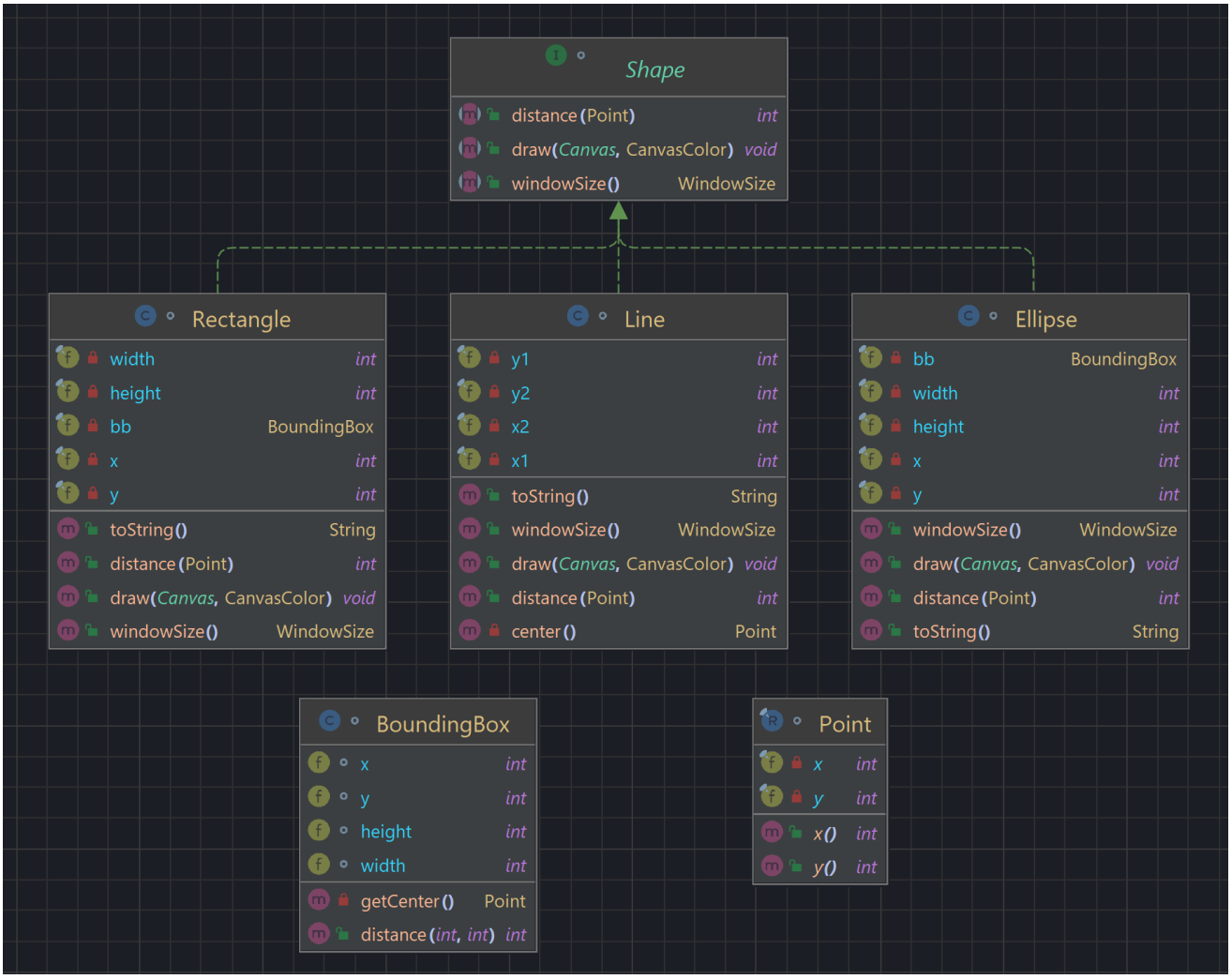
2.1. Présentation globale



Le schéma ci-dessus représente l'ensemble de l'architecture du projet. On peut tout d'abord voir qu'il n'y a pas de notion d'héritage dans l'architecture, uniquement d'implémentation pour les deux interfaces Canvas et Shape.

Les parties suivantes permettront de comprendre plus précisément comment l'architecture fonctionne et quelles sont les responsabilités de chaque classe et interface.

2.2. Partie *Figures*



Cette partie "*figures*" concerne, comme son nom l'indique, tout ce qui tourne autour des formes. Nous allons ainsi retrouver une interface "Shape", comportant 3 méthodes non default. Cette interface va ensuite être implémentée par les 3 classes "Ellipse", "Rectangle" et "Line". Jusqu'ici, rien de très compliqué.

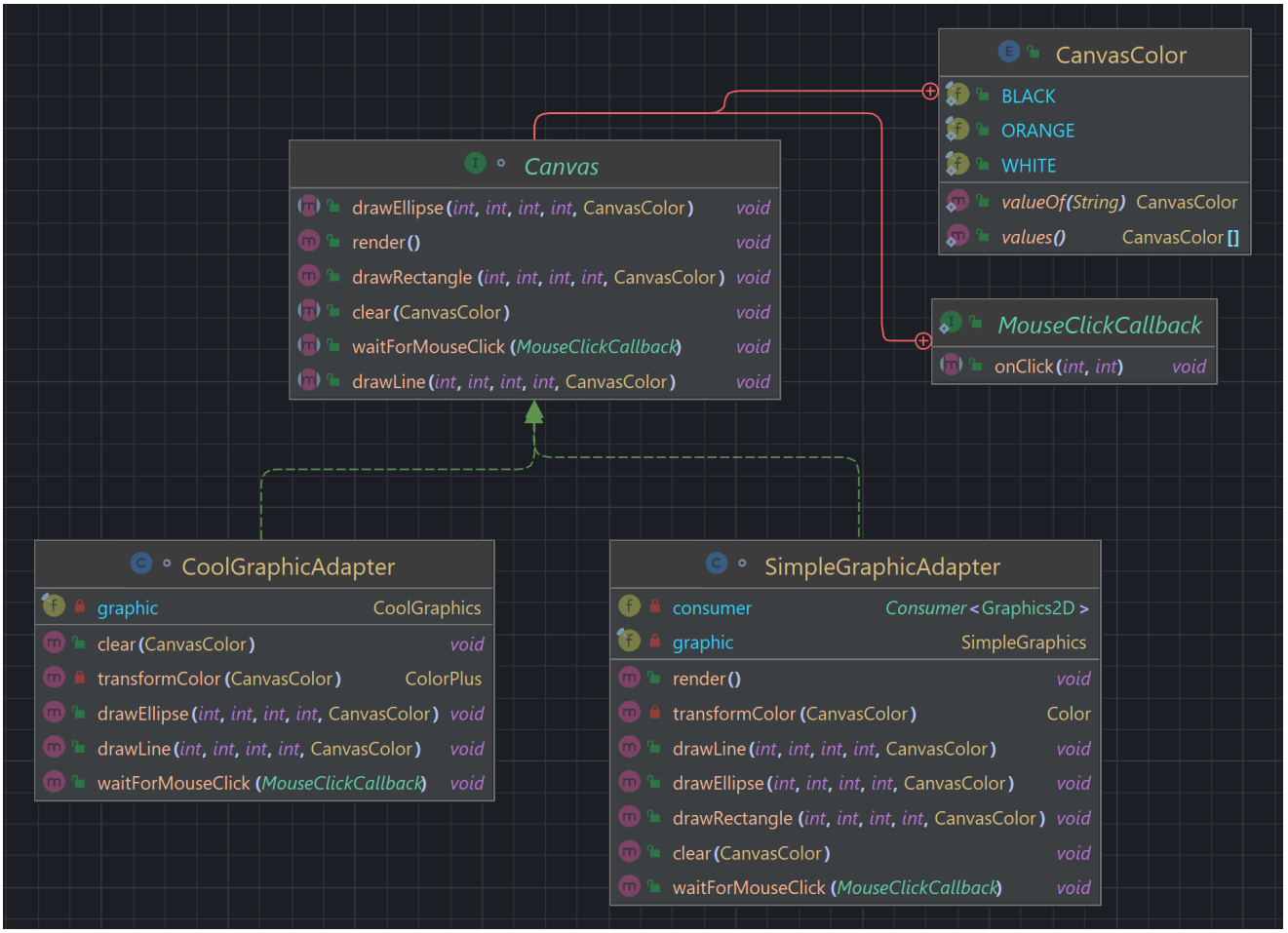
Le problème ici est que le code entre Rectangle et Ellipse était similaire pour le calcul de distance (méthode distance()) et la méthode getCenter(). J'ai tout d'abord créé une classe abstraite nommée "AbstractRectangleEllipse", mais j'ai fini par la retirer pour la classe "BoudingBox". Cette classe permet ainsi de calculer la distance de la même façon pour les ellipse et les rectangles, pour des coordonnées données. Ainsi, les classes Rectangle et Ellipse ont en champ une BoundingBox, créé lors de l'appel au constructeur et le code est ainsi factorisé de manière plus efficace. Si de nouvelles formes venaient à être créées, et que le calcul de distance() était le même que pour un rectangle par exemple, cette nouvelle forme n'aurait qu'à stocker une BoundingBox en champ également.

L'interface Shape est *sealed*, ce qui signifie que seules les classes autorisées (ici les 3 classes de forme) peuvent l'implémenter. Cela permet de ne pas avoir une interface pouvant être

implémentée par n'importe qui, et comme en tant que développeur je suis le seul à choisir quelles formes sont implémentées dans l'application UgePaint, il est plus simple que je choisisse quelle classe peut implémenter l'interface.

De plus, l'ensemble des classes présentées ici sont en visibilité package, car inutile à fournir dans notre cas à un utilisateur de notre librairie.

2.3. Partie *Dessin*



Cette partie est sûrement la plus complexe en termes de code. Ce sont ces classes qui interagissent avec les librairies graphiques "SimpleGraphics" et "Coolgraphics".

Initialement, seule la librairie SimpleGraphics était présente, toutes ces classes n'existaient donc pas et les fonctions de SimpleGraphics étaient directement appelées dans la classe "Drawing". Cependant, lors du rachat de la part de l'entreprise EvilCorp, une nouvelle librairie CoolGraphics a été créée, et nous avons dû proposer aux utilisateurs quelle librairie ils souhaitaient utiliser.

J'ai donc naturellement créé une nouvelle interface "Canvas", avec les 2 implémentations "SimpleGraphicAdapter" et "CoolGraphicAdapter". Cette fois, l'interface Canvas est en visibilité publique, car tous les utilisateurs de ma librairies doivent pouvoir ajouter leur

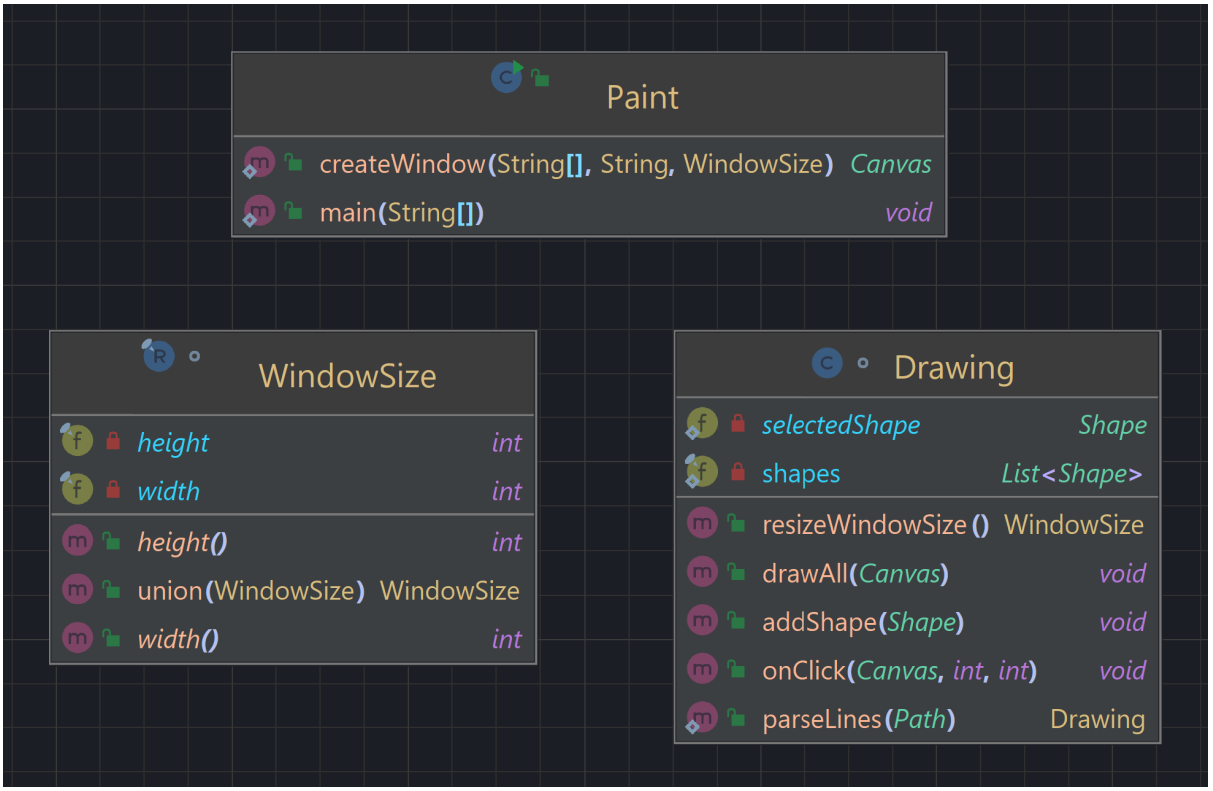
propre librairie graphique s'ils le souhaitent, et cela n'empêchera en aucun cas le code déjà existant de fonctionner. Les classes SimpleGraphicAdapter et CoolGraphicAdapter sont elles bien évidemment en visibilité package, car utiles uniquement pour mon propre code.

Lors de la modification d'UgePaint pour cette version, de nombreux problèmes se sont levés, à savoir ceux de la couleur et ceux du clic souris.

Pour la couleur, les librairies CoolGraphics et SimpleGraphics n'utilisent pas les mêmes classes de couleur ("Color" pour SimpleGraphics et "ColorPlus" pour Coolgraphics).
J'ai alors rajouté à Canvas un simple enum "CanvasColor", donc les couleurs de ma propre application, et ajouté dans les 2 classes "Adapter" une méthode qui transforme les CanvasColor en la couleur de la librairie visée (avec un simple return switch).

Pour la question du clic de souris, cela a été bien plus complexe à gérer. Le problème était que pour utiliser la méthode, nous devions passer un Drawing en paramètre aux classes SimpleGraphicAdapter et CoolGraphicAdapter. Ceci est une très mauvaise chose à faire, car si un utilisateur veut implémenter notre libraire Canvas, il sera forcé d'utiliser un Drawing même s'il ne le veut pas. J'avais en fait lié plusieurs classes entre elles, ce qui est une mauvaise chose à faire. J'ai donc décidé de plutôt créer une interface fonctionnelle (avec le *@FunctionnalInterface*), directement dans l'interface Canvas, avec seulement une méthode *onclick*, qui serait appelée dans les méthodes de gestion du clic de souris (*waitForMouseCLick()*) des 2 classes Adapter. Ainsi, j'avais seulement besoin de passer l'interface fonctionnelle en paramètre des méthodes, et plus un Drawing.

2.4. Partie *Utilisation*



Enfin, dans cette dernière partie reste les 3 classes qui n'ont pas d'histoire d'implémentation, d'héritage ou bien de *bounding*. Tout d'abord, précisons que seule la classe "Paint" est de visibilité package, étant donné qu'il s'agit de la porte d'entrée de notre application, et donc que l'utilisateur doit forcément y avoir accès afin de pouvoir utiliser notre projet UgePaint. Les 2 autres classes sont en visibilité package car encore une fois inutile pour un utilisateur dans notre contexte.

La classe Paint va ainsi s'occuper de créer un Drawing, créer la fenêtre, dessiner toutes les formes et également appeler la méthode vue précédemment pour la gestion du clic de souris, grâce à l'interface Canvas. Le *main* va aussi lire les options indiquées par l'utilisateur, pour savoir s'il faut utiliser la librairie SimpleGraphics ou bien CoolGraphics, dans la méthode *createWindow()*.

La classe "Drawing" va comme son nom l'indique, s'occuper de créer toutes les formes à partir du fichier et de dessiner toutes ces formes. Il est important pour respecter les principes SOLID que cette classe ne s'occupe que de dessiner les formes. C'est pour cela qu'un problème s'est levé lorsque la fenêtre devait maintenant s'adapter en taille en fonction des formes à dessiner. Ma première idée fut de créer des champs *width* et *height* pour garder en paramètre la taille de la fenêtre, et venir augmenter ces paramètres au fur et à mesure de la création des formes. Cependant, comme je viens de l'indiquer, cela n'aurait pas été une très bonne implémentation, car Drawing, qui initialement devait s'occuper de créer et dessiner des formes, aurait maintenant dû gérer les dimensions de la fenêtre. Si

nous avions voulu rajouter des fonctionnalités à cette fenêtre, la classe Drawing se serait encore plus éloignée de son objectif principal.

J'ai donc créé un record WindowSize, s'occupant de stocker les valeurs de taille de la fenêtre. La classe Drawing pouvait ainsi avoir une méthode ne s'occupant que de recalculer la taille de la fenêtre à créer afin d'afficher toutes les formes, et n'avait donc plus aucune valeur à stocker en champs pour tout ce qui concerne la fenêtre.

De plus comme la fenêtre se devait d'être un carré, j'ai dû créer une méthode *union()* qui prend une autre WindowSize en paramètre, et renvoie une nouvelle WindowSize correspondant à la taille max entre les deux WindowSize (celle en paramètre, et le *this*).

2.5. SOLID

Cette partie reprend ce qui a été expliqué plus tôt, et montre en quoi ces intégrations techniques sont bien conformes aux principes SOLID dans la majorité.

2.5.1. Single responsibility principle

Le principe d'une seule et même responsabilité par méthode et par classe est sûrement le principe SOLID le plus connu mais également l'un des plus importants. Il est ici facile de vérifier que ce principe est respecté dans le code. Comme nous venons de le voir, j'ai adapté ma classe Drawing afin qu'elle ne s'occupe pas de stocker des valeurs en rapport avec la fenêtre, mais seulement avec les formes. Cette classe continue ainsi à créer et dessiner des formes seulement. Pour le Paint il est un peu plus complexe de faire cette distinction, mais si l'on part du principe que la mission de cette classe est de gérer l'application, alors en effet le Paint s'occupe de créer les différents outils, et de les laisser travailler, comme cela a été expliqué dans l'introduction sur la partie de "single responsibility principle".

Pour le reste des classes, chacune a été créée pour une seule tâche et ne s'occupe que de remplir celle-ci, comme nous avons pu le voir dans les parties précédentes.

Pour porter un regard un plus critique sur mon code, seule la méthode *parseLines(Path path)* s'éloigne un peu du principe de responsabilité unique, cette méthode s'occupant à la fois de lire dans le fichier, créer les formes et les ajouter dans la liste. Je pourrais argumenter que la méthode s'occupe finalement de créer une liste de formes à partir d'un chemin (vers un fichier), je pense que la méthode aurait pu être divisée en 2 méthodes peut-être pour mieux répondre au principe de responsabilité unique.

2.5.2. Open/closed principle

J'ai également essayé de respecter le principe d'être ouvert aux extensions mais fermé aux modifications. Actuellement, sauf si l'on voulait changer totalement une le principe de l'application, ce qui n'est pas à faire en Java et nous devrions alors créer une autre application, la majorité des classes respecte le principe d'open/close. Si nous voulons par exemple rajouter une librairie graphique comme exprimé plus tôt, il suffirait ainsi de seulement créer un nouvel adaptateur (ou Adapter) qui implémente l'interface Canvas. Il faudrait dans certains cas modifier le code des classes directement mais il s'agirait de modifications minime ne modifiant pas l'utilisation du code pour l'utilisateur. Si par exemple nous souhaitons rajouter une couleur, il nous faudrait alors modifier l'enum CanvasColor ainsi que les méthodes *transformColor* dans les Adapter. Il ne s'agit pas d'une faute très grave car à aucun moment cela ne paralyserait le code, ou ne changerait la façon d'utiliser le code, cependant le principe de fermé aux modifications n'est pas à 100% respecté pour les couleurs, même s'il s'en rapproche.

Pour l'ajout de nouvelles formes, le principe n'est pas non plus complètement respecté. Il faudrait dans un premier temps modifier l'interface Shape, en ajoutant le nouvelle forme aux classes ayant le droit d'implémenter l'interface, mais il faudrait également changer la méthode *parseLines()*, ce qui devient déjà plus contraignant que de simplement rajouter une ligne dans une interface. Il s'agit là des deux seuls cas ne respectant pas complètement le principe d'open/close, tout le reste du code me semble assez modulable et décomposé pour ne pas avoir à modifier des classes déjà existantes pour l'ajout de simples fonctionnalités.

2.5.3. Liskov substitution principle

Le principe de Liskov n'a pas vraiment lieu d'être dans mon projet, étant donné que je n'utilise pas d'héritage. Ce principe était toutefois respecté lorsque j'ai créé et utilisé la classe abstraite AbstractRectangleEllipse, mais cette classe n'existe plus. Pour ce qui est des implémentations, aucune nouvelle implémentation d'une interface n'a causé de problèmes au code.

Le principe de Liskov est également lié de près au principe d'inversion de dépendance présenté plus tard.

2.5.4. Interface segregation principle

Pour le principe de ségrégation d'interface, celle-ci est en grande partie respectée. Il existe peut-être de meilleures implémentations que celles que j'ai développé, mais elles me semblent personnellement allié le mieux les principes SOLID, l'efficacité et la compréhension de code.

Pour l'interface Shape, il n'y a pas de méthode en *default*, et les seules méthodes à implémenter sont bien en rapport avec les formes, à savoir une méthode *draw()* pour dessiner, une méthode *distance()* pour obtenir une distance et une méthode *windowSize()*, pour obtenir la taille en fenêtre que prend cette forme. Cette dernière méthode est plus discutable, étant donné que l'on renvoie une WindowSize dans une interface de forme, mais cela ne me semble pas non plus trop choquant. C'est en tout cas la meilleure solution à laquelle je suis parvenue.

Pour l'interface Canvas, celle-ci est déjà bien plus fournie comme nous avons pu le voir plus tôt. Cependant tout est bien à sa place et lié à l'utilité de la classe, à savoir s'occuper de toute la partie graphique, avec un enum pour les couleurs, une interface fonctionnelle présentée plus tôt pour les clics de souris, les méthodes de dessin directement et une méthode *render()* en default, qui sera présentée plus en détail pour la partie sur l'exercice 9 mais qui s'occupe simplement d'effectuer tous les dessins.

2.5.5. Dependency inversion principle

Ce principe consiste donc à ce qu'une classe ne dépende pas de l'implémentation d'autres classes. L'idée que j'ai essayé de suivre tout au long de ce code, est l'utilisation d'interfaces pour pallier de mauvaises dépendances. En effet, l'utilisation de ces interface (Canvas et Shape) permet que la création de leurs sous-classes ne dépendent pas d'autres classes directement.

3. Exercices supplémentaires

3.1. Classe Drawing

Pour ce premier exercice, on fait un retour sur un problème que j'ai mentionné plus tôt, à savoir le problème de respect du principe "open/close" de la classe Drawing. En effet, il nous faut rajouter dans la méthode directement *fromFile()* un *case* dans le *switch* pour la création des nouvelles formes.

Pour reformuler le problème, on souhaiterait être capable d'ajouter de nouvelles formes sans avoir à modifier la méthode *fromFile()* à chaque fois. J'ai donc opté pour une idée assez simple, à savoir créer une méthode *createShapeFromName()* dans l'interface Shape directement donc, en default et de visibilité package. Elle ne s'occuperait pas de lire le fichier ce qui serait contraire au "single-responsibility" de SOLID, mais seulement de renvoyer une nouvelle Shape en fonction du nom pris en paramètre. J'ai ainsi déplacé le *switch/case* de la méthode *fromFile()* dans cette nouvelle méthode.

On se retrouve ainsi avec cette nouvelle méthode dans Shape :

```
static Shape createShapeFromName(String name, int val1, int val2, int val3, int val4) {
    return switch (name) {
        case "line" -> new Line(val1, val2, val3, val4);
        case "ellipse" -> new Ellipse(val1, val2, val3, val4);
        case "rectangle" -> new Rectangle(val1, val2, val3, val4);
        case default -> throw new UnsupportedOperationException();
    };
}
```

Et la méthode modifiée dans Drawing :

```java
public static Drawing parseLines(Path path) throws IOException {
    List<String> list = new ArrayList<>();
    try (Stream<String> lines = Files.lines(path)) {
        lines.forEach(list::add);
    }
    var drawing = new Drawing();

    list.forEach(line -> {
        String[] tokens = line.split( regex: " ");
        var type :String = tokens[0];
        var val1 :int = Integer.parseInt(tokens[1]);
        var val2 :int = Integer.parseInt(tokens[2]);
        var val3 :int = Integer.parseInt(tokens[3]);
        var val4 :int = Integer.parseInt(tokens[4]);
        drawing.addShape(Shape.createShapeFromName(type, val1, val2, val3, val4));
    });
    return drawing;
}
```

Désormais, lorsque l'on voudra créer une nouvelle Shape, il nous suffira de seulement modifier le code de l'interface Shape, en ajoutant aux permissions la nouvelle Shape et en rajoutant une ligne dans le *switch/case* de la méhtode *createShapeFromName()*.

Je suis conscient que cela ne règle pas tout le problème, car il faudra toujours modifier le code de Shape plus qu'on ne l'aurait souhaité, mais c'est pour le moment ce à quoi je suis arrivé.

3.2. Jar fournit des Canvas

Dans cet exercice, on souhaite revenir sur un précédent exercice sur les ServiceLoader, et expliquer pourquoi le jar doit fournir une classe pour CanvasFactory et pas pour l'interface Canvas.

La raison est simple, on souhaite pouvoir modifier la taille lors de la création du jar, or, le constructeur de Canvas ne peut pas prendre d'argument à cause du ServiceLoader, on souhaite donc utiliser une CanvasFactory pour permettre cela.

3.3. Nouvelle figure *Square*

Pour ce troisième exercice, l'idée est de créer une nouvelle classe "Square", qui implémenterait donc l'interface Shape. Square ne peut pas hériter de Rectangle car il aurait accès à des champs inutiles, ou autres méthodes. Il s'agirait d'une très mauvaise pratique.

J'ai ainsi réfléchi à une première solution, avant de me tourner vers une seconde solution qui me semble bien meilleure.

BoundingBox

Ma première idée était de reprendre la technique utilisée lors de la factorisation du code pour Rectangle et Ellipse. Nous avions alors créé une BoundingBox, présentée plus tôt, afin de stocker tout le code commun entre ces 2 classes.

Cependant cette idée ne me semblait pas très optimisée ici, car il aurait fallu modifier la classe Rectangle, ce qui, comme nous l'avons vu, ne rentre pas dans les principes SOLID, d'autant plus que Rectangle aurait désormais eu 2 BoundingBox en champs, ce qui encore une fois est signe d'une mauvaise implémentation.

Je me suis alors porté sur une seconde idée, bien plus simple et respectant un peu plus les principes SOLID :

Champ Rectangle

Cette idée est donc de stocker un champ Rectangle dans la nouvelle classe Square, qui implémenter Shape bien sûr, et redéfinir le constructeur de Square un *x*, un *y* et un *width* en paramètre. L'idée sera d'initialiser le champ Rectangle de Square en lui passant la valeur *x* pour *x*, *y* pour *y*, *width* pour *width* et *width* pour *height*. La classe Square aura désormais en champs un Rectangle dont la largeur et la hauteur sont égales, ce qui est la définition exacte d'un carré. Pour les méthodes à redéfinir dans Square, rien de plus simple, on rappellera seulement les méthodes de Rectangle à travers le champ Rectangle.

Pour faire cela, il faudra bien évidemment rajouter dans l'interface Shape la permission à Square d'implémenter l'interface, Shape étant pour rappel une interface "sealed" (fermé).

Pour le petit point noir de cette solution, je ne suis pas complètement sûr que le D des principes SOLID (Dependency inversion) soit respecté, la classe Square dépendant de Rectangle pour son utilisation. Cependant, Square ne prend pas de Rectangle directement en paramètre mais seulement des valeurs *int*, et créé ensuite dans la classe directement un Rectangle, ce qui rend peut-être la solution un peu plus viable.

4. Exercice 9

La problématique de l'exercice 9 était que jusqu'à maintenant, on faisait un appel à la méthode *render()* de SimpleGraphics à chaque nouveau dessin d'une forme. Il est cependant possible pour SimpleGraphics de faire tous les dessins en un seul *render()*, à condition que celui-ci connaisse toutes les formes à dessiner bien sûr.

L'idée ici a donc été de faire en sorte que le *render()* pour SimpleGraphics uniquement (car la méthode *render()* de CoolGraphics n'est pas visible) ne soit effectué qu'une seule fois.

Il m'a fallu du temps pour penser à une solution et pour l'implémenter, mais je vais ici directement présenter la solution finale à laquelle j'ai abouti.

J'ai tout d'abord commencé par créer un Consumer, auquel j'ai pu penser grâce au TP2 du cours de Design Pattern, sur les lignes de commandes et les options, où nous utilisions des Consumer. J'ai rapidement vu dans la JavaDoc de Consumer qu'il existait une seconde méthode *andThen()*, qui permet de rajouter une commande à exécuter dans le consumer et renvoie ce consumer. Il m'a fallu un peu de temps et d'aide pour comprendre exactement ce qu'elle faisait et comment l'utiliser, mais une fois cela fait j'ai pu modifier toutes mes méthodes d'affichage dans la classe SimpleGraphicAdapter en remplaçant les anciens "area.render()" par des "this.consumer = consumer.andThen()", comme on peut le voir ci-dessous avec la méthode *drawRectangle()* :

```
@Override
public void drawRectangle(int x, int y, int width, int height, CanvasColor color) {
    this.consumer = consumer.andThen(graphics2D → {
        graphics2D.setColor(transformColor(color));
        graphics2D.drawRect(x, y, width, height);
    });
}
```

J'ai par la même occasion créé un consumer que j'ai initialisé sans lui faire exécuter de commandes, comme ceci :

```
private Consumer<Graphics2D> consumer = Graphics2D → {};
```

Une fois ceci fait, il me fallait une façon d'exécuter le *render()*. J'ai pour cela créé une nouvelle méthode dans l'interface Canvas, que j'ai subtilement nommé "*render()*". J'ai ainsi implémenté ma nouvelle méthode comme ceci :

```
@Override
public void render() {
    graphic.render(consumer);
}
```

Problème lorsque j'ai essayé de lancer mon code, une erreur est apparue. En effet, la classe CoolGraphicAdapter avait également besoin d'implémenter la nouvelle méthode *render().* Ne sachant alors pas quoi faire, j'ai laissé son corps vide étant donné qu'il s'agit d'une méthode *void*, et j'ai pu tester mon code en appelant donc la méthode *render()* après la méthode *drawAll()*, qui avant dessinait toutes les formes, et qui maintenant ajoute seulement les méthodes pour dessiner des formes dans le consumer, seulement lorsque la librairie SimpleGraphics est utilisée bien évidemment. Ci-dessous un extrait de Paint :

```
var canvas : Canvas = createWindow(args, name, window);
drawing.drawAll(canvas);
canvas.render();
```

Une fois avoir vérifié que tout fonctionnait bien, je me suis repenché sur ce problème de méthode vide dans CoolGraphics car cela ne me convenait pas. Cela voulait donc dire que toute personne souhaitant utiliser sa propre librairie graphique devra implémenter une méthode peut-être inutile, et lui laisser un corps vide ?

Après en avoir discuté avec quelques personnes, je suis finalement tombé sur un article (https://toungafranck.com/tutoriels/les-nouveautes-java-8/les-methodes-par-defaut-dans-les-interfaces/) où M. Tounga Franck expliquait comment utiliser des interfaces et des méthodes en default, et notamment qu'il était tout à fait possible de créer une méthode en default dans une interface, avec un corps vide (*"Les méthodes par défaut peuvent être utilisées pour créer des méthodes facultatives et des héritages multiples de comportement"*).

Cette solution me permettait ainsi de ne rien avoir à changer au code de SImpleGraphicAdapter, tout en retirant le fait d'avoir forcément à implémenter cette méthode dans les autres classes. Voici donc la méthode default que j'ai ajouté à mon interface Canvas :

```
1 override
default void render() {
    // Do nothing
}
```

Et avec ceci mon exercice 9 semblait fonctionner correctement, avec une implémentation efficace et non contraignante dans le temps.