Rapport UGE Paint

Table des matières

Exercice 1 à 5....2
1 - Les classes / Types....2
Interface....2
Classes Abstraite....2
Les figures....2
Classe Drawing....3
Liste de Shape....3
Classe Point....3
2 - Si la méthode draw() n'était pas dans l'interface Shape....3
3 - UML des sous-types de Shape....4
4 - Factorisation du code entre Rectangle et Ellipse....4
5 - Responsabilités par chacun des types....4
6 - Classe Drawing respectant le open-close principle....5
Exercice 6 à 9....6
1 - Implémentation du design pattern Adapter....6
2 - ServiceLoader (canards nommées)....7
3 - Pourquoi un jar fournissant CoolGraphics ne pourrait pas fournir l'interface Canvas ?....7
4 - Quelle méthode rajouter à Canvas si on voulait fournir directement Canvas ?....7
5 - Ajout d'une figure Square, sous type de Shape....8
6 - Exercice 9 stocker via une lambda....9

Exercice 1 à 5

1 - Les classes / Types

Les champs représentant les coordonnées d'une figure sont tous *private final* car celle-ci ne seront jamais modifiés.

▪ Interface

- *public interface* Shape → Interface représentant une figure.
- *void* draw(Graphics2D graphics, Color color); → Méthode permettant de dessiner une figure.
- Point center(); → Méthode permettant d'obtenir le centre d'une figure.

▪ Classes Abstraite

- `abstract class` AbstractShape *implements* Shape → Classe abstraite permettant de factoriser le code de Rectangle et Ellipse qui ont des méthodes en communs.
  On a 4 champs (2 coordonnées) pour la position ainsi que largeur et hauteur des figures.
  Les getters pour obtenir les coordonnées dans les classes qui extend cette classe abstraite.
  La classe draw est définie comme abstraite car le corp de la méthode sera définie par les classes qui seront extend par cette classe abstraite étant donné que les méthodes pour dessiner ne sont pas les mêmes pour chaque figure. On peut noter aussi le fait que la classe à la visilibté par defaut (package), c'est à dire que l'accessibilité aux méthodes, champs de classe n'est possible que pour un code situé dans le même package.

▪ Les figures

- *public record* Line(*int* x1, *int* y1, *int* x2, *int* y2) *implements* Shape
- *public class* Rectangle *extends* AbstractShape
- *public class* Ellipse *extends* AbstractShape

Les 3 classes ci-dessus représente les figures géré par notre programme, c'est pourquoi Line implemente Shape de même pour Rectanglet et Ellipse via la classe abstraite. De plus les classes Rectangle et Ellipse extend une classe abstraite pour les raisons expliqué plus haut. Toutes les méthodes ici sont publique car elles ont besoin d'être appelé par d'autres classes.

- **Classe Drawing**

  - `public class` Drawing → permet de réaliser les traitement graphique tel que dessiner les figures, stocker les figures... Elle contient une méthode *public static* Figure parseFile(String fileName) qui permet de lire un fichier contenant les coordonnées des figures à dessiner ces figures sont tout d'abord stocker dans une liste Shape (l'interface décrite plus haut). Cette méthode est déclaré static car elle ne dépend pas d'une instance précise. Elle retourne la liste de Shape après traitement.

- **Liste de Shape**

  - *public class* ListShape → la liste de Shape est encapsulé dans cette classe. Cela permet notamment d'imposer de passer par des méthodes permettant un accès sécurisé. Le champs correspondant à la liste de Shape est final car cela nous permet de modifier son contenu sans pouvoir réassigné le champ à un autre objet. Cette classe contient les méthodes *public void* add(Shape shape) qui permet d'ajouter une figure tout en vérifiant que l'objet ne soit pas nulle. Ainsi que la méthode *public* List<Shape> getShapes() qui renvoie une copie de la liste de Shape respectant la programmation défensive.

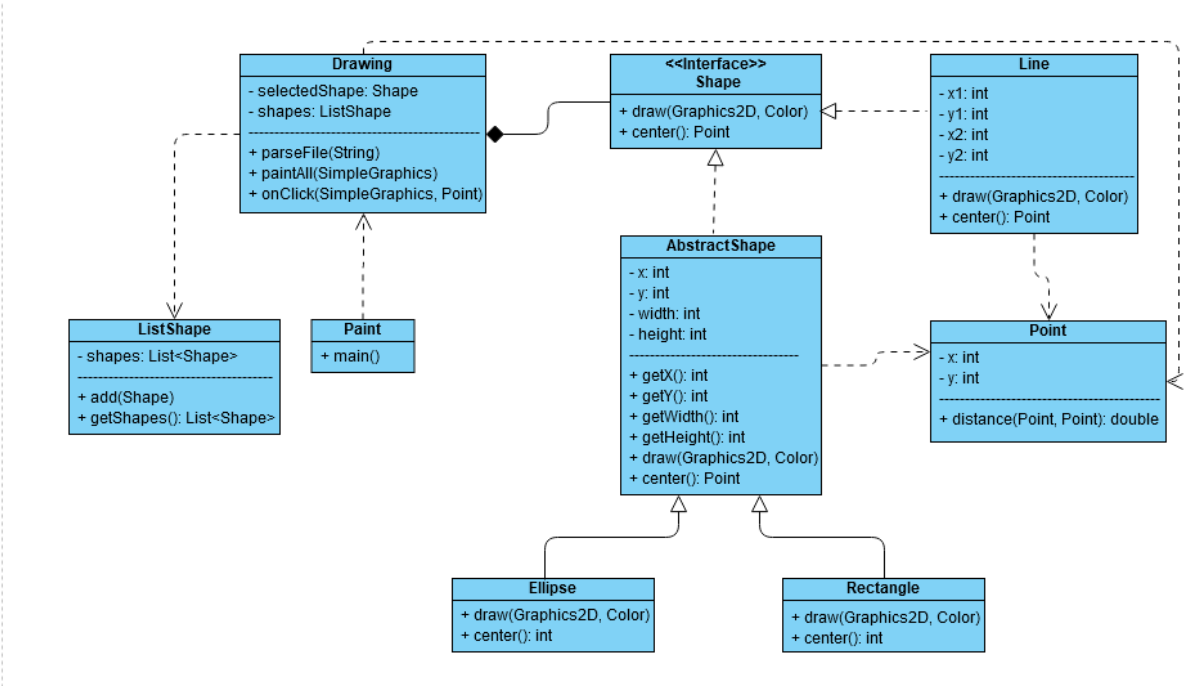
- **Classe Point**

  - *public record* Point(*int* x, *int* y) → permet de représenter une coordonnée (x, y) pour une figure. Elle possède notamment une méthode *public static double* distance(Point point1, Point point2) permettant d'obtenir la distance entre 2 points afin d'obtenir la figure qui à son centre le plus proche du clique de l'utilisateur.

2 - Si la méthode draw() n'était pas dans l'interface Shape

Si la méthode draw() n'était pas dans l'interface Shape, nous aurions été obligé de regarder le type de chaque Shape de notre liste de Shape. C'est à dire faire un switch sur chaque Shape et le comparer au type Line, Rectangle, Ellipse dans notre cas ou autres si nous décidions d'ajouter d'autres figures.

3 - UML des sous-types de Shape



4 - Factorisation du code entre Rectangle et Ellipse

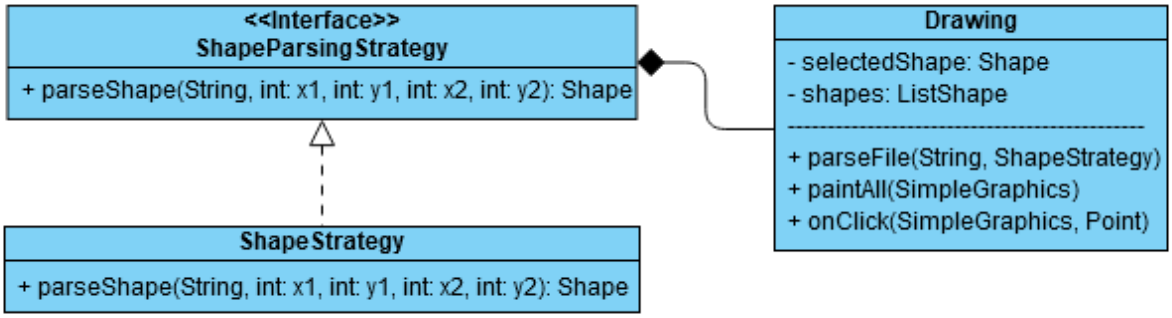
La factorisation du code entre Rectangle et Ellipse a été faite en créant une classe abstraite extend par Rectangle et Ellipse. Etant donné que ces deux classes ont les mêmes champs ainsi que les mêmes méthodes, le corp des méthodes y compris, il suffit de définir les champs et méthodes dans la classe abstraite. Dans Rectangle et Ellipse il suffira donc de faire appel qu'au méthode de la classe abstraite.

5 - Responsabilités par chacun des types

Les classes Line, Rectangle, Ellipse permettent de fournir une figure distinct. La classe ListShape permet de stocker la liste des figures l'encapsulation de la liste dans cette classe offre une sécurité en plus au niveau de la liste. La classe Drawing gère tous les traitement en accord avec les figures tel que le stockage de celle-ci, l'affichage graphique et les évènements. L'interface Shape fourni un ensemble de méthodes décrivant les caractéristiques d'une figure.

6 - Classe Drawing respectant le open-close principle

Pour respecter au minimum le open-close principle on peut utiliser le design pattern strategy en externalisant le parsing de la figure dans une autre classe. Lorsque l'on aura besoin de connaître quelle figure est-ce que l'on doit rajouter à notre liste de figures on fera appel à la méthode qui fera le parsing de la figure.




```java
@FunctionalInterface
public interface ShapeParsingStrategy {
    Shape parseShape(String shapeName, int x1, int y1, int x2, int y2);
}
```

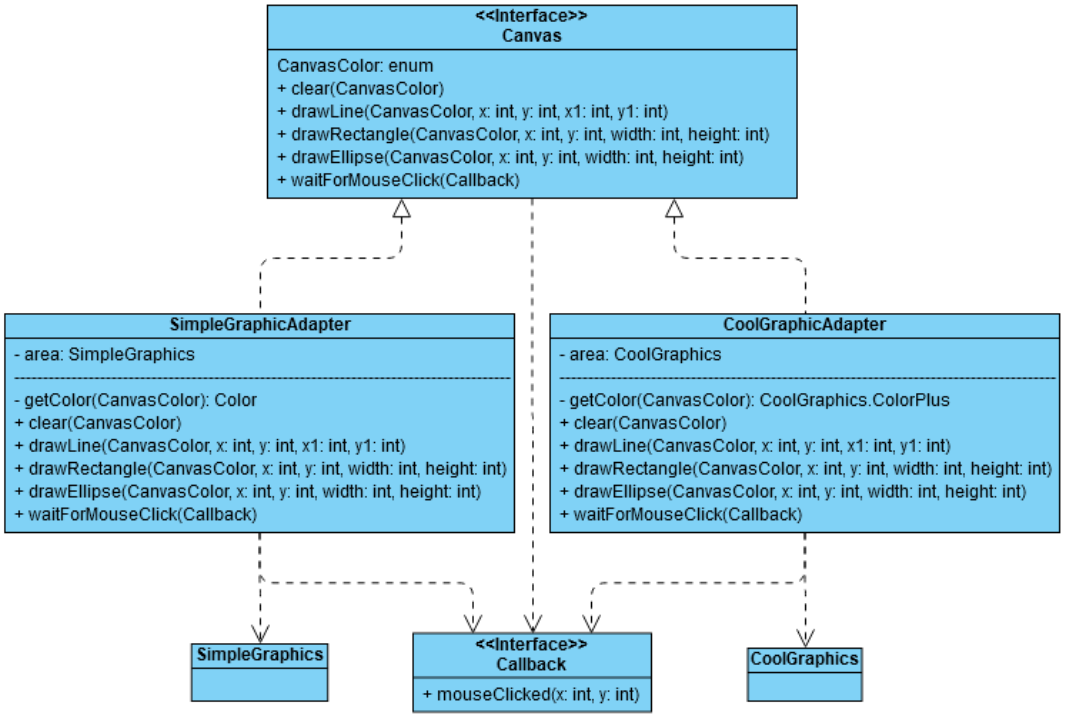
```java
public class ShapeStrategy implements ShapeParsingStrategy {
    @Override
    public Shape parseShape(String shapeName, int x1, int y1, int x2, int y2) {
        switch (shapeName) {
            case "line" -> { return new Line(x1, y1, x2, y2); }
            case "rectangle" -> { return new Rectangle(x1, y1, x2, y2); }
            case "ellipse" -> { return new Ellipse(x1, y1, x2, y2); }
            default -> throw new IllegalArgumentException("Unknown shape");
        }
    }
}
```

```java
public static void parseFile(String fileName, ShapeStrategy shapeStrategy) throws IOException {
    Path path = Paths.get(fileName);

    try(Stream<String> lines = Files.lines(path)) {
        for (String line: lines.toList()) {
            String[] tokens = line.split( regex: " ");
            int x1 = Integer.parseInt(tokens[1]);
            int y1 = Integer.parseInt(tokens[2]);
            int x2 = Integer.parseInt(tokens[3]);
            int y2 = Integer.parseInt(tokens[4]);
            shapes.add(shapeStrategy.parseShape(tokens[0], x1, y1, x2, y2));
        }
    }
}
```

Exercice 6 à 9

1 - Implémentation du design pattern Adapter



Pour gérer les Adapter, les classes SimpleGraphicAdapter et CoolGraphicAdapter ont été créer chacune utilisant leur librairie respective comme on peut le voir sur le schéma ci-dessus. Pour unifier ces deux classes, elles implémente une interface Canvas qui les représente décrivant les méthodes que SimpleGraphicAdapter et CoolGraphicAdater doivent implémenter. Les méthodes permettant de dessiner une figure prennent un enum déclarer dans l'interface Canvas représentant la couleur de la figure ainsi que les coordonées de la figure. Une interface fonctionnelle a été créer afin de gérer le cas des évènements.

2 - ServiceLoader (canards nommées)

Afin de pouvoir fournir des canards nommés sans passer par une factory, on peut ajouter la méthode `public Duck setName(String name, Duck duck)` prenant en paramètre le nom du duck ainsi que les duck chargé par le serviceLoader afin de renvoyer une nouvelle instance de celle-ci mais cette fois ci avec le nom du duck. Il suffira de modifier le code de la classe DuckFarmBetter de la manière suivante :

```java
public class DuckFarmBetter {
    private static Duck setName(String name, Duck duck) throws InvocationTargetException, InstantiationException, IllegalAccessException {
        return (Duck) duck.getClass().getDeclaredConstructors()[1].newInstance(name);
    }

    public static void main(String[] args) throws InvocationTargetException, InstantiationException, IllegalAccessException {
        var serviceLoader :ServiceLoader<Duck> = ServiceLoader.load(Duck.class);

        for (var duck: serviceLoader) {
            System.out.println(setName( name: "Riri", duck).quack());
            System.out.println(setName( name: "Fifi", duck).quack());
            System.out.println(setName( name: "Loulou", duck).quack());
        }
    }
}
```

3 - Pourquoi un jar fournissant CoolGraphics ne pourrait pas fournir l'interface Canvas ?

Un jar fournissant CoolGraphics ne peut pas fournir l'interface Canvas car toutes nos Factory dépendent de l'interface Canvas.

4 - Quelle méthode rajouter à Canvas si on voulait fournir directement Canvas ?

5 - Ajout d’une figure Square, sous type de Shape

Faire hériter Square de Rectangle n’est pas la bonne pratique ici, la sous-classe hérité doit avoir un lien très fort avec la classe dont on hérite ce qui n’est pas le cas ici. On peut implémenter la classe Square de à base de composition sans héritage de la manière suivante :

```java
public class Square implements Shape {
    private final Rectangle square;

    public Square(int x, int y, int width, int height) {
        this.square = new Rectangle(x, y, width, height);
    }

    @Override
    public void draw(Canvas canvas, Canvas.CanvasColor color) {
        square.draw(canvas, color);
    }

    @Override
    public double distance(int x, int y) {
        return square.distance(x, y);
    }

    @Override
    public WindowSize minWindowSize() {
        return square.minWindowSize();
    }
}
```

En effet ici pour implémenter la classe Square il nous suffit de définir un carré comme un Rectangle puis d’implémenter les méthodes de Rectangle qui sont les mêmes que pour un carré.

6 - Exercice 9 stocker via une lambda

```
private Consumer<Graphics2D> graphics2DConsumer = graphics2D -> {};
```

```
@Override
public void drawLine(CanvasColor color, int x, int y, int x1, int y1) {
    graphics2DConsumer = graphics2DConsumer.andThen(Graphics2D -> {
        Graphics2D.setColor(getColor(color));
        Graphics2D.drawLine(x, y, x1, y1);
    });
}

@Override
public void drawRectangle(CanvasColor color, int x, int y, int width, int height) {
    graphics2DConsumer = graphics2DConsumer.andThen(Graphics2D -> {
        Graphics2D.setColor(getColor(color));
        Graphics2D.drawRect(x, y, width, height);
    });
}

@Override
public void drawEllipse(CanvasColor color, int x, int y, int width, int height) {
    graphics2DConsumer = graphics2DConsumer.andThen(Graphics2D -> {
        Graphics2D.setColor(getColor(color));
        Graphics2D.drawOval(x, y, width, height);
    });
}

@Override
public void render() {
    area.render(graphic -> {
        graphics2DConsumer.accept(graphic);
        graphics2DConsumer = graphics2D -> {};
    });
}
```