Programmation orienté objet et Design Pattern

Rapport application Paint

Jonathan CRÉTÉ INFO3

2 novembre 2020

Sommaire

Introduction

Ce document a été rédigé dans le but de récapituler les principes de conception utilisé au sein de l'application Java Paint. Il permet de relire les notions étudiées en cours de design pattern en 3ème année à l'ESIPE. Il peut servir de document de référence en cas de travaux et de mise en pratique de pattern.

Première partie : Exercice 1 à 5

Visibilité et accessibilité des champs et méthodes

Quelle(s) type(s)/classe(s) avez vous créé(e.s)? Avec quels champs/méthodes/modificateurs d'accessibilité? Qu'est ce qui vous guide ou vous permet de justifier chacun de ces membres et leur visibilité?

➙ **Une classe concrète et publique Line** : elle nous permet d'instancier un objet qui représente une ligne géométrique, dont les coordonnées sont extraites via le parsing d'un fichier texte. Les coordonnées d'une ligne sont des champs privés et finaux (on encapsule et les coordonnées géométriques d'une ligne ne sont pas supposées changer). Elle implémente l'interface Shape et le contrat de la classe expose deux méthodes publiques d'instances : Une méthode redéfinie *draw()* qui nous permet de dessiner la ligne en usant du paramètre graphics2D, et une méthode d'affichage.

➙ **Deux classes concrètes Rectangle et Ellipse** : le contrat de la librairie SimpleGraphics nous donnant la possibilité par le biais de méthodes publiques de créer d'autres figures géométriques comme des Rectangle et Ellipse, il est naturel de créer deux classes publiques pour ceux-ci. Ces deux classes exposent de la même façon, une méthode redéfinie *draw()* qui nous permet de dessiner la figure géométrique.

➙ **Une interface publique nommée Shape**, qui nous permet de donner un type commun que vont implémenter les classes concrètes Rectangle et Ellipse et qui possède la signature de la méthode publique draw() et computeDistanceBetweenCenterAndUserClick()

➙ **Une classe concrète publique ContainerShape** qui encapsule les figures. Toujours en respectant le principe d'encapsulation, la classe à un champ privé qui représente une liste de Shape. Son contrat donne les méthodes publiques d'instances *add()* qui permet d'ajouter une Shape à la List, *loopAndDraw()* qui boucle sur la liste et dessine les figures, et enfin *findMinShapeFromDistance()* qui trouve la figure avec la plus petite distance par rapport aux coordonnées passés en paramètres.

➙ **Une classe abstraite AbstractShape** : Celle-ci est de visibilité package et nous permet de partager les comportements similaires entre les classes concrètes Rectangle et Ellipse (champs associés représentant les coordonnées géométriques, ou encore la méthode permettant le calcul de la distance calculer la distance entre le point ou à cliqué l'utilisateur et le centre de la figure). A noter que mettre la visibilité de classe abstraite en publique serait une erreur car on exposerait à l'extérieur tous les détails d'implémentations. On ne met pas les champs privés dans la mesure

où cela nous imposerait d'avoir des getters/setters. On ne les met surtout pas protected car cela permettrait à n'importe quel utilisateur d'un autre package d'hériter de notre classe.

Hiérarchie des sous-types de Shape

Donnez la hiérarchie des sous-types de Shape sous la forme d'un diagramme de classe UML faisant apparaître leurs relations entre-elles et avec les autres types de l'application Paint.

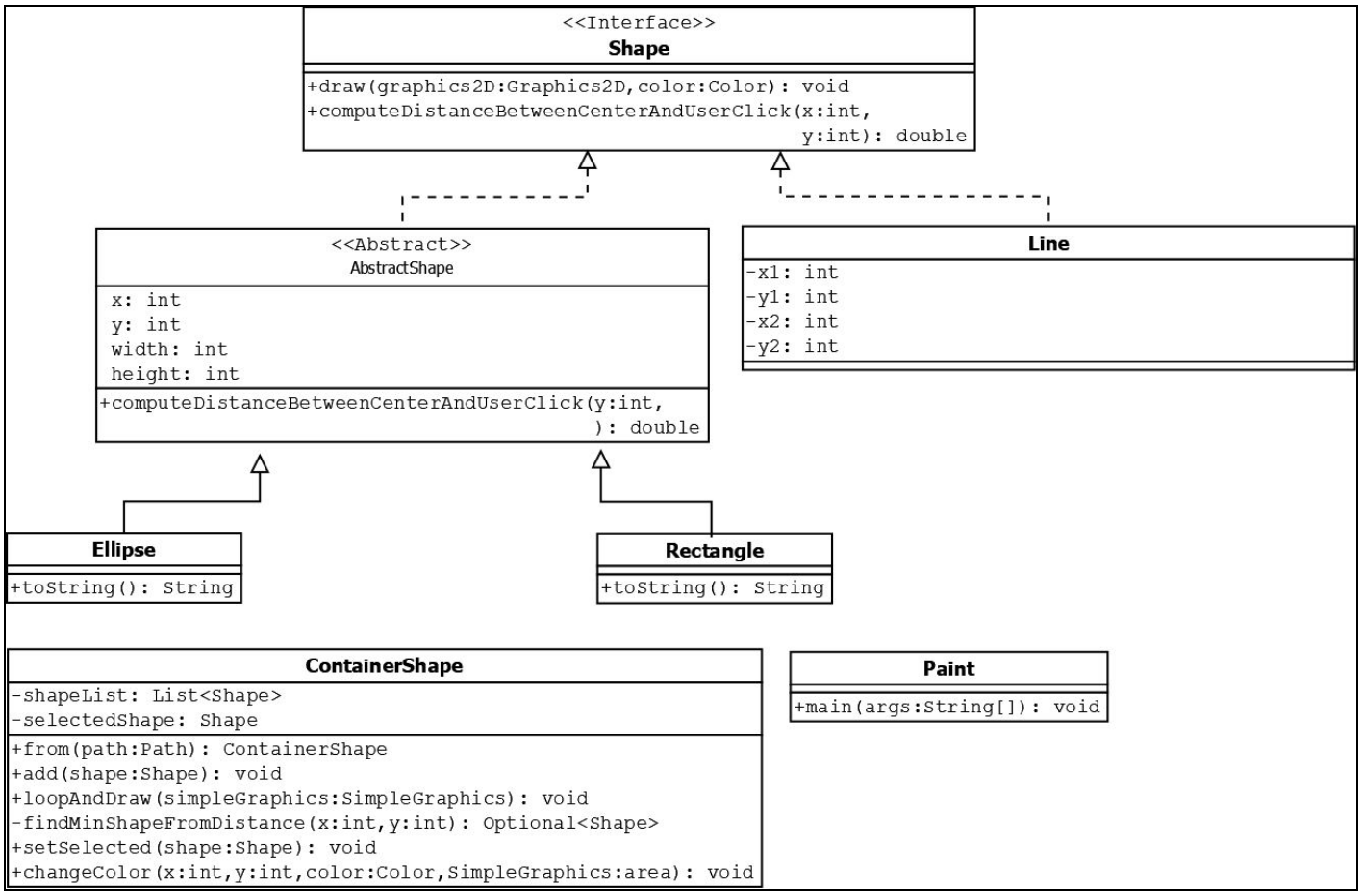


Figure 1 : Diagramme de classes UML des sous-types de Shape

La hiérarchie des classes des sous types de Shape se décompose de la façon suivante :

➔ La classe abstraite AbstractShape et la classe concrète Line **implémentent** l'interface Shape.

➔ Les classes concrètes Ellipse et Rectangle **héritent** de la classe abstraite AbstractShape.

➔ La classe concrète ContainerShape **utilise le type** Shape pour stocker les figures géométrique dans l'ArrayList.

Présence de la méthode draw() dans l'interface Shape

Si la méthode draw() n'était pas dans l'interface Shape, quel code serait-on obligé d'écrire dans la méthode en charge de dessiner toutes les figures?

Si l'on suppose que la méthode draw() n'est pas dans l'interface Shape, alors on devrait écrire dans la classe ContainerShape toutes les méthodes qui nous permettent d'afficher les figures géométriques.

Si l'on admet cette idée, cela pose plusieurs problématiques en terme d'implémentation :

➔ Tout d'abord, si on choisi de rajouter une implémentation (classe concrète) de Shape, par exemple une classe Losange, il faudrait retoucher au code de la classe ContainerShape, ce qui n'est pas du tout une bonne pratique et ne respecte pas le principe Open-closed principle.

➔ En outre, il serait nécessaire d'exposer à l'extérieur les champs des classes concrètes implémentant le type Shape, ce qui n'est pas bon non plus.

Factorisation du code de la classe Rectangle et Ellipse

Justifiez la manière dont vous avez factoriser le code entre Rectangle et Ellipse.

Tout d'abord, nous avons remarqué assez rapidement que le code des deux méthodes **draw()** et **computeDistanceBetweenCenterAndUserClick()** est dupliqué entre les classes Rectangle et Ellipse. Comme on sait que la duplication du code peut engendrer la duplication de bugs, on choisi de factoriser le code et champs communs. De fait, j'ai alors créer une interface publique nommée Shape, qui nous permet de donner un type commun que vont implémenter les classes concrètes Rectangle et Ellipse. Aussi, il faut partager les comportements de calcul de distance entre les deux classes, j'ai donc créé une classe abstraite **AbstractShape** de visibilité package (la mettre publique serait une erreur car on sinon on publierai à l'extérieur les détails d'implémentation) et déplacé dans la celle-ci les méthodes citées précédemment. Cette classe abstraite implémente l'interface Shape et ses champs ont la même visibilité que celle-ci. Les classes concrète Rectangle et Shape hérite de la classe abstraite.

Responsabilités des types

Essayez de décrire brièvement les responsabilités (les fonctionnalités offertes) par chacun des types.

Classe	Responsabilités/Fonctionnalités
AbstractShape	Factoriser/Partage le code et comportement commun entre Rectangle et Ellipse.
ContainerShape	Parser le fichier et instancier les objets Line, Rectangle et Ellipse pour les ajouter dans la liste. Encapsule la liste des figures.
Ellipse	Instancier, dessiner un objet Ellipse et afficher ses coordonnées
Line	Instancier, dessiner un objet Line et afficher ses coordonnées
Paint	Méthode main, point d'entrée du projet.
Rectangle	Instancier, dessiner un objet Rectangle et afficher ses coordonnées
Shape	Fournir un type commun Shape.

Deuxième partie : Exercice 6 à 9

Détails d'implémentation

Expliquez votre implémentation du design pattern Adapter dans l'exercice 7, en fournissant un diagramme UML et en paragraphe d'explication.

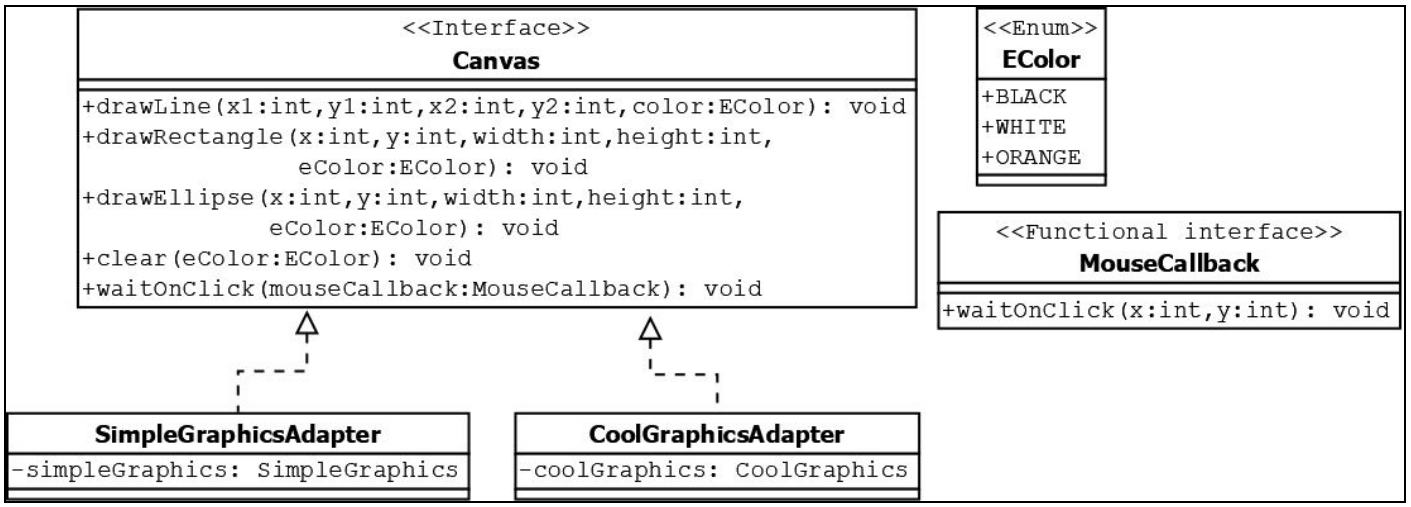


Figure 2 : Diagramme de classes UML d'implémentation du design pattern Adapter

Ici nous avons utilisé le pattern car la **nouvelle librairie CoolGraphics** offre d'autres fonctionnalités, mais par exemple ne propose pas de méthode *drawRectangle()*, tandis que SimpleGraphics oui. Nous avons donc créé une interface Canvas pour donner un type commun, ainsi que des méthodes de même nom et signatures, que nous allons redéfinir dans nos adapters. Utiliser le pattern nous permet ainsi de rendre notre code compatible avec notre application actuelle. Par conséquent, la finalité est que notre code devient alors "adaptatif" et qu'il peut fonctionner avec d'autres implémentations sans avoir modifié le code source des librairies.

A noter que **l'interface fonctionnelle MouseCallback** nous permet de définir une seule méthode abstraite qui est *waitOnclick(int x, int y)* qui nous offre la possibilité de réaliser le callback d'un clic utilisateur. La méthode est alors utilisée via une méthode référence :

```java
@Override
public void waitOnClick(MouseCallback mouseCallback) {

this.simpleGraphics.waitForMouseEvents(mouseCallback::waitOnClick);
}
```

NB : WaitOnclick(..) est redéfinie, il s'agit d'une implémentation de l'interface Canvas.

ServiceLoader

Dans l'exercice sur les ServiceLoader, proposez une solution pour fournir des canards nommés sans passer par une factory et en rajoutant une méthode setName(). Donnez le code de la classe DuckFarmBetter dans ce cas.

1) En premier lieu, on défini la signature de la méthode dans l'interface Duck :

```java
void setName(String name);
```

2) Ensuite on rédéfinie dans les classes **RegularDuck et RubberDuck** la méthode setName() :

```java
@Override
public void setName(String name) {
   Objects.requireNonNull(name);
   this.name = name;
}
```

3) Puis, voici le code de classe **DuckFarmBetter** :

```java
/******* Without factory **********/
ServiceLoader<Duck> loaderDuck =
ServiceLoader.load(fr.uge.poo.ducks.Duck.class);
String[] namesOfDucks = {"Riri", "Fifi", "Loulou"};
var i = 0;
for (var duck : loaderDuck) {
   duck.setName(namesOfDucks[i]);
```

```
    i++;
    System.out.println(duck.quack());
}
```

➜ On charge l'interface Duck avec ServiceLoader.

➜ On stocke dans un tableau de String les noms de canards.

➜ Pour chaque Duck de loaderDuck on lui affecte le nom correspondant, en appelant notre setter.

Le jar CoolGraphics

Expliquez pourquoi un jar fournissant CoolGraphics ne pourrait pas fournir l'interface Canvas ?

Un jar fournissant CoolGraphics ne pourrait pas fournir l'interface Canvas dans la mesure ou, on utilise l'interface Canvas pour tous les jars, à savoir CoolGraphics, SimpleGraphics et possiblement un autre, si une évolution s'impose.

Si l'on voulait fournir directement l'interface Canvas quelle méthode faudrait-il rajouter à l'interface Canvas ? Donnez le morceau de code dans Paint qui réalise la création du Canvas dans ce cas. Est-ce que cela fait sens dans notre contexte ?

Dans le cas où l'on voudrait fournir directement l'interface, une devrions rajouter à l'interface une méthode static de construction pour chaque librairie graphique :

```
static CoolGraphics coolGraphicsFactory(String nameArea, int
windowsWidth, int windowsHeight) {
  return new CoolGraphics(nameArea, windowsWidth, windowsHeight);
}
```

Puis l'appeler de la façon suivante:

```
if(isSimpleGraphics) {
    area = Canvas.simpleGraphicsFactory("Area with simpleGraphics",
800, 800);
} else {
    area = Canvas.coolGraphicsFactory("Area with
```

```
CoolGraphics",800,800);
}
```

Composition plutôt que héritage

*On veut ajouter une figure **Square**, sous-type de Shape. Peut-on la faire hériter de **Rectangle**? Proposer une solution à base de composition sans héritage.*

Effectivement un carré est une forme de “rectangle” donc on pourrait la faire hériter de Rectangle mais ceci est une mauvaise idée, car l’intérêt est limité, on a aucun membre (champs ou méthodes) de la super-classe Rectangle à récupérer.

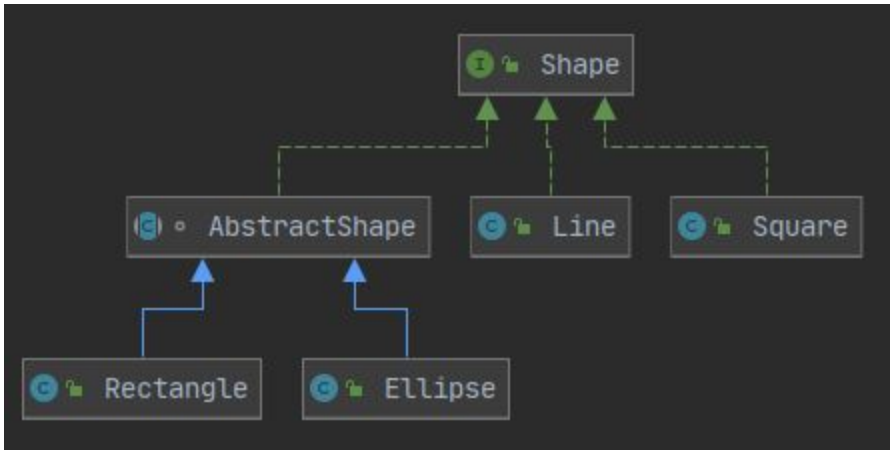
Voici l’implémentation de la classe Square:

```
public class Square implements Shape {

   private final Rectangle square;

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

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

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

Figure 3 : Extrait de la classe Square

Celle-ci possède un champ privé (toujours encapsulation..) final de type Rectangle. Le lien de composition est réalisé lors de l’initialisation dans le constructeur. En paramètre de ce constructeur, on prend les coordonnées géométriques, plutôt que de prendre directement un

objet Rectangle, dans la mesure où ceci est plus pratique et intuitif à l'instanciation de l'objet.



En outre comme Square représente un figure géométrique (comme *Line, Rectangle Ellipse*), il est normal que la classe implémente l'interface Shape.

Redéfinir toString()

Redéfinissez les méthodes toString() des différentes figures de sorte qu'elles produisent des affichages du genre:

- *line x1 y1 x2 y2*
- *ellipse 10 10 400 300*
- *rectangle 10 10 400 500*
- *square 10 10 200 200*

Pour les classes concrètes qui n'héritent pas de AbstractShape (Line et Square), on redéfinie *toString()* simplement de la façon suivante :

```
@Override
public String toString() {
   return "line" +
          " " + x1 +
          " " + x2 +
          " " + y1 +
          " " + y2;
}
```

Pour ce qui est de Rectangle et Ellipse, on écrit deux méthodes simples dans la classe abstraite :

```
public String getShapeType() {
   return getClass().getSimpleName().toLowerCase() + " ";
}

public String getCoords() {
   return x + " " + y + " " + width + " " + height;
}
```

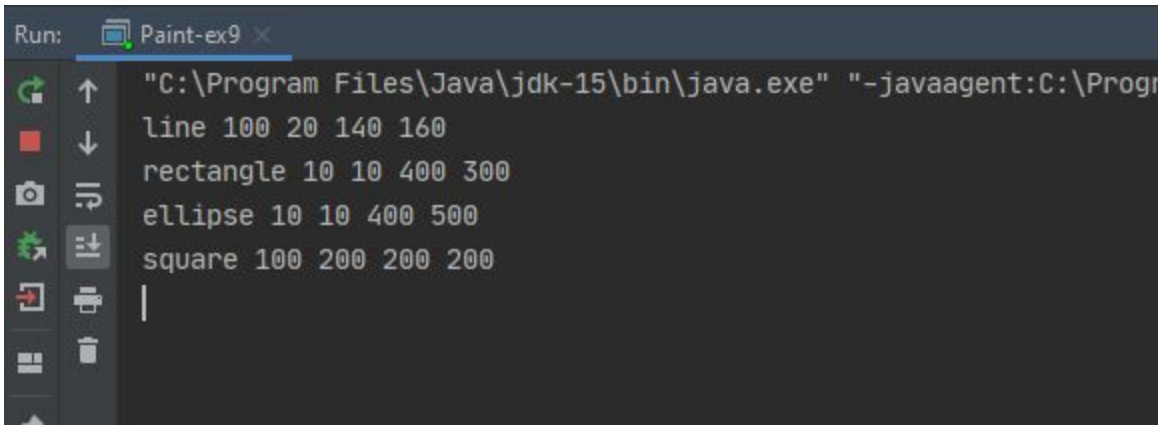
getShapeType() nous permet de récupérer le type (de la classe) et getCoords() les coordonnées de la figure.

Puis, dans chaque classe concrète, on appelle les deux méthodes dans la méthode toString redéfinie :

```
@Override
public String toString() {
   return getShapeType() + getCoords();
}
```

Enfin dans la classe Paint, dans la méthode main() on crée 4 figures et on teste :

```
var line = new Line(100, 20, 140, 160);
var rectangle = new Rectangle(10, 10, 400, 300);
var ellipse = new Ellipse(10 ,10, 400, 500);
var square  =  new Square(100, 200, 200);


System.out.println(line.toString());
System.out.println(rectangle.toString());
System.out.println(ellipse.toString());
System.out.println(square.toString());
```



Stocker une seule lambda

Dans l'exercice 9, il n'est pas nécessaire de stocker une liste de lambdas. On peut stocker une seule lambda (regardez Consumer.andThen pour vous inspirez).

On procède étape par étape :

1) Premièrement, on créer un nouveau champ de type `Consumer<Graphics2D>`

```
private Consumer<Graphics2D> graphics2DConsumer;
```

2) Deuxièmement, on initialise le consumer avec une lambda vide dans le constructeur :

```
public SimpleGraphicsAdapterWithoutListOfConsumer(int widthWindow,
int heightWindow) {
   this.simpleGraphics = new SimpleGraphics("Area with
SimpleGraphicsAdapter", widthWindow, heightWindow);
   this.graphics2DConsumer = graphics2D -> {};
}
```

3) Ensuite, dans chaque méthode drawXXX() on stocke la lambda avec Consumer.*andThen()* dans le consumer **graphics2DConsumer** :

```
@Override
public void drawLine(int x1, int y1, int x2, int y2, EColor color)
{
   this.graphics2DConsumer = graphics2DConsumer.andThen(graphics2D
-> {
      graphics2D.setColor(selectedColor(color));
      graphics2D.drawLine(x1, y1, x2, y2);
   });
}
```

4) Enfin dans la méthode *paint()*, dans le code de la méthode *render()*, on exécute le code du Consumer avec la méthode *accept(), puis on* restocker une lambda vide (pour clear) :

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

Annexe : Problème de concurrence

Lors de la réalisation de l'exercice 9, si l'on utilise une liste de lambda (ArrayDeque<Consumer<Graphics2D>>, le buffer ici) , on se rend rapidement compte qu'un problème de concurrence se présente : Dans la méthode *paint(),* le code de *simplegraphics.render(...)* est exécuté dans une autre thread ! Le buffer est utilisé dans le thread qui fait les dessins et dans le thread main ! (C'est la librairie qui lance un thread) On a donc une datarace, ce qui fait que l'on peut dessiner objet après l'appel à *paint().*

Solution :

Ce que nous voulons c'est dessiner le contenu du buffer exactement lors de l'appel à la méthode *paint()* . Du coup on peut soit utiliser un objet thread safe, soit faire une copie/clone du buffer (c'est ce qu'on choisi ici) :

```
@Override
public void paint() {
   this.simpleGraphics.render(graphics2D -> {
      var copyBuffer = this.buffer.clone();
      while (!copyBuffer.isEmpty()) {
         copyBuffer.pop().accept(graphics2D);
      }
   });
}
```

Liste des notions étudiées

- Design pattern Adapter
- Design pattern Factory
- Factorisation de code
- Interface fonctionnelle
- ServiceLoader
- Diagramme de classe UML
- Consumer et Runnable