Rapport mi-projet dessin vectoriel

Introduction :

1. Première partie : ce que l'on a fait

Présentation de l'architecture utilisée :

Implémenation d'un script:

Contexte :
Nous devons générer un script qui contient donc des instructions décrivant les opérations de l'utilisateur.
Problème : Le sens de script est large car un script peut être une instruction terminale mais aussi un composite ayant une relation d'agrégation avec la classe `Script`.
Solution : Nous utilisons un patron composite puisqu'il a l'avantage de représenter de manière claire les liens entre les différents composants qui représentent un script. Il permet de définir des méthodes communes à chaque instance sans se soucier de quelle classe d'implémentation vient la méthode concernée qui est appelée.
Avantage : Description du patron appliquée à notre problème :

```
script : InstructionTerminale
       | Sequence
       | For
       | Alternative
       ;

Sequence : scripts* ;
For      : 'for 0 to n' Sequence ;
Alternative : condition '?' script ':' script ;
InstructionTerminale : 'dessiner' | 'remplir' | 'insérer' | 'étiqueter' ;
```

Implémenation de Builder :

Contexte :
Nous devons générer un script qui contient donc des instructions décrivant les opérations de l'utilisateur. Ce script est écrit dans un langage que nous avons défini.
Problème : Comment générer les instances de `Script` sans faire appel au mot-clef `new` et que la création du script reste lisible et ressemble à un script textuel.
Solution : Nous nous sommes inspirés de la méthode décrite dans l'article Embedded Typesafe Domain

Specific Languages for Java et générons un script en utilisant des Builders. Le principe est le suivant : Pour créer un script (une instruction terminale ou un composite de Script) nous faisons appel à un builder, représenté par une classe qui correspond à l'instance de `Script` que nous voulons créer (par exemple `DessinerBuilder` pour créer l'instruction `Dessiner`). Le constructeur de `DessinerBuilder` appelé crée une instruction vide que l'on complète progressivement en faisant appel à la (aux) méthode(s) de `DessinerBuilder` qui retournent une instance d'un autre Builder sur lequel nous pourrons ensuite appeler d'autres méthodes pour initialiser d'autres attributs de notre instance de `DessinerBuilder`. Par exemple, la classe `DessinerBuilder` a une méthode

```
public PointsBuilder points(List<Point> points);
```

qui retourne un `PointsBuilder`, classe qui a une méthode public

```
PointsBuilder point(int x, int y);
```

nous permettant de créer un point.

2. Deuxième partie : ce que l'on va faire

- La partie traduction avec deux implémentations différentes
  - traduction vers le langage SVG
  - nous hésitons encore pour la deuxième
- Gestion de variables (l'implémentation de la boucle `For` est incomplète pour l'instant)
- Fonctionnalités diverses
- Remplir
- Tracer un chemin
- D'autres formes que l'on souhaiterait dessiner