Baptiste Maillot
Micaël M'Bagira
Thomas Moisy

Architecture Logicielle :

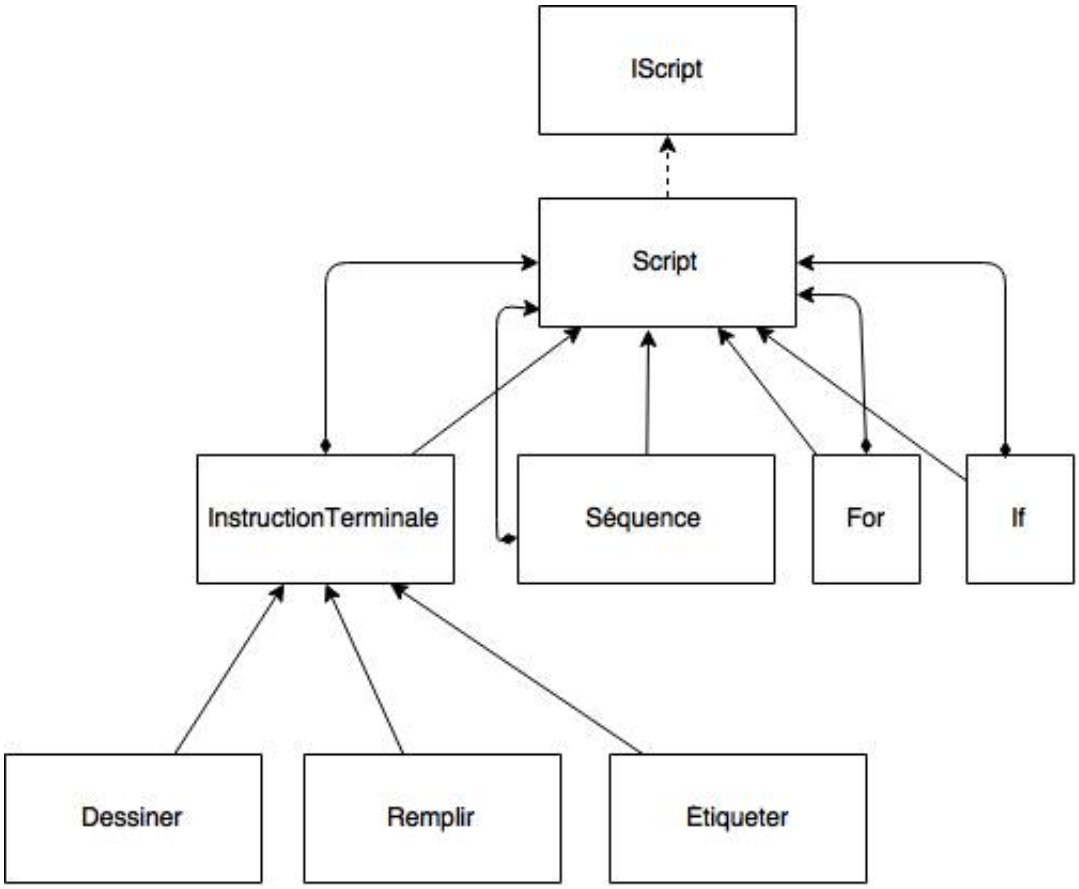
Rapport mi-projet dessin vectoriel

Présentation de l’architecture mise en place

1. Représentation d'un script :

Contexte :
Nous devons générer un script qui contient 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 le patron de conception composite puisqu’il a l’avantage de représenter de manière claire les liens entre les différents composants qui constituent un script. Il permet de définir des méthodes communes à chaque instance sans se soucier de quelle classe d’implémentation provient la méthode qui est appelée.

Description du patron appliqué à notre problème :



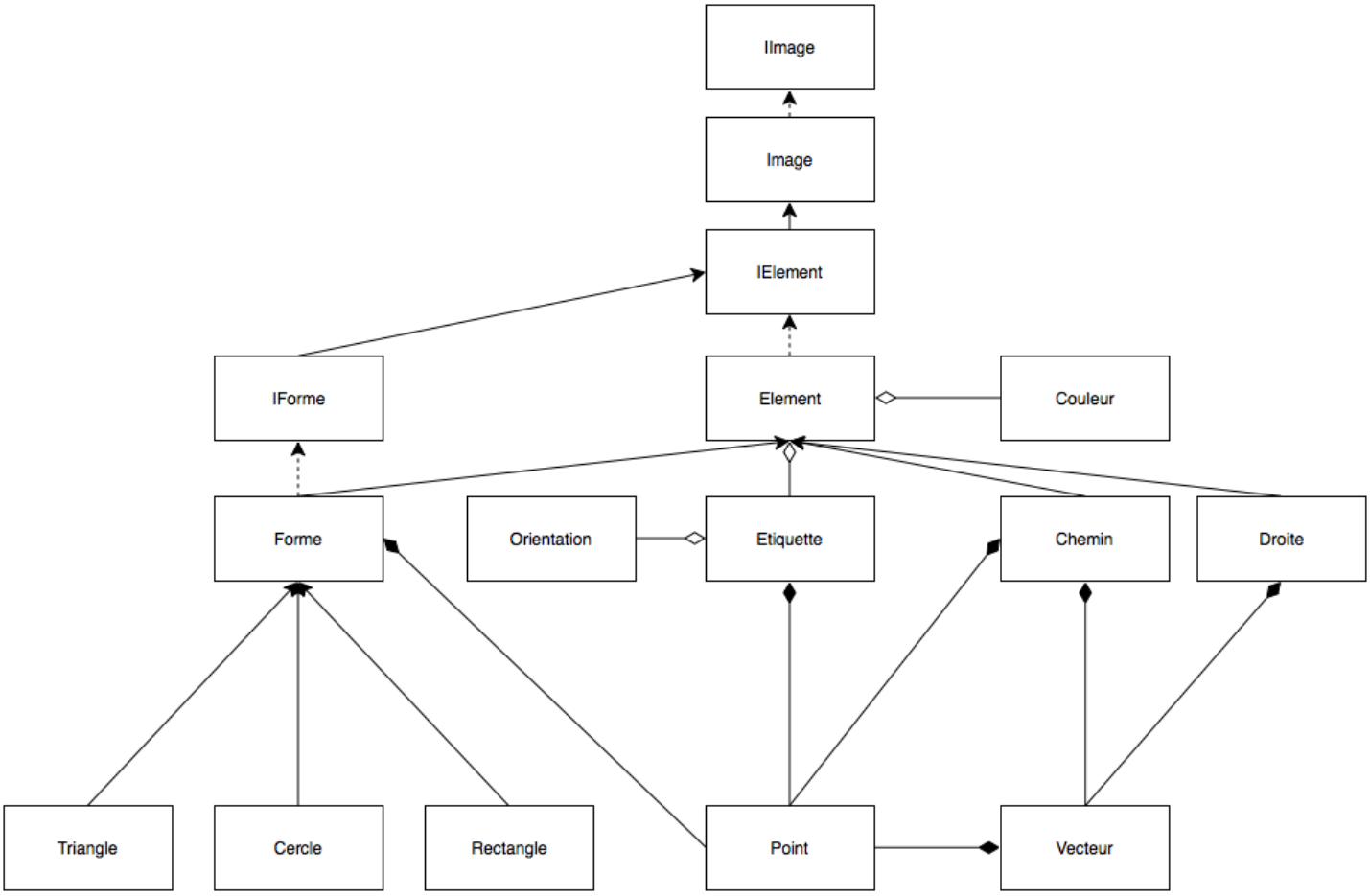

```
script : InstructionTerminale
       | Sequence
       | For
       | Alternative
       ;

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

2. Structure du package dessin vectoriel :

Contexte :
Ce package est destiné à représenter de façon logique le dessin (l'image dans sa globalité) ainsi les éléments qui le composent.
Problème :
Le code doit être le plus modulaire possible notamment pour permettre éventuellement de rajouter de nouvelles formes ou de nouvelles fonctionnalités pour dessiner facilement.
Solution :
Nous avons utilisé une architecture en couches reposant sur des relations d'aggrégation, de composition et d'héritage. Certaines méthodes communes à différents objets sont implémentées dans des classes

abstraites telles que Forme et Element qui constituent la couche haute de cette implémentation et permettent d'éviter la redondance dans le code. La couche basse représente les classes concrètes d'implémentation comme Cercle, Dessin, Chemin etc.



Les différents objets dont le nom est de la forme IObjet sont des interfaces, les classes Element et Forme sont des classes abstraites

3. Utilisation de la méthode Builder :

Contexte :
Nous devons générer un script qui contient les 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);
```

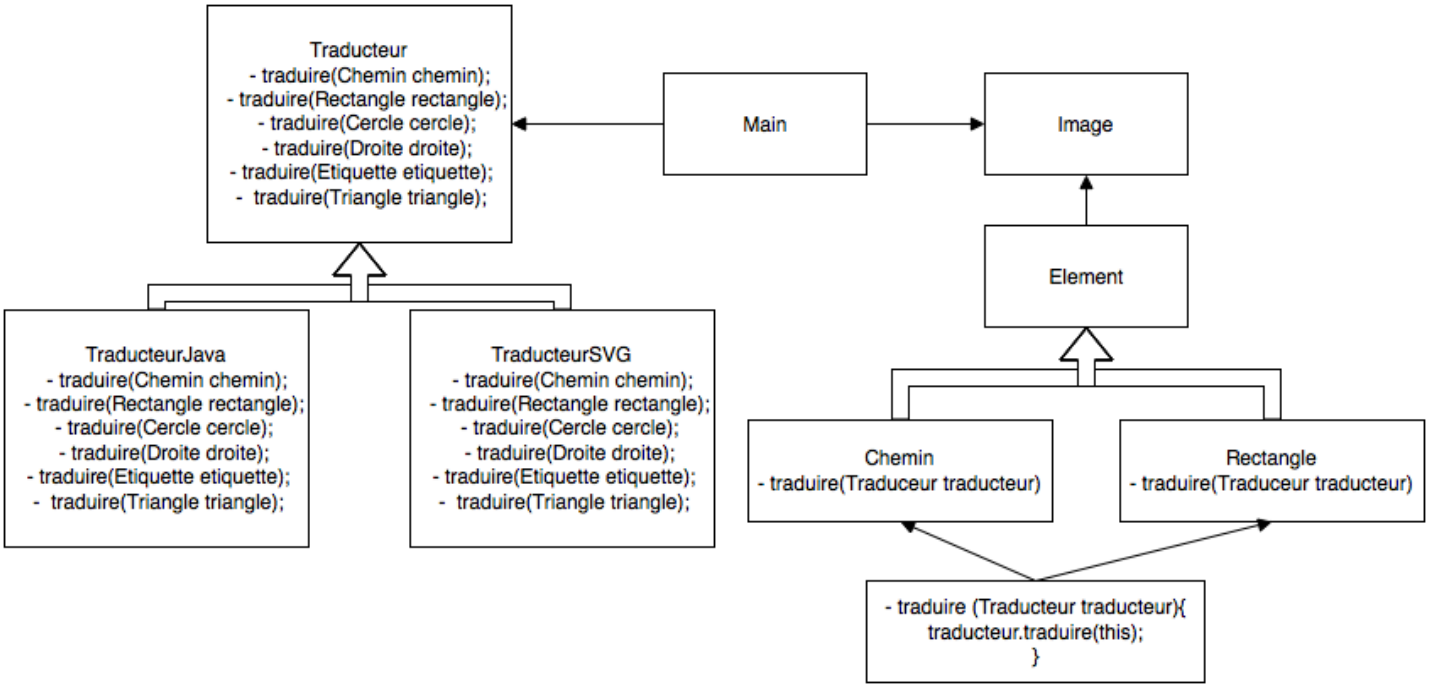
nous permettant de créer un point.

4. Utilisation du patron visiteur pour la traduction :

Contexte :
La logique du dessin vectoriel a donc été implémentée et il est désormais possible de générer des scripts utilisant notre langage. Il reste désormais à réaliser l'étape de traduction de ce script pour pouvoir visualiser le dessin.
Problème :
La partie traduction ne doit pas imposer de contraintes à notre langage et il se peut qu'un utilisateur souhaite ajouter de nouveaux modes de traduction. Il est évidemment préférable qu'il puisse le faire aisément sans avoir à modifier du code existant.
Solution :
Pour résoudre ce problème, le patron Visiteur est utilisé car il permet de visiter toutes les formes de l'image indépendemment du traducteur. C'est lors de la visite de chaque Element, que l'implémentation du traducteur va être appelé avec en paramètre l'élément à traduire.

Dans notre cas, nous avons choisi de réaliser la traduction en utilisant deux classes de traductions pour les deux implémentation suivantes : TraductionJava2D et TraductionSVG, ces deux classes implémentant l'interface Traducteur. L'interface Traducteur a autant de méthode traduire(…) que de forme existantes (traduire(Carre carre), traduire(Chemin chemin) etc.).

Description du patron visiteur appliqué à notre problème :



Ici seuls les éléments Chemin et Rectangle sont mentionnés pour plus de lisibilité mais chaque Element possède une méthode traduire

5. Points à améliorer :

- Insérer : à notre sens, la fonction d'insertion doit permettre d'insérer une forme dans une autre forme. Nous ne savions pas comment déterminer si une forme peut ou ne peut pas être insérée à l'intérieur d'une autre. Nous n'avons donc pas implémenté cette fonctionnalité.