Simon FITOUSSI
Eelaman KETHEESWARAN
Thibault ROMANN

Projet d'architecture logicielle - Un langage embarqué pour dessiner des diagrammes de types

Encadrant : Hervé Grall
12 Décembre 2016

Git : https://github.com/Fitousimon/Projet_ArchitectureLog_Romann-Fitoussi-Ketheeswaran

I/ Définition du modèle du langage

L'architecture comprend 2 couches : La **couche haute** permettant de définir le script du langage (les séquences d'instructions) et la **couche basse** permettant de l'interpréter.

Nous avons créé un paquet *diagramme* contenant les paquets *relation*, *graphe*, et *type*. Ce paquet permet de définir un diagramme des classes avec donc les composants et les relations les liants.

Définir une action

- **Ajouter** : Cette fonctionnalité permet d'ajouter un élément au diagramme en spécifiant sa position si nécessaire. Pour ajouter un élément, on procède comme suit :
    - On commence par définir des variables d'instance :

```
Variable v1 = new Variable("int", "v1");
Variable v2 = new Variable("String", "v2");
```

    - Puis on définit une méthode de classe :

```
Methode m1 = new Methode("int", "getV1");
```

    - Puis de même :

```
Variable param = new Variable("int", "argument");
Methode m2 = new Methode("void", "getV2",Arrays.asList(param));
```

    - Et enfin on crée la classe correspondante grâce à l'utilisation d'ArrayLists ; on définit l'épaisseur de trait et la couleur des éventuels représentations graphiques :

```
Classe Classe1 = new Classe(10, "BLEU", "Classe1",
            Arrays.asList(v1, v2), Arrays.asList(m1, m2));
```

- De même pour ajouter une relation, en supposant avoir à disposition une interface appelée Interface, on peut créer une relation d'implémentation comme suit :

```
Implements i = new Implements("ROUGE", 3, Classe1, Interface);
```

- Enfin, on crée le diagramme général :

```
Diagramme d = new Diagramme(Arrays.asList(i), Arrays.asList(Classe1),
            Arrays.asList(Interface));
```

- Et on finit par l'interprétaion de ce diagramme, ici l'interprétation textuelle :

```
InterpretationTexte inter = new InterpretationTexte(d);
      inter.interpretation();
```

- **Décrire** : Cette fonctionnalité permet de décrire un type, non seulement son contenu (variables, méthodes) mais aussi ses relations.
    - La fonction *display*, dans l'interface IGraph affiche les méthodes et les variables d'instance d'un élément (type ou relation) défini par l'utilisateur.
    - L'introspection est une technique qui sert à accéder de façon dynamique aux informations concernant une classe. Nous avons utilisé cette méthode afin de décrire une classe préexistante dans le but de l'intégrer à un diagramme. Elle permet notamment de récupérer les variables déclarées, la signature des méthodes, les classes parentes et les interfaces implémentées. Toutes les méthodes d'introspection sont définies dans la classe *introspection.ClasseInspecteur,* implémentée en suivant un guide trouvé sur Internet.

  Ainsi les méthodes :
    - **String**[] **getClasseParentes**(**String** nomClasse)
    - **String getInterface**(**String** nomClasse)
    - **String**[] **getMethodesNom**(**String** nomClasse)
    - **String**[] **getMethodesType**(**String** nomClasse)
    - **String**[] **getChampsNom**(**String** nomClasse)
    - **String**[] **getChampsType**(**String** nomClasse)

  de cette classe permettent l'interprétation de classes préexistantes.

  La démonstration de la description introspective se trouve dans le Main de la classe actions.Describe.

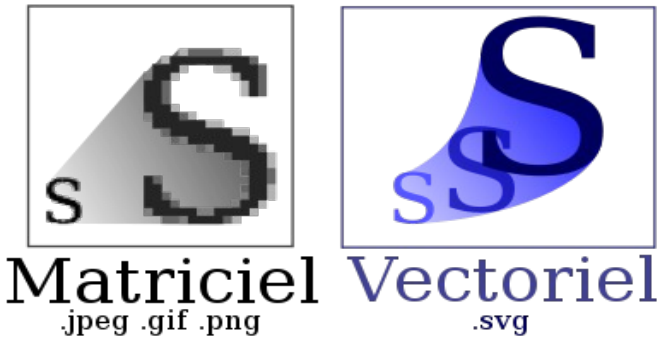
- **Insérer** : Cette fonctionnalité permet d'insérer un diagramme limité par un contour à l'intérieur d'un autre diagramme. L'intérêt du patron de conception Composite est que tous les objets ont le même type parent (ici Graph), ainsi un diagramme sera insérable dans un autre grâce à la même fonction display() que pour les simples objets. Cependant, nous n'avons pas rempli cet objectif..

- **Etiqueter** : Cette commande permet de placer une étiquette à une position donnée à l'intérieur d'un diagramme. De même, cette fonction serait possible à implémenter grâce au patron Composite ; il suffirait d'ajouter une class Label implémentant Graphic. Mais de même nous ne sommes pas allé au bout de cet objectif.

II/ Interprétation du langage

Il était demandé dans le sujet au moins deux interprétations, dont l'une étant une interprétation au format SVG.

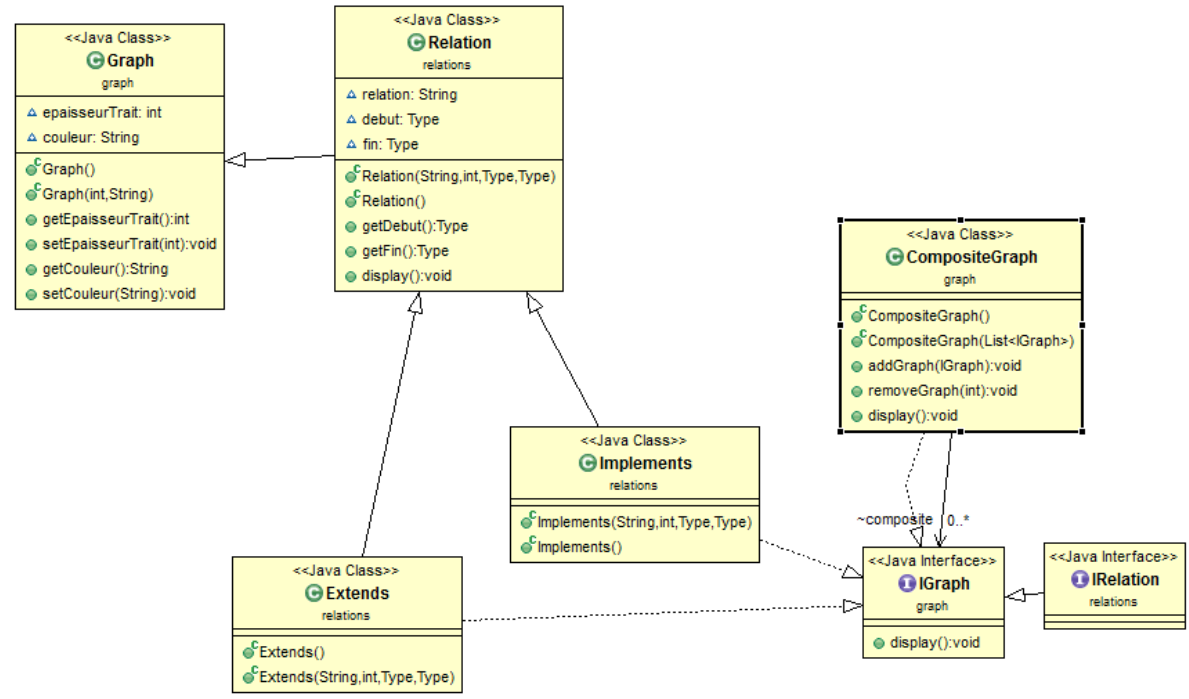
SVG est un format d'images vectorielles basé sur le langage de balisage XML. Il répond parfaitement à des besoins graphiques légers, et donc à la création de diagrammes UML des classes. Nous avons opté pour un rendu lisible par un navigateur Internet.
Pour l'instant, nous n'avons pas commencé l'implémentation d'un générateur de code SVG , mais certaines des fonctions sont d'ores et déjà été déclarées.
Cette interprétation n'a malheureusement pas été développée en raison de difficultés à appréhender le Toolkit Apache Batik, qui semblait approprié. Nous avons donc préféré utiliser le temps qui nous était disponible à faire la seconde interprétation.

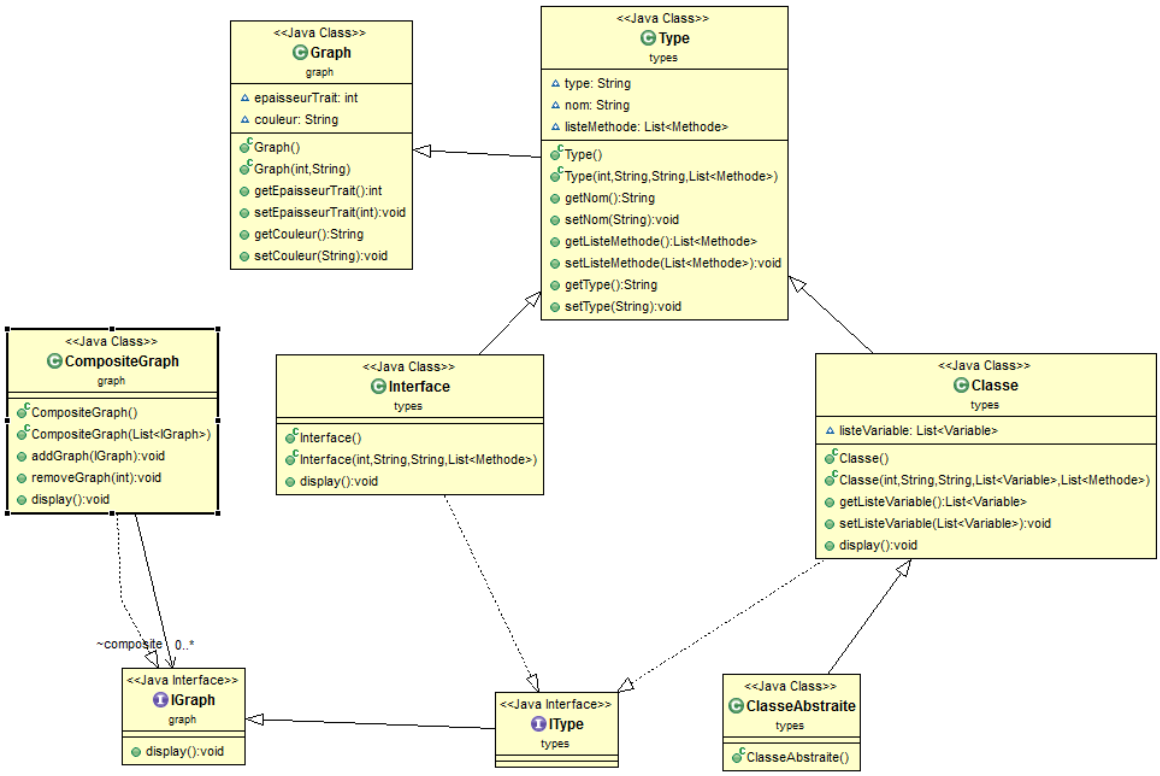


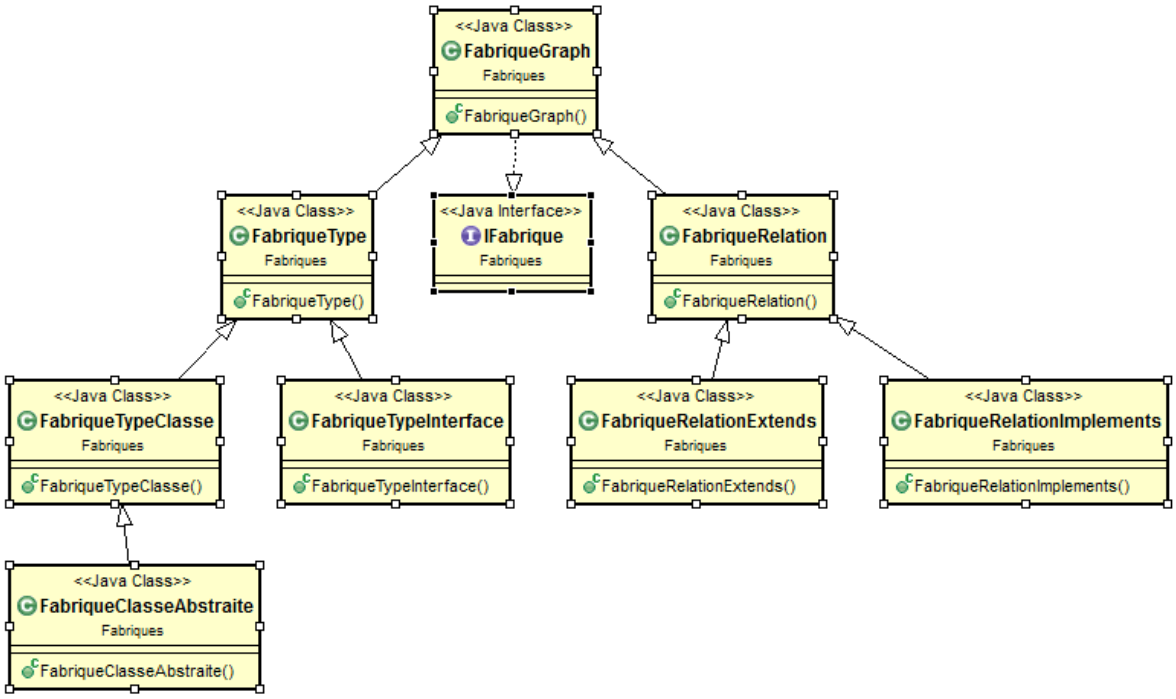
Le second interpréteur est un interpréteur texte qui décrit linéairement le diagramme UML. Il se sert des fonctions *display()* des classes filles de Type. Un exemple de cet interpréteur est donné dans la classe Main.main de notre projet.

III/ Architecture logicielle

1) Patron de conception







2) Patron de conception

Nous avons décidé d'utiliser un patron de conception de type Composite, car celui-ci semblait le plus adapté à notre projet, et ce pour plusieurs raisons ; cette structure permet tout d'abord de traiter de la même façon tous les différents objets qui apparaîtront sur notre graphe. Ces objets (Graph, Classe, Relation, …) seront en effet manipulables par les même fonctions, et notamment imprimables grâce aux même fonctions, ce qui s'adapte parfaitement à notre cas.

De plus, ce patron de conception s'associe bien, selon plusieurs sources internet, avec un patron visiteur. Or, ce patron de conception s'avère très utile dans ce projet, pour permettre à l'utilisateur de choisir entre les différents types d'interprétation (SVG ou texte). Cependant, bien que nous ayons compris l'utilité de ce design pattern, nous n'avons pas réussi à l'implémenter correctement dans notre code..

De même, le patron Singleton semble utile dans ce projet, notamment pour s'assurer qu'il n'y ait qu'un seul Diagramme sur lequel travailler et rajouter les différents éléments du Graph final. Ce patron fonctionne grâce à un constructeur privé, et une fonction Get() vérifiant si une instance de ce Diagramme a déjà été créée auparavant (en quel cas la variable d'instance ne sera plus égale à null), la renvoie si c'est le cas, et en créée une nouvelle sinon. De même nous n'avons pas réussi à passer de cet aspect théorique à une réelle implémentation dans notre code, mais un exemple de structure en singleton a tout de même été créée dans la classe Singleton.

Enfin, la dernière structure utilisée est celle des Fabriques pour la création des différents éléments du Graph, en reproduisant l'architecture utilisée pour ces objets, comme montré sur le schéma UML des Fabriques ci-dessous.

3) Extension du langage

Un des avantages du patron composite que nous n'avons pas mentionné dans la partie précédente, est que celui-ci s'adapte bien aux extensions de langages, tels que des rajouts d'objets ou de feuilles.
Ainsi on peut notamment ajouter un nouvel objet imprimable au même titre que les classes et les liaisons ; il suffit de créer une nouvelle classe MyObject implémentant la classe Graph, et de définir MyObject.display(), et enfin de l'ajouter à un CompositeGraph grâce à la fonction add().
Cet objet sera alors display() sur le diagramme au même titre que les autres objets déjà implémentés.
De plus, des fonctions permettant de changer la couleur, la forme ou encore l'épaisseur de traits sont déjà déclarées dans les types Type et Relation, mais nous ne sommes pas encore allé plus loin.
Enfin, il suffit de rajouter une Fabrique correspondante au nouvel objet implémenté.