Architecture logicielle

Rapport sur le mini-projet

Florian Ducret - Yang Liu - Corentin Normant - 11 décembre 2016

Depot Git : https://github.com/lixxday/TParchilogiciel

Définition du langage de dessin

Le langage de dessin que nous fournissons est majoritairement basé sur le langage UML. Il permet de créer des diagrammes de classes et fournit les fonctionnalités suivantes.

- Représentation des Classes et des Interfaces.
- Représentation des constructeurs et méthodes dans les types ci-dessus.
- Représentation des relations entre différents types.

Le langage en lui-même consiste à transformer toutes les relations et types en un type général afin de mieux gérer l'ajout de composants.

L'interprétation du langage

Type général = rectangle

Haut niveau : Transformer du java en rectangle (DSL)

Bas niveau : Transformer des rectangles en dessin

Le pivot de base de notre langage est la classe Rectangle qui est considérée comme la brique de base pour construire n'importe quel schéma. Un rectangle contient trois parties : un nom, un contenu regroupant par exemple pour une classe, des constructeurs et des méthodes, et des dépendances. Une dépendance sera un autre Rectangle qui participe à sa construction. Par exemple, une classe qui implémente une interface contiendra le nom de celle-ci dans ses dépendances.

L'application est composée de deux parties. Une haut niveau qui permet l'interprétation de code JAVA en Rectangle. Une bas niveau qui transforme ces Rectangles en dessin.

Le but du haut niveau est de référencer des classes et interfaces existantes d'un autre projet (ici app.test), et de les traduire en Rectangles grâce à un visiteur dont nous détaillerons l'architecture après. La partie de la classe « main » ci-dessous représente donc le haut niveau.

```
public class Main {

    public static void main(String[] args) {

        IVisiteur visiteur = new Visiteur();

        ClassRepresentation maClasse1 = new ClassRepresentation(MaClasse1.class);
        ClassRepresentation maClasse2 = new ClassRepresentation(MaClasse2.class);
        InterfaceRepresentation monInterface1 = new InterfaceRepresentation(MonInterface1.class);

        visiteur.visit(maClasse1);
    }
}
```

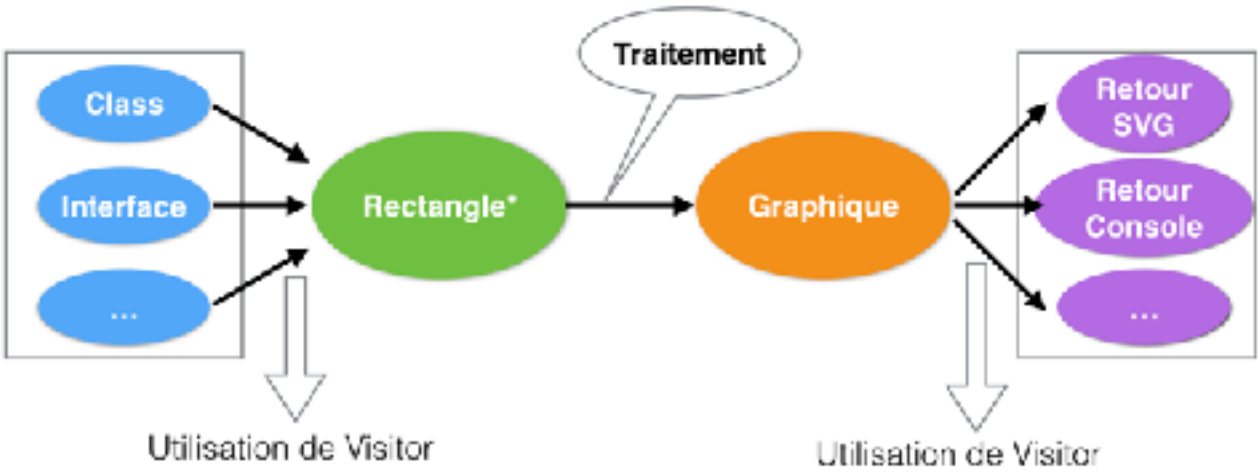
Le bas niveau récupère les Rectangles générés et peut les interpréter de plusieurs façons. De même, pour écrire un code modulable, cette couche utilise un visiteur qui permet d'ajouter facilement une autre forme de sortie, par exemple jpeg.

Par exemple, si on choisit d'écrire les résultats au format SVG, on veut pouvoir écrire un code comme ci dessous.

```
public class Main {

    public static void main(String[] args) {

        // Haut niveau
        IVisiteur visiteur = new Visiteur();

        ClassRepresentation maClasse1 = new ClassRepresentation(MaClasse1.class);
        ClassRepresentation maClasse2 = new ClassRepresentation(MaClasse2.class);
        InterfaceRepresentation monInterface1 = new InterfaceRepresentation(MonInterface1.class);

        // Bas niveau
        Graphique graph = new Graphique();
        IDessineur dessineur = new Dessineur(graph);
        RetourSVG retour = new RetourSVG();
        RetourConsole retour2 = new RetourConsole();

        graph.addGraphique(visiteur.visit(maClasse1));
        graph.addGraphique(visiteur.visit(maClasse2));
        graph.addGraphique(visiteur.visit(monInterface1));

        dessineur.visit(retour);
        dessineur.visit(retour2);
    }
}
```

L'architecture logicielle

Schéma général



Rectangle = L'objet qui fait le lien entre la couche haute et la couche basse.

Patrons de conception utilisés

Nous avons utilisé le patron de conception Visiteur dans deux endroits.

D'un côté pour transformer les différentes représentations hautes du diagramme en une représentation basse commune à toutes les représentations permettant ainsi d'être modulaire.

De l'autre côté pour permettre plusieurs sorties. C'est-à-dire de fournir des diagrammes suivant différentes représentations allant de la console à un diagramme image.

Nous avons choisi ce patron de conception pour rendre notre code très modulaire. En effet, si un programmeur souhaite ajouter un nouvel objet à représenter ou un nouveau type de diagramme en sortie, il lui est très facile de modifier le code puisqu'il n'a qu'à modifier les deux visiteurs implémentés.

Méthode pour étendre le langage et ajouter une interprétation

Il existe trois possibilités pour étendre le langage. On peut ajouter un nouveau type à dessiner, comme le type Enum de Java, ajouter un nouveau type de sortie (diagramme en JPEG par exemple) et modifier les caractéristiques de la couche basse à travers la classe Rectangle, ajouter une couleur par exemple.

Pour ajouter un nouveau type, il suffit de créer une classe dans le package app.visiteur, qui implémente IVisitable, avec les caractéristiques souhaitées. Ensuite, dans l'interface IVisiteur, on ajoute la méthode visit(TypeAjoute o). Dans Visiteur, on implémente cette méthode pour qu'elle retourne un Rectangle adapté à la représentation.

Pour ajouter un nouveau type de sortie, il suffit de créer dans le package app.representation une classe représentant le type de retour, cette classe implémente IDessinable. Ensuite, dans l'interface IDessineur, on ajoute la méthode visit(TypeRetourAjoute o), et on implémente cette méthode dans la classe Dessineur, que l'on adapte à ses besoins.

Si l'on souhaite modifier la représentation de la couche basse, il suffit de modifier la classe Rectangle, en lui ajoutant un attribut par exemple. Dans le cas où on veut modifier en profondeur cette classe, il faudra modifier l'ensemble de l'application. En revanche, si on ne fait qu'ajouter un attribut quelconque comme une couleur, il suffit d'ajouter un attribut, un getter et un setter à la classe Rectangle.

Pistes d'amélioration

La principale amélioration à apporter à notre application est son rendu final. En effet, même si la sortie en texte est claire, l'implémentation de la sortie en SVG est très légère. Un travail sur les représentations des dépendances et sur le placement des différents rectangle est indispensable pour pouvoir représenter des systèmes plus complexes que notre exemple.
Entre autres, ajouter des pointes aux flèches, les pointiller lorsqu'il s'agit de représenter l'implémentation d'une interface, ou colorer les différents rectangles.