Projet d'architecture logicielle – Un langage de dessin vectoriel

Rapport de 1er sprint

Groupe : Cécile Martel, Mathieu Guldner, Pierre Legault

Github du projet : [mguldner/Architecture\_Logicielle\_SVG](https://github.com/mguldner/Architecture_Logicielle_SVG)

1. Terminologie utilisée

Afin de répondre au [cahier des charges](http://grall.name/teaching/softwareArchitecture/2016/project_vectorGraphics.html), 4 types d'éléments ont été retenus:

1. Action : opération élémentaire produisant un dessin. De base, les actions implémentées sont Draw (dessiner), Fill (remplir), Insert (insérer), Labelise (étiqueter).
2. Path (chemin) : ensemble de points reliés d'une certaine manière. Ce chemin peut être fermé ou non. De base, les chemins implémentés (i.e. les manières de relier les points) sont PolygonalPath (polygone) et BezierPath (courbe de Bézier).
3. Tool (outil) : objet qui permet de tracer ou d'écrire. De base, les outils implémentés sont Pen (crayon) et TextTool (outil de texte).
4. Operator (opérateur) : opération algorithmique permettant de définir comment il faut réaliser un dessin. De base, les opérateurs implémentés sont Sequence (séquence), Alternative et Loop (boucle).
5. Définition du langage de dessin
6. Définir un gestionnaire de couleur
7. Pour pouvoir générer la représentation des couleurs (code RGB, code hexadécimal…), un système de gestionnaire de couleurs a été mis en place.
8. Pour initialiser un gestionnaire de couleur :
9. ColorManager cm = new ColorManager("MODE\_CHOISI", CODE\_COULEUR);
10. Où :

* MODE\_CHOISI = « rgb » ou « hex »
* CODE\_COULEUR = int[3] représentant des valeurs de 0 à 250 ou une string représentant le code hexadécimal de la couleur.

**Définir un chemin**

Par défaut, il existe plusieurs types de chemins implémentés : le chemin polygonal et la courbe de Bézier.

Pour initialiser un chemin :

Path polygonalPath = new PolygonalPath(POINTS, CHEMIN\_FERME);

ou

Path bezierPath = new BezierPath(POINTS, CHEMIN\_FERME);

Où :

* POINTS  = Point2D[], le tableau des points par lesquels passe le chemin
* CHEMIN\_FERME = booléen à true si le chemin est fermé, ou à false sinon

**Définir un outil**

**Définir un crayon**

Pour initialiser un crayon :

Tool pen = new Pen(GESTIONNAIRE\_COULEUR, TAILLE);

Où :

* GESTIONNAIRE\_COULEUR = objet du type ColorManager
* TAILLE (optionnel) = int représentant l'épaisseur du trait

**Définir un outil de texte**

Pour initialiser un outil de texte :

Tool textTool = new TextTool(GESTIONNAIRE\_COULEUR, NOM\_POLICE, TAILLE\_POLICE);

Où :

* GESTIONNAIRE\_COULEUR = objet du type ColorManager
* NOM\_POLICE (optionnel) = string représentant le nom de la police
* TAILLE\_POLICE (optionnel) = int représentant la taille de la police

**Définir une action**

**Dessiner**

Dessine un chemin avec un outil.

Drawing draw = new Draw(PATH, TOOL);

Où :

* PATH = objet du type Path
* TOOL = objet du type Tool

**Remplir**

Remplit un chemin avec une couleur

Drawing fill = new Fill(PATH, GESTIONNAIRE\_COULEUR);

Où :

* PATH = objet du type Path
* GESTIONNAIRE\_COULEUR = objet du type ColorManager

I**nsérer**

Insère un dessin dans un chemin. Ne retiens que la partie du dessin dans le chemin.

Drawing insert = new Insert(DESSIN, PATH);

Où :

* DESSIN = objet du type Drawing
* PATH = objet du type Path

**Étiqueter**

Place un texte à l'endroit désiré.

Drawing label = new Label(TEXTE, ORIGINE, OUTIL\_TEXTE);

Où :

* TEXTE = string contenant le texte à afficher
* ORIGINE = Point2D représentant l'origine de l'emplacement du texte
* OUTIL\_TEXT = objet du type TextTool

**Définir un opérateur**

**Séquence**

Effectue le rendu de plusieurs dessins les uns à la suite des autres.

Drawing sequence = new Sequence(DESSINS);

Où :

* DESSINS = objet du type Drawing[]

**Alternative**

Effectue le rendu d'un dessin ou d'un autre.

Drawing alternative = new Alternative(DESSINS, PREMIER\_DESSIN);

Où :

* DESSINS = objet du type Drawing[2]
* PREMIER\_DESSIN = booléen à true si on veut représenter le premier dessin, à false sinon

**Boucle**

Effectue une action en boucle en changeant des paramètres.

Drawing loop = new Loop(DESSINS, NB\_ITERATTIONS, TYPE\_TRANSFORMATION, PARAMETRES);

Où :

* DESSINS = objet du type Drawing[]
* NB\_ITERATIONS = int, le nombre d'itérations de la boucle
* TYPE\_TRANSFORMATION = String représentant la transformation à effectuer au cours de la boucle. Pour l'instant peut etre « translation », « rotation », « scaling »
* PARAMETRES = Object[] qui prend les paramètres de changement à chaque itération
* Rotation : {double angle} angle à rajouter à chaque itération
* Translation : {double x, double y} quantités à ajouter à chaque coordonnée à chaque itération
* Scaling : {double horizontal, double vertical} facteurs d'échelle à appliquer à chaque itération.

1. L'interprétation du langage (ou les interprétations)

Pour l'instant, deux interprétations du langage ont été retenues : une interprétation SVG et une interprétation Java avec java.awt.Graphics2D.

* 1. Interprétation SVG

Cette interprétation correspond à la génération d'un code SVG valide représentant le dessin souhaité par l'utilisateur. Le moteur de rendu est externalisé, étant généralement un navigateur web.

Pour l'instant, cette interprétation génère le code SVG correspondant au dessin et l'affiche dans la console, mais aucun fichier d'export .svg n'est créé. Cette fonctionnalité sera implémentée lors du prochain sprint.

* 1. Interprétation Java avec Graphics2D

Cette interprétation correspond à l'affichage dans une fenêtre, du dessin souhaité par l'utilisateur. Le moteur de rendu est interne.

Cette interprétation est basée sur la bibliothèque java.awt.Graphics2D.

1. Architecture logicielle
   1. Schéma général

Le diagramme de classes ci-dessous tient lieu de schéma général.

Seulement les classes correspondant aux 4 types retenus (Action, Path, Tool, Operator) ont été placées sur le schéma afin d'en améliorer la clarté. Un exemple d'implémentation de chaque type est aussi présent, afin de montrer la structure dans son ensemble.

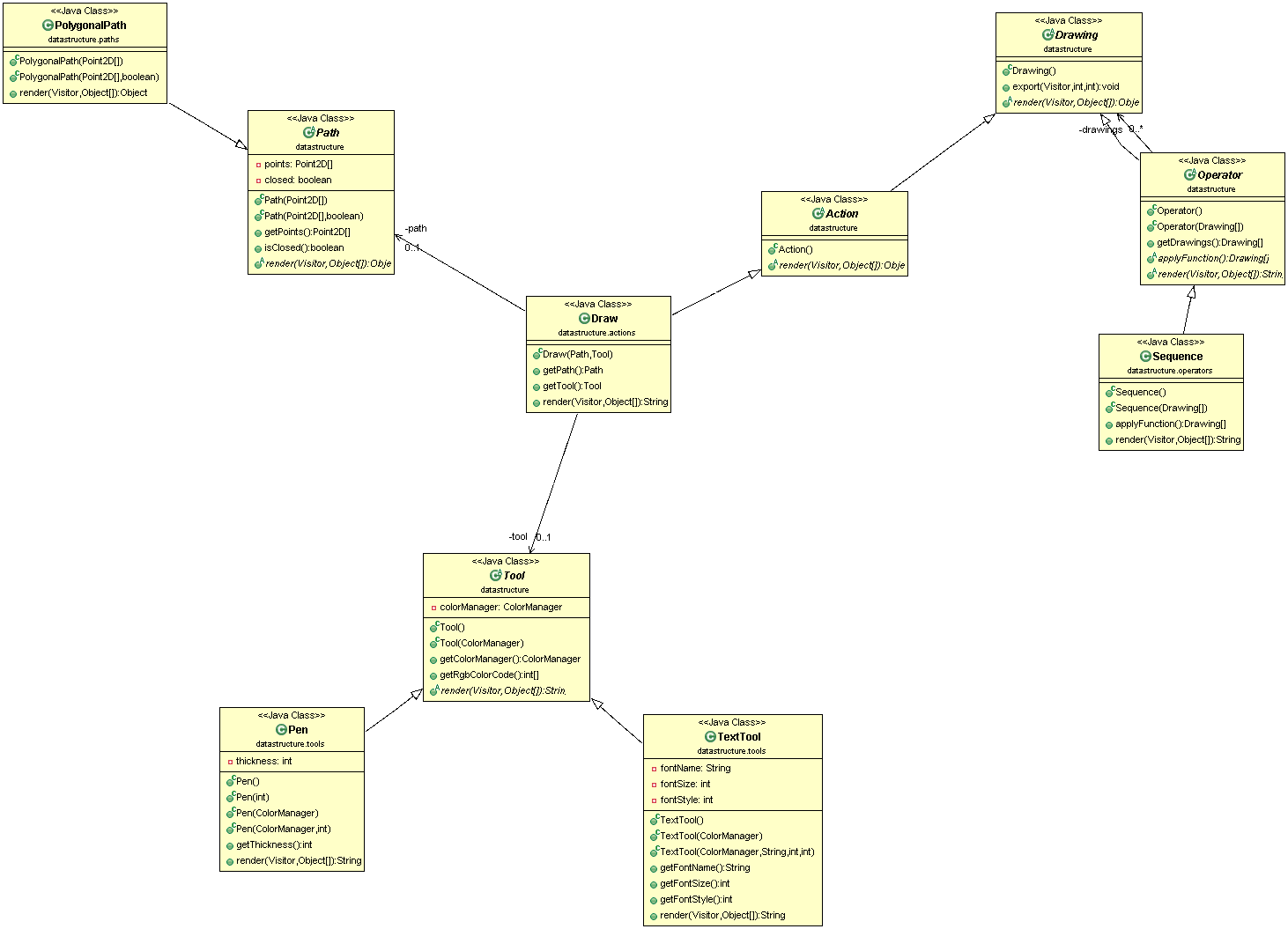


Diagramme de classes partiel du projet.

L'idée sur laquelle se base ce modèle est la suivante : pour faire un dessin, soit l'on effectue une action de base (ex : dessiner un carré rouge), soit l'on réunit un ensemble de dessins déjà existants (ex : placer un carré rouge sur un triangle vert entouré d'un cercle bleu). Ainsi un dessin (Drawing) peut être défini soit par une action (Action) soit par un ensemble d'actions liées par un opérateur (Operator).

* 1. Modularité et patrons de conception
     1. Patron composite

De part la structure d'arbre choisie pour représenter un dessin, la solution d'utiliser un patron composite paraissait adaptée. En effet, ce patron permet de manipuler de la même manière une feuille de l'arbre (une action) et un sous-arbre (un opérateur), ce qui est exactement le cas d'usage ici, où l'on souhaite pouvoir générer l'export de chaque élément du dessin.

Ce patron peut etre observé entre les classes Drawing, Action et Operator.

* + 1. Patron visiteur

Il a été décidé d'utiliser le patron visiteur dans le cadre des fonctions de rendu de chaque composant du dessin. En effet, pour chaque composant (drawing, tool et path), il est nécessaire d'implémenter une fonction de rendu pour chaque mode d'export. Le patron visiteur permet alors d'avoir tout le code de rendu propre à un mode d'export présent dans un seul fichier au lieu d'être réparti dans chacun des composants. L'ajout ou la modification d'une interprétation en est alors facilitée.

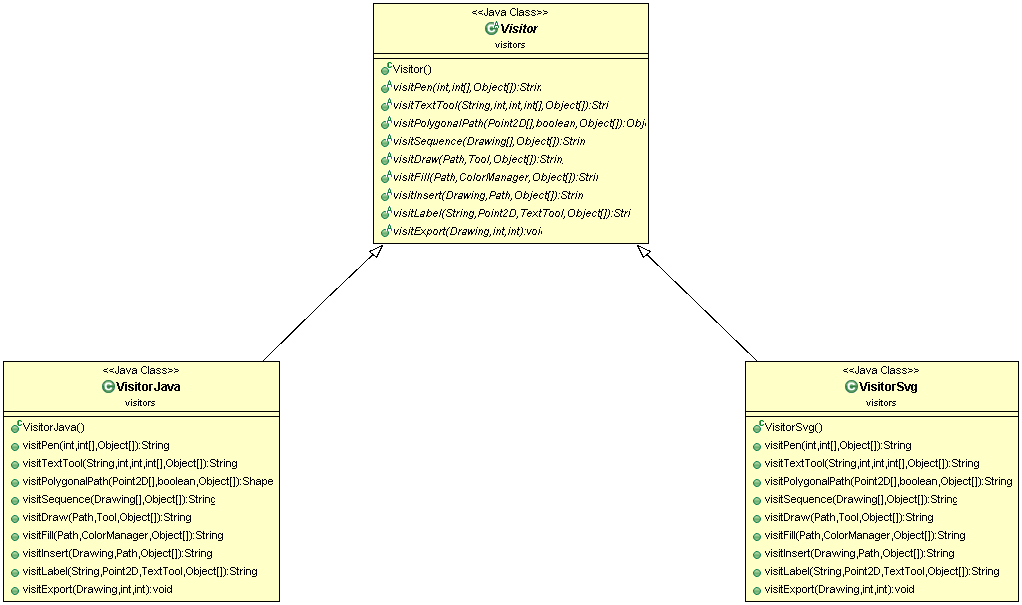


Diagramme de classe montrant la structure d'implémentation du patron visiteur.

* + 1. Cas du gestionnaire de couleurs

Afin de permettre à l'utilisateur de pouvoir utiliser son format de couleur préféré, il a été choisi de créer un gestionnaire de couleur. Celui-ci permet à l'utilisateur d'ajouter une nouvelle façon de coder une couleur (ex : RGBa, code hexadécimal…) sans que cela n'impacte le reste du code.

* 1. Étendre le langage et ajouter une interprétation
     1. Ajouter une action

Pour ajouter une action il suffit de :

1. Créer une classe qui étend la classe Action, la placer dans le package datastructure.actions
2. Définir ses variables d'instance, ses constructeurs et une fonction render appelant un visiteur dédié et sa méthode visitVOTREACTION
3. Ajouter une méthode abstraite visitVOTREACTION à la classe Visitor
4. Implémenter pour chaque mode d'export (dans chaque fichier visiteur) une fonction visiteur adaptée.
   * 1. Ajouter un chemin

Pour ajouter un type de chemin, il suffit de :

1. Créer une classe qui étend la classe Path, la placer dans le package datastructure.paths
2. Définir ses variables d'instance, ses constructeurs et une fonction render appelant un visiteur dédié et sa méthode visitVOTRECHEMIN.
3. Ajouter une méthode abstraite visitVOTRECHEMIN à la classe Visitor
4. Implémenter pour chaque mode d'export (dans chaque fichier visiteur) une fonction visiteur adaptée.
   * 1. Ajouter un outil

Pour ajouter un type d'outil, il suffit de :

1. Créer une classe qui étend la classe Tool, la placer dans le package datastructure.tools
2. Définir ses variables d'instance, ses constructeurs et une fonction render appelant un visiteur dédié et sa méthode visitVOTREOUTIL.
3. Ajouter une méthode abstraite visitVOTREOUTIL à la classe Visitor
4. Implémenter pour chaque mode d'export (dans chaque fichier visiteur) une fonction visiteur adaptée.
   * 1. Ajouter un opérateur

Pour ajouter un opérateur, il suffit de :

1. Créer une classe qui étend la classe Operator, la placer dans le package datastructure.operators
2. Définir ses variables d'instance, ses constructeurs et une variable applyFunction() qui retourne la liste des dessins que l'on va représenter.
3. Définir une fonction render appelant un visiteur dédié, sa méthode visitVOTRECHEMIN avec comme paramètre un appel à la fonction applyFunction en tant que Drawing[]
4. Ajouter une méthode abstraite visitVOTRECHEMIN à la classe Visitor
5. Implémenter pour chaque mode d'export (dans chaque fichier visiteur) une fonction visiteur adaptée.
   * 1. Ajouter une interprétation

Pour ajouter un type d'interprétation, il suffit de :

1. Créer une classe qui étend la classe Visitor, la placer dans le package visitors
2. Implémenter chacune des méthodes nécessaires.