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

Terminologie utilisée

Afin de répondre au cahier des charges, 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).

Définition du langage de dessin

Définir un gestionnaire de couleur

Pour pouvoir générer la représentation des couleurs (code RGB, code hexadécimal…), un système de gestionnaire de couleurs a été mis en place.

Pour initialiser un gestionnaire de couleur :

```
ColorManagerFactory cmf = new ColorManager();
ColorManager cm = cmf.createColorManager("MODE_CHOISI", CODE_COULEUR);
```

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 :

```
PathFactory polygoPathF = new PolygonalPath();
Path polygonalPath = polygoPath.createPath(POINTS, CHEMIN_FERME);
```

ou

```
PathFactory bezPathF = new BezierPath();
Path bezierPath = bezPathF.createPath(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 :

```
PenFactory penF = new Pen();
Tool pen = penF.createPen(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 :

```
TextToolFactory textToolf = new TextTool();
Tool textTool = textToolf.createTextTool(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.

```
DrawFactory drawF = new Draw();
Drawing draw = drawF.createDraw(PATH, TOOL);
```

Où :

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

Remplir

Remplit un chemin avec une couleur

```
FillFactory fillF = new Fill();
Drawing fill = fillF.createFill(PATH, GESTIONNAIRE_COULEUR);
```

Où :

- PATH = objet du type Path
- GESTIONNAIRE_COULEUR = objet du type ColorManager

Insérer

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

```
InsertFactory insertF = new Insert();
Drawing insert = insertF.createInsert(DESSIN, PATH);
```

Où :

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

Étiqueter

Place un texte à l'endroit désiré.

```
LabelFactory labelF = new Label();
Drawing label1 = labelF.createLabel(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.

```
SequenceFactory seqF = new Sequence();
Drawing sequence = seqF.createSequence(DESSINS);
```

Où :

- DESSINS = objet du type Drawing[]

Alternative

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

```
AlternativeFactory altF = new Alternative();
Drawing alternative = altF.createAlternative(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.

```
LoopFactory loopF = new Loop();
Drawing loop = loopF.createLoop(DESSINS, NB_ITERATTIONS, TYPE_TRANSFORMATION, PARAMETRES);
```

Où :

- DESSINS = objet du type Drawing[]
- NB_ITERATIONS = int, le nombre d'itérations de la boucle
- PARAMETRES = HashMap qui prend en clé des chaînes de caractères définissant les types de transformation (« rotation », »translation » ou « scaling »), et en valeur les paramètres de changement pour chaque transformation.
    - 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.

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.

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.

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.

Architecture logicielle

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.

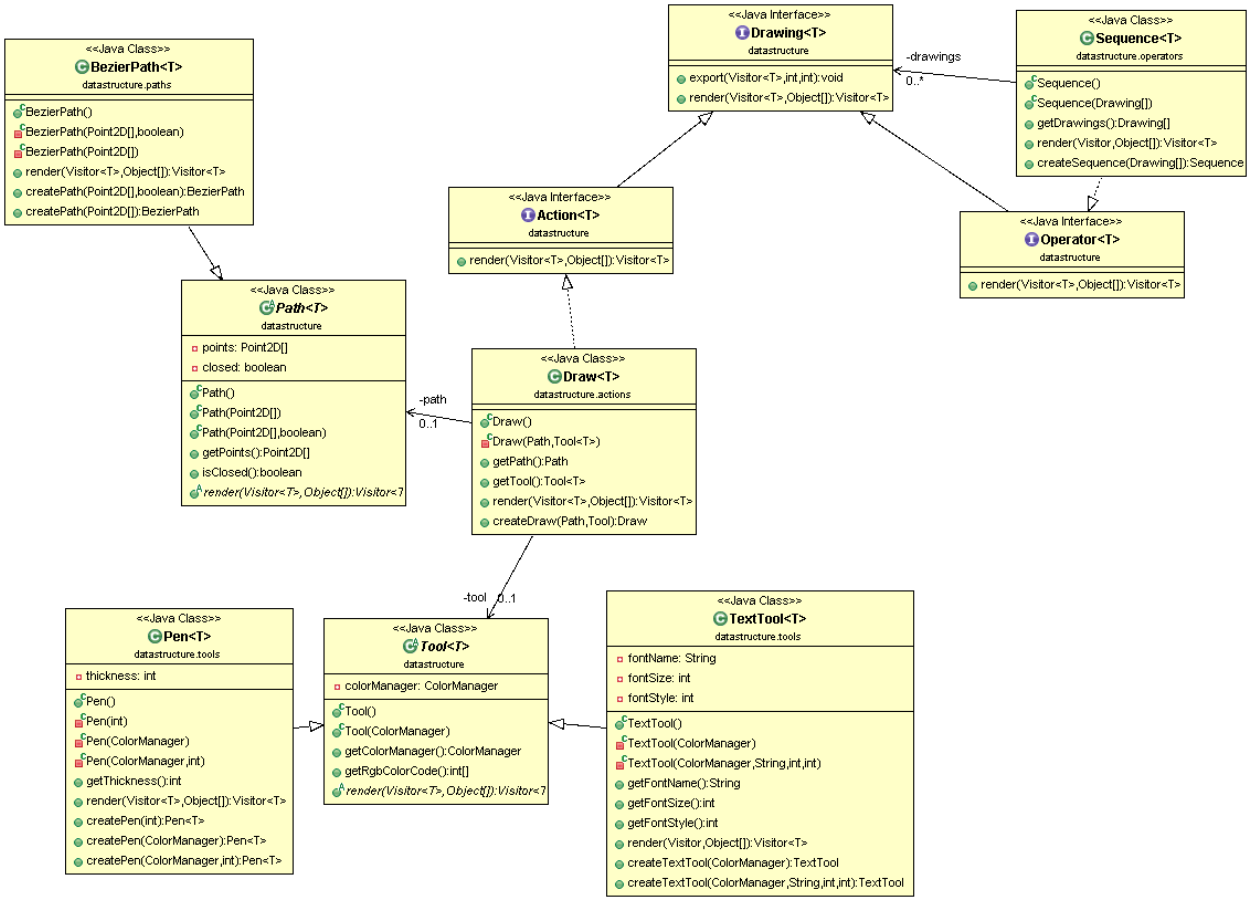

Diagramme de classes partiel du projet.

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).

Modularité et patrons de conception

Patron composite

De par 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 être observé entre les classes Drawing, Action et Operator.

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.

Patron factory

Un patron factory a été utilisé ici également, afin de séparer davantage le langage utilisateur du modèle. Ainsi la dépendance relativement aux classes d'implémentation est restreinte aux fabriques.

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.

Étendre le langage et ajouter une interprétation

Ajouter une action

Pour ajouter une action il suffit de :

1. Ajouter une interface VOTREACTIONFactory dans le package factories.action, avec une fonction createVOTREACTION.
2. Créer une classe qui étend la classe Action, et implémente VOTREACTIONFactory, la placer dans le package datastructure.actions.
3. Définir ses variables d'instance, ses constructeurs et une fonction render appelant un visiteur dédié et sa méthode visitVOTREACTION
4. Implémenter dans VOTREACTION la méthode createVOTREACTION.
5. Ajouter une méthode abstraite visitVOTREACTION à la classe Visitor.
6. Implémenter pour chaque mode d'export (dans chaque fichier visiteur) une fonction visiteur

Ajouter un chemin

Pour ajouter un type de chemin, il suffit de :

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

Ajouter un outil

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

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

Ajouter un opérateur

Pour ajouter un opérateur, il suffit de :

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

6. Ajouter une méthode abstraite visitVOTRECHEMIN à la classe Visitor
7. Implémenter pour chaque mode d'export (dans chaque fichier visiteur) une fonction visiteur adaptée.

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.