Architecture logicielle

Projet “Dessin vectoriel”

Oleksandr AVLESI

**Rendu final**

Le but du projet était de permettre de construire dans un programme en Java un script définissant un dessin logique, puis son interprétation permettant de représenter ce dessin en traduisant les instructions présentes dans le script en commandes associées à une interface graphique particulière. Le but de ce rapport c’est de clarifier une architecture du projet réalisé et expliquer quels étaient les raisons qui ont amené à l’implémentation qui a été faite.

Une classe principale de du projet s’appelle *Dessin*. Toutes les requêtes de la création des objets graphiques vont être adressées à cette classe. On peut la considérer comme une classe manager.

Les fonctions principales que cette classe fournit ce sont :

* draw(Chemin chemin) – permet de dessiner une figure
* label(Text text, int x, int y) – permet d’étiqueter une figure
* fill(Chemin chemin, Color color) – permet de remplir une figure d’un couleur
* insert(Chemin chemin, int x, int y, int width, int height) – permet de clipper une figure dans une zone carrée indiquée

*Chemin* c’est une classe des modèles des figures qui peuvent être dessinées. Pour créer des figures le client peut se servir des fonctions de Dessin suivantes :

* Chemin createPoint(int x, int y) – créer un point
* Chemin createLine(int x1, int y1, int x2, int y2) - créer une ligne
* Chemin createArc(int center\_x, int center\_y, int rx, int ry, int startAngle, int arcAngle) -créer un arc
* Chemin createEllipse(int center\_x, int center\_y, int rx, int ry) – créer un ellipse
* Chemin createCircle(int center\_x, int center\_y, int r) – créer une circle
* Chemin createRectangle(int x1, int y1, int x2, int y2, int x3, int y3, int x4, int y4) – créer un rectangle

Chaque figure peut avoir son propre couleur et ses paramètres de l’épaisseur. Ce paramétrage est fait à l’aide de la classe *Crayon*. A chaque figure on peut associer une instance de Crayon.

Pour créer un crayon, il faut se servir d’une fonction de *Dessin* correspondante :

* Crayon createCrayon(Color color, int thickness)

Pour associer un crayon avec une figure, il faut utiliser une fonction de Chemin

* setCrayon(Crayon crayon)

Si un crayon n’était pas associé explicitement, une figure va utiliser un crayon par défaut dont les paramètres c’est un couleur noir et une épaisseur d’une valeur 1.

Si un client veut obtenir une figure plus complexe, il peut se servir de telles fonctions de Chemin :

* Chemin connectWithLine(Chemin chemin) – fonction permet d’obtenir une nouvelle figure qui se compose de deux figures jointes par une ligne
* Chemin connectWithBezier(Chemin chemin, int x1, int y1, int x2, int y2) – fonction permet d’obtenir une nouvelle figure qui se compose des deux figures jointes par une courbe de Bézier. Ici x1, y1, x2, y2 ce sont des points de contrôle d’une courbe de Bézier cubique.

Dans le projet il y a deux implémentations de la représentation des dessins vectoriels – celle qui est basée sur la librairie awt de java et celle basée sur svg. Pour choisir une représentation que vous voulez, il faut fournir un paramètre correspondant à un constructeur de la classe Dessin.

Voici un exemple d’utilisation des fonctionnalités décrites ci-dessus:

// créer un dessin et choisir une représentation svg

Dessin d = **new** Dessin(DrawType.***SVG***);

// pour choisir une représentation awt il faut faire DrawType.AWT

//créer un crayon

Crayon pen = d.createCrayon(Color.***BLUE***, 1);

//créer un cercle en precisant le center (800, 800) et un radius (100)

Chemin cercle = d.createCircle(800, 800, 100);

//associer un crayon à un cercle

cercle.setCrayon(pen);

// remplir un cercle d'un couleur jaune

d.fill(cercle, Color.***YELLOW***);

//dessiner un cercle

d.draw(cercle);

// ajouter unу texte

d.label("Cool text", 800, 750);

// clipper un cercle dans une zone carrée indiquée

d.insert(cercle, 800, 800, 100, 100);

//faire une mise a jour du dessin du cercle. Si on le fait explicitement,

//la mise a jour n'est pas garantie

d.draw(cercle);

//créer une ligne

Chemin ligne = d.createLine(5, 5, 5, 15);

ligne.setCrayon(crayon);

d.draw(ligne);

//créer une ligne

Chemin ligne2 = d.createLine(500, 5, 500, 15);

d.draw(ligne2);

//joindre par une courbe be Bezier cubique.

Chemin path = ligne.connectWithBezier(ligne2, 200, 100, 400, 100);

d.draw(path);

//créer une ligne

Chemin ligne3 = d.createLine(600, 40, 650, 100);

d.draw(ligne3);

//joindre par une ligne

Chemin path2 = path.connectWithLine(ligne3);

d.draw(path2);

// voir un svg généré dans le navigateur

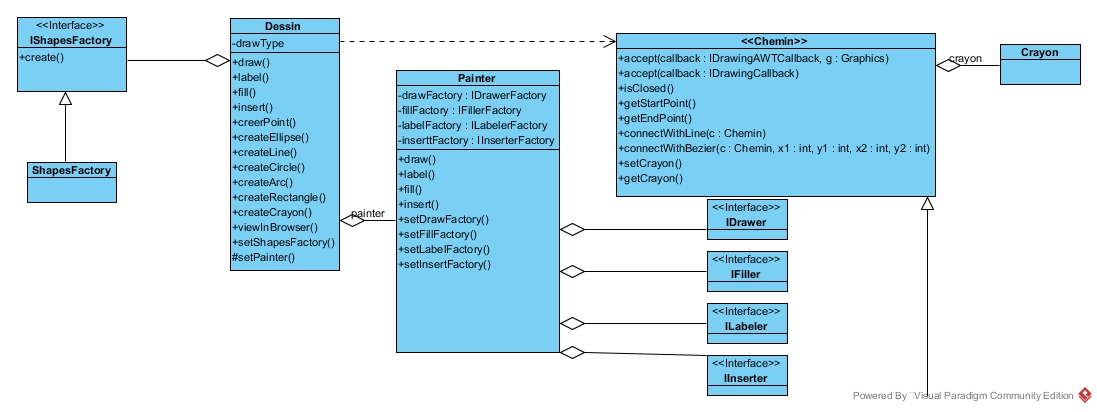
// si un autre DrawType était choisi, la fonction va pas avoir un effet

d.viewInBrowser();

On peut retrouver un exemple pareil dans la classe Main qui est dans le paquet ‘common’.

**Architecture**

Maintenant on va regarder plus en détail sur l’architecture du projet.



Class *Dessin* délègue les opérations liées à la peinture à une classe *Painter*. Cette approche assure une bonne flexibilité. Si en future on décide à changer l’implémentation des opérations de peinture, ça ne va pas toucher *Dessin*. Et si on veut ajouter une nouvelle fonction, on pourra sous-classer de *Dessin* en créant une classe quelconque *CustomDessin*, ajouter là-dedans cette fonction et lui fournir une sous-classe de *Painter* enrichie par l’implémentation de cette nouvelle méthode en appelant une fonction setPainter(Painter p).

Également Dessin délègue la création des figures à la fabrique. Cette fabrique implémente une interface IShapesFactory et si une fois on décide à changer les principes de la création des objets, on peut fournir à Dessin une nouvelle fabrique en appelant une méthode setShapesFactory(IShapesFactory f).

Maintenant parlons plus en détail d’une classe *Painter*. Cette classe agrège 4 interfaces – IDrawer, ILabeler, IFiller et IInserter qui ont des méthodes draw(), label(), fill(), insert() correspondantes. Les instances de ces interfaces sont créées par 4 fabriques correspondantes. Chaque fabrique implémente son interface. Alors par exemple à l’aide d’une méthode setDrawFactory(…) vous pouvez injecter dans le *Painter* une nouvelle factory qui va implémenter une interface IDrawerFactory et qui va générer des instances de l’interface IDrawer selon une autre logique quelconque.

Les classes qui implémentent une interface IDrawer présentent des interprétations graphiques des dessins vectoriels. Comme on a dit avant, on a deux interprétation – awt et svg, c’est pourquoi on a deux classes correspondantes – AwtDrawer et SvgDrawer. Ces classes s’occupent de la peinture. Elles utilisent des classes Filler, Labeler, Inserter (qui implémentent des interfaces IFiller, ILabeler et IInserter) pour stocker l’information correspondante. Par exemple, quand on va dessiner un cercle, dans la méthode de peinture on aura des bouts de code comme ça :

**if**(inserter.contains(c)) {…}

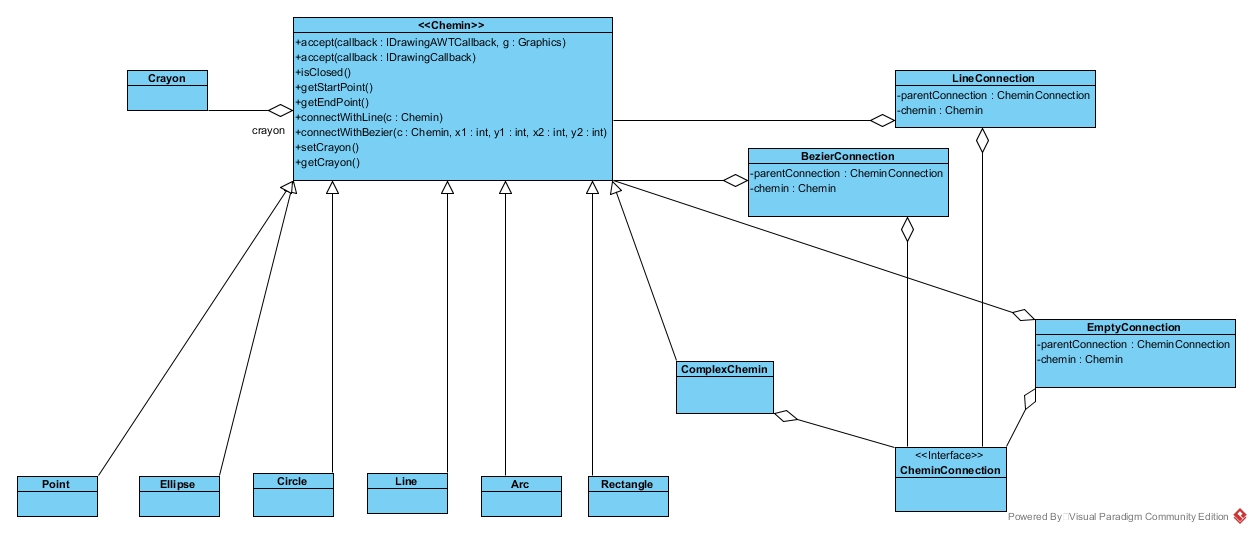
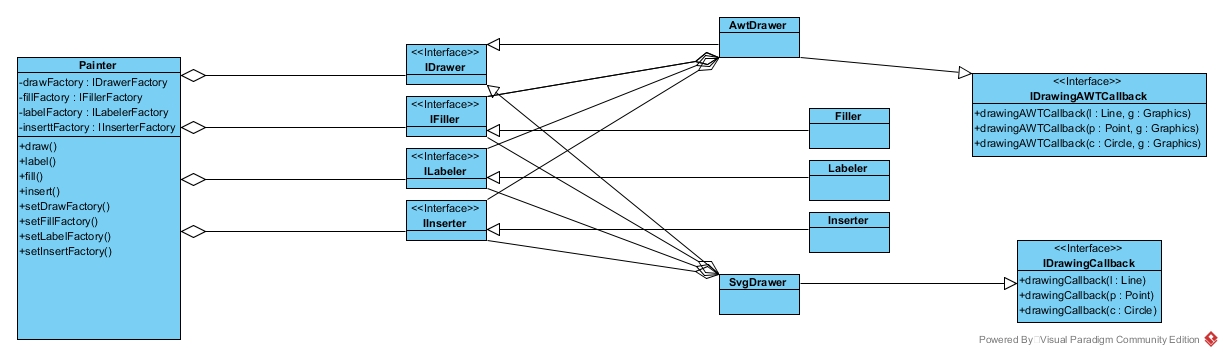
**if**(filler.contains(c)) {…}

En particulier ça signifie qu’on vérifie si dans la liste de toutes les figures qu’il faut remplir, il y a notre cercle *c*, et s’il est là, on récupère une information de son remplissage et on dessine un cercle rempli. Analogiquement pour Inserter.

Pour dessiner des figures on sert d’une approche qui est basée sur le patron de conception Visiteur.

Toutes les figures qu’on peut dessiner sous-classent Chemin (qui lui-même est une classe abstraite). On a défini tels types de figures comme Point, Line, Rectangle, Circle, Arc, Ellipse et ComplexChemin.

Quand un client fait un appel de dessiner une figure, elle s’ajoute dans la liste des figures à dessiner List<Chemin> de Drawer (que ce soit AwtDrawer ou SvgDrawer). Mais pour dessiner une figure on a besoin de l’information spécifique à un type concret. Alors que comme on a dit déjà Drawer a une liste de Chemin qui est abstraite. On pourrait définir un type concret de la figure à l’aide d’une chaine des opérateurs «instance of» ou faire d’autres vérifications similaires pour finalement caster Chemin à un type concret et obtenir des paramètres nécessaires pour dessiner (coordonnées du centre et un radius dans le cas d’un cercle ou quatre points dans le cas d’un rectangle etc). Mais cette solution manque d’élégance. Il est beaucoup mieux d’utiliser une approche qui est basée sur le patron Visiteur comme nous avons fait. Donc on a défini une méthode abstraite accept(…) dans Chemin en obligeant de telle manière chaque classe qui hérite de lui d’implémenter cette méthode.



En fait on va avoir deux versions surchargées de accept() :

void accept(IDrawingCallback callback)

et

void accept(IDrawingAWTCallback callback, Graphics g)

Alors chaque classe qui sous-classe Chemin devra avoir une réalisation comme ça :

**void accept(IDrawingCallback callback) {**

**callback.drawingAWTCallback(this);**

**}**

**void accept(IDrawingAWTCallback callback, Graphics g) {**

**callback.drawingAWTCallback(this, g);**

**}**

IDrawingCallback c’est une interface. Elle a telles méthodes:

**public** **void** drawingCallback(Line l);

**public** **void** drawingCallback(Circle c);

**public** **void** drawingCallback(Point p);

**public** **void** drawingCallback(Ellipse e);

**public** **void** drawingCallback(Arc l);

**public** **void** drawingCallback(Rectangle r);

**public** **void** drawingCallback(ComplexChemin l);

IDrawingAWTCallback c’est aussi une interface. Elle a des méthodes analogiques :

**public** **void** drawingAWTCallback(Line l, Graphics g);

**public** **void** drawingAWTCallback(Circle c, Graphics g);

**public** **void** drawingAWTCallback(Point p, Graphics g);

**public** **void** drawingAWTCallback(Ellipse e, Graphics g);

**public** **void** drawingAWTCallback(Arc l, Graphics g);

**public** **void** drawingAWTCallback(Rectangle r, Graphics g);

**public** **void** drawingAWTCallback(ComplexChemin l, Graphics g);

AwtDrawer implémente IDrawingAWTCallback. Alors pour dessiner la liste des figures on fait comme ça :

**for**(Chemin chemin : chemins) {

chemin.accept(**this**, g);

}

Cela signifie que si chemin est une instance de la classe Point, on aura un appel à une fonction

**public** **void** drawingAWTCallback(Point p, Graphics g){…}

et déjà là une peinture d’un point sera effectuée.

Analogiquement SvgDrawer implémente IDrawingCallback. Alors là on dessine pareil :

**for**(Chemin chemin : chemins) {

chemin.accept(**this**);

}

On pourrait poser une question : pourquoi les fonctions de IDrawingAWTCallback exigent-ils un paramètre supplémentaire Graphics g? En fait la peinture en awt est un peu spécifique. On est obligé de peintre dans une méthode spéciale **paint(Graphics g)** qui vient de JPanel et qui est exécutée dans un thread de dessin à part. Une variable Graphics g existe seulement dans cette fonction et ne doit pas être appelée (dans le cas si on a essayé de la sauvegarder dans une variable d’instance pour une utilisation ultérieure par exemple) en dehors de la portée de paint**.**

Et dans SvgDrawer une peinture se passe sans telles contraintes.

Une autre question qu’on pourrait se poser maintenant c’est pourquoi on a introduit une interface IDrawingCallback alors qu’on pouvait se servir toujours des méthodes de IDrawingAWTCallback et envoyer null pour Graphics dans le cas de SvgDrawer? J’ai décidé que telle solution manquait de clarté. Également je trouve que si on ajoute d’autres interprétations des dessins vectoriels, c’est plutôt IDrawingCallback interface qui sera utilisée pour dessiner à la plupart des cas.

Une dernière chose que je voudrais mentionner dans ce rapport c’est une classe ComplexChemin qui hérite de Chemin. Cette classe sert à la représentation des figures qui sont jointes par des lignes ou de courbes Bézier. ComplexChemin contient des connections entres les autres figures. En fait Chemin et ces enfants représentent une variation du patron Composite : Chemin c’est une classe abstraite, Cercle, Rectangle etc sont des feuilles et ComplexChemin est un composite qui a une liste (qui a une forme spéciale de CheminConnection) des Chemin-s. Une interface CheminConnection est implémenteé par classes : EmptyConnection, LineConnection et BézierConnection. EmptyConnection c’est le cas d’une figure sans connexions, donc sa connexion parentale est null et elle a une référence sur Chemin. LineConnection c’est une connexion par ligne, BézierConnection c’est une connexion par courbe Bézier. Chaque connexion a une référence à une connexion précédente (ou on peut dire parentale) et une référence à un chemin qu’on veut joindre.