# **Mini Projet Dessin Vectoriel**

# Introduction

Le projet sur le dessin vectoriel a pour objectif de définir un langage de dessin vectoriel à partir du langage de programmation Java. Pour ce faire, on se donne trois phases essentielles :

* Définition du langage
* Définition de l’interprétation du langage
* Modularité de ces définitions

L’utilisateur de ce langage doit pouvoir s’affranchir le plus possible du langage de programmation Java au profit de notre langage. Ce qui suppose alors un ensemble riche de types, de structures, de moyens définis dans notre langage et pouvant servir à l’utilisateur. Le script qu’il écrira pour représenter un dessin devrait s’auto-suffire de notre langage (d’où la nécessité d’avoir des moyens comme une alternative (IF THEN ELSE), une boucle (FOR)…

# Le langage

Le langage a été défini en prenant en compte cet aspect modulaire du projet. Nous devions avoir dans notre projet des classes représentant un ensemble de « formes » géométriques prédéfinies que l’utilisateur pourra utiliser en paramétrant des arguments. Par exemple, pour une forme géométrique simple comme une droite ou un rectangle, l’utilisateur utilisera notre implémentation en fournissant deux points pour la droite par exemple.

Ces formes pourraient néanmoins être regroupées dans une entité abstraite et définissant globalement certains aspects et propriétés qu’elles ont en commun. Cette représentation pourrait être très efficace à long terme quand il s’agira de faire évoluer le langage et à court terme pour éviter de la duplication de code. D’où l’aspect modulaire de ce projet encore justifié.

Ci-dessous, les formes disponibles pour le dessin vectoriel (pour chaque forme, les préférences de style sont disponibles. Exemple : couleur du trait, épaisseur du trait, opacité…) :

* **Ligne** : traçant une droite entre deux points donnés en arguments dans l’instruction de l’utilisateur
* **Cercle** : représentant un cercle avec le centre et le rayon donnés en arguments dans l’instruction de l’utilisateur
* **Polygone**: dessinant un polygone avec le nombre de points et les coordonnées des points donnés par l’utilisateur
* **Rectangle**: traçant un rectangle avec un point (le point supérieur gauche), la longueur et la largeur donnés par l’utilisateur

Le langage possède également un certain nombre d’opérateurs :

* **For :** permet de répéter les instructions d’un script passé en argument un nombre de fois choisi par l’utilisateur
* **Alternative :** permet d’exécuter un script (ou une simple instruction) choisi si une condition définie est vraie et d’exécuter un autre script (ou instruction) si cette condition est fausse
* **Sequence :** permet d’exécuter une suite d’instructions dans l’ordre voulu. Il suffit d’ajouter les instructions à la séquence avec la méthode addToSequence() dans l’ordre voulu.

Pour finir le langage permet des manipulations de dessin spécifique :

* **Remplir :** permet de remplir un chemin fermé passé en argument d’une couleur passée en argument sous forme d’une String (ex : « red »)
* **Etiqueter :** permet d’afficher une étiquette sur le dessin avec le texte de son choix et à la position que l’on souhaite

Toutes ces classes sont des classes filles de **Instruction** et elles implémentent toutes une méthode execute() qui applique un traitement particulier pour chacune d’elle mais qui peut être appelée indifféremment par chacune de ces classes.

Cette méthode execute() est appelée par un **Script** pour tracer un dessin, un **Script** étant une suite d’instructions et de scripts éventuellement.

# Architecture

## Schéma général et Modularité

La classe en tête de la hiérarchie est Instruction et contient une méthode execute(). La plupart des classes sont des filles d’Instruction et implémentent la méthode execute() permettant un traitement spécifique pour chaque classe. Ainsi un script étant une liste d’instruction on peut simplement boucler sur cette liste et exécuter chaque instruction avec la méthode execute(). Ainsi quand on exécute l’instruction créerCercle cela va créer une instance de Cercle dans notre dessin et quand on exécute l’instruction Alternative le traitement de execute() est d’appeler execute() sur une instruction 1 si une condition est vraie ou d’appeler execute() sur une instruction 2 si la condition est fausse.

Un dessin est muni d’une liste contenant les différents chemins qui y sont tracés. Cependant, un dessin pouvant en contenir un autre mais un dessin n’étant pas un chemin, on a créé une interface étiquette Objet qu’implémentent Chemin et Dessin. Ainsi un dessin est une liste d’objets donc de chemins et d’autres dessins.

CheminFerme étend Chemin et contient la méthode setCouleurRemplissage(couleur). Toutes les formes fermées comme Cercle, Rectangle ou Polygone étendent CheminFerme et implémentent setCouleurRemplissage(couleur). Ceci est particulièrement pratique pour l’opérateur Remplir qui ne peut être utilisé que sur des chemins fermés. Ainsi si l’on souhaite étendre le langage avec une nouvelle forme fermées comme Triangle par exemple, il suffit d’étendre CheminFerme, implémenter setCouleurRemplissage(couleur) et execute(). Il n’est pas nécessaire de modifier l’implémentation de Remplir pour que cela fonctionne avec le nouvel objet Triangle. Il ne faut pas oublier cependant de coder une interprétation pour le Triangle.

## Patron de conception

Un patron de conception était alors nécessaire. Devant la multitude de classes faisant état de « formes géométriques », une structure adaptée était nécessaire. Structure d’autant plus justifiée que l’écriture de scripts par l’utilisateur et l’interprétation du langage devraient être affranchies le plus possible du langage Java.

On a donc jugé opportun d’utiliser un patron composite pour l’architecture de notre projet. Une première raison pourrait être que la similitude de plusieurs de nos classes représentant des formes au sens des méthodes implémentées, du paramétrage et de l’utilisation pourrait faciliter l’interprétation du langage. Plus concrètement, on aurait comme dans toute structure composite :

* Un **composant** qui sera ici notre objet permettant de faire l’abstraction pour tous les composants. Ce serait donc une classe nommée « Chemin » dans notre projet qui fixe l’utilisation et certaines propriétés (comme *dessiner*) pour toutes les formes. Techniquement, on pourrait le représenter comme une interface ou encore comme une classe abstraite.
* La **feuille** qui ici n’a pas trop d’intérêt puisque notre hiérarchie n’est pas très riche. Mais on pourrait s’imaginer, dans l’enrichissement du langage, des formes géométriques étant des sous-éléments d’autres formes géométriques. Par exemple, un arc de cercle comme sous élément d’un cercle
* Le **composite** qui ici sera représenté par toutes nos classes étendant « Chemin ». Les comportements spécifiques d’une forme géométrique particulière seront définis dans ces classes.

## Extension du langage

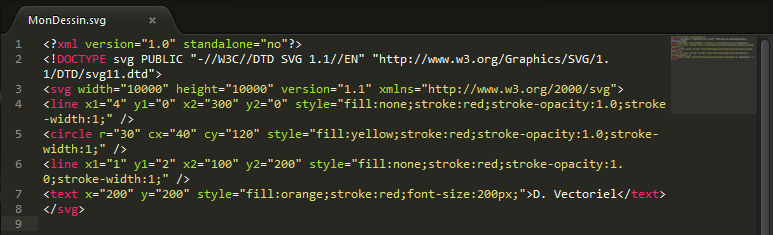
Pour étendre le langage on peut ajouter de nouvelles formes au langage. Comme expliqué précédemment

# Interprétation

La définition d’un dessin par l’utilisateur se fait par l’écriture d’un script. Dans notre modélisation, un script est un ensemble d’instructions représentant ainsi un dessin. Ces instructions peuvent aller d’une simple création de forme au dessin de la forme, mais aussi de commandes plus complexes comme l’écriture d’une boucle **For** ou d’un **IF THEN ELSE**.

Ces instructions sont ensuite regroupées dans l’objet Dessin qui se charger de les exécuter. Mais cette exécution passe par une interprétation, c’est-à-dire une redéfinition de ces instructions par de commandes format svg. L’interpréteur est alors très important dans ce processus de dessin vectoriel, c’est en fait lui qui fait que notre langage soit compréhensible et traductible en une suite de commandes logiques pouvant désormais être tracées et visionnées par n’importe quel logiciel pouvant lire le format svg, typiquement les navigateurs web.

En ce qui nous concerne, l’interprétation du langage se fait par une reconnaissance des instructions. Lorsque l’utilisateur aura écrit et validé son script pour exécution, on va parcourir l’ensemble du script, instruction par instruction pour les traiter. Pour chaque instruction, l’interpréteur regardera de quelle forme il s’agit et récupère l’ensemble des arguments donnés par l’utilisateur. Une fois la forme déterminée, il procédera alors au « tracé » de cette forme avec les commandes svg adéquates en fournissant les données de l’utilisateur. Un exemple de résultat issu de l’interprétation de script est donné ci-dessous :



Pour les alternatives et les boucles qui constituent également des instructions, l’interprétation est la même. Une fois la commande FOR ou IF reconnue, les instructions contenues dans ces commandes seront selon le répétées, testées selon un critère ou ignorées.

En définitive, la fin de l’exécution du script donnera lieu à un fichier au format .svg contenant toutes les instructions du dessin. Vient alors la deuxième phase de l’interprétation qui est l’exécution du fichier svg pour avoir un rendu graphique du dessin. La visualisation de ce fichier peut se faire avec l’aide de navigateurs comme **Firefox** ou **IE**. L’exemple de la visualisation du dessin correspondant aux commandes svg ci-dessus issues de l’interprétation du script de l’utilisateur est donné ci-dessous :

