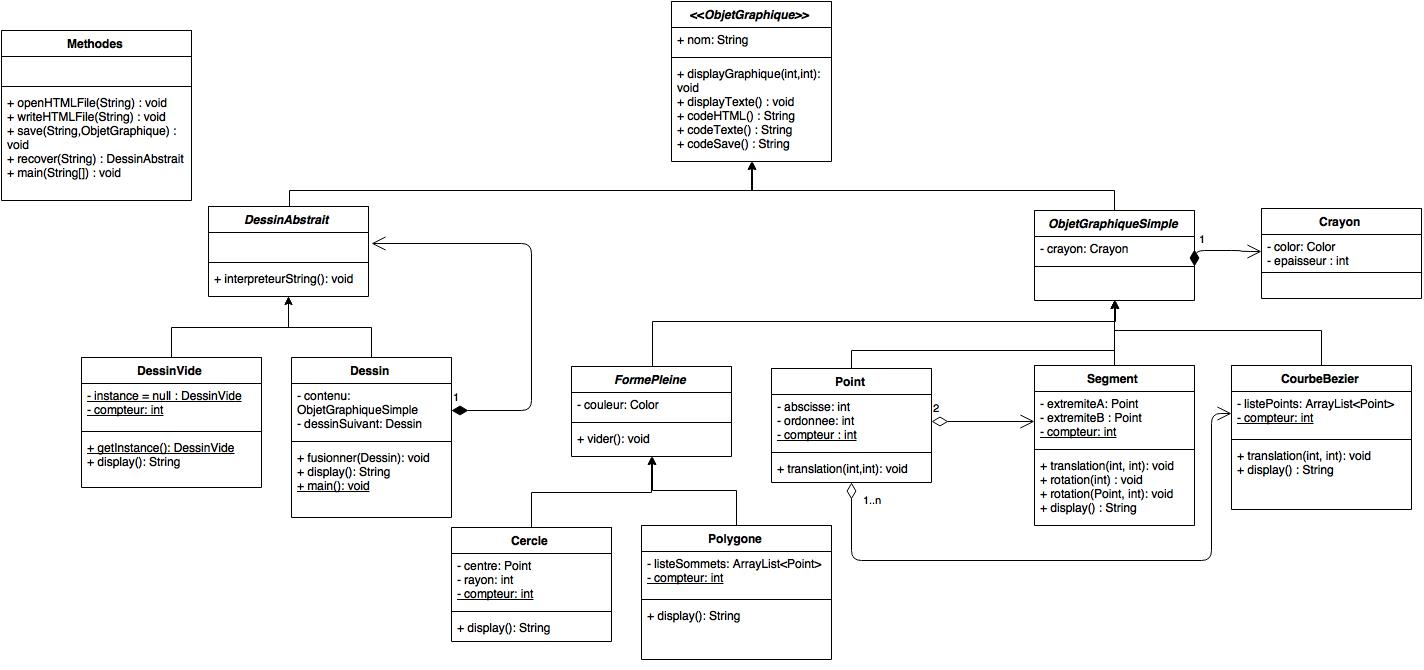
**Rapport de présentation de l’application de dessin vectoriel**

Ce document présente la stratégie mise en œuvre pour l’implémentation de l’application de dessin vectoriel, ainsi que les différents patrons de conception utilisés. Il contiendra également une description des différentes classes implémentées et des patrons de conception employés.

Membres du groupe : Nicolas Boulanger – David Coronel – Marc Lamazouère**I – Overview de l’application en UML**



Ce schéma UML représente l’architecture de l’application de dessin vectoriel.

Le programme permet d’afficher, à travers une console texte ou encore de manière graphique, des figures géométriques variées. Il contient également une fonction de sauvegarde qui permet d’enregistrer un dessin, et de l’éditer ultérieurement.

Nous sommes partis d’une interface abstraite, nommée ObjetGraphique. Toutes les classes du programme implémenteront cette interface. Cette interface possède la déclaration des méthodes d’affichage (displayGraphique() et displayTexte()), mais aussi trois autres méthodes, codeHTML(), codeTexte() et codeSave(). Ces trois dernières méthodes servent à écrire le code correspondant à la tache désirée. Par exemple, codeHTML() va renvoyer une String, qui est un code HTML pour afficher l’objet désigné.

Plus généralement, nous avons choisi de décrire un dessin comme étant une succession d’objets graphiques simples, comme des points, des segments, des courbes ou encore des polygones. Tous ces éléments ont donc été réunis au sein de la classe abstraite ObjetGraphiqueSimple.

Faisons un rapide commentaire sur la classe Crayon. Elle comporte deux variables d’instance, qui sont color et épaisseur. Ces deux variables correspondent respectivement à la couleur et à l’épaisseur du trait dans le cas où l’on voudrait réaliser un affichage graphique des dessins. Ainsi, toute instance d’objet graphique simple doit avoir une couleur et une épaisseur.

Nous avons ensuite choisi de distinguer les éléments constitutifs d’un dessin de deux sous-catégories : les formes pleines, et les autres. Les formes pleines sont (pour le moment) les cercles et les polygones. En effet, le cercle étant un cas particulier, nous avons choisi de le traiter séparément des polygones classiques.

Les points, segments et courbes ont été identifiés comme étant les éléments atomiques composant tout objet graphique simple. C’est pourquoi pour construire des cercles ou des polygones, il est nécessaire d’avoir créé des points au préalable (cf les variables d’instances des classes Cercle et Polygone).

Nous avons essayé de donner des noms explicites aux variables afin de faciliter la compréhension de l’utilisateur. Cependant, certaines d’entre elles méritent tout de même une explication supplémentaire, comme la variable statique compteur, ou encore la présence d’une deuxième couleur au sein de la classe FormePleine.

En ce qui concerne la variable statique compteur, celle-ci sert de repère pour savoir combien il y a d’instances de chaque élément dans un dessin. Lors de l’appel d’un constructeur pour point, on incrémente la valeur du compteur de un. Ainsi, on est capable de savoir combien un dessin comporte de points. C’est exactement la même chose pour les segments, les courbes, etc.

Pour la redondance de la variable couleur : la présence de deux variables couleur est obligatoire dans le cas de formes pleines. Les formes pleines ont la particularité de pouvoir être coloriées, contrairement aux points ou aux segments. Il faut donc une couleur de tracé (comprise dans la classe Crayon) et une couleur de remplissage dans le cas d’un cercle ou d’un polygone.

**II – Mise en exergue des patrons utilisés**

Nous avons utilisé deux patrons de conception différents : le patron singleton et le patron composite. Ces deux patrons sont visibles depuis le schéma UML du projet, et se situent tous deux dans la partie gauche du diagramme.

La classe DessinVide compose à elle seule le patron singleton. Nous l’avons utilisé car nous n’avions besoin que d’un seul objet pour modéliser un dessin vide. Il est inutile d’avoir plusieurs instances de dessin vide, car par définition, le dessin vide est unique (il est vide). Si le dessin vide existe déjà, la méthode getInstance() de la classe DessinVide renvoie la seule instance créée de DessinVide. Autrement dit, si DessinVide n’existe pas, on crée en crée une instance, et s’il existe déjà, on revoie l’instance de DessinVide déjà existante. C’est ici que réside la force du patron singleton. Le système n’a pas besoin d’instances supplémentaires de dessins vides ; c’est pourquoi le choix d’un tel patron dans l’implémentation de notre programme nous a paru indispensable.

Le second patron utilisé est le patron composite. Il permet ici de gérer la succession de dessins de manière très simple et récursive. Nous avons défini un dessin comme ayant un contenu de type ObjetGraphiqueSimple et un autre dessin, appelé dessinSuivant. Pour réaliser un dessin complet, il suffit de chaîner les instances de dessin, et de terminer la chaîne par le dessin vide. Pour compléter un dessin déjà existant, il suffit de parcourir la chaîne, et dès qu’on aperçoit DessinVide, on sait que l’on est inéluctablement arrivé à la fin de la chaîne.

Par ailleurs, l’utilisation du patron composite facilite la clarté et la lisibilité du code, en plus de réduire le nombre de calculs effectués par la machine.