Membres : Schaffo Raphaël, Ryter Nils, Vulliemin Kevin

Equipe : Numéro 6

Destinataire : Monsieur Gobron Stéphane

# Introduction

## Problématique

Dans le cadre du Design Pattern, nous devons développer un programme permettant à l’utilisateur de dessiner une fractal. Ce programme devra contenir dans son implémentation, le pattern de singleton et le pattern composite. Nous avons implémenté la fractal de Cesaro. Pour promouvoir la représentation graphique de la fractale, nous avons décidé d’utiliser Qt Creator.

Qu’est-ce qu’une fractale ? Une fractale est une courbe ou une surface ayant une forme irrégulière ou morcelé. Elle se créer en suivant des règles déterminées. Sur l’image ci-dessous vous pouvez voir la fractale de cesaro. Elle se créer en ajoutant sur chaque ligne un triangle. Sur la première image de gauche, on voit un carré formé de ces 4 lignes sur lesquelles on a ajouté un triangle sur chaque ligne. On applique cette règle sur la nouvelle forme obtenue et ainsi de suite. Après avoir appliqué 3 fois la règle sur la première forme, on obtient la forme située toute à droite de la figure 1.

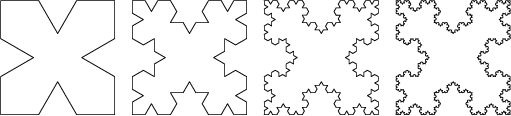


Figure . Évolution *de* la fractal de Cesaro

# Conception

## Singleton

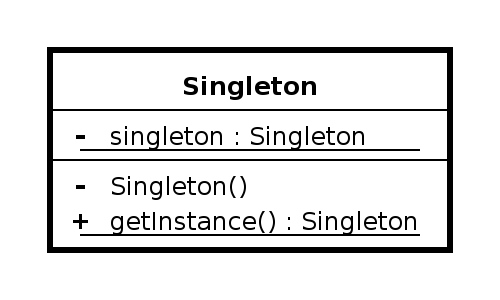
Le singleton est un pattern ayant pour but de limiter, restreindre le nombre d’instanciation d’un objet. Pour implémenter celui-ci il suffit de créer une classe qui va contenir une méthode qui elle, créer une instance d’un objet uniquement si il n’y en a pas encore. S’il y a déjà un objet instancié, la classe renvoie une référence vers cet objet. Le singleton est à manier avec précaution si on l’utilise avec des Threads. Dans le cas où aucune instanciation d’objets n’a encore été faite et que deux threads exécutent la méthode de création il faut bien veiller à ce que seulement un des threads instancie un nouvel objet.

Figure 2. Diagramme du singleton

## Composite

Un composite (un objet composite), est un pattern structurel. Il permet de concevoir une structure d’arbre en limitant à deux le nombre de sous éléments. L’idée de ce pattern est enfaite de pouvoir manipuler un groupe d’objet comme si il s’agissait d’un seul objet. Pour cela il faut donc que les objets regroupés aient des opérations communes. Le pattern de composite peut être décomposé en 3 parties (voir le schéma ci-dessous).

* ***Le Composant :*** Il représente tous les composants, y compris ceux qui sont composés. Il déclare l’interface pour le comportement par défaut.
* ***La Feuille****:* Elle représente un composant qui ne possède pas de sous-éléments. Les feuilles implémentent le comportement par défaut.
* ***Le Composite :*** Il représente un composant qui peut avoir des sous-éléments. Il stocke les composants enfant et permet l’accès à ceux-ci. Il implémente le comportement en utilisant son ou ses enfants.

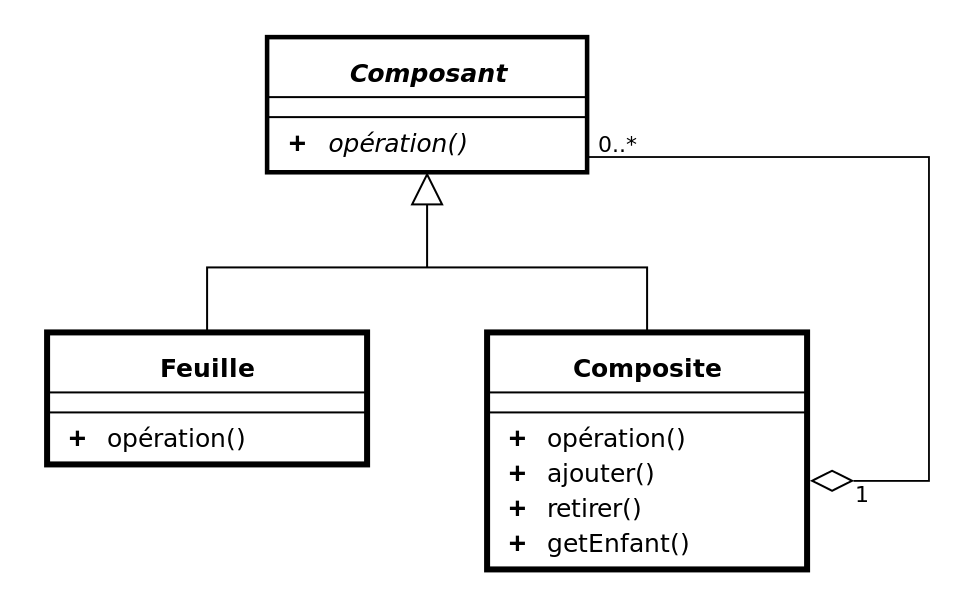
Pour manipuler des objets, l’utilisateur, aussi dit client dans ce cas va utiliser la classe composant.

Figure 3. Diagramme du composite

# Implémentation

## Séparation

Afin de rendre le code plus lisible et compréhensible, le programme a été séparés en package ressemblant à une structure MVC :

* Contrôleur : La partie *algorithme* contient la classe Fractal qui détient les méthodes qui vont effectuer l’algorithme de « Fractalisation ». C’est donc elle qui va pleinement utiliser la structure composite en s’occupant de :
  1. Créer le polygone avec les arguments insérés par l’utilisateur ;
  2. Transformer les lignes du polygone en composite et y inséré les nouvelles lignes selon le fractal choisi.
* Vues : La partie *interfaces* contient les fenêtres et objets destinés à une interface graphique utilisateur afin que celui-ci puisse saisir ces paramètres ainsi que visualiser le résultat ;
* Modèles : La partie *component* contient la hiérarchie des classes du design pattern « Composite ».

## Singleton

On utilisera le singleton dans notre projet afin de garantir l’exclusivité de notre objet mère qui est la classe Fractal. Ce programme ne profite pas pleinement de l’utilité du design pattern « Singleton » mais il a été réalisé dans un but didactique.

## Composite

Après avoir identifié la nature de la fonction voulue, dans notre cas « dessiner », on peut définir la fonction de la classe composant.

1. virtual void *draw*(QPainter &gc) = 0;

Cette méthode sera implémenté dans les classes enfants feuille et composite. Lors de l’appel ce celle-ci, pour chaque composite, on cherchera alors à atteindre tous ses enfants afin de pouvoir dessiner les lignes finales. On va donc itérer sur les enfants tant qu’on ne trouve pas de lignes :

1. foreach(Component \*c, listChildren)
2. {
3. c->*draw*(gc);
4. }

Quand on finira par tomber sur une ligne, on va tout simplement la dessiner :

1. gc.drawLine(\_p1, \_p2);

Cette structure témoigne également d’une sorte de récursivité (appel de la fonction jusqu’à l’ancrage).

## Fractalisation

(Nils : parler de la méthode des matrices)

# Conclusion

Le programme fournit des résultats très satisfaisants. On peut générer rapidement divers formes avec l’algorithme de Cesaro et l’interface graphique permet de visualiser la forme finale produite par un nombre spécifié de fractalisation. L’implémentation du pattern composite était pleinement justifier dans notre programme et a été très utile dans les étapes de transformation.

