/*----------
Design Patterns
Labo 02
Equipe 05 : Loïc Frossard, Valentino Izzo, Luca Meyer, Léon Muller
----------*/

1. Structure source

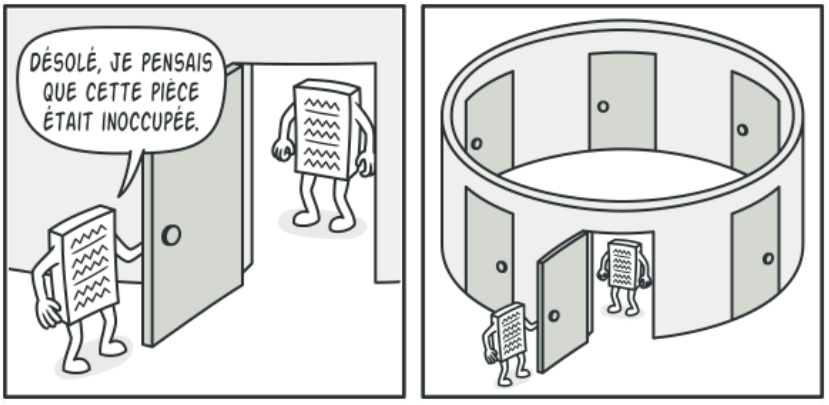
|- Rapport
| Singleton
|- fractal.js
Créer les fractales
|- index.html
Afficher les fractales et en choisir les paramètres
| Composite
|- fractal.js
Créer les fractales
|- index.html
Afficher les fractales et en choisir les paramètres

2. Améliorations possibles

Une amélioration aurait été de mettre plus de temps dans l'infographie d'essayer d'obtenir des arbres visuellement plus esthétiques, courbés, et que l'on voit moins la séparation entre chaque barre par exemple.
Nous aurions aussi pu permettre à l'utilisateur de choisir la forme de la fractale qu'il souhaitait réaliser, en stockant par exemple dans un enum les caractéristiques de chacune des fractales proposées.
Une dernière amélioration aurait pu être de réaliser une combinaison de plusieurs fractales (aléatoirement choisies par exemple) pour créer une image unique.

3. Le DP

a. Singleton



Le but du singleton est de limiter le nombre d'instances d'un objet à une seule. Il permet d'utiliser moins de place mémoire et est aussi très efficace, quand le système est très rapide.

Le principe de ce design pattern est d'avoir une méthode "getInstance()" qui crée une unique instance, si celle-ci n'existe pas encore. Si elle a déjà été créée, la méthode va simplement renvoyer une référence à cette instance.
Le constructeur de la classe doit être en privé, car on ne veut pas créer d'autres instances depuis l'extérieur. Si on veut utiliser cet objet, il faut passer par la méthode "getInstance()". Dans cet objet, une variable privée et statique contient donc cette unique instance.

i. Faiblesses

Le problème principal du pattern est qu'il s'apparente à l'utilisation de variables globales, car une seule instance est partagée à travers la totalité du code, et c'est en général quelque chose que l'on souhaite éviter.

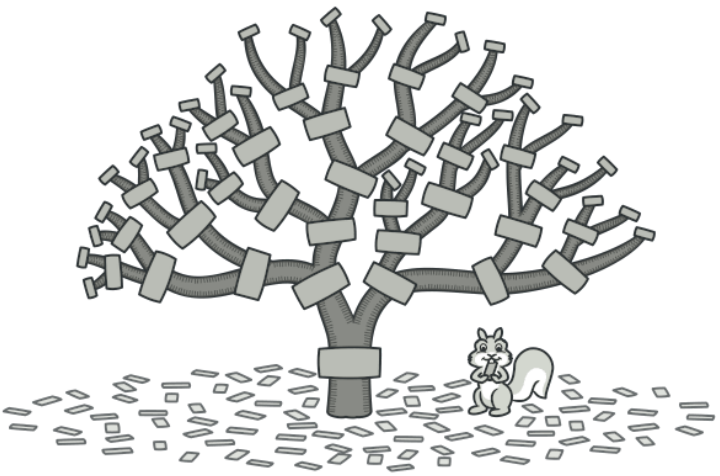
De plus, le pattern n'est pas adapté à une utilisation dans un environnement multithread. Même si l'on peut facilement protéger l'accès aux attributs de la classe avec des mutex (et autres outils), le fait qu'une seule instance soit partagée entre tous les threads ralentira tous les processus, car ceux-ci font fréquemment appel à l'instance unique du singleton.

Un autre inconvénient se retrouve dans la structuration du code : le singleton exécute l'intégralité du code et il n'y a plus de responsabilité unique (on aurait par analogie un "main" conséquent qui résout tous les problèmes).

ii. Forces

Garantit l'utilisation d'une seule instance là où elle est nécessaire.
Par exemple, on ne devrait jamais instancier deux fois une barre de menus dans un programme fenêtré, ou encore interfacer un driver/API, qui lui n'est chargé qu'une seule fois par le système, il est donc possible directement dans le code d'accéder toujours à ce driver/API via une même instance (et potentiellement optimiser les accès si c'est plus rapide à faire en batch que par plein de petits appels). Un contexte OpenGL par exemple, ne devrait être créé qu'une seule fois pour une fenêtre, et stocker les appels à la carte graphique pour ensuite tous les envoyer d'un seul coup sera plus rapide que de constamment communiquer avec elle.

b. Composite

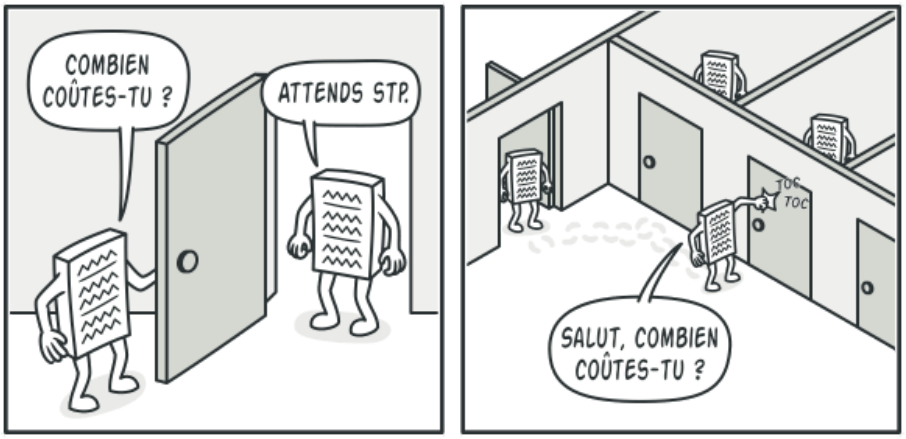


Le but du composite est d'avoir une structure en forme d'arbre : chaque instance est représentée par une feuille/branche qui stocke les instances suivantes, que l'on appelle "enfants". On pourrait prendre comme exemple les dossiers et sous-dossiers dans un ordinateur.
Chaque feuille est donc maîtresse d'elle-même et effectue toutes les opérations (d'affichage par exemple) sur elle-même, avant de dire à ses enfants de faire de même (dans le cas d'un appel récursif).

i. Faiblesse

Difficile d'accéder directement à une instance en particulier : on va pour ce faire devoir implémenter un algorithme de tri pour retrouver l'instance que l'on veut en partant de la racine. De plus, s'il y a trop de différence entre les membres de l'arbre (cas particuliers), l'algorithme deviendra vite complexe.

ii. Forces



Manipulation d'une même manière toutes les instances de la classe.
On peut lancer une opération sur la racine du composite et la faire se répercuter récursivement sur chacun des enfants jusqu'aux feuilles. Ce pattern est utilisé lorsque l'on agit sur plusieurs objets de la même façon et que l'on se retrouve souvent avec les mêmes implémentations pour chacun d'entre eux.

On peut aussi considérer seulement un petit arbre dans l'arbre total, et donc effectuer des opérations sur des parties distinctes qui composent la structure générale.

4. Conclusion

Au travers de ce laboratoire nous avons pu voir l'utilisation des design pattern "Composite" et "Singleton".
Ces deux patterns permettent de gérer des projets ayant un grand nombre d'objets tout en gardant une généricité dans le code et une implémentation courte. Ils permettent de passer par la récursion ou l'itération pour pouvoir réaliser les mêmes opérations en chaîne.

Images :

https://refactoring.guru/images/patterns/content/composite/composite.png?id=73bcf0d94db360b636cd
https://refactoring.guru/images/patterns/content/composite/composite-comic-1-fr.png?id=b318eb1564d5ce4f75a6
https://refactoring.guru/images/patterns/content/singleton/singleton-comic-1-fr.png?id=792833a40401e6e6112b