LOG2410 : Conception logicielle

Conception à base de patrons I

TP4

Présenté par :

Stéphanie Mansour (1935595)

Nanor Janjikian (1901777)

Groupe : B1

Hiver 2019

Patron Composite

1. Patron Composite :
    - **Intention** du patron composite :

Le patron composite porte sur la **structure du design**. Il permet de donner une uniformité au traitement d'éléments composites et de concevoir une **structure d'arbre**. En utilisant une structure d'arbre, des structures composites seront permises et traités de la même manière que les structures non-composites, soit des feuilles. De plus, une structure d'arbre **simplifie** l'utilisation et la compréhension des structures d'un utilisateur. Celui-ci est utilisé pour **manipuler** un groupe d'objets (qui possède des opérations communes) de la même façon que s'il s'agissait d'un seul objet.

- **Structure** des classes réelles participant au patron composite :

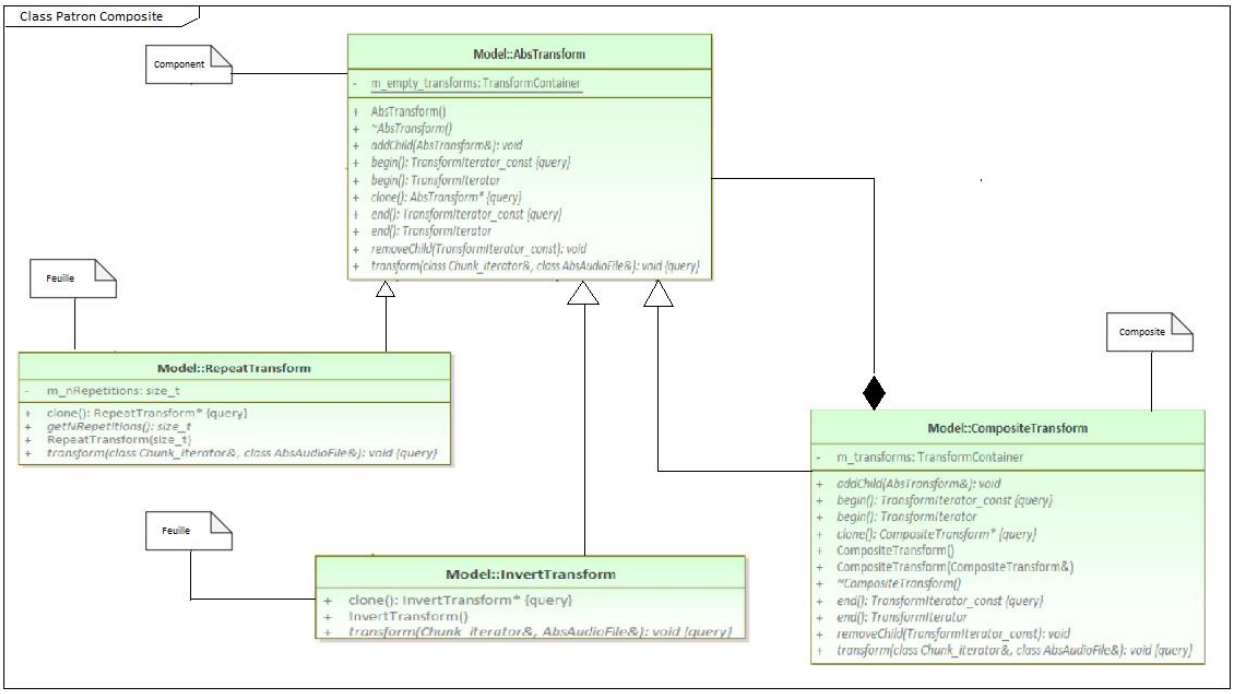

Figure 1. Diagramme de classes du patron Composite

2. Les **abstractions** présentent dans la conception du TP4 et leurs **responsabilités** :

- **AbsAudioFile.h** : représente une **interface** pour AudioFile et MemAudioFile. C'est à travers d'elle que l'utilisateur peut manipuler les méthodes utilisées par les objets, composites ou non. Notre fichier memAudioFile est utilisé comme proxy.

- **AbsTransform.h** : représente une **interface** d'utilisation afin de manipuler et appliquer une ou plusieurs transformations sur les parties audio appelées chunk de notre fichier de sortie de type AbsAudioFile.

3. Dans l'implémentation actuelle du système PolyVersion, quel objet ou classe est responsable de la création de l'arbre des composantes.

   La classe responsable de la création de l'arbre des composantes est la classe **CompositeTransform**. En effet, nous observons que cette classe a des méthodes d'ajout et de retrait de feuilles/nœuds, soient « **addChild** » et « **removeChild** ». Donc, cette classe permet la manipulation de l'arbre.

Patron Proxy

1. Patron Proxy :
   - **Intention** du patron proxy :

Le patron proxy joue le rôle **d'intermédiaire** afin de contrôler **l'accès** a l'objet en question. Ce patron va **dupliquer** la classe « sujet » pour appliquer les modifications voulues **sans changer** l'objet original. Bref, ce patron simplifie une classe complexe et contrôle les accès à une classe.

- **Structure** des classes réelles participant au patron proxy :

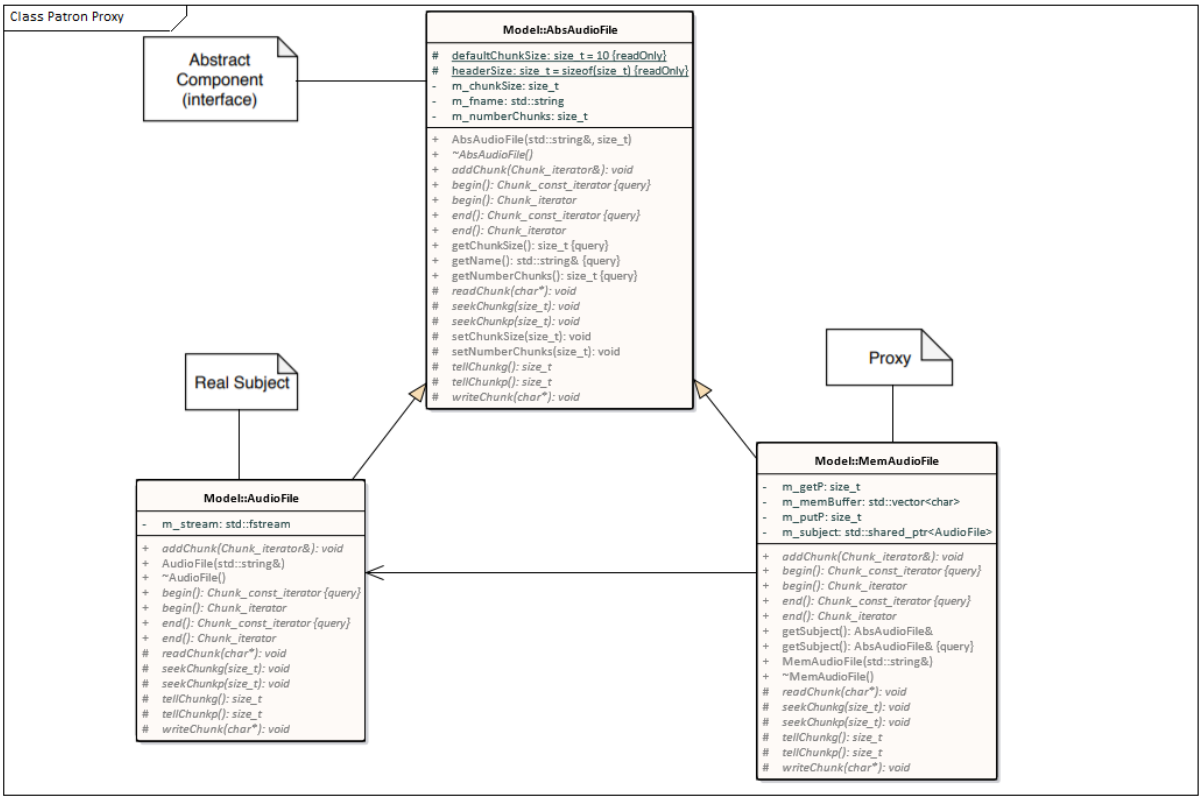

Figure 2. Diagramme de classes du patron Proxy

Patron Itérateur

1. Patron Itérateur :
    - **Intention** du patron itérateur :

Le patron itérateur permet de garder l'état d'itération sans exposer sa structure interne. Alors, il **masque** le comportement interne et fournit une **abstraction** sur les objets. De plus, il offre une méthode d'accès séquentielle aux éléments d'un objet agrégat que ce soit un vecteur, une liste, etc. Ceci **stabilise** les manipulations.

- **La classe de conteneur** de la STL utilisée pour **stocker** les enfants dans la classe Composite et les classes des *Iterators* utilisés dans la conception qui vous a été fournie :

La classe de conteneur de la STL utilisée pour stocker les enfants dans la classe Composite est celle de ***Vector***. De plus, ces classes *iterators* nous ont été fournies : « **TransformBaseIterator** », « **TransformBaseIterator_const** », « **TransformIterator** » et « **TransformIterator_const** ».

2. Expliquez le **rôle** de l'attribut statique **m_empty_transforms** défini dans la classe AbsTransform. Expliquez pourquoi, selon vous, cet attribut est déclaré comme un attribut statique et privé.

M_empty_transforms permet l'implémentation des méthodes begin(), begin(), end() et end(). Ces méthodes doivent être implémentées au sein de la classe « CompositeTransform » puisqu'elles sont présentes dans la classe abstraite AbsTransform et la classe CompositeTransform y dérive. De plus, l'attribut m_empty_transforms peut faire échouer silencieusement ces méthodes (comme mentionné dans les commentaires du code). Il va retourner un objet *Iterator* valide d'un conteneur vide. M_empty_transforms est déclaré **privé** puisqu'il serait inutile de permettre aux classes dérivées de AbsTransform d'hériter d'un attribut qui ne sert qu'à échouer les méthodes héritées de la classe CompositeTransform non nécessaires. Il est aussi déclaré **statique** puisque chaque transformation n'a pas besoin de conteneur et donc n'a pas besoin d'itérateur. De plus, pour n'en avoir qu'une seule copie qui sera utilisé par toutes les instances dérivées de AbsTransforms.

3. Quelles seraient les **conséquences** sur l'ensemble du code si vous décidiez de changer la classe de conteneur utilisée pour stocker les enfants dans la classe Composite? On vous demande de faire ce changement et d'indiquer toutes les modifications qui doivent être faites à l'ensemble du code à la suite du changement. Reliez la liste des changements à effectuer à la notion d'encapsulation mise de l'avant par la programmation orientée-objet. À votre avis, la conception proposée dans le TP4 respecte-t-elle le principe d'encapsulation ?

Si nous décidons de changer la classe de conteneur utilisée pour stocker les enfants dans la class Composite pour des conteneurs de types « list » et de type « deque », il n'y aurait pas de changement qui sera apporté sur le code, comme ces deux types de conteneurs utilisent les mêmes méthodes de notre code actuel. Toutefois, si des conteneurs de types « map », « set », « multimap » « multiset » et d'autres sont choisis, nous devons modifier le code actuel. Certaines méthodes vont être modifiées ou ajoutées pour permettre le fonctionnement de ces types de conteneurs et une bonne manipulation de nos données. De plus, une modification sur le type d'itérateurs utilisé pour parcourir les conteneurs sera nécessaire. A notre avis, ce code respecte le principe d'encapsulation comme les changements à apporter ne sont pas majeurs.

Un exemple de modification de code (en utilisant le consteneur de liste à la place de vector)

```
using TransformContainer = std::list<TransformPtr>;
```

4. Les classes dérivées TransformIterator et TransformIterator_const surchargent les opérateur « * » et « → ». Cette décision de conception a des avantages et des inconvénients. Identifiez un avantage et un inconvénient de cette décision.

   - **Avantage** de la surcharge des opérateurs « * » et « → » : **simplifier** l'accès à l'objet sur lequel pointe l'itérateur. La surcharge des opérateurs « * », « → » déréférence le pointeur et simplifier l'accès à l'objet (*) et aux méthodes (→) puisque le conteneur offre des pointeurs intelligents.

   - **Inconvénient** de la surcharge : confusion si les surcharges ne sont pas conçues de façon optimale et peuvent augmenter la probabilité de bogues.