

**LOG2410**

**CONCEPTION LOGICIELLE**

**TP4**

**Trimestre :** HIVER 2019

Antoine Martin (1929587)

Jaafar Kaoussarani (1932805)

**Présenté à :** François Guibault

École Polytechnique de Montréal

Le mardi 26 mars 2019

# 2 – Patron Composite

1. a) **L’intention du patron Composite** :

Le patron Composite est un patron de conception structurel sur les objets ayant comme objectif de les représenter selon une structure en arbre. Elle nous permet de traiter uniformément les objets de façon récursive. Dans notre cas, cela revient à exécuter une série de transformations d’inversement et de répétition en les stockant dans un objet composite. En d’autres termes, si on utilise le modèle de l’arbre, les sommets intérieurs sont les transformations composites et les deux autres sont les feuilles de ces derniers. Ainsi on peut traiter toutes les transformations de manière uniforme.

b) Voir Annexe 1

* AbsTransform : Classe interface des trois transformations possibles
* RepeatTransform : Transformation qui répète un Chunk n fois dans le fichier de sortie, où n est le nombre de répétitions.
* InvertTransform : Transformation qui inverse le Chunk dans le fichier de sortie
* CompositeTransform : Classe composite composée d’une série de RepeatTransform et de InvertTransform

1. **Identifiez toutes les abstractions**:

Une première abstraction est la classe *AbsTransform*. Cette dernière est la classe interface de nos trois transformations. Par la suite, nous avons la classe *AbsAudioFile,* qui est la classe interface des classes *AudioFile* et *MemAudioFile*.

1. **Quel objet ou classe est responsable de la création de l’arbre des composantes :**

Actuellement, la classe CompositeTransform est responsable de la création de l'arbre des composants. En effet, nous voyons que cette classe a une méthode qui s'appelle «addChild» et une autre qui s'appelle «removeChild». Ces méthodes sont, respectivement, responsables de l'ajout et du retrait des transformations dans l’arbre.

# 3 – Patron Proxy

1. a) **L’intention du patron Proxy**:

L’objectif de ce patron est de dissimuler la structure d’un objet ou d’une classe tout en laissant la possibilité de le ou la modifier. Dans notre situation, la classe *MemAudioFile* est un proxy pour la classe *AudioFile*. En effectuant les méthodes de la classe *MemAudioFile*, on réalise en fait les méthodes de *AudioFile*.

1. Voir Annexe 2

* AbsAudioFile : Classe Interface des deux types de fichier possible. C’est notre sujet abstrait.
* AudioFile : C’est la classe sujet du patron Proxy. C’est l’objet avec la structure qu’on veut garder secrète.
* MemAudioFile: C’est notre classe Proxy qui représente le lien. Elle possède un AudioFile dans ses attributs afin de pouvoir y avoir accès.

# 4 – Patron Iterator

1. a) **L’intention du patron Iterator**:

Dans la situation du TP4, nous ne voulons pas modifier les méthodes du conteneur *vector* directement. On veut toutefois pouvoir utiliser les opérateurs « \* » et « -> » sur nos transformations plus facilement. On crée alors une nouvelle classe itérateur, dans notre cas *TransformIterator*, qui hérite de l’itérateur de *vector* et on surcharge ces opérateurs de la classe *AbsTransform* (nos transformations) à l’intérieur de notre classe itérateur *TransformIterator.* On dissimule ainsi la structure du conteneur *vector* tout en laissant la liberté de le modifier.

b) **La classe conteneur de la STL et les classes des Iterator utilisées** :

Un vecteur nommé *TransformContainer* est utilisé par la classe Composite. Les itérateurs de vecteur abstraits sont *TransformBaseIterator* et *TransformBaseIterator\_const* et leur enfant qui sont des itérateurs concrets sont *TransformIterator* et *TransformIterator\_const*.

1. **Expliquez le rôle de l’attribut statique *m\_empty\_transforms* dans AbsTransform** :

On ne peut pas savoir à l’avance quelle transformation nous auront dans un tableau donné. Toutefois, nous devons effectuer des opérations sur des objet qui seront des transformations valides (Invert, Repeat or Composite) lors de l’exécution. Le compilateur interprètera certain objet comme étant *AbsTransform* alors qu’en réalité il sera dans le futur un objet valide. Si la classe *AbsTransform* ne possède pas de méthode valide alors le programme ne compilera pas car il interprète ces opérations comme effectué sur un objet *AbsTransform*. Pour contourner le compilateur, on défini des méthodes pour une classe abstraite. Pour définir ces méthodes nous devons utiliser l’attribut stratique *m\_empty\_transforms* afin de retourner un élément valide, tout comme celle des objets valides. L’attribut est statique car il n’a pas besoin d’être différent pour chaque objet. Il ne sera pas utilisé de toute façon dans le véritable programme.

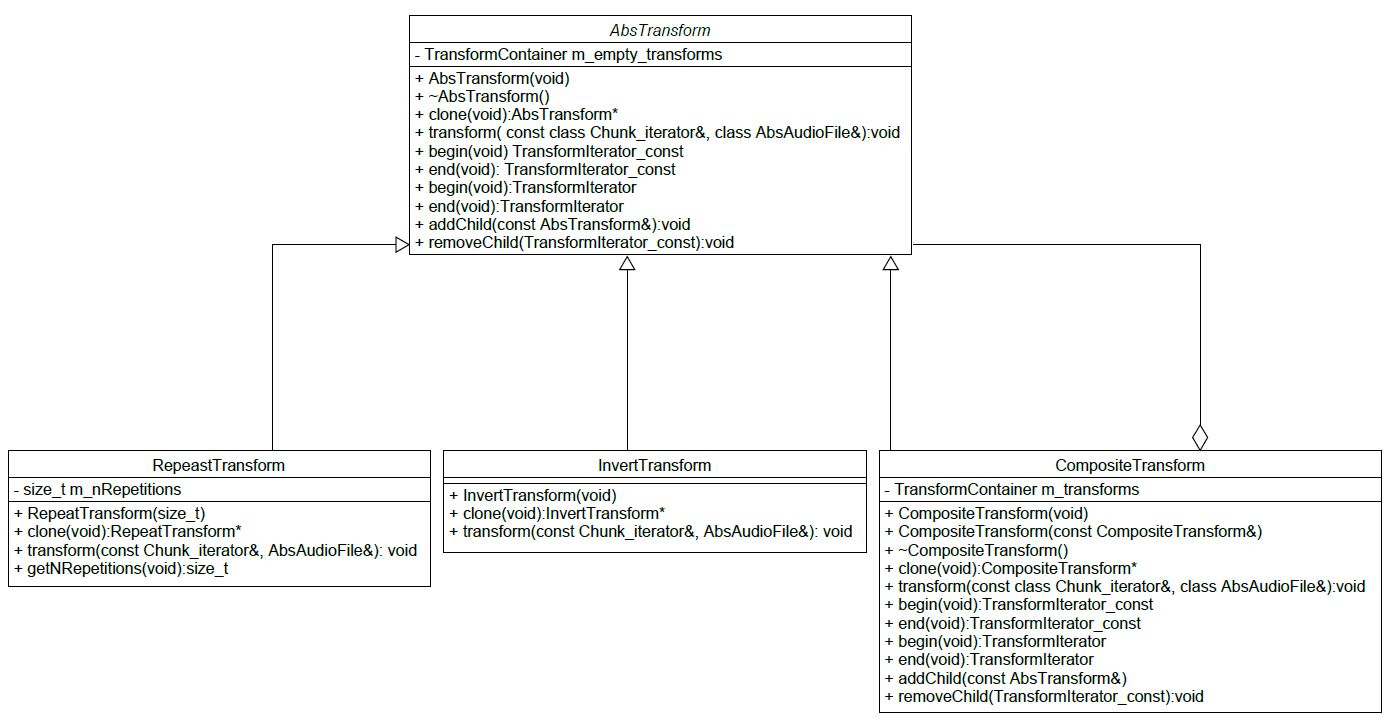
1. **Quelles seraient les conséquences sur l’ensemble du code si vous décidiez de changer la classe de conteneur utilisée pour stocker les enfants dans la classe Composite?**

Il n’y aurait pas de conséquence sur le code à l’extérieur des classes itérateurs créées dans le cadre du Patron Iterator. Les opérations qu’on désire effectuer sur nos objets dans le conteneur sont toutes déterminées dans nos classes itérateurs maisons. Tous les éléments à modifier se retrouvent au même endroit, ce qui facilite grandement le changement. Les éléments de notre conteneur choisi restent cachés, suivant les l’encapsulation de la programmation orienté-objet. Les éléments de notre itérateur sont publiques car on veut pouvoir les modifier à notre guise.

1. **Avantages et inconvénients de la surcharge de « \* » et « -> » par TransformIterator**

Elle facile l’accès à l’objet car on n’a pas besoin d’écrire des double « \* ». Cependant elle réduit la flexibilité car nous n’avons plus accès au premier pointeur. Ce-dernier est déjà déréférencer.

# Annexe 1



# Annexe 2

