

Antoine Gaulin - 1845639

Sébastien Cadorette - 1734603

**Travail pratique**

TP4

Travail présenté à

F. Guibault

Dans le cadre du cours

LOG2410 – Conception Logicielle

Polytechnique Montréal

6 décembre 2017

# Patron Visiteur

## Question 1 : Intention et avantages

L’intention du patron visiteur permet d’ajouter, de supprimer et ainsi de gérer aisément de nouvelles opérations sur une classe sans la modifier. Les opérations deviennent indépendantes de cette classe.

## Question 2 : Diagrammes de classes

Les diagrammes se décomposent en deux parties. D’une première part, on retrouve les visiteurs, et de l’autre on retrouve les éléments sur lesquels les visiteurs agissent. Les visiteurs possèdent une classe abstraite commune qui est *AbsFileVisitor*. S’en suit des deux sous-classes *FileStringReplace* et *FileChecksumCalculator*. Ces deux classes permettent la visitent des éléments, qui eux accepteront la visite. Les éléments possèdent une classe abstraite commune qui est *AbsAudioFile*. S’en suit des deux sous-classes *AudioFile* et *MemAudioFile*.

Voir les diagrammes *DiagrammeDeClasse\_FileChecksumCalculator.pdf* et *DiagrammeDeClasse\_FileStringFindReplace.pdf*.

## Question 3 : Ajout d’une nouvelle sous-classe

Il sera très simple d’ajouter de nouvelles sous-classes dérivées de *AbsAudioFile*, sans pour autant devoir effectuer de nombreux changements aux classes qui y sont reliées. En effet, c’est l’avantage que permet le patron Visiteur. Puisque que chaque sous-classe de *AbsAudioFile* représente un élément qui peut accepter des visiteurs, l’ajout d’une sous-classe ne demandera qu’à ajouter une fonction de visite dans chacun des visiteurs.

## Question 4 : Transformations implémentées comme visiteur

Oui il serait possible d’effectuer les transformations selon un patron visiteur. Les transformations pourraient fonctionner de la même façon que les sous-classes *FileStringReplace* et *FileChecksumCalculator*. Puisque les transformations agissent sur les fichiers audios, on conserve les éléments, et on met visiteur les transformations, et on obtient un diagramme semblable à celui de la question 2.

# Patron Commande

## 1. Généralités

### 1.1 Intention et avantages du patron Commande

L’utilisation du patron visiteur permet d’ajouter, de supprimer et, ainsi, de gérer aisément de nouvelles opérations sur une classe sans avoir à modifier celle-ci. Les opérations sont indépendantes de cette classe.

### 1.2 Structure des classes

Dans le diagramme on retrouve deux parties. D’une part, nous avons les commandes, et d’une autre nous avons l’exécuteur. Les commandes ont une classe abstraite nommée *AbsCommand*, de laquelle elles dérivent. Les commandes sont *TransformCommand*, *ChecksumCommand* et *StringFindReplaceCommand*. L’exécuteur est la classe *CommandExecutor*. L’exécuteur contient toutes les transformations au moyen d’une liste, et permet de les exécuter grâce à la fonction *ExecuteAllCommands()*.

Voir le fichier *DiagrammeDeClasse\_Command.pdf* pour le diagramme de classes du patron Commande.

## 2. Relation avec deux autres patrons de conception

### 2.1 Noms et intentions

La classe *CommandExecutor* participe également aux patrons *Template Method* et *Strategy*.

#### 2.1.1 Template Method

Le patron *Template Method* suggère la définition d’une interface, ce qui permet à ses sous-classes d’avoir un bon point de départ pour leur construction plus élaborée. Il donne avantage principalement à l’implantation d’un algorithme.

#### 2.1.2 Strategy

On utilise une classe pour faciliter l’implémentation d’opérations, plutôt que d’écrire au long à chaque fois l’opération que l’on veut effectuer. Cela permet d’implémenter un algorithme de façon simple, et de créer des sous-classes pour chacune des variantes de cette algorithme. Le code commun est alors placé dans une classe abstraite pour réduire la quantité de code redondante.

### 2.2 Éléments de la classe qui sont caractéristiques à ces patrons

#### 2.2.1 Template Method

Ce patron se retrouve dans le programme en offrant des interfaces pour les différents composants. En effet, on peut associer cela aux classes abstraites telles que *AbsCommand* et *AbsAudioFile*.

#### 2.2.2 Strategy

Chaque commande peut être perçue comme une variante d’un algorithme. En effet, nous retrouvons une classe abstraite *AbsCommand* qui est la base des commandes ayant chacune leur petite variante.

### 2.3 Raisons de l’utilisation du patron Commande

Le patron permet de gérer de façon efficace et simple une série de commande qu’on veut effectuer sur une autre classe. L’utilisation du patron Commande permettra de regrouper toutes ses commandes au même endroit et de les exécuter au moment voulu. De plus, si en cours de développement, une nouvelle commande doit être ajoutée au programme, il sera facile de le faire en y ajoutant qu’une sous-classe à la classe de commandes abstraites. Aucune autre modification n’a besoin d’être apportée au reste du programme.

## 3. Ajout de nouvelles commandes

Non, aucune autre classe n’a besoin d’être modifiée pour ajouter une nouvelle commande, autre que la nouvelle commande elle-même. C’est ce que permet le patron Commande d’ailleurs. L’ajout de nouvelles commandes doit se faire de façon facile et simple, sans avoir à repasser à travers tout le code pour y ajouter les nouvelles fonctionnalités. Une nouvelle commande sera donc simplement ajoutée en créant une nouvelle sous-classe de *AbsCommand*.