Travail Pratique #5 : Conception à base de patrons II

LOG2410 : Conception logicielle

Groupe : B1

Rapport remis par :
Mariam Sarwat (1928777)
Yasmina Abou-Nakkoul (1897266)

École Polytechnique de Montréal
Date de remise (16-04-2019)

1- Patron Visiteur

1) L'intention et les avantages du patron Visiteur

Appartenant à la familles des patrons comportementaux, le patron Visiteur s'intéresse aux comportements entre différents objets. L'intention de ce patron est de définir une nouvelle opération qui doit être appliquée sur plusieurs éléments d'une structure d'objets. L'opération sera définie sans toutefois changer les classes des éléments sur lesquels elle opérera.

Le patron Visiteur peut définir plusieurs méthodes, dont chacune pourrait prendre des arguments contenant des classes différentes. Ceci peut devenir un problème, car afin de choisir la bonne méthode pour un certain objet, il faut vérifier la classe qui y est présente en argument.

Heureusement, le patron Visiteur offre une solution à ce problème en implémentant "double dispatch". Cette méthode fait en sorte que l'opération exécutée dépende du nom de la requête et du type des deux receveurs (type du Visiteur et de l'élément visité). Ceci va à l'encontre du principe du "single dispatch", où l'opération dépend du nom de la requête et du type du receveur (dans ce cas, il n'y aurait pas de noeud Visiteur). L'élément visité aura une méthode "accept()" qui indiquera au visiteur quelle méthode prendre.

Ceci est le seul changement qui sera fait dans les classes des éléments sur lesquelles le patron Visiteur opérera.

Trois avantages notables du patron Visiteur sont:

a) La flexibilité: les "Visitors" et la structure d'objets sont indépendants.

b) Le code associé à une fonctionnalité spécifique se retrouve à un seul endroit bien identifié.

c) L'ajout de nouvelles opérations, sans toucher à la hiérarchie des objets.

2) Diagramme de Classes

Voir les diagrammes DiagrammeDeClasse_FileChecksumCalculator.pdf et DiagrammeDeClasse_FileStringFindReplace.pdf pour les diagramme de classes du patron Visiteur.

3) Ajout d'une nouvelle sous-classe dérivée de *AbsAudioFile*

Si nous décidons d'ajouter une nouvelle sous-classe dérivée de `AbsAudioFile`, nous aurons à ajouter de nouvelles fonctions *visit()* dans la classe `AbsFileVisitor`. Ce dernier est la classe de base abstraite de la hiérarchie de

Visiteur, et dans ses classes dérivées: `FileStringFindReplace` et `FileChecksumCalculator`.

4) Transformations implémenté comme visiteur

L'application des transformations aux fichiers audio pourrait définitivement être implémentée comme un visiteur pour une raison majeure:

1. Les transformations, étant distinctes et n'ayant aucun lien entre elles, à faire sur tous les éléments (chunks), soit différents objets ayant des classes différentes.

Un avantage serait l'augmentation de la flexibilité du système: comme les transformations seraient implémentées dans les Visiteurs, elles seraient séparées de la structure des objets, ce qui ferait en sorte qu'ajouter des transformations n'aurait pas d'effet sur la hiérarchie des classes. De plus, si un seul élément d'une classe devait redéfinir une fonction de transformation, le patron Visiteur s'assure que le code ne se disperse pas puisque toutes les transformations seraient regroupées dans la classe Visitor des transformations.

Cependant, puisqu'`AudioFile` et `MemAudioFile` ont des composantes différentes, il faudra effectuer une duplication à l'implémentation du visiteur ce qui consiste en un désavantage.

2 - Patron Commande

1) L'intention et les avantages du patron Commande

L'utilisation du patron commande permet d'encapsuler une requête à l'aide d'un objet. Cette encapsulation permet d'attribuer une requête différente et une queue de requêtes à chaque client. De plus, ce patron permet le support de l'opération « Annuler », soit un « undo » en anglais.

Ce patron possède plusieurs avantages. Tout d'abord, grâce à celui-ci, il est facile d'implémenter de nouvelles commandes. De plus, comme mentionné précédemment, les commandes sont encapsulées dans des objets qui peuvent être manipulés, comme n'importe quel objet, ce qui amène, à plus de flexibilité. Par la suite, l'objet qui appelle la requête est séparé, soit découplé, de la sous-classe qui peut la satisfaire. Finalement, il est possible d'assembler les commandes afin de créer des commandes composites si nécessaire. Dans ce cas, le patron Commande est combiné au patron Composite afin de pouvoir représenter les commandes composées d'un ensemble d'autres commandes.

2) Diagramme de classe

Voir le fichier DiagrammeDeClasse_Command.pdf pour le diagramme de classes du patron Commande.

3) Relation avec deux autres patrons de conception

a) Les noms et intentions de ces patrons

La classe *CommandeExecutor* est présente dans le patron *Strategy* ainsi que le patron *Template Method*.

Tout d'abord, le patron *Strategy* permet à l'algorithme de varier indépendamment des clients qui l'utilisent. En effet, ce dernier encapsule un algorithme dans une classe afin de la rendre interchangeable. En fait, dans notre cas, la classe *CommandeExecutor,* qui performe une validation sur les données d'entrée, peut utiliser un patron *Strategy* afin de choisir le bon algorithme a implémenter. Ces algorithmes peuvent être caractérisés comme des stratégies.

De son bord, le patron *Template Methode* permet de définir le squelette d'un algorithme dans une opération. Cependant, certaines parties seront configurées dans les sous-classes clientes. En effet, la classe correspondant à ce patron permet aux sous-classes de redéfinir certaines étapes sans changer le squelette de celui-ci. En fait, dans notre cas, la classe *CommandeExecutor* permet cette redéfinition.

b) Éléments de la classe *CommandeExecutor* qui sont caractéristiques de ces patrons

En ce qui concerne le patron *Strategy,* la classe *CommandeExecutor*, comme mentionné précédemment, performe une validation des données d'entrées dans le but de bien identifier l'algorithme à implémenter. En effet, on retrouve la classe abstraite *AbsCommand* qui s'agit comme classe de base pour les différentes commandes qui sont implémentés.

Pour le patron *Template Methode,* la classe *CommandeExecutor* agit comme une interface pour les différentes composantes qui lui sont reliés. Dans notre cas, il y ces composantes sont les classes abstraites *AbsCommand* et *AbsAudioFile.*

c) Raisons d'utilisation de ces patrons

Le patron *Strategy* pourrait être un bon choix pour la raison mentionnée en a) et b), et parce que ce patron définit une interface commune pour plusieurs algorithmes.

Le patron *Template Méthode,* quant à lui, pourrait être un bon choix, car, tout d'abord, il favorise la réutilisation de code. De plus, celui-ci suit le principe Hollywood qui implémente le dicton "ne nous appelez pas, nous vous appellerons". Ce principe mène à ce qu'on appelle une inversion de contrôle. En effet, ce patron permet de définir une classe qui sera en charge de faire

l'appel de la sous-classe correspondant à la requête du client. Finalement, ce patron permet d'imposer des règles de surcharge dans ces classes.

4) Ajout de nouvelles sous-classes de la classe *AbsCommand*

En effet, grâce au patron Commande, l'ajout de nouvelle commande est facile et ne nécessite pas de modification aux classes déjà présentent. En effet, cette addition se fera en implémentant une nouvelle sous-classe dérivant de la classe *AbsCommande.* En effet, cette sous-classe aura ses propres attributs qu'elle utilisera afin d'effectuer les tâche relié à la commande.