Projet-INF224

Toutes les questions ont été traitées. Pour la partie en C++, j'ai fait les questions additionnelles suivantes : - partie 10 : j'ai implémenté deux méthodes pour effacer les médias et les groupes - partie 12 : on peut sérialiser et désérialiser des groupes

Pour la partie Java, j'ai implémenté une télécommande qui permet d'envoyer n'importe quelle requête vers le serveur.

4e étape : Photos et Vidéos

La méthode `play` ne doit pas avoir d'implémentation dans la classe de base. C'est donc une méthode abstraite. On le code de la façon suivante :

```
virtual void play() const = 0;
```

Comme une de ses méthodes est abstraite, la classe de base est désormais abstraite, ce qui signifie qu'on ne peut plus instancier des objets de cette classe.

5e étape : Traitement uniforme

On peut traiter des objets photos et vidéos avec le même code grâce au polymorphisme : les classes `Video` et `Photo` héritent de la superclasse `AbstractMedia`.

Toutefois, on ne code pas exactement de la même façon qu'en Java : on doit utiliser le déréférencement pour accéder aux méthodes des objets vidéos ou photos, parce qu'on a créé un tableau de pointeurs. Par exemple :

```
objects[i]->play();
```

En Java, on écrirait plutôt :

```
objects[i].play();
```

Cette différence est due au fait que le tableau `objects` contient ici des pointeurs vers les objets. Il est également possible de créer des tableaux d'objets en C++ : dans ce cas, on n'aurait pas besoin de déréférencer. En Java, on a une seule possibilité pour les tableaux d'objets : on fait des tableaux de références. Il n'y a donc pas besoin de distinguer deux cas.

Ici, il est nécessaire d'utiliser un tableau de pointeurs parce que les éléments du tableau n'ont pas le même type, et donc ne prennent pas la même taille en mémoire. Or, les éléments du tableau seront attribués la même place en mémoire.

Dans le cas du C++, il est également nécessaire de détruire les pointés quand on ne les utilise plus parce qu'il n'y a pas de ramasse-miettes comme en Java.

6e étape : Films et tableaux

Comme on code un accesseur pour le tableau des durées des chapitres, nous avons besoin de connaître sa longueur. J'ai donc choisi de créer une deuxième variable d'instance `_numberOfChapters`.

De façon cohérente, j'ai implémenté un deuxième accesseur, donc un pour chaque variable d'instance. Ainsi, on peut utiliser correctement le tableau de durée puisqu'on connait sa longueur.

Afin de respecter l'encapsulation, l'accesseur du tableau de durée renvoie une **copie profonde du tableau sur le tas**. De même pour le modifieur qui réalise une copie profonde du tableau donné en argument.

7e étape : Destruction et copie des objets

La seule classe qui générait des fuites mémoires était la classe `Film` car c'est la seule qui avait un pointeur parmi les variables d'instances. Pour corriger ce problème, on libère le pointeur dans le destructeur de l'objet.

De même, puisque la classe `Film` contient ce fameux pointeur, on ne peut pas simplement réaliser des copies superficielles. En effet, le pointeur sera le même pour toutes les classes copiées et elles partageront donc leur objet pointé.

Il faut dont faire une copie en profondeur et recopier l'objet pointé et non le pointeur. J'ai implémenté un copy constructor pour la classe `Film`.

8e étape : créer des groupes

La liste d'objets doit être une liste de pointeurs afin que l'on puisse utiliser le polymorphisme. En Java, en dehors des types de base, on utilise toujours des références pour les listes d'objets, il n'y a donc pas de choix équivalent à faire.

13e étape : traitement des erreurs

J'ai utilisé des exceptions pour gérer les cas suivants : - si on crée plusieurs groupes ou objets ayant le même nom - si on supprime un groupe ou un objet qui n'existe pas