Exercice 22 - ProjectCalendar

Remarque : Le but de ce TD est de se familiariser avec les notions d'association, d'agrégation et de composition entre classes et de leurs conséquences au niveau de l'implémentation des classes.

Remarque : Ce TD utilise le thème du projet de cette année afin de commencer à vous familiariser avec les différentes entités de l'application qui sera à développer. On ne perdra pas de vue que les questions développées dans ce TD ne constituent pas une architecture pour le projet. Celle-ci devra être retravaillée en tenant compte de l'ensemble des entités du sujet de projet.

Remarque : Dans cet exercice, nous utiliserons les classes `Duree`, `Date` et `Horaire` fournies dans les fichiers `timing.h` et `timing.cpp`. Il faut réutilier directement ces classes sans avoir à les redéfinir.

On souhaite développer l'application ProjectCalendar destinée à mixer des fonctionnalités d'un agenda électronique traditionnel et d'un outil de gestion de projet. Dans ce problème, on s'intéresse à une sous-partie simplifiée des fonctionnalités de l'application qui consiste à gérer un ensemble de tâches (à effectuer dans le cadre de la gestion d'un ou plusieurs projets) et à leur programmation (leur associer des dates et horaires de démarrage).

Une tâche est caractérisée par un identificateur (une identité unique), un titre (permettant d'en saisir la nature), une durée, une date de disponibilité (avant laquelle on ne peut pas commencer la tâche) et une date d'échéance (avant laquelle la tâche doit être terminée). Dans la suite une tâche, sera représentée par un objet de la classe `Tache` qui comportera un attribut `identificateur` de type `string`, un attribut `titre` de type `string`, un attribut `duree` de type `Duree`, un attribut `disponibilite` de type `Date` et un attribut `echeance` de type `Date`. **L'unique constructeur** de cette classe a 5 paramètres qui permettent d'initialiser les attributs d'un objet.

Les objets `Tache` utilisés pour l'application sont gérés par un module appelé `TacheManager` qui est responsable de leur création (et destruction). En pratique, un objet `TacheManager` peut créer un ensemble d'objets `Tache` déjà existant à partir d'un nom de fichier transmis en argument ou à partir d'une base de données (cette fonctionnalité ne sera pas implémentée dans l'exercice). La classe possède aussi une méthode `ajouterTache` qui permet de créer une nouvelle tâche en transmettant les caractéristiques de cette nouvelle tâche à la méthode. De plus, la classe possède une méthode `getTache` qui permet d'obtenir une référence sur l'objet `Tache` dont le code est transmis en argument.

Dans l'application, les tâches doivent être ordonnancées, *i.e.* une date et un horaire doivent leur être attribués. Pour cela, on utilise des objets de la classe `Programmation`. Cette classe comporte un attribut `Tache` représentant un objet de la classe `tache`, un attribut `date` de type `Date` et un attribut `horaire` de type `Horaire` renseignants sur l'ordonnancement de la tâche. Elle comporte aussi les accesseurs standards pour communiquer avec les objets de la classe. Les objets `Programamtion` utilisés pour l'application sont gérés par un module appelé `ProgrammationManager` qui est responsable de leur création (et destruction).

Préparation : Créer un projet vide et ajouter trois fichiers `Calendar.h`, `Calendar.cpp` et `main.cpp`. Définir la fonction principale `main` dans le fichier `main.cpp`. Ajouter aussi dans votre projet les 2 fichiers `timing.h` et `timing.cpp` fournis avec le sujet. Après avoir recopié le code ci-après dans le fichier `Calendar.h`, s'assurer que le projet compile correctement. Au fur et à mesure de l'exercice, on pourra compléter la fonction principale en utilisant les éléments créés. Les situations exceptionnelles seront gérées en utilisant la classe d'exception suivante (à recopier dans le fichier `Calendar.h`).

```
#ifndef CALENDAR_h
#define CALENDAR_h
#include<string>
#include<iostream>
#include "timing.h"
using namespace std;
using namespace TIME;

class CalendarException{
public:
 CalendarException(const string& message):info(message){}
 string getInfo() const { return info; }
private:
 string info;
};
#endif
```

Calendar.h

Question 1

Lire le sujet en entier et identifier les différentes entités de l'application. Identifier les associations qui existent entre ces classes. Quel type de lien existe t-il entre un objet `TacheManager` et les objets `Taches` qu'il crée et auxquels il donne accès ? Quel type de lien existe t-il entre un objet `Programmation` et un objet `Tache` auquel il est associé ? Quelle type d'association existe t-il entre la classe `ProgrammationManager` et la classe `Programmation` ? Entre la classe `Tache` et elle-même ?

Question 2

Définir la classe `Tache` ainsi que l'ensemble de ses méthodes. Quel pourrait être l'intérêt d'utiliser des références **`const`** pour les paramètres `string` du constructeur ? La classe `Tache` nécessite t-elle (a priori) un destructeur, un constructeur de recopie et/ou un opérateur d'affectation ? Expliquer. Définir ces méthodes seulement si nécessaire.

Question 3

Est-il possible de définir un tableau (alloué dynamiquement ou non) d'objets `Tache` ? Expliquer. Est-il possible de créer un tableau (alloué dynamiquement ou non) de pointeurs d'objet `Tache` ? Expliquer.

Question 4

Soit la classe `TacheManager` dont voici une définition partielle :

```
class TacheManager {
private:
 Tache** taches;
 unsigned int nb;
 unsigned int nbMax;
 void addItem(Tache* t);
public:
 TacheManager();
 void ajouterTache(const string& id, const string& t, const Duree& dur, const Date&
     dispo, const Date& deadline);
 Tache& getTache(const string& id);
 const Tache& getTache(const string& id) const;
};
```

Calendar.h

La méthode `ajouterTache` permet de créer une nouvelle tâche avec ses caractéristiques transmises en argument. Cette méthode déclenche une exception si la tâche ajoutée possède le même identificateur qu'une tâche qui existe déjà. Pour créer un nouvel objet `Tache`, la méthode *alloue dynamiquement* un objet `Tache` puis fait appel à la méthode privée `addItem` pour sauvegarder l'adresse de ce nouvel objet. Dans la méthode `addItem`, l'adresse est sauvegardée dans un tableau de pointeurs d'objets `Tache` qui a été alloué dynamiquement et dont l'adresse est stockée dans l'attribut `taches` de type `Tache**`. L'attribut `nb` représente le nombre d'adresses sauvegardées dans ce tableau. L'attribut `nbMax` représente le nombre maximum d'adresses qui peut être sauvegardées avant un agrandissement du tableau (*i.e.* la taille du tableau pointé par `taches`). La méthode `addItem` gère les éventuels besoins en agrandissement du tableau. La classe `TacheManager` possède pour l'instant un unique constructeur sans argument. Initiallement, le tableau pointé par `taches` ne contient aucune adresse.
Définir les méthodes de la classe `TacheManager`.

Question 5

La classe `TacheManager` nécessite t-elle le développement d'un destructeur ? Pourquoi ? Si oui, implémenter ce destructeur (on ne tiendra pas compte de la sauvegarde des tâches dans un fichier ou une base de données).

Question 6

Dans l'hypothèse où la duplication d'un objet `TacheManager` est autorisée, la classe `TacheManager` nécessite t-elle le développement d'un constructeur de recopie et/ou d'un opérateur d'affectation ? Si oui, implémenter ces méthodes.

Question 7

Supposons que l'on veuille représenter des contraintes de précédence entre tâches, c'est à dire que l'on veut associer à une tâche i un ensemble de tâches qui doivent avoir été terminées avant de commencer la tâche i. Quelle type d'association lie la classe `Tache` à elle-même ? Quel type d'attribut allez vous utiliser pour implémenter cette association ?

Question 8

Définir la classe `Programmation` ainsi que l'ensemble de ses méthodes.

Question 9

Dans l'hypothèse où la duplication d'un objet `Programmation` est autorisée, la classe `Programmation` nécessite t-elle le développement d'un destructeur, d'un constructeur de recopie, et/ou d'un opérateur d'affectation ? Si oui, implémenter ces méthodes.

Question 10

-*Question à faire à la maison pour s'exercer*- Définir la classe `ProgrammationManager` ainsi que l'ensemble de ses méthodes. En pratique, la classe possède aussi un tableau (dont la capacité sera étendue selon les besoins) pour stocker les différentes programmations des tâches.

Exercice 23 - Gestion de fichier *-Exercice d'approfondissement à faire à la maison-*

Dans cet exercice, on veut mettre en place la partie "*gestion de fichier*" de la classe `TacheManager` de l'Exercice 22. Cet exercice, qui n'est pas à faire dans le cadre de la séance de TD, a pour but de se familiariser avec la gestion des fichiers en C++.

On veut pouvoir construire des objets `Tache` à partir d'un fichier contenant les caractéristiques de ces taches. De plus, afin de rendre persistante les informations recueillies sur des objets `Tache` durant la durée de vie d'un objet `TacheManager`, celles-ci devraient pouvoir être sauvegardées dans un fichier (qui pourra être utilisé à une prochaine recréation de cet objet).
Il existe plusieurs formats de fichier qui peuvent permettre de structurer de telles informationss (comme par exemple XML). Dans cet exercice, on utilisera des fichiers textes où l'on enregistra séquentiellement les différents objets avec exactement une valeur d'attribut par ligne.

Question 1

Dans la classe `TacheManager`, définir une méthode `load` prenant un paramètre de type `string` destiné à recevoir le nom du fichier physique contenant les caractéristiques sur un ensemble d'objets `Tache` à construire. Pour cela, on utilisera la bibliothèque `fstream` du `C++`.

Question 2

Définir une méthode `save` prenant un paramètre de type `string` destiné à recevoir le nom du fichier physique dans lequel seront enregistrés les objets `Tache` contenu dans un objet `TacheManager`. Modifier le destructeur de la classe `TacheManager` de manière à mettre à jour le fichier de taches à la fin du cycle de vie d'un objet `TacheManager`.