TD3 : Le cinéma

On veut gérer les plannings des salles d'un cinéma qui dispose de `n` salles identifiées par un numéro unique entre 1 et `n`. L'ensemble des films sont conservés dans une cinémathèque. Quand un film est programmé au cinéma, il est identifié par son titre, et est caractérisée par sa durée en minutes. Il est diffusé dans une ou plusieurs salles. Une programmation d'un film dans une salle correspond à un intervalle de temps caractérisé par une date de début et une date de fin. Une date est caractérisée par un jour, un mois, une année. Pour chaque salle de cinéma, un planning est établi pour gérer les différentes programmations. Pour un jour donné, une salle ne peut diffuser qu'un seul film. On doit donc faire en sorte que toutes les programmations liées à une salle ne se chevauchent pas : il ne peut pas y avoir deux films le même jour dans une même salle.

On veut construire une application qui permet de gérer les plannings des salles d'un cinéma. Les fonctionnalités immédiates recherchées pour une salle donnée sont :

- L'ajout d'une programmation au planning en vérifiant que chaque nouvelle programmation ne chevauche pas une programmation qui existe déjà.
- L'affichage du planning comprenant les différentes programmations avec le détail des films et ses caractéristiques.

Dans la suite, on réutilisera les classes `Date`, `Duree` et `Intervalle` dont les implémentations sont fournies dans les fichiers `time.h` et `time.cpp`.

Toutes les classes `T` qui seront implémentées devront comporter une méthode `T::Afficher(std::ostream&) const` qui permettra d'afficher un objet de la classe `T` sur un flux `ostream` passé en argument. Réfléchir à un possible argument par défaut pour ces méthodes. Cette méthode sera utilisée par une fonction `operator<<(std::ostream&, const T&)` pour envoyer un objet directement sur un flux `ostream`.

Les situations exceptionnelles seront gérées en utilisant la classe d'exception suivante (à recopier) :

```
class CinemaException {
public:
      CinemaException(const std::string& m):info(m){}
      const std::string& GetInfo() const { return info; }
private:
      std::string info;
};
```

Question 1- Identifier les différentes entités du monde décrit ci-dessus. Etablir un modèle UML où apparaissent les différentes classes utilisées dans l'application. Identifier les associations qui existent entre ces classes. Compléter cette description en lisant attentivement les questions suivantes.

Question 2- Ecrire la classe `Film` en utilisant la classe `std::string` pour l'attribut titre.

Question 3- Ecrire une classe `Cinematheque` qui permettra de stocker des objets `Film`. On fera attention à gérer la mémoire correctement. Cette classe disposera d'une méthode `void AjouterFilm(const std::string& t, int d)` qui permettra d'ajouter un film en fournissant un titre et une durée. Cette classe disposera aussi d'une méthode `const Film& GetFilm(const std::string& t) const` qui permettra de récupérer un objet `Film` à partir de son titre. Si le titre n'existe pas dans la cinémathèque, la méthode déclenchera une exception. Puisque l'on veut éviter d'avoir plus d'un objet `Cinematheque` en mémoire pour que tous les films soient centralisés dans le même objet, utiliser le design pattern Singleton.

Question 4- Ecrire les classes `Programmation, Planning et Salle`. Le planning d'une salle utilisera un tableau alloué dynamiquement d'objets `Programmation` dont la taille sera passée en argument au constructeur. Réfléchir à la pertinence d'utiliser le mot clé `explicit` dans ce cas. On fera attention à gérer la mémoire correctement. On fera en sorte qu'un planning puisse être dupliqué de façon à obtenir deux objets indépendants. Surcharger `operator<<` pour ajouter une nouvelle programmation au planning. Cette méthode renverra un booléen qui indiquera si la nouvelle programmation a pu être ajoutée (suffisamment de place, pas d'intersection avec une programmation existante, ...). Les programmations seront toujours triées par ordre croissant de dates.

Question 5- Afin de pouvoir parcourir les différentes programmations d'un planning, appliquer le design pattern Iterator à la classe `Planning` en déduisant son implémentation de l'exemple donné ci-après.

Question 6- Ecrire la classe `Cinema` qui compose des objets `Salle`. Le constructeur de cette classe prendra en argument le nom du cinéma et son nombre de salles. Empêcher la duplication (par copie et affectation) d'un objet de la classe `Cinema`. Surcharger l'opérateur `operator[]` pour obtenir une référence sur la ième (1<=i<=n) salle d'un cinéma (en versions `const` et non `const`). Déclencher une exception si le numéro de la salle est incorrect.

Dans la fonction principale `main`, programmez vos films préférés pour les deux prochaines semaines.

Le programme suivant vous est fourni à titre d'exemple :

```
int main(){
      try {
            Cinematheque& collection=Cinematheque::GetInstance();
            collection.AjouterFilm("Kill Bill 1", 123);
            collection.AjouterFilm("Kill Bill 2", 156);
            collection.AjouterFilm("2001, Space Odyssey", 152);
            collection.AjouterFilm("School of Rock", 118);
            collection.AjouterFilm("Freaky Friday", 112);
            Cinema cine("Majestic",2);
            cine[1].GetPlanning()<<Programmation(collection.GetFilm("Kill Bill
1"),Date(5,5,2010),Date(11,5,2010));
            cine[2].GetPlanning()<<Programmation(collection.GetFilm("Kill Bill
2"),Date(5,5,2010),Date(11,5,2010));
            cine[1].GetPlanning()<<Programmation(collection.GetFilm("2001, Space
Odyssey"),Date(12,5,2010),Date(18,5,2010));
            cine[1].GetPlanning()<<Programmation(collection.GetFilm("School of
Rock"),Date(19,5,2010),Date(25,5,2010));
            cine[2].GetPlanning()<<Programmation(collection.GetFilm("Freaky
Friday"),Date(12,5,2010),Date(25,5,2010));
            cine.Afficher();
            std::cout<<"\nEssai Iterateur\n";
            for(Planning::iterateur it=cine[1].GetPlanning().begin();
                                          it!=cine[1].GetPlanning().end();++it)
                  std::cout<<*it<<"\n";
            for(Planning::iterateur it=cine[0].GetPlanning().begin();
                                          it!=cine[0].GetPlanning().end();++it)
                  std::cout<<*it<<"\n";
            Cinematheque::LibereInstance();
      }
      catch(CinemaException e){ std::cout<<e.GetInfo()<<"\n"; }
      catch(TimeException e){ std::cout<<e.GetInfo()<<"\n"; }
      std::cout<<"Fin projet cinema\n";
      system("pause"); return 0;
}
```

ainsi que son affichage attendu :

```
Cinema Majestic
SALLE Numero 1
- [05/05/2010 ; 11/05/2010] : Kill Bill 1 (2H03)
- [12/05/2010 ; 18/05/2010] : 2001, Space Odyssey (2H32)
- [19/05/2010 ; 25/05/2010] : School of Rock (1H58)
SALLE Numero 2
- [05/05/2010 ; 11/05/2010] : Kill Bill 2 (2H36)
- [12/05/2010 ; 25/05/2010] : Freaky Friday (1H52)

Essai Iterateur
[05/05/2010 ; 11/05/2010] : Kill Bill 1 (2H03)
[12/05/2010 ; 18/05/2010] : 2001, Space Odyssey (2H32)
[19/05/2010 ; 25/05/2010] : School of Rock (1H58)
erreur : cette salle n'existe pas
Fin projet cinema
Appuyez sur une touche pour continuer...
```