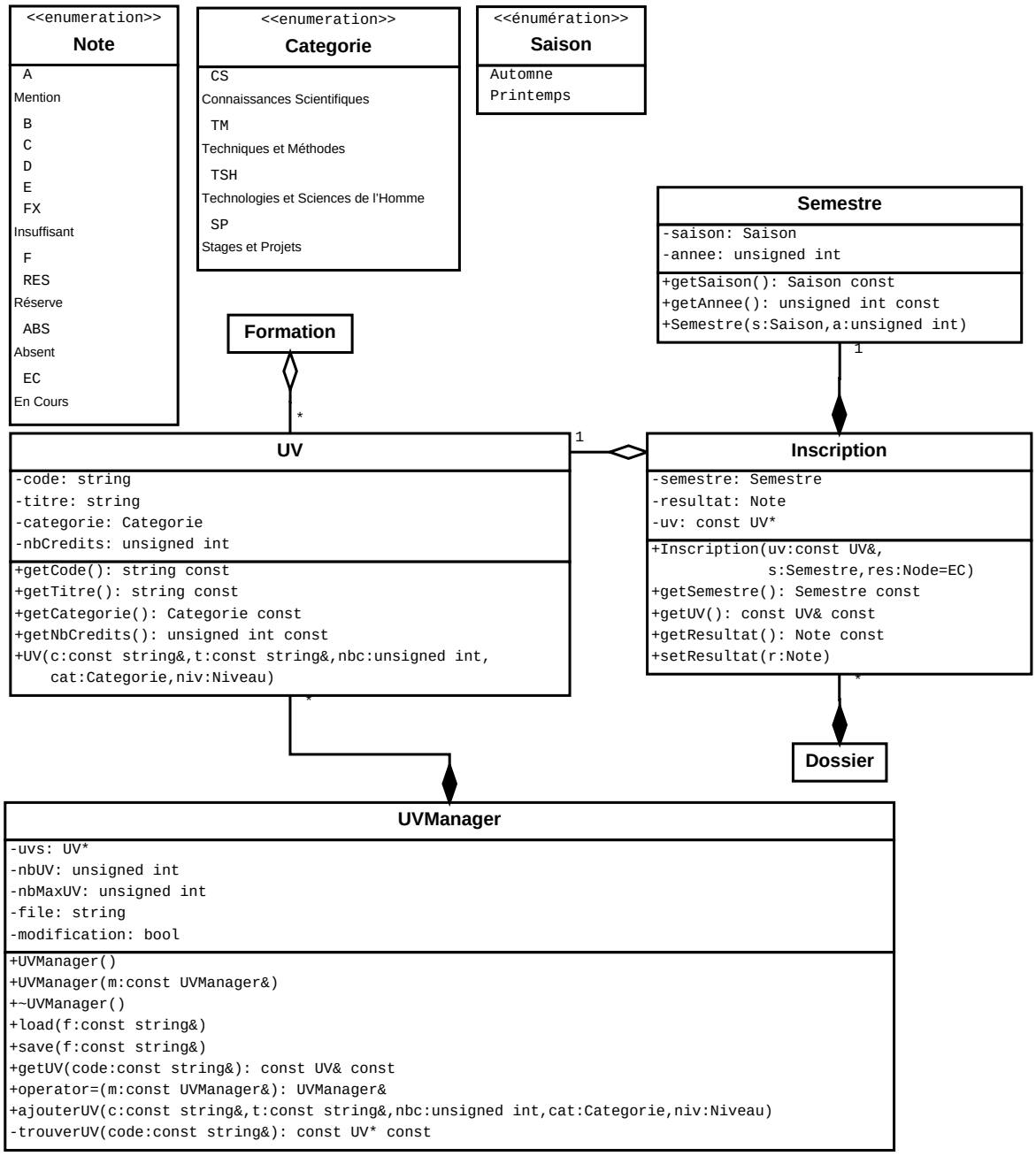
Exercice 24 - Design patterns

On a commencé à développer des classes pour l'application "UT Profiler" destinée à aider un étudiant d'une Université de Technologie à gérer et choisir ses inscriptions aux différentes Unités de Valeur (UV) tout au long des semestres.

L'ensemble des classes déjà développées se trouvent dans une archive à votre disposition. Cette archive contient 3 fichiers. Les fichiers `UTProfiler.h` et `UTProfiler.cpp` contiennent l'ensemble des classes déjà existantes. Cet ensemble est résumé dans le diagramme de classe ci-dessous :



Remarque : Les classes fournies dans l'archive correspondent à celles développées dans le cadre des Exercices 22 et 23. Cependant, les exercices sont indépendants et il suffit de lire la description ci-dessous pour traiter cet exercice.

Dans l'application `UTProfiler`, une UV d'une Université de Technologie est représentée par un objet de la classe `UV`. La classe `UV` comporte 2 attributs `code` et `titre` de type `string`, un attribut `nbCredits` de type **`unsigned int`**, et un attribut `categorie` de type `Categorie`. Les types `Categorie` et `Niveau` sont des énumérations définies dans le code ci-après. Les méthodes `getCode()`, `getTitre()`, `getNbCredits()`et `getCategorie()` permettent de connaître la valeur de ces attributs. **L'unique constructeur** de cette classe a 4 paramètres qui permettent d'initialiser les attributs d'un objet.

Les objets `UV` utilisés pour l'application sont gérés par un module appelé `UVManager` qui est responsable de leur création (et destruction).

En pratique, la classe `UVManager` dispose d'une méthode `load` qui permet de construire un ensemble d'objets `UV` en récupérant leurs caractéristiques à partir d'un fichier dont le nom est transmis en argument. Le fichier `UV_UTC.txt` fourni peut être utilisé comme exemple. Cela pourrait aussi se faire en récupérant ces informations dans une base de données (cette dernière fonctionnalité n'est pas implémentée dans le code fourni). La classe possède aussi une méthode `ajouterUV` qui permet de créer une nouvelle UV en transmettant ses caractéristiques. De plus, la classe possède une méthode `getUV` qui permet d'obtenir une référence sur l'objet `UV` dont le code est transmis en argument. Notons qu'un objet `UVManager` a la responsabilité des objets `UV` qu'il crée (création/destruction). La méthode `save` permet de sauvegarder les caractéristiques de l'ensemble des objets `UV`. Pour assurrer la persistance de nouvelles informations, lorsqu'un objet `UVManager` est détruit, la méthode `save` est utilisée pour mettre à jour le fichier d'UVs en cas d'éventuels ajouts d'UV ou de mise à jour des UVs déjà existantes.
Un dossier étudiant est représenté par un objet de la classe `Dossier` qui comporte plusieurs inscriptions. Une inscription est représentée par un objet de la classe `Inscription`. Cette classe comporte un attribut `uv` représentant un objet de la classe `UV`, un attribut `semestre` de type `Semestre` renseignant sur le semestre d'incription à l'UV et un attribut `resultat` de type `Note` correspondant au résultat obtenu à l'UV. Elle comporte aussi les accesseurs standards pour communiquer avec les objets de la classe.
À chaque sous-formation d'une UT (le TC, une branche, une filière, un parcours, un mineur) correspond un ensemble d'UV qui donne droit à une prise en compte de crédits. Pour simplifier, on représente ici une telle sous-formation par un objet de la classe `Formation`.

Question 1

Expliciter des intérêts de mettre en place le Design Pattern *Singleton* pour la classe `UVManager`. Implémenter ce design pattern. Modifier votre code en conséquence. Mettre à jour le diagramme de classe.

Question 2

On remarque que la duplication malencontreuse d'un objet `UV` pourrait poser des problèmes. Mettre en place les instructions qui permettent d'empêcher la duplication d'un objet `UV`. De plus, faire en sorte que seule l'unique instance de la classe `UVManager` puisse créer des objets `UV`.

Question 3

Afin de pouvoir parcourir séquentiellement les UVs stockées dans un objet `UVManager`, appliquer le design pattern Iterator à cette classe en déduisant son implémentation du code suivant :

```
UVManager& m=UVManager::getInstance();
m.load("UV_UTC.txt");
for(UVManager::Iterator it= m.getIterator();!it.isDone();it.next()){
  std::cout<<it.current()<<"\n";
}
```

Question 4

Refaire la question précédente en proposant une interface d'itérateur similaire à celle utilisée par les conteneurs standards du C++ (STL), *i.e.* qui permet de parcourir séquentiellement les différentes UVs d'un objet `UVManager` avec le code suivant :

```
for(UVManager::iterator it=m.begin();it!=m.end();++it)
   std::cout<<*it<<"\n";
```

Question 5

Implémenter une classe d'iterateur qui permet de parcourir l'ensemble des objets `UV` d'une catégorie donnée contenu dans un objet `UVManager`. On pourra, par exemple, s'inspirer du code suivant :

```
for(UVManager::FilterIterator it= m.getFilterIterator(CS);!it.isDone();it.next())
   std::cout<<it.current()<<"\n";
```