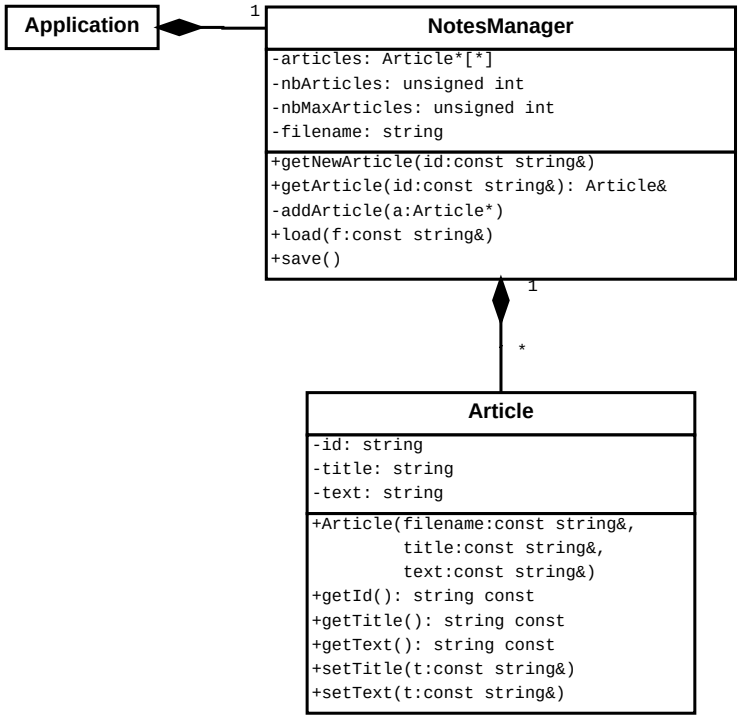
Exercice 24 - Design patterns

On a commencé à développer une application destinée à **éditer et gérer un ensemble de *notes textuelles*** que l'on appelle *article*. Un article peut par exemple correspondre au compte-rendu d'une réunion ou à des notes prises lors d'une séance de cours. Un article est caractérisé par un identificateur, un titre et un texte.

L'ensemble des classes déjà développées se trouvent dans une archive à votre disposition. Ci-dessous, une description des classes déjà existantes (résumée dans le diagramme de classe ci-dessous) est fournie.



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.

D'un point de vue implémentation, un article correspond à un objet instance de la classe `Article`, permettant de le manipuler. Cette classe comporte 3 attributs de type `string` désignant respectivement l'identificateur, le titre et le texte de l'article. L'identificateur permet de distinguer de manière unique un objet `Article`. Les méthodes `getId()`, `getTitle()`, `getText()`, `setTitle()` et `setText()` permettent d'interagir avec les instances de cette classe. L'unique constructeur de cette classe a 3 paramètres de type `const string&` qui permettent d'initialiser les attributs d'un objet.

Dans l'application, l'ensemble des objets `Article` est géré par un module appelé `NotesManager` qui est responsable de leur création (et destruction) et de leur sauvegarde. L'attribut `articles` est un pointeur vers un tableau de pointeurs d'objets `Article` qui est alloué dynamiquement.

La méthode `getNewArticle` permet de créer un nouvel article dont l'identificateur (qui n'existe a priori pas encore) est transmis en argument. Le titre et le texte de cet article sont initialement vides. Si un client essaye de créer un article dont l'identificateur est déjà présent dans l'objet `NotesManager` qui appelle la méthode, une exception est déclenchée. Pour créer un nouvel objet `Article`, la méthode `getNewArticle` *alloue dynamiquement* un objet `Article`. L'adresse de cet objet est alors sauvegardée en appelant la méthode privée `addArticle`.

De plus, la méthode `getArticle` permet d'obtenir un objet `Article` correspondant à un identificateur déjà existant. S'il n'existe pas d'article ayant cet identificateur dans l'objet `NotesManager` auquel la méthode est appliquée, une exception est déclenchée.

La méthode `addArticle` sauvegarde une adresse fournie en argument dans un tableau de pointeurs d'objets `Article`. L'adresse de ce tableau est stockée dans l'attribut `articles` de type `Article**`.
L'attribut `nbArticles` représente le nombre d'adresses sauvegardées dans ce tableau. L'attribut `nbMaxArticles` représente le nombre maximum d'adresses qui peut être sauvegardé avant un agrandissement du tableau (*c.-à-d.* la taille du tableau pointé par `articles`). La méthode `addArticle` gère les éventuels besoins en agrandissement du tableau.

La classe `NotesManager` possède un unique constructeur sans argument. Notons qu'un objet `NotesManager` a la responsabilité des objets `Article` qu'il crée (création/destruction).

Question 1

Expliciter des intérêts de mettre en place le Design Pattern *Singleton* pour la classe `NotesManager`. 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 `Article` pourrait poser des problèmes. Mettre en place les instructions qui permettent d'empêcher la duplication d'un objet `Article`.

Question 3

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

```
NotesManager& m=NotesManager::getInstance();
/*...*/
for(NotesManager::Iterator it= m.getIterator();!it.isDone();it.next()){
 std::cout<<it.current()<<"\n";
 it.current().setText(""); // modification possible
}
//...
const NotesManager& mconst=NotesManager::getInstance();
for(NotesManager::ConstIterator it= mconst.getIterator();!it.isDone();it.next()){
 std::cout<<it.current()<<"\n"; //ok
 //it.current().setText(""); // modification impossible
}
```

Question 4

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

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

Question 5

Écrire une classe d'itérateur qui permet de parcourir séquentiellement les articles dont le texte contient une sous-chaine de caractère donnée en déduisant son implémentation du code suivant :

```
NotesManager& m=NotesManager::getInstance();
/*...*/
for(NotesManager::SearchIterator it= m.getSearchIterator("bidule");!it.isDone();it.
   next()){
 std::cout<<it.current()<<"\n";
}
```