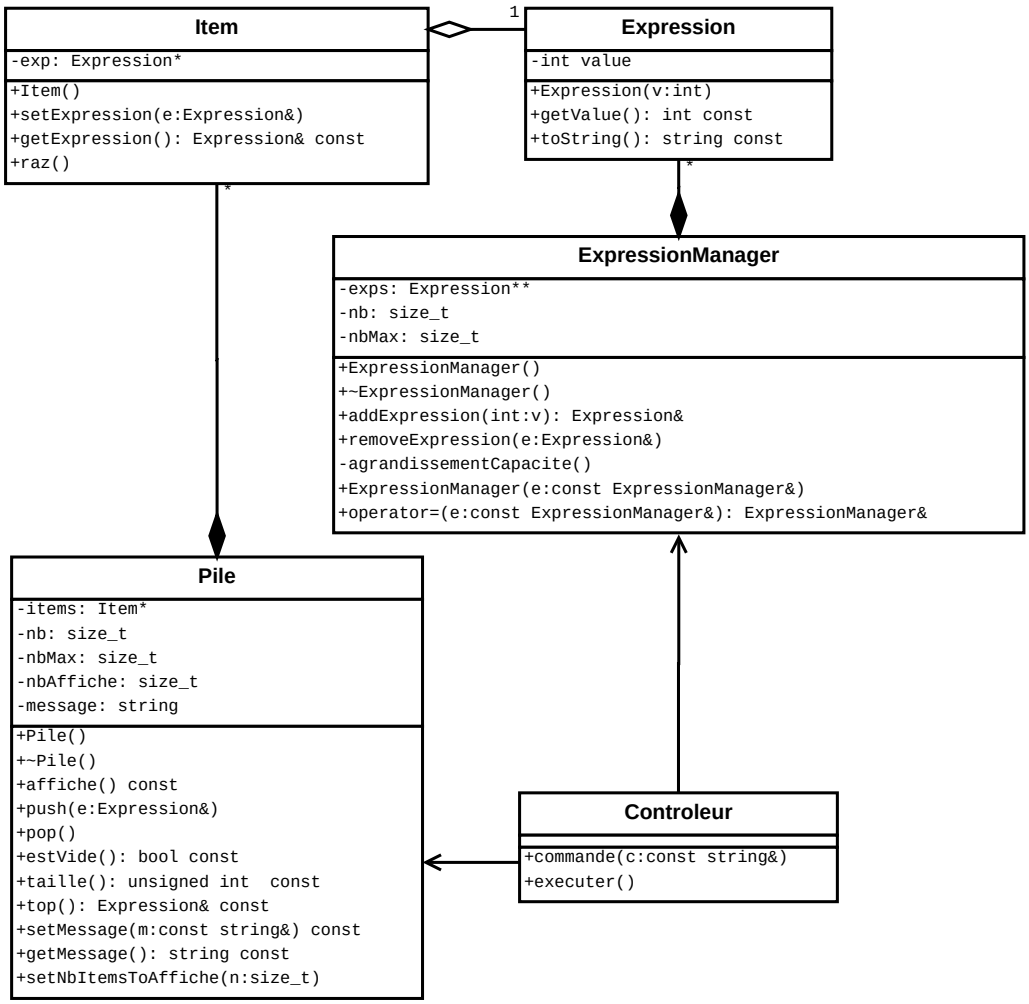
Exercice 24 - Design patterns

Nous avons commencé à développer des classes pour l'application Comp'UT. L'ensemble des classes est résumé dans le diagramme de classes ci-dessous :



Question 1

Expliciter des intérêts de mettre en place le Design Pattern *Singleton* pour la classe `ExpressionManager`. Etudier les différentes possibilités d'implémentation. Implémenter ce design pattern. Modifier votre code en conséquence. Mettre à jour le diagramme de classes.

Question 2

Mettre en place les instructions qui permettent d'empêcher la duplication d'un objet `Expression`. De plus, faire en sorte que seule l'unique instance de la classe `ExpressionManager` puisse créer des objets `Expression`.

Question 3

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

```
ExpressionManager& m=ExpressionManager::getInstance();
/*...*/
for(ExpressionManager::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), *c.-à-d.* qui permet de parcourir séquentiellement les différentes expressions d'un objet `ExpressionManager` avec le code suivant :

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

Ajouter une classe `const_iterator` afin d'offrir un parcours séquentiel uniquement en lecture en proposant une interface d'itérateur similaire à celle utilisée par les conteneurs standards du C++ (STL).

Question 5

Implémenter une classe d'iterateur qui permet de parcourir l'ensemble des expressions associées aux items d'un objet `Pile` dans l'ordre inverse où ils ont été empilés.