SEMAINE 8 : TP 1 - CLASSES ABSTRAITES

Dans cette séance, on voit l'utilité des classes abstraites et de l'héritage.
Nous allons modéliser des ensembles bornés d'entiers à l'aide de différentes représentations.

1ÈRE PARTIE

1 ENSEMBLE D'ENTIERS AVEC UN TABLEAU DE BOOLÉENS

La classe `EnsembleAvecBooleen` représente un ensemble d'entiers compris entre `0` et `MAX` à l'aide d'un tableau de booléens.

Un tableau `tab` de booléens représentera les entiers :

- `tab[i]` vaut `true` si et seulement si `i` appartient à l'ensemble,
- initialement, toutes les cases du tableau sont à `false`.

Les méthodes de cette classe seront :

- `EnsembleAvecBooleen(Integer max)` : crée l'ensemble vide ;
- `boolean add(Integer i)` : ajoute **si possible** l'élément i, retourne vrai dans ce cas et faux sinon ;
- `boolean isEmpty()` : retourne vrai si l'ensemble est vide, faux sinon ;
- `void clear()` : réinitialise l'ensemble ;
- `boolean remove(Integer i)` : supprime **si possible** l'élément i, retourne vrai dans ce cas et faux sinon ;
- `public String toString()` : retourne une chaîne représentant l'ensemble sous la forme: {7,25,33} pour un ensemble contenant les entiers 25, 33 et 7 (on veillera à avoir un affichage dans l'ordre croissant) ;
- `boolean contains(Integer i)` : retourne vrai si i appartient à l'ensemble, faux sinon ;
- `int size()` : retourne le nombre d'élément de l'ensemble.

Écrire la classe `EnsembleAvecBooleen.`

JUNIT

Nous allons utiliser JUnit sous eclipse pour tester votre classe.

Pour cela, créer dans votre projet un dossier tests (au même niveau que votre dossier src). Dans l'explorateur de paquets d'eclipse, faites un clic droit sur la classe **`EnsembleAvecBooleen`**.

1. Dans le menu contextuel, cliquer sur New JUnit Test Case.
2. Choisir le répertoire de sources tests et nommer la classe **EnsembleAvecBooleenTest**. Vérifier que la classe sous test est bien **`EnsembleAvecBooleen.`** Cliquer sur next, ne générer pas les signatures des méthodes de test (elles vous sont fournies) et créez la classe.
3. Eclipse va proposer d'ajouter la bibliothèque de JUnit dans le buildpath du projet, accepter.
4. Remplacer le contenu de la classe **`EnsembleAvecBooleenTest`** fraîchement créée par celui de classe **`EnsembleAvecBooleenTest`** fournie dans les ressources du TP sur Moodle.

Après chaque écriture de méthode, vous vérifierez que le test associé considère votre implémentation valide en faisant un clic droit sur la classe de test puis en choisissant Run As - JUnit test dans le menu contextuel.

2 ENSEMBLE D'ENTIERS AVEC UNE **`ARRAYLIST<INTEGER>`**

Le même type d'ensemble peut être représenté à l'aide d'une **`ArrayList`**, sans changer la signature des méthodes.

Ecrire la classe **`EnsembleAvecArrayList`** qui réalise à l'aide d'une **`ArrayList`** les mêmes fonctionnalités que la classe **`EnsembleAvecBooleen`**.

Note : Même s'il s'agit d'une **`ArrayList`**, il convient d'avoir un attribut **`MAX`** représentant la valeur maximale.

Vérifier votre travail en utilisant la classe de test fournie **`EnsembleAvecArrayListTest.`**

2ÈME PARTIE

BOUML

Lire la suite de l'énoncé intégralement et dessiner avec **Bouml** le diagramme de classes .

1 LA CLASSE ABSTRAITE ENSEMBLEMODIFIABLE

Quels sont les comportements communs aux deux classes?

Si vous avez bien suivi nos recommandations, trois méthodes peuvent être communes aux deux classes. Lesquelles ?

Ecrire une classe abstraite **EnsembleModifiable** qui permet de factoriser les comportements communs aux deux classes précédentes. Pensez au constructeur !

Note : Ces trois méthodes doivent être écrites de la manière la plus générique possible (indépendante des structures de données).

Modifier les classes **EnsembleAvecBooleen** et **EnsembleAvecArrayList** en tenant compte de cette nouvelle classe. Tester de nouveau ces deux classes.

2 LA CLASSE ABSTRAITE ENSEMBLE

On cherche à aller plus loin en terme de factorisation.

Écrire une classe **Ensemble** qui regroupe des ensembles modifiables (on peut leur ajouter ou leur retrancher des éléments) et des ensembles non modifiables. Dans cette classe abstraite, on ne garde que les méthodes **toString()**, **size()** et **isEmpty()**, le constructeur, et la spécification de la méthode **contains(Integer i)**.

Écrire la classe abstraite **Ensemble**. Modifier en conséquence la classe **EnsembleModifiable** et vérifier que les tests passent toujours. Note : Cette classe ne contient plus que le constructeur et trois spécifications de méthodes.

3 SINGLETON

Un singleton est un ensemble contenant un seul élément, et cet élément n'est pas modifiable.

Par exemple : {5} est un singleton.

Écrire la classe **`EnsembleSingleton`** qui hérite de **`Ensemble`** et ne contient qu'un élément. L'élément contenu sera fourni à la construction, on initialisera MAX avec une valeur quelconque plus grande. Tester votre classe avec la classe **`EnsembleSingletonTest`** fournie.

4 UNION DE DEUX ENSEMBLES

On peut réaliser l'union de singletons pour créer des ensembles de plus grande taille.

Ce principe est généralisable en considérant des ensembles d'ensembles.

On veut implémenter la classe **`EnsembleUnion`** qui contient deux **`Ensemble`** (qui peuvent être de n'importe quel type dérivé de **`Ensemble`**). Cette classe est évidemment un **`Ensemble`** !

Ces ensembles ne sont pas modifiables par l'intermédiaire de **`EnsembleUnion`**.

Écrire la classe **`EnsembleUnion`**. Tester la à l'aide de la méthode **`EnsembleUnionTest`** fournie.

5 LA MÉTHODE `EQUALS`

Deux ensembles sont égaux s'ils possèdent les mêmes éléments.

Ajouter la(les) méthode(s) **`boolean equals(Object o)`** dans la(les) classe(s) adéquate(s).

6 MODIFICATION DE L'AFFICHAGE

On souhaite désormais afficher les ensembles avec un seul élément par une ligne.

Modifier la ou les méthodes concernée(s).

Combien de classes ont été modifiées ?