Programmation Orientée Objet

TD 2 – Implémentation Java

Ce TD est dédié à l'implémentation des classes modélisées dans le TD précédent. L'implémentation sera faite à l'aide de l'IDE Eclipse que vous trouverez au chemin /ark/Logiciels/linux64/eclipse64/

Exercice 1 : Résolution d'équation - Implémentation d'une classe simple

Il s'agit ici de réaliser le moteur simplifié de résolution d'équations de degré 1 vu au TD précédent dont la modélisation UML est donnée dans la figure 1.

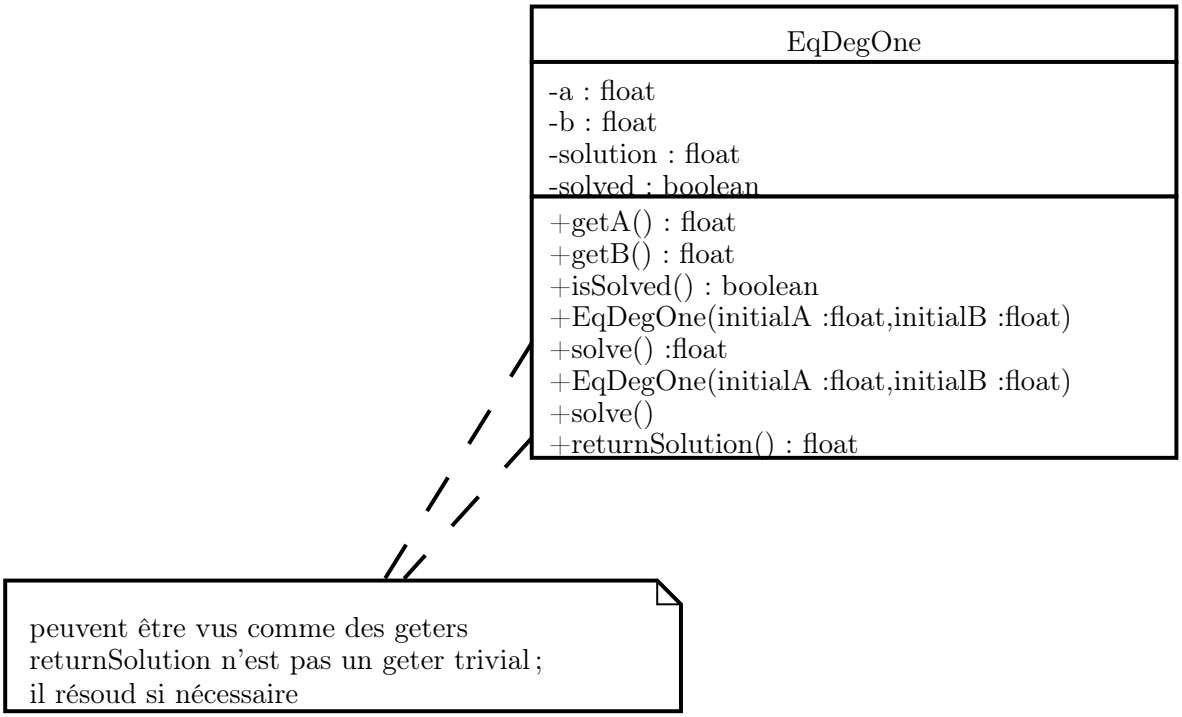

FIGURE 1 – Classe `EqDegreeUn`

On rappelle qu'une telle classe représente une équation de type $(ax + b = 0)$ où a et b sont des coefficients réels. L'état "résolu ou non" est stocké explicitement sous la forme d'un attribut.

a : Écrire un classe Java mettant en œuvre cette classe : définissez et implémentez ses attributs, méthodes et constructeur(s).

b : Définir pour chacun des éléments les commentaires Javadoc spécifiques. Les principaux marqueurs javadoc sont décrits dans l'encadré suivant :

```
/**
* Description courte de la méthode/attribut
* @param nomParam description du premier paramètre (si méthode avec au moins 1 param)
* @param nomParam2 description second paramètre
* @returns description du retour de la méthode (si méthode et si retour)
*/
```

c : Générer et consultez la documentation javadoc ; sur eclipse onglet *project*, *generate javadoc*.

d : Implémenter une classe `TestEqDegOne` testant la classe précédemment codée ; créer une classe `TestEqDegOne` avec un main instanciant un objet de type `EqDegOne`, le manipulant et vérifiant que son comportement est celui attendu.

e : Importer le jar aTester.jar dans votre projet (sous eclipse, Build Path>Configure Build Path>add external Jar). Ce JAR contient un package aTester comportant une classe `FausseEqDegreeUn`. Cette dernière possède les mêmes attributs et méthodes que `EqDegreeUn` ; adapter (sans ajouter de test effectif) `TestEqDegreeUn` afin de la tester. Combien d'erreurs trouvez-vous ?

Exercice 2 : Coffre fort – Quelques classes associées

Il s'agit ici d'implémenter les `Gemstone`s et `Safe`s vus au TD précédent dont la modélisation UML est donnée dans la figure 2. D'autres choix de conception auraient pu être fait, cette figure propose une solution qui n'est donc pas nécessairement unique. On s'attachera néanmoins à ce que l'implémentation colle au plus près à cette conception.

a : Dans deux packages à part (safes et valuables, par exemple), définir les classes `Gemstone` et `Safe`, leurs constructeurs et attributs, sans pour le moment s'intéresser aux méthodes. Définissez pour chacun des éléments ses commentaires javadoc. Pour le contenu du coffre, on utilisera une liste de gemmes de type *ArrayList<Gemstone>*.

b : Implémenter les getters et setters pertinents. Définissez pour chacun des éléments ses commentaires javadoc. Notons que nous ne savons pas, pour le moment, récupérer intelligement la valeur d'un coffre ! Il nous faudrait parcourir la liste "contenu" pour calculer cette valeur...contentons nous donc de bêtement renvoyer la valeur sans la calculer.

c : Implémenter le reste des méthodes.

— Lors de l'expertise d'une `Gemstone`, on pourra dans un premier temps considérer sa valeur fonction de son volume, même si ce n'est pas très satisfaisant !

— À chaque fois qu'un cas problématique apparaît (e.g., ouverture d'un coffre ouvert, tentative de retrait d'une pierre non présente dans le coffre...) affichez simplement un message et arrêtez là la procédure. On pourra utiliser *System.err*.

Définissez pour chacun des éléments ses commentaires javadoc.

d : Implémenter une classe `TestSafe` testant les classes précédemment codées ; créer une classe `TestSafe` dans un package "test" avec un main instanciant des objets de type `Gemstone` et `Safe`, les manipulant et vérifiant que les comportements sont ceux attendus.

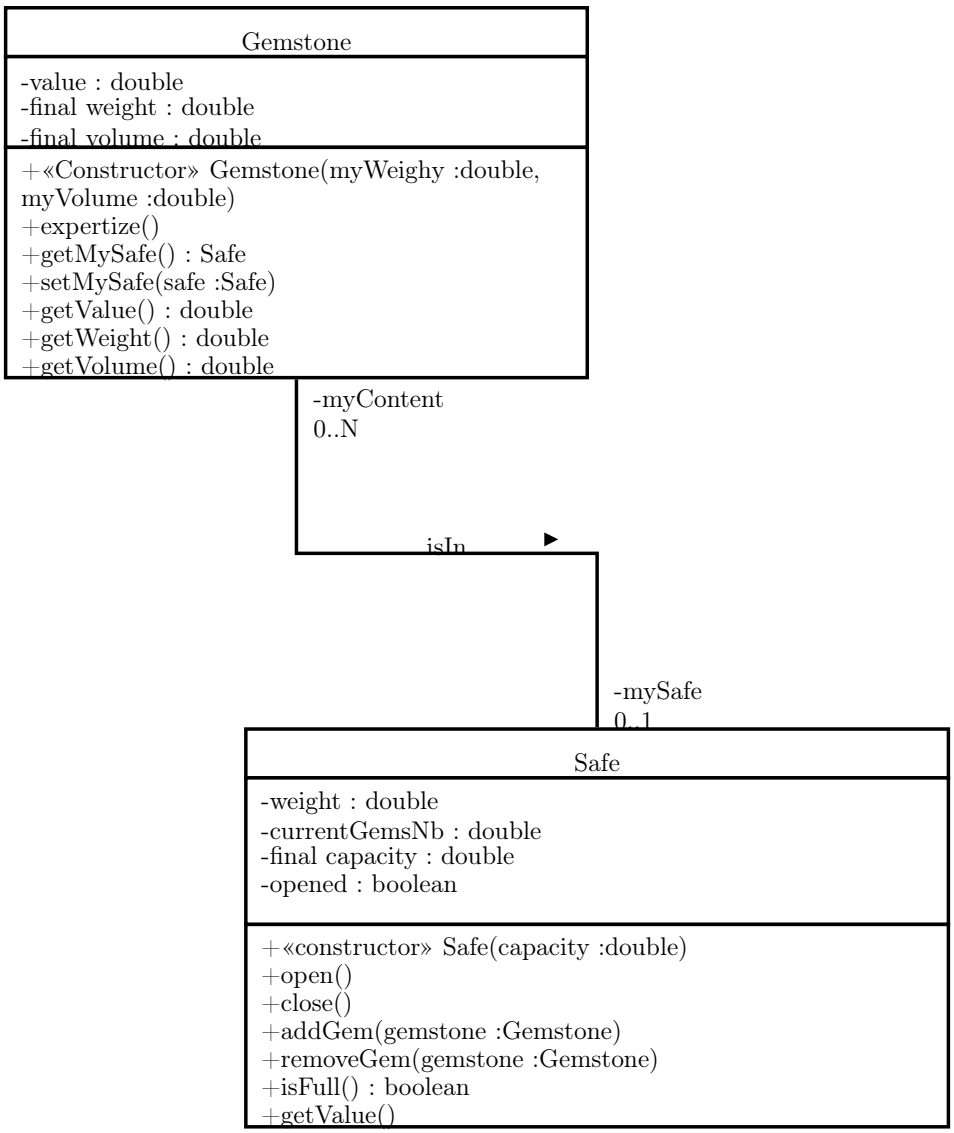

FIGURE 2 – Safe et Gemstone