

## Programmation Orientée Objet

### TD 4 – Généricité, interfaces.

Durant ce TD, nous allons reprendre les coffres et équations de degré un implémentés durant le CM du 30/11/2017 et le TD2, respectivement. Les codes de départ de Coffre (Safe) et Equation sont disponibles sur Celene.

Vous rédigerez un compte rendu de TD dans lequel vous résumez les concepts abordés dans chaque exercice et répondez aux questions posées.

## 1 Coffres et Pierres

On souhaite ajouter des fonctionnalités aux coffres vus au TD2 et complétés en CM. Toute modification doit être testée et documentée.

### Exercice 1 : Initialisation

**a :** Récupérer le code "InitialSafe.zip" disponible sur Celene.

**b :** Générez un diagramme UML à l'aide d'eUML2. Le projet récupéré contient trois packages (tests, safe et valuables) et 7 classes :

- **Gemstone**, une classe abstraite représentant une pierre précieuse. La méthode abstraite *expertize()* est implémentée dans trois spécialisations :
  - **Emerald**
  - **Saphir**
  - **Topaze**
- **Safe**, une classe représentant un coffre simple et possédant une collection de **Gemstone**.
- **SafeWithCode**, une spécialisation de **Safe** où l'ouverture est protégée par un code (entier).
- **FirstTest**, une classe de test.

**c :** Lancez la classe de test.

### Exercice 2 : Parcours de collection

**a :** [*fait en cours!*, mais il faut bien comprendre ce qui se passe] Dans **Safe**, implémentez *getValue()* : après avoir initialisé la valeur du coffre à 0, on parcourra l'intégralité des pierres contenues dans le coffre en ajoutant chacune de leur valeur à celle du coffre.

**b :** Dans **SafeWithCode**, la méthode *getValue()* ne pourra donner le bon résultat que si le coffre est ouvert et 0 sinon. On évitera d'écrire du code inutile (car redondant!).

**c :** Implémentez à présent la recherche d'une **Gemstone** d'une valeur spécifique minimale passée en argument. Sans utiliser d'instruction *break* ou de **return** intempestif, on veillera à arrêter la recherche dès qu'une **Gemstone** acceptable aura été trouvée.

### Exercice 3 : Coffres génériques

On souhaite pouvoir stocker tout type d'objet précieux dans un **Safe** plutôt que de se limiter à des pierres. On considérera pour simplifier que tous les objets stockés dans un coffre sont de même type.

**a :** Créez un nouveau package (**genericSafes**) et copiez-y la classe **Safe** ; La renommer (Refactor/Rename) en **GenericSafe**.

**b :** Déclarez cette classe comme générique en lui ajoutant un type variable paramétré.

**c :** Remplacez toute référence à `Gemstone` par ce type paramétré.

On obtient une erreur ! "The method `getValue()` is undefined for the type `T`". Effectivement, on ne peut pas stocker n'importe quel type d'objet ; il nous faut stocker des objets "précieux" (ou en tout cas pouvant renseigner leur valeur) !

**d :** Définir la(es) méthode(s) devant être proposée(s) par les objets stockés dans le coffre au sein d'une interface.

**e :** Au sein de `Safe`, ajoutez une contrainte signalant que le type paramétré doit implémenter l'interface précédemment définie.

**f :** Rattachez `Gemstone` à cette interface. Au sein de la classe de test, reproduisez les tests à l'aide d'un Coffre générique contenant des Pierres.

**g :** Créez un coffre qui pourra contenir des parapluies, objets très précieux !

**h :** Les Coffres ont eux-mêmes une valeur ! Rattachez `Coffre` à l'interface précédente, puis appliquez les tests précédents à un coffre de coffres de pierres.

**i :** Comment faire un coffre à code générique ?

## 2 Équations génériques

Reprenons à présent la classe `OneDegreeEquation` représentant une équation de type  $aX + b = 0$  avec  $a$  et  $b$  float. Nous allons construire une équation générique permettant de traiter tout type numérique. Tout au long de cet exercice,  $T_A$  et  $T_B$  désigneront le type de  $a$  et  $b$ , respectivement.

### Exercice 4 : Des types numériques simples pour commencer.

Dans un premier temps, on considérera des classes numériques simples de sorte que  $T_A = T_B$ .

**a :** Quel est le type de  $X$  ?

**b :** Quelles opérations doit-on pouvoir effectuer sur  $a$  et  $b$  ? Définir une interface listant ces opérations. Cette interface pourra être elle-même générique si nécessaire.

**c :** Créer une classe `EqDegreUnGenSimple` générique manipulant des objets implémentant l'interface précédemment définie.

**d :** Créer deux classes `Rationnel` et `Complex` implémentant l'interface définie précédemment.

**e :** Tester vos classes `Rationnel` et `Complex`. Tester `EqDegreUnGenSimple` à l'aide d'objets de types `Rationnel` et `Complex`.

### Exercice 5 : Pour aller plus loin. Attention : exercice difficile.

On considère à présent le cas général où  $T_A \neq T_B$ , mais où les opérations opposées et inversion sont stables pour leur domaine de définition.

**a :** Créer des interfaces devant être implémentées dans ce nouveau cas, symbolisant :

- Les opérations propres au type  $T_A$ .
- Les opérations propres au type  $T_B$ .
- Les opérations impliquant  $T_A$  et  $T_B$ .

**b :** Quel est le type de  $X$  ? On pourra dans un premier temps et pour simplifier faire l'hypothèse  $T_X = T_B$ .

**c :** Créer une classe `EqDegreUnGen` générique manipulant des objets de types  $T_A$  et  $T_B$  implémentant les interfaces précédemment définies.

**d :** Cette nouvelle représentation doit être aussi puissante que la précédente, faire tourner un exemple avec les complexes. (i.e.,  $T_A = T_B = \mathbb{C}$ )

**e :** On souhaite résoudre grâce à **EqDegreUnGen** une équation avec  $T_A = \mathbb{Q}$  et  $T_B = \mathbb{C}$ . Implémenter le nécessaire, tester.

**f :** Gérez le cas le plus général où  $T_X \neq T_B$ .