TP de PPO en JAVA

Polytech Lille GIS2A3

Objectifs : hiérarchie de classes, héritage et redéfinition de méthode, typage statique et dynamique, modularité.

1 Hiérarchie de classes de compte vue en cours

Créer un répertoire ‘‘`tp2`’’. Travailler dans un répertoire ‘‘`comptes`’’ qui se situe dans le répertoire que vous venez de créer.

Programmer la hiérarchie de classes `Compte` et `CompteEpargne` vue en cours :

- `Compte : crediter(double x), debiter(double x), solde(), toString()` (pour afficher credit, debit et solde), un constructeur sans paramètre et un contructeur paramétré par un crédit initial.
- sous-classe `CompteEpargne` :
  - `interets()` (qui calcule les intérêts) et `echeance() (qui les crédite sur le compte)`
  - redéfinition de la méthode `debiter()` (pas de solde negatif permis) et de `toString()` (pour afficher en plus les intérêts)
  - un constructeur paramétré par un crédit initial et un taux d'intérêts
  - un constructeur sans paramètre qui initialise le taux d'intérêts à 0.1.

Dans une classe `Banque`, programmer les méthodes suivantes :

- `crediter(Compte c, double valeur)` qui crédite le compte `c` du montant `valeur`.
- `debiter(Compte c, double valeur)` qui débite le compte `c` du montant `valeur`.
- `etat(Compte c` qui retourne la chaîne de caractères correspondant à l'appel de la méthode `toString()` sur le compte `c`.

On ne s'intéressera pas ici à la façon dont les comptes sont stockés dans la banque. La banque sert donc juste ici de moyen d'accéder à un compte et d'agir dessus.

2 Héritage et typage(s)

Ecrire un programme de test dans une classe principale `TestComptes` (contenant une méthode `main`) qui manipule 2 variables : `unCompte` et `unCE` respectivement de type `Compte` et `CompteEpargne`.

Typage statique

Vérifier (à la compilation) que :

- les méthodes de `Compte` sont applicables sur les 2 variables (héritage)
- les méthodes de `CompteEpargne` ne sont applicables que sur `unCE`
- l'affectation : ‘‘`unCompte = unCE;`’’ est valide (sous-typage) et non l'inverse : ‘‘`unCE = unCompte;`’’

Redéfinition de méthode

Créer une instance de `Compte` dans `unCompte` et une instance de `CompteEpargne` dans `unCE`. Vérifier que les méthodes redéfinies `debiter()` et `toString()` s'exécutent différemment selon qu'elles sont appliquées à `unCompte` ou à `unCE`.

Typage dynamique

Dans le `main` tester la portion de code suivante pour vérifier la liaison dynamique des méthodes `debiter()` et `toString()` dans `debiter(c, v)` en fonction du type dynamique de `c` :

```
Banque b = new Banque();
b.crediter(unCompte, 10); // => type dynamique de c = Compte
b.debiter(unCompte, 235); // ou 235 > solde
System.out.println(b.etat(unCompte));

b.crediter(unCE, 10); // => type dynamique de c = CompteEpargne
b.debiter(unCE, 235); // ou 235 > solde
System.out.println(b.etat(unCE));

unCompte = unCE ;

b.crediter(unCompte, 10); // => type dynamique de c =?
b.debiter(unCompte, 235); // ou 235 > solde
System.out.println(b.etat(unCompte));
```

3 Opérations historisées

Créer un répertoire **''operations_historisees''** et y programmer une nouvelle version de la classe `Compte` comme suit. Au lieu de cumuler les montants crédités et débités dans les variables `credit/debit`, l'historique de ces montants est mémorisé. Les variables d'instance `credit/debit` ne sont plus nécessaires et sont remplacées par deux tableaux de `double` de taille `MAX_OPERATIONS` (constante à définir dans la classe `Compte`) `credits` et `debits`, munis de leur indice respectif `dernierCredit` et `dernierDebit`, sur le dernier montant crédité/débité. Les opérations deviennent :

- `crediter(double x)` : range x en fin du tableau `credits`, quand `MAX_OPERATIONS` est atteint, le tableau est réinitialisé avec le cumul des crédits dans `credits[0]`
- `debiter(double x)` : range x en fin du tableau `debits`, quand `MAX_OPERATIONS` est atteint, le tableau est réinitialisé avec le cumul des débits dans `debits[0]`
- `solde()` = $\sum credits[i] - \sum debits[j]$
- `toString()` affiche les historiques `credits` et `debits` et le solde.

4 Modularité

Le protocole de la classe `Compte` n'a pas changé, seule son implantation interne a été modifiée. Les autres classes `CompteEpargne` et `TestComptes` n'ont donc pas à être recompilées. Vérifier cela en copiant simplement leur `.class` (et non leur source) du répertoire **''comptes''** dans **''operations_historisees''** et ré-exécuter directement `TestComptes`.

5 Tester votre travail

Afin de savoir si votre code est de qualité et correspond à ce qui vous est demandé, il vous est possible de le tester. Pour cela, copier sur votre compte dans le répertoire `tp2` le répertoire `test` :

`cp -r ~aetien/public/PPO/tp2/test .`

Mettez vous dans votre répertoire `tp2/test` et exécutez la commande suivante pour tester la première version de compte :

`./runTest.sh`

Exécutez la commande suivante pour tester la deuxième version de compte avec les historiques :

`./runHistoriqueTest.sh`

Si votre code vérifie tous les critères de qualité évalués, vous aurez un message du genre `OK (18 tests)` pour la partie sans historique et `OK (23 tests)`, pour la partie avec. Sinon, vous aurez des messages vous indiquant les erreurs qui peuvent exister dans votre code.