TP5: Comptes bancaires
Algorithmique – Programmation FIP (ING39)

V. Aponte, P. Courtieu

Septembre 2016

Le but de ce TP est de travailler EN BINÔMES (testeur + programmeur), l'écriture de contrats, des tests et le code sur objets.

Préparation de l'environnement

Vous allez travailler en binôme. L'un de vous deux jouera le rôle du *testeur* pendant un certain temps, l'autre celui de *développeur*. Il est recommandé de changer les rôles au moins une fois pendant la durée duTp. Au départ, **le plus expérimenté en java**, prendra le rôle du *testeur* et accessoirement, pourra aider le *programmeur* s'il a des difficultés.

L'un de vous deux (**et seulement l'un de vous**) va créer sur gitlab un projet COMMUN de nom `TpLivretA` et va donner les droits d'accès à l'autre. Pendant toute la durée du tp vous allez travailler sur ce projet commun.

Ce qu'on attend de vous

Vous aurez besoin du squelette de l'interface `LivretA` sur le site du cours.

1. Complétez les contrats de méthodes manquant ou incomplets (voir explications plus bas) dans cette interface, en respectant le format javadoc. La rédaction des contrats doit se faire à deux: vous devez être d'accord sur ce qui est attendu pour chaque méthode. N'écrivez rien d'autre avant de finir de rédiger tous les contrats, commitez et poussez.
2. Le testeur écrira un jeu de tests pour les objets de la classe en suivant les contrats de l'interface.
3. Le développeur écrira une classe `CompteLivretA` qui implante cette interface.
4. Au départ, le MOINS expérimenté en Java prendra le rôle de dévelopeur. Vous changerez les rôles par la suite.
5. Dès que possible, le *développeur* utilisera les tests sur la forge pour tester chacune des méthodes qu'il doit écrire.
6. Le *développeur* ne passera pas à la prochaine méthode à écrire tant que tous les tests sur la forge n'ont pas réussi pour la méthode courante, **ainsi que pour toutes les méthodes testées auparavant** (*non regression*).

Implantation et test de la classe CompteLivretA

Les comptes bancaires sur Livret A possèdent un nom de titulaire (String), un numéro de compte, un solde courant, et un état du compte: ouvert ou fermé. A aucun moment, le solde d'un compte ne doit être négatif ni dépasser un plafond autorisé par la loi. Pour simplifier, nous ne considérons pas les intérêts à abonner périodiquement sur ce type de compte. On se donne les contraintes suivantes: un compte peut être fermé uniquement s'il est ouvert et si son solde est égal à zéro. Un compte fermé n'admet que les opérations ne modifiant pas son état interne ainsi que l'opération

qui permet son ouverture. Les opérations `ouvrir()` sur un compte déjà ouvert, et `fermer()` sur un compte déjà fermé, n'échouent pas, mais n'ont aucun effet sur le compte, et par conséquent, renvoient `false`. Les opérations que l'on souhaite implanter sont décrites par l'interface sur le site du cours (dont voici un extrait):

```
public interface LivretA {
  /**
     * Invariant (contrat) d'état interne:
     *     <pre>
     *          this.getPlafond() > 0
     *          &&  si this.estOuvert()     alors 0 <= this.getSolde()<= this.getPlafond()
     *          &&  si !(this.estOuvert())  alors this.getSolde() = 0
     *     </pre>
     */

    /**
     * Retourne la valeur maximale autorisée pour ce compte.
     * <p> post-condition: this.getPlafond() > 0 </p>
     * @return double strictement positif.
     */
    ....
```

Contrats

Complétez les contrat (javadoc) des méthodes `fermer` et `retrait` dans l'interface `LivretA`.

Implantation

Ecrivez une classe `CompteLivretA` qui implante cette interface, et qui déclare un ou plusieurs constructeurs pour initialiser les données d'un compte. Un compte qui vient d'être crée doit être dans un état qui valide l'invariant d'état interne.

- Le constructeur pour initialiser les variables internes, y compris le plafond maximal du livret.
- Le constructeur doit créer des objets suelement si leur état interne est cohérent (c.a.d, valide l'invariant d'état). Autrement, le constructeur doit échouer.
- Il peut être utile de définir deux constructeurs: un qui crée un compte en état ouvert et, et un autre qui crée un compte en état fermé. Le premier prendra en argument tous les paramètres (sauf l'état, qui sera mis à true); le deuxième ne prendra en paramètre ni l'état (mis à false), ni le solde, qui dans le cas d'un compte fermé doit être nécessairement nul.

Toutes les variables de votre classe seront privées.

Méthode pour tester l'invariant

Ajoutez dans cette classe (et non pas dans la classe de test!!) une méthode `public boolean invariantOK()` permettant de déterminer si l'état interne d'un objet est cohérent (valide l'invariant). Cette méthode auxiliaire servira à réaliser les tests. Elle est placée dans la classe qui implante l'objet pour des raisons de visibilité: elle doit pouvoir accèder aux variables de son état interne. Ainsi, cette méthode sert uniquement aux tests, et donc, seules les méthodes des test (depuis le paquetage de tets) ont le droit de l'invoquer. **Cette méthode ne doit pas être appelée** à partir des méthodes de l'objet.

Tests

Elaborez un jeu de tests d'après la spécification et testez votre implantation. Pour rappel, chaque cas de test Junit doit:

1. Déclarer et créer un nouvel objet c (via un des constructeurs): vous ferez en sorte de lui passer en paramètre les valeurs qui conviennent au cas que vous voulez tester:
   - Pour tester un comportément sans échec: donnez des paramètres qui ne produisent pas d'échec **et** qui satisfont la pré-condition. L'appel doit se faire sur un objet qui **est dans un état cohérent avant l'appel**.
   - Pour tester un comportément avec échec: donnez les paramètres qui devraient le déclencher (avec la clause `expected`) appropriée.
2. Invoquer la méthode à tester, et selon qu'elle retourne ou pas un résultat, vous devez, soit comparer le résultat obtenu avec le résultat attendu, soit tester que les variables internes de l'objet valident la post-condition. En dehors des *accesseurs*, **n'invoquez pas d'autres méthodes** que cella à tester. On veut se concentrer sur une méthode à tester, et de ne pas faire dépendre ce test de comportements externes.
3. Si la méthode testée modifie l'état interne de l'objet, il peut être necéssaire de tester que cet état reste cohérent **après l'appel**. Vous pourrez le faire en invoquant la méthode `invariantOK()`. Ce sera la seule méthode différente d'un accesseur et de celle à tester que l'on s'autorise à invoquer lors d'un test.
4. De manière générale, **on ne teste pas les accesseurs**. On suppose qu'ils sont suffisamment simples pour être trivialement corrects.

Nous vous conseillons d'implanter et de tester les méthodes une à une. Commencez par tester le constructeur qui doit toujours créer des objets qui valident l'invariant. Si les paramètres d'un appel aux constructeurs ne correspondent pas à une initialisation valide, testez que le constructeur se termine par un échec.