ATELIER 2 : TEST DYNAMIQUE - TESTS UNITAIRES AVEC MOCKITO

ENSEIGNANTE : MME. MANEL BEN SALAH

Exercice 1 : Gestion des produits

Question 1

Implémenter un projet représentant un Produit dans un système de gestion de stock simplifié.

1. **Entité Product** : Une classe représentant un produit avec les attributs suivants :
   - productID : un entier représentant l'identifiant du produit.
   - productName : une chaîne de caractères représentant le nom du produit.

2. **Interface ProductDao** : Une interface pour accéder aux produits dans une base de données ou une source de données fictive. Elle contiendra :
   - Product fetchProduct(Integer productID) : méthode pour récupérer un produit par son ID.
   - void update(Product product) : méthode pour mettre à jour un produit.

3. **Service ProductService** : Une classe de service contenant la logique métier pour :
   - Mettre à jour le nom d'un produit si l'ID existe.
   - Retourner true si la mise à jour a réussi, ou false si le produit n'a pas été trouvé.

Question 2

Implémenter les tests unitaires de la classe **ProductService** en simulant les interactions avec **ProductDao**.

Exercice 2 : Gestion des comptes bancaires

On souhaite simuler un système de gestion de comptes bancaires. L'objectif est de tester la classe BankService en utilisant des mocks pour isoler la logique métier.

1. **Classe Compte :**
   - Représente un compte bancaire avec :
     - compteID : l'identifiant du compte.
     - nomC : nom du client possédant le compte
     - solde : le solde du compte.

2. **Interface CompteDao :**
   - Permet d'interagir avec une base de données fictive.
   - Contient deux méthodes :
     - Compte chercherCompte (Integer compteID) : pour rechercher un compte par son identifiant.

- Void updateCompte (Compte compte) : pour mettre à jour les informations d'un compte.

3. **Classe BanqueService :**
   - Une méthode principale retirer pour effectuer un retrait :
     - Reçoit l'ID du compte et le montant à retirer.
     - Vérifie si le solde est suffisant pour le retrait.
     - Si oui, met à jour le solde et retourne true.
     - Si non, retourne false.
   - Une méthode principale verser pour effectuer un virement :
     - Reçoit l'ID du compte et le montant à verser.
     - Vérifie si le solde est suffisant pour le versement.
     - Si oui, met à jour le solde du 2éme compte et retourne true.
     - Si non, retourne false.
   - Une méthode principale verserVers pour effectuer un virement entre deux comptes donnés en paramètre :
     - Reçoit les ID des deux comptes et le montant à verser.
     - Vérifie si le solde est suffisant pour le retrait.
     - Si oui, met à jour le solde du 2éme compte et retourne true.
     - Si non, retourne false.
4. **Test BanqueServiceTest :**
   - Utilisez Mockito pour simuler l'interface CompteDao.
   - Écrivez des tests pour valider la logique métier de la classe BanqueService.

Annexe

Création d'un mock

- Pour créer un mock, il faut utiliser la méthode statique mock ou l'annotation @mock
- Il faut indiquer la classe à mocker ou son interface
- Mockito, crée moi une instance de mock pour cette classe ou cette interface.

```
1  import static org.mockito.Mockito.*;
2  ...
3   C mock1 =mock(C.class); // C est une classe ou une interface
4   C mock2=mock(C.class , "nom");
5   @Mock C mock3;
6   @Mock(name="nom2") C mock4;
```

Pour utiliser les annotations mockito :
Ajouter @RunWith(MockitoJUnitRunner.class) sur la classe de test.
On ne peut pas utiliser l'annotation @Mock sur une variable locale (d'une méthode de test), seulement sur un attribut (normal, c'est une annotation)

Paramétrage d'un mock

- On décrit le comportement du mock avec la méthode méthode when
    - On exprime quelque chose du genre : Mockito, quand (when !) le mock recevra tel appel, alors il faut répondre ceci
- On peut faire des vérifications comportementales avec la méthode verify
    - On vérifie quelque chose du genre : Mockito, est-ce que telle méthode a bien été appelée au moins une fois avec tel paramètre ?
- On peut spécifier des comportements à vérifier un peu complexes grâce à des matchers

Spécification du comportement du mock : méthodes avec retour

Cas d'une méthode avec retour ; valeurs successives

```
1  interface I{ int m();} / the interface to mock
2  ...
3  @Mock I myMock; // the mock
4  ...
5  when(myMock.m()).thenReturn(42, 43, 44);
```

▶ Mockito, quand myMock recevra un appel à la méthode m, retourne 42 au premier appel, puis 43 au deuxième appel, puis 44 au 3ème appel.

Spécification du comportement du mock, méthode avec paramètres

Spécifier le comportement selon la valeur reçue en paramètre

```
1  interface I{ int m(int i);} // the interface to mock
2  ...
3  @Mock I myMock; // the mock
4  ...
5  when(myMock.m(1)).thenReturn(42);
6  when(myMock.m(42)).thenReturn(1);
```

▶Mockito, quand myMock recevra l'appel à la méthode m avec pour paramètre 1, retourne 42, et avec comme paramètre 42, retourne 1.

Spéc. du comportement du mock pour lever des exceptions

Cas d'une méthode avec retour avec levée d'exception

```
1  interface I{ int m() throws E;} // the interface to mock
2  ...
3  @Mock I myMock;
4  ...
5  when(myMock.m()).thenThrow(new E());
```

▶Mockito, quand myMock reçoit un appel à m, jette une exception de type E

Cas d'une méthode sans retour avec levée d'exception

```
1  interface I{ void m(int i) throws E;} // the interface to mock
2  ...
3  @Mock I myMock; // the mock
4  ...
5  doThrow(New E()).when(myMock).m(1);
```

▶Mockito, quand myMock reçoit un appel à m avec pour paramètre 1, jette une exception de type E

Vérification du comportement : verify

Vérifier qu'une méthode est appelée 3 fois

```
1  verify(mock1, times(3)).m();
```

Vérifier qu'une méthode est appelée au moins/au plus 3 fois

```
1  verify(mock1, atLeastOnce()).m();
2     verify(mock1, atMost(3)).m();
```

Vérifier qu'une méthode n'est jamais appelée

```
1  verify(mock1, never()).m();
```