Tests Unitaires TP2

Mockups

Objectifs

Cette suite de TP a pour objectifs de vous apprendre à:

- Comprendre l'importance des tests unitaires dans le cycle de développement logiciel.
- Apprendre à utiliser les mockups pour isoler les composants lors des tests.
- Maîtriser l'utilisation de JUnit5 et Mockito pour la création de tests unitaires et de mockups.
- Faire la différence entre un test d´état et un test des interactions.

Prérequis

Pour réaliser cette suite de TP, vous devez avoir:

- Une version de Java 8 ou supérieure
- Un IDE Intellij IDEA
- Une connexion internet
- Les premiers manuels de TP sur les tests unitaires avec JUnit 5

Les mockups

Les mockups sont des objets qui simulent le comportement d'objets réels de manière contrôlée. Ils sont particulièrement utiles dans les tests unitaires pour les raisons suivantes:

- **Isolation**: Les mockups permettent de tester une partie du code en isolation, sans dépendre d'autres parties du code qui peuvent être coûteuses à mettre en place ou qui peuvent introduire des erreurs.
- **Contrôle**: Avec les mockups, vous avez un contrôle total sur le comportement de l'objet. Vous pouvez définir exactement comment l'objet doit se comporter pour le scénario que vous testez.

Un mockup est une instance d'une classe qui a le même type d'interface que l'objet réel, mais dont le comportement est contrôlé par le testeur. En d'autres termes, un mockup est un faux objet utilisé pour simuler le comportement d'un vrai objet dans un test unitaire.

La bibliothèque Mockito

Mockito est une bibliothèque populaire en Java pour la création de mockups. Elle fournit une API simple pour créer, configurer et utiliser des mockups dans les tests unitaires.

Installation de Mockito

Pour installer Mockito, vous devez ajouter la dépendance Mockito à votre fichier de gestion de dépendances. Voici comment vous pouvez le faire pour Maven:

```
<dependencies>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>4.2.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Importation et configuration de Mockito

Après avoir installé Mockito, vous pouvez l'importer dans vos fichiers de test avec l'instruction d'importation suivante:

```
import static org.mockito.Mockito.*;
```

Deux manières sont possibles pour intégrer Mockito dans les tests Junit :

1- Ajouter l'annotation `@RunWith(MockitoJunitRunner.class)` à la classe de test :

```
@RunWith(MockitoJunitRunner.class)
public class MyTestClass {
}
```

2- Faire appel à la méthode initMocks dans la méthode de SetUp :

```
@Before
public void setUp() {
    MockitoAnnotations.initMocks(this);
}
```

Création d'un mock

La création d'objets mockés se fait soit en appelant la méthode mock(), soit en rajoutant l'annotation @Mock pour les instances de classes.

```
User user = Mockito.mock(User.class);
```

Ou bien

```
@Mock
User user;
```

Mockito encapsule et contrôle tous les appels effectués sur l'objet User. Ainsi user.getLogin() retournera tout le temps null si on ne « stubb » pas la méthode getLogin().

Un autre exemple : pour créer un mock d'une interface `MonInterface`, vous pouvez faire comme suit:

```
MonInterface maMock = mock(MaInterface.class);
```

Stubbing

Le stubbing vous permet de définir le comportement de votre mock. Par exemple, vous pouvez configurer votre mock pour qu'il renvoie une certaine valeur lorsque l'une de ses méthodes est appelée. Voici comment vous pouvez le faire:

```
when(maMock.maMethode()).thenReturn("valeur");
```

Avec cette configuration, chaque fois que `maMethode()` est appelée sur `maMock`, elle renverra la chaîne "valeur".

Vérification des interactions

Mockito vous permet de vérifier si certaines méthodes ont été appelées sur votre mock. Par exemple, pour vérifier si `maMethode()` a été appelée sur `maMock`, vous pouvez faire comme suit:

```
verify(maMock).maMethode();
```

Cela vérifiera que `maMethode()` a été appelée exactement une fois sur `maMock`.

Test d'état et tests d'interactions

Dans les tests unitaires, en particulier lors de l'utilisation de mockups, on distingue généralement deux types de tests : les tests d'état et les tests d'interaction.

- **Test d'état** : Un test d'état vérifie le comportement d'une méthode en examinant l'état de l'objet après que la méthode a été exécutée. Par exemple, si vous avez une méthode qui ajoute un élément à une liste, un test d'état pourrait vérifier que la taille de la liste a augmenté de 1 après l'appel de la méthode. Dans ce type de test, vous vous concentrez sur le résultat de l'opération, pas sur la façon dont le résultat a été obtenu.
- **Test d'interaction** : Un test d'interaction vérifie le comportement d'une méthode en examinant ses interactions avec ses dépendances. Par exemple, si vous avez une

méthode qui utilise un service pour récupérer des données et effectuer une opération sur ces données, un test d'interaction pourrait vérifier que la méthode a bien appelé le service avec les bons arguments. Dans ce type de test, vous vous concentrez sur la façon dont l'opération a été effectuée, pas nécessairement sur le résultat de l'opération.

Lorsque vous utilisez des mockups, vous pouvez utiliser à la fois des tests d'état et des tests d'interaction. Vous pouvez configurer le mockup pour qu'il se comporte d'une certaine manière lorsqu'il est utilisé par l'objet que vous testez (c'est-à-dire, vous pouvez "stubber" le comportement du mockup), puis vous pouvez vérifier que l'objet que vous testez a interagi avec le mockup de la manière attendue.

Exemple de Mockup

Voici un exemple complet d'utilisation de Mockito avec le principe AAA (Arrange, Act, Assert):

```
import org.junit.jupiter.api.Test;
import static org.mockito.Mockito.*;

class MaClasseTest {
    @Test
    void monTest() {
        // Arrange
        MonInterface maMock = mock(MonInterface.class);
        when(maMock.maMethode()).thenReturn("mock");

        MaClasse maClasse = new MaClasse(maMock);

        // Act
        String resultat = maClasse.maMethode();

        // Assert
        assertEquals("mock", resultat);
    }
}
```

Dans cet exemple, nous créons un mock de `MonInterface` et nous configurons son comportement pour que `maMethode()` renvoie `"mock"`. Ensuite, nous utilisons ce mock dans `MaClasse` et nous vérifions que `maMethode()` de `MaClasse` renvoie bien `"mock"`.

Pour ce TP, il est question d'écrire des tests unitaires en isolations pour chaque exercice. *Pusher* vos tests par la suite sur vos comptes Github. N'oublier pas de *pusher* vos réponses aux questions dans un fichier README sur votre *repository*.

Exercice1 : Initiation

On cherche à tester le comportement d'une méthode d'un objet et l'état de l'objet après l'appel de la méthode. Supposons que nous avons une classe Calculatrice avec une méthode `additionner(int a, int b)` qui additionne deux nombres et retourne le résultat.

Voici la classe à tester :

```
public class Calculatrice {
    public int additionner(int a, int b) {
        result = a + b ;
        return result;
    }
    private int result ;
}
```

Sur la base du test donné ci-dessous, compléter les *//TODO*.

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;

import static org.mockito.Mockito.*;

@RunWith(MockitoJUnitRunner.class)
public class CalculatriceTest {

    @Mock
    private Calculatrice calculatrice;

    @Test
    public void testAdditionner() {
        // Définition du comportement de la méthode "additionner"
        when(calculatrice.additionner(2, 3)).thenReturn(5);

        //TODO : Appel de la méthode à tester
        // ...

        //TODO : Vérification du résultat
        // ...

        //TODO : Vérification que la méthode "additionner" a été appelée
        // avec les arguments 2 et 3, utiliser la directive « verify »

        //TODO : Vérification qu'aucune autre méthode n'a été appelée sur
        //l'objet après l'appel de la méthode "additionner", utiliser la
        // méthode « verifyNoMoreInteractions »

        // TODO : Vérification de l'état de l'objet après l'appel de la
        //méthode "additionner", penser à utiliser la méthode
        //« getState() » de la directive « verify » : // exemple :
        verify(objet).getState()
    }
}
```

Exercice 2 : Mocker un service externe (API)

Imaginons une application qui gère des utilisateurs. L'application utilise une classe `UserService` pour interagir avec un service web d'API utilisateurs. La classe `UserService` a une méthode `creerUtilisateur(Utilisateur utilisateur)` qui envoie les informations du nouvel utilisateur à l'API pour le créer.

Ci-dessous vous avez l'implémentation des deux classes : la classe (interface) à tester et du service simulé.

```
public class UserService {

    private final UtilisateurApi utilisateurApi;

    public UserService(UtilisateurApi utilisateurApi) {
        this.utilisateurApi = utilisateurApi;
    }

    public void creerUtilisateur(Utilisateur utilisateur) throws
      ServiceException {
        utilisateurApi.creerUtilisateur(utilisateur);
    }
}

public interface UtilisateurApi {

    void creerUtilisateur(Utilisateur utilisateur) throws ServiceException;
}
```

Sur la base du code donné, et du test ci-dessous, compléter les *//TODO*.

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.junit.MockitoJUnitRunner;

import static org.mockito.Mockito.*;

@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {

    @Mock
    private UtilisateurApi utilisateurApiMock;

    @Test
    public void testCreerUtilisateur() throws ServiceException {
        // Création d'un nouvel utilisateur
        Utilisateur utilisateur = new Utilisateur("Jean", "Dupont",
        "jeandupont@email.com");

        // TODO : Configuration du comportement du mock, utiliser la
        //directive « when » avec sa méthode « thenReturn »
        // ...

        // TODO : Création du service avec le mock
        // ...

        // TODO : Appel de la méthode à tester
```

```
        // ...

        // TODO : Vérification de l'appel à l'API
        // ...

    }
}
```

Exercice 3 : Mocker un service externe avec différents scénarios

Dans le même contexte que celui de l'exercice 2, on veut tester différents scénarios susceptibles d'avoir lieu - dans des conditions réelles - lors de la création de l'utilisateur et lors de son envoi à l'API. Compléter les tests qui afin de répondre aux scénarios suivants :

1. Lever une exception lors de la création de l'utilisateur avec l'exception suivante : `ServiceException("Echec de la création de l'utilisateur").`
2. Tester le comportement en cas d'erreur de validation. Penser à utiliser la méthode `never()`, comme deuxième paramètre pour la directive `« verify ».`

3. Supposons que l'API renvoie un identifiant unique pour l'utilisateur créé, compléter les TODO sur le test donné ci-dessous afin de vérifier l'état de l'objet Utilisateur après son envoi à l'API.

```
when(utilisateurApiMock.creerUtilisateur(utilisateur)).thenReturn(true);

// Définition d'un ID fictif
int idUtilisateur = 123;

// TODO: Configuration du mock pour renvoyer l'ID
//...

// Appel de la méthode à tester
userService.creerUtilisateur(utilisateur);

// TODO: Vérification de l'ID de l'utilisateur
//...
```

4. On veut utiliser des arguments capturés pour vérifier les arguments exacts. Pour cela, implémenter le TODO suivant pour capturer les arguments passés à la méthode `creerUtilisateur` du `mock`.

```
ArgumentCaptor<Utilisateur> argumentCaptor =
ArgumentCaptor.forClass(Utilisateur.class);

when(utilisateurApiMock.creerUtilisateur(any(Utilisateur.class))).thenRetur
n(true);

userService.creerUtilisateur(utilisateur);
Utilisateur utilisateurCapture = argumentCaptor.getValue();

// TODO : Vérification des arguments capturés, utiliser les getters de
//l'objet utilisateurCapture
//...
```

Exercice 4 : Jeu de dés

On s'intéresse à des jeux de hasard et d'argent du type casino en ligne. Le sujet n'est pas très réaliste.

Le jeu qui nous intéresse, le "jeu du 7", se joue avec 2 dés et une banque qui gère les pertes et les gains. Le joueur qui veut jouer propose une mise, qui est la valeur de son pari. Les dés sont du modèle classique à tirage aléatoire entre 1 et 6. La banque encaisse les paris et décaisse les gains. En cas de gains trop importants, la banque peut ne plus être solvable, alors le jeu ferme.

La règle du jeu est la suivante. On ne peut jouer qu'à un jeu ouvert. Le joueur propose interactivement une mise, puis est débité du montant de son pari. S'il est insolvable, le jeu s'arrête là. Sinon la mise est encaissée par la banque. Ensuite, les 2 dés sont lancés. Si la somme des lancers ne vaut pas 7, le joueur a perdu sa mise et le jeu s'arrête là. Si la somme des lancers vaut 7, alors le joueur gagne : la banque paye deux fois la mise, somme créditée au joueur. Le jeu consulte la banque pour savoir si elle est encore solvable, et si ce n'est pas le cas, il ferme.

L'interface de la classe Jeu est la suivante :

```
public Jeu(Banque labanque);
public void jouer(Joueur joueur, De de1, De de2) throwsJeuFermeException;
public void fermer();
public boolean estOuvert();
```

Le sujet consiste à tester en isolation la méthode jouer. Le test sera donc purement unitaire au niveau de la classe. On ne demande pas d'écrire ni de tester les autres classes sauf dernière question : limitez-vous à des interfaces et utilisez des doublures.

On propose les interfaces suivantes :

```
public interface De { public int lancer(); }
public interface Joueur {
    public int mise(); // on suppose que mise positive
    public void debiter(int somme) throws DebitImpossibleException;
    public void crediter(int somme);
}
public interface Banque {
    public void crediter(int somme);
    public boolean est_solvable();
    public void debiter(int somme);
}
```

1. Quels objets dont dépend la classe Jeu sont forcément mockés dans un test pour automatiser jouer ? Pourquoi? Répondre dans le fichier README.
2. Lister les scénarios (classes d'équivalence) que vous allez écrire pour tester jouer, en les décrivant dans le README.
3. Écrire le code Java pour `Jeu`.

4. Commencer par écrire le test le plus simple : le cas où le jeu est fermé. Est-ce un test d'état ou un test des interactions ? Répondre dans le README.
5. Tester le cas où le joueur est insolvable. Comment tester que le jeu ne touche pas aux dés ? Est-ce un test d'état ou un test des interactions ? Répondre dans le README.
6. Continuer avec les autres scénarios.

La banque possède un fond, et un fond minimum à ne pas dépasser (initialisés dans le constructeur). Si elle passe en dessous de ce niveau suite à un pari gagnant, le gain est quand même donné au joueur, mais le jeu ferme car la banque n'est plus solvable.

7. Écrire une implémentation pour la banque, et écrire à nouveau un des tests impliquant la banque, en l'intégrant cette fois (donner un titre parlant). Méditer sur la différence entre les 2 tests.