TP3 Qualité

Compte-rendu du TP3 par CREUZIEUX Dorian & BUISSON Yoann.

Exercice 2

Implémenter chaque cas de test pour les méthodes de la classe*Money*.

A

Même si nous avons effectué des tests sur le constructeur de la classe*Money* dès le début du**TP**, les tests sont solicités dans l'exercice 4 avec les tests de robustesse.

- Nous avons commencé par instancier un objet de type*Money* pour pouvoir l'utiliser dans tous nos cas de tests :

```
class MoneyTest {

    private Money money;

    @BeforeEach
    void init() throws Exception {
        money = new Money(45, "EUR");
    }
```

- Nous n'avons pas pris en compte les cas de tests des méthodes*currency()* et *amount()* puisque ce ne sont que les getter des attributs de la classe.

- Pour la méthode *add(Money m)*, nous voulions que celle-ci retourne 0 si le montant dans l'objet en paramètre était négatif et réaliser le scénario normal pour un montant positif :

```
@Test
void testAvecMontantNegatif() {
    money.add(new Money(-4, "EUR"));
    assertEquals(45, money.amount());
}

@Test testAvecMontantPositif() {
    money.add(new Money(12, "EUR"));
    assertEquals(57, money.amount());
}
```

- Pour cette méme méthode, nous voulions qu'elle fasse la convertion si la devise était différente et ainsi ajouter le montant. Nous voulions de plus qu'elle renvoie une exception si celle-ci était nulle :

```
@Test
void testAvecDeviseDifferente() {
    money.add(new Money(12, "USD"));
    assertEquals((int) (45 + (12 * 0.91)), money.amount());
}

@Test
void testAvecDeviseNulleOuInexistante() {
    assertThrows(NullOrUnknownCurrencyException.class, () -> money.add(12, null));
}
```

Aperçu de la méthode add(Money m) de la classe Money :

```
public void add(Money m) {
    add(m.amount(), m.currency());
}

public void add(int amount, String currency) {
    if (amount < 0) {
        this.amount += 0;
    }
    if(currency == null || currency.isEmpty()) {
        throw new NullCurrencyException("La devise est nulle ou n'existe pas.");
    }
    if(!currency.equals(this.currency()) {
        switch(currency) {
            case "EUR": this.amount += amount * 1.29; break;
            case "USD": this.amount += amount * 0.91; break;
            default: break;
        }
    } else {
        this.amount += amount;
    }
}
```

Exercice 3

Effectuer des tests avec **Mockito** afin de simuler la classe composée *Convertion* et utiliser cet attribut correctement dans les méthodes de la classe *Money*.

A

Nous avons remplacé le switch dans la condition d'une devise différente par un appel à la méthode *unitConvertion(String s)* pour construire une chaîne de caractères avec les deux devises :

```
public void add(int amount, String currency) {
    if (amount < 0) {
        this.amount += 0;
    }
    if(currency == null || currency.isEmpty()) {
        throw new NullCurrencyException("La devise est nulle ou n'existe pas.");
    }
    if(!currency.equals(this.currency()) {
        this.amount += amount * conversion.unitConversion
        (currency + String.format("-%s", this.currency));
    } else {
        this.amount += amount;
    }
}
```

Aperçu de la méthode *unitConvertion(String s)* :

```
public double unitConversion(String s) {
    switch (s) {
        case "EUR-USD": return 1.29;
        case "USD-EUR": return 0.91;
        default: return 1.5;
    }
}
```

B

Au lieu d'appeler réellement la méthode de convertion, nous l'avons simulé dans le classe de test *ConvertionTest* afin qu'elle puisse nous retourner des valeurs particulières en fonction du paramètre donné. Nous avons alors demandé à la méthode de convertion de retourner 1.29 si on voulait convertir de **Euros** à **US Dollars** et 0.91 dans le cas contraire. Ces conditions, ajoutés dans la méthode se lançant avant les cas de tests, seront toujours pris en compte. C'est-à-dire que dès lors que la méthode de convertion est solicité, alors on fera appel à une simulation en retournant une valeur :

```
@ExtendWith(MockitoExtension.class)
@RunWith(JUnitPlatform.class)
class ConversionTest {

    @Mock
    private Conversion conversionMock;

    @InjectMocks
    private Money money = new Money(45, "EUR");

    ConversionTest() throws Exception {
    }

    @Before
    public void init() {
        MockitoAnnotations.initMocks(this);
        when(conversionMock.unitConversion("EUR-USD")).thenReturn(1.29);
        when(conversionMock.unitConversion("USD-EUR")).thenReturn(0.91);
    }
```

C

Il nous a suffit d'ajouter un nouveau when dans la méthode *init()* pour renvoyer une erreur si la chaîne de caractères en paramètre commençait par " " :

```
@Before
public void init() {
        MockitoAnnotations.initMocks(this);
        when(conversionMock.unitConversion("EUR-USD")).thenReturn(1.29);
        when(conversionMock.unitConversion("USD-EUR")).thenReturn(0.91);
        when(conversionMock.unitConversion(startsWith(" ")))
        .thenThrow(IllegalArgumentException.class);
}
```

D

Les tests pour la méthode *sub(Money m)* sont les mêmes que pour la méthode *add(Money m)*. Seul un nouveau test a été réalisé puisqu'il fallait tester les cas où le montant à soustraire était plus petit que le montant donné en paramètre avec mêmes et différentes devises :

```
@Test
void testSoustractionAvecDevisesDifferentes() {
    Money mAmount = new Money(4, "USD");
    money.sub(mAmount);
    assertEquals((int) (45 - (4 * 0.91)), money.amount());
}

@Test
void testSoustractionAvecMontantPositif() {
    Money mAmount = new Money(4, "EUR");
    money.sub(mAmount);
    assertEquals(45 - 4, money.amount());
}

@Test
void testSoustractionAvecMontantNegatif() {
    Money mAmount = new Money(-4, "EUR");
    money.sub(mAmount);
    assertEquals(45, money.amount());
}

@Test
void testSoustractionAvecMontantPlusGrand()
    Money mAmount = new Money(60, "EUR");
        money.sub(mAmount);
        assertEquals(0, money.amount());
}
```

Aperçu de la méthode *sub(Money m)*:

```
public void sub(Money m) {
    if (m.amount < 0) {
        this.amount -= 0;
    }
    if(currency == null || currency.isEmpty()) {
        throw new NullCurrencyException("La devise est nulle ou n'existe pas.");
    }
    if(!currency.equals(this.currency()) {
        int result = amount * conversion.unitConversion
        (currency + String.format("-%s", this.currency));
        if(result >= this.amount) {
            this.amount = 0;
        } else {
            this.amount -= result;
        }
    } else {
        if(amount >= this.amount) {
            this.amount = 0;
        } else {
            this.amount -= amount;
        }
    }
}
```

Exercice 4

Effectuer les tests de robustesse afin d'optimiser le code et d'éviter les dévises nulles et les montants négatifs.

A

Comme expliqué précedemment, en effectuant des vérifications dès le constructeur de la classe*Money*, nous pouvions optimiser le code et même éviter de renvoyer dans chaque méthodes, des erreurs :

```
public Money(int amount, String currency) throws NullOrUnknownCurrencyException {
    if (amount <= 0) {
        this.amount = Math.abs(amount);
    } else {
        this.amount = amount;
    }
    if(currency == null || currency.isEmpty() || currency.length() > 3) {
        throw new NullOrUnknownCurrencyException("Unknown currency.");
    } else {
        this.currency = currency;
    }
    conversion = new Conversion();
}
```

Nous avons décidé de convertir les valeurs négatives par leurs valeur absolues et de renvoyer l'erreur d'une devise nulle, ou composée d'une chaîne de caractères trop grande ou trop petite directement dans le constructeur.

B

Dorénavant, lorsqu'on ajoute avec un montant négatif, alors on ajoutera avec sa valeur absolue. Et lorsque la devise est trop grande, trop petite ou inconnue alors on renverra toujours une erreur :

```
@Test
void testAvecMontantNegatif() {
    money.add(new Money(-4, "EUR"));
    assertEquals(49, money.amount());
}

@Test
void testAvecDeviseTropLongue() {
    assertThrows(NullOrUnknownCurrency.class, () -> money.add(5, "EUROS"));
}

@Test
void testAvecDeviseTropPetite() {
    assertThrows(NullOrUnknownCurrency.class, () -> money.add(5, "EU"));
}

@Test
void testAvecDeviseInconnue() {
    assertThrows(NullOrUnknownCurrency.class, () -> money.add(5, "YEN"));
}
```

Exercice 5

Donner la couverture du code afin que les tests réalisés soient passés par toutes les lignes de toutes les classes testées.

A

B

La couverture du code concerne tous les tests réalisés sur la classe*Money*. C'est aussi dans cette classe que la classe *Convertion* est bien solicitée à 100% :