TD4 : COO

Exercice 1 : Services

On s'intéresse à des services qui peuvent être utilisés par des utilisateurs sous réserve que ceux-ci puissent les payer. Les classes qui apparaissent dans la figure 1 sont utilisées.

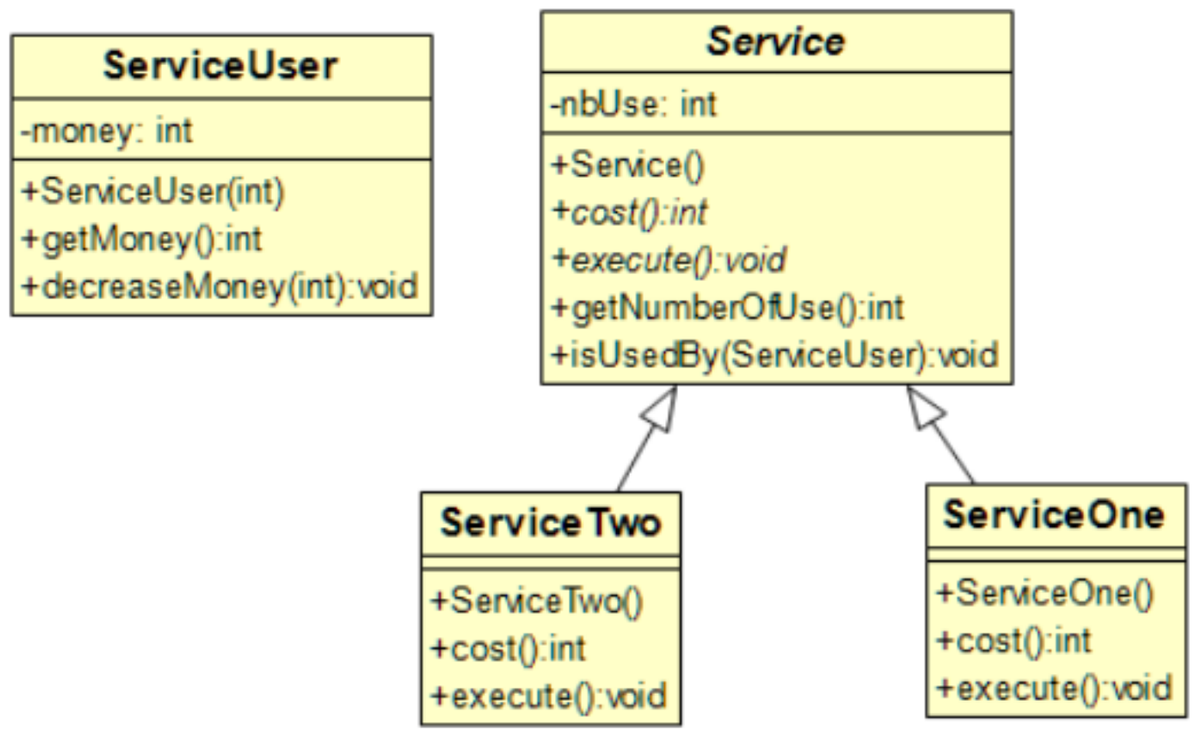


Figure 1: Diagrammes de classes des services

Le type `Service` permet de modéliser ces services. Le coût du service est fourni par la méthode `cost()` spécifique à chaque type de service. L'utilisation d'un service se fait par la méthode `isUsedBy`. Les noms des autres méthodes suffisent pour en comprendre les responsabilités. Voici la documentation et la signature de `isUsedBy` :

```
/** The user uses this service. The user has to pay this service's cost.
* @param user the user of this service
* @throws NotEnoughMoneyException if the user has not enough money to pay this service's cost
*/
public final void isUsedBy(ServiceUser user) throws NotEnoughMoneyException
```

La spécification de `isUsedBy` précise qu'à l'issue de la méthode :

- l'argent de l'utilisateur a été diminué du coût du service s'il en avait assez, sinon l'exception indiquée est levée ;
- le nombre d'utilisations du service a été incrémentée de 1 si l'utilisateur disposait d'assez d'argent.

Q 1 . Donnez le code des **classes de tests** (sauf leur entête et ses `import`) et leurs méthodes permettant de vérifier que l'implémentation de `isUsedBy` vérifie bien cette spécification quels que soient les services considérés, et en particulier pour les services de type « `ServiceOne` » ou ceux de type « `ServiceTwo` ».

Q1 :

```
public abstract class ServiceTest {

    protected ServiceUser user;
    protected Service service;

    protected abstract createService();

@BeforeEach
public void init(){
    this.service = this.createService();
    this.user = new User(this.service.getMoney());
}

@Test
public void testIsUsedByPasses(){
    try {
        this.service.isUsedBy(this.player);
        assertEquals(0, this.player.getMoney());
        assertEquals(1, this.service.getNbOfUse());
    }
```

```
        catch {
            fail();
        }
    }
}

@Test
public void testIsUsedBy(){
    this.user.decreaseMoney(1);
    assertThrows(NotEnoughMoneyException.class,
                 ()-> this.service.isUsedBy(this.player));
}
}
```

```
public class ServiceOneTest extends ServiceTest{

    protected Service createService(){
        Return new ServiceOne();
    }
}

public class ServiceTwoTest extends ServiceTest{

    protected Service createService(){
        Return new ServiceTwo();
    }
}
```

On va s'intéresser plus particulièrement aux méthodes `operate()` et `process()` de `ProductionUnit`.
Le cahier des charges stipule que la méthode `operate()` est `final` et que :

1. L'exécution de la méthode `process(p)` lève une exception `ProductionProblemException` si les ressources disponibles pour l'unité de production sont insuffisantes pour le produit `p`.
2. Si les ressources sont en quantité suffisante, à l'issue de l'exécution de la méthode `process(p)`
   - le nombre de ressources disponibles pour l'unité de production est diminué du montant des ressources requises par le produit `p`.
   - la durée de production du produit `p` est augmentée du temps de production de l'unité.
3. Chaque appel à la méthode `operate()` déclenche l'exécution de la méthode `process(p)`.
4. A l'issue de l'exécution de la méthode `operate(p)`, si la méthode `process(p)` n'a pas déclenché d'exception, le nombre de produits traités par l'unité de production est augmenté de 1.

```
public class MockProduct implements ManufacturedProducts {

public int RRQO{
    Return 3;
}
public int getProductDuration(){
    Return this.time;
}
```

```
public void IPD(int val) {
    this.time += val;
}

public void transformBy(PU unt){
    Return;
}
}
```

```
public abstract class ProductionUnitTest {

    protected ProductionUnit unit;

    protected abstract ProductionUnit createUnit();
```