**UE Conception Orientée Objet**

# Tests

**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.

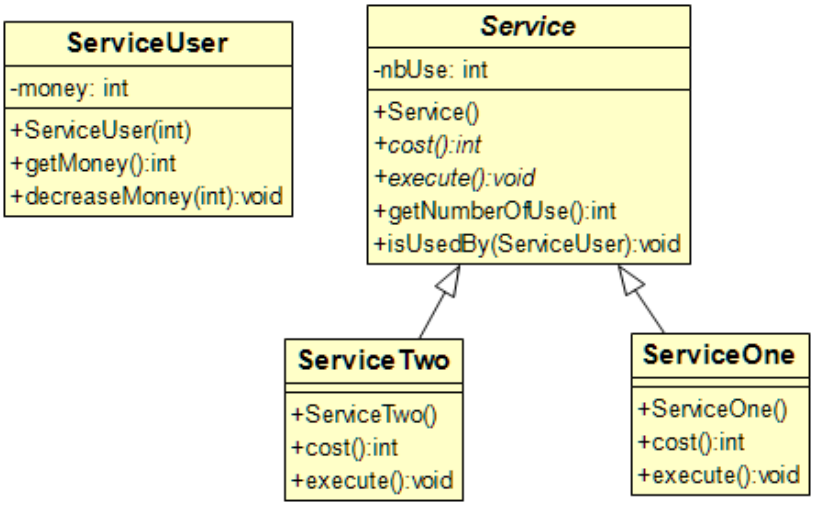



Figure 1: Diagrammes de classes des services

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` ».

**Exercice 2 : Unités de production**

Le contexte général de l'exercice est celui de produits qui sont manufacturés (*manufactured products*) par actions successives de plusieurs unités de production (*production unit*). Pour ce faire, les unités de production utilisent des ressources (*resource*).
On ne considère qu'une version très simplifiée d'un tel problème, suffisante pour les besoins de l'exercice (par exemple on ne s'intéresse pas aux types des ressources ni à comment le produit est transformé par l'unité de production).
Ainsi la modélisation proposée se limite aux classes qui apparaissent dans le diagramme présenté à la figure 2.
Pour `ProductionUnit`, assez naturellement, on a :

- `getAvailableResources()` fournit le nombre de ressources disponibles pour cette unité (afin qu'elle puisse traiter les produits) ;
- `addAvailableResources()` permet d'ajouter des nouvelles ressources disponibles pour cette unité ;
- `getNbProcessedProducts()` fournit le nombre de produits qui ont pu être traités (via sa méthode `process`) par cette unité ;
- `getProductionDuration()` fournit le temps mis par cette unité pour traiter un produit.

CC BY NC SA

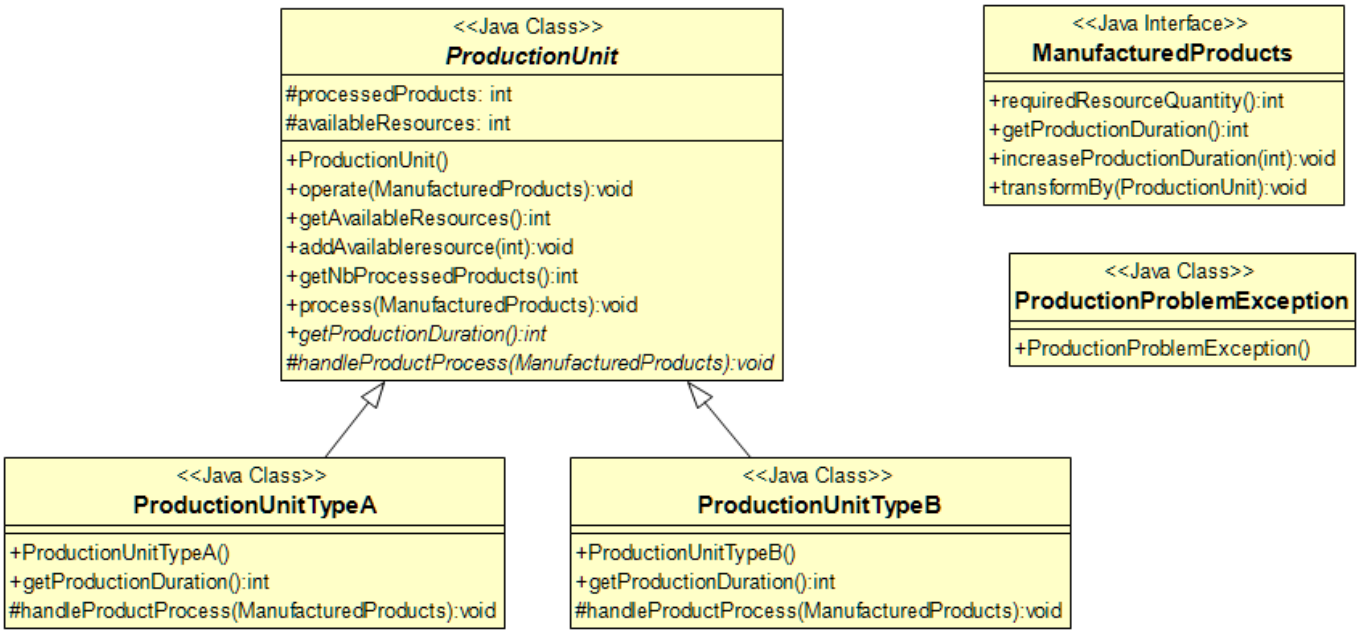



Figure 2: Diagrammes de classes des unités de production

Le code de `ManufacturedProduct` est le suivant :

```
package production.main;

public interface ManufacturedProduct {
  /** @return the amount of resource to be used by a production unit to process this product
   *  (As a simplification it is assumed to be independent from the production unit.)
   */
  public int requiredResourceQuantity();
  /** @return the total time spent (for now) by all the productions unit that processed this product
   */
  public int getProductionDuration();
  /** increase the production duration for this product
   * @param duration the duration to add to the production duration
   */
  public void increaseProductionDuration(int duration);
  /** handle how the unit operates on the product
   * @param unit the production unit that works on this product
   */
  public void transformBy(ProductionUnit unit);
}
```

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.

**Q 1 .** Donnez le code des **classes de tests** (sauf leur entête et ses `import`) avec les méthodes permettant de vérifier que les implémentations des méthodes `process()` et `operate()` vérifient bien les différents points de cahier des charges mentionnés pour chacune des classes `ProductionUnitTypeA` et `ProductionUnitTypeB`.
Le cas échéant, la mise en place des tests de ces méthodes pour de nouvelles sous-classes de `ProductionUnit` doit être facile et demander « peu » de nouveau code.