TP n° 4 & 5 — JUnit et Test Driven Development

Antoine de Roquemaurel (Groupe 3.1)

1 Architecture

Le projet est organisé en deux *packages* distincts : le *package* `achat` et le package `tests`, le premier correspond aux deux classes à tester, le second quant à lui contient les tests unitaires.

1.1 Structures de données

L'exercice nécessitait deux classes, ces deux classes ont été développé en utilisant la méthode TDD : on commence par écrire le test, qui implique ensuite l'écriture de la méthode.

Classe `Produit` Un produit, spécifié par un nom et un prix.

Classe `Panier` Contient les différents produits.

1.2 Tests

Les deux classes sont testé dans le package test, chacune des classes possède une classe de tests associé.

2 Première version

Afin d'écrire la première version de l'application, j'ai commencé par la classe `Produit`.

2.1 `Produit`

2.1.1 Initialisation et getters

Tout d'abord, initialisation du `Produit` à l'aide d'un constructeur initialisant les deux paramètres. Pour cela j'ai créé les méthodes setUp, tearDown, testInitPrice et testInitName. Les deux premières permettent de créer puis de libérer notre structure de données entre chaque tests. Les deux dernières tests qu'une fois créé notre structure contient bien les données qu'on lui à entré dans la méthode setUp.

Afin de faire passer les tests précédents, il a été nécessaire de créer le constructeur, les getteurs et setteurs dans notre classe Produit.

2.1.2 Setters

Une fois les tests précédents passés, il était nécessaire d'avoir deux modificateurs pour nos attributs, deux nouvelles méthodes de tests ont été créé afin de vérifier que l'appel à ses accesseurs changent bien les valeurs. Une fois les deux tests créé, il était nécessaire de créér les méthodes associées afin de faire passer les tests.

2.2 Panier

Le panier est représenté par une `ArrayList`.

2.2.1 Initialisation

Le panier lui nécessitait une initialisation assez simple, la méthode setUp ne contenant qu'un constructeur par défaut.

Nous avons cependant créé deux méthodes vérifiant que notre panier est correctement initialisé une fois créé : les méthodes `testInitProduit` et `testInitTotalPrice` vérifie que nous créons bien un panier vide coutant donc 0€. Ces deux méthodes nécessitait la création d'un constructeur pour notre classe initialisant les valeurs.

2.2.2 Ajout d'un produit

Une fois ceci-fait, j'ai créé des méthodes afin de tester que l'ajout d'un produit s'effectuait correctement, pour cela plusieurs choses à tester :
- Ajout d'un unique produit, signifie une quantité de 1 produit
- Ajout d'un unique produit, nous donne un prix total correspondant à cette quantité
- Ajout de plusieurs produits nous donne le prix total correspondant à la somme des prix.
- Ajout de plusieurs produits nous permet bien d'obtenir une quantité correspondant au nombre de produits ajoutés.

> R Dans cette partie, nous vérifions que si l'utilisateur ajoute des produits de même nom ne pose pas de problèmes au niveau de la quantité et du prix, cependant aucune autre vérifications n'est effectuée.

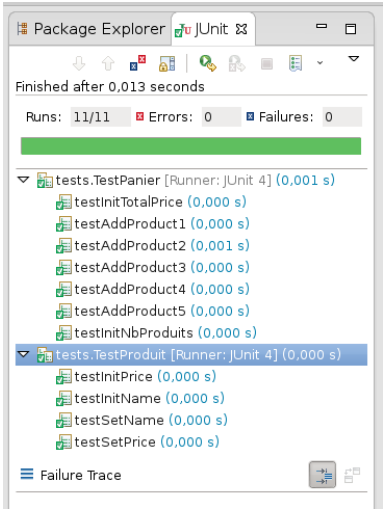

FIGURE 1 – Premiers tests – Passed

3 Version avec quantité

Afin de pouvoir associer une quantité à un produit, nous décidons de remplacer l'`ArrayList` par une `HashMap`, la *map* permettant d'associer un Produit à une quantité.

3.1 Produit

Aucune modification n'est faite sur Produit, cela n'est pas nécessaire.

3.2 Panier

Afin de pouvoir compiler le projet, il est nécessaire dans un premier temps de remplacer la méthode `add` par une méthode put, ajoutant une quantité de 1. En effectuant ceci, les tests passent.

Afin de ne pas toucher aux tests déjà présent, et pouvoir être plus rapide à l'utilisateur, notre panier va posséder deux méthodes `add` : la première ne prend qu'un seul argument, le Produit et l'ajout une seule fois. La seconde prend deux paramètres, le produit et sa quantité. Ainsi, les méthodes de tests déjà présentes appel notre première méthode `add`.

Afin de développer notre méthode `add` avec la quantité, nous avons créé de nouvelles méthodes de tests vérifiant :
– Que le total du prix est égal à la somme des prix
– Que le nombre de produits tiens compte des quantités
Afin de faire passer ces tests, nous avons donc modifié la méthode `getNbProducts()`, parcourant les valeurs de la *map* et la nouvelle méthode `add` quant-à-elle ajoute dans la *map* le produit donné avec sa quantité associée.

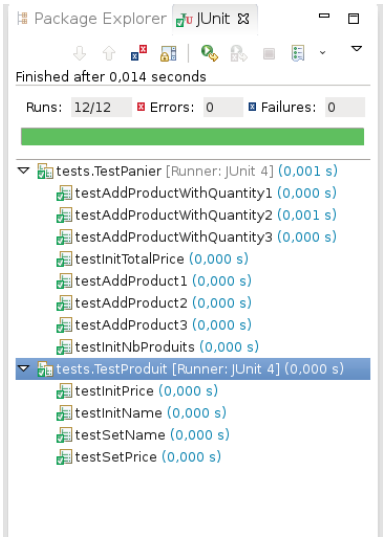

FIGURE 2 – Tests après premier refactoring – Passed

4 Version avec vérification d'unicité

Nous avons ajouter deux nouveaux tests, ceux-ci vérifie que l'application fonctionne si nous décidons d'ajouter des objets existant déjà dans la Map, les tests ne passent pas.

4.1 Produit

Lors du refactoring précédent nous avons utilisé une `HashMap`, cette collection permet très facilement de pouvoir vérifier l'unicité d'un élément dans la collection, pour cela il nous suffit de définir les méthodes `equals` et `hashCode`.

4.2 Panier

Afin de pouvoir ajouter correctement les objets dans la *map*, nous avons modifié la méthode `add` : si l'objet existe déjà, augmenté simple sa quantité. Une fois cette modifications effectuée, tous les tests sont au vert.

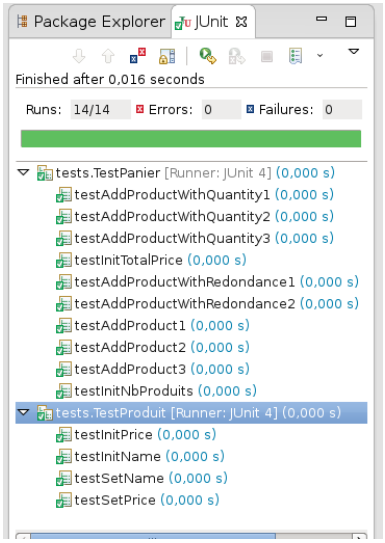

Figure 3 – Tests après V2 – Passed