DEVELOPPEMENT DIRIGE PAR LES TESTS

TDM 5&6

Rapport sur la méthode appliqué pour le TD machine 5 & 6 concernant le développement dirigé par les tests.

Etudiant : Diallo Alpha Oumar Binta
Groupe : 4.1
Professeur chargé du cours : Massié Henri
Année Scolaire : 2012-2013

Diallo Alpha Oumar Binta

21/03/2013

Développement dirigé par les tests

TDM 5&6

Les tests permettent de préciser les spécifications du code, et donc son comportement ultérieur en fonction des situations auxquelles il sera exposé. Ce qui facilite la production d'un code valide en toutes circonstances. On obtient donc un code plus juste et plus fiable. En écrivant les tests d'abord, on utilise le programme avant même qu'il existe. On s'assure ainsi que le code produit est testable unitairement. Il est donc impératif d'avoir une vision précise de la manière dont on va utiliser le programme avant même d'envisager son implémentation. Cela évite souvent des erreurs de conception dues à une précipitation dans l'implémentation avant d'avoir défini les objectifs. De plus, le fait d'avoir des tests augmente la confiance en soi du programmeur lors de la refactorisation du code : il sait qu'à un moment donné les tests ont réussi. Il peut ainsi se permettre des changements radicaux de design en étant sûr, à la fin, d'avoir un programme se comportant toujours de la même façon (si les tests réussissent toujours).

La **méthode traditionnelle** de la rédaction des tests unitaires consiste à rédiger les tests d'une portion d'un programme (unité ou module) afin de vérifier la validité de l'unité implémentée.

Le test découle du code. Ce qui est contraire au principe même du TDD.

La **méthode TDD** quant à elle consiste à rédiger les tests unitaires avant de procéder à la phase de codage.

Le cycle de développement comporte cinq étapes :

1. Ecriture d'un premier test
2. Exécuter le test et vérifier qu'il échoue (car le code n'a pas encore été implémenté)
3. Ecriture de l'implémentation pour faire passer le test
4. Exécution des tests afin de contrôler que les tests passent
5. Remaniement du code afin d'améliorer la qualité en conservant les mêmes fonctionnalités

Exemple de TDD se basant sur le TP 5&6 du cours de TML (Tests et Maintenance de Logiciels).

ENONCE DU PROBLEME

On souhaite réaliser un système pour l'achat en ligne de produits. Dans un tel système, l'utilisateur constitue un panier ou il dépose des produits. Les produits sont caractérisés par un nom et un prix. A chaque ajout de produit, l'utilisateur indique la quantité souhaitée (1 par défaut). Le montant total du panier est actualisé à chaque fois.

Analyse du Probleme

On nous demande de réaliser un système d'achat en ligne de produit. L'utilisateur doit pouvoir constituer un panier où il dépose des produits. Les produits sont caractérisés par un nom et un prix. A chaque ajout d'un produit, l'utilisateur indique la quantité souhaitée (1 par défaut), le montant total du panier est actualisé à chaque ajout. La résolution de ce problème s'effectuera en deux parties, nous allons commencer par traiter le cas de la classe **Panier** puis de la classe **Produit**. Dans la réalisation de la première partie, nous supposerons que la classe Produit existe mais à priori vide.

Realisation de la Classe Panier

Nous commencerons par créer une classe de test unitaire pour tester la classe Panier.
Cette classe est appelé PanierTest.
Que doit faire ce test ?

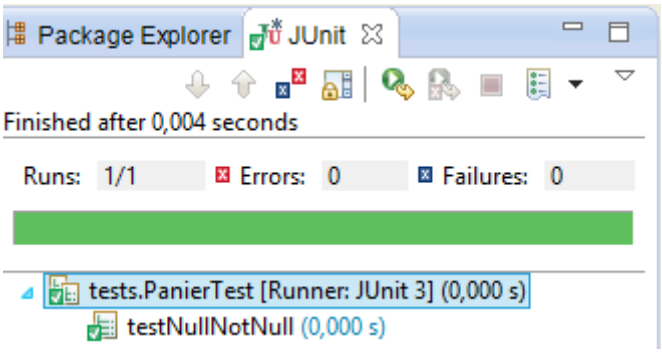
La réponse est relativement simple, il doit créer un objet de type Panier. Sans plus tarder, ci-dessous la classe de test correspondante :

```
/**
 * @author aob
 *
 */
public class PanierTest extends TestCase{

	private Panier panier;

	public PanierTest(String name){
		super(name);
	}
	@Override
	protected void setUp() throws Exception {
		// TODO Auto-generated method stub
		super.setUp();
		panier = new Panier();
	}@Override
	protected void tearDown() throws Exception {
		// TODO Auto-generated method stub
		super.tearDown();
		panier = null;
	}
	public void testNullNotNull(){
		assertTrue(panier != null);
	}
}
```

Le code ne compile pas car la classe Panier n'existe pas encore. Faisons une implémentation rapide de cette classe. Nous obtenons ceci :

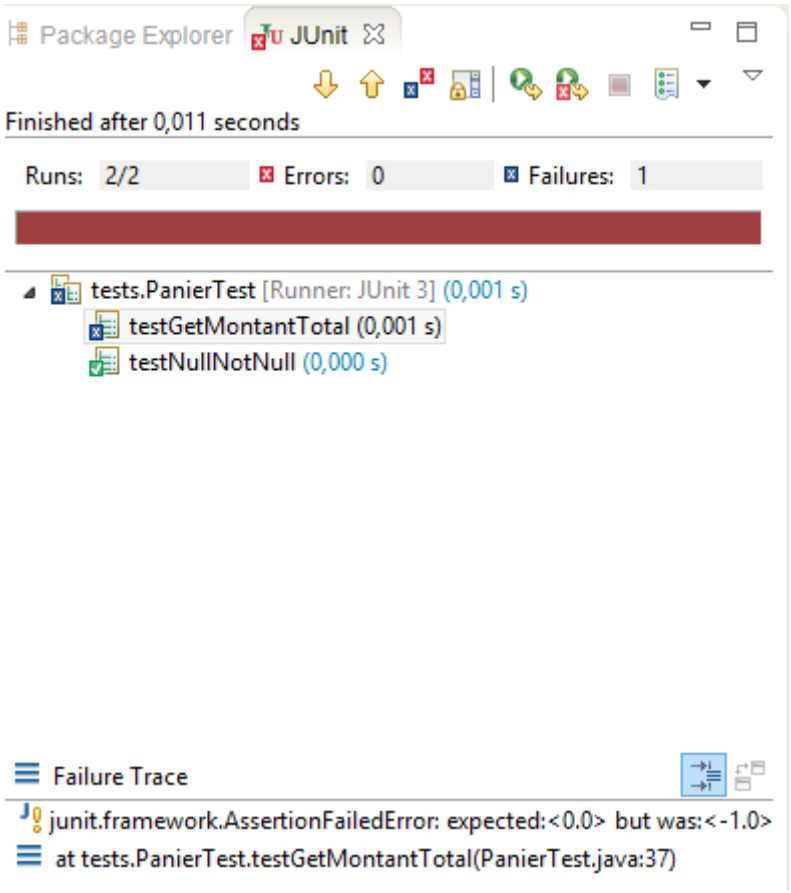
```
public class Panier {

	public Panier(){}
}
```

Après avoir écrit le minimum de code pour la classe panier, que le test passe.



Nous pouvons donc continuer à écrire de nouveau test afin de continuer l'implémentation de la classe Panier. Effectuons un test sur le montant total du panier quand l'utilisateur commence à créer sa liste de course. Le montant est supposé être à 0 à la création du panier.

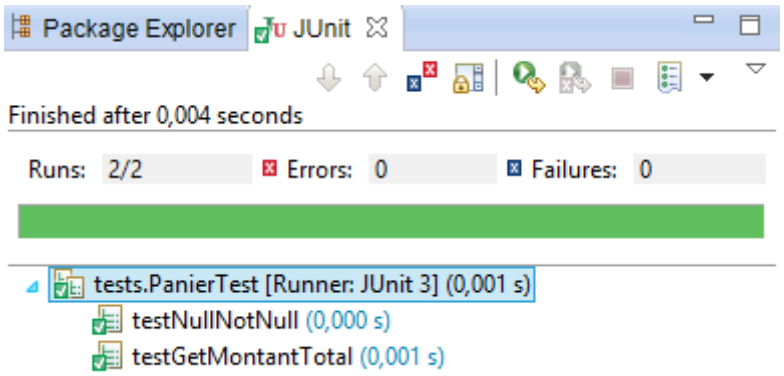
```
public void testGetMontantTotal(){
        double expResult = 0;
        double result = panier.getMontantTotal();
        assertEquals(expResult, result, 0);
    }
```



On constate que le test ne passe pas, le montant total d'un nouveau panier est égale à 0. Donc nous devons avoir un champ **montantTotal** dans la classe Panier. La fonction testé **getMontantTotal** doit donc retourner cette somme. Le champ montantTotal doit être initialisé à 0 par défaut.

```
public class Panier {

	private double montantTotal;
	public Panier(){
		montantTotal = 0.0;
	}

	public double getMontantTotal() {
		// TODO Auto-generated method stub
		return montantTotal;
	}
}
```

Après avoir exécuté le test, nous constatons que le test passe.



Le test suivant porte sur l'ajout d'un produit dans une liste de course de l'utilisateur. Nous supposons que la classe Produit est écrite mais vide.

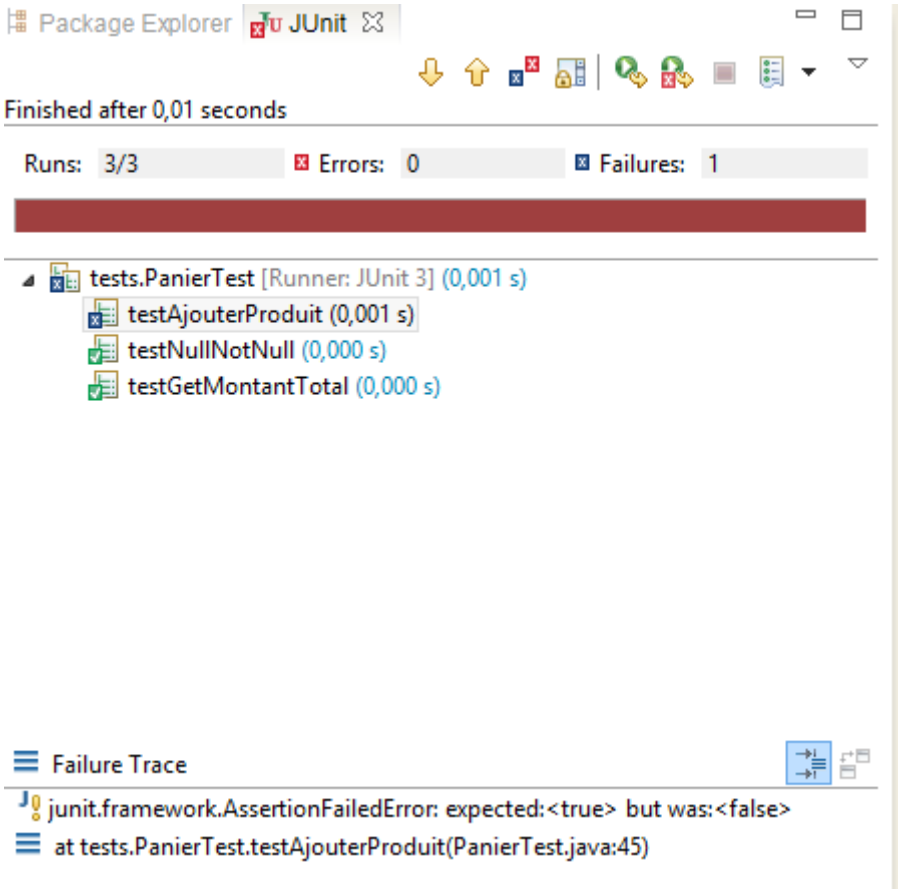
```
/**
 * @author aob
 *
 */
public class Produit {

}
```

Le test effectué est le suivant :

```
public void testAjouterProduit(){
		boolean expectedResult = true;
		Produit produit = new Produit();
		boolean result = panier.ajouterProduit(produit, 1);

		assertEquals(expectedResult, result);
	}
```

Dans la classe Panier la méthode suivante est générer :

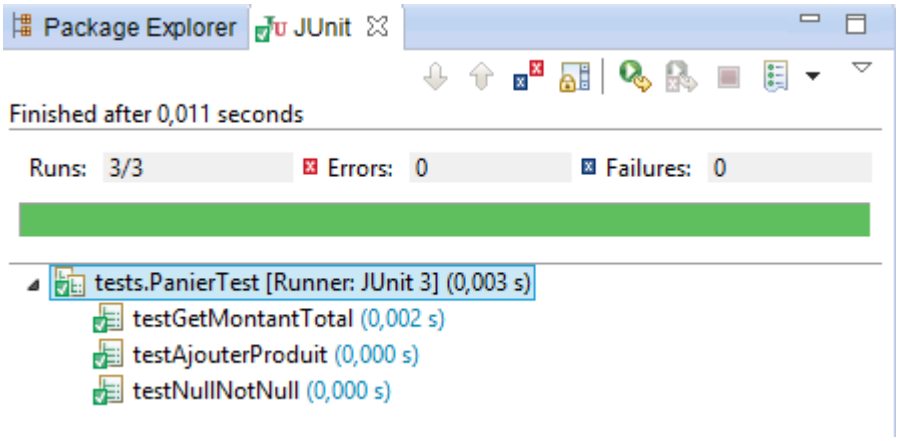
```
public boolean ajouterProduit(Produit produit, int quantite) {
		// TODO Auto-generated method stub
		return false;
	}
```



A première vue le test ne passe pas, cela est évident car la méthode a été générée automatiquement par le biais du menu contextuel d'Eclipse. La méthode **ajouterProduit** prend en argument un produit et une quantité en paramètre, la quantité par défaut est 1 ; si le produit est déjà présent dans le panier, la quantité est augmentée puis le montant total du panier actualisé. Pour sauvegarder l'ensemble des produits ajouté par l'utilisateur, nous opterons pour une **Map**. Une **Map** permet de créer un ensemble de couples clé/valeur (On parle aussi de tableaux associatifs), la clé permettant de retrouver rapidement la valeur qui lui a été associée. La classe HashMap est l'implémentation concrète la plus standard, elle est adaptée à la plupart des situations. Le produit représentera la clé (donc le produit sera unique dans la Map), et la valeur la quantité du produit à ajouter ; donc un champ **listeCourse** sera ajouté dans la classe.

Au niveau de l'implémentation de la méthode ajouterProduit, nous ne toucherons pas le point sur l'actualisation du montant total du panier car la classe Produit est supposé vide. Nous y reviendrons après avoir implémenté cette dernière. Ainsi, nous obtenons le code suivant :

```
public boolean ajouterProduit(Produit produit, int quantite) {
	// TODO Auto-generated method stub
	//si le produit est null ou la quantité inferieur à 0, le
	//produit n'est pas ajouté
	if(produit == null || quantite < 0){
		return false;
	}
	//si la quantité est égale à 0, on la met à 1 par défaut
	int qte = (quantite == 0) ? 1 : quantite;
	//si le panier est vide, la produit est ajouté
	if(listeCourse.containsKey(produit)){
		qte += listeCourse.get(produit);
		listeCourse.put(produit, qte);
	}else{
		listeCourse.put(produit, qte);
	}

	return true;
}
```



Le test est concluant, donc la classe Panier est valide. Dans la seconde partie, nous nous occuperons de la classe Panier, nous supposerons que la classe Panier n'existe pas. A la fin de l'implémentation de cette classe, nous reviendrons sur la classe Panier afin de terminer la méthode d'ajout de produit.

REALISATION DE LA CLASSE PRODUIT

Création de la classe de test ProduitTest

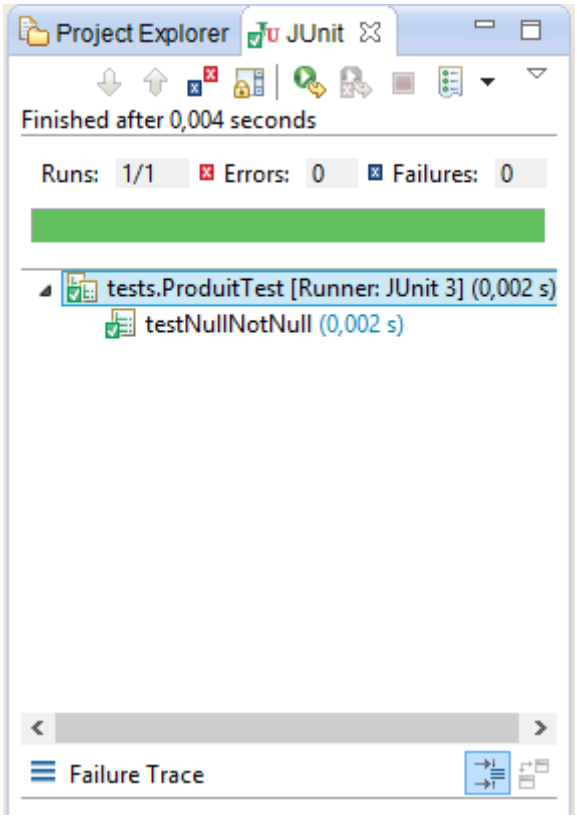
Dans un premier temps, comme la méthode TDD le suggère, la classe de test est d'abord créée avant l'implémentation. Ci-dessous le code

```
/**
 *
 */
package tests;

import junit.framework.TestCase;

/**
 * @author aob
 *
 */
public class ProduitTest extends TestCase {
	public ProduitTest(String name) {
		super(name);
	}

	public void testNullNotNull() {
		Produit produit = new Produit();
		boolean expResult = true;
		boolean result = (produit != null);

		assertEquals("un produit instancie n'est pas null", expResult,
result);
	}

}
```

Les erreurs de compilation signalent que l'objet Produit n'existe pas. La méthode employée est donc la bonne, en écrivant les tests on considère que nous avons besoin de tel objet et méthode pour respecter une règle fonctionnelle. Ici, en l'occurrence il est nécessaire de disposer de l'objet Produit (qui sera créé par la suite). Dans le but de corriger les erreurs de compilation il faut créer l'objet Produit avec le minimum d'information. Voici les implémentations minimum pour l'objet Produit :

```
/**
 *
 */
package code;

/**
 * @author aob
 *
 */
public class Produit {

}
```

Nous avons écrit le test ainsi que le code minimal afin que le test passe, voir ci-dessous.



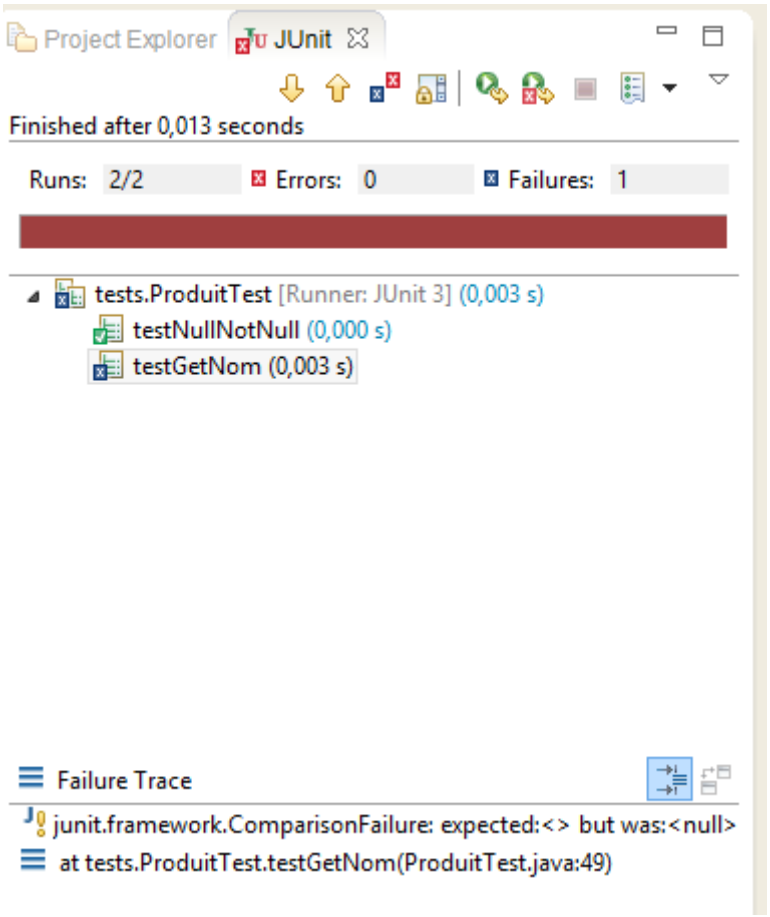
Nous constatons que le test passe, nous pouvons donc continuer à implémenter d'autres tests afin de compléter la classe Produit.

```
protected void setUp() throws Exception {
            // TODO Auto-generated method stub
            super.setUp();
            produit = new Produit();
      }
public void testGetNom(){
            assertEquals("", produit.getNom());
      }
```

Le but est tout d'abord de tester que le produit instancié a pour nom une chaine de caractère vide. C'est pourquoi le nom de la méthode de test porte le nom testGetNom. Le minimum de code ajouté dans la classe Produit est :

```
public class Produit {

      public String getNom() {
            // TODO Auto-generated method stub
            return null;
      }

}
```

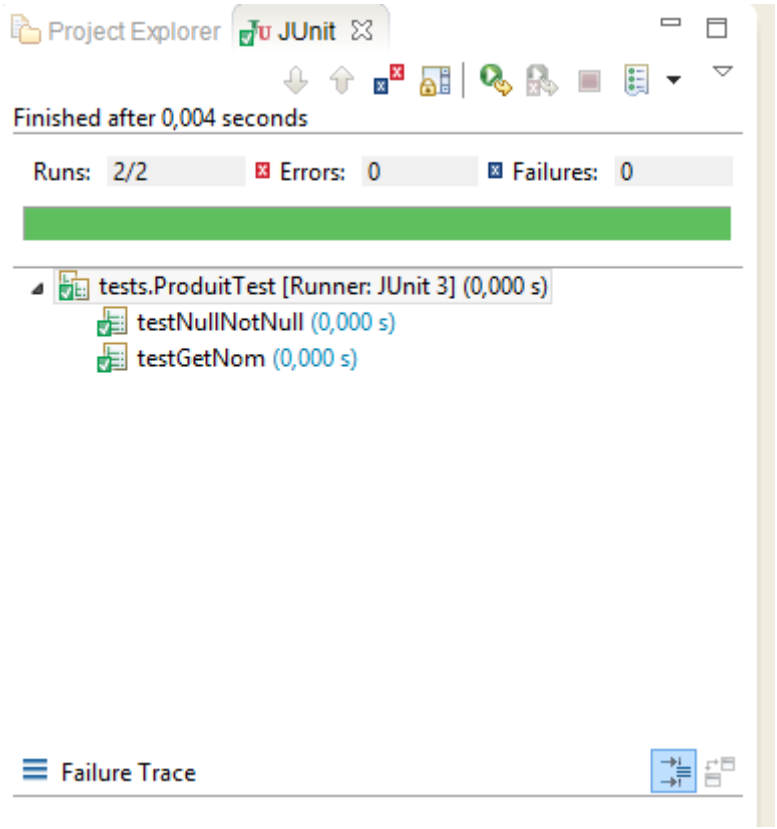
Après exécution du test, on constate qu'il échoue.



Le test échoue et cela est normal car l'implémentation complète n'a pas encore été faite. Le Produit devant avoir un nom, un champ nom est créé et il est initialisé lors de la construction de l'objet à la bonne valeur. Le code corrigé est le suivant :

```
public class Produit {

	private String nom;

	public Produit(){
		nom = "";
	}
	public String getNom() {
		// TODO Auto-generated method stub
		return nom;
	}

}
```

On contrôle que les tests passent.



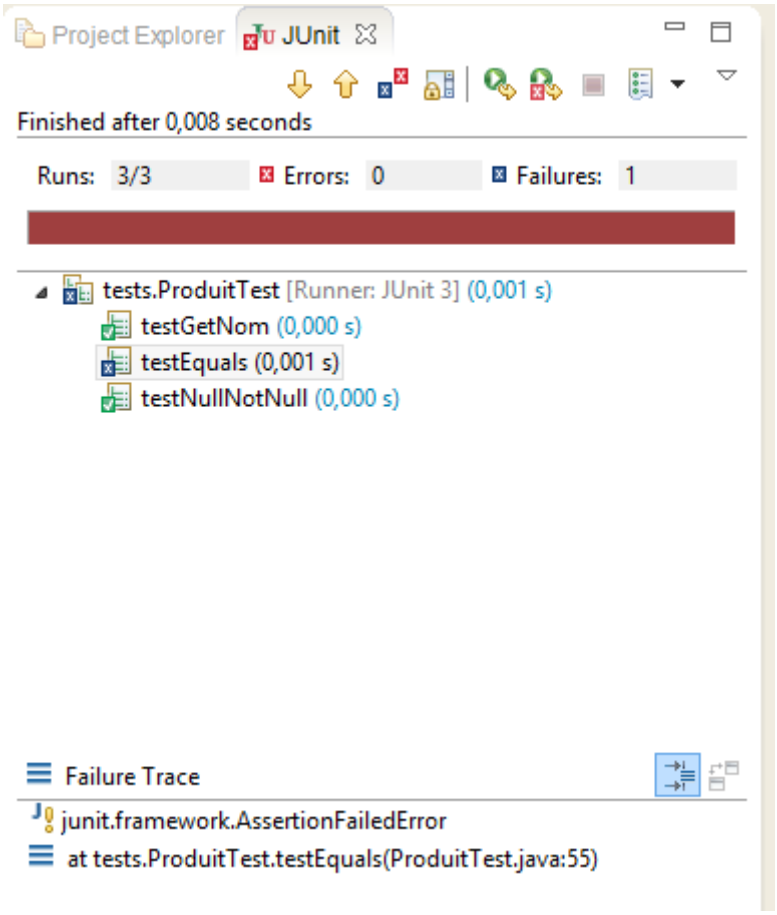
Maintenant on souhaite créer un produit initialisé avec son nom au début. Pour cela les tests vont d'abord être rédigés comme précédemment.

```
public void testEquals(){
            String nom = "yaourt";
            produit = new Produit(nom);
            assertTrue(nom.equals(produit.getNom()));
      }
```

Le minimum de l'implémentation a été écrit afin que le code compile.

```
public Produit(String nom) {
            // TODO Auto-generated constructor stub
      }
```

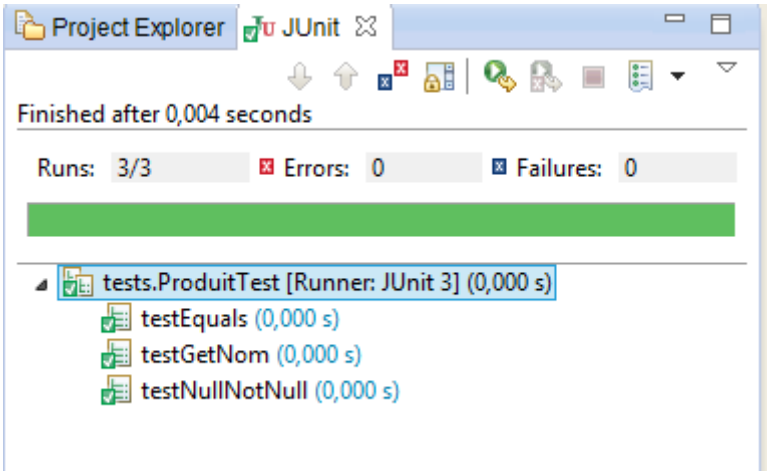
Exécution du test, on remarque que le test à échouer.



On remanie le code afin que le test passe.

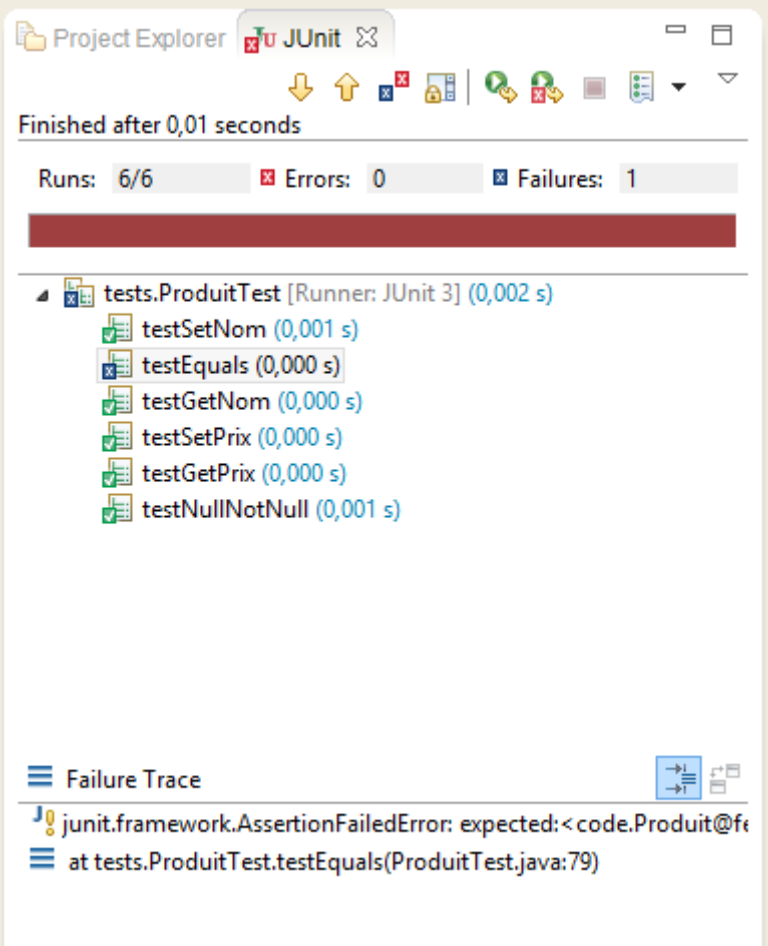
```
public Produit(String nom) {
		// TODO Auto-generated constructor stub
		this.nom = nom;
	}
```

On vérifie que le test passe.



On continue ainsi pour chaque fonctionnalité de la classe. Après avoir répété ce processus, on obtient les résultats suivants.

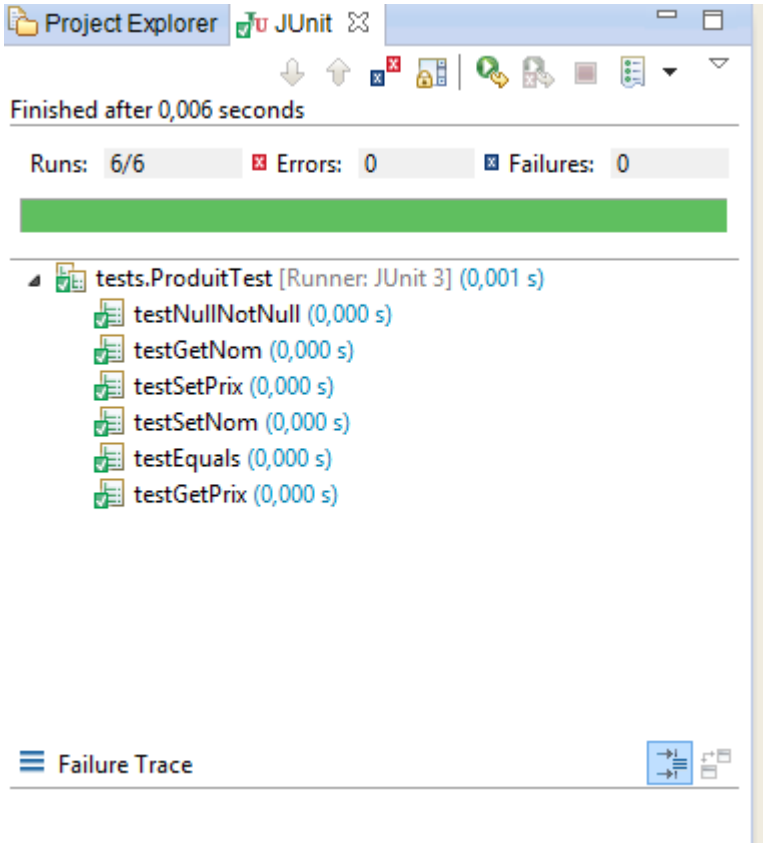
```
public void testEquals(){
            String nom = "yaourt";
            double prix = 1.05;
            produit = new Produit(nom, prix);
            assertTrue(nom.equals(produit.getNom()));
            assertEquals(prix, produit.getPrix());

            Object object = null;
            boolean expResult = false;
            boolean result = produit.equals(object);
            assertEquals(expResult, result);

            Produit instance = new Produit(nom, prix);
            assertEquals(produit, instance);
      }
```



On remarque que le test ne passe pas, pour y remédier, il faut redéfinir la méthode equals, car pour comparer deux objets JAVA se base sur la méthode equals de la classe Object qui compare les adresses mémoires des objets.

```
/* (non-Javadoc)
 * @see java.lang.Object#equals(java.lang.Object)
 */
@Override
public boolean equals(Object obj) {
	if (this == obj)
		return true;
	if(obj == null)
		return false;
	if(obj.getClass() != getClass())
		return false;

	final Produit produit = (Produit) obj;
	if(!produit.nom.equals(nom))
		return false;

	return produit.prix == prix;
}
```

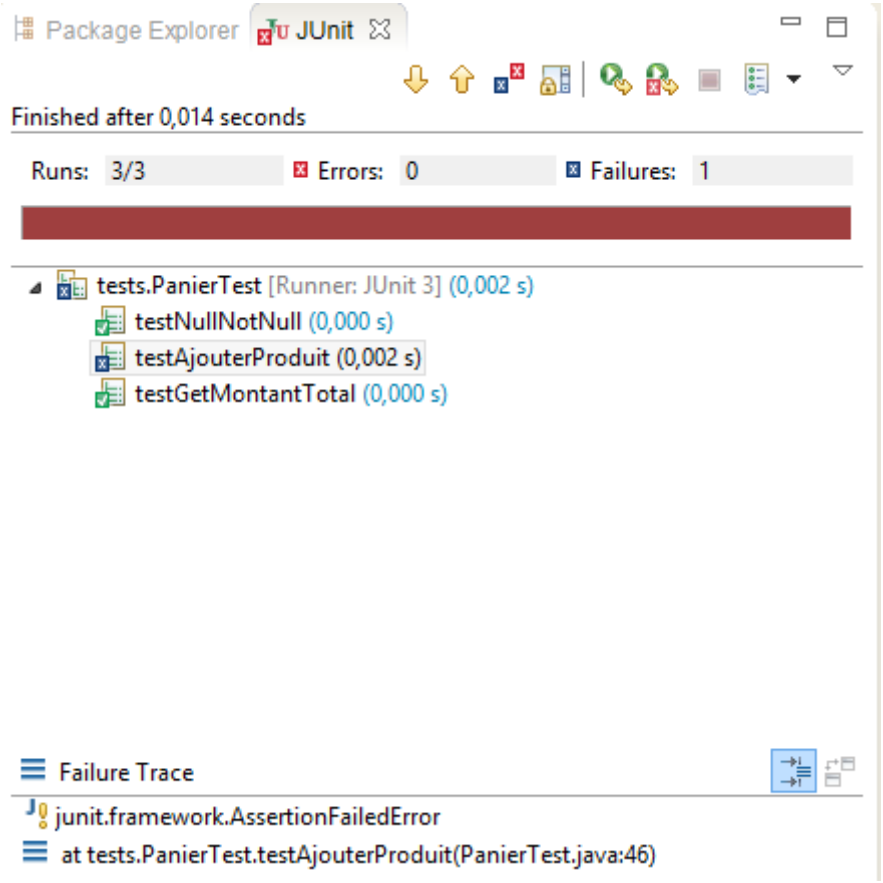
Après exécution des tests, nous obtenons les résultats suivants. La classe Produit est complète et passe tous les tests.



RETOUR SUR LA CLASSE PANIER

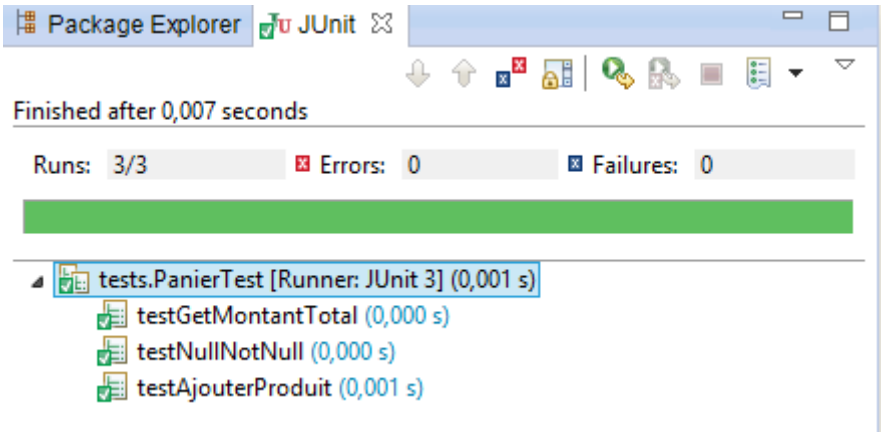
Après avoir fini l'implémentation de la classe Produit, nous pouvons terminer la méthode d'ajout de produit de la classe Panier.

```
public void testAjouterProduit(){
            boolean expectedResult = true;
            Produit produit = new Produit("yaourt", 3);
            boolean result = panier.ajouterProduit(produit, 6);

            assertEquals(expectedResult, result);
            assertTrue(panier.getMontantTotal() == 18);
      }
```



Le test ne passe pas ; cela est normal car la méthode n'est pas encore modifiée dans la classe Panier. Après avoir modifié le code, nous obtenons cela :

```
public boolean ajouterProduit(Produit produit, int quantite) {
		// TODO Auto-generated method stub
		//si le produit est null ou la quantité inferieur à 0, le
		//produit n'est pas ajouté
		if(produit == null || quantite < 0){
			return false;
		}
		//si la quantité est égale à 0, on la met à 1 par défaut
		int qte = (quantite == 0) ? 1 : quantite;
		//si le panier est vide, la produit est ajouté
		if(listeCourse.containsKey(produit)){
			qte += listeCourse.get(produit);
			listeCourse.put(produit, qte);
		}else{
			listeCourse.put(produit, qte);
		}
		//calcul du montant total du panier
		double prixArticleCourant, total = 0;
		for(Entry<Produit, Integer> courant : listeCourse.entrySet()){
			prixArticleCourant = courant.getKey().getPrix() *
			courant.getValue();
			total += prixArticleCourant;
		}

		montantTotal = total;
		return true;
	}
```

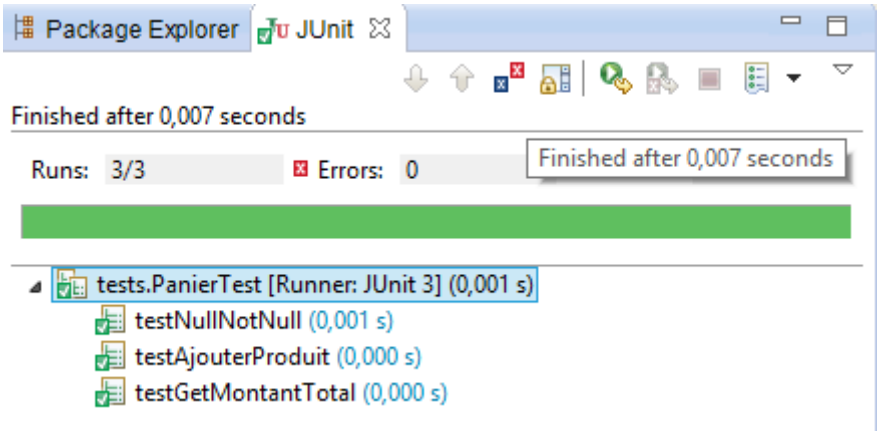


Le test passe, nous pouvons donc remanier le code afin qu'il soit plus lisible, et plus performant. Nous pouvons faire tout cela grâce au menu contextuel proposé par Eclipse (ExtractMethod). Ainsi nous obtenons une nouvelle méthode.

```
public double calculerMontant() {
		//calcul du montant total du panier
		double prixArticleCourant, total = 0;
		for(Entry<Produit, Integer> courant : listeCourse.entrySet()){
			prixArticleCourant = courant.getKey().getPrix() *
			courant.getValue();
			total += prixArticleCourant;
		}
		return total;
	}
```

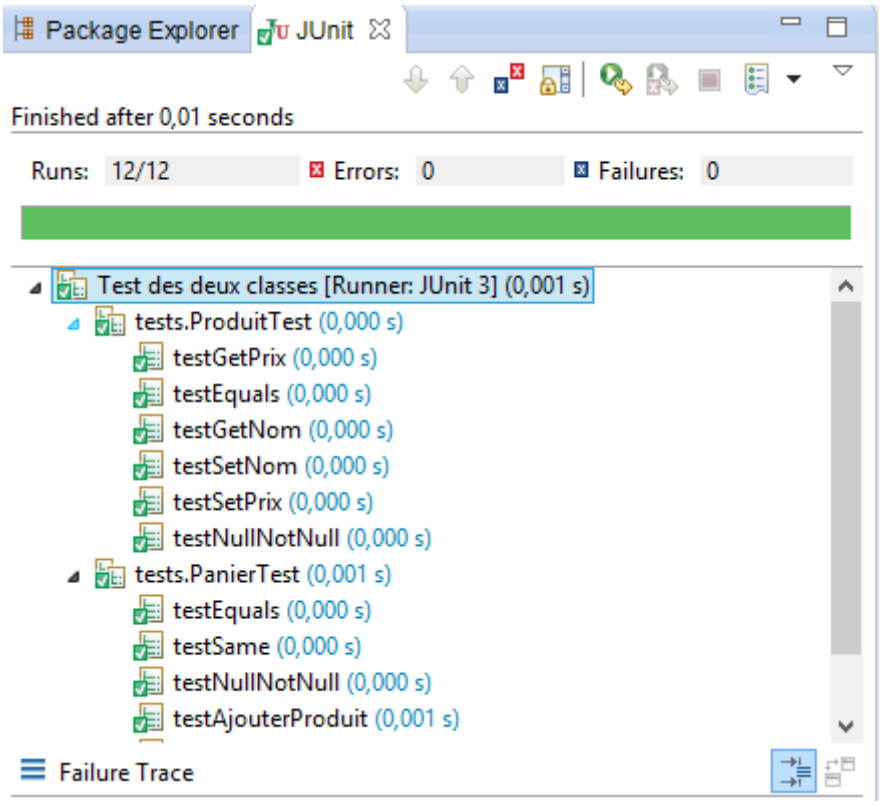
Ainsi, nous obtenons le code suivant :

```
public boolean ajouterProduit(Produit produit, int quantite) {
		// TODO Auto-generated method stub
		//si le produit est null ou la quantité inferieur à 0, le
		//produit n'est pas ajouté
		if(produit == null || quantite < 0){
			return false;
		}
		//si la quantité est égale à 0, on la met à 1 par défaut
		int qte = (quantite == 0) ? 1 : quantite;
		//si le panier est vide, la produit est ajouté
		if(listeCourse.containsKey(produit)){
			qte += listeCourse.get(produit);
			listeCourse.put(produit, qte);
		}else{
			listeCourse.put(produit, qte);
		}
		montantTotal = calculerMontant();
		return true;
	}
```



SUITE DE TEST

```
/**
 *
 */
package tests;

import junit.framework.Test;
import junit.framework.TestSuite;

/**
 * @author aob
 *
 */
public class AllTest {

	public static Test suite(){
		TestSuite suite = new TestSuite("Test des deux classes");

		suite.addTestSuite(ProduitTest.class);
		suite.addTestSuite(PanierTest.class);

		return suite;

	}
}
```



Tous les tests passent avec succès, les deux classes ont été implémentées en utilisant la méthode du TDD.