TP Test Unitaire et pattern en Java

Virtual Pub

Présentation de « Virtual Pub »

Votre société a remporté un marché dans la réalisation d'un environnement de vie virtuelle. Elle a obtenu la réalisation, en Java, de la bibliothèque « Virtual Pub » : bibliothèque de gestion d'un Pub (Public Bar) virtuel. Un Pub virtuel est constitué d'une cave où sont stockées les réserves et d'un bar qui ne peut être approvisionné que par la cave. Ainsi toute livraison du Pub se fait directement dans la cave. Le seul point d'entrée de cette bibliothèque est le Pub, il peut être livré ou servir des boissons aux clients (Remarque : aucune gestion financière ne vous est demandée dans cette version de « Virtual Pub »). « Virtual Pub » doit être capable de gérer deux catégories de boissons : les boissons de base et les cocktails. Les boissons de base se subdivisent en trois catégories : les boissons sans alcool, celles avec alcool et pour finir les boissons chaudes. Les Cocktails quant à eux se décomposent en deux catégories : ceux sans et ceux avec alcool. Un Cocktail est un assemblage proportionné de différentes boissons de base et est considéré alcoolisé dès que l'un de ses éléments de base l'est.

Test Unitaire : jeu de 10 erreurs (voire un petit peu plus …)

Vous êtes dans la position d'une équipe de test et il vous a été fourni une première version de « Virtual Pub », écrite par un stagiaire (Le Fameux, Pierre de son prénom), que vous devez tester. Dans cette position vous n'avez donc pas à modifier les sources qui vous sont fournies, mais à mettre en place un ensemble de tests unitaires et faire remonter les différents problèmes à l'équipe de développement qui vous a fourni ces sources.

Prérequis :

- Logiciel & plug-in :
  - Java version 8 (min)
  - Eclipse IDE version 2020-03 (min)
    - EclEmma version 3.1.3 (min)
    - SonnarLint version 5.0 (min)
  - Bibliothèque
    - JUnit 5

A l'issue de la séance, vous devez rendre une archive comprenant :

- Un plan de test
- Les tests implémentés
- Un rapport de tests intégrant l'état des tests effectués et la couverture du code effectué
- Un rapport donnant les métriques de qualité du code

Différentes aides et exemples vous sont fournis en annexe afin de mettre en place ces jeux de tests.

Annexes :

Dans les méthodes de test unitaire, les méthodes testées sont appelées et leur résultat est testé à l'aide d'assertions :

- **assertEquals**(A, B) : teste si a est égal à b (a et b sont soit des valeurs primitives, soit des objets possédant une méthode equals).
- **assertEquals**(message, A, B) : teste si a est égal à b (a et b sont soit des valeurs primitives, soit des objets possédant une méthode equals) et affiche « message » en cas d'erreur.
- **assertTrue**(boolean condition) teste si la condition est vrai.
- **assertFalse**(boolean condition) teste si la condition est fausse.
- **assertNull**(Object object) teste si l'objet a est nul.
- **assertNotNull**(Object object) teste si l'objet a n'est pas nul.
- **assertSame**(Object a, Object b) teste si a et b réfèrent au même objet.
- **assertNotSame**(Object expected, Object actual) teste si a et b ne réfèrent pas au même objet.
- **fail**() force l'échec du test.

Quelques règles de bonne conduite avec JUnit

- Écrire les test en même temps que le code.
- Tester uniquement ce qui peut vraiment provoquer des erreurs.
- Exécuter ses tests aussi souvent que possible, idéalement après chaque changement de code.
- Écrire un test pour tout bogue signalé (même s'il est corrigé).
- Si un bogue apparaît et qu'aucun test ne l'a détecté c'est qu'il manque probablement un test !
- Ne pas tester plusieurs méthodes dans un même test.
- Attention, les méthodes privées ne peuvent pas être testées !

Junit et Eclipse

Pour chaque Class il est possible de créer « automatiquement » le « TestCase » associé. Ce dernier prendra pour nom normé : NomDeLaClassTest.

Exemple :

- La Class testée :

```
class Money {
	private double fAmount;
	private String fCurrency;

	public Money(double amount, String currency) {
		fAmount= amount;
		fCurrency= currency;
	}

	public double amount() {
		return fAmount;
	}

	public String currency() {
		return fCurrency;
	}

	public Money add(Money m) {
		return new Money(amount()+m.amount(), currency());
```

```
	}

	public boolean equals(Object anObject) {
		if (anObject instanceof Money) {
			Money aMoney= (Money)anObject;
			return aMoney.currency().equals(currency())
				&& amount() == aMoney.amount();
		}
		return false;
	}

	public Money convert(double fact, String currency){
		return new Money(this.amount()*fact, currency);
	}
}
```

- La Class de test :

```
import junit.framework.TestCase;
import junit.framework.Assert;

public class MoneyTest extends TestCase {
	private Money f12CHF;
	private Money f14CHF;

	protected void setUp() {
		f12CHF= new Money(12, "CHF");
		f14CHF= new Money(14, "CHF");
	}
	public void testEquals() {
		Assert.assertTrue(!f12CHF.equals(null));
		Assert.assertEquals(f12CHF, f12CHF);
		Assert.assertEquals(f12CHF, new Money(12, "CHF"));
		Assert.assertTrue(!f12CHF.equals(f14CHF));
	}
	protected void testSimpleAdd() {
		Money expected= new Money(26, "CHF");
		Money result= f12CHF.add(f14CHF);
		Assert.assertTrue(expected.equals(result));
	}
	public void testConvert() {
		Money convertion = f12CHF.convert(0.6, "EUR");
		Assert.assertNotSame(f12CHF, convertion);
		Assert.assertEquals(f12CHF.amount()*0.6, convertion.amount(), 0);
	}

}
```