SQLI

Soignez Vos Tests Unitaires

- Pré-requis :
    - Eclipse avec java 8 d'installé

Import Projet

1. Démarrer eclipse
2. Importer le projet tp-calculette
3. Ajouter la jdk si besoin

Le composant à tester

- Calculator une implémentation de ICalculator qui a 4 opérations
    - sum
    - minus
    - divide
    - multiply

Approche standard

1. Ajoutez un nouveau dossier de sources nommé tests au même niveau d'arborescence que src.
2. Dans l'explorateur, faites un clic droit sur la classe Calculator.
3. Dans le menu contextuel, cliquez sur New – JUnit Test Case.
4. Sélectionnez le bouton radio New JUnit 4 test.
5. Changez le dossier Source folder pour tests.
6. Cliquez sur next et sélectionnez les 4 méthodes Calculator
7. Lancer les tests : Clic droit sur CalculatorTest ⇒ Run as Junit Test

L'approche BDD 1

On se doute que la génération automatique n'est pas la meilleure façon d'écrire un test unitaire soigné pour votre code…

- La première étape quand on veut tester une classe est de se demander ce

qu'on va tester. Ici, au lieu de se focaliser sur la méthode à tester, nous allons nous intéresser à son comportement.

- Voici donc la liste des comportements attendus :
  - si j'ajoute 2 à 3 j'obtiens 5
  - si j'ajoute 2 à -3 j'obtiens -1
  - si a 2 je soustrais 3 j'obtiens -1
  - si a 2 je soustrais -3 j'obtiens 5
  - si je divise 10 par 2 j'obtiens 5
  - si je divise 10 par 0 j'obtiens une erreur
  - si je multiplie 2 à 3 j'obtiens 6
  - si je multiplie 2 par 0 j'obtiens 0
- Cette liste est traduisible directement en classe de test unitaire :
  - le nom de la classe est le nom du composant avec le suffixe Test ;
  - chaque fonctionnalité se traduit par une méthode de test préfixée par should qui énonce le comportement attendu.

```
@Test
public void
should_suggest_ACDC_if_user_was_born_in_the_60s() {
    fail("Not yet implemented");
}
```

1. Ecrivez les 8 tests dans un premier temps en soignant la signature des méthodes avec le corps ci-dessous :
   a. fail("Not yet implemented");
2. Implémenter les tests pour qu'ils deviennent tous verts :), vous allez avoir besoin uniquement des snippets ci-dessous :
   a. @Test(expected = ArithmeticException.class)
   b. Assert.assertEquals

L'approche BDD 2

- Observations :

    - Dès la mise en place du deuxième test on remarque que nous avons du code en commun avec le premier test. En réalité, il concerne deux comportements différents, ce qui n'est pas évident à la lecture du test.

- On va formuler plus précisément les comportements
    - Pour que le comportement à tester dans chaque méthode soit plus clairement identifié, nous allons structurer chaque test en 3 sections (GIVEN, WHEN, THEN).
    - Le plus simple est d'insérer des commentaires :

```
    @Test
    public void
should_suggest_ACDC_if_user_is_born_in_the_60s() {
            // GIVEN
            MusicGuide guide = new RockMusicGuide();
            int birthYear = 1964;

            // WHEN
            List<String> suggestions =
guide.forUserBirthYear(birthYear).suggest();

            // THEN
            assertThat(suggestions).contains("AC/DC");
    }
```

- Observations :
    - la section GIVEN contient la mise en place du contexte d'exécution du test ;
    - la section WHEN permet d'exercer un comportement précis du composant qui est testé : ici, la suggestion en fonction du contenu de la librairie dans le premier test, puis la suggestion en fonction de l'année de naissance dans le second ;
    - la section THEN contient les vérifications concernant le résultat du test : assertions, vérification des appels aux mocks, etc ;
    - pour séparer ces trois sections et alléger la notation, certains développeurs préfèrent utiliser des sauts de ligne ;
    - l'intérêt de cette pratique est de focaliser le test en désignant

précisément le comportement qui est testé, agissant ainsi comme un « mode d'emploi » du composant ;

- Si on veut décliner le test pour plusieurs années de naissance, il est facile de créer un test avec des paramètres sur cette variable. Pour faire ça on peut utiliser junitparams ou à l'annotation @Parameters de Junit.

1. Refactorez les 8 tests
2. Le tp est fini

Conclusion

- Nous devons changer notre attitude traditionnelle envers la construction des programmes :
  - au lieu de considérer que notre tâche principale est de dire à un ordinateur ce qu'il doit faire, appliquons-nous plutôt à expliquer à des êtres humains ce que nous voulons que l'ordinateur fasse.
- L'application du BDD aux tests unitaires se rapproche du literate programming. Le code java devient un support de communication et vous devenez un « programmeur lettré » dont l'oeuvre n'est pas réalisable par un bête générateur de tests.