

Soignez Vos Tests Unitaires Avec Le Behavior-Driven Development (BDD)

¥ Pré-requis :

- ! Eclipse (ou IDE équivalent) installé
- ! JDK 8+ installé
- ! Maven 3.6.3+ installé (facultatif)

Import Projet

1. Démarrer Eclipse
2. Importer le projet `junit-practical-work` comme un projet Maven (menu contextuel *Imports* → *Existing Maven projects* dans la vue *Package Explorer*)

Le composant ^ tester

¥ Le composant représenté par la classe `CalculatorImpl` est une implémentation de l'interface `ICalculator` et comporte quatre opérations simples simulant une calculatrice :

- ! `sum`
- ! `minus`
- ! `divide`
- ! `multiply`

Approche standard

1. Dans l'explorateur, faites un clic droit sur la classe `CalculatorImpl` (sous le package `com.sqli.isc.iut.courses.junit.calculator.impl`).
2. Dans le menu contextuel, cliquez sur *New > JUnit Test Case*.
3. Sélectionnez le bouton radio *New JUnit 4 test* (sélectionné par défaut).
4. Remarquez que le dossier *Source folder* pointe sur le répertoire `junit-practical-work/src/test/java`, répertoire standard en Java où sont placés les tests unitaires.
5. Renommer le nom de la classe de test en `CalculatorTest`.
6. Cliquez sur le bouton *Next >* et sélectionnez les quatre méthodes de `CalculatorImpl`.

7. Lancer les tests : faites un clic droit sur la classe `CalculatorTest` et tapez `Run as JUnit Test`.

Première approche BDD

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 + 3, j'obtiens 5
- ! si j'ajoute 2 + -3, j'obtiens -1
- ! si + 2 je soustrais 3, j'obtiens -1
- ! si + 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 par 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 annonce le comportement attendu. Préférez pour le nom de la méthode la syntaxe `Snake Case` comme par exemple `should_suggest_ACDC_if_user_was_born_in_the_60s` plutôt que le style `Camel Case` `shouldSuggestACDCIfUserWasBornInThe60s` moins lisible.

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

1. Ecrivez les huit 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`

Remarques :

- ¥ L'annotation `@Test` provient du package `org.junit` de la librairie JUnit 4 (sous JUnit 5, l'annotation provient du package `org.junit.jupiter.api`)
- ¥ La classe `org.junit.Assert` de JUnit 4 comporte un ensemble des méthodes d'assertions permettant de réaliser les tests finaux (la méthode *fail* permet d'indiquer que le test est en erreur avec un message associé), sous JUnit 5 la classe a été remplacée par `org.junit.jupiter.api.Assertions` et offre les mêmes fonctionnalités.

Deuxième approche BDD

¥ 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 trois 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 *ç* mode d'emploi *È* 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 cela, on peut utiliser la librairie [JUnitParams](#) ou utiliser l'annotation `@Parameters` de Junit 4.

1. Refactorisez les huit tests
2. Le TP est terminé

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 *ç* programmeur lettré *È* dont l'œuvre n'est pas réalisable par un *b*ête générateur de tests.