L3 Info – INF5A1
Jour 4
le 27 septembre 2021

Tests Unitaires

Sommaire

- Principe des tests unitaires
- Test Driven Development
- JUnit
- Utilsation d’un Mock pour s’affranchir d’une dépendance

Test Unitaire (T.U. ou U.T.)

- Un test unitaire teste une partie très circonscrite du code, en vérifiant qu'elle correspond à sa spécification.
- Un test réellement unitaire ne concerne qu'une classe et une méthode à la fois.
- Le fait qu'il ne concerne qu'une partie très circonscrite du code rend le débogage plus facile.

Test Driven Development

- Idéalement, les tests sont écrits dès la spécification de l'application, avant même l'implémentation
- On écrit alors le code en le testant au fur et à mesure. Toute méthode doit satisfaire son test avant de passer à la prochaine
- Ecrire les tests en amont permet aussi de ne pas se faire influencer par l'implémentation, ce qui pourrait rendre le test biaisé (ne testant que partiellement la spécification).

Exemple

- Une classe fournit une méthode fournissant la valeur absolue d'un nombre.
- Une façon de vérifier qu'elle ne déroge pas à ses spécifications est de vérifier :
  - qu'elle renvoie 1 lorsqu'elle reçoit -1
  - qu'elle renvoie 1 lorsqu'elle reçoit 1

Que prouve un test ?

- Dans de nombreux cas, il est impossible de tester exhaustivement une méthode
- Par exemple, la méthode valeurAbsolue peut prendre en entrée une infinité de valeurs
- Dans ces cas, on teste uniquement sur un ensemble de valeurs que l'on juge représentatif.
- Ainsi, un test qui échoue prouve que le code est incorrect, mais un test qui réussit ne prouve pas formellement que le code est correct...

Non régression du code

- Des modifications faites sur le code d'une méthode (optimisation, extension, etc.) risquent de compromettre son fonctionnement. Il s'agit d'une régression
- Continuer de faire des tests sur ces dernières permet de s'en rendre compte, et de les corriger immédiatement
- Une régression peut aussi survenir sur une méthode sans que l'on ait modifié cette dernière, en raison d'une dépendance à une autre méthode qui a été modifiée. Là encore, les tests unitaires permettent de le vérifier.

JUnit 4

- JUnit est un framework de tests unitaires pour Java
- Il est désormais intégré aux IDE tels que NetBeans
- Il permet de créer pour chaque classe que l'on souhaite tester une classe parallèle dédiée à son test.

JUnit : fonctionnement

- Par convention, la classe de test associée à la classe Toto est nommée TotoTest
- De même, au sein de la classe de Test, une méthode de test concerne une seule méthode, et porte le nom de la méthode testée précédé de « test » (par ex. testValeurAbsolue)
- Chaque méthode de test est de type void, ne prend pas de paramètre, et est précédée par @Test
- Il n'y a pas besoin de main() au niveau des tests, car les méthodes de test sont reconnues et lancées automatiquement par JUnit.

Contenu d’une classe de test

```java
public class DemoTestsUnitairesTest {

    @BeforeClass
    public static void setUpClass() {
    }

    @AfterClass
    public static void tearDownClass() {
    }

    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }

    @Test
    public void testUneCertaineMethode() {
        fail("Test non encore écrit");
    }

}
```

Annotations

- @BeforeClass : du code static lancé lors de la création de la classe de test
- @AfterClass : du code static lancé avant de clore le test de cette classe
- @Before : du code (non static) lancé avant l'exécution de chacune des méthodes de test
- @After : du code (non static) lancé après l'exécution de chacune des méthodes de test
- @Test : une méthode effectuant un test unitaire

Assertions

- Une assertion est une instruction particulière qui effectue une vérification au sein d'une méthode de test
- Un test unitaire est satisfait si toutes ses assertions sont satisfaites
- Différentes assertions existent :
  - assertTrue(expression booléenne);
  - assertFalse(expression booléenne);
  - assertEquals(objet1, objet2)

Exemple d’un test de toString()

```
@Test
public void testToString() {
    Point p = new Point(1, 2);
    assertEquals(p.toString(), "(1.0,2.0)");
}
```

Sur quelles valeurs tester ?

- Les paramètres d'entrée d'une méthode offrent une combinatoire qu'il est souvent impossible de couvrir. Un simple paramètre entier offre déjà 2^32 valeurs possibles.
- Il est donc nécessaire de cibler certaines valeurs représentatives (notamment s'il est possible d'identifier des plages de valeurs correspondant à un comportement identique).
- Par exemple, pour un entier :
  - un nombre négatif
  - zéro
  - un nombre positif
- Pour une méthode prenant 2 entiers, on peut éventuellement tester les 3^*2* combinaisons correspondant au cas précédent (--, -0, -+, 0-, 00, 0+, +-, +0, ++).

Question

- Comment testeriez-vous une méthode de génération d'un nombre pseudo-aléatoire ?

Question

- Comment testeriez-vous une méthode de génération d'un nombre pseudo-aléatoire compris entre 0 et 1 ?
- Idée :
  - poser n = 10000, et m=100
  - créer un tableau de m cases correspondant aux plages de longueur 1/m de l'intervalle [0,1[(dans notre exemple, la case i correspond à une valeur située entre 1/100 x i et 1/100 x (i+1)
  - faire n tirages, et faire tableau[valeurAléatoire/m]++
  - Enfin, vérifier l'homogénéité du tableau. chacune de ses valeurs devrait être proche de n/m = 100. Par exemple, on pourrait faire échouer le test si une valeur du tableau est inférieure à 50 ou supérieure à 150.

Notion de couverture de test

- La couverture de test, aussi appelée couverture de code, correspond au taux de code source testé
- Une couverture de 100% signifierait que chaque ligne de code a été parcourue par les tests
- L'écriture de tests prend du temps, il faut parfois trouver un compromis, et se focaliser sur les partie « métier » fondamentales
- Dans NetBeans : Outils → Modules d'extension → recherche sur « jacocoverage » et l'installer. Ensuite, clic doit sur un projet et « test with Jacocoverage »

Limiter les dépendances ou s'en affranchir

- Un test unitaire doit idéalement concerner une seule classe.
- Cependant, dans de nombreux cas, une méthode fait appel à d'autres méthodes situées dans d'autres classes.
- Il est alors difficile d'isoler réellement un test
- Méthode possible :
  - commencer écrire les tests pour les méthodes sans dépendance
  - puis les méthodes utilisant ces dernières. Ainsi, si leur test ne passe pas, on sait que la cause n'est pas l'une de leurs dépendances

Dépendance (suite): Utilisation d'un Mock

- Parfois, une dépendance n'est pas encore prête pour une raison telle que :
  - présence de bug
  - nécessite des données externes non encore disponibles (BD)
  - en cours de développement par une autre équipe
- On peut alors simuler partiellement (i.e. pour certaines valeurs seulement) le comportement de cette dépendance, pour tester notre classe.
- Un Mock est une classe ad-hoc qui remplace une dépendance

Créer un Mock

- Pour pouvoir créer et utiliser facilement un mock, il faut que le type de la dépendance soit une interface plutôt qu'une classe.
- On peut alors créer une structure un peu à la façon de Proxy :
  - interface uneDependance
  - classe implémentant cette dépendance (celle qui n'est pas encore disponible pour le moment)
  - classe implémentant « en dur » cette dépendance pour quelques valeurs seulement
- Le code de l'application s'appuie entièrement sur l'interface, pas sur une implémentation particulière

Utiliser un Mock dans un test

- Dans la classe de test, on instancie le Mock au lieu de la (future) classe d'implémentation
- On crée le test uniquement sur des valeurs correspondant à ce que sait faire le Mock
- Ainsi, on a un test de notre classe, sans dépendre du tout de la dépendance
- À terme, on pourra intégrer au test la vraie classe d'implémentation lorsqu'elle sera disponible (ce test deviendra plutôt un test d'intégration).