Comwork

Formation sur les tests

www.comwork.io

idriss.neumann@comwork.io

Comwork.io SASU
128 rue de la Boétie 75008 Paris SIRET : 83875798700014
Comwork

Au programme

- Généralités
  - Pourquoi écrire des tests ?
  - Différence entre TI et TU
  - Démarche du TDD
- Le framework JUnit
  - Définition
  - Structure d'un test case
  - Les différentes assertions
  - Assertions sur exception
- L'alternative AssertJ
- Les objets simulacres
  - Différences entre mock et stub
  - Liste des mocking toolkits
- Le framework Mockito
  - Créer un mock
  - Prédire le résultat d'une méthode d'un mock
  - Faire des assertions sur l'invocation d'un mock
  - L'utilisation des annotations
  - Les arguments captors
  - Les espions (spy)
- Mise en pratique en Java
- Equivalences en Python unittest
- Mise en pratique en Python

Généralités

Pourquoi écrire des tests ?

Écrire des scénarios de tests automatisés (tant des tests unitaires que des tests d'intégration) constitue une étape incontournable du cycle de développement logiciel dans un cadre professionnel.

Bien que cette tâche puisse parfois sembler rébarbative, elle est indispensable pour produire des développements de bonne qualité, notamment parce que cela permet entre autres de :

- s'assurer que l'on maîtrise les aspects fonctionnels de l'application (les besoins métier à satisfaire) ;
- détecter au plus tôt les anomalies éventuelles avant la livraison d'un projet aux utilisateurs ;
- détecter d'éventuelles régressions, suite à l'implémentation de nouvelles fonctionnalités, en automatisant l'exécution de l'ensemble de tous les scénarios de tests implémentés sous forme de tests unitaires ou tests d'intégration, de façon à ce qu'ils soient exécutés régulièrement (à minima une fois par jour par exemple).

Plus l'ensemble de vos codes sera couvert par les tests, plus vous serez à même de détecter les régressions, de maîtriser l'ensemble des fonctionnalités de votre application et ainsi de la rendre plus maintenable.

Généralités

Différences entre tests unitaires et tests d'intégration

De manière générale, le test unitaire a pour but de tester une partie précise d'un logiciel (d'où le nom "unitaire") voire une fonction précise, en essayant de couvrir au maximum la combinatoire des cas fonctionnels possibles sur cette portion de code. Cependant, cela ne suffit pas à différencier un test unitaire d'un test d'intégration qui, lui aussi, peut se contenter de tester une portion précise du code.

La différence fondamentale est que le test unitaire doit **uniquement** tester le code et ne doit donc pas avoir d'interactions avec des dépendances externes du module testé telles que des bases de données, des Web services, des appels à des procédures distantes, la lecture ou écriture de fichiers ou encore d'autres fonctions. Les dépendances devront être simulées à l'aide de mécanismes tels que les bouchons ou encore les mocks.

Concernant les tests d'intégration, ceux-ci permettent, de manière générale, de vérifier la bonne intégration de la portion de code testée dans l'ensemble du logiciel et de ses dépendances. Il peut s'agir tout aussi bien de vérifier la bonne insertion de données dans un SGBDR sur une petite portion de code que de scénarios complets du logiciel, correspondant à des enchaînements d'appels de fonctions, de Web services... (c'est ce que l'on appelle aussi des tests « bout en bout »).

Généralités

La démarche du TDD

TDD est un acronyme pour « Test Driven Development » ou encore le développement dirigé par les tests en français. Il s'agit d'une démarche qui recommande au développeur de rédiger l'ensemble des tests unitaires avant de développer un logiciel ou les portions dont il a la charge.

Cette démarche permet de s'assurer que le développeur ne sera pas influencé par son développement lors de l'écriture des cas de tests afin que ceux-ci soient davantage exhaustifs et conformes aux spécifications. Ainsi, cette démarche permet aussi de s'assurer que le développeur maîtrise les spécifications dont il a la charge avant de commencer à coder.

L'approche TDD préconise le cycle court de développement en cinq étapes :

1. écrire les cas de tests d'une fonction ;
2. vérifier que les tests échouent (car le code n'existe pas encore) ;
3. développer le code fonctionnel suffisant pour valider tous les cas de tests ;
4. vérifier que les tests passent ;
5. refactoriser et optimiser en s'assurant que les tests continuent de passer.

Le framework JUnit

Qu’est-ce que JUnit

JUnit est un framework Java prévu pour la réalisation de tests unitaires et d'intégration.

JUnit est le framework le plus connu de la mouvance des frameworks xUnit implémentés dans de nombreuses technologies (nous pourrons également citer Unittest pour Python, PHPUnit pour PHP ou encore xUnit.NET pour C# et .NET par exemple).

JUnit permet de réaliser :

- des TestCase qui sont des classes contenant des méthodes de tests ;
- des TestSuite qui permettent de lancer des suites de classes de type TestCase.

Voir la slide suivante pour avoir la structure globale d’une classe de test écrite en JUnit >= 4.

Le framework JUnit

Structure globale

```
import org.junit.Before;
import org.junit.After;
import org.junit.BeforeClass;
import org.junit.AfterClass;
import org.junit.Test;

public class MaClasseDeTest {

    /** Pre et post conditions */

    @BeforeClass
    public static void
setUpBeforeClass() throws Exception
{
        // Le contenu de cette
méthode ne sera exécuté qu'une fois
avant toutes les autres méthodes
avec annotations
        // (y compris celles ayant
une annotation @Before)
    }

    @AfterClass
    public static void tearDownClass()
throws Exception {
        // Le contenu de cette méthode
ne sera exécuté qu'une fois après
toutes les autres méthodes avec
annotations
        // (y compris celles ayant une
annotation @After)
    }

    @Before
    public void setUp() throws
Exception {
        // Le contenu de cette méthode
sera exécuté avant chaque test (méthode
avec l'annotation @Test)
    }

    @After
    public void tearDown() throws
Exception {
        // Le contenu de cette méthode
sera exécuté après chaque test (méthode
avec l'annotation @Test)
    }

    /** Cas de tests */
    @Test
    public void testCas1() {
        // Code contenant l'exécution du
premier scénario avec les assertions
associées
    }

    @Test
    public void testCas2() {
        // ...
    }
}
```

Le framework JUnit

Les différents types d'assertions

En informatique, une assertion est une expression qui doit être évaluée *vrai* ou faire échouer le programme ou le test en cas d'échec (en levant une exception ou en mettant fin au programme par exemple).

Dans le cas de JUnit et des tests d'une manière générale, on peut assimiler une assertion à une vérification et, en cas d'échec, une exception spécifique est levée (`java.lang.AssertionError`pour JUnit).

Voici une liste non exhaustive des fonctions d'assertions en JUnit :

- `assertEquals` : vérifier l'égalité de deux expressions ;
- `assertNotNull` : vérifier la non-nullité d'un objet ;
- `assertNull` : vérifier la nullité d'un objet ;
- `assertTrue` : vérifier qu'une expression booléenne est vraie ;
- `assertFalse` : vérifier qu'une expression booléenne est fausse ;
- `fail` : échouer le test si cette assertion est exécutée.

Ces méthodes sont surchargées pour de nombreux objets. Par exemple, la méthode assertEquals va être surchargée pour tous les types primitifs et pour les classes implémentant la méthode equals (String par exemple). Il est également possible de fournir un message d'erreur en cas d'échec d'une assertion (en premier paramètre).

Le framework JUnit

Exemples d'assertions simples

Imaginons que l'on veuille tester la classe suivante :

```
public class Addition {
    Integer nb1, nb2;

    public Addition(Integer nb1, Integer nb2){
        this.nb1 = nb1;
        this.nb2 = nb2;
    }
    public Integer somme(){
        return nb1 + nb2;
    }
    public Integer getNb1() {
        return nb1;
    }
    public Integer getNb2() {
        return nb2;
    }
}
```

Voici un exemple de fonction de test :

```
@Test
public void testCasAdditionNominal() {
    Addition add = new Addition(1, 2);
    // vérification, on a bien valorisé l'attribut nb1
    assertEquals(1, add.getNb1());
    // vérification, on a bien valorisé l'attribut nb2
    assertEquals(2, add.getNb2());
    // vérification sur la méthode somme()
    assertEquals("Erreur dans le calcul de la somme",
3, add.somme());
}
```

Remarque : les imports statiques sont préconisés pour les méthodes statiques provenant des frameworks de tests présentés dans cette formation.

Le framework JUnit

Exemple d'assertions sur des exceptions

Il est possible de vérifier la levée d'une exception grâce à l'annotation `@Test(expected=<nom>.class)` :

```
// Ce test va réussir
@Test(expected = java.lang.NullPointerException.class)
public final void testException(){
    Long variablePasInstanciee = null;
    variablePasInstanciee.toString();
}

// Ce test va échouer
@Test(expected = java.lang.NullPointerException.class)
public final void testException2(){
    Long variableInstanciee = 1L;
    variableInstanciee.toString();
}
```

L’alternative AssertJ

Des assertions en fluent coding style pour améliorer la lisibilité des tests

Une alternative aux assertions JUnit sont les assertions d’AssertJ, une librairie mettant à disposition des assertions tout en adoptant le “fluent coding style”.

Voici des exemples d’assertions issues de la documentation officielle d’AssertJ:

```
// unique entry point to get access to all assertThat methods and utility methods (e.g. entry)
import static org.assertj.core.api.Assertions.*;

// basic assertions
assertThat(frodo.getName()).isEqualTo("Frodo");
assertThat(frodo).isNotEqualTo(sauron)
                 .isIn(fellowshipOfTheRing);

// String specific assertions
assertThat(frodo.getName()).startsWith("Fro")
                           .endsWith("do")
                           .isEqualToIgnoringCase("frodo");

// collection specific assertions
assertThat(fellowshipOfTheRing).hasSize(9)
                               .contains(frodo, sam)
                               .doesNotContain(sauron);
```

L'alternative AssertJ

Des assertions en fluent coding style pour améliorer la lisibilité des tests

```
// using extracting magical feature to check fellowshipOfTheRing characters name :)
assertThat(fellowshipOfTheRing).extracting("name")
                               .contains("Boromir", "Gandalf", "Frodo", "Legolas")
                               .doesNotContain("Sauron", "Elrond");

// Extracting with Java 8 love (type safe)
assertThat(fellowshipOfTheRing).extracting(TolkienCharacter::getName)
                               .contains("Boromir", "Gandalf", "Frodo", "Legolas")
                               .doesNotContain("Sauron", "Elrond");

// filter collection before assertion
assertThat(fellowshipOfTheRing).filteredOn("race", HOBBIT)
                               .containsOnly(sam, frodo, pippin, merry);


// filter collection with Java 8 Predicate
assertThat(fellowshipOfTheRing).filteredOn(character -> character.getName().contains("o"))
                               .containsOnly(aragorn, frodo, legolas, boromir);

// combining filtering and extraction (yes we can)
assertThat(fellowshipOfTheRing).filteredOn(character -> character.getName().contains("o"))
                               .containsOnly(aragorn, frodo, legolas, boromir)
                               .extracting(character -> character.getRace().getName())
                               .contains("Hobbit", "Elf", "Man");
```

Les objets simulacres

Différences entre les simulacres (mocks) et bouchons (stubs)

Bien que les concepts d'objets simulacres (mocks) et de bouchons (stubs) soient proches, on peut noter les différences suivantes :

Les bouchons (stubs) sont basés sur une vérification d'état. Ils permettent de s'abstraire d'une dépendance en fournissant des méthodes, Web services, base de données, ayant les mêmes entrants que cette dépendance, mais en fournissant une réponse contrôlée ou prédéfinie.

Ainsi l'utilisation de bouchons permet de :

- tester uniquement la méthode que l'on souhaite tester sans dépendre du résultat fourni par la dépendance en question ;
- de développer autour de dépendances non implémentées.

Les objets simulacres (mocks), eux, sont basés sur une vérification de comportement. Contrairement aux bouchons, il ne peut s'agir que d'objets, mais dont le comportement est également totalement contrôlé. Contrairement aux bouchons qui sont appelés de manière transparente pour l'appelant, l'utilisation de mocks repose sur l'injection de dépendance.

Le gros avantage des mocks par rapport aux simples bouchons : permet d'être mis en place très rapidement à l'aide d'API comme Mockito au sein même d'un testCase JUnit et permet de faire des assertions sur les appels aux méthodes du mock (vérifier qu'on appelle bien telle méthode avec tels arguments par exemple).

Les mocks sont donc plus adaptés à l'écriture de tests unitaires ou l'on ne charge que la classe à tester en essayant de passer par tous les chemins possibles et de faire le maximum d'assertions sur les cas de tests possibles.

Les bouchons, quant à eux, sont davantage adaptés à la mise en place de tests d'intégration où l'on charge une partie de l'environnement des portions de code testées (par exemple : de vrais appels HTTP à des Web services bouchonnés dans le cadre d'un projet SOA).

Les objets simulacres

Les différents framework en Java permettant de faire des mocks

Voici une liste non exhaustive des frameworks permettant de mettre en place des objets simulacres en Java :

- Mockito
- EasyMock
- JMock
- PowerMock
- JMockit

A notre que Mockito est sans doute le plus populaire aujourd'hui. A noter également que PowerMock est essentiellement utilisé en tant que framework complémentaire permettant de mocker des méthodes statiques (ce que ne permet pas les autres frameworks) et propose aux choix la syntaxe EasyMock ou Mockito.

Le framework Mockito

Créer un mock et prédire son résultat

Créer un mock

```
import static org.mockito.Mockito.*;
ClasseAMocker objetMock = mock(ClasseAMocker.class)
```

Prédire le résultat d'une méthode d'un mock

```
import static org.mockito.Mockito.*;
import static org.mockito.Matchers.*;

when(objetMock.methode(any(ClassArgument.class))).thenReturn(objetResultat);
when(objetMock.methode(eq(valeurArgument))).thenReturn(objetResultat);
when(objetMock.methode(valeurArgument)).thenReturn(objetResultat);
```

Le framework Mockito

Faire des assertions sur les invocation de mocks

Faire une assertion sur l'invocation d'un mock

```
import static org.mockito.Mockito.*;
import static org.mockito.Matchers.*;

verify(objetMock).methode(valeurArgument);
verify(objetMock).methode(eq(valeurArgument));
verify(objetMock).methode(any(ClassArgument.class));
verify(objetMock, times(0)).methode(any(ClassArgument.class)); // vérifier que la méthode n'est jamais invoquée
verify(objetMock, times(1)).methode(any(ClassArgument.class)); // vérifier que la méthode n'est invoquée qu'une fois
```

Les matchers

Voici une liste non exhaustive des matchers utilisés dans les instructions verify ou when :

- "eq" pour equals : il est implicite lorsque l'invocation du mock ne fait pas appel à d'autres matchers
- "any" et ses dérivés "anyString", "anyLong", "anyListOf(...)" : **il est préconisé sur ce projet de ne pas les utiliser si vous avez la connaissance de votre jeu de données.**
  Pourquoi ? Car il est préférable de maitriser son jeu de données et de ne pas faire des tests unitaires hasardeux.

Le framework Mockito

Les arguments captor

Ils servent à faire des assertions plus approfondies sur les paramètres passés à un objet simulacres lors d'une assertion de type "verify". Ces captors sont principalement utiles lorsque nous n'avons pas connaissance à l'avance de la référence de l'objet qui va être passé en paramètre (exemple : construction d'une liste au sein de la méthode testée qui va invoquer le mock avec la référence de cette liste). **Dans le cas contraire, il est préconisé sur ce projet d'utiliser le matcher "eq" avec la référence de l'objet.**

```
import org.mockito.ArgumentCaptor;
import static org.mockito.Mockito.*;
import static org.mockito.Matchers.*;

ArgumentCaptor<ObjetParametreAVerifier> captor = ArgumentCaptor.forClass(ObjetParametreAVerifier.class);
verify(ObjetMock).methodeAVerifier(eq(valeur), captor.capture());
ObjetParametreAVerifier objetAVerifier = captor.getValue();
// assertions sur les méthodes de objetAVerifier
```

Pour créer un captor sur un paramètre de type `List<?>`, il faut privilégier l'utilisation des annotations :

```
@Captor
ArgumentCaptor<List<String>> captor;
```

Le framework Mockito

Les espions (Spy)

Cette fois-ci, on s'écarte du principe des simulacres puisqu'il s'agit de faire des assertions sur l'invocation de méthode d'instance réelles d'objets. En effet, Mockito permet également de faire des "verify" sur des "espions" à l'instar des mocks.

Exemple d'espion sur une liste (exemple tirée de la documentation de Mockito) :

```
import static org.mockito.Mockito.*;

@Test
public void whenSpyingOnList_thenCorrect() {
    List<String> list = new ArrayList<String>();
    List<String> spyList = spy(list);
    spyList.add("one");
    spyList.add("two");
    verify(spyList).add("one");
    verify(spyList).add("two");
    assertEquals(2, spyList.size());
}
```

Mise en pratique

En Python

La suite se passe sur ce repo git (à cloner et suivre le README.md): https://gitlab.comwork.io/comwork_training/unit_tests