FORMATION TESTS UNITAIRES

JAVA / JS

Capgemini

SOMMAIRE

I. POURQUOI « PERDRE DU TEMPS » A ECRIRE DES TESTS ?

LE POINT SUR LES TESTS :

LES A PRIORI :

- Pas utile si mon code marche !!
- Ca marche sur ma machine !
- Je perds du temps à écrire du code inutile !!!!
- Tester c'est douter …

LES FAITS :

- Fait partie intégrante de la tâche du développeur
- +/- 40% du temps d'un développeur
- Nécessite autant d'analyse que la tâche de développement
- Souvent besoin de maîtriser une technologie supplémentaire pour bien couvrir son code
- Indice de la qualité de code. Si c'est dur à tester, c'est probablement mal codé

LE BUT :

- Détecter les bugs !!!!
- S'assurer de la stabilité du code en cas de modification
- Pouvoir rejouer la suite de tests à volonté & automatiquement
- Assurer au client une « qualité » de code
- Gain de temps sur le long terme
- Le sentiment du devoir accompli !!

LE TEST « PERSO »

- Mon code

```java
/**
 * Add two integers
 *
 * @param a the first integer
 * @param b the second integer
 * @return a +b
 */
public Integer add(Integer a, Integer b) { return a + b;}
```

Test reproductible ?	❌
Test automatisable ?	❌
Garantie de la couverture ?	❌
Facilement compréhensible ?	❌
Documenté ?	❌
Conclusion	❌

- Mon test

```java
public class Main {

    public static void main(String[] args) {
        Calculator calculator = new Calculator();
        Integer result = calculator.add(3, 2);
        System.out.println("a = " + 3);
        System.out.println("b = " + 2);
        System.out.println("3 + 2 = " + result);
    }
}
```

```
Executing task 'Main.main()'...
> Task :compileJava
> Task :processResources NO-SOURCE
> Task :classes
> Task :Main.main()
a = 3
b = 2
3 + 2 = 5
```

- Après, on supprime les println (Quand on y pense !)

AUTREMENT DIT :

LE TEST « UNITAIRE »

Définition :

- En programmation informatique, le test unitaire est une procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion d'un programme (appelée « unité » ou « module »).
- On écrit un test pour confronter une réalisation à sa spécification. Le test définit un critère d'arrêt (état ou sorties à l'issue de l'exécution) et permet de statuer sur le succès ou sur l'échec d'une vérification.
- Grâce à la spécification, on est en mesure de faire correspondre un état d'entrée donné à un résultat ou à une sortie.
- Le test permet de vérifier que la relation d'entrée / sortie donnée par la spécification est bel et bien réalisée.

Mon code

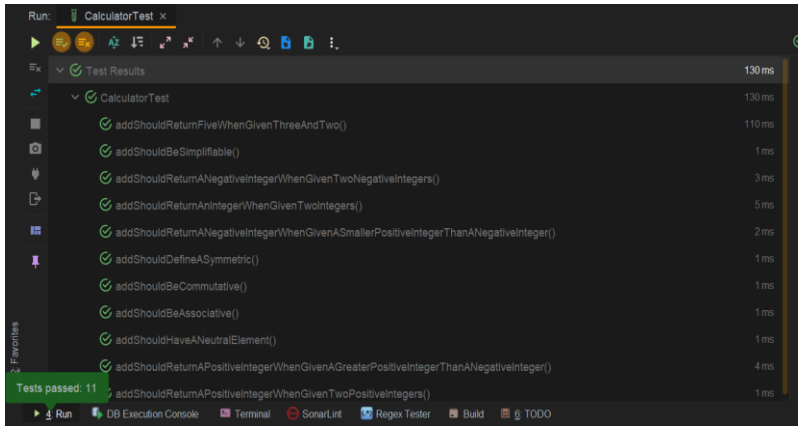
```
/**
 * Add two integers
 *
 * @param a the first integer
 * @param b the second integer
 * @return a +b
 */
public Integer add(Integer a, Integer b) { return a + b;}
```

- **(Analyse) Les propriétés significatives de l'addition :**
  - La somme de deux entiers donne un entier
  - La somme de deux entiers positifs donne un entier positif
  - La somme de deux entiers négatifs donne un entier négatif
  - La somme d'un entier positif ***a*** et d'un entier négatif ***b*** donne un entier positif si ***|a| > |b|***, négatif si ***|a| < |b|***
  - L'addition est commutative : ***a + b = b + a***
  - Elle est associative : ***(a + b) + c = a +(b + c)***
  - Elle est simplifiable : Si ***x + a = y + a*** alors ***x = y***
  - La somme d'un entier et 0 donne l'entier (élément neutre) : ***a + 0 = a***
  - Chaque nombre a un symétrique tel que défini : ***x + (-x) = -x + x = 0***

LE TEST « UNITAIRE »

- **Mon test**

```
class CalculatorTest {

   private Calculator calculator;

   @BeforeEach
   void setUp() {
      this.calculator = new Calculator();
   }

   @Test
   void addShouldReturnAnIntegerWhenGivenTwoIntegers() {
      Integer additionResult = calculator.add(3, 2);
      assertThat(additionResult.getClass()).isEqualTo(Integer.class);
      assertThat(additionResult instanceof Integer).isTrue(); // Alternative syntax
   }

   @Test
   void addShouldReturnAPositiveIntegerWhenGivenTwoPositiveIntegers() {
      int a = 3;
      int b = 2;
      Integer additionResult = calculator.add(a, b);
      if (a < 0 || b < 0)
         throw new IllegalArgumentException("This test needs two positive integers");
      assertThat(additionResult > 0).isTrue();
   }

   @Test
   void addShouldReturnANegativeIntegerWhenGivenTwoNegativeIntegers() {
      int a = -3;
      int b = -2;
      Integer additionResult = calculator.add(a, b);
      if (a > 0 || b > 0)
         throw new IllegalArgumentException("This test needs two negative integers");
      assertThat(additionResult < 0).isTrue();
   }
// Many other tests
```

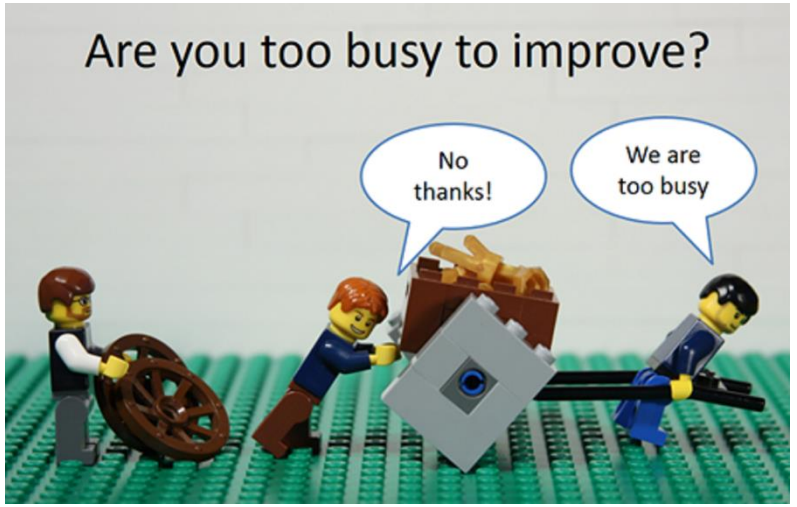


LE TEST « UNITAIRE »

- **Comparaison**

Test reproductible ?	✓
Test automatisable ?	✓
Garantie de la couverture ?	✓
Facilement compréhensible ?	✓
Documenté ?	✓
Conclusion	✓

- **SO ??**



LE TEST « D'INTEGRATION »

- **Définition :**
  - Dans le monde du développement informatique, le test d'intégration est une phase dans les tests, qui est précédée des tests unitaires et est généralement suivie par les tests de validation.
  - Dans le test unitaire, on vérifie le bon fonctionnement d'une portion d'un programme ; dans le test d'intégration, chacun des modules indépendants du logiciel est assemblé et testé dans l'ensemble.
  - Le test d'intégration permet également de vérifier l'aspect fonctionnel, les performances et la fiabilité du logiciel
  - le test d'intégration a pour cible de détecter les erreurs non détectables par le test unitaire.

- **Ou de façon plus parlante, c'est pour éviter ca :**

- **Exemples :**
  - Tests DAO (Incluant la DB)
  - Différents modules d'une application sur la même page
  - Effets de bords …

LES AUTRES TESTS

- **Tests IHM**
  - ▶ Validation de l'UI de l'application
    - Ex : Framework Selenium

- **Tests fonctionnels :**
  - ▶ Souvent établi par le spécificateur
  - ▶ Vérifie que le comportement est conforme à la spécification
- **Tests d'acceptance (UAT) :**
  - ▶ Livraison et déploiement de l'application
  - ▶ Validation par l'utilisateur final ou un représentant (PO)
- **Tests de performance :**
  - ▶ Permet de tester comment l'application résiste à certaines contraintes de charge, de saturation etc.
    - Ex : Framework Gatling

- **Tests de sécurité :**
  - ▶ Evalue la sécurité d'une application et sa résistance à l'intrusion
    - Ex : ZAP, Wfuzz, Wapiti

- **Tests systèmes :**
  - ▶ Axé sur la partie matérielle
- **Tests utilisateur :**
  - ▶ Les utilisateurs utilisent l'application et font remonter des dysfonctionnements
  - ▶ Très utilisés dans certains milieux
    - Ex : Open / Closed Beta
- **Et bien d'autres :**

COMMENT ECRIRE UN TEST ?

1) ANALYSE

- Déterminer les cas significatifs qui permettent de couvrir la fonctionnalité testée
- Ecrire un test pour chaque cas
- Tester les « bornes » et cas particuliers
- Ne pas oublier la gestion des erreurs

2) PREPARER

- Construire les éléments du résultat attendu, généralement nommé« expected »
  - Ex :
```
Integer expectedResult = 5;
```
- Peut être construit, désérialisé depuis un fichier XML ou JSON, mais ne doit pas être issu d'un autre composant de l'application (BDD, autre classe, etc.)

3) AGIR

- Appeler la méthode testée avec les paramètres déterminés par votre analyse
- Stocker le résultat éventuel de la méthode, généralement nommé « actual »
  - Ex :
```
Integer actualResult = calculator.add(5, 0);
```

4) AFFIRMER

- Utiliser une assertion qui fera réussir ou échouer votre test
  - Ex :
```
assertEquals(expectedResult, actualResult);
```
- Eviter de faire plusieurs assertions, sauf si elles doivent être liées (et encore..)
  - Ex : Affirmer qu'un retour est false ET que la liste des messages d'erreurs contient un et un seul message ET qu'il est bien celui attendu (doit rester rare et peut également se réaliser en trois tests)

PERTINENCE ET COUVERTURE DE TEST

PERTINENCE

- Vision d'ensemble de la fonctionnalité, issue de l'analyse
- Représente l'efficacité de votre test au vu de la fonctionnalité
- Difficilement estimable, repose sur la qualité de l'analyse, du développement, de l'expérience du développeur, etc.
- Tester les « edge cases » et « corner cases »

COUVERTURE (CODE COVERAGE)

- Représente le pourcentage de code couvert par les tests
- Indicateur quantitatif et non qualitatif
  - 100% de coverage != 100% pertinent
- Doit être généralement au-dessus des 80%
  - Certaines méthodes ne sont pas testables « unitairement » et seront couvertes pas d'autres types de tests
  - Inutile de tester du code issu de librairies/ frameworks utilisé dans le code (déjà testé par l'auteur)
    - → chute normale du pourcentage de couverture

FRAMEWORKS ET OUTILS JAVA

- **JUnit**
  - Framework de test le plus utilisé
  - Permet de gérer, structurer et écrire les tests unitaires JAVA
  - Version 5 dispo, v4 encore très utilisée

- **Principales annotations**
  - @BeforeAll / @AfterAll
    - Exécute du code avant / après TOUS les tests
  - @BeforeEach / @BeforeAll
    - Exécute du code avant / après CHAQUE test
  - @Test
    - Définit une méthode comme test unitaire
  - @Order
    - Permet d'imposer un ordre d'exécution
  - @DisplayName
    - Permet d'afficher un nom différent (pour caractères spéciaux, emoji, etc.) dans le rapport et dans certains IDE

- @ParametrizedTest
  - → Permet de définir un « jeu de données » qui va être appliqué sur les paramètres.

  - @ValueSource
    - → Définit une source directement dans un tableau

```
@ValueSource(ints = { 1, 2, 3 })
```

  - @EnumSource
    - → La source des paramètres est une enum
  - @MethodSource
    - → La source des paramètres est une méthode

```
@ParameterizedTest
@MethodSource("stringProvider")
void testWithExplicitLocalMethodSource(String argument) {
    assertNotNull(argument);
}

static Stream<String> stringProvider() {
    return Stream.of("apple", "banana");
}
```

  - @CsvSouce / @ScvFileSource
    - → Définit un tableau de valeurs séparés par une virgule ou un fichier csv comme source

FRAMEWORKS ET OUTILS JAVA

- **Assertions JUnit**
  - @AssertEquals / @AssertNotEquals
    - Assertion sur l'égalité entre les deux **objets**
  - @AssertTrue / AssertFalse
    - Assertion sur la valeur d'un booléen
  - @AssertAll
    - Assertions imbriquées, si l'une échoue la chaîne s'arrête. (utilise les lambdas)
  - @AssertSame / @AssertNotSame
    - Assertion sur l'égalité entre les **références** des deux objets.
  - @AssertLinesMatch
    - Assertion sur l'égalité entre deux List<String>
  - @AssertIterableEquals
    - Assertion sur l'égalités entre deux collections implémentant l'interface *Iterable*
  - @AssertNull / @AssertNotNull
    - Assertion sur la nullité du paramètre
  - @AssertThrow / @AssertNotThrow
    - Assertion sur le throw ou non d'une exception (lambdas)

- **AssertJ**
  - Framework d'assertion permettant de les rendre plus « parlantes »

```
assertEquals(expectedResult, actualResult);
```

Devient

```
assertThat(actualResult).isEqualTo(expectedResult);
```

  - Implémente le design pattern Fluent pour chainer les assertions

```
assertThat(frodo.getName()).startsWith("Fro")
      .endsWith("do")
      .isEqualToIgnoringCase("frodo");
```

LET’S PRACTICE AND DO SOME MAGIC

OUTILS ET NOTIONS DE TESTS AGNOSTIQUES

- **MOCKS et autres « simulacres »**
  - Un test pour rester « unitaire » ne doit pas dépendre de librairies ou classes tierces.
    - Les classes tierces sont testées ailleurs dans l'application
    - Les librairies tierces sont testées par leurs équipes de développement
  - Le code d'une méthode « valide » peut être « pollué » par l'appel à une méthode qui ne l'est pas
- **Tester une classe sans dépendance est assez facile mais comment faire lorsqu'il y en a une ou plusieurs ?**
  - Le comportement et l'appel à une dépendance externe doit donc être « simulé »
  - Il existe plusieurs implémentations de ces objets de tests appelé *trop généralement* mocks :
    - Dummy, Stub, Fake, Spy, Mock (et surement encore beaucoup d'autres)

- **Dummy (Fantôme)**
  - Un Dummy est un objet de test avec aucune logique, qui permet juste de compiler (Injection de dépendance, etc)
  - Toutes ses méthodes doivent renvoyer une exception pour faire remonter une éventuelle utilisation erronée.

```
public interface UserInterface {
   public String getPassword();
   public String getUsername();
}

public class BlogPost {

   private UserInterface user;
   private String postContent;

   public BlogPost(UserInterface user) { this.user = user;}

   public String calculateStatsFromContent() {
      String stats = "";
      // Calcul de statistiques et renvoi de la chaine
      return stats;
   }
}

public class DummyUser implements UserInterface {
   @Override
   public String getPassword() { throw new NullPointerException(); }
   @Override
   public String getUsername() { throw new NullPointerException();}
}
```

  - La méthode à tester *calculateStatsFromContent()* n'utilisant pas du tout l'objet *user*, on peut lui passer un Dummy

OUTILS ET NOTIONS DE TESTS AGNOSTIQUES

- **Stub (ou Bouchon)**
  - Un stub est un objet dont le comportement est **exactement** celui que j'attends.

```
public class BlogPost {

    private UserInterface user;
    private String postContent;

    public BlogPost(UserInterface user) {
        this.user = user;
    }

    public String getAuthor() {
        return user.getUsername();
    }
}

public class StubUser implements UserInterface {

    @Override
    public String getPassword() { return "password";}

    @Override
    public String getUsername() { return "john.doe@example.org";}
}
```

  - Je peux faire des assertions maîtrisées sur le retour des méthodes du Stub

OUTILS ET NOTIONS DE TESTS AGNOSTIQUES

- **Fake (Substitut)**
  - Un fake est un stub avec une implémentation partielle de la classe à tester.

```
public interface GenericDAO<T> {
   void save(T t);
   Optional<T> findById(Serializable entityId);
   List<T> findAll();
   void update(T t);
   void delete(T t);
}

public class BlogPostDAO implements GenericDAO<BlogPost> {

   private final OrmEntityManager em;

   public BlogPostDAO(OrmEntityManager em) { this.em = em;}

   @Override
   public void save(BlogPost blogPost) { em.persist(blogPost);}

   @Override
   public Optional<BlogPost> findById(Serializable entityId) {
      return Optional.ofNullable(em.get(BlogPost.class, 1L));
   }

   @Override
   public List<BlogPost> findAll() {
      return em.createQuery("SELECT post FROM BlogPost post");
   }

   @Override
   public void update(BlogPost blogPost) { em.saveOrUpdate(blogPost);}

   @Override
   public void delete(BlogPost blogPost) { em.delete(blogPost);}
}
```

```
public class BlogPostService {

   private BlogPostDAO dao;

   public BlogPostService(BlogPostDAO dao) { this.dao = dao;}

   public String convertPostContentToMarkdown(Long postId) {
      Optional<BlogPost> optionalPost = dao.findById(postId);
      if (optionalPost.isPresent()) {
         BlogPost blogPost = optionalPost.get();
         return markdown(blogPost.getPostContent());
      }
      throw new RuntimeException("No post found with this id");
   }
}
```

```
public class FakeBlogPostDAO implements GenericDAO<BlogPost> {

   private Map<Long, BlogPost> blogPosts = new Hashmap<>();

   @Override
   public void save(BlogPost blogPost) {
      blogPosts.put(blogPost.getId(), blogPost);
   }

   @Override
   public Optional<BlogPost> findById(Serializable entityId) {
      return Optional.empty();
   }

   @Override
   public List<BlogPost> findAll() { return new ArrayList<>(blogPosts.values());}

   @Override
   public void update(BlogPost blogPost) { blogPosts.put(blogPost.getId(), blogPost);}

   @Override
   public void delete(BlogPost blogPost) { blogPosts.remove(blogPost.getId());}
}
```

OUTILS ET NOTIONS DE TESTS AGNOSTIQUES

- **Spy (Espion)**

  - Un Spy va avoir le même comportement qu'un Stub, mais il va nous permettre d'obtenir des informations supplémentaires, une fois le test effectué.

  - Un Spy n'est rien d'autre qu'un Stub qui enregistre des informations pendant le test que l'on pourra aller chercher par la suite.
    - Ex : compter le nombre de fois où une méthode a été appelée

```
public class SpyUser implements UserInterface {

   public int nbCalled = 0;

   @Override
   public String getPassword() { return "password";}

   @Override
   public String getUsername() {
      nbCalled++;
      return "john.doe@example.org";}
}
```

  - Je peux par la suite me servir des informations du spy dans mon test

```
@Test
void shouldCallGetUsernameOnlyOneTime() {
   // Appel de méthode appelant getUsername()
   Integer actualMethodCallNumber = spyUser.nbCalled;
   assertThat(actualMethodCallNumber).isEqualTo(1);
}
```

OUTILS ET NOTIONS DE TESTS AGNOSTIQUES

- **Mock (Simulacre)**
  - Nom attribué généralement à tous les objets précédents. On « mocke » (Abus de langage)
  - Un mock est un objet qui est une substitution complète de l'implémentation originale de la classe à tester
  - Donc pour créer un mock d'une classe, on reprend les mêmes méthodes et on ajoute du code à l'intérieur pour vérifier son comportement.
  - Un mock va plus loin qu'un simple stub. Il peut lever des exceptions s'il ne reçoit pas les bons appels.
  - Il peut également, comme un spy, faire des vérifications pendant l'exécution du processus pour être sûr que tout se passe comme prévu.
  - Le mock va s'occuper de faire lui-même les assertions. Vous n'avez donc plus à tester le mock.
  - Les mocks vont faire leurs propres tests pour savoir ce qu'ils testent (Mock - ception !).
  - Voici donc la principale différence entre un mock et un stub ou un fake : Il peut décider d'échouer.
  - Là où un stub ou un fake doit réussir car on effectue un test précis, un mock peut, par exemple, s'il n'a pas les bons arguments pour une dépendance, décider d'échouer.
  - Avec un mock on ne cherche pas une valeur de retour, mais on s'intéresse plutôt à la méthode qui a été appelée, avec quels arguments ou encore combien de fois elle a été appelée.

OUTILS ET NOTIONS DE TESTS AGNOSTIQUES

```java
public class MockUser implements UserInterface {

    private int actualNbMethodCalls = 0;
    private int expectedNbMethodCalls = 0;

    @Override
    public String getPassword() {return "password";}

    @Override
    public String getUsername() {
        actualNbMethodCalls++;
        return "john.doe@example.org";
    }

    public void setExpectedNbMethodCalls(int expectedNbMethodCalls) {
        this.expectedNbMethodCalls = expectedNbMethodCalls;
    }

    public Boolean verify() {
        if (actualNbMethodCalls != expectedNbMethodCalls)
            throw new RuntimeException(String.format(
                    "Actual number of calls %s, expected %s",
                    actualNbMethodCalls,
                    expectedNbMethodCalls
            ));
        return true;
    }
}
```

```java
@Test
void shouldCallGetUsernameOnlyOneTime() throws Exception {
    // Appel de méthode appelant getUsername()
    mockUser.setExpectedNbMethodCalls(1);
    assertThat(mockUser.getUsername()).isEqualTo("john.doe@example.org");
    mockUser.verify();
}
```

Test Results 84 ms
com.percallgroup.formationtest.CalculatorTest 84 ms
shouldCallGetUsernameOnlyOneTime() 84 ms

```java
@Test
void shouldFailIfExpectedMoreThanOneTime() throws Exception {
    // Appel de méthode appelant getUsername()
    mockUser.setExpectedNbMethodCalls(2);
    assertThat(mockUser.getUsername()).isEqualTo("john.doe@example.org");
    mockUser.verify();
}
```

Test Results — Import Tests from File... 78 ms
com.percallgroup.formationtest.CalculatorTest 78 ms
shouldFailIfExpectedMoreThanOneTime() 78 ms
Tests failed: 1 of 1 test – 82 ms

```
Actual number of calls 1, expected 2
java.lang.Exception: Actual number of calls 1, expected 2
```

- On peut par la suite utiliser toute cette logique dans notre TU

OUTILS ET FRAMEWORKS DE TESTS JAVA

- **MOCKITO**

  - Framework de création de mocks le plus utilisé en JAVA
  - Permet d'accélérer la phase de tests unitaires, au prix de la dépendance à la librairie
  - Peut être utilisé avec d'autres librairies pour accroître encore les possibilités de mock.

- **Utilisation des annotations Mockito**
  - Pour JUnit4, annoter la classe avec :

```
@RunWith(MockitoJUnitRunner.class)
class CalculatorTest {}
```

  - Pour JUnit5, annoter la classe avec :

```
@ExtendWith(MockitoExtension.class)
class CalculatorTest {}
```

- **Principales annotations**
  - @Mock
    - Permet de créer un mock de la dépendance utilisable directement dans les TU
  - @Spy
    - Permet de créer un spy de la dépendance utilisable directement dans les TU
  - @Captor
    - Permet de capturer un argument passé à un mock
  - @InjectMock
    - Permet de créer une instance de la classe et d'y injecter les mocks créés avec @Mock ou @Spy.

OUTILS ET FRAMEWORKS DE TESTS JAVA

- **Gérer les comportements des mocks avec Mockito**

  - When / Then
    - Permet de redéfinir le retour d'une méthode du mock suivant nos besoins :

```
@Test
void shouldReturnFalseWithMockedBehaviorAndTrueWithNormalBehavior() {
  Mockito.when(mockedList.add("mockedBehavior")).thenReturn(false);
  Mockito.when(mockedList.add("normalBehavior")).thenReturn(true);
  boolean mockedBehavior = mockedList.add("mockedBehavior");
  boolean normalBehavior = mockedList.add("normalBehavior");
  assertThat(mockedBehavior).isFalse();
  assertThat(normalBehavior).isTrue();
}
```

```
Test Results                                                    667 ms
  com.percallgroup.formationtest.CalculatorTest                 667 ms
    shouldReturnFalseWithMockedBehavior()                       667 ms
```

```
@Test
void shouldReturnFalseWithMockedBehaviorAndTrueWithNormalBehaviorAlternativeSyntax() {
  Mockito.doReturn(false).when(mockedList).add("mockedBehavior");
  Mockito.doReturn(true).when(mockedList).add("normalBehavior");
  boolean mockedBehavior = mockedList.add("mockedBehavior");
  boolean normalBehavior = mockedList.add("normalBehavior");
  assertThat(mockedBehavior).isFalse();
  assertThat(normalBehavior).isTrue();
}
```

```
Test Results                                                    913 ms
  com.percallgroup.formationtest.CalculatorTest                 913 ms
    shouldReturnFalseWithMockedBehaviorAndTrueWithNormalBehaviorAlternativeSyntax()  913 ms
```

- Permet également de simuler une exception :

```
@Test
void shouldThrowAnExceptionWhenGivenMockedBehaviorAndNotWithNormalBehavior() {
  Mockito.when(mockedList.add("mockedBehavior"))
    .thenThrow(IllegalStateException.class);
  Mockito.when(mockedList.add("normalBehavior")).thenReturn(true);
  boolean normalBehavior = mockedList.add("normalBehavior");
  assertThatIllegalStateException().isThrownBy(
    () -> mockedList.add("mockedBehavior")
  );
  assertThat(normalBehavior).isTrue();
}
```

```
Test Results                                                    733 ms
  com.percallgroup.formationtest.CalculatorTest                 733 ms
    shouldThrowAnExceptionWhenGivenMockedBehaviorAndNotWithNormalBehavior()  733 ms
```

- Peut aussi simuler une exception sur méthode sans retour ni paramètres :

```
@Test
void shouldThrowAnExceptionWhenVoidMethodCalled() {
  Mockito.doThrow(IllegalStateException.class).when(mockedList).clear();
  assertThatIllegalStateException().isThrownBy(() -> mockedList.clear());
}
```

```
Test Results                                                    645 ms
  com.percallgroup.formationtest.CalculatorTest                 645 ms
    shouldThrowAnExceptionWhenVoidMethodCalled()                645 ms
```

OUTILS ET FRAMEWORKS DE TESTS JAVA

- When / Then
  - Permet de définir une séquence de retours de la méthode du mock :

```
@Test
void shouldReturnTrueThenThrowAnException() {
   Mockito.when(mockedList.add("test"))
         .thenReturn(true)
         .thenThrow(IllegalStateException.class);
   assertThat(mockedList.add("test")).isTrue();
   assertThatIllegalStateException().isThrownBy(() -> mockedList.add("test"));
}
```

```
Test Results                                                    650 ms
  com.percallgroup.formationtest.CalculatorTest                 650 ms
    shouldReturnTrueThenThrowAnException()                      650 ms
```

  - Permet de redéfinir le comportement d'un spy :

```
@Test
void shouldReturnTrueWhenMockIsCalledThrowAnExceptionWithSpy() {
   List<String> spy = Mockito.spy(mockedList);
   mockedList.clear();
   Mockito.doThrow(IllegalStateException.class).when(spy).size();
   assertThat(mockedList.size()).isEqualTo(0);
   assertThatIllegalStateException().isThrownBy(spy::size);
}
```

```
Test Results                                                    889 ms
  com.percallgroup.formationtest.CalculatorTest                 889 ms
    shouldReturnTrueWhenMockIsCalledThrowAnExceptionWithSpy()   889 ms
```

  - Permet également de simuler le renvoi systématique du même élément :

```
@Test
void shouldAlwaysReturnTheSameString() {
   mockedList.clear();
   mockedList.add(1, "First element");
   mockedList.add(2, "Second element");
   mockedList.add(3, "Third element");
   Mockito.doAnswer(invocation -> "Any element").when(mockedList).get(1);
   Mockito.doAnswer(invocation -> "Any element").when(mockedList).get(2);
   Mockito.doAnswer(invocation -> "Any element").when(mockedList).get(3);
   assertThat(mockedList.get(1)).isEqualTo("Any element");
   assertThat(mockedList.get(2)).isEqualTo("Any element");
   assertThat(mockedList.get(3)).isEqualTo("Any element");
}
```

```
Test Results                                                    646 ms
  com.percallgroup.formationtest.CalculatorTest                 646 ms
    shouldAlwaysReturnTheSameString()                           646 ms
```

OUTILS ET FRAMEWORKS DE TESTS JAVA

- When / Then
  - Permet d'utiliser la méthode réelle basée sur le mock :

```
public class User implements UserInterface {

  @Override
  public String getPassword() {
    return "realUserPassword";
  }

  @Override
  public String getUsername() {
    return "realUserUsername";
  }
}
```

- On va pouvoir utiliser soit une méthode du mock, soit la méthode issue de la classe dont il est issu.

```
@Test
void shouldUseMockedAndRealMethods() {
  Mockito.when(user.getPassword()).thenReturn("passwordFromMock");
  Mockito.when(user.getUsername()).thenCallRealMethod();
  assertThat(user.getPassword()).isEqualTo("passwordFromMock");
  assertThat(user.getUsername()).isEqualTo("realUserUsername");
}
```

```
Test Results                                   663 ms
  com.percallgroup.formationtest.CalculatorTest  663 ms
    shouldUseMockedAndRealMethods()              663 ms
```

- @Captor
  - Capture un argument passé à la méthode d'un mock

```
@ExtendWith(MockitoExtension.class)
public class DemoMockito {

  @Mock
  private List<String> mockedList;

  @Captor
  ArgumentCaptor<String> argumentCaptor;

  @Test
  void shouldCaptureArgument() {
    mockedList.add("stringToCapture");
    Mockito.verify(mockedList).add(argumentCaptor.capture());
    assertThat(argumentCaptor.getValue()).isEqualTo("stringToCapture");
  }
}
```

```
Test Results                                   717 ms
  com.percallgroup.formationtest.DemoMockito   717 ms
    shouldCaptureArgument()                    717 ms
```

OUTILS ET FRAMEWORKS DE TESTS JAVA

- **ArgumentMatchers**
  - Avec When / Then on peut définir un retour pour un paramètre donnée
  - Avec les ArgumentMatchers on peut élargir cette définition à un type de paramètre.

```java
@Test
void shouldAlwaysReturnTheSameString() {
   mockedList.clear();
   mockedList.add(1, "First element");
   mockedList.add(2, "Second element");
   mockedList.add(3, "Third element");
   String stringToReturn = "Any element";
   Mockito.doAnswer(invocation -> stringToReturn)
      .when(mockedList).get(ArgumentMatchers.anyInt());
   assertThat(mockedList.get(1)).isEqualTo(stringToReturn);
   assertThat(mockedList.get(2)).isEqualTo(stringToReturn);
   assertThat(mockedList.get(3)).isEqualTo(stringToReturn);
      // Non existant indexes
   assertThat(mockedList.get(50)).isEqualTo(stringToReturn);
   assertThat(mockedList.get(100)).isEqualTo(stringToReturn);
}
```

  - Disponible pour la quasi totalité des types JAVA
    - any(), anyBoolean(), anyByte(), anyChar(), anyCollection(), anyDouble(), anyFloat(), anyInt(), anyIterable(), anyLong(), anyMap(), anySet(), anyShort(), anyString()

- Mockito.verify()
  - Permet de vérifier qu'une méthode est appelée, et même de tester le nombre d'appels

```java
@ExtendWith(MockitoExtension.class)
public class DemoMockito {

   @Mock
   private User user;

   @Test
   void shouldPassWhenGetPasswordNotCalledAndGetUsernameCalledOnce() {
      String username = user.getUsername();
      Mockito.verify(user, Mockito.times(0)).getPassword();
      Mockito.verify(user, Mockito.times(1)).getUsername();
   }

   @Test
   void shouldFailWhenGetUsernameCalledOnlyOnce() {
      String username = user.getUsername();
      Mockito.verify(user, Mockito.times(2)).getUsername();
   }
}
```

```
Test Results                                                    537 ms
  com.percallgroup.formationtest.DemoMockito                    537 ms
    shouldPassWhenGetPasswordNotCalledAndGetUsernameCalledOnce  520 ms
    shouldFailWhenGetUsernameCalledOnlyOnce()                    17 ms
```

LET’S PRACTICE AND DO SOME MAGIC

POSTMAN

POSTMAN

- **Un client REST avec des super pouvoirs**
  - Permet d'envoyer des requêtes HTTP en contrôlant tous les paramètres :
    - Request Header
      - Authentification (ApiKey, BasicAuth, BearerToken, etc.)
      - User agent
      - Accept & Accept Encoding
    - Request Parameters
    - Request Body
      - Form-data
      - Raw (JSON, Text, XML, etc.)
      - Binary

POSTMAN

- Postman intègre également un petit framework de test JS qui permet d'écrire des suites de tests pertinentes.
- Bien que non unitaires, ces tests vont nous permettre d'avoir un « matelas » de tests sur lesquels s'appuyer en cas de refactoring
- Pourra également s'intégrer dans une CI / CD

- **Comment on teste « concrètement » ?**
  - Nécessite le même travail d'analyse qu'un test unitaire, mais avec une vision plus globale.
  - Ne pas hésiter à préparer son test avec les Pre-request Scripts
  - Tester le code de retour de la requête, les valeur des variables retournées, les messages d'erreurs
  - Le code reste du javascript donc on peut exploiter les concepts des divers « simulacres » vus précédemment, via des fonctions internes, des callbacks etc.

POSTMAN

- **Un framework de test minimaliste mais efficace**
  - Permet sans fioriture de tester les éléments essentiels d'une réponse HTTP
  - Reprend la notion de Snippets utilisée dans le composer TWx

- **Variabiliser ses requêtes**
  - Le plus efficace est de gérer ses variables globales par « Environnement »
  - Quand on créée un environnement on peut créer des variables (port & url du serveur, paramètre, etc.)
  - Dans l'url on les utilisera sous la forme {{variable}}
  - Dans les scripts on les récupèrera via un petit bout de code (d''ailleurs disponible dans les snippets) :

```
serverUrl = pm.environment.get("serverUrl");
```

  - Une bonne pratique consiste à mettre les déclarations de variables globales dans le Pre-request Script de la **collection**. Ainsi toutes ses requêtes auront accès à ces variables
  - Pour l'utilisation de variables uniquement dans certaines requêtes vous pouvez utiliser la partie « variables » de la collection. On y accède de la même manière

- **Envoyer une requête dans le Pre-request Script**

```
pm.sendRequest({
    url : 'http://' + serverUrl + ':' + serverPort +
          '/Thingworx/Things/Test%2520Bike/Properties/Speed',
    method: 'PUT',
    header: {
      'appKey':'ca1aece6-5616-4d04-8b5b-345bf0cc5941',
      'Accept':'application/json',
      'Content-Type':'application/json'
    },
    body: {
      mode: 'raw',
      raw: JSON.stringify({ Speed: 250 })
    }
}, function (err, response) {
    console.log('Bike is usable in GP');
});
```

  - Construction d'une request par script
  - Equivalent du @BeforeEach
  - Permet de mettre en place les valeurs qui seront utilisés dans le test

POSTMAN

- **Tester le code retour HTTP de la response :**

```
// Assert exception thrown
pm.test("Test IsGpUsable service KO", function() {
    pm.response.to.have.status(500);
});
```

  - Tester systématiquement le 200 et le 500 en cas de test de cas bloquant

- **Tester si le body de la response contient une string donnée :**

```
pm.test("Response text error is conform", function() {
    pm.expect(pm.response.text())
        .to.include(
            "Vehicle speed can't be negative neither equals
            to zero. Please fill speed property with a correct
            number.«
        );
});
```

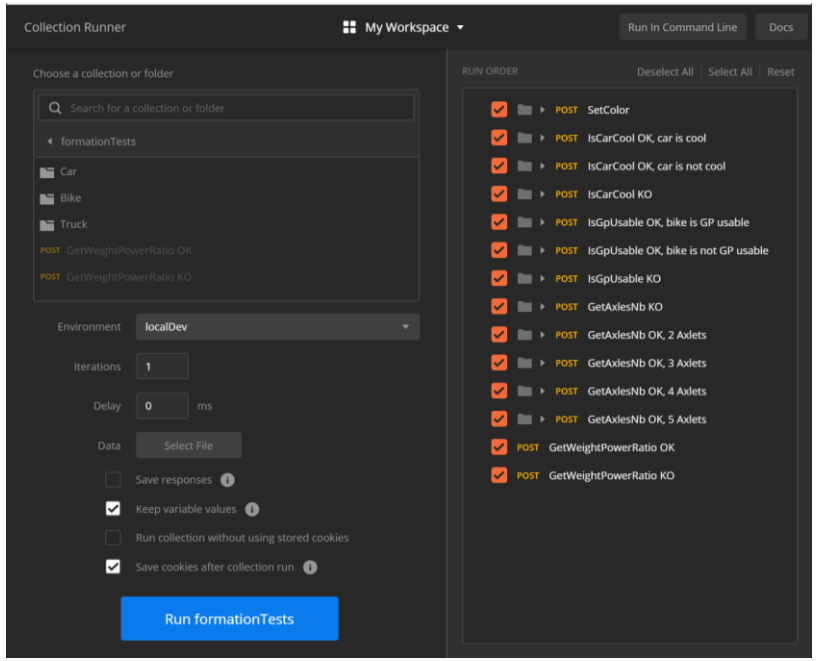
- **Tester la valeur d'un paramètre de la response**

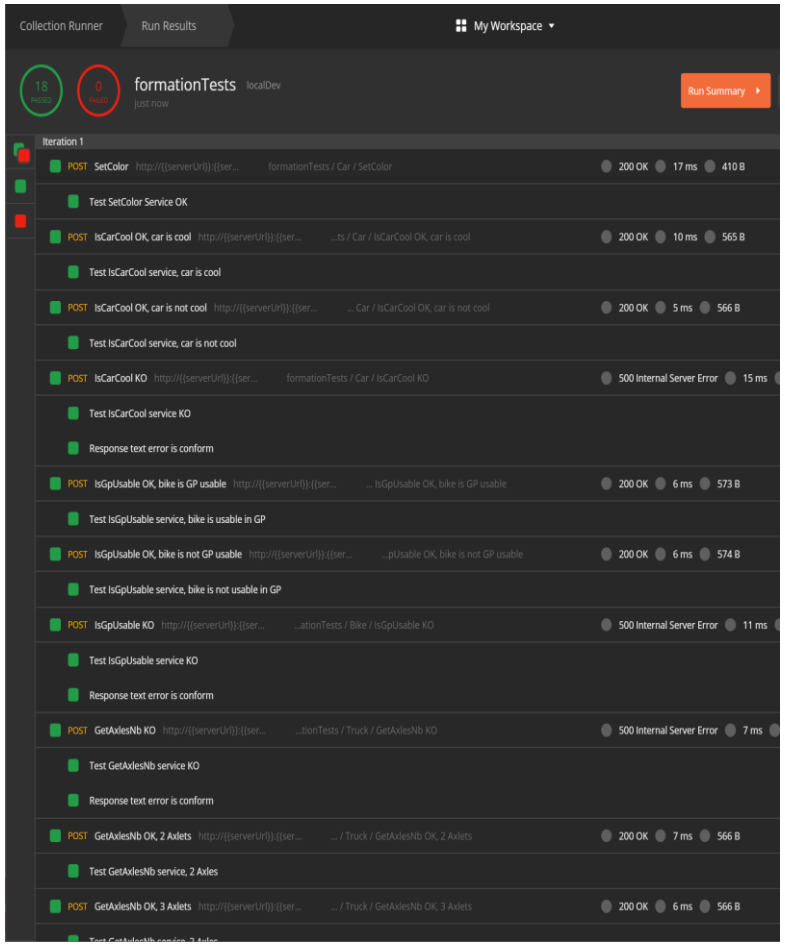
```
// Assert bike is usable in GP
pm.test("Test IsGpUsable service, bike is usable in GP",
    function () {
        var jsonData = pm.response.json();
        pm.expect(jsonData.rows[0].isUsable).to.eql(true);
    }
);
```

  - Voilà pour les principales utilisations. Expérimenter les snippets dispos selon vos besoins

POSTMAN

- Une fois les requêtes définies et les tests écrits on peut jouer la suite de tests à volonté et voir l'état du code pendant un refactoring par exemple





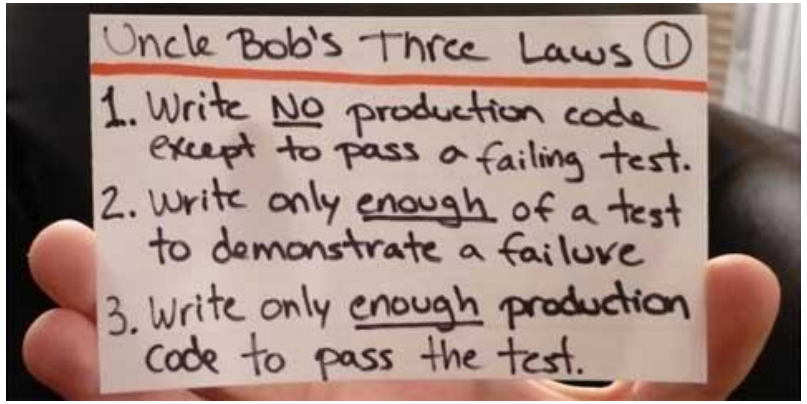
LET’S PRACTICE AND DO SOME MAGIC

LE TDD

- **Test Driven Development, merci Oncle Bob !!**

Il repose sur trois principes très simples mais à respecter quoiqu'il arrive :

1. **JE N'ÉCRIS PAS UNE SEULE LIGNE DE CODE SI CE N'EST POUR FAIRE PASSER UN TEST !!!**
2. **JE N'ÉCRIS QUE LE MINIMUM DE CODE POUR TESTER UNE FONCTIONNALITE**
3. **JE N'ÉCRIS QUE LE MINIMUM DE CODE NÉCESSAIRE À LA RÉUSSITE DU TEST**



- Aucun code n'est écrit avant un test
- Peut très bien se passer de frameworks
- Amène à une couverture de code de 100%
- Mode de réflexion très challengeant
- A tendance à simplifier drastiquement les algorithmes
- Maîtrise du code produit

LE TDD

- **La méthode RED / GREEN / REFACTOR**

- Ok, cette photo ne lui rend pas honneur, mais ça illustre bien le principe
- Et promis, il fait partie des gars les plus BRILLANTS de ces dernières années, visionnaire dans l'architecture logicielle.

1. On écrit le minimum de code pour tester la fonctionnalité
2. Le test échoue (Red)
3. On écrit le minimum de code pour faire passer le test
4. Le test passe (Green)
5. On refactorise notre code, nettoie l'algorithme
6. On écrit un nouveau test, retour au 1.

LET’S PRACTICE AND DO SOME MAGIC

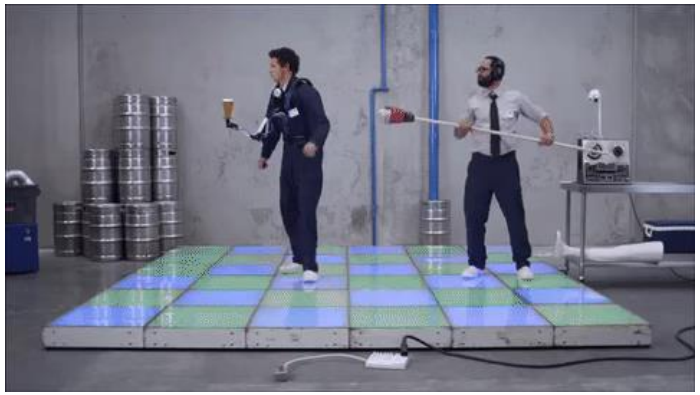
D'AUTRES PISTES

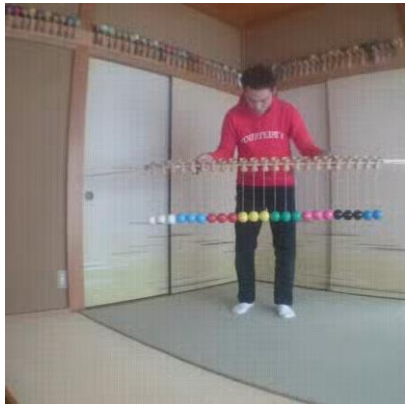
- **Le BDD**
  - Behavior Driven Development
    - Améliore le TDD en n'écrivant plus du code uniquement compréhensible par les développeurs, mais via des scénarios compréhensibles par tous les membres du projet
      - → Syntaxe Gherkin
    - Fonctionne très bien en agile
    - Structure une story suivant un préambule et des steps
      - → Préambule : As a / I want / In order to
      - → Step : GIVEN / WHEN / THEN
    - Peut servir de base solide à une documentation vivante et toujours à jour via les spécifications
    - Exemples de frameworks
      - → Cucumber, CucumberJS, JBehave

- **Thèmes intéressants à approfondir (pas spécifiquement liés aux tests) :**
  - Mutation testing
  - Software Craftmanship
  - Clean Code
  - Design patterns
  - Uncle bob (Robert C. MARTIN), Martin FOWLER, Alistair Cockburn
  - Curiosité intellectuelle

MERCI DE VOTRE ATTENTION

Et que vos applications soient aussi robustes que celles-ci : :-D





About Capgemini

Capgemini is a global leader in partnering with companies to transform and manage their business by harnessing the power of technology. The Group is guided everyday by its purpose of unleashing human energy through technology for an inclusive and sustainable future. It is a responsible and diverse organization of nearly 350,000 team members in more than 50 countries. With its strong 55-year heritage and deep industry expertise, Capgemini is trusted by its clients to address the entire breadth of their business needs, from strategy and design to operations, fueled by the fast evolving and innovative world of cloud, data, AI, connectivity, software, digital engineering and platforms. The Group reported in 2022 global revenues of €22 billion.

Get the future you want | www.capgemini.com

Capgemini