Courte introduction au test de logiciels, test unitaire, et JUnit

2021

Table des matières

1 Introduction

Le test de logiciel fait partie des activités dites de V&V : Vérification et Validation. Le V&V est un ensemble d'activités menées en parallèle du développement et qui vise à s'assurer que le logiciel construit sera conforme aux exigences qui ont dirigé le développement. Dans le cycle en V du logiciel, les phases de V&V correspondent à la "remontée" du V. Dans les développements agiles, les phases de V&V sont omni-présentes et peuvent paradoxalement précéder le développement.

Les défauts du logiciel que l'on cherche à détecter peuvent provenir de différentes phases du développement, par exemple :

— défaut de spécification : un comportement attendu a été mal spécifié, ou oublié.
— défaut de conception : la réalisation d'un comportement a été mal mise en œuvre dans la conception et ne correspond pas à la spécification.
— défaut d'implémentation : l'implémentation n'est pas en accord avec la conception.

Plus les défauts sont détectés tard dans le cycle de vie du logiciel, plus ils sont coûteux. De nombreuses approches de V&V existent, parmi lesquelles les principales sont :

— Test statique. Le test statique (ou analyse statique) consiste à détecter des défauts sans nécessiter l'exécution du code. Il prend souvent la forme d'examen manuel de documents textuels : revues de code, revue de spécifications, revue de documents de conception. Elle peut aussi consister en l'utilisation d'outils d'analyse statique calculant par exemple des métriques sur le code afin d'en déterminer de potentiels défaut.
— Test dynamique. C'est ce que l'on entend le plus souvent quand on parle de test. Il s'agit ici d'exécuter le code afin de déterminer si son comportement est correct.
— Vérification formelle. Il s'agit ici de s'assurer formellement de la cohérence entre un artéfact et ses spécifications. On trouver par exemple dans cette catégorie le model-checking (vérification formelle de propriété dans un modèle), la preuve (on prouve par exemple la correction d'un programme vis à vis de ses spécifications), le raffinement de spécifications vers des modèles proches de l'implémentation (avec des approches à la B).

Les techniques les plus utilisées dans l'industrie sont le test dynamique et dans une moindre mesure le test statique. Nous n'abordons dans la suite que le test dynamique.

Le test dynamique de logiciels

Le test dynamique de logiciel est un domaine fort vaste que nous ne ferons qu'effleurer ici. Il est tout d'abord important de noter que les phases de test accompagnent le développement du logiciel, elles peuvent même le précéder dans le cas de Test-Driven Development. Le test dynamique de logiciel consiste à exécuter le système sous test, et de comparer les sorties obtenues avec les sorties attendues. Il faut donc déterminer des scénarios de test intéressants, avec les données qui les accompagnent ; puis déterminer les éléments observables qui vont permettre de déterminer si on a trouvé ou pas une erreur, ainsi que les valeurs à observer ; et enfin comparer ces valeurs aux valeurs effectivement observées lors de l'exécution du test.

2 Introduction à JUnit

2.1 Historique

JUnit est un framework de test unitaire dédié à Java. C'est le plus populaire de la grande famille des frameworks de test XUnit, pour différents langages (dans cette famille on peut par exemple citer CPPUnit, PiUnit, PHPUnit, ...). Originellement, JUnit est issu des mouvances autour de l'Xtreme Programming, une des premières méthodes agiles, et des méthodes dites de Test-First Development. Il a été créé initialement par Kent Beck (inventeur du concept d'extreme programming) et Erich Gamma (un des auteurs du fameux livre fondateur sur les patrons de conception, communément appelé le Gang of Four [1], et un des initiateurs de l'IDE Eclipse). On peut noter que Kent Beck avait tout d'abord conçu SUnit, un framework de test pour SmallTalk.

2.2 Objectifs

JUnit est un framework open-source (www.junit.org), destiné au test unitaire d'applications Java, et permettant de faciliter la création et l'exécution des tests, permettant par là-même une facilitation du test de non-régression. Junit enchaîne l'exécution des méthodes de test définies par le testeur, facilite la définition des tests grâce à des assertions, des méthodes d'initialisation et de finalisation, et permet de voir rapidement les verdicts émis par les tests. En revanche JUnit n'écrit pas les tests ! Même si l'usage de JUnit impose une certaine organisation des tests (à l'extérieur des classes testées, dans des méthodes de test stigmatisées, ...), JUnit ne propose ni n'impose pas de principes ni de méthodes pour écrire les tests.

JUnit est un framework qui définit toute l'infrastructure nécessaire pour écrire les tests, définir les oracles des tests, exécuter les tests et collecter les verdicts émis. Utiliser JUnit revient donc à définir des tests qui

1. "Design Patterns : Elements of Reusable Object-Oriented Software", écrit par Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides (qui forment le "Gang of Four", souvent abrégé GoF), publié en 1994

embarquent des oracles, et à s'en remettre à JUnit pour leur exécution, sans appeler explicitement les méthodes de test.

2.3 Versions majeures de JUnit

Dans ses versions initiales (1, 2 et 3), JUnit se paramètre par spécialisation des classes TestCase et TestSuite, pour créer des classes de test et des suites de tests. JUnit se base alors sur des conventions de nommage pour identifier les méthodes de test (leur nom doit commencer par la chaîne *test*) et les méthodes auxiliaires pour le test, et sur l'introspection pour les collecter et les exécuter. Le framework intègre un "runner" graphique, qui présente une barre d'exécution plus ou moins verte selon le ratio de tests n'ayant pas détecté d'erreur. C'est pour cela que le slogan de JUnit est : "Keep the bar green to keep the code clean !".

La version 4 intègre au framework les annotations que Java a introduites dans sa version 1.5. Les conventions de nommage disparaissent au profit d'annotations dédiées, et de nouveaux éléments apparaissent (nouvelle méthodes auxiliaires par exemple). En revanche, le "runner" graphique disparaît, il est laissé au soin des IDEs.

La version 5 (ou jupiter) intègre au framework les lambdas introduites par Java dans sa version 8. Les versions inférieures sont toujours présentes dans le framework, mais désormais dans un package nommé *vintage* !

3 Premiers pas avec JUnit

3.1 Les classes de test

Le principe général pour utiliser JUnit est de créer une ou plusieurs classes (dites classes de test) destinées à contenir les tests (sous forme de méthodes de de test). Chaque méthode de test fait appel à une ou plusieurs méthodes du système à tester (communément appelé SUT pour System Under Test). Cela suppose de disposer dans la classe de test d'une instance d'une classe du système à tester (la création d'une telle instance peut être placée à plusieurs endroits, comme nous le verrons plus loin). Les instances à tester sont souvent placées comme attributs de la classe de test.

Les méthodes de test incluent des instructions permettant un verdict automatique, via l'utilisation d'assertions. S'il n'est pas requis stricto sensu par JUnit de placer des assertions dans les méthodes de test, c'est en revanche très fortement conseillé. A défaut, tous les tests "passeront" à l'exception de ceux qui déclenchent une exception inopinée. En aucun cas le verdict ne doit apparaître sous forme d'un affichage.

Les méthodes de test ne sont pas ordonnées : elles sont supposées être exécutées dans un ordre quelconque. En conséquence, les méthodes de test doivent être indépendantes les unes des autres : on ne doit pas supposer qu'un des méthodes de test doit être appelée avant une autre pour un bon fonctionnement. Depuis la version 5 de JUnit, on peut plus ou moins contrôler l'ordre d'appel, mais il faut malgré tout garder l'indépendance des méthodes de test.

La classe de test peut aussi contenir des méthodes auxiliaires (référées comme *fixtures* dans la documentation) permettant de fixer l'environnement de test : ces méthodes permettent d'initialiser le système sous test. Comme les méthodes de test, elles ne sont pas à appeler explicitement, elles sont appelées par JUnit lors de l'exécution des tests. Elles seront décrites un peu plus tard dans ce document.

En JUnit versions <4, une classe de tête a l'entête suivante :

```
1 public class MaJolieClasseDeTest extends TestCase{
```

Depuis JUnit 4, une classe de test n'est pas spécialement stigmatisée ni par un héritage, ni par une annotation.

3.2 Les méthodes de test

Chaque méthode de test devrait ne s'intéresser qu'à une seule unité de code, un seul comportement. Une méthode de test doit également être courte, afin de faciliter ensuite la phase de diagnostique de l'erreur

éventuellement détectée.

Dans les versions <4, les méthodes de test commencent par le mot `test`. Dans les versions ≥4, les méthode des test sont annotées avec l'annotation `@Test`. Elles sont sans paramètres ni type de retour [2], ce qui est logique puisqu'elles vont être appelées automatiquement par JUnit.

Les méthodes de test doivent embarquer un oracle, c'est-à-dire le moyen de décider automatiquement du verdict à émettre à l'exécution (a-t-on trouvé ou pas une erreur). Cet oracle est en général défini en utilisant des assertions. On va par exemple exprimer qu'un certain attribut doit valoir une certaine valeur, que le résultat de l'appel à une méthode est différent de null, etc. Pour exprimer ces assertions, on peut utiliser le assert de Java, mais JUnit fournit des assertions plus riches

JUnit introduit des assertions plus riches que le assert Java, et préconise l'utilisation d'Hamcrest (un petit DSL interne) qui permet d'écrire des assertions ressemblant au langage naturel.

Les verdicts (a t-on détecté ou pas une erreur ?) sont définis grâce aux assertions placées dans les méthodes de test. On distingue globalement le verdict Pass (vert) aucune erreur détectée et le verdict fail ou error (rouge) une erreur a été détectée, soit parce qu'une assertion a été violée, soit parce que le test n'a pas pu s'exécuter correctement.

3.3 L'environnement de test, préambules et postambules

Les méthodes de test ont besoin de manipuler des instances du système sous test. Il faut donc dans une classe de test déclarer et créer des instances du système sous test. En général, les instances sont déclarées comme membres d'instance de la classe de test, et la création des instances et plus globalement la mise en place de l'environnement de test est laissée à la charge de méthodes d'initialisation (de préambule). Les préambules et postambules sont des méthodes que vous trouverez sous le nom de fixtures dans les documentations de JUnit. Les méthodes de préambule sont écrites par le testeur pour mettre en place l'environnement de test. Elles sont appelées automatiquement lors de l'exécution des tests. On distingue les méthodes de préambule appelées avant chaque méthode de test, et celles appelées avant chaque exécution de la classe de test.

Méthodes appelées avant chaque méthode de test

On distingue les versions de JUnit pour ces préabules :

— En JUnit 5, ces méthodes sont annotées avec l'annotation @BeforeEach. Elles sont appelées, comme le nom de l'annotation l'indique, avant chaque appel d'une méthode de test. Quand on écrit une méthode de test, on peut donc supposer que la méthode annotée par @BeforeEach a été appelée juste avant. Plusieurs méthodes peuvent être annotées avec cette annotation dans une même classe, l'ordre d'exécution est alors indéterminé. Ces méthodes sont publiques et non statiques.

— En JUnit 4, ces méthodes sont annotées avec l'annotation @Before. Elles sont appelées avant chaque appel d'une méthode de test. Quand on écrit une méthode de test, on peut donc supposer que la méthode annotée par @Before a été appelée juste avant. Plusieurs méthodes peuvent être annotées avec cette annotation dans une même classe, l'ordre d'exécution est alors indéterminé. Ces méthodes sont publiques et non statiques.

— En Junit 3, cette méthode doit s'appeler setUp.

Méthodes appelées avant l'exécution de la classe de test

On distingue les versions de JUnit pour ces préabules :

— En JUnit 5, ces méthodes sont annotées avec l'annotation @BeforeAll. Les méthodes avec l'annotation @BeforeAll ne sont appelées qu'une seule fois juste avant l'exécution de la classe de test. Il ne peut y avoir qu'une seule méthode annotée avec cette annotation au sein d'une même classe, et cette méthode doit être statique (sauf si le cycle de vie est perClass, mais nous n'abordons pas cette subtilité ici).

2. Cette affirmation n'est pas tout à fait vraie. Depuis JUnit 5

— En JUnit 4, ces méthodes sont annotées avec l'annotation @BeforeClass. Les méthodes avec l'annotation @BeforeClass ne sont appelées qu'une seule fois juste avant l'exécution de la classe de test. Il ne peut y avoir qu'une seule méthode annotée avec cette annotation au sein d'une même classe, et cette méthode doit être statique.
— En Junit 3, il n'y a pas d'équivalent.

Postambules

Les postambules sont les symétriques des préambules, ils sont appelés après les tests (après chaque méthode de test ou après l'exécution d'une classe de test). Ils servent notamment à remettre le système dans l'état dans lequel était le système avant le test. La description est la même que pour les préambules avec les annotations @AfterEach et @AfterAll en Junit 5, les annotations @After et @AfterClass pour JUnit4, et la méthode tearDown pour JUnit 3.

3.4 Premier exemple

Supposons que l'on souhaite tester la classe Heure suivante, qui représente des horaires entre 7h et 23h, avec une granularité de 5 minutes.

```
1  public class Heure {
2          private int heures, minutes;
3          private static int granulariteMinutes=5;
4          private static int heureMax=22;
5          private static int heureMin=7;
6
7          private boolean heuresCorrectes(){
8                  return heures>=heureMin && heures<=heureMax;
9          }
10         private boolean minutesCorrectes(){
11                 boolean result=minutes%granulariteMinutes==0;
12                 if (heures==heureMax&&minutes!=0)result=false;
13                 return result;
14         }
15         public Heure(int heures,int minutes) throws HoraireIncorrectException{
16                 this.heures=heures;
17                 this.minutes=minutes;
18                 if (!heuresCorrectes()||!minutesCorrectes()){
19                         throw new HoraireIncorrectException("heure specifiee incorrecte");
20                 }
21         }
22         public String toString(){
23                 String h=Integer.toString(heures);
24                 String mn=Integer.toString(minutes);
25                 // ajout des 0 non significatifs
26                 if (heures<10)h="0"+h;
27                 if (minutes<10)mn="0"+mn;
28                 return h+":"+mn;
29         }
30 }
```

Si l'on se donne comme objectifs de test :
— de vérifier que la méthode toString est correcte pour des heures bien formées,
— la création d'horaires incorrects déclenche des levées d'exceptions,
— les méthodes estAvant et estStrictementAvant donnent des résultats corrects pour des horaires corrects,
on produit par exemple la classe de test suivante.

```
1  import static org.hamcrest.MatcherAssert.assertThat;
2  import static org.junit.jupiter.api.Assertions.*;
3  import static java.time.Duration.ofMillis;
4  import org.junit.jupiter.api.BeforeEach;
```

```java
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.Test;
import static org.hamcrest.object.HasToString.*;

class TestHeuresJUnit5 {

	Heure h1;
	Heure h2;
	Heure h3;
	Heure h4;
	Heure h5;
	Heure h6;

	@BeforeEach
	public void setUp() throws HoraireIncorrectException{
		h1=new Heure(10,15);
		h2=new Heure(21,55);
		h3=new Heure(8,10);
		h4=new Heure(22,0);
		h5=new Heure(12,05);
		h6=new Heure(8,15);

	}

	@Test
	public void testToStringHeureValide() {
		assertEquals("10:15", h1.toString());
		assertEquals("21:55",h2.toString());
		assertEquals("08:10",h3.toString());
		assertEquals("22:00", h4.toString());
		assertEquals("12:05",h5.toString());
		assertEquals("08:15", h6.toString());
		assertThat(h6,hasToString("08:15")); // avec hamcrest
	}

	@Test
	public void testCreationHeureInvalideDepasseHeureMax() {
		assertThrows(HoraireIncorrectException.class, () -> {
			new Heure(23,05);
		});
	}

	@Test
	public void testCreationHeureInvalideAvantHeureMin()  {
		assertThrows(HoraireIncorrectException.class, () -> {
			new Heure(6,10);
		});
	}

	@Test
	public void testCreationHeureInvalideGranulariteFausse()  {
		assertThrows(HoraireIncorrectException.class, () -> {
			new Heure(7,12);
		});
	}
	@Test
	public void testEstAvant(){
		assertFalse(h1.estStrictementAvant(h1));
		assert(h1.estAvant(h2));
		assert(h1.estAvant(h4));
		assert(h1.estAvant(h5));
		assert(h2.estAvant(h4));
		assert(h3.estAvant(h6));
	}
}
```

Dans cet exemple, la méthode appelée setUp sera appelée 5 fois à l'exécution de cette classe de test : une fois avant chacune des 5 méthodes de test. On remarque qu'avec JUnit, on peut exécuter cette classe de test, bien qu'elle ne contienne pas de main : c'est JUnit qui pilote l'exécution de méthodes de test, ce n'est pas le testeur, qui doit se contenter d'écrire les métodes de test (ce qui est le gros du travail !).

4 Approfondissements

Dans cette section, nous listons en vrac quelques fonctionnalités additionnelles de JUnit, principalement issues de JUnit 4 et 5. On peut noter que si initialement (et globalement jusqu'à la version 3), JUnit était un framework de test extrêmement simple (et rudimentaire), il est devenu beaucoup plus complexe (et sophistiqué) depuis les versions 4 et 5. Nous ne faisons ici qu'explorer quelques fonctionnalités, elles sont légion !

4.1 Les assertions

Nous faisons ici un petit retour sur les assertions. Elles sont utilisées pour embarquer et automatiser l'oracle dans les cas de test (adieu, `println` ...). Les méthodes d'assertion sont statiques, on utilise donc en général un `import static` adéquate pour les utiliser.

```
1 import static org.junit.Assert.*; //JUnit 4
2 import static org.junit.jupiter.api.Assertions.*; // JUnit 5
```

Les assertions, si elles sont violées, lancent des exceptions de type java.lang.AssertionError [3](comme les `assert` java classiques). Il existe beaucoup d'assertions permettant par exemple la comparaison à un delta près (bien utile pour les flottants), la comparaison de tableaux (arrays), la comparaison de lignes de texte (assertLinesMatch), la comparaison de références d'objets (assertSame), le test de vacuité d'un objet (assertNull), la vérification de levée d'exception, la vérification de timeout, la vérification de plusieurs assertions (assertAll), etc.

On peut aussi faire des assertions plus complexes, et émettant des messages d'erreur plus clairs, avec hamcrest. Hamcrest est une petite bibliothèque permettant de définir des "règles de correspondance" (match rules) et de vérifier qu'un objet satisfait bien ces règles. Les règles sont combinables (avec des opérateurs logiques).

Nous ne décrivons pas ici tout ce qu'il est possible de faire avec hamcrest, nous ne donnons que quelques exemples.

```
1  import static org.hamcrest.MatcherAssert.assertThat;
2  import static org.hamcrest.Matchers.*;
3  import java.util.ArrayList;
4
5  import org.junit.jupiter.api.BeforeEach;
6  import org.junit.jupiter.api.Test;
7
8
9  class DiversHamcrest {
10         int x;
11         String responseString;
12         ArrayList<String> myList;
13
14         @BeforeEach
15         void init() {
16                 x=4;
17                 responseString="colr";
18                 myList=new ArrayList<>();
```

3. en fait une sous classe de AssertionError en JUnit 5

```
19                  myList.add("coucou");
20          }
21
22
23          @Test
24          void hamcrestAtAGlimpse1() {
25                  assertThat(x, is(3));
26          }
27          @Test
28          void hamcrestAtAGlimpse2() {
29                  assertThat(x, is(not(4)));
30          }
31          @Test
32          void hamcrestAtAGlimpse3() {
33                  assertThat(responseString, equalToIgnoringCase("Color"));
34          }
35          @Test
36          void hamcrestAtAGlimpse4() {
37
38                  assertThat(responseString, either(containsString("color")).or(containsString
                        ("colour")));
39          }
40
41          @Test
42          void hamcrestAtAGlimpse5() {
43                  assertThat(myList, hasItem("3"));
44          }
45  }
```

Dans cet exemple, le premier test va échouer avec le message :

```
java.lang.AssertionError:
Expected: is <3>
     but: was <4>
```

Le second test va échouer avec le message :

```
java.lang.AssertionError:
Expected: is not<4>
     but: was <4>
```

On voit ici que l'on a combiné le `is(value)` avec le `not(expression)`. On aurait pu utiliser un simple "`!`" mais le message aurait alors été différent.

Le troisième test va échouer avec le message :

```
java.lang.AssertionError:
Expected: equalToIgnoringCase("Color")
     but: was colr
```

On voit ici que l'on a pu tester l'égalité à la casse près très simplement.

Le quatrième test va échouer avec le message :

```
java.lang.AssertionError:
Expected: (a string containing "color" or a string containing "colour")
     but: was "colr"
```

On a ici utilisé une construction either ... or, et on voit bien ici à quel point cela mène à un message explicitant parfaitement le problème en cas de violation.

Le cinquième et dernier test va échouer avec le message :

```
java.lang.AssertionError:
Expected: a collection containing "3"
     but: was "coucou"
```

Ici on illustre les règles sur les collections, avec ici l'utilisation du hasItem (dans le même ordre d'idée on a également le containsInAnyOrder, le hasSize, etc).

4.2 Test et exceptions

Pour tester qu'une méthode déclenche une exception dans les circonstances voulues, plusieurs solutions sont à notre disposition. La plus rustique consiste à :

- appeler la méthode qui doit jeter une exception dans un bloc try
- attraper l'exception dans une clause catch et y placer un assert(true) (pour que le test réussisse, il faut passer par ce catch)
- positionner un fail("la raison du fail") si l'exception n'est pas attrapée.

Cela donne par exemple le code suivant, en réutilisant l'exemple précédent.

```
1  @Test
2          public void testCreationHeureInvalideMauvaiseGranulariteFaçonJUnit3() {
3                  try {
4                          new Heure(7,12);
5                          fail("l'exception aurait du être levée");
6                  }catch (HoraireIncorrectException e) {
7                          assert(true); // tout va bien !
8                  }
9          }
```

JUnit 4 et 5 proposent chacun un mécanisme plus simple pour tester la levée d'exception.

En JUnit 4, l'annotation `@Test` peut prendre en paramètre le type d'exception attendue @Test(expected=monexception.class). Le test est un succès si et seulement si cette exception est lancée.

```
1 @Test(expected=HoraireIncorrectException.class)
2 public void testCreationHeureInvalideDepasseHeureMax()
3   throws HoraireIncorrectException {
4         new Heure(23,05);
5 }
```

En JUnit 5, on utilise l'assertion assertThrows, qui prend en premier paramètre la classe d'exception qui est attendue, et en deuxième paramètre le comportement devant lever cette exception, sous forme de lambda expression (JUnit Jupiter est une version postérieure à Java 8, qui a introduit les streams et les lambdas). Plus techniquement, le deuxième paramètre est typé par Executable, qui est une interface fonctionnelle introduite par JUnit. On rappelle ici que les interfaces fonctionnelles ont été introduites avec Java 8, ce sont des interfaces avec une unique méthode. Quand on veut disposer d'une instance typée par une telle interface, on peut utiliser une lambda donnant le comportement de son unique méthode, ou bien utiliser une référence de méthode (maClasse::maMethode).

```
1 @Test
2 public void testCreationHeureInvalideDepasseHeureMax() {
3         assertThrows(HoraireIncorrectException.class, () -> {
4                 new Heure(23,05);
5         });
6 }
```

Dans l'exemple précédent, on indique donc qu'il faut lever une exception de type HoraireIncorrectException quand on exécute le comportement : new Heure(23,05).

4.3 Test et temps d'exécution

Il est possible depuis JUnit 4 de fixer un temps maximal d'exécution pour un test. Ainsi, le test échoue si le timeout est atteint avant que le test ne finisse son exécution.

En Junit 4, cela donne :

```
@Test(timeout=1)
public void testAvecTimeout() throws HoraireIncorrectException{
        new Heure(7,15);
}
```

En JUnit 5, cela donne :

```
@Test
public void testAvecTimeout() throws HoraireIncorrectException{
        assertTimeout(ofMillis(1), () -> {//java.time.Duration.ofMillis
                new Heure(7,15);
        });
}
```

La méthode assertTimeout prend pour premier paramètre une instance de Duration. Duration est une classe du package java.time, ce package a été introduit avec Java 8 pour gérer le temps beaucoup simplement qu'avec les classes java.util.Time et autre java.util.Calendar. On utilise ici la méthode statique ofMillis qui construit une durée à partir d'un nombre de millisecondes. La méthode assertTimeout prend en second paramètre le comportement pour lequel le temps d'exécution est contraint, ici la création d'une heure correcte. Comme nous l'avons vu avec le assertThrow, ce second paramètre est en fait typé par l'interface Executable.

4.4 Omission d'exécution et exécution conditionnée

Depuis JUnit 4, il est possible d'omettre l'exécution de certains tests (sans les mettre en commentaire ni les effacer, ce qui est bien sûr une autre façon d'en omettre l'exécution !). L'intérêt d'une telle omission peut se trouver par exemple quand on a écrit un test, mais pas encore la partie de code qui devrait faire "passer" ce code (par exemple avec une approche de dévelopement test-first). Signaler les tests comme omis permet qu'ils ne soient pas exécuté tout en gardant une trace de la présence de ces tests omis (lors de l'exécution, on voit clairement que certains tests ont été omis, ce qui n'est pas le cas bien sûr si les tests ont été commentés).

En JUnit 4, on utilise alors l'annotation @Ignore, paramétrée optionnellement par une chaîne de caractère avec la raison pour laquelle le test dit être omis.

```
@Ignore
@Test
public void testNonExecute() {
        // ...
}
```

En JUnit 5, on utilise l'annotation `@Disabled` (paramètre optionnel : du texte) pour désactiver un test.

```
@Disabled
@Test
public void testNonExecute() {
        // ...
}
```

On peut utiliser cette même annotation pour omettre une classe entière.

En JUnit 5, on peut également conditionner l'exécution d'un test en fonction de l'environnement, comme illustré ci-dessous.

```
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.condition.*;
import static org.junit.jupiter.api.condition.OS.*;
import static org.junit.jupiter.api.condition.JRE.*;
```

```
class TestExecutionConditionnee {
        @Test
        @EnabledOnOs(MAC)
        void onlyOnMacOs() {
            // ...
        }

        @Test
        @EnabledOnOs({ LINUX, MAC })
        void onLinuxOrMac() {
            // ...
        }

        @Test
        @DisabledOnOs(WINDOWS)
        void notOnWindows() {
            // ...
        }


        @Test
        @EnabledOnJre(JAVA_8)
        void onlyOnJava8() {
            // ...
        }

        @Test
        @EnabledOnJre({ JAVA_9, JAVA_10 })
        void onJava9Or10() {
            // ...
        }

        @Test
        @DisabledOnJre(JAVA_9)
        void notOnJava9() {
            // ...
        }
}
```

On peut aussi placer à l'intérieur des tests des suppositions conditionnant l'éxécution du reste du test. En Junit 4 on utilise une supposition du style :

```
assumeThat(File.separatorChar, is(''/''));
```

Le reste du test sera ignoré si la supposition n'est pas vérifiée.

```
@Test public void testOnlyOnDeveloperWorkstation() {
        assumeThat(System.getenv("ENV"), is("DEV"));
        assertEquals(1, 1);
}
```

En JUnit 5, on utilise `assumeTrue` et `assumingThat`

```
@Test void testOnlyOnDeveloperWorkstation() {
    assumeTrue("DEV".equals(System.getenv("ENV")),
            () -> "Aborting test: not on developer workstation");
        // remainder of test
    }
@Test void testInAllEnvironments() {
     assumingThat("CI".equals(System.getenv("ENV")),
            () -> {
                // perform these assertions only on the continuous integration server
                assertEquals(2, 2);
            });
        // perform these assertions in all environments
```

```
13          assertEquals("a string", "a string");}
```

AssumeTrue est paramétré par une expression booléenne (premier paramètre) et un Executable (ici une lambda), tandis que assumingThat est paramétré par une règle de correspondance (un matcher) et un Executable (ici une lambda).

4.5 Le test paramétré

Il arrive souvent que l'on veuille exécuter plusieurs fois le même comportement avec des données différentes. Par exemple, si on veut tester une méthode pgcd, on veut plusieurs fois de suite : appeler cette méthode 2 entiers a et b, puis vérifier que le résultat obtenu est bien le pgcd de a et de b. Comme les méthodes de test ne sont pas paramétrées, on se retrouvait souvent en JUnit3 à faire des copier-coller de méthodes de test en changeant juste les données d'entrée et les résultats attendus. C'est dans cette optique que l'on peut utiliser du test paramétré, qui consiste à réutiliser des méthodes de test avec des jeux de données de test différents.

En JUnit 4, les jeux de données sont retournés par une méthode annotée @Parameters. Cette méthode retourne une collection de tableaux contenant les données et éventuellement le résultat attendu, cette méthode est statique. La classe de test dédiée au test paramétré est annotée avec @RunWith(Parameterized.class). Elle contient, en plus de la méthode chargée de retourner les jeux de données, des méthodes de test devant être exécutées avec chacun des jeux de données. A l'exécution par JUnit4 de cette classe de test, pour chaque donnée, la classe est instanciée, et les méthodes de test sont exécutées avec chacun des jeux de données. La classe de test contient un constructeur public paramétré de manière à pouvoir recevoir un jeu de données. Le constructeur est en charge de manipuler le jeu de données (en général en en rangeant les valeurs dans des attributs d'instances), de manière à ce que les méthodes de test (qui ne sont pas paramétrées), puisse s'exécuter correctement.

Cela est illustré ci-dessous sur l'exemple des horaires. Un jeu de test est constitué de 4 entiers (les entrées du test) et d'un booléen (qui servira pour l'oracle). Le constructeur construit les 2 horaires h1 et h2, et les stock dans des attributs d'instance, il stocke de même le booléen. Les méthodes de test manipulent les 2 horaires et le booléen, pour tester la méthode estAvant de Heure, et la création de créneaux (une autre classe non fournie ici).

```
1  import org.junit.Test;
2  import org.junit.runner.RunWith;
3  import org.junit.runners.Parameterized;
4  import org.junit.runners.Parameterized.Parameters;
5  import static org.junit.Assert.*;
6  import java.util.*;
7  @RunWith(Parameterized.class)
8  public class TestParametre {
9          private Heure h1;
10         private Heure h2;
11         private boolean h1Avanth2;
12         public TestParametre(int hh1, int mn1, int hh2, int mn2, boolean h1AvantH2) throws
               HoraireIncorrectException {
13             h1=new Heure(hh1, mn1);
14             h2=new Heure(hh2, mn2);
15             this.h1Avanth2=h1AvantH2;}
16         @Parameters
17         public static Collection testData() {
18             return Arrays.asList(new Object[][] {
19                     { 7, 0, 7, 5, true },{7,0, 12, 5, true },{ 12,30, 7, 5, false },{12,
                           00, 20,15, true}});}
20         @Test public void testEstAVant() {
21             assertEquals(h1Avanth2,h1.estAvant(h2));}
22         @Test public void creationCreneauValide() throws CreneauIncorrectException {
23             Creneau c;
24             if (h1.estAvant(h2)) {
25                     c=new Creneau(JourSemaine.LUNDI, h1, h2);
```

```
26                  } else {
27                          c=new Creneau(JourSemaine.LUNDI, h2, h1);
28                  }}
29          @Test(expected=CreneauIncorrectException.class)
30          public void testCreneauInvalide() throws CreneauIncorrectException{
31                  Creneau c;
32                  if (h1.estAvant(h2)) {
33                          c=new Creneau(JourSemaine.LUNDI, h2, h1);
34                  } else {
35                          c=new Creneau(JourSemaine.LUNDI, h1, h2);
36                  }}
37  }
```

En JUnit 5, on va pouvoir donner à des méthodes de tests des sources de données.

Un exemple de source de donnée est une méthode qui retourne des données. Dans ce cas, on lie les méthodes de test paramétré aux méthodes fournisseurs de données. Les méthodes qui fournissent les données retournent des flux d'Arguments (utilisation des Stream java), elles sont statiques. Dans l'exemple suivant, la méthode heureProvider retourne 4 triplets (entier, entier, booléen). En effet, la méthode statique Arguments.of retourne un Arguments construit avec les valeurs passées en paramètre, et la méthode statique Stream.of construit un flux avec les valeurs passées en paramètres.

On peut maintenant utiliser la méthode source de donnée pour la lier à une méthode de test. C'est ce qui est fait pour la méthode créationHeures. Cette méthode est paramétrée par 2 entiers et un booléen, et est liée à son fournisseur de données par l'annotation @MethodSource("HeureProvider"). C'est une méthode de test paramétré, elle est donc stigmatisée par l'annotation @ParameterizedTest. Cette annotation est ici paramétrée (c'est facultatif!) pour que chaque exécution avec un jeu de données ait un nom sympatique : l'index du jeu de test utilisé (index) puis la valeur de l'heure, la valeur des minutes, et la valeur du booléen oracle.

À l'exécution, la méthode de test creationHeure sera exécutée 4 fois, une fois par jeu de test.

Les sources de données ne sont pas forcément des méthodes. On peut par exemple directement fournir la source de donnée en en énumérant les valeurs, comme on le fait pour la méthode de test testAdherents. Cette méthode teste la création d'adhérents (une ébauche de classe Ahérent vous est donnée). Cette méthode de test paramétré n'est pas liée à une méthode qui fournit les données : ici les données sont directement fournies à la méthode de test paramétré en utilisant l'annotation @ValueSource(strings = "nom1", "nom2", "nom3"). La méthode testAdhérents est en effet paramétrée par une chaîne de caractères. Une @ValueSource peut être paramétrée de différents façons, avec notamment les types primitifs de Java. On a ajouté à la méthode testADhérents l'annotation @nullSource qui est la source de donnée fournissant la valeur null. Finalement, la méthode testAdhérents a donc 2 sources de données. A l'exécution, elle sera exécutée 4 fois, une fois avec comme paramètre nom1, une autre fois avec comme paramètre nom2, une troisième fois avec comme paramètre nom3, une quatrième fois avec la valeur null.

```
1  import static org.junit.jupiter.api.Assertions.*;
2  import java.util.stream.Stream;
3  import org.junit.jupiter.api.DisplayName;
4  import org.junit.jupiter.params.ParameterizedTest;
5  import org.junit.jupiter.params.provider.Arguments;
6  import org.junit.jupiter.params.provider.MethodSource;
7  import org.junit.jupiter.params.provider.ValueSource;
8
9  class TestParametreJunit5 {
10         private static  int nbAdherent=0;
11
12          @DisplayName("création d'heures")
13             @ParameterizedTest(name = "{index} => heure={0}, minutes={1}, correct={2}")
14             @MethodSource("heureProvider")
15             void creationHeure(int h, int mn, boolean correct) {
16                 if (!correct) {
17                         assertThrows(HoraireIncorrectException.class, ()-> {
18                         new Heure(h, mn);
```

```
                        });
                }}

            private static Stream<Arguments> heureProvider() {
                return Stream.of(
                        Arguments.of(10, 12, false),
                        Arguments.of(2, 30, false),
                        Arguments.of(23,10, false),
                        Arguments.of(12, 30, true)
                );}

        @ParameterizedTest
        @ValueSource(strings = { "nom1", "nom2", "nom3" })
        void testAdherents(String name) {
            Adherent a=new Adherent(name);
            nbAdherent++;
            assertEquals(nbAdherent, a.getNumero());
        }
}

public class Adherent {
...

public Adherent(String nom){
        ...
}
...
public int getNumero() {
        ...
}
}
```

D'autres sources de données peuvent être utilisées (et éventuellement combinées) :

- EnumSource : en spécifiant une énumération, tous ses littéraux seront successivement passés à la méthode de test, ou seulement une sélection d'entre eux, ou tous sauf ceux spécifiés.
- CsvSource : permet de spécifier une liste de valeurs séparées par des virgules
- CsvFileSource : permet de spécifier le chemin vers un fichier csv contenant les données de test.
- ArgumentSource : permet de spécifier un fournisseur d'argument générant un flux (stream) d'arguments à passer à la méthode de test
- EmptySource : permet d'unclure une valeur vide, dans e cas où la méthode de test a comme paramètre une chaîne, une collection ou un tableau.
- NullAndEmptySource : combinaison de EmptySource et NullSource

Des exemples d'utilisation de ces sources peuvent être trouvés dans la documentation :
`https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests-sources`

4.6 Les suites de test

Les suites de test sont utilisées pour regrouper les tests présents dans plusieurs classes de test, de manière à en enchaîner l'exécution. En Junit 4, on procède ainsi :

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
@RunWith(Suite.class)
@Suite.SuiteClasses({
        TestHeuresJUnit4.class, TestParametre.class
        })
public class SuiteDeTestJUnit4 {}
```

En Junit 5, on va utiliser des annotations comme @SelectClasses ou @SelectPackages pour sélectionner les éléments à ranger dans la suite de test. Par exemple, on procède ainsi :

```java
import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectClasses;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectClasses({TestHeuresJUnit5.class, TestParametreJunit5.class})
public class SuiteDeTestJUnit5 {

}
```

5 Conclusion

JUnit permet la construction de tests embarquant leur oracle. Il offre depuis ses dernières versions de grandes facilités pour écrire les tests ... mais parfois au prix d'un petit (grand ?) effort initial de compréhension. JUnit est parfaitement adapté pour le test unitaire et le test de non régression. Il fait partie de la grande famille des xUnit

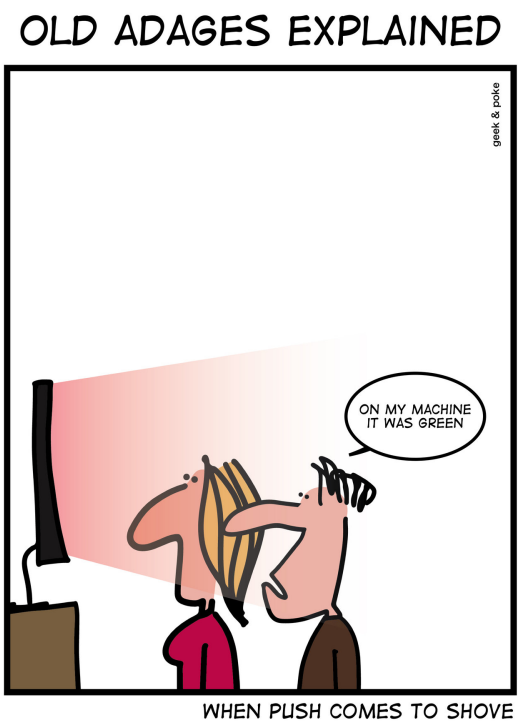
Destiné à faire du test unitaire, JUnit ne contient pas nativement de moyens pour tester une unité logicielle en isolation, quand elle dépend d'autres éléments. Pour parvenir à l'isolation d'une unité logicielle, on a souvent recours à la simulation de l'environnement, avec des outils de simulation pour les test unitaire : les mocks (mockito, easymock, ...)

Les tests unitaires sont écrits par des développeurs (mais pas nécessairement ceux qui ont développé le système sous test). Ils sont écrits le plus tôt possible, éventuellement avant d'écrire le système lui même (TDD, Test Driven development).

L'exécution "initiale" des tests unitaires permet de s'assurer de la qualité d'une unité logicielle, et permet aussi de s'assurer de la non régression : après chaque modification de l'unité logicielle, on relance les tests unitaires. Pour une exécution "continue" : placement des tests sur une plateforme d'intégration continue.

Le test unitaire n'est pas suffisant pour rester un logiciel, on doit également avoir recours au :

— test d'intégration
— test système
— test de recette


OLD ADAGES EXPLAINED
geek & poke
ON MY MACHINE IT WAS GREEN
WHEN PUSH COMES TO SHOVE