Manuel du développeur

Test unitaire en Java avec JUnit 4 (version avancée)

1. Introduction

Ce manuel complète le manuel de base.

2. Assertions personnalisées

Les méthodes de la classe `Assert` offrent des assertions prédéfinies d'égalité ou d'inégalité. Issue du projet Hamcrest, la méthode `assertThat` de cette même classe permet d'aller plus loin dans l'écriture des tests. Elle prend en paramètre un résultat attendu et un test personnalisé formulé à partir d'un ensemble de fonctions, prédicats et connecteurs logiques :

```
assertThat([objetATester], [leTest]);
```

L'intérêt de la méthode `assertThat` est double :

- Elle permet une meilleure lisibilité des tests. Les expressions suivantes de la formulation de tests d'inégalité :

```
assertThat(actual, is(not(equalTo(expected))));
assertThat(actual, is(false));
```

sont plus naturelles que la formulation classique :

```
assertFalse(expected.equals(actual));
assertFalse(expected);
```

- Elle permet surtout une grande flexibilité dans l'écriture des conditions. Il est possible d'exprimer des tests relativement complexes dont voici quelques exemples :

```
assertThat(x, is(3));
assertThat(x, is(not(4)));
assertThat("toto", is(allOf(notNullValue(), instanceOf(String.class),
  equalTo("toto"))));
assertThat("toto", anyOf(is("tata"), containsString("to")));
assertThat("toto", is(any(Object.class)));
assertThat(studentList, hasItem("toto"));
assertThat(people, everyItem(hasProperty("gender", is("male"))));
assertThat("toto", both(containsString("o")).and(containsString("t")))
assertThat("toto", either(containsString("to")).or(containsString("ta")))
```

2.1 Éléments d'assertion

Le test est construit par concaténation de fonctions, prédicats ou connecteurs logiques pour obtenir une combinaison de conditions. Les principales fonctions, prédicats et

connecteurs logiques sont donnés dans la Table 1.

Table 1 : Liste des principales méthodes de la classe assertThat.

`anything`	retourne toujours vrai ; utile si on ne veut pas se soucier de l'objet qui est testé.

`describedAs`	ajoute une description d'échec personnalisée. P. ex :
	`assertThat(a.getStudent(), is(not(nullValue())));`
	Le message d'erreur obtenu : « Expected : not null, Result : null »
	`assertThat(a.getStudent(), describedAs("student", is(not(nullValue()))));`
	Le message d'erreur obtenu : « Expected : student, Result : null »

`is`	teste la condition passée en paramètre. Par exemple, si le paramètre est une classe, alors il s'agit d'un test d'appartenance ; si le paramètre est une instance, alors il s'agit d'un test d'égalité.
`both`	vérifie que l'objet remplit deux conditions. Cette fonction se complète d'un `and` pour la deuxième condition. Elle est comparable à `allOf`, mais elle est limitée à deux conditions.
`either`	vérifie que l'objet remplit au moins une des deux conditions. Cette fonction se complète d'un `or` pour la deuxième condition. Elle est comparable à `anyOf`, mais elle est limitée à deux conditions.
`and, or`	connecteurs logiques utilisables avec `both` et `either`.
`allOf`	vérifie que toutes les conditions sont valides. `allOf` est comparable à `both` mais pour plusieurs valeurs.
`anyOf`	vérifie qu'une condition au moins est valide. `anyOf` est comparable à `either` mais pour plusieurs valeurs.
`not`	vérifie que la condition donnée est fausse.
`equalTo`	teste l'identité d'objet en utilisant la méthode `equals()`.
`equalToIgnoringCase`	teste une chaîne en ignorant la casse.
`equalToIgnoringWhiteSpace`	teste une chaîne string en ignorant les caractères 'blancs'.
`isA`	un raccourci pour `is(instanceOf(SomeClass.class))`.
`notNullValue, nullValue`	teste un objet par rapport à null.
`sameInstance`	teste l'identité d'objet en utilisant l'opérateur ==.
`hasItem, hasItems`	teste si une collection contient l'élément donné à `hasItem` ou les éléments donnés à `hasItems`.
`everyItem`	applique un test sur chaque élément d'une collection.
`closeTo`	teste des flottants à une erreur près .
`hasItemInArray`	teste qu'un tableau contient un élément.
`ContainsString, endsWith, startsWith`	teste la correspondance entre chaînes de caractères.

Le tableau Table 2 donne une correspondance entre l'écriture avec les assertions traditionnelles et celles avec assertThat.

Table 2 : Correspondance entre les assertions traditionnelles et assertThat.

Assertion	Équivalent assertThat
assertEquals(“expected”, “actual”);	assertThat(“actual”, is(“expected”));
assertArrayEquals(new String[] {“test1”, “test2”}, new String[] {“test3”, “test4”});	assertThat(new String[] {“test3”, “test4”}, is(new String[] {“test1”, “test2”}));
assertTrue(value) ; assertFalse(value);	assertThat(actual, is(true)); assertThat(actual, is(false));
assertNull(value); assertNotNull(value);	assertThat(actual, nullValue()); assertThat(actual, notNullValue());
assertSame(expected, actual); assertNotSame(expected, actual);	assertThat(actual, sameInstance(expected)); assertThat(actual, not(sameInstance(expected)));
assertTrue(1 > 3);	assertThat(1, greaterThan(3));
assertTrue(“abc”.contains(“d”));	assertThat(“abc”, containsString(“d”));

2.2 Création de nouveaux éléments d’assertion

JUnit offre la possibilité de construire ses propres fonctions, prédicats et connecteurs. Voici l'exemple d'un prédicat qui teste l'égalité de chaînes :

```
assertThat("toto", MyMatcher.myEquals("tata"));
```

Pour cela, il faut définir une classe qui dérive de `TypeSafeMatcher` :

```
1   public final class MyMatcher extends TypeSafeMatcher<String> {
2     private String _expected;
3     private MyMatcher( String expected ) {
4       _expected = expected;
5     }
6     @Override
7     public void describeTo( Description description ) {
8       description.appendValue(_expected);
9     }
10    @Override
11    protected boolean matchesSafely( String item ) {
12      return item.equals(_expected);
13    }
14    public static MyMatcher myEquals( String expected ) {
15      return new MyMatcher(expected);
16    }
17  }
```

3. Exécuter les tests dans un ordre déterminé

Le moteur d'exécution des méthodes de test par défaut de JUnit, nommé `BlockJUnit4ClassRunner`, n'utilise aucun ordre a priori sur les tests à effectuer. Cela suppose que les tests unitaires sont indépendants.

Toutefois, lorsque l’on développe une fonctionnalité, il arrive que l’on souhaite n’utiliser que quelques tests en particulier ou exécuter des tests dans un ordre spécifique. On peut penser aux cas des tests d’usage d’une collection d’une API où les tests enchaînent la création de la collection, l’insertion, la lecture et la suppression de données.

JUnit fournit deux mécanismes pour exécuter les tests dans un ordre déterminé.

Le premier impose un ordre pour les méthodes dans une classe. Le second regroupe des tests ciblés dans une suite.

3.1 Exécuter les tests d'une classe dans un ordre déterminé

Imposer un ordre sur les tests relève d'une mauvaise pratique puisque les tests unitaires doivent être indépendants les uns des autres. Mais dans certains cas très spécifiques, ce besoin peut exister.

JUnit fournit l'annotation `@FixMethodOrder` pour imposer un ordre d'exécution des méthodes de test. Il y a trois valeurs de paramètre possibles :

- MethodSorters.NAME_ASCENDING : exécute les méthodes de test dans l'ordre lexicographique de leur nom.
- MethodSorters.JVM : exécute les méthodes de test dans un ordre aléatoire. Cette commande peut varier d'une exécution à l'autre.
- MethodSorters.DEFAULT : c'est la valeur par défaut qui garantit un ordre d'exécution constant décidé par la JVM mais qui peut varier d'une JVM à l'autre.

Le code suivant impose un ordre d'exécution lexicographique pour tester trois méthodes d'une API fictive. L'ordre d'exécution est : `createTest()`, puis `editTest()` puis `removeTest()`.

```
1   @FixMethodOrder(MethodSorters.NAME_ASCENDING)
2   public final class TestExecutionOrder {
3      @Test
4      public void editTest() throws Exception {
5         ...
6      }
7      @Test
8      public void createTest() throws Exception {
9         ...
10     }
11     @Test
12     public void removeTest() throws Exception {
13        ...
14     }
15  }
```

3.2 Construire une suite de tests

L'annotation `@RunWith` avec le moteur `Suite.class` permet d'enchaîner plusieurs tests choisis dans un ordre donné. L'exemple qui suit enchaîne les tests des classes de test `JUnit1Test` et `JUnit2Test` après que les tests de `JUnit3Test` aient été exécutés. Il est même possible d'ajouter une suite de tests dans une autre suite de tests :

```
1   @RunWith(Suite.class)
2   @Suite.SuiteClasses({JUnit1Test.class, JUnit2Test.class})
3   public final class JUnit3Test {
4   }
```

Une suite de tests ne peut être exécutée qu'individuellement et c'est ce qui est

attendu puisqu'une suite est utilisée lors de la mise au point d'un code particulier pendant un temps donné. Elle n'est donc pas exécutée avec tous les tests lors de la vérification d'un projet. Avec IntelliJ, on exécute une suite de test de la même manière que l'on exécute un test individuel : il faut utiliser le menu contextuel de l'IDE qui est associé au fichier de la classe définissant la suite de test.

4. Ignorer des tests

Il est quelquefois nécessaire d'ignorer un test écrit. C'est utile par exemple lorsque la méthode à tester n'est pas encore prête ou que le test est devenu faux à cause d'un refactoring en cours. La façon la plus simple d'ignorer un test, c'est le mettre en commentaire. Le problème avec cette façon de faire, c'est que le test mis en commentaire risque d'être oublié au cours du temps et que le test ne sera plus actif jusqu'à ce qu'un développeur s'aperçoive par hasard qu'il avait été mis en commentaire. Une autre situation où il est important d'ignorer un test, c'est quand il dépend de l'environnement d'exécution tel que le type de système d'exploitation ou les ressources disponibles.

JUnit met à disposition des mécanismes pour ignorer des tests temporairement, conditionnellement ou par catégorie. Lorsque l'on exécute les tests, le moteur signale par un message les tests ignorés, ce qui permet au développeur de ne jamais oublier que les tests ignorés temporairement devront un jour être réintégrés.

4.1 Ignorer des tests temporairement

L'annotation `@Ignore` signifie d'ignorer une méthode de test. Un message peut être ajouté pour justifier sa mise à l'écart lors de l'exécution des tests.

```
1    public final class JUnitTest5 {
2       @Ignore("Not Ready to Run")
3       @Test
4       public void test1() {
5          ...
6       }
7    }
```

Si l'annotation `@Ignore` est mise sur la classe alors toutes les méthodes de test de la classe sont ignorées.

```
1    @Ignore("Not implemented correctly")
2    public final class JUnitTest6 {
3       @Test
4       public void test1(){
5          ...
6       }
7       @Test
8       public void test2(){
9          ...
10      }
11   }
```

4.2 Ignorer des tests conditionnellement

La classe `Assume` fournit des méthodes statiques qui peuvent être utilisées pour ignorer des tests à partir d'une condition.

Par exemple, le test suivant est ignoré dans un environnement Linux :

```
1    @Test
2    public void test1() {
3       assumeFalse(System.getProperty("os.name").contains("Linux"));
4       // test...
5    }
```

Dans cet autre exemple, le test n'est effectué que s'il est possible de se connecter à la base de données :

```
1    @Test
2    public void test2() {
3       DBConnection dbc = Database.connect();
4       assumeNotNull(dbc);
5       // test...
6    }
```

Les différentes méthodes utilisables pour ignorer un test sont données dans la Table 3.

Table 3 : Liste des méthodes utilisables pour définir des tests conditionnels.

`assumeNoException()`	vérifie qu'une opération s'est déroulée sans lever d'exception.
`assumeNotNull()`	vérifie qu'aucun des paramètres n'est égal à null.
`assumeThat()`	vérifie qu'une condition est respectée. La condition est construite avec les mêmes assertions que pour la méthode `assumeThat()` présentée en Section Error: Reference source not found.
`assumeTrue()/assumeFalse()`	vérifie que l'expression en paramètre est vraie / fausse.

4.3 Ignorer des tests par catégorie

Les tests peuvent être ignorés par catégorie. L'intérêt des catégories est, par exemple, d'ignorer les tests lents, des tests nécessitant une connexion particulière à un serveur ou des tests non *thread-safe*.

La mise en œuvre est très simple :

- Il faut définir des interfaces marqueurs (ie., *marker interface* ou *tagging interface*), c'est-à-dire des interfaces sans méthode, pour chaque catégorie de test souhaitée.
- Il faut ensuite étiqueter les méthodes de test (ou les classes) avec l'annotation `@Category`.
- Pour choisir les catégories à exécuter, il faut créer une classe qui utilise le moteur de test `Categories.class` et indiquer les catégories à inclure ou à exclure de l'exécution des tests.

L'exécution d'une catégorie consiste alors à exécuter individuellement la classe correspondante, avec le menu contextuel de l'IDE. Ceci fait que l'on ne peut exécuter

qu'une telle classe à la fois. Les catégories ne sont pas prises en compte quand on exécute tous les tests d'un projet en une seule fois. Donc, tous les tests sont exécutés dans ce cas sauf ceux annotés `@Ignore`.

L'exemple classique consiste à définir des catégories en fonction du temps d'exécution estimé et séparer les tests rapides exécutés durant la mise au point des tests lents exécutés avant l'intégration :

```
1   public interface FastTests { }
2
3   public interface SlowTests { }
4
5   public final class A {
6     @Test
7     public void test1() {
8     }
9
10    @Category(SlowTests.class)
11    @Test
12    public void test2() {
13    }
14  }
15
16  @Category({ SlowTests.class, FastTests.class })
17  public final class B {
18    @Test
19    public void test3() {
20    }
21  }
```

La classe suivante permet d'exécuter les tests catégorisés comme lents :

```
1   @RunWith(Categories.class)
2   @IncludeCategory(SlowTests.class)
3   @SuiteClasses({A.class, B.class})
4   public final class SlowTestSuite {
5     // Execute A.test2 et B.test3, mais pas A.test1
6   }
```

Cette autre classe permet aussi d'exécuter les tests lents en excluant ceux qui ont aussi été catégorisés comme rapides :

```
1   @RunWith(Categories.class)
2   @IncludeCategory(SlowTests.class)
3   @ExcludeCategory(FastTests.class)
4   @SuiteClasses({A.class, B.class})
5   public final class SlowTestSuite {
6   // Execute A.test2, mais ni A.test1 ni B.test3
7   }
```

Dans les deux cas, la méthode `A.test1()` n'est jamais exécutée puisqu'elle n'est dans aucune catégorie.

5. Tests paramétrés

Les tests paramétrés évitent d'avoir à dupliquer le code d'un même test pour différents jeux de données. Par exemple, pour tester un validateur de nombres premiers, la même méthode de test peut être utilisée pour les quatre jeux de

données 2, 6, 19 et 22.

5.1 Exemple introductif

Voici un cas de test paramétré qui teste une méthode de validation de nombres premiers :

```
1   @RunWith(Parameterized.class)
2   public final class PrimeNumbervValidatorTest {
3     private Integer _primeNumber;
4     private Boolean _expectedValidation;
5     private PrimeNumberValidator _primeNumberValidator;
6     public PrimeNumberValidatorTest( Integer primeNumber,
7                                      Boolean expectedValidation ) {
8       _primeNumber = primeNumber;
9       _expectedValidation = expected;
10    }
11    @Before
12    public void initialize() {
13      _primeNumberValidator = new PrimeNumberValidator();
14    }
15    @Parameters
16    public static Collection<Object[]> primeNumbers() {
17      return Arrays.asList(new Object[][] {
18                           { 2, true }, { 6, false },
19                           { 19, true }, { 22, false }});
20    }
21
22    @Test
23    public void testPrimeNumberValidator() {
24      assertEquals(_expectedValidation,
25                   _primeNumberValidator.validate(_primeNumber));
26    }
27  }
```

JUnit utilise pour cela le moteur d'exécution `Parameterized.class`. Une méthode statique annotée avec `@Parameters` retourne une `Collection` de tableaux de paramètres avec dans chaque tableau le jeu de données et la valeur attendue. Les éléments de chaque tableau retourné par cette méthode sont transmis au constructeur de la classe du test unitaire comme paramètres.

Les tests paramétrés sont exécutés comme des tests unitaires classiques. La méthode `testPrimeNumberValidator()` sera exécutée quatre fois puisqu'il y a 4 jeux de données.

5.2 Déclaration des paramètres

Une alternative à l'utilisation du constructeur pour récupérer les données de test consiste à déclarer des attributs publiques annotés par `@Parameter`. Le paramètre `value` de l'annotation permet de spécifier l'ordre d'initialisation.

Le cas de test précédent devient :

```
1   @RunWith(Parameterized.class)
2   public final class PrimeNumberValidatorTest {
```

```
3       private PrimeNumberValidator _primeNumberValidator;
4       @Parameter(value = 0)
5       public Integer primeNumber;
6       @Parameter(value = 1)
7       public Boolean expectedValidation;
8
9       @Before
10      public void initialize() {
11      _primeNumberValidator = new PrimeNumberValidator();
12      }
13      @Parameters
14      public static Collection<Object[]> primeNumbers() {
15      return Arrays.asList(new Object[][] {
16                { 2, true }, { 6, false },
17                { 19, true }, { 22, false }});
18      }
19      @Test
20      public void testPrimeNumberValidator() {
21      assertEquals(expectedValidation,
22                   primeNumberValidator.validate(primeNumber));
23  }
24  }
```

5.3 Amélioration du message d'erreur

L'annotation `@Parameters` autorise la spécification d'un nom pour chaque tableau, ce qui permet d'améliorer le message d'erreur généré en cas d'échec du test. Le nom est construit à partir des motifs suivants :

- {index} : représente la valeur d'index courante.
- {0}, {1},... : représentent les paramètres dans l'ordre du tableau.

La méthode statique de génération des jeux de données précédente devient :

```
1       @Parameters(name  = "{index}: prime number {0} = {1}")
2       public static Collection<Object[]> primeNumbers() {
3       return Arrays.asList(new Object[][] {
4                 { 2, true }, { 6, false },
5                 { 19, true }, { 22, false}});
```

L'exécution affiche les messages suivants :

```
0: prime number 2 = true
1: prime number 6 = false
2: prime number 19 = true
3: prime number 22 = false
```

5.4 Limites des tests paramétrés

Le fait que les jeux de tests sont appliqués systématiquement à toutes les méthodes de test interdit de pouvoir mettre dans une même classe des tests avec des jeux de tests différents et interdit aussi de mélanger des tests paramétrés avec des tests ordinaires.

6. JUnit Rules

Les règles JUnit permettent d'ajouter ou redéfinir un comportement aux méthodes d'une classe de test. Elles fonctionnent selon le principe de la programmation orientée aspect (AOP). Le moteur d'exécution intercepte chaque exécution des méthodes de test et applique des actions avant ou après l'exécution de la méthode ou empêche l'exécution de la méthode. L'intérêt est de pouvoir modifier le comportement de la méthode de test de l'extérieur.

Pour utiliser une règle dans une classe de test, il faut déclarer un attribut public dans la classe, l'annoter avec `@Rule` et l'initialiser avec une instance de la classe de la règle choisie. Reste ensuite à utiliser cette règle selon ses prescriptions dans des méthodes de test de la classe.

6.1 Les règles prédéfinies

JUnit fournit plusieurs implémentations de règles.

6.1.1 La règle TemporaryFolder

Cette règle permet la création de fichiers et de dossiers temporaires à l'intérieur d'une méthode de test qui seront détruits à la fin du test, que le test passe ou non.

```
1    public static class HasTempFolderTest {
2       @Rule
3       public final TemporaryFolder folder = new TemporaryFolder();
4       @Test
5       public void testUsingTempFolder() throws IOException {
6          File createdFile = folder.newFile("myfile.txt");
7          File createdFolder = folder.newFolder("subfolder");
8          // ... code de la méthode de test
9       }
10   }
```

La méthode `newFile()` sans paramètre crée un nouveau fichier nommé automatiquement, et `newFolder()` crée un nouveau dossier nommé automatiquement. La méthode `newFolder(String folderNames)` de `TemporaryFolder` crée des dossiers temporaires récursivement.

Par défaut, aucune exception n'est levée si les ressources ne peuvent être détruites. Toutefois, `TemporaryFolder` permet en option de vérifier strictement les ressources supprimées. La règle lève l'exception `AssertionError` si les ressources ne peuvent pas être supprimées. Cette fonctionnalité ne peut être utilisée qu'avec la méthode `builder()`.

```
1 @Rule
2 public TemporaryFolder dir =
  TemporaryFolder.builder().assureDeletion().build();
```

6.1.2 La règle Timeout

La règle `Timeout` correspond strictement à l'annotation `@Test(timeout)`. Elle permet de vérifier la durée maximale d'exécution de toutes les méthodes de test de

la classe.

```
1    public static class HasGlobalTimeoutTest {
2
3       @Rule
4       public final TestRule globalTimeout = Timeout.millis(20);
5
6       @Test
7       public void testInfiniteLoop1() {
8          JUnit2.infiniteLoop();
9       }
10
11      @Test
12      public void testInfiniteLoop2() {
13         JUnit2.infiniteLoop();
14      }
15   }
```

6.1.3 La règle ExpectedException

La règle `ExpectedException` est calquée sur l'annotation `@Test(expected=exception.class)` qui permet de vérifier qu'une exception a bien été levée lors de l'exécution d'un test. Mais, en plus de la vérification du type, elle autorise aussi la vérification du message d'exception attendu. La méthode `expect()` définit l'exception attendue et `expectMessage` définit le message d'exception attendu.

```
1    public static class CalculatorTest {
2       @Rule
3       public final ExpectedException thrown = ExpectedException.none();
4
5       @Test
6       public void should_throw_exception_when_dividing_by_zero() {
7          thrown.expect(ArithmeticException.class);
8          thrown.expectMessage("/ by zero");
9          JUnit3.div(1,0);
10      }
11   }
```

L'objet thrown est réinitialisé après chaque exécution d'une méthode de test.

6.1.4 La règle ExternalResource

La règle `ExternalResource` est utilisée pour réserver une ressource externe (fichier, socket, serveur, base de données, etc.) avant l'exécution d'un test. La ressource est relâchée après chaque test.

```
1    public static class UsesExternalResourceTest {
2       Server _myServer = new Server();
3
4       @Rule
5       public final ExternalResource resource = new ExternalResource() {
6          @Override
7          protected void before() throws Throwable {
8             _myServer.connect();
9          }
10         @Override
```

```
11          protected void after() {
12             _myServer.disconnect();
13          }
14       }
15
16       @Test
17       public void testFoo() {
18          new Client().run(_myServer);
19       }
20    }
```

C'est en fait une règle plus générale qui peut être utilisée pour faire de la programmation orientée aspect dans le sens où elle peut ajouter du code avant et après l'exécution du code des méthodes de test de la classe.

6.1.5 La règle ErrorCollector

Lors de l'écriture de scripts de test, il peut être utile d'exécuter tous les tests même si une ligne de code échoue par exemple en raison d'une défaillance du réseau ou d'une erreur d'assertion dans une base de données. La règle `ErrorCollector` permet de poursuivre l'exécution d'un test après l'échec d'une assertion. Un objet `ErrorCollector` collecte toutes les erreurs et reporte leur affichage après l'exécution de toutes les méthodes de la classe.

Dans la classe `ErrorCollector`, la méthode `checkThat()` permet d'effectuer une assertion et de collecter l'erreur si l'assertion est fausse. La méthode `addError()` permet d'ajouter une erreur au collecteur.

```
1     public static class ErrorCollectorTest {
2        @Rule
3        public ErrorCollector collector = new ErrorCollector();
4        @Test
5        public void should_fail_after_execution() {
6           collector.checkThat("a", equalTo("b"));
7           collector.checkThat(1, equalTo(2));
8           try {
9              Assert.assertTrue("a" == "b");
10          } catch (Throwable t) {
11             collector.addError(t);
12          }
13       }
14    }
```

Dans l'exemple précédent, aucune des vérifications ne passe mais le test s'exécute jusqu'à la fin. Les erreurs ne sont affichées qu'à la fin.

6.1.6 La règle Verifier

La règle `Verifier` permet de mettre un test en échec si la vérification d'un test interne échoue. Le test interne est commun à toutes les méthodes de test de la classe. Il est appliqué à chaque exécution d'une méthode de test.

```
1     public static class ErrorLogVerifierTest {
2        private ErrorLog _errorLog = new ErrorLog();
3        @Rule
```

```
4       public Verifier verifier = new Verifier() {
5         @Override
6         public void verify() {
7           assertTrue(_errorLog.isEmpty());
8         }
9       }
10      @Test
11      public void testThatMightWriteErrorLog() {
12        // ... met à jour ou non la variable _errorLog.
13      }
14    }
```

6.1.7 La règle TestWatcher

La règle `TestWatcher` permet d'inspecter l'exécution d'une méthode test pour en récupérer les données d'exécution sans modifier le comportement. Par exemple, la classe suivante conserve un journal des succès et des échecs d'exécution. À la fin de l'exécution, la règle produit un rapport récapitulatif.

```
1     public final class WatchmanTest {
2       public static String watchedLog;
3       @Rule
4       public final TestRule watchman = new TestWatcher() {
5         @Override
6         public Statement apply( Statement base, Description description )
    {
7           return super.apply(base, description);
8         }
9         @Override
10        protected void succeeded( Description description ) {
11          watchedLog += description.getDisplayName() + " " + "success!\
    n";
12        }
13        @Override
14        protected void failed( Throwable e, Description description ) {
15          watchedLog += description.getDisplayName() + " failed because:
    " +
16                e.getClass().getSimpleName() + "\n";
17        }
18        @Override
19        protected void skipped( AssumptionViolatedException e,
20                                Description description ) {
21             watchedLog += description.getDisplayName()
22                 + " skipped due to assumption " +
23                 e.getClass().getSimpleName() + "\n";
24          }
25        @Override
26        protected void starting( Description description ) {
27          super.starting(description);
28        }
29        @Override
30        protected void finished( Description description ) {
31          super.finished(description);
32        }
33      }
34      @Test
35      public void fails() {
36        fail();
37      }
```

```
38      @Test
39      public void succeeds() {
40        assertTrue(true);
41      }
42    }
```

6.1.8 La règle TestName

La règle `TestName` rend le nom de la méthode de test disponible dans la méthode de test. Le code suivant montre comment le nom du test peut être utilisé dans le test.

```
1     public final class NameRuleTest {
2       @Rule
3       public final TestName name = new TestName();
4       @Test
5       public void testA() {
6         assertEquals("testA", name.getMethodName());
7       }
8       @Test
9       public void testB() {
10        assertEquals("testB", name.getMethodName());
11      }
12    }
```

6.1.9 La règle RuleChain

La règle `RuleChain` permet d'ordonner des `Rules`. On crée une `RuleChain` avec `outerRule` puis on enchaîne les exécutions par `around`.

```
1     public class RuleChainTest {
2       private static final List<String> LOG = new ArrayList<String>();
3
4       private static class LoggingRule extends TestWatcher {
5       private final String _label;
6
7         public LoggingRule( String label ) {
8           _label = label;
9         }
10        @Override
11        protected void starting( Description description ) {
12          LOG.add("starting " + _label);
13        }
14        @Override
15        protected void finished( Description description ) {
16          LOG.add("finished " + _label);
17        }
18      }
19
20      public static class UseRuleChain {
21        @Rule
22        public RuleChain chain = outerRule(new LoggingRule("outer rule"))
23          .around(new LoggingRule("middle rule"))
24          .around(new LoggingRule("inner rule"));
25        @Test
26        public void example() {
27          assertTrue(true);
28        }
29      }
```

```
30    }
```

produit le résultat suivant :

```
starting outer rule
starting middle rule
starting inner rule
finished inner rule
finished middle rule
finished outer rule
```

6.2 L'annotation ClassRule

L'annotation `@ClassRule` étend l'idée des règles `@Rule` aux cas de méthodes statiques qui peuvent affecter le fonctionnement d'une classe entière sur le principe des méthodes statiques annotées `@BeforeClass` et `@AfterClass`.

Toutes les classes de moteur sous-classes de `ParentRunner`, incluant le moteur par défaut `BlockJUnit4ClassRunner` et `Suite`, supporte l'annotation `ClassRule`.

Par exemple, voici une suite de tests qui se connectent à un serveur une fois avant que toutes les méthodes de test de la classe ne s'exécutent et se déconnectent une fois après qu'elles sont toutes terminées.

Cette première version utilise la règle `ExternalResource` :

```
1     @RunWith(Suite.class)
2     @SuiteClasses({A.class, B.class, C.class})
3     public final class UsesExternalResource1Test {
4       public static final Server _myServer = new Server();
5
6       @ClassRule
7       public static final ExternalResource resource = new
    ExternalResource() {
8         @Override
9         protected void before() throws Throwable {
10          _myServer.connect();
11        }
12        @Override
13        protected void after() {
14          _myServer.disconnect();
15        }
16      }
17    }
```

Cette seconde version utilise une méthode statique :

```
1     @RunWith(Suite.class)
2     @SuiteClasses({A.class, B.class, C.class})
3     public class UsesExternalResource2Test {
4       public static final Server _myServer = new Server();
5
6       @ClassRule
7       public static final  ExternalResource getResource() {
8         return new ExternalResource() {
9           @Override
10          protected void before() throws Throwable {
11            _myServer.connect();
```

```
12            }
13            @Override
14            protected void  after() {
15               _myServer.disconnect();
16            }
17         }
18      }
19   }
```

Manque la classe @Test

6.3 Créer de nouvelles règles

6.3.1 Par extension de la règle ExternalResources

La plupart des règles personnalisées peuvent être implémentées par extension de la règle `ExternalResource`.

Un exemple

6.3.2 Par implémentation de l'interface TestRule

Toutefois, s'il faut utiliser plus d'information sur la classe de test ou la méthode en question, il faut créer une classe qui implémente l'interface `TestRule`. Cette interface définit la méthode `apply(Statement, Description)` qui doit renvoyer une instance de `Statement`. Le paramètre de type `Statement` contient le code du test en cours d'exécution et sa méthode `evaluate()` permet de l'exécuter. Le paramètre de type `Description` contient la description de la méthode test. Il permet de lire des informations sur le test via le paquet *Reflection* de Java. Le code de la méthode `apply()` doit retourner une nouvelle instance de `Statement` qui redéfinit la méthode `evaluate()` en ajoutant ou supprimant des instructions au code initial du test.

6.3.3 Exemple de création d'une règle

La règle de test suivante fournit un collecteur de log pour chaque test.

```
1    public final class TestLogger implements TestRule {
2       private Logger _logger;
3
4       public Logger getLogger() {
5          return _logger;
6       }
7
8       @Override
9       public Statement apply( Statement base, Description description ) {
10         return new Statement() {
11            @Override
12            public void evaluate() throws Throwable {
13            // instructions avant l'exécution du code de test (cf. @Before)
14               _logger =
   Logger.getLogger(description.getTestClass().getName()
15                                                + '.'
16                                                +
```

```
   description.getDisplayName());
17             base.evaluate(); // Exécution du code initial du test.
18             // instructions après l'exécution du code de test (cf.
   @After)
19           }
20         }
21       }
22     }
```

La règle peut être appliquée dans une classe de test :

```
1    public final class MyLoggerTest {
2
3       @Rule
4       public final TestLogger logger = new TestLogger();
5
6       @Test
7       public void checkOutMyLogger() {
8          final Logger log = logger.getLogger();
9          log.warning("Your test is showing!");
10      }
11   }
```

Cet autre exemple associe la définition d'une règle avec l'utilisation de l'annotation `ClassRule` :

```
1    public class LoggingRule implemnets TestRule {
2       public class LoggingStatement extends Statement {
3          private final Statement _statement;
4          public LoggingStatement( Statement aStatement, String aName ) {
5             _statement = aStatement;
6          }
7          @Override
8          public void evaluate() throws Throwable {
9             System.out.println("before: " + _name);
10            _statement.evaluate();
11            System.out.println("after: " + _name);
12         }
13      }
14      private final String _name;
15      public LoggingRule( String aName ) {
16         _name = aName;
17      }
18      @Override
19      public Statement apply( Statement statement, Description description
   ) {
20         System.out.println("apply: " + _name);
21         return new LoggingStatement(statement, _name);
22      }
23   }
```

La classe suivante est un exemple de l'utilisation dans un test :

```
1    public class RuleTest {
2       @ClassRule
3       public static LoggingRule classRule = new LoggingRule("classrule");
4       @Rule
5       public static LoggingRule rule = new LoggingRule("rule");
6
7       @Test
8       public void testSomething() {
9          System.out.println("* In TestSomething");
```

```
10          assertTrue(true);
11       }
12
13       @Test
14       public void testSomethingElse() {
15          System.out.println("* In TestSomethingElse");
16          assertTrue(true);
17       }
18 }
```

La sortie de l'exécution donne :

```
apply: classrule
before: classrule
apply: rule
before: rule
* In TestSomething
after: rule
apply: rule
before: rule
* In TestSomethingElse
after: rule
after: classrule
```

7. Utiliser plusieurs moteurs

Dans JUnit, il est impossible d'utiliser plusieurs moteurs en même temps. Par exemple cette situation ne compile pas :

```
1     @RunWith(MockitoJUnitRunner.class)
2     @RunWith(JUnitParamsRunner.class)
3     public class DatabaseModelTest {
4        // some tests
5     }
```

En général, les règles permettent régler le problème. Par exemple, le cas précédent peut se régler en utilisant le moteur `JUnitParamsRunner.class` et la règle de Mockito :

```
1     @RunWith(JUnitParamsRunner.class)
2     public class DatabaseModelTest {
3        @Rule
4        public MockitoRule mockitoRule = MockitoJUnit.rule();
5        // some tests
6     }
```