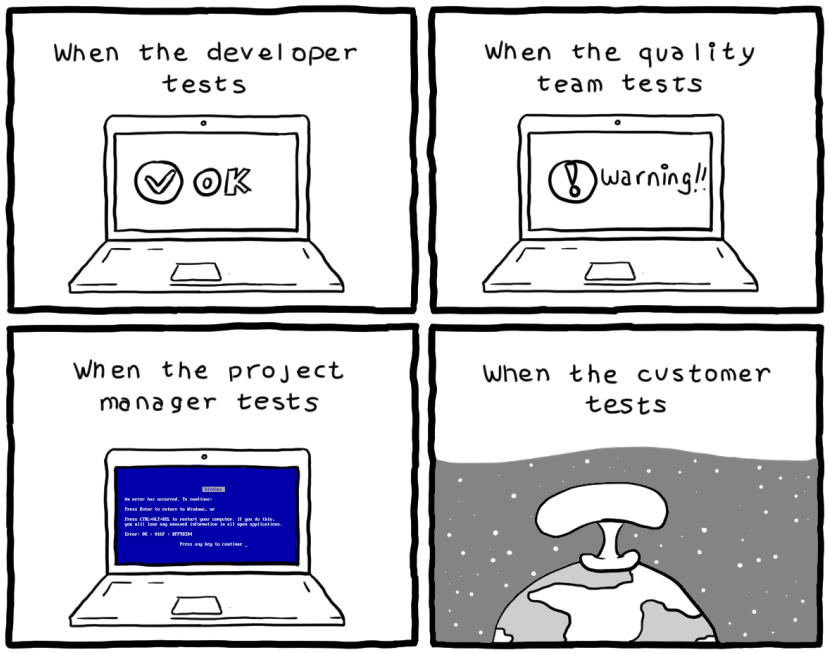
Une introduction au test logiciel et au test unitaire avec JUnit



Faculté des sciences – Université de Montpellier

2021

Sommaire

Sommaire

Le test

Principe

Essayer pour voir si ça marche ...

Essayer ...

- Comment ça marche ?
  - Démarrage du programme ?
  - Interface graphique ? Textuelle ?
  - ça marche comment une API ?
- Quelles entrées ?
  - Données requises ?
- Qu'est-il possible de faire ?
  - Si on veut tout essayer, il faut savoir ce qu'il y a à essayer !
  - Quels enchaînements nécessaires pour essayer une fonctionnalité ?

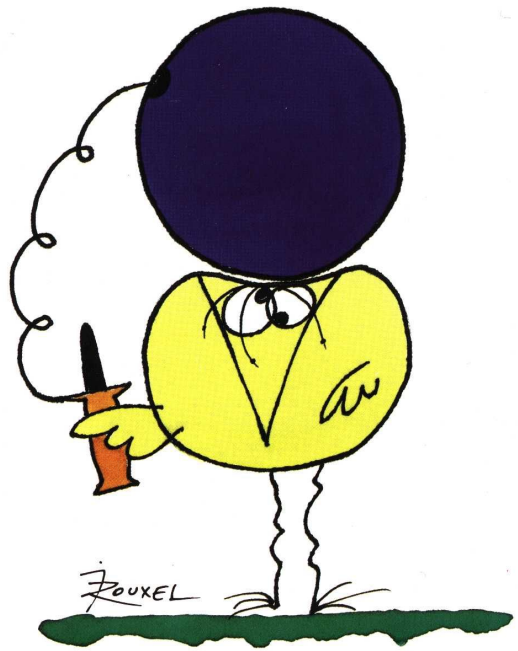
... pour voir ...

- Que peut-on voir ?
  - une couleur dans une interface graphique ?
  - un affichage dans une fenêtre ?
  - la valeur d'une variable ?
  - le résultat d'un calcul intermédiaire ?
- Notion d'observabilité

... si ça marche.

- Comment sait-on que ça marche ?
  - au fait, il doit faire quoi ce programme ?
    - notion de spécifications
  - à partir de ce que l'on peut voir, déterminer si ça marche
    - et si on ne voit pas ce que l'on veut ?
- Et si ça ne marche pas ?
  - Diagnostique
- Et si ça a l'air de marcher ...
  - est-on sûr que ça marche vraiment ?
    - notion de confiance $\neq$certitude
  - et si c'étaient les tests qui étaient mauvais ou insuffisants ?
    - qualité des tests, critère d'arrêt

Les devises Shadok



EN ESSAYANT CONTINUELLEMENT
ON FINIT PAR RÉUSSIR. DONC:
PLUS ÇA RATE, PLUS ON A
DE CHANCES QUE ÇA MARCHE.

Vers une définition ...

Définition de Myers, 1979

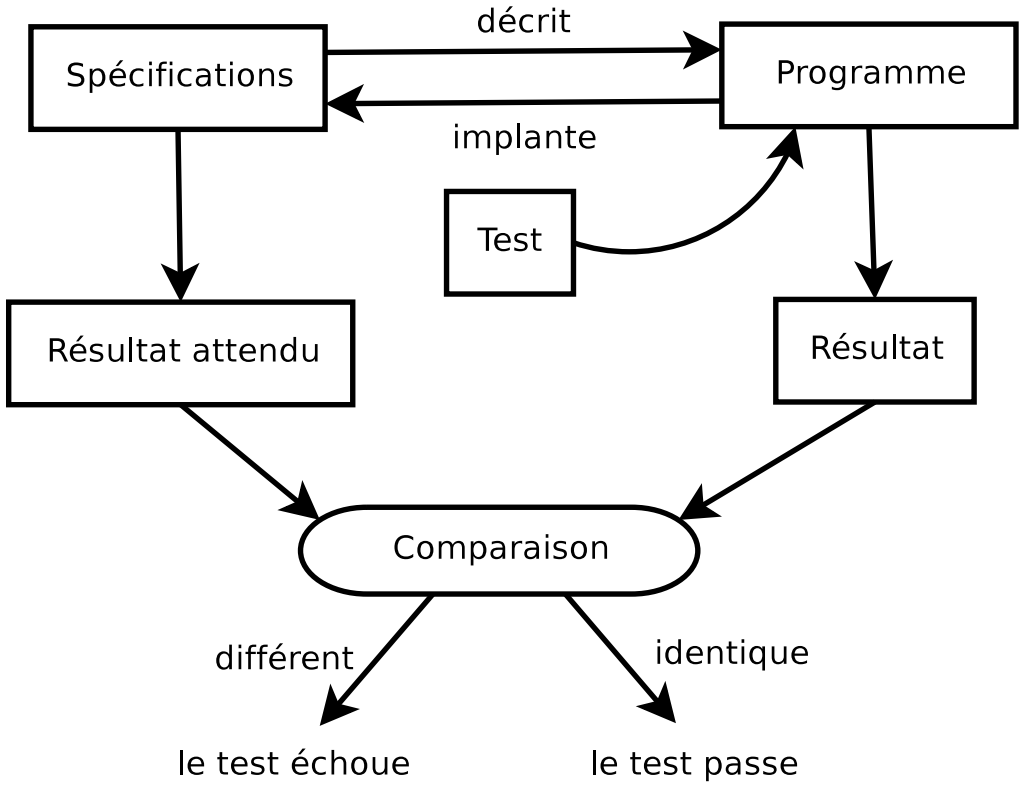
Testing is the process of executing a program with the intent of finding errors. [G. Myers. The Art of Software Testing. 1979]

Qu'est ce qu'on teste ?
(quelles propriétés)

Différentes propriétés à tester

- satisfaction des fonctionnalités requises
- qualité de service (temps de réponse, utilisation mémoire, ...)
- robustesse
- sûreté de fonctionnement
- utilisabilité

Le verdict ...


décrit
Spécifications
Programme
implante
Test
Résultat attendu
Résultat
Comparaison
différent
identique
le test échoue
le test passe

Sommaire

Différents tests

Plusieurs niveaux (échelles)

- Unitaire
- Intégration
- Système
- Acceptation (ou recette)

Différents niveaux d'accessibilité

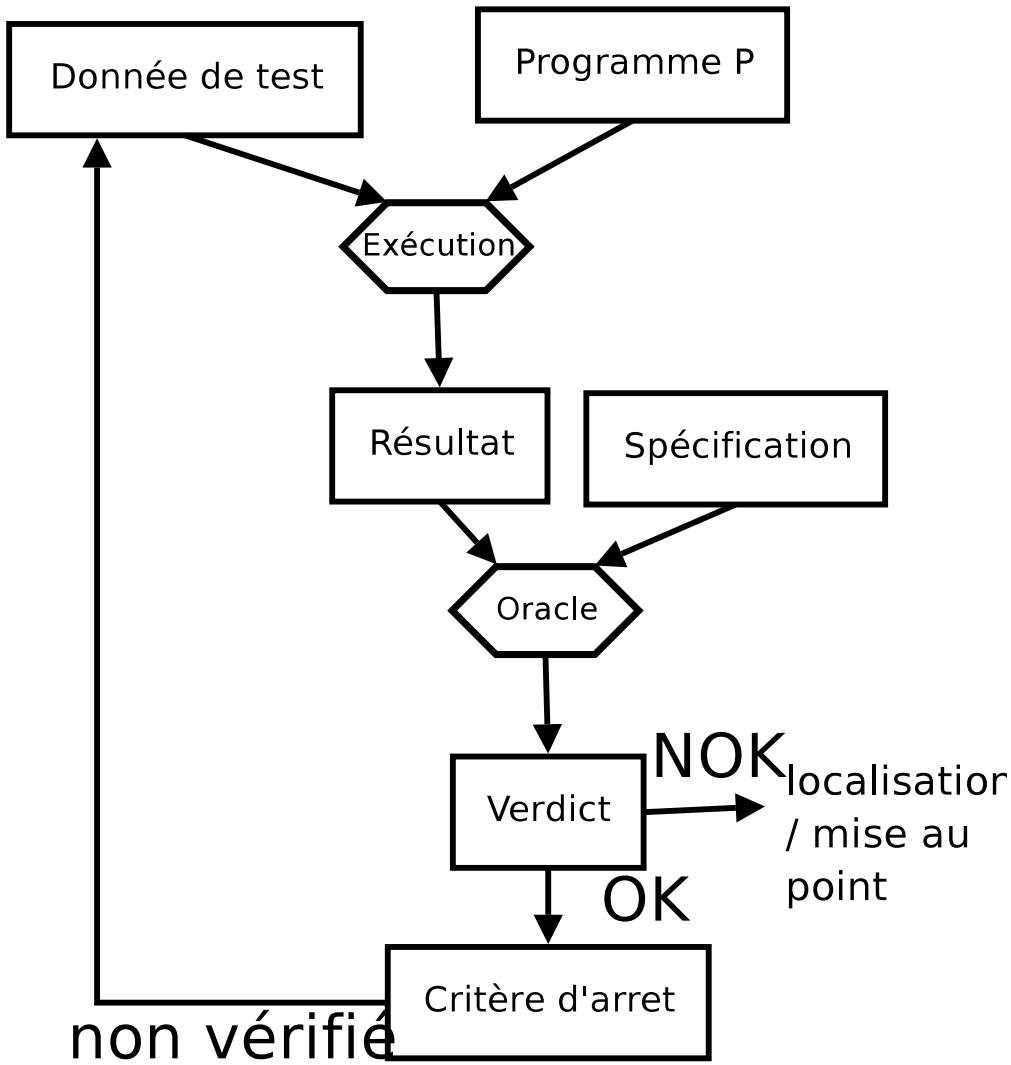
- Test boîte noire (souvent fonctionnel)
- Test boîte blanche (souvent structurel)
- Test boîte grise ?

Plusieurs types classiques

- test fonctionnel
- test de non-régression
- test de montée en charge

Sommaire

Processus


Donnée de test
Programme P
Exécution
Résultat
Spécification
Oracle
Verdict
NOK
localisation
/ mise au
point
OK
Critère d'arret
non vérifié

Vocabulaire

Oracle

- Aussi appelé fonction d'oracle
- Permet de déterminer si le test a réussi ou échoué
  - ie si le résultat obtenu est celui attendu

Critère d'arrêt

- Permet de déterminer si on a fini de tester

Les difficultés

La génération des données de test

- Comment les choisir ? Sur quels critères ?
- Si on en choisit trop, c'est long / cher !
- Il existe des techniques de génération automatique

L'oracle

- Comment savoir si ce qu'on a obtenu est correct ?
  - faire le calcul à la main ?
  - utiliser un autre programme ?
  - en théorie : utiliser la spécification ...
  - en pratique : utiliser la spec, les propriétés du programme, versions antérieures

Les difficultés

Le critère d'arrêt

- Comment savoir quand il n'est plus nécessaire de tester ?
  - on ne trouve plus d'erreurs depuis 5 minutes ?
  - on n'a plus de temps ?
  - on a passé 10h à tester ?
  - on a exécuté une fois chaque intruction ?
  - on a fait au moins 3 tours dans chacune des boucles ?

Sommaire

JUnit

- Origine
  - Xtreme programming (test-first development), méthodes agiles
  - framework de test écrit en Java par E. Gamma et K. Beck
  - open source : www.junit.org
- Objectifs
  - test d'applications en Java
  - faciliter la création des tests
  - tests de non régression

Ce que fait 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
- Permet en un seul clic de savoir quels tests ont échoué/planté/réussi

JUnit (et au delà xUnit) est de facto devenu un standard en matière de test

Ce que ne fait pas JUnit

- JUnit n'écrit pas les tests !
- Il ne fait que les lancer.
- JUnit ne propose pas de principes/méthodes pour structurer les tests

JUnit : un framework

- Le framework définit toute l'infrastructure nécessaire pour :
  - écrire des tests
  - définir leurs oracles
  - lancer les tests
- Utiliser Junit :
  - définir les tests
  - s'en remettre à JUnit pour leur exécution
  - ne pas appeler explicitement les méthodes de test

JUnit : versions initiales, versions 4, versions 5 (jupiter)

Versions initiales

- Paramétrage par spécialisation
- Utilisation de conventions de nommage

Versions 4

- Utilisation d'annotations
- beaucoup de nouvelles fonctionalités dans JUnit 4
- pas de runner graphique en version 4, laissé au soin des IDEs

Versions 5 (Jupiter)

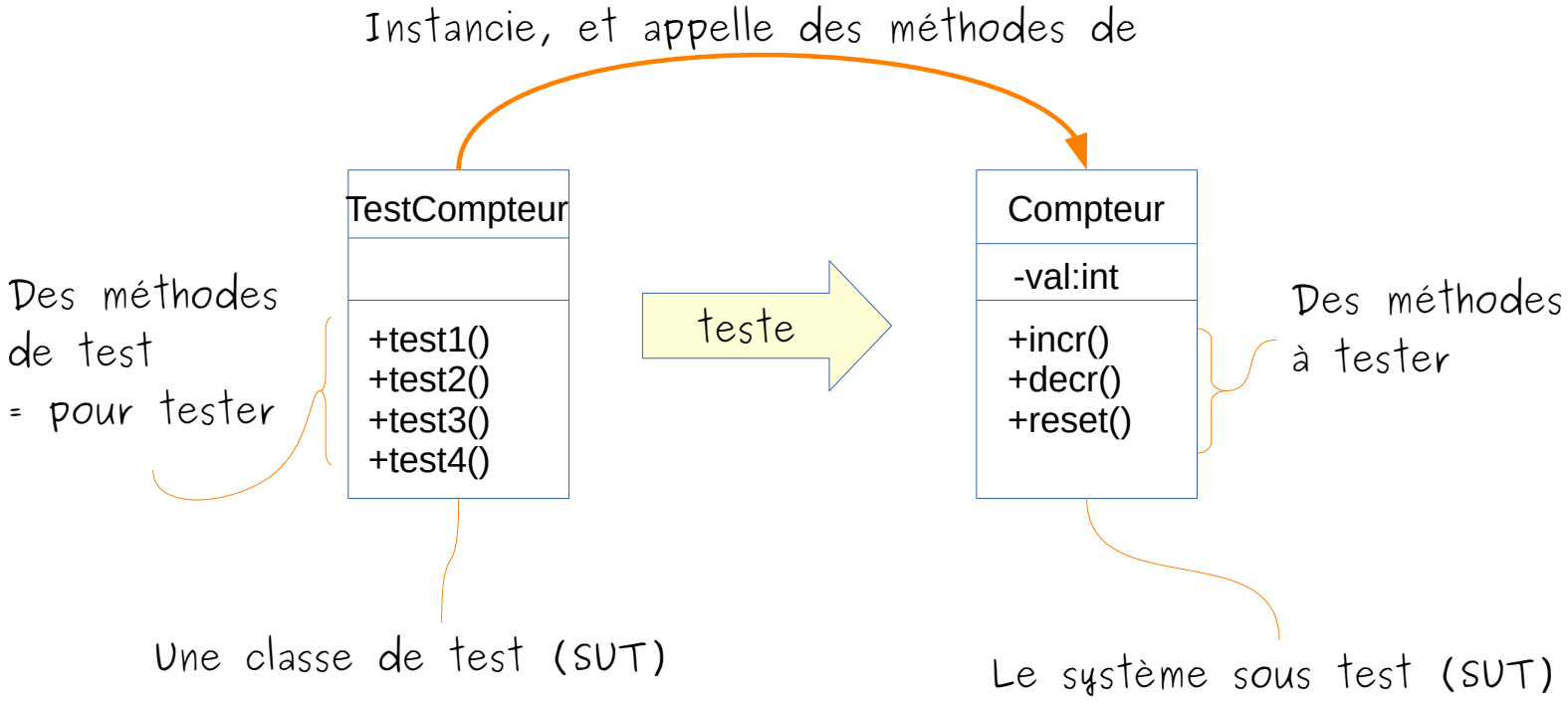
- Utilisation intensive de lambdas
- JUnit versions $\leq$4 dans un package vintage !

Sommaire

Écriture de test : principe général

- On crée une ou plusieurs classes destinées à contenir les tests : les classes de test.
- On y insère des méthodes de test.
- Une méthode de test
  - fait appel à une ou plusieurs méthodes du système à tester (communément appelé SUT, System Under Test),
  - ce qui suppose d'avoir une instance d'une classe du système à tester (la création d'une telle instance peut être placée à plusieurs endroits, voir plus loin),
  - inclut des instructions permettant un verdict automatique : les assertions.

Écriture de test : principe général

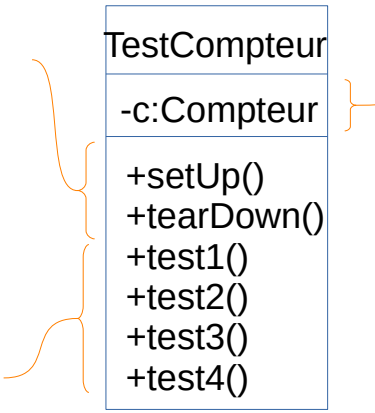


Classe de test

- Contient les méthodes de test (**sans ordre**)
- peut contenir des méthodes particulières pour positionner l'environnement de test (souvent stocké en attribut)
- En JUnit :
  - Junit versions <4 : la classe de test hérite de JUnit.framework.TestCase
  - JUnit versions ≥ 4 : une classe quelconque
  - Jupiter : une classe quelconque

Classe de test

Des méthodes de contrôle de l'environnement de test (init notamment)



L'environnement de test, souvent au minimum une instance du SUT

Des méthodes de test

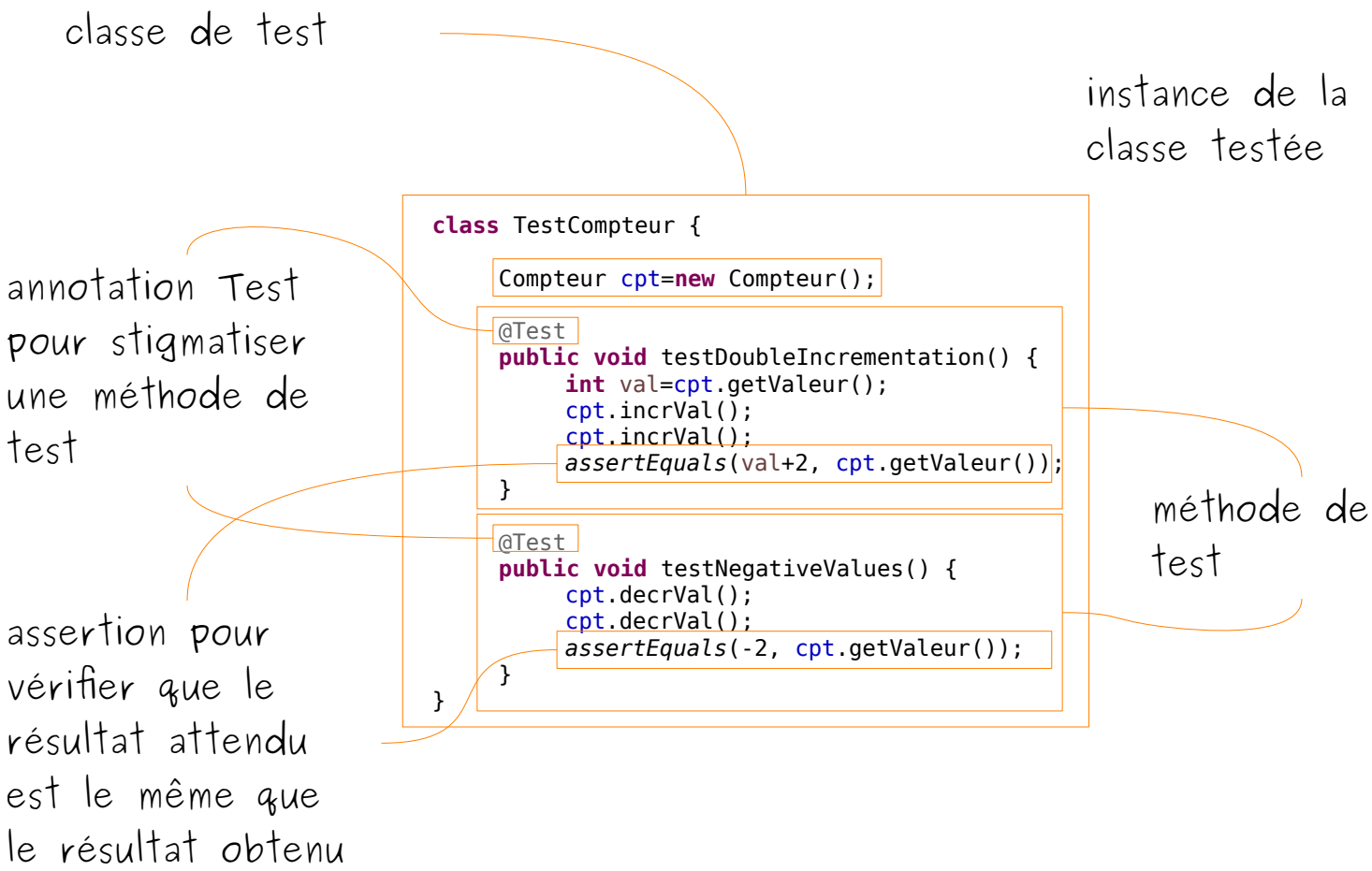
Méthode de test

- s'intéresse à une seule unité de code/ un seul comportement
- doit rester courte
- les méthodes de test sont indépendantes les unes des autres (pas d'ordre !)
  - Junit versions <4 : les méthodes de test commencent par le mot `test`
  - JUnit versions ≥ 4 : annotées `@Test`
- les méthodes de test seront appelées par Junit, dans un ordre supposé **quelconque**.

Les méthodes de test

- sont sans paramètres et sans type de retour (logique puisqu'elles vont être appelées automatiquement par JUnit)
- appellent des méthodes du SUT
- embarquent l'oracle
- i.e. contiennent des assertions
  - x vaut 3
  - le résultat de l'appel de telle méthode est non nul
  - x est plus petit que y
- JUnit introduit des assertions plus riches que le assert Java + utilisation d'Hamcrest (un petit DSL interne)

Un exemple de classe de test et de méthode de test en JUnit 4 ou 5

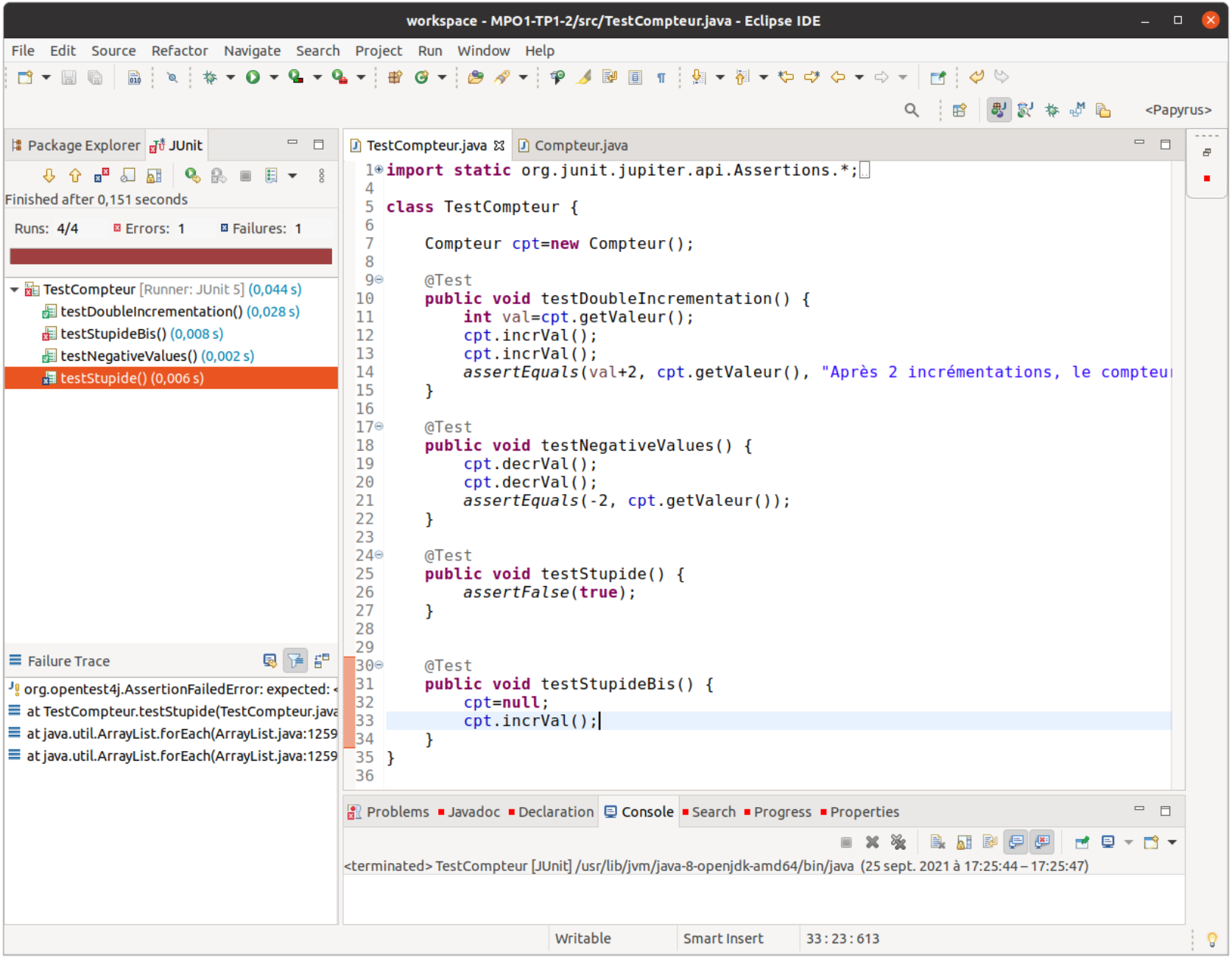


Les verdicts

Sont définis grâce aux assertions placées dans les cas de test.

- Pass (vert) : pas de faute détectée
- Fail (rouge) : échec, violation d'assertion (on attendait un résultat, on en a eu un autre)
- Error : le test n'a pas pu s'exécuter correctement (exception inattendue)

Exécution et verdicts



Exemple – classe à tester

Des heures entre 7h et 23h, avec une granularité de 5 minutes

```
1   public class Heure {
        private int heures, minutes;
3       private static int granulariteMinutes=5;
        private static int heureMax=22;
5       private static int heureMin=7;

7       private boolean heuresCorrectes(){
            return heures>=heureMin && heures<=heureMax;
9       }
        private boolean minutesCorrectes(){
11          boolean result=minutes%granulariteMinutes==0;
            if (heures==heureMax&&minutes!=0)result=false;
13          return result;
        }
15      public Heure(int heures,int minutes) throws HoraireIncorrectException{
            this.heures=heures;
17          this.minutes=minutes;
            if (!heuresCorrectes()||!minutesCorrectes()){
19              throw new HoraireIncorrectException("heure specifiee incorrecte");
            }
21      }
        public String toString(){
23          String h=Integer.toString(heures);
            String mn=Integer.toString(minutes);
25          // ajout des 0 non significatifs
            if (heures<10)h="0"+h;
27          if (minutes<10)mn="0"+mn;
            return h+":"+mn;
29      }
    }
```

Exemple – objectif de test : le toString est correct pour des heures correctes

```
import static org.hamcrest.MatcherAssert.assertThat;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class TestHeures {
    Heure h1, h2, h3, h4, h5, 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
    }
}
```

Remarques

- L'exécution d'une méthode test s'arrête à la première assertion violée
- Donc il est déconseillé d'écrire plusieurs assertions dans le même test
- Plus généralement, l'objectif de test est très vague ici, il serait mieux de faire plusieurs méthodes :
  - testToStringAvecAucunChiffreNonSignificatif
  - testToStringAvec0Minutes
  - testToStringAvecUnitéHeuresNulle
  - etc
- le BeforeEach devient alors inutile si l'on n'a que ces méthodes de test dans la classe.

Exemple – objectif de test : la création d'heures incorrectes lance une HoraireIncorrectException

```
1   import org.junit.jupiter.api.BeforeEach;
    import org.junit.jupiter.api.Test;
3
    public class TestHeures {
5       @Test
        public void testCreationHeureInvalideDepasseHeureMax() {
7           assertThrows(HoraireIncorrectException.class, () -> {
                new Heure(23,05);
9           });
        }
11
        @Test
13      public void testCreationHeureInvalideAvantHeureMin()  {
            assertThrows(HoraireIncorrectException.class, () -> {
15              new Heure(6,10);
            });
17      }

19      @Test
        public void testCreationHeureInvalideGranulariteFausse()  {
21          assertThrows(HoraireIncorrectException.class, () -> {
                new Heure(7,12);
23          });
        }
25  }
```

Remarques

- Nous reviendrons un peu plus tard sur l'utilisation de lambda ici
- Ici chaque méthode vise bien à tester une seule chose, et le nom de la méthode de test exprime ce que l'on veut tester
- Le test échoue si soit aucune exception n'est levée, soit une exception est levée, mais pas du bon type.

Exemple – objectif de test : test de la méthode estAvant

```
1   public boolean estAvant(Heure autreHeure) {
            ...
3   }

5   public boolean estStrictementAvant(Heure autreHeure) {
            ...
7   }

9   public class TestHeures {
        Heure h1, h2, h3, h4, h5, h6;
11      @BeforeEach
        public void setUp() throws HoraireIncorrectException{
13          h1=new Heure(10,15);
            h2=new Heure(21,55);
15          h3=new Heure(8,10);
            h4=new Heure(22,0);
17          h5=new Heure(12,05);
            h6=new Heure(8,15);
19
        }
21      @Test
        public void testEstAvant(){
23          assertFalse(h1.estStrictementAvant(h1));
            assert(h1.estAvant(h2));
25          assert(h1.estAvant(h4));
            assert(h1.estAvant(h5));
27          assert(h2.estAvant(h4));
            assert(h3.estAvant(h6));
29      }
```

Remarques

- Encore une fois ici, il y a trop d'assertions dans la même méthode.
- On pourrait séparer (et mieux tester) ce qui concerne estStrictementAvant
- Il ne semble pas y avoir d'objectif de test bien précis ici, au minimum faudrait-il faire en sorte que toutes les assertions s'exécutent, même en cas de violation de l'une d'elle (voir plus loin)

L'environnement de test

- Les méthodes de test ont besoin d'être appelées sur des instances
- Déclaration et création des instances (par exemple h1, h2, ...)
  - en général, les instances sont déclarées comme membres d'instance de la classe de test
  - la création des instances et plus globalement la mise en place de l'environnement de test est laissé à la charge de méthodes d'initialisation

Préambules et postambules

- Méthodes écrites par le testeur pour mettre en place l'environnement de test.
- JUnit 5 : Méthodes avec annotations @BeforeEach et @AfterEach ; JUnit 4 : Méthodes avec annotations @Before et @After ; JUnit 3 : Méthodes appelées setUp et tearDown
  - exécutées avant/après chaque méthode de test (l'exécution est pilotée par le framework, et pas le testeur)
  - possibilité d'annoter plusieurs méthodes (ordre d'exécution indéterminé)
  - publiques et non statiques
- Méthodes avec annotations @BeforeAll et @AfterAll en JUnit 5 ; Méthodes avec annotations @BeforeClass et @AfterClass en JUnit 4 (pas en JUnit 3)
  - exécutées avant (resp. après) la première (resp. dernière) méthode de test
  - une seule méthode pour chaque annotation
  - publiques et statiques (sauf en JUnit 5 si le cycle de vie est perClass avec : @TestInstance(LifeCycle.PER_CLASS))

```
1   import static org.junit.jupiter.api.Assertions.fail;
    import static org.junit.jupiter.api.Assumptions.assumeTrue;
3   import org.junit.jupiter.api.AfterAll;
    ...
5
    class StandardTests {
7
        @BeforeAll
9       static void initAll() {
        }
11
        @BeforeEach
13      void init() {
        }
15
        @Test
17      void succeedingTest() {
        }
19
        @Test
21      void failingTest() {
            fail("a failing test");
23      }

25      @Test
        @Disabled("for demonstration purposes")
27      void skippedTest() {
            // not executed
29      }

31      @Test
        void abortedTest() {
33          assumeTrue("abc".contains("Z"));
            fail("test should have been aborted");
35      }

37      @AfterEach
        void tearDown() {
39      }

41      @AfterAll
        static void tearDownAll() {
43      }
    }
```

Sommaire

Sommaire

Test de méthode déclenchant des exceptions (JUnit $\geq$4)

JUnit 4

- L'annotation `@Test` peut prendre en paramètre le type d'exception attendue @Test(expected=monexception.class).
- Succès ssi cette exception est lancée.

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

JUnit 5

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

C'est quoi cette notation dans le assertThrows ?

Une lambda ... Regardons la doc ...

```
public static <T extends Throwable> T assertThrows(
    Class<T> expectedType, Executable executable)
```

- Asserts that execution of the supplied executable throws an exception of the expectedType and returns the exception.
- If no exception is thrown, or if an exception of a different type is thrown, this method will fail.
- If you do not want to perform additional checks on the exception instance, simply ignore the return value.

C'est quoi cette lambda dans le assertThrows ?

```
public static <T extends Throwable> T assertThrows(
    Class<T> expectedType, Executable executable)
```

Décortiquons déjà le premier paramètre

- type : Class<T>
- contraint par : T extends Throwable
- donc on attend une classe d'exception
- Introspection en Java : existence d'une classe java Class, paramétrée par un type T ; permet entre autre d'obtenir une instance de la classe (de type T donc) et aussi de manipuler des classes dans un programme Java ...

C'est quoi cette lambda dans le assertThrows ?

```
public static <T extends Throwable> T assertThrows(
    Class<T> expectedType, Executable executable)
```

Et donc c'est quoi le deuxième paramètre ?

- type : Executable

```
@FunctionalInterface
@API(status=STABLE,
     since="5.0")
public interface Executable{...}
```

- Executable is a functional interface that can be used to implement any generic block of code that potentially throws a Throwable.
- The Executable interface is similar to Runnable, except that an Executable can throw any kind of exception.
- Une unique méthode : void execute()

@FonctionalInterface ????

```
@Documented
 @Retention(value=RUNTIME)
 @Target(value=TYPE)
public @interface FunctionalInterface
```

An informative annotation type used to indicate that an interface type declaration is intended to be a functional interface as defined by the Java Language Specification. Conceptually, a functional interface has exactly one abstract method. Since default methods have an implementation, they are not abstract. If an interface declares an abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface's abstract method count since any implementation of the interface will have an implementation from java.lang.Object or elsewhere.
Note that instances of functional interfaces can be created with lambda expressions, method references, or constructor references.
If a type is annotated with this annotation type, compilers are required to generate an error message unless :
The type is an interface type and not an annotation type, enum, or class. The annotated type satisfies the requirements of a functional interface.
However, the compiler will treat any interface meeting the definition of a functional interface as a functional interface regardless of whether or not a FunctionalInterface annotation is present on the interface declaration.

C'est quoi cette lambda dans le assertThrows ?

```
public static <T extends Throwable> T assertThrows(
    Class<T> expectedType, Executable executable)
```

Bref ... Le deuxième paramètre attend une instance d'exécutable ou une lambda pouvant se substituer à l'unique méthode de l'interface Executable ...

Bref

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

La lambda () -> new Heure(23,05) ;) se substitue à une instance d'Executable (car sa signature matche celle de l'unique méthode d'Executable ...)
Et donc on attend une exception quand la lambda s'exécute ...

Sommaire

Test et gestion des temps d'exécution (JUnit ≥4)

JUnit 4

- L'annotation `@Test` peut prendre en paramètre un timeout : @Test (timeout=10) (en ms).
- Fail si la réponse n'arrive pas avant le timeout.

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

JUnit 5

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

Sommaire

Omission de tests à l'exécution (JUnit $\geq$4)

JUnit 4

- annotation `@Ignore` (paramètre optionnel : du texte) pour ignorer le test

```
    @Ignore
2   @Test
    public void testNonExecute() {
4       // ...
    }
```

JUnit 5

- annotation `@Disabled` (paramètre optionnel : du texte) pour désactiver le test

```
1   @Disabled
    @Test
3   public void testNonExecute() {
        // ...
5   }
```

Exécutionnée conditionnée (JUnit 5)

```
1   import org.junit.jupiter.api.Test;
    import org.junit.jupiter.api.condition.*;
3   import static org.junit.jupiter.api.condition.OS.*;
    import static org.junit.jupiter.api.condition.JRE.*;
5   class TestExécutionConditionnee {
        @Test
7       @EnabledOnOs(MAC)
        void onlyOnMacOs() {
9           // ...
        }
11
        @Test
13      @EnabledOnOs({ LINUX, MAC })
        void onLinuxOrMac() {
15          // ...
        }
17
        @Test
19      @DisabledOnOs(WINDOWS)
        void notOnWindows() {
21          // ...
        }
23

25      @Test
        @EnabledOnJre(JAVA_8)
27      void onlyOnJava8() {
            // ...
29      }

31      @Test
        @EnabledOnJre({ JAVA_9, JAVA_10 })
33      void onJava9Or10() {
            // ...
35      }

37      @Test
        @DisabledOnJre(JAVA_9)
39      void notOnJava9() {
            // ...
41      }
    }
```

Sommaire

Les assertions

- Permettent d'embarquer et d'automatiser l'oracle dans les cas de test (adieu, `println` ...)
  - attention, import statique, car les asserts sont des méthodes statiques
  - import static org.junit.Assert.*; //JUnit 4
  - import static org.junit.jupiter.api.Assertions.*; // JUnit 5
- Lancent des exceptions de type java.lang.AssertionError (comme les assert java classiques) (en fait une sous classe de AssertionError en JUnit 5)
- Différentes assertions : comparaison à un delta près, comparaison de tableaux (arrays), ...
- Forte surcharge des méthodes d'assertion.

Les assertions groupées

```
@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
void testToStringHeureValideVersionGroupee() {
    assertAll("toString correct avec heure valide",
        ()-> 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
    );
}
```

Assert that et les matchers hamcrest

- `assertThat([value], [matcher statement]);`
- exemples :
  - assertThat(x, is(3)) ;
  - assertThat(x, is(not(4))) ;
  - assertThat(responseString,
    either(containsString("color")).or(containsString("colour"))) ;
  - assertThat(myList, hasItem("3")) ;
- not(s), either(s).or(ss), each(s)
- Messages d'erreur plus clairs
- En JUnit 4 :
  `http://junit.sourceforge.net/doc/ReleaseNotes4.4.html` +
  `https://junit.org/junit4/javadoc/latest/org/hamcrest/Matcher.html`
- En JUnit 5 : `http://hamcrest.org/JavaHamcrest/`

Suppositions conditionnant la suite du test

JUnit 4

- `assumeThat(File.separatorChar, is("/"))`
- L'assertion suivante sera ignorée si la supposition n'est pas vérifiée

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

JUnit 5

- `assumeTrue` et `assumingThat`

```
1   @Test void testOnlyOnDeveloperWorkstation() {
        assumeTrue("DEV".equals(System.getenv("ENV")),
3               () -> "Aborting test: not on developer workstation");
            // remainder of test
5       }
    @Test void testInAllEnvironments() {
7        assumingThat("CI".equals(System.getenv("ENV")),
                () -> {
9                   // perform these assertions only on the continuous integration server
                    assertEquals(2, 2);
11              });
            // perform these assertions in all environments
13          assertEquals("a string", "a string");}
```

Sommaire

Test paramétré

- Objectif : réutiliser des méthodes de test avec des jeux de données de test différents
- Jeux de données de test
  - retournés par une méthode annotée @Parameters
  - cette méthode retourne une collection de tableaux contenant les données et éventuellement le résultat attendu
- La classe de test
  - annotée @RunWith(Parameterized.class)
  - contient des méthodes devant être exécutées avec chacun des jeux de données
- Pour chaque donnée, la classe est instanciée, les méthodes de test sont exécutées

Test paramétré en JUnit 4

- Un constructeur public qui utilise les paramètres (i.e. un jeu de données quelconque)
- La méthode qui retourne les paramètres (i.e. les jeux de données) doit être statique

Exemple de test paramétré en JUnit 4

```
1   import org.junit.Test;
    import org.junit.runner.RunWith;
3   import org.junit.runners.Parameterized;
    import org.junit.runners.Parameterized.Parameters;
5   import static org.junit.Assert.*;
    import java.util.*;
7   @RunWith(Parameterized.class)
    public class TestParametre {
9       private Heure h1;
        private Heure h2;
11      private boolean h1Avanth2;
        public TestParametre(int hh1, int mn1, int hh2, int mn2, boolean h1AvantH2) throws HoraireIncorrectEx
13          h1=new Heure(hh1, mn1);
            h2=new Heure(hh2, mn2);
15          this.h1Avanth2=h1AvantH2;}
        @Parameters
17      public static Collection testData() {
            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}});}
        @Test public void testEstAVant() {
21          assertEquals(h1Avanth2,h1.estAvant(h2));}
        @Test public void creationCreneauValide() throws CreneauIncorrectException {
23          Creneau c;
            if (h1.estAvant(h2)) {
25              c=new Creneau(JourSemaine.LUNDI, h1, h2);
            } else {
27              c=new Creneau(JourSemaine.LUNDI, h2, h1);
            }}
29      @Test(expected=CreneauIncorrectException.class)
        public void testCreneauInvalide() throws CreneauIncorrectException{
31          Creneau c;
            if (h1.estAvant(h2)) {
33              c=new Creneau(JourSemaine.LUNDI, h2, h1);
            } else {
35              c=new Creneau(JourSemaine.LUNDI, h1, h2);
            }}
37  }
```

Exemple de test paramétré en JUnit 5

```
1   import static org.junit.jupiter.api.Assertions.*;
    import java.util.stream.Stream; import org.junit.jupiter.api.DisplayName; import org.junit.jupiter.params
3   import org.junit.jupiter.params.provider.Arguments;
    import org.junit.jupiter.params.provider.MethodSource;
5   import org.junit.jupiter.params.provider.ValueSource;
    class TestParametreJunit5 {
7       private static  int nbAdherent=0;

9        @DisplayName("création d'heures")
           @ParameterizedTest(name = "{index} : heure={0}, minutes={1}, correct={2}")
11         @MethodSource("HeureProvider")
           void creationHeure(int h, int mn, boolean correct) {
13             if (!correct) {
                   assertThrows(HoraireIncorrectException.class, ()-> {
15                 new Heure(h, mn);
                   });
17             }}

19         private static Stream<Arguments> HeureProvider() {
               return Stream.of(
21                     Arguments.of(10, 12, false),
                       Arguments.of(2, 30, false),
23                     Arguments.of(23,10, false),
                       Arguments.of(12, 30, true)
25             );
           }
27
       @ParameterizedTest
29     @ValueSource(strings = { "nom1", "nom2", "nom3" })
       @NullSource
31     void testAdherents(String name) {
           Adherent a=new Adherent(name);
33         nbAdherent++;
           assertEquals(nbAdherent, a.getNumero());
35     }
   }
37 ...
   public class Adherent {...
39     public Adherent(String nom){...}
   }
41 ...
```

Remarques

- Il y a d'autres sortes de sources de données
- Par exemple au format csv / fichier csv
- Un peu ardu à prendre en main mais si pratique !

Sommaire

Suite de tests

- Rassemble des cas de test pour enchaîner leur exécution
- i.e. groupe l'exécution de classes de test

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

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

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

9   }
```

Sommaire

Conclusion sur JUnit

- Construction rapide de tests
- Exécution rapide
- Très bien adapté pour le test unitaire et test de non régression

JUnit et les autres

- NUnit -> .net
- PiUnit -> python
- JSUnit -> JS
- etc ...

Le test unitaire

Principe

- Tester une unité logicielle en isolation
- Par exemple une classe ou un groupe de classes

Isolation ?

- Que faire en cas de dépendances mutuelles d'un grand nombre de classes ?
- Que faire en cas d'accès à des composants extérieurs de type : FS, DB ?

Simulation

- Pour parvenir à l'isolation d'une unité logicielle, on a souvent recours à la simulation de l'environnement
- Outils de simulation pour les test unitaire : les mocks (mockito, easymock, ...)

Le test unitaire

Ecrire des tests unitaires

- Qui ? des développeurs (mais pas nécesssairement ceux qui ont développé le SUT)
- Quand ? le plus tôt possible, éventuellement avant d'écrire le SUT ! (TDD, Test Driven development)

Exécuter des tests unitaires

- Exécution "initiale" : s'assurer de la qualité d'une unité logicielle
- Non régression : après chaque modification de l'unité logicielle, on relance les tests unitaires
- Exécution "continue" : placement des tests sur une plateforme CI

Écrire des tests unitaires

- Utiliser le pattern Given-When-Then
- Nommer soigneusement les méthodes de test

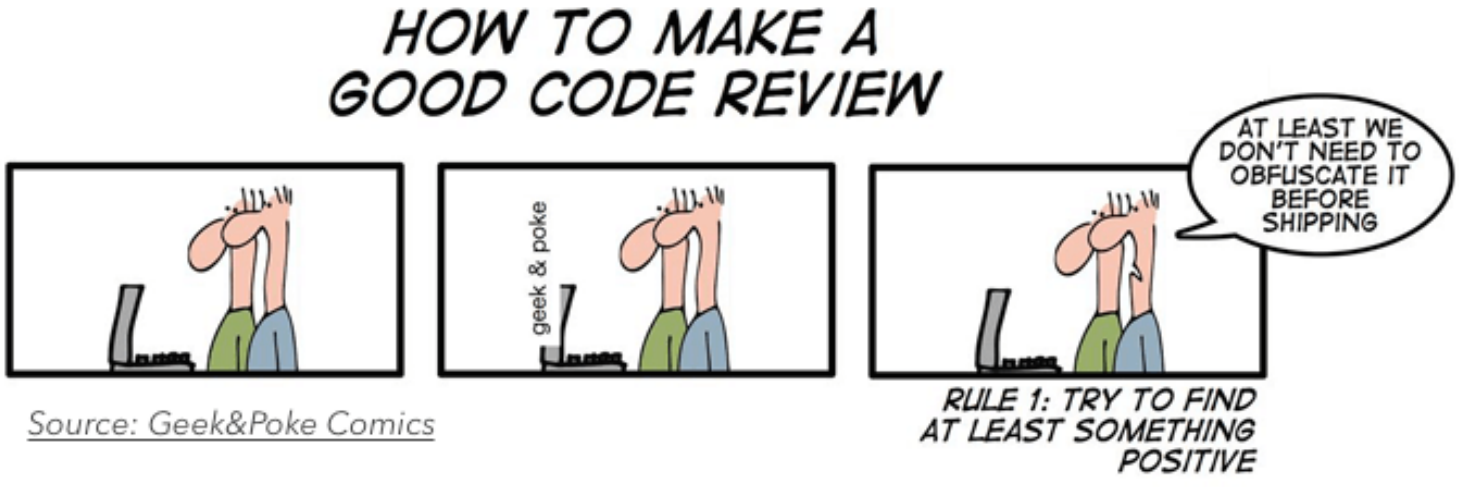
Le test unitaire est-il suffisant ?

Après le test unitaire, les autres tests

- test d'intégration
- test système
- test de recette

Le test unitaire est-il suffisant ?

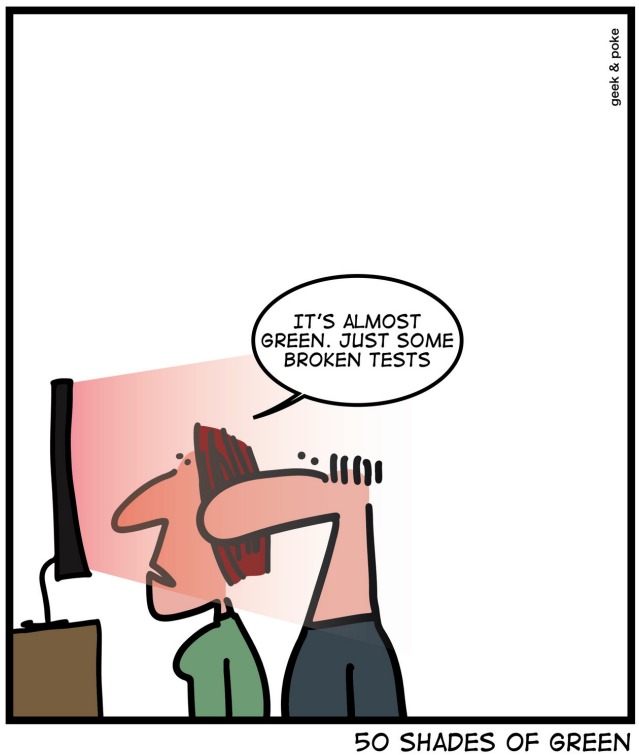
Il n'y a pas que le test pour s'assurer de la qualité d'un logiciel

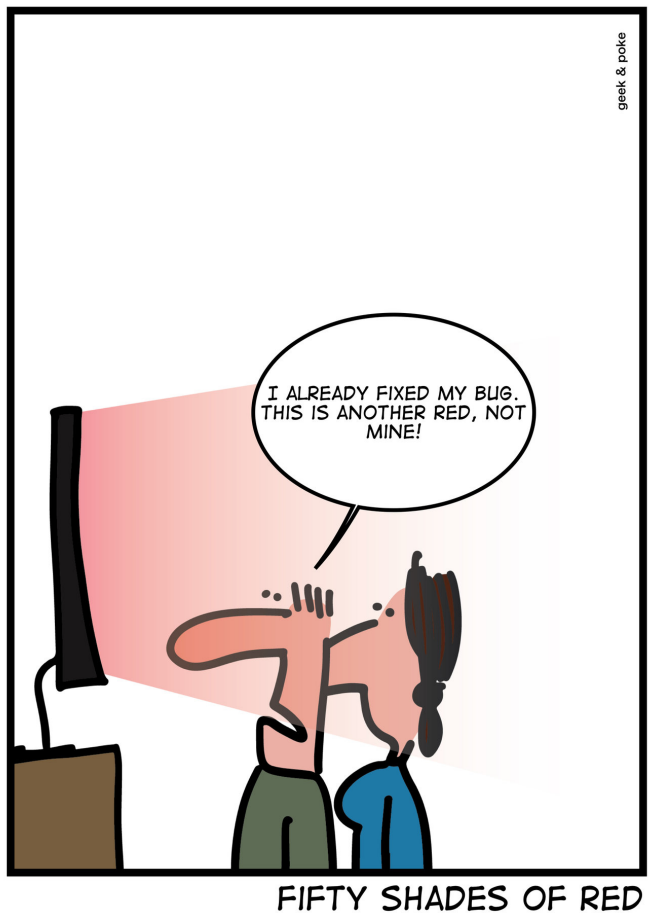


Le bon testeur ...

- pose des questions. Que se passe-t-il si ? Pourquoi ça marche comme ça ?
- est curieux et créatif. Ne s'arrête pas à ce qu'il voit, et cherche des problèmes, sous différents angles.
- communique adroitement. Car il pourvoit en général les mauvaises nouvelles.Car il doit documenter les tests et les rapports de test.
- est patient. Car il doit rester concentré sur sa chasse au bug.
- a le sens des priorités. Car on n'a jamais assez de temps pour "bien tout tester comme il faudrait".
- doit savoir se mettre à la place de l'utilisateur final.
- a des connaissances techniques. Car il faut comprendre ce que l'on teste. Et aussi comprendre les formidables outils de test !
- fait attention aux détails.

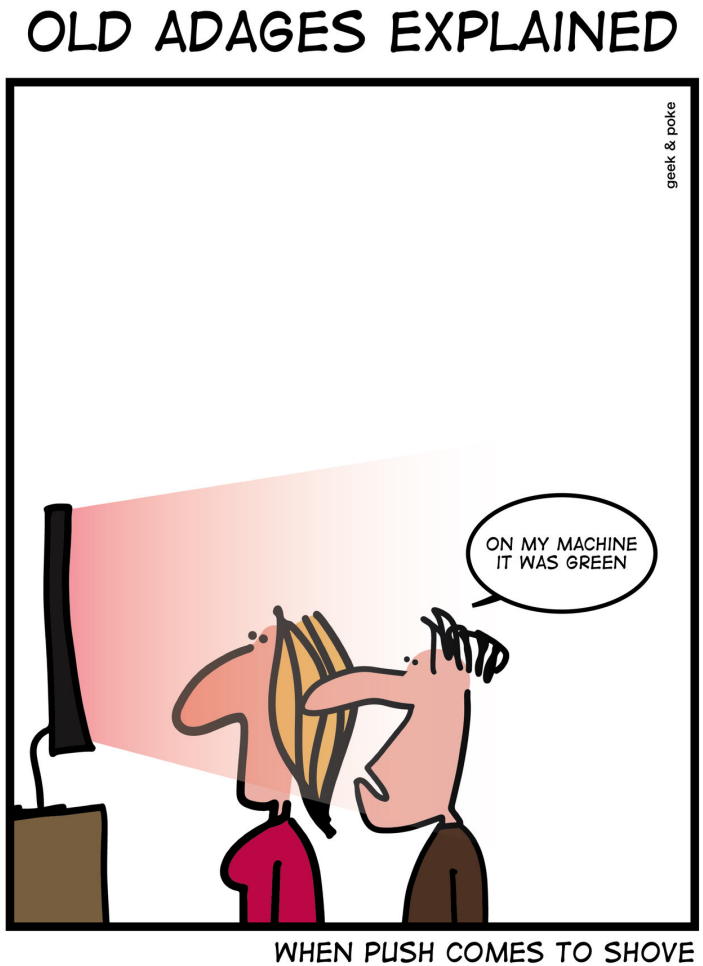
Le test : ingrat mais nécessaire





Idea from Jens Wolfgramm

Bon courage !


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