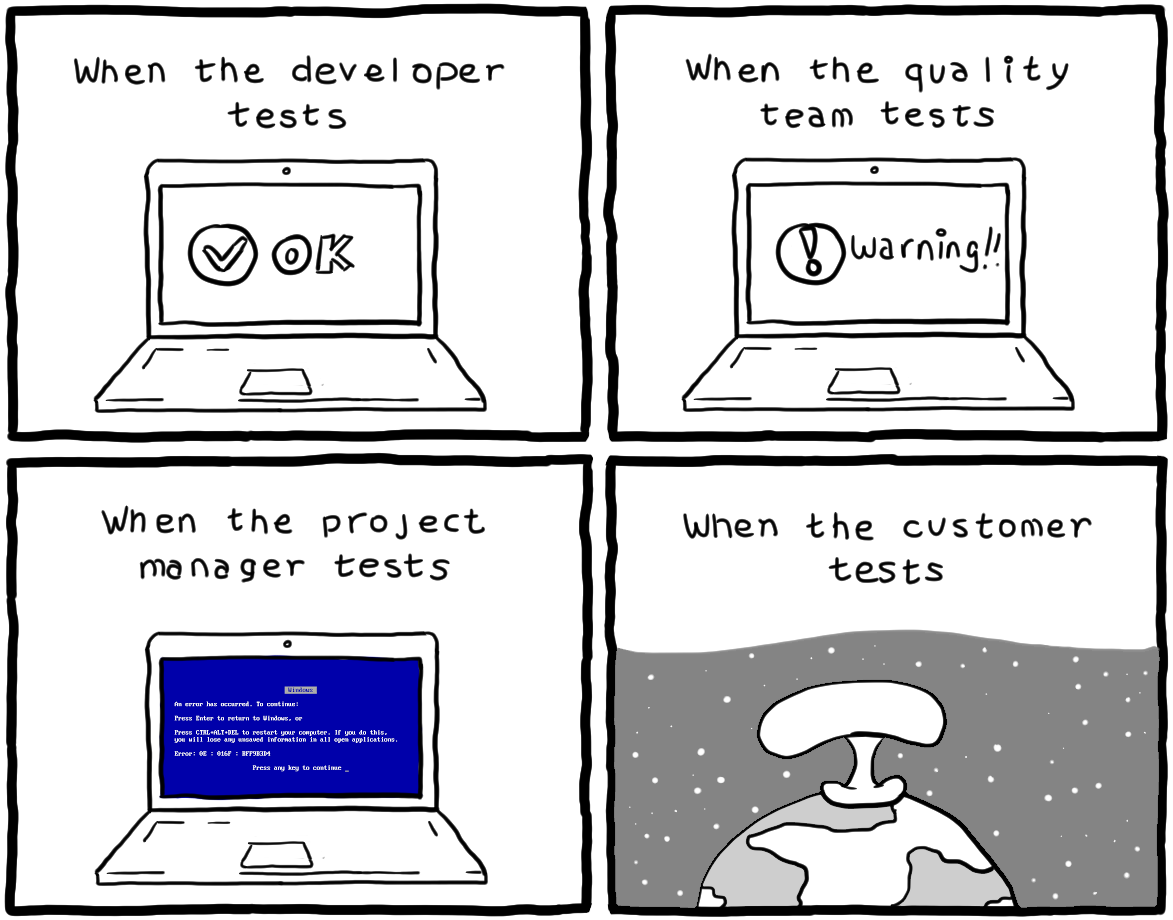
Test Logiciel



27 Septembre 2022

Test Logiciel

Menu Du Jour

(Très) Brève Introduction Au Test

Test Unitaire Avec JUnit

Le Test Logiciel : Essayer Pour Voir Si Ça Marche

- Essayer $\rightarrow$ exécution(s) du système sous test (SUT)
- Pour voir $\rightarrow$ il faut des indicateurs visibles
- Si ça marche $\rightarrow$ il faut une spécification de l'attendu

Le Test Logiciel : Définition

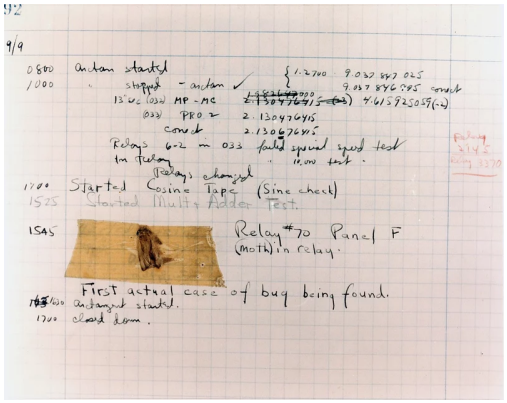
Définition de Myers, 1979

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

- Test
  - $\in$ Procédés de Vérification et Validation (V&V)
  - procédé le plus connu
  - procédé réputé peu coûteux
    - (pas en termes écologiques)
  - procédé simple à mettre en place

- Trouver les bugs !

Bug ?



- Terme attribué à Grace Hopper (1906-1992 ; 1959 : Cobol)
- Mais mot déjà utilisé par Thomas Edison (1847-1931, lampe à incandescence, phonographe)

  > Et c’est alors que les "bugs" — comme on appelle ces petits défauts et problèmes — se manifestent.

- En français on devrait dire bogue

Bugs Et Conséquences : Quelques Bugs Historiques

Les bugs se paient très chers

- THERAC 25 : bug dans l'IHM d'un appareil de traitement anti-cancéreux, surdose de traitement $\Longrightarrow$ des morts
- AT&T : bug dû à un mauvais placement de parenthèse, crash de l'ensemble du téléphone longue distance américain pendant une journée $\Longrightarrow$ 50 millions d'appels bloqués
  https://users.csc.calpoly.edu/~jdalbey/SWE/Papers/att_collapse
- PATRIOT : la chute d'un missile Patriot sur une caserne américaine à Dahran est due à un bug ; erreur d'arrondis mal traités
  https://www.iro.umontreal.ca/~mignotte/IFT2425/Disasters.html $\Longrightarrow$ eviron 50 morts
- Le fameux crash Ariane 5 (1995) : débordement (entier trop grand, les données d'Ariane4 étaient plus petites) $\Longrightarrow$ de 500 millions à 7 milliards de dollars selon comment on compte

Débugguer

- Maurice Wilkes (1913-2010, prix Turing 1967)

> "As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs."
> — Maurice Wilkes discovers debugging, 1949

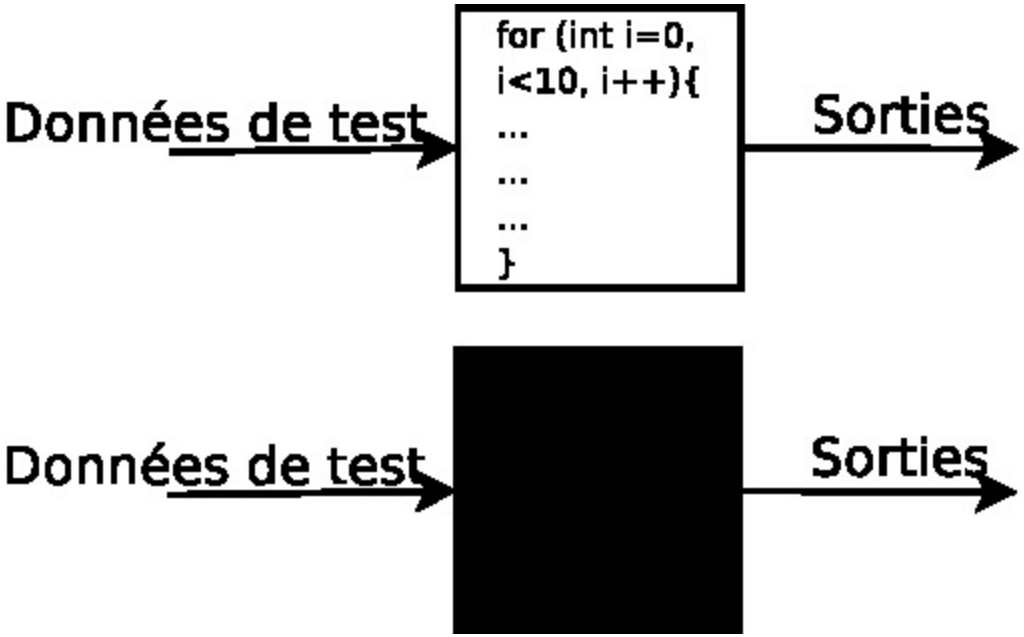
Erreurs ? Quelles Erreurs ??

- Les erreurs sont donc partout
- Donc elles sont dans votre code !
- Et il vous faut partir à la chasse au bug

Que Tester ?

- Différentes propriétés
  - Fonctionnalité
  - Utilisabilité
  - Efficacité
  - Robustesse
  - Sûreté de fonctionnement
  - ...

Boite noire / boite blanche



Plusieurs Sortes De Test En Fonction Des Propriétés À Tester

- Tests de (non) régression (après modifications)
  - ce qui fonctionnait avant la modif fonctionne encore
  - avec des modifs (corrections, évolutions, intégration de fonctionalités)
- Tests de robustesse
  - comportement du SUT en réaction à des entrées non valides
- Tests de performance (le SUT tourne dans son environnement)
  - load testing : résistance à la montée en charge
  - stress testing : résistance aux demandes de ressources anormales

Plusieurs Échelles De Tests

- Tests unitaires : test des composants simples du SUT
  - en général écrits par les développeurs
  - exemple : je teste que ma fonction de somme retourne bien 4 quand je lui envoie l'entrée 2, 2
- Test d'intégration : test de l'assemblage des composants, de leur interaction
  - en général écrits par les développeurs
  - exemple : je teste que quand je sollicite la somme de deux entiers depuis un client web, le serveur web me renvoie un résultat avec un code http correct

Test D'intégration

2 unitary tests, 0 integration tests

https://twitter.com/thepracticaldev/status/687672086152753152

- L'intégration de 2 composants sans défauts apparents peut engendrer des dysfonctionnements.
- Il est en général impossible de réaliser les tests unitaires dans l'environnement cible avec tous modules à disposition.

Plusieurs Échelles De Tests (Suite)

- Les tests système : test du SUT dans son entier, depuis ses interfaces
  - souvent aussi appelés tests fonctionnels, bien qu'ils ne soient pas forcément fonctionnels ...
  - souvent réalisés par des ingénieurs de test
  - exemple : quand je clique sur le bouton submitte, le résultat de la somme de mes 2 entiers s'affiche correctement.
- Les tests d’acceptation : tests finaux
  - réalisés dans l'environnement du client, par le client / les utilisateurs finaux

Test Système

- Création de scénarios de tests centrés sur les fonctionnalités utilisateurs
- Peuvent être facilités par la présence d'US (user stories)
- Les scénarios sont ensuite "joués" sur l'application finale
- D’une manière générale (pas que pour le test système), la préparation de scénarios de tests permet de lever des ambiguïtés, de trouver des oublis, et peuvent participer à la spcéfication du logiciel
- Peuvent donc être écrits avant la création du logiciel ...

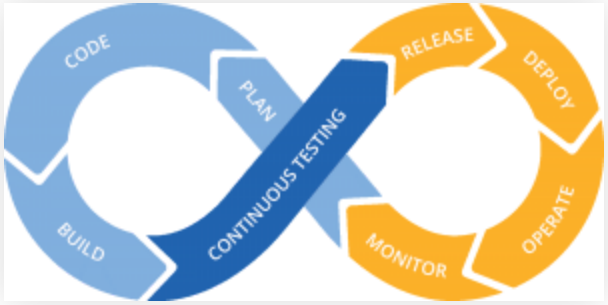
Test : De L'objectif De Test À Son Éxécution

- Objectif de test : qu'est ce que je veux tester
  - on ne teste pas au hazard
- Critère de test
  - quand arrêter de créer de nouveaux tests ?
- Donnée de test
  - Quelle(s) donnée(s) sont à fournir au logiciel pour le test
- Oracle
  - Comment savoir si mon test a détecté une erreur ?
- Implémentation du test
  - si test automatisé, avec des outils adéquats
- Exécution du test
  - verdict du test : PASS, FAIL, INCONCLUSIVE

Test Et Couverture De Code

- Couverture de code par les tests : proportion du code du SUT exécuté lors de l'exécution de ses tests
  - Des variantes
    - couverture des méthodes
    - couverture des instructions
    - couverture des chemins d'exécution

Intégration Continue Du Test



- Automatisation, test plus agile
  - frameworks de test
  - outils de d'exécutions de séries de test
  - xunit
  - cucumber, selenium, cypress
  - outils de mocks
  - approche TDD

Pour Finir

- Test
  - long
  - fastidieux
  - pas vraiment valorisant
- Mais nécessaire

Un Peu De Culture Ne Saurait Nuire

- Cours et exposés de Gérard Berry
  - Notamment "La chasse aux bugs"

Test Logiciel

Test Unitaire Avec JUnit

Le Test Unitaire : Quelques Questions ...

Principe

- Tester une unité logicielle en isolation

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

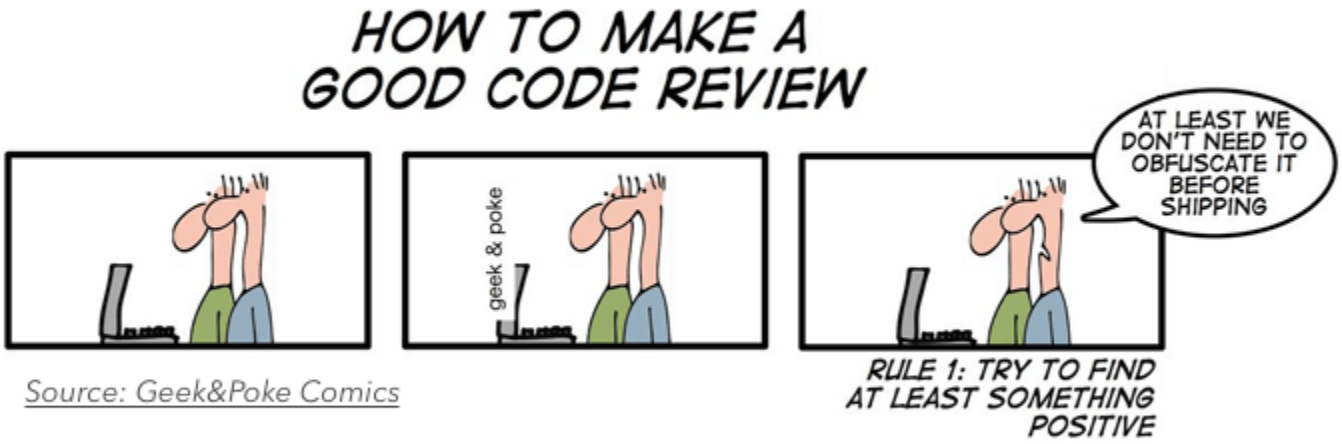
- Exécution "initiale" : s'assurer de la qualité d'une unité logicielle
- Non régression : après chaque modification de l'unité logicielle, on relance les tests unitaires
- Exécution "continue" : placement des tests sur une plateforme CI

Écrire Des Tests Unitaires

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

Le Test Unitaire Est-Il Suffisant ?

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



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

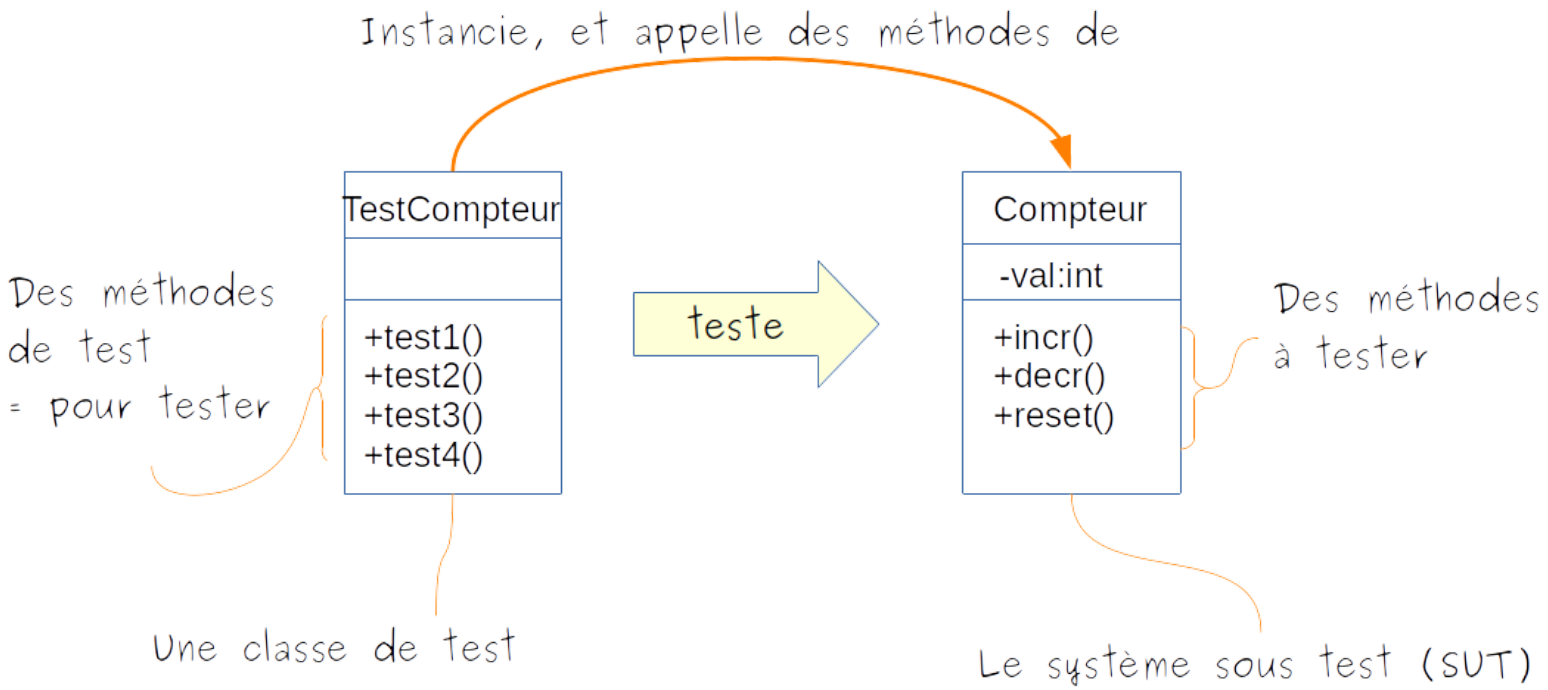
Versions 5 (Jupiter)

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

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.

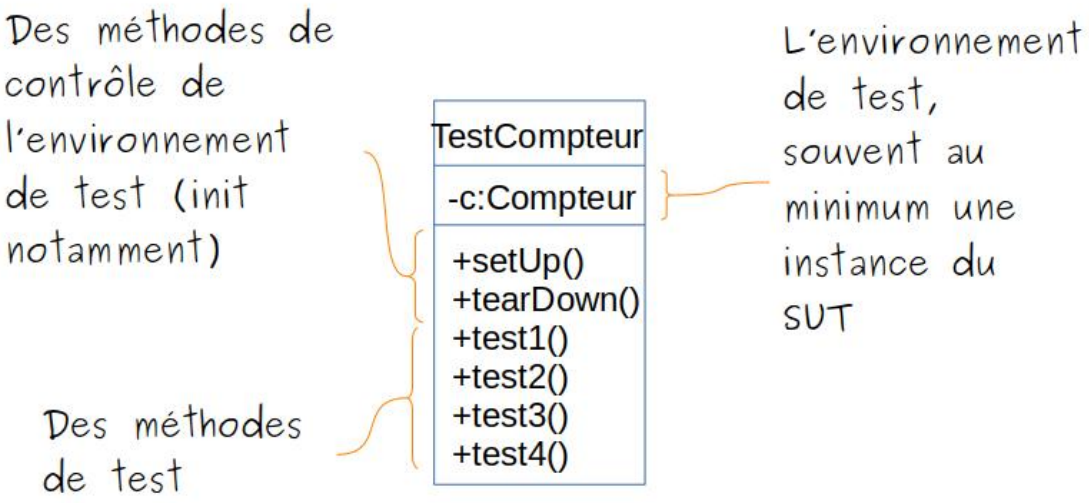
Écriture De Test : Principe Général



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 $\geq$ 4 : une classe quelconque
  - Jupiter: une classe quelconque

Classe De Test



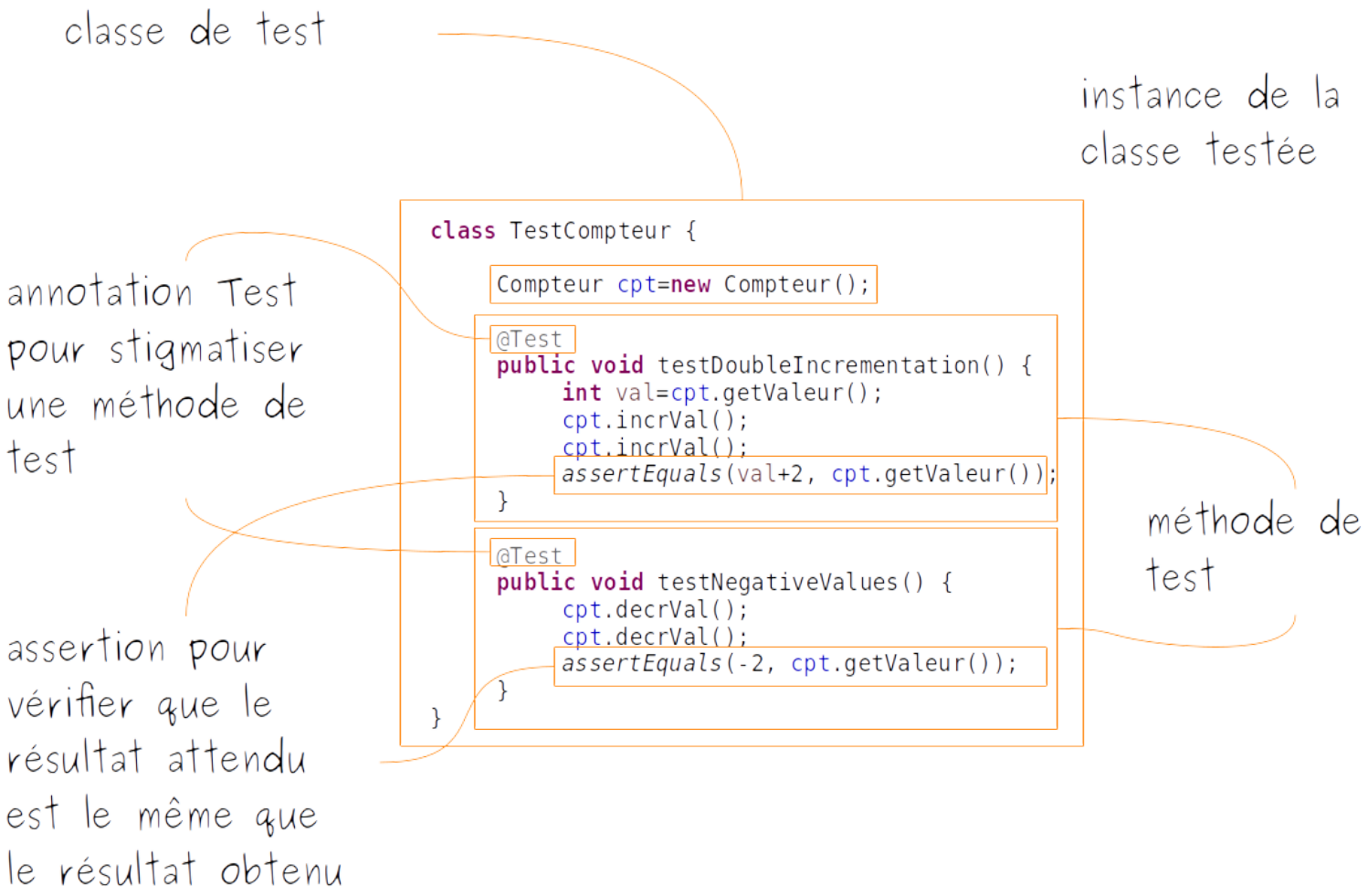
Méthode De Test

- 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 $\geq$ 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

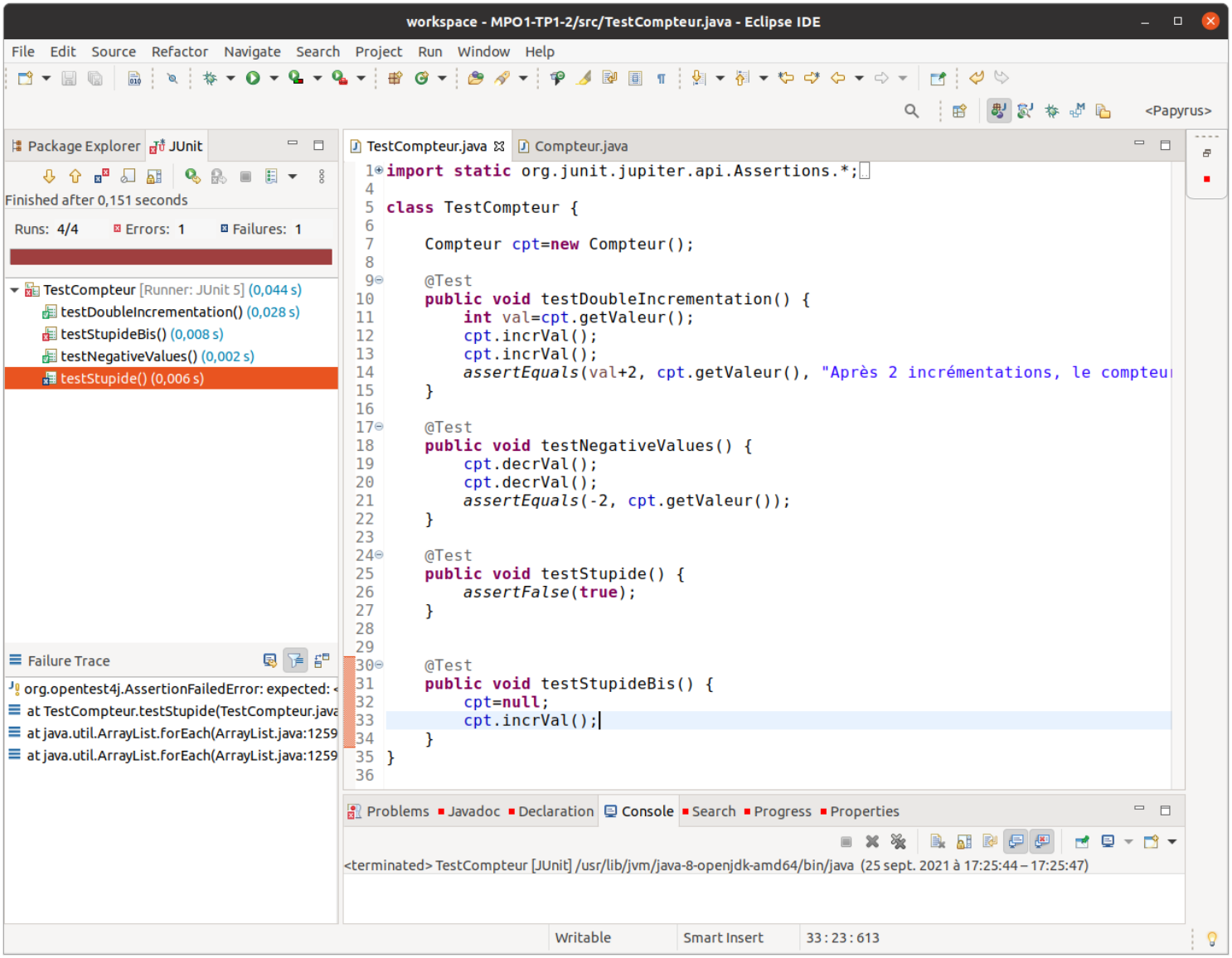


Les Verdicts

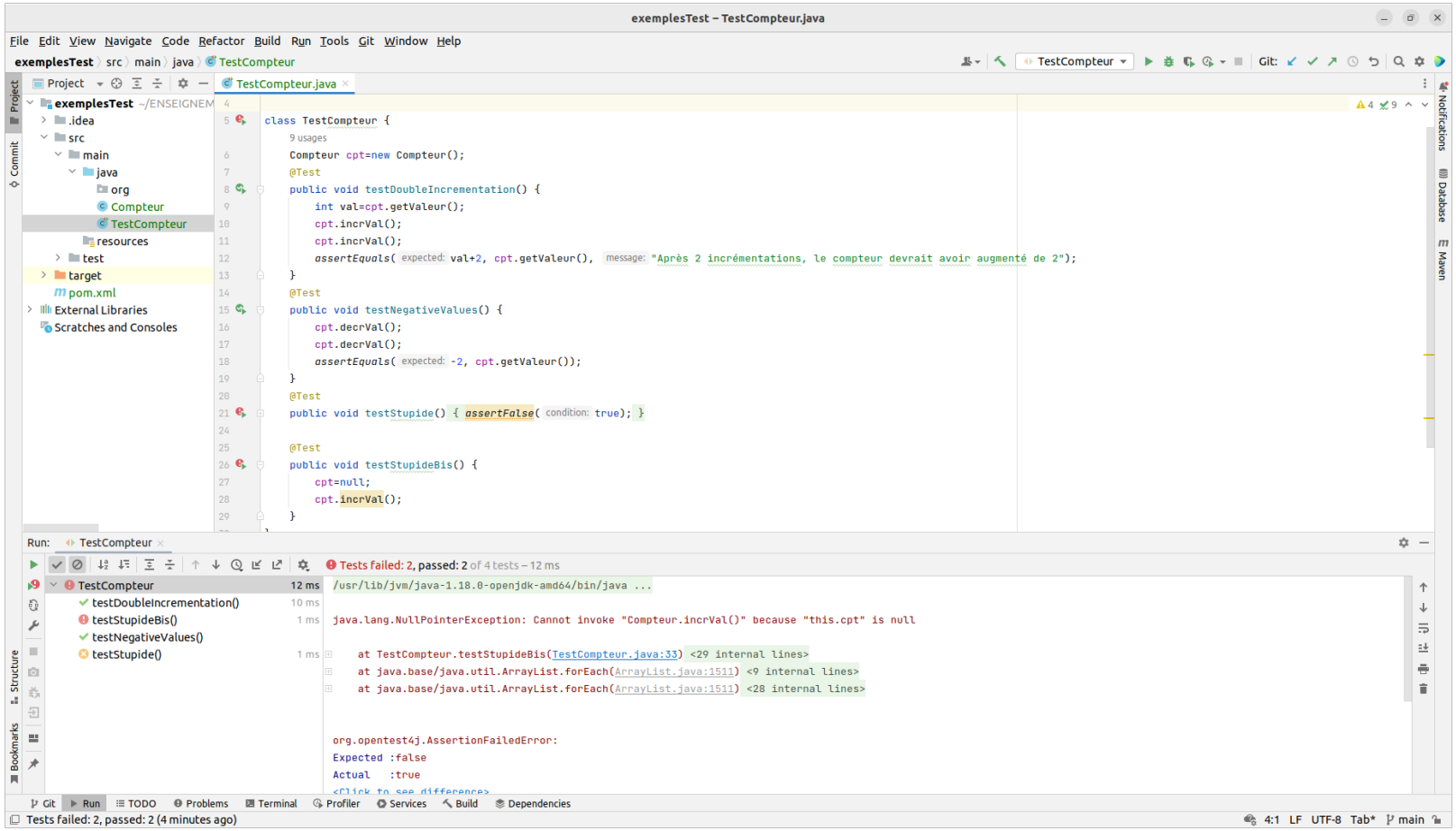
Sont définis grâce aux assertions placées dans les cas de test.

- 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



Exécution Et Verdicts



Exemple -- Classe À Tester

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

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

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

Objectif De Test : ToString Correct Pour Des Heures Correctes

```java
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.

Obj : Créat. Heures Incorr. $\Longrightarrow$ HoraireIncorrectException

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class TestHeures {
    @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);
        });
    }
}
```

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

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

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

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 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));
    }
```

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 Fixtures

- 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

Préambules Et Postambules

- 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))

```
import static org.junit.jupiter.api.Assertions.fail;
import static org.junit.jupiter.api.Assumptions.assumeTrue;
import org.junit.jupiter.api.AfterAll;

class StandardTests {
    @BeforeAll
    static void initAll() {
    }
    @BeforeEach
    void init() {
    }
    @Test
    void succeedingTest() {
    }
    @Test
    void failingTest() {
        fail("a failing test");
    }
    @Test
    @Disabled("for demonstration purposes")
    void skippedTest() {
        // not executed
    }
    @Test
    void abortedTest() {
        assumeTrue("abc".contains("Z"));
        fail("test should have been aborted");
    }
    @AfterEach
    void tearDown() {
    }
    @AfterAll
    static void tearDownAll() {
    }
}
```

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.

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 ...

```
@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 ...

Test Et Gestion Des Temps D'exécution (Ici En JUnit 5)

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

Omission De Tests À L'exécution (Ici En JUnit 5)

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

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

Exécution Conditionnée (JUnit 5)

```java
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 TestExécutionConditionnee {
    @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() {
        // ...
    }
}
```

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

```java
@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 (Ici En JUnit 5)

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

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

Exemple De Test Paramétré En JUnit 5

```
class TestParametreJunit5 {
    private static  int nbAdherent=0;

        @DisplayName("création d'heures")
        @ParameterizedTest(name = "{index} : heure={0}, minutes={1}, correct={2}")
        @MethodSource("HeureProvider")
        void creationHeure(int h, int mn, boolean correct) {
            if (!correct) {
                assertThrows(HoraireIncorrectException.class, ()-> {
                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" })
    @NullSource
    void testAdherents(String name) {
        Adherent a=new Adherent(name);
        nbAdherent++;
        assertEquals(nbAdherent, a.getNumero());
    }
}
...
public class Adherent {...
    public Adherent(String nom){...}
}
...
```

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 !

Conclusion Sur JUnit

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

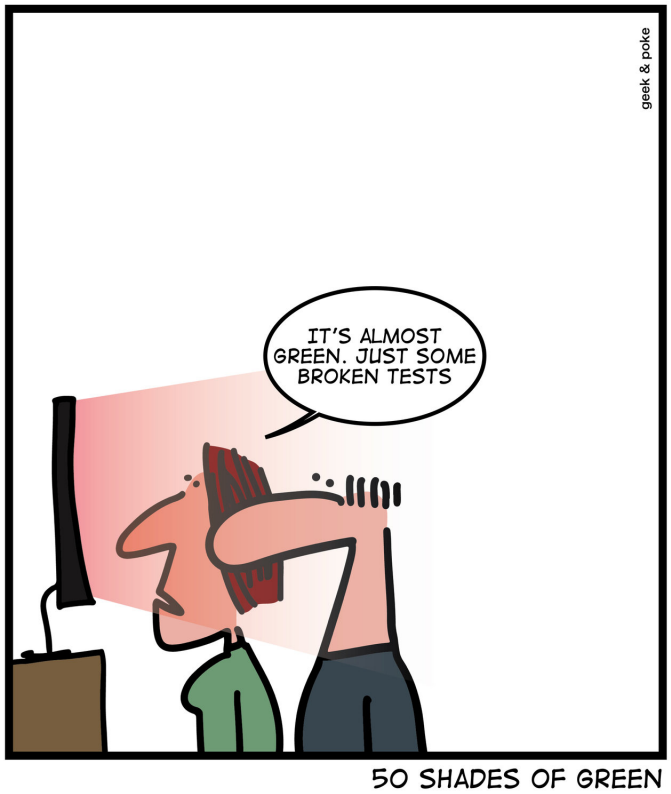
JUnit Et Les Autres

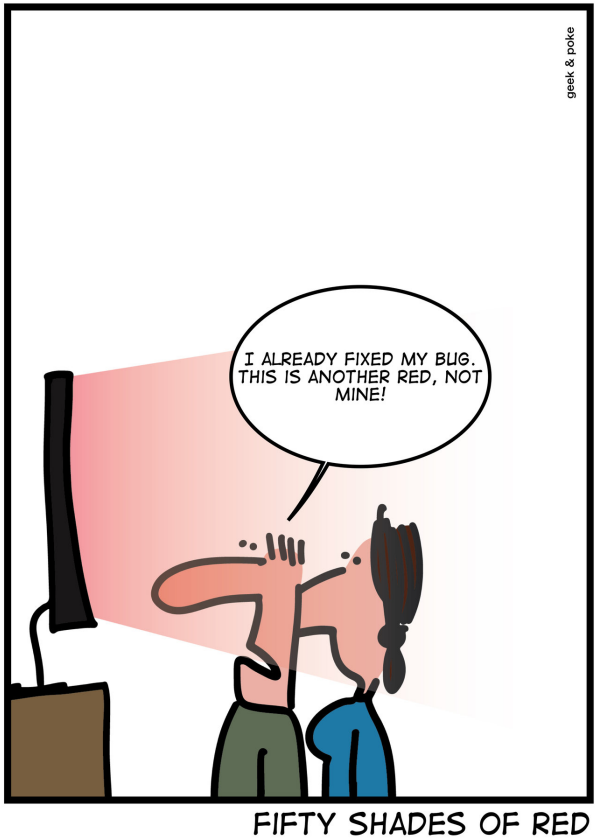
- NUnit -> .net
- PiUnit -> python
- JSUnit -> JS
- etc ...

Le 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.

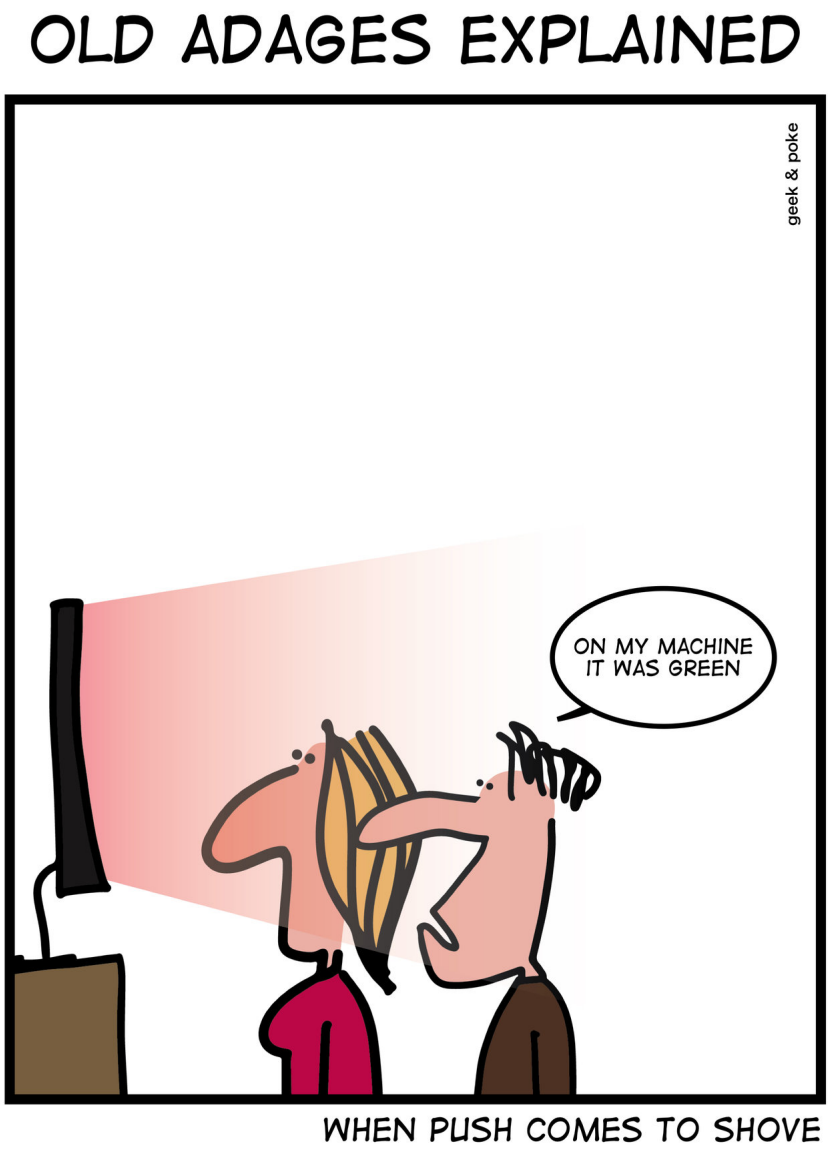
Le Test : Ingrat Mais Nécessaire





Idea from Jens Wolfgramm

Bon Courage !


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