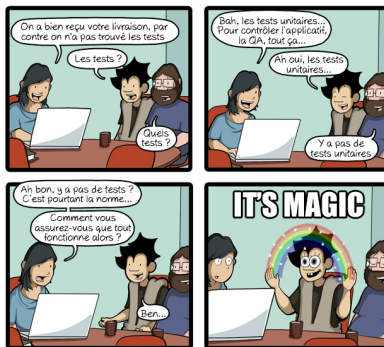


## JUnit, un framework de test unitaire pour Java



CommitStrip.com

Faculté des sciences – Université de Montpellier

22 octobre 2018

## 1 Introduction

## 2 Premiers pas

## 3 Approfondissements

- Test et exceptions
- Test et temps d'exécution
- Omission d'exécution et exécution conditionnée
- Les assertions
- Le test paramétré en JUnit 4
- Les suites de test

## 4 Conclusion

## 5 Le test unitaire et ses limites

# Sommaire

## 1 Introduction

## 2 Premiers pas

## 3 Approfondissements

- Test et exceptions
- Test et temps d'exécution
- Omission d'exécution et exécution conditionnée
- Les assertions
- Le test paramétré en JUnit 4
- Les suites de test

## 4 Conclusion

## 5 Le test unitaire et ses limites

# 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](http://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 fonctionnalités 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 !
- Jupiter pas encore installé dans les salles FDS, donc pas utilisé cette année (restons vintage)



# Sommaire

## 1 Introduction

## 2 Premiers pas

## 3 Approfondissements

- Test et exceptions
- Test et temps d'exécution
- Omission d'exécution et exécution conditionnée
- Les assertions
- Le test paramétré en JUnit 4
- Les suites de test

## 4 Conclusion

## 5 Le test unitaire et ses limites

## 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.

# Classe de test

- Contient les méthodes de test
- Est une collection de cas de test (**sans ordre**)
- peut contenir des méthodes particulières pour positionner l'environnement de test
- En JUnit :
  - Junit versions  $<4$  : la classe de test hérite de `JUnit.framework.TestCase`
  - JUnit versions  $\geq 4$  : une classe quelconque
  - Jupiter : on peut plus ou moins contrôler l'ordre (mais c'est mal)

## Cas de test / méthode de test

- s'intéresse à une seule unité de code/ un seul comportement
- doit rester court
- les cas de test sont indépendants les uns des autres
- Avec Junit, un cas de test  $\equiv$  une méthode (méthode de test)
  - 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)
- 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)

# Les verdicts

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

- Pass (vert) : pas de faute détectée
- Fail (rouge) : échec, on attendait un résultat, on en a eu un autre
- Error : le test n'a pas pu s'exécuter correctement (exception inattendue, ...)
- En JUnit 4, plus de différence entre fail et error

## 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;
    }
}
```

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

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class TestHeures {
    Heure h1, h2, h3, h4, h5, h6;
    @Before
    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());
    }
}
```



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

```
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

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

    @Test(expected=HoraireIncorrectException.class)
    public void testCreationHeureInvalideAvantHeureMin()
        throws HoraireIncorrectException {
        new Heure(6,10);
    }

    @Test(expected=HoraireIncorrectException.class)
    public void testCreationHeureInvalideMauvaiseGranularite()
        throws HoraireIncorrectException {
        new Heure(7,12);
    }
}
```

## Exemple – objectif de test : test de la méthode estAvant

```
public boolean estAvant(Heure autreHeure) {  
    ...  
}  
  
public class TestHeures {  
    Heure h1, h2, h3, h4, h5, h6;  
    @Before  
    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));  
        assertTrue(h1.estAvant(h2));  
        assertTrue(h1.estAvant(h4));  
        assertTrue(h1.estAvant(h5));  
        assertTrue(h2.estAvant(h4));  
        assertTrue(h3.estAvant(h6));  
    }  
}
```

# 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`)

# Sommaire

## 1 Introduction

## 2 Premiers pas

## 3 Approfondissements

- Test et exceptions
- Test et temps d'exécution
- Omission d'exécution et exécution conditionnée
- Les assertions
- Le test paramétré en JUnit 4
- Les suites de test

## 4 Conclusion

## 5 Le test unitaire et ses limites

# Sommaire

## 1 Introduction

## 2 Premiers pas

## 3 Approfondissements

- Test et exceptions

- Test et temps d'exécution

- Omission d'exécution et exécution conditionnée

- Les assertions

- Le test paramétré en JUnit 4

- Les suites de test

## 4 Conclusion

## 5 Le test unitaire et ses limites

# Test de méthode déclenchant des exceptions (JUnit ≥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);
    });
}
```

# Sommaire

## 1 Introduction

## 2 Premiers pas

## 3 Approfondissements

### ■ Test et exceptions

### ■ **Test et temps d'exécution**

### ■ Omission d'exécution et exécution conditionnée

### ■ Les assertions

### ■ Le test paramétré en JUnit 4

### ■ Les suites de test

## 4 Conclusion

## 5 Le test unitaire et ses limites



## Test et gestion des temps d'exécution (JUnit $\geq 4$ )

### 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

## 1 Introduction

## 2 Premiers pas

## 3 Approfondissements

- Test et exceptions
- Test et temps d'exécution
- **Omission d'exécution et exécution conditionnée**
- Les assertions
- Le test paramétré en JUnit 4
- Les suites de test

## 4 Conclusion

## 5 Le test unitaire et ses limites

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

## JUnit 4

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

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

## JUnit 5

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

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

## Exécution conditionnée (JUnit 5)

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

class TestExecutionConditionnee {

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

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

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

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

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

# Sommaire

## 1 Introduction

## 2 Premiers pas

## 3 Approfondissements

- Test et exceptions
- Test et temps d'exécution
- Omission d'exécution et exécution conditionnée
- **Les assertions**
- Le test paramétré en JUnit 4
- Les suites de test

## 4 Conclusion

## 5 Le test unitaire et ses limites

# Les assertions

- Permettent d'embarquer et d'automatiser l'oracle dans les cas de test (adieu, `println` ...)
- Utilisation de `org.junit.Assert.*`
  - attention, import statique, car les asserts sont des méthodes statiques
  - `import static org.junit.Assert.*;`
- Lancer 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.

## 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

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

### 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");  
}
```



# Sommaire

## 1 Introduction

## 2 Premiers pas

## 3 Approfondissements

- Test et exceptions
- Test et temps d'exécution
- Omission d'exécution et exécution conditionnée
- Les assertions
- **Le test paramétré en JUnit 4**
- Les suites de test

## 4 Conclusion

## 5 Le test unitaire et ses limites

## 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

```

import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import static org.junit.Assert.*;
import java.util.*;
@RunWith(Parameterized.class)
public class TestParametre {
    private Heure h1;
    private Heure h2;
    private boolean h1AvantH2;
    public TestParametre(int hh1, int mn1, int hh2, int mn2, boolean h1AvantH2) throws HoraireIncorrectException {
        h1=new Heure(hh1, mn1);
        h2=new Heure(hh2, mn2);
        this.h1AvantH2=h1AvantH2;}

    @Parameters
    public static Collection testData() {
        return Arrays.asList(new Object[][] {
            { 7, 0, 7, 5, true }, {7,0, 12, 5, true }, { 12,30, 7, 5, false }, {12, 00, 20,15, true }
        });
    }

    @Test public void testEstAVant() {
        assertEquals(h1AvantH2, h1.estAvant(h2));}

    @Test public void creationCreneauValide() throws CreneauIncorrectException {
        Creneau c;
        if (h1.estAvant(h2)) {
            c=new Creneau(JourSemaine.LUNDI, h1, h2);
        } else {
            c=new Creneau(JourSemaine.LUNDI, h2, h1);
        }
    }

    @Test(expected=CreneauIncorrectException.class)
    public void testCreneauInvalide() throws CreneauIncorrectException {
        Creneau c;
        if (h1.estAvant(h2)) {
            c=new Creneau(JourSemaine.LUNDI, h2, h1);
        } else {
            c=new Creneau(JourSemaine.LUNDI, h1, h2);
        }
    }
}

```

## Exemple de test paramétré en JUnit 5

```
import static org.junit.jupiter.api.Assertions.*;
import java.util.stream.Stream;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.Arguments;
import org.junit.jupiter.params.provider.MethodSource;
import org.junit.jupiter.params.provider.ValueSource;

class TestParametreJUnit5 {
    private static int nbAdherent=0;

    @DisplayName("cr ation 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" })
    void testAdherents(String name) {
        Adherent a=new Adherent(name);
        nbAdherent++;
        assertEquals(nbAdherent, a.getNumero());
    }
}
```

# Sommaire

## 1 Introduction

## 2 Premiers pas

## 3 Approfondissements

- Test et exceptions
- Test et temps d'exécution
- Omission d'exécution et exécution conditionnée
- Les assertions
- Le test paramétré en JUnit 4
- **Les suites de test**

## 4 Conclusion

## 5 Le test unitaire et ses limites

# Suite de tests

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

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

# Sommaire

- 1 Introduction
- 2 Premiers pas
- 3 Approfondissements
  - Test et exceptions
  - Test et temps d'exécution
  - Omission d'exécution et exécution conditionnée
  - Les assertions
  - Le test paramétré en JUnit 4
  - Les suites de test
- 4 Conclusion
- 5 Le test unitaire et ses limites



## 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 ...

# Sommaire

## 1 Introduction

## 2 Premiers pas

## 3 Approfondissements

- Test et exceptions
- Test et temps d'exécution
- Omission d'exécution et exécution conditionnée
- Les assertions
- Le test paramétré en JUnit 4
- Les suites de test

## 4 Conclusion

## 5 Le test unitaire et ses limites

# 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 nécessairement 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

# Le test unitaire est-il suffisant ?

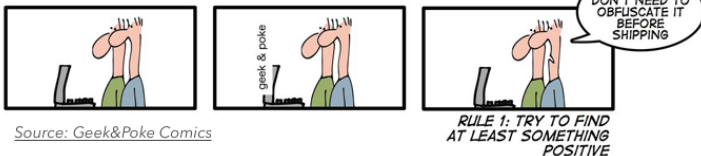
## Après le test unitaire, les autres tests

- 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

### *HOW TO MAKE A GOOD CODE REVIEW*

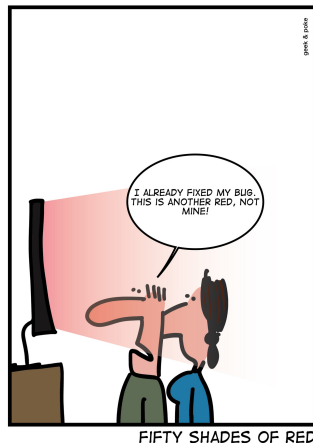


## Le bon testeur ...

- 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 Wolfgang

Bon courage !

## OLD ADAGES EXPLAINED

