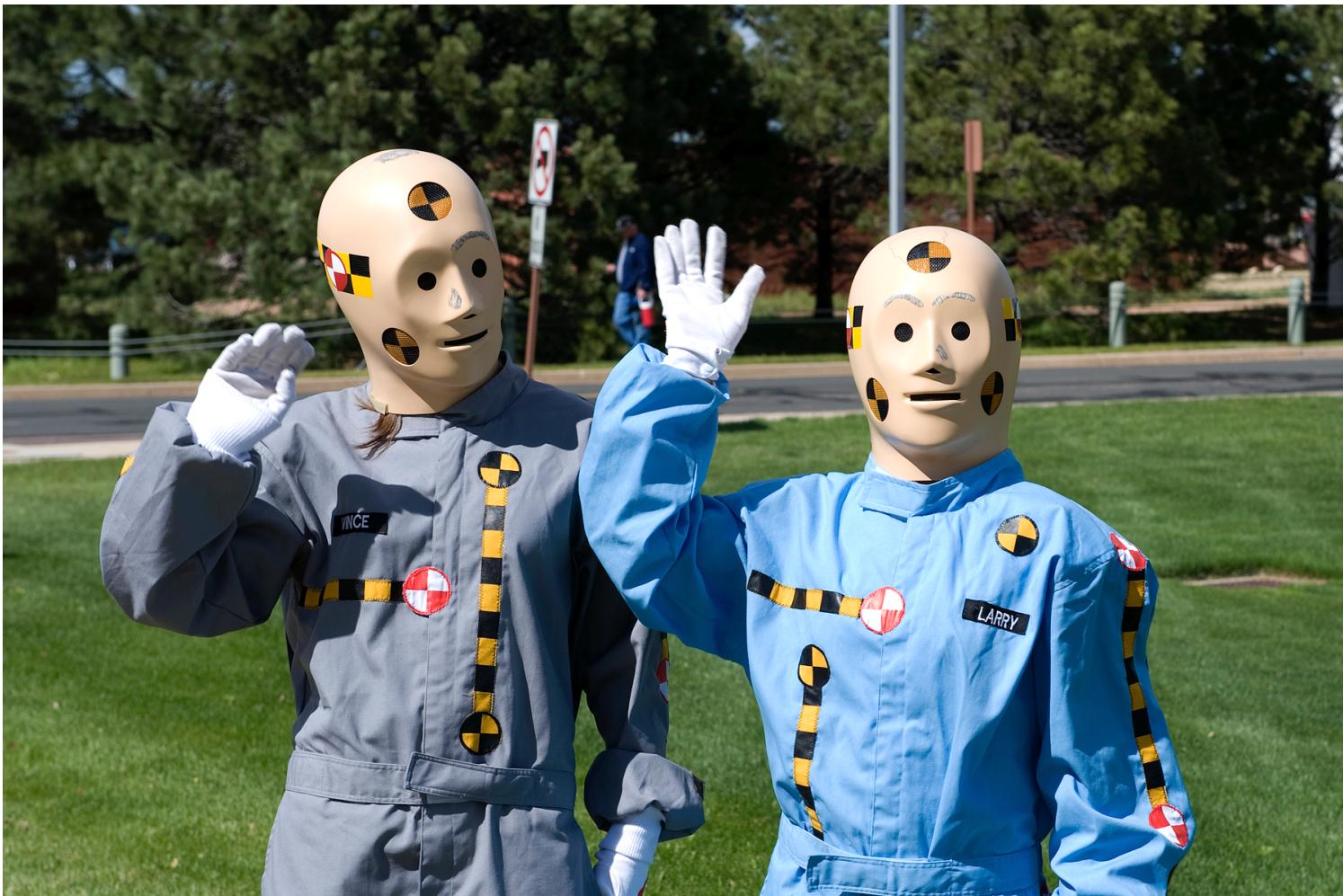


# Tests unitaires - JUnit

Philippe  
Collet

avec des slides  
de S. Mosser

Polytech  
Nice Sophia  
SI3  
2017-2018



# Plan

- V&V
- Tests ?
- JUnit 4
- De Junit 4 à Junit 5
- Ecrire les bons tests ?



# Principes de V&V

- Deux aspects de la notion de qualité :
  - Conformité avec la définition : **VALIDATION**
    - Réponse à la question : **faisons-nous le bon produit ?**
    - Contrôle en cours de réalisation, le plus souvent avec le client
    - **Défauts** par rapport aux besoins que le produit doit satisfaire
  - Correction d'une phase ou de l'ensemble : **VERIFICATION**
    - Réponse à la question : **faisons-nous le produit correctement ?**
    - Tests
    - **Erreurs** par rapport aux définitions précises établies lors des phases antérieures de développement

# Techniques statiques

- Avantages
  - contrôle systématique valable pour toute exécution, applicables à tout document
- Peuvent porter sur du code
  - Pas en situation réelle
  - Preuve de programme impossible à grande échelle (possible sur des propriétés très précises)
  - Analyse statique pour détecter des anomalies « typiques » (par exemple: SONAR)
- Peuvent porter sur d'autres documents
  - Revue, inspection
  - Utile pour détecter des erreurs mais coûteux en temps humain

# Techniques dynamiques

- Nécessitent une exécution du logiciel, une parmi des multitudes d'autres possibles
- Avantages
  - Vérification avec des conditions proches de la réalité
  - Plus à la portée du commun des programmeurs
- Inconvénients
  - Il faut provoquer des expériences, donc écrire du code et construire des données d'essais
  - Un test qui réussit ne démontre pas qu'il n'y a pas d'erreurs

☞ ***Les techniques statiques et dynamiques sont donc complémentaires***

*« Testing is the process of executing a program  
with the intent of finding errors »*

Glen Myers



# Tests : définition...

- Une expérience d'exécution, pour mettre en évidence un défaut ou une erreur
  - **Diagnostic** : quel est le problème
  - Besoin d'un **oracle**, qui indique si le résultat de l'expérience est conforme aux intentions
  - **Localisation** (si possible) : où est la cause du problème ?

- ☞ ***Les tests doivent mettre en évidence des erreurs !***
- ☞ ***On ne doit pas vouloir démontrer qu'un programme marche à l'aide de tests !***

# Le test, c'est du sérieux !



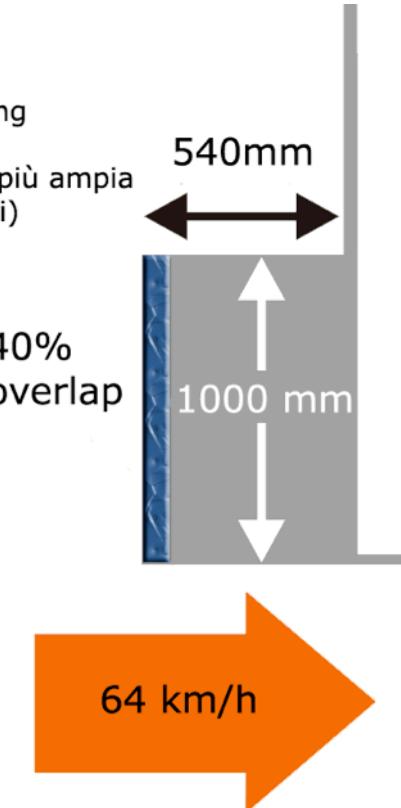
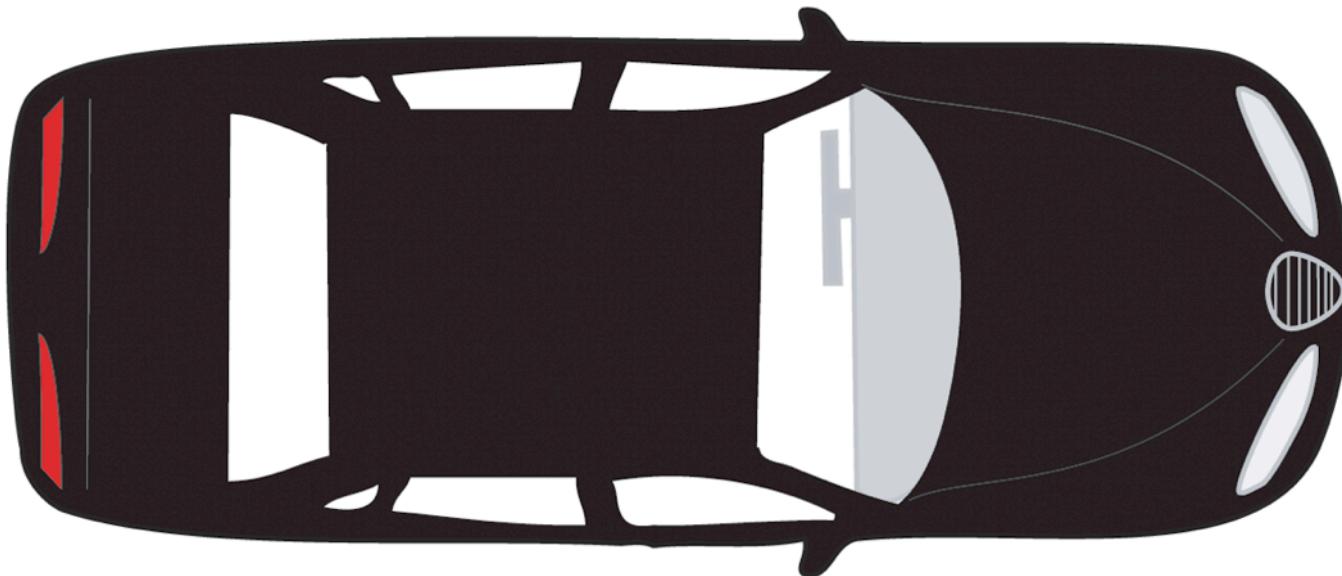
# Un test : un objectif / un cas de test

## FRONTAL IMPACT

- left-hand drive vehicles
- veicoli con guida a sinistra

EN: 40% overlap= 40% of the width of the widest part of the car (not including wing mirrors)

IT: 40% sovrapposizione = 40% della parte più ampia del veicolo ( esclusi specchietti retrovisori)



# Un test : des données de test



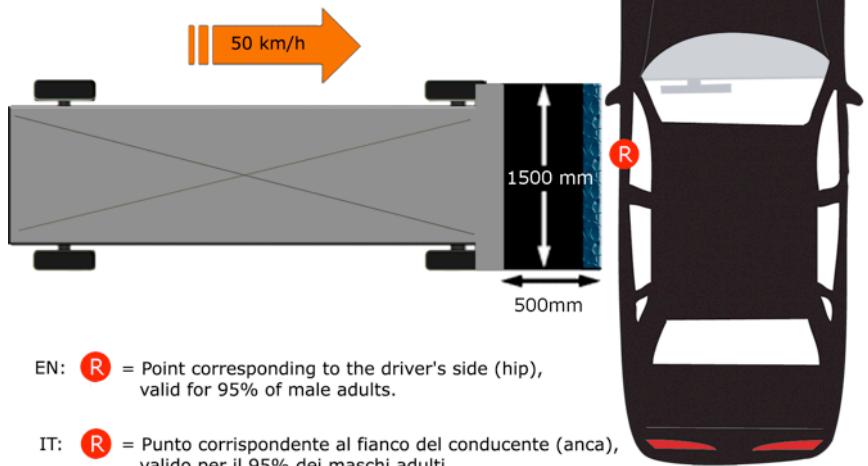
# Un test : des oracles

ADULT OCCUPANT		Total 35 pts   97%
FRONTAL IMPACT	15,4 pts	FRONTAL IMPACT
 Driver		<b>HEAD</b> Driver airbag contact stable Passenger airbag contact stable
 Passenger		<b>CHEST</b> Passenger compartment stable Windscreen Pillar rearward 4mm Steering wheel rearward none Steering wheel upward none Chest contact with steering wheel none
SIDE IMPACT CAR	8 pts	<b>UPPER LEGS, KNEES AND PELVIS</b> Stiff structures in dashboard none Concentrated loads on knees none
 Car		<b>LOWER LEGS AND FEET</b> Footwell Collapse none Rearward pedal movement brake - 11mm Upward pedal movement none
 Pole	7,9 pts	
REAR IMPACT (WHIPLASH)	3,4 pts	<b>SIDE IMPACT</b> Head protection airbag Yes Chest protection airbag Yes
		<b>WHIPLASH</b> Seat description Standard cloth 6 way manual Head restraint type Reactive Geometric assessment 1 pts
		<b>TESTS</b> - High severity 2,5 pts - Medium severity 2,5 pts - Low severity 2,4 pts

# Des tests : des CAS de tests

## SIDE IMPACT

- left-hand drive vehicles
- veicoli con guida a sinistra

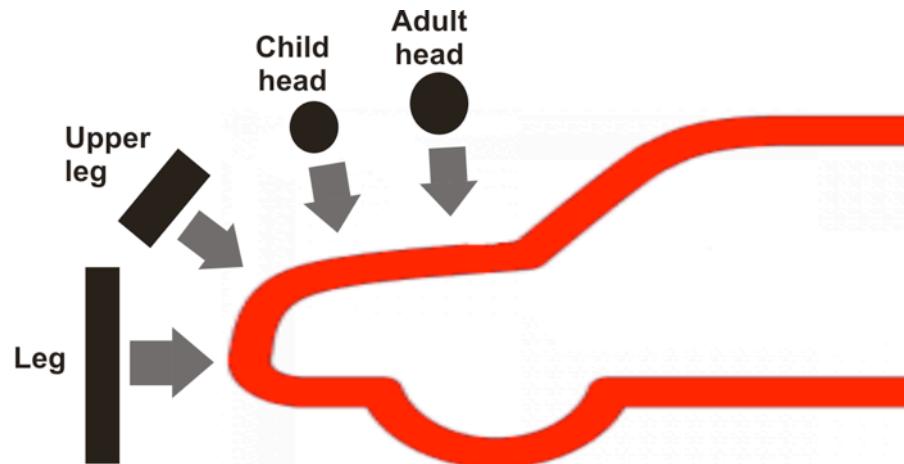
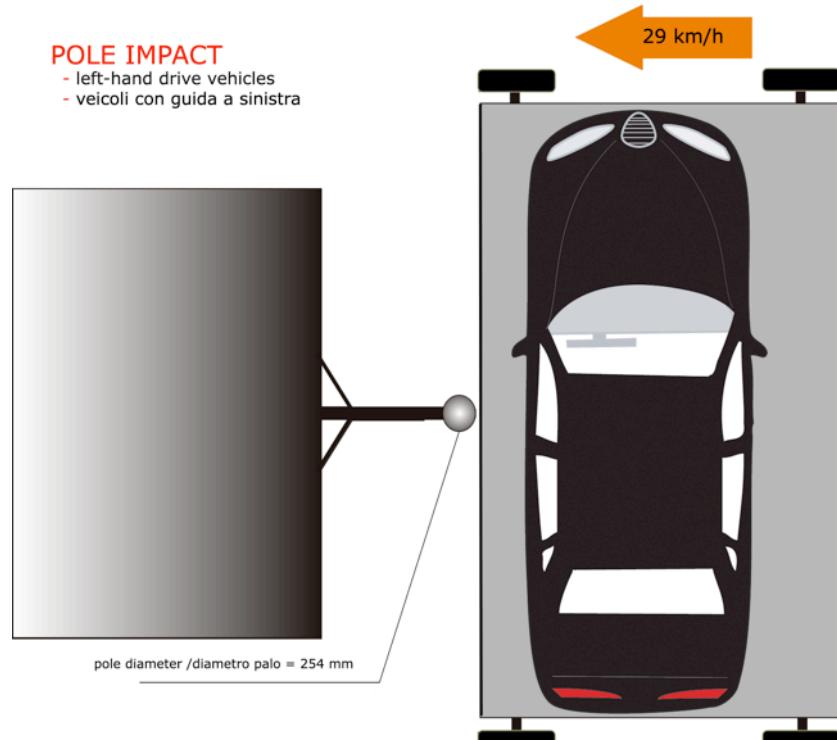


EN: R = Point corresponding to the driver's side (hip), valid for 95% of male adults.

IT: R = Punto corrispondente al fianco del conducente (anca), valido per il 95% dei maschi adulti.

## POLE IMPACT

- left-hand drive vehicles
- veicoli con guida a sinistra



# Des tests : compilation des résultats



Make and model	Overall rating	Adult	Child	Pedestrian	Safety assist	
AUDI Q5	2009		92%	84%	32%	71%
Honda Jazz	2009		78%	79%	60%	71%
Hyundai i20	2009		88%	83%	64%	86%
Kia Soul	2009		87%	86%	39%	86%
Peugeot 3008	2009		86%	81%	31%	97%
Suzuki Alto	2009		55%	46%	35%	29%

# Constituants d'un test



- Nom, objectif, commentaires, auteur
- Données : jeu de test
- Du code qui appelle des routines : cas de test
- Des oracles (vérifications de propriétés)
- Des traces, des résultats observables
- Un stockage de résultats : étalon
- Un compte-rendu, une synthèse...
- **Coût moyen : autant que le programme**

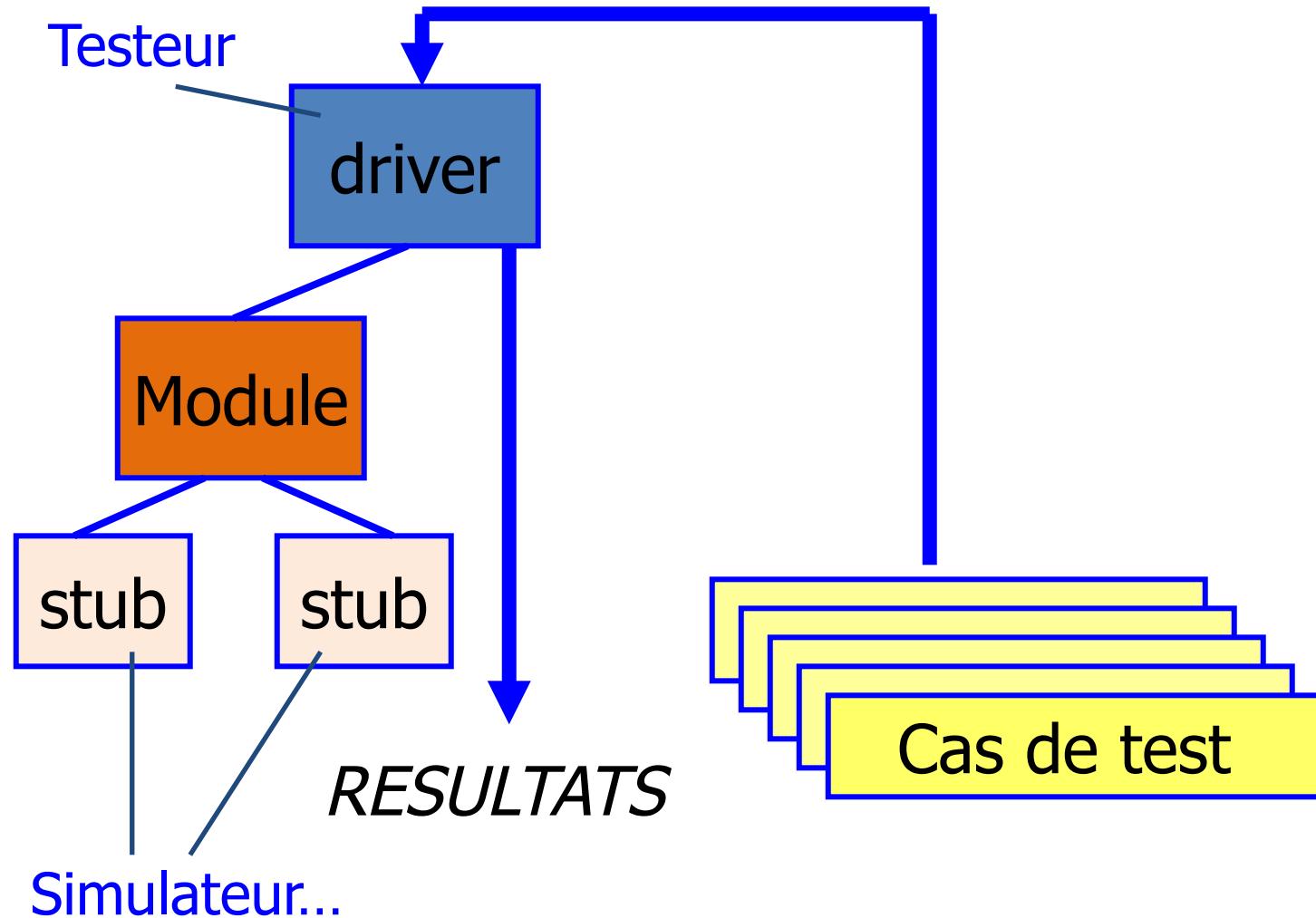
# Un essai n'est pas un test...



# Test vs. Essai vs. Débogage

- On converse les données de test
  - Le coût du test est amorti
  - Car un test doit être **reproductible**
- Le test est différent d'un essai de mise au point
- Le débogage est une enquête
  - Difficilement reproductible
  - Qui cherche à expliquer un problème

# Environnement du test unitaire



# JUnit 4



# JUnit

- La référence du tests unitaires en Java
- Trois des avantages de l'eXtreme Programming appliqués aux tests :
  - Comme les tests unitaires utilisent l'interface de l'unité à tester, ils amènent le développeur à réfléchir à l'utilisation de cette interface tôt dans l'implémentation
  - Ils permettent aux développeurs de détecter tôt des cas aberrants
  - **En fournissant un degré de correction documenté, ils permettent au développeur de modifier l'architecture du code en confiance**

# Exemple

```
class Money {  
    private int fAmount;  
    private String fCurrency;  
    public Money(int amount, String currency) {  
        fAmount= amount;  
        fCurrency= currency;  
    }  
  
    public int amount() {  
        return fAmount;  
    }  
  
    public String currency() {  
        return fCurrency;  
    }  
}
```

# Premier Test avant d'implémenter simpleAdd

```
import static org.junit.Assert.*;  
  
public class MoneyTest {  
    //...  
    @Test public void simpleAdd() {  
        Money m12CHF= new Money(12, "CHF");           // (1)  
        Money m14CHF= new Money(14, "CHF");  
        Money expected= new Money(26, "CHF");  
        Money result= m12CHF.add(m14CHF);             // (2)  
        assertTrue(expected.equals(result));          // (3)  
    } }
```

1. Code de mise en place du contexte de test (*fixture*)
2. Expérimentation sur les objets dans le contexte
3. Vérification du résultat, oracle...

# Les cas de test

- Ecrire des classes quelconques
- Définir à l'intérieur un nombre quelconque de méthodes annotés @Test
- Pour vérifier les résultats attendus (écrire des oracles !), il faut appeler une des nombreuses variantes de méthodes assertXXX() fournies
  - **assertTrue(String message, boolean test)**, **assertFalse(...)**
  - **assertEquals(...)** : test d'égalité avec equals
  - **assertSame(...)**, **assertNotSame(...)** : tests d'égalité de référence
  - **assertNull(...)**, **assertNotNull(...)**
  - **Fail(...)** : pour lever directement une AssertionFailedError
  - ***Surcharge sur certaines méthodes pour les différentes types de base***
  - **Faire un « import static org.junit.Assert.\* » pour les rendre toutes disponibles**

# Application à equals dans Money

```
@Test public void testEquals() {  
    Money m12CHF= new Money(12, "CHF");  
    Money m14CHF= new Money(14, "CHF");  
  
    assertTrue(!m12CHF.equals(null));  
    assertEquals(m12CHF, m12CHF);  
    assertEquals(m12CHF, new Money(12, "CHF"));  
    assertTrue(!m12CHF.equals(m14CHF));  
}
```

```
public boolean equals(Object anObject) {  
    if (anObject instanceof Money) {  
        Money aMoney= (Money) anObject;  
        return aMoney.currency().equals(currency())  
            && amount() == aMoney.amount();  
    }  
    return false;  
}
```

# *Fixture* : contexte commun

- Code de mise en place dupliqué !

```
Money m12CHF= new Money(12, "CHF");  
Money m14CHF= new Money(14, "CHF");
```

- Des classes qui comprennent plusieurs méthodes de test peuvent utiliser les annotations `@Before` et `@After` sur des méthodes pour initialiser, resp. nettoyer, le contexte commun aux tests (= *fixture*)
  - Chaque test s'exécute dans le contexte de sa propre installation, en appelant la méthode `@Before` avant et la méthode `@After` après chacune des méthodes de test
  - Pour deux méthodes, exécution équivalente à :
    - `@Before-method ; @Test1-method(); @After-method();`
    - `@Before-method ; @Test2-method(); @After-method();`
  - Cela doit assurer qu'il n'y ait pas d'effet de bord entre les exécutions de tests
  - **Le contexte est défini par des attributs de la classe de test**

# *Fixture* : application

```
public class MoneyTest {  
    private Money f12CHF;  
    private Money f14CHF;  
  
    @Before public void setUp() {  
        f12CHF= new Money(12, "CHF");  
        f14CHF= new Money(14, "CHF");  
    }  
  
    @Test public void testEquals() {  
        assertTrue(!f12CHF.equals(null));  
        assertEquals(f12CHF, f12CHF);  
        assertEquals(f12CHF, new Money(12, "CHF"));  
        assertTrue(!f12CHF.equals(f14CHF));  
    }  
  
    @Test public void testSimpleAdd() {  
        Money expected= new Money(26, "CHF");  
        Money result= f12CHF.add(f14CHF);  
        assertTrue(expected.equals(result));  
    }  
}
```

# Fixture au niveau de la classe

- **@BeforeClass**
  - 1 seule annotation par classe
  - Evaluée une seule fois pour la classe de test, avant tout autre initialisation @Before
  - Finalement équivalent à un constructeur...
- **@AfterClass**
  - 1 seule annotation par classe aussi
  - Evaluée une seule fois une fois tous les tests passés, après le dernier @After
  - Utile pour effectivement nettoyé un environnement (fermeture de fichier, effet de bord...)

# Quelques autres fonctionnalités

- Tester des levées d'exception
  - @Test(expected= *ClasseDException.class*)

```
@Test(expected = ArithmeticException.class)
public void divideByZero() throws ArithmeticException {
    calculator.divide(0);
}
```

- Tester une exécution avec une limite de temps
  - Spécifiée en millisecondes

```
@Test(timeout=100)
...
```

- *Pas d'équivalent en JUnit 3*

# Quelques autres fonctionnalités

- Ignorer (provisoirement) certains tests
  - Annotations supplémentaire `@Ignore`

```
@Ignore("not ready yet")
@Test
public void multiply() {
    calculator.add(10);
    calculator.multiply(10);
    assertEquals(calculator.getResult(), 100);
}
}
```

- *Pas d'équivalent en JUnit 3*

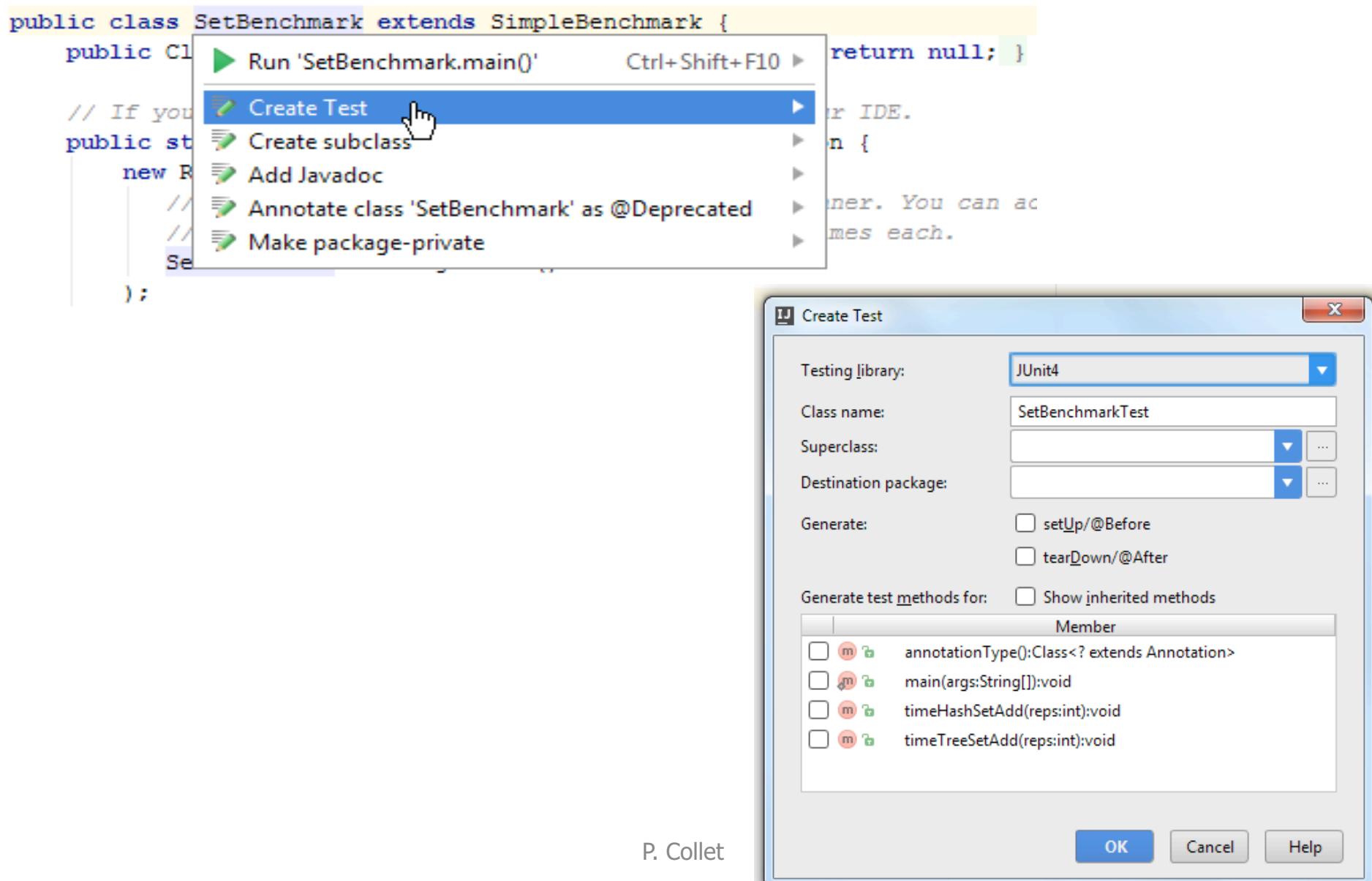
# Exécution des tests

- Par introspection des classes en Junit 4

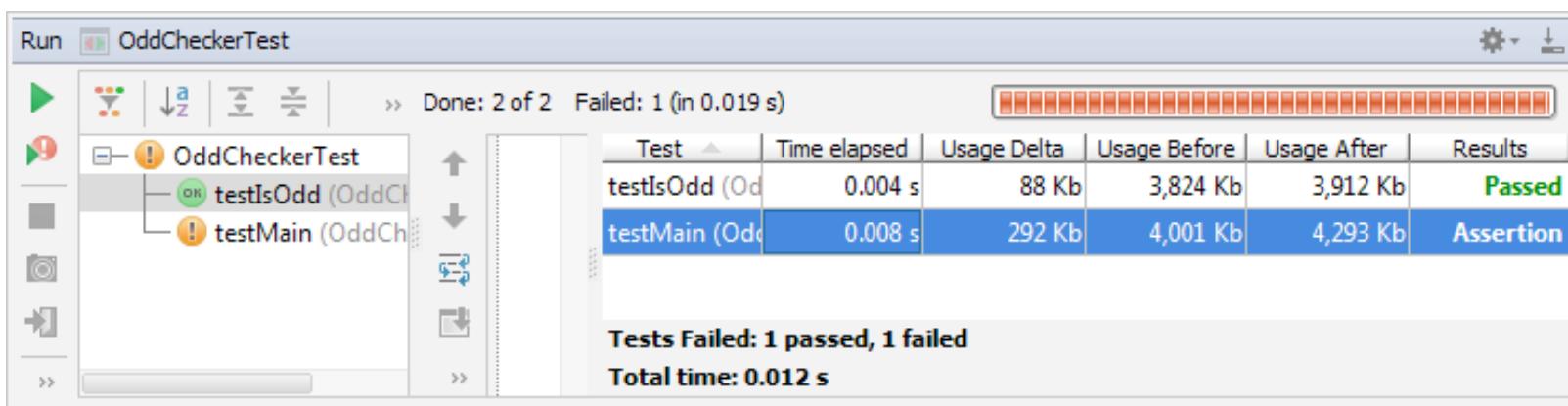
```
org.junit.runner.JUnitCore.runClasses(TestClass1.class, ...);  
    – introspection et exécution de la classe
```

- Récupération des annotations @Before, @After, @Test
- Exécution des tests
- Production d'un objet représentant le résultat
- Résultat juste : uniquement l'information que c'est OK
- Résultat faux : tous les détails (valeurs, ligne fautive) :
  - **Failure = erreur du test (détection d'une erreur dans le code testé)**
  - **Error = erreur/exception dans l'environnement du test (détection d'une erreur dans le code du test)**

# JUnit et IntelliJ: création des tests



# JUnit et IntelliJ : exécution des tests



Run OddCheckerTest

OddCheckerTest

testIsOdd (OddCheckerTest)

testMain (OddCheckerTest)

Navigate to testdata Ctrl+Alt+Home

View assertEquals Difference Ctrl+D

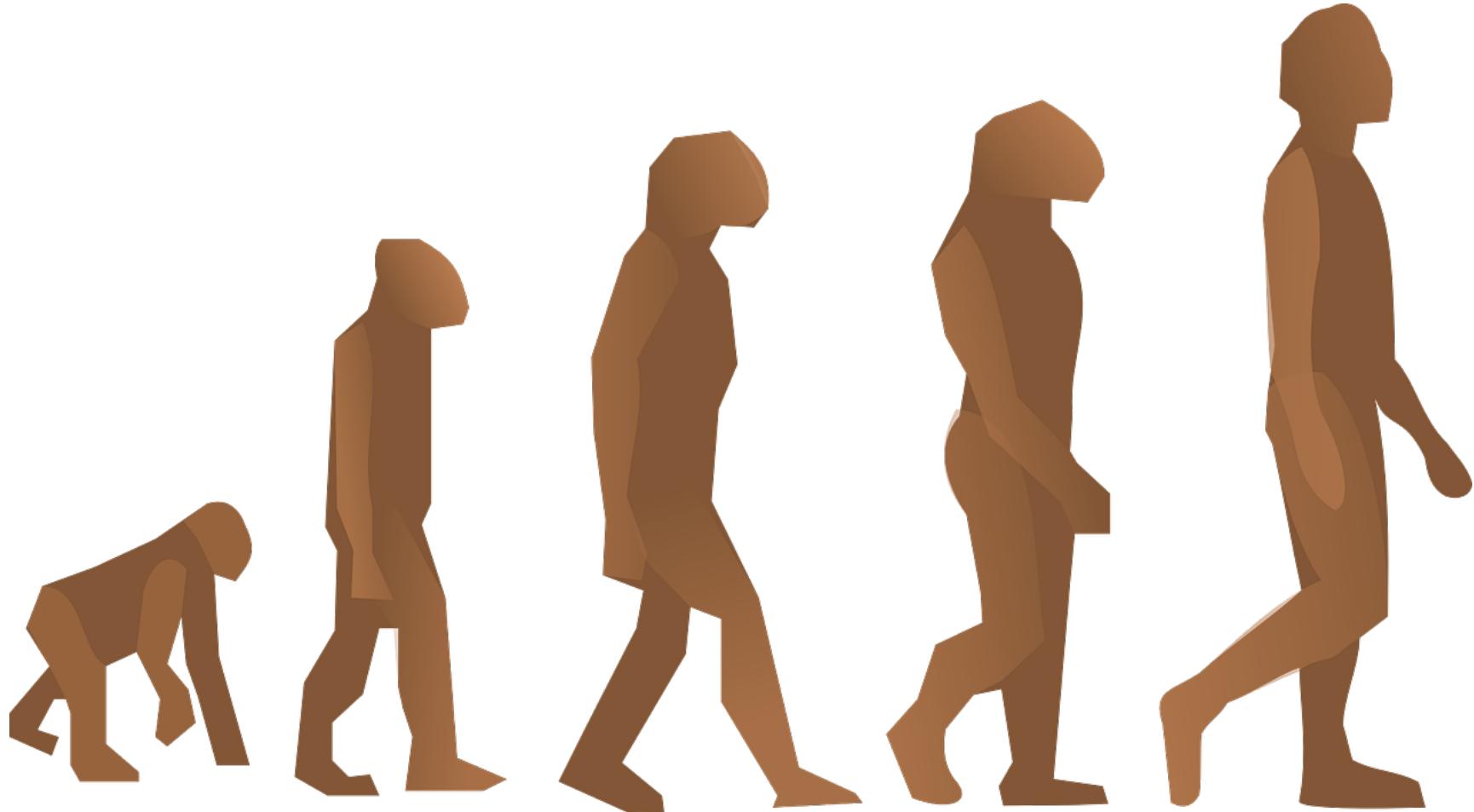
Comparison Failure

Ignore whitespace: Do not ignore

Expected (Read-only)	Actual (Read-only)
true	1
	false

1 difference Deleted Changed

# De Junit 4 à Junit 5



# Quelles améliorations ?

- Au lieu d'un seul gros jar pour toute la bibliothèque Junit, plusieurs sous-parties sont disponibles séparément
- Plusieurs runners de tests peuvent être utilisés simultanément
- Plusieurs fonctionnalités spécifiques à Java 8 (les lambdas en particulier) sont exploitées pour écrire des tests plus facilement

# Changement dans les annotations

- $\text{@Before} \Rightarrow \text{@BeforeEach}$
- $\text{@After} \Rightarrow \text{@AfterEach}$
- $\text{@BeforeClass} \Rightarrow \text{@BeforeAll}$
- $\text{@AfterClass} \Rightarrow \text{@AfterAll}$
- $\text{@Ignore} \Rightarrow \text{@Disabled}$

# Expected ⇒ assertThrows

- Junit 4

```
@Test(expected = ArithmeticException.class)
public void divideByZero() throws ArithmeticException {
    calculator.divide(0);
}
```

- Junit 5

```
@Test
public void divideByZero() throws ArithmeticException {
    Assertions.assertThrows(Exception.class, () -> {
        calculator.divide(0);
    });
}
```

# Timeout ⇒ assertTimeout

- Junit 4

```
@Test(timeout=100)
public void shouldFailBecauseTimeout() {
    ...
}
```

- Junit 5

```
@Test
public void shouldFailBecauseTimeout() {
    Assertions.assertTimeout(Duration.ofMillis(1),
        () -> ...);
}
```

# Et après ?



# Right ?

- validation des résultats en fonction de ce que définit la spécification
  - on doit pouvoir répondre à la question « comment sait-on que le programme s'est exécuté correctement ? »
  - si pas de réponse => spécifications certainement vagues, incomplètes
- Utilise plutôt une valeur commune :
  - Racine carrée ?
  - $\sqrt{4} \Rightarrow 2$

# B : Boundary conditions

- Les premiers à faire après le test Right
- Ceux qui ont le plus de valeur !
- Des bornes ? Des conditions limites ? Partout !
  - L'age d'une personne ?
  - L'email d'un étudiant ?
  - Le nombre de place dans un train ?
  - L'URL d'accès à ce fichier ?
  - $v_0 \Rightarrow ???$
- Boundaries must be **CORRECT**

# Correct: Conformance

- Test avec données en dehors du format attendu
- Un email ?
  - anachroniques ex. : !\*W@V"
  - non correctement formattées ex. : fred@foobar.
  - pas souhaitées : pouetpouet@yopmail.fr

# cOrrect: Order

- Test avec données sans l'ordre attendu
  - Les éléments sont censés être dans un certain ordre, que se passe-t-il s'ils le sont pas ?

# coRrect: Range

- Un intervalle quelque part ?
- Un indice dans un tableau
- L'indice est inférieur à 0...
- L'indice est supérieur à la taille max...

# corRect: Reference

- Quelles sont les hypothèses de la classe que l'on teste ?
  - Un objet est toujours présent ou dans un certain état lorsqu'il est passé en paramètre au constructeur ou à une méthode... Et sinon ?

# corrEct: Existence

- Que se passe-t-il si une valeur n'existe pas ?
- Que tourne stack.pop() ?
  - Null
  - Une exception ? Laquelle ?
  - Un objet bizarre qui représente la pile vide ???

# correCt: Cardinality

- Y-a-t-il un comptage de valeurs ?
- Un nombre d'éléments à vérifier avec un calcul ?
- Combien de poteaux pour faire une barrière de 20m avec 1 poteau tous les 2 mètres ?
- Règle du 0-1-N

# correcT: Time

- **Ordering Time**
  - Une méthode après forcément après une autre ?
- **Absolute Time**
  - Timezone
  - DST ? Passage à l'heure d'été, d'hiver
  - Un rattrapage de secondes ?

# Inverse

- Identifie **les relations inverses** qui permettent de vérifier le comportement
- Test de racine carrée :
  - $\sqrt{a^2} = a$
  - Avec une valeur quelconque de  $a$  (mais pas 1...)

# Cross check

- Identifie **les algorithmes équivalents** qui permettent de vérifier le comportement
- Test de racine carrée :
  - $\sqrt{474} = \text{Math.sqrt}(474)$

Ma version

La version équivalente

# Error condition

- Identifier ce qui se passe quand
  - Le disque, la mémoire sont pleins
  - Il y a perte de connexion réseau
  - Un argument n'est pas dans le domaine attendu
- Equivalent à la notion de robustesse
- Exemple :
  - $\sqrt{-1} \Rightarrow$  Illegal Argument

# Performance

- Vérifier des performances attendues
  - $\text{Time}(\sqrt{474}) < 200 \text{ ms}$
- Identifier des régressions dans les performances
- Attention, cette partie est un domaine de test non-fonctionnel à part entière (charge, performance, etc.).

# Synthèse

- Right : est-ce que les résultats sont corrects ?
- B (Boundary) : est-ce que les conditions aux limites sont correctes ?
- I (Inverse) : est-ce que l'on peut vérifier la relation inverse ?
- C (Cross-check) : est-ce que l'on peut vérifier le résultat autrement ?
- E (Error condition) : est-ce que l'on peut forcer l'occurrence d'erreurs ?
- P (Performance) : est-ce que les performances sont prévisibles ?

# Petits aspects méthodologiques



- **Coder/tester, coder/tester...**
- **lancer les tests aussi souvent que possible**
  - aussi souvent que le compilateur !
- **Commencer par écrire les tests sur les parties les plus critiques**
  - Ecrire les tests qui ont le meilleur retour sur investissement !
- **Si on se retrouve à déboguer à coup de System.out.println(), il vaut mieux écrire un test à la place**
- **Quand on trouve un bug, écrire un test qui le caractérise**

