

Tests orientés objets Junit

Philippe Collet

L3 Informatique
2014-2014



Plan



- Rappel sur V&V
- Tests
- JUnit 4
- Quelques principes méthodologiques

Rappels sur 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



Les tests

*" 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 !***
- Souvent négligé car :
 - les chefs de projet n'investissent pas pour un résultat négatif
 - les développeurs ne considèrent pas les tests comme un processus destructeur

Le test, c'est du sérieux !



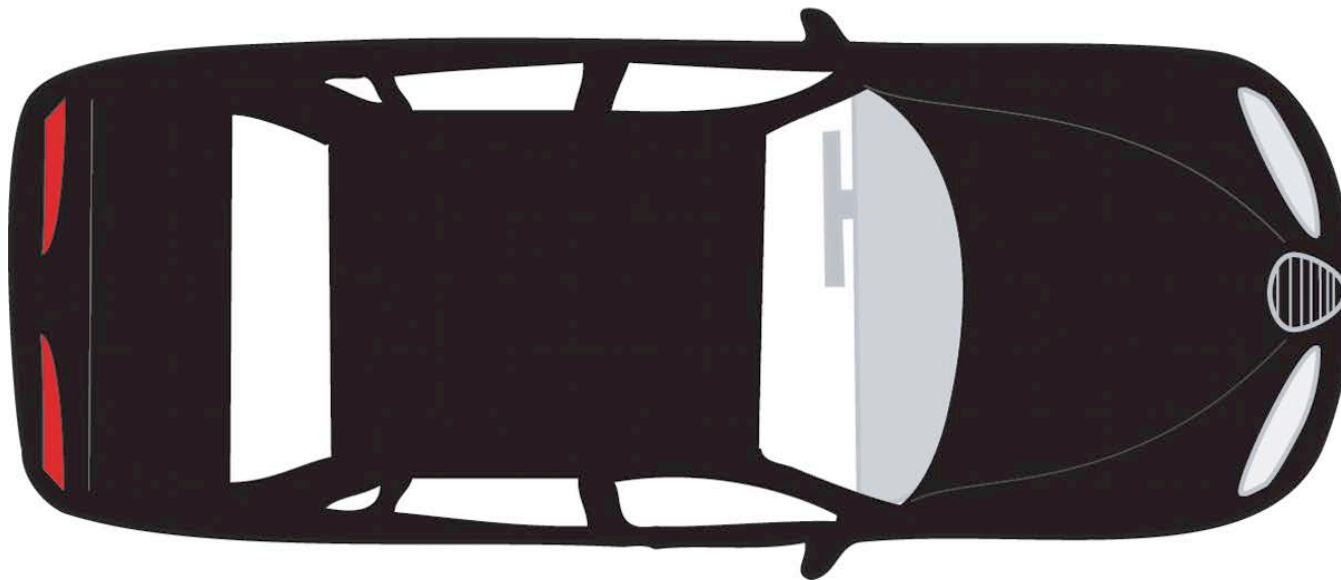
Un test : un objectif / un cas de test

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)



40%
overlap

540mm

1000 mm

64 km/h

Un test : des données de test



Un test : des oracles

ADULT OCCUPANT

Total 35 pts | 97%

FRONTAL IMPACT

15,4 pts



Driver



Passenger

SIDE IMPACT CAR

8 pts

SIDE IMPACT POLE

7,9 pts



Car



Pole

REAR IMPACT (WHIPLASH)

3,4 pts



	GOOD
	ADEQUATE
	MARGINAL
	WEAK
	POOR

FRONTAL IMPACT

HEAD

Driver airbag contact stable

Passenger airbag contact stable

CHEST

Passenger compartment stable

Windscreen Pillar rearward 4mm

Steering wheel rearward none

Steering wheel upward none

Chest contact with steering wheel none

UPPER LEGS, KNEES AND PELVIS

Stiff structures in dashboard none

Concentrated loads on knees none

LOWER LEGS AND FEET

Footwell Collapse none

Rearward pedal movement brake - 11mm

Upward pedal movement none

SIDE IMPACT

Head protection airbag Yes

Chest protection airbag Yes

WHIPLASH

Seat description Standard cloth 6 way manual

Head restraint type Reactive

Geometric assessment 1 pts

TESTS

- High severity 2,5 pts

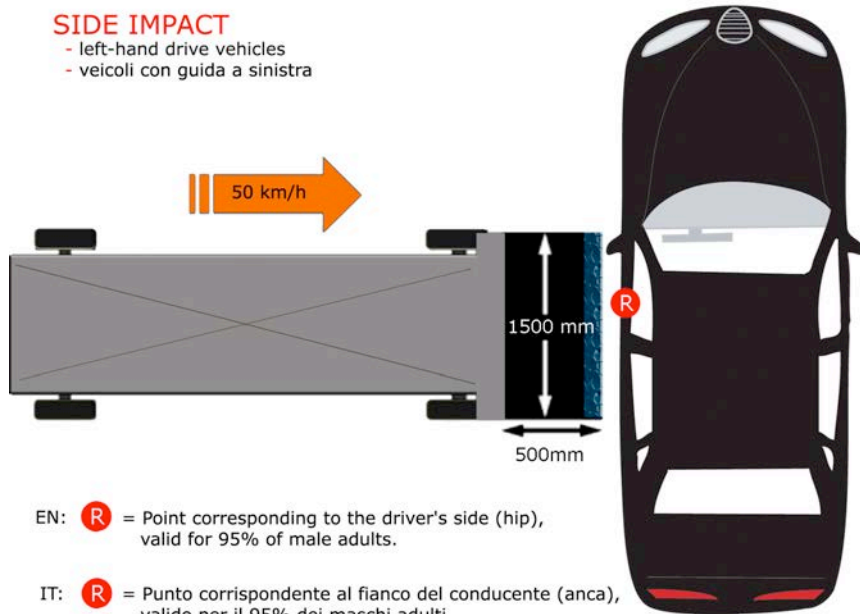
- Medium severity 2,5 pts

- Low severity 2,4 pts

Des tests : des CAS de tests

SIDE IMPACT

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

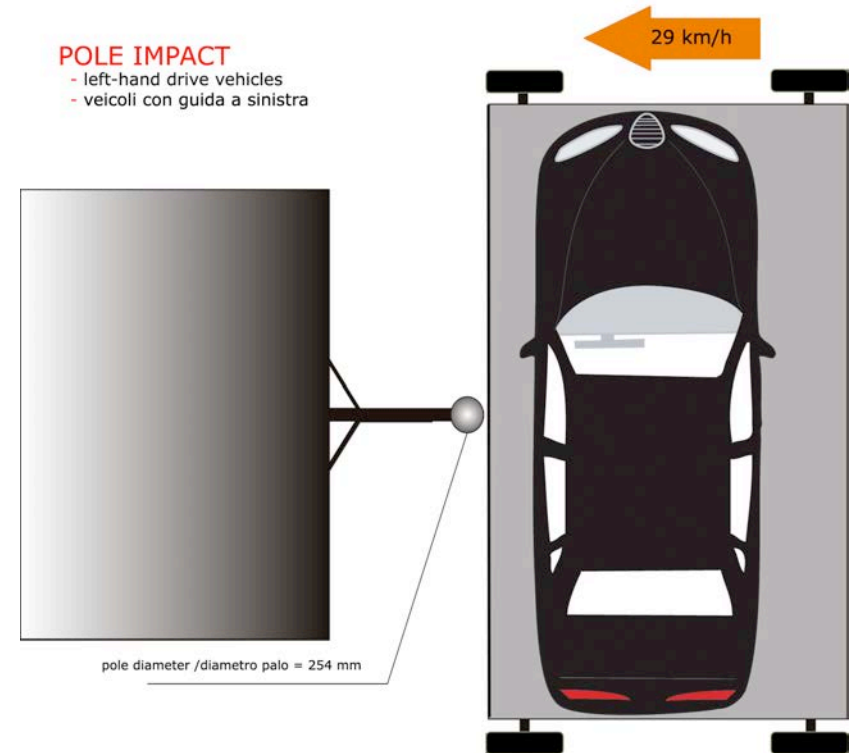


EN: **R** = Point corresponding to the driver's side (hip), valid for 95% of male adults.

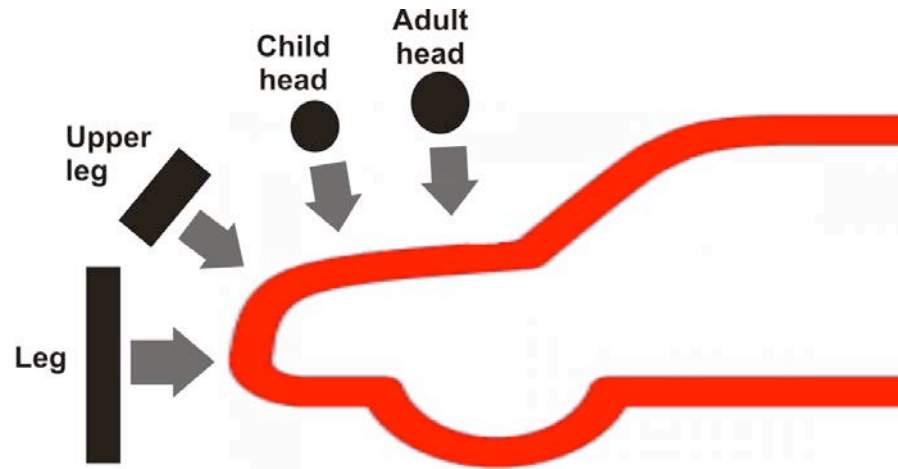
IT: **R** = Punto corrispondente al fianco del conducente (anca), valido per il 95% dei maschi adulti.

POLE IMPACT

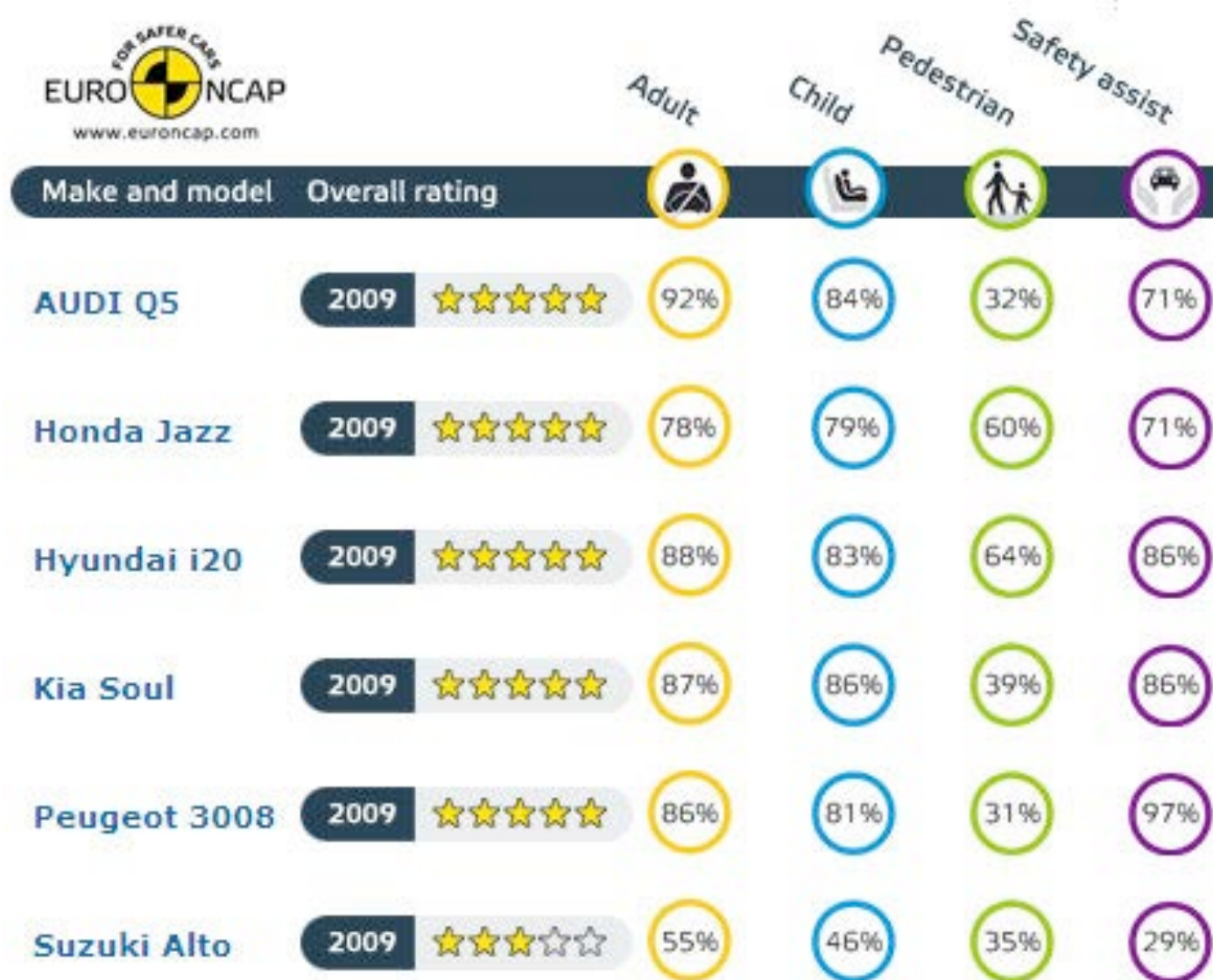
- left-hand drive vehicles
- veicoli con guida a sinistra



pole diameter / diametro palo = 254 mm



Des tests : compilation des résultats



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

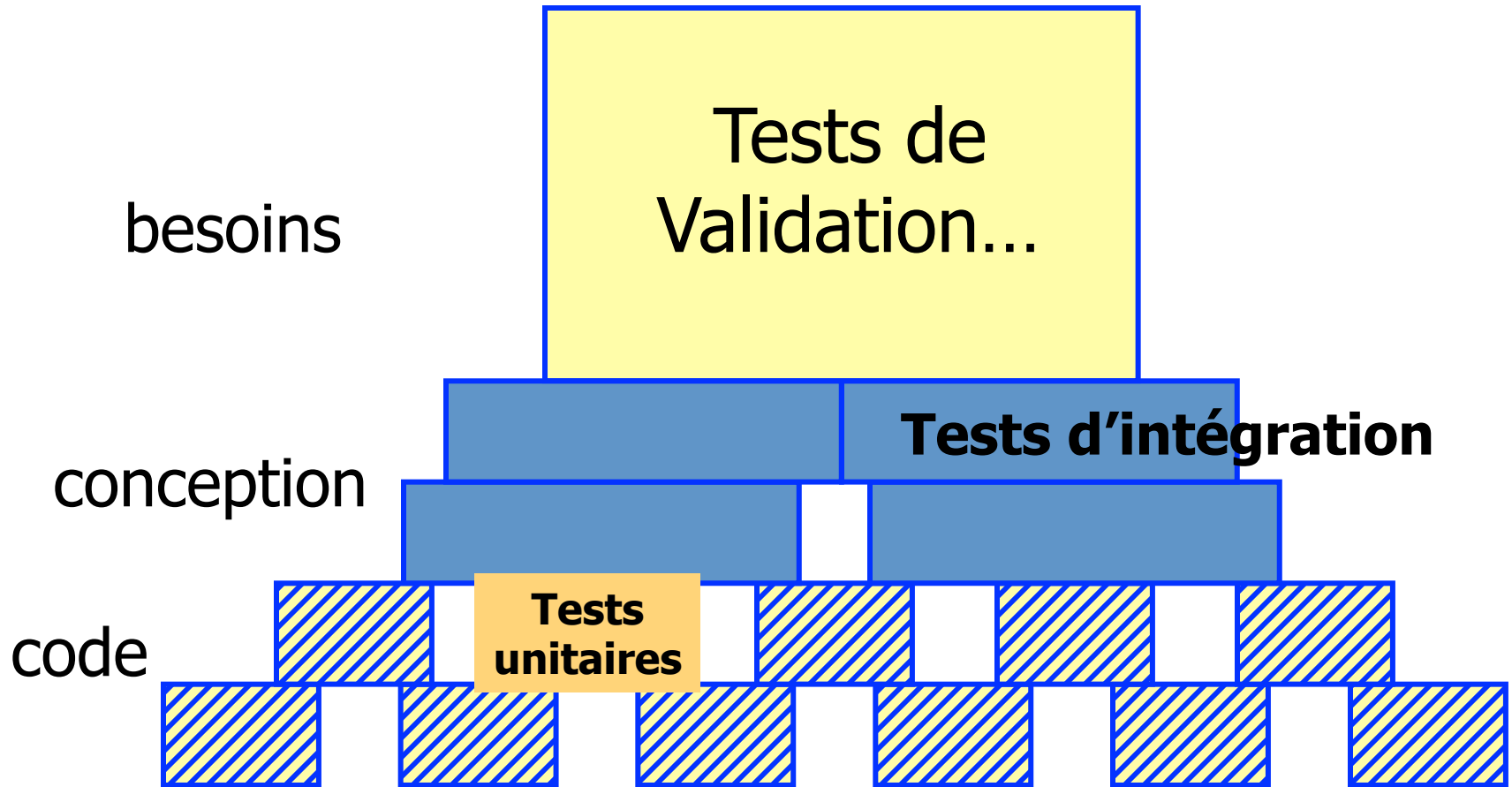
Un essai n'est pas un test...



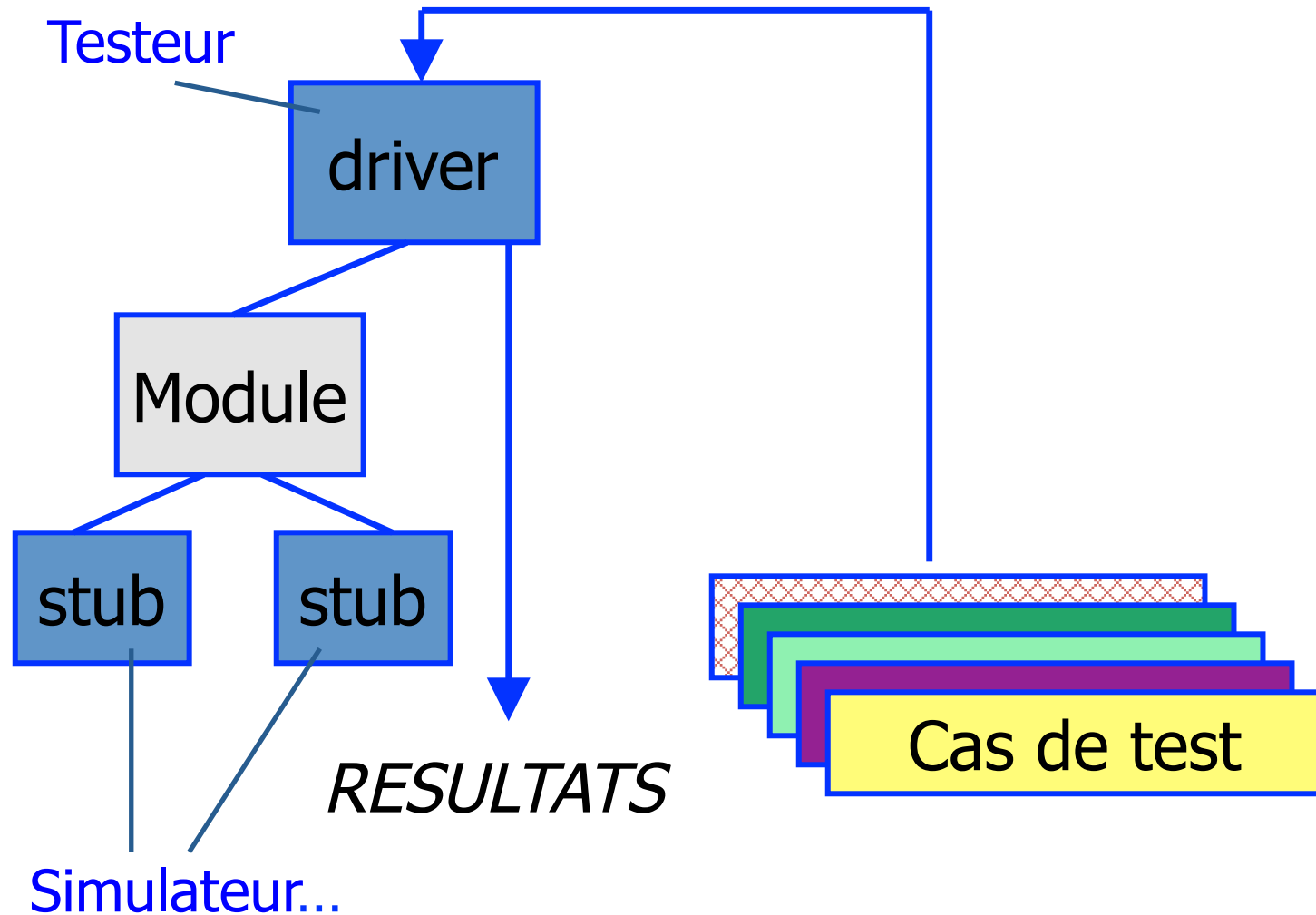
Test vs. Essai vs. Débogage

- 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


Les stratégies de test



Environnement du test unitaire



Stratégies de test OO

- Les classes sont la plus petite unité testable
 - L'héritage introduit de nouveaux contextes pour les méthodes
 - Le comportement des méthodes héritées peut être modifié à cause des méthodes appelées à l'intérieur de leur corps
-  Les méthodes doivent être testées pour chaque classe
- Les objets ont des états : les procédures de test doivent en tenir compte



JUnit

JUnit v4

www.junit.org

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**

Example

```
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
 - assertEquals(...), assertEquals(...) : tests d'égalité de référence
 - assertNull(...), assertNotNull(...)
 - Fail(...) : pour lever directement une AssertionError
 - *Surcharge sur certaines méthodes pour les différents 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
 - Évaluée une seule fois pour la classe de test, avant toute autre initialisation `@Before`
 - Finalement équivalent à un constructeur...
- `@AfterClass`
 - 1 seule annotation par classe aussi
 - Évaluée une seule fois une fois tous les tests passés, après le dernier `@After`
 - Utile pour effectivement nettoyer un environnement (fermeture de fichier, effet de bord de manière générale)

Exécution des tests

- Par introspection des classes

- Classe en paramètre de la méthode

```
org.junit.runner.JUnitCore.runClasses(TestClass1.class, ...);
```

- Introspection à l'exécution de la classe
- Récupération des annotations @Before, @After, @Test
- Exécution des tests suivant la sémantique définie (cf. transp. précédents)
- Production d'un objet représentant le résultat
 - Résultat faux : détail de l'erreur (Stack Trace, etc.)
 - Résultat juste : uniquement le comptage de ce qui est juste

- Précision sur la forme résultat d'un passage de test

- **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)**

Exécution des tests en ligne de commande

- Toujours à travers la classe `org.junit.runner.JUnitCore`

```
java org.junit.runner.JUnitCore com.acme.LoadTester  
com.acme.PushTester
```

- Quelques mots sur l'installation
 - Placer `junit-4.10.jar` dans le CLASSPATH (compilation et exécution)
 - C'est tout...
 - Eclipse le fait presque tout seul...

Autres fonctionnalités

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

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

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

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

```
...
```

- *Pas d'équivalent en JUnit 3*

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*

Paramétrage des tests

```
@RunWith(value=Parameterized.class)
public class FactorialTest {

    private long expected;
    private int value;

    @Parameters
    public static Collection data() {
        return Arrays.asList( new Object[][] {
            { 1, 0 },    // expected, value
            { 1, 1 },
            { 2, 2 },
            { 24, 4 },
            { 5040, 7 },
        });
    }

    public FactorialTest(long expected, int value) {
        this.expected = expected;
        this.value = value;
    }

    @Test
    public void factorial() {
        Calculator calculator = new Calculator();
        assertEquals(expected, calculator.factorial(value));
    }
}
```

Paramétrage des tests

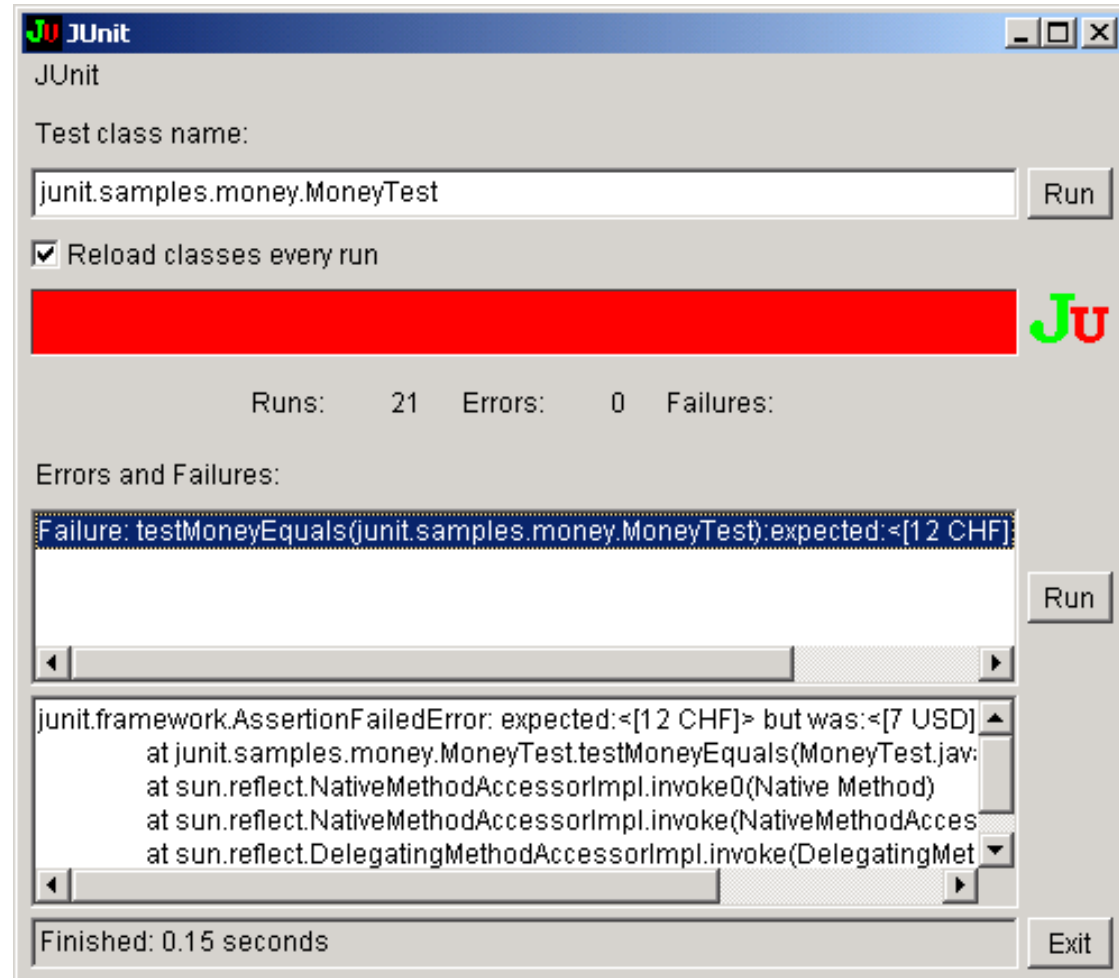
- `@RunWith(value=Parameterized.class)`
 - Exécute tous les tests de la classe avec les données fournies par la méthode annotée par `@Parameters`
- `@Parameters`
 - 5 éléments dans la liste de l'exemple
 - Chaque élément est un tableau utilisé comme arguments du constructeur de la classe de test
 - Dans l'exemple, les données sont utilisées dans `assertEquals`
- Equivalent à :

```
factorial#0:  assertEquals( 1, calculator.factorial( 0 ) );  
factorial#1:  assertEquals( 1, calculator.factorial( 1 ) );  
factorial#2:  assertEquals( 2, calculator.factorial( 2 ) );  
factorial#3:  assertEquals( 24, calculator.factorial( 4 ) );  
factorial#4:  assertEquals( 5040, calculator.factorial( 7 ) );
```

■ *Pas d'équivalent en JUnit 3*

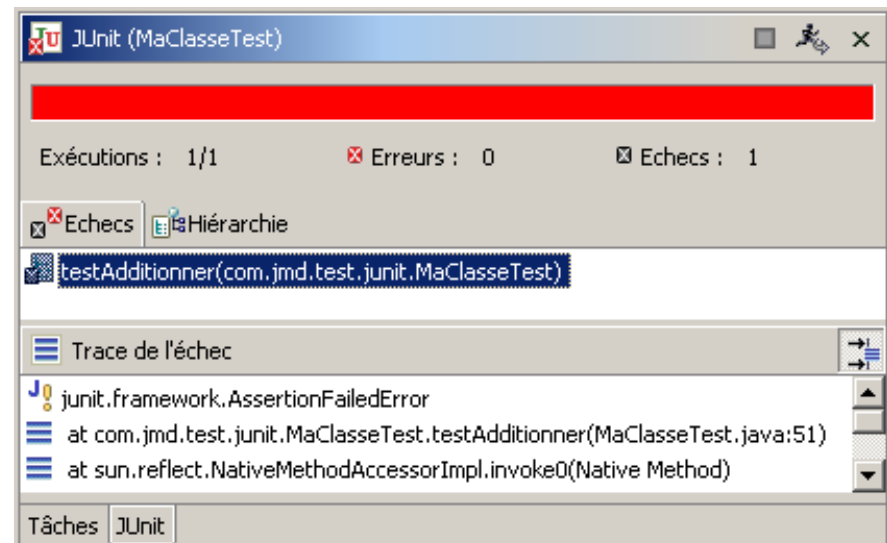
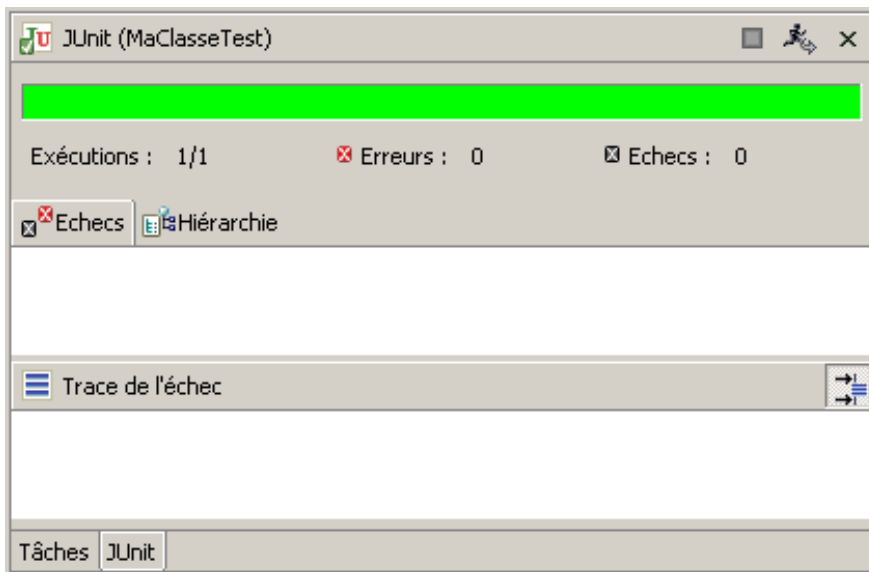
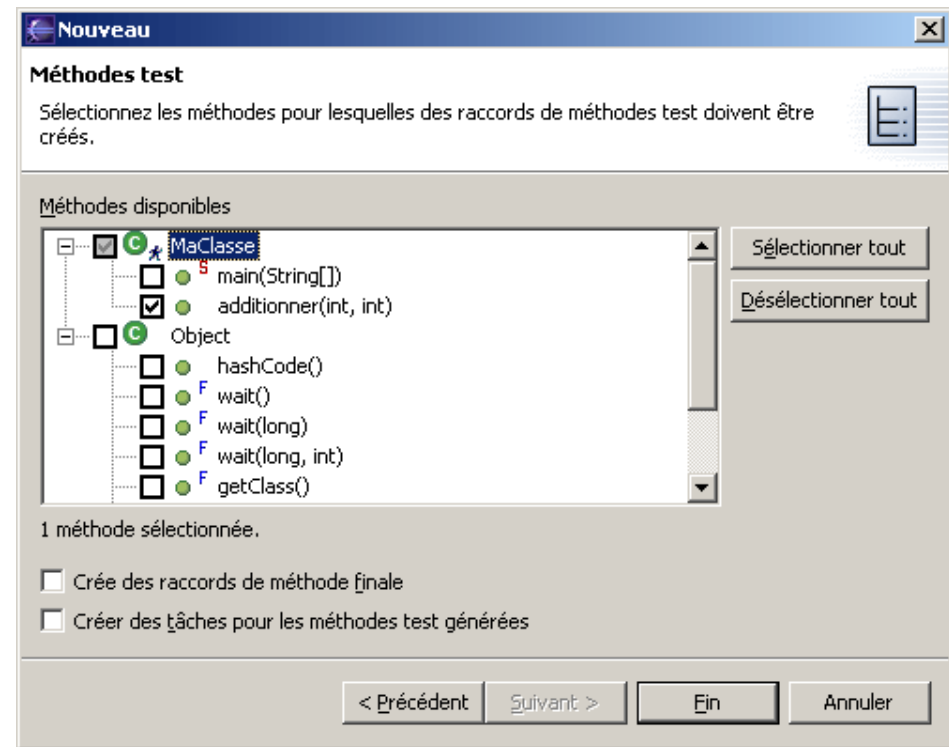
JUnit : TestRunner

- Exécuter et afficher les résultats
- Deux versions dans JUnit (textuelle, graphique)
- Intégration dans des IDE



JUnit dans Eclipse

- Assistants pour :
 - Créer des cas de test
- TestRunner intégré à l'IDE



Et après ?



Quoi tester ? Quelques principes

- **Principe Right-BICEP**

- 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 ?

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
- tests = traduction des spécifications

B : Boundary conditions

- identifier les conditions aux limites de la spécification
- que se passe-t-il lorsque les données sont
 - anachroniques ex. : !*W@V"
 - non correctement formatées ex. : fred@foobar.
 - vides ou nulles ex. : 0, 0.0, "", null
 - extraordinaires ex. : 10000 pour l'age d'une personne
 - dupliquées ex. : doublon dans un Set
 - non conformes ex. : listes ordonnées qui ne le sont pas
 - désordonnées ex. : imprimer avant de se connecter
- Principe « CORRECT » =
 - Conformance : test avec données en dehors du format attendu
 - Ordering : test avec données sans l'ordre attendu
 - Range : test avec données hors de l'intervalle
 - Reference : test des dépendances avec le reste de l'application (précondition)
 - Existence : test sans valeur attendu (pointeur nul)
 - Cardinality : test avec des valeurs remarquables (bornes, nombre maximum)
 - Time : test avec des cas où l'ordre à une importance

Inverse – Cross check

- Identifier
 - les relations inverses
 - les algorithmes équivalents (cross-check)
- qui permettent de vérifier le comportement
- Exemple : test de la racine carrée en utilisant la fonction de mise au carré...

Error condition – Performance

- Identifier ce qui se passe quand
 - Le disque, la mémoire, etc. sont pleins
 - Il y a perte de connexion réseau
- ex. : vérifier qu'un élément n'est pas dans une liste
 - => vérifier que le temps est linéaire avec la taille de la liste
- Attention, cette partie est un domaine de test non-fonctionnel à part entière (charge, performance, etc.).

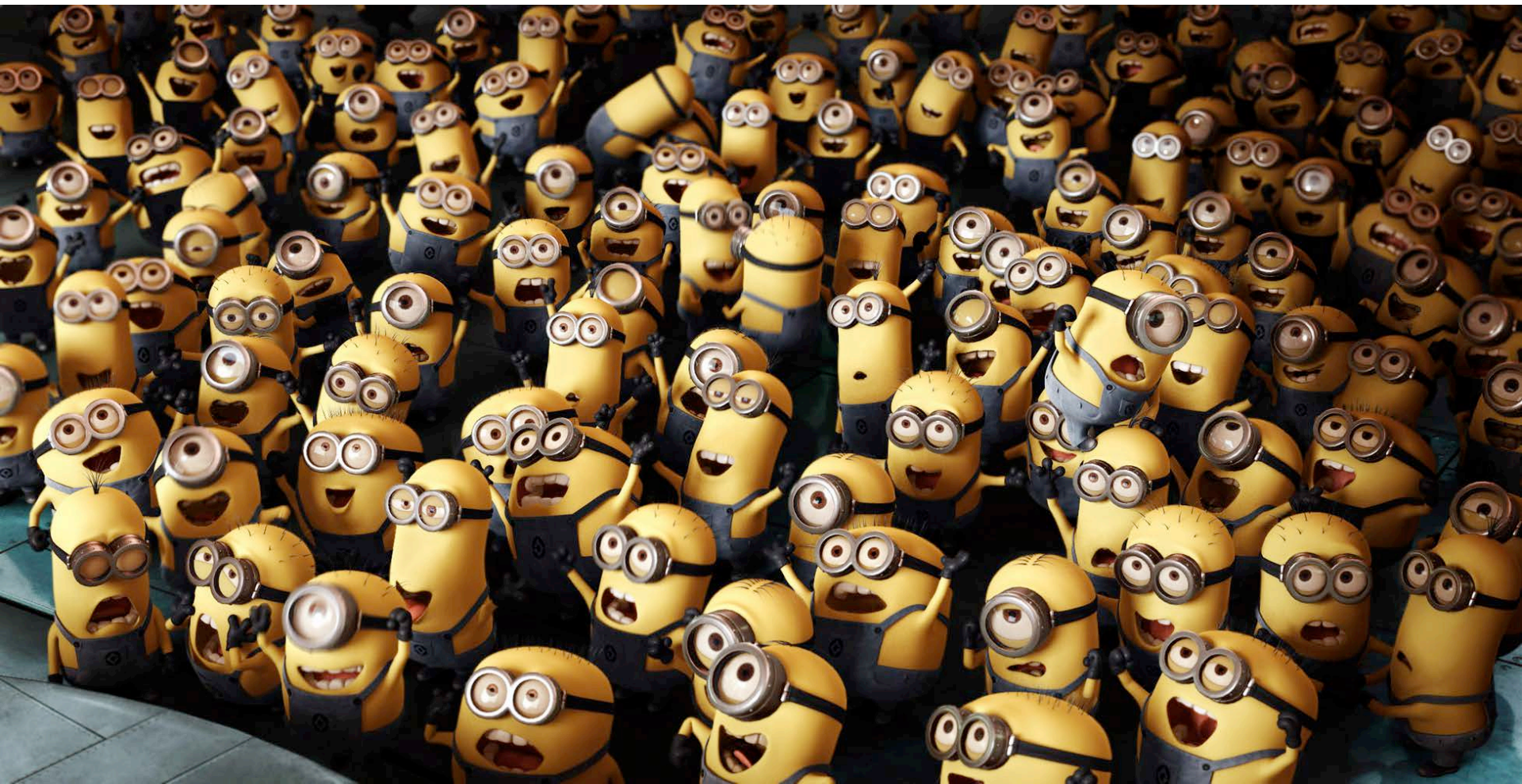
Petits aspects méthodologiques



- **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 !
 - Approche *Extreme Programming*
- **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**

On écrit les tests avant le code !

© Universal Pictures International



Test-Driven Development

- Méthode traditionnelle (incrémentale)
 - Ajouter un peu de code
 - Ajouter un test sur ce bout de code
- Méthode TDD
 - Ajouter un code de test
 - Ajouter du code qui respecte le test
- Mise en pratique (code de couleur Junit)
 - **R (Red)**: écrire un code de test et les faire échouer
 - **G (Green)**: écrire le code métier qui valide le test
 - **R (Refactor)**: remanier le code afin d'en améliorer la qualité

Cycle TDD : 5 étapes

1. Ecriture d'un premier test
2. Exécuter le test et vérifier qu'il échoue
 - car le code qu'il teste n'a pas encore été implémenté
3. Ecriture de l'implémentation pour faire passer le test
 - il existe différentes manières de corriger ce code
4. Exécution des tests afin de contrôler que les tests passent
 - l'implémentation va respecter les règles fonctionnelles des tests unitaires
5. Remaniement (Refactor) du code afin d'en améliorer la qualité
 - mais en conservant les mêmes fonctionnalités
 - Les tests sont repassés après refactoring !!!

TDD : bonnes pratiques

- Livrer les fonctionnalités dont le logiciel a réellement besoin, pas celles que le programmeur croît devoir fournir (une évidence *a priori*)
- Ecrire du code client comme si le code à développer existait déjà et avait été conçu en tout point pour nous faciliter la vie !
- Intégration continue : pendant le développement, le programme marche toujours, peut être ne fait-il pas tout ce qui est requis, mais ce qu'il fait, il le fait bien !

