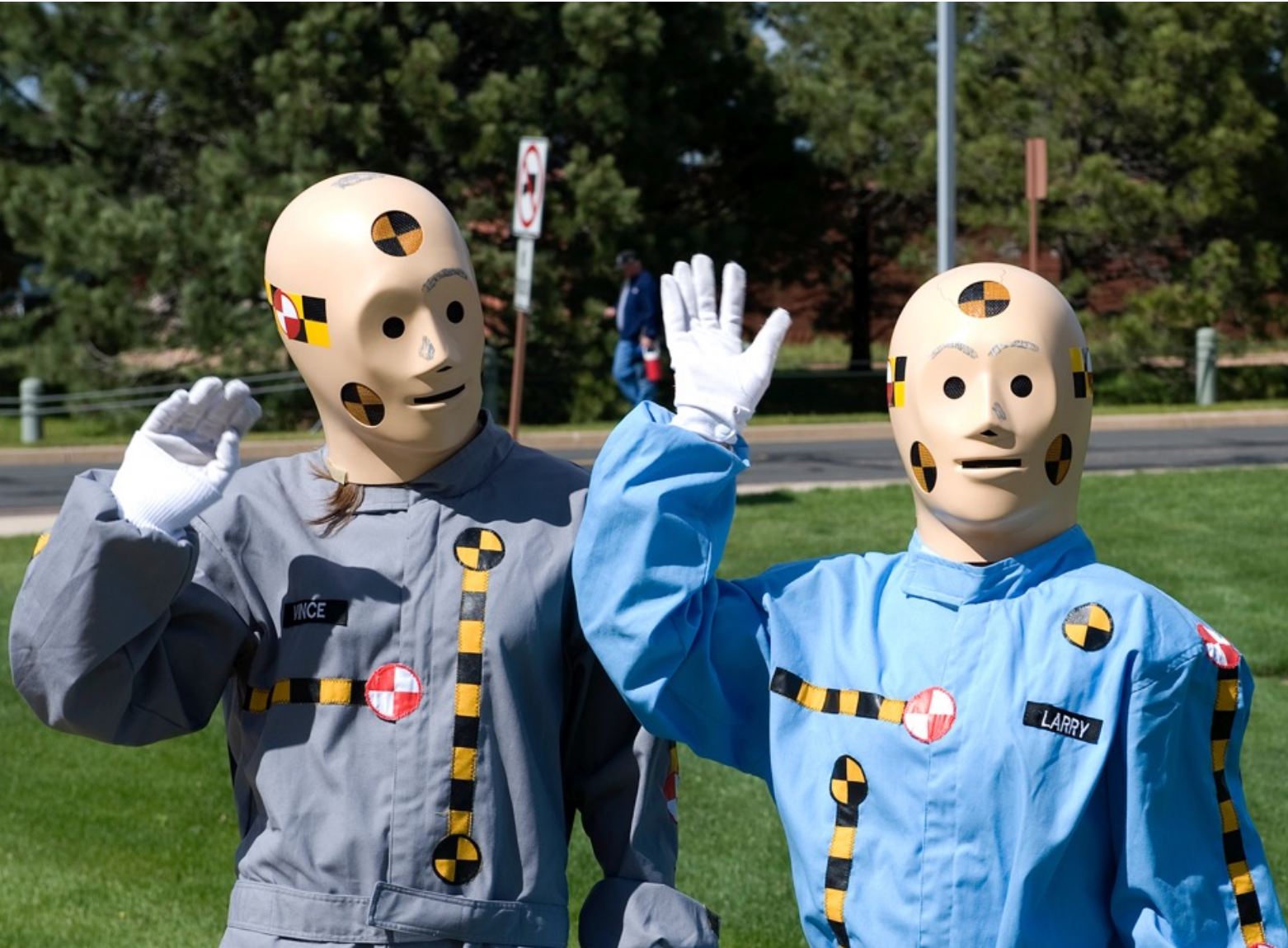


Tests unitaires

JUnit



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, que l'on verra en cours optionnel QGL)
- 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 !



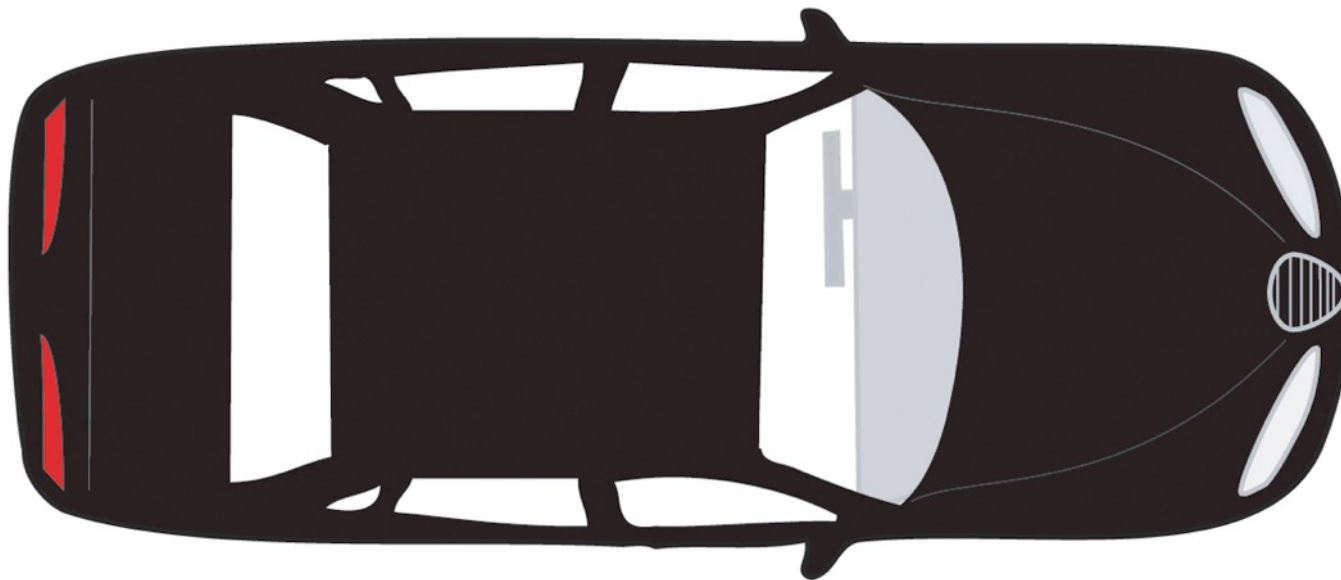
Un test : un objectif / un cas de test

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)



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

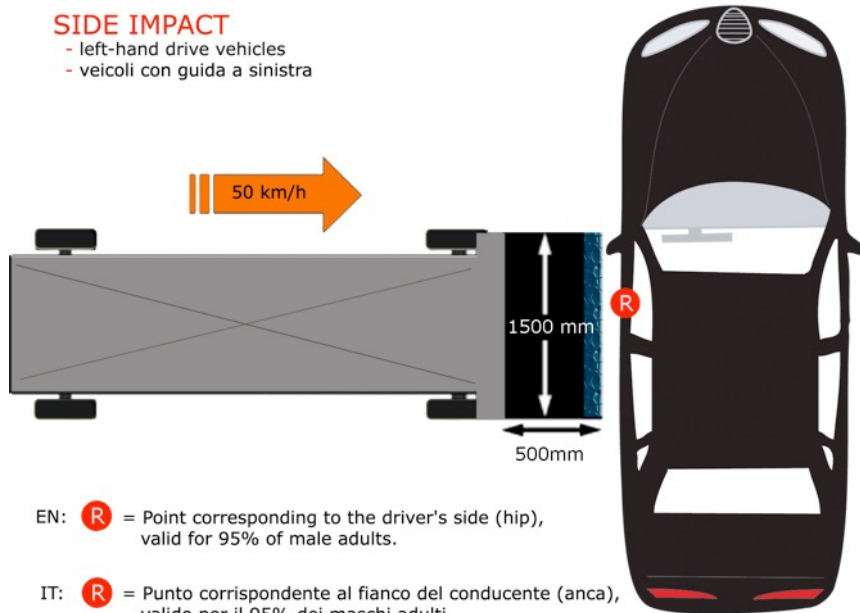
- Medium severity 2,5 pts

- Low severity 2,4 pts

Des tests : des CAS de tests

SIDE IMPACT

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

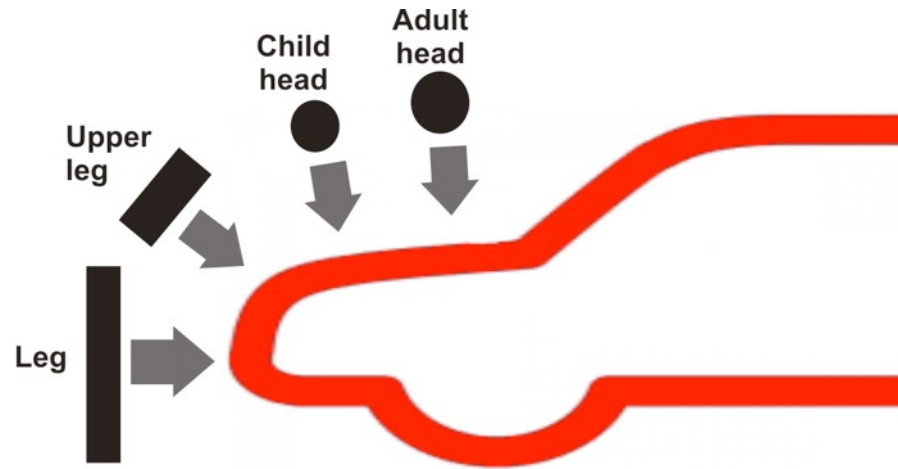
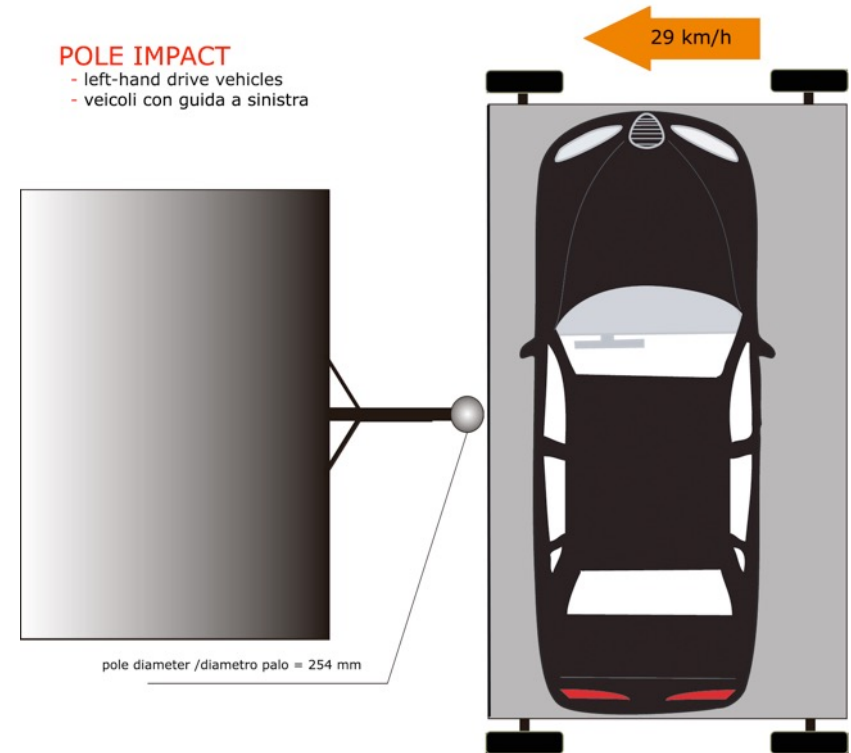


EN: **R** = Point corresponding to the driver's side (hip), valid for 95% of male adults.


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



Des tests : compilation des résultats


 FOR SAFER CARS
 www.euroncap.com

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

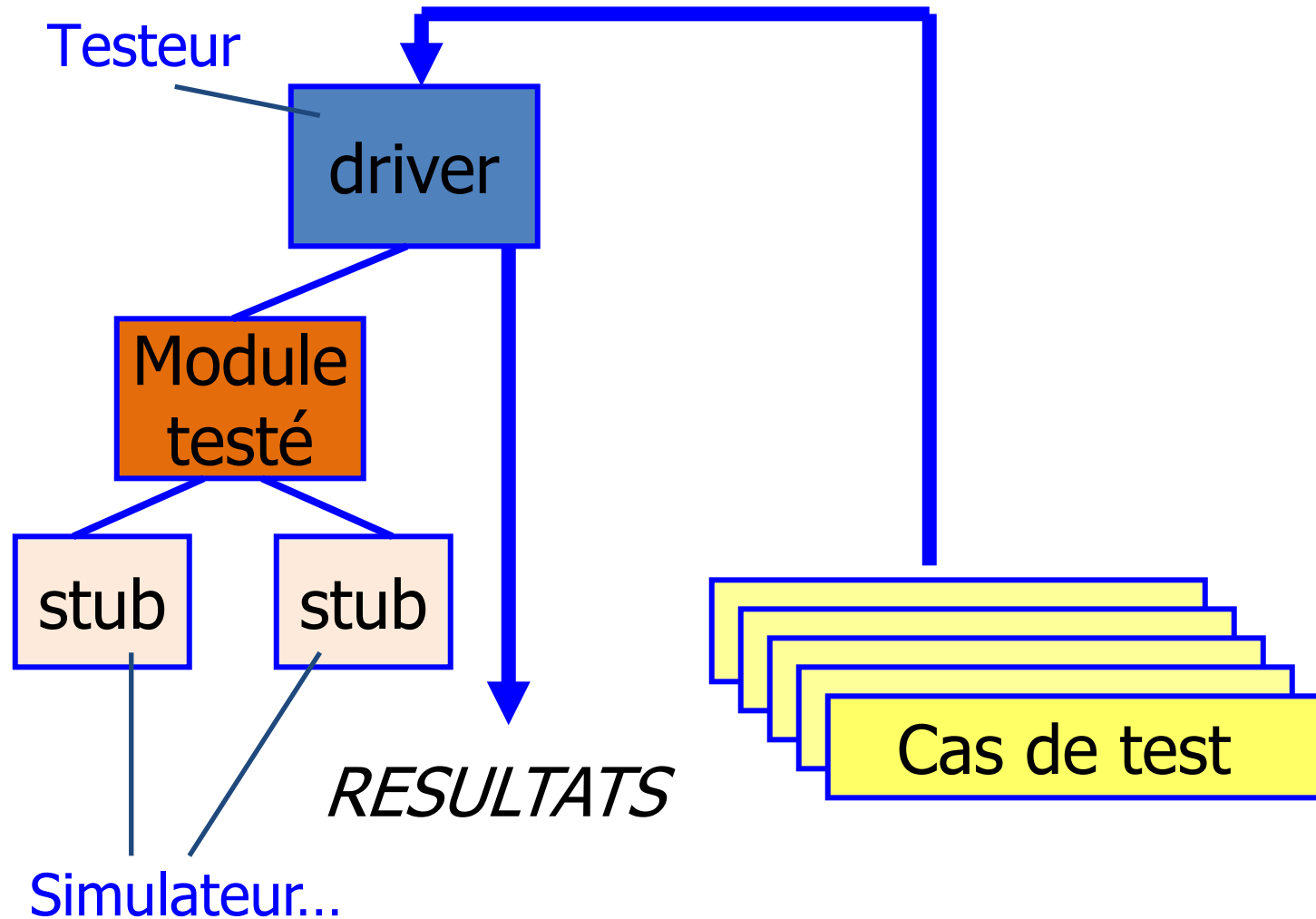
Types de test

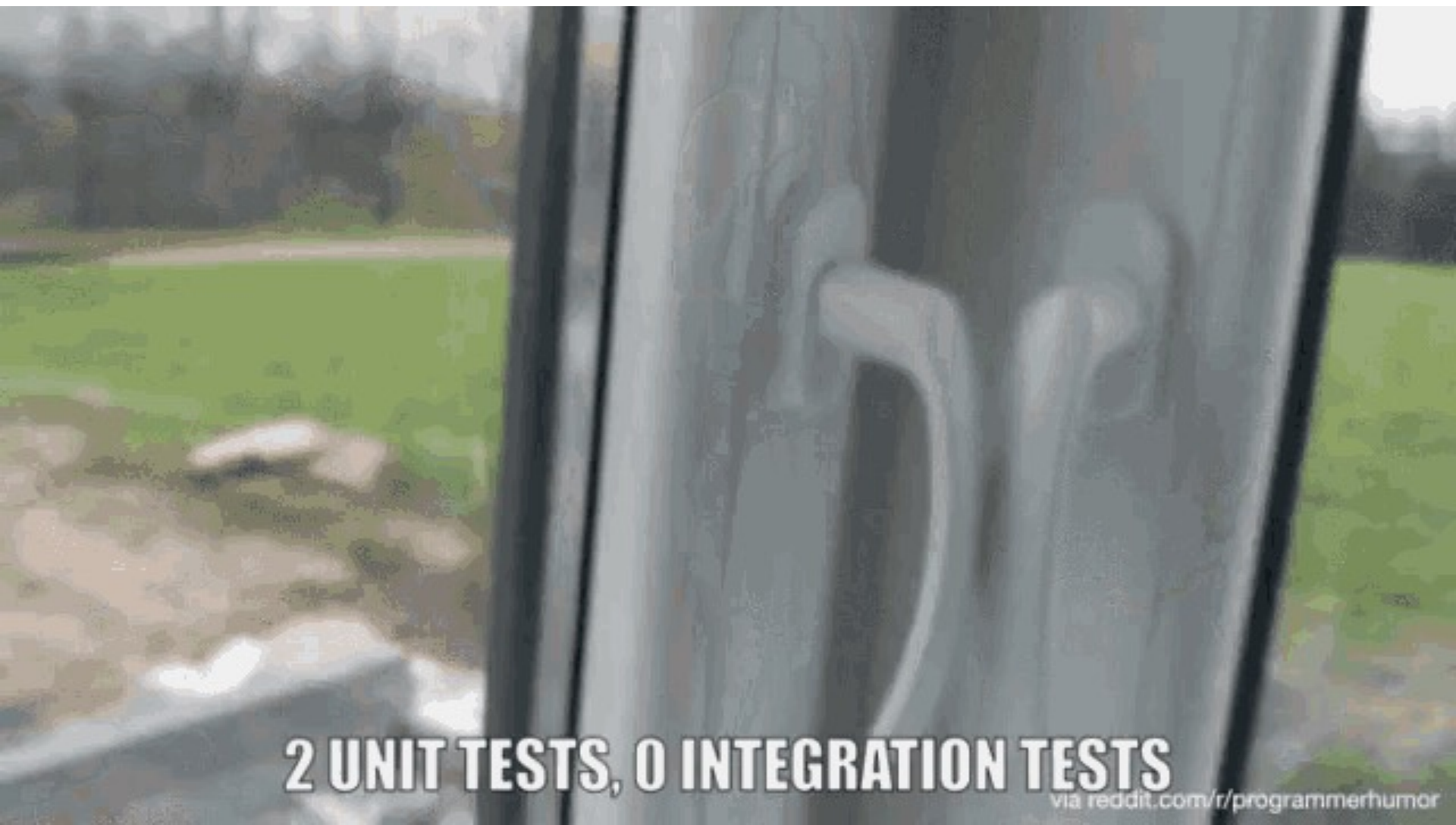
- Tests unitaires
- Tests d'intégration
- Tests fonctionnels (d'acceptation)
- Autre chose ?

Types of tests

- Unit Tests
- Integration Tests
- GUI Tests
- Non-regression Tests
- Coverage Tests
- Load Tests
- Stress Tests
- Performance Tests
- Scalability Tests
- Reliability Tests
- Volume Tests
- Usability Tests
- Security Tests
- Recovery Tests
- L10N/I18N Tests
- Accessibility Tests
- Installation/Configuration Tests
- Documentation Tests
- Platform testing
- Samples/Tutorials Testing
- Code inspections

Environnement du test unitaire





JUnit 5



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.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.api.Test;

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 différents types de base*
 - **Faire les « import » qui vont bien (ou laisser l'IDE le faire...)**

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 `@BeforeEach` et `@AfterEach` 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 `@BeforeEach` avant et la méthode `@AfterEach` après chacune des méthodes de test
 - Pour deux méthodes, exécution équivalente à :
 - `@BeforeEach-method ; @Test1-method(); @AfterEach-method();`
 - `@BeforeEach-method ; @Test2-method(); @AfterEach-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;

    @BeforeEach public void setUp() {
        f12CHF= new Money(12, "CHF");
        f14CHF= new Money(14, "CHF");
    }

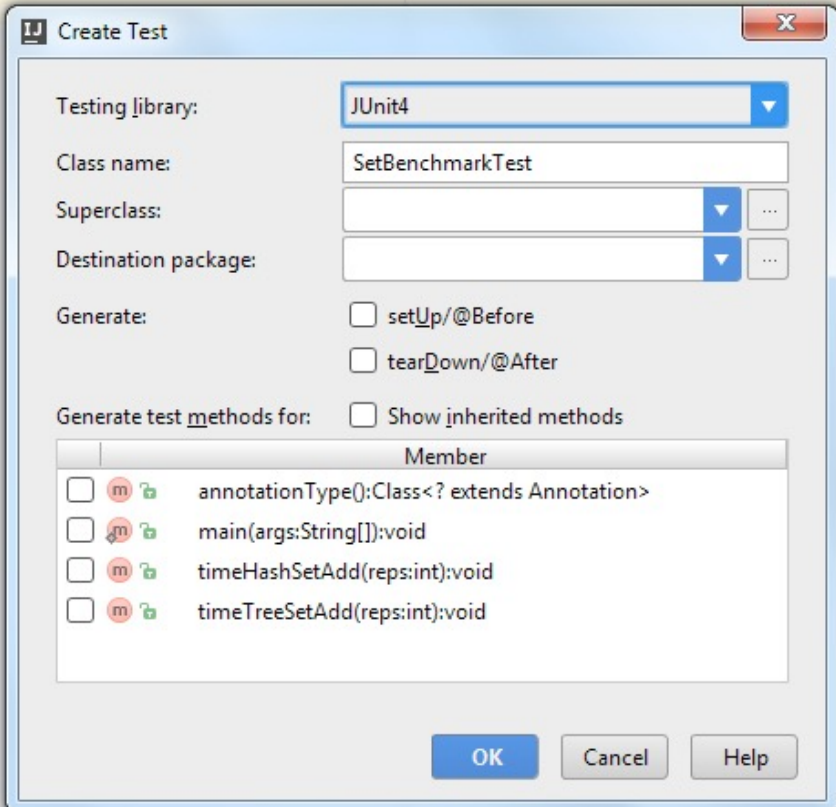
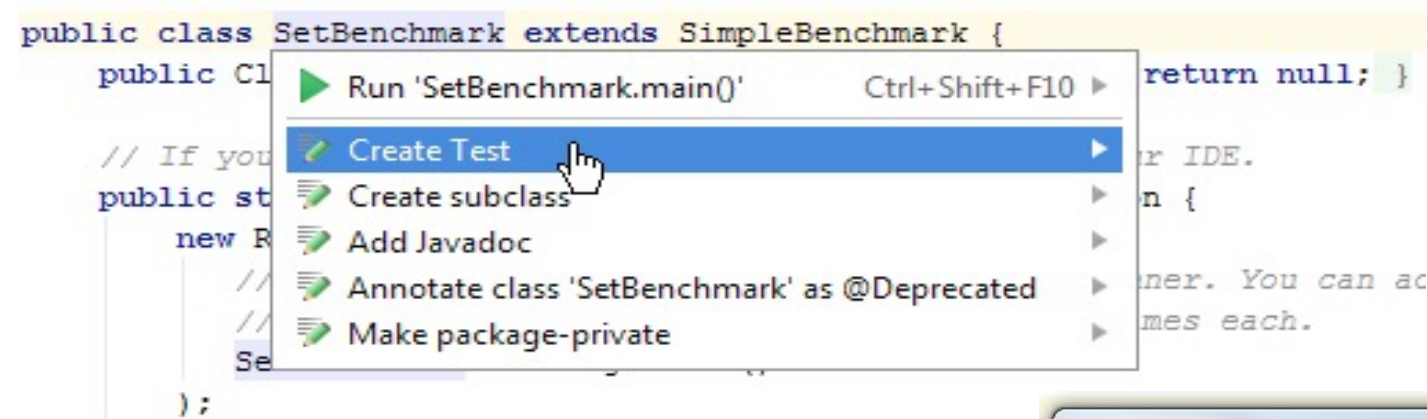
    @Test public void testEquals() {
        assertTrue(!f12CHF.equals(null));
        assertEquals(f12CHF, f12CHF);
        assertEquals(f12CHF, new Money(12, "CHF"));
        assertTrue(!f12CHF.equals(f14CHF));
    }

    @Test public void testSimpleAdd() {
        Money expected= new Money(26, "CHF");
        Money result= f12CHF.add(f14CHF);
        assertTrue(expected.equals(result));
    }
}
```

Exécution des tests

- Dans un IDE
- Plus tard, avec des outils automatisés
- 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

Done: 2 of 2 Failed: 1 (in 0.019 s)

Test	Time elapsed	Usage Delta	Usage Before	Usage After	Results
testIsOdd (OddCheckerTest)	0.004 s	88 Kb	3,824 Kb	3,912 Kb	Passed
testMain (OddCheckerTest)	0.008 s	292 Kb	4,001 Kb	4,293 Kb	Assertion

Tests Failed: 1 passed, 1 failed
Total time: 0.012 s

Run 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	false

1 difference Deleted Changed

Et après ?



Premier test : 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 ?
 - $\sqrt{4} \Rightarrow 2$

Ensuite ? Boundary conditions

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

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**

Que faire en TD ?



- **Setup de l'environnement (JUnit dans IntelliJ)**
- **Ecrire un premier test**
- **Déterminer les parties testables et critiques pour démontrer automatiquement que les éléments unitaires de la main de poker fonctionnent**
- **Grands débutants ? GO GO GO (dès que le TD 1 est terminé)**

