

# Projet de développement

## Eclipse : CDT / PDT V&V et tests unitaires

Philippe Collet

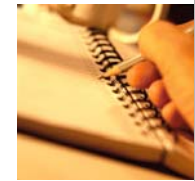
Licence 3 Informatique

2009-2010



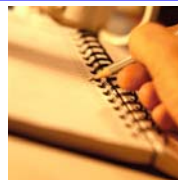
## Organisation

- Cours 1 : principes généraux - svn
- Cours 2 : Redmine et gestion de projet
- Cours 3 : Introduction à Eclipse
- Cours 4 : Eclipse C / PHP, V&V et tests unitaires en Java**
- Cours 5 : Tests C / PHP / Python
- Cours 6 : conclusion , questions...



## Plan

- Eclipse CDT
- Eclipse PDT
- Démo
- V&V
- Spécificités des tests OO
- Tests unitaires en Java avec JUnit
- Démo



## CDT : C/C++ Development Tooling

- Environnement dédié au développement en C et C++
- Projets, perspectives, vues, éditeurs **dédiés**
- Complétion, formatage, génération de documentation **dédiées**
- Refactoring succinct mais présent
  - Renommage, extraction de constantes et de fonctions
- Spécificités
  - Génération de makefile
  - **Nécessité de configuration du compilateur, du débogueur (puisque'on n'est pas en Java)**

- ❑ Environnement de développement dédié à PHP
- ❑ Basé sur le sous-projet « web tools »
  - Développé par Zend, la société leader des outils et solutions serveurs PHP
- ❑ Complétion, formatage, etc.
- ❑ Historique
  - A pris le dessus sur PHPEclipse (premier envt dédié à PHP sous Eclipse)

## Validation et Vérification

- ❑ 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 spécifications fonctionnelles définissent les intentions
  - Elles sont créées lors de la phase d'analyse des besoins
- ❑ La vérification du produit consiste à vérifier la conformité vis-à-vis de ces spécifications fonctionnelles
  - Revues, inspections, analyses, tests fonctionnels et structurels en boîte blanche
- ❑ La validation du produit consiste à vérifier par le donneur d'ordre la conformité vis-à-vis des besoins
  - Le plus souvent, tests fonctionnels en boîte noire
  - Théoriquement, la validation devrait être plutôt faite par les utilisateurs, sans tenir compte du cahier des charges
  - En pratique, la validation s'appuie sur le cahier des charges pour créer des tests d'acceptation...

## Techniques statiques

**Portent sur des documents (plutôt des programmes), sans exécuter le logiciel**

**Avantages**

- contrôle systématique valable pour toute exécution, applicables à tout document

**Inconvénients**

- Ne portent pas forcément sur le code réel
- Ne sont pas en situation réelle (interaction, environnement)
- Vérifications sommaires, sauf pour les preuves
- Ces preuves nécessitent des spécifications formelles et complètes, donc difficiles

## 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**

## 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 ?

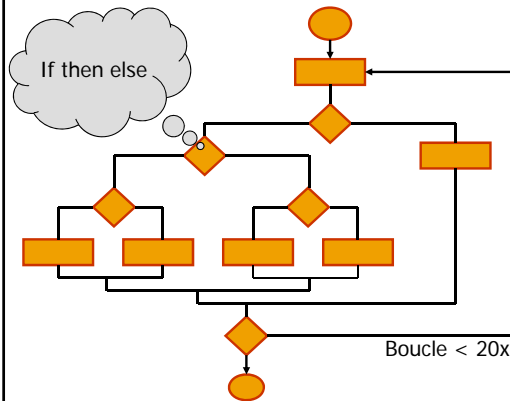
☞ **Les tests doivent mettre en évidence des erreurs !**

☞ **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

## Tests exhaustifs ?



Il y a  $5^{20}$  chemins possibles  
En exécutant 1 test par milliseconde, cela prendrait **3024** ans pour tester ce programme !

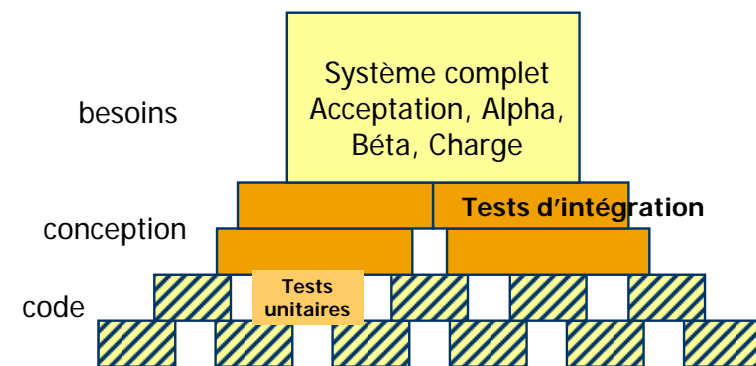
## Constituants d'un test

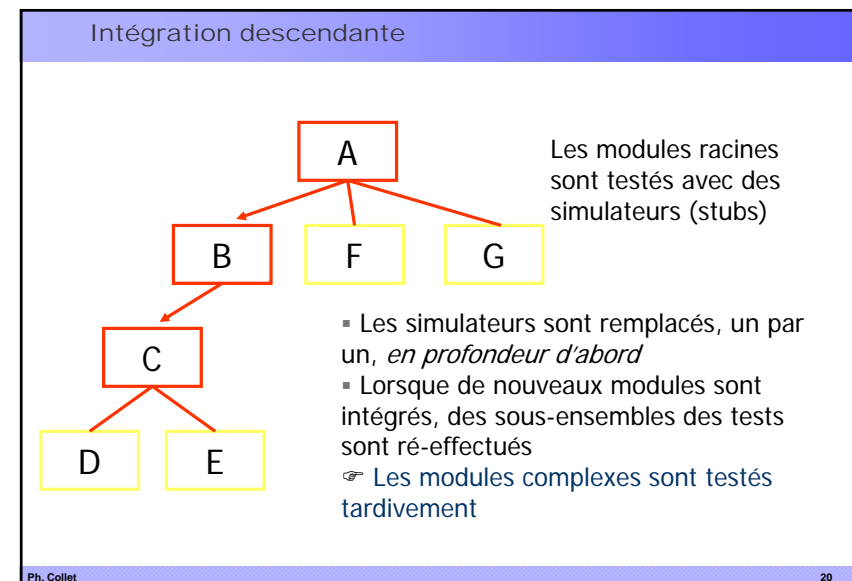
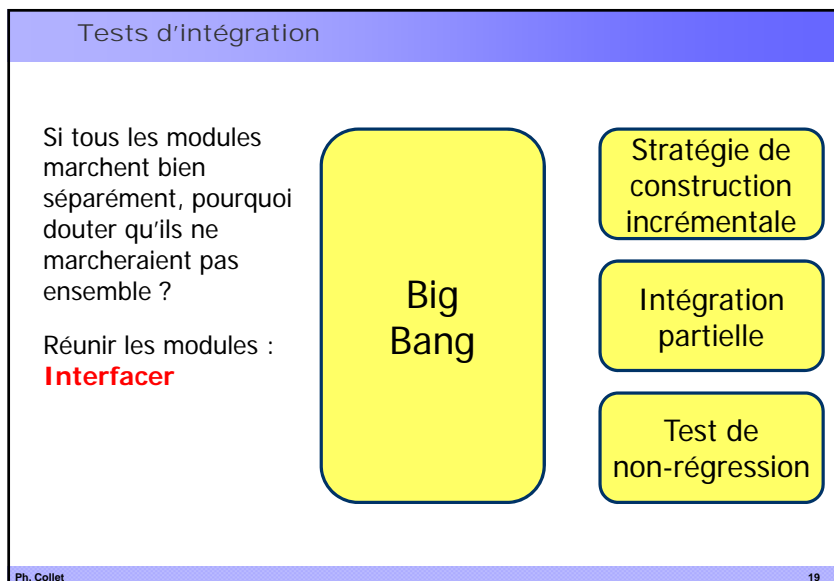
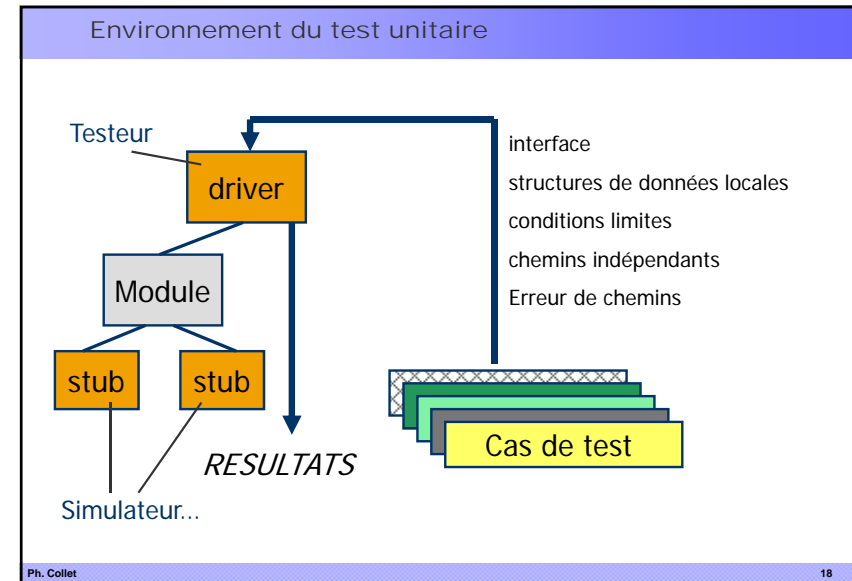
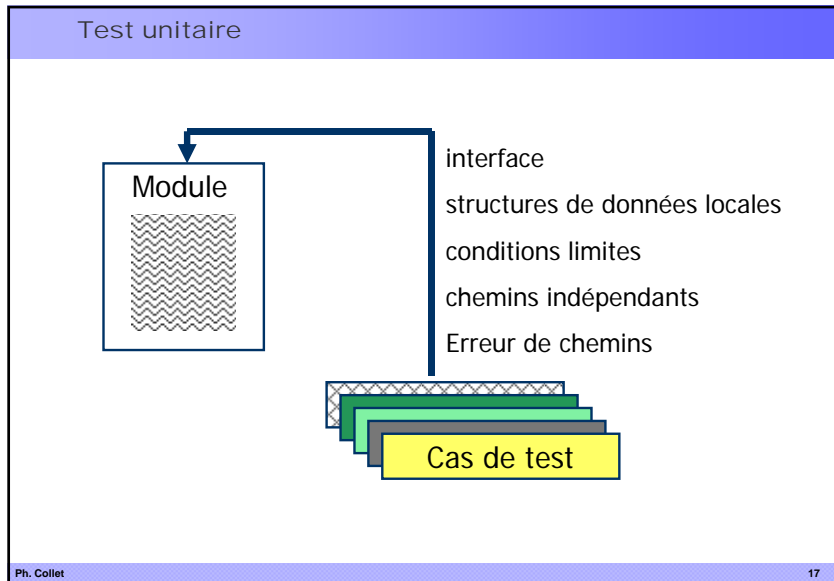
- Nom, objectif, commentaires, auteur
- Données : jeu de test
- Du code qui appelle des routines : cas de test
- Des oracles (vérifications de propriétés)
- Des traces, des résultats observables
- Un stockage de résultats : étalon
- Un compte-rendu, une synthèse...
  
- Coût moyen : autant que le programme

## Test vs. Essai vs. Débogage

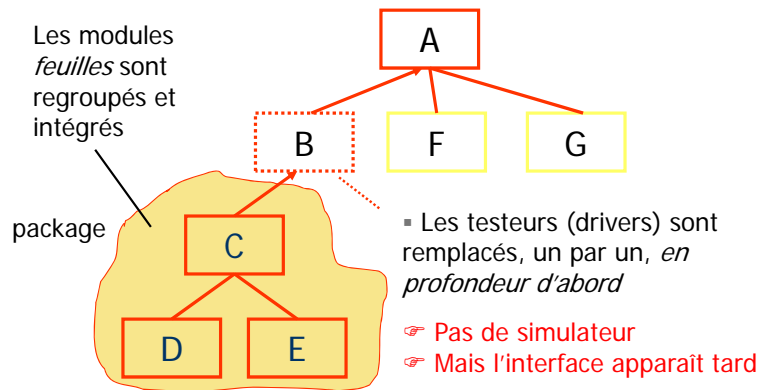
- On converse les données de test
  - Le coût du test est amorti
  - Car un test doit être reproductible
- Le test est différent d'un essai de mise au point
  
- Le débogage est une enquête
  - Difficilement reproductible
  - Qui cherche à expliquer un problème

## Les stratégies de test

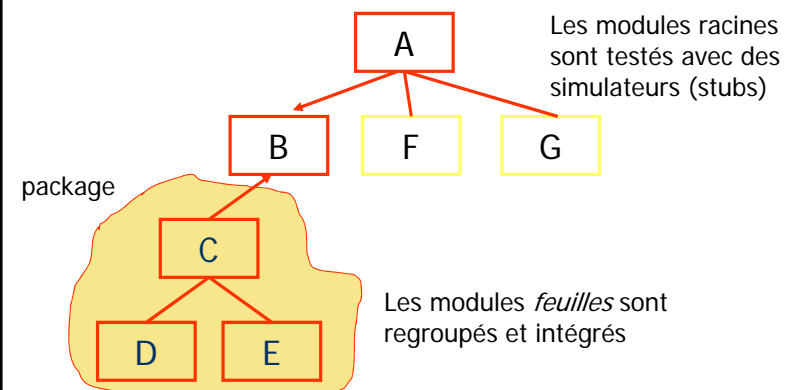




## Intégration ascendante



## Intégration en *Sandwich*



## Tests fonctionnels en boîte noire

### □ Principes

- S'appuient sur des spécifications externes
- Partitionnent les données à tester par classes d'équivalence
  - ◆ Une valeur attendue dans  $1..10$  donne  $[1..10]$ ,  $< 1$  et  $> 10$
- Ajoutent des valeurs « pertinentes », liées à l'expérience du testeur
  - ◆ Tests aux bornes : sur les bornes pour l'acceptation, juste au delà des bornes pour des refus

## Exemple : la recherche binaire

```
Table_binaire <elem -> comparable>
...
lower() : integer
...
chercher(key : elem) : integer
// si l'élément de clé key est dans la table, rend son indice,
sinon rend lower-1
...
```

### □ Classes d'équivalence ?

- Données conformes aux prérequis (?)
- Données non-conformes aux prérequis
- Cas où l'élément recherché existe dans la table
- Cas où l'élément recherché n'existe pas dans la table

## Test : recherche binaire

### Deuxième regroupement plus pertinent

- Table ordonnée, avec élément recherché présent
- Table ordonnée, avec élément recherché absent
- Table non ordonnée, avec élément recherché présent
- Table non ordonnée, avec élément recherché absent

### Ajout de cas limites et *intuitifs*

- Table à un seul élément
- Table à un nombre impair d'éléments (*boite grise*)
- Table à un nombre pair d'éléments (*boite grise*)
- Table où l'élément recherché est le premier de la table
- Table où l'élément recherché est le dernier de la table

## Test : recherche binaire

### Classes d'équivalence

- 1 seul élément, clé présente
- 1 seul élément, clé absente
- Nb pair d'éléments, clé absente
- Nb pair d'éléments, clé en première position
- ...

### Tests en boîte noire résultants

- Nb pair d'éléments, clé ni en première, ni en dernière position
- Nb impair d'éléments, clé absente
- Nb impair d'éléments, clé en première position
- ...

## Spécificités des tests orientés objets

## Tester des applications OO

### Commence par l'évaluation de la correction et de la cohérence de l'analyse et de la conception OO

### Les stratégies de test changent :

- Le concept de *module* s'élargit à cause de l'encapsulation
- L'intégration se concentre sur les classes et leur exécution dans un *thread* ou dans le contexte d'un *cas d'utilisation*
- La validation se fait à l'aide d'approches conventionnelles en boîte noire

### La conception des cas de test repose sur des méthodes classiques mais introduit aussi des spécificités

## 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**

## Intégration OO

- ❑ **Tests basés sur les *threads***
  - Intègrent l'ensemble des classes nécessaires pour répondre à une entrée ou un événement
- ❑ **Tests basés sur les cas d'utilisation**
  - Intègrent l'ensemble des classes nécessaires pour répondre à un cas d'utilisation
  - Démontrent que chaque cas d'utilisation est exécutable par le système
    - ◆ avec ou sans l'interface utilisateur
  - Se focalisent sur ce que fait l'utilisateur, pas sur ce que fait le système

## Jeux de test par classe

- ❑ **Pour chaque classe dans le système**
  - Au moins un cas de test pour
    - ◆ le comportement normal de chaque méthode publique
    - ◆ les exceptions de chaque méthode publique
    - ◆ chaque transition d'état de la classe (...de l'instance)
    - ◆ les transitions d'état invalides de la classe (...de l'instance)
  - Des cas de test en boîte blanche pour les méthodes les plus importantes

## Passage des tests (exécution) : méthode pragmatique

- ❑ **Développer des lanceurs automatiques de test (test drivers) pour chaque classe**
  - La méthode `main` de la classe lance le test de la classe
  - Des méthodes de la classe définissent des (ensemble de) tests
- ❑ **Pour les tests de cas d'utilisation**
  - Développer des classes de test qui réalise les cas d'utilisation
  - Tester les cas d'utilisation à la main avec l'IHM
- ❑ **Le lanceur de tests du système appelle tous les lanceurs des classes et toutes les classes qui testent les cas d'utilisation**



## Implémentation des tests

### □ Le code du lanceur de tests doit :

- Créer l'état spécifique du système décrit dans le cas de test
- Appeler la méthode à tester
- Vérifier si le résultat attendu est produit
- Écrire le résultat du test dans un fichier / sur la console

```
Classe: ResourcePool
méthode: addEngineer
cas de test: No. 142 - test de l'exception de la méthode
résultat: échec
```

- Compter le nombre de tests OK / NOK pour produire un résumé des résultats à la fin

# JUnit

## JUnit v4

[www.junit.org](http://www.junit.org)



## Version « automatique »: 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

### □ Une classe Money

```
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));
    }
}
```

## 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-4.5.jar dans le CLASSPATH (compilation et exécution)
- C'est tout...

## 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)
...
```

## 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);
}
```

## 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 utilisés 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 ) );
```

## 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 nettoyer un environnement (fermeture de fichier, effet de bord de manière générale)

## Suite : organisation des tests

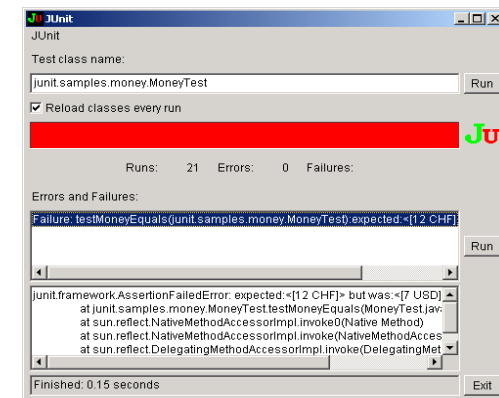
- ❑ Des classes de test peuvent être organisés en hiérarchies de « Suite »
  - Elles appellent automatiquement toutes les méthodes @Test de chaque classe de test
  - Une « Suite » est formée de classes de test ou de suites de test
  - Les tests peuvent être assemblés dans une hiérarchie de niveau quelconque, tous les tests seront toujours appelés automatiquement et uniformément en une seule passe.

```
@RunWith(value=Suite.class)
@SuiteClasses(value={CalculatorTest.class, AnotherTest.class})
public class AllTests {
    ...
}
```

- Une « suite » peut avoir ses propres méthodes annotées @BeforeClass et @AfterClass, qui seront évaluées 1 seule fois avant et après l'exécution de toute la suite

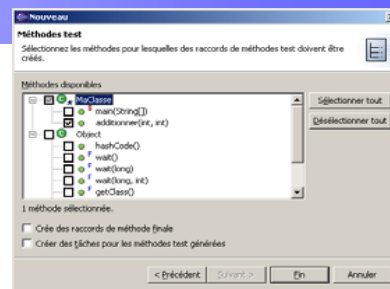
## JUnit : TestRunner

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

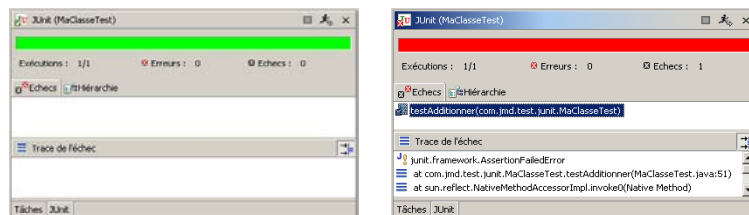


## JUnit dans Eclipse

- ❑ Assistants pour :
  - Créer des cas de test



- ❑ TestRunner intégré à l'IDE



## Aspects méthodologiques

- ❑ Coder/tester, coder/tester...
  - aussi souvent que le compilateur !
- ❑ lancer les tests aussi souvent que possible
  - Ecrire les tests qui ont le meilleur retour sur investissement !
  - Approche *Extreme Programming*
- ❑ Quand on ajoute des fonctionnalités, on écrit d'abord les tests
  - *Test-Driven Development...*
- ❑ 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

## TDD: Test-Driven Development

### ❑ Qu'est-ce vraiment ?

- Livrer les fonctionnalités dont le logiciel a réellement besoin, pas celles que le programmeur croit devoir fournir (une évidence *a priori*)

### ❑ Comment ?

- Ecrire du code client comme si le code à développer existait déjà et avait été conçu en tout point pour nous faciliter la vie !
- Le code client, ce sont les tests : Ecrire un test.
- Le regarder échouer (il n'y a pas de code fonctionnel pour l'instant).
- Ecrire du code pour que ça compile
- Modifier le code au fur et à mesure pour que le test passe (et les tests précédents aussi)
- *Refactorer* le tout (code, test) pour rendre les choses toujours plus simples pour soi
- **Rincer et répéter**

### ❑ + Principe d'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 !

## 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

## Right-BICEP

### ❑ 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

## Right-BICEP

### ❑ 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.).