Université Nice SOPHIA ANTIPOLIS

Université Nice SOPHIA ANTIPOLIS

Tests orientés objets
Junit, Mockito
Rappel ?

Philippe Collet

Master 1 IFI

2014-2015

http://deptinfo.unice.fr/twiki/bin/view/Minfo/GenieLog1415

Plan

- Retour sur V&V
- JUnit 4
- Mockito

Principes de V&V (rappel ?)

- 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

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

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…

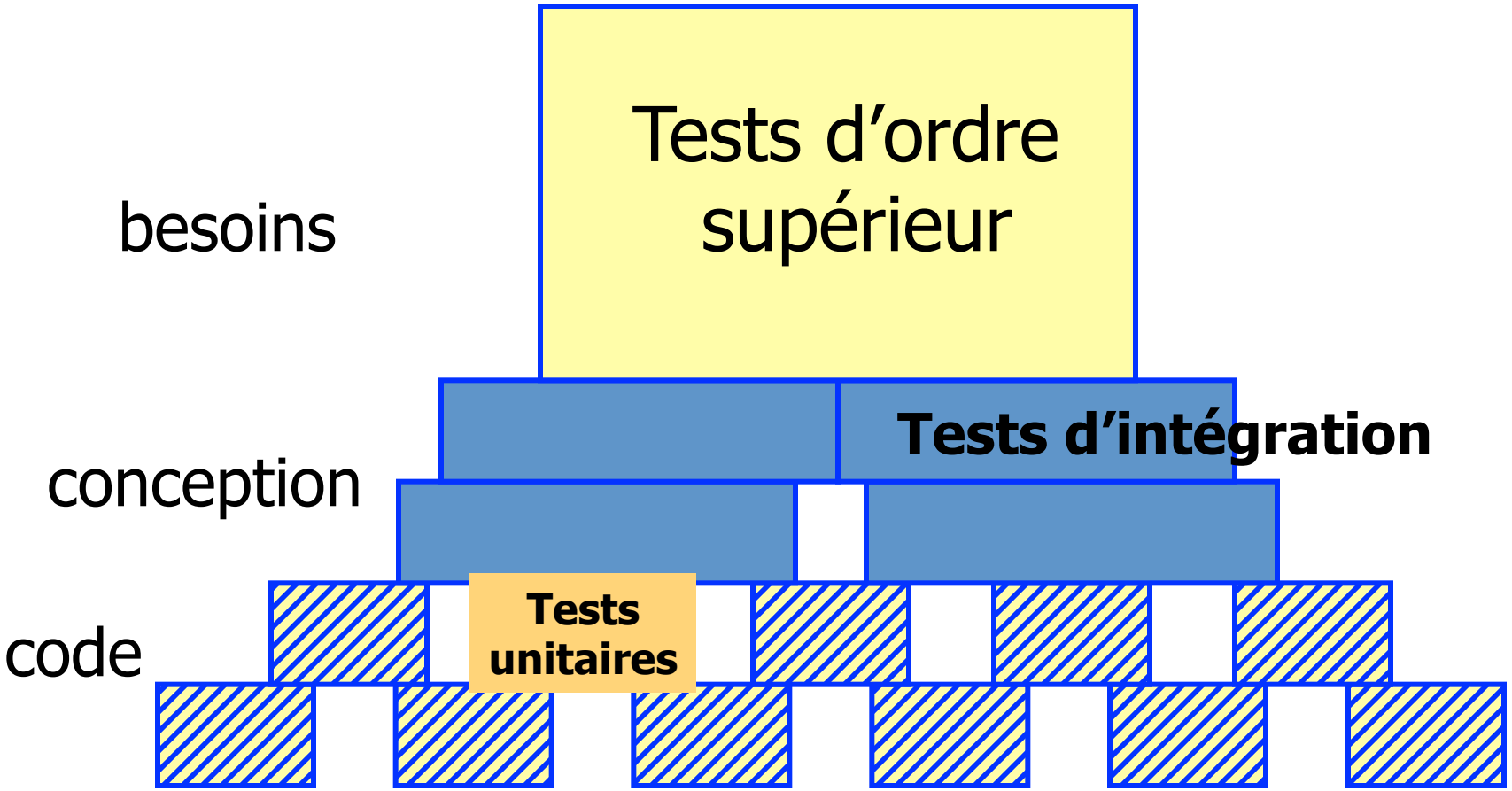
- Coût moyen : autant que le programme

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



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

Université Nice SOPHIA ANTIPOLIS

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**

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));
}
```

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)
…
```

- *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 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 ) );
```

- *Pas d'équivalent en JUnit 3*

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 de manière générale)

Suite : organisation des tests

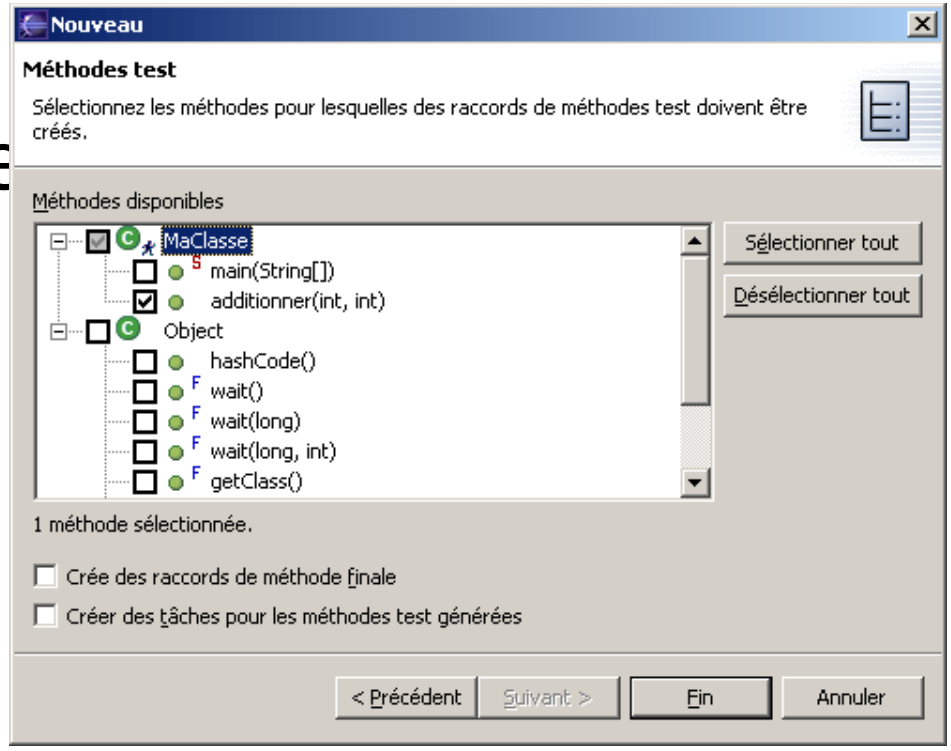
- Des classes de test peuvent être organisés en hiérarchies de « Suite »
  - Elles appellent automatiquement toutes les méthodes @Test de chaque classe de test
  - Une « Suite » est formée de classes de test ou de suites de test
  - Les tests peuvent être assemblés dans une hiérarchie de niveau quelconque, 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 {
   ...
}
```

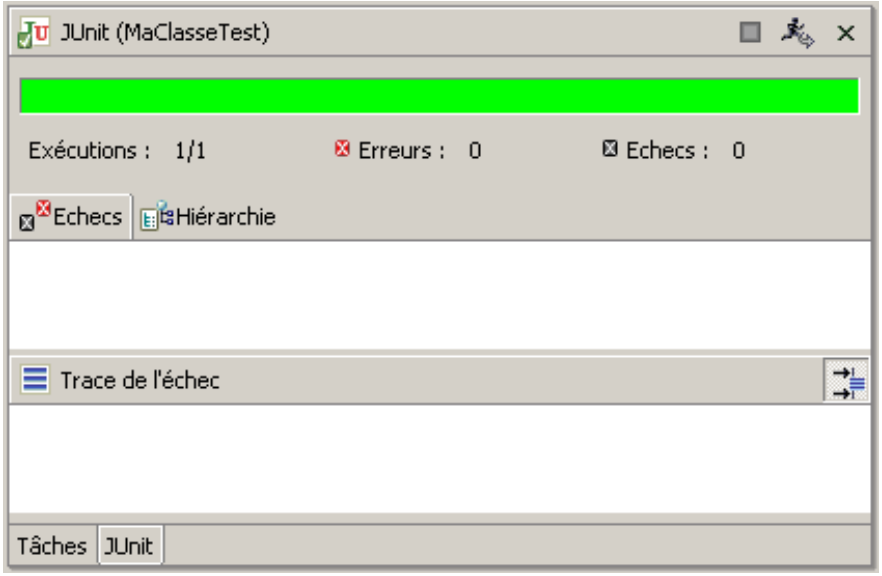
  - **Une « suite » peut avoir ses propres méthodes annotées @BeforeClass et @AfterClass, qui seront évaluées 1 seule fois avant et après l'exécution de toute la suite**

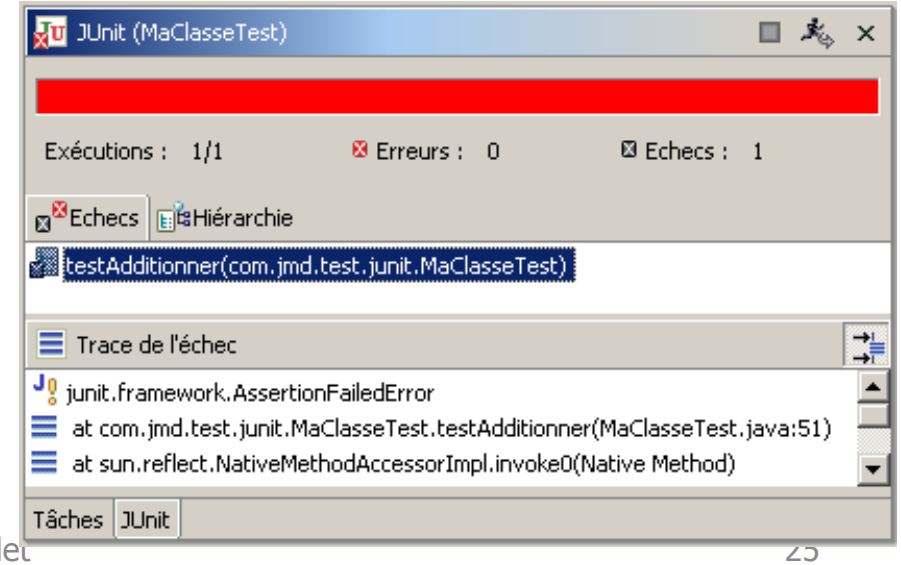
JUnit dans Eclipse

- Assistants pour :
  - Créer des cas de test



- TestRunner intégré à l'IDE





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-BICEP : right

- **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 : boundary

- **B : Boundary conditions**
  - identifier les conditions aux limites de la spécification
  - que se passe-t-il lorsque les données sont
    - anachroniques ex. : !*W@\/"
    - non correctement formattées 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

Right-BICEP : boundary

- **Pour établir correctement les « bornes »**
- **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…

Right-BICEP

- **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.).

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*
- **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**

De l'eXtreme Programming au Test-Driven Development

- **Qu'est-ce vraiment ?**
  - Livrer les fonctionnalités dont le logiciel a réellement besoin, pas celles que le programmeur croît 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 !
  - Modifier le code pour compiler les tests
  - Implémenter le code au fur et à mesure
  - *Refactorer* le tout pour rendre les choses toujours plus simples pour soi
  - Ecrire un test, ecrire du code, refactorer…
- **+ 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 !

Université Nice SOPHIA ANTIPOLIS

Université Nice SOPHIA ANTIPOLIS

Test & Objet Mock

Définition

- Mock = Objet factice
- les mocks (ou Mock object) sont des objets simulés qui reproduisent le comportement d'objets réels de manière contrôlée
- On teste ainsi le comportement d'autres objets, réels, mais liés à un objet inaccessible ou non implémenté
- Cet objet est remplacé par un mock

Définition(s)

- dummy (pantin, factice) : objets vides qui n'ont pas de fonctionnalités implémentées
- stub (bouchon) : classes qui renvoient en dur une valeur pour une méthode invoquée
- fake (substitut, simulateur) : implémentation partielle qui renvoit toujours les mêmes réponses selon les paramètres fournis
- spy (espion) : classe qui vérifie l'utilisation qui en est faite après l'exécution
- mock (factice) : classes qui agissent comme un stub et un spy

Exemple d'utilisation

- Comportement non déterministe (l'heure, un senseur)
- Initialisation longue (BD)
- Classe pas encore implémentée ou implémentation changeante, en cours
- Etats complexes difficiles à reproduire dans les tests (erreur réseau, exception sur fichiers)
- Pour tester, il faudrait ajouter des attribut ou des méthodes

Principe

- Un mock a la même interface que l'objet qu'il simule
- L'objet client ignore s'il interagit avec un objet réel ou un objet simulé
- La plupart des frameworks de mock permettent
  - De spécifier quelles méthodes vont être appelées, avec quels paramètres et dans quel ordre
  - De spécifier les les valeurs retournées par le mock

Université Nice SOPHIA ANTIPOLIS

Mockito

http://code.google.com/p/mockito/

Eléments de cours de M. Nebut lifl.fr

Mockito

- Générateur automatique de doublures
- Léger
  - Focalisation sur le comportement recherché et la vérification après l'exécution
- Simple
  - Un seul type de mock
  - Une seule manière de les créer

Principes

- Fonctionnement en mode espion (spy) :
  - Création des mocks
    - méthode mock ou annotation @mock
  - Description de leur comportement
    - Méthode when
  - Mémorisation à l'exécution des interactions
    - Utilisation du mock dans un code qui teste un comportement spécifique
  - Interrogation, à la fin du test, des mocks pour savoir comme ils ont été utilisés
    - Méthode verify

import static org.mockito.Mockito.*

Création

- Par une interface ou une classe (utilisation de .class)
  - UneInterface mockSansNom = mock(UneInterface.class);
  - UneInterface mockAvecNom =
    mock(UneInterface.class, "ceMock");
  - @Mock UneInterface ceMock;
- Comportements par défaut
  - assertEquals("ceMock", monMock.toString());
  - assertEquals("type numerique : 0 ", 0,
    monMock.retourneUnEntier());
  - assertEquals("type booleéen : false",
    false, monMock.retourneUnBooleen());
  - assertEquals("type collection : vide",
    0, monMock.retourneUneList().size());

Stubbing

- Pour remplacer le comportement par défaut des méthodes
- Deux possibilités
  - Méthode qui a un type de retour :
    - when + thenReturn ;
    - when + thenThrow ;
  - Méthode de type void :
    - doThrow + when ;

Stubbing
retour d'une valeur unique

```
// stubbing
when(monMock.retourneUnEntier()).thenReturn(3);

// description avec JUnit
assertEquals("une premiere fois 3", 3, monMock.retourneUnEntier());
assertEquals("une deuxieme fois 3", 3, monMock.retourneUnEntier());
```

Stubbing
valeurs de retour consécutives

```
// stubbing
when(monMock.retourneUnEntier()).thenReturn(3, 4, 5);

// description avec JUnit
assertEquals("une premiere fois : 3", 3, monMock.retourneUnEntier());
assertEquals("une deuxieme fois : 4", 4, monMock.retourneUnEntier());
assertEquals("une troisieme fois : 5", 5, monMock.retourneUnEntier());

when(monMock.retourneUnEntier()).thenReturn(3, 4);
// raccourci pour      .thenReturn(3).thenReturn(4);
```

Stubbing
Levée d'exceptions

```
public int retourneUnEntierOuLeveUneExc() throws BidonException;
// stubbing
when(monMock.retourneUnEntierOuLeveUneExc()).thenReturn(3)
                    .thenThrow(new BidonException());

// description avec JUnit
assertEquals("1er appel : retour 3",
                3, monMock.retourneUnEntierOuLeveUneExc());

try {
    monMock.retourneUnEntierOuLeveUneExc(); fail();
} catch (BidonException e) {
    assertTrue("2nd appel : exception", true);
}
```

Levée d'exception + méthode void = doThrow

Remarques

- Les méthodes equals() et hashcode() ne peuvent pas être *stubbées*
- Un comportement de mock non exécuté ne provoque pas d'erreur
- Il faut utiliser *verify*
  - Quelles méthodes ont été appelées sur un
  - Combiendefois, avec quels paramètres, dans quel ordre
- Une exception est levée si la vérification échoue, le test échouera aussi

Verify

- Méthode appelée une seule fois :
  - verify(monMock).retourneUnBooleen();
  - verify(monMock, times(1)).retourneUnBooleen();
- Méthode appelée au moins/au plus une fois:
  - verify(monMock, atLeastOnce()).retourneUnBooleen();
  - verify(monMock, atMost(1)).retourneUnBooleen();
- Méthode jamais appelée :
  - verify(monMock, never()).retourneUnBooleen();
- Avec des paramètres spécifiques :
  - verify(monMock).retourneUnEntierBis(4, 2);

Verify

- import org.mockito.InOrder;
- Pour vérifier que l'appel (4,2) est effectué avant l'appel (5,3) :
  - InOrder ordre = inOrder(monMock);
  - ordre.verify(monMock).retourneUnEntierBis(4, 2);
  - ordre.verify(monMock).retourneUnEntierBis(5, 3);
- Avec plusieurs mocks :
  - InOrder ordre = inOrder(mock1, mock2);
  - ordre.verify(mock1).foo();
  - ordre.verify(mock2).bar();

Espionner un objet classique

- Pour espionner autre chose qu'un objet mock (un objet « réel »):
  - Obtenir un objet espionné par la méthode spy :

```
List list = new LinkedList();
List spy = spy(list);
```

  - Appeler des méthodes « normales » sur le spy :

```
spy.add("one");
spy.add("two");
```

  - Vérifier les appels à la fin :

```
verify(spy).add("one");
verify(spy).add("two");
```

Matchers

- Mockito vérifie les valeurs en arguments en utilisant equals()
- Assez souvent, on veut spécifier un appel dans un « when » ou un « verify » sans que les valeurs des paramètres aient vraiment d'importance

- On utilise pour cela des *Matchers (import org.mockito.Matchers)*

  – when(mockedList.get(**anyInt**())).thenReturn("element");

  – verify(mockedList).get(**anyInt**());

Matchers

- Les Matchers très souvent utilisés :
  - any() , static <T> T anyObject()
  - anyBoolean(), anyDouble(), anyFloat() , anyInt(), anyString()…
  - anyList(), anyMap(), anyCollection(), anyCollectionOf(java.lang.Class<T> clazz) ,anySet(), <T> java.util.Set<T> anySetOf(java.lang.Class<T> clazz)
  - …

- Attention ! Si on utilise des matchers, tous les arguments doivent être des matchers :
  - verify(mock).someMethod(anyInt(), anyString(), "third argument");
  - n'est pas correct !

Alternatives à Mockito

- EasyMock, Jmock
- Tous basés sur expect-run-verify