Plan du cours

- Définitions autour du test
  - Métriques de qualité/fiabilité logicielle
  - Positionnement du test dans le cycle de vie du logiciel
  - Test et méthodes agiles
  - Qu'est-ce que le test ?
- Processus de test simplifié
- Comment choisir les scénarios de test à jouer ?
  - Les méthodes de test Boîte Noire
  - Les méthodes de test Boîte Blanche
- Tests unitaires avec JUnit et EclEmma
  - Présentation de JUnit
  - Couverture de test avec EclEmma
- Utilisation de doublures (mocks) pour les tests unitaires avant intégration

Junit, un outil pour le TDD

JUnit

- Junit
  - plateforme de tests unitaires pour Java (déclinée pour de nombreux autres langages) écrite par Erich Gamma and Kent Beck
  - Dédié au test « boite blanche » de type unitaire ou intégration
  - Intégré à Eclipse et Maven
  - Version actuelle 4.x basée sur l'utilisation des annotation Java 5
  - Disponible à http://www.junit.org/
- Junit propose
  - D'écrire facilement des tests unitaires
  - D'utiliser des assertions pour exprimer les oracles
  - Le lancement automatique de suites de test
  - Un formatage du diagnostic

Tests en JUnit

JUnit

- Les tests sont regroupés selon leur application aux méthodes d'une classe donnée
- Chaque test vérifie le comportement d'une méthode sous certaines conditions
  - Vérification des comportements corrects
  - Vérification des comportements incorrects (valeurs incohérentes des paramètres, ...)
  - Vérification des levées d'exceptions
- Écrits par le programmeur de la classe
- Surtout utiles pour tester qu'une classe reste valide après des modifications du code
- Les tests sont regroupés sous forme de « suite de tests » exécutés par un « Suite runner »

Test par Assertions

JUnit

- Une assertion = une confrontation avec l'oracle
- Méthodes statiques de la classe `org.junit.Assert`
  - `assertEquals`
  - `assertTrue`
  - `assertFalse`
  - `assertNull`
  - `assertNotNull`
  - `assertSame`
  - `assertNotSame`
- Paramètres d'invocation
  - `expected, actual`
  - `expected, actual, delta`
  - condition
  - ... + message

Méthode de test

- Méthode qui exécute un test unitaire
- Convention de nommage *test[méthode à tester]()*
  - Mais aucune obligation (nom quelconque possible)
- Utilise l'annotation **`@Test`**
- Publique, sans paramètre, type de retour `void`

```
@Test
public void testSetMontant(){
  compte.setMontant(100000.0);
  assertEquals(compte.getMontant(), 100000.0);
}
```

- L'annotation **`@Ignore`** permet d'ignorer un test

Levée d'exception et échec

- Le test d'une levée d'exception s'écrit comme suit

```
@Test(expected=ArrayIndexOutOfBoundsException.class)
public void testElementAt(){
  Vector<String> v = new Vector<String>();
  String s = v.elementAt(2);
}
```

- Une méthode peut provoquer l'échec d'un test

```
@Test
public void testSetMontant() {
  if (compte == null)
    fail("Erreur d'initialisation du test");
  compte.setMontant(100000.0);
  assertEquals(compte.getMontant(), 100000.0);
}
```


Méthodes d'initialisation et de finalisation

- Méthodes d'initialisation utilisée avant chaque test
  - setUp() est une convention de nommage
  - L'annotation **@Before** est utilisée

```
@Before
public void setUp(){
  compte = new Compte();
}
```

- Méthodes de finalisation utilisée après chaque test
  - tearDown() est une convention de nommage
  - L'annotation **@After** est utilisée

```
@After
public void tearDown(){
  declaration = null;
}
```

- Dans les 2 cas, possibilité d'annoter plusieurs méthodes
  - Ordre d'exécution indéterminé

Méthodes d'initialisation et de finalisation

- Méthodesd'initialisation globale
  - Annotée **`@BeforeClass`**
  - Publique et statique
  - Exécutée une seule fois avant la première méthode de test
- Méthode de finalisation globale
  - Annotée **`@AfterClass`**
  - Publique et statique
  - Exécutée une seule fois après la dernière méthode de test
- Dans les 2 cas, une seule méthode par annotation

JUnit

Tests unitaires fonctionnels et structurels

- Outils utilisé
  - Eclipse
  - JUnit
  - EclEmma

- *Exemple*: fonction retournant la somme de 2 entiers modulo 20 000

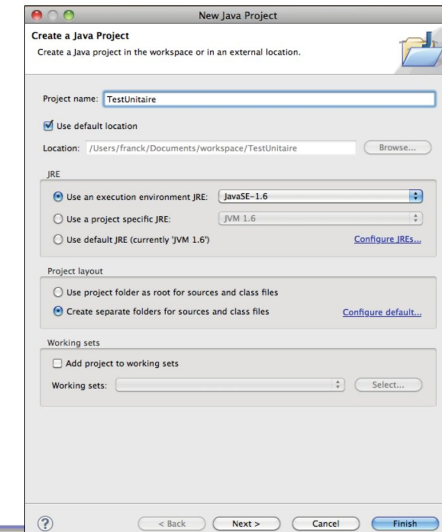
```
fun (x:int, y:int) : int =
        if (x==500 and y==600) then x-y (bug 1)
                                 else x+y (bug 2)
```

  - Avec l'approche fonctionnelle (boîte noire),
    - le bug 1 est difficile à détecter, le bug 2 est facile à détecter
  - Avec l'approche structurelle (boîte blanche),
    - le bug 1 est facile à détecter, le bug 2 est difficile à détecter

JUnit

Tests unitaires fonctionnels et structurels

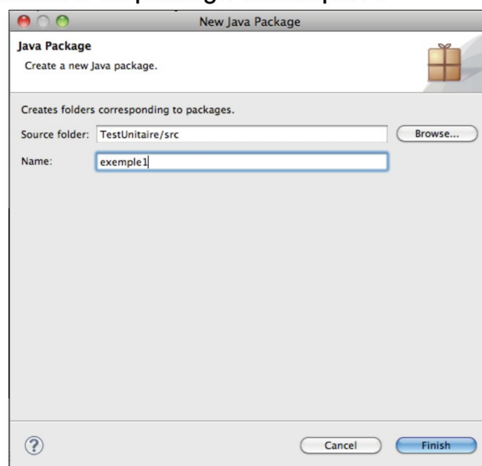
- Création d'un projet Eclipse Java → TestUnitaire



JUnit

Tests unitaires fonctionnels et structurels

- Dans le projet, création d'un package « exemple1 »

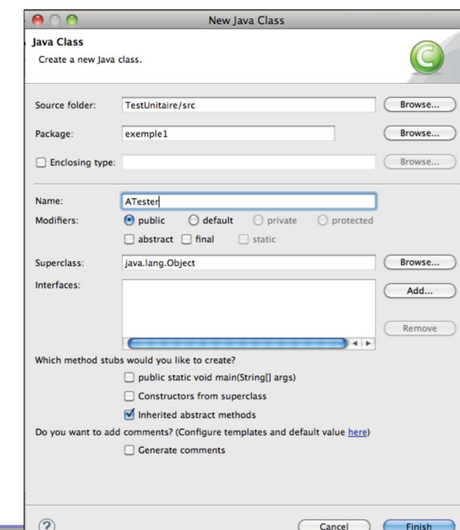


- Puis d'un package exemple1.test

JUnit

Tests unitaires fonctionnels et structurels

- Dans le package « exemple1 », création d'une classe « ATester »



Tests unitaires fonctionnels et structurels

JUnit

- Enfin écriture du code de l' opération fun

```
fun (x:int, y:int) : int =
        if (x==500 and y==600) then x-y (bug 1)
                                else x+y (bug 2)
```

```java
package exemple1;

public class ATester {
    public int f(int x, int y){
        if(x==500 && y==600)
            return y-x;
        else
            return x+y;
    }
}
```

Tests unitaires fonctionnels et structurels

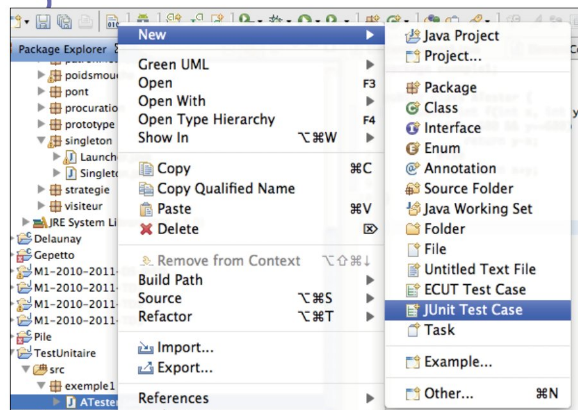
JUnit

- On commence par le test fonctionnel
  - *La fonction doit retourner la somme de 2 entiers modulo 20000*
  - Test 1 - DT : {x=10, y=20}, prédiction de l' oracle : 30
  - Test 2 - DT : {x=10000, y=10050}, prédiction de l' oracle : 50

Tests unitaires fonctionnels et structurels

JUnit

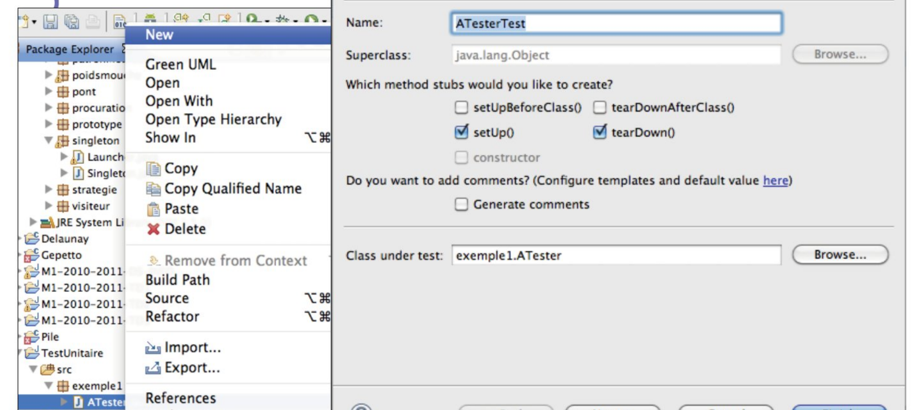
- On commence par le test fonctionnel
  - *La fonction doit retourner la somme de 2 entiers modulo 20000*
  - Test 1 - DT : {x=10, y=20}, prédiction de l' oracle : 30
  - Test 2 - DT : {x=10000, y=10050}, prédiction de l' oracle : 50



Tests uni

JUnit

- On commence par
  - *La fonction d*
  - Test 1 - DT :
  - Test 2 - DT :



Tests unitaires fonctionnels et structurels

JUnit

- On commence par le test fonctionnel
  - *La fonction doit retourner la somme de 2 entiers modulo 20000*
  - Test 1 - DT : {x=10, y=20}, prédiction de l'oracle : 30
  - Test 2 - DT : {x=10000, y=10050}, prédiction de l'oracle : 50

```java
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.Assert;

public class ATesterTest {

    ATester a;
    @Before
    public void setUp() throws Exception {
        a = new ATester();
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void testFonctionnel1(){
        int r = a.f(10, 20);
        Assert.assertEquals(30, r);
    }
}
```

Ecriture du test 1

Lancement des tests

JUnit — Finished after 0,012 seconds — Runs: 1/1 Errors: Failures: — exemple1.ATesterTest [Runne... — testFonctionnel1 (0,000 s) — Failure Trace

Tests unitaires fonctionnels et structurels

JUnit

- On commence par le test fonctionnel
  - *La fonction doit retourner la somme de 2 entiers modulo 20000*
  - Test 1 - DT : {x=10, y=20}, prédiction de l'oracle : 30
  - Test 2 - DT : {x=10000, y=10050}, prédiction de l'oracle : 50

```java
public class ATesterTest {

    ATester a;
    @Before
    public void setUp() throws Exception {
        a = new ATester();
    }

    @After
    public void tearDown() throws Exception {
    }
...
    }
```

Ecriture du test 2

Lancement des tests

JUnit — Finished after 0,05 seconds — Runs: 2/2 Errors: 0 Failures: 1 — exemple1.ATesterTest [Runner: JUnit 4] (0,029 s) — testFonctionnel1 (0,000 s) — testFonctionnel2 (0,029 s) — Failure Trace — java.lang.AssertionError: expected:<50> but was:<20050> — at exemple1.ATesterTest.testFonctionnel2(ATesterTest.java:30)

Tests unitaires fonctionnels et structurels

JUnit

- On commence par le test fonctionnel
  - *La fonction doit retourner la somme de 2 entiers modulo 20000*
  - Test 1 - DT : {x=10, y=20}, prédiction de l'oracle : 30
  - Test 2 - DT : {x=10000, y=10050}, prédiction de l'oracle : 5
- Correction du code et on relance le test

```java
public class ATester {
    public int f(int x, int y){
        if(x==500 && y==600)
            return y-x;
        else
            return (x+y)%20000;
    }
}
```

JUnit — Finished after 0,011 seconds — Runs: 2/2 Errors: 0 Failures: 0 — exemple1.ATesterTest [Runner: JUnit 4] (0,000 s) — testFonctionnel1 (0,000 s) — testFonctionnel2 (0,000 s) — Failure Trace

Tests unitaires fonctionnels et structurels

EclEmma

- Et maintenant le test structurel
  - Lancement d'une première couverture de code basée sur les tests

Menu: New, Go Into, Green UML, Open in New Window, Open Type Hierarchy, Show In, Copy, Copy Qualified Name, Paste, Delete, Remove from Context, Build Path, Source, Refactor, Import..., Export..., Refresh, Close Project, Close Unrelated Projects, Assign Working Sets..., Run As, Debug As, Profile As, Coverage As, Validate, Import Fit files — Coverage As: 1 Java Application, 2 JUnit Test, Coverage Configurations...

Tests unitaires fonctionnels et structurels

EclEmma

- Et maintenant le test structurel
  - Lancement d'une première couverture de code basée sur les tests

```
public class ATester {
    public int f(int x, int y){
        if(x==500 && y==600)
            return y-x;
        else
            return (x+y)%20000;
    }
}
```

Tout le code n'est pas couvert !!

En vert, ligne couverte,
En rouge, ligne non couverte,
En jaune, ligne partiellement couverte

EclEmma fait de la couverture de nœuds en splittant les conditions multiples

Coverage | Console

ATesterTest (19 sept. 2011 22:31:13)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
TestUnitaire	86,3 %	44	7	51
src	86,3 %	44	7	51
exemple1	63,2 %	12	7	19
ATester.java	63,2 %	12	7	19
ATester	63,2 %	12	7	19
f(int, int)	56,2 %	9	7	16
exemple1.test	100,0 %	32	0	32
ATesterTest.java	100,0 %	32	0	32

Tests unitaires fonctionnels et structurels

- Et maintenant le test structurel
  - Lancement d'une première couverture de code basée sur les tests
  - Ajout d'un test unitaire

```
@Test
public void testStructurel1(){
    int r = a.f(500, 600);
    Assert.assertEquals(1100, r);
}
```

```
public class ATester {
    public int f(int x, int y){
        if(x==500 && y==600)
            return y-x;
        else
            return (x+y)%20000;
    }
}
```

Couverture complète atteinte

Coverage | Console

ATesterTest (19 sept. 2011 22:32:24)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
TestUnitaire	82,3 %	51	11	62
src	82,3 %	51	11	62
exemple1.test	74,4 %	32	11	43
exemple1	100,0 %	19	0	19
ATester.java	100,0 %	19	0	19
ATester	100,0 %	19	0	19
f(int, int)	100,0 %	16	0	16

Couverture partielle de blocs avec EclEmma

EclEmma

- Une ligne est partiellement couverte car l'évaluation du && est fainéante
  - x vaut 1 dans le test
  - Donc x==2 est évalué mais pas y==1

```
public class A {

    public boolean testAND(int x, int y){
        if (x==2 && y==1)
            return true;
        else
            return false;
    }
}
```

```
@Test
public void testStructurel3() {
    Assert.assertEquals(false, a.testAND(1,2));

}
```

Couverture partielle de blocs avec EclEmma

EclEmma

- Avec x=2, le test y==1 est effectué

```
public class A {

    public boolean testAND(int x, int y){
        if (x==2 && y==1)
            return true;
        else
            return false;
    }
}
```

```
@Test
public void testStructurel3() {
    Assert.assertEquals(false, a.testAND(2,3));

}
```

Couverture partielle de blocs avec EclEmma

EclEmma

- Il en va de même avec l'évaluation de l'opérateur ||

```
@Test
public void testStructurel4() {
    Assert.assertEquals(true, a.testOR(2,1));
}
```

```
@Test
public void testStructurel4() {
    Assert.assertEquals(true, a.testOR(1,2));
}
```

```
public boolean testOR(int x, int y){
    if (x==2 || y==1)
        return true;
    else
        return false;
}
```

```
public boolean testOR(int x, int y){
    if (x==2 || y==1)
        return true;
    else
        return false;
}
```

Couverture partielle de blocs avec EclEmma

EclEmma

- La ligne 6 est un `if…then…else`,
  - `vi` valant 1 seule une branche de la condition est couverte
- Ceci implique pour la ligne 8, que
  - `vk<vj` est évalué à faux pour `vk=0`
  - Donc `++vk` n'est jamais exécuté, d'ou une couverture partielle aussi

```
 1 public class MyClass
 2 {
 3     public static void main (final String [] args)
 4     {
 5         int vi = 1;
 6         int vj = vi > 0 ? -1 : 1;
 7
 8         for (int vk = 0; vk < vj; ++ vk)
 9         {
10             System.out.println ("vk = " + vk);
11         }
12     }
13
14     public MyClass () {}
15 }
```

Couverture partielle de blocs avec EclEmma

EclEmma

- L'attribut `m_i` est initialisé lors de l'appel au constructeur de la classe
  - 2 constructeurs (défaut ou argument entier)
  - Seule le constructeur avec une valeur entière exécutée
  - Donc couverture partielle de l'initialisation de `m_i`

```
 1 public class MyClass
 2 {
 3     public static void main (final String [] args)
 4     {
 5         MyClass mc = new MyClass (5);
 6     }
 7
 8     public MyClass ()
 9     {
10     }
11
12     public MyClass (int size)
13     {
14         m_i = new Integer (size);
15     }
16
17     private Integer m_i = new Integer (0);
18 }
```

Plan du cours

- Définitions autour du test
  - Métriques de qualité/fiabilité logicielle
  - Positionnement du test dans le cycle de vie du logiciel
  - Test et méthodes agiles
  - Qu'est-ce que le test ?
- Processus de test simplifié
- Comment choisir les scénarios de test à jouer ?
  - Les méthodes de test Boîte Noire
  - Les méthodes de test Boîte Blanche
- Tests unitaires avec JUnit et EclEmma
  - Présentation de JUnit
  - Couverture de test avec EclEmma
- Utilisation de doublures (mocks) pour les tests unitaires avant intégration

Mocks, doublures ou simulacres

- Les frameworks de mock (Jmock, Mockito, EasyMock, MockRunner, etc.) fournissent une solution à des questions comme :
  - Comment faire si pour un test, vous avez besoin d'un objet qui n'est pas encore codé ou géré par un autre développeur qui ne l'a pas intégré ?

    → Dans le cas du test d'un composant avant son intégration
  - Comment faire si un test doit faire appel à une base de donnée et que la connexion est lente ou que la BD est absente, le disque dur saturé ?

    → dans le cas du test avant test système
- Un mock ou doublure ou simulacre permet de
  - Renvoyer des résultats prédéterminés ;
  - D'obtenir des états particuliers du contexte ;
  - D'invoquer des ressources qui prennent du temps ;
  - De simuler/doubler un objet au comportement spécifié mais pas implémenté.

Mockito

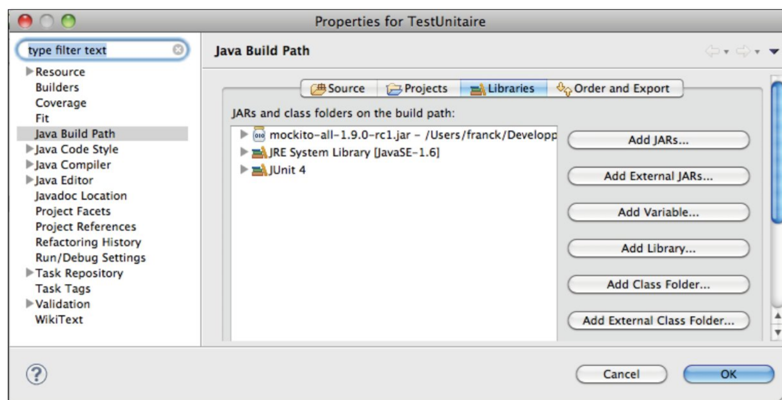
Les principes

- Mockito est un générateur automatique de doublures/simulacres
- Il fonctionne sur le mode de l'espion :
  - On crée les mocks
  - On décrit leur comportement (mode fake ou substitut) ;
  - Ensuite, à l'exécution, toutes les interactions avec les mocks sont mémorisées ;
  - A la fin du test, on interroge les mocks pour savoir comment ils ont été utilisés

Mockito

Installation et utilisation de Mockito

- Récupérer la dernière release mockito-all-1.9.0-rc1.jar
- Placer cette archive dans un répertoire pour vos développements
- importer-le dans le build path de votre projet



Mockito

Exemple d'un jeu de plateau

- Vous développez la classe `GameBoard`
- Un autre développeur se charge de la classe `Cell`
  - Interface bien définie (très important pour le travail par équipe)
- Vous voulez faire des tests unitaires des opérations `add()`, `getNbCells()` et `getCell()` de la classe `GameBoard`

```java
public interface Cell {
    public String getName();

    public GameBoard getGameBoard();
}
```

```java
public class GameBoard {

    private ArrayList<Cell> cells;
    public GameBoard() {
        cells = new ArrayList<Cell>();
    }

    public void add(Cell c){
        cells.add(c);
    }

    public int getNbCells(){
        return cells.size();
    }

    public Cell getCell(int i){
        Cell c=null;
        if (i>=0 && i<cells.size())
            c = cells.get(i);
        return c;
    }
```

Mockito

Exemple d'un jeu de plateau

Mockito

- On va créer un objet doublure de notre interface

Import des fonctionnalités de mocking

```
public interface Cell {
    public String getName();

    public GameBoard getGameBoard();
}
```

Création d'une doublure de cellule

```
import static org.junit.Assert.*;
import junit.framework.Assert;
import org.junit.Before;
import org.junit.Test;
import static org.mockito.Mockito.*;

import exempleMockito.GameBoard;
import exempleMockito.Cell;

public class TestGameBoard {

    GameBoard gb;
    @Before
    public void setUp() throws Exception {
        gb = new GameBoard();
    }

    @Test
    public void testAdd(){
        Cell c = mock(Cell.class);
        gb.add(c);
        Assert.assertEquals(1,gb.getNbCells());
    }

}
```

Exemple d'un jeu de plateau

Mockito

- On va créer un objet doublure de notre interface
- Puis lui attribuer des comportements en cas d'appel aux opérations

```
public interface Cell {
    public String getName();

    public GameBoard getGameBoard();
}
```

Création d' une doublure de cellule

Spécification de la réponse en cas d' appel à l' opération `getName()` sur l'objet `c`

```
@Test
public void testGetCell(){
    Cell c = mock(Cell.class);
    when(c.getName()).thenReturn("case 1");
    gb.add(c);

    Cell c2 = gb.getCell(0);
    Assert.assertEquals("case 1",c2.getName());
}
```

Les principes

Mockito

- On importe le package qui va bien
  `import static org.mockito.Mockito.*;`
- On crée les doublures à l' aide de la méthode `mock`
  `MonItf doublure = mock(MonItf.class);`
- On décrit le comportement attendu de la doublure avec la méthode `when`
  - par exemple
    `when(doublure.monOpe()).thenReturn(12);`
- On crée l'objet de la classe à tester en utilisant les doublures, on décrit le corps du test
- On vérifie que l'interaction avec les doublures est correcte par la méthode `verify`.

Fonctionnement de la méthode mock

Mockito

```
public static <T> T
mock(java.lang.Class<T> classToMock)
```

- Cette méthode permet de créer une doublure
  - Pour une interface
  - Comme pour une classe
- En théorie, en test, vous ne devez pas faire des doublure de classes mais seulement d'interfaces qui décrivent les services des des objets collaborant avec l'objet à tester
- Nommage du mock pour des messages d' erreurs plus clairs

```
public static <T> T
mock(java.lang.Class<T> classToMock, String name)
```

Mockito

Fonctionnement de la méthode mock – Comportement par défaut

- On suppose donnée une interface `Itf` contenant les méthodes
  - `exempleInt`
  - `exempleBool`
  - `exempleList`

  avec les types de retour correspondant.

```
Itf mock = mock(Itf.class, «Itf»)
```

- Dès l'exécution de cet appel, lobjet `mockItf` est créé avec des **comportements par défaut** :

```
assertEquals("Itf", mock.toString());
assertEquals("type int : 0 ", 0,mock.exempleInt());
assertEquals("type bool : false",false,
                              mock.exempleBool());
assertEquals("collection : vide",0,
                    mock.exempleList().size());
```

Mockito

Stubbing et méthode When

- Pour définir le comportement de son choix (**stubbing**), on utilise la méthode `when` associée aux fonctions de stubbing :

```
thenReturn(T value);
thenReturn(T value1, ...,T valuen);
thenThrow(Throwable t);
thenThrow(Throwable t1, ..., Throwable tn)
```

```
when(mock.exempleInt()).thenReturn(3);
```

- Utilisation avec Junit

```
assertEquals("une fois 3",3, mock.exempleInt());
assertEquals("deux fois 3",3, mock.exempleInt());
```

Retourne la valeur stubbée autant de fois que nécessaire.

Mockito

Fonctionnement de la méthode when

Valeurs de retour consécutives

```
when(mock.exempleInt()).thenReturn(3, 4, 5);
```

- Utilisation avec JUnit

```
assertEquals("une fois : 3",3, mock.exempleInt());
assertEquals("deux fois : 4",4, mock.exempleInt());
assertEquals("trois fois: 5",5, mock.exempleInt());
```

- `when(mock.exempleInt()).thenReturn(3, 4);`

  est un raccourci pour

  `when(mock.exempleInt()).thenReturn(3).thenReturn(4);`

Mockito

Fonctionnement de la méthode when

Fonctionnement différent suivant la valeur des paramètres

```
public int exempleIntBis(int i, int j);
```

- On écrira par exemple

```
when(mock.exempleIntBis(4,2)).thenReturn(4);
when(mock.exempleIntBis(5,3)).thenReturn(5);
```

- Utilisation avec JUnit

```
assertEquals("param 4 2:4" , 4,
                    mock.exempleIntBis(4,2));
assertEquals("param 5 3 : 5", 5,
                    mock.exempleIntBis(5,3));
```

Mockito

Fonctionnement de la méthode when

Fonctionnement avec levée d'exceptions et void

- Attention, la méthode when ne permet pas de stubber des méthodes de type void

```
public void opVoidThrowExc() throws ExeException;
```

- On écrira

```
doThrow(new ExeException()).when(mock).
                                  opVoidThrowExc();
```

- Utilisation avec JUnit

```
try {
    mock.opVoidThrowExc();
    fail();
} catch (ExeException e) {
    assertTrue("levee exception", true);
}
```

Mockito

Fonctionnement de la méthode verify

- La méthode `exempleBool` doit avoir été appelée. . .
  - exactement une fois :
    ```
    verify(mock).exempleBool();
    verify(mock,times(1)).exempleBool(); //équivalent
    ```
  - Au moins / au plus une fois:
    ```
    verify(mock, atLeastOnce()).exempleBool();
    verify(mock, atMost(1)).exempleBool();
    ```
  - Jamais (ce qui est faux) :
    ```
    verify(mock, never()).exempleBool();
    ```
- Avec des paramètres

```
verify(mock).exempleIntBis(4, 2);
```

Mockito

Fonctionnement de la méthode verify

- Ordre des appels

```
import org.mockito.InOrder;
```

- Pour vérifier que l'appel (4,2) est effectué avant l'appel (5,3)

```
InOrder inOrder = inOrder(mock);
inOrder.verify(mock).methodeIntBis(4, 2);
inOrder.verify(mock).methodeIntBis(5, 3);
```

- Marche aussi avec plusieurs mocks :

```
InOrder inOrder = inOrder(mock1, mock2);
inOrder.verify(mock1).foo();
inOrder.verify(mock2).yo();
```

Mockito

Pour finir

- Vérification qu'il n'y a pas d'interaction :
  - `verifyNoMoreInteractions(mock);`
  - `verifyZeroInteractions(mock);`
- Possibilité de décrire des contraintes sur les paramètres
  - `ArgumentMatcher`

Pour conclure

- Le test de logiciels est une activité très coûteuse qui gagne à être supportée par des outils.

- Les principales catégories d'outils concernent :
  - L'exécution des tests
  - La gestion des campagnes
  - Le test de performance
  - La génération de tests fonctionnels
  - La génération de tests structurels

- N'oubliez pas l'importance du test !!!