

## **Méthode de développement**

### **Objectifs**

- Améliorer la qualité du logiciel
- Permettre au développeur d'avoir plus de confiance dans son travail.

### **Cycle TDD**

- Ecrire un test
- Vérifier qu'il échoue, car le code à tester n'existe pas encore
- Ecrire le code à tester de façon à simplement permettre le passage du test
- Vérifier que le test se déroule bien
- Refactoriser le code, afin de l'améliorer
- Vérifier que le test se déroule toujours bien

## **Principes à respecter**

### **Bases du TDD**

- On écrit du code SEULEMENT si un test échoue
- AUCUNE duplication de code n'est permise

### **Implications**

- Le code métier évolue de façon incrémentale
- Les tests et le code métier sont écrits par la même personne
- L'environnement de développement doit permettre la réalisation rapide de petits changements
- Le code doit être composé d'éléments très cohérents et peu couplés, afin d'être facilement testables

## Processus détaillé

- ETAPE 1: Ecrire un petit test qui échoue dans un premier temps
- ETAPE 2:
  - Développer le plus rapidement possible le code permettant au test de réussir
- ETAPE 3: éliminer les duplications apparues entre les deux premières étapes

### Possibilités étape 2

- Implémentation évidente
- Commencer par simuler, puis remplacer par le véritable code
- Généraliser à partir d'exemples

## *Framework Spring 4*

### *Fakes et Mocks dans les tests unitaires*

---

- Tests unitaires dans des environnements complexes.
- Implémenter des fakes et des mocks pour isoler des classes.

## Limite des tests unitaires ?

- Les classes à tester sont rarement totalement isolées.
- Elles dépendent d'autres classes.
- Un test unitaire ne doit se concentrer que sur une classe à la fois.
- Les tests unitaires ne doivent pas se transformer en test d'intégration.

```
public class LivreMetier {  
    /** LivreDAO est une interface */  
    private LivreDAO livreDAO;  
  
    public void libererTout() throws MetierException {  
        try {  
            Livre[] livres = livreDAO.listerLivres();  
            for (Livre livre : livres) {  
                livreDAO.modifierEtatLivre(livre, Livre.ETAT_LIBRE);  
            }  
        } catch (DAOException ex) {  
            throw new MetierException("Impossible de libérer les livres.");  
        }  
    }  
}
```

## Une première solution : les fakes

- Un fake est une classe exposant la même interface que la classe dont dépend la classe testée.
- Les méthodes d'un fake renvoient toujours les mêmes résultats, ils ne dépendent que d'eux même.
- Trois méthodes pour injecter un fake dans la classe testée :
  - Le fake implémente l'interface voulue et est injecté dans la classe testée via un accesseur. C'est la solution la plus simple, la plus propre et est aisément mise en oeuvre si l'application gère des notions d'**inversion de contrôle** ou d'**injection de dépendance**.
  - Le fake hérite de la classe qu'elle remplace et est injecté dans la classe testée via un accesseur. Solution possible uniquement si la classe n'est pas **final** et que l'accesseur existe.
  - Donner au fake le nom exact de la classe qu'il doit remplacer et jouer sur les priorités du **ClassLoader** afin que ce soit lui qui soit chargé. C'est la seule solution si la classe testée référence directement (instancie) la classe qu'elle utilise.

## Une première solution : les fakes

### Développement d'un fake

```
public class LivreDAOFake implements LivreDAO {
    private Livre[] listeLivres;

    public Livre[] listerLivres() throws DAOException {
        return listeLivres;
    }

    public void modifierEtatLivre(Livre livre, int etat) throws DAOException {
        EtatLivre etatLivre = new EtatLivre();
        etatLivre.setIdEtatLivre(etat);
        livre.setEtatLivre(etatLivre);
    }

    public Livre[] getListeLivres() {
        return listeLivres;
    }

    public void setListeLivres(Livre[] listeLivres) {
        this.listeLivres = listeLivres;
    }
}
```

## Une première solution : les fakes

### Préparation du test

```
public class LivreMetierTest {
    private LivreMetier metier;
    private LivreDAOFake fake;

    @Before
    public void init() {
        metier = new LivreMetier();
        fake = new LivreDAOFake();

        Livre livre = new Livre();
        Auteur auteur = new Auteur();
        auteur.setNom("Zola");
        auteur.setPrenom("Emile");
        livre.setAuteur(auteur);
        livre.setTitre("Les misérables");

        fake.setListeLivres(new Livre[] {livre});

        metier.setLivreDAO(fake);
    }
}
```

## Une première solution : les fakes

### Méthode de test

```
@Test
public void libererToutOK() {
    try {
        metier.libererTout();
        for (Livre livre : fake.getListeLivres()) {
            assertEquals(Livre.ETAT_LIBRE, livre.getEtatLivre().getIdEtatLivre());
        }
    } catch (MetierException e) {
        fail("Exception : " + e);
    }
}
```

- Le comportement de la classe LivreMetier ne dépend plus que du fake dont le comportement est parfaitement maîtrisé.

### Limite des objets fake

- Les fakes sont longs et pénibles à développer.
- Si la méthode testée appelle plusieurs fois la même méthode du fake, le résultat sera toujours le même (sauf développement couteux du fake).
- Si dans une série de test, les méthodes testées appellent plusieurs fois la même méthode du fake, le résultat sera toujours le même (sauf développement couteux du fake).
- Les interactions entre la classe testée et le fake ne peuvent pas être contrôlées.

---

Besoin de créer un fake plus "intelligent".

---

## Les mocks

- Objets fakes rendus dynamiques et intelligents.
- Possibilité de spécifier, pour chaque méthode, le résultat de chaque invocation (y compris des exceptions).
- Possibilité de spécifier le nombre d'invocations attendu pour chaque méthode et de demander une vérification.
- Possibilité de spécifier les paramètres d'appel de chaque méthode et de demander une vérification.
- Permet de tester le comportement interne d'une classe de façon complète :
  - On vérifie son résultat final comme dans tout test unitaire (adéquation avec la conception générale)
  - On vérifie la façon dont le résultat est obtenu (communication précise entre objets - adéquation avec la conception détaillée).
- Tout en restant dans le cadre du test unitaire (une seule classe testée).

---

Nécessité d'outiller la génération de mocks

---

## Générateurs de Mocks Java

- **jMockit** : projet mis à disposition par Google permettant de mettre en place des mocks sans limitation
- **MockMaker** : projet sourceforge n'évoluant plus depuis Juillet 2002. Son plugin Eclipse est cependant toujours compatible avec Eclipse 3.3. Cet outil génère des mocks statiques.
- **easyMock** : projet sourceforge actif. Les mocks sont générés dynamiquement.



- **jMock** : autre générateur de mocks dynamique.



- **RMock** : extension de JUnit autorisant des assertions plus fines et générants des mocks dynamiques.
- **SevenMock**

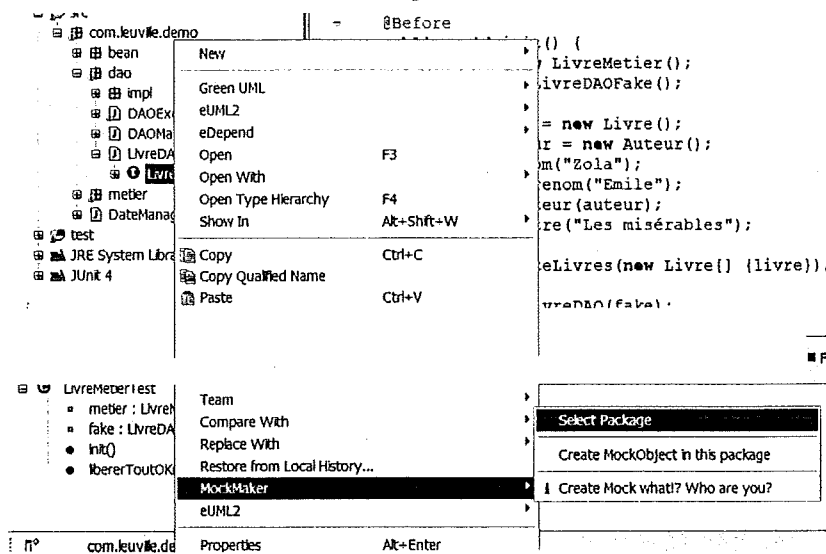


- **Mockito** : autre générateur de mocks dynamique, basé sur easyMock, avec simplification des fonctionnalités et de la syntaxe.



## Exemple d'utilisation d'un mock statique

- Génération d'un mock avec mockMaker sous Eclipse :



## Exemple d'utilisation d'un mock statique

```
public class MockLivreDAO implements LivreDAO{
    private ExpectationCounter myListerLivresCalls = new ExpectationCounter("com.leuville.de
    private ReturnValues myActualListerLivresReturnValues = new ReturnValues(false);
    private ExpectationCounter myModifierEtatLivreCalls = new ExpectationCounter("com.leuvi
    private ReturnValues myActualModifierEtatLivreReturnValues = new VoidReturnValues(false
    private ExpectationList myModifierEtatLivreParameter0Values = new ExpectationList("com.
    private ExpectationList myModifierEtatLivreParameter1Values = new ExpectationList("com.
    public void setExpectedListerLivresCalls(int calls){
        myListerLivresCalls.setExpected(calls);
    }
    public Livre[] listerLivres() throws DAOException{
        myListerLivresCalls.inc();
        Object nextReturnValue = myActualListerLivresReturnValues.getNext();
        if (nextReturnValue instanceof ExceptionalReturnValue && ((ExceptionalReturnValue)nex
            throw (DAOException)((ExceptionalReturnValue)nextReturnValue).getException();
        if (nextReturnValue instanceof ExceptionalReturnValue && ((ExceptionalReturnValue)nex
            throw (RuntimeException)((ExceptionalReturnValue)nextReturnValue).getException();
        return (Livre[]) nextReturnValue;
    }
}
```

## Exemple d'utilisation d'un mock statique

```
public class LivreMetierTest {

    private LivreMetier metier;
    private MockLivreDAO mock;

    @Before
    public void init() {
        metier = new LivreMetier();
        mock = new MockLivreDAO();
    }
}
```

## Exemple d'utilisation d'un mock statique

```
@Test
public void libererToutOK() {
    Livre livre = new Livre();
    Auteur auteur = new Auteur();
    auteur.setNom("Zola");
    auteur.setPrenom("Emile");
    livre.setAuteur(auteur);
    livre.setTitre("Les misérables");

    mock.setupListerLivres(new Livre[] {livre});
    mock.addExpectedModifierEtatLivreValues(livre, Livre.ETAT_LIBRE);
    mock.setExpectedListerLivresCalls(1);
    mock.setExpectedModifierEtatLivreCalls(1);
    metier.setLivreDAO(mock);

    try { metier.libererTout(); }
    catch (MetierException e) { fail("Exception : " + e); }

    mock.verify();
}
```



## *Framework Spring 4*

### *Tests unitaires avec JUnit*

---

- Définition du test unitaire.
- Présentation de JUnit

#### **Test unitaire**

- Le **test unitaire** est un procédé permettant de s'assurer du fonctionnement correct d'une **partie déterminée d'un logiciel** ou d'une **portion d'un programme** (appelée « unité »).
- Il s'agit pour le programmeur de tester un module, **indépendamment du reste du programme**, ceci afin de s'assurer qu'il répond aux spécifications fonctionnelles et qu'il fonctionne correctement en toutes circonstances. Cette vérification est considérée comme essentielle, en particulier dans les applications critiques. Elle s'accompagne couramment d'une vérification de la **couverture de code**, qui consiste à s'assurer que le test conduit à exécuter l'ensemble (ou une fraction déterminée) des instructions présentes dans le code à tester.
- L'ensemble des tests unitaires doit être rejoué après une modification du code afin de vérifier qu'il n'y a pas de **régressions** (l'apparition de nouveaux dysfonctionnements).
- Dans les applications non critiques, l'écriture des tests unitaires a longtemps été considérée comme une tâche secondaire. Cependant, la méthode **Extreme programming (XP)** a remis les tests unitaires, appelés « tests du programmeur », au centre de l'activité de programmation.
- La méthode XP préconise d'écrire les tests **en même temps, ou même avant la fonction à tester** (Test Driven Development). Ceci permet de définir précisément l'interface du module à développer. En cas de découverte d'un bogue, on écrit la procédure de test qui reproduit le bogue. Après correction on relance le test, qui ne doit indiquer aucune erreur.

## Assertions

- Les tests unitaires sont basés sur des assertions.
- Une **assertion** est une expression booléenne décrivant une condition particulière qui doit être satisfaite pour indiquer que la méthode réussit un test.
- Les assertions sont incluses dans le langage Java depuis le JDK 1.4.

## Classe "jumelle" de test

- Dans une application, presque chaque classe (applicative) possède une classe « jumelle » de test.
- La classe applicative évolue avec sa classe de test.
- Lorsqu'on ajoute une nouvelle méthode à une classe applicative:
  - on commence d'abord par des nouvelles assertions / cas de test dans la classe de test.
  - on implémente la nouvelle méthode dans la classe applicative et on fait attention pour être certain qu'elle passe les tests!
- Il n'est pas rentable de tester toutes les méthodes de toutes les classes applicatives.
  - Cependant, la règle de base est : « **Test everything that could possibly break** », c'est à dire, tester tout ce qui peut avoir des erreurs.

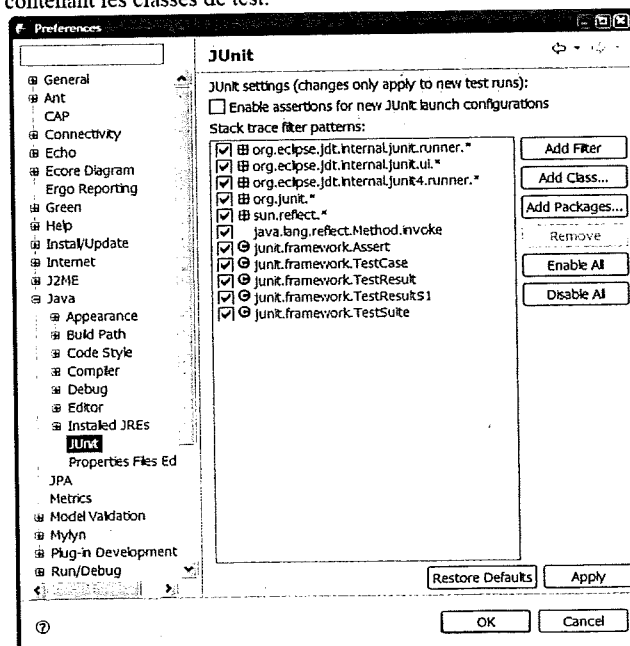
## Présentation de JUnit

- Imaginé et développé en Java par **Kent Beck** et **Erich Gamma**.
- Framework de rédaction et d'exécutions de tests unitaires en Java.
- Avec JUnit, l'unité de test est une classe dédiée qui regroupe des cas de tests. Ces cas de tests exécutent les tâches suivantes :
  - création d'une instance de la classe et de tous autres objets nécessaires aux tests
  - appel de la méthode à tester avec les paramètres du cas de test
  - comparaison du résultat obtenu avec le résultat attendu : en cas d'échec, une exception est levée

<http://junit.org/>

## Installation de JUnit

- Directement intégré dans la plupart des IDE java.
- Il est également possible de télécharger l'archive junit.jar depuis le site officiel et de l'ajouter au CLASSPATH de l'application contenant les classes de test.



## Ecriture d'une classe de test avec JUnit 4

- Créer la classe "jumelle" de la classe devant être testée.
- Créer les méthodes de test. Ces méthodes :
  - doivent être annotées avec l'annotation `org.junit.Test` (depuis JUnit 4)
  - créent une instance de la classe devant être testée et invoquent ses méthodes.
  - font des assertions permettant de vérifier le résultat des invocations par rapport au résultat attendu.

```
public class DateManagerTest {
    @Test
    public void parseDateOK() {
        DateManager manager = new DateManager();
        Date date = manager.parseDate("08/12/1978");

        assertNotNull(date);
        Calendar cal = Calendar.getInstance();
        cal.setTime(date);
        assertEquals(8, cal.get(Calendar.DATE));
        assertEquals(Calendar.DECEMBER, cal.get(Calendar.MONTH));
        assertEquals(1978, cal.get(Calendar.YEAR));
    }
}
```

## Assertions JUnit

- **Forme générale des méthodes `assertXXX` ([message], valeur attendue, valeur testée)**
  - message à utiliser si l'assertion se révèle fausse [OPTIONNEL]
- **`assertEquals`** : vérifie l'égalité entre des valeurs. **`assertFalse`** : vérifie qu'une expression booléenne est fausse.
- **`assertTrue`** : vérifie qu'une expression booléenne est vraie.
- **`assertNotNull`** : vérifie qu'une valeur est non nulle (pointeur).
- **`assertNull`** : vérifie qu'une valeur est nulle.
- **`assertNotSame`** : vérifie que deux objets ne sont pas identiques au sens de l'identité objet.
- **`assertSame`** : vérifie que deux variables identifient le même objet.
- **`fail`** : fait automatiquement échouer le test.
- **`assertArrayEquals`** : vérifie que deux tableaux contiennent les mêmes valeurs.

## assertThat

- Nouvelle méthode d'assertion plus lisible et compréhensible

```
assertThat(x, is(3));
assertThat(x, is(not(4)));
assertThat(responseString,
    either(containsString("color")).or(containsString("colour")));
assertThat(myList, hasItem("3"));
```

- Forme de la méthode:  
assertThat([value], [matcher statement]);
- Familles de matcher:
  - JUnit: import static org.junit.matchers.JUnitMatchers.\*
  - Hamcrest: import static org.hamcrest.CoreMatchers.\*

## JUnit Matchers

- **containsString**: vérifie si une chaîne de caractère est présente
- **both**: `assertThat(string, both(containsString("a")).and(containsString("b")));`
- **either**: `assertThat(responseString, either(containsString("color")).or(containsString("colour")));`
- Vérification sur des listes:
  - **everyItem**,
  - **hasItem**,
  - **hasItems**

## Hamcrest Matchers

- Core: `anything()`, `is(..)`
- Logiques: `allOf(..)`, `anyOf(...)`, `not (...)`
- Objet: `equalTo(..)`, `instanceOf(..)`, `notNullValue()`, `nullValue()`

```
assertThat("Hello", is(allOf(notNullValue(), instanceOf(String.class), equalTo("Hello"))));
```

```
assertThat("Hello", is(notNullValue()));
```

```
assertThat("Hello", is(allOf(notNullValue(), instanceOf(String.class), equalTo("Hello"))));
```

## Test des exceptions

- Il est également possible de vérifier que les exceptions attendues lors d'un traitement sont effectivement lancées.
- Avec l'utilisation de l'attribut `expected` de l'annotation `org.junit.Test`.

```
import java.util.ArrayList;
import org.junit.Test;

public class ListTest {

    @Test(expected = IndexOutOfBoundsException.class)
    public void listeVide() {
        new ArrayList<String>().get(0);
    }

}
```

## Test des exceptions

- Utilisation d'un bloc try/catch

```
import java.util.ArrayList;
import org.junit.Test;

public class ListTest {
    @Test
    public void testListeVide() {
        try {
            new ArrayList<Object>().get(0);
            fail("Erreur: IndexOutOfBoundsException attendue");
        } catch (IndexOutOfBoundsException ex) {
            assertEquals(ex.getMessage(), "Index: 0, Size: 0");
        }
    }
}
```

## Durée d'exécution des tests

- Il est possible de faire échouer des tests mettant trop de temps à s'exécuter
- Paramètre timeout de l'annotation @Test

```
@Test(timeout = 1000)
public void testTimeout() {
    //.....
}
```

- Utilisable d'une règle (@Rule) applicable à l'ensemble des tests de la classe

```
public class HasGlobalTimeout {
    public static String log;
    @Rule
    public Timeout globalTimeout = new Timeout(10000); // 10s max par test
    @Test
    public void testInfiniteLoop1() {log += "ran1"; for (;;) { } }

    @Test
    public void testInfiniteLoop1() {log += "ran1"; for (;;) { } }
}
```

## JUnit Fixture

- Mécanisme permettant de créer un environnement de test.
  - Permet de préciser des opérations à effectuer avant chaque test et après chaque test d'une classe de test.
  - Utile pour initialiser l'objet testé et / ou ses dépendances.
- Utilise les annotations `org.junit.Before` et `org.junit.After`

```
public class DateManagerTest {
    private DateManager manager;

    @Before
    public void init() {
        manager = new DateManager();
    }

    @After
    public void destroy() {
        manager = null;
    }

    @Test
    public void parseDateOK() {
        Date date = manager.parseDate("08/12/1978");
    }
}
```

## JUnit Fixture

- Utilisation des annotations `@BeforeClass` et `@AfterClass` permettent de faire des initialisations communes à l'ensemble des tests de la classe (par exemple dans le cas d'utilisation de ressources externe - connection base de données par exemple)
  - La méthode annotée avec `@BeforeClass` sera appelée après l'instanciation de la classe de test,
  - La méthode annotée avec `@AfterClass` sera appelée juste après l'exécution du dernier cas de test.

```
public class PropTest {
    private static Properties prop; // les propriétés
    private static FileInputStream propFile; //le fichier de propriétés

    @BeforeClass
    public static void setUpBeforeClass() throws Exception {
        prop = new Properties();
        //charge le fichier de propriétés
        propFile = new FileInputStream("src/config.properties");
        prop.load(propFile);
    }

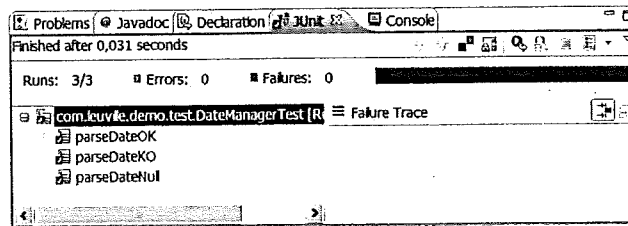
    @AfterClass
    public static void tearDownAfterClass() throws Exception {
        propFile.close(); //fermeture fichier de propriétés
    }
}
```



## Lancer un test

### Depuis Eclipse

- Menu contextuel de la classe de test > **Run As> JUnit Test**



### Depuis du code Java

- `JUnitCore.runClasses(DateManagerTest.class);`
- A la façon des testSuite JUnit 3 :

```
public static junit.framework.Test suite() {  
    return new JUnit4TestAdapter(DateManagerTest.class);  
}
```

- Avec des annotations :

```
@RunWith(Suite.class)  
@SuiteClasses(value={DateManagerTest.class, LivreMetierTest.class})  
public class TestLauncher {
```

## *Framework Spring 4*

### *Couplage Spring avec JUnit*

---

- Extensions de Spring pour JUnit

## **Extensions pour JUnit**

### **Lanceur de tests spécifiques**

- `org.springframework.test.context.junit4.SpringJUnit4ClassRunner`

### **Mocks permettant de simuler des objets JavaEE**

- Définis dans les sous-paquetages de `org.springframework.mock`

### **Versions**

- Spring 2.5 supporte JUnit 4.4
- Spring 3.0 nécessaire pour JUnit 4.5 et au delà

## **Tests d'intégration**

### **Annotations de `org.springframework.test.context`**

- Chargement d'un contexte Spring pour chaque test
- Mise en cache automatique des contextes partagés entre plusieurs tests
- Injection de dépendances dans les tests
- Possibilité d'ajout d'objets réagissant au cycle de vie des tests

## Annotations

### Exemple

- `@RunWith` délègue l'exécution à une classe Spring
- `@ContextConfiguration` permet de préciser les fichiers constituant le contexte Spring

```
package com.leuville.junit;

import junit.framework.Assert;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import com.leuville.bean.DAO;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class DAOTest {

    @Autowired
    private DAO dao; // dépendance vers classe à tester

    @Test
    public void initOK() {
        Assert.assertNotNull(dao); // teste si injection dépendance
    }

    // ...
}
```

## @ContextConfiguration

### Comportement par défaut

- Si la classe portant l'annotation est `com.leuville.MaClasse`
- Le contexte Spring sera chargé à partir de `classpath:/com.leuville.MaClasse-context.xml`

### Attribut locations


- Permet de définir un ou plusieurs fichiers de configuration du contexte pour un test

```
package com.leuville.junit;

...
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={
    "/path1/fic1.xml",
    "/path2/fic2.xml"
})
public class DAOTest {
}
```

## Ecoute de l'exécution des tests

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
@TestExecutionListeners({MyTestListener.class})
public class DAOTest {
}
```



```
package com.leuville.junit;

import org.springframework.test.context.TestContext;
import org.springframework.test.context.TestExecutionListener;

public class MyTestListener implements TestExecutionListener {

    public void afterTestClass(TestContext arg0) throws Exception { }

    public void afterTestMethod(TestContext arg0) throws Exception { }

    public void beforeTestClass(TestContext arg0) throws Exception { }

    public void beforeTestMethod(TestContext arg0) throws Exception { }

    public void prepareTestInstance(TestContext arg0) throws Exception { }
}
```

## *Framework Spring 4* *Java Persistence API, les bases*

---

- Concepts essentiels
- Exemples