# Java Enterprise Edition - Architectures et données

Gaël Guibon

E-mails: prenom.nom@lis-lab.fr

Aix*Marseille université

1. **La fiabilité**
2. **Les tests**
3. **Les tests unitaires**
4. **Spring et tests**

# La fiabilité

# La fiabilité

## Fiabilité

- Fonctionnelle dans différents contextes
- Chaque brique logicielle est vérifiée
- Les options et possibilités principales sont effectuées

# La fiabilité

Fiabilité != Sécurité

## Sécurité

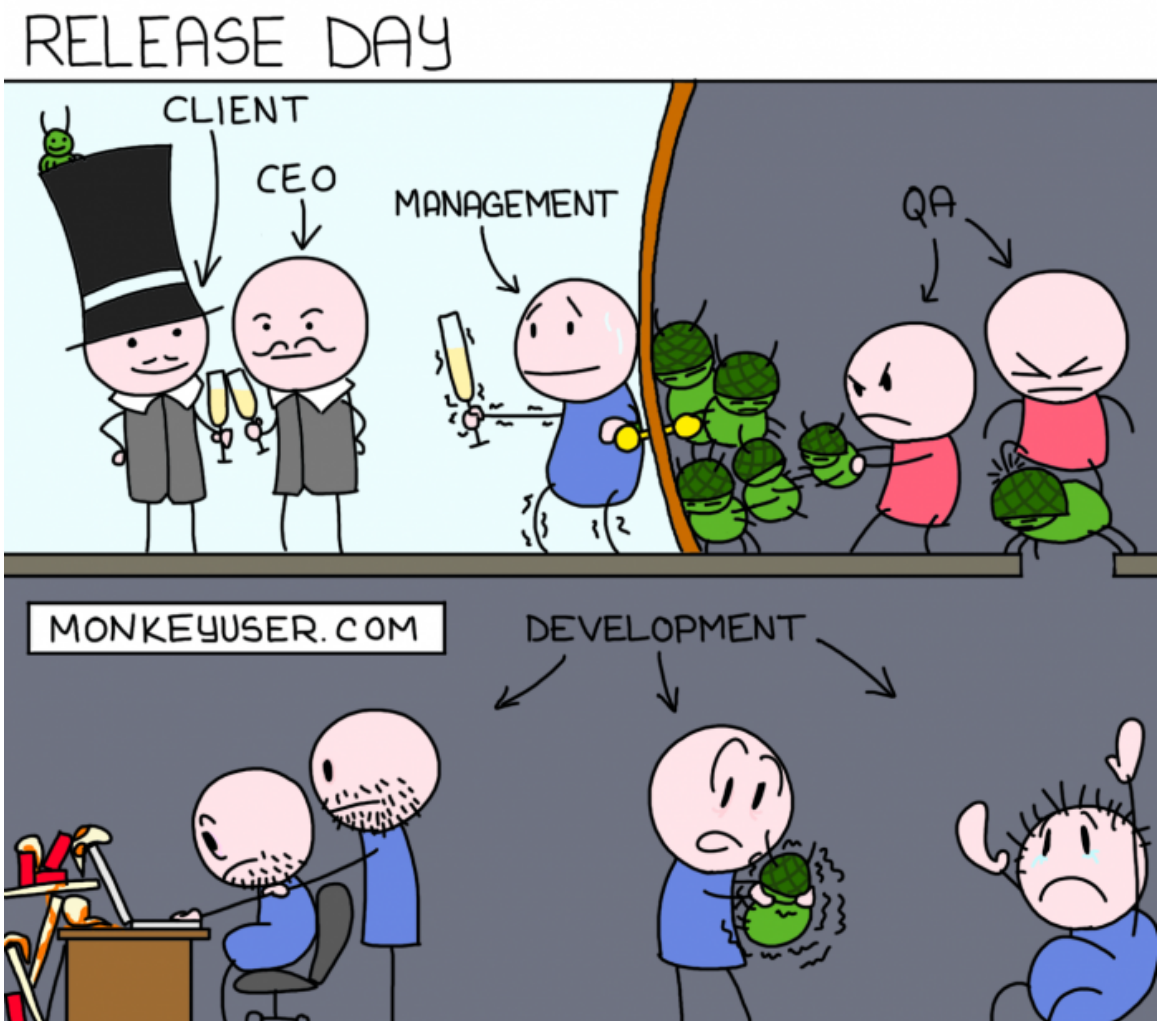
- Gestion des logins et authentification
- Conservation des informations sensibles côté serveur
- Parades aux failles habituelles : brute force, etc.

**La fiabilité de la sécurité peut être vérifiée et testée !**

# La fiabilité

## Mais dans la vraie vie...

Il y a toujours des bugs. Aucun n'est désiré.



# La fiabilité

## Intérêts du client

- Robustesse
- Mise en production possible dans des contextes à risques
  - Bourse, finance, médical...
- Confiance du client en l'entreprise
- Mise à l'échelle possible

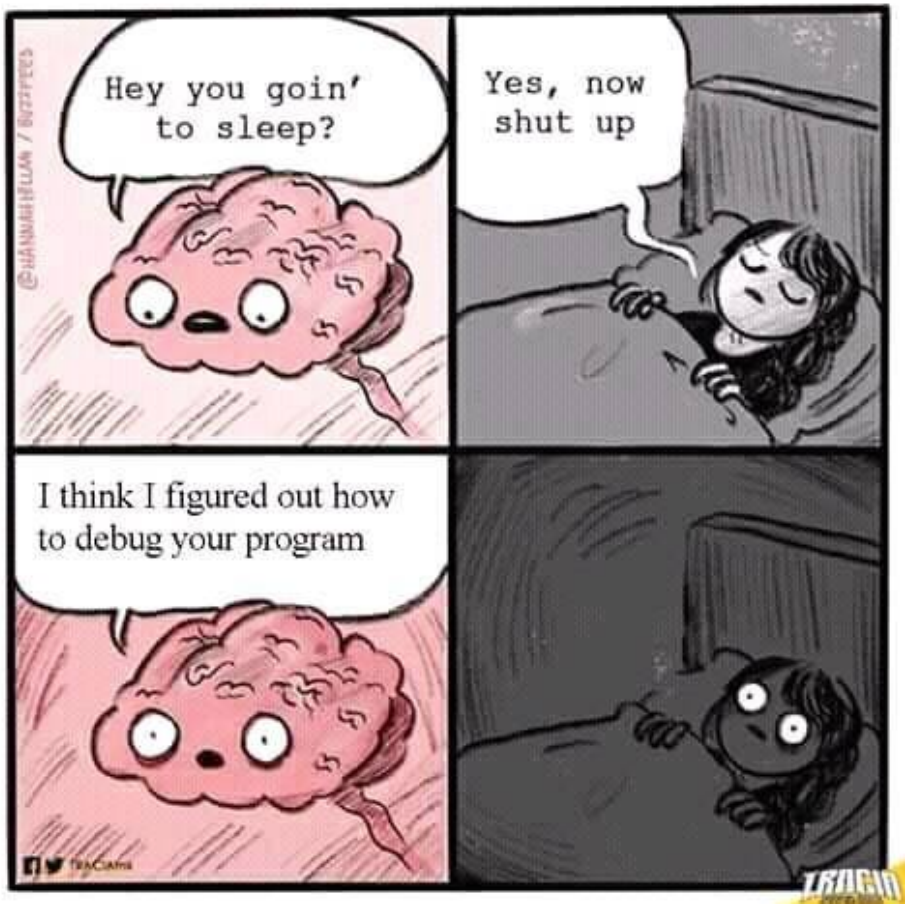
# La fiabilité

## Intérêts du développeur

- Gain de temps
- Investissement dans le temps
- Moins de sollicitations
- Partage et transmission facilités
- Réputation

# La fiabilité

**Ça prend du temps !!!** Comment l'automatiser ?



# La fiabilité

## Solution : Les tests

- Tests d'acceptation (par le client final)
- Tests du système (fonctionnalités, système entier)
- Tests d'intégration (Interaction des morceaux, Environnement, DB, etc.)
- Tests unitaires (Petits morceaux du programme)

Les tests unitaires et un peu ceux d'intégration nous intéresse.

# La fiabilité

## Phases de tests

1. Test de régression : pas de perte de fonctionnalités, on ne régresse pas en qualité
2. Test de robustesse : tester les entrées non prévus et cas particuliers
3. Test sous stress : tester les limites du système : mise à l'échelle et tentatives de crash

# Les tests

# Les tests

## En simple c'est :

- Un essai d'exécution
  - Selon le contexte
- Une comparaison avec la spécification
  - SI test passé : ACCEPT : True
  - SINON : FAIL : False

# Les tests

## Non exhaustifs

- Ne vérifie pas l'absence d'erreurs
- Ne cherche pas à trouver toutes les erreurs
- Ne vérifie pas les fonctionnalités
- Ne corrige pas les erreurs

## But

Selon un contexte, trouver des erreurs d'exécution.

# Les tests

Ils suivent des spécifications (oracle) qui contient des informations complètes.

## Objectif de la spécification/oracle

- Trouver le plus d'erreurs
- Programmer sans erreurs

Certains utilisent les tests pour définir les attentes du programme.

# Les tests

## Avantages

- Simplicité
- Faisabilité
- Rapport performance / coût
- Populaire (programmation orientée tests)

## Inconvénients

- Peu rigoureux (dépend des spécifications)
- Non-exhaustif (impossible de l'être)
- Qualité subjective

# Les tests

## Stratégie de test

Comment détecter le plus d'erreurs ?

## Couverture du code

Le test est-il efficace ?

## Modèles de spécification de code

Comment formaliser le test ?

# Les tests

## Couverture des tests

- Tout tester est impossible et trop long, voire infini.
- Trouver le maximum d'erreurs avec le minimum de chemins.

# Les tests

## Test en boîte blanche


## Test en boite noire

# Les tests : boite blanche

## Test “boîte blanche”

- Accès au contenu et fonctionnement.
- Précis

## Approche

- Exécution lente pas à pas : 30 à 70% des erreurs
- Test structurel statique
- Test structurel dynamique
  - Couverture du flot de données
  - Couverture du flot de contrôle
- Test par mutation : adaptation des test aux changements

# Les tests : boite noire

## Fonctionnement

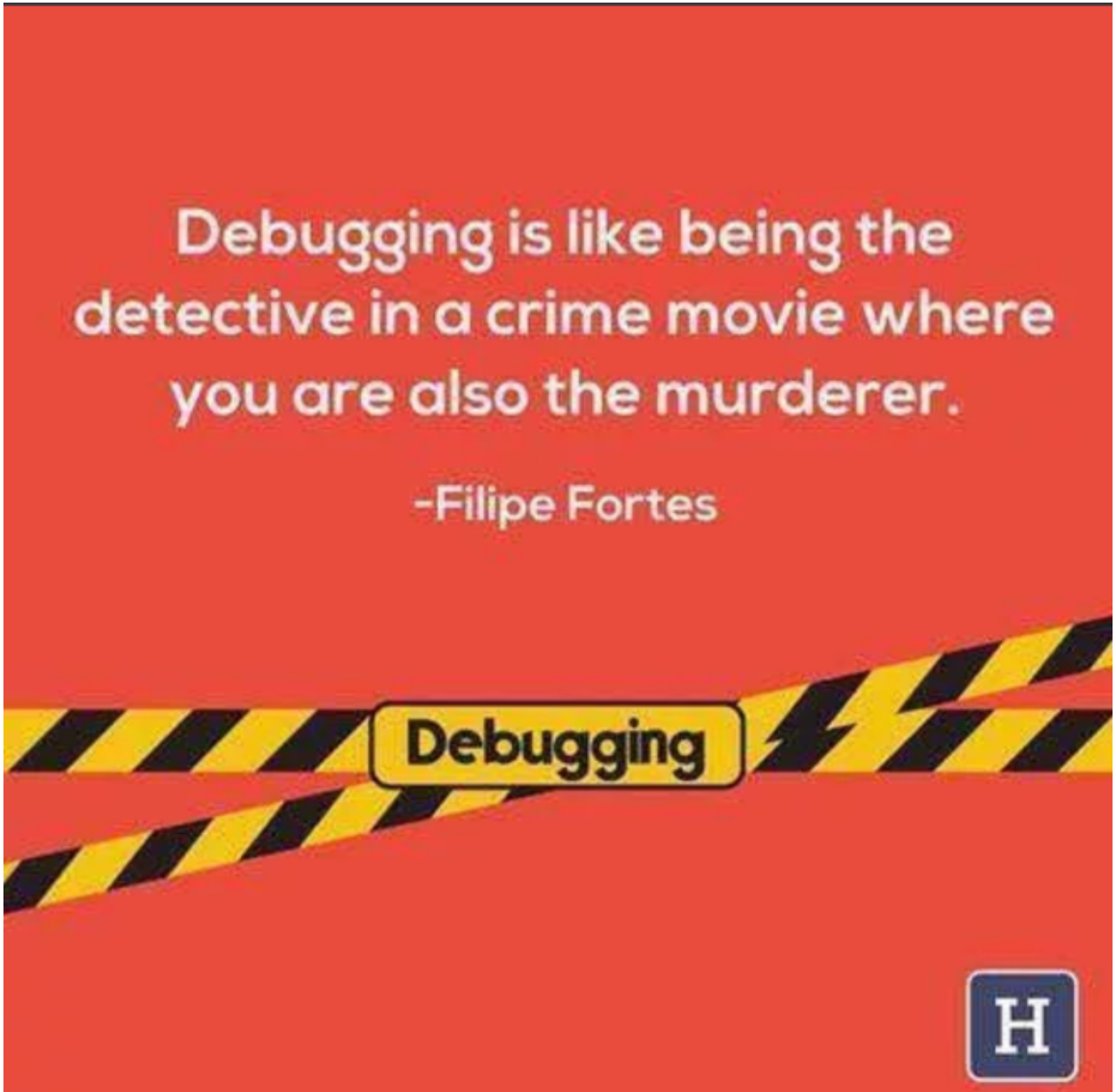
- Structure et fonctionnement interne nono connus
- Respecte à la lettre les spécifications, mais pas leur implémentation interne.
- Pour tests systèmes et tests unitaires isolés.

## Approches

- Partition des entrées : 1 entrée 1 classe
- Analyse du comportement
- Vérification de sortie attendue
- etc.

# Les tests unitaires

# Les tests unitaires



# Les tests unitaires

Vous avez tous fait ceci :

```
public void maMethode(){
 ....
 System.out.println(x);
 ...
 for(String str : list) System.out.println(str);
 ...
}
```

C'est INTERDIT !

# Les tests unitaires

Vous avez tous fait ceci :

```
public void maMethode(){
 ....
 System.out.println(x);
 ...
 for(String str : list) System.out.println(str);
 ...
}
```

**C'est INTERDIT !**

# Les tests unitaires

## Règle 1 : les logs

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
...
private static final Logger log =
     LoggerFactory.getLogger(Application.class);
log.info(maVar);
log.debug(maVar);
...
```

Très important de logger une application web !

# Les tests unitaires

**Les logs ne montrent pas tout**

Se connecter à l'application puis afficher quelque chose. Aller à telle ou telle url. Vérifier la bonne construction d'un méthode. etc.

# Les tests unitaires

## Règle 2 : les tests

Pour l'implémentation de la méthode suivante :

```
User getUser(String id);
```

On peut vérifier sa véracité.

```
@Test
public void createUser(){
 Assert.assertTrue(us.getUser("fsdfhk") instanceof User);
}
```

# Les tests unitaires

| | **Tests unitaires** | **Test Sys.o.print** |
|---|---|---|
| **Reproductible** | Oui | Non |
| **Compréhensible** | Oui | Non |
| **Documenté** | Oui | Non |
| **Externe au code** | Oui | Non (*bloated*) |
| **Conclusion** | **Production & Finalisation** | **Quick&Dirty** |

# Les test unitaires

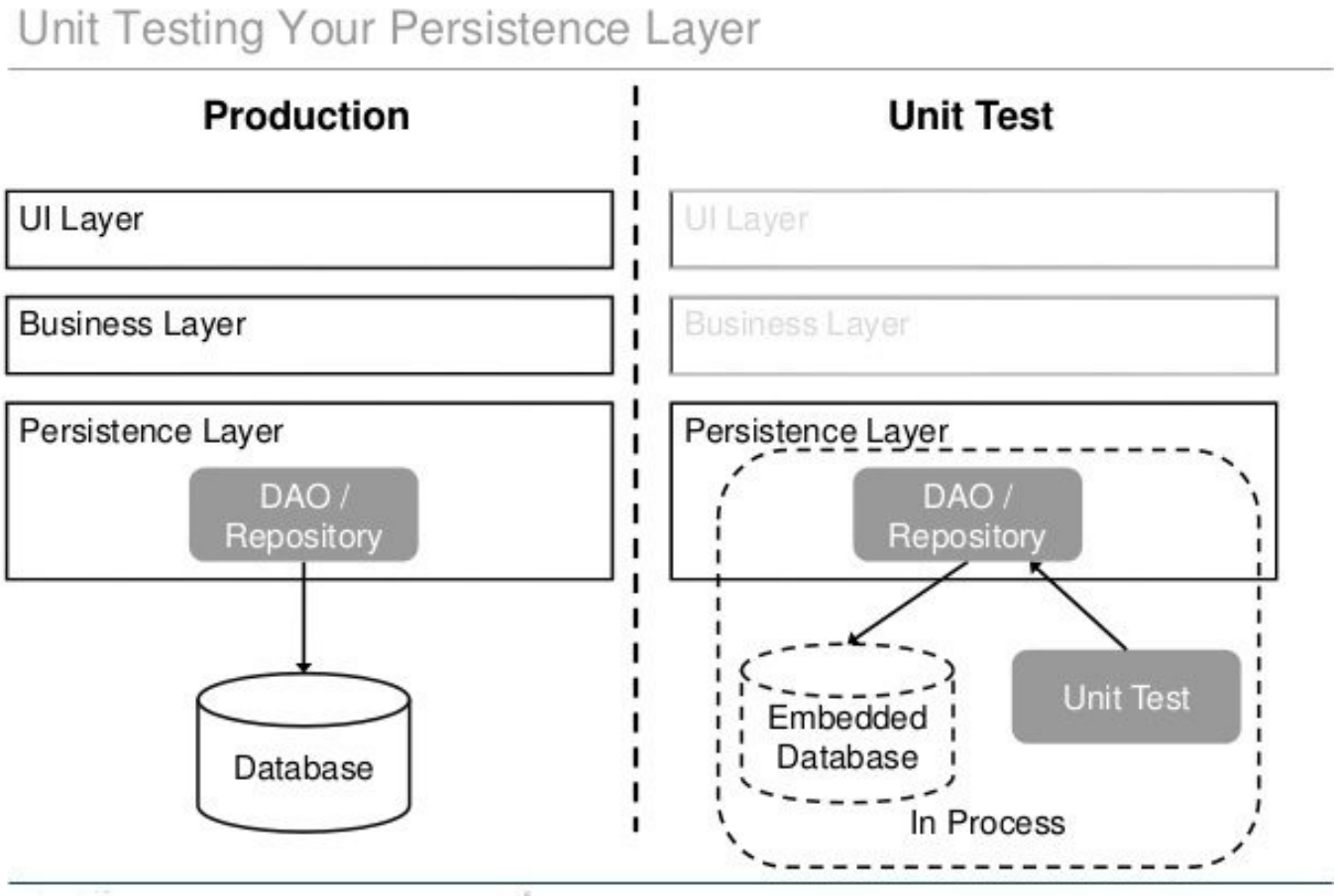
## Un test c'est :

1. Une définition du contexte
   - Factice pour des test unitaires
   - Réel pour des tests d'intégration
2. Une entrée et son état
3. Une sortie et son état
4. Une comparaison des changements d'état attendus (dit "oracle")
5. Une redirection ou non vers un état cible

## Tests et interfaces

Un test sur IMonDAO.java pourra s'appliquer à plusieurs MonDAOImpl.java possibles.

# Les test unitaires



# Les test unitaires

## Règles

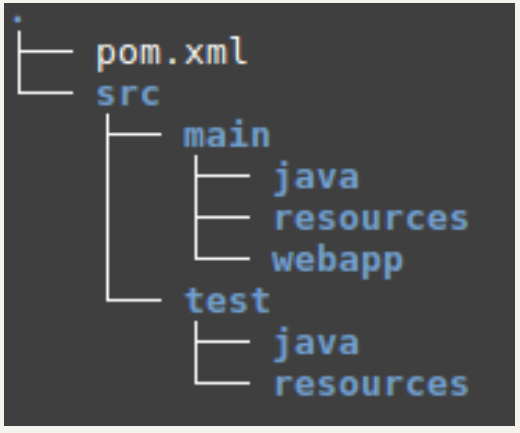
- 1 méthode = 1 test
- 1 classe java = 1 classe java de test
- Si ça marche une fois, ça marchera les autres fois (cf. SDZ)
- Si ça marche pour quelques valeurs, ça marchera pour toutes les autres (cf. SDZ)

## Objectif

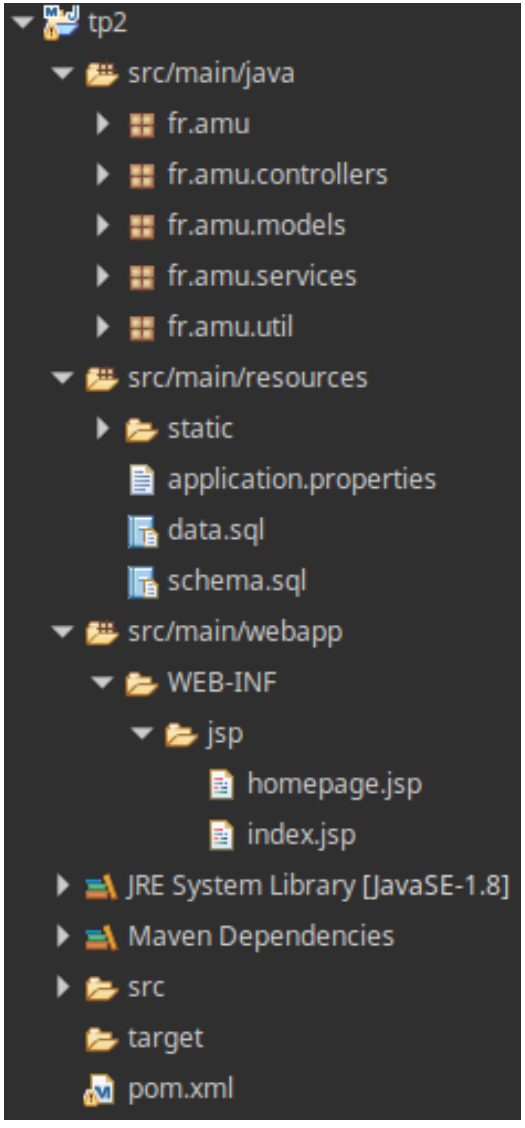
**Signaler les erreurs !** Éviter les bugs habituels, et les plus fréquents. Pouvoir se contentrer que sur les vrais bugs importants.

# Les tests unitaires

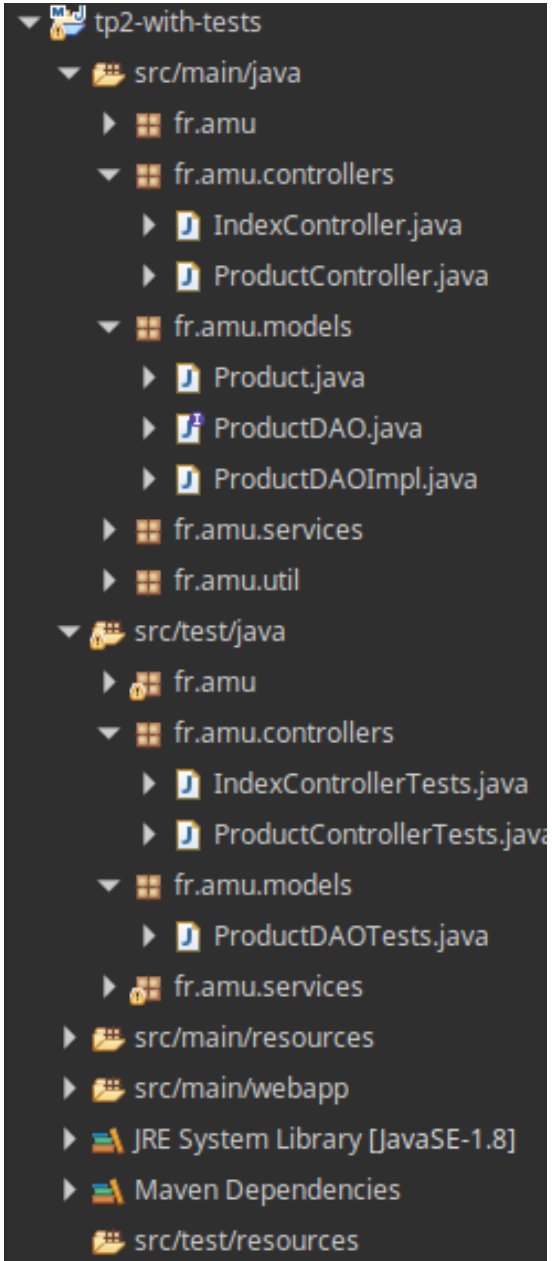
## Sous Maven



# Les tests unitaires



# Les tests unitaires



# Les test unitaires

```java
import static org.junit.Assert.*;
import org.junit.Test;

@Test
public void faire() {
    assertTrue(message, condition); // condition doit être vraie
    assertFalse(message, condition); // condition doit être fausse
    assertEquals(message, expected, actual); //pour des objets ou des
        longs
    assertNotNull(message, object); // vérifie la sortie
}
```

- Un import static de JUnit
- Une annotation @Test
- Des conditions attendues à valider Assert.assertTrue ...

# Les tests unitaires

```
@BeforeClass
public static void setUpBeforeClass() throws Exception { }

@AfterClass
public static void tearDownAfterClass() throws Exception { }

@Before
public void setUp() throws Exception { }

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

- Avant ou après tout : @BeforeClass @AfterClass
- Avant ou après chaque test : @Before @After

# Les tests unitaires et mocks

## Les mocks

- Tests avec injections de dépendances nécessaires (autres classes)
- C'est classe *Dummy* qui en remplace une autre
- Simule le contexte de test

# Les tests unitaires : vers Spring

## Heureusement, Spring est là !

# Spring et tests

# Spring et tests

## Questions en suspens

- Comment créer des tests pour des contrôleurs avec urls, GET, POST, etc ?
- Comment tester les DAO et les BDD ?

## Solutions Spring boot

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
```

# Spring et tests : l'application

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class ApplicationTests {

  @Test
  public void contextLoads() {
  }

}
```

- @RunWith(SpringRunner.class) : se lancera au démarrage de Spring
- @SpringBootTest : indique une classe de test Spring boot avec tests cachés habituels à faire
- @Test : le même qu'avant

# Spring et tests : les DAO

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class ProductDAOTests {
    // la base de données est auto remplie grâce à
         src/main/java/resources/data.sql
    @Autowired
    ProductDAO prdao;

    @Test // dire que c'est un test (annotation classique de JUnit)
    @Transactional // pour gérer les transactions
    @Rollback(true) // pour remettre la BDD dans son état initial
    public void add() {
   Integer generatedId = prdao.add(product);
   List<Product> products = prdao.findAll();
   Assert.assertEquals(generatedId,
        Integer.valueOf(products.get(products.size()-1).getId() ));
    }
```

# Spring et tests : les DAO

- @Rollback(true) : remet la BDD dans son état initial après le test
- @Transactional : il gère les transactions

# Spring et tests : les contrôleurs

```
@RunWith(SpringRunner.class)
@WebMvcTest(ProductController.class)
public class ProductControllerTests {
  @Autowired
  private ProductController controller;

  @Autowired
  MockMvc mvc;

  @MockBean
  ProductService ps;
```

# Spring et tests : les contrôleurs

- @WebMvcTest(class) : permet d'avoir accès aux requêtes, réponses, url, etc.
- MockMvc : ensemble de classes et méthodes pour simuler les requêtes, etc.
- @MockBean : injecter un mock auto-généré des beans nécessaires dans la méthode

# Spring et tests : les contrôleurs

```
@Before
public void setup() {
   InternalResourceViewResolver viewResolver = new
        InternalResourceViewResolver();
   viewResolver.setPrefix("/WEB-INF/jsp");
   viewResolver.setSuffix(".jsp");
   mvc =
        standaloneSetup(controller).setViewResolvers(viewResolver).build();
}
```

- Utilisation du MockMvc pour créer une configuration standaloneSetup
- Faite avant chaque test (@Before de JUnit)
- Fournit les chemins prefix et suffix des vues (JSP)

# Spring et tests : les contrôleurs

```
@Test
public void contexLoads() throws Exception {
   assertThat(controller).isNotNull(); // on vérifie que le
        controller est bien lancé
}
```

D'abord vérifier que le contrôleur s'initialise bien.

# Spring et tests : les contrôleurs

```
@Test
public void add() throws Exception {
   mvc.perform(post("/add").contentType(MediaType.APPLICATION_FORM_URLENCODED
        "value")
         .param("category", "value").param("description", "value"))
   .andExpect(view().name("homepage"));
}
```

- Vérifier chaque méthode en donnant des paramètres factices .param()
- Choisir une méthode HTTP et une URL : post("/add")
- Donner la vue attendue : andExpect(view().name("index"))
  - SI vue, paramètres, requête et url sont corrects : TRUE : ACCEPT
  - SINON : False : FAIL

# Spring et tests : les services

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class ProductServiceTests {
   @Autowired
   ProductService ps;

   @Autowired
   ProductDAO prdao;
```

Dépendances si nécessaires

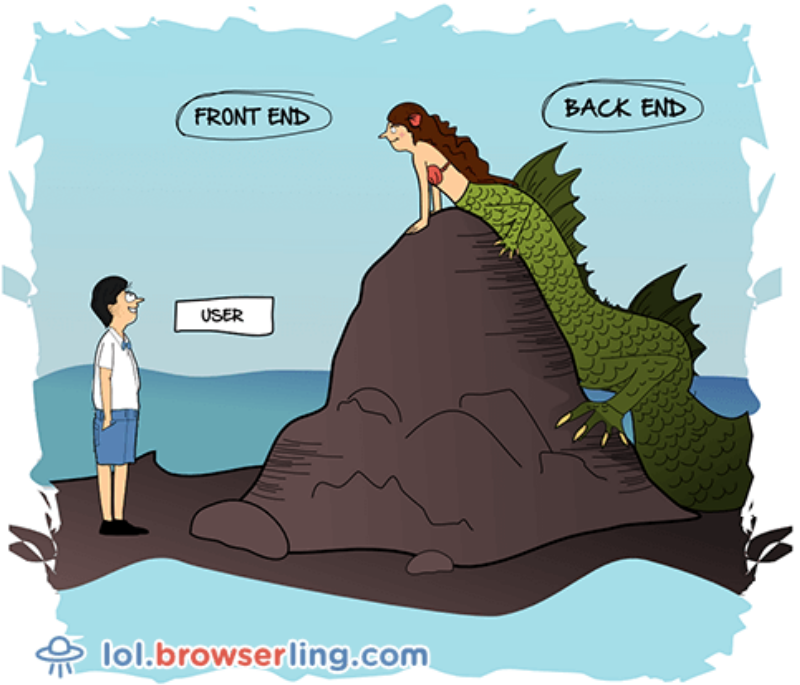
# Spring et tests : les services

```
@Test
public void getProduct() {
   ps.addProduct(product);
   Product product = ps.getProduct(0);
   Assert.assertTrue(ps.getProduct(0) instanceof Product);
   Assert.assertTrue(product.getId() == 0);
}
```

Tests classiques de JUnit : les services sont des traitements JavaSE bien souvent.

# A vous de jouer !

Reprenez **AMUzone** et intégrez-y vous **tests** !




Front end vs. Back end.

# Sources

- Documentation Oracle
- OpenClassroom : Hedi Radhouane
- Cours de Paris7 : Mihaela Sighireanu