

## Formation tests unitaires

Olivier LEVITT à partir du travail d'Emmanuel L' HOUR,  
Sébastien LICHTENAUER et Mélanie MARTIN

# Sommaire

- 1 Pourquoi ?
  - Pourquoi tester ?
  - Revendications
- 2 JUnit
  - Mise en place
  - Utilisation
  - Exercice
- 3 DBUnit
  - DBUnit
  - Exercice
- 4 Mockito
  - Mockito
  - Exercice
- 5 Bien tester
  - Quoi tester
  - Rentabiliser les tests
- 6 Conception par les tests
  - Approches

# Sommaire

- 1 Pourquoi ?
  - Pourquoi tester ?
  - Revendications
- 2 JUnit
  - Mise en place
  - Utilisation
  - Exercice
- 3 DBUnit
  - DBUnit
  - Exercice
- 4 Mockito
  - Mockito
  - Exercice
- 5 Bien tester
  - Quoi tester
  - Rentabiliser les tests
- 6 Conception par les tests
  - Approches

# Pourquoi tester ?

En développement :

- S'assurer que ce qu'on code fonctionne
- S'assurer que ce qu'on a déjà codé continue de fonctionner

En maintenance

- Vérifier que le bug est résolu
- S'assurer qu'il ne se reproduira plus
- Effets de bord / régressions ?

Test “**unitaire**” ?

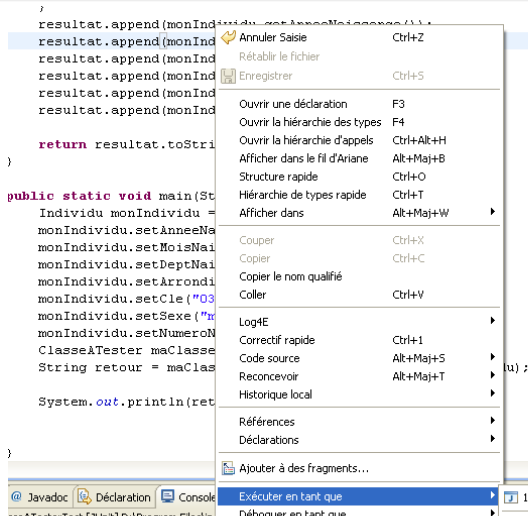
# Le test manuel, exemple

```
1 public String recupererDebutNumeroSS(Individu monIndividu) throws
   MonAppliException {
2     StringBuffer resultat = new StringBuffer();
3     if (monIndividu != null) {
4         if ("M".equals(monIndividu.getSexe())) {
5             resultat.append("1");
6         } else {
7             resultat.append("2");
8         }
9         resultat.append(monIndividu.getAnneeNaissance());
10        resultat.append(monIndividu.getMoisNaissance());
11        resultat.append(monIndividu.getDeptNaissance());
12        resultat.append(monIndividu.getArrondissementNaissance());
13        resultat.append(monIndividu.getNumeroNaissance());
14        resultat.append(monIndividu.getCle());
15    } else {
16        throw new MonAppliException("Pas d'individu");
17    }
18    return resultat.toString();
19 }
```

# Le test manuel, exemple

```
1 public static void main(String[] args) throws Exception {  
2     Individu monIndividu = new Individu();  
3     monIndividu.setAnneeNaissance("1983");  
4     monIndividu.setMoisNaissance("01");  
5     monIndividu.setDeptNaissance("75");  
6     monIndividu.setArrondissementNaissance("014");  
7     monIndividu.setCle("03");  
8     monIndividu.setSexe("m");  
9     monIndividu.setNumeroNaissance("004");  
10    ClasseATester maClasse = new ClasseATester();  
11    String retour = maClasse.recupererDebutNumeroSS(monIndividu);  
12  
13    System.out.println(retour);  
14  
15 }
```

# Le test manuel, exemple



# Le test manuel, exemple





# Le test manuel, conclusion

- Coûteux
- Partiel
- Sujet à erreurs
- Non-collaboratif
- Non-durable

# Revendications

Qu'est ce qu'on veut ?

- Efficacité
- Carré
- Cadré
- Reproductibilité
- Automatisation
- Isolation
- Durabilité
- Collaboration

Et quand est ce qu'on le veut ?

- **Maintenant !**

# Sommaire

1

## Pourquoi ?

- Pourquoi tester ?
- Revendications

2

## JUnit

- Mise en place
- Utilisation
- Exercice

3

## DBUnit

- DBUnit
- Exercice

4

## Mockito

- Mockito
- Exercice

5

## Bien tester

- Quoi tester
- Rentabiliser les tests

6

## Conception par les tests

- Approches

# Mise en place

Comme d'hab, une dépendance

```
1 <dependency>
2 <groupId>junit</groupId>
3 <artifactId>junit</artifactId>
4 <version>4.4</version>
5 <scope>test</scope>
6 </dependency>
```

- Et la version 3 ?
- Et la version 5 ?

Pourquoi ?

**JUnit**

DBUnit

Mockito

Bien tester

Conception par les tests

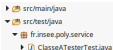
Mise en place

Utilisation

Exercice

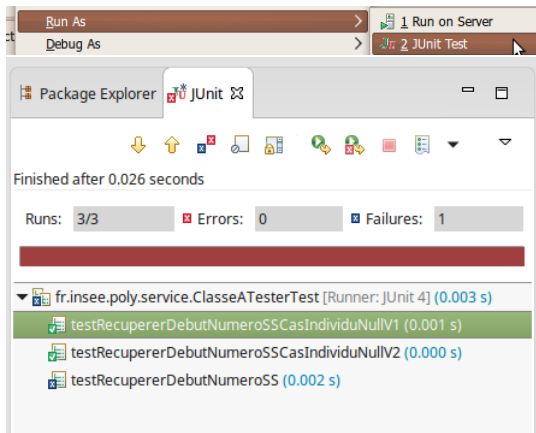
# Configuration

# Hello world



```
1  import static org.junit.Assert.assertEquals;
2  import org.junit.Test;
3
4  public class ClasseATesterTest {
5      @Test
6      public void testRecupererDebutNumeroSS() {
7          // GIVEN
8          Individu monIndividu = new Individu();
9          monIndividu.setAnneeNaissance("1983");
10         monIndividu.setDeptNaissance("75");
11         monIndividu.setMoisNaissance("01");
12         monIndividu.setArrondissementNaissance("014");
13         monIndividu.setCle("03");
14         monIndividu.setSexe("m");
15         monIndividu.setNumeroNaissance("004");
16         ClasseATester maClasse = new ClasseATester();
17
18         // WHEN
19         String retour = maClasse.recupererDebutNumeroSS(monIndividu);
20
21         // THEN
22         assertEquals("183017411400403", retour);
23     }
24 }
```

# Hello world



# Utilisation

- Le dossier src/test
- Conventions
- Given, when, then
- C'est quoi un import static ?
- 1 assert = 1 test
- `Assert.assertEquals(["message erreur"], resultatAttendu, resultatReel)`
- `Assert.assertTrue(["message erreur"], expression)`
- `fail()`



# Utilisation avancée, les hooks

```
1 public class ClasseATesterTest {
2     @BeforeClass
3     public void beforeClass() {
4         //Une seule fois au debut
5     }
6
7     @Before
8     public void before() {
9         //Avant chaque test
10    }
11
12    @After
13    public void after() {
14        //Après chaque test
15    }
16
17    @AfterClass
18    public void afterClass() {
19        //Une seule fois a la fin
20    }
21
22    @Test
23    public void testMethode() {
24
25    }
26 }
```

# Utilisation avancée, les exceptions

```
1 public class ClasseATesterTest {
2
3     @Test(expected = IllegalArgumentException.class)
4     public void testExplosif() throws Exception {
5         racineCarree(-10);
6     }
7 }
```

## Alternative :( :

```
1 public class ClasseATesterTest {
2
3     @Test
4     public void testExplosif() {
5         try {
6             racineCarree(-10);
7             fail();
8         }
9         catch (IllegalArgumentException e) {
10             assertEquals("Nombre inferieur a 0", e.getMessage());
11         }
12     }
13 }
```

# Exercice

- Parcourir le module 1 pour le fun
- Module 2 package DbUnit : en mode maintenicien
- Tester en l'état la méthode `isSoldeSuffisant()` de `ServiceCompte` dans la classe `ServiceCompteTest`

# Sommaire

1

## Pourquoi ?

- Pourquoi tester ?
- Revendications

2

## JUnit

- Mise en place
- Utilisation
- Exercice

3

## DBUnit

- DBUnit
- Exercice

4

## Mockito

- Mockito
- Exercice

5

## Bien tester

- Quoi tester
- Rentabiliser les tests

6

## Conception par les tests

- Approches

## Previously on les tests unitaires

```
1 public class ClasseATesterTest {
2     @Test
3     public void testRecupererDebutNumeroSS() {
4         // GIVEN
5         Individu monIndividu = new Individu();
6         monIndividu.setAnneeNaissance("1983");
7         monIndividu.setDeptNaissance("75");
8         monIndividu.setMoisNaissance("01");
9         monIndividu.setArrondissementNaissance("014");
10        monIndividu.setCle("03");
11        monIndividu.setSexe("m");
12        monIndividu.setNumeroNaissance("004");
13        ClasseATester maClasse = new ClasseATester();
14
15        // WHEN
16        String retour = maClasse.recupererDebutNumeroSS(monIndividu);
17
18        // THEN
19        assertEquals("183017411400403", retour);
20    }
21 }
```

# Nouvelle problématique

- Notre méthode touche à la base de données
- Par exemple, elle désactive un compte

```
1 public void desactiverCompte(String identifiant) {  
2     PreparedStatement ps = connection.prepareStatement("UPDATE utilisateurs SET  
3     actif = 0 WHERE identifiant = ?");  
4     ps.setString(1, identifiant);  
5     ps.executeUpdate();  
6 }
```

# 1ère approche

```
1 public class ClasseATesterTest {
2     @Test
3     public void testDesactiverCompte() {
4         // GIVEN
5
6         // WHEN
7         desactiverCompte("ABCDEF");
8
9         // THEN
10        assertFalse("Le compte est maintenant inactif", isActif("ABCDEF"));
11    }
12 }
```

## 2ème approche

```
1 public class ClasseATesterTest {
2     @Test
3     public void testDesactiverCompte() {
4         // GIVEN
5
6         // WHEN
7         desactiverCompte("ABCDEF");
8
9         // THEN
10        assertFalse("Le compte est maintenant inactif", isActif("ABCDEF"));
11    }
12
13    @Before
14    public void avantTests() {
15        activerTousLesComptes();
16    }
17
18    @After
19    public void afterTests() {
20        activerTousLesComptes();
21    }
22 }
```



# C'est quoi les problèmes ?

- Pas de reproductibilité
- Pas d'isolation
- Travail en équipe ?
- Maintenabilité ?

=> DBUnit se charge de réinitialiser la BDD

# Mise en place

Comme d'hab, une dépendance

```
1 <dependency>
2 <groupId>org.dbunit</groupId>
3 <artifactId>dbunit</artifactId>
4 <version>2.4.8</version>
5 <scope>test</scope>
6 </dependency>
```

# Configuration

```
1 <dataset>
2 <compte numero="1" montant="1000" auto_decouv="100" id_client="1" />
3 <compte numero="2" montant="300" auto_decouv="200" id_client="1" />
4 <compte numero="3" montant="1000" auto_decouv="100" id_client="1" />
5 <compte numero="5" montant="50" auto_decouv="0" id_client="9" />
6 <client id="1" nom="Jean" prenom="Dupont" />
7 <compteConnu id_numero="5" id_client="1" />
8 </dataset>
```

- Le dossier test/resources
- 1 fichier par ensemble de classes
- Le schéma est défini implicitement

# Utilisation

On prend une connexion et on en fait une connexion DBUnit :

```
1 DatabaseConnection myDbConnection = new DatabaseConnection(connection);
```

Ensuite on charge le XML :

```
1 FlatXmlProducer producer = new FlatXmlProducer(new  
2 InputSource("src/test/resources/fr/insee/DbUnit/dao/impl/CompteDAOTest.xml"));  
3 IDataset mySetUpDataset = new FlatXmlDataSet(producer);
```

Enfin on peut insérer les données :

```
1 DatabaseOperation.CLEAN_INSERT.execute(myDbConnection, mySetUpDataset);
```

- Où placer ce code ?

# Problem solved ?

Et l'isolation ?

- Les bases en mémoire (H2, HSQLDB)

La vie en rose ?

- Limitations (FULL OUTER JOIN)
- SQL non standard :(
- Reproduire tout le schéma ?
- Maintenir le schéma ?

# Utilisation avancée

- Le schéma est dérivé de la première ligne
- NULL : ne rien mettre
- Sensible à la casse

# Exercice

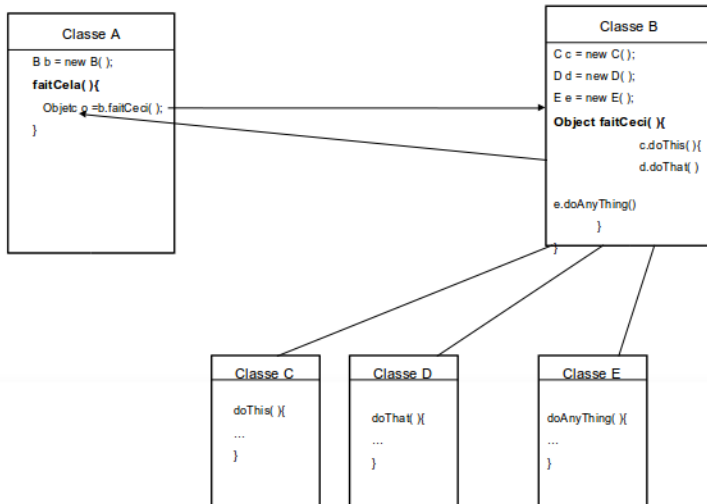
- Tester CompteDAO  
(findIdComptesDetenusEtConnusByIdClient puis update,  
ajouterCompteAuClient)

# Sommaire

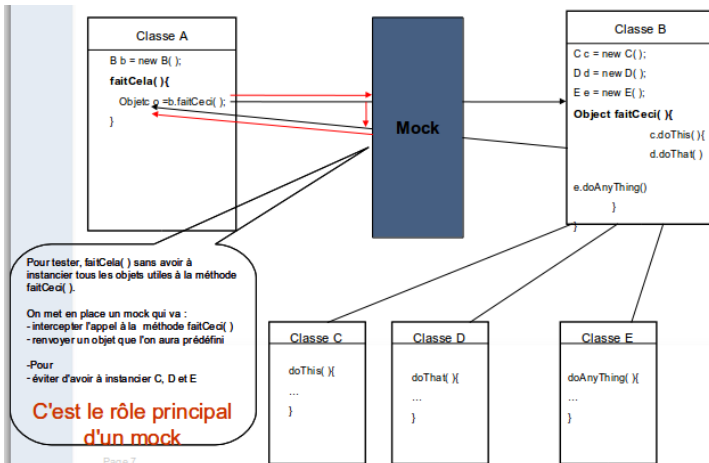
- 1 Pourquoi ?
  - Pourquoi tester ?
  - Revendications
- 2 JUnit
  - Mise en place
  - Utilisation
  - Exercice
- 3 DBUnit
  - DBUnit
  - Exercice
- 4 **Mockito**
  - **Mockito**
  - **Exercice**
- 5 Bien tester
  - Quoi tester
  - Rentabiliser les tests
- 6 Conception par les tests
  - Approches



# Nouvelle problématique



# Mock



# Comment ça marche

- Mock = objet simulé
- Reproduire le comportement de façon contrôlée
- Flexibilité
- Simulacre, émulateur, proxy, stub, dummy, fake, espion ?

# Le choix de la bibliothèque

- Jmock
- Easymock
- Mockito

A l'INSEE : Mockito

# Cas d'utilisation classiques

- Math.random
- new Date()
- DAO / SQLException
- ...

# Mise en place

Comme d'hab, une dépendance

```
1 <dependency>  
2 <groupId>org.mockito</groupId>  
3 <artifactId>mockito-all</artifactId>  
4 <version>1.8.5</version>  
5 <scope>test</scope>  
6 </dependency>
```

Pourquoi ?

JUnit

DBUnit

**Mockito**

Bien tester

Conception par les tests

**Mockito**

Exercice

# Configuration

# Exemple

```
1 public class ComptesServiceImpl implements IComptesService {
2
3     private IComptesDAO comptesDAO;
4
5     public ComptesServiceImpl() {
6         comptesDAO = new ComptesDAOImpl();
7     }
8
9     public void setComptesDAO(IComptesDAO comptesDAO) {
10         this.comptesDAO = comptesDAO;
11     }
12
13     //Methode a tester
14     public int getSoldeTotal() {
15         List<Compte> comptes = comptesDAO.getAll();
16         int soldeTotal = 0;
17         for (Compte compte : comptes) {
18             soldeTotal += compte.getSolde();
19         }
20         return soldeTotal;
21     }
22 }
```

- Comment tester `getSoldeTotal()` ?



# Exemple

```
1 public class CompteServiceImplTest {
2
3     private ICompteService compteService;
4
5     @Test
6     public void getSoldeTotalTest() {
7         //GIVEN
8         List<Compte> comptesDeTest = new ArrayList<Compte>();
9         comptesDeTest.add(new Compte(20));
10        comptesDeTest.add(new Compte(10));
11        comptesDeTest.add(new Compte(-5));
12
13        compteService = new CompteServiceImpl();
14        ICompteDAO compteDAO = Mockito.mock(ICompteDAO.class);
15        Mockito.when(compteDAO.getAll()).thenReturn(listeComptes);
16        compteService.setCompteDAO(compteDAO);
17
18        //WHEN
19        int solde = compteService.getSoldeTotal();
20
21        //THEN
22        assertEquals(25, solde);
23    }
24 }
```

# Personnalisation du comportement

- Par défaut : ne fait rien / “nice values”

```
1 Mockito.when(mockRandomService.random()).thenReturn(0.5);
2 Mockito.when(mockRandomService.randomInt(10)).thenReturn(5);
3 Mockito.when(mockFichierService.lireFichier()).thenThrow(new
4 FileNotFoundException());
5 Mockito.when(mockRandomService.random()).thenCallRealMethod();
```

# Les matchers

```
1 Mockito.when(mockRandomService.randomInt(6)).thenReturn(5);
2 Mockito.when(mockRandomService.randomInt(7)).thenReturn(5);
3 Mockito.when(mockRandomService.randomInt(8)).thenReturn(5);
4 Mockito.when(mockRandomService.randomInt(Matchers.anyInt())).thenReturn(5);
```

```
1 Mockito.when(mockRandomService.randomInt(2,5)).thenReturn(3);
2 Mockito.when(mockRandomService.randomInt(2,6)).thenReturn(3);
3 Mockito.when(mockRandomService.randomInt(2,7)).thenReturn(3);
4 Mockito.when(mockRandomService.randomInt(Matchers.anyInt(),Matchers.anyInt())).
    thenReturn(3);
```

## D'autres goodies

- DISCLAIMER

```
1 Mockito.verify(mockRandomService, times(2)).randomInt(Matchers.anyInt());
```

```
1 InOrder ordre = Mockito.inOrder(mockRandomService);  
2 ordre.verify(mockRandomService).randomInt(2);  
3 ordre.verify(mockRandomService).randomInt(5);
```

- Powermock ?
- DISCLAIMER

# Injection de dépendances

```
1 public class ExempleDependant {
2
3     private IService service;
4
5     public ExempleDependant() {
6         service = new ServiceImpl();
7     }
8
9     public void methodeATester() {
10         service.whatever();
11     }
12 }
```

# Injection de dépendances, fait main

```
1 public class ExempleDependant {
2
3     private IService service;
4
5     public ExempleDependant() {
6         service = new ServiceImpl();
7     }
8
9     public ExempleDependant(IService service) {
10         //Constructeur utile aux tests
11         this.service = service;
12     }
13
14     public void methodeATester() {
15         service.whatever();
16     }
17
18     //Setter utile aux tests
19     public void setService(IService service) {
20         this.service = service;
21     }
22 }
```

## ● Ca fait envie ?

# Injection de dépendances, comme les pros

- Frameworks modernes
- Ex : spring

# Exercice

- Tester `recupererClientById` de la classe `ServiceClient`
- Y compris les cas aux limites
- Utiliser la classe déjà présente `ServiceClientTest`



# Sommaire

- 1 Pourquoi ?
  - Pourquoi tester ?
  - Revendications
- 2 JUnit
  - Mise en place
  - Utilisation
  - Exercice
- 3 DBUnit
  - DBUnit
  - Exercice
- 4 Mockito
  - Mockito
  - Exercice
- 5 Bien tester**
  - Quoi tester**
  - Rentabiliser les tests**
- 6 Conception par les tests
  - Approches

# Quoi tester ?

- Public, private, protected ?
- Le contrat fait foi
- Couverture des tests
- Getters / Setters ?
- Service, DAO ?
- Code legacy ?
- Test first ?

# Rentabiliser les tests

- Intégration avec la MOA
- Autodocumentation
- Passage en maintenance
- Changement d'environnement
- PIC

# Outillage

- Intégration dans Eclipse
- Plugins
- Alertes
- Démo ! (Eclemma, Infinitest, Moreunit)

# Sommaire

- 1 Pourquoi ?
  - Pourquoi tester ?
  - Revendications
- 2 JUnit
  - Mise en place
  - Utilisation
  - Exercice
- 3 DBUnit
  - DBUnit
  - Exercice
- 4 Mockito
  - Mockito
  - Exercice
- 5 Bien tester
  - Quoi tester
  - Rentabiliser les tests
- 6 Conception par les tests
  - Approches

# Un principe, des mouvances

- Tests first
- Only tests we trust

Décliné en tous les sigles possibles :

- TDD
- BDD
- DDD
- ATDD
- TDR

Pourquoi ?

JUnit

DBUnit

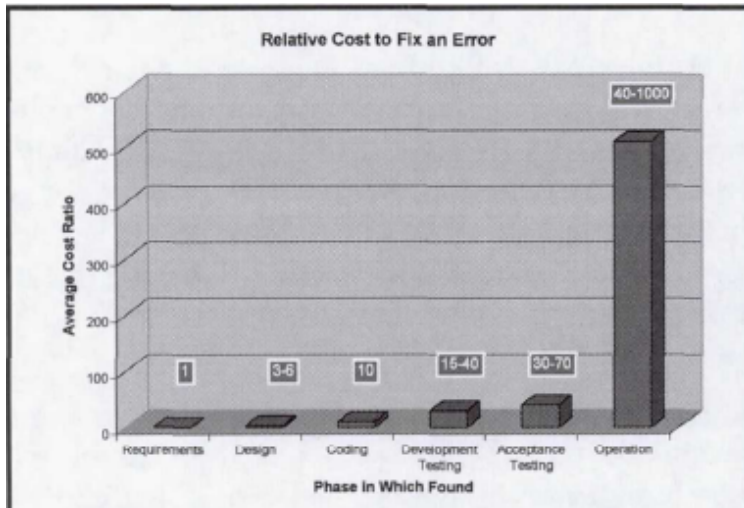
Mockito

Bien tester

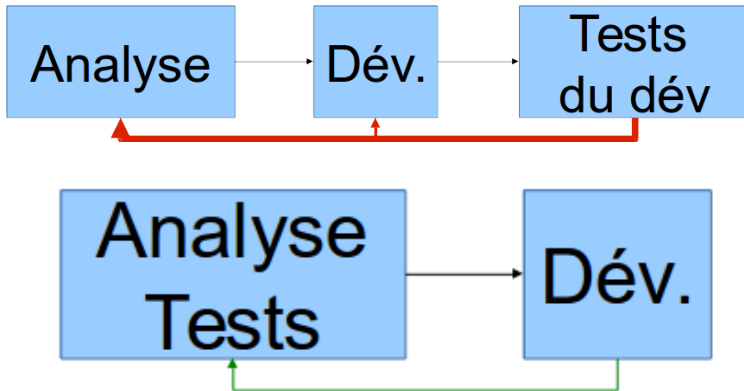
Conception par les tests

Approches

# Un constat



## Un constat

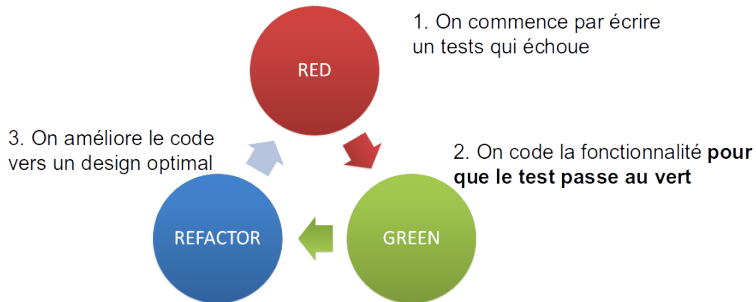




# TDD

› *Tests Driven Design* : Red – Green - Refactor

› Conception orientée par les tests



## Quand ça marche bien

- Documentation exhaustive et exécutable
- Confiance
- Qualité de code
- Couverture de test : 100%

## Plus loin, BDD / DDD

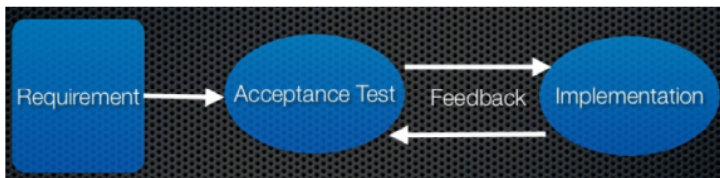
- Tester les comportements plutôt que les méthodes
- Discussion avec MOA
- Esprit boîte noire
- Plus trop unitaire
- Adieu le 100%

## Plus loin, TDR

- Specs pilotées par les tests
- Clarifie la spec
- Cas de test tout prêt
- Recette réduite (déplacée)

## Encore plus loin, ATDD

- Conception orientée par les tests fonctionnels
- Outils : Selenium, FitNesse, Cucumber



## Encore plus loin, ATDD

