Tests

Programmation avancée – Java

F5 – ISIMA 2020/2021

Olivier Goutet
o.goutet@openium.fr

13 octobre 2020

Introduction – Tests

- Des tests ?

Introduction – Tests

- Des tests ?
- Pourquoi tester ?

Introduction – Tests

- Des tests ?
- Pourquoi tester ?
- Comment tester ?

Introduction – Tests

- Des tests ?
- Pourquoi tester ?
- Comment tester ?
- Fondamental dans le développement logiciel moderne
  - Pour améliorer la qualité
  - Pour développer plus vite
  - Pour éviter les régressions

Pourquoi ne pas tester

- C'est trop long de tester

Pourquoi ne pas tester

- C'est trop long de tester
  - Pas si on le fait dès le début du projet
  - En prenant un peu de temps a tester tout au long du développement on gagne **beaucoup** de temps a la fin

Pourquoi ne pas tester

- C'est trop long de tester
  - Pas si on le fait dès le début du projet
  - En prenant un peu de temps a tester tout au long du développement on gagne **beaucoup** de temps a la fin
- Les test mettent trop de temps a se lancer

Pourquoi ne pas tester

- C'est trop long de tester
  - Pas si on le fait dès le début du projet
  - En prenant un peu de temps a tester tout au long du développement on gagne **beaucoup** de temps a la fin
- Les test mettent trop de temps a se lancer
  - ça doit être instantané ou presque
  - Sortir les très longs tests des jeu de tests courants

Pourquoi ne pas tester

- C'est trop long de tester
  - Pas si on le fait dès le début du projet
  - En prenant un peu de temps a tester tout au long du développement on gagne **beaucoup** de temps a la fin
- Les test mettent trop de temps a se lancer
  - ça doit être instantané ou presque
  - Sortir les très longs tests des jeu de tests courants
- Ce n'est pas mon boulot de tester mon code

Pourquoi ne pas tester

- C'est trop long de tester
  - Pas si on le fait dès le début du projet
  - En prenant un peu de temps a tester tout au long du développement on gagne **beaucoup** de temps a la fin
- Les test mettent trop de temps a se lancer
  - ça doit être instantané ou presque
  - Sortir les très longs tests des jeu de tests courants
- Ce n'est pas mon boulot de tester mon code
  - Le boulot d'un développeur est de faire du code qui marche
  - Si votre code est toujours buggué et ralenti tout le monde...

Pourquoi ne pas tester

- C'est trop long de tester
  - Pas si on le fait dès le début du projet
  - En prenant un peu de temps a tester tout au long du développement on gagne **beaucoup** de temps a la fin
- Les test mettent trop de temps a se lancer
  - ça doit être instantané ou presque
  - Sortir les très longs tests des jeu de tests courants
- Ce n'est pas mon boulot de tester mon code
  - Le boulot d'un développeur est de faire du code qui marche
  - Si votre code est toujours buggué et ralenti tout le monde...
- Je ne sais pas trop comment le code est supposer se comporter donc je ne peux pas le tester

Pourquoi ne pas tester

- C'est trop long de tester
  - Pas si on le fait dès le début du projet
  - En prenant un peu de temps a tester tout au long du développement on gagne **beaucoup** de temps a la fin
- Les test mettent trop de temps a se lancer
  - ça doit être instantané ou presque
  - Sortir les très longs tests des jeu de tests courants
- Ce n'est pas mon boulot de tester mon code
  - Le boulot d'un développeur est de faire du code qui marche
  - Si votre code est toujours buggué et ralenti tout le monde...
- Je ne sais pas trop comment le code est supposer se comporter donc je ne peux pas le tester
  - Ne pas s'en servir si on ne le connais pas
  - Prendre le temps d'apprendre

Pourquoi ne pas tester

- Mais ça compile

Pourquoi ne pas tester

- Mais ça compile
  - ...
  - Différence entre une analyse textuelle et une exécution

Pourquoi ne pas tester

- Mais ça compile
  - ...
  - Différence entre une analyse textuelle et une exécution
- Je suis payé pour écrire du code, pas de le tester

Pourquoi ne pas tester

- Mais ça compile
  - ...
  - Différence entre une analyse textuelle et une exécution
- Je suis payé pour écrire du code, pas de le tester
  - Pas payé non plus pour débugguer toute la journée
  - Vous êtes payés pour faire du code qui marche

Pourquoi ne pas tester

- Mais ça compile
  - ...
  - Différence entre une analyse textuelle et une exécution
- Je suis payé pour écrire du code, pas de le tester
  - Pas payé non plus pour débugguer toute la journée
  - Vous êtes payés pour faire du code qui marche
- Je me sens coupable de prendre le travail de l'équipe de la qualité

Pourquoi ne pas tester

- Mais ça compile
  - ...
  - Différence entre une analyse textuelle et une exécution
- Je suis payé pour écrire du code, pas de le tester
  - Pas payé non plus pour débugguer toute la journée
  - Vous êtes payés pour faire du code qui marche
- Je me sens coupable de prendre le travail de l'équipe de la qualité
  - Tests unitaires != tests de la qualité

Pourquoi ne pas tester

- Mais ça compile
  - ...
  - Différence entre une analyse textuelle et une exécution
- Je suis payé pour écrire du code, pas de le tester
  - Pas payé non plus pour débugguer toute la journée
  - Vous êtes payés pour faire du code qui marche
- Je me sens coupable de prendre le travail de l'équipe de la qualité
  - Tests unitaires != tests de la qualité
- Mon entreprise ne me laissera pas exécuter les tests sur le système.

Pourquoi ne pas tester

- Mais ça compile
  - ...
  - Différence entre une analyse textuelle et une exécution
- Je suis payé pour écrire du code, pas de le tester
  - Pas payé non plus pour débugguer toute la journée
  - Vous êtes payés pour faire du code qui marche
- Je me sens coupable de prendre le travail de l'équipe de la qualité
  - Tests unitaires != tests de la qualité
- Mon entreprise ne me laissera pas exécuter les tests sur le système.
  - Un test doit être local, sur la machine de développement
  - Ne doit pas être dépendant d'un système extérieur dans la mesure du possible

Plan

Plan

Framework de test

- En fonction du language
  - Java – Junit
  - C# – Nunit
  - C/C++ – Cunit
  - PHP – PhpUnit
- Basé sur des assertions
  - assertTrue(valeur)
  - assertEquals(valeur1,valeur2)
  - assertNull(valeur)
  - ...

Plan

Calculatrice

```
public class Calculatrice {

    public int add(int val1, int val2){
        return val1 + val2;
    }

    public int mul(int val1, int val2){
        return val1 * val2;
    }
}
```

Calculatrice Tests

```
public class CalculatriceTest extends TestCase {
    private Calculatrice mCalculatrice;
    protected void setUp() throws Exception {
        super.setUp();
        mCalculatrice = new Calculatrice();
    }
    protected void tearDown() throws Exception {
        super.tearDown();
        mCalculatrice = null;
    }
    public void testAdd(){
        assertEquals(4,mCalculatrice.add(3,1));
        assertEquals(0,mCalculatrice.add(0,0));
    }
    public void testMul(){
        assertEquals(6,mCalculatrice.mul(3, 2));
        assertEquals(0,mCalculatrice.mul(0, 0));
        assertEquals(1,mCalculatrice.mul(1,1));
    }
}
```

Calculatrice Tests – Exécution

- Compiler le test

```
javac -cp .:junit-4.XX.jar CalculatorTest.java
```

- Exécuter le test

```
java -cp .:junit-4.XX.jar org.junit.runner.JUnitCore
     CalculatorTest
```

Les exemples calculatrices

- Ne correspondent pas à la réalité
- La question n'est pas comment faire une assertion, mais comment écrire du code testable
- Code testable
  - Du code calculatrice !
  - Des classes, des méthodes où l'on peux faire varier les arguments
  - Des méthodes où on a une valeur de retour
  - Peu de dépendances
- Concevoir dès le début du code testable
  - Ecrire les tests avant de coder ?

Code non testable

```
public class Parser {

    public void parse() throws Exception {
        Reader reader = new FileReader("/tmp/toto.txt");
        BufferedReader br = new BufferedReader(reader);
        String line;
        while ((line = br.readLine())!= null){
            String[] parts = line.split(",");
            Etudiant etudiant = new Etudiant();
            etudiant.nom = parts[0];
            etudiant.prenom = parts[1];
            // ...
            mListEtudiant.add(etudiant);
        }
        br.close();
    }
}
```

Plan

Types de tests

- Tests unitaires
- Tests fonctionnels
- Tests d'intégration
- Tests de performance
- Tests de charge
- Tests fumée
- Tests de client simulé

Types de tests – Tests unitaires

- Tester une seule classe
- Pas de dépendances sur d'autres classes
- Valide une seule fonction

Ex calculatrice

Types de tests – Tests fonctionnels

- Tester un ensemble de classes et leurs intéractions
- Tester un module de l'application
- Tester toute l'application

Ex Tester que la température a clermont est bien de 8 degré

Types de tests – Tests d'intégration

- Tester l'intégration d'un autre produit avec votre programme
- Tester le fonctionnement d'une nouvelle version d'une base de donnée

Plan

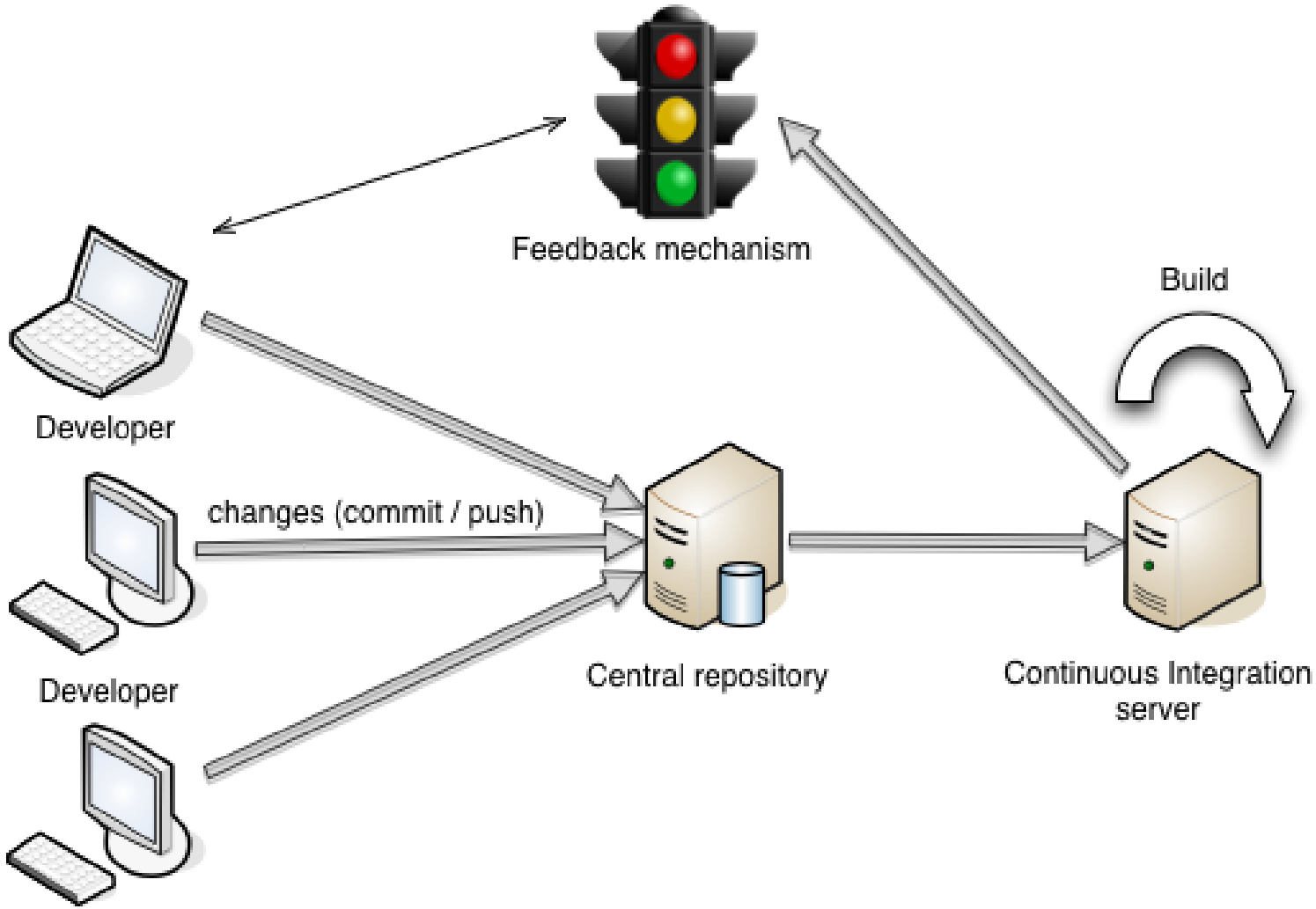
Test en entreprise – Le test dans la vraie vie

- Intégration continue
- Mock Class
- Couverture de code

Intégration continue

- Système qui lance les tests automatiquement pour vous a chaque changement du dépôt
- Nécessite
  1. Un gestionnaire de version (SVN, GIT, ...)
  2. Des scripts de build
  3. Un serveur mail
- Idéalement
  1. Des machines dédiées
  2. Une lava lamp
  3. Un girophare

Intégration continue



Classe de mock

- Wikipedia : Les mocks sont des objets simulés qui reproduisent le comportement d'objets réels de manière contrôlée. Un programmeur crée un mock dans le but de tester le comportement d'autres objets, réels, mais liés à un objet inaccessible ou non implémenté. Ce dernier est alors remplacé par un mock.
- Utile dès que l'on souhaite tester unitairement
- Utile si on ne peux pas facilement récupérer les données transmises à un objet externe
  - Réseau
  - Base de donnée

Calcul de hash – Non testable

```
public class HashBuilder {

    public String getHash(){
        TimeGenerator timeGenerator = new TimeGenerator();
        long time = timeGenerator.getTime();
        return DigestUtils.md5Hex("isima"+ time +"2014");
    }
}
```

Calcul de hash – Testable avec un Mock

```
public class HashBuilder {
  public String getHash(ITimeGenerator iTimeGenerator){
    long time = iTimeGenerator.getTime();
    return DigestUtils.md5Hex("isima"+ time +"2014");
  }

  public class MockTimeGenerator implements ITimeGenerator{
    public long getTime(){
      return 12;
    }
  }
  @Test
  public void testHashBuilderWithMock(){
    HashBuilder hb = new HashBuilder();
    assertEquals("EF12AC....",
                 hb.getHash(new MockTimeGenerator()));
  }
}
```

Couverture de code

- % de code couvert par les tests
  - Par classe
  - Par méthodes
  - Par sections de code
  - Par ligne
- En java
  - EMMA
  - Jacoco
- 50% – 70%

Conclusion

- Le test demande
  - Du temps de mise en place
  - De la pratique
  - De concevoir du code testable
- Retour sur investissement rapide si les tests écrits sont de bons tests
  - Faire des tests utiles
  - Tester beaucoup le classes à problème
  - Tester un peu les autres

Références

- Pragmatic Unit Testing (Andrew Hunt – David Thomas)
- Ship It ! (Jared Richardson – William Gwaltney Jr.)
- Schéma CI : http ://chercheurs.lille.inria.fr/ demarey/Main/ContinuousIntegration
- `http://junit.org/`