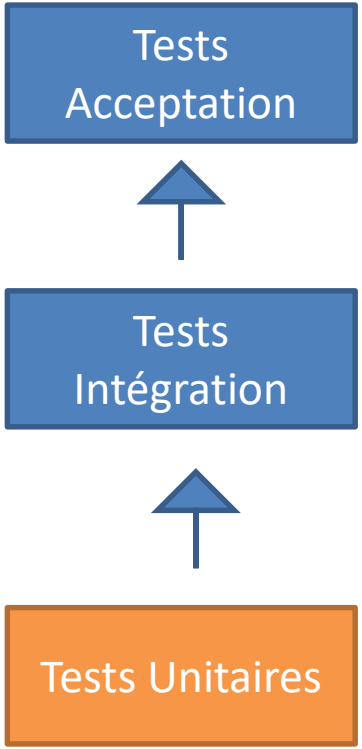
Tests unitaires

association nationale
pour la formation professionnelle
des adultes

Vincent BOST

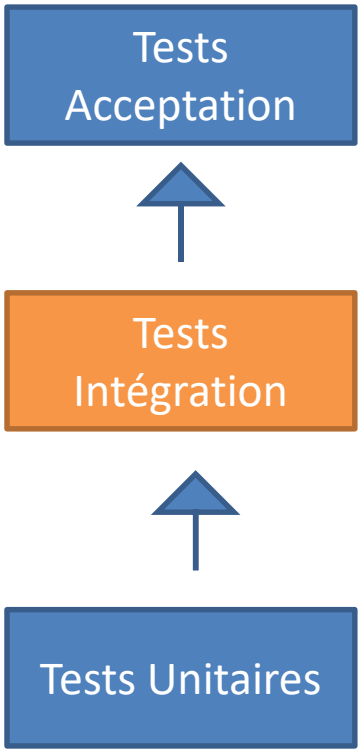
Sommaire

Les différentes types – Tests Unitaires



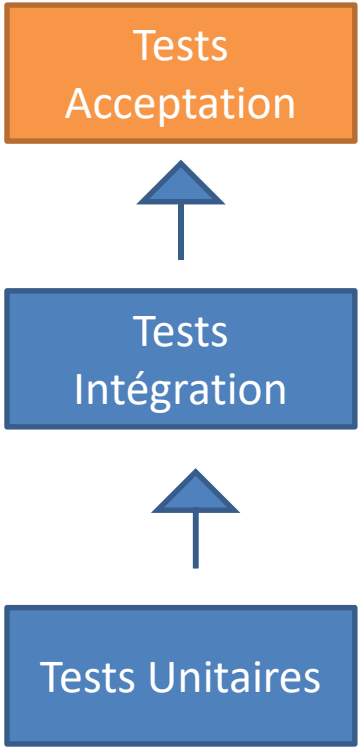
- Réalisés par le développeur
- En lien avec le fonctionnel et la spécification des exigences
- Chaque portion de code est testé individuellement
- Tests boite blanche
- Objectif d'une couverture de tests proche de 100 %
- Peuvent être organisés avant la phase de codage (TDD)

Les différentes types – Tests Intégration



- Intégration des différents modules
- Réalisés par l'équipe technique (développeur + expert technique + système)
- Ne seront pas validés si un test unitaire échoue
- Intégration / accès base de données ou service cloud
- Intégration / module messagerie authentification
- Interfaces entre 2 modules

Les différentes types – Tests Acceptation



Partie Fonctionnelle

Ecrits par les responsables métier et tests réalisés par des utilisateurs (ou testeurs MOA)

Vérification adéquation / besoin fonctionnel

Vérification Aptitude ou VA Bon Fonctionnement

Partie technique

Vérification de l'exploitabilité

Tests de performances

C'est à l'issue de ces tests que les fonctionnalités sont acceptées ou non (Recette)

Autres tests

- Revue de code automatique / manuel afin de produire un code de qualité
- Tests de stress, tests de montée en charge pour vérifier son fonctionnement dans la vraie vie…
- Tests de non régression

Les tests unitaires

Les tests unitaires testent chaque méthode d'un composant individuellement

L'objectif est , au travers des cas testés, de vérifier l'exactitude de la méthode et adéquation du résultat au besoin

Sous la responsabilité du développeur

Ne doivent pas intégrer les dépendances à l'architecture comme les bases de données ou une infrastructure Web.

Les tests unitaires

- Ils peuvent être manuels ou automatisés
- Toujours sous la responsabilité du développeur.
- Contrairement aux idées reçues, la programmation de tests automatisés, qui sollicite initialement plus de ressources, permet un retour sur investissement très rapide
- Ils peuvent être réalisés en amont du développement proprement dit des fonctions (TDD) : permet au développeur de vérifier sa bonne compréhension du besoin, d'identifier les conditions et cas de tests. Phase de refactorisation du code dans un second temps.
- Doivent toujours être faits au plus tôt

Recours à des framework spécialisés

Pour pouvoir les organiser, les suivre et les rejouer facilement de nombreux frameworks de tests

Junit pour Java

Nunit pour .Net et core

PhpUnit pour Php

...

Quelques principes

Isolé : Chaque test doit être isolé de toute dépendance

Ainsi, nous utilisons des Design Pattern et des frameworks pour éviter de solliciter des dépendances (Base de données, Messagerie, Service web, …) pour les tests de nos méthodes métier

Simple : Un test doit être le plus petit et le plus simple possible

Indépendant : Le résultat d'un test ne doit pas dépendre d'un autre test

Quelques principes

Répétable. L'exécution d'un test unitaire doit être cohérente avec ses résultats. Il retourne toujours le même résultat aucun changement n'est fait entre les exécutions (prédictif)

Se vérifier seul. Le test doit pouvoir détecter automatiquement son état de réussite ou d'échec sans aucune interaction humaine.

Découplage : Un produit testable impose une bonne factorisation du code

Un test se déroule en 3 actes A A A

Premier A pour Arrange : première étape de préparation où nous allons définir les objets nécessaires au bon déroulement du test

Second A pour Act : La méthode à tester est exécutée dans le contexte préparée à l'étape précédente

Troisième A pour Assert : Vérification du résultat produit par l'action ou Audit

Bonnes pratiques

- Définir une classe de test par classe à tester
- Le nom de la méthode de test doit permettre d'identifier son objectif
- Une seule assertion par test
- Eviter de programmer des blocs conditionnels ou itératifs au sein d'une méthode de tests. Ces options indiquent le plus souvent la nécessité de distinguer plusieurs cas à tester ou d'écrire plusieurs méthodes de tests.
- Identifier tous les cas d'exception
- Définir au moins un cas usuel

Exemple

Classe à tester

```
public class Calculatrice
{
    1 référence | 1/1 ayant réussi
    public int Additionner(int a, int b)
    {
        return a + b;
    }
}
```

Classe de test

```
[TestFixture]
0 références
public class TestCalculatrice
{
    [Test]
    | 0 références
    public void AdditionnerDeuxEntiersQuelconques()
    {
        int a = 3, b = 4;// Arrange
        int expected = 7;
        Calculatrice.BLL.Calculatrice calculatrice
            = new Calculatrice.BLL.Calculatrice();
        Assert.AreEqual(expected, calculatrice.Additionner(a, b));
    }
}
```

Les faux, simulacres et bouchons

Nous devrons souvent recourir à des framework de mocking pour produire des faux en lieu et place des objets réels

Dans tous les cas, nous devons bénéficier d'une abstraction pour créer un faux (classe abstraite ou interface)

Ainsi, pour qu'un test ne soit pas dépendant de la base de données, nous créerons un bouchon (stub) pour isoler notre tests de la couche de persistance. Pour cela nous utiliserons l'interface du Repository.

Nous pouvons aussi créer un Mock pour proposer un comportement de substitution en lieu et place de l'objet réel.