A41 – Qualité de Développement

Romain Orhand

rorhand@unistra.fr

Université de Strasbourg

IUT Robert Schuman
Institut universitaire de technologie
Université de Strasbourg

Retour sur vos TP

Attention à la faisabilité des fonctionnalités proposées ☺

Organisation du développement : groupe, git, …

-> Principal problème rencontré : « *merging* »

Objectif principal du TP : **garantir le bon fonctionnement de l'ensemble de l'application**

-> Vérifier l'existant

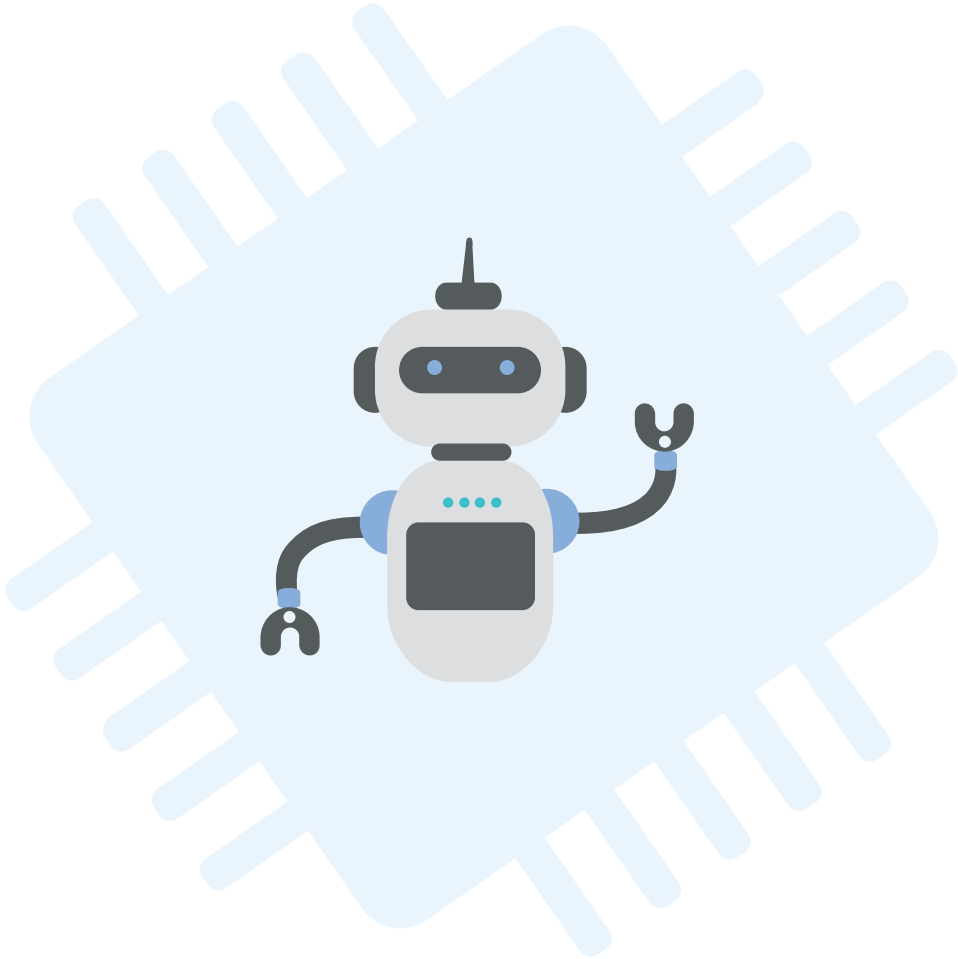
-> Bien définir ce que doivent faire vos nouvelles fonctionnalités

-> Vérifier vos ajouts et leur comportement avec l'existant

Parce que …

Parce que

Sommaire



Tester, c’est douter.

Du second degré, certes …

Tester, c'est vérifier.

Le test est **l'évaluation** d'un **composant** ou d'un **système**, par des moyens **manuels ou automatiques**, pour **vérifier** qu'il répond bien à ses **spécifications** et **identifier** les **différences** entre les résultats attendus et les résultats obtenus.

En quatre étapes :

1. Quel est l'objectif du test ?
2. Quels éléments en entrée du test ?
3. Quelle séquence d'actions pour le test ?
4. Quels éléments en sortie du test ?

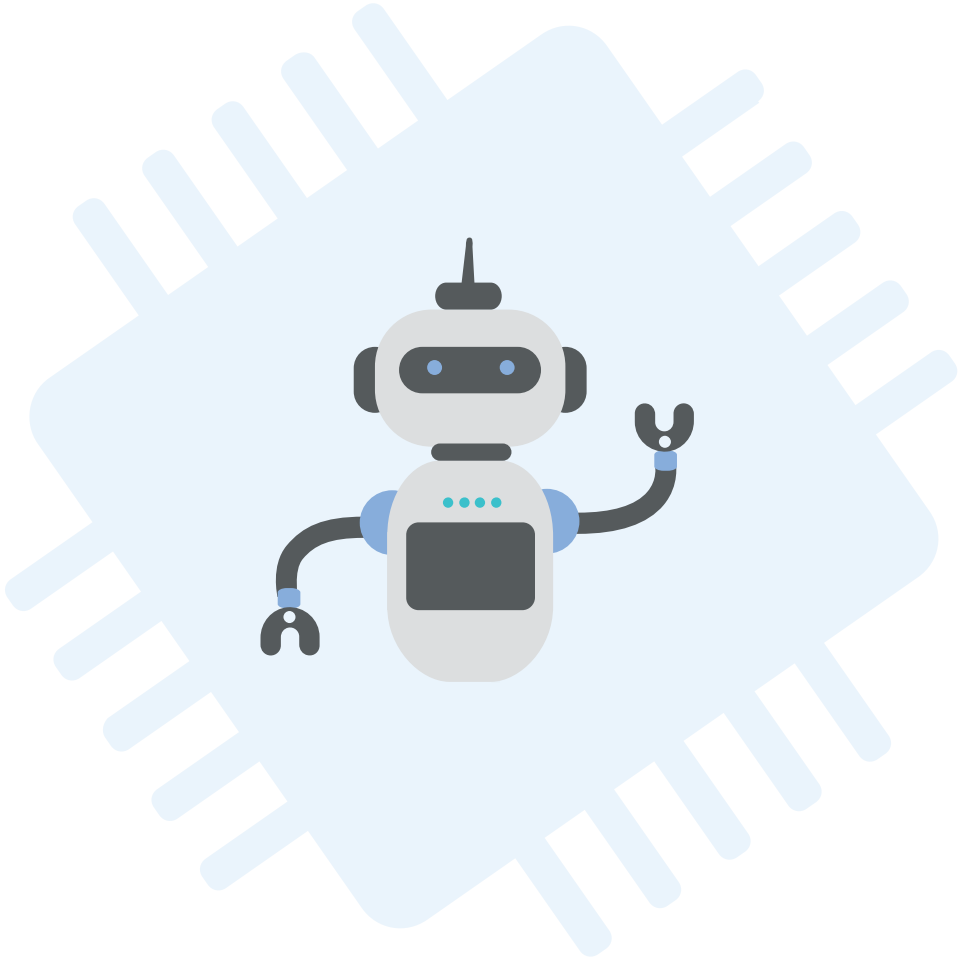
Pourquoi tester ?

- Anticiper l'inattendu
- Gagner du temps
- Réduire les coûts
- Faciliter la correction et la maintenance du code
- Communiquer

Quand tester ?

- Dépend du fonctionnement de votre équipe, entreprise, etc.
- A la portée de tous et sans surcharge de travail…
- C’est une étape qui se planifie dans votre projet !

Sommaire



Les tests unitaires

Il s'agit de **vérifier** si une **unité** de **code source**, dont le but est **défini et unique**, se comporte de manière attendue dans tous les cas et dans tous les contextes.

Deux approches :

-> les tests en **boîte blanche** qui se concentrent sur le fonctionnement et la structure d'une application.

-> les tests en **boîte noire** qui se concentrent sur les fonctionnalités d'une application.

Principes *FIRST*

- *Fast* : L'ensemble de vos tests sont rapidement compilés et exécutés.
- *Independent / Isolated* : Un test ne doit pas dépendre d'un autre test.
- *Repeatable* : Un test produit toujours le même résultat.
- *Self-validating* : Un test doit déterminer si son résultat est celui attendu ou non.
- *Timely* : Un test est écrit au même moment que le code testé.

Arrange – Act – Assert et *Given – When – Then*

1. ***Arrange / Given*** : Initialiser toutes les données nécessaires au test et le système à tester.
   (Etant donné un contexte, …)

2. ***Act / When*** : Exécuter l'élément à tester avec les données définies en amont.
   (… lorsque certaines actions sont effectuées …)

3. ***Assert / Then*** : Vérifier si ce qui est attendu est obtenu ou non.
   (… alors on doit pouvoir constater que.)

Nomenclature


```
class MyFunctionnalityShould {

    @Nested
    class InSomeContext {
        @Test
        void doThisUseCase() {
            // Arrange - Given
            // Act - When
            // Assert - Then
        }
        /* Do other tests */
    }
    /* Do other tests */
}
```

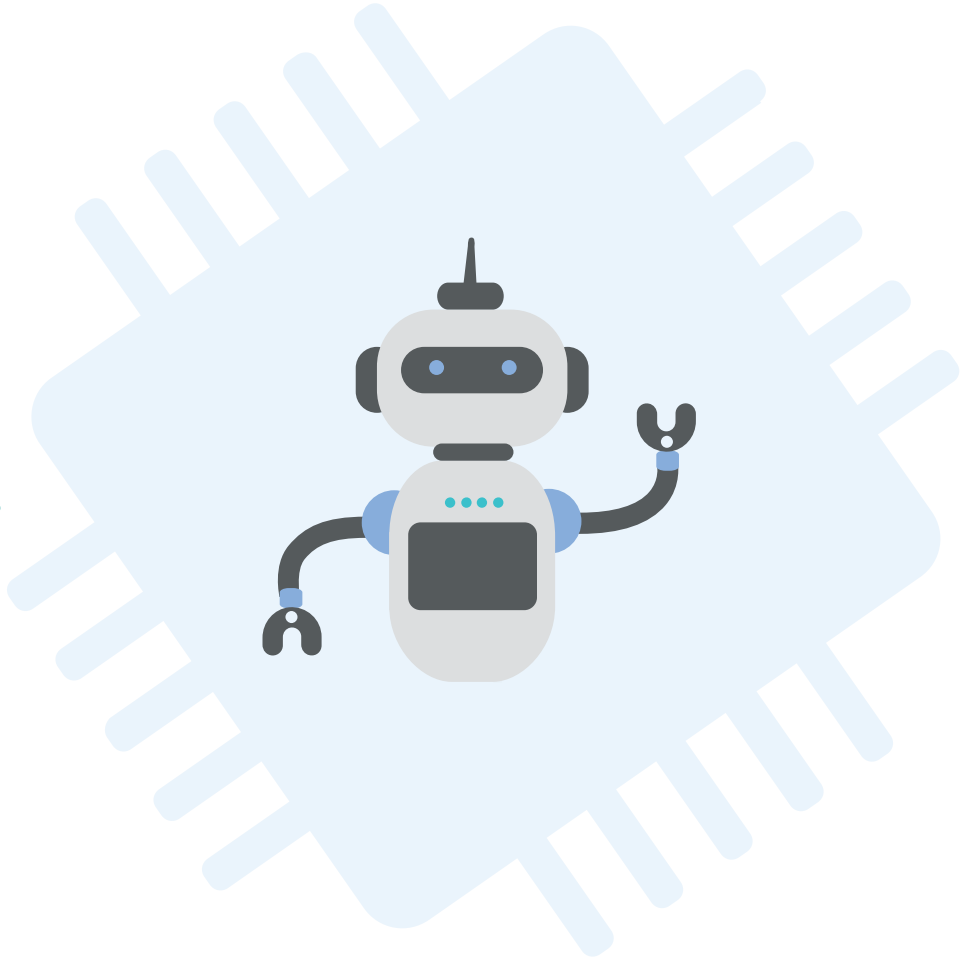
Exercice

1 Ecrire avec **la notation AAA ou GWT** des tests unitaires couvrant **la cuisson d'un muffin** avec **JUnit 5.**

2 Quels autres tests unitaires êtes-vous capables d'identifier ?

```java
class MyOvenShould {

    private Oven oven;

    @BeforeEach
    public void initOven() {
        oven = new Oven();
    }

    @Nested
    class InTheContextOfBakingOneMuffin{
        @Test
        void bakeOneAzukiMuffin() {
            // Arrange - Given
            Flavor flavor = Flavor.Azuki;
            // Act - When
            Muffin muffin = oven.bakeMuffin(flavor);
            // Assert - Then
            assertTrue(muffin.getFlavor().compareTo(anotherString: "Azuki") == 0);
        }

        /* And all other tests... */
    }

    @AfterEach
    public void resetOven() {
        oven = null;
    }
}
```

Sommaire



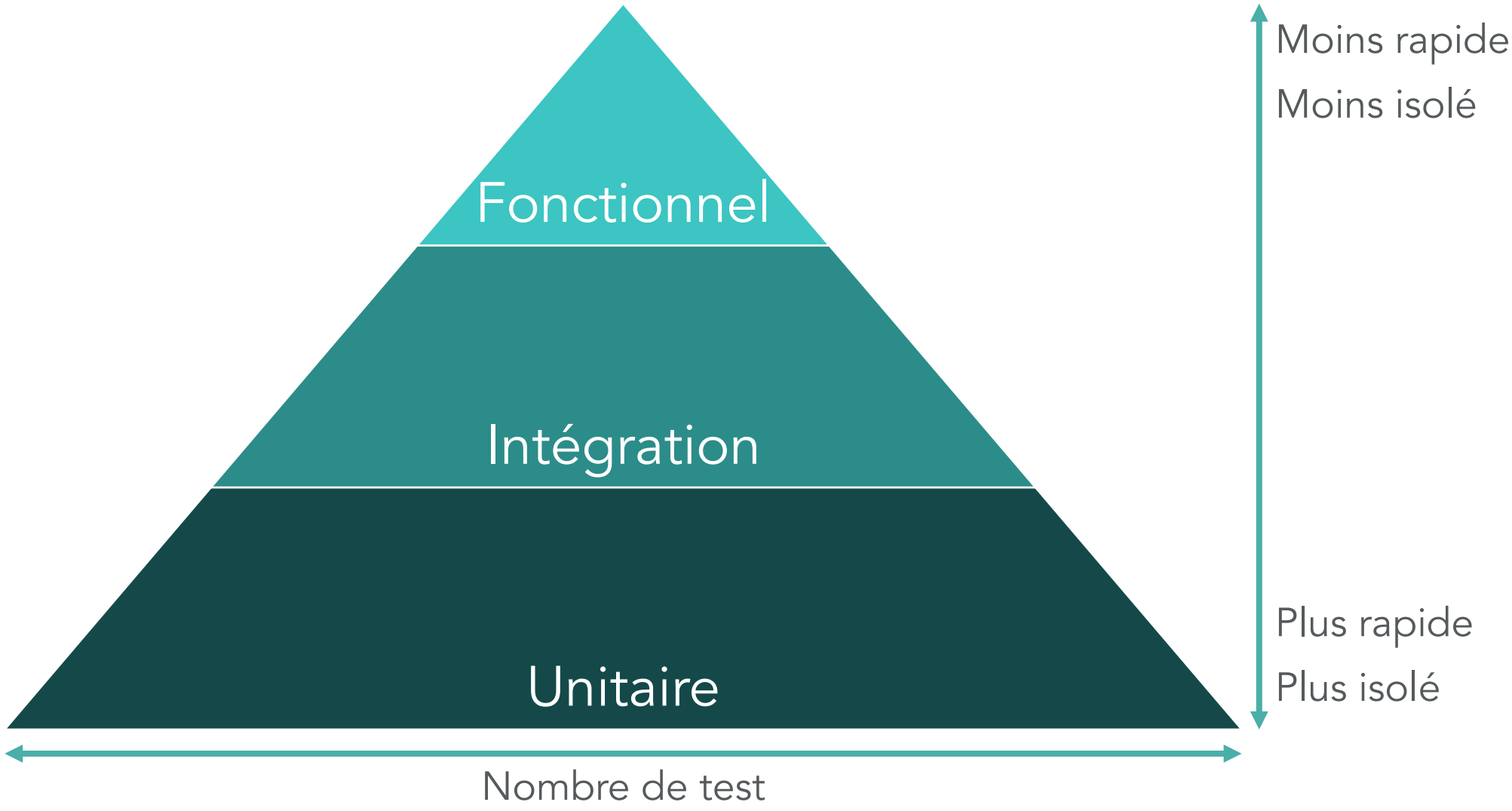
Test unitaire, test d'intégration et test fonctionnel

Test unitaire : Il s'agit de **vérifier** si une **unité** de **code source**, dont le but est **défini et unique**, se comporte de manière attendue dans tous les cas et dans tous les contextes.

Test d'intégration : Il s'agit de **vérifier** si **différentes unités** se comportent correctement lorsqu'elles sont combinées. L'accent est mis sur les **interactions**.

Test fonctionnel (système) : Il s'agit de **vérifier** si **l'ensemble** de l'application se comporte selon ce qui est attendu dans des situations réelles, défini dans les spécifications.

Pyramide des tests



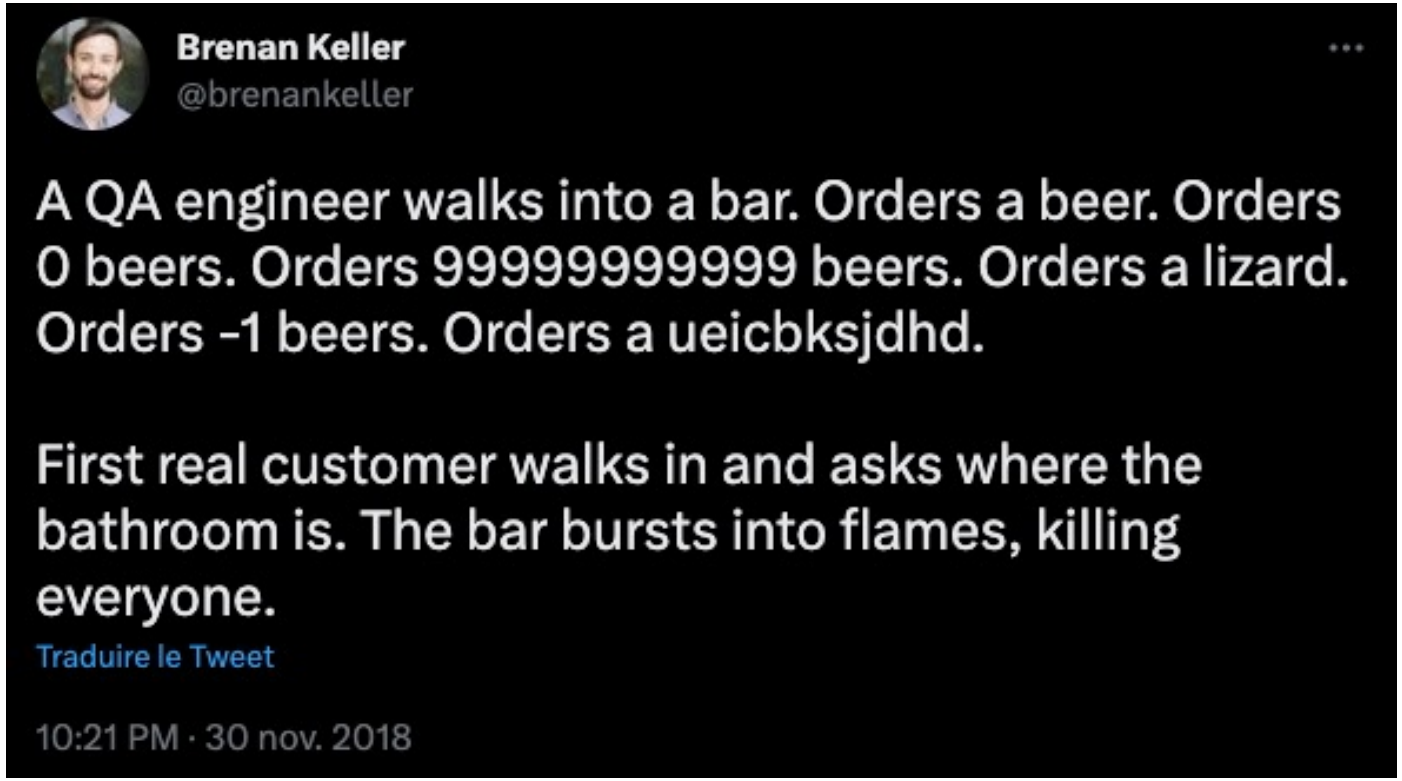
Couverture de test

Deux couvertures différentes :

-> la **couverture du code** qui quantifie la part du code traversée par des tests ;

-> la **couverture des cas** qui indique combien de cas d'utilisation sont couverts par des test.

Vers quelle couverture aller ?



Toujours *AAA ou GWT*

1. *Arrange / Given* : Initialiser toutes les données nécessaires au test et le système à tester.
(Etant donné un contexte, …)

2. *Act / When* : Exécuter l'élément à tester avec les données définies en amont.
(… lorsque certaines actions sont effectuées …)

3. *Assert / Then* : Vérifier si ce qui est attendu est obtenu ou non.
(… alors on doit pouvoir constater que.)

Les *test doubles* ou doublures de test

Parfois, il peut être nécessaire d'utiliser des éléments qui **ressemblent et se comportent comme leurs homologues** que l'on cherche à tester, tout en étant des versions **simplifiées** de ces homologues. Il s'agit des ***test doubles*** qui permettent **d'isoler** une classe de ses dépendances.

- ***Dummy*** : Utilisé lorsqu'une entrée est nécessaire à un test, sans qu'elle soit utilisée.

Les *test doubles* ou doublures de test

Parfois, il peut être nécessaire d'utiliser des éléments qui **ressemblent et se comportent comme leurs homologues** que l'on cherche à tester, tout en étant des versions **simplifiées** de ces homologues. Il s'agit des ***test doubles*** qui permettent **d'isoler** une classe de ses dépendances.

- ***Dummy*** : Utilisé lorsqu'une entrée est nécessaire à un test, sans qu'elle soit utilisée.
- ***Stub*** : Utilisé pour implémenter une entrée qui répond exactement ce qui est attendu.

Les *test doubles* ou doublures de test

Parfois, il peut être nécessaire d'utiliser des éléments qui **ressemblent et se comportent comme leurs homologues** que l'on cherche à tester, tout en étant des versions **simplifiées** de ces homologues. Il s'agit des ***test doubles*** qui permettent **d'isoler** une classe de ses dépendances.

- ***Dummy*** : Utilisé lorsqu'une entrée est nécessaire à un test, sans qu'elle soit utilisée.
- ***Stub*** : Utilisé pour implémenter une entrée qui répond exactement ce qui est attendu.
- ***Spy*** : Un *stub* qui enregistre des informations pendant un test et manipulables après.

Les *test doubles* ou doublures de test

Parfois, il peut être nécessaire d'utiliser des éléments qui **ressemblent et se comportent comme leurs homologues** que l'on cherche à tester, tout en étant des versions **simplifiées** de ces homologues. Il s'agit des ***test doubles*** qui permettent **d'isoler** une classe de ses dépendances.

- ***Dummy*** : Utilisé lorsqu'une entrée est nécessaire à un test, sans qu'elle soit utilisée.
- ***Stub*** : Utilisé pour implémenter une entrée qui répond exactement ce qui est attendu.
- ***Spy*** : Un *stub* qui enregistre des informations pendant un test et manipulables après.
- ***Mock*** : Utilisé pour substituer complètement l'implémentation de l'élément testé.

Les *test doubles* ou doublures de test

Parfois, il peut être nécessaire d'utiliser des éléments qui **ressemblent et se comportent comme leurs homologues** que l'on cherche à tester, tout en étant des versions **simplifiées** de ces homologues. Il s'agit des ***test doubles*** qui permettent **d'isoler** une classe de ses dépendances.

- ***Dummy*** : Utilisé lorsqu'une entrée est nécessaire à un test, sans qu'elle soit utilisée.
- ***Stub*** : Utilisé pour implémenter une entrée qui répond exactement ce qui est attendu.
- ***Spy*** : Un *stub* qui enregistre des informations pendant un test et manipulables après.
- ***Mock*** : Utilisé pour substituer complètement l'implémentation de l'élément testé.
- ***Fake*** : Utilisé comme une implémentation d'un élément testé inadaptée à la mise en production.

Les tests *doubles : dummy*

Dummy : Utilisé lorsqu'une entrée est nécessaire pour réaliser un test, sans qu'elle soit utilisée.

Exercice :

1. Identifier un ou plusieurs tests qui nécessiteraient l'utilisation d'une doublure de test *dummy* et expliquer pourquoi.

2. Ecrire pour l'un de ces tests la doublure *dummy* nécessaire.

Injection de dépendance

Parfois, il est nécessaire de découpler les dépendances entre objets pour pouvoir mettre en place nos doublures de tests.
Avec l'injection de dépendance, les dépendances entre composants logiciels ne sont plus exprimées dans le code de manière statique mais déterminées dynamiquement à l'exécution.
Si une classe A dépend d'une classe B :

1. Créer une interface I qui contiendra toutes les méthodes que A peut appeler de B.
2. Indiquer que B implémente I.
3. Remplacer toutes les références de B dans A par des références à I.

Un exemple de *dummy*

Dummy : Utilisé lorsqu'une entrée est nécessaire pour réaliser un test, sans qu'elle soit utilisée.

```
public class DummyOven implements OvenInterface {

    @Override
    public Muffin bakeMuffin(Flavor f) {
        /* ou throw new Exception("Ne devrait pas être appelé");
         * avec ajout de throws dans la méthode
         */
        return null;
    }

}
```

Les tests *doubles : stub*

Stub : Utilisé pour implémenter une entrée qui répond exactement ce qui est attendu.

Exercice :

1. Identifier un ou plusieurs tests qui nécessiteraient l'utilisation d'une doublure de test *stub* et expliquer pourquoi.

2. Ecrire une doublure *stub* d'une caisse renvoyant un montant fixe et d'un four cuisant un muffin au parfum azuki.

Deux exemples de *stub*

Stub : Utilisé pour implémenter une entrée qui répond exactement ce qui est attendu.

```java
public class StubCashRegister implements CashRegisterInterface {

    @Override
    public Double getIncomes() {
        return 5.9;
    }

    @Override
    public void increaseIncomes(Double incomes) {}

    @Override
    public void clearCashRegister() {}

}
```

```java
public class StubOven implements OvenInterface {

    @Override
    public Muffin bakeMuffin(Flavor f) {
        return new MuffinAzuki();
    }

}
```

Les tests *doubles : spy*

Spy : Un *stub* qui enregistre des informations pendant un test et manipulables après.

Exercice :

1. Identifier un ou plusieurs tests qui nécessiteraient l'utilisation d'une doublure de test *spy* et expliquer pourquoi.

2. Ecrire une doublure *spy* d'un four comptabilisant le nombre de muffins qu'il a cuit.

Un exemple de *spy*

Spy : Un *stub* qui enregistre des informations pendant un test et manipulables après.

```
public class SpyOven implements OvenInterface {

    public Integer nbOfBakedMuffins = 0;

    @Override
    public Muffin bakeMuffin(Flavor f) {
        nbOfBakedMuffins++;
        return new MuffinAzuki();
    }

}
```

Les tests *doubles : fake*

Fake : Un *stub*, mais avec un peu de logique, comme une petite implémentation de la classe testée.

-> Implémentation parfois partielle du contrat représenté par l'objet à laquelle il manque des caractéristiques pour être utilisable en production.

-> Pour vérifier que cet objet remplit bien son contrat, on peut utiliser les tests unitaires associés au contrat.

-> Utile par exemple dans le cadre de la persistance des données.

Un exemple de *fake*

Fake : Un *stub*, mais avec un peu de logique, comme une petite implémentation de la classe testée.

```java
public class BDD implements BDDInterface {

    // All java sql items, connection object, ...

    @Override
    public String read(Integer id) {
        // SQL request
        return "";
    }

    @Override
    public String update(Integer id, String newValue) {
        // SQL request
        return "";
    }

}
```

```java
import java.util.HashMap;

public class FakeBDD implements BDDInterface {

    HashMap<Integer, String> fakeBDD = new HashMap<Integer, String>();

    @Override
    public String read(Integer id) {
        return fakeBDD.get(id);
    }

    @Override
    public String update(Integer id, String newValue) {
        return fakeBDD.replace(id, newValue);
    }

}
```

Les tests *doubles : mock*

Mock : Utilisé pour substituer complètement l'implémentation de l'élément testé.

-> Objet au comportement précâblé très spécifique : « *quand on te demande ça tu fais ça* ».

-> Peut lever des exceptions s'il ne reçoit pas les bons appels et décider d'échouer.

-> Peut réaliser des vérifications comme un *spy* et faire en plus **ses propres assertions**.

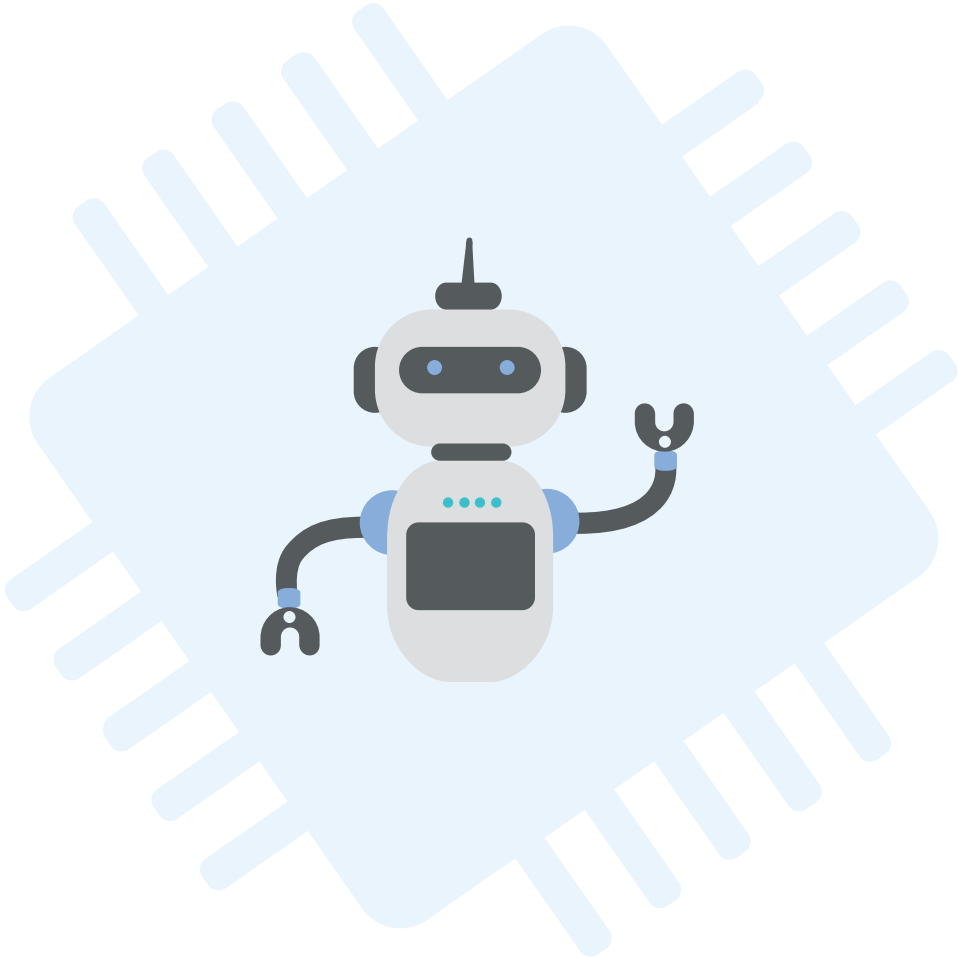
-> Surtout utile pour vérifier le comportement de votre application.

Attention, bien souvent un *fake* ou un *spy* peuvent être suffisants pour écrire vos tests !

Un exemple de *mock*


```
public class MockOven implements OvenInterface {

    public Integer nbOfBakedMuffins = 0;
    public Integer numberOfExpectedBakedMuffin = 0;

    @Override
    public Muffin bakeMuffin(Flavor f) {
        nbOfBakedMuffins++;
        return new MuffinAzuki();
    }

    public void setNumberOfExpectedBakedMuffin(Integer nb) {
        numberOfExpectedBakedMuffin = nb;
    }

    public void verifyNumberOfExpectedBakedMuffin() {
        assertTrue(numberOfExpectedBakedMuffin == nbOfBakedMuffins);
    }

}
```

Sommaire



Politique de test

« Document de haut niveau décrivant les principes, approches et objectifs majeurs de l'organisation concernant l'activité de test. »

ISTQB

La politique de test est un document qui décrit pourquoi on fait des tests et quelles pratiques peuvent être mises en œuvre.

Stratégie de test

*« Document de haut niveau **définissant**, pour un programme, les niveaux de tests à exécuter et les tests dans chacun de ces niveaux (pour un ou plusieurs projets). »*

ISTQB

La stratégie de test est un document qui décrit quels tests sont effectués sur un ou plusieurs projets au sein d'une organisation, à partir de la politique de test :

-> Quelle approche des tests ?

-> Quels types de tests à exécuter ?

-> Quels outils et environnements ?

Plan de test

« Document décrivant l'étendue, l'approche, les ressources et le planning des activités de test prévues.

Il identifie entre autres les éléments et caractéristiques à tester, l'affectation des tâches, le degré d'indépendance des testeurs, l'environnement de test […]. Il constitue la documentation du processus de planification de test. »

ISTQB

En bref, un plan de test définit donc ce que l'on va tester, comment on va le tester et aussi, ce qui ne va pas être testé.

Politique, Stratégie et Plan de test

Politique de test

Stratégie de test

Plan de test

Avec tous les détails

Ecrire un plan de test

Faire concis et efficace :

1. Quelle est l'application que vous testez ? La décrire.
2. Quelles fonctionnalités seront testées et en quelle mesure ? Pourquoi ?
3. Quelles fonctionnalités ne seront pas testées ? Pourquoi ?
4. Quels risques votre plan de test comporte-t-il ?
5. Quels outils de tests sont utilisés ?
6. Quel est le planning de la mise en place de vos tests ? Qui, quoi, quand ?

Fiche de test / Rapport de test

Au moins 7 éléments attendus :

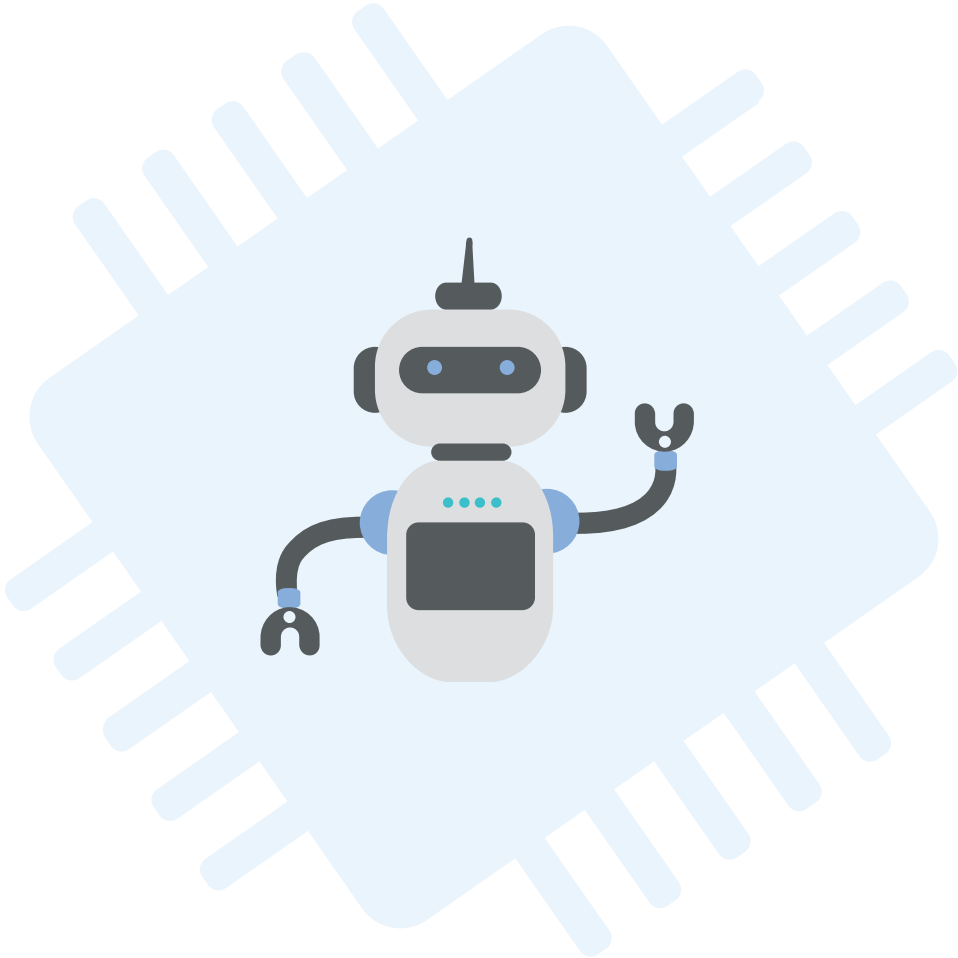
1. Type de test
2. Titre du test
3. Scénario
4. Résultat attendu
5. Résultat observé
6. Résultat du test
7. Commentaire

Exercice :
Ecrire le rapport de test associé au test unitaire lié à la cuisson d'un muffin azuki.

Un exemple de rapport de test

Type de test	Titre	Scénario	Attendu	Observé	Résultat	Commentaires
Unitaire	Cuire un muffin azuki	**Etant donné** un parfum azuki, un four **cuit** un muffin de ce parfum, **afin de** fournir un muffin azuki.	Un muffin azuki	Un muffin azuki	Ok	
…		…	…	…	…	

Sommaire



Plein d'autres tests existent ! (1/3)

Test Interface Homme Machine - S'assurer que l'interface utilisateur graphique d'une application répond à ses spécifications.

Test de Performance - Déterminer la performance (temps de réponse utilisateurs, temps de réponse réseau, temps de traitement) d'une application.

Test de Sécurité - Processus consistant à trouver des failles dans la sécurité des systèmes d'information.

Test de Compatibilité (coexistence et Interopérabilité) - Evaluer le degré d'un logiciel à réaliser ses tâches en étant intégré dans un environnement avec d'autres logiciels et s' assurer que les informations entre plusieurs systèmes sont bien échangées.

Test de Régression - Vérifier si des modifications apportées à un logiciel ont cassé une fonctionnalité qui fonctionnait auparavant.

Plein d'autres tests existent ! (2/3)

Test à Données Aléatoires (*fuzzing*) - Fournir des données invalides, inattendues ou aléatoires en entrée d'un programme informatique et le surveiller pour détecter les exceptions telles que les plantages, les défaillances des assertions de code intégrées ou les fuites de mémoire potentielles.

Test d'Utilisabilité - Evaluer un produit en le faisant tester par des utilisateurs.
Test d'Accessibilité - Test utilisateur lorsque les utilisateurs considérés présentent des handicaps qui affectent la manière dont ils utilisent une application.

Test d'Acceptation - Test visant à déterminer si les exigences d'une spécification ou d'un contrat sont satisfaites.

Test Non Fonctionnel - Test d'un logiciel informatique pour la façon dont il fonctionne plutôt que pour des comportements ou des fonctions spécifiques.

Test d'Intégrité - Evaluer rapidement si une affirmation ou le résultat d'un calcul peut être vrai.

Plein d’autres tests existent ! (3/3)

Test de Charge - Processus consistant à solliciter un système et à mesurer sa réponse.
Test de Montée en Charge (capacité) - Simulation d’un nombre d'utilisateurs sans cesse croissant de façon à déterminer quelle charge limite le système peut supporter.

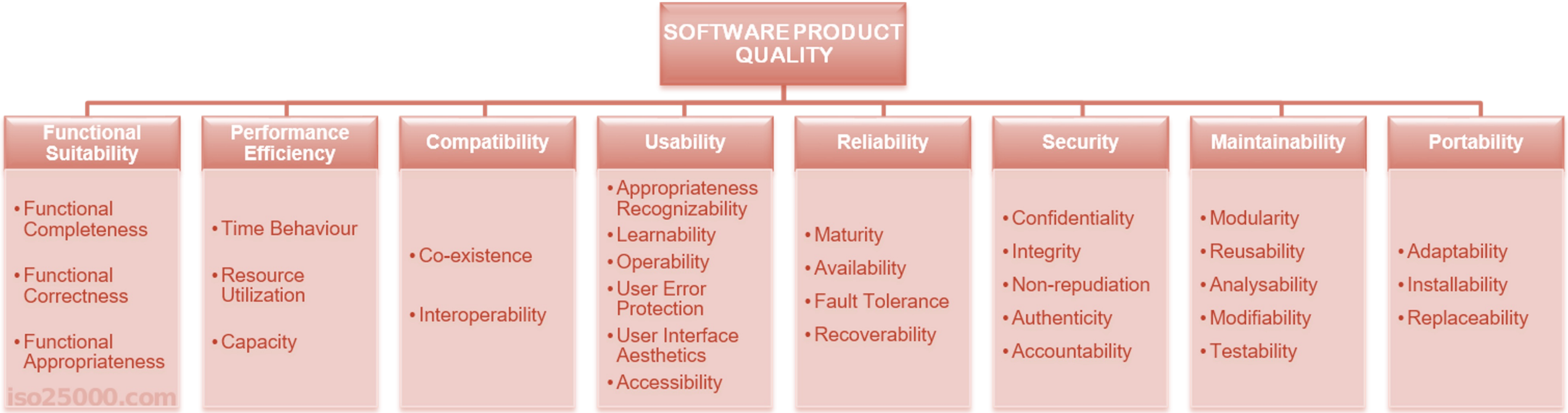
Test de Robustesse - Forme de test délibérément intense ou approfondie, utilisée pour déterminer la stabilité d'un système, d'une infrastructure critique ou d'une entité donnée.

Test de Résilience - Tenter de faire échouer un logiciel de manière incontrôlée, afin de tester sa robustesse et d'aider à établir les limites dans lesquelles le logiciel fonctionnera de manière stable et fiable.

Test de Localisation - Vérifier que le produit soit utilisable par les utilisateurs d'une région particulière.

Test de Stress - Etablir comment un système réagit au maximum de l’activité attendue des utilisateurs

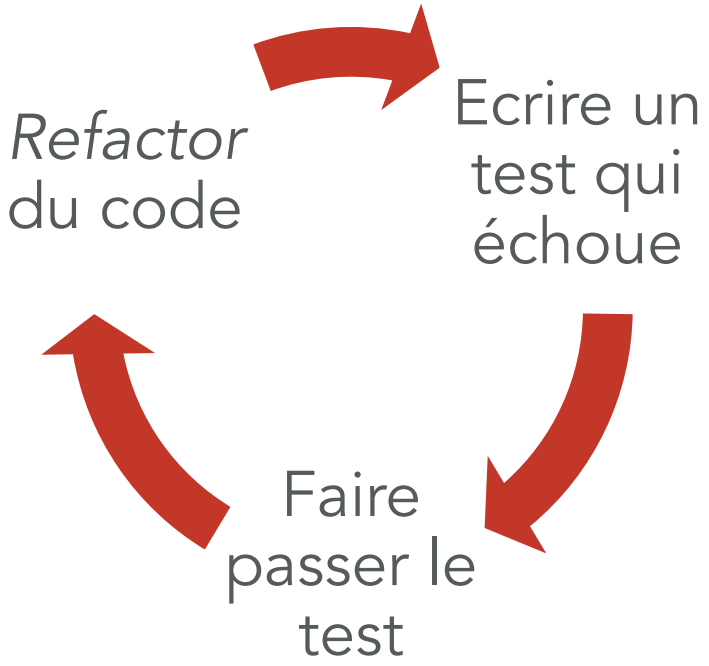
ISO/IEC 25010



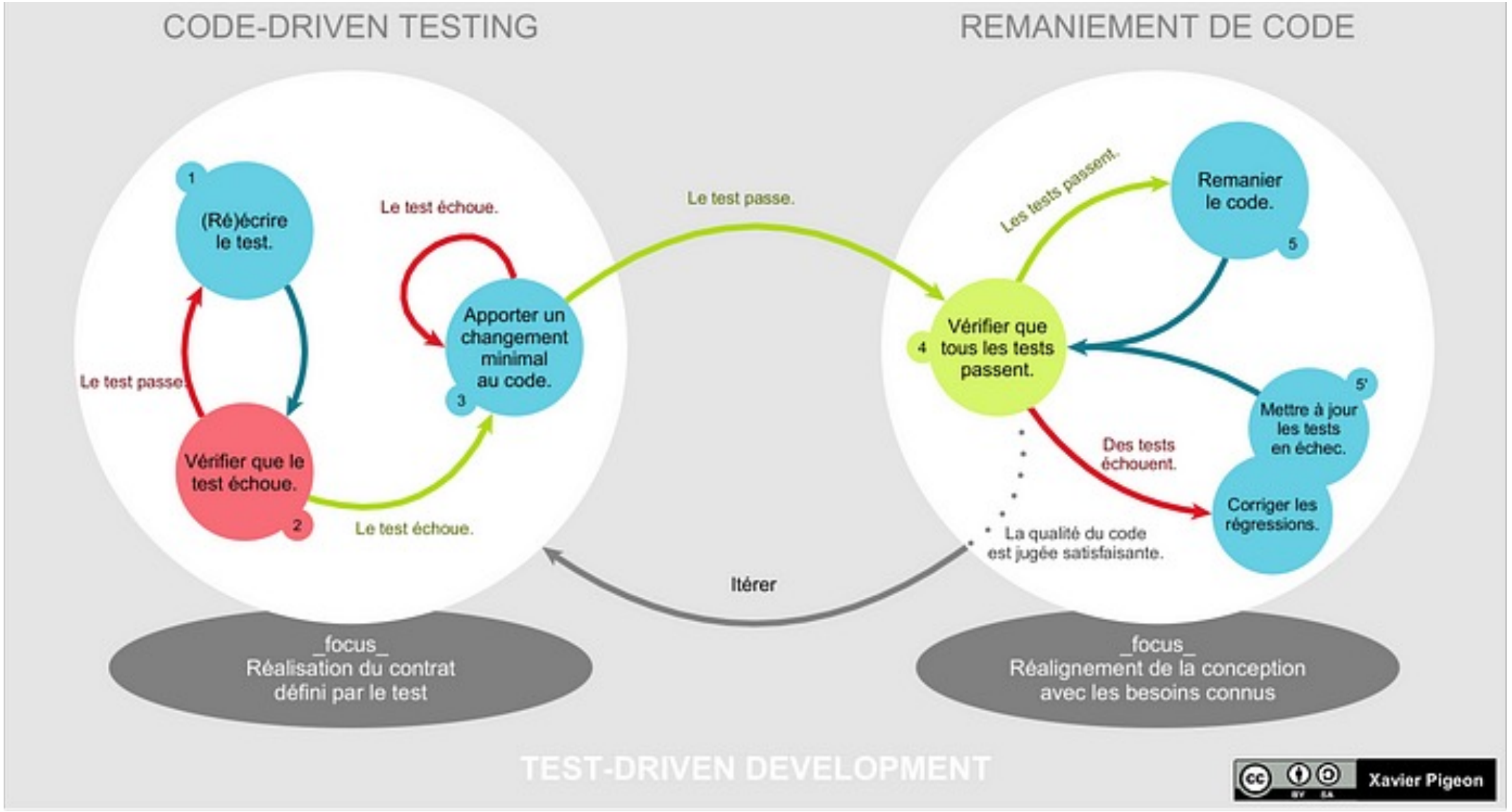
Source : https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

Test Driven Development (1/2)

Le **Développement Piloté par les Tests** est une méthode de développement de logiciel qui consiste à concevoir un logiciel **par des itérations successives très courtes**, telles que chaque itération est accomplie en formulant un sous-problème **à résoudre sous forme d'un test** avant d'écrire le code source correspondant, et où le code est continuellement remanié dans une volonté de simplification.



Test Driven Development (2/2)



Intégration Continue et Déploiement Continu

Intégration Continue : Pratiques pour vérifier à chaque modification du code qu'il n'y a pas de **régression** dans l'application.

Déploiement Continu : Extension de l'intégration continue, il s'agit de faire en sorte que les utilisateurs ont **automatiquement** une version de votre application la plus à jour.

1. Planifier le développement
2. Compiler et **intégrer** le code
3. **Tester** le code
4. **Mesurer** la qualité du code
5. Gérer les livrables de l'application

Projet : A vous de jouer !

Back to the Muffin Shop Part II

Une nouvelle version de l'application construite à partir du TP.

Objectifs, **en binôme** :

1. **Proposer un plan de test** de l'application en cohérence avec une stratégie de test donnée.
2. **Concevoir et exécuter les tests unitaires et d'intégration** selon le plan que vous aurez établi.

Lisez tout bien le sujet !

Rendu : 9 avril 23h59 au plus tard.