

# **Qualité**

# *Logicielle*

Youssef Touati ESIEA

15 Octobre 2025



## Présentation

Youssef TOUATI

Poste actuel : Test Lead

Domaine : Assurance

## Parcours professionnel

Expérience dans plusieurs secteurs :

- Automobile
- Fintech
- Paris sportifs
- Assurance

## Formation

Ingénieur en systèmes embarqués

## Mon Quotidien :

- Planification et pilotage des tests
- Coordination de l'équipe QA
- Analyse des exigences et risques
- Suivi de la qualité et reporting
- Amélioration continue & automatisation des process

# Introduction au Cours - Qualité Logicielle



## Ma passion ...

Rechercher

+ Créer

9+

Accueil

Shorts

Abonnements

Vous >

- Historique
- Playlists
- Vos vidéos
- À regarder plus tard
- Vidéos "J'aime"
- Vos clips

Abonnements >

- TEDx Talks
- Bloomberg Orig... (0)
- edureka!
- 24 / FRANC... (0)
- Cuisine olfa ... (0)
- Elhiwar Ettounsi
- SIKANA Français

Apprendre les tests de logiciels avec Youssef

@Apprendre\_Avec\_Youssef · 6 k abonnés · 30 vidéos

LinkedIn: Youssef Touati ...plus

Personnaliser la chaîne Gérer les vidéos

Accueil Vidéos Playlists Posts

ISTQB (version 2023)- Chapitre 1: Fondamentaux des tests

15322 vues • il y a 2 ans

19:21 / 3:16:01

Pour vous

Présenté Par :

Présenté par :

Souha BEJAOUI Head of QA Department

ISTQB International Software Testing Qualification

Chapitre 2 : Tester pour la vie du dév

Livraison à 75019 Paris Mettre à jour l'emplacement

Toutes Amazon Haul Monoprix Meilleur

Ebooks Kindle Abonnement Kindle Prime Reading

PRÉPARATION  
À L'ISTQB  
FOUNDATION  
V4 : EXERCICES  
PAR CHAPITRE

YOUSSEF TOUATI

Votre guide complet pour réussir la certification ISTQB Foundation 2023



## Organisation du module "Qualité Logicielle" (24h)

Composante	Durée	Contenu
Cours magistral	6h	Concepts clés de la qualité logicielle, métriques, ROI, automatisation
TP – Tests manuels (Jira/Xray)	6h	Création, exécution et suivi de cas de test manuels
TP – Automatisation des tests	12h	Implémentation de scripts, outils, stratégie d'automatisation

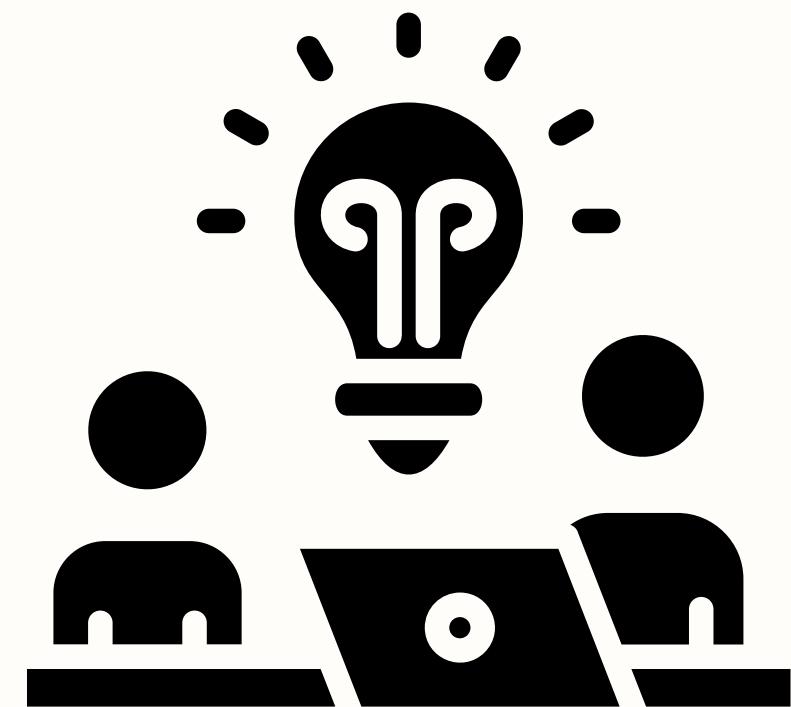
## Répartition des groupes

**Cours magistral :** Tous les 3 groupes réunis

**TP :**

1 groupe continue avec Youssef

2 groupes encadrés par Adrien

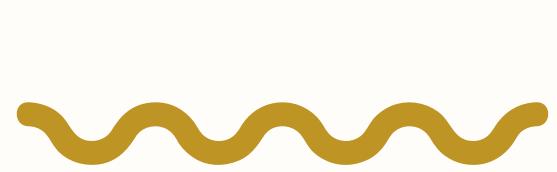




## Modalités d'évaluation

<b>Épreuve</b>	<b>Poids</b>	<b>Détails</b>
Cours magistral	50%	20 QCM américains (⚠ mauvaise réponse = point négatif)
Travaux pratiques	50%	Évaluation sur la qualité des tests manuels et automatisés réalisés

# Pourquoi ce cours est unique pour vous ?



Dans quelques mois, vous suivrez deux ou 3 chemins principaux :

- Développeur,
- Testeur,
- Chef de projet ou Product Owner.

Ce module vous aidera à :

- Comprendre comment chaque rôle perçoit la qualité,
- Collaborer efficacement entre métiers,
- Parler un langage commun autour du produit,

 Parce que la qualité, ce n'est pas qu'une affaire de tests... c'est une affaire d'équipe.



## L'objectif de ce module : créer des ponts

### Pour les futurs développeurs :

- Comprendre pourquoi les testeurs formulent certaines demandes
- Anticiper les cas de test dès la phase de développement
- Collaborer efficacement avec les équipes qualité

### Pour les futurs testeurs :

- Découvrir les enjeux business derrière les aspects techniques
- Maîtriser les outils et méthodes de test modernes
- Comprendre la psychologie et le langage des développeurs pour mieux communiquer

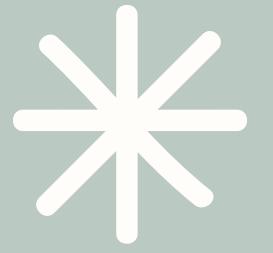
### Pour tous :

- Parler un langage commun
- Comprendre les contraintes et priorités de chacun
- Co-créer des produits fiables et exceptionnels



## Ce que j'attends de vous :

- **Je veux que ce cours soit interactif**
- **Vos questions - surtout celles qui vous semblent naïves**



# **Chapitre 1 : Introduction à la Qualité Logicielle**



# Chapitre 1 : Introduction à la Qualité Logicielle



- **1.1 Le coût de la non-qualité : Réalités et conséquences**
  - 1.1.1 Études de cas emblématiques
  - 1.1.2 Impacts business de la non-qualité
- **1.2 Définitions fondamentales**
  - 1.2.1 Qu'est-ce que la qualité logicielle ?
  - 1.2.2 Définition du test logiciel
  - 1.2.3 Vérification vs Validation
- **1.3 Les 7 principes fondamentaux du test (ISTQB)**
- **1.4 La pyramide des tests : Organisation par niveaux**
  - 1.4.1 Présentation des 4 niveaux fondamentaux
  - 1.4.2 Pourquoi la pyramide est efficace ?



# I.I Le coût de la non-qualité : Réalités et conséquences



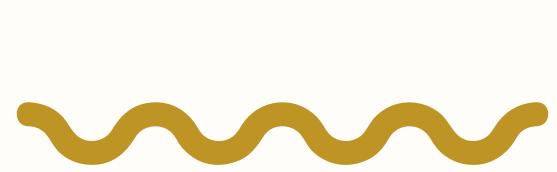
### Exemple 1 : Crash d'Instagram (2022)

Pendant plusieurs heures, des millions d'utilisateurs n'ont pas pu se connecter à la plateforme.

- **Cause** : déploiement en production d'une mise à jour mal testée .
- **Impact** : perte de revenus publicitaires + frustration des utilisateurs.



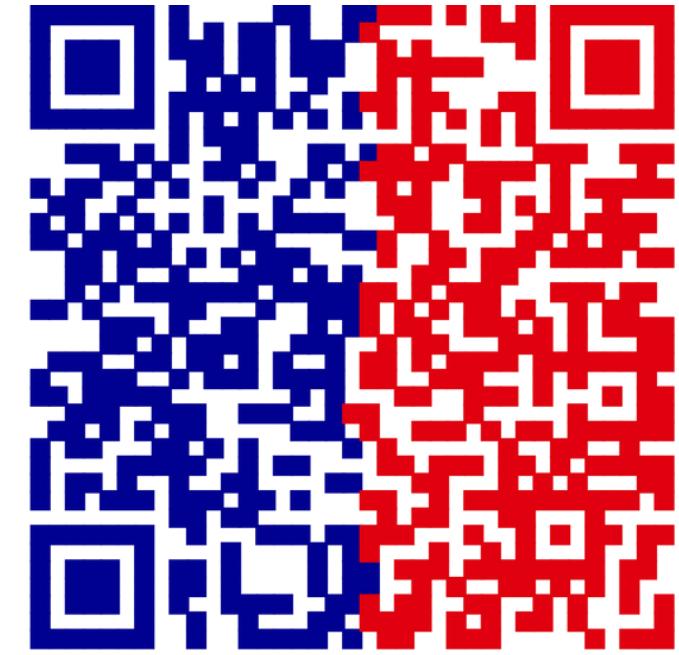
## 1.1.1 Études de cas emblématiques



### Exemple 2 : Bug de l'application TousAntiCovid (France, 2021)

Certains QR codes valides étaient refusés à l'entrée des lieux publics.

- **Cause** : problème d'interprétation des données lors de la lecture du certificat.
- **Impact** : blocages aux entrées, mécontentement des usagers et forte médiatisation.



#Tous  
AntiCovid



### Panne mondiale de Microsoft Teams (2023)

Coupure mondiale de plusieurs heures.

- **Cause** : erreur de configuration lors d'une mise à jour du cloud Azure.
- **Impact** : millions de salariés bloqués → perte de productivité énorme.



## 1.1.1 Études de cas emblématiques



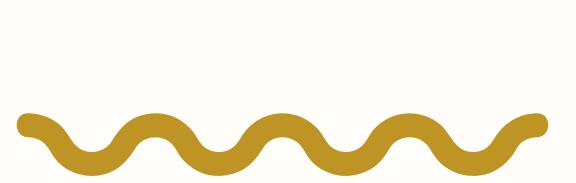
### Bug Tesla (2021)

Une mise à jour OTA a désactivé l'Autopilot de milliers de voitures.

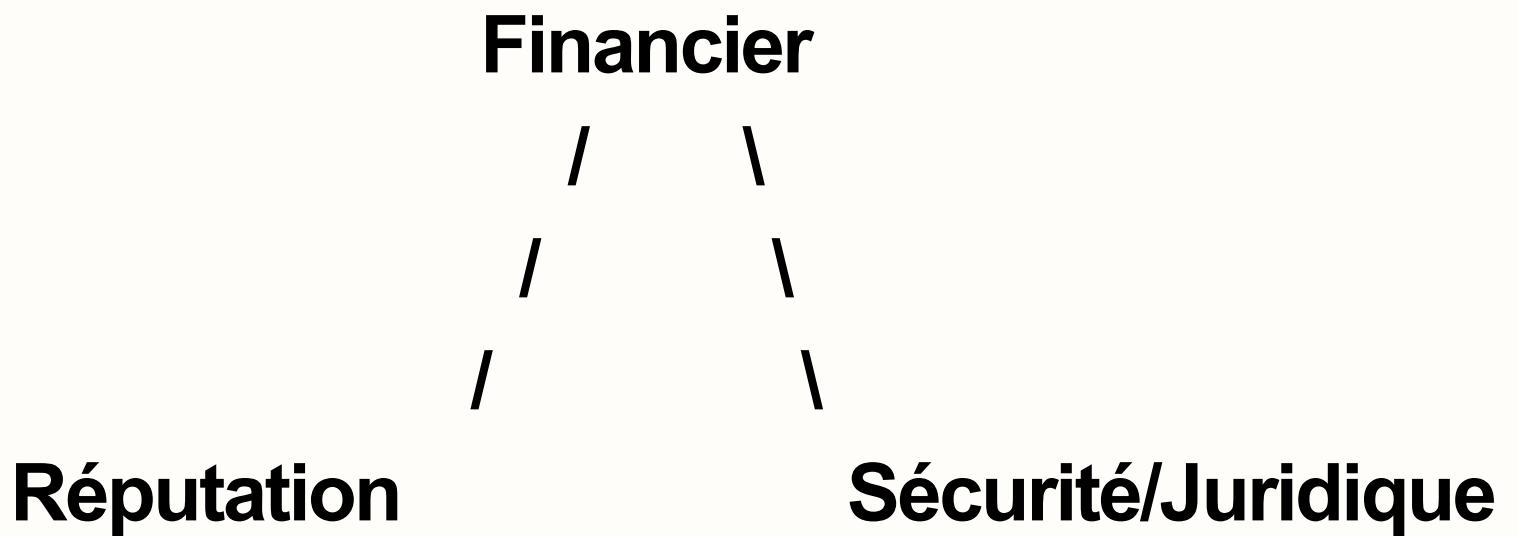
- **Cause** : erreur logicielle liée au traitement des capteurs.
- **Impact** : rappel immédiat + risque sécurité routière.



## 1.1.2 Impacts business de la non-qualité



### Triangle des conséquences :

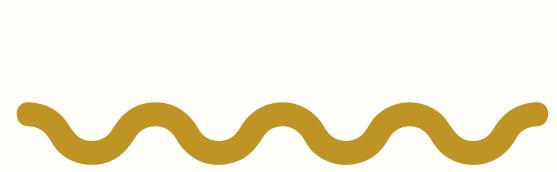


Catégorie	Impact court terme	Impact long terme
Financier	Remboursements, pénalités, perte de CA	Décote boursière, augmentation des primes d'assurance
Réputation	Communication de crise, perte de clients	Perte de parts de marché
Juridique	Amendes réglementaires (GDPR, etc.)	Procès, responsabilités civiles/pénales
Opérationnel	Temps de correction, support surchargé	Démotivation équipes, retard roadmap

→ un simple bug peut avoir des répercussions à tous les niveaux de l'entreprise.



## I.2 Définitions fondamentales



**Question: c'est quoi “la qualité logicielle” pour vous en un mot ?**

## 1.2.1 Qu'est-ce que la qualité logicielle ?



| La qualité, c'est bien plus qu'un code qui fonctionne...

### Définition ISO 25010 :

“La qualité d'un produit logiciel est sa capacité à satisfaire des besoins explicites et implicites dans des conditions d'utilisation spécifiées”

### Les 4 dimensions de la qualité

Dimension	Description synthétique
<b>Qualité externe</b> (ce que voit l'utilisateur)	Comportement observable du logiciel (vision <b>objective</b> )
<b>Qualité interne</b> (ce que voit le développeur)	Structure et maintenabilité du code
<b>Qualité perçue</b> (expérience utilisateur)	Ressenti et satisfaction visuelle / émotionnelle ( vision <b>subjective</b> )
<b>Qualité en usage</b> (résultats en conditions réelles)	Efficacité du produit dans son contexte d'utilisation ( Performance ..)

## 1.2.2 Définition du test logiciel



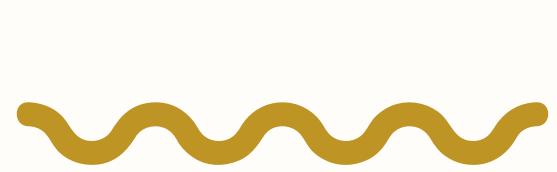
**Définition ISTQB** (organisme international de référence pour la certification des testeurs logiciels)

"Le test est un Processus consistant à exécuter un programme ou système avec l'intention de trouver des défauts"

### Les 3 objectifs complémentaires :

1. **Déetecter les défauts** → Améliorer le produit
2. **Fournir de la confiance** → Aider à la décision de mise en production
3. **Fournir de l'information** → Éclairer les parties prenantes sur la qualité

## 1.2.3 Vérification vs Validation : Le double questionnement



- Vérifier, c'est construire correctement.
- Valider, c'est construire ce qu'il faut.

Ces deux notions sont souvent confondues – voici comment les distinguer :

Aspect	Vérification	Validation
Question	"Construit-on le produit <b>BIEN</b> ?"	"Construit-on le <b>BON</b> produit ?"
Objectif	Conformité aux spécifications	Adéquation aux besoins utilisateur
Périmètre	Processus de développement	Produit final et valeur business
Méthodes	Revue de code, tests unitaires, analyse statique	Tests utilisateur, beta tests
Acteurs	Équipe technique	Clients, utilisateurs finaux, PO

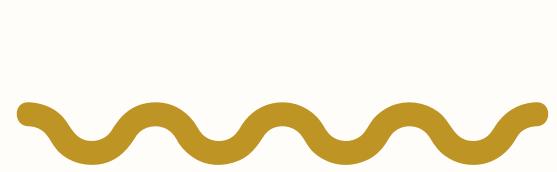
**Vérification** → Le bouton "payer" est-il **vert** comme prévu sur le plan ?

**Validation** → Est-ce que le processus de paiement est simple et sécurisé pour l'utilisateur ?

→ **Il faut faire les deux !**



# I.3 Les 7 principes fondamentaux du test (ISTQB)



## Principe 1 – Les tests montrent la présence de défauts, pas leur absence.

### Explication :

Les tests permettent de révéler les défauts, mais ne prouvent pas que le logiciel est parfait.

### Exemple concret :

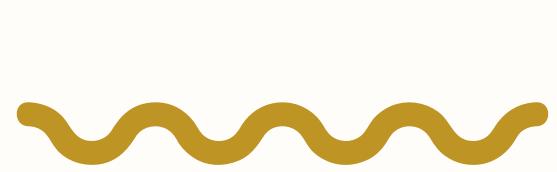
- On teste une calculatrice et on découvre que  $5 + 3 = 7$  au lieu de 8.
- Ce test met en évidence un bug... mais rien ne garantit qu'il n'y en ait pas d'autres ailleurs.



### Message clé :

"Tester, c'est comme chercher des aiguilles dans une botte de foin - quand on en trouve, on sait qu'il y en a, mais on ne sait pas combien il en reste."





## Principe 2 : Les tests exhaustifs sont impossibles

Tout tester, c'est tester... pour l'éternité.

### Explication :

Tester toutes les combinaisons de données, de navigateurs, de parcours et de configurations prendrait un temps et des ressources infinies ..

### Exemple concret :

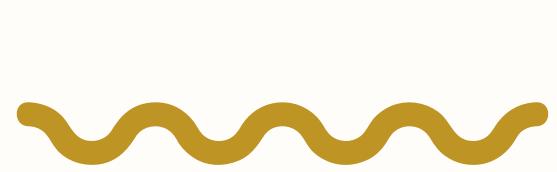
- Un formulaire avec 10 champs ayant chacun 5 valeurs possibles
- $\rightarrow 5^{10} = 9\ 765\ 625$  combinaisons à tester !
- En pratique, on sélectionne les cas critiques et représentatifs



### Message clé :

Le testeur ne teste pas tout — il teste ce qui compte :

→ le test, ce n'est pas la quantité, c'est la pertinence.



## Principe 3 : Tester tôt est économique

### Explication :

Plus un défaut est détecté tôt dans le cycle de développement, moins il coûte cher à corriger.

### Exemple concret :

- Bug trouvé en conception → correction = 1 heure
- Même bug trouvé en production → correction = 100 heures (+ impact client)

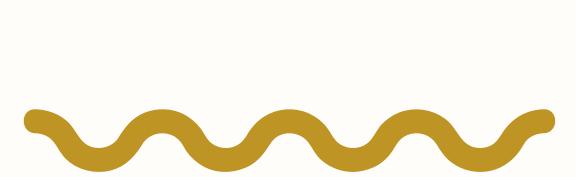


### Message clé :

"1 euro investi tôt en vaut 100 en production."

→Corriger un bug en conception, c'est changer une ligne de code.

En production, c'est éteindre un incendie."



## Principe 4 – Regroupement des défauts : la loi du 80/20 (Pareto)

La plupart des problèmes viennent d'une minorité de zones

### Explication :

La majorité des défauts se concentrent dans une minorité de modules ou de composants.

Ces zones sont souvent **complexes, anciennes** ou **souvent modifiées**, ce qui augmente le risque d'erreurs.

En test, tout n'a pas la même probabilité de casser

### Exemple concret :

- Sur 100 bugs trouvés, 80 proviennent de seulement 3 modules sur 20

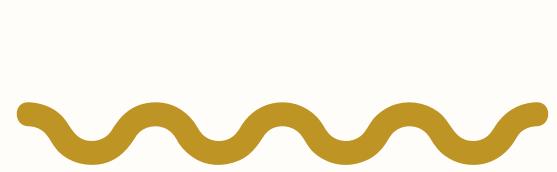


### Message clé :

"Concentrez vos efforts là où ça fait mal."

→ C'est toujours les mêmes modules qui cassent — testez-les plus, pas plus tard.





## Principe 5 : Paradoxe du pesticide

Trop de routine tue le test.

### Explication :

Si on répète toujours les mêmes tests, on ne trouve plus de nouveaux défauts.

Le système évolue, mais pas les tests : ils deviennent inefficaces face aux nouvelles erreurs.

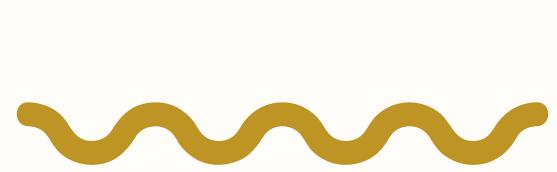
### Exemple concret :

- Pendant 6 mois, les mêmes tests automatisés passent toujours
- Soudain, un nouveau type de bug apparaît et n'est pas détecté
- Résultat : tout est "vert" dans le rapport... mais le produit est cassé



### Message clé :

"Faites évoluer vos tests comme évolue votre logiciel."



## Principe 6 : Les tests sont contextuels

Chaque projet a sa propre météo qualité.

### Explication :

La façon de tester dépend du contexte: type de projet, niveau de risque, exigences métier, budget et délais → chaque contexte impose ses priorités.

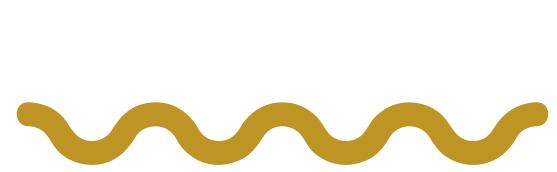
### Exemple concret :

- Application bancaire → priorité à la sécurité et à la fiabilité
- Jeu mobile → priorité à la performance et à l'expérience utilisateur



### Message clé :

"Adapter sa stratégie de test, c'est comme adapter sa tenue à la météo."



## Principe 7 : L'illusion de l'absence d'erreur

Un logiciel sans bug n'est pas forcément un bon produit.

### Explication :

Un logiciel sans défaut apparent n'est pas forcément utilisable ou adapté aux besoins réels des utilisateurs.

### Exemple concret :

- Application techniquement parfaite mais que personne n'utilise car inadaptée aux besoins utilisateur
- Fonctionnalités implémentées correctement mais qui ne résolvent pas le vrai problème



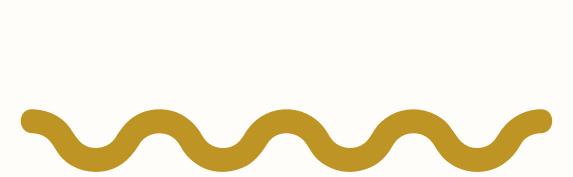
### Message clé :

"Zéro bug ≠ succès garanti. Pensez valeur business !"

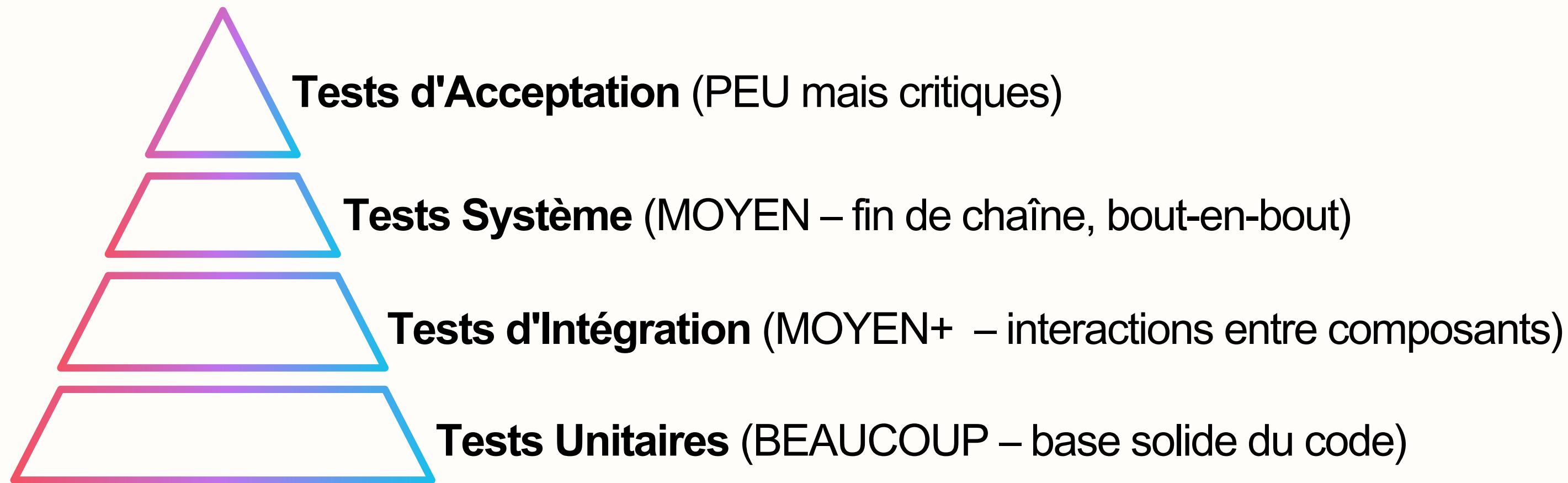


## I.4 La pyramide des tests : Organisation par niveaux

#### 1.4.1 Présentation des 4 niveaux fondamentaux



**Pyramide idéale** : la pyramide illustre la répartition idéale des tests selon leur **niveau**, leur **coût** et leur **rapidité d'exécution**



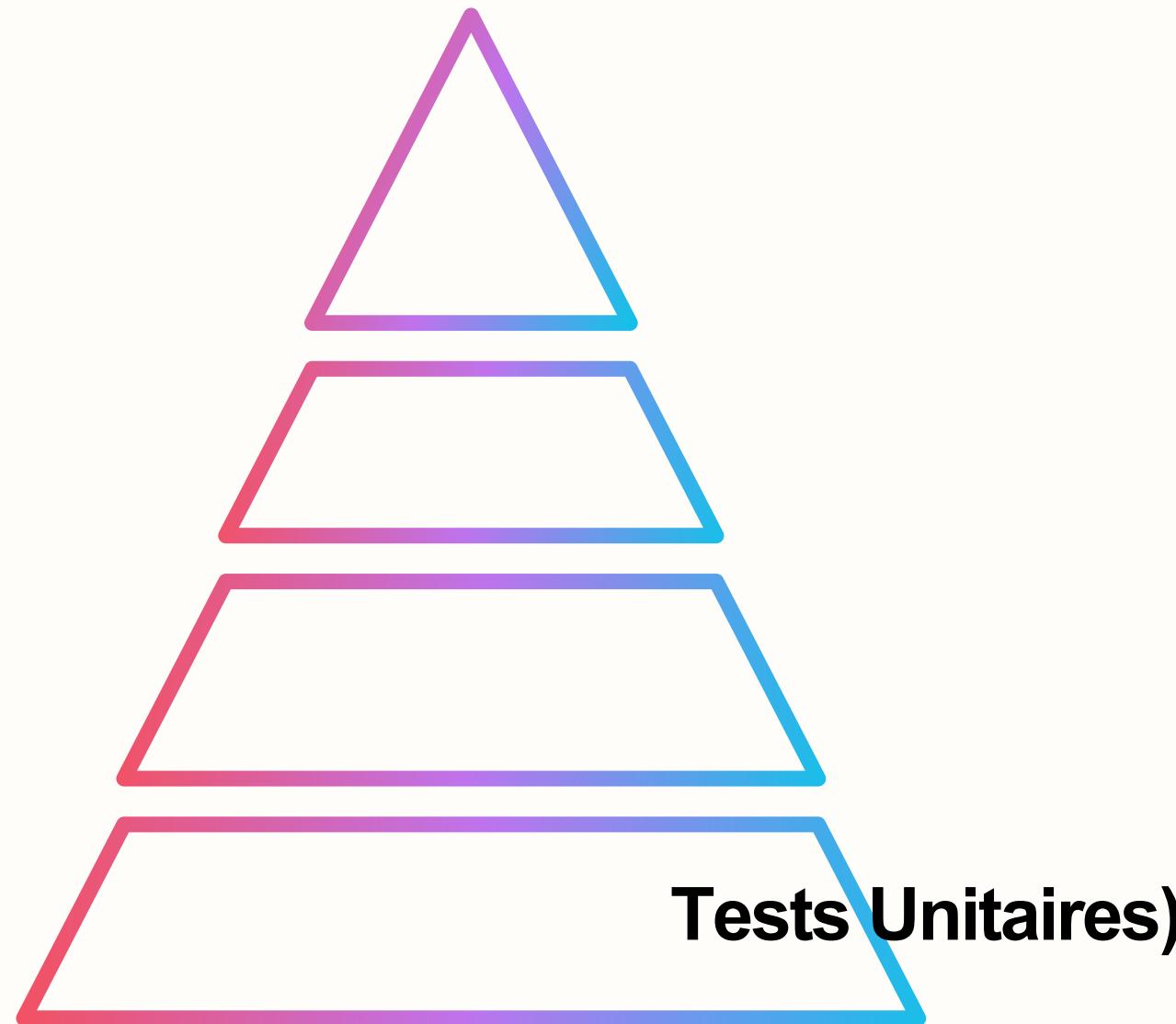
**Plus on descend, plus on automatise ; plus on monte, plus on valide la valeur.**

## 1.4.1 Présentation des 4 niveaux fondamentaux



### A. Tests Unitaires

Le premier rempart contre les régressions.

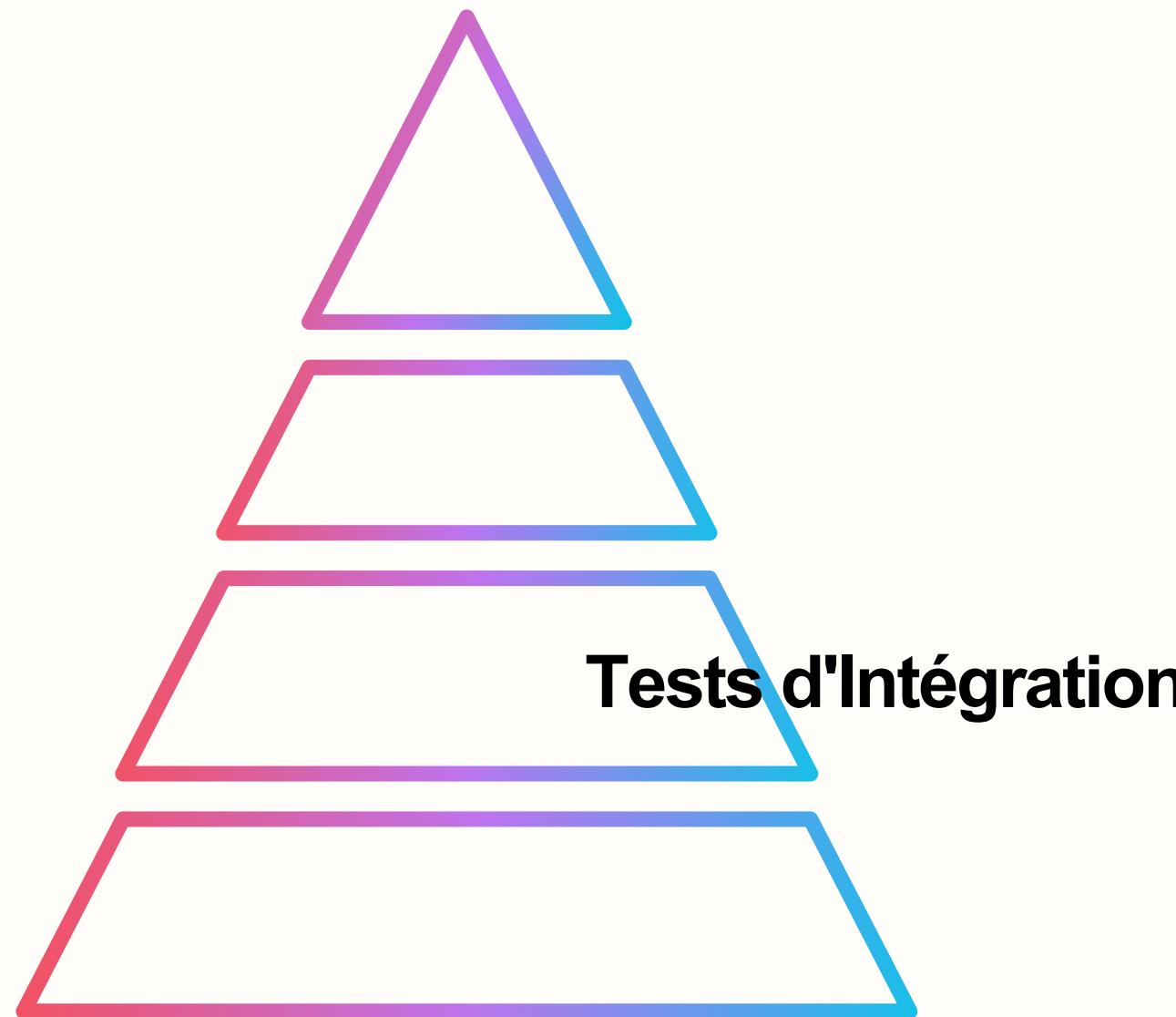


- **Objectif** : Vérifier le bon fonctionnement d'une unité de code (fonction, méthode, classe)
- **Périmètre** : Code isolé (mocking des dépendances)
- **Responsables** : Développeurs
- **Fréquence** : À chaque commit, exécution très rapide
- **Couverture cible** : 70-80% des règles business



### B. Tests d'Intégration

S'assurer que les pièces du puzzle fonctionnent ensemble.



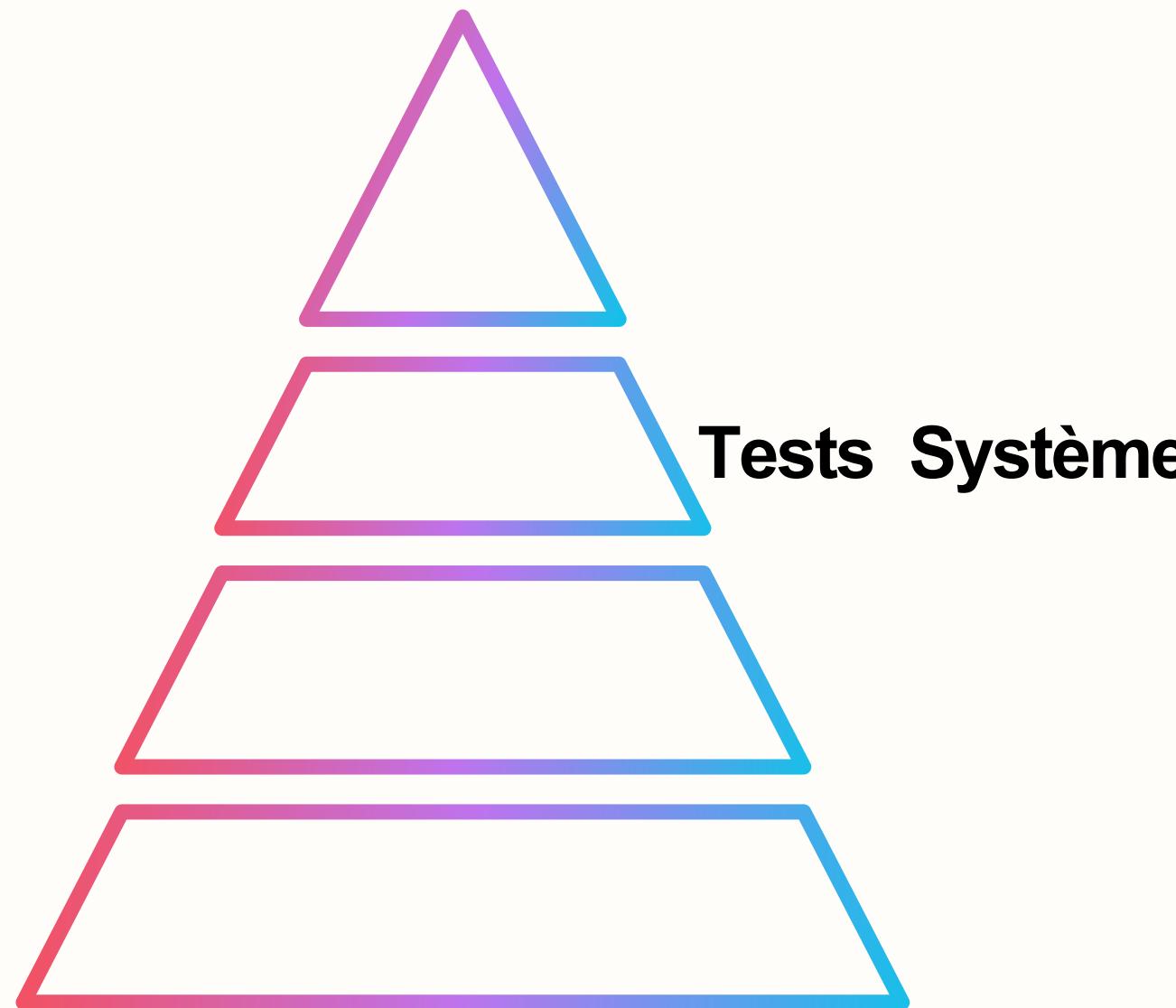
- **Objectif** : Vérifier que les différents composants interagissent correctement entre eux (interfaces, flux, échanges de données).
- **Périmètre** : 2+ composants, bases de données, APIs externes
- **Responsables** : Développeurs/testeurs techniques
- **Fréquence** : Plusieurs fois par jour dans le cadre des builds CI/CD
- **Focus** : Flux de données, API ..

## 1.4.1 Présentation des 4 niveaux fondamentaux



### C. Tests Système

Valider le fonctionnement global du produit dans un environnement réaliste.

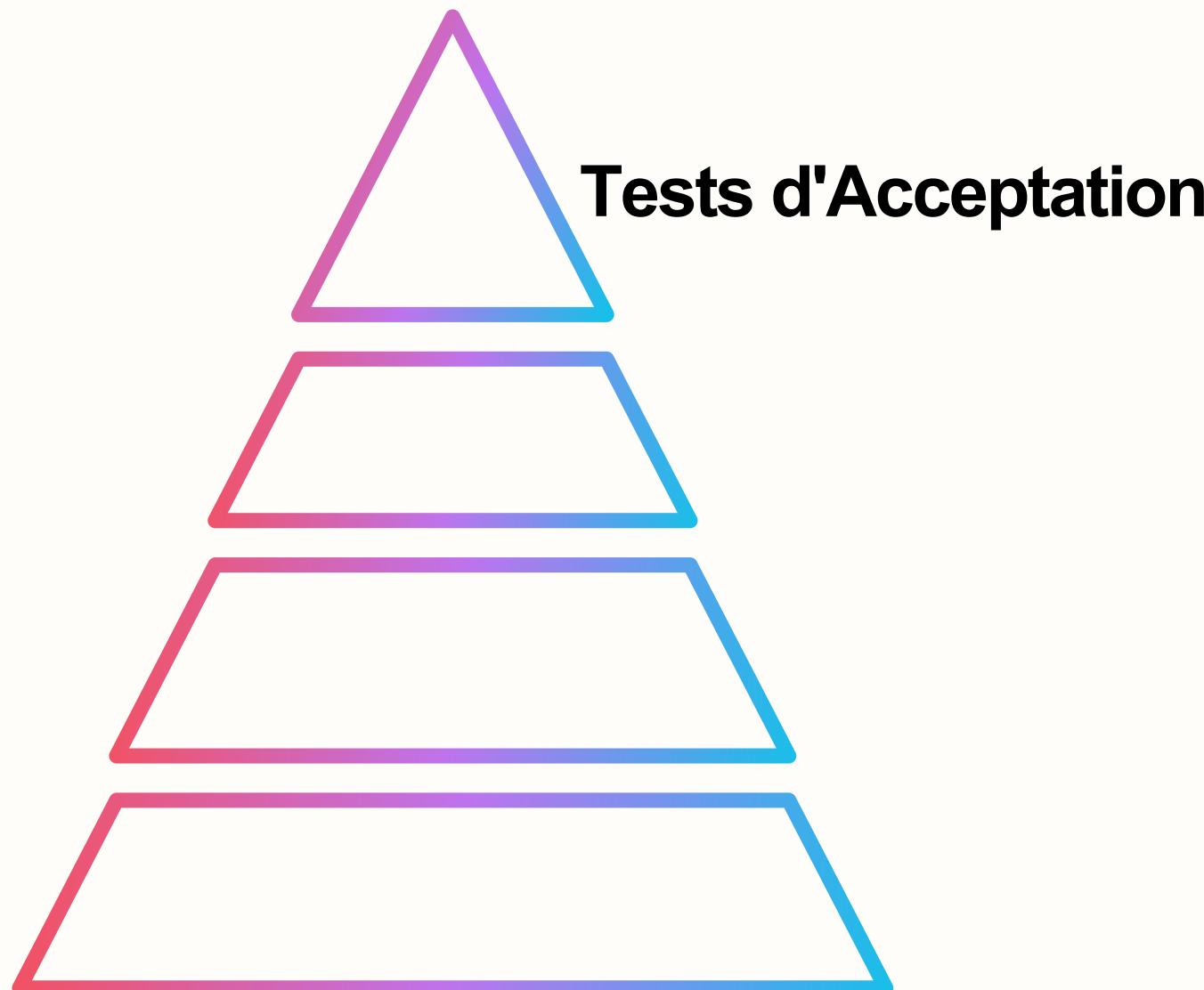


- **Objectif** : Vérifier le système complet from end-to-end
- **Périmètre** : Environnement de pré-production
- **Responsables** : Équipe QA / testeurs fonctionnels.
- **Fréquence** : généralement chaque nuit ou à chaque déploiement majeur.
- **Focus** : Tests fonctionnels, tests de performance, sécurité, compatibilité, et scénarios utilisateurs critiques.



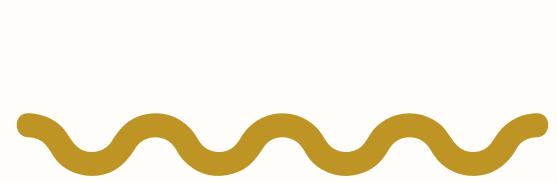
### D- Tests d'Acceptation

La validation finale : prouver que le produit répond au besoin réel.



- **Objectif** : Validation finale par le client/utilisateur
- **Périmètre** : Scénarios métier complets, sur un environnement identique à la production.
- **Responsables** : Clients/Product Owners
- **Fréquence** : Avant la mise en production
- **Focus** : Valeur business, expérience utilisateur

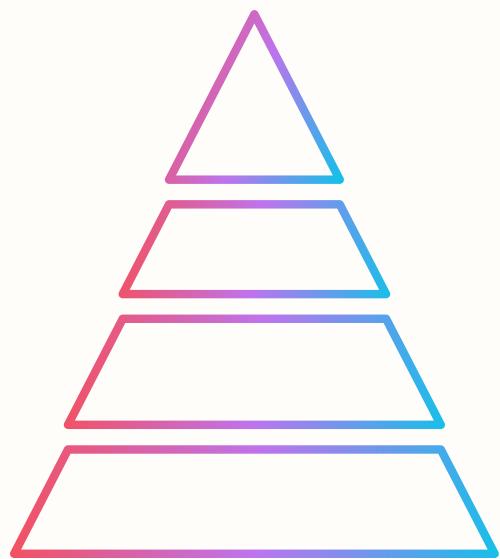
## 1.4.1 Présentation des 4 niveaux fondamentaux



## 1.4.2 Pourquoi la pyramide est efficace ?

Moins de coûts, plus de valeur et de feedback rapide.

Aspect	Tests Unitaires	Tests E2E
Coût création	Faible	Élevé
Temps exécution	Secondes	Minutes/heures
Maintenance	Facile	Complexe
Debugging	Simple	Complexe
Feedback	Immédiat	Différé



→ **Plus on monte**, plus les tests sont **coûteux et lents**.

Si on inverse cette logique, la qualité devient un frein au lieu d'un levier



# I.4 Méthodologies de Test Avancées

Des approches modernes pour tester mieux, plus vite et plus intelligemment

## 1.4.1 Risk-Based Testing

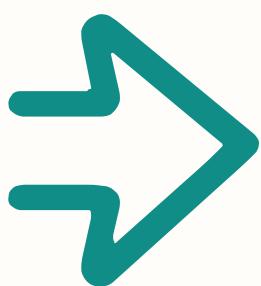
Concentrer les efforts là où le risque est le plus fort.



### Matrice de risque avancée :

#### FACTEURS DE RISQUE :

- Complexité technique
- Criticité métier
- Fréquence d'utilisation
- Visibilité externe
- Historique de défauts

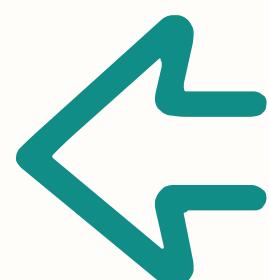


#### **Calcul du risque :**

$$\text{Risk Score} = (\text{Probabilité} \times \text{Impact}) + \text{Urgence}$$

- **Probabilité d'occurrence** du défaut
- **Impact métier** si le défaut survient
- **Urgence** de correction

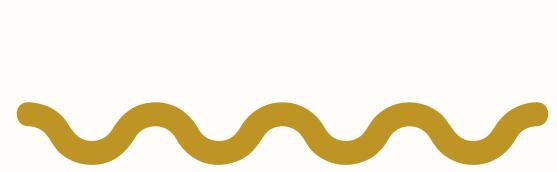
Niveau de risque	Type de tests recommandés
Elevé	Tests automatisés + tests manuels approfondis
Moyen	Tests automatisés de base
Faible	Tests exploratoires occasionnels



💡 Plus le **score** est élevé, plus la **priorité** de test augmente.



## 1.4.1 Risk-Based Testing - Exercices :



### Contexte :

Application e-commerce avec les fonctionnalités suivantes :

Connexion utilisateur

Recherche produit

Paiement carte bancaire

Envoi e-mail de confirmation

Gestion du panier

### Consigne :

Attribuez à chaque fonctionnalité une note de 1 à 5 selon :

Probabilité d'occurrence du défaut

Impact métier si le défaut survient

Urgence de correction

Calculez :

Risk Score = (Probabilité × Impact) + Urgence

### Tableau à remplir :

Fonctionnalité	Probabilité	Impact	Urgence	Risk Score	Priorité
Connexion					
Recherche produit					
Paiement CB					
Envoi e-mail					
Panier					

## 1.4.1 Risk-Based Testing - Exercices :



Fonctionnalité	Probabilité	Impact	Urgence	Risk Score	Priorité
Paiement CB	5	5	5	30	● Haute
Connexion	4	5	4	24	● Haute
Panier	3	4	3	15	● Moyenne
Recherche produit	2	3	2	8	● Basse
Envoi e-mail	2	2	2	6	● Basse

### Interprétation :

**Tests automatisés** → sur paiement et connexion (haute priorité, récurrents, critiques)

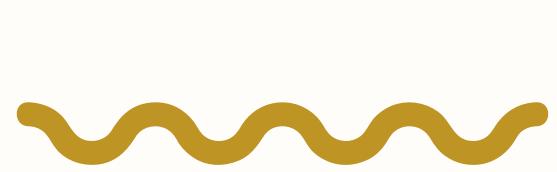
**Tests manuels exploratoires** → sur panier et recherche produit

**Tests occasionnels / non bloquants** → sur envoi e-mail



### Points clés à retenir :

- **La Qualité est un Investissement** : Ce n'est pas un coût, mais une assurance pour la réputation et la pérennité du produit.
- **Vérification vs Validation** : "Construit-on le produit BIEN ?" vs "Construit-on le BON produit ?". Les deux sont indispensables.
- **Les 7 Principes ISTQB** sont le fondement éthique et technique du métier de testeur. Ils rappellent les limites et les objectifs réels des tests.
- **La Pyramide des Tests** : Guide l'optimisation des efforts. Prioriser les tests unitaires (nombreux, rapides) et monter progressivement vers les tests métier.
- **Penser Risque** : Le Risk-Based Testing permet de concentrer les efforts là où le risque métier et technique est le plus fort.



### 1. Selon le principe fondamental des tests, lequel des énoncés suivants est VRAI ?

- a) Les tests exhaustifs sont possibles si l'équipe dispose de suffisamment de temps.
- b) Les tests montrent la présence de défauts, mais ne prouvent pas leur absence.
- c) L'absence de défauts prouve que le logiciel est de haute qualité.
- d) Tester tôt est toujours plus coûteux.

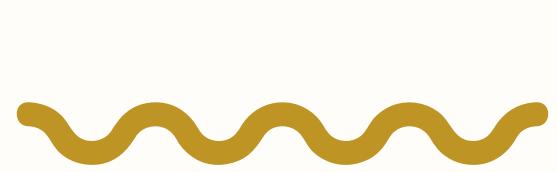


### 1. Selon le principe fondamental des tests, lequel des énoncés suivants est VRAI ?

- a) Les tests exhaustifs sont possibles si l'équipe dispose de suffisamment de temps.
- b) Les tests montrent la présence de défauts, mais ne prouvent pas leur absence.**
- c) L'absence de défauts prouve que le logiciel est de haute qualité.
- d) Tester tôt est toujours plus coûteux.

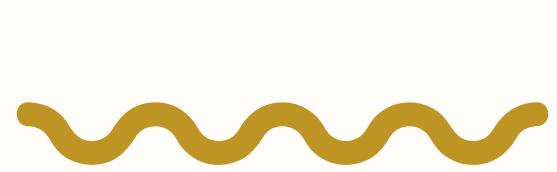
#### ➤ Correction : B

Explication : C'est le Principe 1 de l'ISTQB. Les tests peuvent révéler des défauts, mais même avec une couverture étendue, il est impossible de prouver qu'un logiciel ne contient pas des défauts.



### 2. Quel est l'objectif principal des tests logiciels, selon l'ISTQB ?

- a) Prouver que le logiciel ne contient aucun défaut.
- b) Exécuter tous les cas de test possibles pour être exhaustif.
- c) Fournir de la confiance et trouver des défauts.
- d) Valider uniquement les fonctionnalités critiques.

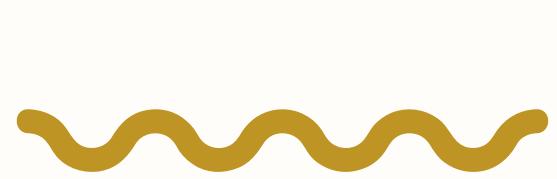


### 2. Quel est l'objectif principal des tests logiciels, selon l'ISTQB ?

- a) Prouver que le logiciel ne contient aucun défaut.
- b) Exécuter tous les cas de test possibles pour être exhaustif.
- c) Fournir de la confiance et trouver des défauts.**
- d) Valider uniquement les fonctionnalités critiques.

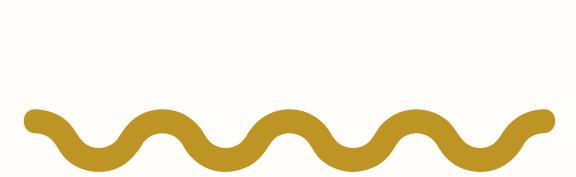
➤ Correction : C

Explication : Les tests ont pour objectifs principaux de détecter les défauts, de fournir de la confiance aux parties prenantes sur le niveau de qualité et de fournir des informations pour aider à la décision.



**3. Dans la pyramide des tests, quel niveau de test devrait être le plus nombreux et le plus automatisé ?**

- a) Tests d'acceptation
- b) Tests système
- c) Tests d'intégration
- d) Tests unitaires

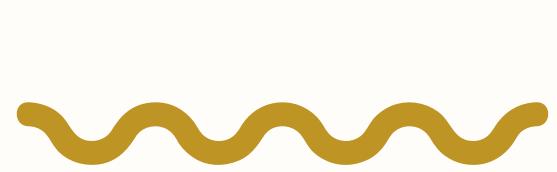


### 3. Dans la pyramide des tests, quel niveau de test devrait être le plus nombreux et le plus automatisé ?

- a) Tests d'acceptation
- b) Tests système
- c) Tests d'intégration
- d) Tests unitaires**

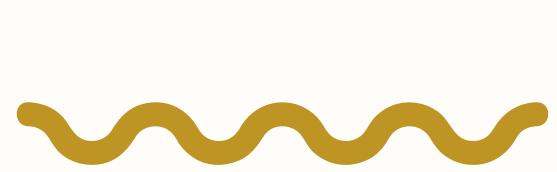
➤ Correction : D

Explication : La base de la pyramide est constituée des tests unitaires. Ils sont les plus rapides à exécuter, les moins chers à créer et à maintenir, et fournissent un feedback immédiat aux développeurs.



**4. Le fait qu'un défaut détecté pendant la phase de conception coûte beaucoup moins cher à corriger que s'il est détecté en production illustre quel principe ?**

- a) Le regroupement des défauts
- b) L'importance de tester tôt
- c) Le contexte de dépendance des tests
- d) L'illusion de l'absence d'erreur

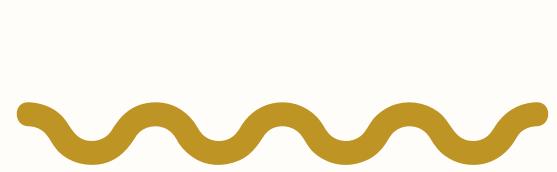


**4. Le fait qu'un défaut détecté pendant la phase de conception coûte beaucoup moins cher à corriger que s'il est détecté en production illustre quel principe ?**

- a) Le regroupement des défauts
- b) L'importance de tester tôt**
- c) Le contexte de dépendance des tests
- d) L'illusion de l'absence d'erreur

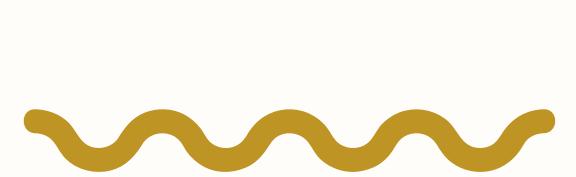
➤ **Correction : B**

Explication : C'est le Principe 3 de l'ISTQB : "Tester tôt est économique". Plus un défaut est détecté tard dans le cycle de vie, plus son coût de correction est élevé.



**5. Lors de la planification des tests, il est décidé de se concentrer davantage sur les modules à forte complexité technique et à haut impact métier. Cette approche est connue sous le nom de :**

- a) Test basé sur l'expérience.
- b) Test basé sur le risque (Risk-Based Testing).
- c) Test exploratoire.
- d) Test par équivalence.

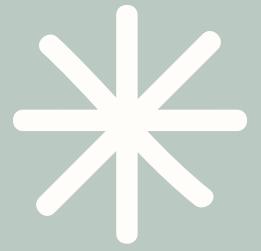


**5. Lors de la planification des tests, il est décidé de se concentrer davantage sur les modules à forte complexité technique et à haut impact métier. Cette approche est connue sous le nom de :**

- a) Test basé sur l'expérience.
- b) Test basé sur le risque (Risk-Based Testing).**
- c) Test exploratoire.
- d) Test par équivalence.

➤ **Correction : B**

Explication : Le test basé sur le risque est une technique de planification qui consiste à identifier les zones à plus haut risque et à y concentrer les efforts de test pour optimiser l'efficacité et la couverture.



## **Chapitre 2 : Conception et Organisation des Tests**



## Chapitre 2 : Conception et Organisation des Tests



- **2.1 Le Processus Fondamental du Test**
  - 2.1.1 Vue d'ensemble du processus structuré
- **2.2 Concevoir des Cas de Test Efficaces**
  - 2.2.1 Les caractéristiques d'un bon cas de test (CRUUSC)
  - 2.2.2 Structure standard d'un cas de test
- **2.3 Introduction à la Matrice de Traçabilité**
  - 2.3.1 Définition et objectifs
  - 2.3.2 Structure de la matrice
  - 2.3.3 Métriques de couverture
- **2.4 Tests de Régression & Confirmation**
  - 2.4.1 Tests de Régression
  - 2.4.2 Tests de Confirmation

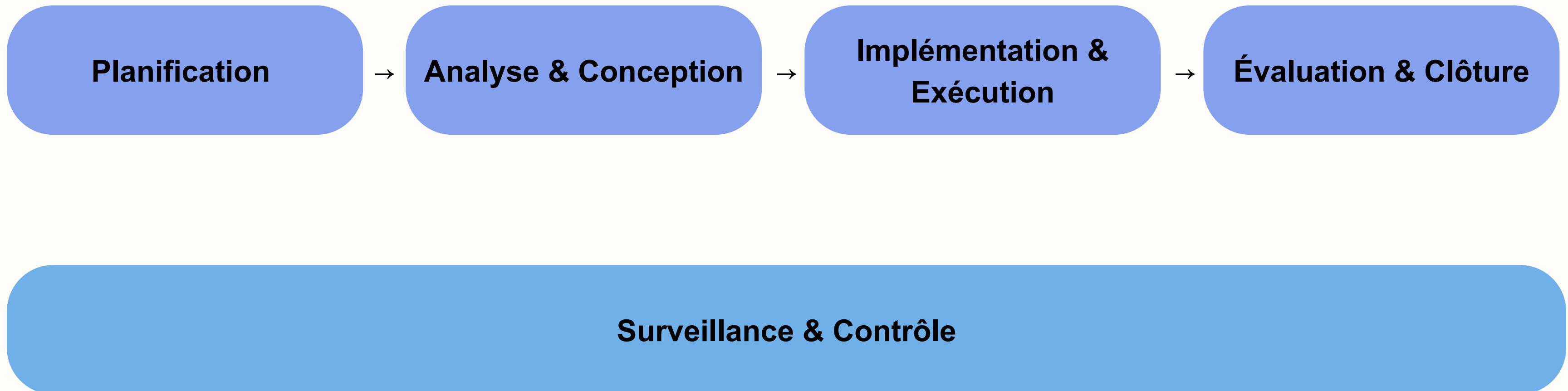


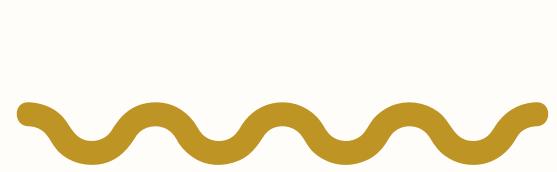
## 2.1 Le Processus Fondamental du Test

## 2.1.1 Vue d'ensemble du processus structuré



**Cycle de vie standardisé (ISTQB) :**





### Phase 1 : Planification stratégique : “Définir la route avant de tester.”



#### Objectifs de la planification :

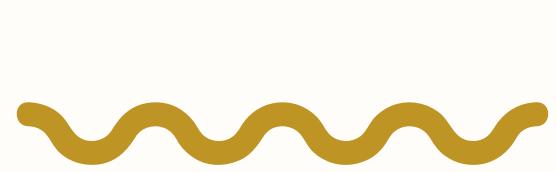
- Définir le **périmètre** et les **objectifs** de test
- Estimer l'**effort** et les **ressources** nécessaires
- Définir la **stratégie** de test globale

#### Livrables de planification :

- **Plan de Test (Test Plan)**

Un plan de test bien conçu, c'est la boussole du projet qualité

1. INTRODUCTION
  - Portée et objectifs
  - Références (spécifications, réglementations)
2. STRATÉGIE DE TEST
  - Niveaux de test couverts
  - Types de test (fonctionnel, performance, sécurité)
  - Critères d'entrée/sortie pour chaque niveau
3. RESSOURCES ET ENVIRONNEMENTS
  - Équipe (rôles et responsabilités)
  - Environnements de test (DEV, INT, PREPROD)
  - Outils (Jira/Xray, Selenium, JMeter)
4. PLANIFICATION TEMPORELLE
  - Jalons principaux
  - Estimation effort (hommes/jours)
  - Livrables et dates clés
5. RISQUES ET MITIGATION
  - Risques identifiés (techniques, ressources, planning)
  - Plans de contournement



### Phase 2 : Analyse et Conception

“Transformer les exigences en cas de test concrets.”

#### Activités clés :

- Analyser les **bases de test** (spécifications, user stories, diagrammes)
- Identifier les **conditions de test** (ce qu'il faut vérifier)
- **Concevoir les cas de test** et les données associées
- Définir l'architecture des tests automatisés (si applicable)

Modèle de cas de test												
Nom du projet	Créé par	Description	Date de	Date de	ID du cas de test	Conditions préalables	Scénario du cas de test	Cas de test	Données de test	Résultat attendu	Résultat réel	Statut d'exécution

“Un bon test case décrit **quoi** vérifier, **comment** et **pourquoi**.”



### Phase 3 : Implémentation et Exécution

“Passer du plan à l'action”

#### Séquence d'activités :

1. **Implémentation** : Développer et prioriser les procédures de test. Créer les suites de test.
2. **Préparation** : Configurer l'environnement et préparer les données.
3. **Exécution** : Exécuter les procédures de test.

#### Processus d'exécution :

- Exécuter les cas de test **manuellement** ou **automatiquement**.
- **Comparer** les résultats **obtenus** aux résultats **attendus**.
- Signaler les écarts ou **anomalies** détectés.
- Effectuer les **re-tests** après correction des défauts.
- Lancer les tests de **non-régression** pour vérifier les impacts collatéraux.





### Phase 4 : Évaluation et Clôture

#### Critères d'évaluation :

##### A. Critères de Sortie (Exit Criteria)

- **Couverture des exigences** : 100% des critères critiques
- **Taux de réussite** :  $\geq 95\%$  des cas de test passés
- **Défauts bloquants** : Aucun défaut critique/bloquant ouvert
- **Performance** : Objectifs de performance atteints

“Clôturer, c'est mesurer la **confiance**, pas seulement la couverture.”

##### B. Rapport de Clôture

###### # RAPPORT DE CLÔTURE DE TEST

###### ## Résumé Exécutif

- Objectifs atteints/partiellement atteints
- Principaux risques résiduels

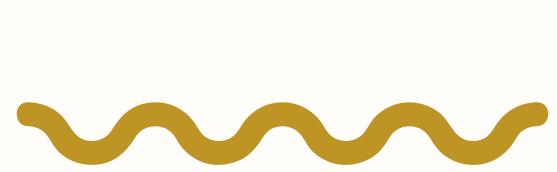
###### ## Métriques Clés

- Nombre de cas de test exécutés/passés/échoués
- Densité de défauts par composant
- Tendance de fermeture des défauts

###### ## Recommandations

- Aptitude à la mise en production : OUI/NON/AVEC RÉSERVES
- Zones à surveiller en production
- Améliorations pour les prochains cycles

## 2.1 Le Processus Fondamental du Test : Résumé



### Phase 1 – Planification stratégique

- Déterminer le périmètre, les ressources et les risques.
- Produire le plan de test et les critères de sortie (DOD)

### Phase 2 – Analyse & Conception

- Identifier les conditions de test.
- Concevoir les cas de test et jeux de données.
- Préparer les procédures et scripts.

### Phase 3 – Implémentation & Exécution

- Mettre en place les environnements de test.
- Exécuter les tests manuels / automatisés.
- Documenter les résultats et les anomalies.

### Phase 4 – Évaluation & Clôture

- Comparer résultats vs objectifs.
- Évaluer les risques résiduels.
- Rédiger le rapport de clôture et décider Go / No-Go.



## 2.2 Concevoir des Cas de Test Efficaces

## 2.2.1 Anatomie d'un bon cas de test



### Caractéristiques d'un cas de test efficace (CRUSC) :

<b>Complet</b>	Toutes les informations nécessaires sont présentes (préconditions, étapes, données, résultats attendus).
<b>Reproductible</b>	Le même résultat est obtenu peu importe qui exécute le test.
<b>Univoque</b>	Les étapes sont claires, sans ambiguïté ni interprétation possible.
<b>Spécifique</b>	Le test vérifie une seule condition à la fois.
<b>Concis</b>	Aucun pas inutile, chaque étape a une valeur.



un bon cas de test peut être lu et compris par quelqu'un qui n'a jamais vu l'application

## 2.2.2 Structure standard d'un cas de test



### Template détaillé :



IDENTIFIANT: TC-LOGIN-001

TITRE: Connexion réussie avec identifiants valides

COMPOSANT: Module d'authentification

PRIORITÉ: Haute

TYPE: Fonctionnel

AUTEUR: John Doe

DATE: 2024-01-15

PRÉCONDITIONS:

1. L'application est accessible via l'URL <https://app.example.com>
2. Un utilisateur test existe (login: "testuser", password: "Test123!")

DONNÉES DE TEST:

- Login: testuser
- Password: Test123!

ÉTAPES:

1. Naviguer vers la page de login
2. Saisir "testuser" dans le champ "Nom d'utilisateur"
3. Saisir "Test123!" dans le champ "Mot de passe"
4. Cliquer sur le bouton "Se connecter"

RÉSULTAT ATTENDU:

1. L'utilisateur est redirigé vers la page d'accueil
2. Le message "Connexion réussie" s'affiche
3. Le nom d'utilisateur "testuser" apparaît dans l'en-tête
4. Les menus utilisateur sont accessibles

RÉSULTAT RÉEL: [À remplir pendant l'exécution]

STATUT: [Pass/Échec/Bloqué/Non Exécuté]

## 2.2.4 – Erreurs fréquentes dans l'évaluation de la priorité



### 1. Confondre Impact et Fréquence

Ex. : “Le logo mal aligné est vu par tous → donc c'est critique.”

**Non X** : il a une forte fréquence, mais un faible impact métier.

Toujours raisonner en valeur business, pas en visibilité.

### 2. Sous-estimer les impacts cachés

“Le bug n'arrive qu'en cas limite, donc pas grave.”

**Erreur** : un défaut rare peut casser un processus critique (paiement, sécurité).

Évaluer toujours le risque métier associé à chaque scénario.



## 2.3 Introduction à la Matrice de Traçabilité

## 2.4.1 Définition et objectifs



“Relier chaque exigence à ses tests, ses défauts et ses livrables.”

### Qu'est-ce que la traçabilité ?

La **traçabilité**, c'est la capacité à relier chaque exigence aux artefacts produits tout au long du cycle de vie logiciel  
(spécifications, cas de test, anomalies, livrables, etc.)

### Objectifs principaux :

- 1. Couverture** : Vérifier que toutes les exigences sont testées
- 2. Impact analysis** : Comprendre l'impact des changements
- 3. Reporting** : Communiquer l'avancement aux parties prenantes

“Sans traçabilité, on teste — mais on ne sait pas quoi on teste, ni pourquoi.”

## 2.4.2 Structure de la matrice



**Format basique :**

Exigence ID	Description Exigence	Cas de Test ID	Priorité	Statut	Couverture
REQ-001	User login	TC-LOGIN-001	Haute	Pass	100%
REQ-001	User login	TC-LOGIN-002	Moyenne	Pass	100%
REQ-001	User login	TC-LOGIN-003	Basse	Non exécuté	100%
REQ-002	Password reset	TC-PWD-001	Haute	Échec	100%
REQ-003	User profile	TC-PROF-001	Haute	Pass	100%
REQ-004	Email notifications	-	-	-	0%

## 2.4.3 Métriques de couverture



### Formules de calcul :

#### Couverture des exigences :

Couverture = (Nombre d'exigences couvertes / Total d'exigences) × 100

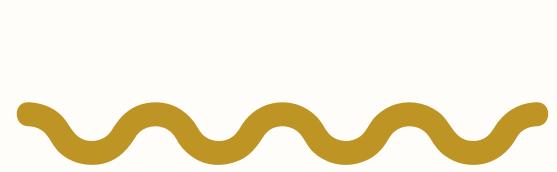
#### Couverture des tests :

Couverture = (Nombre de cas de test exécutés / Total de cas de test) × 100

#### Taux de réussite :

Taux de réussite = (Nombre de tests passés / Nombre de tests exécutés) × 100

## Cas 1 – Couverture des exigences



**Contexte :** Un projet comporte 120 exigences fonctionnelles. L'équipe de test a couvert 90 d'entre elles avec des cas de test.

**Question :** Calculez la couverture des exigences.

**Formule :** **Couverture = (Nombre d'exigences couvertes / Total d'exigences) × 100**

**Réponse attendue :** Couverture =  $(90 / 120) \times 100 = 75\%$

## Cas 2 – Couverture des tests



**Contexte** : Sur un total de 200 cas de test prévus, 160 ont été exécutés.

**Question** : Calculez la couverture des tests.

**Formule** : **Couverture = (Nombre de cas de test exécutés / Total de cas de test) × 100**

**Réponse attendue** : Couverture des tests =  $(160 / 200) \times 100 = 80\%$

## Cas 3 – Taux de réussite



**Contexte :** Parmi les 160 cas de test exécutés, 140 ont été passés avec succès.

**Question :** Calculez le taux de réussite.

**Formule :** **Taux de réussite = (Nombre de tests passés / Nombre de tests exécutés) × 100**

**Réponse attendue :** Taux de réussite =  $(140 / 160) \times 100 = 87.5\%$



## 2.4 Tests de Régression & Confirmation



### Tests de Confirmation

#### Definition:

- | Les **tests de confirmation** sont exécutés **après la correction d'un défaut** pour vérifier que celui-ci a bien été **résolu**.
  - Ils valident le **correctif lui-même**, pas les autres fonctionnalités.
- Processus en 4 étapes :

#### Procédure typique :

1. Identifier le **cas de test lié au défaut** (ex. TC-BUG-045).
2. Rejouer **exactement le même scénario**.
3. Comparer le résultat obtenu avec le **résultat attendu post-correction**.
4. Si tout est conforme → le défaut passe à l'état “**Closed**” dans Jira/Xray.



## 2.4.2 Tests de Régression & Confirmation



### Tests de Régression (1/2)

“S’assurer que ce qui marchait **hier** marche encore **aujourd’hui**.”

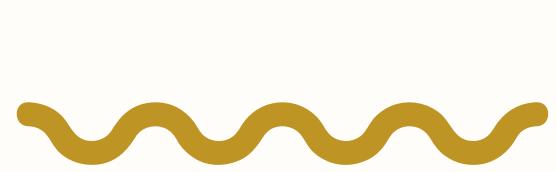
#### Définition:

| Ensemble de tests qui garantissent que les modifications ne dégradent pas les fonctionnalités existantes.

#### Quand les exécuter ?

- Après chaque modification de code
- Avant chaque mise en production
- Durant les sprints Agile
- Après des refactoring importants





### Tests de Régression (2/2)

#### Stratégies clés :

- **Suite automatisée** : Les tests critiques (login, paiement, recherche...) sont rejoués automatiquement à chaque build CI/CD.
- **Sélection intelligente** : On teste uniquement les zones impactées par les changements récents.
- **Tests smoke** : Vérification rapide des fonctions vitales avant de lancer la campagne complète.
- **Priorisation par risque** : On exécute d'abord les cas liés aux modules à fort impact métier.

#### Exemple concret :

Après l'ajout d'une fonctionnalité “paiement en plusieurs fois”, il faut vérifier que :

- Le paiement simple fonctionne toujours,
- La connexion utilisateur reste opérationnelle,
- Le panier conserve les articles,
- Les commandes passées restent consultables.



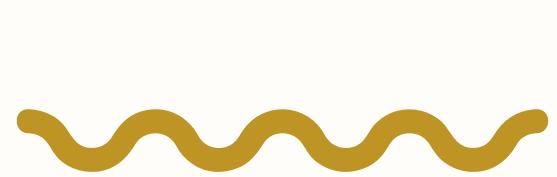
## Test de Régression VS Tests de Confirmation

	Test de Confirmation	Test de Régression
Objectif	Vérifier qu'un bug est corrigé	Vérifier qu'aucune autre partie n'est cassée
Portée	Limitée à un bug spécifique	Large (ensemble des fonctionnalités)
Fréquence	Après chaque correction	Avant chaque livraison
Automatisation	Souvent manuelle	Majoritairement automatisée



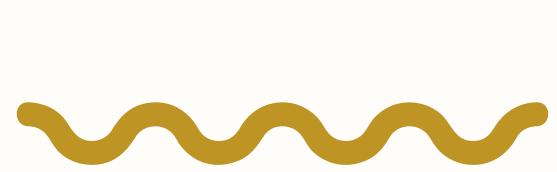
### Points clés à retenir :

- **Processus Structuré** : Suivre le cycle Planification → Conception → Exécution → Clôture garantit un testing rigoureux et mesurable.
- **CRUSC** : Un bon cas de test est **Complet**, **Reproductible**, **Univoque**, **Spécifique** et **Concis**.
- **Traçabilité** : La matrice de traçabilité est essentielle pour assurer la couverture des exigences et analyser l'impact des changements.
- **Priorité & Criticité** : Évaluer un défaut en fonction de son urgence de correction (priorité) et de son impact métier (criticité).
- **Régression vs Confirmation** : Le test de confirmation valide un correctif ; le test de régression protège les fonctionnalités existantes.



### 1. Quel est l'objectif principal d'une Matrice de Traçabilité ?

- a) Automatiser l'exécution des tests.
- b) Établir un lien entre les exigences et les cas de test.
- c) Calculer le coût total du projet de test.
- d) Déterminer la priorité des défauts.

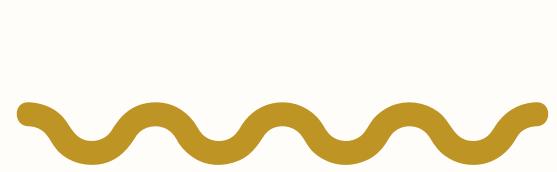


### 1. Quel est l'objectif principal d'une Matrice de Traçabilité ?

- a) Automatiser l'exécution des tests.
- b) Établir un lien entre les exigences et les cas de test.**
- c) Calculer le coût total du projet de test.
- d) Déterminer la priorité des défauts.

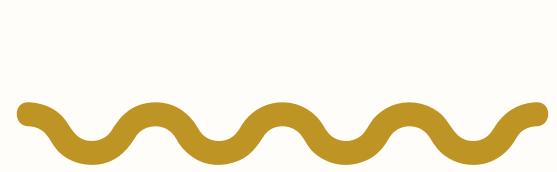
➤ **Correction : B**

Explication : La matrice de traçabilité a pour objectif principal de relier les exigences aux cas de test. Cela permet de s'assurer que chaque exigence est couverte par au moins un test (couverture) et de réaliser une analyse d'impact en cas de modification d'une exigence.



**5. Vous devez vérifier qu'un correctif de bug a bien résolu le problème initial sans avoir introduit de nouvelles erreurs. Quel type de test allez-vous principalement exécuter ?**

- a) Test de performance
- b) Test de régression
- c) Test de confirmation
- d) Test d'acceptation

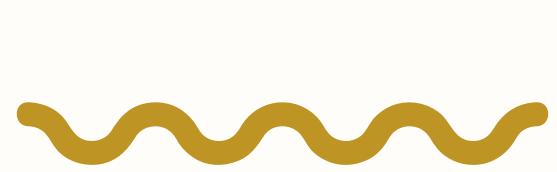


**2. Vous devez vérifier qu'un correctif de bug a bien résolu le problème initial sans avoir introduit de nouvelles erreurs. Quel type de test allez-vous principalement exécuter ?**

- a) Test de performance
- b) Test de régression
- c) **Test de confirmation**
- d) Test d'acceptation

➤ **Correction : C**

Explication : Le test de confirmation (ou re-test) a pour objectif spécifique de vérifier qu'un défaut préalablement rapporté a bien été corrigé. Le test de régression a un objectif plus large : s'assurer que les modifications n'ont pas dégradé les fonctionnalités existantes.



**3. Un "Test de Smoke" (test de fumée) a pour principal objectif de :**

- a) Tester toutes les fonctionnalités en profondeur.
- b) Vérifier rapidement que les fonctionnalités vitales de l'application sont opérationnelles après un déploiement.
- c) Simuler une charge utilisateur importante sur le système.
- d) Valider l'interface utilisateur avec des utilisateurs finaux.



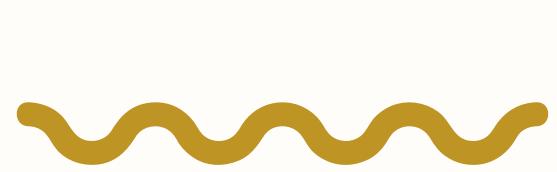
### 3. Un "Test de Smoke" (test de fumée) a pour principal objectif de :

- a) Tester toutes les fonctionnalités en profondeur.
- b) Vérifier rapidement que les fonctionnalités vitales de l'application sont opérationnelles après un déploiement.**
- c) Simuler une charge utilisateur importante sur le système.
- d) Valider l'interface utilisateur avec des utilisateurs finaux.

#### ➤ Correction : B

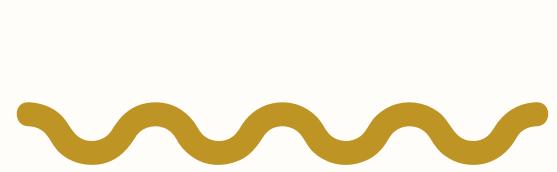
Explication : Un test de smoke est un sous-ensemble de la suite de régression.

Son but est de faire une vérification rapide des fonctionnalités les plus critiques ("vitales") après une nouvelle mise en build, pour décider si des tests plus poussés peuvent être lancés sans perte de temps.



### 4. Dans la phase d'évaluation et clôture, le testeur :

- a) Supprime tous les tests échoués
- b) Mesure la couverture, rédige le rapport et évalue les risques résiduels
- c) Replanifie le projet
- d) Automatise les tests

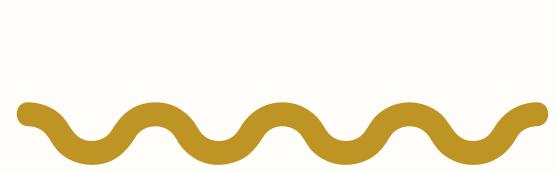


### 4. Dans la phase d'évaluation et clôture, le testeur :

- a) Supprime tous les tests échoués
- b) Mesure la couverture, rédige le rapport et évalue les risques résiduels**
- c) Replanifie le projet
- d) Automatise les tests

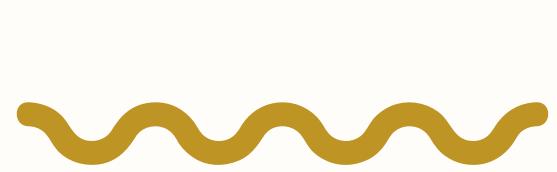
➤ Correction : B

Explication : Cette étape mesure les objectifs atteints, analyse les écarts et formalise la décision Go / No-Go.



### 5. Quelle erreur est fréquente dans la priorisation des tests ?

- a) Tester les cas critiques d'abord
- b) Confondre impact métier et fréquence d'occurrence
- c) Évaluer le risque avant de prioriser
- d) Documenter les anomalies

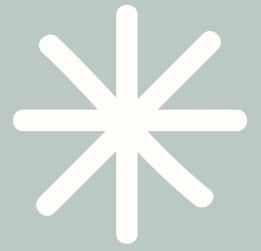


### 5. Quelle erreur est fréquente dans la priorisation des tests ?

- a) Tester les cas critiques d'abord
- b) Confondre impact métier et fréquence d'occurrence**
- c) Évaluer le risque avant de prioriser
- d) Documenter les anomalies

➤ **Correction : B**

Un bug très visible n'est pas forcément critique : la priorité dépend de l'impact business, pas seulement de la fréquence..



## **Chapitre 3 - Au-Delà du Fonctionnel - Performance, Sécurité et Stratégie Qualité**



## Chapitre 3 : Au-Delà du Fonctionnel



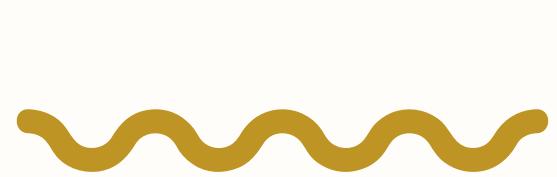
- **3.1 Tests Fonctionnels**
- **3.2 Tests Non-Fonctionnels**
  - 3.2.1 Tests de Performance
  - 3.2.2 Tests de Sécurité
  - 3.2.3 Tests d'Utilisabilité (UX)



## 3.1 Les Tests Essentiels au Métier

- Tests Fonctionnels -

### 3.1.1 Tests Fonctionnels



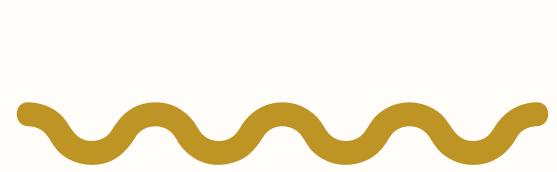
#### Objectif:

| S'assurer que chaque fonctionnalité répond exactement aux besoins métiers et aux exigences utilisateurs.

#### Exemples typiques :

Type de test	Exemple concret	Outil ou pratique courante
Test de saisie	Vérifier qu'un utilisateur peut créer un compte avec un email valide	Formulaire web
Test de recherche	Vérifier que la recherche renvoie les bons résultats	Scénario Selenium ou Postman
Test de panier	Ajouter un produit, appliquer un code promo, valider le paiement	Test fonctionnel e-commerce
Test d'affichage	Vérifier le rendu d'une page selon le rôle utilisateur	Test exploratoire ou automatisé

### 3.1.1 Tests Fonctionnels



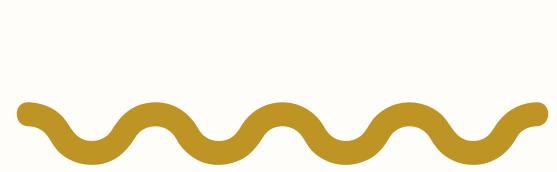
- **Les tests fonctionnels** garantissent le “**Quoi**” (ce que le système fait).
- **Les tests non fonctionnels** garantissent le “**Comment**” (la manière dont il le fait) — performance, sécurité, fiabilité, etc.



## 3.1 Les Tests Essentiels au Métier

-Tests Non-Fonctionnels-

### 3.1.1 Tests de Performance - Vérifier que ça tient la charge



#### Objectif:

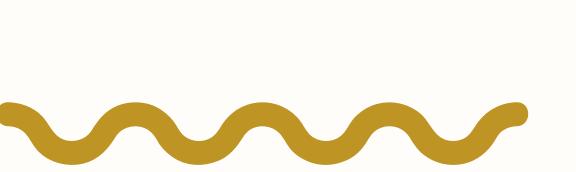
| S'assurer que l'application **supporte la charge, reste fluide et répond rapidement**, même sous une forte utilisation.

#### Pourquoi c'est important :

- **Expérience utilisateur** : une application lente fait fuir les utilisateurs.
- **Perte de revenus** : un site e-commerce qui plante pendant les soldes = catastrophe.
- **Exigence métier** : en entreprise, on attend du testeur qu'il **garantisse la stabilité et la scalabilité**.
- **Image de marque** : la performance influence directement **la réputation du produit**.

**“Les tests de performance protègent la satisfaction client et la réputation du produit.”**

### 3.1.1 Tests de Performance - Vérifier que ça tient la charge



Type	Objectif	Exemple concret
<b>Test de charge</b>	Vérifier le comportement à un volume d'utilisateurs attendu	500 utilisateurs simultanés
<b>Test de stress</b>	Pousser le système jusqu'à ses limites	Test jusqu'à crash du serveur
<b>Test d'endurance</b>	Vérifier la stabilité dans le temps	48h d'utilisation continue
<b>Test de montée en charge (scalabilité)</b>	Vérifier la capacité à absorber une croissance	+10% d'utilisateurs toutes les 5 min

#### Etude de cas :

Après un nouveau déploiement, une API “/checkout” passe de 200 ms à 1,2 s de temps de réponse moyen.

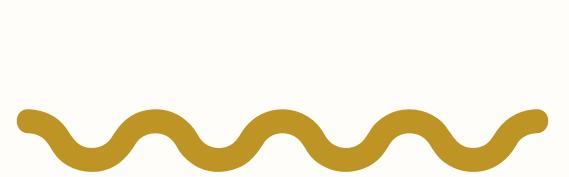
#### Questions :

Quelle hypothèse formules-tu ?

Quel type de test exécuterais-tu pour confirmer ?

Quelle serait ta recommandation finale à l'équipe de dev ?

### 3.1.1 Tests de Performance - Vérifier que ça tient la charge



#### Etude de cas :

Après un nouveau déploiement, une API “/checkout” passe de 200 ms à 1,2 s de temps de réponse moyen.

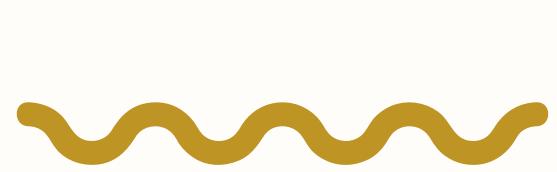
#### Réponse

**Hypothèse** → Saturation du serveur ou fuite mémoire après le nouveau déploiement.

**Type de test** → Test de charge (pour reproduire le trafic réel) puis test de stress (pour voir à quel seuil ça bloque).

**Recommandation** → Optimiser les requêtes SQL / équilibrer la charge serveur / vérifier la configuration réseau.

### 3.1.2 Tests de Sécurité - Protéger les données



#### Objectif :

| Vérifier que le système protège correctement les **données sensibles** et résiste aux **attaques malveillantes**.

Type de risque	Exemple concret
<b>Injection SQL</b>	username = 'admin' OR '1'='1' password = 'anything' Si ça marche → GROS problème
<b>Authentification faible</b>	Tentatives illimitées de login Risque de brute force
<b>Données non chiffrées</b>	Mot de passe stocké en clair HTTP au lieu de HTTPS

**Un testeur Agile ne cherche pas que les bugs fonctionnels — il vérifie aussi la robustesse face aux attaques.**

### 3.1.3 Tests d'Utilisabilité - L'expérience utilisateur



#### Objectif :

| S'assurer qu'un **produit fonctionne bien pour l'utilisateur final**, et pas seulement pour les développeurs.

“Un produit peut être fonctionnel... mais inutilisable s'il n'est pas pensé pour l'utilisateur.”

#### Ce que les utilisateurs attendent :

- **Simplicité** → trouver rapidement ce qu'ils cherchent.
- **Rapidité d'exécution** → peu de clics, navigation fluide.
- **Interface intuitive** → logique, cohérente, sans apprentissage nécessaire.



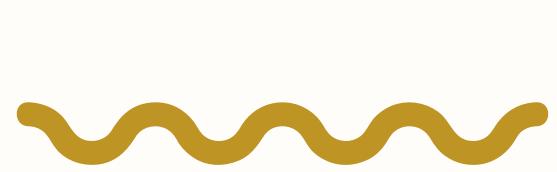
### Points clés à retenir :

- **Fonctionnel vs Non-Fonctionnel** : Le "quoi" vs le "comment". La qualité d'usage dépend autant des deux.
- **Tests de Performance** : Vérifient le comportement sous charge (Load), la stabilité dans le temps (Endurance) et les limites (Stress).
- **Tests de Sécurité** : Protègent les données et l'intégrité du système contre les menaces (Injection SQL, authentification...).
- **Tests d'Utilisabilité (UX)** : Un logiciel fonctionnel mais inutilisable est un échec. L'expérience utilisateur est primordiale.
- **Penser "Qualité Totale"** : La performance, la sécurité et l'UX sont des exigences business, pas des options techniques.



### 1. Le principal objectif des tests fonctionnels est de :

- a) Vérifier la performance du système
- b) Vérifier que le logiciel fait ce qu'il doit faire
- c) Tester la robustesse du code source
- d) Identifier les failles de sécurité



### 1. Le principal objectif des tests fonctionnels est de :

- a) Vérifier la performance du système
- b) Vérifier que le logiciel fait ce qu'il doit faire**
- c) Tester la robustesse du code source
- d) Identifier les failles de sécurité

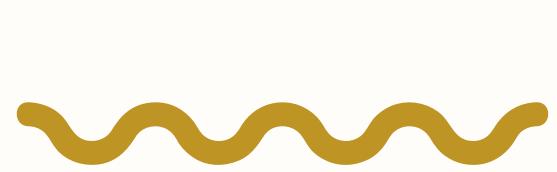
#### ➤ Correction : B

Les tests fonctionnels valident le “Quoi” (comportement attendu) selon les exigences métier, pas le “Comment”.



### 2. Les tests non fonctionnels évaluent principalement :

- a) Les spécifications métier
- b) La conformité du code aux standards internes
- c) Les qualités d'usage comme performance, sécurité, fiabilité
- d) Le nombre de tests exécutés

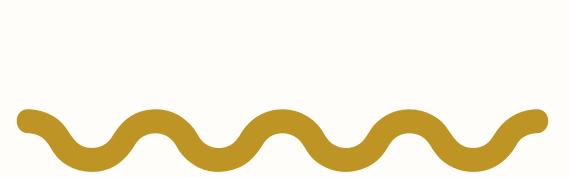


### 2. Les tests non fonctionnels évaluent principalement :

- a) Les spécifications métier
- b) La conformité du code aux standards internes
- c) **Les qualités d'usage comme performance, sécurité, fiabilité**
- d) Le nombre de tests exécutés

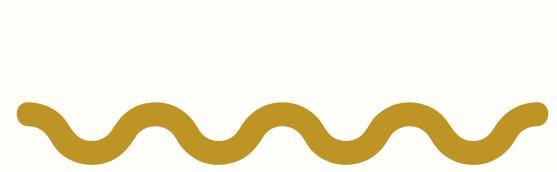
➤ **Correction : C**

Les tests non fonctionnels se concentrent sur les caractéristiques de qualité : performance, sécurité, utilisabilité, etc.



### 3. Un test de charge consiste à :

- a) Vérifier la stabilité après plusieurs jours d'exécution
- b) Évaluer le comportement sous un volume d'utilisateurs attendu
- c) Pousser le système jusqu'à son crash
- d) Simuler des attaques malveillantes

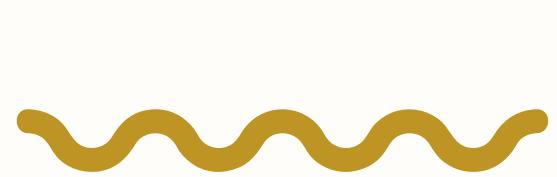


### 3. Un test de charge consiste à :

- a) Vérifier la stabilité après plusieurs jours d'exécution
- b) Évaluer le comportement sous un volume d'utilisateurs attendu**
- c) Pousser le système jusqu'à son crash
- d) Simuler des attaques malveillantes

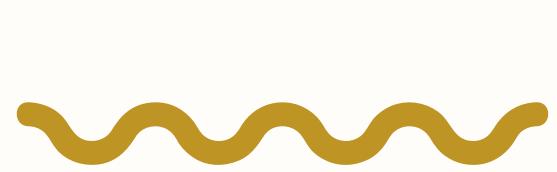
➤ **Correction : B**

Le test de charge reproduit le **trafic réel** pour vérifier la tenue en charge et la réponse du système sous contrainte normale.



### 4. Le test de stress vise à :

- a) Vérifier la rapidité du login
- b) Tester uniquement les API
- c) Contrôler la conformité RGPD
- d) Identifier le point de rupture du système

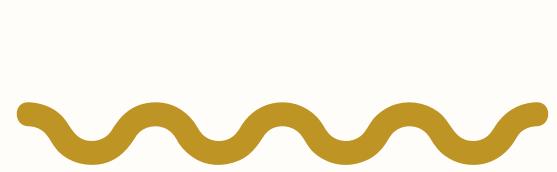


### 4. Le test de stress vise à :

- a) Vérifier la rapidité du login
- b) Tester uniquement les API
- c) Contrôler la conformité RGPD
- d) Identifier le point de rupture du système

➤ Correction : D

Le test de stress pousse le système **au-delà de ses limites** pour observer sa tolérance à la surcharge et son comportement en cas de panne.



### 5. Complétez :

**“Un produit peut être fonctionnel, mais inutilisable s'il n'est pas \_\_\_\_.”**

- a) Conçu pour l'utilisateur
- b) Automatisé
- c) Conforme aux normes ISO
- d) Testé en production



### 5. Complétez :

“Un produit peut être fonctionnel, mais inutilisable s'il n'est pas \_\_\_\_.”

a) Conçu pour l'utilisateur

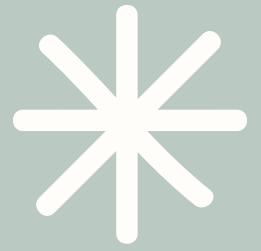
b) Automatisé

c) Conforme aux normes ISO

d) Testé en production

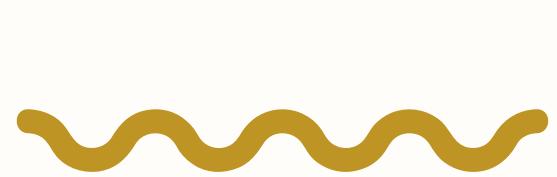
➤ Correction : A

Cette phrase clé du cours rappelle que la qualité en usage dépend de **la prise en compte des besoins utilisateurs réels**, pas seulement de l'absence de bugs.



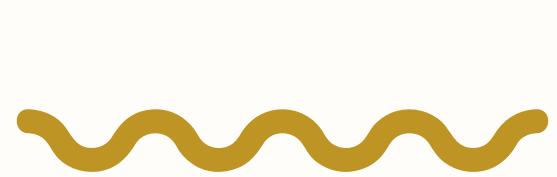
# **Exercices de rappel- Séance 1**





**Le paradoxe du pesticide s'applique quand :**

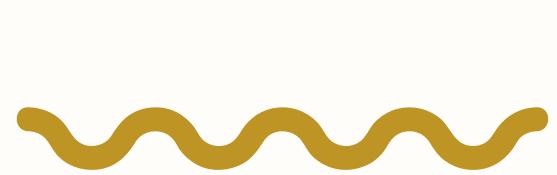
- a) Les tests sont trop nombreux
- b) Les mêmes tests ne trouvent plus de nouveaux défauts
- c) Les outils d'automatisation sont obsolètes
- d) Les testeurs manquent d'expérience



**Le paradoxe du pesticide s'applique quand :**

- a) Les tests sont trop nombreux
- b) Les mêmes tests ne trouvent plus de nouveaux défauts**
- c) Les outils d'automatisation sont obsolètes
- d) Les testeurs manquent d'expérience

Explication : Répéter les mêmes tests rend le système "immunisé" contre ces tests.



**La validation diffère de la vérification car elle répond à la question :**

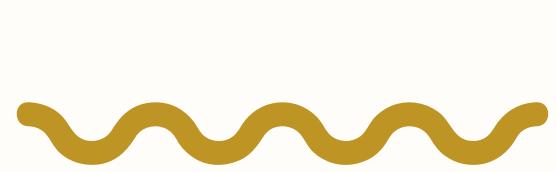
- a) "Construit-on le produit bien ?"
- b) "Construit-on le bon produit ?"
- c) "Le code est-il optimisé ?"
- d) "Les tests sont-ils automatisés ?"



**La validation diffère de la vérification car elle répond à la question :**

- a) "Construit-on le produit bien ?"
- b) "Construit-on le bon produit ?"**
- c) "Le code est-il optimisé ?"
- d) "Les tests sont-ils automatisés ?"

Explication : Validation = bon produit pour l'utilisateur, vérification = produit conforme aux specs.



**Dans la pyramide des tests, les tests unitaires sont caractérisés par :**

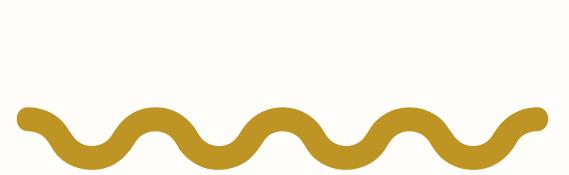
- a) Un coût élevé et un feedback lent
- b) Une couverture métier étendue
- c) Une exécution rapide et un débogage simple
- d) Une validation par le client



**Dans la pyramide des tests, les tests unitaires sont caractérisés par :**

- a) Un coût élevé et un feedback lent
- b) Une couverture métier étendue
- c) **Une exécution rapide et un débogage simple**
- d) Une validation par le client

Explication : Ils forment la base de la pyramide : nombreux, rapides, faciles à maintenir.



## Le coût de correction d'un défaut :

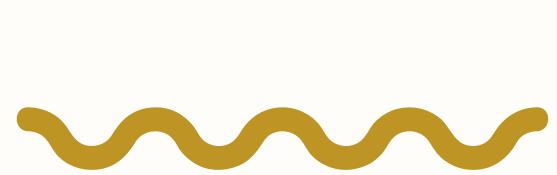
- a) Est constant tout au long du cycle de vie
- b) Diminue plus il est détecté tard
- c) Multiplié par 10 entre la conception et la production
- d) N'influe pas sur la rentabilité du projet



## Le coût de correction d'un défaut :

- a) Est constant tout au long du cycle de vie
- b) Diminue plus il est détecté tard
- c) Multiplié par 10 entre la conception et la production
- d) N'influe pas sur la rentabilité du projet

**Explication : Plus un défaut est détecté tard, plus son coût de correction est exponentiel.**



## **Le Risk-Based Testing priorise les tests en fonction :**

- a) Du nombre de lignes de code
- b) De la complexité technique et de l'impact métier
- c) Des préférences de l'équipe de développement
- d) De la date de livraison



## Le Risk-Based Testing priorise les tests en fonction :

- a) Du nombre de lignes de code
- b) De la complexité technique et de l'impact métier**
- c) Des préférences de l'équipe de développement
- d) De la date de livraison

Explication : On concentre les efforts sur les zones à plus haut risque métier et technique.



**Le principe "Les tests montrent la présence de défauts" signifie que :**

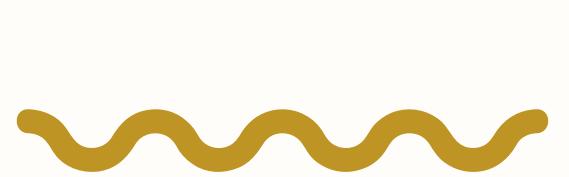
- a) On peut prouver qu'un logiciel est sans défaut
- b) On ne peut jamais garantir l'absence totale de défauts
- c) Les tests sont inutiles si aucun défaut n'est trouvé
- d) Seuls les tests manuels sont efficaces



**Le principe "Les tests montrent la présence de défauts" signifie que :**

- a) On peut prouver qu'un logiciel est sans défaut
- b) On ne peut jamais garantir l'absence totale de défauts**
- c) Les tests sont inutiles si aucun défaut n'est trouvé
- d) Seuls les tests manuels sont efficaces

Explication : Les tests peuvent révéler des défauts mais ne prouvent pas leur absence.



## **La traçabilité des exigences permet notamment :**

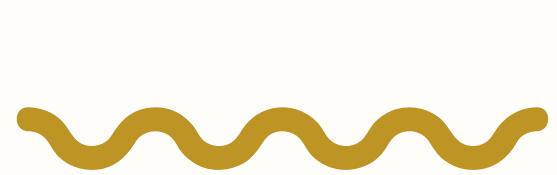
- a) De calculer la vitesse d'exécution des tests
- b) D'identifier l'impact d'un changement sur les cas de test
- c) De calculer le taux de couverture des tests
- d) De choisir les outils d'automatisation



## La traçabilité des exigences permet notamment :

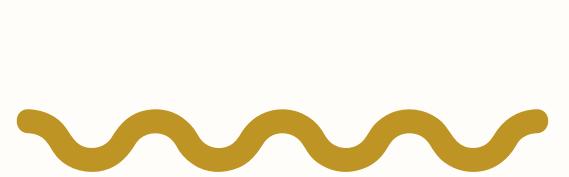
- a) De calculer la vitesse d'exécution des tests
- b) D'identifier l'impact d'un changement sur les cas de test**
- c) De calculer le taux de couverture des tests
- d) De choisir les outils d'automatisation

Explication : Elle permet de voir quels tests sont impactés par un changement d'exigence.



**Le test de confirmation est spécifiquement utilisé pour :**

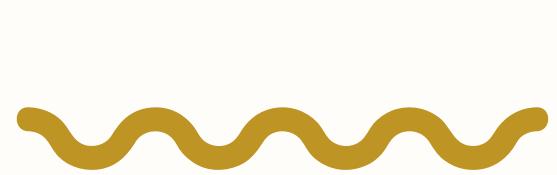
- a) Vérifier qu'un correctif résout le défaut rapporté
- b) Tester de nouvelles fonctionnalités
- c) Valider les performances du système
- d) Vérifier la sécurité après un déploiement



**Le test de confirmation est spécifiquement utilisé pour :**

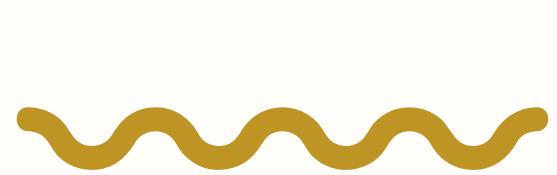
- a) Vérifier qu'un correctif résout le défaut rapporté
- b) Tester de nouvelles fonctionnalités
- c) Valider les performances du système
- d) Vérifier la sécurité après un déploiement

Explication : Il valide qu'un défaut spécifique a bien été corrigé.



**Un test de smoke est généralement :**

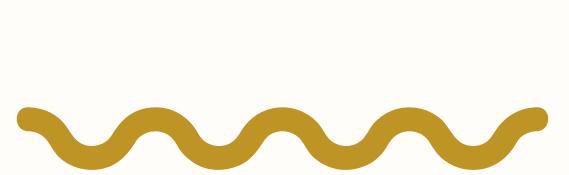
- a) Exhaustif et long à exécuter
- b) Rapide et limité aux fonctionnalités vitales
- c) Réservé aux tests de performance
- d) Effectué uniquement en production



**Un test de smoke est généralement :**

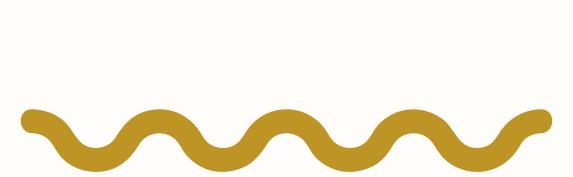
- a) Exhaustif et long à exécuter
- b) Rapide et limité aux fonctionnalités vitales**
- c) Réservé aux tests de performance
- d) Effectué uniquement en production

Explication : Vérification rapide des fonctions critiques après un déploiement.



**Dans un cas de test bien structuré, les "préconditions" servent à :**

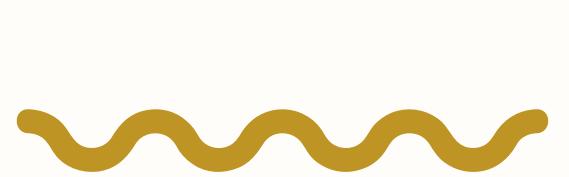
- a) Décrire le résultat attendu
- b) Spécifier l'état du système avant exécution
- c) Lister les outils utilisés
- d) Indiquer l'identifiant unique du test



**Dans un cas de test bien structuré, les "préconditions" servent à :**

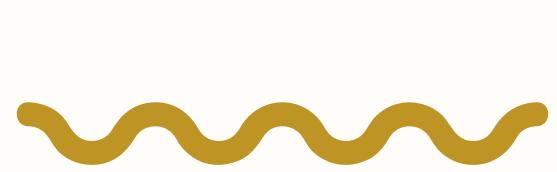
- a) Décrire le résultat attendu
- b) Spécifier l'état du système avant exécution**
- c) Lister les outils utilisés
- d) Indiquer l'identifiant unique du test

Explication : Pousse le système au-delà de ses limites pour trouver son point de rupture.



**Un test d'utilisabilité évalue principalement :**

- a) La vitesse de traitement des données
- b) La satisfaction de l'utilisateur
- c) La sécurité des transactions
- d) La couverture du code



**Un test d'utilisabilité évalue principalement :**

- a) La vitesse de traitement des données
- b) La satisfaction de l'utilisateur**
- c) La sécurité des transactions
- d) La couverture du code

Explication : Mesure l'expérience utilisateur et la facilité d'utilisation.



**Le test de performance qui mesure le temps de réponse sous charge normale est un :**

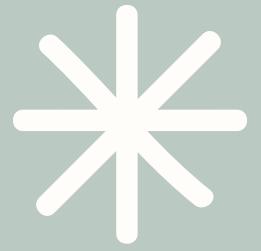
- a) Test de stress
- b) Test de charge
- c) Test d'endurance
- d) Test de smoke



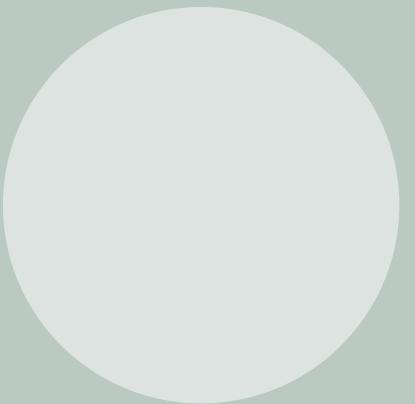
**Le test de performance qui mesure le temps de réponse sous charge normale est un :**

- a) Test de stress
- b) Test de charge**
- c) Test d'endurance
- d) Test de smoke

Explication : Test de charge = comportement sous volume d'utilisateurs attendu.



# **Chapitre 4 : Intégration du test dans les Méthodologies Modernes**



## Chapitre 4 : Intégration du test dans les Méthodologies Modernes

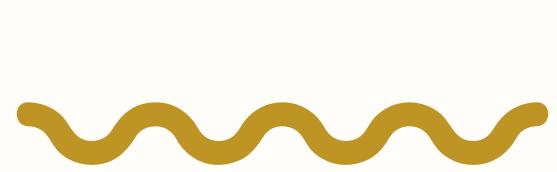


- **4.1 Les Tests dans les Méthodologies Agiles**
  - 4.1.1 De la méthode classique à l'Agile
  - 4.1.2 Les pratiques orientées test : BDD & TDD
  - 4.1.3 Rôle du testeur dans une équipe Scrum
- **4.2 Tests et DevOps/CI-CD**
  - 4.2.1 C'est quoi DevOps et CI/CD ?
  - 4.2.2 Les "Portes de Qualité" (Quality Gates)
  - 4.2.3 Shift-Left : Tester plus tôt
- **4.3 Stratégie d'Automatisation des Tests**
  - 4.3.1 Pourquoi automatiser ?
  - 4.3.2 Quoi automatiser en priorité ?
  - 4.3.3 Les outils d'automatisation en 2024



## 4.i Les Tests dans les Méthodologies Agiles

#### 4.1.1 De la méthode classique à l'Agile : le rôle du testeur se transforme



##### Tester tôt, souvent et en équipe

Méthodologie Classique (Cycle en V)	Méthodologie Agile (Scrum / Kanban)
Tests en <b>fin de cycle</b> , après développement	Tests <b>en continu</b> , à chaque sprint
Testeur = <b>rôle séparé</b> , validation "après coup"	Testeur <b>intégré à l'équipe</b> , collab. dev & PO
Documentation <b>lourde</b> et <b>statique</b>	Documentation <b>légère</b> et <b>vivante</b> (BDD, Jira, Xray)
Mesure de couverture <b>tardive</b>	Feedback <b>rapide</b> et visible (CI/CD, dashboards)
Corrections <b>retardées</b>	Corrections <b>immédiates</b> grâce à la proximité équipe

“En Agile, le test n'est plus une étape finale, mais un réflexe collectif.”

#### 4.1.1 De la méthode classique à l'Agile : le rôle du testeur se transforme

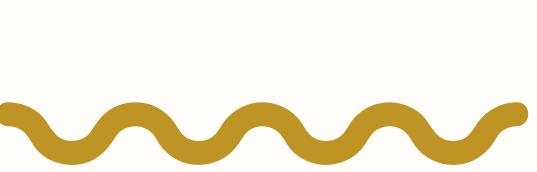


##### Les 4 piliers du testing Agile :

- Feedback continu
  - Détection précoce des défauts grâce aux tests automatisés et aux revues fréquentes.
- Collaboration étroite
  - Travail en synergie entre Testeurs, Développeurs et Product Owner.
- Adaptabilité
  - Capacité à ajuster les tests à chaque évolution du produit.
- Qualité collective
  - “La qualité est l'affaire de tous” — chaque membre contribue à la validation.

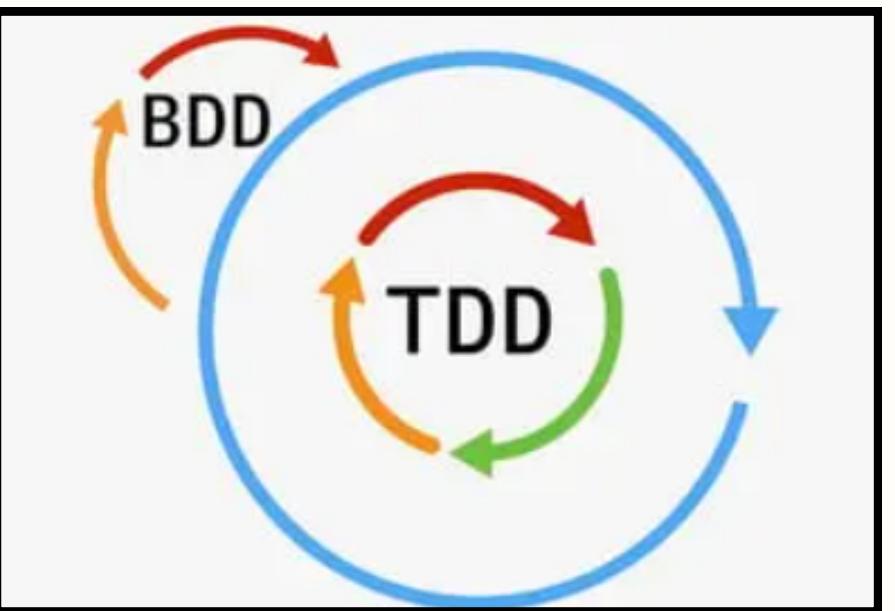
“Tester tôt, tester souvent, tester ensemble.”

## 4.1.2 Les pratiques orientées test : BDD & TDD



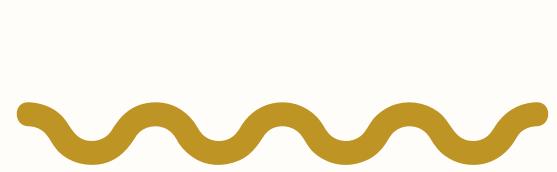
### Objectif :

- Renforcer la qualité dès la phase de développement, en intégrant le test au cœur du code et des exigences.



“En Agile, on ne teste pas **après**... on teste **pendant**.”

## 4.1.2 Les pratiques orientées test : BDD & ATDD



### BDD - Behavior Driven Development

🗣 **Le langage qui unit technique et métier** : Le BDD vise à créer un langage commun entre les équipes techniques et métiers à travers des scénarios exécutables.

Format Gherkin - Compréhensible par tous :

gherkin

Fonctionnalité: Paiement en ligne sécurisé

Scénario: Paiement par carte réussi

Étant donné que j'ai des articles dans mon panier

Quand je saisis un numéro de carte valide "4111-1111-1111-1111"

Et que je clique sur "Payer"

Alors la commande devrait être confirmée

Et je devrais recevoir un email de confirmation

Les 3 piliers du BDD :

- 1. Collaboration:** Ateliers communs
- 2. Langage commun :** Format Given-When-Then
- 3. Documentation vivante :** Scénarios exécutables

Outils : Cucumber, SpecFlow, Behave

## 4.1.2 Les pratiques orientées test : BDD & ATDD



### TDD - Test Driven Development

● RED → ● GREEN → ● REFACTOR

#### Le cycle en 3 étapes :

##### Étape 1 : ● RED - Le test échoue,

On écrit un test avant le code.

Il échoue logiquement, car la fonctionnalité n'existe pas encore.

##### Étape 2 : ● GREEN - Code minimal

On écrit juste le code nécessaire pour que le test passe.

faire passer le test, rien de plus.

##### Étape 3 : ● REFACTOR - Amélioration

On améliore la qualité du code (lisibilité, duplication, performance)

Sans casser les tests.

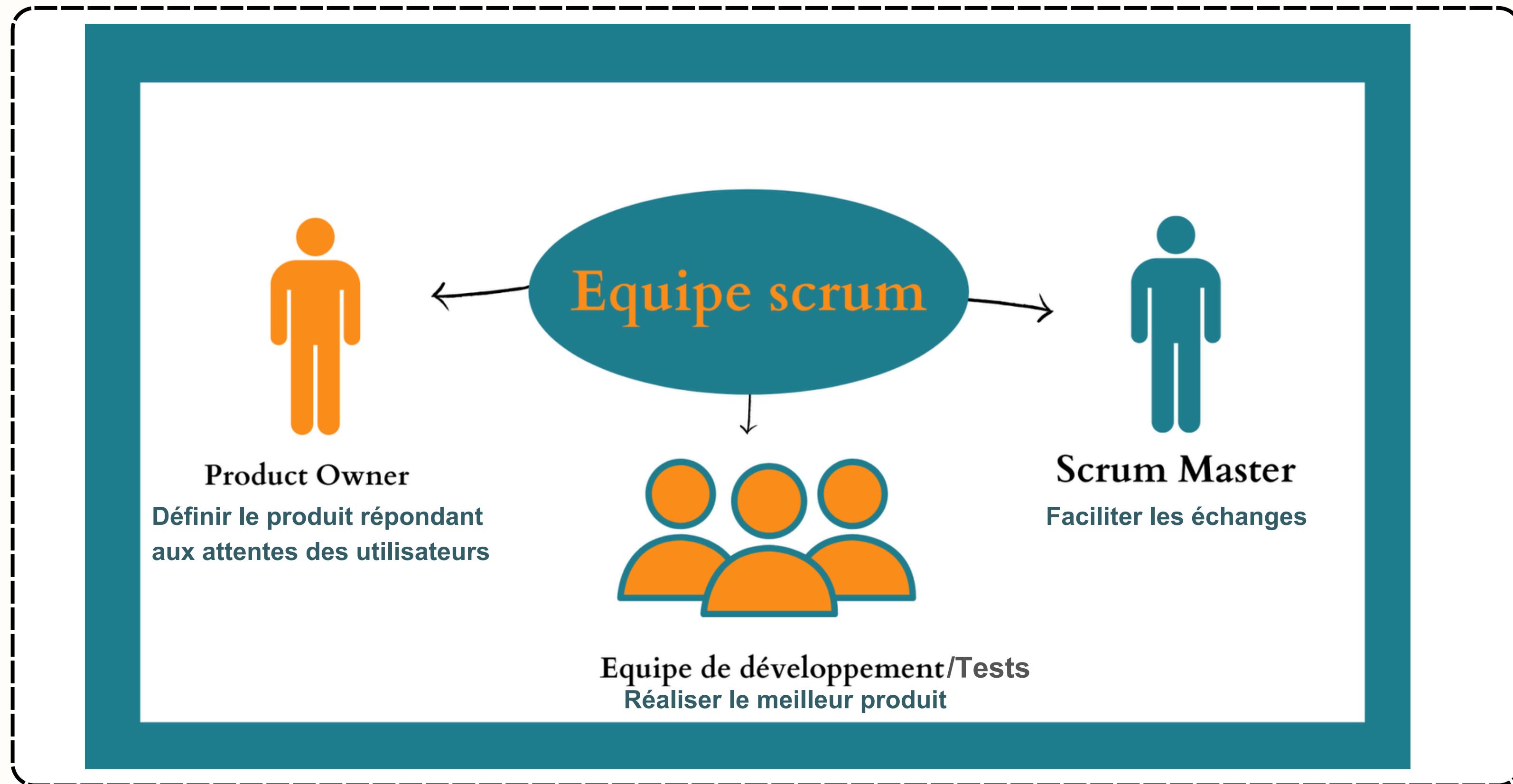
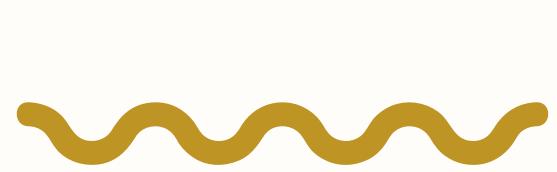
“écrire le test avant le code — on code pour faire passer le test.”

```
@Test  
public void testCalculerTVA() {  
    CalculateurTVA calc = new CalculateurTVA();  
    assertEquals(20.0, calc.calculerTVA(100.0, 20.0), 0.01);  
}  
// ✗ Échec - classe n'existe pas
```

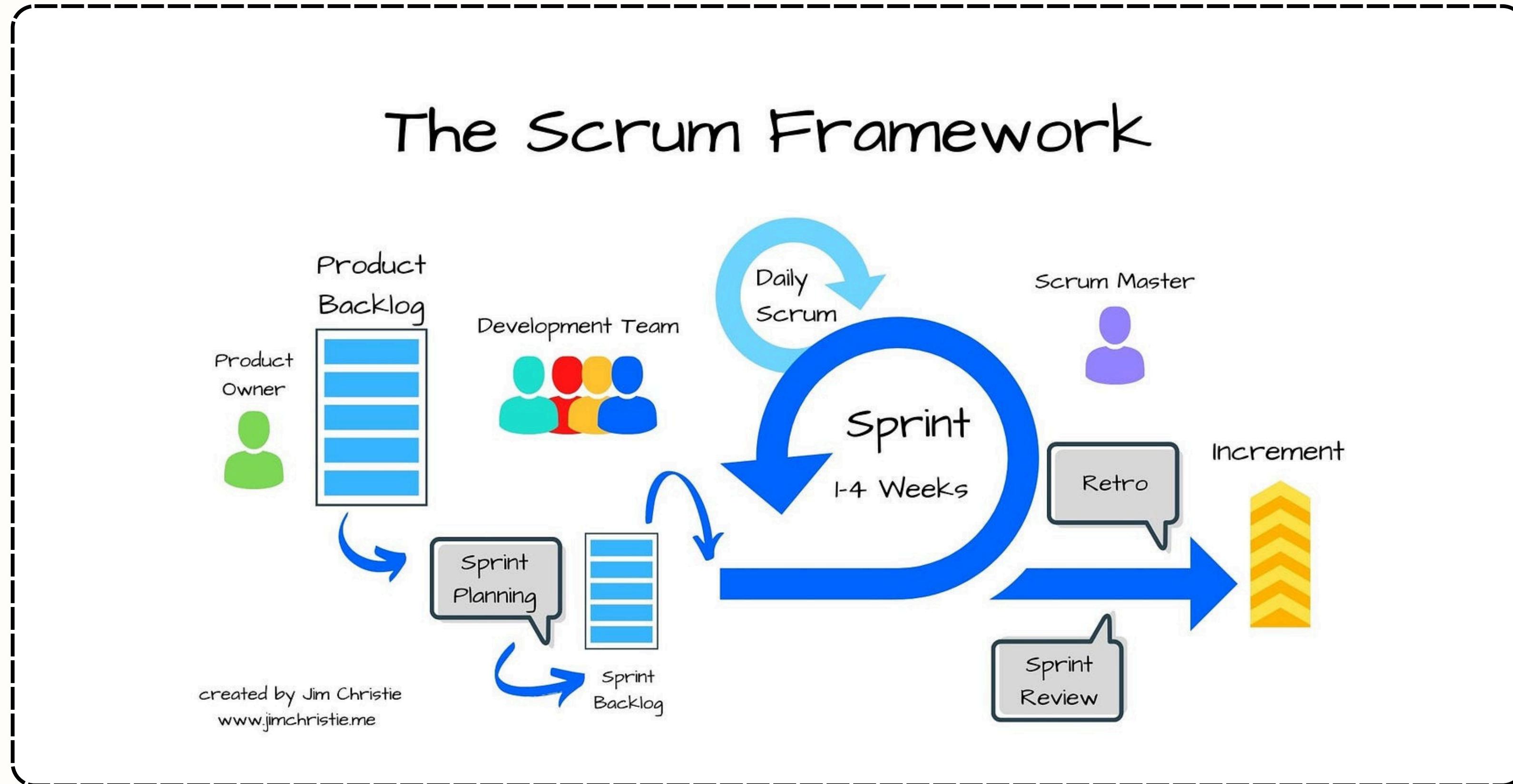
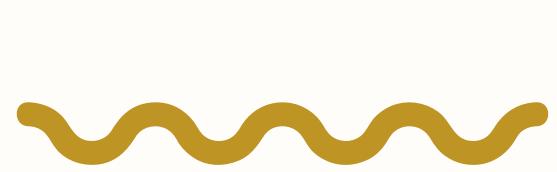
```
public class CalculateurTVA {  
    public double calculerTVA(double montant, double taux) {  
        return 20.0; // Solution minimale  
    }  
}  
// ✅ Succès - code basique
```

```
public class CalculateurTVA {  
    public double calculerTVA(double montant, double taux) {  
        return montant * (taux / 100); // Solution propre  
    }  
}  
// ✅ Succès - code optimisé
```

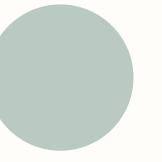
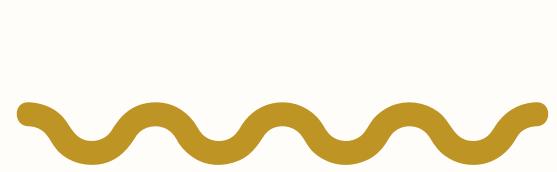
#### 4.1.3 Rôle du testeur dans une équipe Scrum : Rappel : Équipe Scrum



#### 4.1.3 Rôle du testeur dans une équipe Scrum: Rappel : Scrum Framework



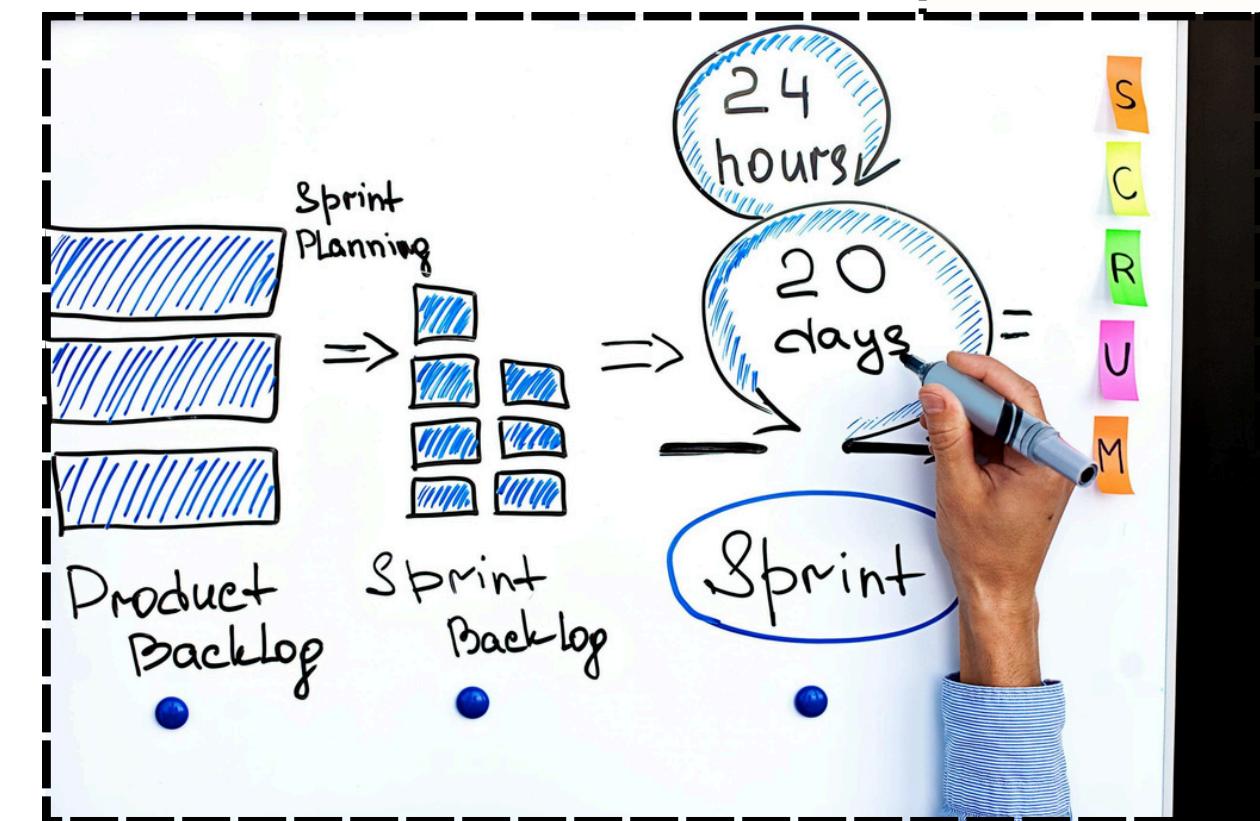
#### 4.1.3 Rôle du testeur dans une équipe Scrum



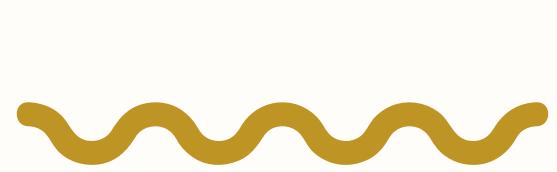
##### A. Sprint Planning :

###### Rôle du testeur:

- Estimer **l'effort de test** pour chaque user story
- S'assurer que les critères d'acceptation **sont testables**
- Identifier les besoins en données/environnements de test



#### 4.1.3 Rôle du testeur dans une équipe Scrum



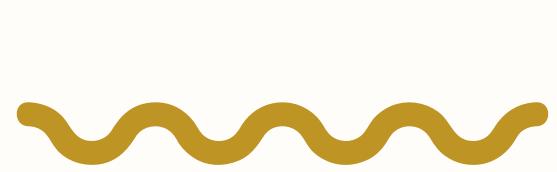
##### B. Daily Stand-up :

###### Ce que le testeur dit:

- "Hier j'ai testé les stories X et Y, j'ai trouvé 3 défauts"
- "Aujourd'hui je vais terminer Z et commencer les tests de régression"
- "J'ai un blocage: l'environnement de test est instable"



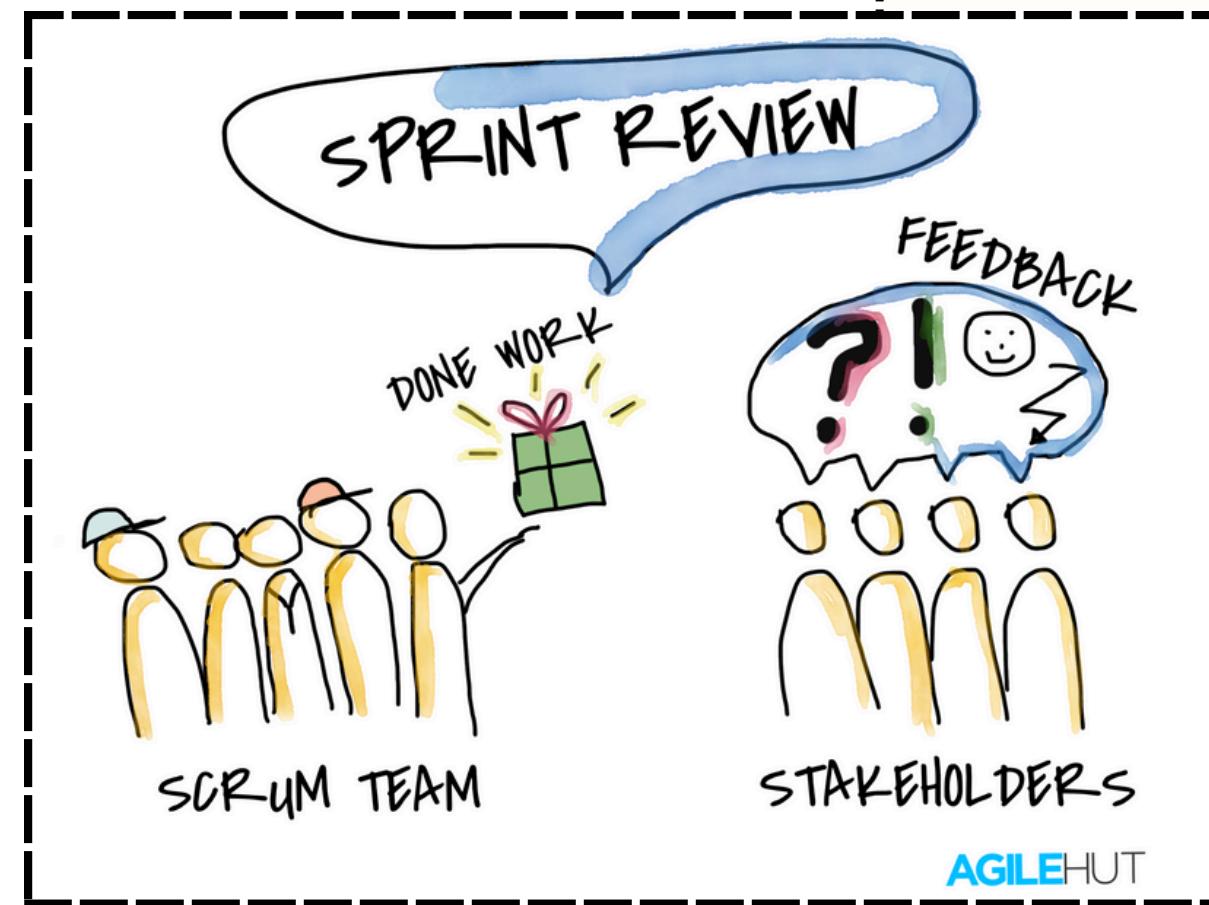
#### 4.1.3 Rôle du testeur dans une équipe Scrum



##### C. Sprint Review :

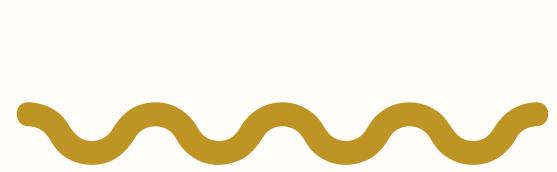
###### Contributions du testeur:

- Présenter le **rapport qualité** du sprint
- Démontrer les **fonctionnalités testées**
- Recueillir le **feedback** des **parties prenantes**



AGILEHUT

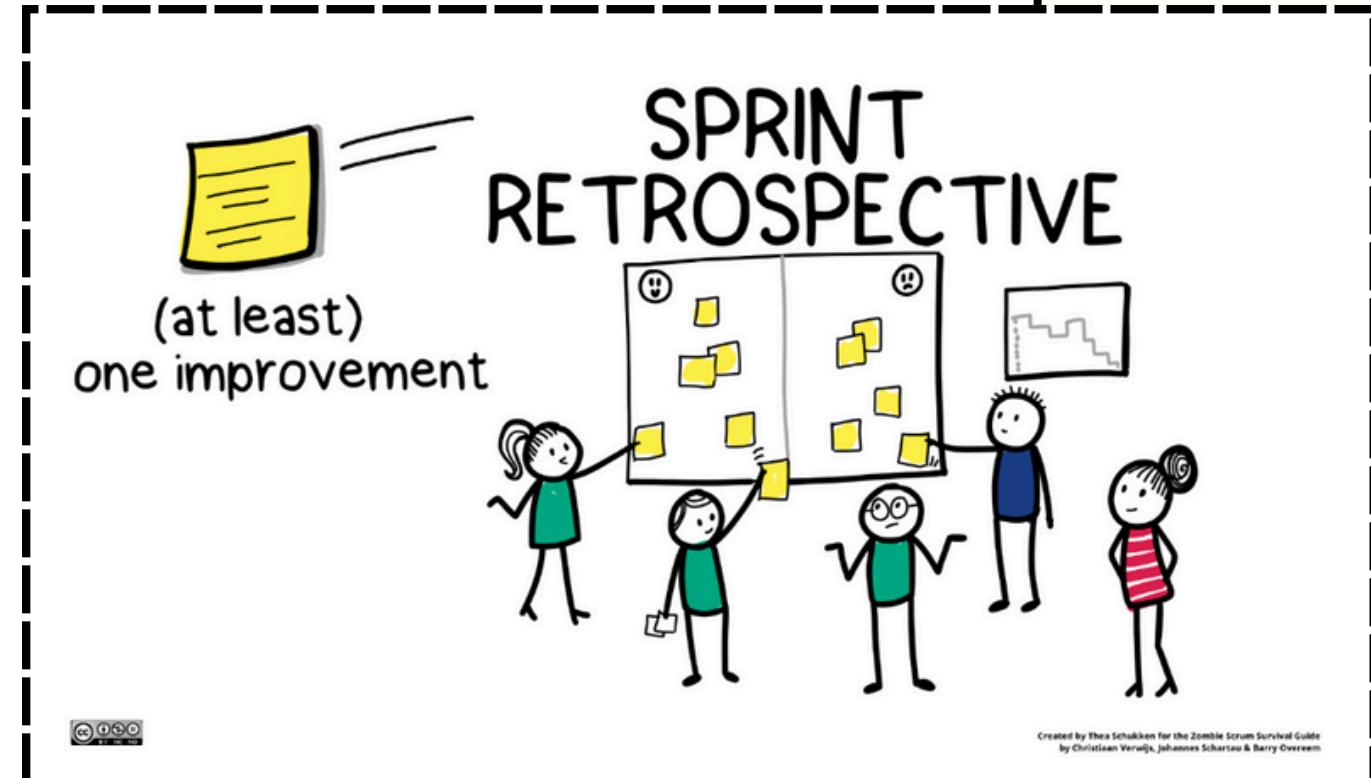
#### 4.1.3 Rôle du testeur dans une équipe Scrum



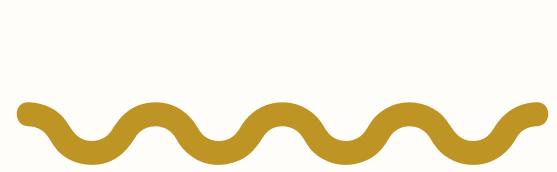
##### D. Sprint Retrospective :

Améliorations qualité:

- "Qu'est-ce qui a bien fonctionné pour la qualité ?"
- "Qu'est-ce qu'on peut améliorer dans nos processus de test ?"
- "Comment réduire les défauts échappés ?"



#### 4.1.3 Definition of Done (DoD) - pour une User Story



**DoD = tout ce qui doit être vrai pour dire “c'est fini”**

##### **Qualité du code**

- Code développé et revue effectuée
- Tests unitaires écrits et  $\geq 80\%$  de couverture
- Aucun défaut critique ou bloquant ouvert

##### **Validation technique**

- Tests d'intégration écrits et passants
- Tests manuels exécutés et validés
- Tests automatisés ajoutés à la suite de régression

##### **Livraison et documentation**

- Code déployé sur environnement d'intégration
- Documentation technique mise à jour
- Performance dans les cibles définies
- Story acceptée par le Product Owner

**“Une User Story n'est Done que si elle répond à tous les critères du DoD.”**



## 4.2 Tests et DevOps/CI-CD

## 4.2.1 C'est quoi DevOps et CI/CD ?



DevOps = Développement + Opérations

- **Avant** : Les devs et les ops travaillaient séparément
- **Maintenant** : Ils collaborent pour livrer plus vite et plus fiable

Objectif :

- Réduire le temps entre l'idée et la mise en production
- Automatiser le maximum d'étapes manuelles

## CI/CD

	Signification	Rôle
CI	Intégration Continue	Chaque commit est testé automatiquement
CD	Livraison Continue	Le code testé peut être déployé en prod à tout moment

### ■ Exemple :

Quand un développeur pousse un commit :

CI → les tests automatisés tournent

CD → si tout est vert, la nouvelle version peut être déployée

Commit → Build → Test → Deploy

↑              ↑              ↑

Intégration   Qualité   Livraison

Continue      Gate      Continue

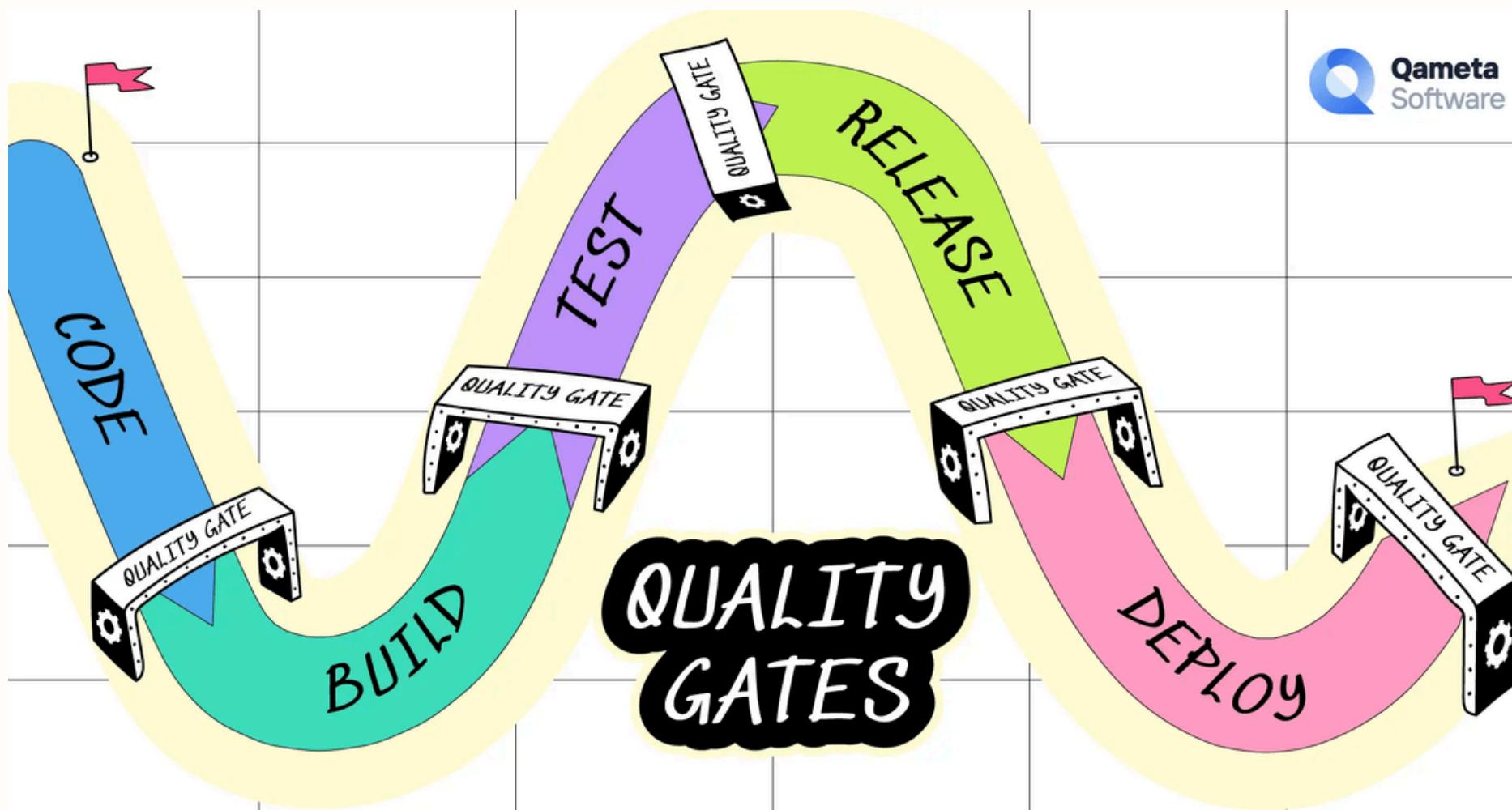
## 4.2.2 Les "Portes de Qualité" (Quality Gates) – contrôler la qualité à chaque étape du pipeline CI/CD



“Comme dans une usine, chaque étape de fabrication du logiciel passe un **contrôle qualité**

Chaque étape du pipeline CI/CD devient une **porte de validation**.

Si une porte **échoue** → le pipeline est **bloqué** jusqu'à correction.



- ◆ **Porte Qualité 1** → Vérifie le code et la syntaxe
- ◆ **Porte Qualité 2** → Vérifie les tests unitaires
- ◆ **Porte Qualité 3** → Vérifie l'intégration et les dépendances
- ◆ **Porte Qualité 4** → Vérifie les scénarios métiers (E2E) et la stabilité avant déploiement



Les Quality Gates assurent que la qualité est vérifiée en continu, et non en fin de projet.

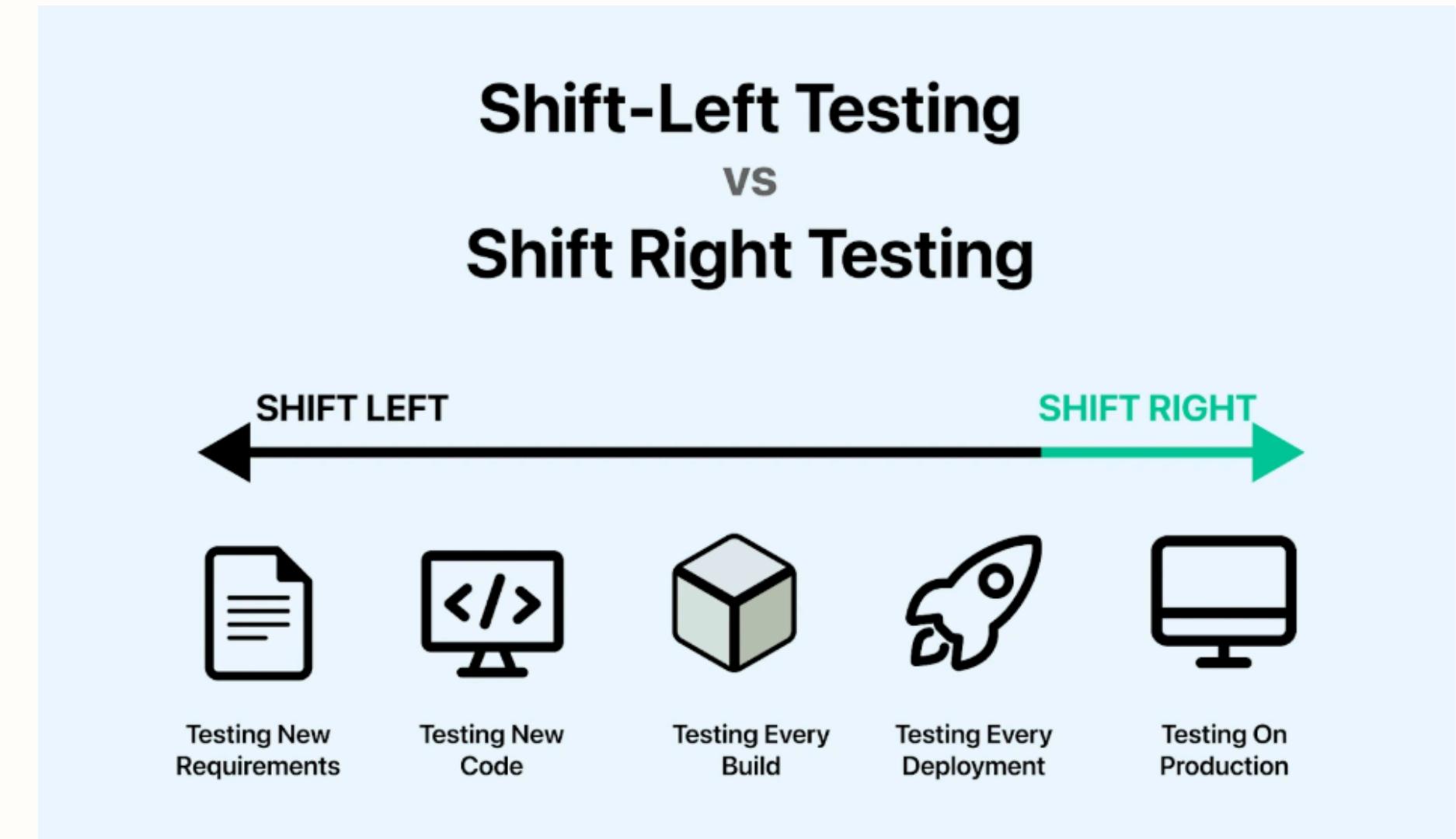
## 4.2.3 Shift-Left : Tester plus tôt pour prévenir les défauts



**Idée :** Attraper les bugs le plus tôt possible  
("prévention" plutôt que "détexion")

### ACTIVITÉS SHIFT-LEFT:

- Vérification des specs **avant dev**
- Revue de code / **analyse statique**
- Tests uanitaires et d'intégration précoce
- Tests de sécurité dès la conception



### BÉNÉFICES:

- Détection des anomalies jusqu'à 10× plus rapide
- Réduction des coûts de correction
- Collaboration renforcée entre QA et Dev



## 4.3 Stratégie d'Automatisation des Tests

### 4.3.1 Pourquoi automatiser ? Mesurer avant d'agir



Dans les projets Agiles, les tests sont rejoués à chaque sprint.

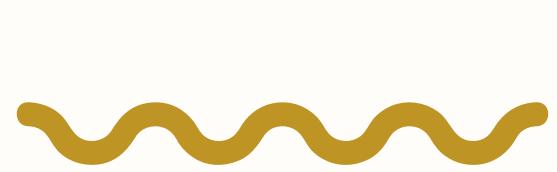
| Automatiser permet de gagner en efficacité, en qualité et en rapidité.

#### Objectifs de l'automatisation :

Objectif	Impact direct
<b>Réduire les efforts manuels</b>	Moins de répétition, plus de valeur ajoutée
<b>Accélérer les cycles de livraison</b>	Intégration continue fluide (CI/CD)
<b>Améliorer la détection des défauts</b>	Feedback plus rapide pour les devs
<b>Renforcer la fiabilité</b>	Tests rejouables, traçables, stables
<b>Gagner du temps</b>	Tests de régression exécutés en quelques minutes

Automatiser sans mesurer, c'est investir sans garantie de retour.

### 4.3.1 Pourquoi automatiser ? Mesurer avant d'agir



“L’automatisation n’est rentable que si les gains dépassent les coûts

Calcul du Return on Investment :

$$\text{ROI} = (\text{Gains} - \text{Coûts}) / \text{Coûts}$$

#### GAINS:

- Temps d'exécution économisé
- Réduction défauts en production
- Livraison plus rapide

#### COÛTS:

- Développement et maintenance des scripts
- Outils et infrastructures
- Formation des équipes

#### Exemple concret avec un sprint de 2 semaines :

Scénario 1 suite de tests manuels de 8h exécutée 2 fois par sprint

Avant automatisation : 20 000 €/an

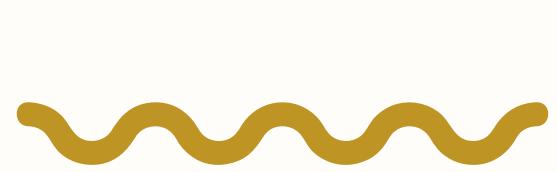
Après automatisation :

- Dev initial : 4 000 €
- Maintenance : 7 600 €/an
- ROI = 163 % la 1<sup>re</sup> année

⚠ L’automatisation n’est pas toujours rentable.

Elle doit être ciblée sur les scénarios à fort impact (fréquence, stabilité, criticité).

### 4.3.2 Quoi automatiser en priorité ?



#### Grille de décision simple :

Critère	À automatiser	À réfléchir	Pas prioritaire
Fréquence	Tous les jours	1 fois/semaine	1 fois/mois
Importance	Fonction vitale	Utile	Accessoire
Stabilité	Ne change pas	Évolue parfois	En refonte
Complexité	Long et difficile	Moyen	Rapide et simple

#### Exemples concrets :

##### À automatiser ABSOLUMENT :

**Login** : Testé à chaque build

**Paiement** : Critique pour l'entreprise

**Calculs financiers** : Complexes et risqué

##### À automatiser PLUS TARD :

**Couleurs UI** : Peu critique

**Fonctions en cours de modification**

**Rapports peu utilisés**



**l'automatisation est une décision stratégique, pas juste technique**

#### 4.3.4 Les outils d'automatisation en 2024



##### Tests API :

- Postman : Débutant, interface facile
- REST Assured : Expert Java, puissant



POSTMAN

##### Tests Web :

- Selenium : Standard, tous langages
- Playwright : Moderne, performant
- Cypress : Rapide, bon pour frontend



Selenium

##### Tests Mobile :

- Appium : Standard, multi-plateforme
- Espresso : Android natif, rapide



Appium

##### Tests Performance :

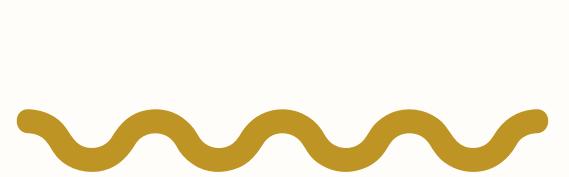
- JMeter : Classique, complet
- k6 : Moderne, intégration DevOps





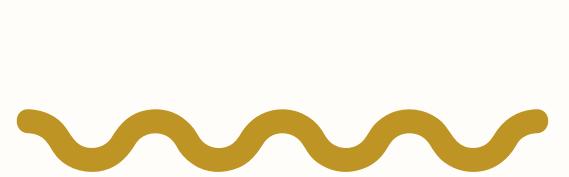
### Points clés à retenir :

- **Testeur Agile = Role Collaboratif** : Intégré à l'équipe, il contribue de la spécification (DoD, BDD) à la démonstration.
- **Shift-Left** : Tester le plus tôt possible pour détecter les défauts lorsque leur correction est moins coûteuse.
- **DevOps & CI/CD** : L'automatisation des tests est le moteur de la livraison continue. Les "Quality Gates" garantissent la qualité à chaque étape.
- **Automatisation Stratégique** : Automatiser en priorité ce qui est stable, critique et fréquemment exécuté pour un ROI maximal.
- **TDD & BDD** : Des pratiques qui intègrent la qualité directement dans le processus de développement et de spécification.



### 1. Dans une approche Agile, le test est :

- a) Réalisé uniquement à la fin du projet
- b) Intégré en continu tout au long du développement
- c) Effectué par une équipe séparée des développeurs
- d) Remplacé par des revues de code

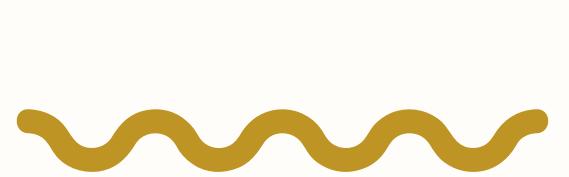


### 1. Dans une approche Agile, le test est :

- a) Réalisé uniquement à la fin du projet
- b) Intégré en continu tout au long du développement**
- c) Effectué par une équipe séparée des développeurs
- d) Remplacé par des revues de code

➤ Correction : B

En Agile, le test est **intégré à chaque sprint**. Il devient une activité **collective et continue**, pas une étape finale.



### 2. Le TDD (Test Driven Development) consiste à

- a) Écrire le test avant le code
- b) Tester après avoir écrit le code
- c) Tester uniquement les fonctionnalités critiques
- d) Automatiser les tests en fin de sprint



### 2. Le TDD (Test Driven Development) consiste à

- a) Écrire le test avant le code
- b) Tester après avoir écrit le code
- c) Tester uniquement les fonctionnalités critiques
- d) Automatiser les tests en fin de sprint

➤ Correction : A

Le cycle **RED** → **GREEN** → **REFACTOR** du TDD commence toujours par écrire un test qui échoue, puis le code minimal pour le faire passer.



### 3. Le BDD (Behavior Driven Development) vise à :

- a) Automatiser les tests unitaires
- b) Remplacer la documentation technique
- c) Tester uniquement les interfaces
- d) Décrire les comportements métiers en langage commun

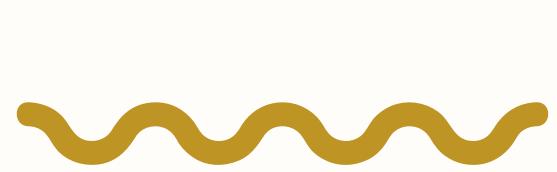


### 3. Le BDD (Behavior Driven Development) vise à :

- a) Automatiser les tests unitaires
- b) Remplacer la documentation technique
- c) Tester uniquement les interfaces
- d) Décrire les comportements métiers en langage commun**

➤ **Correction : D**

Le BDD utilise un langage partagé entre métiers et techniques (format Gherkin : Given / When / Then) pour aligner la compréhension du besoin.



### 4. Dans une chaîne CI/CD, le but du Continuous Integration (CI) est :

- a) Déployer automatiquement le produit en production
- b) Créer des rapports de performance
- c) Vérifier chaque commit avec des tests automatisés
- d) Tester uniquement les interfaces utilisateur



### 4. Dans une chaîne CI/CD, le but du Continuous Integration (CI) est :

- a) Déployer automatiquement le produit en production
- b) Créer des rapports de performance
- c) Vérifier chaque commit avec des tests automatisés
- d) Tester uniquement les interfaces utilisateur

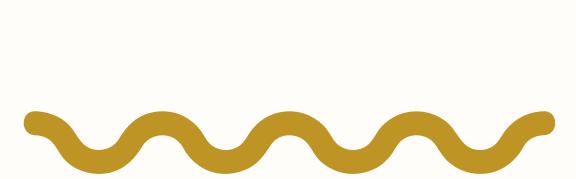
➤ Correction : C

Le CI exécute automatiquement **les tests unitaires et d'intégration** à chaque commit pour détecter les défauts rapidement.



### Le principe Shift-Left signifie :

- a) Reporter les tests après la livraison
- b) Déplacer les tests plus tôt dans le cycle de développement
- c) Faire uniquement des tests exploratoires
- d) Exécuter les tests à gauche de l'écran

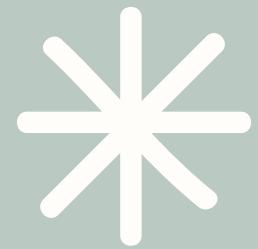


**Le principe Shift-Left signifie :**

- a) Reporter les tests après la livraison
- b) Déplacer les tests plus tôt dans le cycle de développement**
- c) Faire uniquement des tests exploratoires
- d) Exécuter les tests à gauche de l'écran

➤ **Correction : B**

“Tester plus tôt” permet de **prévenir les défauts plutôt que les détecter tard**, via revues, tests unitaires et sécurité dès la conception.



## Chapitre 5 : Etude de cas



## Cas 1 — Le paiement fantôme (E-Shopia)

### Situation

E-Shopia, un site de vente en ligne, ajoute un nouveau **paiement par carte VISA** avant les soldes.

Dès le lancement, plusieurs clients se plaignent :

“Ma commande est validée, mais je n'ai jamais été débité !”



### Ce qui s'est passé

- Le site affichait “Paiement réussi” même quand la transaction n'avait **pas abouti**.
- En réalité, le serveur de la banque (le **back-end**) renvoyait une erreur 500 (**panne du service**)
- Le site (le **front-end**) ne gérait pas cette erreur et montrait quand même le message de succès.

### Conséquences

- Plus de 200 commandes “**fantômes**” non payées qui étaient envoyées sans paiement effectif
- Pertes financières: bloquer les ventes jusqu'à la correction du bug

## Cas 1 — Le paiement fantôme (E-Shopia)- L'analyse et la solution

### Pourquoi le bug est passé entre les mailles

1. Le testeur a **vérifié l'affichage du message**, mais pas le **résultat réel de la transaction**.
2. Aucun **test d'intégration** entre le site et l'API de paiement n'a été réalisé.
3. Le développeur n'avait pas prévu de **gestion d'erreur** en cas d'échec du serveur bancaire.



### Comment y remédier

- **Créer des tests “bout-en-bout”**

Tester le parcours complet : clic “payer” → appel API → enregistrement dans la base → confirmation.

- **Prévoir des scénarios d'échec**

Exemple : carte refusée, API en panne, solde insuffisant.

- **Automatiser le test critique de paiement**

Ce test doit s'exécuter avant chaque mise en production pour détecter toute régression.

### Leçon à retenir

Ce n'est pas parce qu'un message “succès” s'affiche que le système a vraiment réussi.

Toujours vérifier la **réalité du résultat**, pas seulement ce que l'écran montre.

## Cas 2 — Le virement surprise (FinTrack)

### Situation

La plateforme bancaire **FinTrack** permet de faire des virements en ligne.

Une règle métier indique :

“Aucun virement **supérieur à 5000 €** ne doit être autorisé.”

Mais un client réussit à transférer **7000 €** via **PayPal** sans blocage.

Le système ne signale aucune erreur et affiche : “Virement effectué **avec succès.**”

### Ce qui s'est passé

- Le contrôle de limite (5000 €) avait bien été programmé... mais uniquement pour les paiements par carte bancaire.
- Le canal PayPal n'avait pas cette même règle.
- Résultat : un client a pu contourner la limite sans que le système ne réagisse.



### Conséquences

- Risque financier réel : virement non autorisé = perte de contrôle interne.
- Non-conformité réglementaire (la banque ne respecte pas ses propres règles).
- Perte de confiance des utilisateurs (“système pas sécurisé”).
- Correction urgente à faire sur toutes les API de paiement.

## Cas 2 — Le virement surprise (FinTrack)- L'analyse et la solution

### Pourquoi le bug est passé entre les mailles

1. Les tests ne couvraient que les **paiements par carte**.
2. Aucun test de **valeurs limites** n'a été fait (juste en dessous / au-dessus de 5000 €).
3. Pas de **vérification transversale** entre tous les canaux de paiement.
4. Les testeurs ont **supposé** que "la règle s'applique partout".



### Comment y remédier

#### Créer des jeux de données complets

- Tester les valeurs : 4999 €, 5000 €, 5001 € sur chaque méthode de paiement.

#### Ajouter un test automatique pour chaque canal

- CB, virement classique, PayPal, etc.

#### Uniformiser les règles métiers

- La validation de montant doit être gérée au même endroit dans le code (ex. au niveau du serveur, pas dans chaque module).

## Cas 3 — Les données qui disparaissent (App Gestion+)

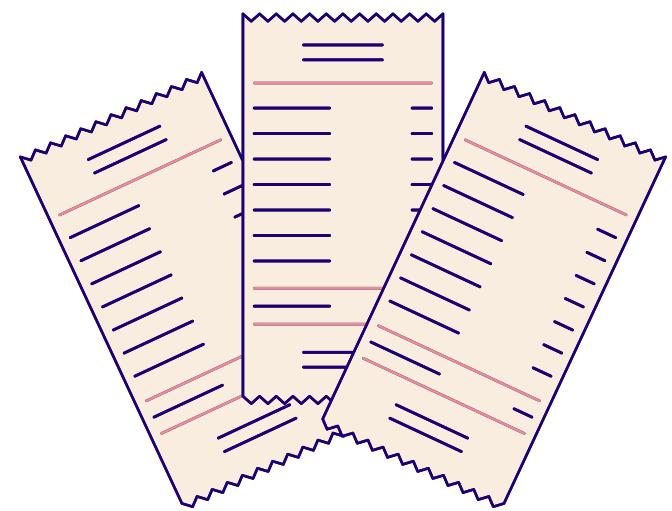
### Situation

L'application **Gestion+** permet à des entreprises de suivre leurs factures et devis.

Après une mise à jour, plusieurs clients signalent :

“Mes **anciennes** factures ont **disparu** !”

“J'ai perdu mes données après la **nouvelle version** !”



### Ce qui s'est passé

- Une nouvelle fonctionnalité “**Archivage automatique**” a été déployée.
- Lors du transfert de données vers la nouvelle base, **une erreur de script SQL** a supprimé certaines factures.
- **Aucun test** n'avait vérifié que les données **anciennes** étaient bien **migrées** avant la mise en production.

### Conséquences

- **Perte** partielle de données clients.
- Risque **juridique** (les factures sont des documents obligatoires).
- **Temps perdu** pour restaurer des sauvegardes manuelles.
- **Perte de confiance** et mauvaise image auprès des entreprises utilisatrices.

## Cas 3 — Le virement surprise (FinTrack)- L'analyse et la solution

### Pourquoi le bug est passé entre les mailles

1. Aucun **test de migration de données** avant le déploiement.
2. Les testeurs ont vérifié la nouvelle fonctionnalité (archivage) mais pas **l'intégrité des données existantes**.
3. Il n'y avait pas de **sauvegarde automatisée** avant la mise à jour.
4. Le script SQL n'avait pas été **revérifié par un autre développeur** (pas de revue de code).

### Comment y remédier

#### Créer un environnement de préproduction

- Tester la mise à jour sur une copie réelle de la base de production avant de déployer.

#### Mettre en place des tests de migration

- Vérifier que le nombre de données avant/après est identique, et que les valeurs sont correctes.

#### Sauvegarder avant chaque déploiement

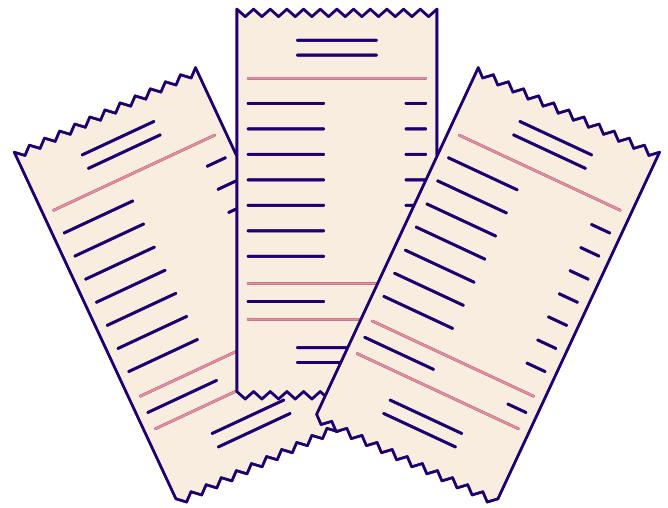
- Avoir une sauvegarde automatique quotidienne de la base de données pour restaurer en cas d'erreur.

#### Faire une revue de code SQL

- Chaque script de modification de données doit être validé par un second développeur.

### Leçon à retenir

Tester, ce n'est pas seulement valider de nouvelles fonctionnalités, c'est **aussi protéger ce qui existait déjà**



## Cas 4 — La fonctionnalité mal comprise (Projet BookTime)

### Situation

L'application **BookTime** permet de réserver des salles de réunion.

Le client a demandé :

“Je veux que la salle soit bloquée automatiquement dès qu'une réservation est confirmée.”

Après la mise en production, des utilisateurs se plaignent :

“Deux personnes ont réussi à **réserver la même salle à la même heure !**”



### Ce qui s'est passé

- Les développeurs ont compris “bloquer” comme : empêcher de nouvelles réservations après validation.
- Le client voulait dire : empêcher les réservations en double dès la saisie.
- L'équipe de test n'a pas vérifié ce cas, car la spécification était ambiguë.

### Conséquences

- **Réservations en conflit** → salles doublement occupées.
- **Mécontentement** des utilisateurs internes (“logiciel non fiable”).
- Temps perdu pour **corriger les plannings à la main**.

## Cas 4 — Le virement surprise (FinTrack)- L'analyse et la solution

### Pourquoi le bug est passé entre les mailles

1. L'exigence n'était pas claire dès le départ.
2. Aucun atelier de clarification n'a été fait entre client, devs et testeurs.
3. Le testeur a testé uniquement le "cas simple" : créer une réservation, pas les cas concurrents.
4. Pas de revue de spécifications avant développement.



### Comment y remédier

#### Clarifier les besoins dès le début

- Organiser une revue d'exigences : le client, le PO (Product Owner) et les testeurs valident ensemble ce que chaque phrase veut dire.

#### Documenter les décisions

- Chaque clarification doit être notée dans le cahier des charges ou l'outil de gestion (Jira, Confluence...).

### Leçon à retenir

La plupart des bugs viennent d'un **malentendu humain**, pas d'une erreur technique.

**Une exigence claire = moitié du travail de test déjà fait.**

## Cas 5 — Les tests sans fin (Projet FastBank)

### Situation

L'équipe **FastBank** livre une application bancaire mobile avec des **mises à jour toutes les 2 semaines** (sprints).

Avant chaque livraison, une suite de tests **manuels de 8 heures** est exécutée **deux fois** par sprint.

Les testeurs passent donc plus de la moitié de leur temps à répéter les mêmes scénarios.

### Ce qui s'est passé

- Les livraisons commencent à prendre du **retard**.
- Certains tests sont “**sautés**” pour gagner du temps.
- Plusieurs bugs réapparaissent en production (non-régression non couvertes’)

### Conséquences

- Temps de test **trop long** → baisse de la productivité.
- Coût annuel élevé : ~20 000 € en main-d'œuvre répétitive.
- Régressions fréquentes non détectées à temps.
- Démotivation des testeurs (“on refait toujours les mêmes tests”).

## Cas 5 — Les tests sans fin (Projet FastBank)

### Pourquoi le bug est passé entre les mailles

1. Aucun plan d'automatisation n'a été prévu dès **le départ du projet**.
2. Les tests ont été pensés uniquement en mode manuel, **sans priorisation**.
3. L'équipe **sous-estimait** la valeur économique des tests automatisés.

### Comment y remédier

#### Identifier les tests répétitifs à automatiser

- Scénarios candidats : connexion, création de compte, virement, paiement carte.

#### Investir dans une automatisation initiale

- Ex : 4 000 € de dev + 7 600 € de maintenance/an.
- Coût manuel remplacé : 20 000 €/an → ROI = 163 % la première année.

#### Mettre les tests dans la CI/CD

- Les exécuter automatiquement à chaque build pour éviter les régressions.

#### Mesurer les gains concrets

- Temps économisé : 16 h par sprint.
- Réduction des bugs en prod : -40 %.
- Livraisons plus rapides et fiables.

## Le mot de passe visible (App SecureMail)

### Situation

L'application **SecureMail** permet aux employés d'une entreprise de se connecter pour consulter leurs mails professionnels.

Après le déploiement d'une nouvelle version, plusieurs utilisateurs remarquent :

“Quand je me connecte, mon **mot de passe s'affiche en clair** sur l'écran pendant une seconde avant de disparaître !”

### Ce qui s'est passé

- Le champ de saisie du mot de passe n'était pas configuré en **mode “masqué” (type="password")**.
- Une erreur d'intégration dans le code front-end a supprimé cette propriété.
- Les testeurs ont bien vérifié la connexion... **mais pas l'affichage visuel du champ.**

### Conséquences

- Risque de vol de mot de passe (écran visible ou filmé).
- Perte de crédibilité pour une application censée être “sécurisée”.
- Signalements d'utilisateurs → correctif d'urgence déployé.
- Retard sur la feuille de route produit (priorité donnée à la sécurité).

## Le mot de passe visible (App SecureMail)

### Pourquoi le bug est passé entre les mailles

1. Aucun test d'affichage (UI) n'a été prévu pour la page de connexion.
2. Les testeurs se sont concentrés sur la fonctionnalité logique ("ça se connecte ?") et pas sur la sécurité visuelle.
3. Aucune revue de sécurité n'a été effectuée avant la mise en production.

### Comment y remédier

#### Inclure la sécurité dans les tests fonctionnels

- Ajouter un test visuel : "Le mot de passe doit rester masqué pendant la saisie."

#### Automatiser les vérifications UI

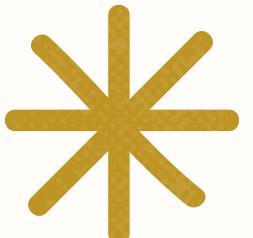
- Exécuter un test Selenium ou Playwright qui capture l'écran pendant la saisie et vérifie l'absence du mot de passe visible.

#### Faire des revues de code orientées sécurité

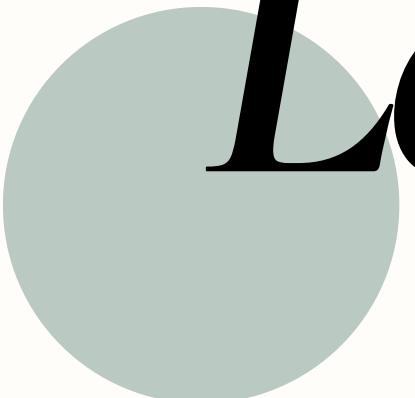
- Chaque merge doit être validé avec une checklist sécurité front-end.

#### Sensibiliser l'équipe QA et dev

- Expliquer que les erreurs de présentation peuvent aussi être des failles de sécurité.



# Qualité

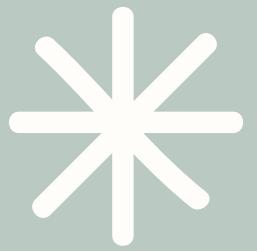


# *Logicielle*

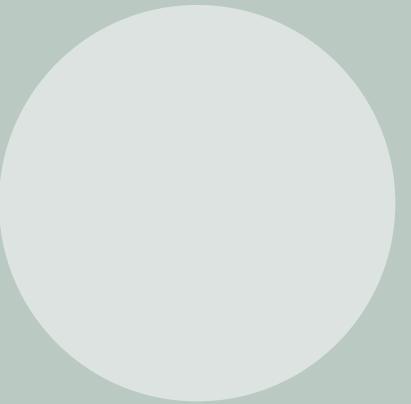
---

Youssef Touati ESIEA

---



# Exercices de révision



## Travail en groupe

1. Formez des groupes de **5 à 6 personnes maximum.**
2. Choisissez un **nom d'équipe** ( inspiré du thème de la qualité logicielle).
  - Désignez un responsable de groupe, chargé de :
  - Renseigner la liste complète des membres de l'équipe,
  - Et m'envoyer ce récapitulatif par mail à : [youssef.touati@ext.esiea.fr](mailto:youssef.touati@ext.esiea.fr)
  - ( Nom du groupe – Liste des participants)
3. Chaque groupe notera **ses réponses aux QCM** pendant l'activité.
4. **Un représentant par groupe** viendra ensuite au tableau pour présenter la réponse du groupe.
5. À la fin, on calcule la **somme des réponses valides**.

### Barème bonus / malus

- **Cas spécial 1er** : s'il n'y a qu'un seul groupe à finir 1er, il obtient **+2 points**.
- **Top 3 (sinon)** : les 3 premiers groupes obtiennent **+1 point** chacun.
- **Bottom 3** : les 3 derniers groupes auront **-1 point** chacun.
- **Cas spécial dernière position**: s'il n'y a qu'un seul groupe en dernière position, il obtient **-2 points**



## Format du mail

**Objet : Nom du groupe – Et m'envoyer ce récapitulatif par mail à : [youssef.touati@ext.esiea.fr](mailto:youssef.touati@ext.esiea.fr)**

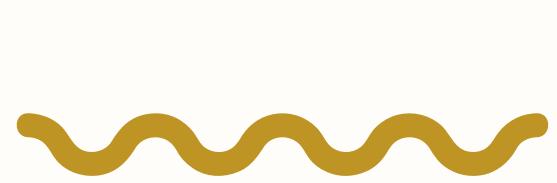
**Corps :**

Nom du groupe : < nom du groupe>

Responsable : <Nom Prénom>

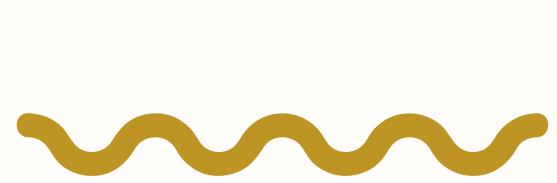
Membres :

- <Nom Prénom>



## 1- Le principe “Tester tôt est économique” implique principalement que :

- a) Les tests précoce garantissent l'exhaustivité.
- b) Plus un défaut est détecté tôt, plus son coût de correction est faible.
- c) Tester tôt réduit le nombre total de tests nécessaires.
- d) Tester tôt augmente la couverture des tests unitaires.



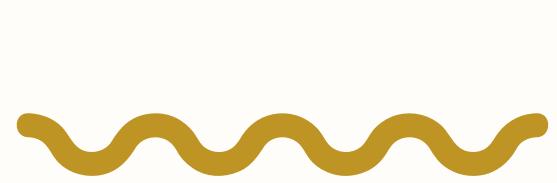
### 2. Quelle affirmation illustre le mieux le principe “Les tests sont contextuels” ?

- a) Les mêmes tests peuvent être réutilisés dans tous les projets.
- b) Les tests sont universels et indépendants du domaine.
- c) La stratégie de test dépend du type d'application, du risque et des contraintes.
- d) Tester un jeu mobile et une appli bancaire nécessite les mêmes critères de qualité.



**3. La différence essentielle entre vérification et validation est que :**

- a) La vérification est effectuée par les clients, la validation par les développeurs.
- b) La vérification évalue la conformité aux besoins utilisateurs.
- c) La validation mesure la conformité au cahier des charges.
- d) La vérification s'assure qu'on construit le produit bien, la validation qu'on construit le bon produit.



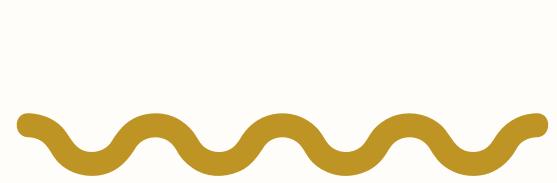
**4. Dans la pyramide des tests, un déséquilibre où les tests E2E dominent entraîne :**

- a) Un feedback plus rapide mais moins pertinent.
- b) Des coûts de maintenance faibles.
- c) Une stratégie inefficace et des délais accrus.
- d) Une réduction du risque de régression.



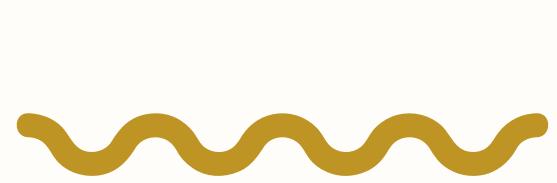
## 5. Quelle est la différence principale entre le test statique et le test dynamique ?

- a) Le test statique est effectué par le client
- b) Le test statique n'exécute pas le code, le test dynamique l'exécute
- c) Le test statique est moins important
- d) Le test dynamique ne détecte pas de défauts



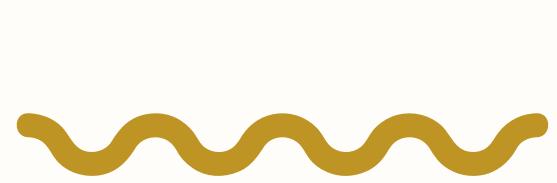
**6. Un cas de test est dit “Univoque” lorsqu'il :**

- a) Peut être réexécuté sans dépendre de l'environnement.
- b) Est rédigé sans ambiguïté ni interprétation multiple.
- c) Couvre plusieurs scénarios fonctionnels.
- d) Contient des données dynamiques pour s'adapter aux changements.



**7. Dans une matrice de traçabilité, l'absence de lien entre une exigence et un cas de test indique :**

- a) Une exigence non testée.
- b) Une exigence réussie.
- c) Une exigence rejetée.
- d) Une exigence facultative.



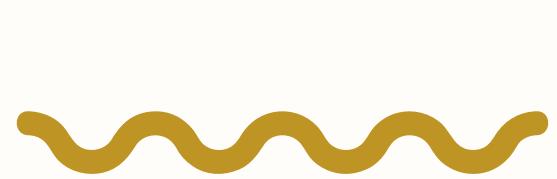
**8. L'erreur la plus courante lors de la priorisation des tests est :**

- a) Sous-estimer les impacts cachés.
- b) Confondre fréquence d'occurrence et impact métier.
- c) Donner trop de priorité aux tests automatisés.
- d) Classer tous les cas en priorité haute.



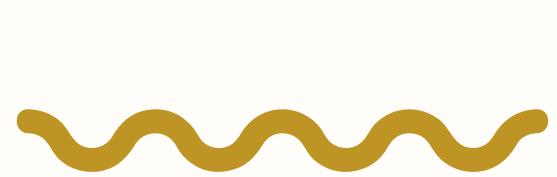
## 9. Le test de confirmation diffère du test de régression car :

- a) Il s'exécute avant toute modification de code.
- b) Il vise uniquement à vérifier la correction d'un défaut spécifique.
- c) Il couvre tout le système.
- d) Il est systématiquement automatisé.



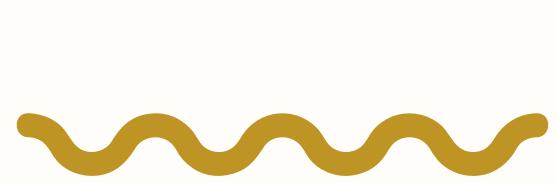
**10. Une suite de tests “Smoke” est principalement conçue pour :**

- a) Vérifier les fonctionnalités secondaires.
- b) Exécuter tous les cas critiques avant chaque build.
- c) Vérifier les fonctions vitales avant de poursuivre la campagne complète.
- d) Mesurer la performance après chaque livraison.



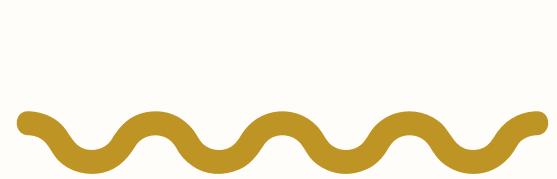
**11. Quelle affirmation distingue le mieux les tests fonctionnels des non-fonctionnels ?**

- a) Les tests fonctionnels évaluent la performance, les non-fonctionnels la logique métier.
- b) Les tests fonctionnels valident le “quoi”, les non-fonctionnels le “comment”.
- c) Les tests fonctionnels sont manuels, les non-fonctionnels automatisés.
- d) Les tests non-fonctionnels remplacent les tests d’intégration.



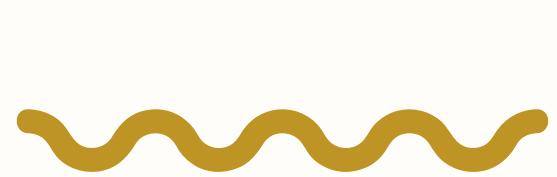
**12. Un test d'endurance vise à :**

- a) Identifier le point de rupture du système.
- b) Vérifier la stabilité sur une longue durée d'exécution.
- c) Pousser la charge jusqu'au crash du serveur.
- d) Simuler des attaques répétées.



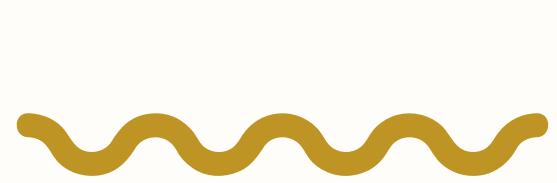
**13. Lors d'un test de charge, le principal indicateur à suivre est :**

- a) La fréquence CPU uniquement.
- b) Le nombre de sessions échouées.
- c) Le temps de réponse moyen sous charge nominale.
- d) Le nombre de requêtes par seconde uniquement.



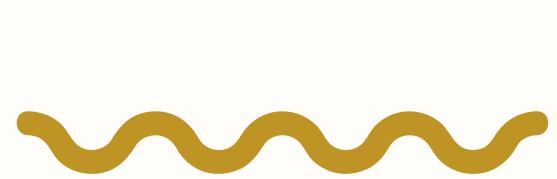
**14. Un produit logiciel présentant un code propre et maintenable, mais dont les utilisateurs se plaignent d'une interface confuse, souffre principalement d'un manque de :**

- a) Qualité interne
- b) Qualité externe
- c) Qualité perçue
- d) Qualité en usage



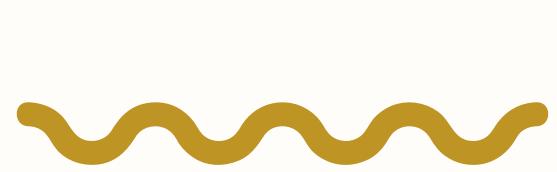
## 15. L'objectif principal des tests d'utilisabilité est :

- a) Identifier les vulnérabilités de sécurité.
- b) Mesurer la satisfaction et la facilité d'usage pour l'utilisateur.
- c) Vérifier la conformité RGPD.
- d) Tester la robustesse du backend.



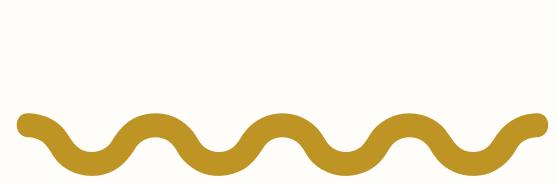
### 16. En approche Agile, le test devient :

- a) Une activité terminale après la revue du sprint.
- b) Une activité continue, intégrée à chaque itération.
- c) Un audit externe indépendant.
- d) Une tâche optionnelle selon la vitesse.



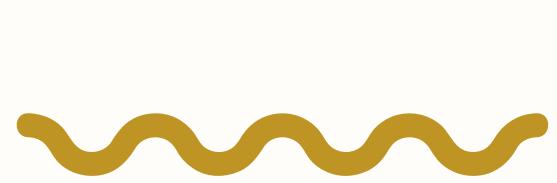
**17. Le cycle RED → GREEN → REFACTOR illustre :**

- a) Le processus BDD.
- b) Le cycle de revue QA.
- c) Le Test Driven Development (TDD).
- d) Le pipeline CI/CD.



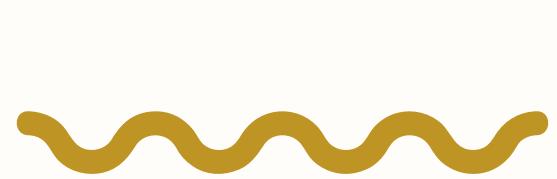
**18. Le BDD a pour finalité principale de :**

- a) Simuler des tests unitaires avant le développement.
- b) Décrire les comportements métiers dans un langage commun exécutable.
- c) Automatiser les tests de performance.
- d) Remplacer les spécifications techniques.



**19. Le Shift-Left Testing se traduit par :**

- a) L'ajout de tests après la mise en production.
- b) Le déplacement des activités de test plus tôt dans le cycle de développement.
- c) L'automatisation exclusive des tests unitaires.
- d) Le regroupement des tests dans la phase de validation.

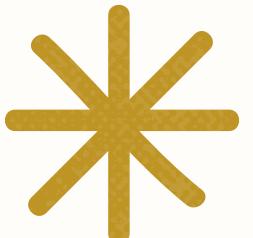


**20. Une Quality Gate dans une chaîne CI/CD sert à :**

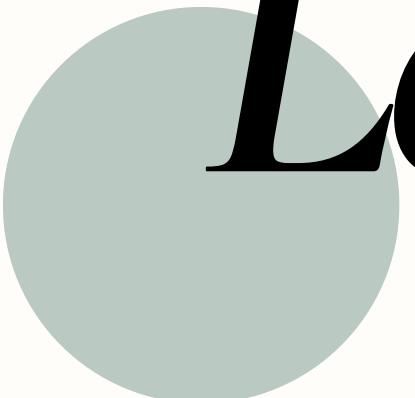
- a) Permettre la mise en production immédiate sans vérification.
- b) Bloquer le pipeline si une étape de qualité échoue.
- c) Créer des tests manuels supplémentaires.
- d) Supprimer les anomalies mineures automatiquement.

## Correction

N° Question	Réponse correcte
1	b
2	c
3	d
4	c
5	b
6	b
7	a
8	b
9	b
10	c
11	b
12	b
13	c
14	c
15	b
16	b
17	c
18	b
19	b
20	b



# Qualité



# *Logicielle*

---

Youssef Touati ESIEA

---