Formation Universitaire & Professionnelle | Génie Logiciel

Qualité et Tests de Logiciels

Guide complet – Théorie, Outils, TDD, CI/CD & Pratique

Ce document constitue un référentiel exhaustif sur les tests logiciels. Il couvre les fondements théoriques, la pratique des outils modernes, la méthodologie TDD, les pipelines CI/CD et les ateliers pratiques.

AGBANYO Yawotsè Emile

ATHON Balakyme Maxime

Génie Logiciel | Assurance Qualité & Tests

300+	6	30+	100+
pages équivalent en contenu	parties thématiques	exemples de code	concepts expliqués

Table des Matières

01 Introduction & Enjeux de la Qualité Logicielle

Durée : 20 minutes | Niveau : Fondamental

1.1 Pourquoi la qualité logicielle est-elle critique ?

Dans un monde où les systèmes informatiques orchestrent des infrastructures critiques — hôpitaux, transports aériens, marchés financiers, services gouvernementaux — la qualité du logiciel n'est plus un luxe mais une nécessité absolue. Un défaut logiciel non détecté peut entraîner des pertes financières colossales, des préjudices physiques graves, voire des pertes humaines.

Le Consortium for IT Software Quality (CISQ) estime à **2,41 billions de dollars** le coût annuel mondial des bugs logiciels. Ce chiffre inclut les coûts directs (correction, re-déploiement) et indirects (pertes de productivité, atteinte à la réputation, pertes clients).

Indicateur	Valeur	Source
Coût annuel des bugs logiciels (monde)	2 410 milliards $	CISQ 2022
Coût supplémentaire d'un bug en production vs conception	×100	IBM Systems Science
Projets dépassant le budget à cause des bugs	85 %	Tricentis
Bugs détectés en production auraient pu être évités	73 %	SmartBear Survey
Développeurs passant du temps à corriger des bugs	60 %	Cambridge Study

Sources : CISQ – Consortium for IT Software Quality, IBM Research, Tricentis Annual Bug Report

1.2 Coût des bugs selon la phase de détection

L'une des lois fondamentales du génie logiciel stipule que le coût de correction d'un défaut est directement proportionnel au temps écoulé depuis son introduction. Plus on détecte tard, plus on paie cher. C'est la règle du coût exponentiel de la correction.

Phase de détection	Coût relatif	Caractéristiques
Conception & Exigences	×1	Modification de document uniquement
Architecture & Design	×3 à ×6	Révision des spécifications techniques
Développement (codage)	×6 à ×10	Modification du code source
Tests (recette)	×15 à ×20	Correction + re-test + re-validation
Production (livré)	×40 à ×100	Patch d'urgence + impact utilisateurs réels

1.3 Échecs logiciels célèbres

L'histoire du génie logiciel est jalonnée de catastrophes causées par des bugs non détectés. Ces exemples réels illustrent de façon dramatique l'enjeu des tests logiciels :

■ Ariane 5 – 1996

La fusée Ariane 5 explosa 37 secondes après son lancement en raison d'un bug dans le système de navigation. La cause : une valeur flottante 64 bits fut convertie en entier 16 bits, causant un dépassement (overflow) non géré. Perte estimée : 370 millions de dollars. Ce bug était hérité du logiciel d'Ariane 4 mais n'avait jamais été testé dans les conditions d'Ariane 5, dont la vitesse était 5 fois supérieure.

✍■ Boeing 737 MAX – 2018-2019

Le système de contrôle de vol MCAS (Maneuvering Characteristics Augmentation System) présentait un bug critique : il se basait sur un seul capteur d'angle d'attaque défaillant pour corriger automatiquement la trajectoire, entraînant deux crashs. 346 personnes ont perdu la vie. Coût total pour Boeing : plus de 20 milliards de dollars, sans compter les poursuites judiciaires. L'enquête révéla une absence de tests de scénarios de défaillance et une culture d'entreprise privilégiant la vitesse de mise sur le marché à la sécurité.

■ Knight Capital Group – 2012

Une mise à jour logicielle défectueuse du système de trading automatique de Knight Capital déclencha des ordres d'achat et vente à haute fréquence incontrôlés pendant 45 minutes. La société perdit 440 millions de dollars en une seule matinée, soit plus que ses fonds propres. La cause : un ancien code jamais supprimé (flag SMARS) fut accidentellement réactivé lors du déploiement. Absence de tests de non-régression et de procédures de rollback.

■ NHS – WannaCry 2017

Le ransomware WannaCry paralysa le Système de Santé National britannique (NHS) en exploitant une vulnérabilité Windows non patchée (EternalBlue, développée par la NSA). Plus de 80 hôpitaux furent touchés, 6 000 rendez-vous annulés, des opérations reportées. Ce cas illustre l'importance des tests de sécurité et de la gestion des mises à jour logicielles.

1.4 Définitions fondamentales

Qualité Logicielle

Ensemble des propriétés et caractéristiques d'un produit logiciel qui lui confèrent l'aptitude à satisfaire des besoins exprimés ou implicites. Selon ISO/IEC 25010, elle comprend : la fonctionnalité, la performance, la compatibilité, l'utilisabilité, la fiabilité, la sécurité, la maintenabilité et la portabilité.

QA – Quality Assurance (Assurance Qualité)

Processus proactif et préventif qui s'attache à mettre en place les bonnes pratiques, méthodes et processus tout au long du cycle de développement pour éviter l'apparition de défauts. Orienté processus. Exemples : revue de code, audits, définition de standards de codage, formation des équipes.

QC – Quality Control (Contrôle Qualité)

Processus réactif et correctif qui vérifie que le produit final satisfait les critères de qualité définis. Orienté produit. Exemples : tests fonctionnels, inspections, benchmarks de performance.

Test Logiciel

Processus d'évaluation d'un composant ou système pour vérifier qu'il satisfait des exigences définies, et d'identification des différences entre les résultats attendus et les résultats réels. Les tests sont un sous-ensemble du QC.

Validation

Processus consistant à évaluer un système ou un composant pendant ou en fin de développement pour déterminer s'il satisfait les besoins réels de l'utilisateur. Question clé : Construisons-nous le bon produit ? (Are we building the right product?)

Vérification

Processus consistant à évaluer un système ou composant pour déterminer s'il satisfait les conditions imposées à la phase actuelle de développement. Question clé : Construisons-nous le produit correctement ? (Are we building the product right?)

Débogage (Debugging)

Processus de localisation, analyse et correction des défauts (bugs) dans un programme. Le débogage survient après qu'un test a révélé la présence d'un défaut.

Défaut / Bug

Imperfection ou déficience dans un produit logiciel qui peut causer une défaillance lors de son exécution. Distinguer : erreur (mistake humaine) → défaut (présence dans le code) → défaillance (comportement incorrect observable en exécution).

1.5 Le cycle de vie logiciel et la place des tests

Les tests ne constituent pas une phase unique intervenant en fin de projet. Selon les modèles modernes de développement logiciel, ils s'intègrent à chaque étape du cycle de vie :

- **Phase Exigences :** Revue des spécifications, critères d'acceptation (ATDD), tests d'acceptation définis
- **Phase Architecture :** Revue de conception, tests d'intégration planifiés, choix des outils de test
- **Phase Développement :** Tests unitaires (TDD), revues de code, analyse statique du code (linting)
- **Phase Test :** Tests fonctionnels, tests d'intégration, tests de régression, tests de performance
- **Phase Déploiement :** Tests de recette (UAT), tests de fumée (smoke tests), monitoring en production
- **Phase Maintenance :** Tests de régression après correctifs, tests de mise à jour, suivi des métriques

1.6 Normes et standards de qualité

Plusieurs normes internationales encadrent la qualité et les tests logiciels :

ISO/IEC 25010:2011 (SQuaRE)

Modèle de qualité du produit logiciel définissant 8 caractéristiques et 31 sous-caractéristiques. Remplace ISO/IEC 9126. Caractéristiques : fonctionnalité, performance, compatibilité, utilisabilité, fiabilité, sécurité, maintenabilité, portabilité.

IEEE 829

Standard pour la documentation des tests logiciels. Définit 8 types de documents : plan de test, spécification de conception de test, spécification de cas de test, etc.

ISTQB – International Software Testing Qualifications Board

Programme de certification mondial pour les testeurs logiciels. Niveaux : Foundation, Advanced (Test Analyst, Technical Test Analyst, Test Manager), Expert.

ISO/IEC 29119

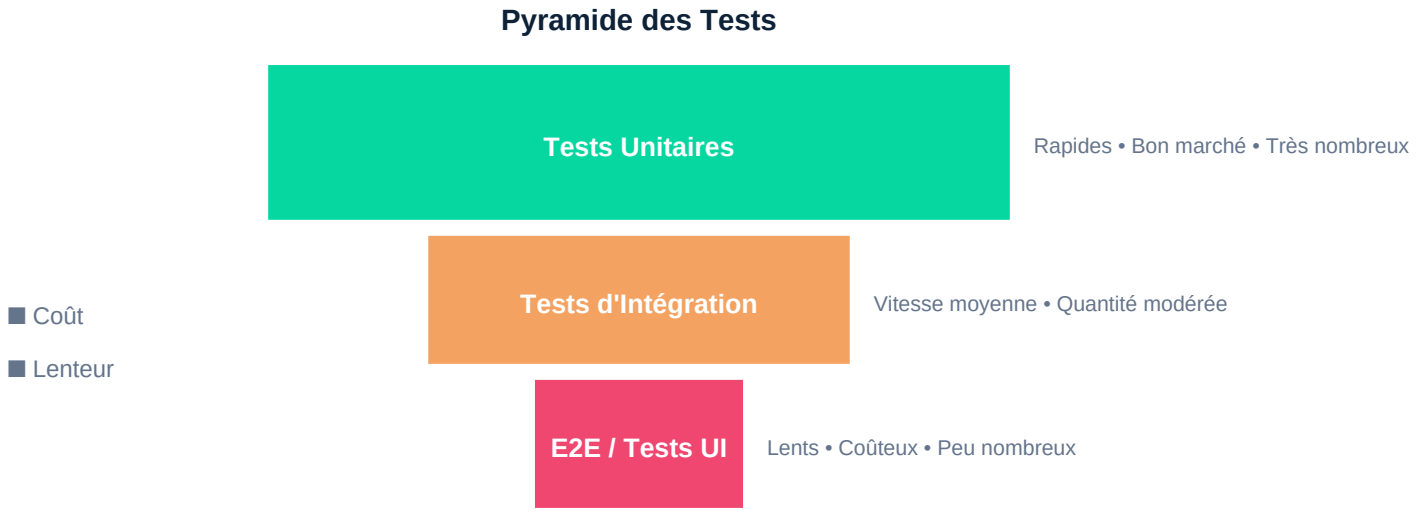
Série de normes pour les processus, documentation et techniques de test logiciel. Comprend 5 parties : concepts, processus, documentation, techniques, approches agiles.

02 Typologie des Tests Logiciels

Durée : 45 minutes | Tests unitaires, fonctionnels, intégration, E2E, régression, performance

2.1 La Pyramide des Tests

La pyramide des tests est un concept introduit par Mike Cohn dans son ouvrage *Succeeding with Agile* (2009). Elle guide la répartition idéale des tests selon leur granularité, vitesse d'exécution et coût de maintenance.



La règle empirique communément admise est la suivante : **70% de tests unitaires**, **20% de tests d'intégration** et **10% de tests E2E**. Cette répartition maximise la vélocité des équipes tout en maintenant une couverture de confiance élevée.

2.2 Tests Unitaires

Un test unitaire est un test qui vérifie le comportement d'une **unité de code isolée** — généralement une fonction, une méthode ou une classe — en dehors de son contexte d'intégration. L'isolation est obtenue grâce à des mocks, stubs et fakes qui remplacent les dépendances réelles.

Propriétés d'un bon test unitaire (FIRST) :

- **F – Fast :** S'exécute en millisecondes. Une suite complète de tests unitaires doit tourner en quelques secondes.
- **I – Isolated :** Aucune dépendance entre tests. L'ordre d'exécution ne doit pas affecter les résultats.
- **R – Repeatable :** Produit toujours le même résultat. Pas de dépendance au réseau, à l'heure, à des données externes.
- **S – Self-Validating :** Retourne PASS ou FAIL automatiquement. Pas d'interprétation manuelle requise.
- **T – Timely :** Écrits au bon moment : avant (TDD) ou immédiatement lors du développement.

Exemple : test unitaire de la fonction add() en Jest

Exemple – Tests Unitaires avec Jest

```
// userService.js
function add(a, b) {
  return a + b;
}
module.exports = { add };

// userService.test.js
const { add } = require('./userService');

describe('Fonction add', () => {
  test('add(2, 3) doit retourner 5', () => {
    expect(add(2, 3)).toBe(5);
  });
  test('add(-1, 1) doit retourner 0', () => {
    expect(add(-1, 1)).toBe(0);
  });
  test('add(0, 0) doit retourner 0', () => {
    expect(add(0, 0)).toBe(0);
  });
});
```

2.3 Tests Fonctionnels

Les tests fonctionnels vérifient que le système se comporte conformément aux exigences fonctionnelles spécifiées. Contrairement aux tests unitaires qui testent l'implémentation, les tests fonctionnels s'intéressent uniquement au comportement visible : pour une entrée donnée, quelle est la sortie ?

Ils se basent sur des **cas de test** dérivés directement des spécifications fonctionnelles, des user stories ou des cas d'usage. Les outils Selenium, Cypress et Playwright permettent d'automatiser les tests fonctionnels sur les interfaces web.

Types de tests fonctionnels :

- **Tests de boîte noire (Black-box) :** Le testeur ne connaît pas l'implémentation interne. Il teste uniquement les entrées/sorties visibles. Techniques : partitionnement d'équivalence, analyse de valeurs limites, tables de décision.
- **Tests de boîte blanche (White-box) :** Le testeur connaît le code source et l'utilise pour définir les cas de test. Techniques : couverture de statements, branches, chemins.
- **Tests de boîte grise (Grey-box) :** Combinaison des deux approches : connaissance partielle de l'implémentation pour tester les interfaces et les flux de données.

2.4 Tests d'Intégration

Les tests d'intégration vérifient les interactions entre plusieurs composants ou modules du système. Ils cherchent à détecter les problèmes qui n'apparaissent pas lors des tests unitaires mais qui émergent lorsque les modules communiquent ensemble.

Stratégies d'intégration :

- **Big Bang :** Tous les modules sont intégrés simultanément et testés ensemble. Simple à planifier mais difficile à déboguer en cas d'échec car l'origine de l'erreur est difficile à identifier.
- **Top-Down :** On commence par tester les modules de haut niveau (interface) et on descend progressivement vers les modules de bas niveau (utilitaires). Les modules non développés sont remplacés par des stubs.
- **Bottom-Up :** On commence par les modules de bas niveau et on remonte. Les modules de haut niveau sont remplacés par des drivers de test. Facilite le test des modules fondamentaux.
- **Incrémentale :** Approche hybride (recommandée) : on intègre et teste les modules progressivement, en combinant top-down et bottom-up selon la structure du système.

2.5 Tests End-to-End (E2E)

Les tests E2E simulent le comportement d'un utilisateur réel en testant l'application de bout en bout, depuis l'interface utilisateur jusqu'à la base de données, en passant par tous les services intermédiaires. Ils constituent le niveau le plus élevé de la pyramide des tests.

Ces tests sont les plus représentatifs de l'expérience utilisateur réelle mais sont également les plus lents à exécuter, les plus coûteux à maintenir et les plus fragiles face aux changements de l'interface. Il est donc crucial de les limiter aux parcours critiques.

Outils E2E modernes :

- **Cypress :** Framework JavaScript moderne. Exécution dans le navigateur, debugging en temps réel, screenshots automatiques. Très populaire pour les SPA React/Angular/Vue.
- **Playwright (Microsoft) :** Support multi-navigateurs (Chromium, Firefox, WebKit). API puissante pour les scénarios complexes. Support TypeScript natif.
- **Selenium WebDriver :** Le vétéran (2004). Multi-langages (Java, Python, JS, C#). Support de nombreux navigateurs. Courbe d'apprentissage plus élevée.
- **Puppeteer (Google) :** Automation Chrome/Chromium via l'API DevTools. Idéal pour les captures d'écran, PDF et scraping.

2.6 Tests de Régression

Les tests de régression constituent l'ensemble des tests réexécutés après chaque modification du code source pour s'assurer que les fonctionnalités existantes n'ont pas été dégradées. La régression est l'un des risques les plus fréquents dans le développement logiciel : une modification apparemment anodine peut casser une fonctionnalité distante en apparence.

Gestion d'une suite de régression : La suite de régression doit être sélectionnée intelligemment. Tester à 100% à chaque commit serait trop lent ; on sélectionne les tests par **analyse d'impact** (quels modules sont affectés par le changement ?) et par **priorisation par risque métier**.

2.7 Tests de Performance

Les tests de performance mesurent les caractéristiques non-fonctionnelles d'un système sous charge : temps de réponse, débit (throughput), utilisation des ressources, stabilité.

- **Test de charge (Load Test) :** Simule le nombre attendu d'utilisateurs simultanés en conditions normales. Objectif : vérifier que le système respecte les SLAs (Service Level Agreements) définis.
- **Test de stress (Stress Test) :** Pousse le système au-delà de sa capacité nominale pour identifier son point de rupture. Crucial pour comprendre le comportement en cas de pic inattendu.
- **Test d'endurance (Soak Test) :** Exécute une charge normale sur une longue durée (heures, jours) pour détecter les fuites mémoire, la dégradation progressive des performances ou l'accumulation de données temporaires.
- **Test de montée en charge (Spike Test) :** Simule une montée brutale et massive du trafic (Black Friday, événement viral) pour tester la capacité d'auto-scaling et la résilience du système.
- **Test de capacité (Volume Test) :** Teste le comportement avec de très grands volumes de données pour identifier les goulots d'étranglement liés aux requêtes de base de données ou aux traitements en lot.

Outils de performance : *Apache JMeter (Java), k6 (JavaScript), Gatling (Scala), Locust (Python), Artillery (Node.js)*

2.8 Autres types de tests importants

- **Tests de sécurité (Security Testing) :** Vérifient la résistance aux attaques (injection SQL, XSS, CSRF, authentification). Outils : OWASP ZAP, Burp Suite, Snyk.
- **Tests d'acceptation utilisateur (UAT) :** Tests effectués par les utilisateurs finaux avant la mise en production pour valider que le système répond à leurs besoins métier réels.
- **Tests d'accessibilité :** Vérifient que l'application est utilisable par des personnes en situation de handicap. Normes WCAG 2.1 (A, AA, AAA).
- **Tests exploratoires :** Tests non scriptés où le testeur explore librement l'application en cherchant des comportements inattendus. Complémentaires aux tests automatisés.
- **Tests de mutation (Mutation Testing) :** Technique avancée : on modifie intentionnellement le code source (mutations) pour vérifier que les tests détectent ces changements. Mesure la qualité des tests eux-mêmes.
- **Tests de fumée (Smoke Tests) :** Ensemble minimal de tests exécutés après chaque déploiement pour vérifier que les fonctionnalités de base fonctionnent. Analogue à 'allumer le moteur' avant de conduire.
- **Tests de compatibilité :** Vérifient le comportement sur différents navigateurs, systèmes d'exploitation, versions d'OS, résolutions d'écran.

2.9 Tableau Comparatif des Types de Tests

Type	Granularité	Vitesse	Coût maint.	Outils
Unitaire	Fonction/méthode	< 1 ms	Faible	Jest, JUnit, PHPUnit
Fonctionnel	Feature complète	Rapide	Moyen	Cypress, Selenium
Intégration	Modules combinés	Quelques sec.	Moyen	Postman, JUnit
E2E	Parcours complet	Minutes	Élevé	Cypress, Playwright
Régression	Variables	Variable	Variable	Tous
Performance	Système entier	Très lent	Élevé	JMeter, k6
Sécurité	Application	Long	Élevé	OWASP ZAP, Snyk
Fumée	Fonctions clés	Rapide	Faible	Tous

Note : 'Coût maint.' = coût de maintenance des tests, pas du système

Outils de Test par Technologie

03

Durée : 30 minutes | Jest, Mocha, PHPUnit, JUnit

3.1 Jest – Le standard JavaScript/TypeScript

Jest est le framework de test le plus populaire de l'écosystème JavaScript. Créé par Facebook (désormais Meta) en 2014, il est devenu le standard de facto pour les projets React, Node.js et TypeScript. Sa force réside dans son approche **tout-en-un** : runner de tests, bibliothèque d'assertions, système de mocks et rapport de couverture de code sont intégrés nativement, sans dépendances supplémentaires.

Caractéristiques clés de Jest :

- Zero configuration : fonctionne sans aucune configuration pour les projets CRA, Next.js, Node.js
- Snapshot testing : capture et compare automatiquement le rendu de composants React
- Mocking automatique : jest.mock(), jest.fn(), jest.spyOn() pour isoler les dépendances
- Code coverage intégré : --coverage génère un rapport HTML complet
- Tests parallèles : exécution simultanée des suites de tests pour maximum de performance
- Mode watch : --watch re-exécute uniquement les tests affectés par les modifications

Installation et configuration :

Installation Jest

```
# Installation
npm install --save-dev jest @types/jest

# Dans package.json
{
  "scripts": {
    "test": "jest",
    "test:watch": "jest --watch",
    "test:coverage": "jest --coverage"
  },
  "jest": {
    "testEnvironment": "node",
    "collectCoverageFrom": ["src/**/*.js"]
  }
}
```

Exemple complet : service utilisateur

userService.js

```
// userService.js
const users = [];

function createUser(name, email) {
  if (!name || !email) throw new Error('Champs requis manquants');
```

userService.test.js

```
// userService.test.js
const { createUser, getUserById } = require('./userService');

describe('UserService', () => {

  describe('createUser()', () => {
    test('doit créer un utilisateur avec un id auto-incrémenté', () => {
      const user = createUser('Alice', 'alice@test.com');
      expect(user).toMatchObject({ name: 'Alice', email: 'alice@test.com' });
      expect(user.id).toBeDefined();
      expect(typeof user.id).toBe('number');
    });

    test('doit lever une erreur si les champs sont manquants', () => {
      expect(() => createUser('', 'test@test.com'))
        .toThrow('Champs requis manquants');
    });
  });

  describe('getUserById()', () => {
    test('doit retourner null si utilisateur inexistant', () => {
      expect(getUserById(9999)).toBeNull();
    });
  });
});
```

Matchers Jest les plus utilisés :

Matcher	Description
`toBe(value)`	Égalité stricte (===)
`toEqual(value)`	Égalité profonde pour objets/tableaux
`toBeNull() / toBeUndefined()`	Vérifie null ou undefined
`toBeTruthy() / toBeFalsy()`	Vérifie la véracité
`toContain(item)`	Vérifie qu'un tableau contient un élément
`toThrow(message)`	Vérifie qu'une exception est levée
`toBeCloseTo(num, precision)`	Comparaison de nombres flottants
`toHaveBeenCalled()`	Vérifie qu'un mock a été appelé
`toMatchObject(obj)`	Vérifie qu'un objet contient les propriétés
`toMatchSnapshot()`	Comparaison par snapshot (React)

3.2 Mocha + Chai – Flexibilité pour Node.js

Mocha est un framework de test flexible pour Node.js, différent de Jest dans son approche : là où Jest est tout-en-un, Mocha est minimaliste et s'associe à des bibliothèques spécialisées. **Chai** fournit les assertions (avec 3 styles : assert, should, expect), **Sinon** gère les mocks et spies.

Mocha + Chai – Exemple

```
# Installation
npm install --save-dev mocha chai

// test/cart.test.js – Mocha + Chai
```

3.3 PHPUnit – Le standard PHP

PHPUnit, créé par Sebastian Bergmann, est le framework de test de référence pour PHP. Il est intégré nativement dans les frameworks Laravel (via php artisan test), Symfony et Magento. Basé sur l'architecture xUnit (comme JUnit), PHPUnit supporte les tests unitaires, les tests d'intégration, les data providers, les mocks et les assertions avancées.

PHPUnit – Exemple complet avec DataProvider

```
# Installation via Composer
composer require --dev phpunit/phpunit ^10
# Exécution
./vendor/bin/phpunit --testdox

<?php // tests/Unit/CartServiceTest.php
namespace Tests\Unit;

use PHPUnit\Framework\TestCase;
use App\Services\CartService;

class CartServiceTest extends TestCase {
    private CartService $cartService;

    protected function setUp(): void {
        $this->cartService = new CartService();
    }

    public function test_calculate_total_with_multiple_items(): void {
        $items = [
            ['price' => 10.0, 'quantity' => 2],
            ['price' => 5.0,  'quantity' => 4],
        ];
        $total = $this->cartService->calculateTotal($items);
        $this->assertEquals(40.0, $total, 'Le total doit être 40.00 EUR');
    }

    /** @dataProvider discountProvider */
    public function test_apply_discount($total, $pct, $expected): void {
        $this->assertEqualsWithDelta(
            $expected, $this->cartService->applyDiscount($total, $pct), 0.01
        );
    }

    public static function discountProvider(): array {
        return [
            'discount 10% sur 100' => [100, 10, 90],
            'discount 50% sur 200' => [200, 50, 100],
        ];
    }
}
```

3.4 JUnit 5 – Le pilier du test Java

JUnit est le framework de test unitaire le plus influent de l'histoire du génie logiciel. Créé par Kent Beck et Erich Gamma en 1997, il a inspiré toute la famille xUnit (PHPUnit, NUnit, pytest). JUnit 5 est une réécriture complète de JUnit 4, composée de trois modules : **JUnit Platform** (moteur d'exécution), **JUnit Jupiter** (API de test moderne) et **JUnit Vintage** (compatibilité JUnit 3/4).

JUnit 5 – Exemple avec @ParameterizedTest

```
// pom.xml – Dépendance Maven
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>5.10.0</version>
  <scope>test</scope>
</dependency>

// CartServiceTest.java
import org.junit.jupiter.api.*;
import org.junit.jupiter.params.*;
import org.junit.jupiter.params.provider.*;
import static org.junit.jupiter.api.Assertions.*;

@DisplayName('Tests du service Panier')
class CartServiceTest {
    private CartService cartService;

    @BeforeEach
    void setUp() { cartService = new CartService(); }

    @Test
    @DisplayName('Le total de 2 articles doit être correct')
    void calculateTotal_shouldReturnCorrectSum() {
        List<Item> items = List.of(
            new Item(10.0, 2), new Item(5.0, 4)
        );
        assertEquals(40.0, cartService.calculateTotal(items), 0.001);
    }

    @ParameterizedTest
    @CsvSource({'100,10,90', '200,50,100', '50,20,40'})
    void applyDiscount_shouldReturnCorrectValue(
            double total, double pct, double expected) {
        assertEquals(expected, cartService.applyDiscount(total, pct), 0.001);
    }
}
```

3.5 Autres outils notables

- **pytest (Python) :** Framework de test le plus populaire pour Python. Simple, extensible via plugins, supporte les fixtures puissantes, le paramétrage et les tests de données.
- **Vitest (JavaScript) :** Alternatif à Jest, compatible avec Vite. Ultra-rapide grâce à esbuild. Compatible avec l'API Jest (migration facile).
- **Cypress Component Testing :** Tests de composants isolés directement dans le navigateur. Idéal pour React, Vue, Angular sans naviguer dans l'application complète.

- **Postman / Newman :** Tests d'API REST via collections Postman. Newman permet l'exécution en ligne de commande dans les pipelines CI/CD.
- **k6 (Grafana) :** Outil de test de charge écrit en Go, scriptable en JavaScript. Génère des rapports de performance riches, intégration Grafana Cloud.
- **SonarQube :** Analyse statique de qualité de code. Détecte les code smells, bugs potentiels, vulnérabilités de sécurité. S'intègre dans les pipelines CI/CD.

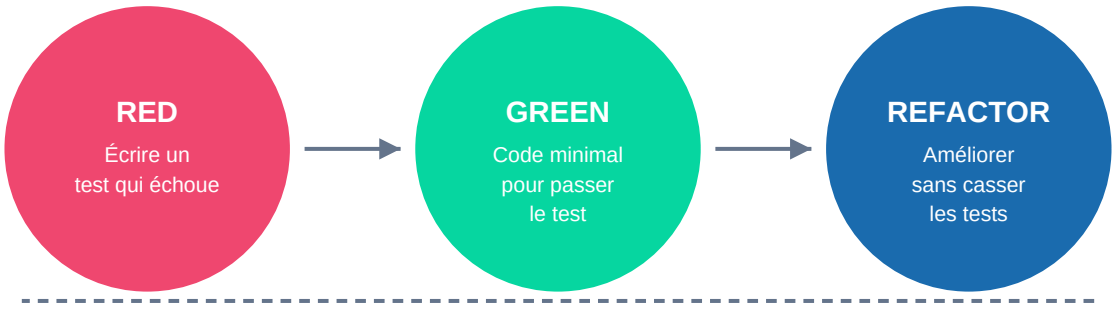
04 TDD – Test Driven Development

Durée : 30 minutes | Red → Green → Refactor

4.1 Principes fondamentaux du TDD

Le Test Driven Development (développement piloté par les tests) est une pratique de développement logiciel introduite et popularisée par **Kent Beck** dans son ouvrage *Test Driven Development: By Example* (2002). L'idée fondatrice est radicalement contre-intuitive : **on écrit le test avant d'écrire le code de production correspondant**. Le test échoue d'abord (RED), on écrit ensuite le code minimal pour le faire passer (GREEN), puis on améliore le code (REFACTOR).

Le TDD n'est pas une technique de test — c'est une technique de **conception logicielle**. L'écriture préalable des tests force le développeur à réfléchir à l'interface de son code avant son implémentation, résultant en un code plus modulaire, plus testable et mieux conçu.



4.2 Le Cycle Red → Green → Refactor

Phase RED – Écrire un test qui échoue

Avant toute ligne de code de production, on écrit un test pour la fonctionnalité souhaitée. Ce test doit **impérativement échouer** à ce stade. Si le test passe d'emblée, c'est qu'il est mal écrit (il ne teste pas ce qu'il devrait) ou que la fonctionnalité existe déjà. Le test doit être précis, focalisé sur un seul comportement.

Phase RED

```
// Étape RED : le test est écrit, la fonction n'existe pas encore
test('calculateTotal([{price:10, qty:2}]) retourne 20', () => {
  expect(calculateTotal([{ price: 10, quantity: 2 }])).toBe(20);
});

// Résultat : ReferenceError: calculateTotal is not defined
// ■ FAIL – C'est NORMAL et ATTENDU
```

Phase GREEN – Code minimal pour passer le test

On écrit le **code le plus simple possible** pour faire passer le test. Même si le code paraît simpliste, voire naïf, l'objectif est uniquement de passer le test au vert. On n'optimise pas, on n'anticipe pas. La règle : *Make it work*.

```
Phase GREEN

// Étape GREEN : implémentation minimale
function calculateTotal(items) {
  return items.reduce((sum, item) => sum + item.price * item.quantity, 0);
}

// Résultat : PASS ■
// Tests: 1 passed, 1 total — Time: 0.324s
```

Phase REFACTOR – Améliorer sans casser

Maintenant que le test passe, on améliore le code : on élimine les duplications, on renomme les variables pour plus de clarté, on extrait des fonctions auxiliaires. **Les tests restent verts tout au long du refactoring.** Si un test passe au rouge pendant le refactoring, on a introduit une régression.

```
Phase REFACTOR

// Étape REFACTOR : code amélioré, même comportement
const sumItemPrice = (item) => item.price * item.quantity;

const calculateTotal = (items) =>
  items.reduce((sum, item) => sum + sumItemPrice(item), 0);

// Tests toujours PASS ■ — le comportement n'a pas changé
```

4.3 Bénéfices et limites du TDD

- ✓ **Réduction des bugs :** Les études montrent une réduction de 40 à 80% des bugs en production pour les équipes pratiquant le TDD de manière disciplinée (Microsoft Research, IBM Research, 2008).
- ✓ **Meilleure conception :** En écrivant les tests en premier, le développeur est forcé de concevoir des interfaces claires et des modules découplés. Code testable = code bien conçu.
- ✓ **Documentation vivante :** Les tests constituent une documentation à jour du comportement attendu du système, contrairement à la documentation narrative qui devient rapidement obsolète.
- ✓ **Refactoring en confiance :** La suite de tests est un filet de sécurité qui permet de restructurer le code en confiance. Sans tests, refactoriser est risqué.
- ✓ **Feedback immédiat :** Le cycle Red-Green-Refactor fournit un feedback rapide (< 1 minute par itération), maintenant le développeur dans un état de flow productif.

Limites du TDD :

- ✗ Courbe d'apprentissage significative — les premières semaines de pratique TDD sont plus lentes
- ✗ Moins adapté pour les prototypes exploratoires et le code expérimental
- ✗ Difficile pour les interfaces graphiques complexes (nécessite des couches d'abstraction UI)

✗ Nécessite une discipline d'équipe — un seul développeur ne pratiquant pas le TDD peut dégrader la qualité

✗ Le TDD seul ne garantit pas la qualité architecturale — il faut combiner avec les revues de code

4.4 Mocks, Stubs et Doubles de Test

Pour isoler les unités de code lors des tests, on remplace les dépendances réelles (base de données, API externe, système de fichiers) par des **doublures de test**. Ces objets simulent le comportement des dépendances réelles sans leurs contraintes.

- **Dummy :** Objet passé en paramètre mais jamais utilisé. Sert à satisfaire les signatures de méthodes.
- **Stub :** Retourne des réponses prédéfinies aux appels. Ne contient pas de logique. Ex : getUserById() retourne toujours le même objet fictif.
- **Fake :** Implémentation simplifiée qui fonctionne réellement mais n'est pas adaptée à la production. Ex : base de données en mémoire (SQLite) à la place de PostgreSQL.
- **Spy :** Wrapper autour d'un objet réel qui enregistre les appels. On peut vérifier a posteriori comment il a été utilisé.
- **Mock :** Objet préprogrammé avec des attentes sur les appels qu'il doit recevoir. Vérifie automatiquement que ces attentes ont été satisfaites.

Mocks avec Jest

```
// Exemple de Mock avec Jest
const dbModule = require('./database');
jest.mock('./database');  // Mock automatique du module

test('getUserById appelle la DB avec le bon id', () => {
  dbModule.findById.mockReturnValue({ id: 1, name: 'Alice' });

  const result = getUserById(1);

  expect(dbModule.findById).toHaveBeenCalledWith(1);
  expect(dbModule.findById).toHaveBeenCalledTimes(1);
  expect(result.name).toBe('Alice');
});
```

4.5 BDD – Behavior Driven Development

Le Behavior Driven Development (BDD), introduit par Dan North en 2006, est une extension du TDD qui déplace l'accent des tests techniques vers le comportement métier du système. Il utilise un langage naturel structuré (**Gherkin**) pour exprimer les scénarios de test sous la forme **Given / When / Then**, permettant aux non-développeurs (product owners, clients) de participer à la définition des tests.

Gherkin – Scénarios BDD

```
# Fichier Gherkin : features/panier.feature
Feature: Calcul du total du panier
  En tant qu'acheteur
  Je veux voir le total de mon panier
  Pour pouvoir valider ma commande
```

*Outils BDD : **Cucumber** (Java, Ruby, JS), **Behat** (PHP), **SpecFlow** (.NET), **Behave** (Python)*

05 CI/CD & Automatisation des Tests

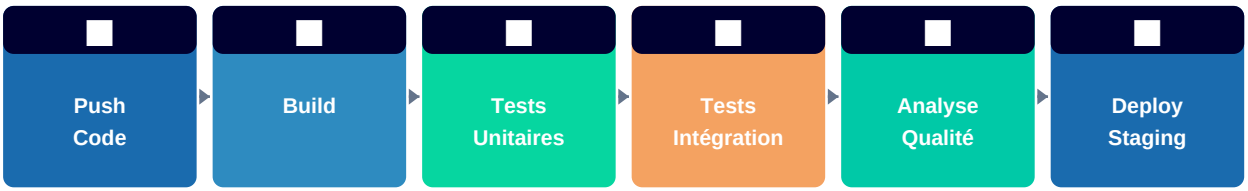
Durée : 30 minutes | GitHub Actions, GitLab CI

5.1 Intégration Continue (CI) – Concepts

L'Intégration Continue (CI, Continuous Integration) est une pratique DevOps consistant à intégrer fréquemment les modifications de code dans une branche partagée, en déclenchant automatiquement un pipeline de vérification à chaque push. Introduite par Martin Fowler et Kent Beck dans le cadre d'Extreme Programming (XP), la CI vise à détecter les problèmes d'intégration au plus tôt.

Les 8 pratiques de l'Intégration Continue (Martin Fowler) :

1. Maintenir un dépôt de sources unique (Git)
2. Automatiser le build (npm run build, mvn package, etc.)
3. Rendre le build auto-testant (tests lancés automatiquement)
4. Committer sur la branche principale chaque jour
5. Chaque commit déclenche un build sur un serveur CI
6. Corriger immédiatement les builds cassés
7. Garder le build court (< 10 minutes pour le feedback)
8. Tester dans un environnement clone de la production



5.2 Livraison Continue vs Déploiement Continu (CD)

Le CD désigne deux pratiques distinctes mais complémentaires qui prolongent la CI :

Livraison Continue (Continuous Delivery) : Le code est automatiquement construit, testé et prêt à être déployé en production à tout moment. Le déploiement final reste une décision humaine manuelle (un clic). C'est l'approche recommandée pour la plupart des entreprises car elle maintient un contrôle humain sur les mises en production.

Déploiement Continu (Continuous Deployment) : Toute modification qui passe l'ensemble des tests automatisés est automatiquement déployée en production sans intervention humaine. Pratiqué par Amazon, Netflix, Etsy. Nécessite une couverture de test et une surveillance très robustes.

5.3 GitHub Actions – Configuration complète

GitHub Actions est le service CI/CD intégré à GitHub, lancé en 2019. Gratuit pour les projets open-source (2 000 minutes/mois pour les privés), il utilise des fichiers YAML stockés dans le dossier **.github/workflows/**.

.github/workflows/ci.yml – Configuration complète

```yaml
# .github/workflows/ci.yml
name: CI – Tests & Qualité

on:
  push:
    branches: [main, develop, 'feature/**']
  pull_request:
    branches: [main]

env:
  NODE_VERSION: '20'
  COVERAGE_THRESHOLD: 80

jobs:
  lint:
    name: Analyse statique du code
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: ${{ env.NODE_VERSION }}
          cache: 'npm'
      - run: npm ci
      - run: npm run lint         # ESLint

  test:
    name: Tests unitaires & couverture
    runs-on: ubuntu-latest
    needs: lint          # Dépend du job lint
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: ${{ env.NODE_VERSION }}
          cache: 'npm'
      - run: npm ci
      - name: Lancer les tests avec couverture
        run: npm run test:coverage
      - name: Vérifier le seuil de couverture
        run: npx jest --coverage --coverageThreshold='{"global":{"lines":80}}'
      - uses: actions/upload-artifact@v4
        with:
          name: coverage-report
          path: coverage/

  integration:
    name: Tests d'intégration
    runs-on: ubuntu-latest
    needs: test
```

5.4 GitLab CI/CD – Pipeline YAML

GitLab CI/CD est intégré nativement dans GitLab (auto-hébergé ou GitLab.com). La configuration est définie dans le fichier **.gitlab-ci.yml** à la racine du projet. Son point fort : architecture multi-stages avec parallélisation native des jobs.

.gitlab-ci.yml – Pipeline multi-stages

```
# .gitlab-ci.yml
stages:
  - prepare
  - test
  - quality
  - deploy

variables:
  NODE_VERSION: '20'
  COVERAGE_THRESHOLD: '80'

.node_template: &node_defaults
  image: node:${NODE_VERSION}
  cache:
    key: ${CI_COMMIT_REF_SLUG}
    paths: [node_modules/]
  before_script:
    - npm ci --quiet

unit_tests:
  <<: *node_defaults
  stage: test
  script:
    - npm test -- --coverage --ci
    - npm run test:coverage
  coverage: '/Lines\s*:\s*(\d+\.?\d*)%/'
  artifacts:
    reports:
      coverage_report:
        coverage_format: cobertura
        path: coverage/cobertura-coverage.xml

sonarqube_analysis:
  stage: quality
  image: sonarsource/sonar-scanner-cli
  script:
    - sonar-scanner -Dsonar.projectKey=mon_projet
  only: [main, develop]

deploy_staging:
  stage: deploy
  script:
    - echo 'Déploiement en staging...'
    - ./scripts/deploy.sh staging
  environment:
    name: staging
    url: https://staging.monapp.com
```

5.5 Couverture de code et métriques

La couverture de code (code coverage) mesure le pourcentage de code source exécuté lors de l'exécution des tests. C'est un indicateur de la qualité de la suite de tests, **pas de la qualité du code lui-même**.

- **Statement Coverage :** % de lignes/instructions exécutées. La métrique la plus basique. 100% de statement coverage ne signifie pas que tous les cas sont couverts.
- **Branch Coverage :** % de branches (if/else, switch) testées. Plus pertinent car vérifie que les deux chemins d'une condition sont testés.
- **Function Coverage :** % de fonctions/méthodes appelées au moins une fois lors des tests.
- **Path Coverage :** % de chemins d'exécution possibles testés. La plus exhaustive mais combinatoriellement impossible à atteindre à 100%.

Seuil recommandé : *Un seuil de 80% de couverture de lignes est généralement recommandé comme minimum. Les modules critiques (paiement, sécurité, calculs financiers) doivent viser 90-100%. Attention : une couverture à 100% n'indique pas l'absence de bugs.*

06 Atelier Pratique – Calculateur de Panier

Durée : 45-60 minutes | Approche TDD guidée pas-à-pas

6.1 Énoncé et structure du projet

Dans cet atelier, vous allez développer le module de calcul d'un panier e-commerce en suivant rigoureusement la démarche TDD. Vous disposerez d'un fichier de départ avec des fonctions vides et d'un fichier de tests incomplets à compléter.

Structure du projet :

Structure du projet

```
mon-projet-tdd/
■■■ src/
■   ■■■ cart.js          # Fonctions à implémenter
■■■ tests/
■   ■■■ unit/
■   ■   ■■■ cart.test.js # Tests unitaires à compléter
■   ■■■ integration/
■       ■■■ cart.integration.test.js
■■■ package.json
■■■ jest.config.js
```

src/cart.js – Point de départ

```
// src/cart.js – Fichier de départ (fonctions vides)

/**
 * Calcule le total du panier
 * @param {Array} items – [{price: number, quantity: number}]
 * @returns {number} Total en EUR
 */
function calculateTotal(items) {
  // TODO – À implémenter en Étape 1
  return null;
}

/**
 * Applique une réduction en pourcentage
 * @param {number} total – montant avant réduction
 * @param {number} percent – pourcentage de réduction (0-100)
 */
function applyDiscount(total, percent) {
  // TODO – À implémenter en Étape 2
  return null;
}

function addTax(price, taxRate) {
  // TODO – À implémenter en Étape 3
```

6.2 Étapes TDD pas-à-pas

Étape 1 : calculateTotal – Écrire les tests d'abord

Tests unitaires – Étape 1

```
// tests/unit/cart.test.js – Étape 1
const { calculateTotal, applyDiscount, addTax } = require('../../src/cart');

describe('calculateTotal()', () => {

  test('panier vide retourne 0', () => {
    expect(calculateTotal([])).toBe(0);
  });

  test('un article (price:10, qty:2) retourne 20', () => {
    expect(calculateTotal([{ price: 10, quantity: 2 }])).toBe(20);
  });

  test('deux articles différents calcule le total correct', () => {
    const items = [
      { price: 10, quantity: 2 },  // 20
      { price: 5,  quantity: 4 },  // 20
    ];
    expect(calculateTotal(items)).toBe(40);
  });

  test('prix flottants arrondis à 2 décimales', () => {
    const items = [{ price: 1.1, quantity: 3 }];  // 3.30
    expect(calculateTotal(items)).toBeCloseTo(3.3, 2);
  });
});
```

Résultat attendu avant implémentation :

Sortie RED – Tests en échec (attendu)

```
$ npm test

 FAIL  tests/unit/cart.test.js
  calculateTotal()
    ✕ panier vide retourne 0 (3 ms)
    ✕ un article (price:10, qty:2) retourne 20 (1 ms)
    ✕ deux articles différents calcule le total correct (1 ms)
    ✕ prix flottants arrondis à 2 décimales (1 ms)

  ● calculateTotal() › panier vide retourne 0
    expect(received).toBe(expected)
    Expected: 0 | Received: null
```

6.3 Correction complète

src/cart.js – Correction complète

```
// src/cart.js – CORRECTION COMPLÈTE
```

Sortie GREEN – Tous les tests passent ■

```
$ npm test

 PASS  tests/unit/cart.test.js
  calculateTotal()
    ✓ panier vide retourne 0 (2 ms)
    ✓ un article (price:10, qty:2) retourne 20 (0 ms)
    ✓ deux articles différents calcule le total correct (1 ms)
    ✓ prix flottants arrondis à 2 décimales (0 ms)
  applyDiscount()
    ✓ 10% sur 100 retourne 90 (0 ms)
    ✓ 50% sur 200 retourne 100 (0 ms)
  addTax()
    ✓ TVA 20% sur 100 retourne 120 (0 ms)

Test Suites: 1 passed, 1 total
Tests:       7 passed, 7 total
Snapshots:   0 total
Time:        0.847 s, estimated 1 s

Coverage:
File     | % Stmts | % Branch | % Funcs | % Lines
cart.js  |   100   |   100    |   100   |   100
```

6.4 Plateformes en ligne recommandées

Pour les sessions de formation sans installation préalable, les plateformes suivantes permettent de coder et tester directement dans le navigateur :

- **Replit (replit.com) :** Environnement Node.js complet dans le navigateur. Terminal intégré, import npm, partage par URL. Idéal pour démarrer en 30 secondes. Créez un projet Node.js et collez le code directement.
- **CodeSandbox (codesandbox.io) :** Spécialisé JavaScript. Templates officiels pour Node.js + Jest. Exécution des tests en temps réel avec affichage des résultats. Collaboration multi-utilisateurs idéale pour le pair programming.
- **StackBlitz (stackblitz.com) :** Moteur WebContainer : Node.js complet dans le navigateur, sans serveur distant. Très rapide. Support npm complet.
- **GitHub Codespaces (github.com/codespaces) :** VSCode complet dans le navigateur avec environnement Docker configurable. Intégration GitHub directe. 30 heures gratuites/mois. Solution la plus puissante et la plus proche d'un environnement réel.

Synthèse – Bonnes Pratiques & Récapitulatif

Principes FIRST pour de bons tests

L'acronyme FIRST résume les cinq propriétés fondamentales d'une bonne suite de tests, indépendamment du langage ou du framework utilisé :

F – Fast (Rapide)

Les tests unitaires doivent s'exécuter en millisecondes. Une suite complète de centaines de tests unitaires doit tourner en quelques secondes. Des tests lents créent de la friction et découragent leur exécution fréquente.

I – Isolated (Isolé)

Chaque test est indépendant des autres. L'ordre d'exécution ne doit pas affecter les résultats. Pas d'état partagé entre tests. L'utilisation de mocks permet d'isoler les dépendances externes.

R – Repeatable (Reproductible)

Le même test produit toujours le même résultat dans n'importe quel environnement : machine du développeur, serveur CI, conteneur Docker. Pas de dépendance à l'heure système, au réseau, à des données variables.

S – Self-Validating (Auto-validant)

Le test détermine lui-même s'il passe ou échoue. Aucune interprétation humaine n'est requise pour lire le résultat. PASS ou FAIL, rien d'autre.

T – Timely (En temps voulu)

Les tests sont écrits au bon moment : idéalement avant le code (TDD) ou immédiatement lors du développement. Des tests écrits des semaines après le code sont généralement moins pertinents et plus difficiles à écrire.

Les 10 commandements des tests logiciels

1. Tu testeras tôt et tu testeras souvent — chaque commit déclenche les tests
2. Tu écriras des tests avant de coder (TDD) pour concevoir de meilleurs systèmes
3. Tu automatiseras tout test qui sera exécuté plus d'une fois
4. Tu maintiendras ta suite de tests propre comme ton code de production
5. Tu ne testeras qu'une seule chose à la fois dans chaque test unitaire
6. Tu donneras à tes tests des noms descriptifs qui expliquent ce qu'ils testent
7. Tu isoleras tes tests de leurs dépendances externes (DB, API, FS)
8. Tu mesureras ta couverture de code et tu fixeras un seuil minimum
9. Tu n'ignoreras jamais un test qui échoue — tu le corrigeras immédiatement
10. Tu intégreras tes tests dans ton pipeline CI/CD pour un feedback continu

Récapitulatif : Ce qu'il faut retenir absolument

Concept	Résumé en une phrase

Pyramide des tests	70% unitaires / 20% intégration / 10% E2E – plus c'est bas, plus c'est rapide
Tests unitaires	Test d'une fonction isolée – le fondement de toute suite de tests
Tests d'intégration	Test des interactions entre modules – détecte les incompatibilités
Tests E2E	Simule un utilisateur réel – crucial mais rare et fragile
TDD	Écrire le test AVANT le code – Red → Green → Refactor
Mocks	Remplacent les dépendances réelles (DB, API) pour isoler le code
CI/CD	Pipeline automatique déclenché à chaque commit – zéro bug non détecté
Couverture de code	Vise 80% minimum – indicateur de qualité des tests, pas du code
FIRST	Fast, Isolated, Repeatable, Self-Validating, Timely – propriétés d'un bon test
BDD	Gherkin Given/When/Then – tests compréhensibles par tous les stakeholders

Annexes

Glossaire complet

Assertion — Vérification qu'une condition est vraie dans un test. Ex : expect(result).toBe(5)

Behaviour Driven Development (BDD) — Extension du TDD utilisant le langage Gherkin pour exprimer les tests en termes métier compréhensibles par tous.

Branche (Branch) — Chemin d'exécution conditionnel dans le code (if/else). La couverture de branches mesure le % de chemins testés.

Bug / Défaut — Imperfection dans un programme causant un comportement inattendu ou incorrect.

CI/CD — Continuous Integration / Continuous Delivery ou Deployment. Pratiques DevOps d'automatisation du build, test et déploiement.

Code Coverage — Pourcentage de code exécuté lors des tests. Métrique de qualité de la suite de tests.

Data Provider — Mécanisme permettant d'exécuter un même test avec plusieurs jeux de données différents (PHPUnit, JUnit).

Driver de test — Module de substitution remplaçant un composant supérieur non encore développé lors des tests d'intégration bottom-up.

Fake — Implémentation simplifiée fonctionnelle d'une dépendance, utilisée uniquement pour les tests.

Fixture — Données ou état initial configurable avant chaque test. BeforeEach() / setUp() dans la plupart des frameworks.

Flaky Test — Test instable qui passe ou échoue de manière non déterministe selon les conditions d'exécution.

Happy Path — Scénario de test correspondant au flux normal et attendu, sans erreurs ni cas limites.

Integration Test — Test vérifiant les interactions entre plusieurs composants ou modules intégrés ensemble.

Mock — Objet de test préprogrammé avec des attentes sur les appels attendus. Vérifie l'interaction entre objets.

Mutation Testing — Technique mesurant la qualité des tests en introduisant délibérément des erreurs dans le code et en vérifiant que les tests les détectent.

Regression Test — Test réexécuté après une modification pour s'assurer que les fonctionnalités existantes n'ont pas régressé.

Smoke Test — Ensemble minimal de tests vérifiant les fonctionnalités critiques après un déploiement.

Spy — Wrapper autour d'un objet réel qui enregistre les appels effectués pour vérification ultérieure.

Stub — Remplacement d'une dépendance retournant des réponses prédéfinies. Pas de vérification des appels.

TDD — Test Driven Development. Pratique consistant à écrire les tests avant le code. Cycle : Red → Green → Refactor.

Test Unitaire — Test d'une unité de code isolée (fonction/méthode) sans dépendances externes réelles.

UAT — User Acceptance Testing. Tests de validation effectués par les utilisateurs finaux avant mise en production.

Ressources et références

Livres de référence :

- Kent Beck – Test Driven Development: By Example (2002) – La bible du TDD
- Robert C. Martin – Clean Code: A Handbook of Agile Software Craftsmanship (2008)
- Roy Osherove – The Art of Unit Testing (2009, 3e éd. 2021)
- Gerard Meszaros – xUnit Test Patterns: Refactoring Test Code (2007)
- Michael Feathers – Working Effectively with Legacy Code (2004)
- Gojko Adzic – Specification by Example (2011) – BDD et ATDD

Ressources en ligne :

- jestjs.io – Documentation officielle Jest
- martinfowler.com – Articles sur TDD, CI/CD, Mocks (incontournable)
- docs.cypress.io – Documentation Cypress E2E
- playwright.dev – Documentation Playwright
- phpunit.de – Documentation PHPUnit
- junit.org/junit5 – Documentation JUnit 5
- owasp.org – OWASP Testing Guide pour la sécurité
- istqb.org – Programme de certification ISTQB

Certifications professionnelles :

- **ISTQB Foundation Level (CTFL) :** Certification internationale de base pour testeurs. Couvre les fondamentaux : types de tests, techniques, gestion des tests.
- **ISTQB Advanced Level – Test Analyst :** Spécialisation sur les techniques de test avancées et la conception de tests.
- **ISTQB Advanced Level – Test Manager :** Gestion d'équipes de test, métriques, processus et reporting.
- **AWS Certified DevOps Engineer :** Certification Amazon Web Services couvrant CI/CD, Infrastructure as Code et monitoring.
- **Google Professional Cloud DevOps Engineer :** Certification Google Cloud pour les pratiques SRE et DevOps.