# TESTING (1h30min)

# AU MENU CE JOUR

# Introduction : Testing & QSI

**La qualité dans un SI** : une pro-activité nécessaire

**Le rôle du testing** : un levier stratégique

# Les différentes catégories de tests

**Les doublures de tests**

**Les classiques** : unitaire, intégration, e2e...

**Autres types** : pbt, visuel, manuels (chromatic)...

# Élaborer une stratégie de testing

**Testing trophee & pyramide** : quelle stratégie adopter ?

**TDD** : une stratégie de test ?

**Automatisation** : les bonnes pratiques CI/CD

# TESTING & QSI

# LA QUALITÉ DANS UN SI NE SE RÉSUME PAS À L'ABSENCE DE BUG

# LA QUALITÉ DANS UN SI

## UNE PROACTIVITÉ NÉCESSAIRE ET CONTEXTUELLE

| | |
|---|---|
| **FIABILITÉ** | Garantir un fonctionnement sans interruptions imprévues |
| **PERFORMANCE** | Assurer une réponse rapide |
| **SÉCURITÉ** | Protéger le SI contre les attaques |
| **ÉVOLUTIVITÉ** | Permettre au SI de grandir et rester compétitif |
| **PERTINENCE** | Répondre au besoin de ses utilisateurs et du métier |

***Spoiler** : dans certaines phases de développement (PoC, MVP...) on accepte parfois des concessions.*

# LE RÔLE DU TESTING

## UN LEVIER STRATÉGIQUE DE LA QUALITÉ

### PRÉVENIR + QUE GUÉRIR

Un bug en production peut coûter + cher qu'en développement* (indisponibilité de paiement, connexion...)

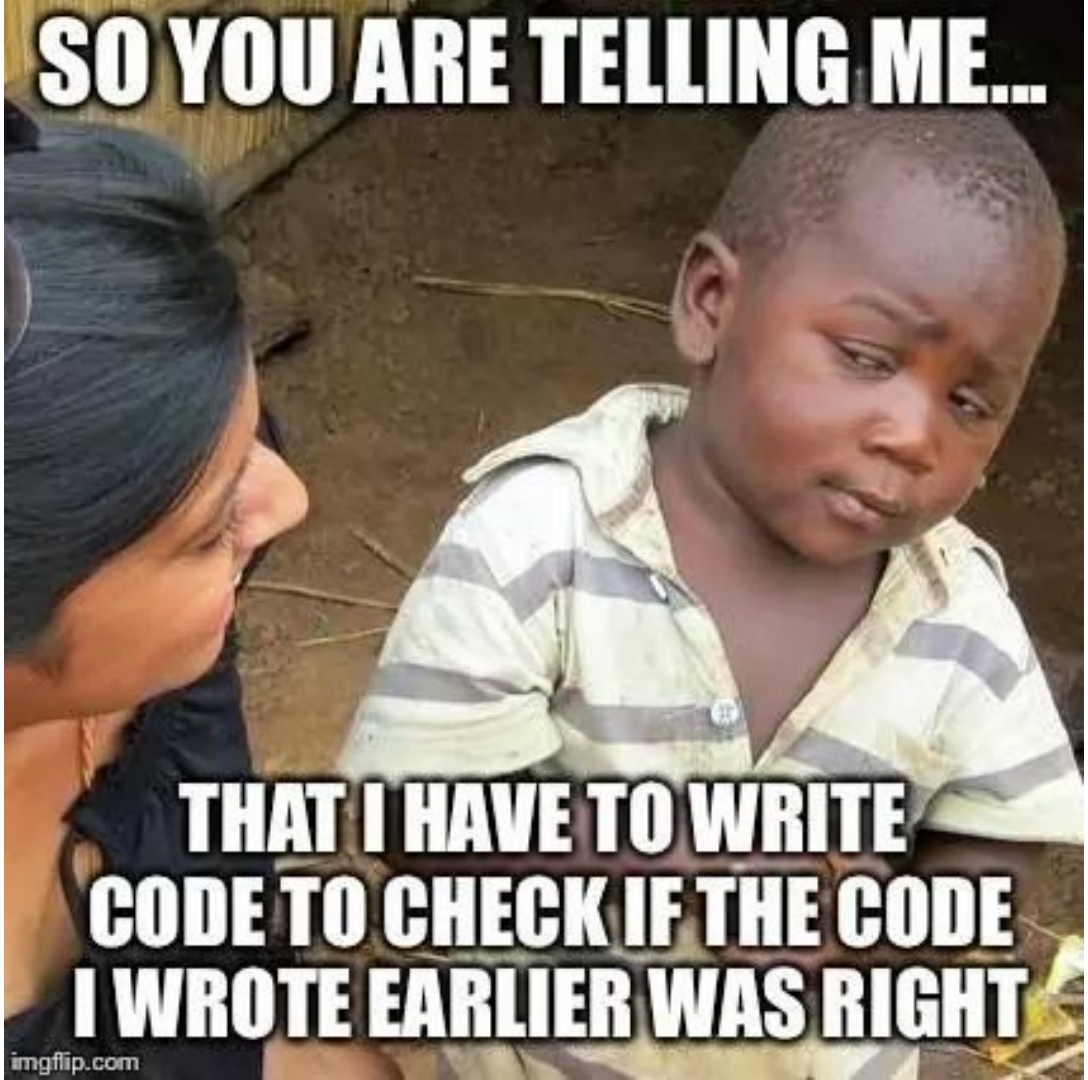
### LIMITER LE RISQUE

Valider le comportement du système dans des scénarios variés pour limiter et identifier les incidents au + tôt

### ASSURER LA COHÉRENCE AU BESOIN

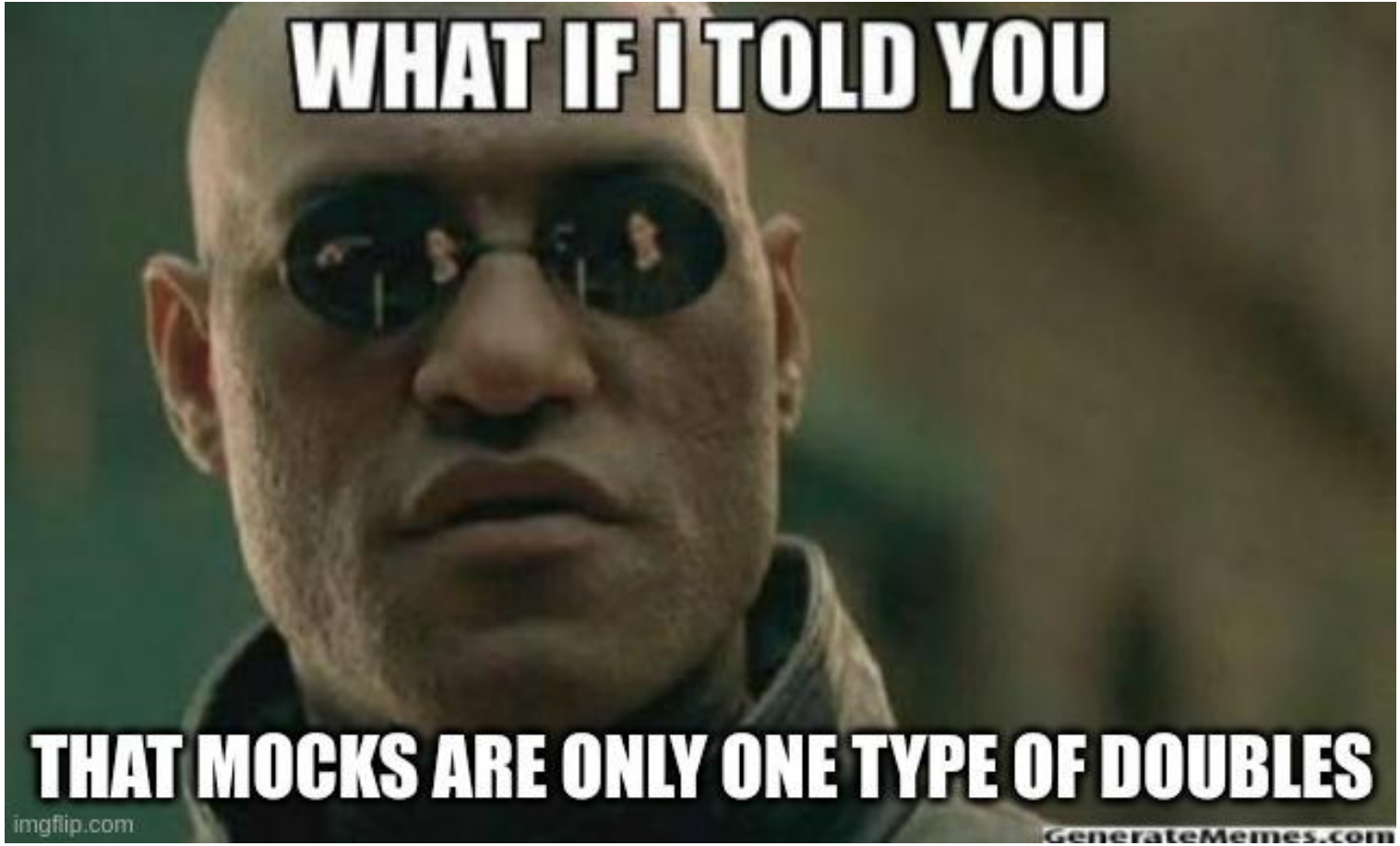
Valider les scénarios correspondants au besoin utilisateur nous assure une cohérence technique / fonctionnelle

***** : *selon le type de produit que vous construisez, vous n'avez pas les mêmes enjeux*


SO YOU ARE TELLING ME...
THAT I HAVE TO WRITE
CODE TO CHECK IF THE CODE
I WROTE EARLIER WAS RIGHT
imgflip.com

# DIFFÉRENTES CATÉGORIES

# LES DOUBLURES DE TESTS C'EST COMME AU CINÉMA


WHAT IF I TOLD YOU
THAT MOCKS ARE ONLY ONE TYPE OF DOUBLES
imgflip.com
GenerateMemes.com

# LES DIFFÉRENTES CATÉGORIES DE TESTS

## DOUBLURES DE TESTS

On parle souvent et abusivement de “mock” pour évoquer les doublures de tests.
Mais finalement, les doublures sont aussi constituées de … ?

# LES DIFFÉRENTES CATÉGORIES DE TESTS

## DOUBLURES DE TESTS - DUMMIES

Dummies

```ts
interface Logger {
  log(message: string): void;
}

class OrderService {
  constructor(private logger: Logger) {}

  placeOrder(orderId: number): void {
    // ... make things
    this.logger.log(`Order ${orderId} placed successfully.`);
  }
}

class DummyLogger implements Logger {
  log(message: string): void {}
}

const dummyLogger = new DummyLogger();
const orderService = new OrderService(dummyLogger);
```

Une implémentation de classe dont on se fiche des réponses données.
Une méthode d'un Dummy retourne généralement void.

# LES DIFFÉRENTES CATÉGORIES DE TESTS

## DOUBLURES DE TESTS - STUB

Stub

```
interface PaymentGateway {
  processPayment(amount: number): boolean;
}

class OrderService {
  constructor(private paymentGateway: PaymentGateway) {}

  placeOrder(amount: number): string {
    const paymentSuccess = this.paymentGateway.processPayment(amount);

    // ... make things
  }
}

class PaymentGatewayStub implements PaymentGateway {
  processPayment(amount: number): boolean {
    return true;
  }
}
```

Une implémentation de classe dont les méthodes vont retourner exactement ce que j'attends dans le test.

# LES DIFFÉRENTES CATÉGORIES DE TESTS

## DOUBLURES DE TESTS - FAKE

Fake

```
class FakeDatabase implements Database {
  private dataStore: { id: number; name: string }[] = [];

  save(data: { id: number; name: string }): void {
    this.dataStore.push(data);
  }

  getById(id: number): { id: number; name: string } | null {
    return this.dataStore.find((entry) => entry.id === id) || null;
  }
}
```

Une implémentation de classe dérivée de votre classe concrète.
Par exemple le InMemoryDB à la place du RealDB.

# LES DIFFÉRENTES CATÉGORIES DE TESTS

## DOUBLURES DE TESTS - SPY

Spy

```
class LoggerSpy implements Logger {
  public calls: string[] = []; // Enregistre les messages logués

  log(message: string): void {
    this.calls.push(message); // Enregistre l'appel
  }
}
```

Comme son nom l'indique, on a ici affaire à un espion qui va enregistrer les actions sur la classe que l'on teste, enregistrer le nombre de fois où on l'appelle, par exemple.

# LES DIFFÉRENTES CATÉGORIES DE TESTS

## DOUBLURES DE TESTS - MOCK

Mock

```
class EmailServiceMock implements EmailService {
  public sendEmailCalls: { to: string; subject: string; body: string }[] = [];

  sendEmail(to: string, subject: string, body: string): void {
    this.sendEmailCalls.push({ to, subject, body }); // Enregistre l'appel
  }

  // Méthode pour vérifier les appels
  wasCalledWith(to: string, subject: string, body: string): boolean {
    return this.sendEmailCalls.some(
      (call) => call.to === to && call.subject === subject && call.body === body
    );
  }

  get callCount(): number {
    return this.sendEmailCalls.length;
  }
}
```

Une super classe, ultra complète, qui va à la fois être un Stub et un Spy.
Un mock va vérifier qu'un process' a bien été réalisé, par exemple: telle fonction est appelée X fois, sinon je lève une exception.

# LES DIFFÉRENTES CATÉGORIES DE TESTS

## DEUX APPROCHES - CHICAGO OU LONDRES ?

**Detroit - Classiques - Inside-out**

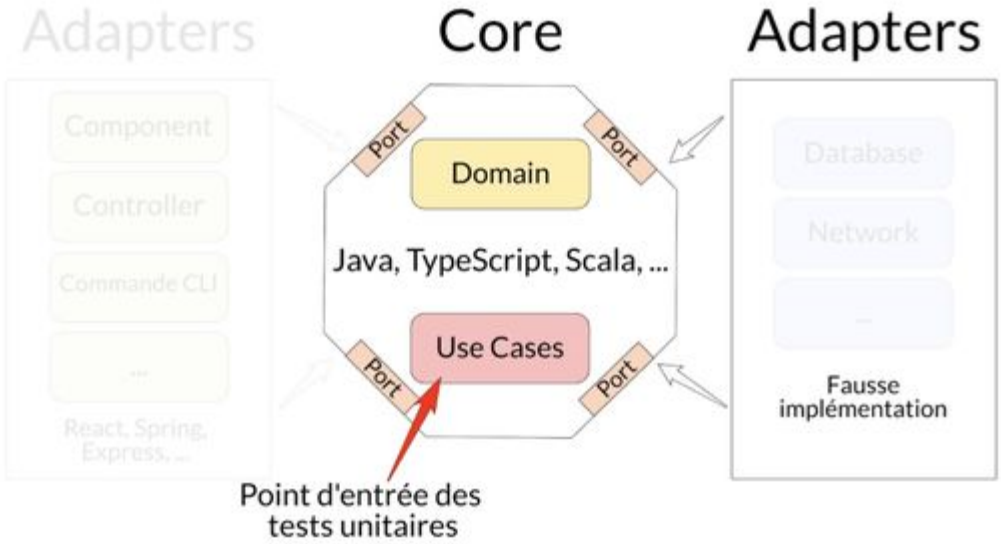
Se suffit en majorité des stubs
Le design émerge pendant le refactoring
Moins d'over-engineering
On part du coeur pour aller vers l'extérieur
Adapté aux débutants, de part l'émergence
Popularisé par Kent Beck

**London - Mockistes - Outside-in**

Utilisation de mocks
Partir de l'API pour aller vers le coeur
+ d'OE potentiel avec le "pré-senti"
On commence avec de l'Acceptance
Bien dans un contexte DDD
Nécessite des connaissances en design

# LES DIFFÉRENTES CATÉGORIES DE TESTS

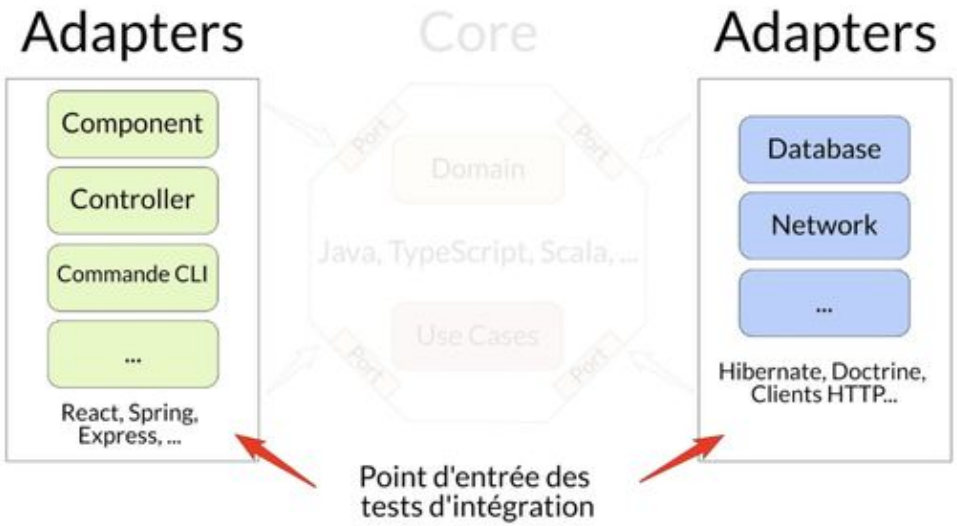
## TESTS UNITAIRES




Un TU doit **tester unitairement un comportement**/une intention utilisateur et non une implémentation possible. Le **nom du TU** doit également le refléter, ce qui permet d'avoir une **documentation fonctionnelle**.

# LES DIFFÉRENTES CATÉGORIES DE TESTS

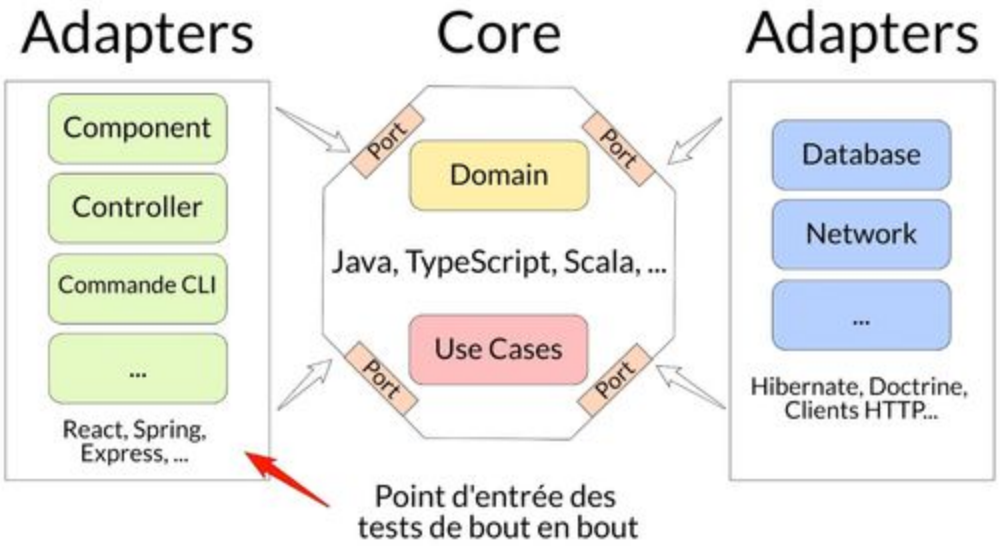
## TESTS D'INTÉGRATION




Vise à **tester l'intégration d'un système tiers** (lib, frameworks...) pour **vérifier son comportement lors d'appels de use-cases**.
Souvent **très petit**, on ne se préoccupe pas du métier ici : vérification de code HTTP / structure d'objet en retour, un happy + wrong path maximum.

# LES DIFFÉRENTES CATÉGORIES DE TESTS

## TESTS E2E



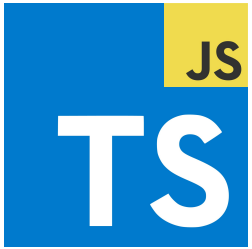

L'objectif : tester que **l'ensemble du système fonctionne correctement avec les différentes couches connectées**.

# ET LES AUTRES ?

# LES DIFFÉRENTES CATÉGORIES DE TESTS

## PROPERTY BASED TESTING


**junit-quickcheck**




**fast-check**

**QuickCheck**


**RapidCheck**

S'aider de **règles mathématiques** pour tester de **multiples exécutions** avec des **inputs aléatoires** afin de **vérifier une sortie et un comportement**.
Utile si on est face à un **algorithme complexe à implémenter, mais simple à tester**.

# LES DIFFÉRENTES CATÉGORIES DE TESTS

## PROPERTY BASED TESTING - CAS METIER

Une fonction **hashPassword** génère un hachage sécurisé pour un mot de passe donné.
Elle doit respecter les propriétés suivantes :

- Le hachage est **toujours une chaîne non vide**.
- Le hachage est **déterministe pour le même mot de passe et le même sel**.
- Le hachage **change si le mot de passe ou le sel change**.

# LES DIFFÉRENTES CATÉGORIES DE TESTS

## PROPERTY BASED TESTING - CAS METIER

```
import * as fc from "fast-check";

describe("hashPassword - Property-Based Testing", () => {
  it("should always return a non-empty string", () => {
    fc.assert(
      fc.property(
        fc.record({
          password: fc.string({ minLength: 1 }),
          salt: fc.string({ minLength: 1 }),
        }),
        ({ password, salt }) => {
          const hash = hashPassword({ password, salt });
          expect(hash).toBeDefined();
          expect(hash).toBeTruthy(); // La chaîne n'est pas
vide     }
      )
    );
  });
});
```

# LES DIFFÉRENTES CATÉGORIES DE TESTS

## PROPERTY BASED TESTING - CAS METIER

```
import * as fc from "fast-check";

describe("hashPassword - Property-Based Testing", () => {
  it("should be deterministic for the same password and salt", () => {
    fc.assert(
      fc.property(
        fc.record({
          password: fc.string({ minLength: 1 }),
          salt: fc.string({ minLength: 1 }),
        }),
        ({ password, salt }) => {
          const hash1 = hashPassword({ password, salt });
          const hash2 = hashPassword({ password, salt });
          expect(hash1).toBe(hash2); // Le même mot de passe et sel génèrent le même
hachage }
      )
    );
  });
});
```

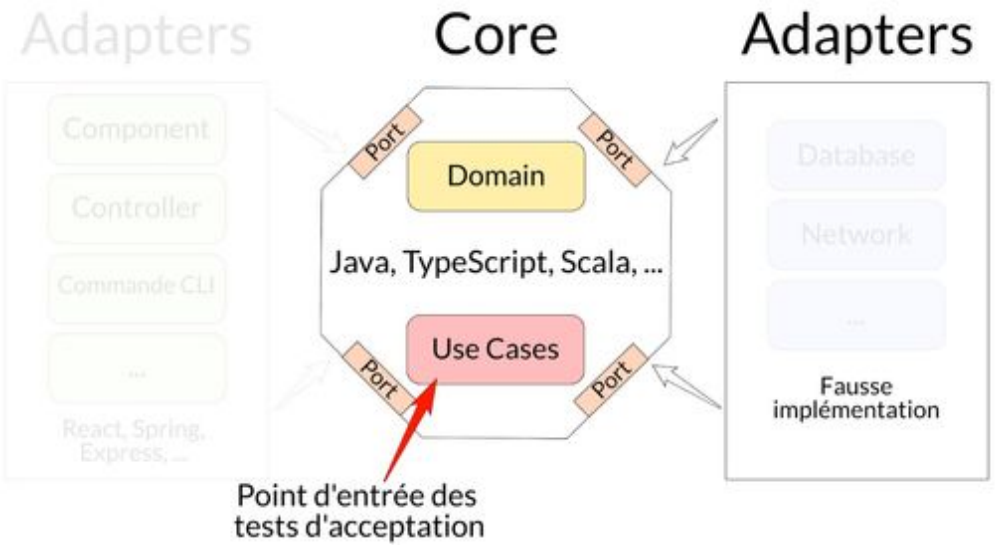
# LES DIFFÉRENTES CATÉGORIES DE TESTS

## PROPERTY BASED TESTING - CAS METIER

```
import * as fc from "fast-check";

describe("hashPassword - Property-Based Testing", () => {
  it("should produce different hashes for different passwords or salts", () => {
    fc.assert(
      fc.property(
        fc.record({
          password: fc.string({ minLength: 1 }),
          salt: fc.string({ minLength: 1 }),
        }),
        fc.record({
          password: fc.string({ minLength: 1 }),
          salt: fc.string({ minLength: 1 }),
        }),
        (input1, input2) => {
          if (input1.password !== input2.password || input1.salt !== input2.salt) {
            const hash1 = hashPassword(input1);
            const hash2 = hashPassword(input2);
            expect(hash1).not.toBe(hash2); // Les hachages diffèrent si mot de passe ou sel
diffère   }
        }
      )
    );
  });
});
```

# LES DIFFÉRENTES CATÉGORIES DE TESTS

## ATDD




Similaire à la démarche de tests unitaires, mais se focalise sur le “quoi”, les spécifications, **les discussions.**
Démarche d’écriture en langage naturel (Gherkin : Given When Then ...)

# LES DIFFÉRENTES CATÉGORIES DE TESTS

## ATDD - Exemple

```
Feature: Withdraw cash from ATM

  Scenario: Successful withdrawal
    Given an account with a balance of 100€
    When the user withdraws 50€
    Then the withdrawal is accepted
    And the account balance should be 50€

  Scenario: Withdrawal refused due to insufficient balance
    Given an account with a balance of 30€
    When the user withdraws 50€
    Then the withdrawal is refused
    And the account balance should be 30€
```

Gherkin

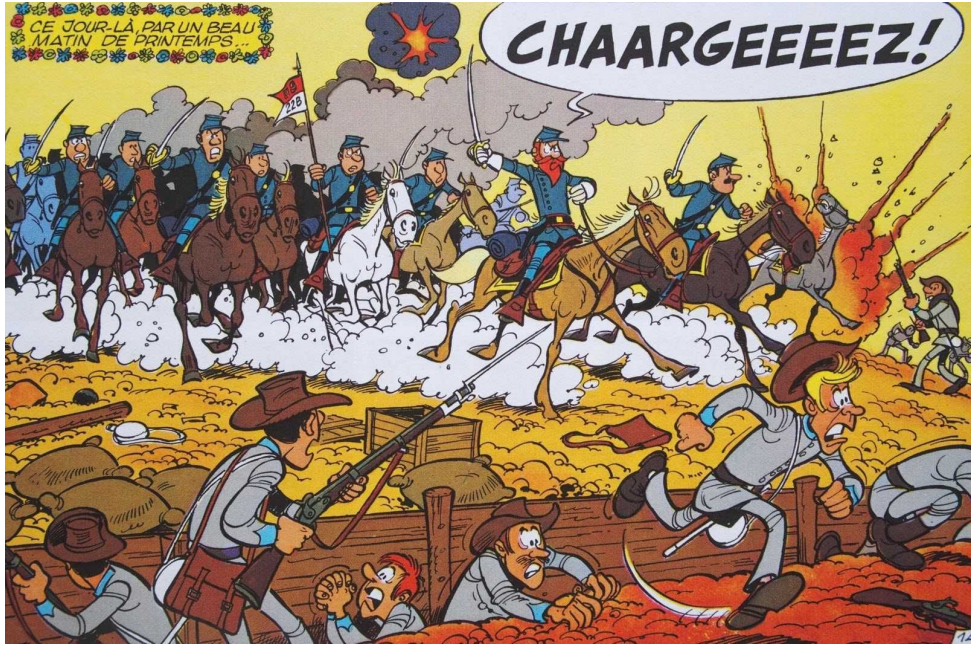
➡️

```
import { Given, When, Then } from "@cucumber/cucumber";
import assert from "assert";

class Account {
  constructor(private balance: number) {}

  withdraw(amount: number): { accepted: boolean } {
    if (amount > this.balance) {
      return { accepted: false };
    }
    this.balance -= amount;
    return { accepted: true };
  }

  getBalance() {
    return this.balance;
  }
}

let account: Account;
let result: { accepted: boolean };

Given("an account with a balance of {int}€", function (balance: number) {
  account = new Account(balance);
});

When("the user withdraws {int}€", function (amount: number) {
  result = account.withdraw(amount);
});

Then("the withdrawal is accepted", function () {
  assert.equal(result.accepted, true);
});

Then("the withdrawal is refused", function () {
  assert.equal(result.accepted, false);
});

Then("the account balance should be {int}€", function (expected: number) {
  assert.equal(account.getBalance(), expected);
};
```

Cucumber.js

# LES DIFFÉRENTES CATÉGORIES DE TESTS

## TESTS DE CHARGE




Simuler le comportement d'utilisateurs à travers des scénarios afin d'évaluer la gestion de charge d'une plateforme.

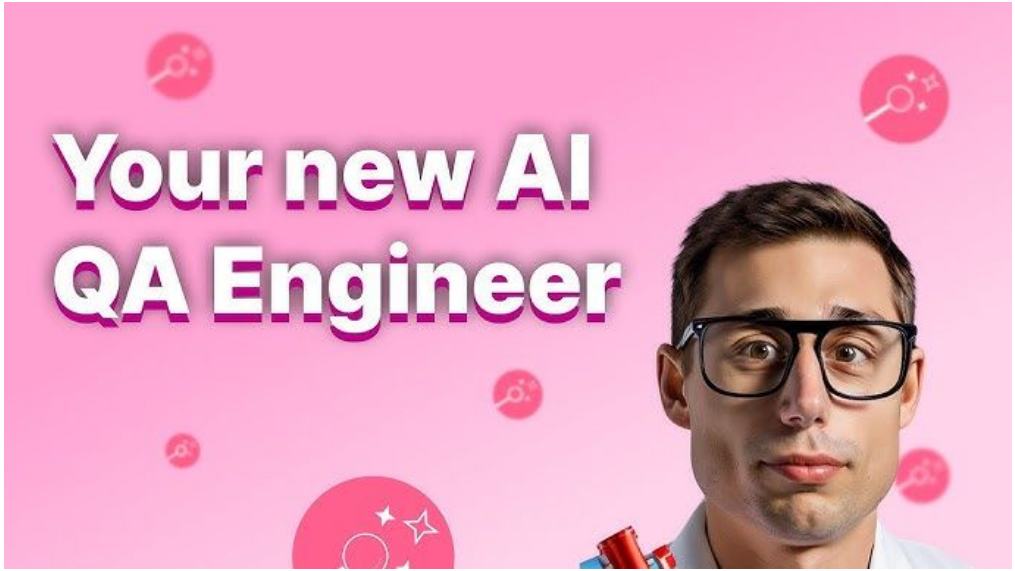
# LES DIFFÉRENTES CATÉGORIES DE TESTS

## TESTS D'INTRUSION & RED TEAM


Trouver les failles dans un système informatique à partir de connaissances +/- limitées, afin d'améliorer la sécurité générale du SI.

# LES DIFFÉRENTES CATÉGORIES DE TESTS

## TESTING AUTOMATISÉ... POWERED BY AI




Un public non technique peut rédiger des scénarios de tests facilement, à l'aide de l'IA.
"Clique sur le bouton de connexion", "Rempli le champ d'email avec XXX"

# STRATÉGIE DE TESTING

# STRATÉGIE DE TESTING

## TROPHY & PYRAMID

THE FOUR TYPES OF TESTS

### End to End

A helper robot that behaves like a user to click around the app and verify that it functions correctly.

Sometimes called "functional testing" or e2e.

### Integration

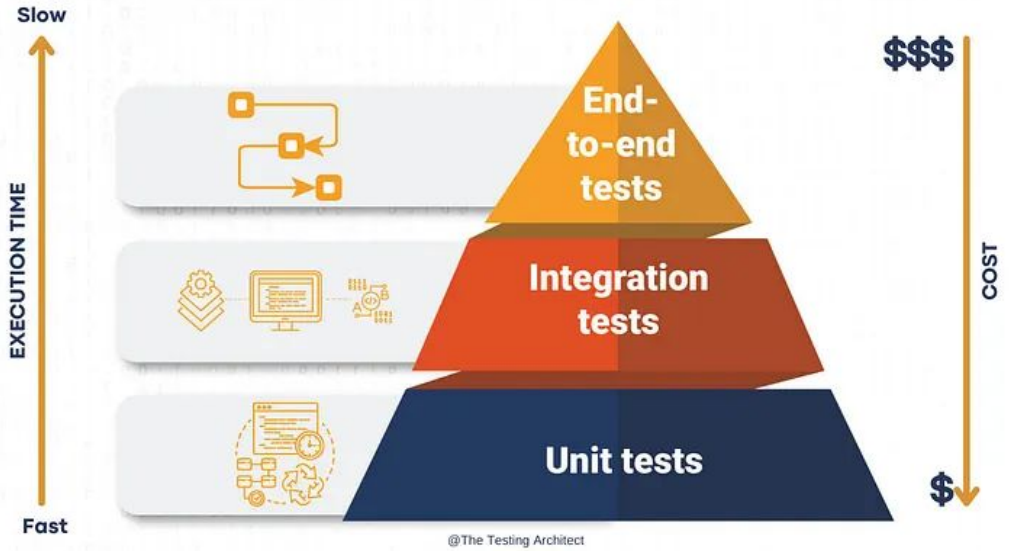
Verify that several units work together in harmony.

### Unit

Verify that individual, isolated parts work as expected.

### Static

Catch typos and type errors as you write the code.

*Ces stratégies sont idéales dans des cas de construction d'applications...*

# STRATÉGIE DE TESTING

## ET LE TDD ALORS ?

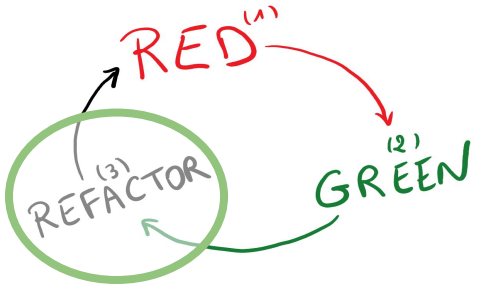
Le TDD est une approche de **code plutôt que de test**

Le TDD **ne sert pas à faire de la non-régression** (c'est un side-effect)

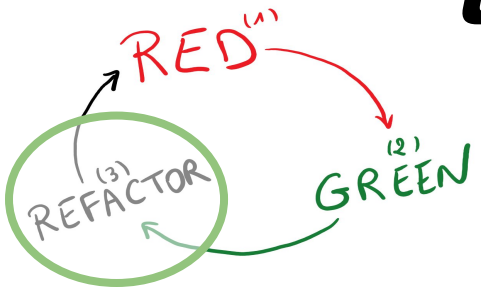
Le TDD **n'amène pas forcément de la qualité**

**Tous les scénarios ne sont pas couverts** grâce au TDD, seulement le business

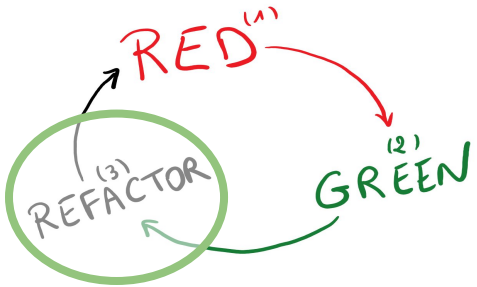
Le TDD permet d'**être plus productif si maîtrisé**



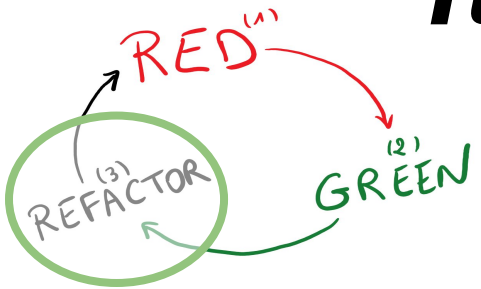
# *Tu n'écriras pas plus de code de test qu'il ne suffit pour que ce dernier échoue.*



# *Tu n'écriras pas plus de code qu'il ne suffit pour faire passer ce test au vert.*



# Tu n'écriras point de code de prod', si ce n'est que pour faire passer un test.



# STRATÉGIE DE TESTING

AUTOMATISATION

| PRE-COMMIT | PRE-PUSH / MR | INTÉGRATION BRANCHE SENSIBLE | INTÉGRATION ENVIRONNEMENT PREPROD |
|---|---|---|---|
| **Lint + build Git hook** | **Tests unitaires** **(Tests intégration)** | **Tests d'intégration (Tests E2E)** | **Tests E2E** |

Les bonnes pratiques générales :
paralléliser / stabiliser les environnements de tests / penser à la DX

# À VOUS DE JOUER

# RESSOURCES

- https://blog.octo.com/un-test-peut-en-cacher-un-autre-un-peu-de-theorie
- https://github.com/dubzzz/fast-check