Tests et Performance

Plan et objectifs

- Les requis
- L’importance des tests
- Tests manuel et tests automatisés
- Types de tests
  - tests unitaires
  - tests d’intégration
  - tests de bout en bout
  - tests d’acceptation
  - tests de fumée
  - tests système
  - tests de performance
- Outils de tests

Les requis

Avant de se lancer dans la construction d'une application web (ou tout autre logiciel), il faut bien définir ce qui est attendu - les requis.

Selon le type de projet et les ressources à disposition, les requis peuvent prendre plusieurs formes:

- un document décrivant l'ensemble des fonctionnalités
- des maquettes de l'interface utilisateur et le flot de l'application
- un ensemble de scénarios d'utilisation
- la définition d'un produit minimum viable
- etc

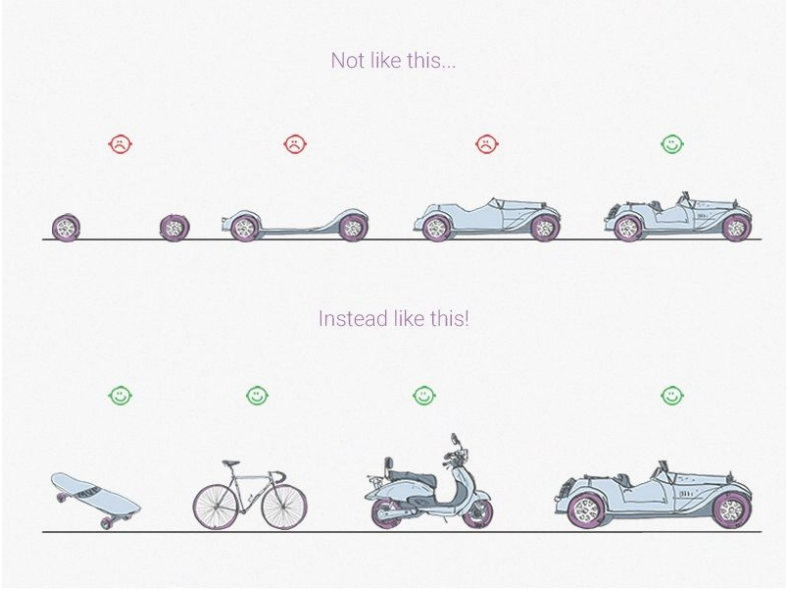
Selon la méthode de développement, les requis peuvent être définis en amont ou par itération.

Produit minimum viable

Le **produit minimum viable** est le produit le plus simple qui permet de combler le ou les besoins des utilisateurs.

On minimise les efforts de développement afin de maximiser les retours des utilisateurs.

Il est utilisé pour valider des hypothèses et obtenir des retours utilisateurs afin de mieux cerner les besoins et le marché.



Requis fonctionnels et non-fonctionnels

Il faut faire la distinction entre les requis fonctionnels et les requis non-fonctionnels, car différents types de tests permettront de les valider.

Requis fonctionnels

Un requis fonctionnel décrit **comment l'application doit se comporter**. Par exemple: l'application doit permettre d'ajouter un produit.

Requis non-fonctionnels

Un requis non-fonctionnel décrit une **propriété générale du système**. Par exemple: l'application doit avoir un temps de réponse sous les 2 secondes.

Pourquoi tester?

L’objectif premier est de vérifier que l’application se comporte comme prévu (valider les requis).

Un test qui échoue permet d’identifier un problème et de valider la correction.

Une suite de tests qui s'exécute avec succès ne prouve pas l’absence de bogues.

Les tests automatisés diminuent les risques liés à la refactorisation du code, car exécutés de façon régulière, ces tests permettent de déterminer si des modifications ont altéré le comportement de l’application de façon inattendue (régression).

Tests manuels ou automatisés

Tests manuels

Tests exécutés manuellement par un individu. Lents, mais permet plus de nuances et de flexibilité dans l'exécution et l’évaluation du résultat.

Tests automatisés

Suite de tests exécutés automatiquement. Rapides, mais demande des compétences en programmation, une infrastructure et un effort de maintenance qui peut ralentir le développement. On préfère les tests automatisés aux tests manuels.

Boîte blanche, boîte noire

Tests boîte blanche

Le testeur connaît le fonctionnement interne de l'application à tester. Ces tests sont écrits par des développeurs et permettent de valider le fonctionnement de façon plus granulaire. Par contre, les tests sont souvent fortement couplés à l'implémentation, ce qui implique des changements fréquents lorsque le code change.

Tests boîte noire

Le testeur ne connaît pas le fonctionnement interne. Il se base sur les requis pour définir les scénarios de tests.

Anatomie d’un bon scénario de test

- Nom ou id - permet de référer au scénario de façon précise
- Prérequis et mise en place - ensemble des conditions qui doivent être vraie au départ
- Données de test - les données à utiliser pour le test
- Séquence - les étapes à suivre pour exécuter le test
- Résultats attendus - les conditions de validation (binaire: passe ou échoue)
- Nettoyage - s’assurer que l’exécution du test n’affectera pas les tests subséquents

Types de tests

Par ordre de taille des composants testés:

- tests unitaires
- tests d’intégration
- tests de bout en bout
- tests d’acceptation
- tests de fumée
- tests de performance
- tests système

Tests unitaires

Type: tests boîtes blanches automatisés

Les caractéristiques des tests unitaires:

- appliqués sur une unité logique du programme en isolation (e.g. une classe)
- écrits par l'auteur du code testé
- automatisés (fait partie de pipeline d'intégration continue)
- les dépendances sont simulées à l'aide de *mocks* afin de contrôler de façon précise l'état du composant à tester

Tests unitaires - Bonnes pratiques

Quelques bonnes pratiques pour les tests unitaires:

- Structure:
  - mise en place
  - test
  - validation
  - nettoyage
- indépendant de l’ordre d’exécution
- déterministe
- valide un élément précis
- très rapide
- code maintenu comme du code de production

Avantages des tests unitaires

Les avantages des tests unitaires sont nombreux:

- permettent d'identifier des bogues rapidement
- donnent un feedback rapide au programmeur
- permettent de valider des changements rapidement et d'éviter des régressions
- documentent le code
- formalisent les requis
- outils précieux pour la validation d'une correction de bogue
- permette de tester des scénarios d'erreur plus facilement

Les tests unitaires ont aussi des désavantages qui sont principalement

- écrire des bons tests demandent du temps
- les tests doivent être maintenus

Les tests unitaires et l’évolution d’une base de code

Sans tests unitaires, il est risqué de faire de la refactorisation de code.

Sans refactorisation, il devient difficile de garder une base de code propre (accumulation de dette technique).

Sans base de code propre, il devient de plus en plus difficile de faire évoluer une application.

Une application qui n’évolue pas…meurt.

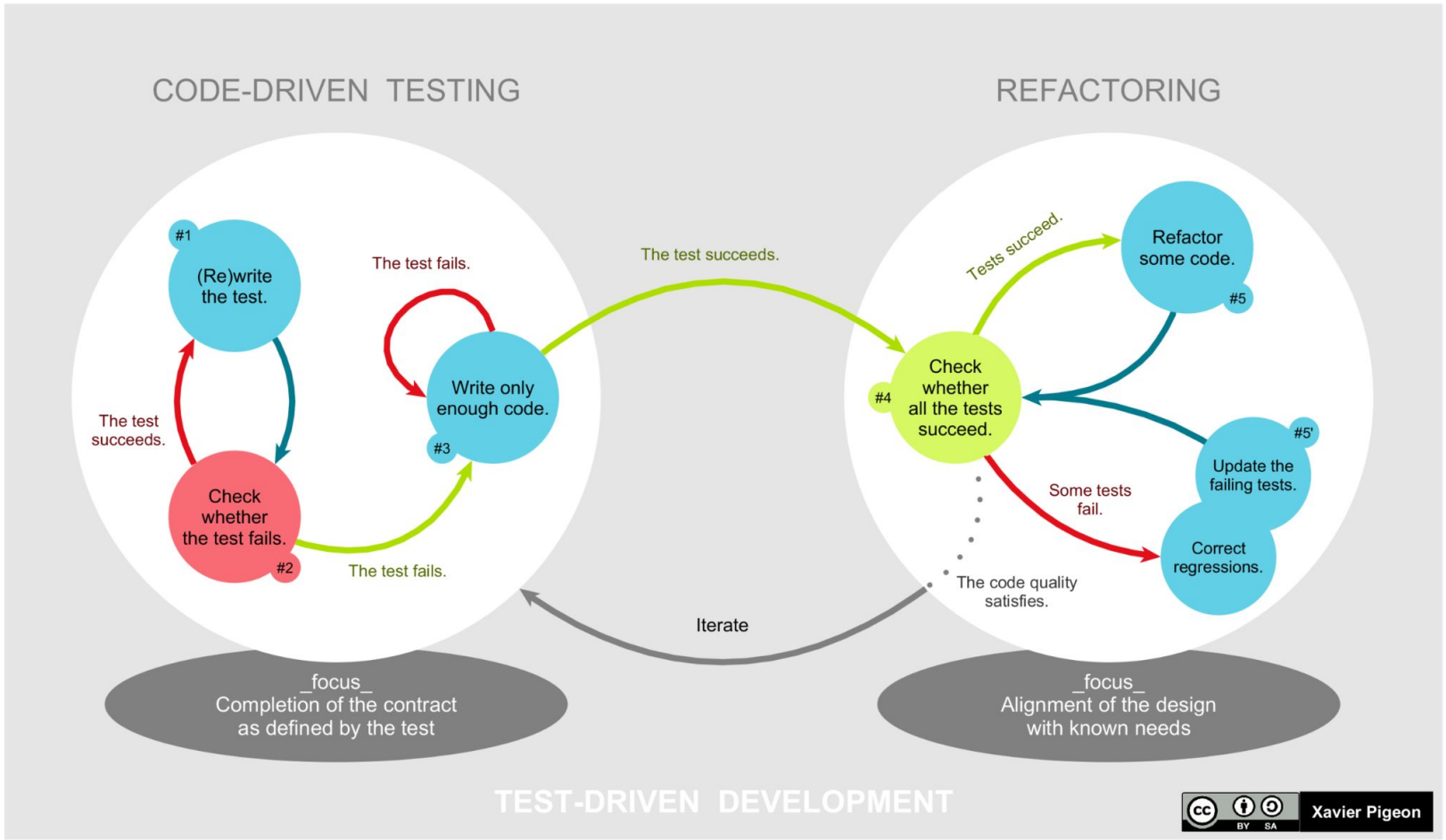
TDD ou ne pas TDD

Le développement piloté par les tests (TDD, Test Driven Development) suggère d’écrire un test unitaire qui échoue avant d’écrire le code d’une partie de la fonctionnalité à implémenter (i.e. ne jamais écrire de code de production sans un test qui échoue).

On répète ensuite le même processus jusqu’à ce que la fonctionnalité complète soit implémentée (et testée).

L’objectif est double:

- le code de la fonctionnalité est le plus simple possible (pour faire passer les tests)
- le code est testé


CODE-DRIVEN TESTING
REFACTORING
#1
(Re)write the test.
The test fails.
The test succeeds.
Write only enough code.
#3
The test succeeds.
Check whether the test fails.
#2
The test fails.
Tests succeed.
Refactor some code.
#5
#4
Check whether all the tests succeed.
#5'
Update the failing tests.
Some tests fail.
Correct regressions.
The code quality satisfies.
Iterate
focus
Completion of the contract as defined by the test
focus
Alignment of the design with known needs
TEST-DRIVEN DEVELOPMENT
Xavier Pigeon

Le TDD

Utiliser l'approche du TDD n'est pas toujours intuitive et requiert de la discipline.

Elle peut parfois amener le programmeur dans un design sous-optimal étant donné qu'il se concentre sur l'implémentation morceau par morceau.

Il est parfois utile de prendre un peu de recul et d'explorer le code et les options de design avant d'investir dans les tests (qui figent un peu le code).

Par contre, il est important d'écrire le code et les tests conjointement et de ne pas les remettre à plus tard.

Une fonctionnalité ne devrait pas être complétée (intégrée) si le code ne contient pas les tests unitaires. Les tests devraient faire partie des critères d'acceptation d'une tâche.

Peu importe si vous utilisez le TDD ou non, vous devriez toujours avoir vu un test échoué au moins une fois.

Test basé sur les propriétés - Property Based Testing

Stratégie d'écriture de tests unitaires qui contraste avec la stratégie traditionnelle (tester des exemples et valider le résultat pour un cas bien précis).

L'objectif du Property Based Testing est de valider des *invariants*.

On détermine un *invariant* (une propriété qui devrait être toujours vraie). Par exemple: le nombre de produits retournés doit être <= au paramètre `limit`.

On utilise (ou construit) un générateur de données valides (déterministe ou aléatoire). Par exemple: les nombres de -100 à 100.

Le test consiste à exécuter le code à tester avec l'ensemble des données générées et de valider l'invariant pour chacun.

Voir https://jqwik.net/ (bibliothèque pour Java).

Tests unitaires - Angular

Angular offre des outils pour écrire des tests unitaires pour les composants et les services.

Angular utilise:

- jasmine - https://jasmine.github.io/ - bibliothèque style *piloté par le comportement*
- karma - https://karma-runner.github.io/ - environnement d'éxécution (dans navigateur web)

Documentation officielle: https://angular.io/guide/testing

Tests unitaires - Spring Boot

Spring Boot offre aussi des outils pour développer des tests unitaires en utilisant principalement JUnit (https://junit.org/) et Mockito (https://site.mockito.org/) pour les *mocks*.

Exemple et documentation: https://spring.io/guides/gs/testing-web/.

Tests d’intégrations

Type: tests boîtes blanches automatisés

Les tests d’intégrations combinent plusieurs composants qui sont ensuite testés ensembles.

Dans le contexte d’une application web, on ajoute des tests d’intégration pour valider les *endpoints* d’une API par exemple.

Dans ce cas, les tests d’intégration démarrent le conteneur web, mais ne *mock* pas les dépendances (en général). Ils utilisent plutôt des instances dédiés de ces dépendances (e.g. bases de données) qui s'exécutent **localement**. Celles-ci sont nettoyées entre chaque batterie de tests (ou chaque test!).

Le test valide les scénarios d’utilisation à plus haut niveau et permet de s’assurer que les différents composants communiquent bien entre eux.

Tests d’intégration

Ces tests sont écrits et exécutés par les développeurs et par les outils d’intégration continue.

Concevoir et maintenir des tests d’intégration est plus coûteux, mais permet d’automatiser une batterie de tests qui reflète mieux la réalité.

Le temps d’exécution des tests d’intégration doit être relativement rapide (mais il sera plus lent qu’un test unitaire).

Le plus difficile est de bien gérer les dépendances (émulateurs, environnement de test dédié), de s’assurer que l’état des dépendances est nettoyé entre chaque test (peu importe si le test passe ou échoue) et que le test est stable.

Test d’intégrations avec Spring Boot

Spring Boot permet aussi d’écrire des tests d’intégration qui démarrent le conteneur web. Le test utilise ensuite le **`TestRestTemplate`** pour effectuer les requêtes HTTP.

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class HttpRequestTest {
      @LocalServerPort
      private int port;

      @Autowired
      private TestRestTemplate restTemplate;

      @Test
      public void testABC() throws Exception {
      }
 }
```

Conseil: Tests et correction de bogues

Lorsqu'un bogue est découvert, il est souvent utile de créer un test d'intégration et/ou un test unitaire pour le reproduire.

Une fois le ou les tests écrits, il devient facile de valider la correction.

Tests de bout en bout

Type: tests boîtes noires automatisés

Aussi connu sous le nom de *end to end testing*, ces tests automatisés valident le système dans son ensemble (frontend et backend).

Dans le cas d'une application web, on utilisera souvent un environnement local complet et dédié pour ce type de test.

Ces tests sont écrits par les développeurs et exécutés par les outils d'intégration continue.

Les tests simulent le comportement des utilisateurs et valide l'application du point de vue de l'utilisateur en interagissant directement avec la page web.

Tests de bout en bout

Ces tests utilisent des outils spécifiques (Selenium, Cypress) pour exécuter des scripts de tests.

Ils sont encore plus coûteux que les tests d'intégration à développer, mais ajoutent un niveau supplémentaire de réalisme au scénario.

Ces tests peuvent aussi être très coûteux à faire évoluer, car ils sont fortement liés à la navigation et au contenu des pages de l'application.

Ces tests risquent d'être instable et leur temps d'exécution est non négligeable.

https://www.selenium.dev/

https://docs.cypress.io/

Tests d’acceptation

Type: tests boîtes noires manuels

Les tests d’acceptation sont des tests manuels dont l’objectif est de valider du point de vue de l’utilisateur si oui ou non l’application est conforme aux spécifications (requis, contrat, etc).

Ces tests peuvent être exécutés par une équipe de testeurs, des clients, des gestionnaires de produit, des utilisateurs, etc.

Tests de fumée

Type: tests boîtes noires automatisés

Les tests de fumée s'exécutent au niveau du système dans son ensemble, mais ne comportent qu'un petit nombre de scénarios de test.

L'objectif est de valider la stabilité des éléments critiques de système (e.g. login) après la création ou le déploiement d'une nouvelle version.

Tests de performance

Type: tests boîtes noires automatisés

On utilise les tests de performance pour mesurer et valider la capacité d’une application web et de découvrir ses limites.

Ces tests mesurent généralement le temps de réponse (en ms), le débit (nombre de requêtes par seconde) et le taux d’erreur (%). Il n’y a pas ou peu de validation.

Ces tests permettent d’identifier les limites actuelles du système et de valider que des changements de code ou de configuration (matériel ou logiciel) n'impactent pas négativement les performances de l’application.

Tests de performance

Ces tests sont exécutés contre un environnement dédié similaire à l'environnement de production.

Ce sont des tests de longue durée.

Ces tests utilisent des outils dédiés (plus ou moins complexes) pour décrire les scénarios de test et déterminer leurs paramètres (durée, nombre de threads, etc).

https://jmeter.apache.org/

https://github.com/wg/wrk

Tests système

Type: tests boîtes noires automatisés ou manuels

Les tests systèmes ont pour objectif de valider la stabilité et la résilience d'un système.

L’objectif est de déstabiliser le système (augmentation de la latence réseau, panne d'un composant, etc) et vérifier que le système peut récupérer de lui-même.

Les tests peuvent être manuels (avec un ordre défini) ou aléatoires (chaos engineering).

Les problèmes rencontrés sont parfois très difficiles à reproduire et peuvent demander beaucoup de temps et d’effort côté développement pour les régler.

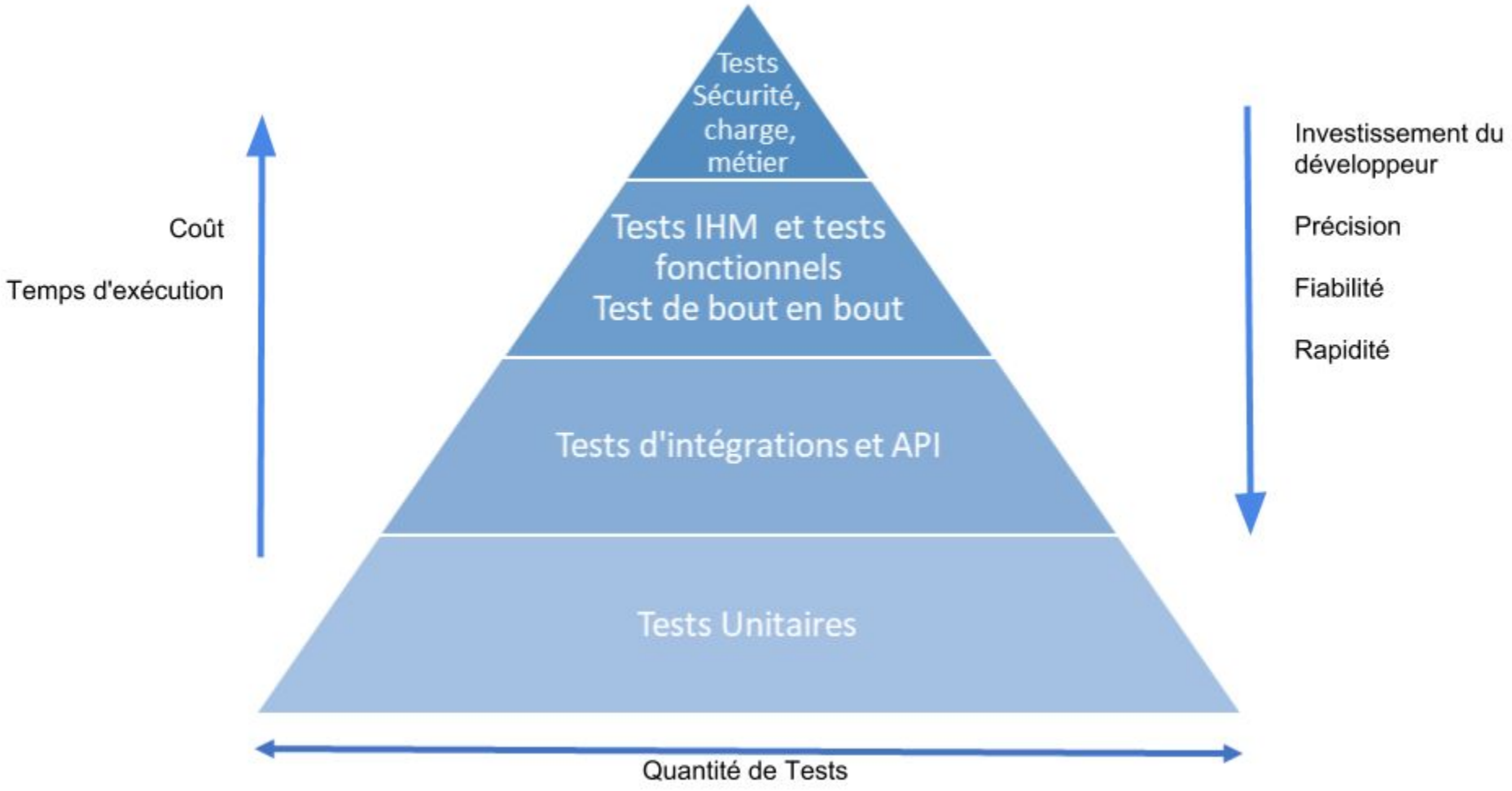
Tests: la réalité

Malgré le fait qu’on aimerait une couverture complète, en pratique ce n’est pas possible.

Chaque type de test se concentre sur l'identification de bogues de différents types et qui surviennent dans des contextes différents.

Il convient de choisir de tester les éléments critiques en premier en fonction de l’ampleur du projet et des ressources disponibles.

La pyramide des tests



https://jaayap.github.io/Unity_Best_Practices/Fr/Unit_Test_And_TDD.html

Tests en production?

Dans l'idéal: non.

Mais parfois une équipe n'a pas les ressources pour créer et maintenir tous les environnements de test pour tous les types de test.

Si le processus de développement est rigoureux et que l'environnement de production est bien surveillé et stable, il est possible d'utiliser un environnement *canari* pour déployer une nouvelle version sur une fraction du trafic de l'application et ainsi détecter les anomalies rapidement.

Il est aussi possible d'ajouter une configuration qui active une modification sur un sous-ensemble des utilisateurs.

Dans tous les cas, il faut aussi être en mesure de réagir rapidement avec un retour en arrière (rollback).

Outils de performance - Visual VM

Dans l'écosystème Java, il existe des outils intéressants qui permettent d'enquêter sur les problèmes de performance.

Il existe de nombreux outils de profilage commerciaux de très bonne qualité (JProfiler, YourKit, etc).

Visual VM est un outil gratuit de visualisation de la JVM et un outil de profilage simple.

Il vous permet d'inspecter la mémoire de la JVM, d'échantillonner le CPU et la mémoire, de générer un *thread dump* (utile pour les problèmes de contention) ou un *heap dump* (utile pour les problèmes de fuite de mémoire).