# Introduction au Génie Logiciel

## Séance 8 : Tests

L. Laversa
laversa@irif.fr

Université Paris Cité

25 mars 2025

. Remerciements à J. Lefebvre

# Pourquoi des tests ?

- Détecter les bugs le plus tôt possible
- Réduire les régressions
- Améliorer la maintenabilité
- Assurer la qualité logicielle

# Niveaux de tests fonctionnels

- **Tests unitaires** : composants isolés
- **Tests d'intégration** : interaction entre composants
- **Tests de validation** : besoin du client

# Tests unitaires

## Objectif

Vérifier que chaque composant fonctionne correctement de façon indépendante

- Détecter rapidement les erreurs lors du développement
- Tester les cas limites
- Faciliter la maintenance et éviter les régressions

# Tests unitaires - Lien SOLID

Cohérents avec pratiques SOLID

*Rappel :*

## Pratiques SOLID - Objectif

Avoir un code modifiable et évolutif de façon à éviter les régressions et les effets de bord.

# Tests unitaires - Structure

- Dossier `tests` avec la même architecture que `main`

→ Pas de lien inutile avec le code source !

# Tests unitaires - Bonnes pratiques

- Approche Arrange-Act-Assert
- Tests indépendents
- Nettoyer entre les tests
- Utilisation de *mocks* pour ne tester qu'un élément à la fois

# Tests unitaires - Bonnes pratiques

- Approche Arrange-Act-Assert
- Tests indépendents
- Nettoyer entre les tests
- Utilisation de *mocks* pour ne tester qu'un élément à la fois

> Les tests sont du code !

- Noms clairs
- Maintenance du code implique maintenance des tests
- Éviter d'avoir du code mort

# Différents outils

## conteneur docker

Machine virtuelle très légère pour virtualiser l'environnement d'exécution et ne pas être dépendant de la machine sur laquelle le code tourne.

## *mocks* et *stubs*

Pour simuler, de façon plus ou moins complexe, le comportement d'objets ou de services externes sans avoir besoin de les créer ou d'y avoir accès. Les mocks sont beaucoup plus configurables que les stubs.

# Mocks

La création des tests soulève la complexité du code :

- Environnement complexe → multiplication des mocks → tests compliqués
- Dans le projet : gros risques d'effets de bord
- Complexité justifiée ?

# Tests d'intégration

## Objectif

Vérifier que plusieurs composants fonctionnent bien ensemble.

Environnements spécifiques à gérer comme :

- bases de données (stockées en mémoire par ex.)
- mocks et stubs
- conteneur docker
- etc.

# Tests d'intégration - Exemples

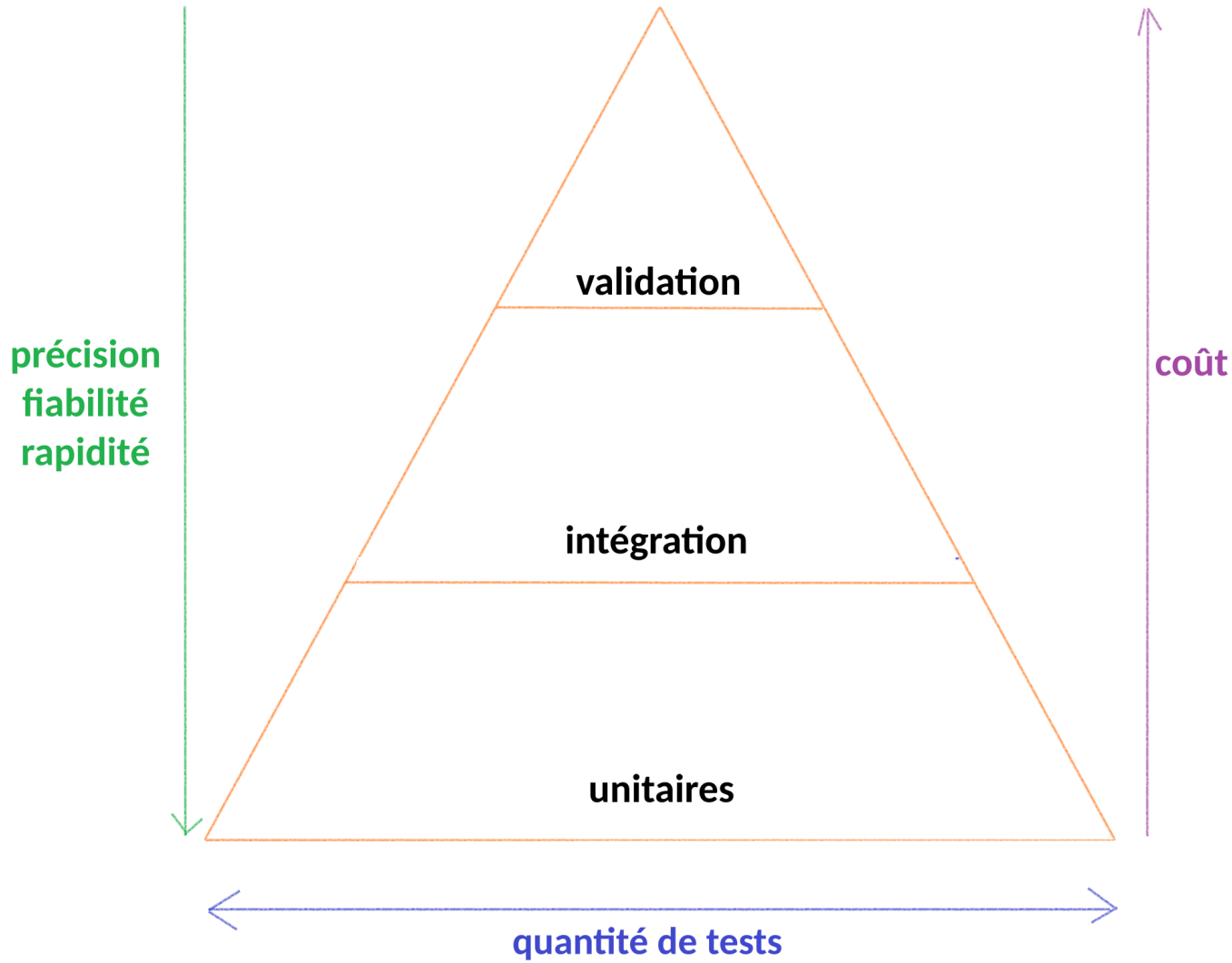
- Connexion et test d'une requête sur une base de données
- Interaction entre deux classes
- Interaction avec un service tiers
- etc.

# Tests de validation

## Objectif

Vérifier que le logiciel fonctionne, qu'il répond aux besoins des utilisateurs finaux, aussi bien d'un point de vue fonctionnel que non-fonctionnel.

# Liens entre tests



# Limites du système

Les tests aident à la définition des limites :

- des fonctions,
- des composants,
- d'un test,
- du système ...

# Fonctionnels VS Non-fonctionnels

## Tests fonctionnels

Évaluent **si** le système fonctionne.

## Tests non-fonctionnels

Évaluent **comment** le système fonctionne.

# Tests non-fonctionnels

- **performance** : vitesse
- **charge** : mise à l'échelle
- **stress** : utilisation extrême
- **destructif** : charge + stress
- **accessibilité** : utilisable par tous
- et pour tous les attributs non-fonctionnels

Le projet définit la priorité des attributs non-fonctionnels à tester.

- Ex : pour une banque, la sécurité est primordiale, la performance moins.

# Test non-fonctionnels - Chaos testing

## Objectif

Tester la résistance aux pannes matérielles.

Outil : **Chaos Monkey** - par Netflix
Fait tomber une instance au hasard.

Pour en savoir plus :
`https://fr.wikipedia.org/wiki/Chaos_Monkey`