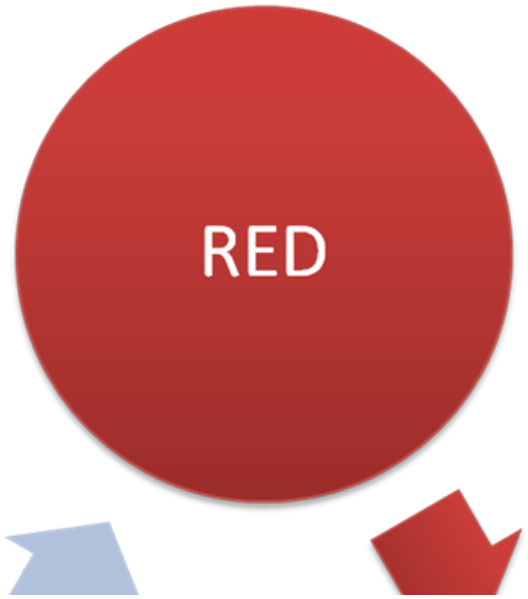
Introduction

Semifir



Les Tests

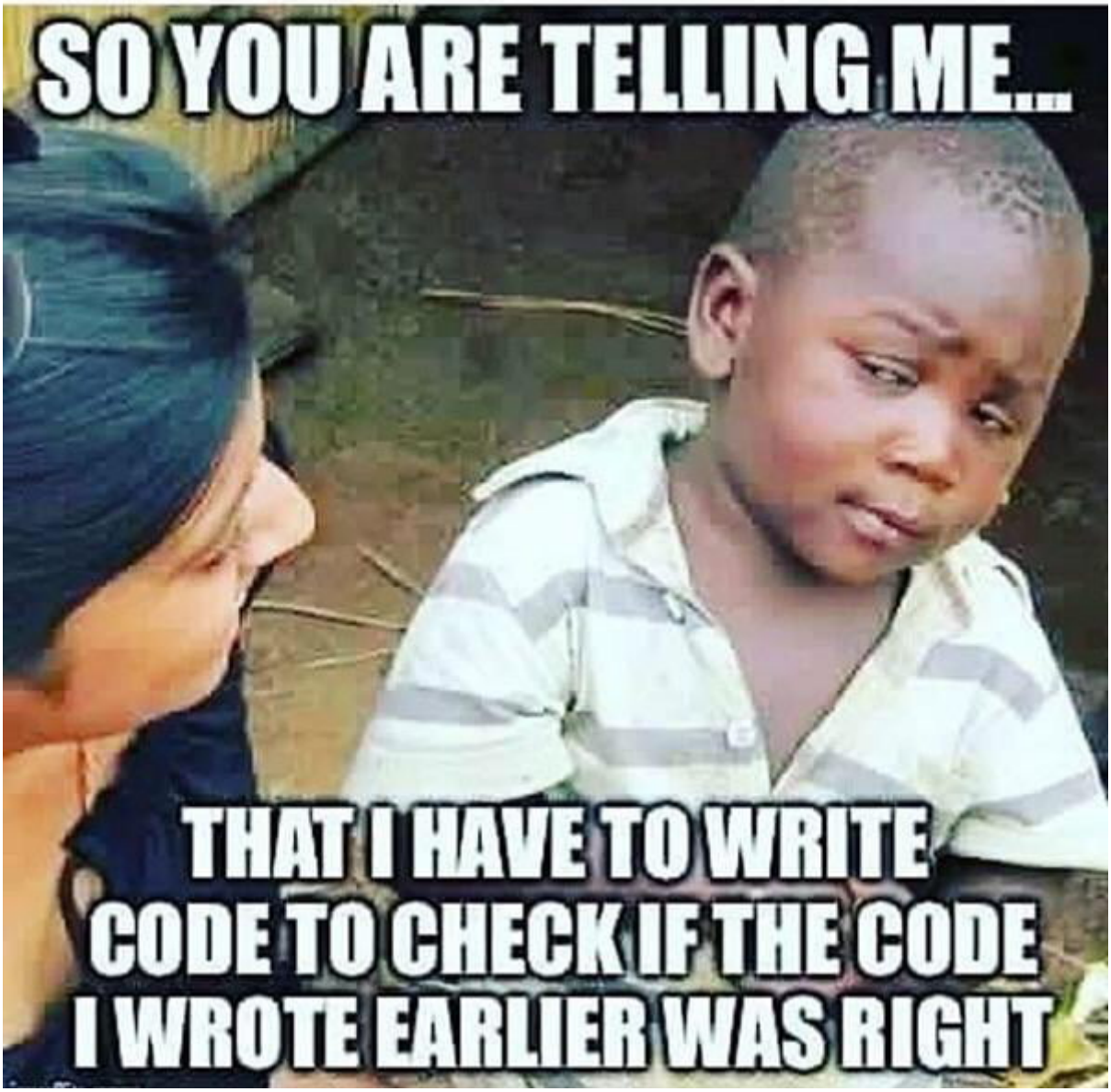
- S'assurent que la solution fonctionne correctement.
- Détectent tout problème potentiel avant sa mise en production.
- Sont aussi une documentation importante du code source.

> C'est une étape cruciale du développement.

Les tests (suite)

- Ce sont des sortes de "scripts" qui s'exécutent automatiquement.
- Ils sont exécutés selon certains critères et peuvent être de plusieurs types.

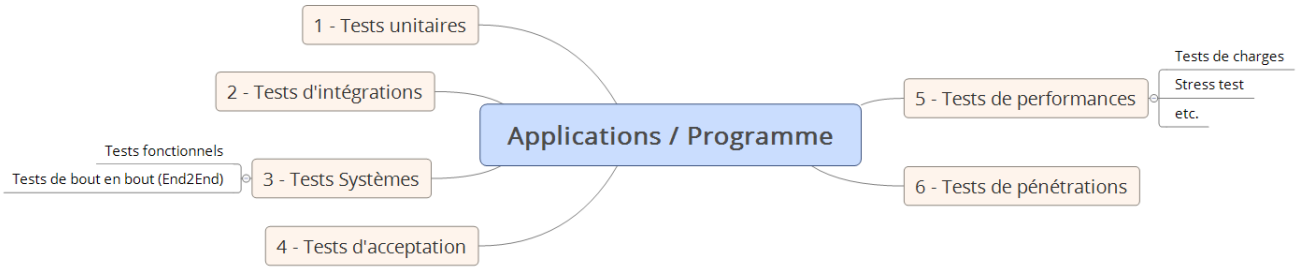
Why ?!



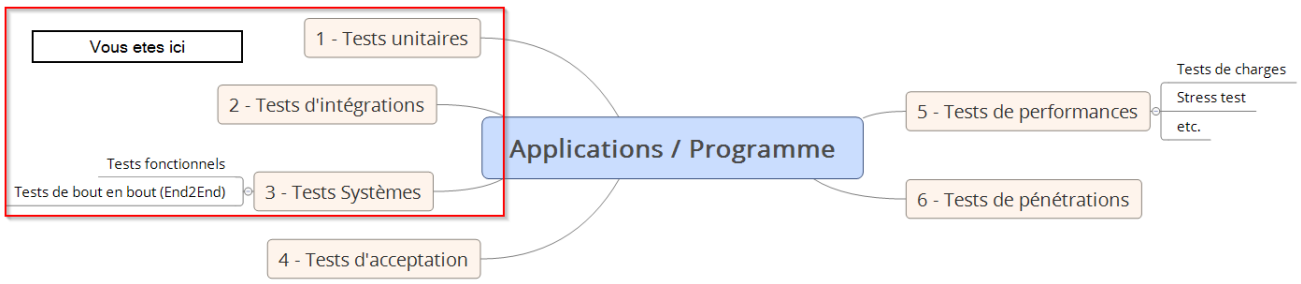
Tester c'est douter ?

Une multitude de types de tests

- Classés selon différents critères (but, portée, granularité...)
- Peuvent être de plusieurs types :



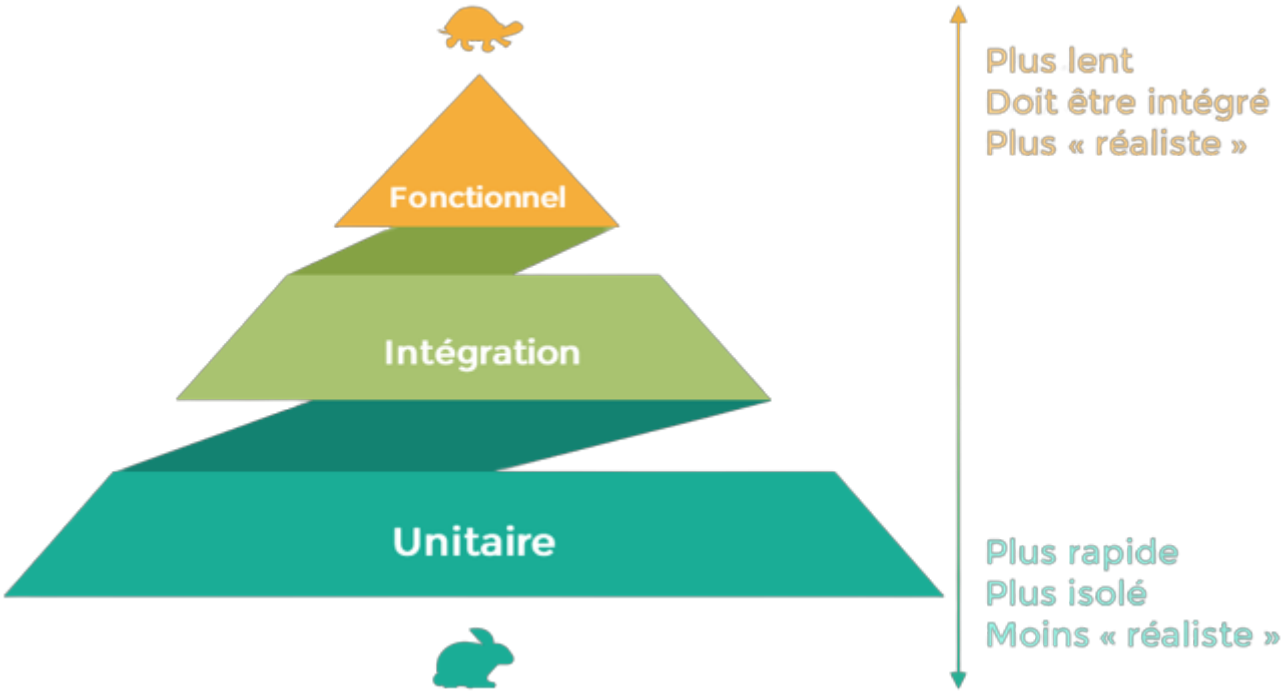
Une multitude de types de tests (suite)



> La TDD se concentre principalement sur cette partie

---

la pyramide des tests



Note: Les tests système n'ont pas été inclus, mais sont bien présents dans la pyramide. Ils ne nous intéressent cependant pas dans le cadre de la TDD.

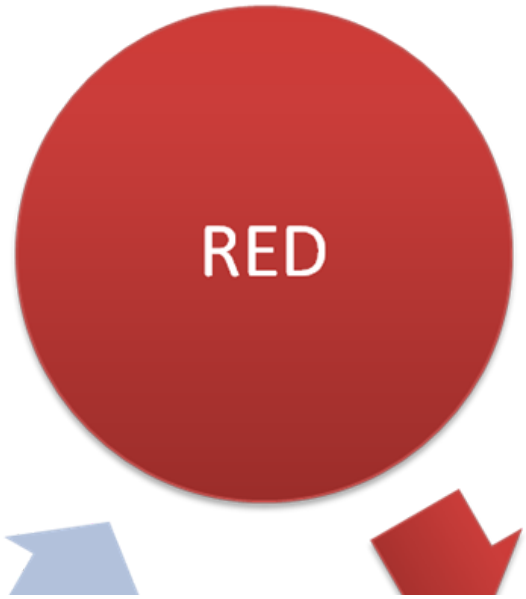
---

la pyramide des tests : explications

- Modèle suggérant un équilibre entre différents types de tests dans un projet.
- Basée sur l'idée que les tests des basses couches sont plus nombreux et moins chronophage à réaliser.

Les types de tests

Semifir



---

3 grandes familles

À chaque étape son type de test :

- **Tests Unitaires** : Vérifier une fonction spécifique
- **Tests d'Intégration** : Vérifier que différentes unités fonctionnent ensemble
- **Tests Fonctionnels** : Vérifier le fonctionnement "d'une fonctionnalité"

Tests Unitaires

> On zoom dans la fonction et on la teste dans une "bulle"

Les Tests Unitaires

- Tester une fonction spécifique, de manière isolée
- Vérifier que le résultat est cohérent et correspond aux attendus
- S'assurer que le code est correct

Mais du coup ...

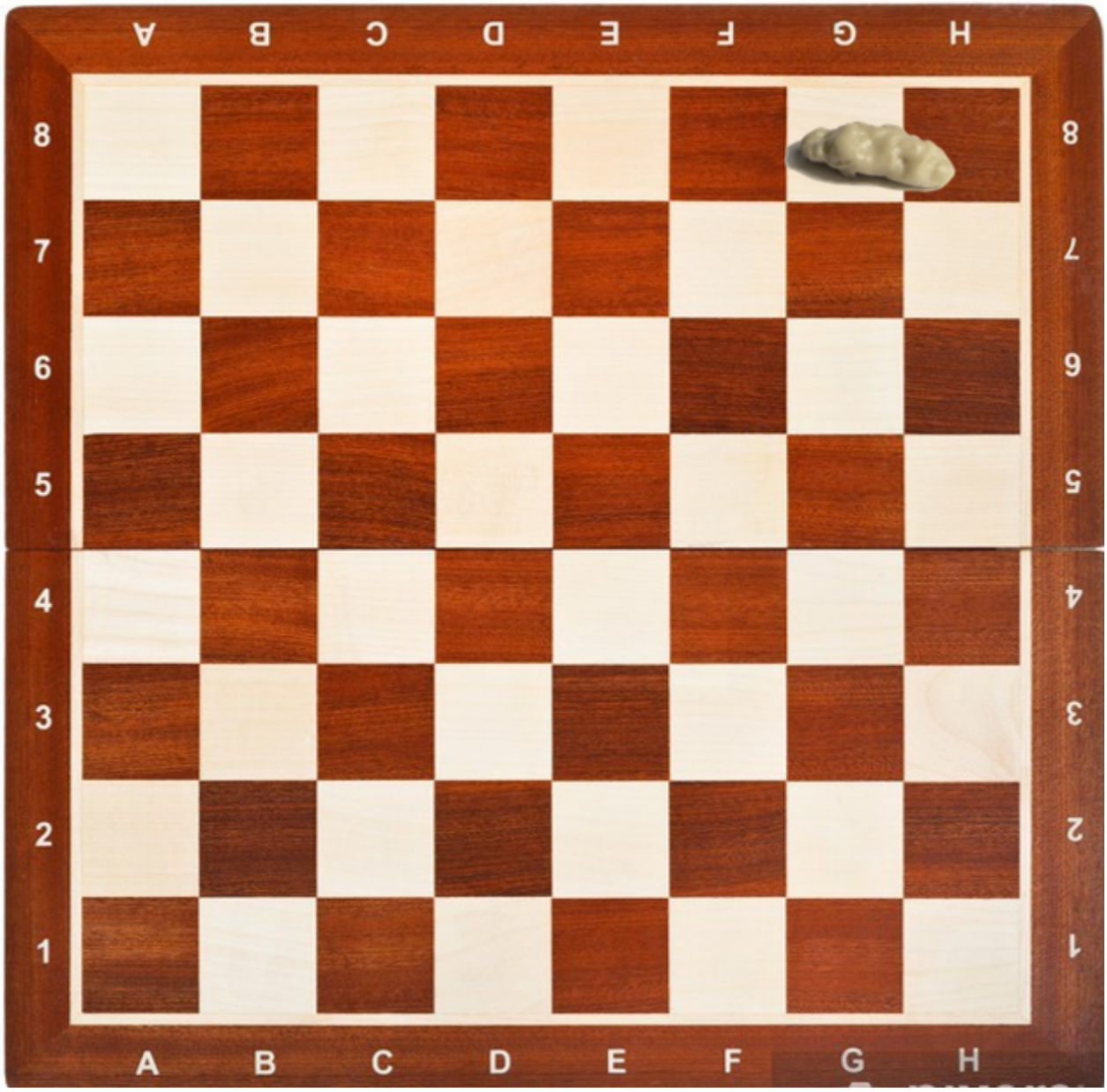
Comment réaliser des tests unitaires sur une classe qui dépend d'une autre ? (aggregation ou composition)

Les mocks

- Les mocks sont des objets qui simulent le comportement d'autres objets.

- Ils peuvent être basés sur une classe réelle ou sur une interface.

En image



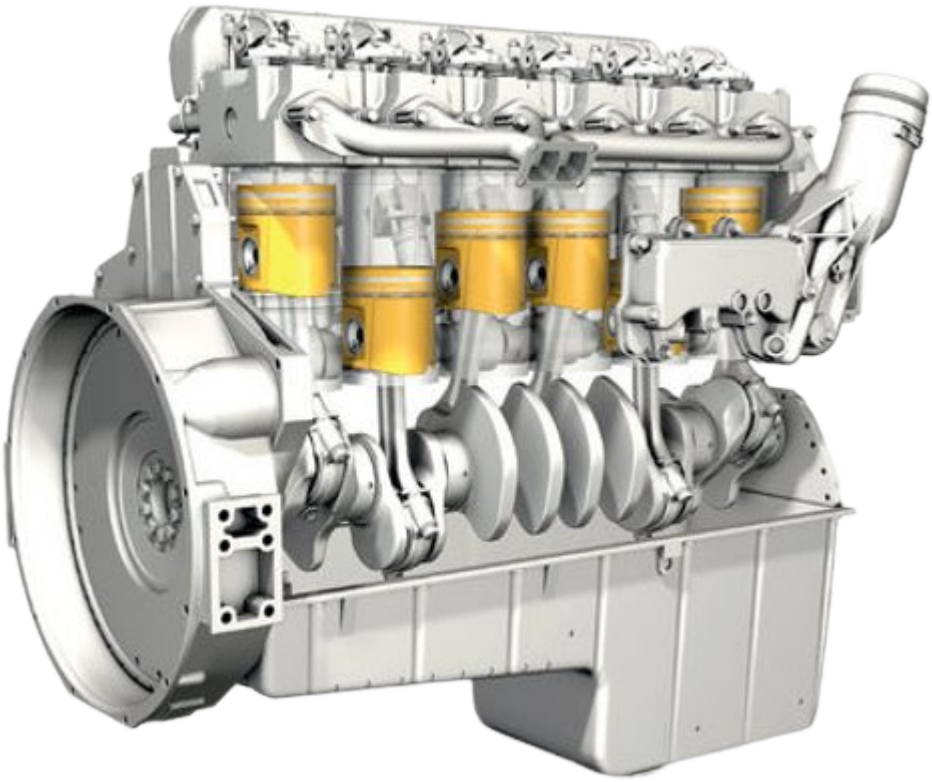
> On pourrait traduire mock par "simulacre"

Pourquoi s'embêter ?

- Lors des tests unitaires, on souhaite tester une classe en **isolation**.
- Si on inclut une instance de la classe dépendante, un bug dans cette dernière pourrait fausser le résultat du test.
- Avec un mock, on s'assure que le test concerne bien uniquement notre classe

---

Tests d'Intégration



> On prend de la hauteur pour s'assurer que les différentes parties fonctionnent **ensemble**

Les Tests d'Intégration

Explications

- Tester les interactions entre différentes classes d'une application
- S'assurer que les résultats sont cohérents

Les Tests Fonctionnels

- Tester le comportement d'une application
- Vérifier que les **fonctionnalités** répondent aux exigences
- Se base sur la logique métier et fonctionnelle

> Nous arrivons sur la partie "utilisateur" de la pyramide

Les tests d'acceptation

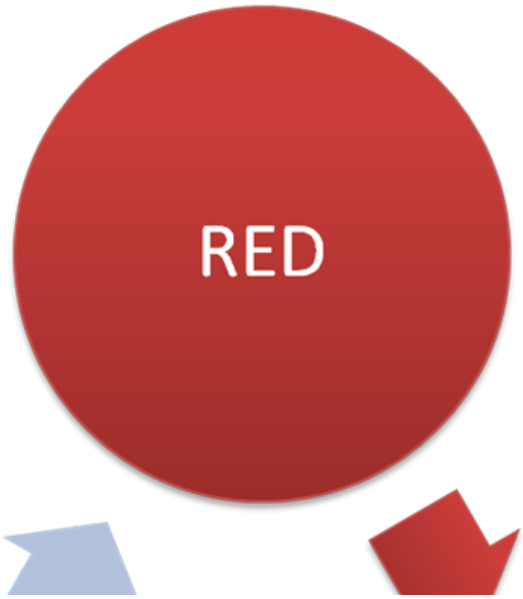
- Reprend la logique des tests fonctionnels
- Cette fois-ci : ils sont validés par le client !

> C'est ce qui nous fait passer de la préprod à la prod !

Bonne pratique

F.I.R.S.T

Semifir



Définition

- **FIRST** est un acronyme qui décrit les caractéristiques d'un bon test.
- Il est utilisé pour définir les tests unitaires.
- **F**ast, **I**solated, **R**epeatable, **S**elf-Validating, **T**imely

---

Fast

- Les tests doivent être rapides à exécuter.
- Le développeur ne doit pas hésiter à cause du temps de test.
- Ils ne doivent pas représenter une contrainte de temps

Isolated / Independent

- Les tests doivent être indépendants les uns des autres.
- On utilise la méthode des 3A : Arrange, Act, Assert

Isolated / Independent

Les 3A

- Arrange : Préparer les données pour le test
- Act : Exécuter la fonction à tester
- Assert : Vérifier que le résultat est correct

> En BDD, on utilise le Given, When, Then

Repeatable

- Les tests doivent être répétables.
- Ils doivent toujours donner le même résultat.

> En d'autres termes, il ne doit pas être dépendant ni aléatoire.

Self-Validating

- Le test doit pouvoir se valider lui-même sans intervention humaine.
- On évitera de créer des tests éphémères à la main.

> Pourquoi tester à la main quand on peut **directement** créer un test automatique ?

Timely ou Thorough

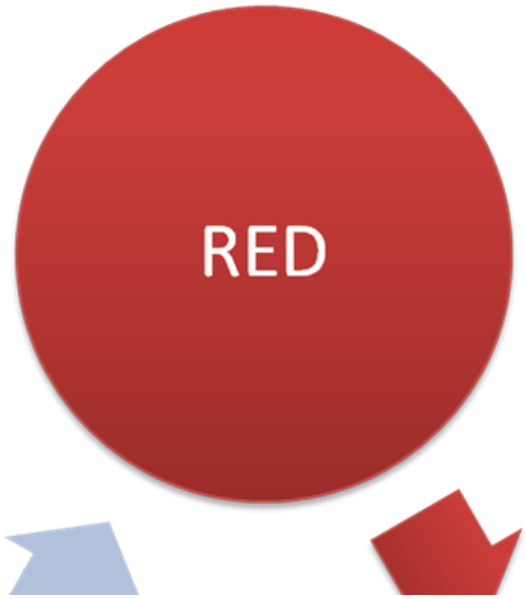
- Les tests doivent être rapides
- Ils doivent tester les fonctionnalités et non les données.

> Tester 100% des données rendrait le test trop long !

Test Driven Development (TDD)

En bref


Semifir


RED

Mais du coup, c'est quoi la TDD ?

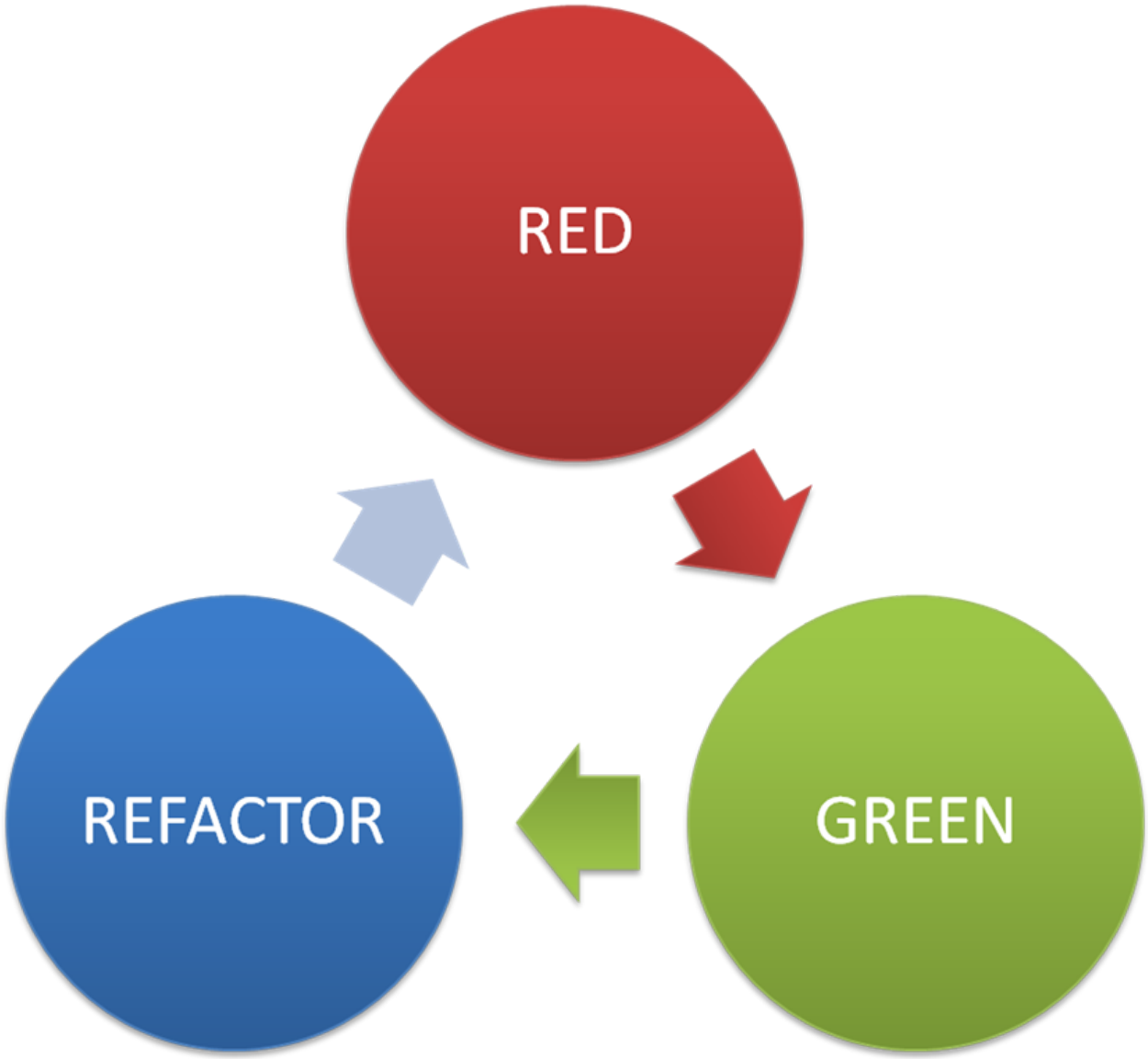
En bref

- Approche consistant à **écrire des tests avant d'écrire du code**.
- Améliorer la qualité du code.
- S'assure que toutes les unités fonctionnent correctement et répondent aux exigences définies.

Mais encore ?

> En TDD, on ne teste pas les erreurs, on teste les **résultats** attendus !

Red Green Refactor



Les étapes en bref

- **RED** : Écrire un test qui **échoue**
- **GREEN** : Créer le code **minimum** pour que le test passe
- **REFACTOR** : Rédiger le code fonctionnel

Avantages

- Penser de manière découpée et développer de manière plus structurée et organisée.
- Aide à réduire les bugs et les problèmes de maintenance à long terme.
- Evite de créer des tests juste pour dire d'écrire des tests.

Exemple sans TDD

- Je suis testeur, et je dois tester la fonctionnalité `addition` de `calculatrice` rédigé par un autre dev.

- Je vais donc écrire un test qui vérifie que `1 + 1 = 2`.
- Problème : le code est mal rédigé : j'attends `2` et je reçois `11`.

Le piège

- Je ne connais pas le résultat attendu, je pars du principe que c'est normal.
- Je rédige un test qui passe.
- Je déclare m'attendre à recevoir "11".

Les limites de cet exemple

- On sait tous que 1 + 1 ne doit pas retourner 11.
- Mais dans la vraie vie, on ne peut pas savoir avec exactitude quel est le comportement attendu.
- On risque donc de se contenter de rédiger "des tests qui passent" et de ne pas s'assurer que le code est correct.

Exemple avec TDD

- Je dispose d'une tâche précise qui me dit que je dois rédiger une calculatrice.
- Mes User Stories étant bien faites, je sais exactement ce que je dois faire.

> "En tant qu'utilisateur, lorsque j'additionne deux nombres, je veux en connaitre la somme".

Etape 1 : RED

- Je rédige un test qui échoue : c'est normal, je n'ai pas encore écrit de code.

- Assurez-vous que le module `pytest` soit installé sur votre machine.
  Si ce n'est pas le cas vous pouvez l'installer avec la commande suivante :

```
pip install pytest
```

- Créer un fichier `test_addition.py` dans le dossier `tests`.
- Le nom du fichier doit commencer par "test_" et le nom de la fonction doit commencer par "test_" pour être exécutée automatiquement par `pytest`.

```
# Fichier test_addition.py
def test_unPlusUnEgalDeux():
    # Test de la fonction d'addition
    resultat = calculatrice.addition(1, 1)

    # Vérification que le résultat est correct
    assert resultat == 2
```

Execution du test

- Ouvrez un terminal dans le dossier `tests`.
- Exécutez la commande suivante :

```
pytest test_addition.py
```

- Le test échoue, c'est normal, je n'ai pas encore écrit de code.

Etape 2 : GREEN (1)

- Je rédige le code minimum pour que le test passe.

```
# Fichier test_addition.py

def test_unPlusUnEgalDeux():
    calculatrice = Calculatrice()
    resultat = calculatrice.addition(1, 1)
    assert resultat == 2
```

```
class Calculatrice:
    def addition(self, a, b):
        # Non, ce n'est pas fonctionnel, et ce n'est pas le but !
        return 2
```

> Mon test réussit et passe au vert ! J'ai fini !

RED 2 : Nouveau test !

```
# Fichier test_addition_bis.py
def test_un_plus_moins_un_egal_zero():
    resultat = calculatrice.addition(1, -1)
    assert resultat == 0
```

RED : Tout est rouge !

- Mon test est rouge
- C'est le moment d'ajouter la logique !

> Ca ne marche pas, donc ça marche

Etape 2 Bis : GREEN (2)

- Je rédige le code fonctionnel.

```
# Fichier test_addition.py
def test_un_plus_moins_un_egal_zero():
    calculatrice = Calculatrice()
    resultat = calculatrice.addition(1, -1)
    assert resultat == 0
# Dans le même fichier
class Calculatrice:
    def addition(self, a, b):
        return a + b
```

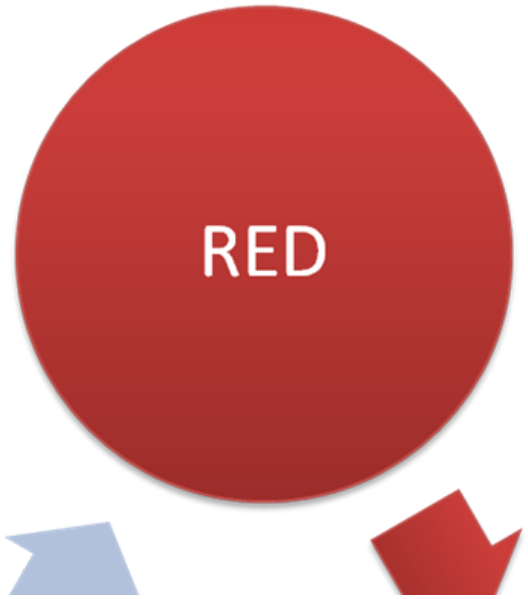
> Je vérifie au fur et à mesure que mon test soit toujours au vert.

La suite

La TDD en pratique

- Java
- TypeScript
- C#
- Python

Semifir




REFACTOR
GREEN