Écrire des tests unitaires

18 octobre 2024

I. Informatique Moderne

I. Informatique Moderne

D'un point de vue culturel, la façon dont on fait de l'informatique depuis trente ans a régulièrement évolué

- culture agile : laisser les ingénieurs organiser le travail en mettant le feedback sur la valeur au centre du développement
- culture CICD : déployer chaque commit aux parties prenantes pour collecter du feedback encore plus rapidement

Ces changements sont articulés autour de la valeur produite par le logiciel

Cette accélération de la collecte du feedback repose sur de l'automatisation

- du build du logiciel
- de la vérification au moyen de tests, de linteurs, d'analyses en tout genres
- du provisionnement d'environnements et de services à la demande
- du monitoring

II. Enjeux

II. Enjeux

Enjeux du test ?

- Un outil de gestion du risque
- Une pratique qui produit du feedback

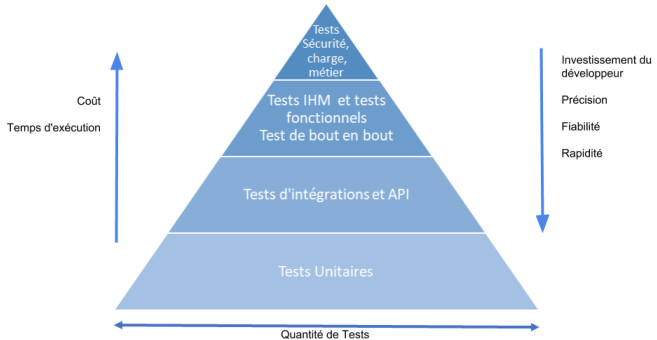
II. Enjeux - tests automatisés

Enjeux du test automatisé ?

- Un test automatisé sert à donner du feedback automatisé
- Un test automatisé est un programme pour comprendre et évaluer un autre programme
- ~~Exclure tout bug~~ il y aura toujours des bugs
- Parvenir à un ratio coût/bénéfice convenable pour publier le logiciel

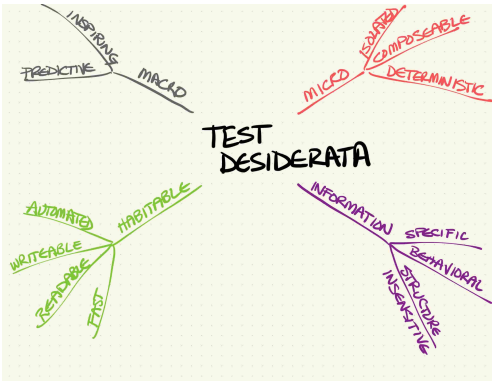
II. Enjeux - cadre théorique

Pyramide des tests



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

II. Enjeux - test desiderata



https://kentbeck.github.io/TestDesiderata/

II. Enjeux - compromis

Chaque type de test a ses forces et ses limites

- Les tests End-To-End offrent une grande confiance
- Plus un test implique de programmes, d'appels réseau, ou d'I/O, plus il est coûteux et lent
- Plus le test couvre une large quantité de données et d'instructions, plus il est lent et gourmand en mémoire
- La spécificité des tests diminue à mesure que leur complexité augmente

II. Enjeux - synthèse

L'objectif est de trouver un équilibre entre

- coût en temps de développement car au plus une situation est difficile à répliquer au plus elle est difficile à automatiser
- prédictabilité, car au plus des tests sont réalisés dans une situation proche de la production au plus ils permettent de prédire le comportement une fois en production
- rapidité d'execution, car au plus il y a de couches de programme, au plus les tests ralentissent

Pour valider automatiquement la publication du build

II. Enjeux - bénéfices

Tester automatiquement le build avec ses dépendances mises à jour par dependabot ou renovate permet de diviser le coût de MCO

III. Tests Unitaires

III. Tests Unitaires

Enjeux des tests unitaires ?

- un type spécial de test automatisé
- accent sur la rapidité du feedback et leur déterminisme (cf test desiderata)
- parfois appelés «tests de programmeurs»

IV. Frameworks de tests en python

IV. Frameworks de tests en python - tests simples avec pytest

```
def test_sum() -> None:
  assert 10 == 5 + 5

def test_product() -> None:
    assert 25 == 5 * 5
```

des tests unitaires sur des calculs

IV. Frameworks de tests en python - tests simples avec pytest

```
def test_sum() -> None:
  assert 10 == 5 + 5

def test_product() -> None:
    assert 25 == 5 * 5
```

des tests unitaires sur des calculs

```
pytest --quiet

tests/test_sums.py::test_sum PASSED [ 50%]
tests/test_sums.py::test_product PASSED [100%]
```

lancement de pytest

IV. Frameworks de tests en python - présentation de pytest

pytest est un framework de test

- il est très simple : le système d'assertion est largement suffisant dans beaucoup de cas
- Il se pilote avec une interface de ligne de commande (*cli*) simple
- Il est externe, il faut l'installer via *pip*

IV. Frameworks de tests en python - présentation de unittest

unittest est un autre framework de test

- il est expressif (famille xunit)
- Il se pilote avec une interface de ligne de commande (*cli*) simple
- Il est un peu trop verbeux
- Il fait partie de la Python standard library

V. Vocabulaire

V. Vocabulaire - assertion

une **assertion** ou an **assertion**

```
assert 10 == 5 + 5
```

C'est la base d'un test unitaire, l'instruction qui compare l'attendu et la réalité

On vérifie que 10 est égal à 5 + 5
... asserte ...
Assert [that] 10 is equal to 5 + 5

V. Vocabulaire - test case

un **cas de test** ou **test case**, la méthode (voire la classe) entière

```
def test_cinq_plus_cinq_egal_dix() -> None:
  assert 10 == 5 + 5
```

Le test cinq plus cinq egal 10

V. Vocabulaire - test harness

un **harnais de test** ou **test harness**, une notion plus rarement évoquée ou connue

```
import dotenv
from os import getenv
dotenv.load_dotenv()

def test_cleaning_process_removes_empty_names() -> None:
    database = Database(getenv('DATABASE_URL'))
    fill_dirty_data(database)
    clean_data(database)
    assert_all_names_are_neither_null_nor_empty(database)
    database.close()
```

on a mis la fonction clean data dans un harnais de test
we put clean data in a test harness

V. Vocabulaire - test suite

un **suite de tests** ou test suite

Une suite de tests permet de référer à un ensemble de tests regroupés arbitrairement

> *J'ai lancé la suite de tests d'intégration pour valider les interactions entre les modules.*
> *I ran the integration test suite to validate the integration between the modules.*

VI. Exemples d'assertions

VI. Exemples d'assertions

Voici quelques exemples d'assertions faites avec **pytest**

VI. Exemples d'assertions

vérifier un booléen

```
assert True
assert False
assert 1 < 10
```

VI. Exemples d'assertions

vérifier une expression

```
assert expected == actual
```

Quand deux valeurs sont comparées, la convention c'est de mettre la valeur attendue *avant* et la valeur calculée *après*
les termes *expected* et *actual* sont également conventionnels

VI. Exemples d'assertions

vérifier un entier

```
assert 29 == jours_fevrier(2024)
```

VI. Exemples d'assertions

vérifier un double ou un float
C'est un cas particulier à cause de la représentation en mémoire des décimaux

```
import pytest

assert abs(x−y) < 0.0001
# ou
assert 2.2 == pytest.approx(2.3, 0.1)
```

VI. Exemples d'assertions

vérifier une string

```
assert "test" in "methodologie du test"
```

VI. Exemples d'assertions

vérifier une liste

```
assert 3 in [1, 2, 3]
```

VI. Exemples d'assertions

vérifier une dictionnaire

```
assert { "a": 1 } == { "a": 1 }
```

VI. Exemples d'assertions

vérifier les erreurs

```
import pytest

with pytest.raises(ZeroDivisionError):
    1 / 0
```

VII. Quoi tester

VII. Quoi tester - quoi tester ?

Le plus important :

- les **calculs**
- les **ifs**
- les **cas limites**
  - division par zéro (with pytest.raises(ZeroDivisionError))
  - inputs invalides mais permis par le type (jours_restants: int $= -1$)
  - des cas extrèmes (2'000'000'000 pour un entier, si la donnée est envisageable)
- les **use-cases**
  - *Quand un utilisateur donne un feedback, on enregistre une version anonymisée de ses informations*
  - *Quand on propose des candidats à un utilisateur, la liste est dédoublonnée*

VII. Quoi tester - quoi ne pas tester ?

Ce qui fait baisser la valeur des tests :

- les passe-plats (retourner une valeur non transformée)
- les comportements déjà couverts par des tests
- l'intégralité des possibles
- les comportements rendus impossibles par d'autres outils tels que pylint

VII. Quoi tester - le TDD

Le **test-driven-development** est une méthode de développement dans laquelle on écrit des tests avant d'écrire du code de production
Il suit un rythme en trois phases :

- écrire un test qui échoue, le lancer
- écrire le code pour faire passer le test, sans écrire plus que nécéssaire
- refactoriser, puis pousser le code

Une technique qui implique une testabilité sans pareil, et qui force à se placer du point de vue de l'utilisateur

VIII. Exercices

VIII. Exercices

Récupération de l'environement de travail pour le TD et le TP

```
git clone https://github.com/PierreAntoineGuillaume/methodologie-du-test
cd methodologie-du-test
python3 -m venv venv
. venv/bin/activate
pip install -r requirements.txt
pytest --quiet
```