Tests Logiciels - Master MS2D

Christophe Brun

Campus Saint-Michel IT

21 septembre 2033

Compétences:

- Savoir développer et mettre en place des tests unitaires sur une application.
- Savoir développer et mettre en place des tests d'intégration sur une application.
- Savoir configurer un pipeline d'intégration continue sur une application.

Programme:

1. Les tests unitaires
   - Qu'est-ce qu'un test ?
   - Les tests unitaires
   - Les bonnes pratiques
   - Le Test Driven Development
   - Le code coverage
   - Tests avec langage naturel
2. Les tests d'intégration
   - Les tests d'intégration
   - Le fonctionnement
   - Les bonnes pratiques
3. L'intégration continue
   - Mise en place d'une plateforme d'intégration continue (GitLab CI)
   - Conteneurisation d'une application (API)
   - Configuration d'un pipeline de tests

Intervenant sur le module Tests Logiciels

Christophe Brun, CTO d'In France

- 1ere année d'intervenant à Saint-Michel 🤩.
- 7 ans de conseil en développement au sein d'SSII.
- 6 ans de conseil en développement à mon compte PapIT.
- Directeur technique et associé d'In France.
- Passionné!

Qu'est-ce qu'un test ?

Définition de l'International Software Testing Qualifications Board et IBM

L'ISTQB définit les termes suivants dans son glossaire[1] :
Test : Un ensemble d'un ou plusieurs cas de tests.
Cas de test : Un ensemble de conditions préalables, de données d'entrée, d'actions (le cas échéant), de résultats attendus et de postconditions, élaboré sur la base des conditions de test.

Selon IBM[2] :
"Le test logiciel est le processus qui consiste à évaluer et à vérifier qu'un produit ou une application logicielle fait ce qu'il ou elle est censé(e) faire."

---

[1] ISTQB, Glossaire des termes utilisés en tests de logiciels, https://www.cftl.fr/wp-content/uploads/2018/10/Glossaire-des-tests-logiciels-v3_2F-ISTQB-CFTL-1.pdf

[2] IBM, Qu'est-ce que le test logiciel ?, https://www.ibm.com/fr-fr/topics/software-testing

Histoire du test logiciel moderne

- ISTQB, l'International Software Testing Qualifications Board a été fondée en 1998.[3]
- Dans son livre Extreme Programming Explained, Kent Beck parle de "Test-first Programming" en 1999.
- Les standards modernes datent de la fin des années 90, début des années 2000.
- JUnit, le framework de test le plus utilisé en Java date de 2002.[4]
- Début du BDD (Behavior-Driven Development) avec JBehave, censé remplacer JUnit en utilisant le behavior au lieu de test, date de 2023.[5]

[3] ISTQB, About Us, https://www.istqb.org/about-us/who-we-are

[4] Steven J. Zeil, Unit Testing Frameworks, https://www.cs.odu.edu/~zeil/cs350/latest/Public/junit/index.html

[5] Cucumber, https://cucumber.io/docs/bdd/history/

Brief sur les outils de test logiciel

- Les frameworks de test retournent 3 statuts, OK, FAILED et ERROR.
- OK si aucune erreur n'a lieu et FAILED en cas d'erreur d'assertion, `AssertionError` en Python et Java.
- Pas de standard universellement accepté mais un format domine, le XML JUnit. Un XML le définit par la grammaire `https://windyroad.com.au/dl/Open%20Source/JUnit.xsd`.
- Vient de l'écosystème Java avec le package du même nom, JUnit, mais est présent dans la plupart des frameworks de test comme PHPUnit, Pytest.
- L'interopérabilité entre les outils est permise par cette XSD commune mais pas plus de contrainte. Par exemple, l'interopérabilité entre les frameworks de test et les outils de CI/CD.
- Certains outils, souvent des solutions propriétaires, rechignent toujours à exporter un JUnit contenant les résultats de test pour fermer leur environnement.

L'assertion error

En Python

L'instruction `assert` vérifie une condition. Si la condition est vraie, cela ne fait rien et votre programme continue simplement à s'exécuter. Mais si la condition d'assertion est fausse, elle lève une exception `AssertionError`.

```
assert 23 % 2 == 0, "Le restant de la division est différent de 0"
```

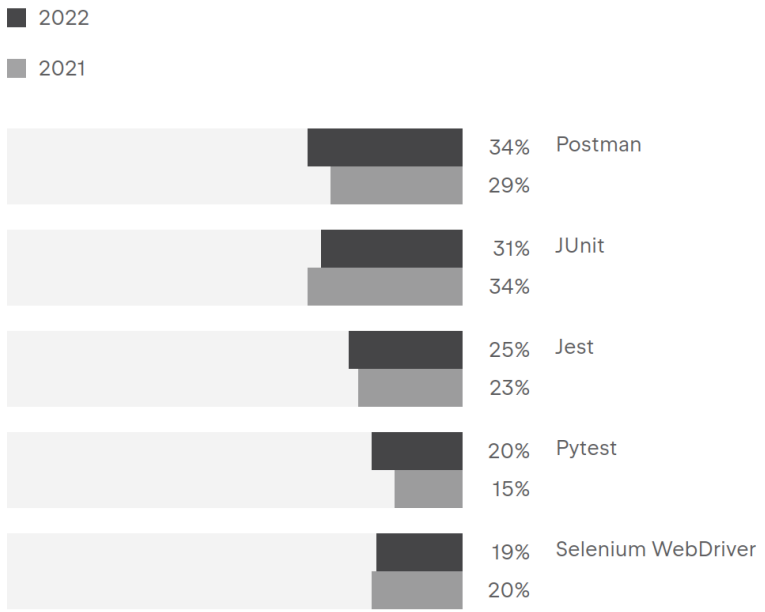
Étant toujours fausse, le programme crash.

```
chrichri@chrichri-HKD-WXX:~$ python3 -c 'assert 23 % 2 == 0, "Le restant de la
    division est différent de 0"'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
AssertionError: Le restant de la division est différent de 0
```

Pourquoi Pytest pour le testing

"pytest is a mature full-featured Python testing tool that helps you write better programs.[6]"

Pytest est le framework de test en Python le plus utilisé selon les sondages de JetBrains[7].
Seuls JUnit et Pytest sont pour tous usages, les autres sont orientés web.



[6] pytest: helps you write better programs , https://docs.pytest.org

[7] JetBrains, Which test frameworks , https://www.jetbrains.com/lp/devecosystem-2022/testing/

- Un module de test Pytest est un module Python préfixé par `test_`.
- Toutes les méthodes préfixées par `test_` sont exécutées par Pytest. Qu'elles soient dans une classe ou non. Les autres méthodes ne sont pas exécutées sauf si les tests ne les appellent.

Ici par exemple, la précédente assertion est intégrées dans une méthode nommée `test_divide` dans le module de test `test_division.py`.

```
def test_divide(): # Un test Pytest est préfixé par test_
    assert 23 % 2 == 0, "Le restant de la division est différent de 0."
```

Lancer avec la commande `python -m pytest test_divison.py`, Pytest affiche le rapport de test.

```
...
def test_divide(): # Un test Pytest est préfixé par test_
> assert 23 % 2 == 0, "Le restant de la division est différent de 0."
E AssertionError: Le restant de la division est différent de 0.
E assert (23 % 2) == 0

test_divison.py:2: AssertionError
...
```

De très utiles et nombreuses options peuvent compléter cette ligne de commande. Pour les découvrir, lancez `python -m pytest --help` et “RTFM”.

Les tests paramétriques sont des tests qui prennent un ensemble de paramètres en entrées.
Ils permettent de tester plusieurs cas avec un seul test.
Le rapport génère un cas de test par paramètre. Il est donc plus détaillé et permet de trouvé les tests en échec plus facilement que si l'`assert` était dans une boucle, ce qui ne donne qu'un statut OK ou Fail dans le rapport.

```
import pytest
...
@pytest.mark.parametrize("dividend", range(100)) # Paramétrage du test
def test_divide_from_0_to_99(dividend): # Doit avoir un argument présent dans le
      paramétrage
   assert dividend % 2 == 0, "Le restant de la division est différent de 0."
```

Génère 100 cas de tests. Met en évidence les dividendes dont le restant de la division par 2 est différent de 0, ici 1 par exemple.

```
collecting ... collected 100 items

test_divison.py::test_divide_from_0_to_99[0] PASSED [ 1%]
test_divison.py::test_divide_from_0_to_99[1] FAILED [ 2%]
test_divison.py:7 (test_divide_from_0_to_99[1])
1 != 0

Expected :0
Actual :1
<Click to see difference>

dividend = 1

    @pytest.mark.parametrize("dividend", range(100)) # Paramétrage du test
    def test_divide_from_0_to_99(dividend): # Doit avoir un argument présent dans
          le paramétrage
> assert dividend % 2 == 0, "Le restant de la division est différent de 0."
E AssertionError: Le restant de la division est différent de 0.
E assert (1 % 2) == 0
```

Pytest

Plantage dans un test, quel statut?

Ce test crash avant l'assert.

```
def test_fail_or_error(): # Une erreur donne un fail ou error?
    dividende = 23 / 0
    assert dividende % 2 == 0, \
        "Le restant de la division est différent de 0."
```

Malgré l'absence d'`assertionError`, le test est en échec et non en erreur.

```
test_divison.py::test_fail_or_error FAILED [100%]
test_divison.py:12 (test_fail_or_error)
def test_fail_or_error(): # Une erreur donne un fail ou error?
> dividende = 23 / 0
E ZeroDivisionError: division by zero

test_divison.py:14: ZeroDivisionError
```

> Pour éviter les faux négatifs, un test doit tendre, tant que faire se peut, vers un `assert`. Pour cela pensez à utiliser les fonctions `parametrize` et `fixture`.

Cf. https://github.com/St-Michel-IT/testing/blob/main/test_customer_database.py

Un `return` ou un `yield` envoie l'objet au test dans l'état voulu.

```
def customer_without_table():
    """..."""
    customer = Customer() # Avant le test, c'est le setup
    yield customer # Un yield évite de sortir de la fonction
    customer.con.close() # Après le test, le teardown
```

Le test n'a qu'une seule ligne, qu'un `assert` l'échec ne pourrait venir que de là.

```
def test_instantiation(customer_without_table):
    """..."""
    assert isinstance(customer_without_table.con, Connection)
```

En cas de plantage dans la fixture, i.e. avant ou après le test, la sanction serait erreur et non échec.
Le rapport reflétera fidèlement le test et sera exempt de faux négatif.

Pytest

Le scope des fixtures

L'argument `scope` du décorateur `pytest.fixture` défini la durée de vie de la fixture. Il peut prendre 3 valeurs :

- `function` : La valeur par défaut si on ne met rien. La fixture est exécutée à chaque fonction de test `def test_...` qui l'appelle directement ou indirectement, donc les éventuels setup et teardown de cette dernière aussi.
- `module` : Une fois par module `test_....py`.
- `session` : Une seule fois durant la session de test quel que soit le nombre de modules et de tests exécutés.

```
@pytest.fixture(scope="function") # Scope de la fixture, par default function
def customer_without_table():
    """
    Connection to in memory database using the Customer class
```

Pour tester une API en étant sûr le token n'est pas périmé, on peut utiliser le scope `function` pour en avoir un nouveau à chaque appel.
Si préparer les conditions initiales, le setup, prend du temps, on évitera si possible de se répéter à chaque test avec les scopes `module` `session`.

Pytest

Le `conftest.py`

Il n'a qu'une seule particularité, c'est d'être importé automatiquement par Pytest lorsqu'il est présent dans le dossier courant des tests.
Il permet de factoriser les fixtures et les paramétrages de tests utilisés dans plusieurs modules de tests, car ces derniers seront automatiquement disponible sans même un import dans le module.

```
./
├── conftest.py
├── test_no_directory.py
├── unit
│   └── test_integration.py
└── functional
    └── test_functional.py
```

Pour tous ces tests, s'ils sont lancés depuis la racine avec la commande `pytest`, le `conftest.py` sera importé automatiquement.

PyCharm intègre par défaut Pytest comme lanceur de tests et fournit l'autocomplétion pour les fixtures du `conftest.py`.

Pytest

Le secret de la réussite

Pytest est un framework complet voir complexe.
Mais quasi toutes les fonctionnalités auxquelles on peut penser sont déjà implémentées et décrites une documentation de qualité.
Comme avec toutes les grosses libraries, il faut avoir confiance en leur design et chercher dedans avant de réinventer la roue.
La liste des fonctionnalités est trop longue, nous venons juste d'en découvrir les principales.

Donc une fois encore, comme pour toutes les bonnes libraires à connaître, le secret est "**RTFM**" !