

# Tests Logiciels - Master MS2D

Christophe Brun

Campus Saint-Michel IT

21 septembre 2033



# Table des matières

1 Programme du module

2 Généralités

# Tests Logiciels

Compétences acquises au cours des 3 jours du module

- 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.



# Tests Logiciels

Le programme des 3 jours du module

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

- 1<sup>ère</sup> 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 !



# L'histoire du testing

- ISTQB, l'International Software Testing Qualifications Board a été fondée en 1998<sup>1</sup>
- 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<sup>2</sup>
- Début du BDD (Behavior-Driven Development) avec JBehave, censé remplacer JUnit en utilisant le behavior au lieu de test, date de 2003<sup>3</sup>

---

<sup>1</sup>ISTQB, About Us, <https://www.istqb.org/about-us/who-we-are>

<sup>2</sup>Steven J Zeil, Unit Testing Frameworks,

<https://www.cs.odu.edu/~zeil/cs350/latest/Public/junit/index.html>

<sup>3</sup>Cucumber, <https://cucumber.io/docs/bdd/history/>

# 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<sup>4</sup> :

**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<sup>5</sup> :

“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.”

---

<sup>4</sup>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](https://www.cftl.fr/wp-content/uploads/2018/10/Glossaire-des-tests-logiciels-v3_2F-ISTQB-CFTL-1.pdf)

<sup>5</sup>IBM, Qu'est-ce que le test logiciel ?,

<https://www.ibm.com/fr-fr/topics/software-testing>

# Qu'est-ce qu'un test unitaire ?

Définition de la taverne du testeur<sup>6</sup>

Il obéit au principe “F.I.R.S.T.” :

- **Fast** : S'exécute rapidement et est donc automatisé
- **Isolated** : Est indépendant des facteurs externes et des autres tests
- **Repeatable** : Isole les bugs automatiquement
- **Self-validating (autonome)** : N'est pas ambiguë (pas sujet à interprétation, ne demande pas une action manuelle pour vérifier le résultat)
- **Timely (tot)** : Écrit en même temps que le code (même avant en TDD)

Si le test échappe à un de ces principes, il n'est probablement pas unitaire.

Pour isoler les bugs automatiquement, un test unitaire doit tester le code unitairement, et isolément. Il doit donc être écrit dans le but de tester la logique d'une ligne de code, et quasiment si un test unitaire échoue vous pouvez connaître la ligne de code incriminée.

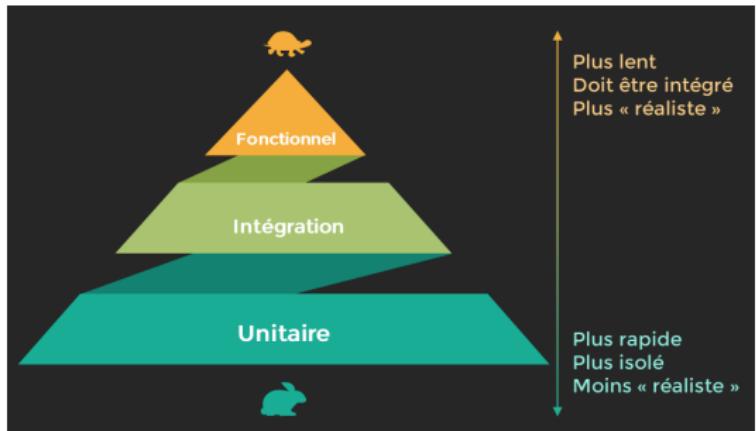
<sup>6</sup>Mais c'est quoi un test unitaire ?, <https://lataverne du testeur.fr/2018/04/11/mais-cest-quoi-un-test-unitaire/>

# Testing, les frameworks de test unitaire

- UnitTest, le framework de test standard de Python
- PyTest, un framework pour tout faire en Python
- JUnit, le framework de test le plus utilisé en Java
- PHPUnit, le framework de test standard de PHP
- Google Test, pour le C++
- Jest pour JS, Babel, pour TypeScript, Node, React, Angular, Vue et plus encore
- etc

# La Pyramide des tests

Différents types de tests peuvent être identifiés. La classification la plus classique étant la suivante<sup>7</sup> :



Mais d'autres valent le coup d'être découvertes...

<sup>7</sup>OPENCLASSROOMS, Testez votre code Java pour réaliser des applications de qualité, <https://openclassrooms.com/fr/courses/6100311-testez-votre-code-java-pour-realiser-des-applications-de-qualite/6616481-decouvrez-les-tests-dintegration-et-les-tests-fonctionnels>

# Testing

- Un programme a pour entrée le standard input, et sorties les standard output et standard error, ce sont les standard streams
- Un programme se termine avec un code retour, le plus souvent un entier différent de 0 en cas de problème
- Les frameworks de tests retournent 3 statuts, OK, FAILED et ERROR
- Les OK si aucune erreur n'a lieu et FAILED en cas d'erreur d'assertion

```
christr@christr-HKD-WXX:~/Documents/Campus St Michel IT$ cat test_stdout_stderr.py
import sys

def test_hello():
    print('Hello testing')
    print('stderr during testing', file=sys.stderr)
    assert False

christr@christr-HKD-WXX:~/Documents/Campus St Michel IT$ python3 -m pytest test_stdout_stderr.py 1> /dev/null
christr@christr-HKD-WXX:~/Documents/Campus St Michel IT$ python3 -m pytest test_stdout_stderr.py 2> /dev/null
=====
platform linux -- Python 3.11.4, pytest-7.2.1, pluggy-1.8.0+repack
rootdir: /home/christr/Documents/Campus St Michel IT
collected 1 item

test_stdout_stderr.py F [100%]

=====
===== FAILURES =====
test_hello

def test_hello():
    print('Hello testing')
    print('stderr during testing', file=sys.stderr)
>     assert False
E     assert False

test_stdout_stderr.py:6: AssertionError
----- Captured stdout call -----
Hello testing
----- Captured stderr call -----
stderr during testing
===== short test summary info =====
FAILED test_stdout_stderr.py::test_hello - assert False
1 failed in 0.10s
christr@christr-HKD-WXX:~/Documents/Campus St Michel IT$ echo $?
1
```

# 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.

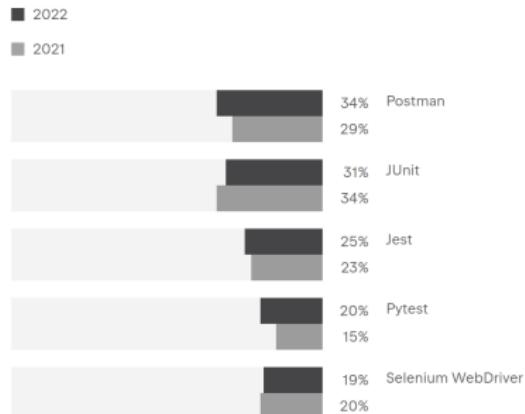
```
chrichri@chrichri-HKD-WXX:~$ python3 -c 'assert 23 % 2 == 0, "Le restant de la
division est différent de 0"'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
AssertionError: Le restant de la division est différent de 0
```

# Pourquoi Pytest pour le testing

“pytest is a mature full-featured Python testing tool that helps you write better programs.<sup>8</sup>”

Pytest est le framework de test en Python le plus utilisé selon les sondages de JetBrains<sup>9</sup>.

Seuls JUnit et Pytest sont pour tous usages, les autres sont orientés web.



<sup>8</sup> pytest: helps you write better programs , <https://docs.pytest.org>

<sup>9</sup> JetBrains, Which test frameworks ,

<https://www.jetbrains.com/lp/devcosystem-2022/testing/>

# Pytest

## Rédiger un test

- 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 exécutées uniquement si les tests les appellent.

Ici par exemple, la précédente assertion est intégrée 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."
```

# Pytest

## Lancer les tests

Lancer avec la commande `python -m pytest test_division.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_division.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”.

# Pytest

## Les tests paramétriques

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 trouver 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."
```

# Pytest

## Les tests paramétriques

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.

# Pytest

## Les fixtures

Cf. [https://github.com/St-Michel-IT/testing/blob/main/test\\_customer\\_database.py](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

Les fixtures, le moyen du pattern A.A.A<sup>10</sup>

Le pattern du A.A.A, Arrange, Act, Assert, est facile à implémenter avec les fixtures. Arrange et Act sont isolé dans la fixture, Assert seul est dans le test.

```
def customer_without_table():
    """
    ...
    """
    customer = Customer() # Arrange et Act dans le setup avant yield ou return
    yield customer
    customer.con.close() # Pas de nom en A, ils l'ont oublié?
```

Le test tend vers un assert quasi seul.

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

Le B.D.D. suit le pattern A.A.A sous un autre nom : Given-When-Then. Le langage Gherkin utilise les étapes Given-When-Then pour spécifier les comportements dans les scénarios.

<sup>10</sup>Arrange-Act-Assert: A Pattern for Writing Good Tests <https://automationpanda.com/2020/07/07/arrange-act-assert-a-pattern-for-writing-good-tests/>

# Pytest

## Le scope des fixtures

L'argument `scope` du décorateur `pytest.fixture` définit 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 le 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 disponibles 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 coverage

Pas de support natif du coverage par Pytest, il faut installer le plugin `pytest-cov`<sup>II</sup>. Pour appeler le plugin, il faut passer en argument le module à tester et le(s) test(s) de ce dernier.

```
pytest --cov=\textcolor{flatgreen}{customer_database} ./test_customer_database.py
...
rootdir: /home/chrichri/Documents/Campus-St-Michel-IT/testing
plugins: cov-4.1.0
collected 6 items

test_customer_database.py ..... [100%]

----- coverage: platform linux, python 3.11.4-final-0 -----
Name      Stmts  Miss  Cover
-----
customer_database.py  9      0   100%
-----
TOTAL    9      0   100%
```

---

<sup>II</sup>Welcome to `pytest-cov`'s documentation!,

<https://pytest-cov.readthedocs.io/en/latest/>

# Pytest

## Le coverage

Coverage for **customer\_database.py**: 100%

9 statements 9 run 0 missing 0 excluded

« prev ^ index » next coverage.py v7.3.1, created at 2023-09-28 22:38 +0200

```
1 import sqlite3
2
3
4 class Customer:
5     """
6         Customer class containing all the method to interact with the customer table
7     """
8
9     def __init__(self, path=":memory:") -> None:
10         """
11             Connect to in memory database by default, or to a database file if
12             specified.
13         """
14         self.con = sqlite3.connect(path)
15
16     def create_table(self) -> None:
17         """
18             Create the customer table if not exists
19         """
20         self.con.execute("""
21             CREATE TABLE IF NOT EXISTS customer (
22                 id INT PRIMARY KEY NOT NULL,
23                 email TEXT NOT NULL
24             )
25         """)
26
27     def insert(self, customer_id: int, email: str) -> None:
28         """
29             Insert a new customer in the table.
30
31             :param customer_id: The customer id as an int
32             :param email: The customer email as a string
33         """
34         self.con.execute("""
35             INSERT INTO customer (
36                 id,
37                 email
38             ) VALUES (
39                 ?,
40             )""", (customer_id, email))
41         self.con.commit()

    « prev ^ index » next coverage.py v7.3.1, created at 2023-09-28 22:38 +0200
```

L'option `--cov-report=html` permet de générer un rapport HTML plus détaillé qui met en lumière quelles parties du code sont couvertes par les tests et lesquelles ne le sont pas.

**Exercice :** Trouvez quel test du module [https://github.com/St-Michel-IT/testing/blob/main/test\\_customer\\_database.py](https://github.com/St-Michel-IT/testing/blob/main/test_customer_database.py) couvre quelle partie de ce code:

# Pytest

## Le secret de la réussite

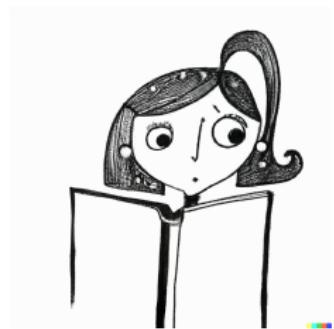
Pytest est un framework complet voir complexe.

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**” !



# Pytest

## Travaux dirigés sur les tests unitaires

Les tests unitaires peuvent-être défini comme ceux testant les classes et fonctions d'une seule librairie.

Exemple de la librairie du repo

<https://github.com/St-Michel-IT/testing/>, le module `customer_database.py` testé par `test_customer_database.py` est donc un exemple de tests unitaires.

**Exercice :** Toujours dans le même repo, le module `witness_number.py` est un début implémentation d'une généralisation du petit théorème de Fermat expliquée sur la chaîne Youtube Numberphile par Matt Parker<sup>12</sup>. Cette librairie n'est pas déjà libre sur le web, finissons son développement avec des standards de qualité élevés grâce à nos connaissances en testing.

---

<sup>12</sup>Numberphile, Witness Numbers (and the truthful 1,662,803),

[https://www.youtube.com/watch?v=\\_MscGSN5J6o](https://www.youtube.com/watch?v=_MscGSN5J6o)

# Tests avec langage naturel

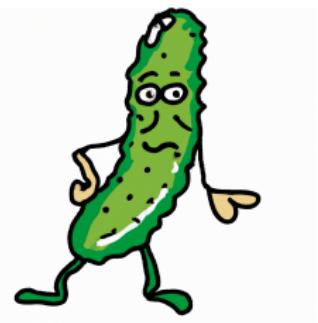
A.K.A le BDD

Les slides qui suivent sont truffés de biais négatifs sur le BDD suite à de mauvaises expériences 😅.

Le BDD est le sigle de Behavior-Driven Development.

Vous remarquerez qu'il n'y a ici aucune mention liée au testing?

Alors pourquoi est-il régulièrement associé au testing ?



Parce que le BDD est un outil de communication entre les développeurs et les métiers qui permet à ceux qui spécifient le produit de le décrire par rapport au comportement (behavior) attendu. Le tout dans une syntaxe proche du langage naturel. Ce langage peut être utilisé pour générer des tests. On pourrait presque finir par croire que le métier peut écrire les tests.

# Spécifier avec le BDD

Exemple de Cucumber, le plus répandu

Développé en début des années 2000 par Daniel Terhorst-North, il fait entre autre le constat que : “Behaviour” is a more useful word than “test”<sup>13</sup>.

Il utilise les mots clés “Given”, “When” et “Then” pour décrire le comportement attendu du produit en développement dans un scénario.

```
Given some initial context (the givens),  
When an event occurs,  
Then ensure some outcomes.
```

Par exemple :

```
Given I just opened a new private window in my browser  
When I browse the https://in-france.fr website  
Then I want to discover the french economy by activity  
And by region.
```

On comprend facilement que ce sont des critères d’acceptation haut niveau. Les tests liés seront donc des tests d’acceptation/fonctionnels.

<sup>13</sup>Daniel Terhorst-North, Introducing BDD, <https://dannorth.net/introducing-bdd/>