Course: Code Quality

Ali ZAINOUL <ali.zainoul.az@gmail.com>

CrystalClearCode
March 30, 2025

5. Atelier Pratique – Mise en Œuvre des Tests Unitaires
6. Q&A et Récapitulatif de la Journée
7. Retour sur la Journée 1 et Objectifs du Jour 2
8. Les Différents Types de Tests
   - Définition et Différenciation
   - Étude de Cas
9. Test Coverage et Test Quality
   - Introduction au Test Coverage
   - Atelier Pratique : Génération de Rapports de Coverage
   - Supervision de la Qualité des Tests
10. Méthode TDD (Test Driven Development)
    - Présentation Théorique du TDD
    - Atelier Pratique du TDD
11. Méthode BDD (Behavior-Driven Development)
    - Introduction à la BDD
12. Conclusion et Clôture de la Formation

** Qualité de Code et Tests Unitaires **

Plan de cours

- Présentation des objectifs de la formation.
- Présentation de l'agenda et des modalités pédagogiques (alternance théorie/pratique, travaux en groupe).

Histoire : Échecs de tests ayant coûté cher

Exemple 1 : Ariane 5
Le vol inaugural d'Ariane 5 a échoué en raison d'une erreur de conversion numérique dans le logiciel, entraînant l'autodestruction de la fusée. Cet échec a coûté plusieurs centaines de millions d'euros.
Source : `https://en.wikipedia.org/wiki/Ariane_5`

Exemple 2 : Boeing 737 MAX
Des tests insuffisants sur le système MCAS ont contribué aux accidents tragiques du Boeing 737 MAX, causant la perte de centaines de vies et d'importantes pertes financières pour le constructeur.
Source : `https://en.wikipedia.org/wiki/Lion_Air_Flight_610`

Conventions de Code

- **Normes et Standards :**
  - **PEP-8** pour Python[1] : Guide officiel de style Python.
  - **PSR** pour PHP[2] : Standards recommandés pour le développement en PHP.
- **Importance :**
  - Assurer la lisibilité et la maintenabilité du code.
  - Faciliter la collaboration entre développeurs.
- **Références :**
  - Livre *Clean Code* de Robert C. Martin.
  - Articles et tutoriels sur les bonnes pratiques de codage.

[1] `https://www.python.org/dev/peps/pep-0008/`
[2] `https://www.php-fig.org/psr/`

Exemple 1: l'indentation tu respecteras

L'exemple qui suit montre le non-respect du standard PEP-8 en termes d'indentation:

```
# Example of code where indentation is not respected

def calculate_salary(base, bonus):
total = base + bonus
return total
```

Exemple 2: la clarté tu favoriseras

L'exemple qui suit montre le non-respect du standard PEP-8 en termes de clarté du code:

```
# Example of code where clarity is missing

def f(a, b):
    # No descriptive names or comments to explain the operation
    return a >> b

result = f(5, 3)
print(result)
```

Exemple 3: le nom ”parfait” tu donneras

L’exemple qui suit montre le non-respect du standard PEP-8 en termes de nommage de variables, de méthodes, de classes et de modules & packages:

```
# Example of code where naming conventions are not respected

# Importing a class from a poorly named package and module
from MyModule.bad_package import contact_manager

Obj = contact_manager()
Obj.AddContact("Badd, Example!")
```

Essentiels du standard PEP-8

- **Indentation** : Utilisez 4 espaces par niveau et assurez-vous que vos tabulations en Python sont configurées pour respecter cette convention.
- **Nommage des variables et fonctions** : Adopter le *snake_case* (ex. `user_name`, `calculate_total`).
- **Nommage des modules et des packages** Respecter le snake_case.
- **Nommage des classes** : Utiliser le *CamelCase* (ex. `ContactManager`).
- **Espacement** : Ajouter des espaces autour des opérateurs et après les virgules pour une meilleure lisibilité.

- **Longueur des lignes** : Limiter à 79 caractères par ligne.
- **Annotations de type** : Utiliser des annotations pour clarifier les signatures de fonctions (ex. `def add_contact(self, name: str) -> bool:`).
- **Commentaires et documentation** : Rédiger des commentaires clairs et concis en anglais, et maintenir une documentation à jour.

Atelier pratique

- **Objectif :** Implémentez une classe `ContactManager` en Python qui gère une liste de contacts.
- **Instructions :**
  - Créez une classe `ContactManager` avec un attribut interne `contacts` initialisé comme une liste vide.
  - Implémentez les méthodes suivantes :
    - `add_contact(contact: str) -> None` : ajoute un contact à la liste.
    - `remove_contact(contact: str) -> bool` : supprime le contact s'il existe et retourne `True`, sinon retourne `False`.
    - `list_contacts() -> list` : retourne la liste des contacts.
  - Respectez les conventions PEP-8 (indentation, nommage, etc.).
  - Rédigez un script principal pour démontrer l'utilisation de cette classe : ajouter quelques contacts (faker), les lister, puis supprimer un contact et réafficher la liste.

Introduction aux Linters

- **Définition** : Outils d'analyse statique[3] qui vérifient le respect des conventions de code.
- **Rôle** :
  - Détecter les erreurs de syntaxe et les incohérences.
  - Encourager l'application des bonnes pratiques de développement.
- **Sources** :
  - Documentation de PyLint[4].
  - Documentation d'ESLint[5].

[3] **Analyse statique de code** : Processus d'examen du code source sans l'exécuter, visant à détecter des erreurs, des vulnérabilités et à vérifier la conformité aux standards de développement.

[4] `https://pylint.pycqa.org/`

[5] `https://eslint.org/`

Présentation et Démonstration de PyLint

- **Outils présentés** :
  - **PyLint** et **Black** pour Python.

- **Démonstration** :
  - Analyse en direct d'un projet avec les linters.
  - Visualisation des erreurs et suggestions de correction.

Exercice Pratique sur les Linters

- Correction d'un projet simple grâce aux linters.
- Application de **PyLint** et **Black** pour améliorer le formatage et la qualité du code.
- Discussion sur les corrections apportées et les retours d'expérience.

Introduction aux Tests Unitaires

- **Définition** : Vérification du bon fonctionnement des plus petites unités de code (fonctions, méthodes, classes).
- **Objectifs** :
  - Détecter rapidement les erreurs.
  - Faciliter le débogage et la maintenance.
- **Méthodologies** :
  - Approche TDD (Test Driven Development).
  - Utilisation de mocks et stubs[6] pour isoler les unités.
- **Sources** :
  - Tutoriels sur les tests unitaires et documentation associée.

[6] Les *stubs* sont des morceaux de code utilisés pour simuler des modules ou des fonctions non encore implémentés dans un programme. Voir Wikipédia : `https://fr.wikipedia.org/wiki/Bouchon_(informatique)`

Exemple des stubs en programmation

```python
class BankService:
    def get_balance(self, user_id):
        """Simulate a call to an external bank service API"""
        raise NotImplementedError("This method should interact
            with an external API.")

# Stub Test
def test_get_balance_with_stub():
    bank_service = BankService()
    bank_service.get_balance = lambda user_id: 1000
    # Stub : always returns 1000
    assert bank_service.get_balance(123) == 1000
```

Exemple des mocks en programmation

```python
from unittest.mock import Mock

# Mocking Test
def test_get_balance_with_mock():
    bank_service = Mock()
    bank_service.get_balance.return_value = 2000
    # Simulate a return value of 2000

    # Verification
    assert bank_service.get_balance(456) == 2000

    # Verification that the method have indeed been called
    bank_service.get_balance.assert_called_once_with(456)
```

Frameworks de Test

- **Python :** PyTest[7].
- **Java :** JUnit[8].
- **PHP :** PHPUnit[9].
- **NodeJS :** JEST[10].

[7] https://docs.pytest.org/
[8] https://junit.org/
[9] https://phpunit.de/
[10] https://jestjs.io/

Module `pytest`

- Le module `pytest` est un framework de test populaire et flexible pour Python.
- Il permet d'écrire des tests simples et complexes de manière concise.
- `pytest` offre des fonctionnalités avancées telles que la découverte automatique des tests, la gestion des fixtures et des rapports détaillés.
- Il supporte également les tests de performance et les tests de manière asynchrone.

Exemple d'utilisation

```python
# Test example with pytest

def test_upper():
    assert 'hello'.upper() == 'HELLO'

def test_isupper():
    assert 'HELLO'.isupper()
    assert not 'Hello'.isupper()
```

- Il suffit de naviguer dans le dossier où se trouve le fichier des tests unitaires (exemple: `test_myfile.py`) et de lancer la commande: `pytest` afin de lancer les tests unitaires.

Utilisation des Fixtures

- Les fixtures permettent de configurer un état initial avant d'exécuter des tests.
- Elles peuvent être partagées entre plusieurs tests, ce qui évite la répétition de code.

```python
import pytest

@pytest.fixture
def sample_data():
    return [1, 2, 3]

def test_sum(sample_data):
    assert sum(sample_data) == 6
```

Démonstration : ContactManager en Python

- **Projet Exemple : ContactManager**
  - Une classe unique `ContactManager` qui gère des opérations CRUD.
  - **Méthodes typiques** :
    - `add_contact(contact: str)` : Ajouter un contact.
    - `get_contact(contact: str)` : Récupérer les informations d'un contact.
    - `update_contact(contact: str, new_info: str)` : Mettre à jour les informations d'un contact.
    - `delete_contact(contact: str)` : Supprimer un contact.
    - `remove_contact(contact: str) -> bool` : Supprime le contact s'il existe et retourne `True`, sinon retourne `False`.
    - `list_contacts() -> list` : Liste tous les contacts.

Test Unitaire et Discussion

- **Test Unitaire** :
  - Utilisation de **PyTest** pour tester chaque méthode indépendamment.
  - Tests des méthodes :
    - Test de `add_contact()` : Vérification que l'ajout d'un contact modifie correctement l'état interne de l'objet.
    - Test de `get_contact()` : Vérification qu'un contact peut être récupéré correctement après ajout.
    - Test de `update_contact()` : Vérification que la mise à jour des informations d'un contact fonctionne correctement.
    - Test de `delete_contact()` : Vérification que la suppression d'un contact fonctionne correctement.
    - Test de `remove_contact()` : Vérification que la suppression d'un contact existant retourne `True`, sinon `False`.
    - Test de `list_contacts()` : Vérification que tous les contacts sont listés correctement.

Atelier Pratique – Tests Unitaires

- Travail en binômes ou en petits groupes.
- Implémentation des tests unitaires sur des exemples de code fournis.
- Application concrète sur le projet **ContactManager** pour vérifier les opérations CRUD.
- Échange sur les bonnes pratiques et retour collectif sur les difficultés rencontrées.

Q&A et Récapitulatif

- **Discussion** :
  - Avantages de tester chaque méthode de manière isolée.
  - Impact sur la maintenance et l'évolution du code.
- **Sources** :
  - Documentation PyTest[11].
  - Tutoriels sur les tests unitaires en Python.
- Synthèse des points clés abordés durant la journée.
- Session de questions / réponses pour clarifier d'éventuelles zones d'ombre.
- Discussion sur l'importance d'une démarche qualité continue et des outils de tests.

[11] `https://docs.pytest.org/`

** Tests d'Intégration, Coverage et Méthodes TDD/BDD **

Retour sur la Journée 1 et Objectifs du Jour 2

- Rappel des concepts abordés lors de la première journée : qualité de code, conventions, linters et tests unitaires.
- Introduction des nouveaux thèmes :
  - Tests d'intégration et tests fonctionnels.
  - Mesure de la couverture de tests (Test Coverage) et qualité des tests.
  - Approfondissement des méthodes TDD et BDD.
- Réutilisation du projet **ContactManager** pour illustrer les concepts.

Définition et Différenciation des Tests

- **Tests Unitaires** : Vérifient le bon fonctionnement de chaque méthode ou fonction de manière isolée.
- **Tests d'Intégration** : Valident l'interaction entre plusieurs composants ou modules, par exemple, l'interaction entre le gestionnaire de contacts et une base de données fictive.

Définition et Différenciation des Tests (suite)

- **Tests Fonctionnels** : Simulent des scénarios utilisateur pour vérifier le comportement global de l'application.
- **Tests de Régression** : Vérifient que les modifications apportées à un programme, telles que la correction de bugs ou l'ajout de nouvelles fonctionnalités, n'ont pas introduit de nouveaux défauts dans les fonctionnalités déjà existantes. Ils permettent de s'assurer que le comportement attendu du logiciel reste inchangé après des modifications dans le code.

Sources :

- `https://www.softwaretestinghelp.com/types-of-software-testing/`

Étude de Cas

- **Scénario réel** : Utilisation du projet **ContactManager**.
- **Exemple** :
  - Un test unitaire pour chaque méthode CRUD.
  - Un test d'intégration pour vérifier l'interaction entre `ContactManager` et le module de persistance (fichier ou base de données simulée).
  - Un test fonctionnel simulant l'ajout, la modification et la suppression d'un contact via une interface utilisateur.
- Analyse des avantages et limites de chaque type de test.

Introduction au Test Coverage

- **Définition** : Mesure la part de code exécutée lors de l'exécution de la suite de tests.
- **Importance** :
  - Permet d'identifier les zones non testées.
  - Contribue à améliorer la qualité globale du code.
- **Outils** :
  - **pytest-cov** pour Python[12].

[12] `https://pytest-cov.readthedocs.io/en/latest/`

Atelier Pratique - Génération de Rapports de Coverage

- **Objectif** : Générer et analyser des rapports de couverture en HTML et en PDF.
- **Exercice** :
  - Utilisation de **pytest-cov** sur le projet **ContactManager**.
  - Export et analyse du rapport de couverture.
- **Références** :
  - Documentation **pytest-cov**.

Supervision de la Qualité des Tests

- **Outils et Méthodes :**
  - Intégration de rapports de coverage dans des outils CI/CD (ex. Jenkins, GitLab CI).
  - Analyse continue de la qualité des tests et du code.
- **Bénéfices :**
  - Détection précoce des régressions.
  - Amélioration de la maintenabilité du projet.
- **Sources :**
  - Articles sur la supervision des tests et outils d'intégration continue.

Présentation Théorique du TDD

- **Concepts Clés :**
  - Écrire d'abord le test qui définit la fonctionnalité.
  - Implémenter le code pour faire passer le test.
  - Refactoriser le code tout en conservant la validité des tests.
- **Avantages :**
  - Conception orientée test.
  - Réduction des bugs et amélioration de la qualité du code.
- **Références :**
  - *Test Driven Development: By Example* de Kent Beck[13].

[13] https://www.agilealliance.org/glossary/tdd/

Atelier Pratique du TDD

- **Exercice :**
  - Rédiger un test pour une nouvelle fonctionnalité dans le projet **ContactManager** (exemple : ajout d'une vérification d'un numéro de téléphone).
  - Implémenter la fonctionnalité pour faire passer le test.
  - Refactoriser le code en respectant les principes du TDD.
- **Bénéfices :**
  - Développement itératif et sécurisé.
  - Documentation vivante du code.

Introduction à la BDD

- **Concepts de base** :
  - Collaboration entre développeurs, testeurs et métiers.
  - Rédaction de scénarios compréhensibles par tous avec le langage **Gherkin**.
- **Outils associés** :
  - **Behave** pour l'exécution des scénarios BDD ...
- **Avantages** :
  - Amélioration de la communication et de la compréhension des besoins.
  - Tests alignés sur les exigences métiers.

Conclusion et Clôture de la Formation

- **Synthèse** des concepts abordés sur les tests d'intégration, la couverture de tests, le TDD et la BDD.
- **Récapitulatif** des points forts et des apprentissages réalisés avec le projet **ContactManager**.
- **Session de Q&A** : Échanges et retour d'expérience sur l'ensemble de la formation.
- **Évaluation** de la formation et discussion sur les prochaines étapes.