TP3 - Organisation et Automatisation des Tests

Objectifs

- Apprendre à organiser un projet avec une structure de tests propre
- Maîtriser les conventions de nommage et d'arborescence
- Automatiser l'exécution des tests avec pytest
- Générer et analyser des rapports de couverture
- Découvrir l'intégration continue basique

Prérequis

- Python 3.6+ installé
- Installation des outils : `pip install pytest pytest-cov`

Partie 1 : Créer la structure du projet

Étape 1 : Analyser une structure mal organisée

Créez d'abord cette structure **problématique** pour comprendre les enjeux :

```
mauvais_projet/
├── calculs.py
├── utilisateurs.py
├── test1.py
├── test_calcul_bidule.py
├── TestUtilisateur.py
└── autre_test.py
```

Questions de réflexion :

1. Quels problèmes voyez-vous dans cette organisation ?
2. Comment un nouveau développeur s'y retrouverait-il ?
3. Comment lancer tous les tests d'un coup ?

Étape 2 : Créer la structure recommandée

Maintenant, créez cette structure **propre** :

```
bibliotheque_projet/
├── src/
│   └── bibliotheque/
│       ├── __init__.py
│       ├── book.py          # À créer
│       ├── library.py       # À créer
│       └── user.py          # À créer
```

```
├── tests/
│   ├── __init__.py
│   └── test_book.py          # À créer
│   └── test_library.py       # À créer
│   └── test_user.py          # À créer
├── requirements.txt          # À créer
├── pytest.ini               # À créer
└── README.md                 # À créer
```

Mission : Créez tous ces dossiers et fichiers vides. Observez la différence avec la première structure.

Partie 2 : Développer le code métier

Étape 3 : Créer la classe `Book`

Dans `src/bibliotheque/book.py`, implémentez cette classe étape par étape :

```python
class Book:
    """Représente un livre dans la bibliothèque"""

    def __init__(self, title, author, isbn):
        # TODO: Initialisez les attributs
        # - Validez que title et author sont non vides
        # - Validez que isbn a exactement 13 caractères
        # - Levez ValueError si conditions non respectées
        pass

    def is_available(self):
        # TODO: Retournez True si le livre n'est pas emprunté
        pass

    def borrow(self):
        # TODO: Marquez le livre comme emprunté
        # Retournez True si succès, False si déjà emprunté
        pass

    def return_book(self):
        # TODO: Marquez le livre comme disponible
        # Retournez True si succès, False si pas emprunté
        pass
```

Instructions détaillées :

- `__init__` doit valider les paramètres et lever `ValueError` avec message explicite
- Ajoutez un attribut `borrowed` (booléen) pour suivre l'état
- Implémentez la logique métier des emprunts/retours

Étape 4 : Créer la classe `User`

Dans `src/bibliotheque/user.py` :

```
class User:
    """Représente un utilisateur de la bibliothèque"""

    def __init__(self, name, email):
        # TODO: Validez email (doit contenir @)
        # TODO: Validez name (non vide)
        pass

    def can_borrow(self, max_books=3):
        # TODO: Vérifiez si l'utilisateur peut emprunter
        # (limite de max_books livres)
        pass

    def add_borrowed_book(self, book):
        # TODO: Ajoutez un livre à la liste des emprunts
        pass

    def remove_borrowed_book(self, book):
        # TODO: Retirez un livre de la liste des emprunts
        pass
```

À implémenter :

- Liste `borrowed_books` pour suivre les emprunts
- Validation d'email basique (contient @)
- Logique de limite d'emprunts

Étape 5 : Créer la classe `Library`

Dans `src/bibliotheque/library.py` :

```
from .book import Book
from .user import User

class Library:
    """Gestionnaire de bibliothèque"""

    def __init__(self, name):
        # TODO: Initialisez nom et collections
        pass

    def add_book(self, book):
        # TODO: Ajoutez un livre à la collection
        pass

    def find_book_by_isbn(self, isbn):
        # TODO: Trouvez un livre par ISBN
        pass
```

```
    def borrow_book(self, user, isbn):
        # TODO: Gérez l'emprunt complet
        # 1. Trouvez le livre
        # 2. Vérifiez que user peut emprunter
        # 3. Vérifiez que le livre est disponible
        # 4. Effectuez l'emprunt
        # Retournez True/False selon le succès
        pass
```

Partie 3 : Écrire les tests organisés

Étape 6 : Tester la classe `Book`

Dans `tests/test_book.py`, suivez cette structure :

```
import pytest
from src.bibliotheque.book import Book

class TestBookCreation:
    """Tests de création de livre"""

    def test_create_valid_book(self):
        """Test création livre valide"""
        # TODO: Créez un livre avec paramètres valides
        # TODO: Vérifiez que les attributs sont correctement assignés
        pass

    def test_create_book_empty_title_raises_error(self):
        """Test titre vide lève une erreur"""
        # TODO: Utilisez pytest.raises(ValueError) pour tester l'exception
        pass

    def test_create_book_invalid_isbn_raises_error(self):
        """Test ISBN invalide lève une erreur"""
        # TODO: Testez avec ISBN trop court et trop long
        pass

class TestBookBorrowing:
    """Tests d'emprunt de livre"""

    def setup_method(self):
        """Fixture : prépare un livre pour chaque test"""
        # TODO: Créez self.book avec des données valides
        pass

    def test_new_book_is_available(self):
        """Test livre neuf disponible"""
        # TODO: Vérifiez que is_available() retourne True
        pass
```

```
    def test_borrow_available_book_success(self):
        """Test emprunt livre disponible"""
        # TODO: Empruntez le livre
        # TODO: Vérifiez que borrow() retourne True
        # TODO: Vérifiez que is_available() retourne False
        pass

    def test_borrow_already_borrowed_book_fails(self):
        """Test emprunt livre déjà emprunté"""
        # TODO: Empruntez une première fois
        # TODO: Tentez d'emprunter une seconde fois
        # TODO: Vérifiez que la seconde tentative retourne False
        pass
```

Conseils d'implémentation :

- Utilisez `pytest.raises(ValueError)` pour tester les exceptions
- Nommez vos tests de façon explicite : `test_what_when_expected`
- Séparez les tests par fonctionnalité avec des classes

Étape 7 : Tester les interactions complexes

Dans `tests/test_library.py` :

```
import pytest
from src.bibliotheque.library import Library
from src.bibliotheque.book import Book
from src.bibliotheque.user import User

class TestLibraryOperations:

    def setup_method(self):
        """Fixture complexe : bibliothèque avec livres et utilisateurs"""
        # TODO: Créez une bibliothèque
        # TODO: Créez 2-3 livres avec différents ISBN
        # TODO: Créez 2 utilisateurs
        # TODO: Ajoutez les livres à la bibliothèque
        pass

    def test_borrow_flow_success(self):
        """Test flux complet d'emprunt réussi"""
        # TODO: Empruntez un livre
        # TODO: Vérifiez tous les changements d'état
        pass

    def test_user_cannot_borrow_more_than_limit(self):
        """Test limite d'emprunts par utilisateur"""
        # TODO: Faites emprunter 3 livres (limite)
        # TODO: Tentez un 4ème emprunt
```

```
        # TODO: Vérifiez que c'est refusé
        pass
```

Partie 4 : Configuration et outils

Étape 8 : Configurer pytest

Créez `pytest.ini` avec cette configuration :

```
[tool:pytest]
# TODO: Configurez le répertoire des tests
testpaths = tests

# TODO: Ajoutez des options par défaut
addopts = -v --tb=short

# TODO: Configurez les marqueurs de tests
markers =
    slow: marks tests as slow
    integration: marks tests as integration tests
```

Recherchez et complétez :

- Quelle option affiche plus de détails ? (verbose)
- Comment ignorer les warnings ? (--disable-warnings)

Étape 9 : Créer le fichier requirements

Dans `requirements.txt` :

```
# TODO: Listez les dépendances de production
# (aucune pour ce projet simple)

# TODO: Listez les dépendances de développement
pytest>=7.0.0
pytest-cov>=4.0.0
```

Partie 5 : Lancement et couverture

Étape 10 : Lancer les tests de différentes façons

Testez ces commandes et notez les différences :

```
# Commande 1
pytest
```

```
# Commande 2
pytest -v

# Commande 3
pytest tests/test_book.py

# Commande 4
pytest tests/test_book.py::TestBookCreation::test_create_valid_book
```

Mission : Lancez chaque commande et décrivez ce qui change dans l'affichage.

Étape 11 : Générer le rapport de couverture

Suivez ces étapes :

1. **Lancez avec couverture :**

```
pytest --cov=src/bibliotheque
```

2. **Générez le rapport HTML :**

```
pytest --cov=src/bibliotheque --cov-report=html
```

3. **Analysez le rapport :**

- Ouvrez `htmlcov/index.html` dans votre navigateur
- Cliquez sur chaque fichier pour voir les lignes non testées

Questions d'analyse :

- Quel pourcentage de couverture avez-vous ?
- Quelles lignes ne sont pas testées ? Pourquoi ?
- Comment améliorer la couverture ?

Étape 12 : Améliorer la couverture

Identifiez les lignes non testées et ajoutez les tests manquants :

```
# Dans test_book.py, ajoutez :
def test_return_book_not_borrowed_fails(self):
    """Test retour livre non emprunté"""
    # TODO: Tentez de retourner un livre jamais emprunté
    # TODO: Vérifiez que return_book() retourne False
    pass

def test_return_borrowed_book_success(self):
```

```
    """Test retour livre emprunté"""
    # TODO: Empruntez puis retournez un livre
    # TODO: Vérifiez que return_book() retourne True
    # TODO: Vérifiez que le livre redevient disponible
    pass
```

Objectif : Atteignez au moins 90% de couverture.

Partie 6 : Automatisation avancée

Étape 13 : Créer un Makefile simple

Créez un `Makefile` :

```
# TODO: Cible pour installer les dépendances
install:
    pip install -r requirements.txt

# TODO: Cible pour lancer les tests
test:
    pytest

# TODO: Cible pour la couverture
coverage:
    pytest --cov=src/bibliotheque --cov-report=html

# TODO: Cible pour nettoyer
clean:
    rm -rf htmlcov/ .coverage .pytest_cache/
```

Test : Lancez `make test` et `make coverage`

Étape 14 : GitHub Actions basique

Créez `.github/workflows/tests.yml` :

```
name: Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python
```

```
      uses: actions/setup-python@v4
      with:
        python-version: '3.9'

    # TODO: Étape pour installer les dépendances

    # TODO: Étape pour lancer les tests

    # TODO: Étape pour générer la couverture
```

Mission : Complétez les étapes manquantes en vous inspirant du Makefile.

Partie 7 : Analyse et bonnes pratiques

Étape 15 : Analyser votre organisation

Répondez à ces questions :

1. **Structure des tests :**
   - Vos tests suivent-ils la même organisation que le code source ?
   - Les noms de fichiers sont-ils cohérents ?
2. **Nommage des tests :**
   - Peut-on comprendre ce que teste chaque fonction sans lire le code ?
   - Les classes de test regroupent-elles logiquement les tests ?
3. **Couverture :**
   - Quelles parties du code ne sont jamais testées ?
   - Y a-t-il des tests redondants ?
4. **Automatisation :**
   - Combien de commandes faut-il pour lancer tous les tests ?
   - Un nouveau développeur peut-il facilement contribuer ?

Étape 16 : Refactoring des tests

Identifiez les améliorations possibles :

```python
# Avant : duplication
def test_borrow_book_user1(self):
    user = User("John", "john@email.com")
    book = Book("Title", "Author", "1234567890123")
    # ... test logic

def test_borrow_book_user2(self):
    user = User("Jane", "jane@email.com")  # Duplication !
    book = Book("Title", "Author", "1234567890123")  # Duplication !
```

```
    # ... test logic

# Après : fixtures
@pytest.fixture
def sample_user():
    return User("John", "john@email.com")

@pytest.fixture
def sample_book():
    return Book("Title", "Author", "1234567890123")

def test_borrow_book_success(self, sample_user, sample_book):
    # ... test logic sans duplication
```

Mission : Refactorisez vos tests pour éliminer la duplication.

Partie 8 : Rapport final

Étape 17 : Générer le rapport complet

Lancez cette séquence complète :

```
# Nettoyage
make clean

# Tests avec couverture détaillée
pytest --cov=src/bibliotheque --cov-report=html --cov-report=term-missing

# Analyse du rapport
```

Étape 18 : Documentation du projet

Complétez le `README.md` :

```
# Projet Bibliothèque

## Installation
# TODO: Instructions d'installation

## Tests
# TODO: Comment lancer les tests

## Couverture
# TODO: Comment générer la couverture

## Structure
# TODO: Expliquez l'organisation du code
```

Questions de réflexion finale

1. **Organisation :** Quels avantages voyez-vous à cette structure par rapport à un projet mal organisé ?
2. **Automatisation :** Comment ces outils facilitent-ils le travail en équipe ?
3. **Couverture :** Un taux de couverture élevé garantit-il un code sans bugs ? Pourquoi ?
4. **Intégration continue :** Comment GitHub Actions peut-il prévenir les régressions ?
5. **Maintenance :** Comment cette organisation facilite-t-elle l'ajout de nouvelles fonctionnalités ?

Pour aller plus loin

Outils avancés à explorer

- **tox** : Tests sur plusieurs versions Python
- **pre-commit** : Hooks Git pour la qualité
- **mypy** : Vérification de types statique
- **black** : Formatage automatique du code

Métriques avancées

- **Couverture de branches** : `--cov-branch`
- **Tests de mutation** : `mutmut`
- **Complexité cyclomatique** : `xenon`

Félicitations ! Vous maîtrisez maintenant l'organisation et l'automatisation des tests ! 🚀