Rappels de TDD et Clean Code

Cycle TDD

Red → Green → Refactor :

1. **Red** : écrire un test qui échoue.
2. **Green** : coder le minimum pour passer le test.
3. **Refactor** : améliorer le code sans casser les tests.

Principes Clean Code essentiels

— Noms explicites pour fonctions, classes et variables.

— Fonctions courtes et simples.

— Séparation des responsabilités (Single Responsibility Principle).

— Eviter les répétitions (DRY — Don't Repeat Yourself).

— Commentaires utiles seulement pour expliquer le *pourquoi*, pas le *comment*.

— Tests unitaires couvrants et indépendants.

Exemples guidés

Exemple 1 — Fonction simple avec TDD

Test (Red)

```python
# test_main.py
import unittest
from main import addition

class TestAddition(unittest.TestCase):
    def test_addition(self):
        self.assertEqual(addition(2, 3), 5)

if __name__ == "__main__":
    unittest.main()
```

Code minimal (Green)

```python
# main.py
def addition(a, b):
    return a + b
```


Exemple 2 — Refactor et validation

Ajouter vérification des types

```
def addition(a, b):
    if not isinstance(a, (int, float)) or not isinstance(b, (int,
       float)):
        raise TypeError("Les param tres doivent  tre  num riques
           ")
    return a + b
```

Exercices guidés (avec solution)

Exercice 1 — Gestion de panier simple

— Créer une classe `Panier` avec : ajouter, retirer, total.

— Écrire les tests avant le code.

Solution rapide

```
class Panier:
    def __init__(self):
        self.articles = {}

    def ajouter(self, nom, prix):
        self.articles[nom] = prix

    def retirer(self, nom):
        if nom in self.articles:
            del self.articles[nom]

    def total(self):
        return sum(self.articles.values())
```

Exercice 2 — Calculatrice avec TDD

— Fonctions : addition, soustraction, multiplication, division.

— Lever une exception pour la division par zéro.

Solution

```
def division(a, b):
    if b == 0:
        raise ZeroDivisionError("Division par z ro interdite")
    return a / b
```


Exercices challengés (sans solution)

1. Implémentez une classe `CompteBancaire` :
   — Méthodes : `deposer(montant)`, `retirer(montant)`, `solde()`.
   — Tests unitaires pour chaque méthode.
   — Vérifier que le solde ne devient jamais négatif.
2. Créez une fonction `fibonacci(n)` utilisant TDD.
3. Refactorez la fonction `fibonacci` pour la rendre plus lisible et efficiente.
4. Implémentez une classe `TodoList` :
   — Ajouter une tâche, marquer terminée, supprimer tâche.
   — Écrire des tests pour tous les scénarios.
5. Créez un module `MathUtils` avec : `factorial`, `is`$_{p}rime$, gcd.$TDDobligatoire$.
5. Implémentez une fonction qui lit un fichier CSV et retourne un dictionnaire, testez la fonction avant de coder.
6. Refactorisez un code donné avec mauvaise lisibilité et doublons.
7. Implémentez une classe Logger simple qui garde les logs en mémoire et écrit dans un fichier.
8. Créez une classe Livre et une classe Bibliotheque :
   - La bibliothèque doit permettre d'ajouter, retirer, rechercher des livres.
   - Respectez les principes SOLID.
9. Implémentez un petit jeu Python (ex: pierre-papier-ciseaux) avec TDD, en testant toutes les combinaisons.
10. Créez un décorateur Python @retry qui relance une fonction en cas d'exception. Testez-le avant d'implémenter.
11. Refactorisez une classe Personne avec noms et âges pour utiliser des propriétés, tests avant/après.
12. Créez une fonction normalize$_{t}extquinettoieuntexte(minuscules, suppressionponctuation), a$

12. Implémentez un système de vote simple :
   - Ajouter candidats, voter, compter les votes.
   - Tests unitaires obligatoires.
13. Challenge final : combinez plusieurs classes (Panier, CompteBancaire, Logger) dans un mini-projet testable.

Conclusion

— Les tests unitaires guident le développement et assurent la qualité.

— Clean Code et TDD permettent un code lisible, fiable et maintenable.

— Les exercices challengés renforcent la pratique autonome et l'application des principes SOLID et DRY.

"Écrivez les tests d'abord, codez avec clarté, refactorez avec rigueur."