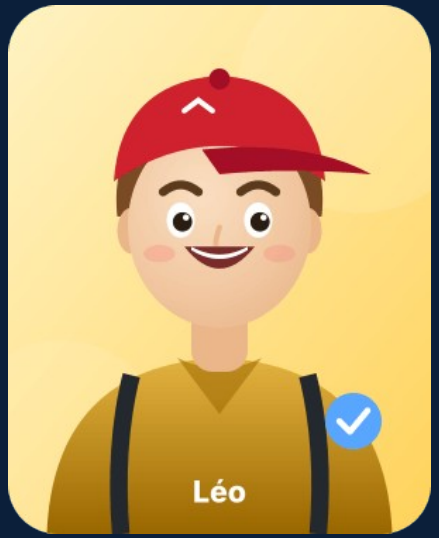
HEF-110 · MISSION 08 / 10

TDD et refactoring



Écrire le test avant le code, avancer par petits pas, nettoyer sans peur : un rythme qui rend la conception plus simple et le code plus sûr.

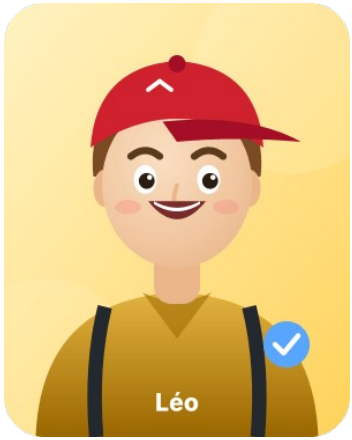


Badge : Danseur rouge-vert

110 min · intermédiaire

L'histoire

Kodi doit écrire un convertisseur de nombres en chiffres romains pour les certificats de HashBox. Il ouvre l'éditeur et commence par la boucle la plus compliquée, avec les cas spéciaux (4, 9, 40, 90…). Trois heures plus tard, 60 lignes de `if` imbriqués, des bugs dans 1994, et plus personne n'ose y toucher.



> *« Autre méthode, propose Sam. Écris un seul test tout simple. Regarde-le échouer. Fais-le passer avec le code le plus bête possible. Puis le test suivant. Le code grossit par petits pas, toujours vérifié. Et à chaque fois qu'il est vert, tu peux nettoyer. »*

Tu apprends le cycle rouge-vert-refactor, à avancer par petits pas, à utiliser des tests comme filet de sécurité pour refactorer, et à apprivoiser du code existant sans tests.

Ta mission

À la fin, je serai capable de…

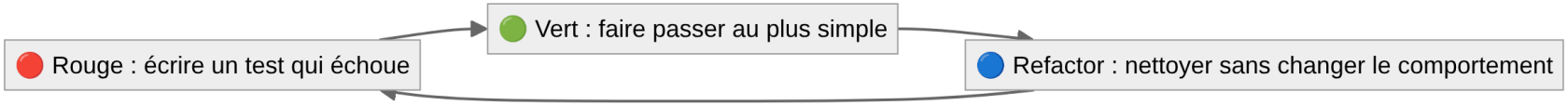
1. pratiquer le cycle rouge → vert → refactor
2. avancer par petits pas avec la solution la plus simple
3. refactorer sous la protection d'une suite de tests
4. écrire des tests de caractérisation pour du code existant
5. identifier les cas où le TDD aide et ceux où il aide moins

Défi d'échauffement

Tu construis un pont de cordes : à chaque planche posée, tu tires dessus pour vérifier qu'elle tient avant d'avancer. Compare avec quelqu'un qui pose 200 planches puis teste en marchant dessus. Que se passe-t-il quand une planche cède au milieu ? Où est le défaut ?

Le concept

Le cycle du TDD



Description du schéma : trois étapes en boucle : rouge (écrire un test qui échoue), vert (écrire le code le plus simple qui le fait passer), refactor (améliorer le code en gardant les tests verts), puis retour au rouge pour le comportement suivant.

Étape	Règle
Rouge	un test **qui échoue pour la bonne raison**, petit, qui exprime un seul comportement nouveau

Le concept

Le cycle du TDD (suite)

Étape	Règle
Vert	le **code le plus simple** qui passe tous les tests, même « bête » (renvoyer une constante)
Refactor	supprimer la duplication, clarifier les noms, simplifier — **sans nouveau comportement**, tests toujours verts

Les trois lois (Robert C. Martin)

1. N'écris pas de code de production sans un test qui échoue.
2. N'écris pas plus de test qu'il n'en faut pour échouer.
3. N'écris pas plus de code qu'il n'en faut pour passer le test.

Le concept

Pourquoi ça aide

- **Conception guidée** : tu écris d'abord l'**usage** (l'API), donc elle est plus agréable.
- **Petits pas** : un bug a été introduit dans les dernières minutes, donc il est facile à situer.
- **Couverture naturelle** : tout comportement a un test qui l'a motivé.
- **Confiance** : le refactoring devient routine.

Triangulation et « faux » vert

Pour éviter de coder trop tôt une règle générale, on peut d'abord faire passer avec une constante (`return 1`), puis ajouter un deuxième exemple qui **force** la généralisation : c'est la **triangulation**.

Le concept

Refactoring : changer la structure, pas le comportement

Un **refactoring** modifie la forme du code **sans changer** son comportement observable. Les tests en sont la preuve. Mouvements courants :

Mouvement	Exemple
Renommer	`x` → `nombre_restant`
Extraire une fonction	sortir un bloc nommé
Remplacer des `if` par une table	dictionnaire de correspondances
Supprimer la duplication	factoriser
Simplifier une condition	utiliser une clause de garde

Le concept

Refactoring : changer la structure, pas le comportement (suite)

Règle : un pas de refactoring à la fois, tests après chaque pas. Ne mélange **jamais** refactoring et changement de comportement dans le même pas.

Du code existant sans tests : caractériser

Avant de refactorer du code hérité :

1. appelle-le avec des entrées variées et **note ce qu'il renvoie réellement** ;
2. transforme ces observations en **tests de caractérisation** (ils fixent le comportement *actuel*, même s'il est bizarre) ;
3. refactore sous leur protection ;
4. corrige les vrais bugs séparément, avec de vrais tests.

Le concept

Limites du TDD

- Peu adapté à l'**exploration** pure (on ne sait pas encore ce qu'on veut) : on peut écrire un prototype jetable, puis recommencer en TDD.
- Les interfaces graphiques, l'apprentissage automatique, les systèmes physiques demandent d'autres stratégies.
- Ce n'est pas une religion : c'est un outil, très utile pour la **logique métier**.

🧠 ***Astuce*** *: si tu ne sais pas quel test écrire, tu ne sais pas encore ce que le code doit faire. Retourne à la mission 2 : décris le comportement avec un exemple.*

Exemple pas à pas

Problème : implémenter la conversion en chiffres romains en TDD, **étape par étape** (script qui rejoue le cycle), puis refactorer sous protection.

```
import io, unittest

def lancer(source_tests, source_code):
    espace = {"unittest": unittest}
    exec(source_code, espace)                                   # le code de produc
   tion
    exec(source_tests, espace)                                  # les tests (classe
    TestRomains)
    suite = unittest.defaultTestLoader.loadTestsFromTestCase(espace["TestRomains"]
   )
    return unittest.TextTestRunner(stream=io.StringIO()).run(suite)

T = "class TestRomains(unittest.TestCase):\n"
def tests(*cas):
    corps = T
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    for i, (n, attendu) in enumerate(cas):
        corps += f"    def test_{n}(self):\n        self.assertEqual(romain({n}),
   '{attendu}')\n"
    return corps

ETAPES = [
    # (cas de test cumulés, code de production « le plus simple »)
    ([(1, "I")],
     "def romain(n):\n    return 'I'\n"),                                    #
    constante !
    ([(1, "I"), (2, "II"), (3, "III")],
     "def romain(n):\n    return 'I' * n\n"),                               #
    généralisation forcée par 2 et 3
    ([(1, "I"), (2, "II"), (3, "III"), (5, "V"), (6, "VI"), (8, "VIII")],
     "def romain(n):\n    r = ''\n    if n >= 5:\n        r += 'V'; n -= 5\n    re
   turn r + 'I' * n\n"),
    ([(1, "I"), (3, "III"), (4, "IV"), (5, "V"), (6, "VI"), (9, "IX"), (10, "X"),
   (14, "XIV")],
     "def romain(n):\n    r = ''\n    if n >= 10:\n        r += 'X' * (n // 10); n
```

Exemple pas à pas (suite)

```
    %= 10\n    if n == 9:\n        return r + 'IX'\n    if n >= 5:\n        r += '
   V'; n -= 5\n    if n == 4:\n        return r + 'IV'\n    return r + 'I' * n\n")
   ,
]
for i, (cas, code) in enumerate(ETAPES, 1):
    precedent = ETAPES[i - 2][1] if i > 1 else "def romain(n):\n    return None\n"
    rouge = lancer(tests(*cas), precedent)                    # 🔴 le nouveau te
   st contre l'ANCIEN code
    vert = lancer(tests(*cas), code)                          # 🟢 puis contre l
   e nouveau code
    print(f"étape {i} : 🔴 {len(rouge.failures) + len(rouge.errors)} test(s) rouge
   (s)  →  🟢 {vert.testsRun} test(s) verts")
    assert not rouge.wasSuccessful() and vert.wasSuccessful()

# --- 🔵 Refactor : on remplace les if par une table. Le filet de tests dit si on
   a cassé quelque chose ---
CAS_COMPLETS = [(1, "I"), (4, "IV"), (9, "IX"), (14, "XIV"), (40, "XL"), (90, "XC"
   ), (400, "CD"), (1994, "MCMXCIV"), (2024, "MMXXIV"), (3999, "MMMCMXCIX")]
```

Exemple pas à pas

Exemple pas à pas (suite)

```
def romain_avant(n):                                          # version en casca
   de de if (étape 4 prolongée, partielle)
    r = ''
    for valeur, signe in ((1000, 'M'), (900, 'CM'), (500, 'D'), (400, 'CD'), (100,
    'C'), (90, 'XC'), (50, 'L'), (40, 'XL'), (10, 'X'), (9, 'IX'), (5, 'V'), (4, '
   IV'), (1, 'I')):
        while n >= valeur:
            r += signe
            n -= valeur
    return r

TABLE = [(1000, "M"), (900, "CM"), (500, "D"), (400, "CD"), (100, "C"), (90, "XC")
   , (50, "L"), (40, "XL"), (10, "X"), (9, "IX"), (5, "V"), (4, "IV"), (1, "I")]

def romain(n):
    if not 1 <= n <= 3999:
        raise ValueError("entre 1 et 3999")
    morceaux = []
    for valeur, signe in TABLE:
```

Exemple pas à pas (suite)

```
        fois, n = divmod(n, valeur)
        morceaux.append(signe * fois)
    return "".join(morceaux)

class TestRomains(unittest.TestCase):
    def test_cas_connus(self):
        for n, attendu in CAS_COMPLETS:
            with self.subTest(n=n):
                self.assertEqual(romain(n), attendu)
    def test_equivalence_avec_l_ancienne_version(self):
        for n in range(1, 4000):
            self.assertEqual(romain(n), romain_avant(n), n)
    def test_hors_limites(self):
        for n in (0, -1, 4000):
            with self.assertRaises(ValueError):
                romain(n)

r = unittest.TextTestRunner(stream=io.StringIO()).run(unittest.defaultTestLoader.l
   oadTestsFromTestCase(TestRomains))
```

Exemple pas à pas (suite)

```
print("🔵 refactor sous protection :", r.testsRun, "tests verts sur 3 999 nombres
   comparés")
assert r.wasSuccessful(), r.failures + r.errors
```

Lecture :

1. À chaque étape, on vérifie **deux choses** : le nouveau test est rouge contre l'ancien code (donc il teste bien du neuf), puis vert avec le nouveau.
2. La première implémentation est volontairement **bête** (`return 'I'`) ; ce sont les exemples qui obligent à généraliser (**triangulation**).
3. Le refactoring (une table au lieu de cascades de `if`) est protégé par un test d'équivalence sur **tous** les nombres de 1 à 3 999 : si l'on casse un détail, on le voit instantanément.
4. Le refactoring et le changement de comportement (validation hors limites) sont séparés dans le temps : la validation a été ajoutée avec son propre test rouge.

À toi, avec aide

Fais un mini-kata TDD : **FizzBuzz** paramétrable `fizzbuzz(n)` (3 → « Fizz », 5 → « Buzz », 15 → « FizzBuzz », sinon le nombre en texte). Écris la séquence de cycles dans l'ordre :

1. test pour 1 → « 1 » (le plus bête qui passe : `return "1"`) ;
2. test pour 2 → « 2 » (force `str(n)`) ;
3. test pour 3 → « Fizz » ;
4. test pour 5 → « Buzz » ;
5. test pour 15 → « FizzBuzz » ;
6. refactor : élimine la duplication avec une construction de texte par morceaux.

Montre le code après chaque étape (les tests doivent rester verts).

À toi, avec aide · réponse

Indices progressifs

Indice 1

À l'étape 5, vérifie 15 **avant** 3 et 5, ou construis `"Fizz" * (n%3==0) + "Buzz" * (n%5==0)`.

Indice 2

Le refactor final doit passer les mêmes 5 tests sans en modifier un seul.

Indice 3

Garde la liste des tests cumulés.

À toi, avec aide · réponse

Réponse

```
import unittest, io

def verifier(fizzbuzz):
    cas = [(1, "1"), (2, "2"), (3, "Fizz"), (4, "4"), (5, "Buzz"), (6, "Fizz"), (1
   0, "Buzz"), (15, "FizzBuzz"), (30, "FizzBuzz")]
    for n, attendu in cas:
        assert fizzbuzz(n) == attendu, (n, fizzbuzz(n), attendu)

# étape 1 : le plus bête
def v1(n): return "1"
# étape 2 : généralisation
def v2(n): return str(n)
# étape 3-4
def v3(n):
    if n % 3 == 0: return "Fizz"
    if n % 5 == 0: return "Buzz"
    return str(n)
# étape 5 : cas combiné
def v4(n):
```

À toi, avec aide · réponse

Réponse (suite)

```
    if n % 15 == 0: return "FizzBuzz"
    if n % 3 == 0: return "Fizz"
    if n % 5 == 0: return "Buzz"
    return str(n)
# étape 6 : refactor (même comportement, moins de duplication)
def v5(n):
    return ("Fizz" if n % 3 == 0 else "") + ("Buzz" if n % 5 == 0 else "") or str(
   n)

for v in (v4, v5):
    verifier(v)
for v in (v1, v2, v3):
    try:
        verifier(v); raise SystemExit("aurait dû être rouge")
    except AssertionError:
        pass
assert all(v4(n) == v5(n) for n in range(1, 1000))
print("les versions 1 à 3 sont rouges sur la suite complète, 4 et 5 sont vertes et
    équivalentes")
```

Réponse (suite)

Les versions intermédiaires sont **rouges** contre la suite complète (c'est normal : elles ne couvrent que les premiers tests). Le refactor v5 est prouvé équivalent à v4 sur 1 000 valeurs.

À toi, en solo

Caractériser puis refactorer du code existant. Voici une fonction « héritée » dont personne ne connaît toutes les règles :

```
def calc(a, b, t):
    if t == 1:
        if a > 100: r = a * 0.9
        else: r = a
    elif t == 2:
        r = a * b
        if r > 500: r = r - 50
    elif t == 3:
        r = a / b if b != 0 else 0
    else:
        r = -1
    return round(r, 2)
```

À toi, en solo (suite)

1. écris des **tests de caractérisation** à partir de ses sorties réelles (une grille de 30 entrées au moins, y compris `b = 0` et `t` inconnu) ;
2. refactore en trois fonctions nommées + un dispatch par table, **sans changer** le comportement ;
3. prouve l'équivalence sur 5 000 entrées aléatoires (graine fixée) ;
4. ajoute ensuite, séparément, un test qui révèle le comportement discutable de `b = 0` (renvoie 0) et décide ce qu'il faudrait faire (sans le changer sans accord).

À toi, en solo · réponse

Pistes de correction

```
import random

def calc_ancien(a, b, t):
    if t == 1:
        if a > 100: r = a * 0.9
        else: r = a
    elif t == 2:
        r = a * b
        if r > 500: r = r - 50
    elif t == 3:
        r = a / b if b != 0 else 0
    else:
        r = -1
    return round(r, 2)

# 1. tests de caractérisation : on enregistre ce que fait réellement le code
GRILLE = [(a, b, t) for a in (0, 50, 100, 101, 200) for b in (0, 2, 7) for t in (1
   , 2, 3, 9)]
OBSERVE = {entree: calc_ancien(*entree) for entree in GRILLE}
```

À toi, en solo · réponse

Pistes de correction (suite)

```
assert len(GRILLE) == 60 and OBSERVE[(0, 0, 3)] == 0 and OBSERVE[(50, 2, 9)] == -1

# 2. refactoring : fonctions nommées + table
def remise_gros_montant(a, b):
    return a * 0.9 if a > 100 else a

def produit_avec_rabais(a, b):
    r = a * b
    return r - 50 if r > 500 else r

def quotient_ou_zero(a, b):
    return a / b if b != 0 else 0

OPERATIONS = {1: remise_gros_montant, 2: produit_avec_rabais, 3: quotient_ou_zero}

def calc(a, b, t):
    operation = OPERATIONS.get(t)
    return round(operation(a, b), 2) if operation else -1
```

À toi, en solo · réponse

Pistes de correction (suite)

```
# 3. équivalence : la grille, puis l'aléatoire
for entree, attendu in OBSERVE.items():
    assert calc(*entree) == attendu, entree
rng = random.Random(5)
for _ in range(5000):
    entree = (rng.choice([0, rng.uniform(0, 300)]), rng.choice([0, rng.randint(-5,
    9), rng.uniform(1, 9)]), rng.choice([1, 2, 3, 4, 0]))
    assert calc(*entree) == calc_ancien(*entree), entree

# 4. comportement discutable, documenté par un test (sans le changer)
assert calc(10, 0, 3) == 0           # diviser par zéro renvoie 0 : à discuter (er
   reur explicite ? None ?)
print("60 cas caractérisés, 5000 aléatoires équivalents, comportement discutable d
   ocumenté")
```

La caractérisation **fige** le comportement actuel, même discutable ; l'ancien et le nouveau code sont prouvés équivalents. Corriger `b = 0` est une **décision de comportement**, à prendre séparément, avec un vrai test, et l'accord des utilisateurs de la fonction.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Écrire un gros test avant un gros bout de code	On perd les petits pas	Un comportement par cycle
Refactorer avec des tests rouges	On ne sait plus si on a cassé quelque chose	Toujours partir du vert
Mélanger refactoring et nouvelle fonctionnalité	Impossible d'isoler la cause d'un bug	Deux commits, deux pas distincts
Tests de caractérisation pris pour des specs	On fige des bugs	Les marquer ; corriger plus tard avec décision explicite

Mini-boss

Mini-boss

Kata complet : le score d'un jeu de bowling. Implémente `score(lancers)` (liste de quilles tombées par lancer, 10 manches, spares et strikes avec bonus, dernière manche spéciale) **en TDD strict**.

1. note la liste ordonnée de tes cycles (test, code, refactor) avec au moins 12 cycles ;
2. fais rejouer par un script chaque étape comme dans l'exemple (rouge avant, vert après) ;
3. refactore en séparant `decouper_en_manches` et `score_manche` sous la protection des tests ;
4. prouve l'invariant : le score max (12 strikes) est 300 et le score minimum 0 ; 5000 parties aléatoires valides ne dépassent jamais 300 ;
5. rédige 10 lignes : qu'est-ce que le TDD a changé dans ta façon de concevoir l'API ?

Mini-boss · réponse

Pistes de correction

Un bon travail : une première étape `score([0]*20) == 0`, puis `score([1]*20) == 20`, puis spare (`[5,5,3] + zéros → 16`), puis strike (`[10,3,4] + zéros → 24`), puis dernière manche (spare et strike avec lancers bonus), puis partie parfaite (300). Le refactor extrait une fonction qui lit les manches avec un indice. Le générateur aléatoire doit produire des parties **valides** (le second lancer ne peut dépasser les quilles restantes). L'apport attendu du TDD : l'API (`score(liste)`) est apparue naturellement de l'usage, et les cas limites de la dernière manche ont été découverts un par un.

Quiz éclair

Quiz éclair

1. Le cycle du TDD est :

A. vert, rouge, déployer
B. refactor, rouge, vert
C. rouge, vert, refactor

2. L'étape « vert » consiste à :

A. écrire le code le plus simple qui fait passer le test
B. écrire le code parfait immédiatement
C. supprimer le test

3. Un test de caractérisation sert à :

A. spécifier un nouveau comportement

Quiz éclair

Quiz éclair (suite)

B. figer le comportement actuel d'un code existant avant de le refactorer
C. accélérer le code

4. 🔁 *Révision mission 07* : Un stub sert à :

A. enregistrer des appels pour les vérifier
B. stocker des données en base
C. fournir des réponses préparées au code testé

5. 🔁 *Révision mission 07* : Pour rendre testable un code dépendant de l'heure, on :

A. injecte l'horloge en paramètre
B. attend minuit
C. lit l'horloge dans chaque fonction

Quiz éclair · réponse

Réponses

1. **C**.
2. **A**.
3. **B**.
4. **C**.
5. **A**.

Je réfléchis

1. Sur quel code de ton projet n'ose-t-on plus toucher, faute de tests ?
2. Quel serait le premier test de caractérisation que tu écrirais ?

À retenir

À retenir

1. Rouge → vert → refactor, par petits pas.
2. Le code le plus simple qui passe ; triangulation pour généraliser.
3. Refactorer = changer la forme, pas le comportement, sous protection des tests.
4. Code hérité : caractériser d'abord, corriger ensuite.
5. Le TDD guide la conception de l'API ; ce n'est pas un dogme.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Danseur rouge-vert

Prochaine mission : Flakiness et qualité des suites

