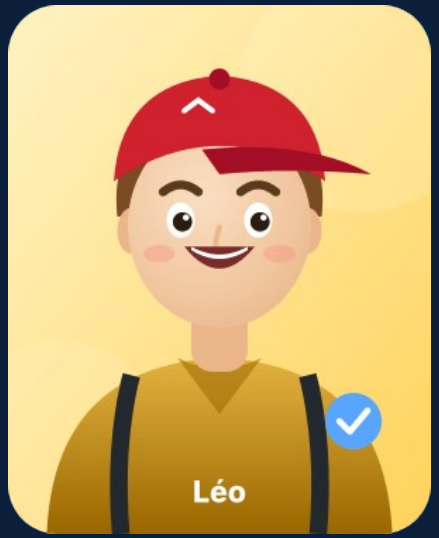
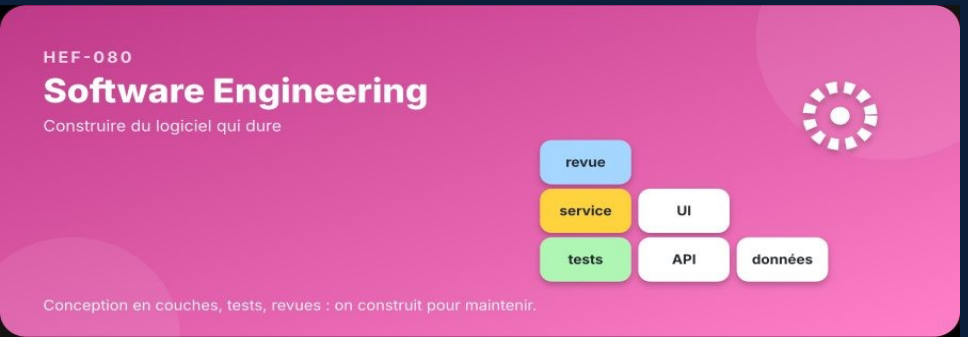
HEF-080 · MISSION 04 / 10

Clean Code : écrire pour être lu

Le code est écrit une fois et lu cent fois. Il doit se lire comme de la prose claire.





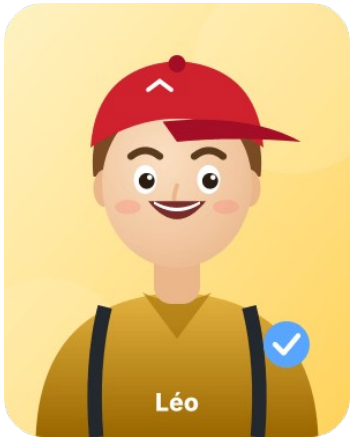
Badge : Artisan du code lisible

80 min · intermédiaire

L'histoire

Kodi relit une fonction écrite il y a six mois par… lui-même :

```
def f(d, t):
    r = 0
    for x in d:
        if x[2] > t and x[3] != 0:
            r += x[1] * 0.2
    return r
```



« Qui est d ? Que vaut x[2] ? D'où sort ce 0.2 ? » se demande-t-il. « Ton futur toi est un collègue qui n'a pas ta mémoire, dit Sam. Écris pour lui. »

Tu apprends les pratiques qui rendent un code immédiatement compréhensible.

Ta mission

À la fin, je serai capable de…

1. choisir des noms qui révèlent l'intention
2. écrire de petites fonctions qui font une chose
3. remplacer les nombres magiques par des constantes nommées
4. simplifier la logique avec des retours anticipés
5. écrire des commentaires utiles (le pourquoi, pas le quoi)

Défi d'échauffement

Défi d'échauffement

Tu lis deux phrases : (A) « Appliquer la remise de fidélité aux clients qui ont plus de 2 ans d'ancienneté. » (B) « Faire x = y * 0,85 pour z dans la liste q si w > 2. » Quelle phrase comprends-tu tout de suite ? Qu'est-ce qui rend l'autre obscure ? Liste trois règles pour qu'un texte (ou un code) soit facile à lire.

Le concept

1. Des noms qui disent l'intention

❌	✅
`d`	`membres`
`lst2`	`membres_actifs`
`process(x)`	`calculer_remise_fidelite(membre)`
`flag`	`est_abonne`

Règles : prononçables, cherchables, **sans abréviations obscures**, un mot par concept. Un booléen se nomme comme une question (`est_`, `a_`, `peut_`). Une fonction est un **verbe**, une classe un **nom**.

Le concept

2. Des fonctions petites, qui font une chose

- Une fonction = une idée, un niveau d'abstraction.
- Peu de paramètres (idéalement ≤ 3). Au-delà : regrouper dans un objet.
- Éviter les paramètres booléens qui changent le comportement (`envoyer(msg, True)`) : deux fonctions plus claires.

3. Pas de nombres magiques

```
# ❌
if age >= 18 and score > 60: ...
# ✅
AGE_MAJORITE = 18
SCORE_MINIMUM = 60
if age >= AGE_MAJORITE and score > SCORE_MINIMUM: ...
```

Le concept

4. La logique plate

Les `if` imbriqués sur plusieurs niveaux fatiguent. Traite les **cas d'erreur d'abord** (retours anticipés / *guard clauses*), puis le cas principal.

```
def remise(membre):
    if membre is None:
        return 0
    if not membre.actif:
        return 0
    if membre.anciennete < 2:
        return 0
    return 0.15
```

Le concept

5. Les commentaires

- Un commentaire qui répète le code est du bruit : `i += 1  # incrémente i`.
- Un bon commentaire explique le **pourquoi** : une règle métier, une contrainte, un piège.
- Si tu ressens le besoin d'un commentaire pour expliquer *quoi*, **renomme** ou **extrais une fonction**.
- Les **docstrings** documentent le contrat (paramètres, retour, erreurs).

6. La gestion des erreurs

- Lever des **exceptions** précises plutôt que renvoyer `None` ou `-1` ambigus.
- Ne pas avaler une exception avec un `except:` vide.
- Messages d'erreur **utiles** : quoi, avec quelle valeur, quelle attente.

Le concept

7. Le formatage et les outils

Un style cohérent facilite la lecture : un formateur automatique (`black`) et un analyseur (`ruff`, `flake8`) évitent les discussions sans fin. Les tests documentent aussi le comportement.

8. La règle du boy-scout

> *Laisse le code un peu plus propre que tu ne l'as trouvé.*

De petites améliorations constantes évitent la dégradation lente.

Exemple pas à pas

Exemple pas à pas

Problème : rendre le code de Kodi lisible.

Avant :

```
def f(d, t):
    r = 0
    for x in d:
        if x[2] > t and x[3] != 0:
            r += x[1] * 0.2
    return r
```

On découvre (en lisant le code appelant) que `d` est une liste de ventes `(id, montant, ancienneté_en_jours, actif)`, `t` un seuil d'ancienneté, et 0,2 le taux de commission.

Après :

Exemple pas à pas

Exemple pas à pas (suite)

```
from dataclasses import dataclass

TAUX_COMMISSION = 0.20

@dataclass
class Vente:
    identifiant: int
    montant: float
    anciennete_jours: int
    est_active: bool

def est_eligible(vente, anciennete_minimum_jours):
    return vente.est_active and vente.anciennete_jours > anciennete_minimum_jours

def commission(vente):
    return vente.montant * TAUX_COMMISSION

def total_commissions(ventes, anciennete_minimum_jours):
    """Somme des commissions des ventes actives plus anciennes que le seuil."""
```

Exemple pas à pas (suite)

```
    return sum(commission(v) for v in ventes if est_eligible(v, anciennete_minimum
   _jours))

ventes = [Vente(1, 100, 30, True), Vente(2, 200, 5, True), Vente(3, 50, 90, False)
   ]
assert total_commissions(ventes, 10) == 20.0
print(total_commissions(ventes, 10))
```

Vérification d'équivalence :

```
def f(d, t):
    r = 0
    for x in d:
        if x[2] > t and x[3] != 0:
            r += x[1] * 0.2
    return r

anciennes = [(1, 100, 30, 1), (2, 200, 5, 1), (3, 50, 90, 0)]
```

Exemple pas à pas (suite)

```
nouvelles = [Vente(i, m, a, bool(act)) for i, m, a, act in anciennes]
assert f(anciennes, 10) == total_commissions(nouvelles, 10)
```

Ce qui a changé : les noms racontent le métier ; le 0,2 a un nom ; les règles d'éligibilité et de commission sont **séparées** donc modifiables indépendamment ; la fonction principale se lit comme une phrase : *total des commissions des ventes éligibles*.

Un bon et un mauvais commentaire :

```
# mauvais : répète le code
total = prix * quantite  # multiplie prix et quantité

# bon : explique une règle non évidente
# Les ventes de moins de 5 jours peuvent encore être annulées (délai légal de rétr
   actation).
DELAI_RETRACTATION_JOURS = 5
```

À toi, avec aide

À toi, avec aide

Rends ce code lisible :

```
def g(l, a):
    o = []
    for i in l:
        if i["a"] >= a:
            if i["s"] == 1:
                o.append(i["n"].upper())
    return o
```

On sait que `l` est une liste de membres (`n` nom, `a` âge, `s` statut où 1 = actif) et `a` un âge minimum.

1. Donne de bons noms.
2. Aplatis la logique.
3. Écris un test d'équivalence.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Introduis une fonction `est_membre_eligible(membre, age_minimum)`.

Indice 2

Remplace le statut `1` par une constante `STATUT_ACTIF`.

Indice 3

Une compréhension de liste exprime bien « les noms en majuscules des membres éligibles ».

À toi, avec aide · réponse

Réponse

```
STATUT_ACTIF = 1

def est_membre_eligible(membre, age_minimum):
    return membre["a"] >= age_minimum and membre["s"] == STATUT_ACTIF

def noms_en_majuscules_des_membres_eligibles(membres, age_minimum):
    return [m["n"].upper() for m in membres if est_membre_eligible(m, age_minimum)
   ]

def g(l, a):
    o = []
    for i in l:
        if i["a"] >= a:
            if i["s"] == 1:
                o.append(i["n"].upper())
    return o

membres = [{"n": "ada", "a": 30, "s": 1}, {"n": "bob", "a": 17, "s": 1}, {"n": "cl
   eo", "a": 40, "s": 0}]
```

Réponse (suite)

```
assert noms_en_majuscules_des_membres_eligibles(membres, 18) == g(membres, 18) ==
   ["ADA"]
print("ok")
```

À toi, en solo

Revue de lisibilité. Voici une fonction avec de nombreux défauts :

```
def proc(x, y, z=False):
    # calcule
    if x != None:
        if len(x) > 0:
            t = 0
            for i in range(len(x)):
                if x[i]["p"] > 100:
                    t = t + x[i]["p"] * 0.9
                else:
                    t = t + x[i]["p"]
            if z == True:
                t = t + y
            return t
        else:
            return -1
    else:
```

À toi, en solo (suite)

```
        return -1
```

1. Liste au moins 8 problèmes de lisibilité.
2. Réécris-la proprement (noms, constantes, retours anticipés, paramètre booléen remplacé, erreurs explicites).
3. Prouve l'équivalence sur les cas valides et explique ce que tu as volontairement changé (le `-1`).

À toi, en solo · réponse

Pistes de correction

Problèmes : noms obscurs (`proc`, `x`, `y`, `z`, `t`, `i`) ; commentaire inutile ; `!= None` au lieu de `is not None` ; imbrication ; `range(len(...))` au lieu d'itérer directement ; nombres magiques (100, 0,9) ; `z == True` ; paramètre booléen qui change le comportement ; `-1` ambigu pour signaler une erreur.

```
SEUIL_REMISE = 100
TAUX_REMISE = 0.10

def prix_remise(prix):
    return prix * (1 - TAUX_REMISE) if prix > SEUIL_REMISE else prix

def total_panier(articles):
    """Total d'un panier non vide ; une remise de 10 % s'applique aux articles de
   plus de 100."""
    if not articles:
        raise ValueError("le panier est vide")
    return sum(prix_remise(a["p"]) for a in articles)

def total_avec_livraison(articles, frais_livraison):
```

Pistes de correction (suite)

```
    return total_panier(articles) + frais_livraison

def proc(x, y, z=False):
    if x != None:
        if len(x) > 0:
            t = 0
            for i in range(len(x)):
                if x[i]["p"] > 100:
                    t = t + x[i]["p"] * 0.9
                else:
                    t = t + x[i]["p"]
            if z == True:
                t = t + y
            return t
        else:
            return -1
    else:
        return -1
```

À toi, en solo · réponse

Pistes de correction (suite)

```
panier = [{"p": 50}, {"p": 200}, {"p": 100}]
assert total_panier(panier) == proc(panier, 7) == 330.0
assert total_avec_livraison(panier, 7) == proc(panier, 7, True) == 337.0
try:
    total_panier([])
except ValueError as e:
    print("erreur explicite :", e)
```

Changement volontaire : le `-1` (qui pourrait être confondu avec un total) est remplacé par une exception claire. Le paramètre booléen `z` est remplacé par deux fonctions.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Noms trop courts ou trompeurs	Il faut deviner l'intention	Nom = intention, pas type
Commentaires qui répètent le code	Bruit, et ils deviennent faux	Commenter le pourquoi
Fonctions de 100 lignes	Plusieurs idées mélangées	Extraire des fonctions nommées
Renvoyer `-1` ou `None` pour une erreur	Confusion avec une vraie valeur	Lever une exception explicite

Mini-boss

Chantier de nettoyage. Écris (volontairement mal) un script de 60 lignes qui traite des commandes : lecture d'une liste de dictionnaires, validation, calcul de remises selon le client et la quantité, calcul de TVA, arrondi, et génération d'un récapitulatif texte.

1. Écris les tests qui figent son comportement (au moins 10 cas).
2. Réécris-le en appliquant toutes les pratiques de la mission (noms, fonctions courtes, constantes, retours anticipés, exceptions, docstrings).
3. Vérifie que les tests passent toujours.
4. Mesure avec l'outil d'audit de la mission 1 : complexité, longueur, imbrication avant/après.
5. Rédige 5 lignes : quelles améliorations ont eu le plus d'effet sur la lisibilité ?

Mini-boss · réponse

Pistes de correction

Un bon travail : fonctions `valider_commande`, `taux_remise`, `calculer_ht`, `calculer_tva`, `formater_recapitulatif`, chacune avec une docstring d'une ligne ; constantes `TAUX_TVA`, `SEUIL_QUANTITE`, `REMISES_PAR_CLIENT` ; exceptions `CommandeInvalide` avec message précis ; tests d'équivalence parametrés ; mesures montrant la baisse de la complexité maximale (par exemple de 14 à 4) et de la profondeur d'imbrication (de 5 à 2). Les améliorations les plus efficaces sont généralement le renommage et l'extraction de fonctions nommées d'après l'intention.

Quiz éclair

Quiz éclair

1. Un bon nom de fonction est généralement :

A. un verbe qui exprime l'intention
B. une lettre
C. un nombre

2. Les retours anticipés servent à :

A. accélérer le code
B. réduire l'imbrication
C. supprimer les tests

3. Un bon commentaire explique :

A. le quoi évident

Quiz éclair

Quiz éclair (suite)

B. la syntaxe
C. le pourquoi

4. 🔁 *Révision mission 03* : Le principe ouvert/fermé dit :

A. ouvert à l'extension, fermé à la modification
B. ouvert à la modification, fermé à l'extension
C. tout est ouvert

5. 🔁 *Révision mission 03* : Inverser les dépendances, c'est :

A. supprimer les dépendances
B. dépendre d'abstractions plutôt que de détails
C. tout mettre dans une classe

Quiz éclair · réponse

Réponses

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

Je réfléchis

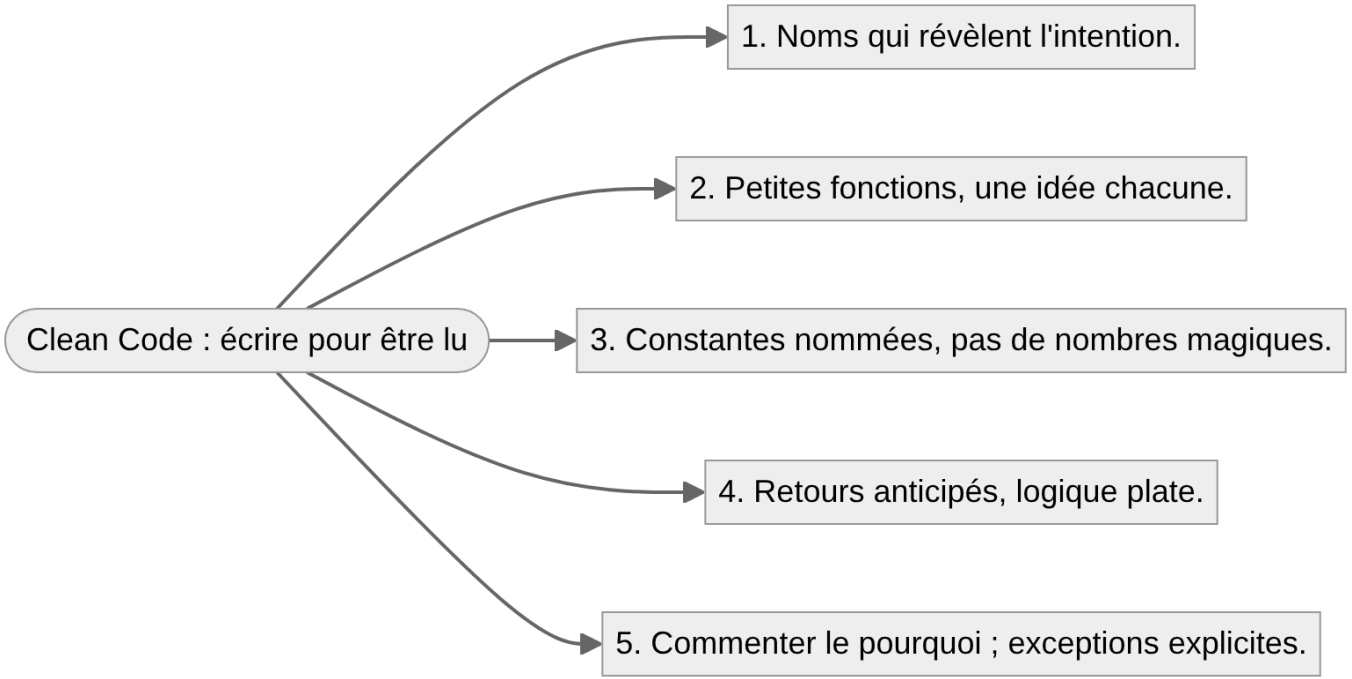
1. Quel nom de variable ou de fichier dans ton travail ne dit pas ce qu'il fait ?
2. Comment la règle du boy-scout pourrait-elle s'appliquer à tes projets ?

À retenir

1. Noms qui révèlent l'intention.
2. Petites fonctions, une idée chacune.
3. Constantes nommées, pas de nombres magiques.
4. Retours anticipés, logique plate.
5. Commenter le pourquoi ; exceptions explicites.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Artisan du code lisible

Prochaine mission : Interfaces et abstractions

