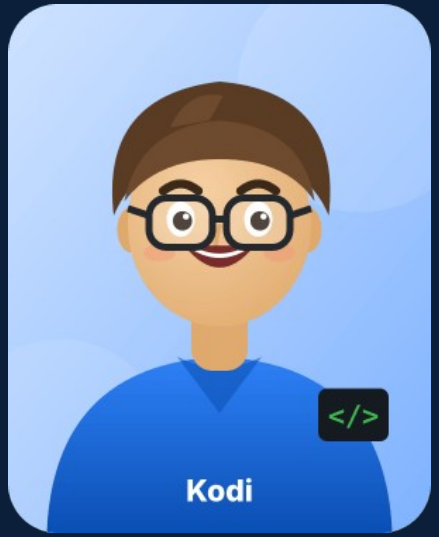
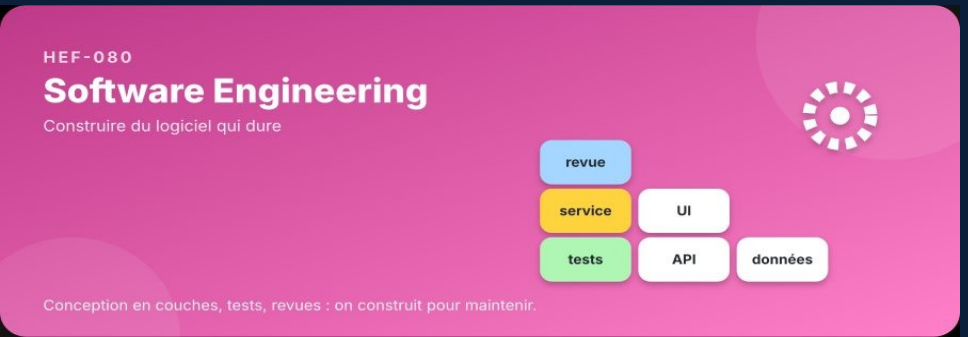
HEF-080 · MISSION 01 / 10

Qualité et maintenabilité

Un programme qui marche, c'est le minimum. Un programme qu'on peut encore modifier dans deux ans, c'est de l'ingénierie.



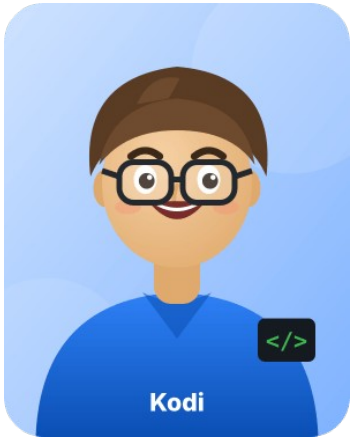


Badge : Sentinelle de la qualité

75 min · intermédiaire

L'histoire

Kodi hérite de `calcul_prix`, une fonction de 80 lignes écrite par un ancien collègue. Elle marche. Mais quand Maya demande « une remise pour les étudiants », Kodi passe trois jours à comprendre où la glisser, casse deux autres remises, et personne ne sait pourquoi.



> *« Le code est lu dix fois plus souvent qu'il n'est écrit, dit Sam. Écrire du code qui marche est la partie facile. Écrire du code qui reste modifiable, c'est le vrai métier. »*

Dans cette mission, tu apprends à définir et à mesurer la qualité d'un logiciel.

Ta mission

À la fin, je serai capable de…

1. lister les qualités d'un bon logiciel au-delà de « ça marche »
2. expliquer KISS, DRY et YAGNI avec un exemple
3. calculer une complexité cyclomatique simple
4. identifier un code difficile à modifier
5. améliorer une fonction en conservant son comportement (tests à l'appui)

Défi d'échauffement

Défi d'échauffement

Tu reçois deux recettes de cuisine qui donnent le même plat. L'une est claire, avec des étapes numérotées et des quantités. L'autre est un seul paragraphe illisible plein de ratures. Dans six mois, tu voudras **adapter la recette** pour un végétarien. Laquelle préfères-tu hériter ? Quelles propriétés rendent une recette « facile à modifier » ? Transpose ces propriétés au code.

Le concept

Les qualités d'un logiciel

Qualité	Question
Correction	fait-il ce qui est demandé ?
Lisibilité	un autre développeur le comprend-il vite ?
Testabilité	peut-on le vérifier automatiquement ?
Modifiabilité	un changement est-il local et sûr ?
Performance	est-il assez rapide pour l'usage ?
Sécurité, fiabilité	résiste-t-il aux abus et aux pannes ?

La **maintenabilité** regroupe lisibilité, testabilité et modifiabilité. Elle domine le coût d'un logiciel : la plupart de l'argent est dépensé **après** la première livraison.

Le concept

Le coût du changement

Un bon design garde le coût du changement **bas** : ajouter une règle de remise doit toucher un endroit, pas dix. Un mauvais design le fait **croître** à chaque fonctionnalité.

Trois principes de bon sens

Principe	Sens	Piège évité
KISS (*Keep It Simple, Stupid*)	la solution la plus simple qui répond au besoin	sur-ingénierie
DRY (*Don't Repeat Yourself*)	chaque connaissance a **une** représentation	corriger un bug à dix endroits
YAGNI (*You Aren't Gonna Need It*)	ne pas construire ce qui n'est pas demandé	complexité spéculative

Le concept

Trois principes de bon sens (suite)

Attention : DRY concerne la **connaissance** dupliquée, pas deux morceaux de code qui se ressemblent par hasard. Fusionner deux codes qui changent pour des raisons différentes crée du couplage.

Mesurer la complexité

La **complexité cyclomatique** compte les chemins d'exécution indépendants : 1 + nombre de décisions (`if`, `for`, `while`, `and`/`or`, `except`…). Repères courants : jusqu'à 10 : acceptable ; au-delà : à découper.

```
import ast

def complexite(source):
    n = 1
    for noeud in ast.walk(ast.parse(source)):
        if isinstance(noeud, (ast.If, ast.For, ast.While, ast.ExceptHandler, ast.I
   fExp)):
            n += 1
        elif isinstance(noeud, ast.BoolOp):
```

Le concept

Mesurer la complexité (suite)

```
            n += len(noeud.values) - 1
    return n

print(complexite("def f(a, b):\n    if a and b:\n        return 1\n    return 0"))
      # 3
```

Les signes d'un code difficile à modifier

- fonctions très longues, imbriquées sur 4 niveaux ;
- noms trompeurs (`data2`, `tmp`) ;
- le même nombre « magique » à plusieurs endroits ;
- pas de tests : personne n'ose toucher ;
- un changement oblige à modifier de nombreux fichiers.

Le concept

Le filet de sécurité : les tests

Avant de rendre un code plus propre, on écrit des **tests** qui figent son comportement actuel. Ensuite on peut nettoyer en toute confiance.

Exemple pas à pas

Problème : `calcul_prix` est illisible. On la refactore sans changer son comportement.

```
def calcul_prix_v1(quantite, client, jour_ferie):
    if quantite > 0:
        if client == "etudiant":
            if quantite >= 10:
                prix = quantite * 8 * 0.8
            else:
                prix = quantite * 8 * 0.9
        elif client == "pro":
            if quantite >= 10:
                prix = quantite * 8 * 0.7
            else:
                prix = quantite * 8 * 0.85
        else:
            if quantite >= 10:
                prix = quantite * 8 * 0.9
            else:
```

Exemple pas à pas (suite)

```
                prix = quantite * 8
        if jour_ferie:
            prix = prix * 1.1
        return round(prix, 2)
    else:
        return 0
```

Étape 1 : figer le comportement avec des tests.

```
import itertools

CAS = list(itertools.product([0, 1, 5, 10, 25], ["etudiant", "pro", "particulier"]
   , [False, True]))
ATTENDU = {cas: calcul_prix_v1(*cas) for cas in CAS}       # le comportement actue
   l sert de référence
```

Étape 2 : réécrire de façon lisible. Les remises sont une **donnée**, pas du code.

Exemple pas à pas

Exemple pas à pas (suite)

```
PRIX_UNITAIRE = 8
REMISES = {                          # client -> (remise petite quantité, remise g
   rande quantité)
    "etudiant": (0.10, 0.20),
    "pro": (0.15, 0.30),
    "particulier": (0.00, 0.10),
}
SEUIL_GROSSE_QUANTITE = 10
MAJORATION_JOUR_FERIE = 1.10

def calcul_prix_v2(quantite, client, jour_ferie):
    if quantite <= 0:
        return 0
    petite, grande = REMISES[client]
    remise = grande if quantite >= SEUIL_GROSSE_QUANTITE else petite
    prix = quantite * PRIX_UNITAIRE * (1 - remise)
    if jour_ferie:
        prix *= MAJORATION_JOUR_FERIE
    return round(prix, 2)
```

Exemple pas à pas (suite)

Étape 3 : prouver l'équivalence.

```
assert all(calcul_prix_v2(*cas) == attendu for cas, attendu in ATTENDU.items())
print("comportement identique sur", len(CAS), "cas")
```

Gain : la complexité passe de 8 à 4 environ ; « ajouter une remise pour un nouveau type de client » devient **une ligne** dans `REMISES`. Voilà une mesure concrète de la maintenabilité : le coût d'un changement a baissé.

À toi, avec aide

À toi, avec aide

Voici une fonction :

```
def statut(note, absences, rattrapage):
    if note >= 10:
        if absences > 5:
            return "valide avec avertissement"
        else:
            return "valide"
    else:
        if rattrapage:
            if note >= 8:
                return "valide au rattrapage"
            else:
                return "non valide"
        else:
            return "non valide"
```

1. Calcule sa complexité cyclomatique avec la fonction `complexite`.

À toi, avec aide (suite)

2. Écris des tests qui figent son comportement.
3. Réécris-la avec des retours anticipés (*early return*) et vérifie l'équivalence.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Chaque `if` compte pour 1 : ici, 4 décisions → complexité 5.

Indice 2

Teste les combinaisons de notes (5, 8, 9, 10, 15), d'absences (0, 6) et de rattrapage (vrai/faux).

Indice 3

Traite d'abord les cas particuliers, puis le cas général, sans `else` imbriqués.

À toi, avec aide · réponse

Réponse

```
import ast, itertools

def complexite(source):
    n = 1
    for noeud in ast.walk(ast.parse(source)):
        if isinstance(noeud, (ast.If, ast.For, ast.While, ast.ExceptHandler, ast.I
   fExp)):
            n += 1
        elif isinstance(noeud, ast.BoolOp):
            n += len(noeud.values) - 1
    return n

def statut_v1(note, absences, rattrapage):
    if note >= 10:
        if absences > 5:
            return "valide avec avertissement"
        else:
            return "valide"
    else:
```

À toi, avec aide · réponse

Réponse (suite)

```
        if rattrapage:
            if note >= 8:
                return "valide au rattrapage"
            else:
                return "non valide"
        else:
            return "non valide"

def statut_v2(note, absences, rattrapage):
    if note >= 10:
        return "valide avec avertissement" if absences > 5 else "valide"
    if rattrapage and note >= 8:
        return "valide au rattrapage"
    return "non valide"

for cas in itertools.product([5, 8, 9, 10, 15], [0, 6], [False, True]):
    assert statut_v1(*cas) == statut_v2(*cas), cas
print("équivalentes")
```

Réponse (suite)

La complexité cyclomatique reste proche (5 avant, 5 après : le nombre de décisions n'a pas changé), mais la **profondeur d'imbrication** passe de 3 niveaux à 1 et la logique se lit de haut en bas. Une métrique ne dit pas tout : ici c'est la lisibilité, pas la complexité cyclomatique, qui s'améliore.

À toi, en solo

Chasse au code dupliqué. Voici trois fonctions d'une plateforme :

```
def prix_cours(base, membre):
    if membre["statut"] == "etudiant":
        return round(base * 0.8, 2)
    if membre["statut"] == "pro":
        return round(base * 0.9, 2)
    return base

def prix_atelier(base, membre):
    if membre["statut"] == "etudiant":
        return round(base * 0.8, 2)
    if membre["statut"] == "pro":
        return round(base * 0.9, 2)
    return base + 5

def prix_livre(base, membre):
    if membre["statut"] == "etudiant":
```

À toi, en solo (suite)

```
        return round(base * 0.8, 2)
    if membre["statut"] == "pro":
        return round(base * 0.9, 2)
    return base
```

1. Identifie la **connaissance** dupliquée (qui est-ce que DRY doit unifier ?).
2. Écris une version DRY avec une seule table de remises, en gardant le comportement exact (y compris le `+ 5` d'`atelier`).
3. Montre qu'ajouter un statut « senior » à 15 % de remise ne demande qu'**un** changement.

À toi, en solo · réponse

Pistes de correction

```
REMISES = {"etudiant": 0.20, "pro": 0.10}

def appliquer_remise(base, membre):
    return round(base * (1 - REMISES.get(membre["statut"], 0)), 2)

def prix_cours(base, membre):
    return appliquer_remise(base, membre)

def prix_livre(base, membre):
    return appliquer_remise(base, membre)

def prix_atelier(base, membre):
    prix = appliquer_remise(base, membre)
    return prix if membre["statut"] in REMISES else prix + 5

assert prix_cours(100, {"statut": "etudiant"}) == 80.0
assert prix_atelier(100, {"statut": "pro"}) == 90.0
assert prix_atelier(100, {"statut": "particulier"}) == 105
REMISES["senior"] = 0.15
```

À toi, en solo · réponse

Pistes de correction (suite)

```
assert prix_livre(100, {"statut": "senior"}) == 85.0
print("ok")
```

La connaissance dupliquée est la **règle de remise par statut**. Attention à ne pas tout fusionner : le supplément de 5 € des ateliers est une règle **propre** aux ateliers, elle reste dans `prix_atelier`.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Juger la qualité seulement sur « ça marche »	Le coût de maintenance explose	Mesurer lisibilité, tests, facilité de changement
Dédupliquer à l'excès	Couplage entre choses qui divergent	DRY = une connaissance, une représentation
Anticiper des besoins hypothétiques	Complexité sans valeur (YAGNI)	Construire ce qui est demandé, rester ouvert
Refactorer sans tests	On casse sans le savoir	Figer le comportement avant de toucher

Mini-boss

Mini-boss

Audit de maintenabilité. Écris un petit outil `audit(dossier)` qui parcourt des fichiers Python et, pour chaque fonction, calcule : nombre de lignes, profondeur maximale d'imbrication, complexité cyclomatique, nombre de paramètres.

1. Affiche les fonctions qui dépassent des seuils (complexité > 10, lignes > 40, imbrication > 3, paramètres > 5).
2. Applique l'outil à un code d'exemple que tu écris avec trois fonctions « sales ».
3. Refactore la plus mauvaise en gardant son comportement (tests d'équivalence) et relance l'audit pour montrer l'amélioration.

Mini-boss · réponse

Pistes de correction

```
import ast, textwrap

def profondeur(noeud, niveau=0):
    blocs = (ast.If, ast.For, ast.While, ast.Try, ast.With)
    m = niveau
    for enfant in ast.iter_child_nodes(noeud):
        m = max(m, profondeur(enfant, niveau + 1) if isinstance(enfant, blocs) els
   e profondeur(enfant, niveau))
    return m

def complexite(f):
    n = 1
    for x in ast.walk(f):
        if isinstance(x, (ast.If, ast.For, ast.While, ast.ExceptHandler, ast.IfExp
   )):
            n += 1
        elif isinstance(x, ast.BoolOp):
            n += len(x.values) - 1
    return n
```

Mini-boss · réponse

Pistes de correction (suite)

```
def audit(source, seuils=(10, 40, 3, 5)):
    rapport = []
    for f in [n for n in ast.walk(ast.parse(source)) if isinstance(n, ast.Function
   Def)]:
        lignes = f.end_lineno - f.lineno + 1
        mesures = (complexite(f), lignes, profondeur(f), len(f.args.args))
        if any(m > s for m, s in zip(mesures, seuils)):
            rapport.append((f.name, mesures))
    return rapport

sale = textwrap.dedent('''
def f(a, b, c, d, e, g):
    if a:
        if b:
            for x in c:
                if x and d:
                    while e:
                        if g:
```

Mini-boss · réponse

Pistes de correction (suite)

```
                            return 1
    return 0
def propre(x):
    return x + 1
''')
print(audit(sale))      # [('f', (...))]
```

L'outil désigne `f` (imbrication, nombre de paramètres, complexité). La refactorisation utilise des retours anticipés et regroupe les paramètres dans un objet.

Quiz éclair

1. La maintenabilité désigne surtout :

A. la facilité à comprendre, tester et modifier
B. la vitesse d'exécution
C. la taille du code

2. DRY signifie :

A. ne pas écrire de commentaires
B. ne pas dupliquer une connaissance
C. répéter les tests

3. YAGNI conseille de :

A. tout prévoir à l'avance

Quiz éclair

Quiz éclair (suite)

B. éviter les tests
C. ne pas construire ce qui n'est pas demandé

4. 🔁 *Révision HEF-070, mission 08* : Une transaction garantit :

A. tout ou rien
B. des requêtes plus rapides
C. des données chiffrées

5. 🔁 *Révision HEF-070, mission 01* : Une clé étrangère :

A. identifie la ligne
B. référence la clé primaire d'une autre table
C. est un index

Quiz éclair · réponse

Réponses

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

Je réfléchis

Je réfléchis

1. Quel code (le tien ou celui d'un autre) as-tu déjà eu peur de modifier ? Qu'est-ce qui te faisait peur ?
2. Où, dans ta vie, dupliques-tu une information qui finira par se désynchroniser ?

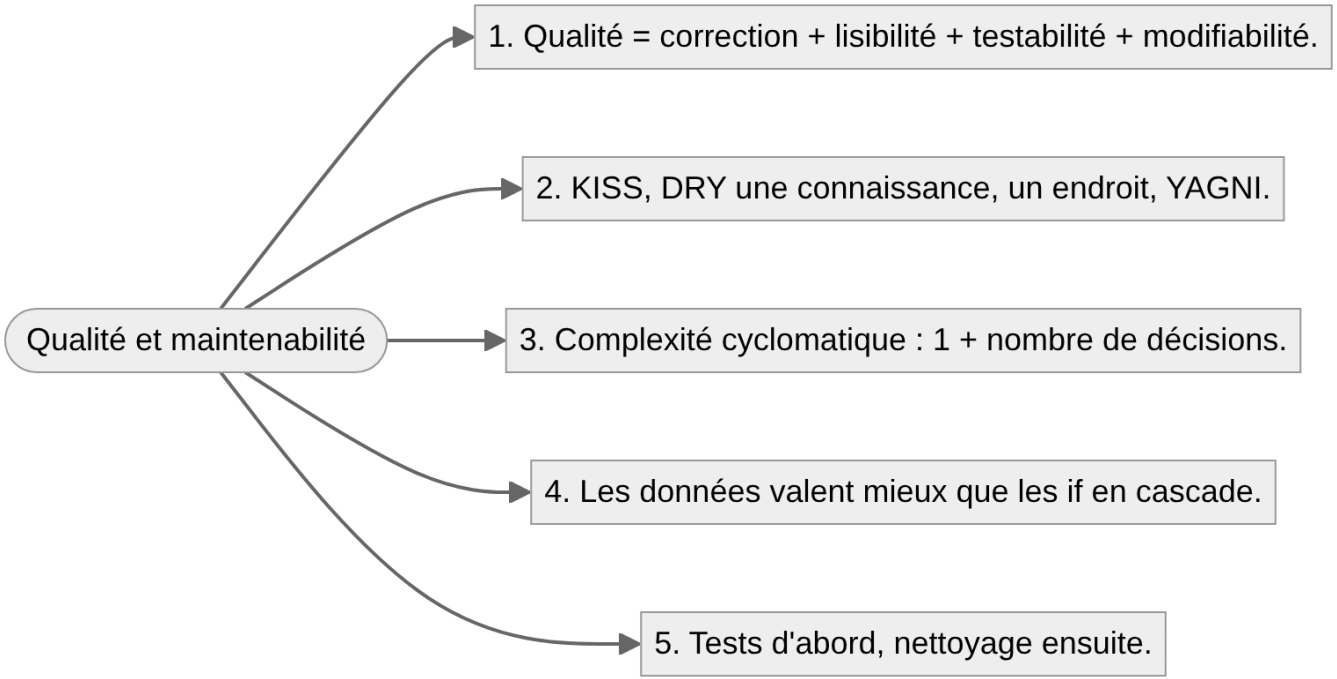
À retenir

À retenir

1. Qualité = correction + lisibilité + testabilité + modifiabilité.
2. KISS, DRY (une connaissance, un endroit), YAGNI.
3. Complexité cyclomatique : 1 + nombre de décisions.
4. Les données valent mieux que les `if` en cascade.
5. Tests d'abord, nettoyage ensuite.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Sentinelle de la qualité

Prochaine mission : Cohésion et couplage

