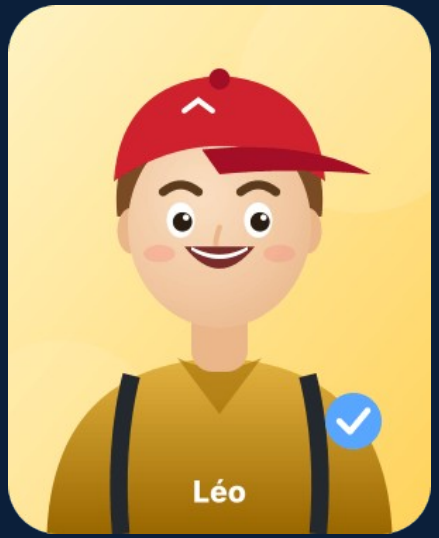
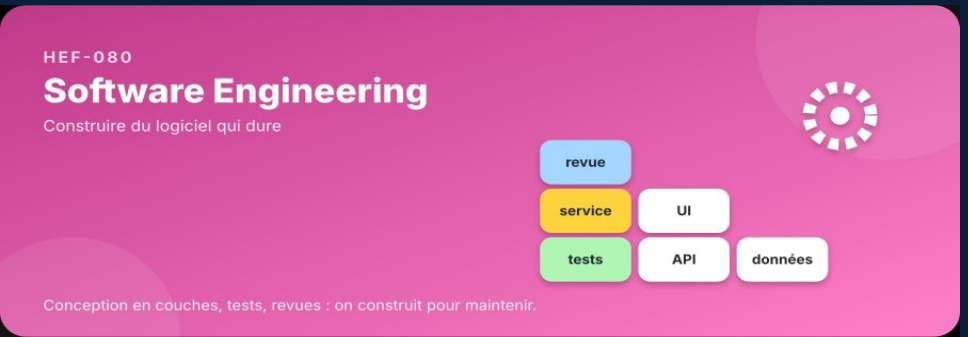
HEF-080 · MISSION 08 / 10

Refactoring sécurisé

Changer la structure d'un code sans changer son comportement : une discipline qui s'apprend.





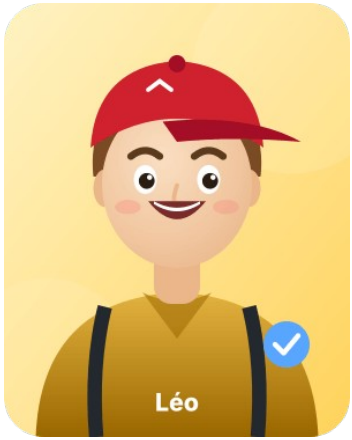
Badge : Chirurgien du code

90 min · intermédiaire

L'histoire

Kodi doit modifier `generer_facture`, une fonction de 150 lignes sans aucun test, écrite par quelqu'un qui est parti. Il n'ose pas y toucher. Sam lui dit :

> *« Tu ne vas pas la réécrire. Tu vas écrire des tests qui décrivent ce qu'elle fait aujourd'hui, puis la transformer en petits pas, en relançant les tests à chaque pas. »*



Tu apprends la discipline du refactoring : petits pas, tests, filet de sécurité.

Ta mission

À la fin, je serai capable de…

1. définir le refactoring et le distinguer d'une réécriture ou d'une nouvelle fonctionnalité
2. écrire des tests de caractérisation sur du code legacy
3. appliquer des refactorings élémentaires (renommer, extraire, introduire un paramètre)
4. remplacer des conditionnelles par du polymorphisme
5. utiliser la méthode des petits pas et savoir revenir en arrière

Défi d'échauffement

Défi d'échauffement

Tu veux réorganiser ta cuisine pendant que tu cuisines pour 20 invités. Comment procèdes-tu pour ne pas rater le repas : tout vider d'un coup, ou déplacer un objet à la fois en vérifiant que tu retrouves toujours ce dont tu as besoin ? Qu'est-ce qui joue le rôle de « test » dans cette analogie ?

Le concept

Qu'est-ce que le refactoring ?

***Refactorer**, c'est améliorer la **structure interne** d'un code **sans changer son comportement observable**.*

	Refactoring	Réécriture	Nouvelle fonctionnalité
Comportement	inchangé	peut changer	change volontairement
Taille des pas	petits, vérifiés	un grand saut	variable
Risque	faible avec des tests	élevé	moyen

Règle des deux chapeaux (Kent Beck) : on ne porte qu'**un chapeau à la fois** : soit on **refactore** (aucune nouvelle fonctionnalité), soit on **ajoute** une fonctionnalité (pas de nettoyage). Les mélanger rend les erreurs introuvables.

Le concept

Le filet de sécurité : les tests

Sans tests, on ne refactore pas : on **espère**. Pour du code existant sans tests, on écrit d'abord des **tests de caractérisation** :

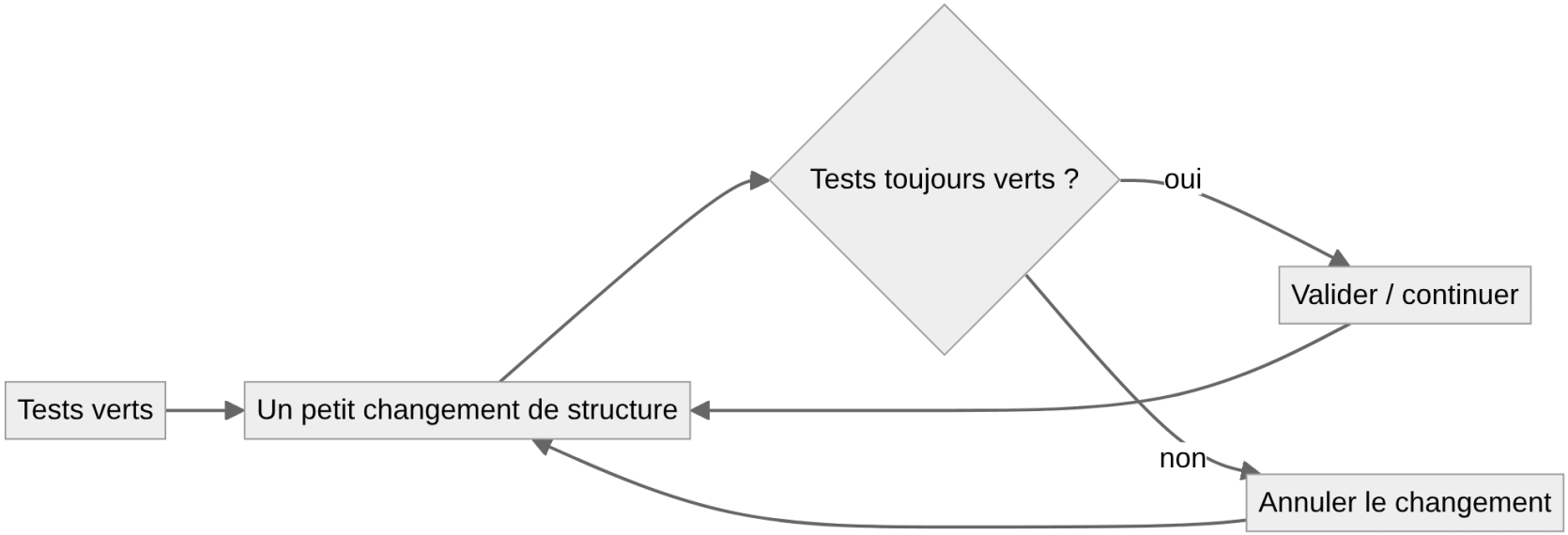
1. appeler le code avec des entrées variées ;
2. noter ce qu'il **renvoie réellement** (même si c'est bizarre) ;
3. en faire des assertions : elles **figent** le comportement actuel.

Un test de caractérisation ne dit pas que le comportement est *bon*, seulement qu'on ne l'a *pas changé*.

La boucle du refactoring

Le concept

La boucle du refactoring (suite)



Le concept

La boucle du refactoring (suite)

Description du schéma : on part de tests au vert, on fait un petit changement, on relance les tests. S'ils passent, on valide et on continue ; sinon on annule le dernier changement et on recommence plus petit.

Quelques refactorings élémentaires

Refactoring	Quand
Renommer	le nom ne dit pas l'intention
Extraire une fonction	un bloc a un sens propre, ou est dupliqué
Extraire une variable / constante	expression ou nombre magique à nommer
Inline	une fonction ou variable n'apporte rien
Introduire un objet-paramètre	trop de paramètres liés

Le concept

Quelques refactorings élémentaires (suite)

Refactoring	Quand
Remplacer une conditionnelle par du polymorphisme	`if type == …` répété
Décomposer une conditionnelle	condition complexe à nommer
Déplacer une méthode	elle manipule surtout les données d'une autre classe

Les « odeurs » de code (*code smells*)

Des signes qu'un refactoring serait utile : fonction trop longue, trop de paramètres, duplication, classe fourre-tout, commentaires qui expliquent du code confus, conditionnelles répétées, envie de fonctionnalité (*feature envy*), chaînes d'appels.

Le concept

Refactorer à grande échelle

Pour remplacer un grand système : le **motif de l'étranglement** (*strangler fig*). On construit la nouvelle version **autour** de l'ancienne, on redirige progressivement les usages, jusqu'à pouvoir supprimer l'ancienne. Jamais de grand soir.

Les outils

Les environnements de développement proposent des refactorings **automatiques** (renommer partout, extraire une fonction), plus sûrs que des modifications à la main. Le contrôle de version (Git) permet de revenir en arrière à chaque petit pas : **un commit par pas** vérifié.

Exemple pas à pas

Problème : `generer_facture` est un monolithe sans tests. On le transforme en petits pas.

Le code d'origine :

```
def generer_facture(client, lignes, pays):
    total = 0
    texte = "FACTURE\n"
    texte += "Client: " + client + "\n"
    for l in lignes:
        if l[2] > 0:
            t = l[1] * l[2]
            if l[3] == "A":
                t = t * 0.8
            elif l[3] == "B":
                t = t * 0.9
            total += t
            texte += l[0] + " x" + str(l[2]) + " = " + str(round(t, 2)) + "\n"
    if pays == "FR":
```

Exemple pas à pas (suite)

```
        tva = total * 0.2
    elif pays == "DE":
        tva = total * 0.19
    else:
        tva = 0
    texte += "Total HT: " + str(round(total, 2)) + "\n"
    texte += "TVA: " + str(round(tva, 2)) + "\n"
    texte += "Total TTC: " + str(round(total + tva, 2)) + "\n"
    return texte
```

Étape 1 : tests de caractérisation. On fige la sortie actuelle.

```
ENTREES = [
    ("Ada", [("Stylo", 2.0, 10, "A"), ("Cahier", 3.5, 4, "B"), ("Sac", 25.0, 1, ""
   )], "FR"),
    ("Bob", [("Règle", 1.2, 0, "A"), ("Gomme", 0.8, 5, "")], "DE"),
    ("Cléo", [], "US"),
    ("Dan", [("Livre", 12.0, 3, "C")], "ES"),
```

Exemple pas à pas (suite)

```
]
REFERENCE = [generer_facture(*e) for e in ENTREES]        # sortie actuelle = véri
   té de référence

def verifier(fonction):
    for entree, attendu in zip(ENTREES, REFERENCE):
        assert fonction(*entree) == attendu, entree
    return "ok"
```

Étape 2 : extraire les règles en petits pas, en relançant `verifier` à chaque pas.

```
REMISES = {"A": 0.20, "B": 0.10}                        # pas 1 : nombres magiques →
    table nommée
TAUX_TVA = {"FR": 0.20, "DE": 0.19}                     # pas 2 : idem pour la TVA

def prix_ligne(prix_unitaire, quantite, categorie):   # pas 3 : extraire une fonct
   ion
    return prix_unitaire * quantite * (1 - REMISES.get(categorie, 0))
```

Exemple pas à pas

Exemple pas à pas (suite)

```
def lignes_facturables(lignes):                        # pas 4 : nommer la conditio
   n
    return [l for l in lignes if l[2] > 0]

def formater_ligne(nom, quantite, montant):            # pas 5 : extraire la présen
   tation
    return f"{nom} x{quantite} = {round(montant, 2)}\n"

def generer_facture_v2(client, lignes, pays):
    facturables = lignes_facturables(lignes)
    montants = [prix_ligne(l[1], l[2], l[3]) for l in facturables]
    total = sum(montants)
    taux = TAUX_TVA.get(pays)
    tva = total * taux if taux else 0              # garde l'entier 0 du code d'or
   igine (pas 0.0)
    corps = "".join(formater_ligne(l[0], l[2], m) for l, m in zip(facturables, mon
   tants))
    return ("FACTURE\n" + f"Client: {client}\n" + corps
```

Exemple pas à pas (suite)

```
            + f"Total HT: {round(total, 2)}\n" + f"TVA: {round(tva, 2)}\n" + f"Tot
    al TTC: {round(total + tva, 2)}\n")

print(verifier(generer_facture_v2))
```

Lecture : à chaque pas, `verifier` prouve que le comportement est identique. D'ailleurs, une première version écrivait `tva = total * TAUX_TVA.get(pays, 0)` : les tests ont immédiatement signalé une différence, car le code d'origine affichait `TVA: 0` (entier) alors que la nouvelle version affichait `TVA: 0.0`. C'est précisément le genre de détail qu'on ne voit pas à l'œil nu. À la fin, la facture est composée de fonctions nommées : changer une remise ou un taux de TVA se fait **dans une table**. Les nombres magiques ont des noms, la présentation est séparée du calcul.

Remplacer une conditionnelle par du polymorphisme (quand les `if` portent sur un « type ») :

Exemple pas à pas (suite)

```
class Remise:
    taux = 0.0
    def appliquer(self, montant): return montant * (1 - self.taux)

class RemiseA(Remise): taux = 0.20
class RemiseB(Remise): taux = 0.10

REMISES_OBJETS = {"A": RemiseA(), "B": RemiseB()}
```

Ajouter une catégorie « C » = ajouter une classe, sans modifier les existantes.

À toi, avec aide

À toi, avec aide

Voici une fonction legacy :

```
def tarif(age, est_membre, jour):
    if age < 12:
        p = 5
    elif age < 65:
        p = 10
    else:
        p = 6
    if est_membre == True:
        p = p - 2
    if jour == "samedi" or jour == "dimanche":
        p = p + 3
    return p
```

1. Écris des tests de caractérisation couvrant les combinaisons d'âges (5, 12, 40, 65, 80), de statut et de jours (lundi, samedi, dimanche).

À toi, avec aide (suite)

2. Refactore par petits pas (constantes nommées, fonctions extraites, simplification de `== True` et du test de jour), en relançant les tests.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Génère toutes les combinaisons avec `itertools.product`.

Indice 2

Extrais `tarif_de_base(age)`, `reduction_membre(est_membre)`, `supplement_week_end(jour)`.

Indice 3

`jour in ('samedi', 'dimanche')` remplace le `or`.

À toi, avec aide · réponse

Réponse

```
import itertools

def tarif_legacy(age, est_membre, jour):
    if age < 12:
        p = 5
    elif age < 65:
        p = 10
    else:
        p = 6
    if est_membre == True:
        p = p - 2
    if jour == "samedi" or jour == "dimanche":
        p = p + 3
    return p

CAS = list(itertools.product([5, 12, 40, 65, 80], [False, True], ["lundi", "samedi
   ", "dimanche"]))
REFERENCE = {cas: tarif_legacy(*cas) for cas in CAS}
```

À toi, avec aide · réponse

Réponse (suite)

```
TARIF_ENFANT, TARIF_ADULTE, TARIF_SENIOR = 5, 10, 6
AGE_ADULTE, AGE_SENIOR = 12, 65
REDUCTION_MEMBRE = 2
SUPPLEMENT_WEEK_END = 3
JOURS_WEEK_END = {"samedi", "dimanche"}

def tarif_de_base(age):
    if age < AGE_ADULTE:
        return TARIF_ENFANT
    if age < AGE_SENIOR:
        return TARIF_ADULTE
    return TARIF_SENIOR

def tarif(age, est_membre, jour):
    prix = tarif_de_base(age)
    if est_membre:
        prix -= REDUCTION_MEMBRE
    if jour in JOURS_WEEK_END:
        prix += SUPPLEMENT_WEEK_END
```

À toi, avec aide · réponse

Réponse (suite)

```
    return prix

assert all(tarif(*cas) == attendu for cas, attendu in REFERENCE.items())
print("refactoring sûr sur", len(CAS), "cas")
```

À toi, en solo

Refactoring d'un code avec conditionnelle répétée. Voici un code legacy de HashBox :

```
def envoyer(notif):
    if notif["type"] == "mail":
        return "MAIL à " + notif["dest"] + " : " + notif["texte"]
    elif notif["type"] == "sms":
        return "SMS à " + notif["dest"] + " : " + notif["texte"][:160]
    elif notif["type"] == "push":
        return "PUSH vers " + notif["dest"] + " : " + notif["texte"][:60]
    else:
        raise ValueError("type inconnu : " + notif["type"])

def cout(notif):
    if notif["type"] == "mail":
        return 0.0
    elif notif["type"] == "sms":
        return 0.06
    elif notif["type"] == "push":
```

À toi, en solo (suite)

```
        return 0.01
    else:
        raise ValueError("type inconnu : " + notif["type"])
```

1. Écris les tests de caractérisation (y compris textes longs et type inconnu).
2. Refactore en **remplaçant la conditionnelle par du polymorphisme** : une classe par type de notification, avec `formater(dest, texte)` et `cout`.
3. Vérifie l'équivalence, puis ajoute un type « slack » sans modifier le code existant.

À toi, en solo · réponse

Pistes de correction

```
def envoyer_legacy(notif):
    if notif["type"] == "mail":
        return "MAIL à " + notif["dest"] + " : " + notif["texte"]
    elif notif["type"] == "sms":
        return "SMS à " + notif["dest"] + " : " + notif["texte"][:160]
    elif notif["type"] == "push":
        return "PUSH vers " + notif["dest"] + " : " + notif["texte"][:60]
    else:
        raise ValueError("type inconnu : " + notif["type"])

def cout_legacy(notif):
    if notif["type"] == "mail":
        return 0.0
    elif notif["type"] == "sms":
        return 0.06
    elif notif["type"] == "push":
        return 0.01
    else:
        raise ValueError("type inconnu : " + notif["type"])
```

À toi, en solo · réponse

Pistes de correction (suite)

```
CAS = [{"type": t, "dest": "ada", "texte": x} for t in ("mail", "sms", "push") for
    x in ("court", "z" * 300)]
REF = [(envoyer_legacy(n), cout_legacy(n)) for n in CAS]

class Canal:
    nom, prefixe, longueur_max, cout = "", "", None, 0.0
    def formater(self, dest, texte):
        if self.longueur_max is not None:
            texte = texte[:self.longueur_max]
        return f"{self.prefixe} {dest} : {texte}"

class Mail(Canal):
    nom, prefixe = "mail", "MAIL à"
class SMS(Canal):
    nom, prefixe, longueur_max, cout = "sms", "SMS à", 160, 0.06
class Push(Canal):
    nom, prefixe, longueur_max, cout = "push", "PUSH vers", 60, 0.01
```

À toi, en solo · réponse

Pistes de correction (suite)

```
CANAUX = {c.nom: c for c in (Mail(), SMS(), Push())}

def canal(notif):
    try:
        return CANAUX[notif["type"]]
    except KeyError:
        raise ValueError("type inconnu : " + notif["type"]) from None

def envoyer(notif): return canal(notif).formater(notif["dest"], notif["texte"])
def cout(notif): return canal(notif).cout

assert [(envoyer(n), cout(n)) for n in CAS] == REF

class Slack(Canal):                                # extension sans modifier l'exi
   stant
    nom, prefixe, cout = "slack", "SLACK", 0.0
CANAUX["slack"] = Slack()
assert envoyer({"type": "slack", "dest": "#equipe", "texte": "salut"}) == "SLACK #
   equipe : salut"
```

À toi, en solo · réponse

Pistes de correction (suite)

```
print("ok")
```

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Refactorer sans tests	On casse sans le voir	Tests de caractérisation d'abord
Mélanger refactoring et nouvelle fonctionnalité	Erreurs impossibles à localiser	Un chapeau à la fois
Trop gros pas	Un échec ne dit plus d'où il vient	Un petit changement, un test, un commit
Tout réécrire « proprement »	Bugs et retard	Étranglement progressif

Mini-boss

Mini-boss

Sauvetage d'un module legacy. Écris (volontairement mal) un module de 60 lignes de HashBox qui calcule le **bulletin d'un étudiant** : moyennes pondérées par matière, bonus, mentions, rattrapage, texte du bulletin. Utilise des noms obscurs, des conditions imbriquées, des nombres magiques, du code dupliqué.

1. Écris une batterie d'au moins 25 cas de caractérisation (graine fixe pour les cas aléatoires).
2. Refactore en au moins 10 petits pas, en validant les tests après chacun (note chaque pas dans un journal : refactoring appliqué, tests verts).
3. Introduis une nouvelle fonctionnalité (mention « très bien avec félicitations ») **après** le refactoring, en suivant la règle des deux chapeaux.
4. Compare avant/après avec l'outil d'audit (complexité, longueur, imbrication).

Mini-boss · réponse

Pistes de correction

Un bon travail : une liste `JOURNAL` des étapes (« renommer `x` en `note` », « extraire `moyenne_ponderee` », « constantes de mentions »…) avec la vérification `assert verifier(bulletin)` après chaque pas ; des fonctions finales `moyenne_ponderee`, `bonus`, `mention`, `texte_bulletin` ; la nouvelle fonctionnalité ajoutée dans un commit séparé avec ses propres tests ; l'audit montre la complexité maximale passer d'environ 15 à moins de 5 et l'imbrication de 5 à 2.

Quiz éclair

Quiz éclair

1. Refactorer, c'est :

A. ajouter une fonctionnalité
B. réécrire tout le code
C. améliorer la structure sans changer le comportement

2. Un test de caractérisation :

A. fige le comportement actuel du code
B. vérifie que le code est correct
C. remplace la documentation

3. La règle des deux chapeaux dit :

A. écrire deux fois le code

Quiz éclair

Quiz éclair (suite)

B. ne pas refactorer et ajouter une fonctionnalité en même temps
C. utiliser deux langages

4. 🔁 *Révision mission 07* : Dans l'architecture hexagonale, le domaine :

A. dépend de la base de données
B. dépend du framework web
C. ne dépend d'aucun détail technique

5. 🔁 *Révision mission 07* : Un objet-valeur est :

A. défini par ses valeurs et immuable
B. défini par son identité
C. toujours persistant

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quelle chose chez toi a besoin d'être réorganisée par petits pas plutôt que d'un coup ?
2. Qu'est-ce qui te rassure le plus dans la boucle « petit pas, test, valider » ?

À retenir

À retenir

1. Refactorer = même comportement, meilleure structure.
2. Tests de caractérisation pour le code legacy.
3. Petits pas, tests à chaque pas, un commit par pas.
4. Deux chapeaux : refactoring ou fonctionnalité, jamais les deux.
5. Étranglement pour les grands chantiers.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Chirurgien du code

Prochaine mission : La dette technique

