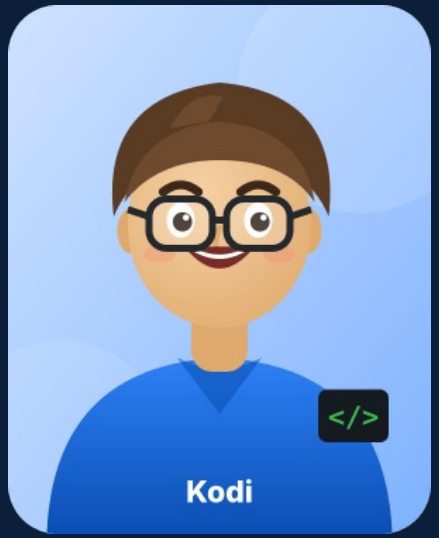
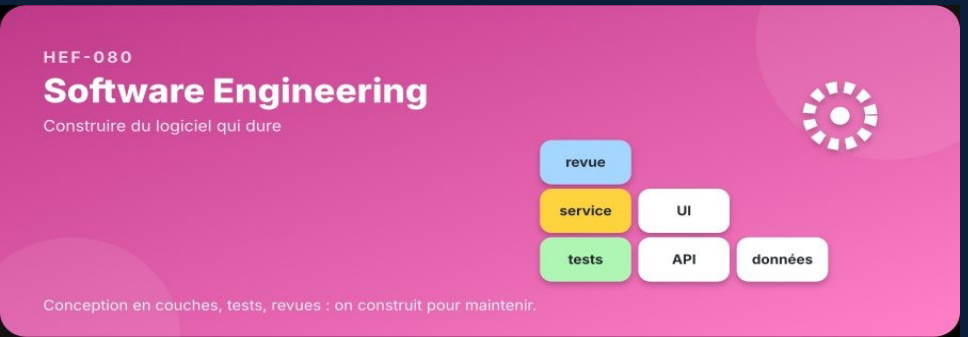
HEF-080 · MISSION 09 / 10

La dette technique

Un raccourci aujourd'hui, des intérêts demain.
Comment emprunter à bon escient, et rembourser.





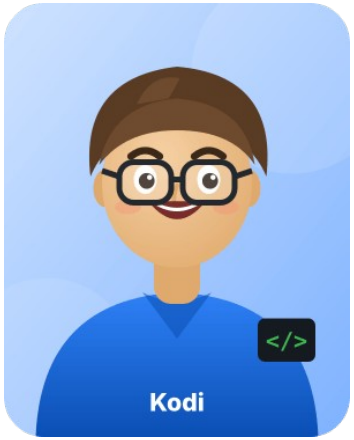
Badge : Gestionnaire de dette

80 min · intermédiaire

L'histoire

Pour livrer à temps une démonstration, Kodi a copié-collé un module au lieu de l'adapter. « On nettoiera après. » Six mois plus tard, une correction doit être appliquée aux deux copies, l'une a été oubliée, et un client perd des données.

« Tu as emprunté, dit Maya. Ce n'est pas interdit. Mais personne n'a noté la dette ni prévu de la rembourser. »



Tu apprends à reconnaître, mesurer, prioriser et rembourser la dette technique.

Ta mission

À la fin, je serai capable de…

1. définir la dette technique et ses « intérêts »
2. distinguer dette délibérée, accidentelle, prudente et imprudente
3. repérer les points chauds (complexité × fréquence de changement)
4. prioriser le remboursement avec un tableau valeur/coût
5. intégrer la dette au quotidien (boy-scout, budget, ADR)

Défi d'échauffement

Défi d'échauffement

Tu empruntes de l'argent : tu obtiens quelque chose tout de suite, mais tu paieras des **intérêts**. Dans quels cas est-ce raisonnable ? Dans quels cas est-ce dangereux ? Quelle information est indispensable pour décider (montant, taux, durée) ? Comment transposer cette analogie à un raccourci dans un projet logiciel ?

Le concept

La métaphore de la dette

Ward Cunningham a comparé les raccourcis de conception à une **dette** : on gagne du temps maintenant, mais on paie des **intérêts** à chaque changement futur (le code est plus dur à comprendre, tester, modifier). Si on ne rembourse pas, les intérêts finissent par absorber toute l'énergie de l'équipe.

Les quatre quadrants (Martin Fowler)

	Imprudente	Prudente
Délibérée	« Pas le temps pour la conception. »	« On livre maintenant et on gérera les conséquences. »
Involontaire	« C'est quoi, la couche d'abstraction ? »	« Maintenant on sait comment on aurait dû faire. »

Le concept

Les quatre quadrants (Martin Fowler) (suite)

Seule la dette **prudente et délibérée** est un choix d'ingénierie : elle est **tracée**, avec une date de remboursement.

Les formes de dette

Forme	Exemples
Code	duplication, fonctions géantes, noms obscurs
Tests	couverture faible, tests instables
Architecture	couplage fort, cycles de dépendances
Documentation	décisions non expliquées, README périmé
Dépendances	versions obsolètes, bibliothèques abandonnées
Infrastructure	déploiements manuels, absence de supervision

Le concept

Les intérêts

Le coût de la dette se voit dans :

- le **temps** pour ajouter une fonctionnalité (qui augmente) ;
- le nombre de **bugs** liés à une zone ;
- la **peur** de toucher à certains fichiers ;
- l'**intégration** de nouveaux arrivants (plus longue).

Mesurer : les points chauds

Une zone vraiment coûteuse est à la fois **complexe** et **souvent modifiée**. Un module horrible mais jamais touché coûte peu. On croise donc :

Le concept

Mesurer : les points chauds (suite)

Priorité ≈ complexité × fréquence de changement

L'historique Git fournit la fréquence ; un outil d'audit fournit la complexité.

Prioriser le remboursement

Valeur du remboursement	Coût faible	Coût élevé
Élevée	**Faire tout de suite**	Planifier en projet
Faible	Faire en passant (boy-scout)	Ne pas faire (ou accepter)

Gérer la dette au quotidien

- **La rendre visible** : un registre (liste) de la dette avec impact, coût estimé, propriétaire.

Le concept

Gérer la dette au quotidien (suite)

- **Boy-scout** : laisser le code un peu plus propre à chaque passage.
- **Budget régulier** : par exemple 15 à 20 % du temps de l'équipe, planifié.
- **Définition de « terminé »** : inclure tests, doc, absence de duplication nouvelle.
- **Journaliser les décisions** (ADR) : « nous avons choisi X par manque de temps ; on revient dessus avant le trimestre prochain ».
- **Automatiser** : intégration continue, analyse statique, mise à jour des dépendances.

Ce que la dette n'est pas

Un mauvais code écrit par négligence n'est pas de la dette stratégique : c'est de la **mauvaise qualité**. Et « on le refera plus tard » sans plan n'est pas un emprunt, c'est un oubli.

Le concept

Un modèle simple du ralentissement

Si chaque nouvelle fonctionnalité ajoute un peu de dette non remboursée, la vitesse de livraison **baisse** progressivement, jusqu'à un point où l'équipe passe l'essentiel de son temps à contourner l'existant.

Exemple pas à pas

Problème : prioriser la dette avec un calcul simple, puis modéliser son effet sur la vitesse.

```
# Chaque module : complexité (audit), nombre de modifications sur 6 mois (historiq
   ue), bugs liés, coût de remboursement (jours)
MODULES = {
    "facturation":  dict(complexite=28, changements=40, bugs=12, cout_jours=8),
    "inscriptions": dict(complexite=15, changements=35, bugs=5,  cout_jours=3),
    "export_pdf":   dict(complexite=35, changements=2,  bugs=1,  cout_jours=10),
    "notifications":dict(complexite=9,  changements=25, bugs=2,  cout_jours=2),
}

def priorite(m):
    point_chaud = m["complexite"] * m["changements"]              # intérêts prob
   ables
    valeur = point_chaud + 50 * m["bugs"]                          # on pondère l
   es bugs
    return valeur / m["cout_jours"]                                # valeur par j
   our de remboursement
```

Exemple pas à pas (suite)

```
classement = sorted(MODULES.items(), key=lambda kv: priorite(kv[1]), reverse=True)
for nom, m in classement:
    print(f"{nom:14s} point chaud={m['complexite'] * m['changements']:5d}  score={
   priorite(m):7.1f}")
```

Lecture : `export_pdf` est le module le plus complexe (35) mais presque jamais modifié (2 fois) : sa dette coûte peu, on **l'accepte**. `facturation` et `inscriptions` cumulent complexité et changements fréquents : c'est là que chaque jour de remboursement rapporte le plus.

Effet de la dette sur la vitesse :

```
def simuler(sprints, capacite=20, dette_par_sprint=2, remboursement=0.0):
    """Chaque sprint : une part de la capacité part dans les intérêts de la dette.
   """
    dette, total = 0.0, 0.0
    historique = []
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    for _ in range(sprints):
        interets = min(capacite * 0.9, dette * 0.15)              # la dette ralen
   tit
        rembourse = min(dette, capacite * remboursement)
        utile = capacite - interets - rembourse
        dette = dette + dette_par_sprint - rembourse
        total += utile
        historique.append(round(utile, 1))
    return total, historique

sans, h1 = simuler(20, remboursement=0.0)
avec, h2 = simuler(20, remboursement=0.15)            # 15 % de la capacité pour r
   embourser
print("travail utile cumulé :", round(sans, 1), "sans remboursement ;", round(avec
   , 1), "avec 15 % de remboursement")
print("sprint 20 :", h1[-1], "vs", h2[-1])
```

Exemple pas à pas (suite)

Lecture : au début, ne rembourse rien semble plus rapide (toute la capacité va aux fonctionnalités). Mais à mesure que la dette grandit, les intérêts mangent la capacité : la vitesse utile **par sprint** de l'équipe qui rembourse dépasse très vite celle de l'équipe qui ne rembourse pas (17,7 contre 14,3 au 20e sprint), et le **cumul** finit lui aussi par la dépasser (356 contre 343). C'est ce point d'équilibre qu'il faut savoir expliquer à la direction.

Un registre de dette (extrait) :

Id	Description	Impact	Coût	Décision	Revue
D-12	La facturation duplique le calcul de TVA du module de commandes	2 bugs/trimestre	3 jours	Rembourser T2	juin
D-15	Export PDF en code legacy, non testé	faible (modifié 2 fois/6 mois)	10 jours	Accepter, tests de caractérisation si modifié	:

À toi, avec aide

Voici quatre situations. Classe chacune dans un des quatre quadrants de dette, dis si c'est un emprunt acceptable, et propose un remboursement :

1. L'équipe livre en deux jours un prototype pour un salon en sachant que le code sera jeté après.
2. Le projet utilise une bibliothèque d'authentification abandonnée depuis 3 ans, personne ne l'a remarqué.
3. Un développeur copie une fonction à trois endroits « pour aller plus vite » sans en parler.
4. Après avoir livré, l'équipe comprend qu'un autre découpage en modules aurait été meilleur.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Délibérée = choisie ; involontaire = découverte après coup.

Indice 2

Prudente = avec un plan ; imprudente = sans réflexion.

Indice 3

Un prototype jeté n'est quasi pas une dette.

Réponse

1. **Délibérée et prudente** : emprunt acceptable car le code sera jeté (aucun intérêt à payer). À écrire dans le registre pour ne pas qu'il survive par accident.
2. **Involontaire et imprudente** : dette de sécurité sérieuse. Remboursement prioritaire : remplacer ou migrer la bibliothèque, mettre en place une veille des dépendances.
3. **Délibérée et imprudente** : à rembourser vite (extraire une fonction commune, tests) ; en parler à l'équipe.
4. **Involontaire et prudente** : normale, on apprend. Planifier une refonte progressive (étranglement), consigner la décision dans un ADR.

À toi, en solo

Outil de points chauds. Écris un script qui, à partir de deux sources, calcule les points chauds d'un projet :

- un dictionnaire `complexite_par_fichier` (tu fournis des valeurs) ;
- un historique de commits simulé : une liste de `(commit, [fichiers modifiés])`.

Pour chaque fichier, calcule la fréquence de changement, le score `complexité × changements`, et affiche le top 5. Ajoute le **couplage temporel** : les paires de fichiers souvent modifiées ensemble (un signe de dépendance cachée). Teste avec 40 commits aléatoires (graine fixe).

À toi, en solo · réponse

Pistes de correction

```
import itertools, random
from collections import Counter

complexite = {"facturation.py": 28, "inscriptions.py": 15, "export_pdf.py": 35, "n
   otifications.py": 9, "utils.py": 20, "modeles.py": 6}

rng = random.Random(4)
fichiers = list(complexite)
poids = [10, 8, 1, 4, 6, 5]
historique = []
for i in range(40):
    modifies = list(set(rng.choices(fichiers, weights=poids, k=rng.randint(1, 3)))
   )
    if "facturation.py" in modifies and rng.random() < 0.8:
        modifies.append("utils.py") if "utils.py" not in modifies else None     #
   couplage caché
    historique.append((f"c{i}", modifies))

changements = Counter(f for _, fs in historique for f in fs)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
scores = {f: complexite[f] * changements[f] for f in complexite}
print("Top points chauds :")
for f, s in sorted(scores.items(), key=lambda x: -x[1])[:5]:
    print(f"  {f:18s} complexité={complexite[f]:3d} changements={changements[f]:3d
   } score={s}")

paires = Counter()
for _, fs in historique:
    for a, b in itertools.combinations(sorted(set(fs)), 2):
        paires[(a, b)] += 1
print("Couplage temporel (fichiers modifiés ensemble) :")
for (a, b), n in paires.most_common(3):
    print(f"  {a} + {b} : {n} commits")
```

Si `facturation.py` et `utils.py` apparaissent ensemble très souvent, c'est un signe d'une dépendance qu'il faudrait expliciter (ou une duplication à supprimer).

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Dette invisible	Personne ne la gère, les intérêts augmentent	Registre de dette
Tout nettoyer par principe	Temps perdu sur ce qui ne bouge pas	Prioriser par points chauds
Pas de budget de remboursement	La dette ne baisse jamais	Part fixe de la capacité
Confondre dette et négligence	On excuse le mauvais travail	Distinguer emprunt prudent et erreur

Mini-boss

Mini-boss

Plan de remboursement. Tu reprends un projet fictif de 12 modules. Pour chacun, invente des mesures plausibles (complexité, fréquence de changement, bugs, absence de tests, dépendances obsolètes, coût de remboursement).

1. Construis le registre de dette (au moins 10 entrées) avec catégorie et quadrant.
2. Calcule les priorités avec un score documenté et affiche une feuille de route sur 3 trimestres (avec une charge de 15 % de la capacité).
3. Simule avec le modèle de ralentissement l'effet de ton plan sur la vitesse sur 12 sprints, comparé à « ne rien faire ».
4. Rédige une note d'une page pour la direction : le problème, l'investissement, le gain attendu, les risques.

Mini-boss · réponse

Pistes de correction

Un bon travail : un tableau de registre en code (liste de dictionnaires) trié par priorité ; un score qui combine points chauds, bugs et risque de sécurité, divisé par le coût ; une feuille de route qui remplit chaque trimestre sans dépasser le budget ; un graphique textuel ou tableau comparant la vitesse « avec plan » et « sans plan » avec le point de croisement identifié ; une note pour la direction écrite sans jargon, avec un indicateur suivi (par exemple délai moyen de livraison d'une fonctionnalité) et des critères d'arrêt (quand cesser de rembourser un module).

Quiz éclair

Quiz éclair

1. La dette technique désigne :

A. un retard de paiement
B. un raccourci de conception dont on paie les intérêts plus tard
C. un bug en production

2. Un point chaud est un code :

A. très ancien
B. sans commentaires
C. à la fois complexe et souvent modifié

3. Une bonne gestion de la dette passe par :

A. la rendre visible et prévoir un budget de remboursement

Quiz éclair (suite)

B. ne jamais emprunter
C. ignorer les petits raccourcis

4. 🔁 *Révision mission 08* : Refactorer, c'est :

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

5. 🔁 *Révision mission 08* : Un test de caractérisation :

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

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quelle « dette » personnelle (rangement, administratif, apprentissage) paies-tu en intérêts en ce moment ?
2. Comment expliquerais-tu à une personne non technique pourquoi il faut du temps pour rembourser une dette technique ?

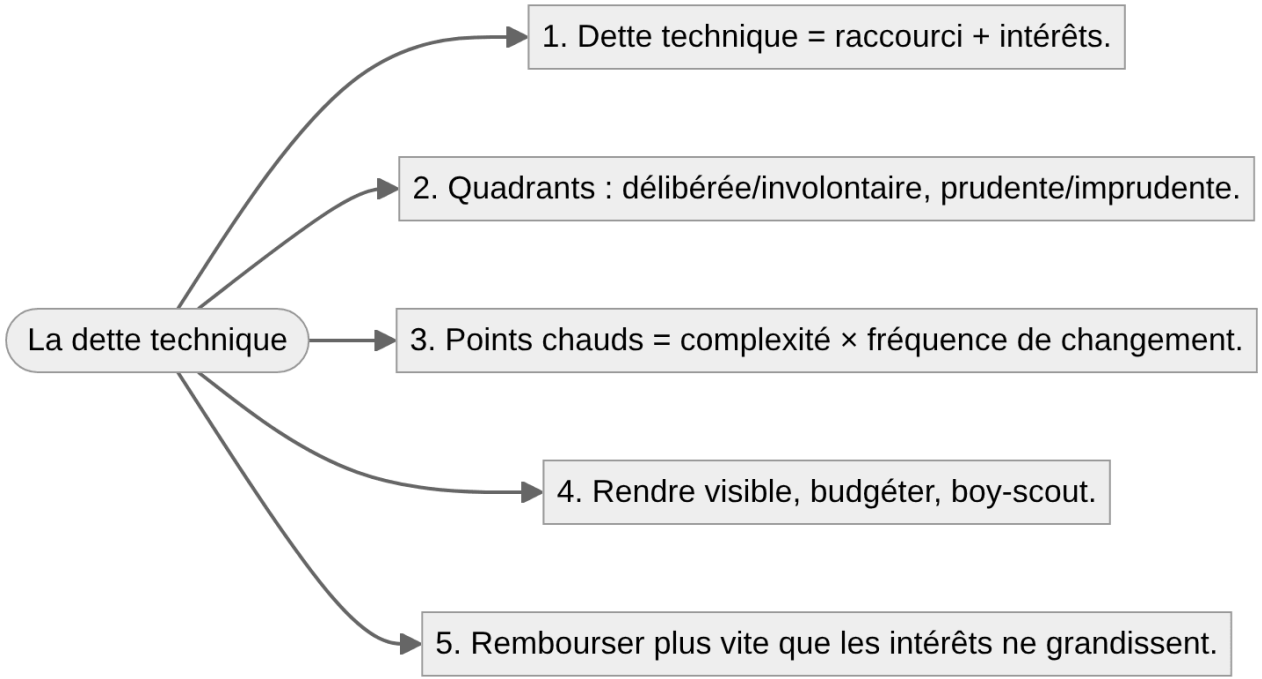
À retenir

À retenir

1. Dette technique = raccourci + intérêts.
2. Quadrants : délibérée/involontaire, prudente/imprudente.
3. Points chauds = complexité × fréquence de changement.
4. Rendre visible, budgéter, boy-scout.
5. Rembourser plus vite que les intérêts ne grandissent.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Gestionnaire de dette

Prochaine mission : Revue de code et décision d'ingénierie

