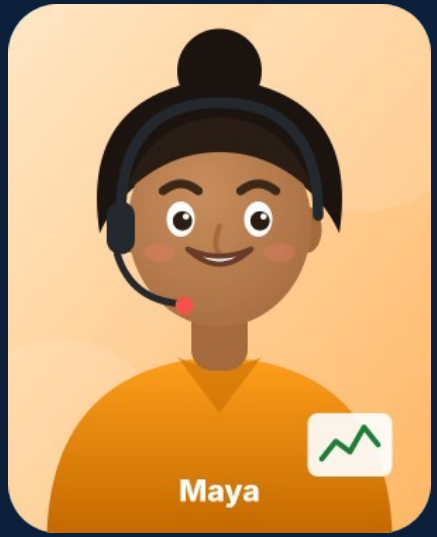
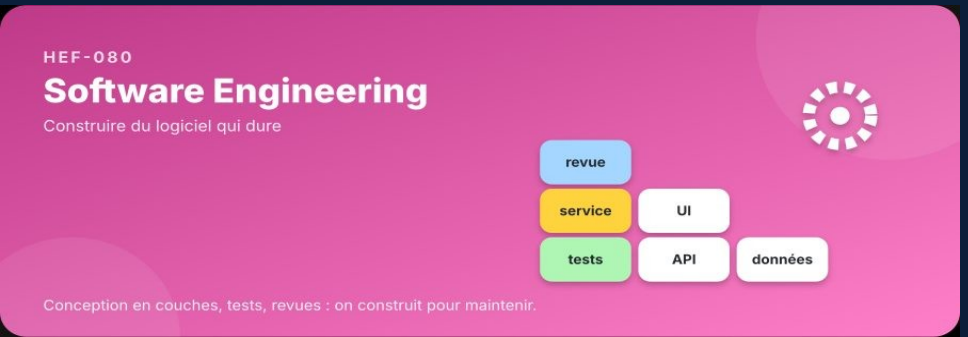
HEF-080 · MISSION 10 / 10

Revue de code et décision d'ingénierie

Faire progresser le code, et l'équipe, par la relecture et par des décisions écrites, argumentées, révisables.



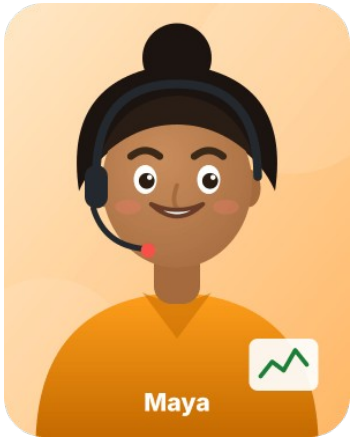


Badge : Ingénieur·e logiciel

95 min · intermédiaire

L'histoire

Kodi soumet sa première demande de fusion. Quinze commentaires arrivent. Un seul dit : « Ce choix de base de données ne tiendra pas la charge, as-tu comparé les alternatives ? » Kodi se vexe, puis réalise que c'est le plus utile.



> *« Une bonne revue ne juge pas la personne, dit Sam. Elle questionne le code, vérifie ce qui compte, et laisse une trace des décisions. »*

Dans cette dernière mission, tu assembles le module : relire, décider, justifier.

Ta mission

À la fin, je serai capable de…

1. mener une revue de code structurée avec une checklist
2. formuler des commentaires constructifs et classés par importance
3. comparer des alternatives avec une matrice de décision pondérée
4. rédiger un ADR (Architecture Decision Record)
5. présenter une décision et ses compromis à une équipe

Défi d'échauffement

Défi d'échauffement

On te donne un texte de 2 pages écrit par un collègue, à relire avant publication. Que vérifies-tu en premier : les fautes d'orthographe, la structure, la justesse des faits, la clarté pour le lecteur ? Dans quel ordre ? Comment tournes-tu une critique pour qu'elle soit **utile** et **bien reçue** ? Écris trois phrases de commentaire : une mauvaise, une meilleure.

Le concept

Pourquoi relire le code ?

Bénéfice	Explication
Qualité	on trouve des bugs et des oublis avant la production
Partage de connaissance	plusieurs personnes comprennent chaque partie
Cohérence	l'équipe garde un style et des conventions
Apprentissage	relecteurs et auteurs progressent

Que regarder (par ordre d'importance)

1. **Le bon problème** : le changement résout-il ce qui était demandé ?
2. **La conception** : place des responsabilités, couplage, simplicité.
3. **La correction** : cas limites, erreurs, concurrence, sécurité.

Le concept

Que regarder (par ordre d'importance) (suite)

4. **Les tests** : couvrent-ils le comportement ? Sont-ils lisibles ?
5. **La lisibilité** : noms, structure, commentaires utiles.
6. **La performance et l'exploitation** : journaux, métriques, migrations.
7. **Le style** : de préférence automatisé (formateur, analyseur).

Une checklist de revue

- Je comprends ce que fait ce changement et pourquoi.
- Le changement est **petit** et cohérent (une seule intention).
- Les cas limites et erreurs sont gérés.
- Pas de secrets, de données sensibles, d'entrées non validées.
- Les tests couvrent le nouveau comportement et l'échec.
- Pas de duplication ni de dette ajoutée sans raison.

Le concept

Une checklist de revue (suite)

- La documentation / les migrations / la configuration suivent.
- Le changement est facile à retirer (retour arrière).

Écrire de bons commentaires

❌	✅
« C'est nul. »	« Cette fonction fait deux choses (valider et enregistrer). Pourrait-on les séparer pour tester la validation seule ? »
« Pourquoi tu fais ça ? »	« Je ne vois pas l'intérêt du cache ici. Quel problème de performance résout-il ? »
« Change ça. »	« Suggestion : `if not membre: return` évite deux niveaux d'imbrication. »

Bonnes pratiques :

Le concept

Écrire de bons commentaires (suite)

- **Parler du code, pas de la personne.**
- **Poser des questions** quand on ne comprend pas.
- **Classer** : *bloquant* (doit être corrigé), *suggestion*, *détail* (préférence), *question*, *félicitation*.
- **Expliquer le pourquoi** du commentaire.
- **Reconnaître le bon travail.**
- Répondre aux commentaires **sans défensive** : le but est le produit, pas l'ego.

Des petites demandes de fusion

La qualité d'une revue baisse vite avec la taille : à 200 lignes, les relecteurs voient des problèmes ; à 2 000, ils survolent. Mieux vaut plusieurs petits changements ciblés.

Le concept

Prendre et documenter des décisions

Une décision d'ingénierie est un **choix entre alternatives**, sous contraintes. Pour qu'elle soit solide :

1. **Énoncer le problème et le contexte.**
2. **Lister les alternatives** (y compris « ne rien faire »).
3. **Choisir des critères** et leur donner un poids.
4. **Évaluer** chaque option sur chaque critère.
5. **Décider** et **écrire les conséquences** (positives et négatives).
6. **Prévoir la revue** : à quelles conditions reconsidérer ?

Le concept

La matrice de décision pondérée

Critère	Poids	Option A	Option B	Option C
Coût	3	4	2	5
Simplicité	2	5	3	2
Évolutivité	3	2	5	3

Score = somme(poids × note). Attention : les chiffres **aident** à discuter, ils ne remplacent pas le jugement ; les poids sont subjectifs. On vérifie la **sensibilité** : le résultat change-t-il si un poids bouge un peu ?

Le concept

L'ADR (Architecture Decision Record)

Un ADR est un court document daté qui garde une décision :

```
# ADR-007 — Utiliser PostgreSQL comme base principale

Statut : accepté (2025-03-12)
Contexte : besoin de transactions, de requêtes complexes, 50 000 utilisateurs à 12
    mois.
Décision : PostgreSQL géré par l'hébergeur.
Alternatives : SQLite (pas de concurrence d'écriture), MongoDB (modèle trop souple
    pour nos jointures).
Conséquences : (+) transactions, outillage, compétences ; (−) coût, administration
   , migrations à gérer.
Revue : si le volume dépasse 2 To ou si une fonctionnalité exige un moteur de rech
   erche dédié.
```

Le concept

Quand décider vite, quand décider bien

Décision	Réversibilité	Effort de réflexion
Nom d'une variable	facile à changer	minimal
Format d'une API publique	difficile	élevé
Choix de la base de données	coûteux	élevé

🧠 ***Astuce*** *: investis la réflexion en proportion du* ***coût de revenir en arrière****.*

Exemple pas à pas

Problème : relire un changement, puis décider d'une solution de stockage avec une matrice pondérée.
Le code soumis :

```
def inscrire(nom, email, cours):
    import sqlite3
    bd = sqlite3.connect("production.db")
    bd.execute("INSERT INTO membres VALUES ('" + nom + "', '" + email + "')")
    bd.commit()
    for c in cours:
        bd.execute("INSERT INTO inscriptions VALUES ('" + email + "', '" + c + "')
   ")
    bd.commit()
    print("ok")
```

Une revue structurée (extrait) :

Exemple pas à pas

Exemple pas à pas (suite)

Gravité	Commentaire
🔴 **Bloquant**	*Sécurité* — Les valeurs sont concaténées dans le SQL : injection possible (`nom = "x'); DROP TABLE membres;--"`). Utiliser des requêtes paramétrées (`?`).
🔴 **Bloquant**	*Atomicité* — Deux `commit` séparés : si l'inscription aux cours échoue, le membre est créé sans cours. Une seule transaction.
🟠 Important	*Conception* — La fonction ouvre sa propre connexion vers `production.db` : impossible à tester sans toucher la production. Injecter la connexion ou un dépôt.
🟠 Important	*Tests* — Aucun test. Au minimum : succès, e-mail en double, échec au milieu.
🟡 Suggestion	`print("ok")` : préférer renvoyer une valeur ou journaliser.
🟢 Félicitation	Bonne idée d'inscrire en même temps aux cours.

Exemple pas à pas

Exemple pas à pas (suite)

Le code après revue :

```
import sqlite3

def inscrire(bd, nom, email, cours):
    with bd:                                              # une seule transaction
        bd.execute("INSERT INTO membres (nom, email) VALUES (?, ?)", (nom, email))
        bd.executemany("INSERT INTO inscriptions (email, cours) VALUES (?, ?)", [(
   email, c) for c in cours])

bd = sqlite3.connect(":memory:")
bd.executescript('''
CREATE TABLE membres (nom TEXT NOT NULL, email TEXT PRIMARY KEY);
CREATE TABLE inscriptions (email TEXT NOT NULL REFERENCES membres(email), cours TE
   XT NOT NULL CHECK (cours <> ''));
''')
inscrire(bd, "x'); DROP TABLE membres;--", "a@x.fr", ["HEF-000", "HEF-010"])
   # traité comme du texte
```

Exemple pas à pas (suite)

```
assert bd.execute("SELECT COUNT(*) FROM membres").fetchone()[0] == 1
try:
    inscrire(bd, "Bob", "b@x.fr", ["HEF-000", ""])        # le 2e cours est inval
   ide
except sqlite3.IntegrityError:
    pass
assert bd.execute("SELECT COUNT(*) FROM membres").fetchone()[0] == 1      # rien
   n'est resté à moitié
print("revue prise en compte")
```

Décision : quelle solution de stockage ?

```
CRITERES = {"fiabilité": 4, "coût": 2, "simplicité": 3, "évolutivité": 3}
OPTIONS = {
    "SQLite":      {"fiabilité": 4, "coût": 5, "simplicité": 5, "évolutivité": 2},
    "PostgreSQL":  {"fiabilité": 5, "coût": 3, "simplicité": 3, "évolutivité": 4},
    "Service cloud géré": {"fiabilité": 5, "coût": 2, "simplicité": 4, "évolutivit
   é": 5},
```

Exemple pas à pas

Exemple pas à pas (suite)

```
}

def scores(poids):
    return {o: sum(poids[c] * notes[c] for c in poids) for o, notes in OPTIONS.ite
   ms()}

base = scores(CRITERES)
print(base)                                                    # classement de dép
   art
# Sensibilité : que se passe-t-il si le coût devient deux fois plus important ?
variante = dict(CRITERES, coût=4)
print(scores(variante))
meilleur = lambda s: max(s, key=s.get)
print("choix stable ?", meilleur(base) == meilleur(variante))
```

Si le classement change quand on bouge un poids, la décision est **fragile** : il faut discuter ce critère plutôt que se cacher derrière le chiffre.

À toi, avec aide

Relis ce code et rédige **cinq commentaires de revue** classés par gravité (bloquant, important, suggestion, question, félicitation) :

```
def verifier_mot_de_passe(utilisateur, saisi):
    f = open("mots_de_passe.txt")
    for ligne in f:
        u, p = ligne.split(":")
        if u == utilisateur and p.strip() == saisi:
            print("connecté")
            return True
    return False
```

À toi, avec aide · réponse

Indices progressifs

Indice 1

Pense sécurité : où et comment sont stockés les mots de passe ?

Indice 2

Pense ressources : le fichier est-il fermé ?

Indice 3

Pense conception : que fait la fonction en plus de vérifier ?

À toi, avec aide · réponse

Réponse

- 🔴 **Bloquant (sécurité)** : les mots de passe sont stockés et comparés **en clair**. Stocker un hachage salé (`bcrypt`, `argon2`, ou `hashlib.scrypt`) et comparer avec `hmac.compare_digest`.
- 🔴 **Bloquant (ressources)** : le fichier n'est jamais fermé ; utiliser `with open(...) as f`.
- 🟠 **Important (conception)** : la fonction vérifie *et* affiche (`print`) *et* lit un fichier en dur. Séparer la source des identifiants (injectée) du contrôle.
- 🟠 **Important (robustesse)** : `ligne.split(":")` casse si le mot de passe contient `:` ou si une ligne est vide ; utiliser `split(":", 1)` et ignorer les lignes mal formées.
- 🟡 **Suggestion** / 🟢 : préférer lever une exception ou journaliser plutôt que `print` ; bonne idée d'avoir isolé la fonction (facile à tester une fois la source injectée).

À toi, en solo

Décision d'ingénierie documentée. HashBox doit choisir comment envoyer ses e-mails transactionnels (confirmations, mots de passe oubliés) :

- A : serveur SMTP maison ;
- B : service d'e-mails géré (API) ;
- C : bibliothèque d'envoi + file de tâches maison.

1. Définis 5 critères (par exemple délivrabilité, coût, effort d'exploitation, simplicité de test, dépendance à un fournisseur) avec poids.
2. Note chaque option, calcule les scores, fais une analyse de **sensibilité** (fais varier chaque poids de ±1 et indique si le classement change).
3. Rédige l'ADR (contexte, décision, alternatives, conséquences, conditions de revue).
4. Écris un petit programme qui produit l'ADR en Markdown à partir de tes données.

À toi, en solo · réponse

Pistes de correction

```
CRITERES = {"délivrabilité": 5, "coût": 2, "effort d'exploitation": 4, "testabilit
   é": 3, "indépendance fournisseur": 2}
OPTIONS = {
    "A SMTP maison":            {"délivrabilité": 2, "coût": 4, "effort d'exploita
   tion": 1, "testabilité": 3, "indépendance fournisseur": 5},
    "B service géré":           {"délivrabilité": 5, "coût": 3, "effort d'exploita
   tion": 5, "testabilité": 4, "indépendance fournisseur": 2},
    "C bibliothèque + file":    {"délivrabilité": 3, "coût": 4, "effort d'exploita
   tion": 2, "testabilité": 4, "indépendance fournisseur": 4},
}

def scores(poids):
    return {o: sum(poids[c] * n[c] for c in poids) for o, n in OPTIONS.items()}

base = scores(CRITERES)
gagnant = max(base, key=base.get)
instable = []
for critere in CRITERES:
    for delta in (-1, 1):
```

Pistes de correction (suite)

```
        variante = dict(CRITERES, **{critere: max(1, CRITERES[critere] + delta)})
        s = scores(variante)
        if max(s, key=s.get) != gagnant:
            instable.append((critere, delta))

def adr(numero, titre, contexte, decision, alternatives, consequences, revue):
    return (f"# ADR-{numero:03d} — {titre}\n\nStatut : accepté\n\n"
            f"## Contexte\n{contexte}\n\n## Décision\n{decision}\n\n"
            f"## Alternatives\n" + "\n".join(f"- {a}" for a in alternatives) +
            "\n\n## Conséquences\n" + "\n".join(f"- {c}" for c in consequences) +
   f"\n\n## Revue\n{revue}\n")

print(base, "gagnant :", gagnant, "| fragile si :", instable or "aucune variation
   testée ne change le choix")
print(adr(12, "Envoyer les e-mails transactionnels via un service géré",
          "Les confirmations d'inscription doivent arriver; notre équipe n'a pas d
   e compétence d'exploitation SMTP.",
          f"Nous utilisons l'option : {gagnant}.",
          [o for o in OPTIONS if o != gagnant],
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        ["(+) bonne délivrabilité, peu d'exploitation", "(−) coût variable, dépe
   ndance au fournisseur : isolée derrière une interface Messagerie"],
        "Reconsidérer si le coût dépasse 500 euros par mois ou si le fournisseur
    change ses conditions."))
```

La sensibilité montre si la décision tient quand les priorités bougent un peu. L'interface `Messagerie` (mission 5) limite le coût de changer d'avis.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Critiquer la personne	Défensive, conflit	Parler du code, expliquer le pourquoi
Demandes de fusion énormes	Revue superficielle	Petits changements ciblés
Décision orale non écrite	Personne ne se souvient du pourquoi	ADR court et daté
Matrice de décision comme vérité	Les poids sont subjectifs	Analyse de sensibilité et jugement

Mini-boss

Mini-boss

Dossier d'ingénierie final du module HEF-080. Reprends l'application d'inscriptions en architecture hexagonale de la mission 7 (ou une application de ton choix) et mène une démarche complète :

1. **Audit** : exécute l'outil de la mission 1 (complexité, longueur, imbrication) et identifie 3 points faibles.
2. **Revue** : rédige au moins 8 commentaires de revue classés par gravité sur le code (dont une injection de dépendances manquante, un cas limite oublié et une duplication).
3. **Refactoring** : corrige les trois plus importants en petits pas, avec tests de caractérisation et journal des pas.
4. **Décision** : choisis un ajout (par exemple un cache, un deuxième canal de notification ou un nouveau moteur de stockage) avec une matrice pondérée, sensibilité, et un ADR.
5. **Dette** : tiens un registre de dette de 6 entrées et un plan de remboursement.
6. **Synthèse** : une page pour une équipe qui reprendrait le projet : architecture, décisions, dette, prochaines étapes.

Mini-boss · réponse

Pistes de correction

Un bon dossier : audit avant/après avec chiffres ; commentaires de revue formulés de façon constructive avec gravité et justification ; refactoring en pas vérifiés (tests de caractérisation, journal) ; une matrice de décision dont la sensibilité est discutée et un ADR complet (contexte, décision, alternatives, conséquences, revue) ; un registre de dette avec priorités calculées par points chauds ; une synthèse qui permet à un nouvel arrivant de comprendre où est le cœur, ce qui est volontairement imparfait et pourquoi. Il relie explicitement chaque choix à une notion du module (cohésion/couplage, SOLID, patrons, couches, refactoring, dette).

Quiz éclair

Quiz éclair

1. Un bon commentaire de revue :

A. porte sur le code et explique le pourquoi
B. juge la personne
C. est toujours bloquant

2. Un ADR sert à :

A. lister les bugs
B. garder une décision et ses raisons
C. remplacer les tests

3. L'analyse de sensibilité d'une matrice de décision vérifie :

A. la vitesse du code

Quiz éclair

Quiz éclair (suite)

B. la couverture des tests
C. si le choix change quand les poids varient

4. 🔁 *Révision mission 09* : La dette technique désigne :

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

5. 🔁 *Révision mission 09* : Un point chaud est un code :

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

Quiz éclair · réponse

Réponses

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

Je réfléchis

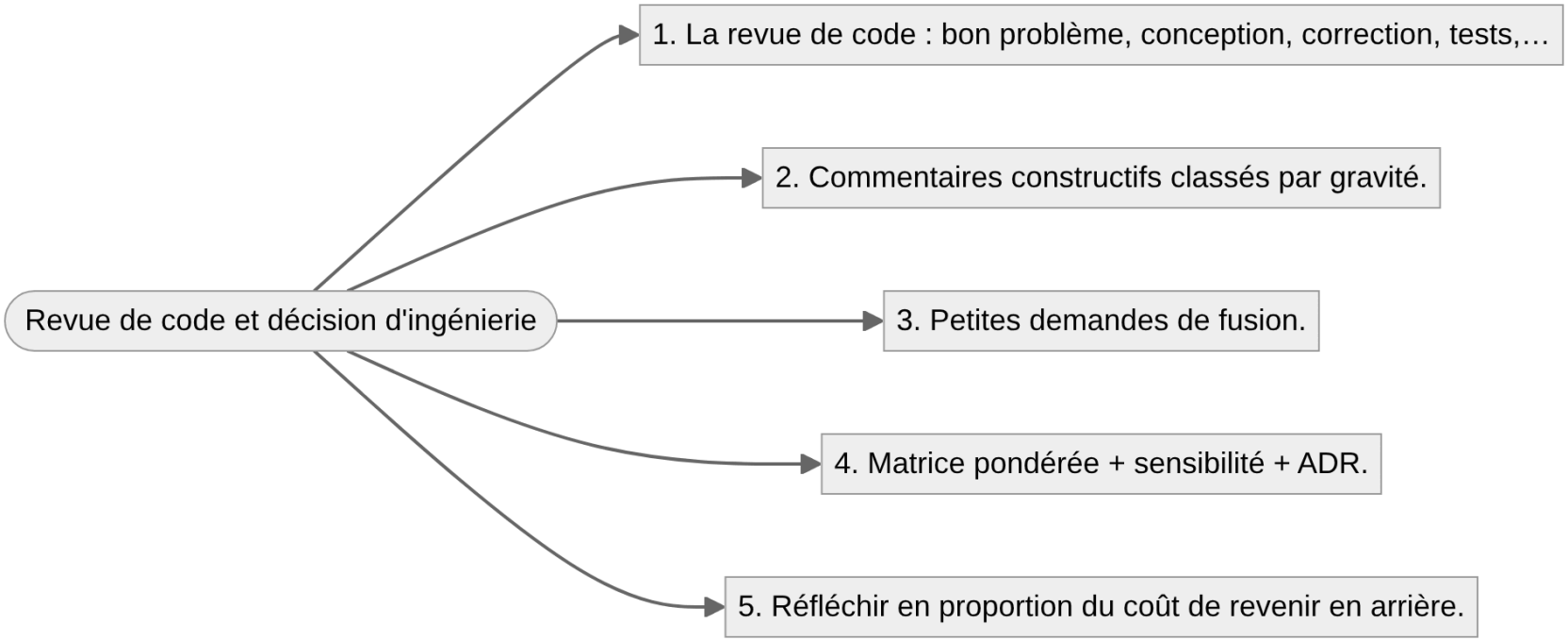
1. Quel commentaire reçu (en revue, en retour de travail) t'a le plus fait progresser, et pourquoi ?
2. Quelle décision que tu prends souvent mériterait d'être écrite pour que tu t'en souviennes dans un an ?

À retenir

1. La revue de code : bon problème, conception, correction, tests, lisibilité.
2. Commentaires constructifs classés par gravité.
3. Petites demandes de fusion.
4. Matrice pondérée + sensibilité + ADR.
5. Réfléchir en proportion du coût de revenir en arrière.

À retenir

Le schéma de la mission



Félicitations

Félicitations

Tu as terminé **HEF-080 — Génie logiciel**. Tu sais juger la qualité d'un code, structurer avec cohésion et faible couplage, appliquer SOLID et les patrons, concevoir des architectures propres, refactorer en sécurité, gérer la dette technique, relire du code et documenter des décisions.
→ Prochain module : **HEF-090 — Compilateurs et langages**

MISSION ACCOMPLIE

+120 XP

Badge débloqué : Ingénieur·e logiciel

Fin du module HEF-080. Bravo !

