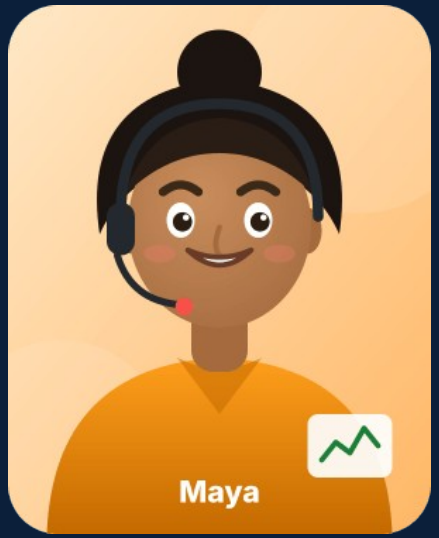
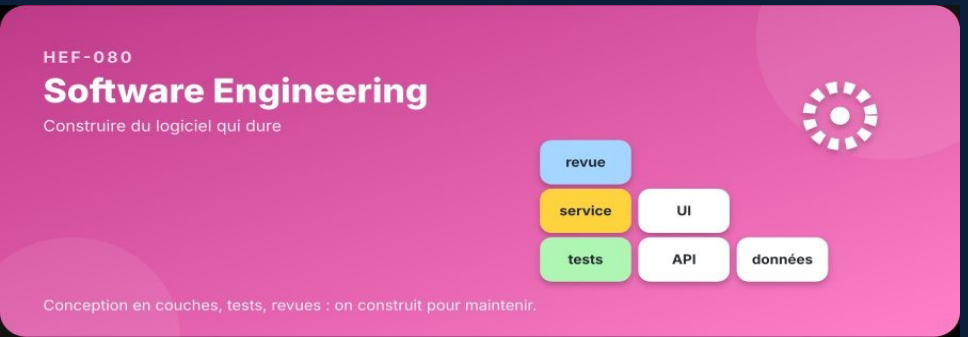
HEF-080 · MISSION 02 / 10

Cohésion et couplage



Un bon module fait une chose et dépend de peu. Deux mots pour juger n'importe quel design.

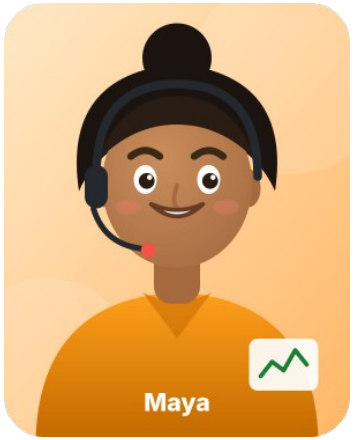


Badge : Architecte de modules

75 min · intermédiaire

L'histoire

La classe `GestionnaireMembres` de HashBox lit la base, valide les données, calcule des statistiques, formate des rapports PDF et envoie des e-mails. Quand Kodi change le format du PDF, les e-mails cessent de fonctionner. Quand il veut tester le calcul des statistiques, il doit installer une base de données et un serveur de mail.



> *« Cette classe a trop de raisons de changer, dit Sam, et elle dépend de tout. Il faut la découper : chaque pièce soudée en interne, reliée aux autres par des fils fins. »*

Tu apprends à juger et à améliorer la structure d'un code avec la cohésion et le couplage.

Ta mission

À la fin, je serai capable de…

1. définir cohésion et couplage et expliquer pourquoi viser forte cohésion et faible couplage
2. reconnaître les formes de couplage (données, global, temporel, par implémentation)
3. découper une classe qui a plusieurs responsabilités
4. utiliser l'injection de dépendances pour réduire le couplage
5. rendre un code testable sans base de données ni réseau

Défi d'échauffement

Une cuisine de restaurant : le cuisinier cuisine, le serveur sert, le caissier encaisse. Quels sont les avantages si chacun a **un seul rôle clair** ? Que se passe-t-il quand le cuisinier doit aussi servir, encaisser et faire le ménage ? Imagine maintenant que le serveur doit connaître **la recette** de chaque plat pour prendre une commande : qu'est-ce que cela change si la recette évolue ?

Le concept

La cohésion

La **cohésion** mesure à quel point les éléments d'un module (fonctions d'une classe, classes d'un paquet) **appartiennent ensemble**.

- **Forte cohésion** : le module fait **une chose** bien définie ; tout ce qu'il contient y contribue.
- **Faible cohésion** : un « fourre-tout » (`utils.py`, `Manager`, `Helper`).

Test : peux-tu décrire le module en **une phrase sans « et »** ?

Le couplage

Le **couplage** mesure à quel point un module **dépend** des autres. Plus il est fort, plus un changement ici casse ailleurs.

Le concept

Le couplage (suite)

Forme de couplage	Exemple	Gravité
Par **données** (paramètres simples)	`calculer(prix, quantite)`	faible, acceptable
Par **interface** (un contrat stable)	dépend d'un `Protocol`	faible
Par **implémentation**	utilise les détails internes d'une classe	forte
Par **état global**	variable globale modifiée par plusieurs modules	forte
Temporel	« appelle `ouvrir()` avant `lire()` »	moyenne

L'objectif

> ***Forte cohésion, faible couplage.***

Un module cohésif change pour **une seule raison** ; un module peu couplé change **sans entraîner** les autres.

Le concept

L'injection de dépendances

Au lieu de **créer** ses dépendances à l'intérieur, un objet les **reçoit** de l'extérieur.

```
# Fortement couplé : la classe construit elle-même sa dépendance
class Notification:
    def __init__(self):
        self.messagerie = ServeurMail("smtp.example.org")     # impossible à rempl
   acer

# Faiblement couplé : la dépendance est fournie
class Notification:
    def __init__(self, messagerie):
        self.messagerie = messagerie                           # n'importe quoi qu
   i a .envoyer()
```

Le concept

L'injection de dépendances (suite)

Avantages : on peut fournir un **faux** (pour les tests), changer de fournisseur sans toucher à la classe, et les dépendances sont **visibles**.

La loi de Déméter (parler seulement à ses voisins)

Éviter les chaînes `a.b().c().d()` : elles lient le code à la structure interne de plusieurs objets. Préfère demander à l'objet proche de faire le travail (`a.faire_x()`).

Mesurer

- Nombre d'`import` entre modules (dépendances).
- Nombre de raisons de changer d'une classe.
- Difficulté à écrire un test : si tester une classe demande de monter la moitié du système, le couplage est trop fort.

Exemple pas à pas

Problème : découper `GestionnaireMembres` et le rendre testable.

Avant : tout est mélangé, impossible à tester sans base de données.

```
class GestionnaireMembresAvant:
    def rapport_mensuel(self, mois):
        import sqlite3
        bd = sqlite3.connect("production.db")                    # dépendance co
   dée en dur
        lignes = bd.execute("SELECT nom, score FROM membres").fetchall()
        moyenne = sum(s for _, s in lignes) / len(lignes)
        texte = f"Rapport {mois}\n" + "\n".join(f"{n}: {s}" for n, s in lignes) +
   f"\nMoyenne: {moyenne:.1f}"
        print("envoi d'un e-mail à direction@hashbox.example")      # effet de bor
   d direct
        return texte
```

Exemple pas à pas (suite)

Après : trois responsabilités, trois objets, des dépendances injectées.

```
class DepotMembres:
    """Accès aux données : seule classe qui connaît la source."""
    def __init__(self, lignes):
        self._lignes = lignes
    def tous(self):
        return list(self._lignes)

def statistiques(membres):
    """Calcul pur : aucune dépendance externe."""
    scores = [s for _, s in membres]
    return {"nombre": len(scores), "moyenne": sum(scores) / len(scores) if scores
   else 0}

def formater_rapport(mois, membres, stats):
    """Présentation : texte uniquement."""
    corps = "\n".join(f"{n}: {s}" for n, s in membres)
```

Exemple pas à pas (suite)

```
    return f"Rapport {mois}\n{corps}\nMoyenne: {stats['moyenne']:.1f}"

class ServiceRapport:
    """Orchestration : reçoit ses dépendances."""
    def __init__(self, depot, messagerie):
        self.depot, self.messagerie = depot, messagerie

    def envoyer_rapport_mensuel(self, mois, destinataire):
        membres = self.depot.tous()
        rapport = formater_rapport(mois, membres, statistiques(membres))
        self.messagerie.envoyer(destinataire, rapport)
        return rapport

# --- test sans base ni réseau : on injecte des faux ---
class FausseMessagerie:
    def __init__(self):
        self.envoyes = []
    def envoyer(self, destinataire, texte):
        self.envoyes.append((destinataire, texte))
```

Exemple pas à pas

Exemple pas à pas (suite)

```
depot = DepotMembres([("Ada", 90), ("Bob", 50)])
messagerie = FausseMessagerie()
rapport = ServiceRapport(depot, messagerie).envoyer_rapport_mensuel("mars", "direc
   tion@hashbox.example")
assert "Moyenne: 70.0" in rapport
assert messagerie.envoyes[0][0] == "direction@hashbox.example"
assert statistiques([]) == {"nombre": 0, "moyenne": 0}
print("tests passés sans base de données ni serveur de mail")
```

Lecture :

- chaque pièce a **une raison de changer** (source de données, calcul, présentation, envoi) ;
- `statistiques` et `formater_rapport` sont des **fonctions pures** : triviales à tester ;
- `ServiceRapport` ne sait pas **comment** on envoie ou où on lit : il reçoit des objets qui ont les bonnes méthodes ;

Exemple pas à pas (suite)

- remplacer le mail par un message Slack revient à écrire une autre classe envoyer, sans toucher au reste.

À toi, avec aide

À toi, avec aide

Voici une classe :

```
class Commande:
    def __init__(self, lignes):
        self.lignes = lignes
    def total(self):
        return sum(p * q for p, q in self.lignes)
    def sauvegarder(self):
        open("commandes.txt", "a").write(str(self.lignes) + "\n")
    def envoyer_confirmation(self, email):
        print("mail envoyé à", email)
    def en_pdf(self):
        return "PDF: " + str(self.total())
```

1. Décris en une phrase par responsabilité pourquoi la cohésion est faible.
2. Propose une découpe en 3 ou 4 éléments.

À toi, avec aide (suite)

3. Réécris le code avec injection et écris un test qui n'écrit aucun fichier et n'envoie aucun mail.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Responsabilités : calculer, persister, notifier, présenter.

Indice 2

`Commande` ne devrait contenir que ses données et son total.

Indice 3

Passe un `depot` et une `messagerie` à un service qui orchestre.

À toi, avec aide · réponse

Réponse

Responsabilités mélangées : calcul (métier), persistance (fichier), notification (mail), présentation (PDF). Elles changent pour des raisons différentes.

```
class Commande:
    def __init__(self, lignes):
        self.lignes = lignes
    def total(self):
        return sum(p * q for p, q in self.lignes)

class DepotCommandes:
    def __init__(self):
        self.stockees = []
    def sauvegarder(self, commande):
        self.stockees.append(commande)

class ServiceCommande:
    def __init__(self, depot, messagerie):
        self.depot, self.messagerie = depot, messagerie
```

À toi, avec aide · réponse

Réponse (suite)

```
    def valider(self, commande, email):
        self.depot.sauvegarder(commande)
        self.messagerie.envoyer(email, f"Commande de {commande.total()} euros")

def en_pdf(commande):
    return "PDF: " + str(commande.total())

class FausseMessagerie:
    def __init__(self): self.envoyes = []
    def envoyer(self, a, texte): self.envoyes.append((a, texte))

depot, mail = DepotCommandes(), FausseMessagerie()
ServiceCommande(depot, mail).valider(Commande([(10, 2), (5, 1)]), "a@x.fr")
assert depot.stockees[0].total() == 25 and mail.envoyes == [("a@x.fr", "Commande d
   e 25 euros")]
assert en_pdf(depot.stockees[0]) == "PDF: 25"
print("ok")
```

À toi, en solo

Mesurer le couplage. Écris un outil qui analyse une petite « application » représentée par un dictionnaire `{module: [modules importés]}` et calcule pour chaque module :

- le couplage **sortant** (de combien de modules il dépend) ;
- le couplage **entrant** (combien de modules dépendent de lui) ;
- s'il appartient à un **cycle** de dépendances (A → B → A).

Teste sur une application de 7 modules qui contient un cycle. Propose ensuite une modification (introduire une interface) qui casse le cycle.

À toi, en solo · réponse

Pistes de correction

```
def analyser(app):
    entrant = {m: 0 for m in app}
    for m, deps in app.items():
        for d in deps:
            entrant[d] = entrant.get(d, 0) + 1

    def dans_cycle(m):
        vus, pile = set(), list(app.get(m, []))
        while pile:
            x = pile.pop()
            if x == m:
                return True
            if x not in vus:
                vus.add(x)
                pile.extend(app.get(x, []))
        return False

    return {m: {"sortant": len(app[m]), "entrant": entrant.get(m, 0), "cycle": dan
   s_cycle(m)} for m in app}
```

À toi, en solo · réponse

Pistes de correction (suite)

```
app = {
    "web": ["services", "modeles"],
    "services": ["modeles", "depot", "mail"],
    "depot": ["modeles", "services"],            # cycle services <-> depot
    "modeles": [],
    "mail": [],
    "cli": ["services"],
    "outils": [],
}
for m, info in analyser(app).items():
    print(f"{m:9s} {info}")
```

Le cycle `services ↔ depot` se casse en faisant dépendre `services` d'une **interface** `Depot` (définie dans `services`), et en faisant implémenter cette interface par `depot` : la dépendance s'inverse (`depot` → `services`), le cycle disparaît.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Classe « manager » qui fait tout	Faible cohésion, changements en cascade	Une raison de changer par classe
Créer les dépendances à l'intérieur	Impossible à tester ou remplacer	Injection de dépendances
Variables globales partagées	Couplage caché	Passer l'état en paramètre
Chaînes d'appels `a.b.c.d`	Dépendance à la structure interne	Demander à l'objet proche de faire le travail

Mini-boss

Mini-boss

Réorganiser un script monolithique. Voici la description d'un script de 200 lignes de HashBox : il lit un fichier CSV d'inscriptions, valide chaque ligne, écrit les valides dans une base SQLite, calcule des statistiques par cours, génère un rapport texte, l'écrit dans un fichier et envoie un e-mail récapitulatif.

1. Écris une version monolithique minimale (une seule fonction de 40 lignes environ).
2. Découpe-la en modules à forte cohésion : `lecture`, `validation`, `depot`, `statistiques`, `rapport`, `notification`, `service` (orchestration).
3. Injecte les dépendances externes (lecture du fichier, base, messagerie).
4. Écris au moins 8 tests unitaires qui n'utilisent ni fichier, ni base réelle, ni réseau.
5. Dessine (Mermaid) le graphe de dépendances avant et après, avec description en texte.

Mini-boss · réponse

Pistes de correction

Un bon travail : fonctions pures pour `valider`, `statistiques`, `formater` ; une classe `DepotInscriptions` avec un constructeur qui reçoit une connexion (SQLite en mémoire pour les tests) ; un `Protocol` ou simple convention `.envoyer(destinataire, texte)` pour la messagerie ; `ServiceImport.executer(lignes_csv)` reçoit tout par le constructeur. Les tests : validation de lignes valides/invalides, statistiques par cours, rapport formaté, import avec faux dépôt et fausse messagerie vérifiant les appels. Le schéma « avant » montre une flèche unique vers tout ; « après » montre `service` au centre, dépendant d'interfaces, avec les détails (SQLite, SMTP) aux bords.

Quiz éclair

Quiz éclair

1. Une forte cohésion signifie :

A. le module dépend de beaucoup d'autres
B. le module est très long
C. le module fait une chose bien définie

2. Un faible couplage permet :

A. de modifier un module sans casser les autres
B. de réduire le nombre de tests
C. d'éviter les interfaces

3. L'injection de dépendances consiste à :

A. créer les dépendances dans la classe

Quiz éclair (suite)

B. fournir les dépendances de l'extérieur
C. supprimer les dépendances

4. 🔁 *Révision mission 01* : DRY signifie :

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

5. 🔁 *Révision mission 01* : La complexité cyclomatique compte :

A. les chemins d'exécution indépendants
B. les lignes de code
C. les fichiers

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quel « fourre-tout » de ta vie (tiroir, dossier, fichier) mélange des choses sans rapport ?
2. Pourquoi un code facile à tester est-il presque toujours un code bien structuré ?

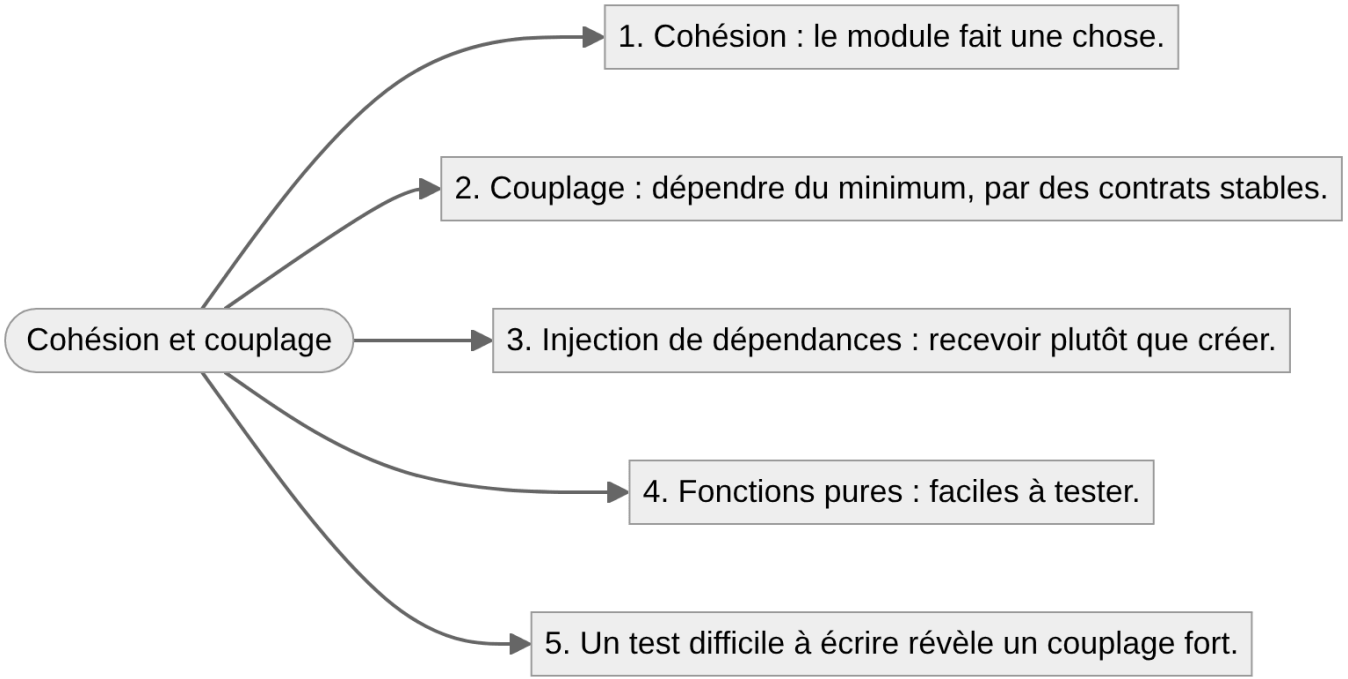
À retenir

À retenir

1. Cohésion : le module fait une chose.
2. Couplage : dépendre du minimum, par des contrats stables.
3. Injection de dépendances : recevoir plutôt que créer.
4. Fonctions pures : faciles à tester.
5. Un test difficile à écrire révèle un couplage fort.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Architecte de modules

Prochaine mission : Les principes SOLID

