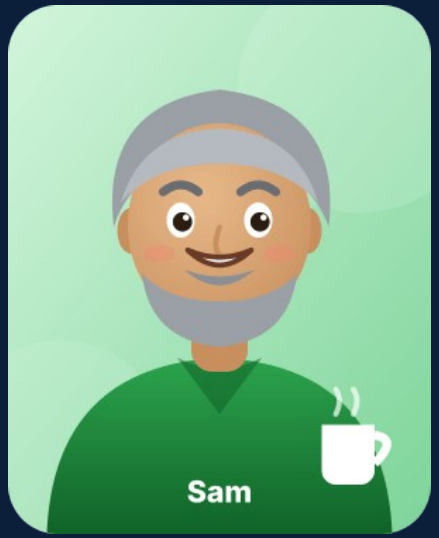
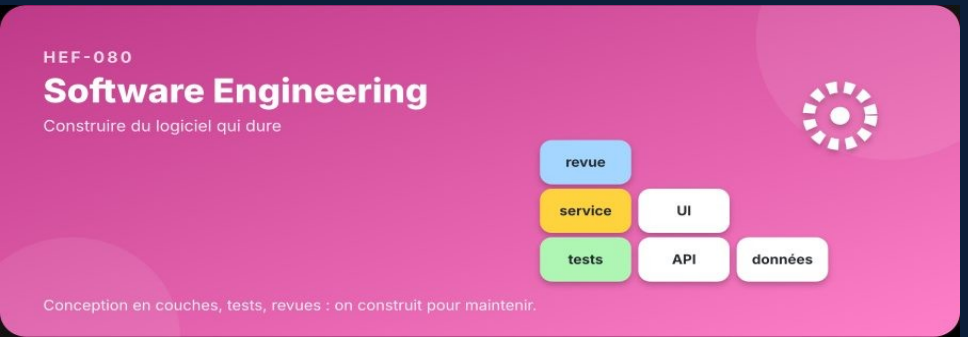
HEF-080 · MISSION 03 / 10

Les principes SOLID

Cinq principes de conception qui rendent le code facile à étendre sans le casser.





Badge : Maître des cinq principes

90 min · intermédiaire

L'histoire

HashBox ajoute un nouveau moyen de paiement toutes les deux semaines. À chaque fois, Kodi ouvre la fonction `payer`, ajoute un `elif`, et casse parfois les paiements existants.

> *« Ton code est ouvert à la casse et fermé à l'extension, plaisante Sam. Il faut l'inverser. SOLID, c'est cinq règles pour y parvenir. »*



Tu apprends les cinq principes SOLID avec des exemples concrets en Python.

Ta mission

À la fin, je serai capable de…

1. énoncer les cinq principes S-O-L-I-D
2. reconnaître la violation de chacun dans un code
3. refactorer un code vers le principe ouvert/fermé
4. utiliser des `Protocol` pour l'inversion de dépendances
5. savoir quand ne pas appliquer SOLID à la lettre

Défi d'échauffement

Défi d'échauffement

Un restaurant propose aujourd'hui trois modes de paiement (carte, espèces, chèque-repas). Le manager veut en ajouter un quatrième tous les mois. Pour chaque ajout, faut-il **réécrire** le fonctionnement de la caisse ou simplement **brancher** un nouveau module de paiement ? Dessine l'idée d'une caisse qui accepte n'importe quel module respectant une même « prise ».

Le concept

S — Responsabilité unique (*Single Responsibility*)

> *Une classe doit avoir **une seule raison de changer**.*

Reprise de la mission précédente : une classe qui calcule, sauvegarde et formate a trois raisons de changer.

O — Ouvert/fermé (*Open/Closed*)

> *Le code doit être **ouvert à l'extension**, **fermé à la modification**.*

Ajouter un comportement = ajouter du code nouveau, pas modifier du code qui marche.

Le concept

O — Ouvert/fermé (*Open/Closed*) (suite)

```
# Violation : chaque nouveau moyen de paiement oblige à modifier cette fonction
def payer(moyen, montant):
    if moyen == "carte":
        ...
    elif moyen == "especes":
        ...
```

L — Substitution de Liskov (*Liskov Substitution*)

> *Un sous-type doit pouvoir **remplacer** son type de base sans casser le programme.*

Exemple de violation : `Carre` hérite de `Rectangle`, mais modifier la largeur d'un `Carre` modifie aussi sa hauteur, ce qui surprend le code écrit pour des rectangles. Un sous-type ne doit pas **renforcer les préconditions** ni **affaiblir les garanties**.

Le concept

I — Ségrégation des interfaces (*Interface Segregation*)

> *Mieux vaut **plusieurs petites interfaces** qu'une grosse.*

Une interface `Appareil` avec `imprimer`, `scanner`, `faxer` oblige une imprimante simple à implémenter `scanner` (qu'elle ne sait pas faire).

D — Inversion des dépendances (*Dependency Inversion*)

> *Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau ; **tous deux dépendent d'abstractions**.*

Le service de paiement dépend d'une interface `Paiement`, pas de `Stripe` ou `PayPal` directement.

Le concept

Un résumé

Lettre	Question à se poser
S	Combien de raisons de changer ?
O	Puis-je ajouter un cas sans modifier l'existant ?
L	Puis-je substituer un sous-type sans surprise ?
I	Les clients dépendent-ils de méthodes qu'ils n'utilisent pas ?
D	Mon cœur dépend-il de détails (base, API) ou d'abstractions ?

Les `Protocol` de Python

Python permet de décrire une interface par ses méthodes (*typage structurel*) :

Le concept

Les `Protocol` de Python (suite)

```
from typing import Protocol

class Paiement(Protocol):
    def payer(self, montant: float) -> str: ...
```

Toute classe qui possède une méthode `payer(montant)` satisfait l'interface, sans héritage.

Attention à l'excès

SOLID sert à **gérer le changement**. Appliqué partout, par réflexe, il produit des dizaines de petites classes inutiles. On applique un principe **quand la douleur apparaît** (un deuxième moyen de paiement), pas avant (YAGNI).

Exemple pas à pas

Problème : passer d'une chaîne de `elif` à une conception ouverte/fermée.

Avant :

```
def payer_v1(moyen, montant):
    if moyen == "carte":
        return f"carte: {montant:.2f} EUR (frais 1,5 %: {montant * 0.015:.2f})"
    elif moyen == "especes":
        return f"especes: {montant:.2f} EUR"
    elif moyen == "cheque_repas":
        return f"cheque_repas: {min(montant, 25):.2f} EUR (plafond 25)"
    raise ValueError("moyen inconnu")
```

Après : chaque moyen de paiement est une classe qui respecte une interface ; le service ne change jamais.

Exemple pas à pas (suite)

```
from typing import Protocol

class MoyenPaiement(Protocol):
    nom: str
    def payer(self, montant: float) -> str: ...

class Carte:
    nom = "carte"
    def payer(self, montant):
        return f"carte: {montant:.2f} EUR (frais 1,5 %: {montant * 0.015:.2f})"

class Especes:
    nom = "especes"
    def payer(self, montant):
        return f"especes: {montant:.2f} EUR"

class ChequeRepas:
    nom = "cheque_repas"
    plafond = 25
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    def payer(self, montant):
        return f"cheque_repas: {min(montant, self.plafond):.2f} EUR (plafond {self
   .plafond})"

class Caisse:
    """Dépend de l'abstraction MoyenPaiement (D) ; ouverte à l'ajout de moyens (O)
    """
    def __init__(self, moyens):
        self.moyens = {m.nom: m for m in moyens}
    def payer(self, nom, montant):
        if nom not in self.moyens:
            raise ValueError("moyen inconnu")
        return self.moyens[nom].payer(montant)

caisse = Caisse([Carte(), Especes(), ChequeRepas()])

# Même comportement qu'avant
for moyen, montant in [("carte", 100), ("especes", 12.5), ("cheque_repas", 40)]:
    assert caisse.payer(moyen, montant) == payer_v1(moyen, montant)
```

Exemple pas à pas

Exemple pas à pas (suite)

```
# Extension sans modifier le code existant : un nouveau moyen
class Virement:
    nom = "virement"
    def payer(self, montant):
        return f"virement: {montant:.2f} EUR"

caisse.moyens["virement"] = Virement()
print(caisse.payer("virement", 80))
```

Ce qu'on a gagné : ajouter `Virement` n'a modifié **ni** `Caisse`, **ni** les autres moyens, **ni** leurs tests. C'est l'ouverture/fermeture.

Une violation de Liskov, pour comprendre :

Exemple pas à pas

Exemple pas à pas (suite)

```
class Rectangle:
    def __init__(self, l, h):
        self.l, self.h = l, h
    def definir_largeur(self, l):
        self.l = l
    def aire(self):
        return self.l * self.h

class Carre(Rectangle):
    def definir_largeur(self, l):
        self.l = self.h = l          # surprend le code qui attend un rectangle

def agrandir_largeur(r):
    r.definir_largeur(10)
    return r.aire()                  # on attend 10 * h

print(agrandir_largeur(Rectangle(2, 5)))    # 50
print(agrandir_largeur(Carre(2, 2)))        # 100 : le comportement attendu n'est
   plus respecté
```

À toi, avec aide

Identifie le principe SOLID violé dans chaque cas et propose la correction :

1. Une classe `Utilisateur` contient `valider_email`, `sauvegarder_en_base`, `envoyer_mail_bienvenue` et `afficher_en_html`.
2. Pour chaque nouveau format d'export (CSV, JSON, PDF), on ajoute un `elif` dans `exporter(format, donnees)`.
3. Une interface `Oiseau` impose `voler()`, et `Pingouin` implémente `voler()` en levant une exception.
4. Un `ServiceCommande` crée lui-même `ConnexionMySQL()` dans son constructeur.
5. Une interface `Travailleur` impose `travailler()` et `manger()`, et un `Robot` doit implémenter `manger()` pour rien.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Pense aux lettres dans l'ordre : S, O, L, D, I.

Indice 2

Le 3 : un sous-type qui ne sait pas faire ce que promet son type de base.

Indice 3

Le 5 : une interface trop large impose des méthodes inutiles.

Réponse

1. **S** (responsabilité unique) → séparer validation, persistance, notification, présentation.
2. **O** (ouvert/fermé) → un `Exporteur` par format derrière une interface commune, choisi dans un dictionnaire.
3. **L** (Liskov) → un `Pingouin` n'est pas un `OiseauVolant` : séparer `Oiseau` et `OiseauVolant`.
4. **D** (inversion des dépendances) → injecter un `Depot` (interface), pas une classe concrète.
5. **I** (ségrégation des interfaces) → `Travailleur` et `Mangeur` séparés ; `Robot` n'implémente que `Travailleur`.

À toi, en solo

Exports extensibles. Écris `exporter(donnees, format)` d'abord avec des `if/elif` pour CSV, JSON et texte brut. Puis refactore-la :

1. un `Protocol` `Exporteur` avec `exporter(donnees) -> str` ;
2. trois classes qui le satisfont ;
3. un `ServiceExport` qui reçoit les exporteurs par injection ;
4. ajoute un format `markdown` **sans modifier** le service ni les exporteurs existants ;
5. teste que l'ancien comportement est inchangé.

À toi, en solo · réponse

Pistes de correction

```
import json
from typing import Protocol

def exporter_v1(donnees, format):
    if format == "csv":
        return "\n".join(",".join(str(v) for v in ligne.values()) for ligne in don
   nees)
    if format == "json":
        return json.dumps(donnees, sort_keys=True)
    if format == "texte":
        return "\n".join(" | ".join(f"{k}={v}" for k, v in ligne.items()) for lign
   e in donnees)
    raise ValueError(format)

class Exporteur(Protocol):
    format: str
    def exporter(self, donnees) -> str: ...

class CSV:
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    format = "csv"
    def exporter(self, donnees):
        return "\n".join(",".join(str(v) for v in ligne.values()) for ligne in don
   nees)

class JSON:
    format = "json"
    def exporter(self, donnees):
        return json.dumps(donnees, sort_keys=True)

class Texte:
    format = "texte"
    def exporter(self, donnees):
        return "\n".join(" | ".join(f"{k}={v}" for k, v in ligne.items()) for lign
   e in donnees)

class ServiceExport:
    def __init__(self, exporteurs):
        self._par_format = {e.format: e for e in exporteurs}
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    def exporter(self, donnees, format):
        if format not in self._par_format:
            raise ValueError(format)
        return self._par_format[format].exporter(donnees)

donnees = [{"nom": "Ada", "score": 90}, {"nom": "Bob", "score": 45}]
service = ServiceExport([CSV(), JSON(), Texte()])
for f in ("csv", "json", "texte"):
    assert service.exporter(donnees, f) == exporter_v1(donnees, f)

class Markdown:
    format = "markdown"
    def exporter(self, donnees):
        cles = list(donnees[0])
        lignes = ["| " + " | ".join(cles) + " |", "|" + "---|" * len(cles)]
        lignes += ["| " + " | ".join(str(l[c]) for c in cles) + " |" for l in donn
   ees]
        return "\n".join(lignes)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
service._par_format["markdown"] = Markdown()
print(service.exporter(donnees, "markdown"))
```

Ajouter `Markdown` n'a nécessité aucune modification du service ni des autres exporteurs : le service est ouvert à l'extension et fermé à la modification.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Appliquer SOLID avant d'en avoir besoin	Dizaines de petites classes inutiles	Attendre le deuxième cas, puis généraliser
Héritage pour réutiliser du code	Violations de Liskov	Préférer la composition
Interfaces énormes	Classes qui implémentent du vide	Petites interfaces ciblées
Abstraction qui fuit les détails	L'interface expose le fournisseur	Concevoir l'interface selon le besoin du client

Mini-boss

Mini-boss

Système de notifications SOLID. HashBox envoie des notifications : e-mail, SMS, notification mobile. Règles : un utilisateur a des préférences de canaux ; certaines notifications sont urgentes (tous les canaux) ; chaque canal peut échouer et on doit réessayer ou basculer.

1. Écris d'abord une version naïve (une fonction avec `if/elif`).
2. Refactore en respectant S, O, L, I, D : interface `Canal`, implémentations, `PolitiqueEnvoi` (choix des canaux), `Service` qui reçoit tout par injection.
3. Ajoute un canal « Slack » sans modifier le service.
4. Écris les tests avec de faux canaux (succès, échec, échec puis succès).
5. Explique en 6 lignes quel principe justifie chaque choix.

Mini-boss · réponse

Pistes de correction

Un bon travail : `Canal(Protocol)` avec `envoyer(utilisateur, message) -> bool` ; `PolitiqueEnvoi.canaux_pour(utilisateur, urgent)` (responsabilité unique : choisir) ; `Service.notifier` parcourt les canaux, gère le réessai et le basculement (responsabilité : orchestration) ; les canaux concrets sont injectés. Les tests utilisent `FauxCanal(reussit=True/False/[False, True])` et vérifient qu'une urgence passe par tous les canaux, qu'un échec bascule sur le suivant et que `Slack` s'ajoute par une simple instance. L'explication associe S à `PolitiqueEnvoi`, O à l'ajout de Slack, L à la substituabilité des canaux, I à la petite interface `Canal`, D à l'injection.

Quiz éclair

Quiz éclair

1. Le principe ouvert/fermé dit :

A. ouvert à la modification, fermé à l'extension
B. ouvert à l'extension, fermé à la modification
C. tout est ouvert

2. Liskov est violé quand :

A. une classe a deux méthodes
B. on utilise des interfaces
C. un sous-type ne peut pas remplacer son type de base sans surprise

3. Inverser les dépendances, c'est :

A. dépendre d'abstractions plutôt que de détails

Quiz éclair

Quiz éclair (suite)

B. supprimer les dépendances
C. tout mettre dans une classe

4. 🔁 *Révision mission 02* : Une forte cohésion signifie :

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

5. 🔁 *Révision mission 02* : L'injection de dépendances consiste à :

A. créer les dépendances dans la classe
B. supprimer les dépendances
C. fournir les dépendances de l'extérieur

Quiz éclair · réponse

Réponses

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

Je réfléchis

Je réfléchis

1. Quel principe SOLID te semble le plus naturel, et lequel le plus difficile à appliquer ?
2. Dans quel cas appliquer SOLID serait-il de la sur-ingénierie ?

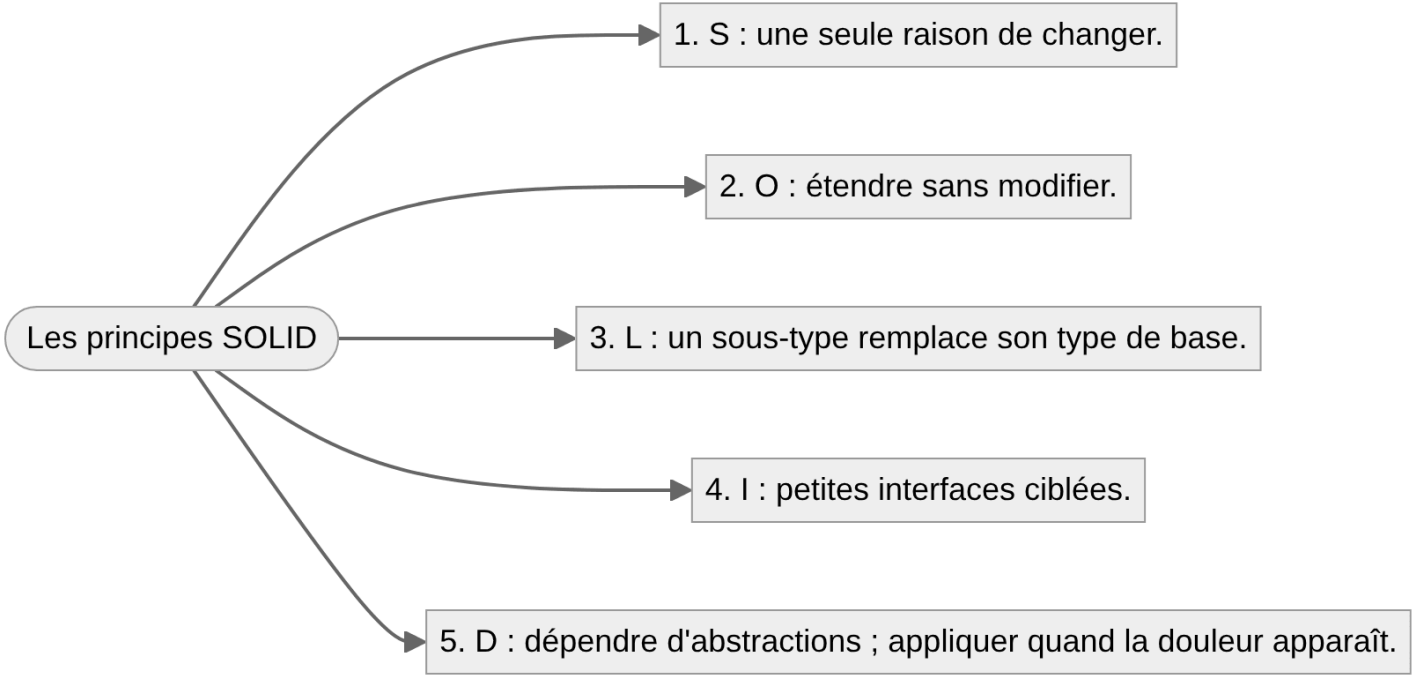
À retenir

À retenir

1. S : une seule raison de changer.
2. O : étendre sans modifier.
3. L : un sous-type remplace son type de base.
4. I : petites interfaces ciblées.
5. D : dépendre d'abstractions ; appliquer quand la douleur apparaît.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Maître des cinq principes

Prochaine mission : Clean Code : écrire pour être lu

