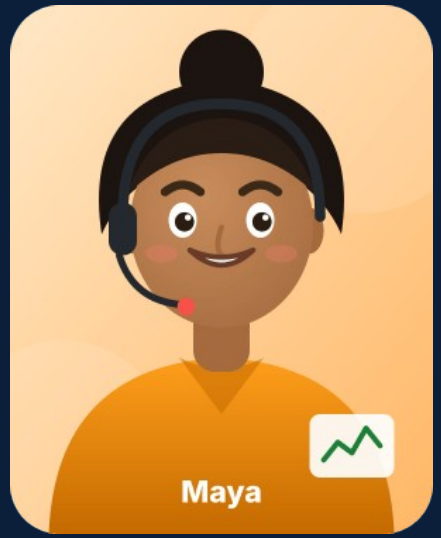
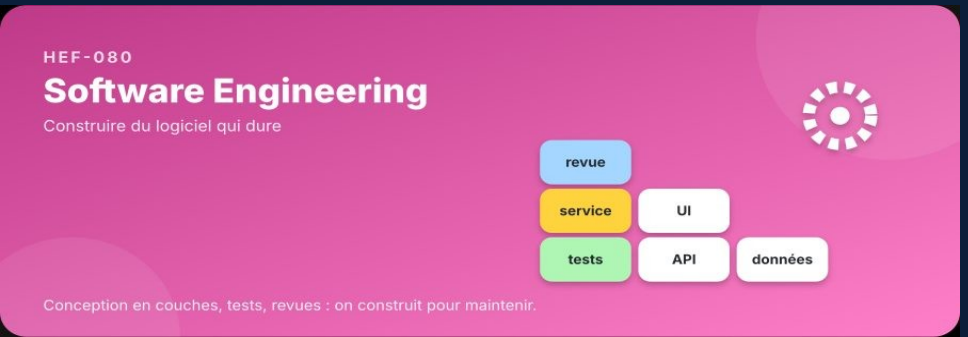
HEF-080 · MISSION 06 / 10

Les patrons de conception

Des solutions éprouvées à des problèmes qui reviennent sans cesse. Un vocabulaire commun entre ingénieurs.



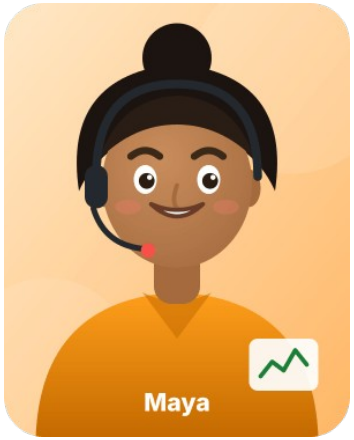


Badge : Collectionneur de patrons

90 min · intermédiaire

L'histoire

En revue de code, Kodi entend : « Ici, on pourrait utiliser une *Strategy*. Là, un *Observer*. Et ce `if type(x) == ...`, c'est un *Factory* qui manque. » Il ne comprend pas la moitié des mots.



> *« Les patrons ne sont pas des recettes à copier, dit Sam. Ce sont des noms pour des solutions que tout le monde finit par réinventer. Les connaître évite de les réinventer mal. »*

Tu apprends six patrons de conception, leur intention, leur code, et surtout quand les utiliser.

Ta mission

À la fin, je serai capable de…

1. expliquer ce qu'est un patron de conception et à quoi il sert
2. implémenter Strategy, Observer, Factory, Decorator, Adapter et Singleton (et ses limites)
3. reconnaître le problème que chaque patron résout
4. choisir entre un patron et une solution plus simple (fonction, dictionnaire)
5. éviter la sur-utilisation des patrons

Défi d'échauffement

Un restaurant propose plusieurs façons de livrer (retrait, vélo, camion). Le gérant veut changer de méthode **selon la commande** sans réécrire la caisse. Un autre restaurant veut que le chef, le serveur et le comptable soient **prévenus automatiquement** quand une commande est prête. Comment organiserais-tu le code dans chaque cas ? Quel point commun vois-tu avec « brancher des modules interchangeables » ?

Le concept

Un patron de conception, c'est quoi ?

Un **patron de conception** (*design pattern*) est une **solution générale et nommée** à un problème de conception récurrent. Il décrit : le **problème**, la **structure** de la solution, ses **conséquences**. On les classe en trois familles :

Famille	But	Exemples
Création	créer des objets de façon flexible	Factory, Singleton, Builder
Structure	assembler des objets	Adapter, Decorator, Facade
Comportement	répartir des responsabilités	Strategy, Observer, Command

Le concept

Strategy : changer d'algorithme

Encapsuler une famille d'algorithmes interchangeables.

Problème : plusieurs façons de faire la même chose (calcul de frais de livraison, tri, remise).

En Python, une **fonction** passée en paramètre suffit souvent : pas besoin d'une classe par stratégie.

```
def frais_retrait(poids): return 0
def frais_velo(poids): return 3 + poids * 0.5
def frais_camion(poids): return 10 + poids * 0.2

def total(prix, poids, strategie_livraison):
    return prix + strategie_livraison(poids)
```

Le concept

Observer : prévenir les intéressés

> *Quand un objet change, tous ses observateurs sont prévenus, sans qu'il les connaisse.*

Problème : plusieurs parties du système doivent réagir à un événement (nouvelle inscription : mail, statistiques, badge).

Factory : centraliser la création

> *Une fonction ou classe qui décide **quel objet créer**.*

Problème : le code appelant ne doit pas connaître les classes concrètes (`creer_exporteur("csv")`).

Le concept

Decorator : ajouter un comportement

> *Envelopper un objet pour lui ajouter une fonction, avec la **même interface**.*

Problème : ajouter des journaux, du cache, de la compression sans modifier la classe d'origine. (Python propose aussi la syntaxe `@decorateur` pour les fonctions.)

Adapter : faire coopérer des interfaces incompatibles

> *Traduire l'interface d'une classe vers celle que le client attend.*

Problème : une bibliothèque externe n'a pas la forme de ton interface.

Le concept

Singleton : une seule instance (à utiliser avec prudence)

Garantir qu'une classe n'a qu'une instance.

C'est un **état global déguisé** : il couple tout le code qui l'utilise et complique les tests. En Python, un simple **module** (importé une seule fois) joue déjà ce rôle. On préfère **créer l'objet une fois** et le **passer** aux endroits qui en ont besoin (injection).

Quand utiliser un patron ?

Signal	Patron probable
`if/elif` sur un type ou un mode, qui grossit	Strategy ou Factory
plusieurs réactions à un même événement	Observer

Le concept

Quand utiliser un patron ? (suite)

Signal	Patron probable
besoin d'ajouter journalisation/cache/réessai autour d'un objet	Decorator
bibliothèque dont l'interface ne convient pas	Adapter
création complexe ou conditionnelle	Factory / Builder

🧠 ***Astuce*** *: ne choisis pas un patron puis cherche un problème. Reconnais un problème,* ***puis*** *vois si un patron le résout simplement. Le meilleur patron est parfois… une fonction.*

Exemple pas à pas

Problème : un système d'inscription qui doit calculer des frais, prévenir plusieurs services, créer des exporteurs, tracer ses appels et utiliser une bibliothèque externe.

```
# --- Strategy : le calcul de frais est une fonction interchangeable ---
def frais_standard(montant): return montant * 0.02
def frais_etudiant(montant): return 0.0
def frais_premium(montant): return montant * 0.01 + 1

def total_inscription(montant, strategie_frais):
    return round(montant + strategie_frais(montant), 2)

assert total_inscription(100, frais_standard) == 102.0
assert total_inscription(100, frais_etudiant) == 100.0

# --- Observer : les services s'abonnent à un événement ---
class Evenement:
    def __init__(self):
        self._abonnes = []
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    def abonner(self, fonction):
        self._abonnes.append(fonction)
    def declencher(self, *args):
        for f in self._abonnes:
            f(*args)

inscription_faite = Evenement()
journal = []
inscription_faite.abonner(lambda nom: journal.append(f"mail de bienvenue à {nom}")
   )
inscription_faite.abonner(lambda nom: journal.append(f"statistiques : +1 ({nom})")
   )
inscription_faite.declencher("Ada")
assert len(journal) == 2

# --- Factory : un point unique de création ---
class ExporteurCSV:
    def exporter(self, donnees): return "\n".join(",".join(map(str, l)) for l in d
   onnees)
```

Exemple pas à pas (suite)

```
class ExporteurTexte:
    def exporter(self, donnees): return "\n".join(" ".join(map(str, l)) for l in d
   onnees)

def creer_exporteur(format):
    fabriques = {"csv": ExporteurCSV, "texte": ExporteurTexte}
    if format not in fabriques:
        raise ValueError(f"format inconnu : {format}")
    return fabriques[format]()

assert creer_exporteur("csv").exporter([(1, 2)]) == "1,2"

# --- Decorator : tracer les appels sans modifier la fonction ---
import functools

def tracer(fonction):
    @functools.wraps(fonction)
    def enveloppe(*args, **kwargs):
        resultat = fonction(*args, **kwargs)
```

Exemple pas à pas (suite)

```
        journal.append(f"{fonction.__name__}{args} -> {resultat}")
        return resultat
    return enveloppe

@tracer
def additionner(a, b):
    return a + b

additionner(2, 3)
assert journal[-1] == "additionner(2, 3) -> 5"

# --- Adapter : on fait ressembler une bibliothèque externe à notre interface ---
class ServiceMeteoExterne:                      # interface imposée, qu'on ne peu
   t pas changer
    def get_temp_fahrenheit(self, ville): return 68.0

class AdaptateurMeteo:                          # notre interface : temperature_c
   elsius(ville)
    def __init__(self, service): self.service = service
```

Exemple pas à pas (suite)

```
    def temperature_celsius(self, ville):
        return round((self.service.get_temp_fahrenheit(ville) - 32) * 5 / 9, 1)

assert AdaptateurMeteo(ServiceMeteoExterne()).temperature_celsius("Lyon") == 20.0
print("six patrons en action")
```

Lecture :

- **Strategy** : ajouter une règle de frais = ajouter une fonction, sans toucher à `total_inscription` (ouvert/fermé).
- **Observer** : l'événement ne connaît pas ses abonnés ; en ajouter un ne modifie pas le code d'inscription.
- **Factory** : le code appelant ne connaît pas les classes concrètes.
- **Decorator** : la traçabilité s'ajoute autour de la fonction sans la modifier.

Exemple pas à pas (suite)

- **Adapter** : le reste du code reste écrit dans notre vocabulaire (°C), et la bibliothèque (°F) est confinée à l'adaptateur.

À toi, avec aide

Pour chaque situation, nomme le patron le plus adapté et justifie en une phrase :

1. Le code contient `if canal == "mail": ... elif canal == "sms": ... elif canal == "push": ...` à six endroits.
2. Tu veux enregistrer le temps d'exécution de dix fonctions sans les modifier.
3. Un module de paiement externe expose `make_payment(cents, cur)` mais ton application attend `payer(montant)`.
4. Quand un membre termine une mission, il faut mettre à jour son score, lui donner un badge et envoyer un e-mail, sans que `terminer_mission` connaisse ces trois services.
5. Tu as besoin d'un objet de configuration partagé par toute l'application.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Pour 1, les branches représentent des variantes d'un même comportement.

Indice 2

Pour 2, on enveloppe sans modifier.

Indice 3

Pour 5, réfléchis à l'alternative à l'état global : créer une fois et passer.

Réponse

1. **Strategy** (ou **Factory** si le problème est de créer l'objet) : une table `canal → fonction d'envoi` remplace les `if`.
2. **Decorator** : un `@chronometrer` appliqué aux dix fonctions.
3. **Adapter** : une classe qui traduit `payer(montant)` en `make_payment(cents, cur)`.
4. **Observer** : `terminer_mission` déclenche un événement ; les trois services s'y abonnent.
5. Un **Singleton** est tentant, mais on préfère **créer la configuration une fois au démarrage et l'injecter** (ou un module dédié) : plus facile à tester.

À toi, en solo

Boutique extensible. Construis un petit module avec :

1. **Strategy** : trois façons de calculer une remise (aucune, pourcentage, palier : 5 € dès 50 €, 15 € dès 100 €).
2. **Observer** : une commande validée déclenche un événement auquel s'abonnent : la gestion du stock, les statistiques de vente, l'envoi d'un reçu.
3. **Decorator** : un décorateur `@reessayer(n)` qui rejoue une fonction jusqu'à `n` fois en cas d'exception.
4. **Factory** : `creer_remise("palier")` renvoie la bonne stratégie, erreur claire si inconnue.

Écris des tests pour chaque partie, y compris le cas où un observateur échoue (les autres doivent quand même être prévenus).

À toi, en solo · réponse

Pistes de correction

```
import functools

def sans_remise(montant): return 0.0
def pourcentage(taux):
    return lambda montant: round(montant * taux, 2)
def palier(montant):
    return 15.0 if montant >= 100 else 5.0 if montant >= 50 else 0.0

def creer_remise(nom):
    fabriques = {"aucune": lambda: sans_remise, "pourcentage": lambda: pourcentage
   (0.10), "palier": lambda: palier}
    if nom not in fabriques:
        raise ValueError(f"remise inconnue : {nom}")
    return fabriques[nom]()

class Evenement:
    def __init__(self):
        self.abonnes, self.erreurs = [], []
    def abonner(self, f):
```

Pistes de correction (suite)

```
        self.abonnes.append(f)
    def declencher(self, *args):
        for f in self.abonnes:
            try:
                f(*args)
            except Exception as e:                  # un observateur défaillant ne
    bloque pas les autres
                self.erreurs.append(e)

def reessayer(n):
    def decorateur(f):
        @functools.wraps(f)
        def enveloppe(*a, **k):
            for essai in range(1, n + 1):
                try:
                    return f(*a, **k)
                except Exception:
                    if essai == n:
                        raise
```

Pistes de correction (suite)

```
        return enveloppe
    return decorateur

assert creer_remise("palier")(120) == 15.0 and creer_remise("palier")(60) == 5.0 a
   nd creer_remise("palier")(10) == 0.0
assert creer_remise("pourcentage")(200) == 20.0
try:
    creer_remise("inconnue")
except ValueError:
    pass

commande_validee, stock, ventes = Evenement(), [], []
commande_validee.abonner(lambda c: stock.append(c))
commande_validee.abonner(lambda c: 1 / 0)               # observateur défaillant
commande_validee.abonner(lambda c: ventes.append(c))
commande_validee.declencher("cmd-1")
assert stock == ["cmd-1"] and ventes == ["cmd-1"] and len(commande_validee.erreurs
   ) == 1
```

À toi, en solo · réponse

Pistes de correction (suite)

```
essais = []
@reessayer(3)
def instable():
    essais.append(1)
    if len(essais) < 3:
        raise RuntimeError("panne")
    return "ok"
assert instable() == "ok" and len(essais) == 3
print("ok")
```

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Patron pour le patron	Complexité sans bénéfice	Partir du problème, pas de la solution
Singleton partout	État global, tests difficiles	Créer une fois et injecter
Observer sans gestion d'erreur	Un abonné défaillant bloque les autres	Isoler les échecs, journaliser
Oublier `functools.wraps`	Le décorateur masque le nom de la fonction	`@functools.wraps(fonction)`

Mini-boss

Moteur de règles de notification. Conçois un petit moteur en combinant au moins quatre patrons :

- **Observer** : événements (`inscription`, `paiement_echoue`, `mission_terminee`) ;
- **Strategy** : chaque règle décide si et comment notifier ;
- **Factory** : création des règles à partir d'une configuration (dictionnaire) ;
- **Decorator** : journalisation et réessai autour de l'envoi ;
- **Adapter** : brancher un faux service SMS externe dont l'interface diffère de la tienne.

Écris les tests, puis explique en un paragraphe quel patron tu as **refusé** d'utiliser et pourquoi (par exemple Singleton pour la configuration).

Mini-boss · réponse

Pistes de correction

Un bon travail : un `Bus` d'événements (Observer) auquel s'abonnent des `Regle`; chaque règle a `appliquer(evenement) -> list[Notification]` (Strategy); `creer_regle({"type": "seuil", ...})` choisit la classe (Factory); `@journaliser` et `@reessayer(3)` entourent `envoyer` (Decorator); `AdaptateurSMS(ServiceSMSExterne)` traduit `envoyer(destinataire, texte)` vers `send_sms(number, body)`. Les tests utilisent un faux canal qui échoue deux fois puis réussit, vérifient les appels journalisés et le comportement quand une règle lève une exception. Le Singleton est refusé : la configuration est lue une fois dans `main()` et passée aux constructeurs, ce qui évite l'état global et rend les tests indépendants.

Quiz éclair

Quiz éclair

1. Strategy permet de :

A. créer une seule instance
B. changer d'algorithme sans modifier le code appelant
C. traduire une interface

2. Observer sert à :

A. chiffrer des données
B. créer des objets
C. prévenir plusieurs abonnés d'un événement sans les connaître

3. Un Adapter :

A. traduit une interface vers celle qu'attend le client

Quiz éclair

Quiz éclair (suite)

B. ajoute de la sécurité
C. supprime des classes

4. 🔁 *Révision mission 05* : Une interface décrit :

A. comment c'est implémenté
B. ce qu'on peut faire, avec un contrat
C. la couleur de l'interface graphique

5. 🔁 *Révision mission 05* : Un test de contrat sert à :

A. mesurer la vitesse
B. supprimer des bugs automatiquement
C. vérifier que toutes les implémentations respectent le même comportement

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quels problèmes de ta vie professionnelle ressemblent à Observer (prévenir plusieurs personnes d'un changement) ?
2. Pourquoi le Singleton est-il souvent un piège pour les tests ?

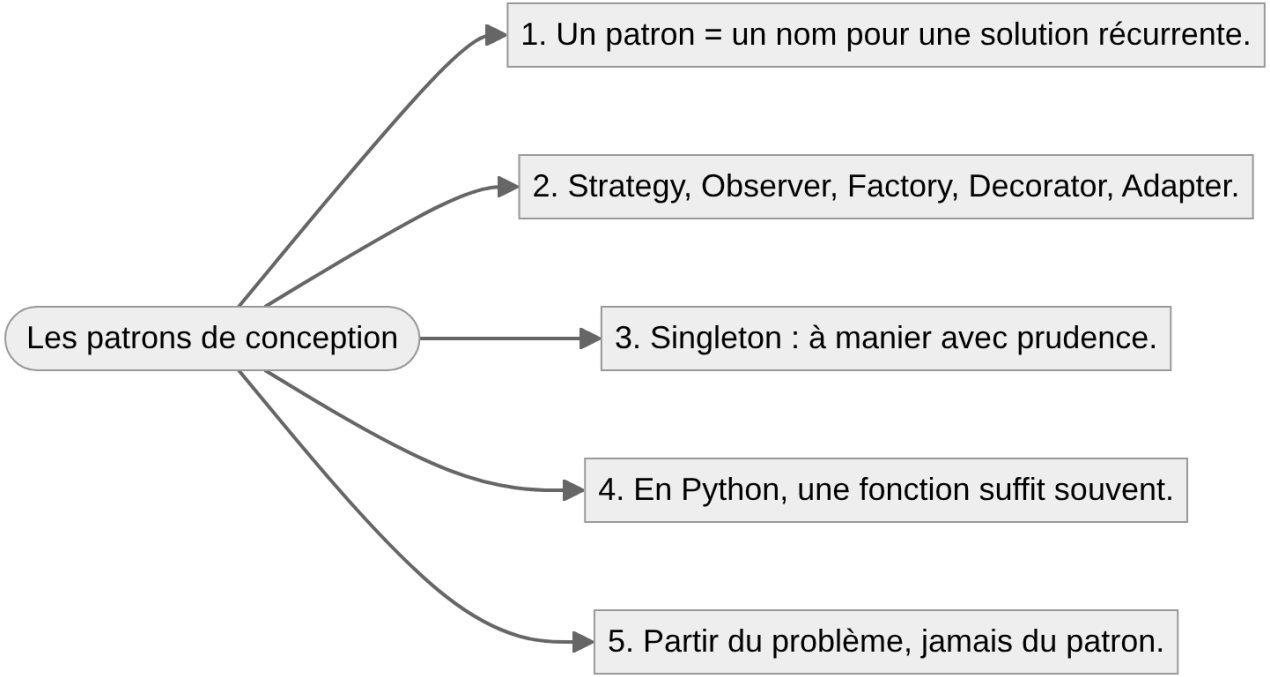
À retenir

À retenir

1. Un patron = un nom pour une solution récurrente.
2. Strategy, Observer, Factory, Decorator, Adapter.
3. Singleton : à manier avec prudence.
4. En Python, une fonction suffit souvent.
5. Partir du problème, jamais du patron.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Collectionneur de patrons

Prochaine mission : Architecture en couches et frontières

