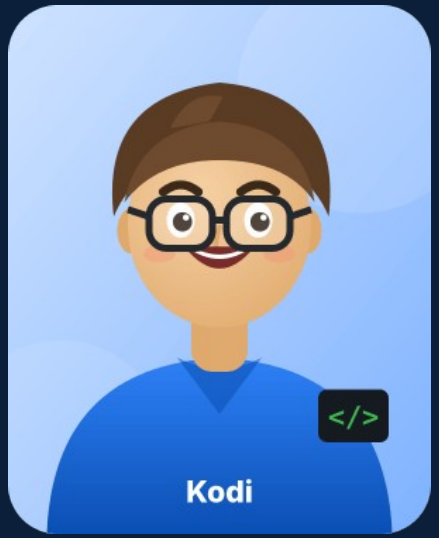
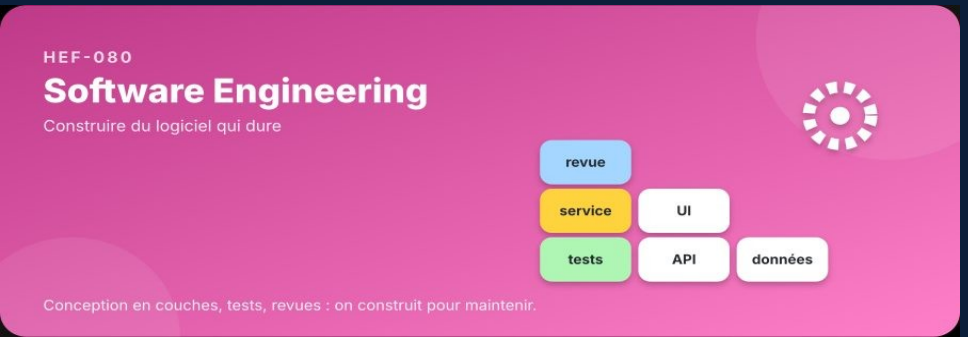
HEF-080 · MISSION 05 / 10

Interfaces et abstractions

Une bonne abstraction cache ce qui change et expose ce qui dure. Une mauvaise cache ce qu'il faudrait voir.



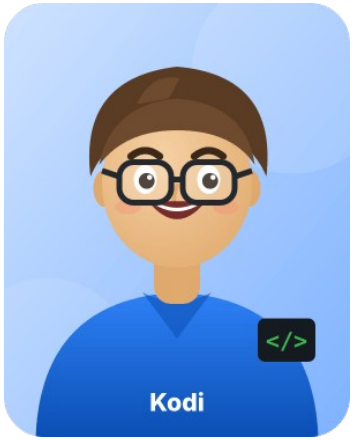


Badge : Concepteur de contrats

80 min · intermédiaire

L'histoire

HashBox veut pouvoir stocker les cours soit dans SQLite, soit dans des fichiers, soit chez un fournisseur cloud. Kodi crée une interface `Stockage` avec… 23 méthodes qui reflètent exactement l'API du fournisseur. Quand HashBox change de fournisseur, tout est à réécrire.



> *« Ton abstraction reproduit les détails au lieu de les cacher, dit Sam. Une interface doit être écrite pour les besoins du client, pas à l'image du fournisseur. »*

Tu apprends à concevoir des interfaces qui durent.

Ta mission

À la fin, je serai capable de…

1. définir interface, abstraction, implémentation et contrat
2. concevoir une interface à partir des besoins du client
3. écrire plusieurs implémentations interchangeables d'une même interface
4. utiliser une fausse implémentation (fake) pour tester
5. reconnaître les abstractions inutiles ou qui fuient (leaky abstractions)

Défi d'échauffement

Défi d'échauffement

Une prise électrique : n'importe quel appareil se branche, quelle que soit la centrale électrique derrière. Que doit garantir la prise (forme, tension) ? Que cache-t-elle ? Imagine maintenant une prise qui exposerait le type de centrale (nucléaire, solaire, charbon) : pourquoi ce serait un problème ?

Le concept

Abstraction, interface, implémentation

- **Abstraction** : un modèle simplifié qui garde l'essentiel et cache les détails.
- **Interface** : l'ensemble des opérations qu'un client peut utiliser, avec leur **contrat**.
- **Implémentation** : la façon concrète de réaliser ce contrat.

Le client dépend de l'**interface**. On peut changer l'implémentation sans le toucher.

Définir une interface en Python

Avec `Protocol` (typage structurel : pas d'héritage nécessaire) :

Le concept

Définir une interface en Python (suite)

```
from typing import Protocol

class DepotCours(Protocol):
    def trouver(self, code: str) -> dict | None: ...
    def ajouter(self, cours: dict) -> None: ...
    def lister(self) -> list[dict]: ...
```

Avec une classe abstraite (ABC) : on impose l'héritage et on peut vérifier que les méthodes sont définies.

```
from abc import ABC, abstractmethod

class DepotCours(ABC):
    @abstractmethod
    def trouver(self, code): ...
```

Le concept

Concevoir une bonne interface

1. **Partir du besoin du client** : de quoi le code appelant a-t-il vraiment besoin ?
2. **Petite et ciblée** (ségrégation) : quelques opérations bien choisies.
3. **Exprimée dans le vocabulaire du métier**, pas du fournisseur (`marquer_comme_payee`, pas `execute_sql_update`).
4. **Un contrat clair** : préconditions, résultat, erreurs.
5. **Stable** : l'interface change moins vite que les implémentations.

Plusieurs implémentations

Une même interface, des réalisations différentes : mémoire (tests), fichier, base SQL, service distant. Le code client ne sait pas laquelle il utilise.

Le concept

Les doublures de test

Doublure	Rôle
Fake	implémentation simplifiée mais fonctionnelle (dépôt en mémoire)
Stub	renvoie des réponses prédéfinies
Mock	vérifie qu'on l'a appelé comme prévu
Spy	enregistre les appels pour les vérifier après

Les **fakes** donnent souvent les tests les plus lisibles.

Quand NE PAS abstraire

Le concept

Quand NE PAS abstraire (suite)

- **Règle de trois** : on abstrait au troisième cas semblable, pas au premier. Une abstraction prématurée devine mal les besoins futurs.
- Une interface avec **une seule implémentation** qui ne changera jamais est souvent du bruit.
- Une abstraction qui **ne cache rien** (un simple renommage de l'API du fournisseur) ajoute une couche sans valeur.

Les abstractions qui fuient

La loi des abstractions qui fuient : **toute abstraction non triviale laisse transpirer ses détails**. Exemples : un dépôt « générique » dont les requêtes deviennent lentes selon le fournisseur ; une interface « fichier » qui masque la latence réseau. Mieux vaut une abstraction **honnête** sur ses limites (délais d'attente, erreurs possibles) qu'une abstraction qui promet l'impossible.

Exemple pas à pas

Problème : un dépôt de cours avec deux implémentations interchangeables et un service qui ne sait rien du stockage.

```
from typing import Protocol
import sqlite3

class DepotCours(Protocol):
    def trouver(self, code: str) -> dict | None: ...
    def ajouter(self, cours: dict) -> None: ...
    def lister(self) -> list[dict]: ...

class DepotMemoire:
    """Fake : parfait pour les tests."""
    def __init__(self):
        self._cours = {}
    def trouver(self, code):
        return self._cours.get(code)
    def ajouter(self, cours):
```

Exemple pas à pas

Exemple pas à pas (suite)

```
        if cours["code"] in self._cours:
            raise ValueError("code déjà utilisé")
        self._cours[cours["code"]] = dict(cours)
    def lister(self):
        return sorted(self._cours.values(), key=lambda c: c["code"])

class DepotSQLite:
    def __init__(self):
        self._bd = sqlite3.connect(":memory:")
        self._bd.execute("CREATE TABLE cours (code TEXT PRIMARY KEY, titre TEXT NO
   T NULL)")
    def trouver(self, code):
        ligne = self._bd.execute("SELECT code, titre FROM cours WHERE code = ?", (
   code,)).fetchone()
        return {"code": ligne[0], "titre": ligne[1]} if ligne else None
    def ajouter(self, cours):
        try:
            with self._bd:
                self._bd.execute("INSERT INTO cours VALUES (?, ?)", (cours["code"]
```

Exemple pas à pas

Exemple pas à pas (suite)

```
   , cours["titre"]))
        except sqlite3.IntegrityError:
            raise ValueError("code déjà utilisé") from None
    def lister(self):
        return [{"code": c, "titre": t} for c, t in self._bd.execute("SELECT code,
    titre FROM cours ORDER BY code")]

class Catalogue:
    """Le service ne connaît que l'interface."""
    def __init__(self, depot: DepotCours):
        self.depot = depot
    def publier(self, code, titre):
        if not titre.strip():
            raise ValueError("le titre est obligatoire")
        self.depot.ajouter({"code": code, "titre": titre.strip()})
    def titre_de(self, code):
        cours = self.depot.trouver(code)
        return cours["titre"] if cours else "cours inconnu"
```

Exemple pas à pas

Exemple pas à pas (suite)

```
# Les mêmes tests s'appliquent aux deux implémentations (le contrat est le même)
for depot in (DepotMemoire(), DepotSQLite()):
    catalogue = Catalogue(depot)
    catalogue.publier("HEF-000", "Mentalité")
    catalogue.publier("HEF-010", "Algorithmes")
    assert catalogue.titre_de("HEF-010") == "Algorithmes"
    assert catalogue.titre_de("XXX") == "cours inconnu"
    try:
        catalogue.publier("HEF-000", "Doublon")
        raise AssertionError("doublon accepté")
    except ValueError:
        pass
    assert [c["code"] for c in depot.lister()] == ["HEF-000", "HEF-010"]
print("les deux implémentations respectent le même contrat")
```

Lecture :

Exemple pas à pas (suite)

- la même suite de vérifications s'applique à **toutes** les implémentations : c'est un **test de contrat** ;
- les tests rapides du service utilisent le **fake** mémoire ; un test plus lent vérifie que `DepotSQLite` respecte le contrat ;
- le contrat inclut le comportement en cas d'erreur (`ValueError` sur un doublon), pas seulement le cas heureux : c'est ce que l'on promet aux clients.

À toi, avec aide

Conçois l'interface d'un **service d'envoi de messages** pour HashBox :

1. Écris la liste des besoins du client (le code qui notifie les membres) en une phrase.
2. Définis un `Protocol` `Messagerie` minimal.
3. Écris deux implémentations : `MessagerieConsole` (affiche) et `MessagerieEnregistrement` (garde les messages en mémoire, pour les tests).
4. Écris un `ServiceBienvenue` qui reçoit une `Messagerie` et teste-le avec l'enregistrement.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Le besoin : « envoyer un texte à une adresse ».

Indice 2

Une seule méthode suffit : `envoyer(destinataire, texte)`.

Indice 3

L'enregistrement garde une liste de tuples `(destinataire, texte)`.

À toi, avec aide · réponse

Réponse

```
from typing import Protocol

class Messagerie(Protocol):
    def envoyer(self, destinataire: str, texte: str) -> None: ...

class MessagerieConsole:
    def envoyer(self, destinataire, texte):
        print(f"[à {destinataire}] {texte}")

class MessagerieEnregistrement:
    def __init__(self):
        self.messages = []
    def envoyer(self, destinataire, texte):
        self.messages.append((destinataire, texte))

class ServiceBienvenue:
    def __init__(self, messagerie: Messagerie):
        self.messagerie = messagerie
    def accueillir(self, nom, email):
```

À toi, avec aide · réponse

Réponse (suite)

```
        self.messagerie.envoyer(email, f"Bienvenue {nom} sur HashBox !")

m = MessagerieEnregistrement()
ServiceBienvenue(m).accueillir("Ada", "ada@x.fr")
assert m.messages == [("ada@x.fr", "Bienvenue Ada sur HashBox !")]
ServiceBienvenue(MessagerieConsole()).accueillir("Bob", "bob@x.fr")
```

Une interface d'une seule méthode, dans le vocabulaire du client, suffit : elle n'expose rien du fournisseur (SMTP, SMS…).

À toi, en solo

Test de contrat réutilisable. Écris une fonction `verifier_contrat_depot(creer_depot)` qui reçoit une *fabrique* de dépôts et vérifie le contrat suivant (pour une interface `DepotMembres` avec `ajouter(nom, email)`, `par_email(email)`, `supprimer(email)`, `compter()`) :

- ajouter puis retrouver par e-mail ;
- ajouter un e-mail en double lève `ValueError` ;
- supprimer un e-mail inexistant lève `KeyError` ;
- `compter` reflète les ajouts et suppressions ;
- la recherche par e-mail est **insensible à la casse**.

Implémente deux dépôts (mémoire et SQLite) et vérifie que les deux passent. Introduis volontairement un bug dans l'un et montre que le test de contrat l'attrape.

À toi, en solo · réponse

Pistes de correction

```
import sqlite3

class DepotMemoire:
    def __init__(self):
        self._d = {}
    def ajouter(self, nom, email):
        cle = email.lower()
        if cle in self._d:
            raise ValueError("e-mail déjà utilisé")
        self._d[cle] = nom
    def par_email(self, email):
        nom = self._d.get(email.lower())
        return None if nom is None else {"nom": nom, "email": email.lower()}
    def supprimer(self, email):
        del self._d[email.lower()]
    def compter(self):
        return len(self._d)

class DepotSQLite:
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    def __init__(self):
        self._bd = sqlite3.connect(":memory:")
        self._bd.execute("CREATE TABLE m (email TEXT PRIMARY KEY, nom TEXT NOT NUL
    L)")
    def ajouter(self, nom, email):
        try:
            with self._bd:
                self._bd.execute("INSERT INTO m VALUES (?, ?)", (email.lower(), no
    m))
        except sqlite3.IntegrityError:
            raise ValueError("e-mail déjà utilisé") from None
    def par_email(self, email):
        r = self._bd.execute("SELECT nom, email FROM m WHERE email = ?", (email.lo
    wer(),)).fetchone()
        return None if r is None else {"nom": r[0], "email": r[1]}
    def supprimer(self, email):
        with self._bd:
            if self._bd.execute("DELETE FROM m WHERE email = ?", (email.lower(),))
    .rowcount == 0:
```

À toi, en solo · réponse

Pistes de correction (suite)

```
                raise KeyError(email)
    def compter(self):
        return self._bd.execute("SELECT COUNT(*) FROM m").fetchone()[0]

def verifier_contrat_depot(creer_depot):
    d = creer_depot()
    d.ajouter("Ada", "Ada@X.fr")
    assert d.par_email("ada@x.fr")["nom"] == "Ada"
    assert d.compter() == 1
    try:
        d.ajouter("Autre", "ADA@x.fr"); raise AssertionError("doublon accepté")
    except ValueError:
        pass
    try:
        d.supprimer("inconnu@x.fr"); raise AssertionError("suppression inexistante
    acceptée")
    except KeyError:
        pass
    d.supprimer("ada@X.FR")
```

À toi, en solo · réponse

Pistes de correction (suite)

```
    assert d.compter() == 0 and d.par_email("ada@x.fr") is None

for fabrique in (DepotMemoire, DepotSQLite):
    verifier_contrat_depot(fabrique)
print("les deux dépôts respectent le contrat")

class DepotBogue(DepotMemoire):
    def ajouter(self, nom, email):            # oublie la normalisation de la cass
   e
        if email in self._d:
            raise ValueError
        self._d[email] = nom
try:
    verifier_contrat_depot(DepotBogue)
except (AssertionError, KeyError, ValueError, TypeError):
    print("bug détecté par le test de contrat")
```

Un test de contrat s'écrit une fois et s'applique à toute implémentation présente et future.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Interface calquée sur le fournisseur	Changer de fournisseur = tout réécrire	Interface dans le vocabulaire du métier
Abstraire trop tôt	Interface devinée, mal ajustée	Règle de trois
Interface trop large	Implémentations pénibles	Petites interfaces ciblées
Contrat non testé	Les implémentations divergent	Test de contrat commun

Mini-boss

Mini-boss

Architecture de stockage de fichiers. HashBox stocke des supports de cours. Besoins du client : `enregistrer(nom, contenu) -> identifiant`, `lire(identifiant) -> contenu`, `supprimer(identifiant)`, `existe(identifiant)`.

1. Définis l'interface `StockageFichiers`.
2. Implémente trois versions : mémoire (fake), dossier local (fichiers sur disque dans un dossier temporaire) et une version qui **ajoute de la compression** autour d'un autre stockage (décorateur).
3. Écris un test de contrat commun, y compris pour les erreurs (lire un identifiant inconnu).
4. Écris un `ServiceSupports` qui dépend uniquement de l'interface et teste-le avec le fake.
5. Explique où l'abstraction **fuit** (latence, taille maximale, atomicité) et comment tu le documentes dans le contrat.

Mini-boss · réponse

Pistes de correction

Un bon travail : `Protocol` à quatre méthodes ; `StockageMemoire` (dictionnaire) ; `StockageDossier` (identifiant = hachage SHA-256 du contenu ou compteur, fichiers dans un `TemporaryDirectory`) ; `StockageCompresse(stockage)` qui compresse avec `zlib` à l'écriture et décompresse à la lecture, tout en respectant le même contrat ; `FichierInconnu(KeyError)` pour les lectures invalides ; test de contrat paramétré. Les fuites documentées : la lecture disque peut être lente et lever des `OSError`, la taille maximale diffère, l'écriture n'est pas atomique sans fichier temporaire + renommage.

Quiz éclair

Quiz éclair

1. Une interface décrit :

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

2. Une doublure « fake » est :

A. une implémentation simplifiée mais fonctionnelle
B. un mot de passe
C. un type d'index

3. Un test de contrat sert à :

A. mesurer la vitesse

Quiz éclair

Quiz éclair (suite)

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

4. 🔁 *Révision mission 04* : Un bon nom de fonction est généralement :

A. une lettre
B. un nombre
C. un verbe qui exprime l'intention

5. 🔁 *Révision mission 04* : Les retours anticipés servent à :

A. réduire l'imbrication
B. accélérer le code
C. supprimer les tests

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quelle « prise » (interface) stable utilises-tu au quotidien sans te soucier de ce qu'il y a derrière ?
2. Quand as-tu abstrait trop tôt et regretté ?

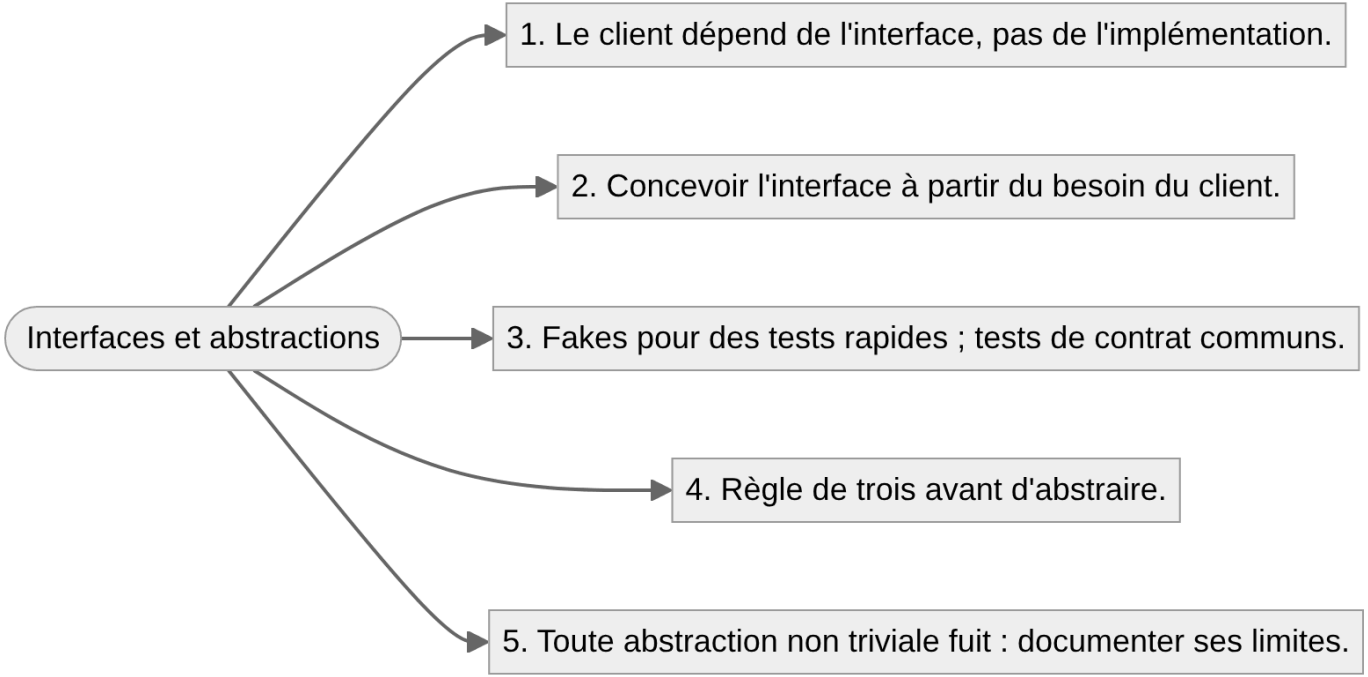
À retenir

À retenir

1. Le client dépend de l'interface, pas de l'implémentation.
2. Concevoir l'interface à partir du besoin du client.
3. Fakes pour des tests rapides ; tests de contrat communs.
4. Règle de trois avant d'abstraire.
5. Toute abstraction non triviale fuit : documenter ses limites.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Concepteur de contrats

Prochaine mission : Les patrons de conception

