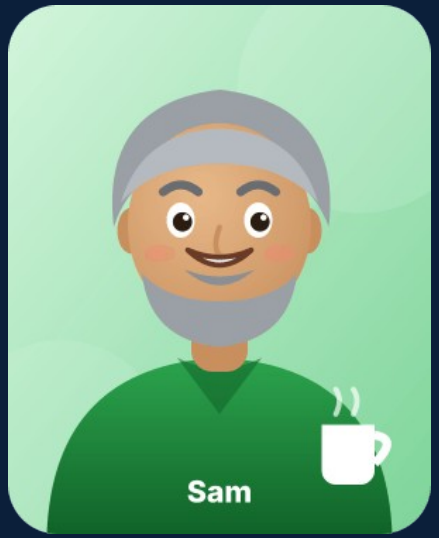
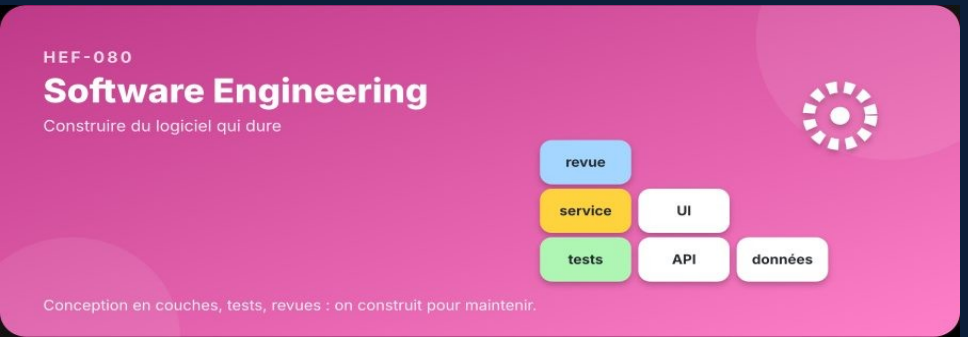
HEF-080 · MISSION 07 / 10

Architecture en couches et frontières

Où va chaque morceau de code ? L'architecture répond : cœur au centre, détails aux bords.





Badge : Urbaniste logiciel

95 min · intermédiaire

L'histoire

L'application HashBox grandit : pages web, règles de calcul, accès à la base, envoi de mails. Le code se mélange : une page web contient du SQL, une règle métier appelle directement le serveur de mail, et changer de base de données nécessite de modifier cinquante fichiers.

> *« Il faut décider où vivent les choses, dit Sam, et dans quel sens vont les dépendances. Les règles métier doivent être au centre et ne dépendre de rien. »*



Tu apprends les architectures en couches, MVC, hexagonale et les bases de DDD.

Ta mission

À la fin, je serai capable de…

1. décrire l'architecture en couches et sa règle de dépendance
2. expliquer MVC et son rôle
3. expliquer l'architecture hexagonale (ports et adaptateurs)
4. utiliser entités, objets-valeurs et services (bases de DDD)
5. construire une petite application dont le cœur ne dépend d'aucun détail technique

Défi d'échauffement

Une ville est organisée : le centre historique (règles qu'on ne change pas à la légère), les quartiers (usages), les routes et réseaux (électricité, eau) qui peuvent être refaits sans reconstruire les maisons. Si on doit changer de fournisseur d'électricité, veut-on casser les maisons ? Quelles sont les « maisons » et les « réseaux » dans une application ?

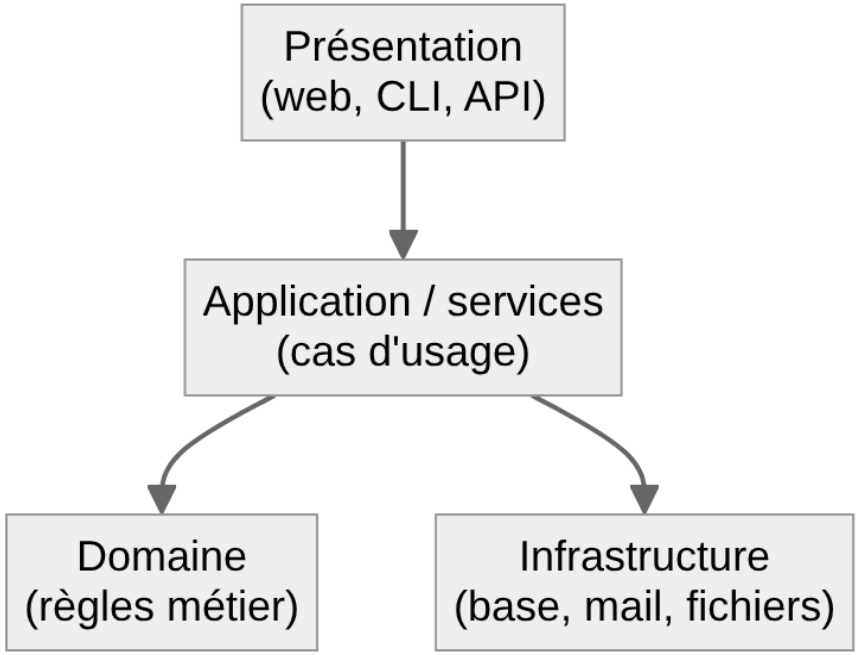
Le concept

Architecture en couches

On organise le code en **couches** empilées, chacune ne dépendant que de celle d'en dessous :

Le concept

Architecture en couches (suite)



Le concept

Architecture en couches (suite)

Description du schéma : la présentation appelle la couche application (cas d'usage), qui utilise le domaine (règles métier) et l'infrastructure (base de données, mail, fichiers).

Avantage : séparer les préoccupations. Limite : si le domaine dépend de l'infrastructure, les règles métier sont liées à la base de données.

MVC

Modèle – Vue – Contrôleur sépare une interface en trois rôles :

Rôle	Responsabilité
Modèle	données et règles métier
Vue	affichage
Contrôleur	reçoit les actions de l'utilisateur, coordonne modèle et vue

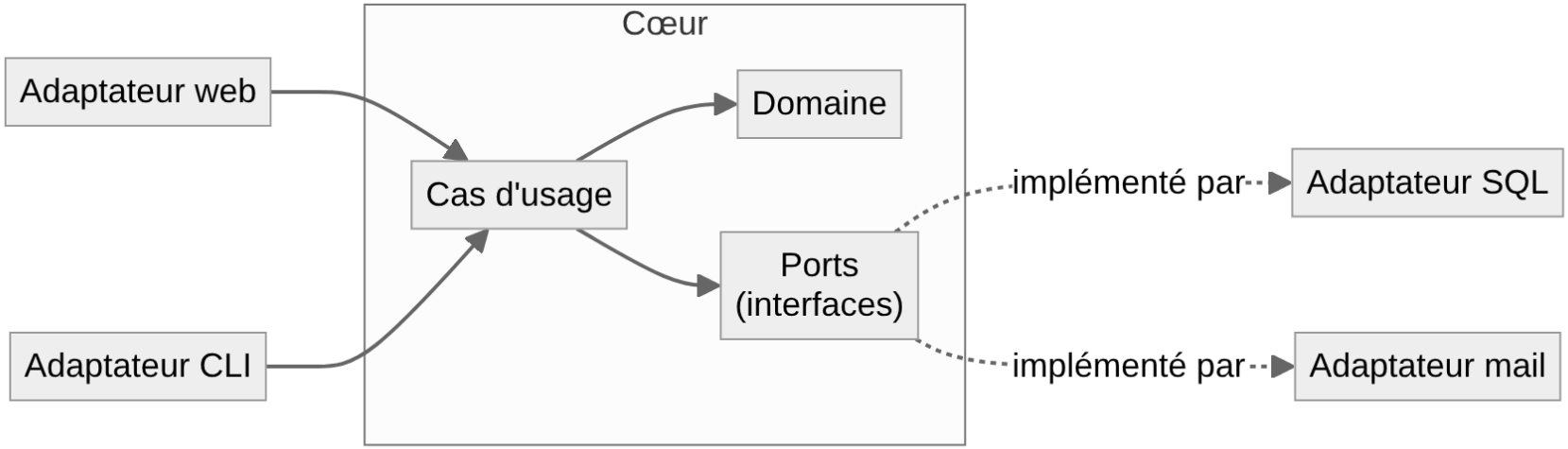
Le concept

L'architecture hexagonale (ports et adaptateurs)

On **inverse** la dépendance : le **domaine** est au centre et ne connaît **que des interfaces** (les **ports**). Les détails techniques (**adaptateurs**) implémentent ces ports.

Le concept

L'architecture hexagonale (ports et adaptateurs) (suite)



Description du schéma : à gauche, des adaptateurs d'entrée (web, ligne de commande) appellent les cas d'usage du cœur. Le cœur contient le domaine et des ports (interfaces). À droite, les adaptateurs de sortie (SQL, mail) implémentent ces ports.

Le concept

L'architecture hexagonale (ports et adaptateurs) (suite)

Règle de dépendance : les flèches de dépendance pointent **vers le centre**. Le domaine ne connaît ni SQL, ni HTTP, ni mail.

Conséquences : on teste le cœur sans infrastructure ; on change de base ou de framework en écrivant un nouvel adaptateur.

Les bases de DDD (Domain-Driven Design)

La conception pilotée par le domaine propose un vocabulaire pour modéliser le métier :

Concept	Sens	Exemple
Langage omniprésent	les mots du métier, utilisés partout (code compris)	« inscription », « mission terminée »
Entité	objet défini par son **identité**	`Membre` (id 42)

Le concept

Les bases de DDD (Domain-Driven Design) (suite)

Concept	Sens	Exemple
Objet-valeur	défini par ses **valeurs**, immuable	`Montant(10, "EUR")`, `Email`
Agrégat	groupe d'objets modifié comme un tout, via une racine	`Commande` avec ses lignes
Service de domaine	règle qui n'appartient à aucune entité	calcul de remise croisée
Dépôt (*repository*)	interface d'accès aux agrégats	`DepotMembres`
Contexte délimité	frontière dans laquelle un modèle est valide	« Facturation » vs « Catalogue »

Les frontières

Une **frontière** est l'endroit où l'on traduit entre deux modèles ou deux mondes (données HTTP → objets du domaine ; lignes SQL → entités). On **valide à la frontière**, et à l'intérieur on manipule des objets fiables.

Le concept

Choisir

Contexte	Architecture raisonnable
Script, prototype	un module, des fonctions
Petite application	couches simples
Application métier durable	hexagonale avec domaine isolé
Très grand système	contextes délimités, services séparés

🧠 ***Astuce*** *: une architecture se juge à la* ***facilité des changements probables****. Fais la plus simple qui permet de changer de base, d'interface et de règle sans tout casser.*

Exemple pas à pas

Problème : une petite application d'inscriptions en architecture hexagonale, avec deux adaptateurs de stockage et un test du cœur sans infrastructure.

```
from dataclasses import dataclass
from typing import Protocol

# ===== DOMAINE : aucune dépendance technique =====
@dataclass(frozen=True)
class Email:
    valeur: str
    def __post_init__(self):
        if "@" not in self.valeur or self.valeur.startswith("@") or self.valeur.en
   dswith("@"):
            raise ValueError(f"e-mail invalide : {self.valeur!r}")

@dataclass
class Membre:
    id: int | None
```

Exemple pas à pas (suite)

```
    nom: str
    email: Email
    cours: set

    def s_inscrire(self, code_cours, capacite_max, deja_inscrits):
        if code_cours in self.cours:
            raise ValueError("déjà inscrit")
        if deja_inscrits >= capacite_max:
            raise ValueError("cours complet")
        self.cours.add(code_cours)

# ===== PORTS : interfaces que le cœur exige =====
class DepotMembres(Protocol):
    def enregistrer(self, membre: Membre) -> Membre: ...
    def par_email(self, email: Email) -> Membre | None: ...
    def compter_inscrits(self, code_cours: str) -> int: ...

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

Exemple pas à pas (suite)

```
# ===== CAS D'USAGE (application) =====
CAPACITE_COURS = 2

class InscrireMembre:
    def __init__(self, depot: DepotMembres, messagerie: Messagerie):
        self.depot, self.messagerie = depot, messagerie

    def executer(self, nom, adresse, code_cours):
        email = Email(adresse)                                  # validation à la
   frontière
        membre = self.depot.par_email(email) or Membre(None, nom, email, set())
        membre.s_inscrire(code_cours, CAPACITE_COURS, self.depot.compter_inscrits(
   code_cours))
        membre = self.depot.enregistrer(membre)
        self.messagerie.envoyer(email, f"Bienvenue dans {code_cours}, {nom} !")
        return membre

# ===== ADAPTATEURS (infrastructure) =====
```

Exemple pas à pas

Exemple pas à pas (suite)

```
class DepotMemoire:
    def __init__(self):
        self._par_email, self._suivant = {}, 1
    def enregistrer(self, membre):
        if membre.id is None:
            membre.id, self._suivant = self._suivant, self._suivant + 1
        self._par_email[membre.email.valeur] = membre
        return membre
    def par_email(self, email):
        return self._par_email.get(email.valeur)
    def compter_inscrits(self, code):
        return sum(code in m.cours for m in self._par_email.values())

class MessagerieEnregistrement:
    def __init__(self):
        self.envoyes = []
    def envoyer(self, email, texte):
        self.envoyes.append((email.valeur, texte))
```

Exemple pas à pas

Exemple pas à pas (suite)

```
# ===== TEST du cœur : aucune base, aucun réseau =====
depot, mail = DepotMemoire(), MessagerieEnregistrement()
cas_usage = InscrireMembre(depot, mail)
cas_usage.executer("Ada", "ada@x.fr", "HEF-000")
cas_usage.executer("Bob", "bob@x.fr", "HEF-000")
assert mail.envoyes[0] == ("ada@x.fr", "Bienvenue dans HEF-000, Ada !")
for erreur, (n, a) in (("cours complet", ("Cléo", "cleo@x.fr")), ("déjà inscrit",
   ("Ada", "ada@x.fr"))):
    try:
        cas_usage.executer(n, a, "HEF-000")
        raise AssertionError("aurait dû échouer")
    except ValueError as e:
        assert erreur in str(e)
try:
    cas_usage.executer("Zoé", "pas-un-email", "HEF-010")
except ValueError as e:
    print("refusé à la frontière :", e)
print("cœur testé sans infrastructure")
```

Exemple pas à pas (suite)

Lecture :

- le **domaine** (`Email`, `Membre`) contient les règles (cours complet, déjà inscrit) et ne connaît ni base ni mail ;
- les **ports** (`DepotMembres`, `Messagerie`) sont les contrats exigés par le cœur ;
- les **adaptateurs** (`DepotMemoire` ici, un `DepotSQLite` ensuite) les implémentent ;
- `Email` est un **objet-valeur** : s'il existe, il est valide, donc plus besoin de revalider partout ;
- le cas d'usage orchestre : valider, charger, appliquer la règle, enregistrer, notifier.

À toi, avec aide

Ajoute un deuxième adaptateur de stockage à l'exemple : `DepotSQLite`, qui implémente `DepotMembres` avec une base SQLite en mémoire (tables `membres` et `inscriptions`).

1. Implémente `enregistrer`, `par_email`, `compter_inscrits`.
2. Fais passer **exactement les mêmes vérifications** (cours complet, déjà inscrit) à `InscrireMembre` avec ce nouveau dépôt.
3. Que prouve le fait de ne pas avoir touché au cœur ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

`Membre` a un `set` de cours : stocke-les dans une table `inscriptions(membre_id, cours)`.

Indice 2

`par_email` doit reconstruire un `Membre` complet avec ses cours.

Indice 3

Le cas d'usage reçoit le dépôt par son constructeur : tu n'as rien à y changer.

À toi, avec aide · réponse

Réponse

```
import sqlite3

class DepotSQLite:
    def __init__(self):
        self._bd = sqlite3.connect(":memory:")
        self._bd.executescript('''
        CREATE TABLE membres (id INTEGER PRIMARY KEY, nom TEXT NOT NULL, email TEX
   T NOT NULL UNIQUE);
        CREATE TABLE inscriptions (membre_id INTEGER NOT NULL REFERENCES membres(i
   d), cours TEXT NOT NULL, PRIMARY KEY (membre_id, cours));
        ''')

    def enregistrer(self, membre):
        with self._bd:
            if membre.id is None:
                cur = self._bd.execute("INSERT INTO membres (nom, email) VALUES (?
   , ?)", (membre.nom, membre.email.valeur))
                membre.id = cur.lastrowid
            self._bd.execute("DELETE FROM inscriptions WHERE membre_id = ?", (memb
```

À toi, avec aide · réponse

Réponse (suite)

```
    re.id,))
            self._bd.executemany("INSERT INTO inscriptions VALUES (?, ?)", [(membr
    e.id, c) for c in membre.cours])
         return membre

     def par_email(self, email):
         ligne = self._bd.execute("SELECT id, nom FROM membres WHERE email = ?", (e
    mail.valeur,)).fetchone()
         if ligne is None:
             return None
         cours = {c for (c,) in self._bd.execute("SELECT cours FROM inscriptions WH
    ERE membre_id = ?", (ligne[0],))}
         return Membre(ligne[0], ligne[1], email, cours)

     def compter_inscrits(self, code):
         return self._bd.execute("SELECT COUNT(*) FROM inscriptions WHERE cours = ?
    ", (code,)).fetchone()[0]
```

À toi, avec aide · réponse

Réponse (suite)

En réutilisant les assertions de l'exemple avec `InscrireMembre(DepotSQLite(), mail)`, les mêmes résultats sont obtenus. Cela prouve que les **règles métier sont indépendantes du stockage** : changer de base revient à écrire un adaptateur.

À toi, en solo

Ajouter une règle métier sans toucher à l'infrastructure. Nouvelle règle : un membre ne peut pas être inscrit à plus de 3 cours simultanément, sauf s'il est « premium » (nouvel attribut booléen).

1. Modifie **uniquement** le domaine (`Membre`) et les tests.
2. Vérifie que les adaptateurs (dépôt, messagerie) n'ont pas besoin d'être changés (à l'exception du stockage de l'attribut `premium` dans l'adaptateur SQL si tu veux le persister).
3. Écris les tests : 3 cours acceptés, un 4e refusé, un membre premium accepté.
4. Que montre cet exercice sur la valeur de l'architecture ?

À toi, en solo · réponse

Pistes de correction

```
# Dans Membre :
#   premium: bool = False
#   def s_inscrire(self, code_cours, capacite_max, deja_inscrits):
#       ...règles existantes...
#       if not self.premium and len(self.cours) >= 3:
#           raise ValueError("limite de 3 cours atteinte")
#       self.cours.add(code_cours)

from dataclasses import dataclass, field

@dataclass
class Membre:
    id: int | None
    nom: str
    email: str
    cours: set = field(default_factory=set)
    premium: bool = False

    def s_inscrire(self, code_cours, capacite_max, deja_inscrits):
```

À toi, en solo · réponse

Pistes de correction (suite)

```
        if code_cours in self.cours:
            raise ValueError("déjà inscrit")
        if deja_inscrits >= capacite_max:
            raise ValueError("cours complet")
        if not self.premium and len(self.cours) >= 3:
            raise ValueError("limite de 3 cours atteinte")
        self.cours.add(code_cours)

m = Membre(1, "Ada", "ada@x.fr")
for code in ("A", "B", "C"):
    m.s_inscrire(code, 10, 0)
try:
    m.s_inscrire("D", 10, 0)
    raise AssertionError("4e cours accepté")
except ValueError as e:
    assert "limite" in str(e)
p = Membre(2, "Bob", "bob@x.fr", premium=True)
for code in ("A", "B", "C", "D"):
    p.s_inscrire(code, 10, 0)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
assert len(p.cours) == 4
print("ok")
```

Le cas d'usage, les ports et les adaptateurs n'ont pas bougé : la règle vit dans le domaine, et c'est le seul endroit qui change.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
SQL dans les contrôleurs	Règles métier et accès aux données mélangés	Passer par un port (dépôt)
Un domaine qui importe le framework	Impossible à tester seul	Le domaine ne dépend de rien
Architecture trop lourde pour un script	Complexité sans bénéfice	Adapter l'architecture à la durée de vie du logiciel
Valider partout, ou nulle part	Données invalides à l'intérieur	Valider à la frontière, objets-valeurs ensuite

Mini-boss

Mini-boss

Application de réservation de salles en architecture hexagonale.

Domaine : `Salle` (capacité), `Reservation` (entité), `Creneau` (objet-valeur : début, fin, valide si fin > début), règle de non-chevauchement. Ports : `DepotReservations`, `Horloge` (pour contrôler le temps dans les tests), `Notificateur`. Cas d'usage : `ReserverSalle`, `AnnulerReservation`, `ListerReservationsDuJour`. Adaptateurs : dépôt mémoire, dépôt SQLite, notificateur console, adaptateur « ligne de commande » qui traduit des textes en appels de cas d'usage.

1. Implémente le tout.
2. Écris au moins 12 tests du cœur avec uniquement des fakes (chevauchement, annulation, capacité, créneau invalide, horloge fixée).
3. Fais passer le même test de contrat aux deux dépôts.
4. Dessine (Mermaid) les dépendances et vérifie par un petit script que le domaine n'importe ni `sqlite3` ni `sys`.

Mini-boss · réponse

Pistes de correction

Un bon travail : `Creneau` est un `@dataclass(frozen=True)` qui valide dans `__post_init__` ; `Reservation.chevauche(autre)` ; le cas d'usage `ReserverSalle` récupère les réservations de la salle via le dépôt, refuse en cas de chevauchement, enregistre et notifie ; `Horloge.maintenant()` permet de refuser les réservations passées sans dépendre de l'heure réelle ; la vérification d'architecture lit les fichiers du domaine avec `ast` et échoue si un import hors bibliothèque standard « pure » (`dataclasses`, `typing`, `datetime`) est trouvé. Le schéma Mermaid montre les flèches vers le centre.

Quiz éclair

Quiz éclair

1. Dans l'architecture hexagonale, le domaine :

A. ne dépend d'aucun détail technique
B. dépend de la base de données
C. dépend du framework web

2. Un objet-valeur est :

A. défini par son identité
B. défini par ses valeurs et immuable
C. toujours persistant

3. Un port est :

A. une connexion réseau

Quiz éclair

Quiz éclair (suite)

B. un fichier de configuration
C. une interface exigée par le cœur

4. 🔁 *Révision mission 06* : Strategy permet de :

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

5. 🔁 *Révision mission 06* : Un Adapter :

A. ajoute de la sécurité
B. traduit une interface vers celle qu'attend le client
C. supprime des classes

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quels éléments de ta vie professionnelle sont « le cœur » (qu'on protège) et lesquels sont des « détails » (qu'on peut remplacer) ?
2. Pourquoi valider à la frontière réduit-il les bugs à l'intérieur ?

À retenir

À retenir

1. Couches : présentation, application, domaine, infrastructure.
2. MVC : modèle, vue, contrôleur.
3. Hexagonale : domaine au centre, ports et adaptateurs.
4. DDD : entités, objets-valeurs, agrégats, dépôts.
5. Règle de dépendance : vers le centre.

MISSION ACCOMPLIE

+100 XP

Badge débloqué : Urbaniste logiciel

Prochaine mission : Refactoring sécurisé

