

# CM1 : Fondamentaux de l'architecture logicielle

---

BUT Informatique — Ressource R4.01 « Architecture logicielle »

Enseignant : Marc Ennaji

 Objectif du cours :




Comprendre **pourquoi** l'architecture logicielle est essentielle et maîtriser les **principes fondamentaux** qui guident toute bonne conception.

## Plan du cours

1. Pourquoi une architecture logicielle ?
2. L'architecture à l'ère des assistants de codage IA
3. **Principes fondamentaux :**
  - Cohésion
  - Couplage
  - Gestion des dépendances
  - Séparation des responsabilités
  - Inversion de dépendances
4. **Architecture hexagonale** (Ports & Adapters)
5. Présentation du projet ticketing

## **Ce que vous allez construire**


Vous allez appliquer ces principes sur un **projet fil rouge** :

-  **Système de tickets** (simplifié, type Trello/Jira)
-  **Architecture hexagonale imposée** (vous comprendrez pourquoi)
-  **20h de TD** pour maîtriser les fondamentaux

 **Ce CM vous donne les clés pour réussir le projet.**

# 1. Pourquoi parler d'architecture ?






Sans vraie architecture, on obtient vite :

- Du **code spaghetti**  (*attention aux indigestions*)
- Une application **difficile à comprendre**
- Des bugs qui reviennent en boucle
- Une application **impossible à tester**
- Une appli qui ne supporte pas bien les évolutions

👉 L'architecture sert à organiser le logiciel pour qu'il soit **vivable** sur le long terme.

## Objectifs d'une bonne architecture

Une bonne architecture doit aider à :

-  **Maintenir** : corriger, faire évoluer
-  **Modulariser** : pouvoir changer une partie sans tout casser
-  **Tester** : isoler le métier pour le tester sans tout l'environnement
-  **Faire évoluer** : ajouter des fonctionnalités sans tout réécrire
-  **Comprendre** : nouveaux développeurs qui arrivent sur le projet

*Plus l'architecture est pensée, moins on "jette et réécrit" les applis.*

## 2. L'architecture à l'ère de l'IA

« Avec Copilot, ChatGPT, Cursor... je code par "intuition" et ça marche.  
L'architecture, c'est moins important ? »

✗ **FAUX.** C'est même l'inverse.

 *Le "vibe coding" (coder à l'instinct avec l'IA) a sa place pour prototyper.  
Mais en production sans maîtrise des fondamentaux → dette technique garantie.*



# Pourquoi l'architecture devient PLUS importante

## 1. "Vibe coding" = productivité court terme, chaos moyen terme

- L'IA + votre intuition → Code qui marche *maintenant*
- Mais sans vision architecturale → Dette technique exponentielle
- *Dans 6 mois : "Qui a écrit ce code ?" — Spoiler : c'était vous + l'IA*

## 2. L'IA ne conçoit pas de systèmes

- Elle respecte une architecture *si vous lui expliquez laquelle*
- Elle amplifie vos décisions (bonnes **ou** mauvaises)

## 3. Le "vibe" ne scale pas

- 100 lignes → intuition OK | 10 000 lignes → structure nécessaire | 100 000 lignes → principes indispensables



## À retenir !



L'IA code très bien. Aucune IA n'est ingénieure logicielle.

**Votre valeur** = comprendre le système, pas juste générer du code.

**Usages légitimes du "vibe coding" :**

-  Prototypage rapide / POC
-  Scripts one-shot

**Mais en production sans fondamentaux = illusion de compétence :**

- Ça marche maintenant → mais ça ne scale pas → personne ne comprend dans 3 mois



**Ce cours vous donne les fondamentaux** pour concevoir des systèmes cohérents que l'IA pourra ensuite vous aider à implémenter.



### 3. Principes fondamentaux

Ces principes sont **universels** — ils s'appliquent quelle que soit l'architecture choisie.

Les maîtriser, c'est pouvoir :

- Évaluer la qualité d'un code existant
- Guider une IA efficacement
- Faire les bons choix de conception

## 3.1 La cohésion

Ce qui va ensemble doit rester ensemble.

Une classe, un module, un service doit avoir une **responsabilité claire et focalisée**.

✓ **Forte cohésion** (bien) :

```
class ShoppingCart:
    def add_item(self, item): ...
    def remove_item(self, item): ...
    def calculate_total(self): ...
    def apply_discount(self, code): ...
```

## 3.1 La cohésion

### ✗ Faible cohésion (problème) :

```
class ShoppingCart:
    def add_item(self, item): ...
    def send_email(self, to, subject): ... # ✗ Rien à voir !
    def generate_pdf_report(self): ...     # ✗ Pas sa responsabilité
```

## 3.1 La cohésion — pourquoi c'est important ?

**Faible cohésion = problèmes garantis :**

- 🐛 Modifications à un endroit cassent des choses sans rapport
- 🧪 Tests difficiles : il faut mocker des choses non liées
- 🤖 Code difficile à comprendre : "cette classe fait quoi exactement ?"
- 🔄 Réutilisation impossible : tout est mélangé

**Forte cohésion = bénéfices :**

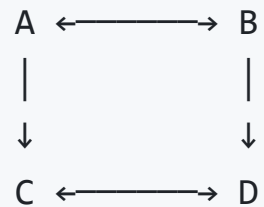
- ✅ Code auto-documenté par sa structure
- ✅ Tests ciblés et simples
- ✅ Évolutions localisées

## 3.2 Le couplage

Moins les modules dépendent les uns des autres, mieux c'est.

Le couplage mesure à quel point un module est **lié** à d'autres.

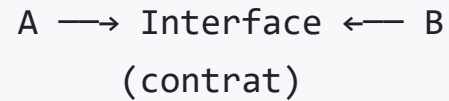
Fort couplage



Tout est connecté à tout



→ Modifier A impacte B, C, D

Faible couplage



Les modules ne se connaissent  
que via des abstractions

## 3.2 Le couplage — comparaison

 Fort couplage	 Faible couplage
<b>Code :</b>	<b>Code :</b>
<pre>class OrderService:</pre>	<pre>class OrderService:</pre>
<pre>    self.db = MySQLDatabase()</pre>	<pre>    def __init__(self, repo: OrderRepository,</pre>
<pre>    self.mailer = SmtplibMailer()</pre>	<pre>        notifier: Notifier):</pre>
	<pre>        self.repo = repo # Interface</pre>
<b>Problèmes :</b>	<b>Bénéfices :</b>
<ul style="list-style-type: none"><li>• Impossible de tester sans MySQL/SMTP</li></ul>	<ul style="list-style-type: none"><li>• Testable avec fakes</li></ul>
<ul style="list-style-type: none"><li>• Changer de DB = réécrire le service</li></ul>	<ul style="list-style-type: none"><li>• Changer MySQL → PostgreSQL = 0 impact</li></ul>
<ul style="list-style-type: none"><li>• Changer d'email = réécrire</li></ul>	<ul style="list-style-type: none"><li>• Changer SMTP → SMS = 0 impact</li></ul>

## 3.3 Les dépendances

Une **dépendance** = quelque chose dont votre code a besoin pour fonctionner.

Type	Exemples	Risque
<b>Infrastructure</b>	Base de données, système de fichiers	Changement coûteux
<b>Framework</b>	Spring, Django, Symfony	Couplage au cycle de vie du framework
<b>Services externes</b>	API paiement, météo, IA	Indisponibilité, changements d'API
<b>Bibliothèques</b>	PDF, logging, validation	Obsolescence, failles

👉 **Plus votre code dépend directement de ces éléments, plus il est fragile.**

## 3.4 Séparation des responsabilités

Chaque composant doit avoir **UNE** raison de changer.

C'est le principe **SRP** (Single Responsibility Principle).

**✗ Classe "God Object"** qui fait tout :

```
class OrderManager:
    def create_order(self): ...
    def validate_payment(self): ...
    def send_confirmation_email(self): ...
    def generate_invoice_pdf(self): ...
    def update_stock(self): ...
    def calculate_shipping(self): ...
    def apply_loyalty_points(self): ...
```

→ 7 raisons de changer cette classe = 7 sources de bugs potentiels à chaque modif.



## 3.4 Séparation — la bonne approche

✓ Chaque responsabilité isolée :

```
class OrderService:           # Création de commande
class PaymentService:         # Validation paiement
class NotificationService:    # Envoi emails/SMS
class InvoiceGenerator:        # Génération PDF
class StockService:           # Gestion stock
class ShippingCalculator:      # Calcul livraison
class LoyaltyService:         # Points fidélité
```

### Avantages :

- Chaque classe est simple et focalisée
- On peut modifier le calcul de livraison sans risquer de casser les emails
- On peut tester chaque responsabilité indépendamment

## 3.5 Inversion de dépendances

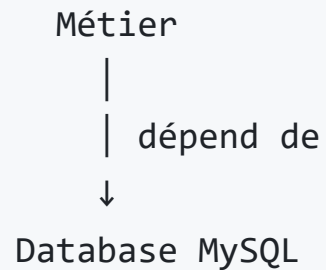
Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau.

Les deux doivent dépendre d'abstractions.

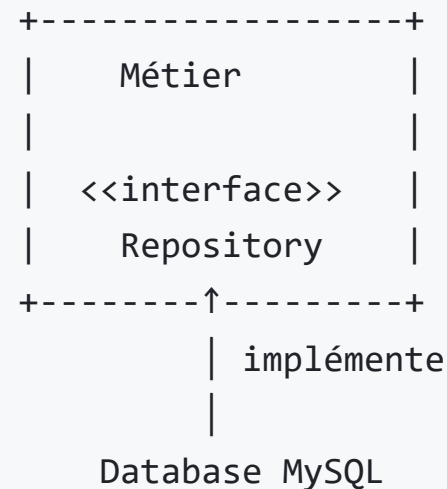
C'est le **D** de SOLID — et c'est **fondamental** pour l'architecture hexagonale.

## 3.5 Inversion — avant/après

✗ Classique (problème)



✓ Inversé (solution)



**Avant** : Métier dépend de la DB → Tester = installer MySQL → Changer de DB = réécrire métier

**Après** : Métier définit l'interface → Tester = injecter un fake → Changer de DB = nouvel adapter

## 3.6 Le rôle des tests dans l'architecture

Les tests ne servent pas qu'à détecter les bugs.  
Ils révèlent (et forcent) la qualité de votre architecture.

**Code difficile à tester = Code mal architecturé**




Si vous devez :

- Instancier 15 dépendances pour tester une fonction → **×** Trop couplé
- Lancer une DB pour tester une règle métier → **×** Pas d'inversion de dépendances
- Mocker la moitié de l'application → **×** Faible cohésion

**👉 Les tests sont un détecteur de problèmes architecturaux.**

## 3.6 TDD : piloter l'architecture par les tests

**TDD (Test-Driven Development)** : Écrire le test **AVANT** le code.

1.  Écrire un test qui échoue (Red)
2.  Écrire le code minimal pour passer (Green)
3.  Refactorer pour améliorer (Refactor)

### Bénéfices architecturaux :

- Force la testabilité et réduit le couplage
- Impose la cohésion (test complexe = trop de responsabilités)
- Garantit l'inversion (le test définit l'interface)

 *TDD ne garantit pas une bonne architecture, mais une mauvaise architecture ne survit pas au TDD.*

## Récapitulatif des principes

Principe	Question à se poser
<b>Cohésion</b>	Cette classe/module a-t-elle une responsabilité claire et unique ?
<b>Couplage</b>	Si je modifie ce module, combien d'autres sont impactés ?
<b>Dépendances</b>	Mon code métier dépend-il directement de la technique ?
<b>Responsabilités</b>	Combien de raisons cette classe a-t-elle de changer ?
<b>Inversion</b>	Qui définit les interfaces : le métier ou la technique ?

 Ces principes guident **TOUTES** les décisions architecturales.

## Pause conceptuelle

Récapitulatif rapide :

- ✓ **5 principes fondamentaux** = outils universels pour évaluer et concevoir du code
- ✓ **1 objectif commun** = code maintenable, testable, évolutif

**? Questions avant de passer à l'architecture hexagonale ?**

*Prochain sujet : comment ces principes se concrétisent dans une architecture réelle*

## 4. Architecture hexagonale (Ports & Adapters)

### 4.1 Le problème à résoudre

✗ Code "framework-first" typique :

```
@app.post("/tickets")
def create_ticket(request: Request, db: Session = Depends(get_db)):
    data = request.json()

    # Validation métier dans le controller 🤖
    if len(data["title"]) < 3:
        raise HTTPException(400, "Titre trop court")

    # Accès direct à la DB 🦷
    ticket = TicketModel(title=data["title"], status="open")
    db.add(ticket)
    db.commit()

    return {"id": ticket.id}
```

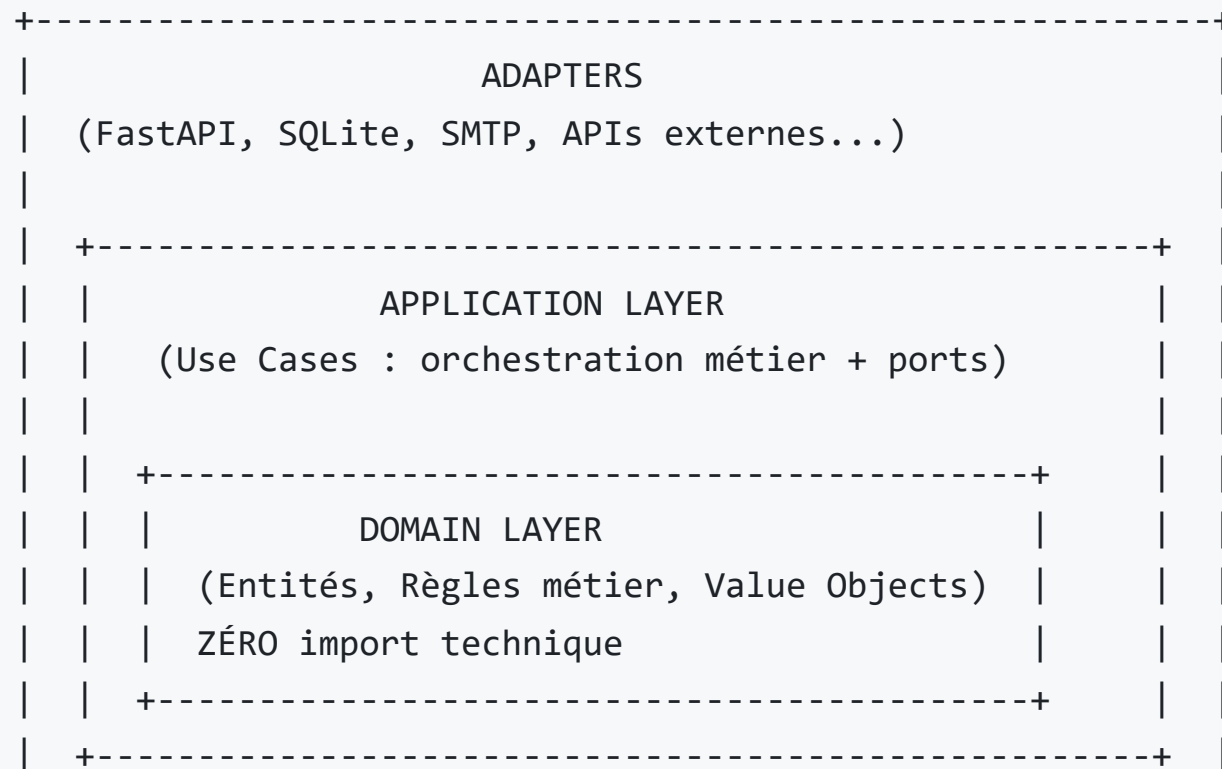


## 4.2 La solution : séparer le métier de la technique

### Principe central de l'hexagonale :

**Le domaine métier au centre, indépendant de toute technique.**

La technique s'adapte au métier, pas l'inverse.



## 4.3 Les 3 couches (1/3)

### ● DOMAIN (le cœur)

#### Contenu :

- Entités ( `Ticket` , `User` )
- Règles métier ( `ticket.assign_to()` , `ticket.close()` )
- Value Objects ( `TicketStatus` , `Email` )

#### Règle d'or :

Aucun import de framework ou lib technique (FastAPI, SQLAlchemy, etc.)

## 4.3 Les 3 couches — exemple DOMAIN (2/3)

```
# domain/ticket.py
from dataclasses import dataclass
from enum import Enum

class Status(Enum):
    OPEN = "open"
    IN_PROGRESS = "in_progress"
    RESOLVED = "resolved"
    CLOSED = "closed"

@dataclass
class Ticket:
    id: int
    title: str
    status: Status
    assignee_id: int | None = None

    def assign(self, user_id: int) -> None:
        """Règle métier : on ne peut assigner qu'un ticket ouvert."""
        if self.status != Status.OPEN:
            raise ValueError("Impossible d'assigner un ticket non ouvert")
        self.assignee_id = user_id
        self.status = Status.IN_PROGRESS
```

## 4.3 Les 3 couches — PORTS (3/4)

### ● PORTS (interfaces)

Des **contrats** (interfaces) définis par le métier :

```
# ports/ticket_repository.py
from abc import ABC, abstractmethod

class TicketRepository(ABC):
    @abstractmethod
    def save(self, ticket: Ticket) -> None: pass

    @abstractmethod
    def get(self, ticket_id: int) -> Ticket | None: pass

    @abstractmethod
    def list_all(self) -> list[Ticket]: pass
```

👉 Le métier **définit** ce dont il a besoin, sans savoir **comment** c'est implémenté.

## 4.3 Les 3 couches — APPLICATION (4/5)

### ● APPLICATION (orchestration)

**Use cases** qui coordonnent le métier et les ports :

```
# application/usecases/create_ticket.py
class CreateTicket:
    def __init__(self, ticket_repository: TicketRepository):
        self.repository = ticket_repository

    def execute(self, title: str) -> Ticket:
        ticket = Ticket(id=None, title=title, status=Status.OPEN)
        self.repository.save(ticket)
        return ticket
```

## 4.3 Les 3 couches — ADAPTERS (5/5)

Implémentations concrètes des ports :

```
# adapters/db/ticket_repository_inmemory.py
class InMemoryTicketRepository(TicketRepository):
    def __init__(self):
        self.tickets: dict[int, Ticket] = {}
        self.next_id = 1

    def save(self, ticket: Ticket) -> None:
        if ticket.id is None:
            ticket.id = self.next_id
            self.next_id += 1
        self.tickets[ticket.id] = ticket
```

## 4.4 Pourquoi c'est puissant ?

### ✓ Testabilité :

```
# Test du domaine (ZÉRO dépendance)
def test_cannot_assign_closed_ticket():
    ticket = Ticket(id=1, title="Bug", status=Status.CLOSED)
    with pytest.raises(ValueError):
        ticket.assign(user_id=42)

# Test du use case (InMemory fake)
def test_create_ticket():
    repo = InMemoryTicketRepository()
    use_case = CreateTicket(repo)
    ticket = use_case.execute("Bug critique")
    assert ticket.status == Status.OPEN
```



✓ **Évolutivité** : Passer de InMemory → SQLite → PostgreSQL sans toucher au métier

✓ **Clarté** : Chaque couche a un rôle précis

## 4.6 Justification pédagogique (1/2)




**Question légitime :** *Pourquoi l'hexagonale et pas une autre architecture ?*

**Réponses :**

1.  **Impose structurellement les bons principes**
  - Séparation domaine/infrastructure visible immédiatement
  - Impossible de contourner l'inversion de dépendances
2.  **Adaptée au format 20h TD**
  - Ni trop simple (layered classique), ni trop complexe (microservices)
  - Juste assez de contraintes pour apprendre les fondamentaux



## 4.6 Justification pédagogique (2/2)

3.  **Naturellement testable** : Tests par couche sans dépendances
    - Domain : pur (0 mock)
    - Use cases : fake repository (pas de vraie DB)
    - E2E : API complète
  4.  **Transférable** : Fondation pour comprendre toutes les archi modernes
    - Clean Architecture, Onion, DDD → mêmes concepts
    - Compatible TDD, microservices, event-driven
-  **Pour aller plus loin** : Voir les annexes pour comparaisons détaillées des architectures et discussion monolithe vs microservices

## ? Questions ou clarifications ?

**Avant de passer au projet concret :**

- Architecture hexagonale claire ?
- Différence Domain / Ports / Application / Adapters ?
- Inversion de dépendances compréhensible ?

*Prochain sujet : votre projet fil rouge (système de tickets)*

## 5. Le projet : Ticketing System

### 5.1 Vue d'ensemble (1/2)

Vous allez implémenter un **système de tickets** (simplifié) en architecture hexagonale.

#### Domaine métier :

- `Ticket` : id, titre, statut, assigné à
- `User` : id, username
- `Status` : OPEN, IN\_PROGRESS, RESOLVED, CLOSED

## 5.1 Vue d'ensemble (2/2)

### Use cases :

- Créer un ticket
- Assigner un ticket à un utilisateur
- Changer le statut d'un ticket
- Récupérer un ticket / liste de tickets

### Adapters :

- Persistance : InMemory → SQLite
- API : FastAPI (REST)

## 5.2 Progression des TDs

TD	Objectif	Couche
<b>TD0</b>	Setup environnement, workflow Git	-
<b>TD1</b>	Modéliser le domaine ( Ticket , User , Status )	Domain
<b>TD2</b>	Créer les use cases et ports	Application + Ports
<b>TD3</b>	Implémenter le repository SQLite	Adapters (DB)
<b>TD4</b>	Exposer l'API REST	Adapters (API)

## 5.3 Évaluation

### Répartition :

- 30% : Projet final (GitHub, code fonctionnel)
- 40% : Exercices de TD (livrables intermédiaires)
- 30% : Contrôle final (analyse de code architectural)

### Important :

- 70% de la note **sans IA** (TD présentiel + Contrôle)
- L'IA est **autorisée** pour le projet à la maison
- Mais **comprendre** l'architecture reste indispensable

 Grille détaillée : `td/evaluation.md`

## 5.4 Ressources

 **Template de code :**

[https://github.com/Marcennaji/ticketing\\_starter](https://github.com/Marcennaji/ticketing_starter)

 **Documentation TDs :**

<https://github.com/Marcennaji/architecture-logicielle-BUT2-ressources>

 **Technologies :**

- Python 3.11+
- FastAPI (web framework)
- SQLite (base de données, module `sqlite3` intégré)
- pytest (tests)

 **Prérequis :** Guide de démarrage à suivre **AVANT le TD0**

# **Récapitulatif & prochaines étapes**

**Vous avez maintenant :**

- ✓ Compris **pourquoi** l'architecture est essentielle (encore plus avec l'IA)
- ✓ Découvert les **5 principes fondamentaux**
- ✓ Découvert l'**architecture hexagonale** (Domain, Ports, Application, Adapters)
- ✓ Une vision du **projet ticketing**

 **Prochaine étape** : TD0 (prise en main environnement + workflow)

 **Ressources complémentaires :**

- Annexes (PDF) : [Comparaison architectures](#), [TDD](#), [Monolithe vs Microservices](#)
- Articles : [Architecture Hexagonale \(OCTO\)](#), [Hexagonale expliquée \(Les Dieux du Code\)](#)
- Toutes les ressources : <https://github.com/Marcennaji/architecture-logicielle-BUT2->