🏛️ Annexe : Architecture vs Design

Comprendre la distinction

🎯 **Objectif** : Clarifier la différence entre architecture et design (conception), deux notions souvent confondues.

Slides :

1. Définitions et portée
2. Exemples concrets
3. La frontière floue
4. Cas pratiques

📖 Définitions

🏛️ Architecture

Scope : Décisions **structurelles** à l'échelle du système

Caractéristiques :

- Impact **global** (plusieurs composants)
- Difficile/coûteux à changer après coup
- Visible dans la structure des dossiers
- Définit les **contraintes** pour le design

Questions typiques :

- Comment découper le système en couches ?
- Où placer les frontières entre modules ?

🎨 Design (Conception)

Scope : Décisions **locales** à l'intérieur d'un composant

Caractéristiques :

- Impact **localisé** (un module, une classe)
- Plus facile à refactorer
- Invisible depuis l'extérieur du composant
- Respecte les **contraintes** définies par l'architecture

Questions typiques :

- Comment structurer cette classe ?
- Quel pattern utiliser ici ? (Factory, Strategy, etc.)
- Comment nommer cette méthode ?
- Quelle signature pour cette fonction ?

🏠 La métaphore de la maison

Niveau	🏛️ Architecture	🎨 Design
Maison	• Position des murs porteurs • Fondations • Structure du toit • Étages	• Agencement des meubles • Décoration • Couleur des murs • Choix des luminaires
Changement	Nécessite permis, travaux lourds	Réaménagement le weekend
Coût erreur	Très élevé (démolition)	Faible (repeindre)
Visibilité	Depuis l'extérieur	Seulement à l'intérieur

Transposition logicielle :

- **Architecture** = Structure qui tient le projet debout
- **Design** = Organisation interne de chaque pièce

💻 Exemples Architecture

Décision 1 : Découpage en couches

```
✅ Architecture (impact global)

Décision : "Le domaine métier sera isolé de l'infrastructure"

Conséquence :
src/
├── domain/       ← Aucune dépendance technique
├── ports/        ← Interfaces définies par le métier
├── application/  ← Use cases
└── adapters/     ← Implémentations (API, DB)

→ Tous les développeurs doivent respecter ce découpage
```

Décision 2 : Flux de dépendances

```
✅ Architecture (impact global)
```

💻 Exemples Design

Décision 1 : Pattern de création

```python
# 🎨 Design (local à la création de tickets)

# Option A : Factory simple
def create_ticket(title: str) -> Ticket:
    return Ticket(id=None, title=title, status=Status.OPEN)

# Option B : Builder (si complexe)
class TicketBuilder:
    def __init__(self):
        self._ticket = Ticket()

    def with_title(self, title: str) -> "TicketBuilder":
        self._ticket.title = title
        return self

    def build(self) -> Ticket:
        return self._ticket
```

Décision 2 : Nommage et signature

```python
# 🎨 Design (choix de conception locale)

# Option A : Méthode vs fonction
class Ticket:
    def assign_to(self, user_id: int) -> None:  # Méthode
        self.assignee = user_id

# vs

def assign_ticket(ticket: Ticket, user_id: int) -> Ticket:  # Fonction
    ticket.assignee = user_id
    return ticket

# Option B : Nom explicite vs concis
def assign_to_user(user_id: int)  # Explicite
def assign(user_id: int)           # Concis
```

Impact : Localisé à l'interface de `Ticket`

🌫️ La frontière est floue

Pourquoi la distinction n'est pas toujours claire ?

1. Échelle relative

Contexte	Architecture	Design
Application complète	Hexagonale (couches)	Organisation d'une classe
Module métier	Structure du module	Implémentation d'une méthode
Classe	Responsabilités de la classe	Algorithme interne

💡 **L'architecture d'un module = le design du système**

2. Impact contextuel

Exemple : Héritage vs Composition

```
# Cas 1 : Design pur (reste dans une couche)
class TicketPrioritaire:
    def __init__(self, ticket: Ticket, priority: int):  # Composition
        self.ticket = ticket
        self.priority = priority

# ✅ Design : choix local, pas d'impact architectural
```

```
# Cas 2 : Architectural (traverse les couches)
class TicketEntity(SQLAlchemyModel):  # Héritage
    # ❌ Architecture : le domaine hérite de l'infrastructure
    # → Viole l'inversion de dépendances
```

Règle : Si ça couple des couches → Architecture

📊 Tableau comparatif approfondi

Critère	🏛️ Architecture	🎨 Design
Portée	Système, plusieurs modules	Module, classe, fonction
Durée de vie	Long terme (années)	Moyen terme (mois)
Coût changement	Très élevé	Faible à moyen
Responsable	Architecte, Lead Dev	Développeur
Documentation	ADR (Architecture Decision Records)	Code comments, docstrings
Tests	Tests d'intégration, E2E	Tests unitaires
Outils	Diagrammes C4, UML composants	UML classes, diagrammes de séquence
Revue	Architecture review board	Code review

🔍 Cas pratiques

Cas 1 : "Préférez la composition à l'héritage"

Question : Architecture ou Design ?

Réponse : Majoritairement Design (85%)

Pourquoi ?

- Décision locale à la classe
- N'impacte pas les frontières entre couches
- Peut être refactoré facilement

Exception (15%) : Si ça crée un couplage entre couches

```
class DomainEntity(ORMBase):  # ❌ Architecture
    # Le domaine dépend de l'infrastructure
```

Cas 2 : "Utiliser des interfaces pour les dépendances"

Question : Architecture ou Design ?

Réponse : Architecture

Pourquoi ?

- Définit le contrat entre couches
- Impact sur toute l'application
- Difficile à changer après (tous les clients)

```
# 🏛️ Architecture
class TicketRepository(ABC):  # Interface définie
    @abstractmethod
    def save(self, ticket: Ticket) -> None:
        pass

# Tous les adapters doivent l'implémenter
# → Décision structurante
```

Cas 3 : "Découper une classe God Object"

Question : Architecture ou Design ?

Réponse : Ça dépend !

Scénario A : Découpage interne

```
# Avant
class OrderManager:
    def create_order(self): ...
    def calculate_total(self): ...  # Design : extraire dans Calculator

# 🎨 Design si ça reste dans le même module
```

Scénario B : Découpage en modules

```
# Après
domain/order.py        → Order entity
application/create_order.py  → Use case
domain/pricing.py      → Pricing logic
```

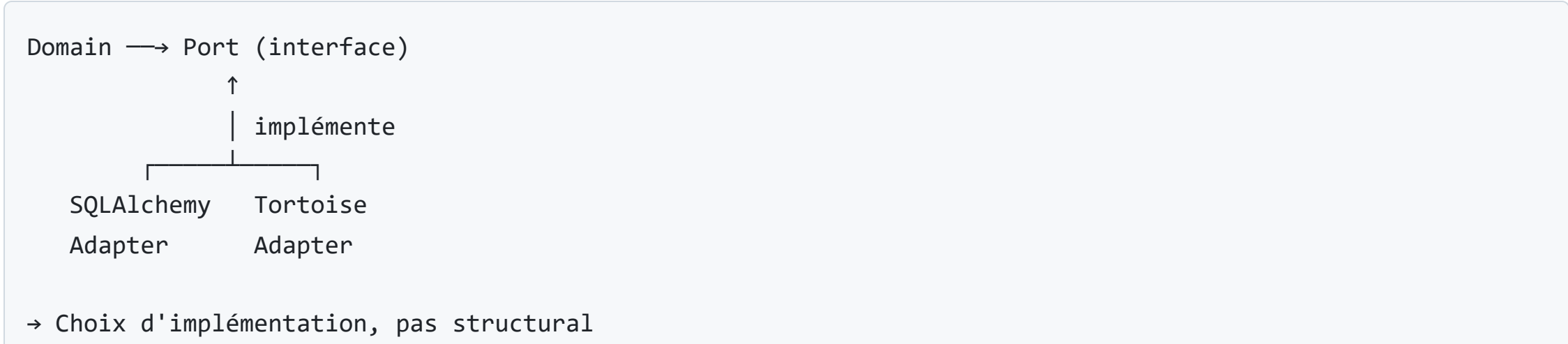
Cas 4 : "Choisir un ORM (SQLAlchemy vs Tortoise)"

Question : Architecture ou Design ?

Réponse : Design (si architecture hexagonale)

Pourquoi ?

- L'ORM est dans un **adapter** (couche infrastructure)
- Le domaine ne le voit pas (inversion de dépendances)
- Peut être changé sans impacter l'architecture



🎯 Règles pratiques pour distinguer

C'est de l'Architecture si :

- ✅ Ça impacte **plusieurs modules/couches**
- ✅ Ça définit des **contraintes** pour les développeurs
- ✅ Ça apparaît dans la **structure de dossiers**
- ✅ Ça nécessite un **ADR** (Architecture Decision Record)
- ✅ Difficile à changer **une fois en production**

C'est du Design si :

- ✅ Ça reste **dans un module/une classe**
- ✅ Ça peut être **refactoré facilement**
- ✅ Ça ne change pas les **dépendances entre couches**
- ✅ Ça relève d'un **pattern de conception**

💡 Principe clé

> **Architecture = "Quoi et où ?"** (décisions structurelles)
> **Design = "Comment ?"** (implémentation locale)

Analogie musicale :

- 🏛️ **Architecture** = Choisir les instruments d'un orchestre (cuivres, cordes, percussions)
- 🎨 **Design** = Comment chaque musicien joue sa partition

Analogie culinaire :

- 🏛️ **Architecture** = Menu (entrée, plat, dessert)
- 🎨 **Design** = Recette de chaque plat

📚 Exemple complet : Ticketing System

Décisions d'Architecture

1. ✅ Hexagonale (couches : domain/ports/adapters)
2. ✅ Domain pur Python (zéro import technique)
3. ✅ Ports définis par le métier
4. ✅ FastAPI pour l'API, SQLAlchemy pour la DB

Décisions de Design

1. 🎨 `Ticket` = dataclass (vs class normale)
2. 🎨 `Status` = Enum (vs constantes)
3. 🎨 Validation dans `Ticket.assign()` (vs validator externe)
4. 🎨 Repository InMemory = dict (vs list)

🤔 Zones grises courantes

1. Patterns architecturaux vs patterns de conception

Type	Exemples
Patterns architecturaux	Hexagonale, Layered, CQRS, Event Sourcing
Patterns de conception	Factory, Strategy, Observer, Decorator

💡 Les patterns de conception **servent** l'architecture

2. SOLID : Architecture ou Design ?

Réponse : Les deux !

- **S, R, P** (Single Responsibility) → Plutôt Design (échelle classe)
- **O** (Open/Closed) → Les deux (extensibilité)
- **L** (Liskov Substitution) → Design (héritage)

🎓 Conséquences pratiques

Pour le cours

Slides sur l'Architecture :

- Hexagonale (couches, dépendances)
- Inversion de dépendances
- Séparation métier/infrastructure

Slides sur le Design :

- Patterns (Factory, Strategy)
- SOLID (S, R, P, L, I)
- Clean Code (nommage, fonctions)

Pour le projet

? Questions fréquentes

Q : Microservices, c'est de l'architecture ou du design ?

R : Architecture (découpage système en services autonomes)

Q : Choisir entre MySQL et PostgreSQL ?

R : Design (si architecture hexagonale → juste un adapter)

Q : TDD, c'est de l'architecture ou du design ?

R : Les deux ! TDD influence design (testabilité) ET architecture (couplage)

Q : DRY (Don't Repeat Yourself) ?

R : Plutôt Design (refactoring local), mais peut devenir architectural (duplication entre modules)

🏁 Fin de l'annexe

À retenir :

- Architecture = décisions **structurelles** difficiles à changer
- Design = décisions **locales** faciles à refactorer
- La frontière est **contextuelle** et parfois floue
- **Les deux sont importants** et complémentaires

➡️ Retour au cours principal