🧪 Annexe : TDD et Tests en Architecture

Les tests comme détecteur architectural

🎯 **Objectif** : Comprendre comment les tests révèlent (et améliorent) la qualité architecturale.

Slides :

1. Les tests révèlent l'architecture
2. TDD : Red-Green-Refactor
3. TDD et architecture hexagonale
4. Exemples concrets
5. Pièges à éviter

🔍 Les tests : plus qu'un outil de validation

Vision classique (limitée)

> "Les tests servent à détecter les bugs avant la production."

✅ Vrai, mais **incomplet**

Vision architecturale (complète)

> "Les tests révèlent la qualité de votre architecture.
> Code difficile à tester = Code mal architecturé."

Principe fondamental :

- Si tester est **simple** → Bonne architecture ✅
- Si tester est **complexe** → Mauvaise architecture ❌

🚨 Symptômes d'une mauvaise architecture

Vous savez que votre architecture a un problème quand :

1. Dépendances impossibles à mocker

```
# ❌ Impossible à tester
class OrderService:
    def create_order(self, order):
        db = MySQLDatabase()  # Instanciation directe
        db.insert(order)
```

Symptôme : Couplage fort

Diagnostic : Pas d'inversion de dépendances

2. Setup de test gigantesque

```python
# ❌ Setup cauchemardesque
def test_assign_ticket():
    # 50 lignes de setup
    db = setup_database()
    migrate_schema()
    seed_test_data()
    app = create_app()
    client = TestClient(app)

    # 3 lignes de test
    response = client.post("/tickets/1/assign", json={"user_id": 42})
    assert response.status_code == 200
```

Symptôme : Faible cohésion → Métier mélangé à l'infrastructure

3. Besoin d'infrastructure réelle

```
# ❌ Nécessite MySQL, Redis, RabbitMQ...
def test_business_rule():
    db = MySQLConnection()  # Vraie DB
    cache = RedisConnection()  # Vrai Redis
    queue = RabbitMQConnection()  # Vraie queue

    result = calculate_discount(user, cart)
```

Symptôme : Dépendances directes → Pas d'abstraction (ports)

4. Tests qui cassent pour de mauvaises raisons

```
# ❌ Test casse si on change de MySQL à PostgreSQL
def test_create_ticket():
    ticket = Ticket(title="Bug")
    # Ce test ne devrait tester QUE le métier
    # Mais il casse si on change la DB
```

Symptôme : Couplage infrastructure

Diagnostic : Métier dépend de détails techniques

✅ Signes d'une bonne architecture

Votre architecture est bonne quand :

1. Tests unitaires simples

```python
# ✅ Test du domaine : pur, rapide, fiable
def test_cannot_assign_closed_ticket():
    ticket = Ticket(id=1, title="Bug", status=Status.CLOSED)

    with pytest.raises(ValueError):
        ticket.assign(user_id=42)
```

Caractéristiques :

- Zéro dépendance externe
- Setup minimal (1-2 lignes)
- Rapide (< 1ms)

2. Mocks/fakes faciles

```
# ✅ Fake repository simple
class FakeTicketRepository(TicketRepository):
    def __init__(self):
        self.tickets = {}
    def save(self, ticket):
        self.tickets[ticket.id] = ticket

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

Caractéristiques : Interface claire, Fake en 10 lignes

3. Pyramide de tests équilibrée

```
      /\
     /E2E\         ← Peu (lents, fragiles)
    /──────\
   / Integ \       ← Quelques-uns (adapters)
  /──────────\
 /  Unitaires \    ← Beaucoup (rapides, fiables)
/──────────────\
```

Répartition saine :

- **70%** Tests unitaires (domaine + use cases)
- **20%** Tests d'intégration (adapters)
- **10%** Tests E2E (API complète)

Architecture hexagonale → Favorise cette pyramide naturellement

🔴🟢♻️ TDD : Test-Driven Development

Le cycle Red-Green-Refactor

```
1. 🔴 RED : Écrire un test qui ÉCHOUE
   ↓
2. 🟢 GREEN : Écrire le code MINIMAL pour passer
   ↓
3. ♻️ REFACTOR : Améliorer le code (sans casser les tests)
   ↓
   └→ Retour à 1 (nouveau test)
```

Principe : Le test **pilote** le code, pas l'inverse

Exemple concret : Créer un ticket

🔴 Étape 1 : RED

```
# test_ticket.py
def test_new_ticket_has_open_status():
    ticket = Ticket(title="Bug critique")
    assert ticket.status == Status.OPEN
```

Résultat : ❌ Test échoue (classe `Ticket` n'existe pas)

🟢 Étape 2 : GREEN

```python
# ticket.py
class Status(Enum):
    OPEN = "open"

class Ticket:
    def __init__(self, title: str):
        self.title = title
        self.status = Status.OPEN
```

Résultat : ✅ Test passe

♻️ Étape 3 : REFACTOR

```python
# ticket.py (amélioration)
from dataclasses import dataclass

@dataclass
class Ticket:
    title: str
    status: Status = Status.OPEN
```

Résultat : ✅ Test passe toujours (refactoring réussi)

Pourquoi TDD améliore l'architecture ?

1. Force la testabilité

Si vous ne pouvez pas écrire le test **avant** le code :

- Soit le design est trop couplé
- Soit vous n'avez pas pensé aux dépendances

→ TDD vous oblige à réfléchir à l'interface **avant** l'implémentation

2. Réduit le couplage

```python
# ❌ Sans TDD (couplage direct)
class OrderService:
    def __init__(self):
        self.db = MySQLDatabase()  # Comment tester ?

# ✅ Avec TDD (inversion de dépendances)
class OrderService:
    def __init__(self, repo: OrderRepository):
        self.repo = repo  # Mockable !
```

TDD vous force à injecter les dépendances (sinon impossible de tester)

3. Impose la cohésion

```
# Si votre test devient complexe...
def test_create_order():
    # Setup 30 lignes
    # ...
    # Test 3 lignes

# → Signe que la classe fait trop de choses
# → TDD vous pousse à découper
```

Règle d'or TDD : Setup complexe = responsabilités mal découpées

4. Garantit l'inversion de dépendances

```
# Le test définit ce dont il a besoin
def test_create_ticket():
    repo = FakeTicketRepository()  # Interface
    use_case = CreateTicket(repo)

    ticket = use_case.execute("Bug")
    assert ticket in repo.tickets.values()

# → Le test est le "client" qui force l'interface
# → L'implémentation s'adapte au test
```

TDD inverse naturellement les dépendances

🛡️ TDD + Architecture Hexagonale = ❤️

Synergie parfaite

Aspect	Hexagonale	TDD	Combo
Testabilité	Forcée structurellement	Forcée méthodologiquement	Double garantie
Inversion	Par design	Par nécessité	Cohérent
Domaine pur	Zéro import technique	Testable sans infra	Rapide
Adapters	Implémentations swappables	Mockables facilement	Flexible

TDD rend l'hexagonale plus facile

L'hexagonale rend TDD plus naturel

Workflow TDD-Hexagonal

Ordre de développement :

```
1. 🔴 Test du domaine (entité)
   ↓
2. 🟢 Implémenter l'entité
   ↓
3. 🔴 Test du use case (avec fake repo)
   ↓
4. 🟢 Implémenter le use case
   ↓
5. 🔴 Test de l'adapter (ex: repo SQL)
   ↓
6. 🟢 Implémenter l'adapter
   ↓
7. 🔴 Test E2E (API complète)
   ↓
8. 🟢 Câbler tout ensemble dans main.py
```

Avantage : Chaque couche est testée **avant** d'être intégrée

💻 Exemple complet : Ticketing avec TDD

Étape 1 : Domaine

```
# ❌ Test échoue
def test_assign_ticket():
    ticket = Ticket(title="Bug", status=Status.OPEN)
    ticket.assign(user_id=42)

    assert ticket.assignee_id == 42
    assert ticket.status == Status.IN_PROGRESS
```

```
# ✅ Implémentation
class Ticket:
    def assign(self, user_id: int):
        if self.status != Status.OPEN:
            raise ValueError("Ticket non ouvert")
        self.assignee_id = user_id
        self.status = Status.IN_PROGRESS
```

Étape 2 : Use Case

```python
# ❌ Test échoue
def test_create_ticket_use_case():
    repo = FakeTicketRepository()
    use_case = CreateTicket(repo)

    ticket = use_case.execute(title="Bug critique")

    assert ticket.status == Status.OPEN
    assert ticket in repo.tickets.values()
```

```python
# ✅ Implémentation
class CreateTicket:
    def __init__(self, repo: TicketRepository):
        self.repo = repo

    def execute(self, title: str) -> Ticket:
        ticket = Ticket(title=title, status=Status.OPEN)
        self.repo.save(ticket)
        return ticket
```

Étape 3 : Adapter

```python
# ❌ Test échoue
def test_sql_repository_save():
    repo = SQLTicketRepository(db_session)
    ticket = Ticket(title="Bug", status=Status.OPEN)

    repo.save(ticket)

    saved = repo.get(ticket.id)
    assert saved.title == "Bug"
```

```python
# ✅ Implémentation
class SQLTicketRepository(TicketRepository):
    def save(self, ticket: Ticket):
        model = TicketModel.from_entity(ticket)
        self.session.add(model)
        self.session.commit()
```

Étape 4 : E2E

```
# ❌ Test échoue
def test_api_create_ticket():
    client = TestClient(app)

    response = client.post("/tickets", json={"title": "Bug"})

    assert response.status_code == 201
    assert response.json()["status"] == "open"
```

```
# ✅ Implémentation (router)
@app.post("/tickets")
def create_ticket(data: CreateTicketRequest):
    use_case = get_create_ticket_use_case()  # DI
    ticket = use_case.execute(data.title)
    return TicketResponse.from_entity(ticket)
```

🎯 Pyramide de tests en Hexagonale

```
            /\
           /E2E\          ← 5 tests (API complète)
          /─────\
         / Integ \        ← 10 tests (adapters SQL, API)
        /─────────\
       /Application\      ← 20 tests (use cases + fakes)
      /─────────────\
     /    Domain     \    ← 50 tests (entités pures)
    /_________________\

Total : ~85 tests
Temps d'exécution : ~2 secondes
```

Répartition idéale :

- **60%** Domain (rapide, fiable, zéro infra)
- **25%** Application (fake repos, pas de DB)

⚠️ Pièges à éviter avec TDD

1. Tester l'implémentation, pas le comportement

```
# ❌ Test fragile (couplé à l'implémentation)
def test_ticket_uses_dict():
    repo = InMemoryTicketRepository()
    assert isinstance(repo.tickets, dict)  # ❌ Détail interne

# ✅ Test comportemental
def test_save_and_retrieve_ticket():
    repo = InMemoryTicketRepository()
    ticket = Ticket(title="Bug")

    repo.save(ticket)
    retrieved = repo.get(ticket.id)

    assert retrieved.title == "Bug"  # ✅ Comportement
```

2. Over-mocking

```
# ❌ Too many mocks
def test_create_ticket():
    mock_repo = Mock()
    mock_logger = Mock()
    mock_notifier = Mock()
    mock_validator = Mock()
    mock_event_bus = Mock()

    use_case = CreateTicket(mock_repo, mock_logger, ...)
    # ...

# → Signe que le use case a trop de responsabilités
```

Solution : Simplifier le design (cohésion)

3. Tests trop couplés

```python
# ❌ Test qui connaît trop de détails
def test_create_ticket():
    use_case = CreateTicket(repo)

    # On teste l'implémentation interne
    assert use_case._validate_title("Bug") == True
    assert use_case._generate_id() is not None

# ✅ Test de l'interface publique
def test_create_ticket():
    use_case = CreateTicket(repo)

    ticket = use_case.execute("Bug")
    assert ticket.title == "Bug"
```

4. Ignorer le refactor

```
Red → Green → ❌ STOP

→ Code fonctionne mais technique debt s'accumule
```

TDD complet :

```
Red → Green → Refactor → Red → Green → Refactor ...

→ Code fonctionne ET est maintenable
```

Refactor = partie intégrante de TDD, pas optionnelle

💡 TDD : Mythes vs Réalité

Mythe	Réalité
"TDD ralentit le développement"	TDD accélère sur le moyen terme (moins de bugs, moins de refonte)
"TDD = 100% de couverture"	TDD vise le comportement, pas la métrique
"TDD remplace l'architecture"	TDD **révèle** les problèmes, ne les résout pas automatiquement
"TDD impose le design"	TDD **guide** vers un bon design, ne l'impose pas
"TDD = écrire tous les tests avant"	TDD = cycle court (1 test → 1 impl → refactor)

À retenir :

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

🧰 Outils et bonnes pratiques

Frameworks de test Python

```python
# pytest (recommandé)
def test_ticket_creation():
    ticket = Ticket(title="Bug")
    assert ticket.status == Status.OPEN

# unittest (stdlib)
class TestTicket(unittest.TestCase):
    def test_creation(self):
        ticket = Ticket(title="Bug")
        self.assertEqual(ticket.status, Status.OPEN)
```

pytest > unittest pour TDD (moins verbeux)

Fakes vs Mocks vs Stubs

```
# Fake : implémentation simple réelle
class FakeTicketRepository(TicketRepository):
    def __init__(self):
        self.tickets = {}
    def save(self, ticket):
        self.tickets[ticket.id] = ticket

# Mock : objet configuré pour vérifier les appels
mock_repo = Mock(spec=TicketRepository)
use_case.execute("Bug")
mock_repo.save.assert_called_once()

# Stub : retourne des valeurs prédéfinies
stub_repo = Mock()
stub_repo.get.return_value = Ticket(title="Bug")
```

Préférence : Fakes > Stubs > Mocks (dans cet ordre)

Coverage : utile mais pas suffisant

```
pytest --cov=src --cov-report=html
```

Métriques saines :

- Domain : 100% (facile, code pur)
- Application : 90%+ (use cases critiques)
- Adapters : 70%+ (intégration)

⚠️ **Attention :**

- 100% coverage ≠ bons tests
- Tester le **comportement**, pas juste la couverture

🎓 Pour votre projet

Stratégie recommandée

TD1 : Domain

- TDD strict (test avant code)
- Tous les tests unitaires (rapides)

TD2 : Application

- TDD avec fake repositories
- Tests de use cases

TD3 : Adapters

- Tests d'intégration (vraie DB)
- TDD optionnel (setup plus complexe)

Checklist TDD

Avant de commit, vérifier :

- [ ] Chaque fonctionnalité a au moins 1 test
- [ ] Tous les tests passent (vert)
- [ ] Pas de code non testé "pour plus tard"
- [ ] Tests rapides (< 1s pour les unitaires)
- [ ] Pas de dépendance externe dans tests unitaires
- [ ] Refactoring fait (pas de dette technique volontaire)

? Questions fréquentes

Q : Faut-il TOUJOURS faire du TDD ?

R : Non. TDD est très utile pour le métier critique. Moins pour du glue code simple.

Q : TDD pour des bugs ?

R : Oui ! 1) Écrire test qui reproduit le bug (RED), 2) Fixer (GREEN), 3) Refactor.

Q : TDD avec une DB ?

R : TDD du domain/application (sans DB), puis tests d'intégration pour l'adapter.

Q : TDD ralentit au début ?

R : Oui (courbe d'apprentissage). Mais accélère dès la 2e semaine (moins de bugs).

🏁 Fin de l'annexe

À retenir :

- Tests = détecteur de qualité architecturale
- Code difficile à tester = code mal architecturé
- TDD force la testabilité → améliore l'architecture
- TDD + Hexagonale = synergie naturelle
- Pyramide : beaucoup d'unitaires, peu d'E2E

Citation clé :

> "Si vous ne pouvez pas tester facilement votre code,
> c'est que votre architecture a un problème."

➡️ Retour au cours principal