# Manuel de Révision Condensé FULL — 2-3 Heures

## Guide de Révision Intensive des Concepts Essentiels

---

**Objectif :** Ce manuel concentre les concepts clés de Design Patterns, DSA, ERD, SQL et UML en un format digestible en 2-3 heures. Chaque section présente l'essentiel avec des exemples concrets et des solutions rapides.

**Public cible :** Développeurs se préparant à des entretiens techniques ou examens nécessitant une révision rapide mais complète.

**Structure :** 5 sections principales, chacune avec théorie condensée + exemples pratiques + pièges courants.

---

## Table des Matières



---

## 1. Design Patterns Essentiels

### Patterns Creational (Création d'objets)

**Singleton — Une seule instance** **Quand l'utiliser :** Configuration globale, connexion DB, logger

```python
class Config:
    _instance = None
    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance.env = "prod"
        return cls._instance
```

```java
public class Config {
    private static final Config INSTANCE = new Config();
    private Config() {}
    public static Config getInstance() { return INSTANCE; }
}
```

**Pièges :** Thread-safety en environnement concurrent, difficile à tester (état global).

---

**Factory Method — Déléguer la création** **Quand l'utiliser :** Créer différents types d'objets sans `switch` dans le code client.

```python
class CompteFactory:
    @staticmethod
    def create(type, titulaire):
        if type == "COURANT":
            return CompteCourant(titulaire)
        elif type == "EPARGNE":
            return CompteEpargne(titulaire)
        raise ValueError("Type inconnu")
```

```
interface CompteFactory {
    Compte creer(String titulaire);
}
class CompteCourantFactory implements CompteFactory {
    public Compte creer(String titulaire) { return new CompteCourant(titulaire); }
}
```

**Avantages :** Découplage, extensibilité (ajouter un nouveau type sans modifier le code existant).

---

**Builder — Construction complexe** **Quand l'utiliser :** Objets avec beaucoup de paramètres optionnels.

```
class CompteBuilder:
    def __init__(self, titulaire):
        self.titulaire = titulaire
        self.devise = "HTG"
        self.services = []

    def with_devise(self, devise):
        self.devise = devise
        return self

    def add_service(self, service):
        self.services.append(service)
        return self

    def build(self):
        return Compte(self.titulaire, self.devise, self.services)

# Usage
compte = CompteBuilder("Alice").with_devise("USD").add_service("SMS").build()
```

**Avantages :** Lisibilité, validation à la construction, immutabilité possible.

---

**Patterns Structural (Structure et composition)**

**Adapter — Convertir une interface** **Quand l'utiliser :** Intégrer une API externe avec interface incompatible.

```
class ExternalAPI:
    def get_usd_to_htg(self): return 132.5

class CurrencyAdapter:
    def __init__(self, api):
        self.api = api

    def convert(self, from_currency, to_currency, amount):
        if from_currency == "USD" and to_currency == "HTG":
            return amount * self.api.get_usd_to_htg()
        raise ValueError("Conversion non supportée")
```

---

**Decorator — Ajouter des fonctionnalités dynamiquement** **Quand l'utiliser :** Ajouter logging, validation, cache sans modifier la classe de base.

```
class Compte:
    def retirer(self, montant):
```

```
        self.solde -= montant

class AuditDecorator:
    def __init__(self, compte):
        self.compte = compte

    def retirer(self, montant):
        print(f"AUDIT: Retrait de {montant}")
        return self.compte.retirer(montant)
```

**Alternative moderne :** Décorateurs Python (`@audit`), Proxy Java.

---

**Facade — Simplifier une interface complexe** **Quand l'utiliser :** Masquer la complexité d'un sous-système (KYC, cartes, comptes).

```
class BankFacade:
    def __init__(self):
        self.kyc = KYCService()
        self.account = AccountService()
        self.card = CardService()

    def onboard_client(self, client_data):
        if not self.kyc.verify(client_data):
            raise ValueError("KYC échoué")
        account = self.account.create(client_data)
        card = self.card.issue(account)
        return {"account": account, "card": card}
```

---

### Patterns Behavioral (Comportement et communication)

**Observer — Notifier les abonnés** **Quand l'utiliser :** Notifications, event-driven architecture.

```
class Observable:
    def __init__(self):
        self._observers = []

    def subscribe(self, observer):
        self._observers.append(observer)

    def notify(self, event):
        for obs in self._observers:
            obs.update(event)

class EmailObserver:
    def update(self, event):
        print(f"Email envoyé: {event}")

# Usage
compte = Observable()
compte.subscribe(EmailObserver())
compte.notify("Solde faible")
```

---

**Strategy — Algorithmes interchangeables** **Quand l'utiliser :** Différentes façons de calculer (frais, intérêts, taxes).

```python
class FeeStrategy:
    def calculate(self, amount): pass

class StandardFee(FeeStrategy):
    def calculate(self, amount): return amount * 0.01

class PremiumFee(FeeStrategy):
    def calculate(self, amount): return amount * 0.005

class Transaction:
    def __init__(self, amount, fee_strategy):
        self.amount = amount
        self.fee_strategy = fee_strategy

    def total(self):
        return self.amount + self.fee_strategy.calculate(self.amount)
```

**Avantage :** Évite les `if/else` ou `switch` massifs.

---

### Résumé Design Patterns

| Pattern | Problème résolu | Exemple bancaire |
|---|---|---|
| Singleton | Instance unique | Configuration système |
| Factory | Création polymorphe | Types de comptes |
| Builder | Construction complexe | Compte avec options |
| Adapter | Interface incompatible | API externe change |
| Decorator | Ajouter fonctionnalités | Audit, logging |
| Facade | Simplifier complexité | Onboarding client |
| Observer | Notification événements | Alertes transactions |
| Strategy | Algorithmes multiples | Calcul de frais |

---

## 2. Data Structures & Algorithms (DSA)

### Structures de Données Fondamentales

**Arrays & Lists** **Complexités :** - Accès: O(1) - Insertion début: O(n) (array), O(1) (linked list) - Recherche: O(n) non trié, O(log n) trié avec binary search

**Problème classique : Two Sum**

```python
def two_sum(nums, target):
    seen = {}
    for i, n in enumerate(nums):
        need = target - n
        if need in seen:
            return [seen[need], i]
        seen[n] = i
    return []
```

**Complexité :** O(n) temps, O(n) espace.

---

**Hash Tables (Dict/Map)** **Utilisation :** Lookup O(1), compter fréquences, détecter doublons.

**Problème : Premier caractère unique**

```python
from collections import Counter

def first_unique_char(s):
    counts = Counter(s)
    for i, ch in enumerate(s):
        if counts[ch] == 1:
            return i
    return -1
```

**Pièges :** Collisions (rarement un problème en Python/Java), consommation mémoire.

---

**Stacks & Queues** **Stack (LIFO) :** Parenthèses valides, undo/redo, DFS.

```python
def is_valid_parentheses(s):
    stack = []
    pairs = {')': '(', ']': '[', '}': '{'}
    for ch in s:
        if ch in pairs.values():
            stack.append(ch)
        elif ch in pairs:
            if not stack or stack.pop() != pairs[ch]:
                return False
    return len(stack) == 0
```

**Queue (FIFO) :** BFS, traitement de tâches.

---

**Trees (Arbres)** **Arbre Binaire de Recherche (BSR) :** - Recherche, insertion, suppression : O(log n) si équilibré, O(n) pire cas.

**Traversées :** - **Inorder** (gauche, root, droite) → ordre croissant pour BSR - **Preorder** (root, gauche, droite) - **Postorder** (gauche, droite, root) - **Level-order** (BFS par niveau)

```python
def inorder(root):
    if not root:
        return []
    return inorder(root.left) + [root.val] + inorder(root.right)
```

**BFS Level-Order :**

```python
from collections import deque

def level_order(root):
    if not root:
        return []
    result, queue = [], deque([root])
    while queue:
        level = []
        for _ in range(len(queue)):
            node = queue.popleft()
            level.append(node.val)
            if node.left: queue.append(node.left)
            if node.right: queue.append(node.right)
        result.append(level)
    return result
```

## Algorithmes Essentiels

### Tri (Sorting)

| Algorithme | Complexité Moyenne | Espace | Stable |
|---|---|---|---|
| QuickSort | O(n log n) | O(log n) | Non |
| MergeSort | O(n log n) | O(n) | Oui |
| HeapSort | O(n log n) | O(1) | Non |
| TimSort (Python) | O(n log n) | O(n) | Oui |

**En pratique :** Utilisez `sorted()` (Python) ou `Arrays.sort()` (Java) sauf besoin spécifique.

---

**Recherche Binaire** **Prérequis :** Tableau trié.

```python
def binary_search(nums, target):
    lo, hi = 0, len(nums) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            lo = mid + 1
        else:
            hi = mid - 1
    return -1
```

**Complexité :** O(log n).

**Variante :** Recherche du premier/dernier élément satisfaisant une condition.

---

**Two Pointers** **Problème : Paires avec somme donnée (tableau trié)**

```python
def two_sum_sorted(nums, target):
    left, right = 0, len(nums) - 1
    while left < right:
        current = nums[left] + nums[right]
        if current == target:
            return [left, right]
        elif current < target:
            left += 1
        else:
            right -= 1
    return []
```

**Complexité :** O(n) temps, O(1) espace.

---

**Sliding Window** **Problème : Sous-chaîne maximale sans répétition**

```python
def length_of_longest_substring(s):
    seen = {}
    left = max_len = 0
```

```
    for right, ch in enumerate(s):
        if ch in seen and seen[ch] >= left:
            left = seen[ch] + 1
        seen[ch] = right
        max_len = max(max_len, right - left + 1)
    return max_len
```

**Complexité :** O(n).

---

**Backtracking Problème : Générer toutes les permutations**

```
def permutations(nums):
    result = []
    def backtrack(path, remaining):
        if not remaining:
            result.append(path[:])
            return
        for i in range(len(remaining)):
            backtrack(path + [remaining[i]], remaining[:i] + remaining[i+1:])
    backtrack([], nums)
    return result
```

**Utilisation :** Sudoku, N-Queens, génération de combinaisons.

---

**Résumé DSA**

**Structures à maîtriser :** - Array/List, Hash Table, Stack, Queue, Linked List, Tree, Heap, Graph

**Algorithmes à maîtriser :** - Binary Search, Two Pointers, Sliding Window, BFS/DFS, Backtracking, DP (niveau avancé)

**Complexités courantes :** - O(1) : Hash lookup - O(log n) : Binary search, heap operations - O(n) : Linear scan, BFS/DFS - O(n log n) : Sorting, divide & conquer - $O(n^2)$ : Nested loops (à éviter si possible)

---

## 3. Entity-Relationship Diagrams (ERD)

**Concepts Fondamentaux**

**ERD (Entity-Relationship Diagram) :** Représentation visuelle de la structure d'une base de données.

**Composants :** - **Entité :** Objet du monde réel (Client, Compte, Transaction) - **Attribut :** Propriété d'une entité (nom, email, solde) - **Relation :** Lien entre entités (un Client possède plusieurs Comptes) - **Cardinalité :** Nombre d'instances (1:1, 1:N, M:N)

---

**Notation Crow's Foot (la plus utilisée)**

```
  ENTITE       ← Rectangle = Entité

PK id          ← PK = Primary Key
   nom         ← Attributs
FK autre_id    ← FK = Foreign Key
```

**Cardinalités :**

```
|   Zéro ou un
    Exactement un
<    Zéro ou plusieurs
<    Un ou plusieurs
```

---

**Exemple Bancaire Complet**

```
     CLIENT

PK client_id           <
   nom                          Relation 1:N
   email (UQ)                   Un client → plusieurs comptes
   telephone


                          COMPTE

                    PK compte_id               <
                    FK client_id
                       numero_compte
                       type_compte
                       solde



                                          TRANSACTION

                                     PK transaction_id
                                     FK compte_id
                                        type_transaction
                                        montant
                                        date_transaction
```

**Relations expliquées :** 1. **CLIENT** **< COMPTE** : Un client a 0 ou plusieurs comptes 2. **COMPTE** **< TRANSACTION** : Un compte a 0 ou plusieurs transactions

---

**Types de Relations**

**1:1 (Un à Un) — Carte bancaire unique par compte**

```
     COMPTE                   CARTE_BANCAIRE

 PK compte_id            PK carte_id
    solde                    FK compte_id(UQ)
```

**SQL :** La FK doit être UNIQUE.

```sql
CREATE TABLE cartes_bancaires (
    carte_id INT PRIMARY KEY,
    compte_id INT UNIQUE NOT NULL,  -- UNIQUE force le 1:1
    FOREIGN KEY (compte_id) REFERENCES comptes(compte_id)
);
```

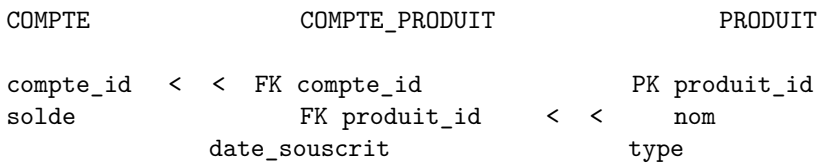
---

**1:N (Un à Plusieurs) — Cas le plus courant** Déjà montré ci-dessus (Client → Comptes).

**Règle :** La FK va dans la table "plusieurs" (côté N).

---

**M:N (Plusieurs à Plusieurs) — Table d'association** **Problème :** Un compte peut avoir plusieurs produits financiers (prêt, assurance), et un produit peut être associé à plusieurs comptes.




**SQL :**

```sql
CREATE TABLE compte_produit (
    compte_id INT NOT NULL,
    produit_id INT NOT NULL,
    date_souscrit DATE NOT NULL,
    PRIMARY KEY (compte_id, produit_id),  -- Clé composite
    FOREIGN KEY (compte_id) REFERENCES comptes(compte_id),
    FOREIGN KEY (produit_id) REFERENCES produits(produit_id)
);
```

---

**Normalisation (Éviter la Redondance)**

**1NF (Première Forme Normale) :** Attributs atomiques (pas de listes).

**Incorrect :**

```
clients (id, nom, telephones)
→ telephones = "123, 456, 789"  (multi-valué)
```

**Correct :**

```
clients (id, nom)
telephones (id, client_id, numero)
```

**2NF :** Chaque attribut non-clé dépend de toute la clé primaire.

**3NF :** Pas de dépendance transitive (un attribut ne dépend pas d'un autre attribut non-clé).

---

**Checklist ERD**

Chaque entité a une clé primaire (PK) Les FK sont clairement identifiées Les cardinalités sont définies Attributs atomiques (1NF) Pas de redondance (normalisation) Contraintes UNIQUE identifiées

---

## 4. SQL Fondamental et Avancé

**DDL (Data Definition Language) — Créer la structure**

```sql
CREATE TABLE clients (
    client_id      INT PRIMARY KEY AUTO_INCREMENT,
```

```
    nom             VARCHAR(100) NOT NULL,
    email           VARCHAR(120) UNIQUE NOT NULL,
    date_inscrip    DATE NOT NULL DEFAULT CURRENT_DATE
);

CREATE TABLE comptes (
    compte_id       INT PRIMARY KEY AUTO_INCREMENT,
    client_id       INT NOT NULL,
    type_compte     VARCHAR(20) NOT NULL CHECK (type_compte IN ('COURANT', 'EPARGNE')),
    solde           DECIMAL(12,2) NOT NULL DEFAULT 0,
    date_ouverture DATE NOT NULL,
    FOREIGN KEY (client_id) REFERENCES clients(client_id) ON DELETE CASCADE
);

CREATE TABLE transactions (
    transaction_id INT PRIMARY KEY AUTO_INCREMENT,
    compte_id       INT NOT NULL,
    type_tx         VARCHAR(10) NOT NULL,
    montant         DECIMAL(12,2) NOT NULL,
    date_tx         TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (compte_id) REFERENCES comptes(compte_id)
);
```

**Contraintes importantes :** - `PRIMARY KEY` : Identifiant unique - `FOREIGN KEY` : Référence à une autre table - `UNIQUE` : Pas de doublons - `NOT NULL` : Valeur obligatoire - `CHECK` : Validation de valeur - `DEFAULT` : Valeur par défaut

---

### DML (Data Manipulation Language) — Manipuler les données

#### INSERT

```
-- Simple
INSERT INTO clients (nom, email, date_inscrip)
VALUES ('Jean Pierre', 'jean@demo.ht', '2024-01-15');

-- Multiple
INSERT INTO comptes (client_id, type_compte, solde, date_ouverture) VALUES
(1, 'COURANT', 1000.00, '2024-01-15'),
(1, 'EPARGNE', 5000.00, '2024-01-20'),
(2, 'COURANT', 500.00, '2024-02-01');
```

---

#### SELECT — Requêtes de base

```
-- Tous les clients
SELECT * FROM clients;

-- Colonnes spécifiques
SELECT nom, email FROM clients;

-- Filtre WHERE
SELECT * FROM comptes WHERE solde > 1000;

-- Tri ORDER BY
SELECT * FROM clients ORDER BY nom ASC;
```

```sql
-- Limite LIMIT
SELECT * FROM transactions ORDER BY date_tx DESC LIMIT 10;
```

---

### UPDATE

```sql
-- Mise à jour simple
UPDATE comptes
SET solde = solde + 100
WHERE compte_id = 5;

-- Mise à jour conditionnelle
UPDATE clients
SET email = 'inconnu@demo.ht'
WHERE email IS NULL;
```

**Attention :** Toujours utiliser `WHERE` sinon toutes les lignes sont modifiées !

---

### DELETE

```sql
-- Suppression avec condition
DELETE FROM transactions WHERE date_tx < '2023-01-01';

-- Suppression complète (dangereux)
DELETE FROM clients;  -- Supprime TOUT

-- Mieux : TRUNCATE (plus rapide)
TRUNCATE TABLE transactions;
```

---

### Jointures (JOINS)

#### INNER JOIN — Seules les correspondances

```sql
-- Comptes avec nom du client
SELECT c.nom, co.type_compte, co.solde
FROM comptes co
INNER JOIN clients c ON c.client_id = co.client_id;
```

**Résultat :** Seulement les clients qui ont des comptes.

---

#### LEFT JOIN — Tous de la table gauche

```sql
-- Tous les clients, même sans compte
SELECT c.nom, co.compte_id, co.solde
FROM clients c
LEFT JOIN comptes co ON co.client_id = c.client_id;
```

**Résultat :** Clients sans compte auront `NULL` pour `compte_id` et `solde`.

---

#### RIGHT JOIN — Tous de la table droite

```sql
-- Tous les comptes, même si client supprimé (rare)
SELECT c.nom, co.compte_id
```

```sql
FROM clients c
RIGHT JOIN comptes co ON co.client_id = c.client_id;
```

---

### Jointures Multiples

```sql
-- Transactions avec nom du client
SELECT c.nom, t.type_tx, t.montant, t.date_tx
FROM transactions t
JOIN comptes co ON co.compte_id = t.compte_id
JOIN clients c ON c.client_id = co.client_id
ORDER BY t.date_tx DESC;
```

---

## Agrégations (GROUP BY)

**Fonctions d'agrégation :** `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`

```sql
-- Nombre de comptes par client
SELECT c.client_id, c.nom, COUNT(co.compte_id) AS nb_comptes
FROM clients c
LEFT JOIN comptes co ON co.client_id = c.client_id
GROUP BY c.client_id, c.nom
ORDER BY nb_comptes DESC;

-- Total des dépôts par compte
SELECT compte_id, SUM(montant) AS total_depots
FROM transactions
WHERE type_tx = 'DEPOT'
GROUP BY compte_id;

-- Solde moyen par type de compte
SELECT type_compte, AVG(solde) AS solde_moyen
FROM comptes
GROUP BY type_compte;
```

**HAVING** — Filtrer après agrégation

```sql
-- Clients avec plus de 2 comptes
SELECT c.nom, COUNT(co.compte_id) AS nb_comptes
FROM clients c
JOIN comptes co ON co.client_id = c.client_id
GROUP BY c.client_id, c.nom
HAVING COUNT(co.compte_id) > 2;
```

**Différence WHERE vs HAVING :** - `WHERE` : Filtre avant agrégation (sur les lignes) - `HAVING` : Filtre après agrégation (sur les groupes)

---

## Sous-requêtes (Subqueries)

### Sous-requête scalaire

```sql
-- Clients avec solde total > 10000
SELECT nom
FROM clients
WHERE client_id IN (
    SELECT client_id
```

```
    FROM comptes
    GROUP BY client_id
    HAVING SUM(solde) > 10000
);
```

**Sous-requête avec EXISTS**

```
-- Clients ayant au moins une transaction
SELECT c.nom
FROM clients c
WHERE EXISTS (
    SELECT 1
    FROM comptes co
    JOIN transactions t ON t.compte_id = co.compte_id
    WHERE co.client_id = c.client_id
);
```

**Performance :** `EXISTS` souvent plus rapide que `IN` pour grandes tables.

---

### Transactions ACID

**Propriétés ACID :** - **Atomicity :** Tout ou rien - **Consistency :** État valide avant et après - **Isolation :** Transactions concurrentes isolées - **Durability :** Modifications persistantes

**Exemple de virement atomique :**

```
START TRANSACTION;

-- Vérification du solde
SELECT solde INTO @solde_source FROM comptes WHERE compte_id = 100 FOR UPDATE;

-- Validation
IF @solde_source >= 500 THEN
    -- Débit
    UPDATE comptes SET solde = solde - 500 WHERE compte_id = 100;

    -- Crédit
    UPDATE comptes SET solde = solde + 500 WHERE compte_id = 200;

    COMMIT;
ELSE
    ROLLBACK;
END IF;
```

**Sans transaction :** Risque d'état incohérent si crash entre débit et crédit.

---

### Index (Performance)

```
-- Créer un index
CREATE INDEX idx_client_email ON clients(email);
CREATE INDEX idx_compte_client ON comptes(client_id);
CREATE INDEX idx_tx_date ON transactions(date_tx);

-- Index composite
CREATE INDEX idx_compte_type_solde ON comptes(type_compte, solde);
```

**Avantages :** Accélère les recherches (WHERE, JOIN). **Inconvénients :** Ralentit INSERT/UPDATE/DELETE, utilise de l'espace.

**Règle :** Indexer les FK, colonnes fréquemment filtrées, colonnes de jointure.

---

**Résumé SQL**

**DDL :** `CREATE`, `ALTER`, `DROP` (structure) **DML :** `SELECT`, `INSERT`, `UPDATE`, `DELETE` (données) **Jointures :** `INNER`, `LEFT`, `RIGHT`, `FULL OUTER` **Agrégations :** `COUNT`, `SUM`, `AVG`, `MIN`, `MAX` + `GROUP BY` + `HAVING` **Transactions :** `START TRANSACTION`, `COMMIT`, `ROLLBACK` **Performance :** Index, éviter `SELECT *`, utiliser EXPLAIN

---

## 5. UML & Conception Orientée Objet

**Principes SOLID**

**S — Single Responsibility Principle** **Règle :** Une classe = une seule responsabilité.

**Incorrect :**

```
class Compte:
    def __init__(self): ...
    def deposer(self): ...
    def retirer(self): ...
    def send_email(self): ...  # Responsabilité en trop
```

**Correct :**

```
class Compte:
    def deposer(self): ...
    def retirer(self): ...

class EmailService:
    def send_email(self, recipient, message): ...
```

---

**O — Open/Closed Principle** **Règle :** Ouvert à l'extension, fermé à la modification.

**Incorrect :** Modifier une classe existante pour ajouter un type de compte.

**Correct :** Créer une sous-classe.

```
class Compte(ABC):
    @abstractmethod
    def calculer_frais(self): pass

class CompteCourant(Compte):
    def calculer_frais(self): return 10.0

class CompteEpargne(Compte):
    def calculer_frais(self): return 0.0
```

---

**L — Liskov Substitution Principle** **Règle :** Les sous-classes doivent être substituables à leur classe parent.

```
def afficher_solde(compte: Compte):
    print(f"Solde: {compte.get_solde()}")

# Doit fonctionner avec CompteCourant ET CompteEpargne
```

```
afficher_solde(CompteCourant(...))
afficher_solde(CompteEpargne(...))
```

---

**I — Interface Segregation Principle** **Règle :** Pas d'interface "fat" — interfaces spécifiques.

**Incorrect :**

```
class Compte(ABC):
    @abstractmethod
    def deposer(self): pass
    @abstractmethod
    def appliquer_interets(self): pass  # Pas pour tous les comptes
```

**Correct :**

```
class Compte(ABC):
    @abstractmethod
    def deposer(self): pass

class CompteAvecInterets(Compte):
    @abstractmethod
    def appliquer_interets(self): pass
```

---

**D — Dependency Inversion Principle** **Règle :** Dépendre d'abstractions, pas de classes concrètes.

**Incorrect :**

```
class Banque:
    def __init__(self):
        self.db = MySQLDatabase()  # Couplage fort
```

**Correct :**

```
class Banque:
    def __init__(self, db: DatabaseInterface):
        self.db = db  # Dépend de l'interface
```

---

**Diagrammes UML Essentiels**

**Diagramme de Classes**

```
      Compte

 - solde: double
 - titulaire: String

 + deposer(m: double)
 + retirer(m: double)


         (Héritage)
```

```
Courant    Epargne
```

**Relations :** - Association - Agrégation (contient, peut exister indépendamment) - Composition (contient, ne peut exister sans) - Implémentation interface - Héritage

---

**Diagramme de Séquence**

```
Client          Banque          CompteA        CompteB

    virement() >
                   getSolde()  >
                  < solde

                   retirer()   >
                  < OK

                   deposer()         >
                  < OK
   <  OK
```

**Usage :** Visualiser les interactions temporelles entre objets.

---

**Encapsulation, Héritage, Polymorphisme**

**Encapsulation**

```python
class Compte:
    def __init__(self):
        self._solde = 0  # Privé (convention Python)

    @property
    def solde(self):
        return self._solde

    def deposer(self, montant):
        if montant > 0:
            self._solde += montant
```

**Avantage :** Contrôle d'accès, validation, traçabilité.

---

**Héritage**

```python
class Compte(ABC):
    def __init__(self, titulaire):
        self.titulaire = titulaire
        self._solde = 0

    @abstractmethod
    def calculer_frais(self): pass

class CompteCourant(Compte):
    def calculer_frais(self): return 10.0
```

```
class CompteEpargne(Compte):
    def calculer_frais(self): return 0.0
```

**Relation "est-un" :** Un CompteCourant **est un** Compte.

---

**Polymorphisme**

```
def appliquer_frais_mensuels(comptes: List[Compte]):
    for compte in comptes:
        frais = compte.calculer_frais()  # Polymorphisme
        compte.retirer(frais)

# Fonctionne avec tous les types de comptes
comptes = [CompteCourant("Alice"), CompteEpargne("Bob")]
appliquer_frais_mensuels(comptes)
```

**Avantage :** Code générique, extensible.

---

**Résumé UML & POO**

**SOLID :** - S : Une responsabilité par classe - O : Extension sans modification - L : Substituabilité des sous-classes - I : Interfaces spécifiques - D : Dépendre d'abstractions

**Diagrammes :** - **Classes :** Structure statique (attributs, méthodes, relations) - **Séquence :** Interactions temporelles - **États-Transitions :** Comportement dynamique d'un objet

**POO :** - **Encapsulation :** Protéger les données - **Héritage :** Réutiliser et spécialiser - **Polymorphisme :** Code générique

---

## 6. Checklist Finale

**Design Patterns**

- ☐ Je connais Singleton, Factory, Builder
- ☐ Je comprends Adapter, Decorator, Facade
- ☐ Je maîtrise Observer et Strategy
- ☐ Je sais quand utiliser quel pattern

**DSA**

- ☐ Je maîtrise Arrays, Hash Tables, Stacks, Queues
- ☐ Je connais Binary Search et Two Pointers
- ☐ Je sais traverser un arbre (BFS, DFS)
- ☐ Je comprends les complexités O(1), O(log n), O(n), $O(n^2)$

**ERD**

- ☐ Je sais dessiner un ERD avec notation Crow's Foot
- ☐ Je maîtrise les cardinalités (1:1, 1:N, M:N)
- ☐ Je comprends les FK et PK
- ☐ Je sais identifier les erreurs de normalisation

**SQL**

- ☐ Je maîtrise SELECT, INSERT, UPDATE, DELETE
- ☐ Je sais faire des JOINS (INNER, LEFT)

☐ Je connais GROUP BY et HAVING
☐ Je comprends les transactions (ACID)

**UML & POO**

☐ Je connais les principes SOLID
☐ Je sais dessiner un diagramme de classes
☐ Je maîtrise encapsulation, héritage, polymorphisme
☐ Je comprends les relations entre classes

---

## Conseils pour l'Entretien/Examen

1. **Clarifiez les exigences** avant de coder
2. **Pensez à voix haute** pour montrer votre raisonnement
3. **Commencez simple** puis optimisez
4. **Testez avec des exemples** (cas limites, valeurs négatives, null)
5. **Discutez les trade-offs** (temps vs espace, simplicité vs performance)
6. **Posez des questions** sur les contraintes (taille des données, fréquence d'accès)

---

## Ressources Complémentaires

- **Design Patterns :** "Head First Design Patterns", "Refactoring Guru"
- **DSA :** LeetCode, HackerRank, "Cracking the Coding Interview"
- **SQL :** SQLBolt, Mode Analytics SQL Tutorial
- **UML :** Lucidchart, draw.io, PlantUML

---

**Dernière révision :** Janvier 2026

**Bon courage pour votre préparation !**