QCM — Architectures logicielles (correction détaillée)

Question 1 — Monolithe

Bonne réponse : B. Une dette technique accrue à cause du couplage fort

Pourquoi ?
Dans un monolithe, les modules partagent souvent le même déploiement, runtime et parfois le même schéma de base. Les dépendances internes deviennent faciles mais **très serrées**, rendant les refactorings risqués et coûteux.

Pourquoi pas les autres ?

- A. Faux : les appels sont **in-process**, pas réseau.
- C. Faux : au contraire, **la cohérence transactionnelle est plus simple** (une seule base/transaction).
- D. Faux : rien n'empêche d'utiliser une base relationnelle.

Question 2 — Architecture n-tiers

Bonne réponse : C. À encapsuler la logique métier et les règles de gestion

Pourquoi ?
La **couche métier (services)** porte les **invariants**, **politiques** et **règles de gestion**. Elle ne dépend pas de l'UI ni du stockage concret : elle **orches tre** des opérations métier et **utilise** la couche d'accès aux données via des abstractions.

Pourquoi pas les autres ?

- A. Faux : c'est le rôle de la **couche données** (repositories/DAO).
- B. Faux : c'est la **couche présentation**.
- D. Faux : la scalabilité est **transversale** et pas propre à la couche métier.

Question 3 — Dépendances en n-tiers

Bonne réponse : A. Présentation → Métier → Données

Pourquoi ?
Le sens classique est **top-down** : l'UI **appelle** la couche métier, qui **appelle** les accès aux données. Cela évite le couplage de l'UI vers l'infrastructure.

Pourquoi pas les autres ?

- B. Faux : les dépendances **bidirectionnelles** créent du couplage cyclique.
- C. Faux : inverser le sens casse l'isolation et n'a pas de sens architectural.
- D. Faux : l'UI ne doit **jamais** parler directement à la base.

Question 4 — Architecture Clean

Bonne réponse : C. Au cœur du domaine, indépendantes de toute technologie

Pourquoi ?
Les **Entities** (modèle riche) sont la **couche la plus interne**. Elles portent les **invariants métier** et **ne dépendent d'aucun framework** ni d'infrastructure. Elles sont donc **faciles à tester** et **pérennes**.

Pourquoi pas les autres ?

- A. Faux : l'infrastructure est **externe**.
- B. Faux : la périphérie (frameworks/drivers) dépend **du domaine**, pas l'inverse.
- D. Faux : les Use Cases orchestrent ; les Entities **n'y résident pas**.

Question 5 — Cas d'usage (Clean)

Bonne réponse : A. Orchestrer une logique métier à partir d'entrées externes

Pourquoi ?
Un **Use Case** coordonne les **Entities** et **Ports** pour réaliser un **scénario métier** (application service). Il **ne fait pas de persistance** lui-même : il **dépend d'abstractions**.

Pourquoi pas les autres ?

- B. Faux : la persistance est un **détail d'implémentation** derrière un port/repository.
- C. Faux : UI/IHM = **adapter entrant**, pas dans l'Use Case.
- D. Faux : la gestion de threads est **technique**, pas applicative.

Question 6 — Architecture hexagonale

Bonne réponse : B. Une dépendance dont le core a besoin (ex. repository, mailer)

Pourquoi ?
Un **port OUT** est une **interface définie par le domaine** qui exprime ce dont **le core a besoin** du monde extérieur (p. ex. `LoadProductPort`, `SendEmailPort`). Des **adapters sortants** (SQLite, SMTP…) **implémentent** ces ports.

Pourquoi pas les autres ?

- A. Faux : ça, c'est plutôt un **port IN** (interface d'entrée).
- C. Faux : un adaptateur de présentation est **entrant**, pas un port OUT.
- D. Faux : une entité n'est **pas un port**.

Question 7 — Adaptateurs (Hexagonale)

Bonne réponse : A. Traduire entre le core et le monde extérieur (infra, UI, DB…)

Pourquoi ?
Les **adapters** sont des **traducteurs** : REST/CLI/GraphQL côté **entrant** ; SQL/SMTP/S3 côté **sortant**. Ils **convertissent** formats et protocoles vers/depuis le langage du domaine.

Pourquoi pas les autres ?

- B. Faux : ils **n'enlèvent** pas de règles métier.
- C. Faux : l'optimisation réseau n'est pas leur **raison d'être**.
- D. Faux : ils ne **remplacent** pas les Entities (confusion DTO/Entity).

Question 8 — Microservices

Bonne réponse : C. Isoler chaque contexte métier avec sa persistance

Pourquoi ?
Chaque microservice est **autonome** : **code + base de données**. On évite la base **partagée** pour préserver l'**indépendance de déploiement** et l'**encapsulation**.

Pourquoi pas les autres ?

- A. Faux : NoSQL n'est pas obligatoire (on peut mixer).
- B. Faux : les microservices sont **découpés par domaine** (bounded context), pas par couches.
- D. Faux : on fait des **tests unitaires** (et contract tests, etc.).

Question 9 — Contrôles et cohérence

Bonne réponse : C. S'appuyer sur des événements/messages (cohérence éventuelle)

Pourquoi ?
La cohérence dans les microservices est **asynchrone** via **events** (pub/sub), **outbox**, **sagas/process managers**. On accepte l'**eventual consistency** pour garder l'autonomie.

Pourquoi pas les autres ?

- A. Faux : une **base centrale** casse l'isolation.
- B. Faux : les **transactions distribuées** synchrones (2PC) nuisent à la **résilience** et à la **scalabilité**.
- D. Faux : on **gère** la cohérence, on ne l'abandonne pas.

Question 10 — Dépendances et inversion de contrôle

Bonne réponse : B. L'inversion de dépendances (Dependency Inversion)

Pourquoi ?
Le **DIP** (principe 'D' de SOLID) impose que les **détails dépendent des abstractions**. Le **domaine** définit des **ports** ; l'infrastructure **implémente** ces ports. C'est la base de Clean/Hexagonal.

Pourquoi pas les autres ?

- A. SRP traite de **responsabilités**, pas des **dépendances**.
- C. Le polymorphisme est une **technique**, pas un **principe d'orientation des dépendances**.
- D. LSP traite de **substituabilité**, pas du **sens des dépendances**.

Résumé "à retenir"

- **Séparer** strictement **métier** (Entities/Use Cases) et **technique** (adapters/infrastructure).
- **Ports** = **interfaces** définies **par le domaine** ; **adapters** = implémentations techniques.
- **Dépendances entrantes** vers le **domaine**, jamais l'inverse.
- **Microservices** = **autonomie** (code + base) + **cohérence éventuelle** via événements.