L’Architecture Logicielle :

Une présentation complète

Formation : Master en Informatique

Étudiant(e) : Nom Complet

Directeur(rice) : Prof. Dr. Nom Complet

Paris, 22 décembre 2025

Sommaire

Pourquoi cette présentation ?

Imaginez que vous avez construit une maison. Jusqu'à maintenant, nous avons parlé de :

- **L'architecture applicative** : c'est comme le plan des pièces de votre maison (cuisine, salon, chambres...)
- **L'architecture technique** : ce sont les matériaux (briques, béton, tuyaux, électricité...)

Maintenant

Nous allons découvrir **ce qui se passe À L'INTÉRIEUR de chaque pièce** !

Qu’est-ce que la « Boîte Blanche » ?

Boîte Noire

- Vue de l’extérieur
- On ne voit pas comment ça fonctionne
- On voit juste que ça marche

Boîte Blanche

- Vue de l’intérieur
- On découvre les composants
- On comprend l’organisation
- On voit les communications

Analogie : C’est comme ouvrir le capot d’une voiture pour voir le moteur !

Le Couplage Fort entre Architecture Logicielle et Technique

Important

Les architectures logicielles et techniques sont en « **couplage fort** ».

Qu'est-ce que ça veut dire ?

- Imaginez deux danseurs qui dansent un tango :
  - Si l'un fait un pas à gauche, l'autre DOIT suivre
  - Les choix de l'un influencent directement l'autre
  - Ils sont **couplés** : ils doivent bouger ensemble !

De la même façon :

- Les choix pour l'architecture **logicielle** (comment organiser le code)
- Influencent directement les choix pour l'architecture **technique** (serveurs, base de données...)
- Et vice-versa !

Les 3 Contraintes Essentielles

En plus de faire fonctionner l'application, l'architecture logicielle doit garantir :

1. **Sécurité**
   - Comme mettre une serrure sur votre porte
   - Empêcher les méchants d'entrer
   - Protéger vos données

2. **Disponibilité**
   - Comme avoir de l'électricité 24h/24
   - L'application doit fonctionner tout le temps

3. **Performance**
   - Comme avoir une voiture rapide
   - Répondre vite, même avec beaucoup d'utilisateurs

Définition Simple

L'Architecture Logicielle, c'est :

La façon dont on organise et on structure le code d'une application pour qu'elle fonctionne bien

C'est comme un plan de construction qui explique :

- Comment découper l'application en morceaux
- Comment ces morceaux communiquent entre eux
- Quelles règles ils doivent respecter

Les Deux Rôles de l'Architecture Logicielle

Rôle n°1 : Structurer

Elle organise les solutions **les mieux adaptées** pour répondre aux besoins de l'utilisateur.

Exemple : Pour une application e-commerce :

- Où mettre le code du panier ?
- Où mettre le calcul des prix ?
- Où mettre l'affichage ?

Rôle n°2 : Décomposer

Elle découpe l'application en petits morceaux faciles à comprendre et à modifier.

Analogie : Comme un puzzle géant !

- Chaque pièce = un morceau de code
- L'architecture = les règles d'assemblage

Les 5 Concepts Clés de Décomposition (1/3)

1. Les Couches (Layers)

- Comme un gâteau à étages
- Chaque étage = une couche avec un rôle précis
- **Exemple :** Couche interface → Couche logique → Couche données

2. Les Modules

- Un gros dossier qui regroupe des fonctionnalités liées
- **Exemple :** Module « Gestion des Clients », Module « Gestion des Commandes »

Les 5 Concepts Clés de Décomposition (2/3)

3. Les Composants

- Un morceau de code réutilisable, comme une brique de LEGO
- **Exemple :** Un bouton « Valider », un formulaire de connexion

4. Les Design Patterns

- Une recette de cuisine pour résoudre un problème courant
- **Exemple :**
  - Pattern « Singleton » : une seule connexion à la BD
  - Pattern « Observer » : prévenir tous les intéressés quand quelque chose change

Les 5 Concepts Clés de Décomposition (3/3)

5. Les Frameworks

- Une boîte à outils toute prête avec plein de fonctions déjà codées
- **Exemples :**
  - React (pour faire des interfaces web)
  - Spring (pour faire des applications Java)
  - Django (pour faire des applications Python)

Pourquoi C'est Important ?

Une bonne architecture permet :

- ✓ **Comprendre** le code facilement
- ✓ **Modifier** sans tout casser
- ✓ **Réutiliser** des morceaux
- ✓ **Travailler en équipe**
- ✓ **Éviter les bugs**
- ✓ **Améliorer les performances**

Une mauvaise architecture

= un **code spaghetti**

- × Tout est mélangé
- × Impossible de s'y retrouver
- × Ajouter une fonctionnalité = tout casse !

Comment Créer une Architecture Logicielle ?

Principe

Créer une architecture logicielle, c'est comme construire une maison : on ne commence pas par le toit !

Il faut suivre **4 étapes**, et on les répète plusieurs fois (**de manière itérative et incrémentale**).

C'est quoi « itérative et incrémentale » ?

- **Itérative** = on répète les étapes plusieurs fois, en améliorant à chaque fois
- **Incrémentale** = on ajoute des morceaux petit à petit, comme des couches de peinture

ÉTAPE 1 : Analyser les Besoins

But : Comprendre ce que l'application doit faire.

Questions à se poser :

- Qui va utiliser l'application ? (des clients ? des employés ?)
- Qu'est-ce que l'application doit faire ? (vendre des produits ? gérer des stocks ?)
- Combien de personnes vont l'utiliser ? (10 ? 1000 ? 1 million ?)
- Quelles sont les contraintes ? (budget ? délai ? sécurité ?)

Exemple concret : Application e-commerce

- Les clients doivent pouvoir chercher des produits
- Les clients doivent pouvoir payer en ligne
- L'application doit supporter 10 000 clients simultanés
- Les paiements doivent être ultra-sécurisés

ÉTAPE 2 : Définir la Structure

But : Choisir comment organiser l'application.

Décisions à prendre :

1. **Quel type d'architecture choisir ?**
   - Monolithique ? (tout en un seul bloc)
   - Microservices ? (plein de petits services indépendants)
   - Client-Serveur ? (un client qui parle à un serveur)
2. **Combien de couches ?**
   - 3 couches ? (Présentation, Logique, Données)
   - 5 couches ? (encore plus de séparation)
3. **Quels frameworks utiliser ?**
   - React pour l'interface ?
   - Spring Boot pour le backend ?

ÉTAPE 2 : Définir la Structure (Analogie)

Analogie

C’est comme décider si votre maison sera :

- Une maison individuelle (monolithique)
- Un immeuble avec plusieurs appartements (microservices)
- Le nombre d’étages (les couches)

ÉTAPE 3 : Concevoir les Composants (1/2)

But : Créer les briques de base de l'application.

Ce qu'on fait :

1. **Identifier les composants nécessaires**
   - Composant « Liste des produits »
   - Composant « Panier d'achat »
   - Composant « Formulaire de paiement »
2. **Définir comment ils communiquent**
   - Le panier envoie les infos au paiement
   - Le paiement vérifie avec la banque
   - La banque renvoie OK ou KO

ÉTAPE 3 : Concevoir les Composants (2/2)

3. **Choisir les design patterns**
   - Pattern MVC pour séparer l'affichage de la logique
   - Pattern Repository pour accéder aux données

Analogie

C'est comme décider :

- Quels meubles mettre dans chaque pièce
- Où placer les prises électriques
- Comment les pièces sont connectées

ÉTAPE 4 : Valider et Ajuster

But : Vérifier que l'architecture fonctionne bien.

Ce qu'on vérifie :

1. **La qualité**
   - Est-ce que le code est propre et bien organisé ?
   - Est-ce qu'on peut facilement ajouter de nouvelles fonctionnalités ?
2. **Les performances**
   - Est-ce que l'application est rapide ?
   - Est-ce qu'elle peut gérer beaucoup d'utilisateurs ?
3. **La sécurité**
   - Est-ce que les données sont protégées ?
   - Est-ce qu'on peut pirater l'application ?

Si ça ne va pas

On retourne à l'étape 2 ou 3 et on améliore !

Pourquoi « Itératif et Incrémental » ?

Parce qu'on ne peut pas tout prévoir !

Au début du projet, on ne sait pas tout. Donc :

- On fait une première version simple (Itération 1)
- On la teste
- On l'améliore (Itération 2)
- On ajoute des fonctionnalités (Incrémental)
- On teste encore
- Et ainsi de suite...

Exemple

- **Itération 1** : Afficher une liste de produits
- **Itération 2** : Ajouter un panier
- **Itération 3** : Ajouter le paiement
- **Itération 4** : Ajouter la gestion des comptes clients

Pourquoi 2 Vues ?

Pour comprendre une architecture, on a besoin de 2 points de vue différents, comme pour une maison :

Vue d'architecte

Le plan dessiné sur papier

→ **Vue logique**

Vue du chantier

Les vraies machines et matériaux

→ **Vue physique**

Les deux sont nécessaires !

VUE N°1 : La Vue en Couches (Layer View) - Vue LOGIQUE

C'est quoi ?
La vue en couches montre comment l'application est organisée **conceptuellement**, dans notre tête.

Comme un sandwich

- Le pain du haut = l'interface utilisateur
- La salade = la logique métier
- La viande = les services
- Le pain du bas = la base de données

Les Couches Typiques (1/3)

Couche 1 : Présentation / Interface Utilisateur (UI)

- **Rôle** : Ce que l'utilisateur voit et touche
- **Exemples** : Boutons, formulaires, menus, pages web
- **Technologies** : HTML, CSS, JavaScript, React, Angular
- **Analogie** : La vitrine d'un magasin

Couche 2 : Contrôleur / Logique de Présentation

- **Rôle** : Recevoir les demandes et organiser la réponse
- **Exemple** : Quand vous cliquez sur « Ajouter au panier », c'est le contrôleur qui reçoit la demande
- **Analogie** : Le chef de salle dans un restaurant (il prend votre commande et la transmet à la cuisine)

Les Couches Typiques (2/3)

Couche 3 : Logique Métier / Domain

- **Rôle** : Les règles du métier, les calculs, la vraie intelligence de l'application
- **Exemples** :
  - Calculer le prix total (produits + TVA + frais de port)
  - Vérifier si un produit est en stock
  - Appliquer une réduction si le client est VIP
- **Analogie** : Les cuisiniers qui préparent le plat selon la recette

Couche 4 : Services / Logique d'Application

- **Rôle** : Coordonner plusieurs opérations métier
- **Exemple** : Orchestrer le processus de commande complet
- **Analogie** : Le chef de cuisine qui coordonne tous les cuisiniers

Les Couches Typiques (3/3)

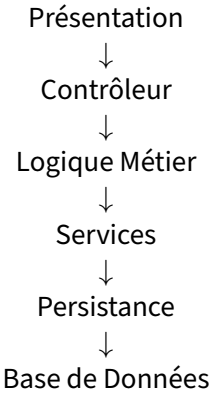
Couche 5 : Persistance / Accès aux Données

- **Rôle** : Parler à la base de données
- **Exemples** :
  - Sauvegarder une commande
  - Récupérer la liste des clients
  - Modifier un produit
  - Supprimer un compte
- **Technologies** : SQL, Hibernate, JPA, Repository Pattern
- **Analogie** : Le garde-manger où on stocke tous les ingrédients

Règles de Communication entre Couches

Règle d'Or

On communique **uniquement avec la couche d'en dessous** (comme une chaîne)



Règles de Communication : Pourquoi ?

Pourquoi cette règle ?

- Pour éviter le désordre (code spaghetti)
- Pour pouvoir changer une couche sans casser les autres
- Pour que le code reste propre et organisé

Mauvaise pratique

L'interface utilisateur qui parle directement à la base de données

→ Si on change la BD, il faut changer l'interface !

Bonne pratique

Interface → Contrôleur → Logique → Persistance → BD

→ Chaque couche a son rôle bien défini !

VUE N°2 : La Vue en Niveaux (Tier View) - Vue PHYSIQUE

C'est quoi ?
La vue en niveaux montre où les morceaux de l'application sont **physiquement installés** (sur quels serveurs, ordinateurs, etc.)

La différence avec les couches

- **Couches (Layers)** = organisation logique du code (dans notre tête)
- **Niveaux (Tiers)** = organisation physique des machines (dans la vraie vie)

Architecture 1-Tier (Monolithique)

Tout est sur la même machine !

- Interface utilisateur
- Logique métier
- Base de données

Exemple : Une vieille application Windows qu'on installe sur son PC.

Avantages :

- ✓ Simple
- ✓ Rapide (tout est local)

Inconvénients :

- × Si l'ordinateur plante, tout est perdu
- × Impossible de partager
- × Difficile à mettre à jour

Architecture 2-Tiers (Client-Serveur)

Séparation entre le client et le serveur

teur Client Interface utilisateur → Logique métier
Serveur (affichage) ← Base de données

Exemple : Une application bancaire

Avantages :

- ✓ Données centralisées
- ✓ Plusieurs utilisateurs
- ✓ Plus sécurisé

Inconvénients :

- × Si le serveur plante, tout le monde est bloqué
- × Le serveur peut être surchargé

Architecture 3-Tiers (Classique Web)

Séparation en 3 machines

[Client] → [Serveur Web] → [Serveur de Données]
Navigateur Logique métier Base de données

Exemple : Un site web e-commerce

Avantages :

- ✓ Chaque niveau peut être amélioré indépendamment
- ✓ Meilleure performance
- ✓ Plus sécurisé

Inconvénients :

- × Plus complexe à installer et maintenir
- × Plus cher (3 machines au lieu d'une)

Architecture N-Tiers (Moderne)

Séparation en plein de machines

Exemple : Netflix

- Votre TV/PC = l'interface
- Des milliers de serveurs web répartis dans le monde
- Des serveurs de cache (pour que la vidéo soit rapide)
- Des bases de données énormes

Avantages :

- ✓ Très performant
- ✓ Très disponible
- ✓ Scalable

Inconvénients :

- × Très complexe
- × Très cher
- × Nécessite des experts

Relation entre Couches et Niveaux

Important à comprendre

- Les **couches** peuvent être sur le **même niveau** (même machine)
- Ou les **couches** peuvent être sur des **niveaux différents** (machines différentes)

Exemple 1 : Monolithique

- Toutes les couches sur 1 seul niveau (1 machine)

Exemple 2 : Architecture 3-Tiers typique

- Niveau 1 (Client) = Couche Présentation
- Niveau 2 (Serveur Web) = Couches Contrôleur + Logique Métier + Services
- Niveau 3 (Serveur BD) = Couche Persistance + Base de Données

Tableau Comparatif

Aspect	Couches (Layers)	Niveaux (Tiers)
Nature	Logique / Conceptuelle	Physique / Matérielle
Question	Comment organiser le code ?	Où installer les composants ?
Exemple	Couche Présentation	Serveur Web
Communication	Verticale (haut en bas)	Horizontale (réseau)
But	Organiser le code	Distribuer la charge
Diagramme UML	Diagramme de packages	Diagramme de déploiement

Les Principes Fondamentaux

Pour créer une bonne architecture, il faut respecter certains principes.

Pensez à eux comme les **règles d'or** de l'architecture logicielle !

Nous allons voir :

1. L'Abstraction
2. L'Encapsulation
3. La Séparation des Responsabilités
4. Le Faible Couplage
5. La Forte Cohésion
6. La Modularité
7. L'Extensibilité
8. La Réutilisabilité

Principe 1 : L'Abstraction

C'est quoi ? Cacher les détails compliqués et ne montrer que l'essentiel.

Analogie

Quand vous conduisez une voiture :

- Vous appuyez sur l'accélérateur → la voiture avance
- Vous ne savez pas comment le moteur fonctionne (et vous n'avez pas besoin de le savoir !)

Dans le code :

Interface simple (abstraction)

```
envoyer_email(destinataire, message)
```

On ne voit pas les détails : connexion SMTP, encodage, gestion des erreurs...

Pourquoi c'est important ? Plus facile à utiliser, on peut changer l'implémentation, moins de bugs

Principe 2 : L'Encapsulation

C'est quoi ? Mettre les données et les fonctions qui les manipulent dans une « boîte fermée ».

Analogie

Une télécommande de TV :

- À l'intérieur : des circuits compliqués
- À l'extérieur : juste des boutons simples
- Vous ne pouvez pas toucher directement les circuits (c'est protégé !)

Dans le code :

- Données PRIVÉES (protégées) : `solde = 1000€`
- Fonctions PUBLIQUES (accessibles) : `deposer(montant)`, `retirer(montant)`

Pourquoi c'est important ? Les données sont protégées, on contrôle les modifications, impossible de mettre un solde négatif !

Principe 3 : La Séparation des Responsabilités

C'est quoi ? Chaque morceau de code doit avoir **une seule responsabilité** bien définie.

Analogie

Dans une cuisine de restaurant :

- Le chef cuisine
- Le serveur sert
- Le plongeur fait la vaisselle
- Chacun fait SON travail, pas celui des autres !

Mauvais exemple

Une fonction qui :

- Affiche à l'écran
- Calcule le prix
- Sauvegarde en BD

Bon exemple

4 fonctions séparées :

- `afficher_progression()`
- `calculer_prix()`
- `sauvegarder_commande()`

Principe 4 : Le Faible Couplage

C'est quoi ? Les morceaux de code doivent être **le moins dépendants possible** les uns des autres.

Analogie

- **Fort couplage** : Des jumeaux siamois (si l'un bouge, l'autre doit bouger)
- **Faible couplage** : Deux amis (chacun peut faire sa vie de son côté)

Mauvais (fort couplage)

Code qui appelle directement MySQL

[×] Si on change de BD, il faut tout changer !

Bon (faible couplage)

Code qui passe par une abstraction (Repository)

[✓] On peut changer de BD facilement !

Principe 5 : La Forte Cohésion

C'est quoi ? Les éléments d'un même module doivent être **fortement liés** entre eux.

Analogie

Dans une boîte à outils :

- **Forte cohésion** : Toutes les clés ensemble, tous les tournevis ensemble
- **Faible cohésion** : Un marteau avec une brosse à dents et un livre de cuisine (ça n'a aucun sens !)

Bon exemple

Module GestionDesClients :

- `creer_client()`
- `modifier_client()`
- `supprimer_client()`
- `trouver_client()`

Mauvais exemple

Module Utilitaires :

- `calculer_prix()`
- `envoyer_email()`
- `afficher_menu()`
- `convertir_date()`

Principes 6, 7 et 8

6. La Modularité

- Diviser l'application en **modules indépendants**
- Comme des pièces de LEGO : on peut les assembler de différentes façons
- Plus facile à tester, réutiliser, maintenir

7. L'Extensibilité (Open/Closed Principle)

- Le code doit être **ouvert** à l'extension (on peut ajouter des fonctionnalités)
- Le code doit être **fermé** à la modification (on ne touche pas au code existant)

8. La Réutilisabilité

- Écrire du code qu'on peut **utiliser à plusieurs endroits**
- Comme une recette de pâte à crêpes : utilisable pour crêpes sucrées, salées, galettes...

Les Patterns Architecturaux

Définition

Les patterns architecturaux sont des **modèles de solutions** pour des problèmes courants.

Nous allons voir :

1. Pattern MVC (Modèle-Vue-Contrôleur)
2. Pattern Repository
3. Pattern Singleton
4. Pattern Observer

Pattern MVC (Modèle-Vue-Contrôleur)

Le Problème : Comment séparer l'affichage de la logique métier ?

La Solution MVC :

$$[\text{VUE}] \leftrightarrow [\text{CONTRÔLEUR}] \leftrightarrow [\text{MODÈLE}]$$

- **MODÈLE** = Les données et la logique métier (ex : classe `Produit`)
- **VUE** = L'affichage (ex : page HTML qui affiche les produits)
- **CONTRÔLEUR** = Le chef d'orchestre qui coordonne

Pourquoi c'est bien ?

- ✓ On peut changer l'affichage sans toucher à la logique
- ✓ On peut changer la logique sans toucher à l'affichage
- ✓ Facile à tester

Pattern Repository

Le Problème : Comment accéder aux données sans dépendre de la base de données ?

La Solution : Créer une « couche intermédiaire » qui fait le lien.

[Code Métier] → [Repository] → [Base de Données]

Exemple :

- Interface : `ClientRepository` avec `trouver_par_id()`, `sauvegarder()`, `supprimer()`
- Implémentation : `ClientRepositoryMySQL` avec code SQL spécifique

Pourquoi c'est bien ?

- ✓ Le code métier ne connaît pas la BD utilisée
- ✓ On peut changer de BD (MySQL → PostgreSQL) facilement
- ✓ Plus facile à tester (FakeRepository)

Pattern Singleton

Le Problème : Comment s'assurer qu'on a **qu'une seule instance** d'un objet ?

Exemple concret : Une connexion à la base de données : on en veut qu'une seule !

La Solution :

- Variable privée qui stocke l'unique instance
- Constructeur privé (on ne peut pas faire `new ConnexionBD()`)
- Fonction statique `obtenir_instance()` qui retourne toujours la même instance

Utilisation :

```
connexion1 = ConnexionBD.obtenir_instance()
connexion2 = ConnexionBD.obtenir_instance()
```

→ connexion1 et connexion2 sont LA MÊME connexion !

Pattern Observer

Le Problème : Comment notifier plusieurs objets quand quelque chose change ?

Exemple concret : Quand un produit arrive en stock, il faut :

- Envoyer un email aux clients intéressés
- Mettre à jour l'affichage du site
- Enregistrer dans les logs

La Solution :

- Le produit a une liste d'observateurs
- Quand le stock change, on notifie tous les observateurs
- Chaque observateur réagit à sa manière

Avantage : Les observateurs ne se connaissent pas entre eux !

Les Qualités d'une Bonne Architecture

Qualité	Description
Maintenabilité	Facile à corriger et à améliorer
Testabilité	Facile à tester
Évolutivité	Facile d'ajouter des fonctionnalités
Scalabilité	Peut gérer plus d'utilisateurs
Performance	Rapide et efficace
Sécurité	Protégé contre les attaques
Simplicité	Facile à comprendre
Réutilisabilité	Les morceaux sont réutilisables

Introduction

Il existe de nombreux **styles d'architecture logicielle**. Chacun a ses avantages et ses inconvénients.

C'est comme les styles de maison

- Maison individuelle
- Appartement
- Immeuble
- Château

Chaque style est adapté à des besoins différents !

Architecture Monolithique Traditionnelle

C'est quoi ? Une application où **tout est dans un seul gros bloc**, un seul programme.

Caractéristiques :

- Un seul code source
- Un seul exécutable
- Un seul déploiement
- Tout est fortement couplé

Avantages :

- ✓ Simple à développer
- ✓ Simple à déployer
- ✓ Rapide
- ✓ Facile à tester

Inconvénients :

- × Code spaghetti
- × Difficile à comprendre
- × Difficile à modifier
- × Impossible de scaler

Quand l'utiliser ? Petite application, équipe réduite, prototype

Architecture Monolithique Modulaire

C'est quoi ? Comme le monolithique, mais **bien organisé en modules indépendants**.

Analogie

Un immeuble avec plusieurs appartements : tout est dans le même bâtiment, mais chaque appartement est indépendant.

Caractéristiques :

- Un seul déploiement (comme le monolithique)
- Mais divisé en modules avec frontières claires
- Faible couplage entre modules
- Chaque module a sa responsabilité

Avantages : Bien organisé, facile à comprendre, facile à tester, équipes autonomes

Quand l'utiliser ? Application moyenne à grande, équipe moyenne, migration vers microservices

Architecture Client-Serveur

C'est quoi ? L'application est divisée en deux parties :

- Le **Client** : l'interface utilisateur
- Le **Serveur** : la logique et les données

Analogie

Un restaurant :

- Le client = vous, à votre table
- Le serveur (de restaurant) = prend vos commandes et les apporte
- La cuisine = prépare les plats

Avantages : Données centralisées, plusieurs clients, mise à jour facile, sécurité

Inconvénients : Dépendance réseau, serveur = point unique de défaillance

Quand l'utiliser ? Applications d'entreprise (ERP, CRM), jeux en réseau, applications bancaires

Architecture N-Tiers

C'est quoi ? L'application est divisée en **plusieurs niveaux physiques**, chacun sur des machines différentes.

Architecture 3-Tiers classique :

- **Niveau 1 : Présentation** (Navigateur Web) : HTML/CSS/JS, Interface UI
- **Niveau 2 : Application** (Serveur Web/API) : Logique, Services, API REST
- **Niveau 3 : Données** (Serveur BD) : MySQL/PostgreSQL, Stockage

Avantages : Scalabilité horizontale, indépendance, sécurité, spécialisation

Inconvénients : Complexité, coût, latence

Quand l'utiliser ? Applications web, beaucoup d'utilisateurs, besoin de performance

Architecture en Couches (Layered)

C'est quoi ? Organisation **logique** du code en couches empilées.

Différence avec N-Tiers

- **N-Tiers** = physique (machines différentes)
- **En Couches** = logique (dans le code)

Couches typiques :

1. Couche Présentation (Controllers, UI)
2. Couche Application (Use Cases, Workflows)
3. Couche Métier / Domain (Logique métier, Entities)
4. Couche Services (Services externes, APIs)
5. Couche Persistance (Repository, DAO)

Quand l'utiliser ? La plupart des applications, combiné avec d'autres patterns

Architecture Microservices (1/2)

C'est quoi ? L'application est **découpée en plein de petits services indépendants**.

Chaque service :

- Fait **une seule chose**
- A sa **propre base de données**
- Peut être **déployé indépendamment**
- Communique avec les autres via **des APIs**

Analogie

Une ville avec plein de commerces indépendants :

- La boulangerie vend du pain
- La pharmacie vend des médicaments
- La banque gère l'argent

Chaque commerce est indépendant, mais ils travaillent ensemble !

Architecture Microservices (2/2)

Avantages :

- ✓ Scalabilité
- ✓ Flexibilité technologique
- ✓ Équipes autonomes
- ✓ Déploiement indépendant
- ✓ Résilience
- ✓ Innovation

Inconvénients :

- × Très complexe
- × Coût élevé
- × Latence réseau
- × Cohérence des données
- × Tests complexes
- × Monitoring sophistiqué

Quand l'utiliser ? Grandes applications (Netflix, Amazon, Uber), grandes équipes (50+ développeurs), millions d'utilisateurs

Architecture Orientée Services (SOA)

C'est quoi ? Comme les microservices, mais **à l'échelle de l'entreprise**, avec un **bus de services** central.

Différence avec Microservices

- **SOA** : Communication via un **ESB** (Enterprise Service Bus) - un bus central
- **Microservices** : Communication directe entre services

Avantages : Réutilisation, interopérabilité, centralisation, standards

Inconvénients : ESB = point unique de défaillance, complexe, lourd, coût

Quand l'utiliser ? Grandes entreprises avec beaucoup de systèmes existants, besoin d'intégration, environnements legacy

Architecture Hexagonale (Ports & Adapters)

C'est quoi ? Mettre la **logique métier au centre**, et l'isoler de tout le reste (BD, interfaces, APIs externes).

Le principe :

- Au **centre** : le domaine métier (la vraie valeur de l'application)
- Autour : des **ports** (interfaces)
- Encore autour : des **adaptateurs** (implémentations concrètes)

Analogie

Une prise électrique :

- Le **port** = la prise dans le mur (interface standardisée)
- Les **adaptateurs** = les différents chargeurs qu'on peut brancher
- Le **centre** (métier) = l'appareil qu'on veut charger

Quand l'utiliser ? Applications avec logique métier complexe, besoin de testabilité maximale, Domain-Driven Design (DDD)

Architecture Event-Driven (Orientée Événements)

C'est quoi ? Les composants communiquent en s'envoyant des **événements** (messages).

Principe :

- Quelque chose se passe → Un événement est émis
- Les composants intéressés **réagissent** à l'événement

Analogie

Un système d'alarme :

- **Événement** : « Quelqu'un a ouvert la porte »
- **Réactions** : La sirène sonne, SMS envoyé, police appelée, vidéo enregistrée

Tous ces systèmes réagissent à l'événement, mais ils ne se connaissent pas entre eux !

Avantages : Découplage total, scalabilité, asynchrone, résilience, flexibilité

Quand l'utiliser ? Systèmes temps réel, IoT, e-commerce, systèmes complexes

Architecture MVC

Les 3 composants :

1. **MODÈLE** : Les données, la logique métier, les règles de validation
2. **VUE** : L'interface utilisateur, l'affichage, les formulaires
3. **CONTRÔLEUR** : Reçoit les demandes, interroge le modèle, choisit la vue

Les 2 Variantes :

- **MVC Push (Action-Based)** : Le contrôleur **pousse** les données vers la vue (ex : Spring MVC)
- **MVC Pull (Component-Based)** : La vue **tire** les données du modèle (ex : JSF)

Quand l'utiliser ? Applications web traditionnelles, applications desktop, frameworks (Spring MVC, Django)

Architecture Domain-Driven Design (DDD)

C'est quoi ? Une approche qui met le **domaine métier** au cœur de l'architecture.

Concept clé : Bounded Context

- Un « Bounded Context » = un **contexte métier** bien délimité
- Chaque contexte a son propre **modèle** et son propre **langage**

Concepts DDD :

1. **Entities** : Objets avec identité unique (ex : Client avec ID)
2. **Value Objects** : Objets sans identité (ex : Adresse)
3. **Aggregates** : Groupe d'objets traités comme une unité (ex : Commande + Lignes)
4. **Repositories** : Accès aux données (ex : ClientRepository)
5. **Services** : Opérations qui ne rentrent pas dans les entités (ex : CalculateurDePrix)

Quand l'utiliser ? Domaine métier complexe, grandes applications d'entreprise, collaboration avec experts métier

Tableau Comparatif des Architectures

Architecture	Complex.	Scala.	Maint.	Coût	Quand l'utiliser ?
Monolithique	Faible	Faible	Faible	Faible	Petites applis
Monolith. Modulaire	Moyenne	Moyenne	Bonne	Faible	Applis moyennes
Client-Serveur	Moyenne	Faible	Moyenne	Moyen	Apps entreprise
N-Tiers	Moyenne	Bonne	Moyenne	Moyen	Sites web
En Couches	Faible	Moyenne	Bonne	Faible	Plupart applis
Microservices	Élevée	Excell.	Moyenne	Élevé	Grandes applis
SOA	Élevée	Bonne	Faible	Élevé	Grandes entr.
Hexagonale	Moyenne	Moyenne	Excell.	Moyen	Apps complexes
Event-Driven	Élevée	Excell.	Moyenne	Moyen	Temps réel, IoT
MVC	Faible	Moyenne	Bonne	Faible	Apps web class.
DDD	Élevée	Moyenne	Bonne	Élevé	Domaine complexe

Pourquoi Cartographier ?

Cartographier une architecture = créer des **plans** et des **schémas** pour :

- **Comprendre** l'existant
- **Communiquer** avec les équipes
- **Documenter** pour le futur
- **Décider** des évolutions

Analogie

Comme un plan de ville :

- Pour comprendre où sont les rues
- Pour expliquer un trajet
- Pour planifier de nouvelles routes

Les 3 Vues d'Architecture (selon UML)

UML (Unified Modeling Language) propose 3 vues pour cartographier une architecture :

1. **VUE LOGIQUE (Logical View)**
   - **But** : Montrer l'organisation **conceptuelle** du système
   - **Diagramme UML** : Diagramme de Packages

2. **VUE D'IMPLÉMENTATION (Implementation View)**
   - **But** : Montrer les **composants physiques** et leurs connexions
   - **Diagramme UML** : Diagramme de Composants

3. **VUE DE DÉPLOIEMENT (Deployment View)**
   - **But** : Montrer où les composants sont **physiquement installés**
   - **Diagramme UML** : Diagramme de Déploiement

Diagramme de Packages (Vue Logique)

C'est quoi ? Un package = un dossier qui regroupe des éléments liés.

Exemple : Application E-commerce

- Package **Clients** : Service + Repository
- Package **Commandes** : Service + Repository (dépend de Clients et Produits)
- Package **Produits** : Service + Repository
- Package **Paiements** : Service + Repository (dépend de Clients)
- Package **Infrastructure** : Database, Email, Logging

Signification

- Le package « Commandes » dépend de « Clients » (une commande a un client)
- Le package « Commandes » dépend de « Produits » (une commande contient des produits)
- Tous dépendent de « Infrastructure » (pour accéder à la BD, envoyer des emails...)

Diagramme de Composants (Vue d'Implémentation)

Exemple : Architecture Web 3-Tiers

Couche Présentation :

- Frontend (React) : Components + Pages

Couche Application : (via HTTP/REST)

- Backend API (Spring Boot) : Controllers + Services

Couche Données : (via JDBC)

- Database (PostgreSQL) : Tables + Views

Composants externes :

- Stripe API (Paiements)
- SendGrid API (Emails)

Diagramme de Déploiement (Vue Physique)

Exemple : Déploiement Cloud (AWS)

Internet → Load Balancer (AWS ELB)

3 Serveurs Web :

- EC2 Instance 1, 2, 3 : app.jar (Spring Boot)

Base de données :

- RDS Database (PostgreSQL) : ecommerce_db

Autres nœuds :

- S3 Bucket (Stockage de fichiers)
- CloudFront (CDN pour les images)

Diagrammes Complémentaires

Diagramme de Classes (pour la vue détaillée)

- Pour zoomer dans un module et voir les classes
- **Exemple** : Classe `Commande` avec attributs (id, date, client, statut) et méthodes (calculerTotal, valider, annuler)
- Relation avec classe `LigneCommande`

Diagramme de Séquence (pour les interactions)

- Pour montrer comment les composants interagissent dans le temps
- **Exemple** : Processus « Passer une commande » :
  - Utilisateur → Interface → Contrôleur → ServiceCommande → Repository → BD
  - Puis retour avec confirmation

Outils de Cartographie

Outils de Modélisation UML :

- Enterprise Architect (professionnel, payant)
- Visual Paradigm (complet, payant)
- StarUML (gratuit, open source)
- PlantUML (code as diagram, gratuit)
- Draw.io / diagrams.net (gratuit, en ligne)
- Lucidchart (en ligne, freemium)

Outils spécialisés Architecture :

- Archimate (pour l'architecture d'entreprise)
- C4 Model (pour l'architecture logicielle moderne)
- Structurizr (architecture as code)

Le Modèle C4 (Moderne et Simplifié)

Le **modèle C4** est une approche moderne pour cartographier les architectures.

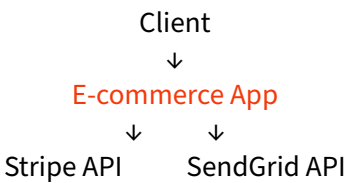
Les 4 Niveaux du C4 :

1. **Context (Contexte)** : Vue d'ensemble - qui utilise le système et avec quoi il interagit
2. **Containers (Conteneurs)** : Les applications et les bases de données
3. **Components (Composants)** : Les modules à l'intérieur d'un conteneur
4. **Code** : Les classes et leurs relations (diagrammes de classes UML)

Modèle C4 : Niveau 1 - Context

Vue d'ensemble : qui utilise le système et avec quoi il interagit.

Exemple E-commerce :



Modèle C4 : Niveau 2 - Containers

Les applications et les bases de données

Exemple E-commerce :

React App (Frontend)
↓ HTTP
Spring Boot API (Backend)
↓ JDBC
PostgreSQL DB

Modèle C4 : Niveau 3 - Components

Les modules à l’intérieur d’un conteneur

Exemple : Spring Boot API

Controllers
↓
Services
↓
Repositories

Modèle C4 : Niveau 4 - Code

Les classes et leurs relations

C’est le niveau le plus détaillé : on utilise les diagrammes de classes UML classiques.

Exemple :

- Classe `CommandeController`
- Classe `CommandeService`
- Classe `CommandeRepository`
- Classe `Commande`
- Classe `LigneCommande`

Exemple Complet : Cartographie E-commerce

Étape 1 : Vue Logique (Packages)

- `com.shop.customers` : Customer, CustomerService, CustomerRepository
- `com.shop.products` : Product, ProductService, ProductRepository
- `com.shop.orders` : Order, OrderLine, OrderService, OrderRepository
- `com.shop.payments` : Payment, PaymentService, StripeAdapter
- `com.shop.infrastructure` : database, email, logging

Étape 2 : Vue d'Implémentation (Composants)

- shop-frontend.war (Application React)
- shop-backend.jar (API Spring Boot)
- shop-database (PostgreSQL)
- Composants externes : Stripe API, SendGrid API, AWS S3

Exemple Complet (suite)

Étape 3 : Vue de Déploiement (Infrastructure)

Production :

- 3 serveurs web (EC2)
- 1 load balancer (AWS ELB)
- 1 serveur base de données (RDS PostgreSQL)
- 1 bucket S3 (images)
- 1 CloudFront (CDN)

Développement :

- 1 laptop développeur
- 1 base de données locale (Docker)

Récapitulatif

Nous avons fait un **voyage complet** dans l'univers de l'architecture logicielle !

Ce que nous avons appris :

1. **L'Architecture Logicielle** = Comment organiser le code d'une application
2. **Les Principes Fondamentaux** : Abstraction, Encapsulation, Séparation des responsabilités
3. **La Démarche en 4 Étapes** : Analyser → Définir → Concevoir → Valider
4. **Les 2 Vues Essentielles** : Couches (logique) et Niveaux (physique)
5. **Les Types d'Architectures** : Monolithique, Microservices, SOA, Hexagonale...
6. **La Cartographie** : Diagrammes de Packages, Composants, Déploiement

Les Points Clés à Retenir (1/2)

1. Il n'y a pas UNE seule bonne architecture

- Chaque architecture a ses avantages et inconvénients
- Le choix dépend : de la taille de l'application, de l'équipe, des besoins de performance, du budget, du temps

2. On peut combiner plusieurs architectures

- Architecture en Couches (organisation du code)
- + Pattern MVC (pour la présentation)
- + Architecture Hexagonale (pour isoler le métier)
- + Déploiement N-Tiers (pour la scalabilité)

Les Points Clés à Retenir (2/2)

3. L'architecture évolue avec le temps

- On peut commencer simple et évoluer :
- Monolithique → Monolithique Modulaire → Microservices

4. La qualité de l'architecture est cruciale

Bonne architecture

- ✓ Code facile à comprendre
- ✓ Code facile à modifier
- ✓ Code facile à tester
- ✓ Application performante
- ✓ Application sécurisée

Mauvaise architecture

- × Code spaghetti
- × Bugs en cascade
- × Impossible d'ajouter des fonctionnalités
- × Équipe frustrée

Aller Plus Loin

Pour les Débutants :

- Bien comprendre les **couches** (Présentation, Métier, Données)
- Apprendre le **pattern MVC**
- Pratiquer la **séparation des responsabilités**

Pour les Intermédiaires :

- Approfondir l'**architecture en Couches** (5 couches)
- Maîtriser les **Design Patterns** (Repository, Singleton, Observer...)
- Découvrir l'**architecture Hexagonale**

Pour les Avancés :

- Explorer les **Microservices**
- Approfondir le **Domain-Driven Design (DDD)**
- Maîtriser l'**Event-Driven Architecture**

Ressources pour Continuer

Livres :

- « Clean Architecture » - Robert C. Martin
- « Domain-Driven Design » - Eric Evans
- « Building Microservices » - Sam Newman
- « Patterns of Enterprise Application Architecture » - Martin Fowler

Sites Web :

- <https://martinfowler.com/> (articles sur l'architecture)
- <https://microservices.io/> (patterns microservices)
- <https://c4model.com/> (modélisation d'architecture)

Pratique :

- Créez de petits projets
- Refactorez du code existant
- Lisez du code open source
- Participez à des code reviews

Mot de la Fin

L'architecture logicielle, c'est comme construire une maison

- On ne met pas le toit avant les fondations
- On planifie avant de construire
- On peut rénover et améliorer avec le temps
- Une bonne architecture dure des années

Mais surtout

« La meilleure architecture est celle qui répond aux besoins actuels tout en permettant l'évolution future »

N’oubliez pas

- **Commencez simple** (KISS : Keep It Simple, Stupid)
- **N’ajoutez de la complexité que quand c’est nécessaire** (YAGNI : You Ain’t Gonna Need It)
- **Refactorez régulièrement** (amélioration continue)

Mission Accomplie

Vous savez maintenant :

- ✓ Ce qu'est l'architecture logicielle
- ✓ Pourquoi c'est important
- ✓ Comment la concevoir
- ✓ Quels types d'architectures existent
- ✓ Comment la cartographier

Félicitations !

Vous êtes maintenant capable de comprendre et de discuter d'architecture logicielle comme un pro !

Annexe : Glossaire des Termes Principaux

Terme	Définition Simple
Abstraction	Cacher les détails compliqués
API	Interface pour communiquer avec un système
Couplage	Degré de dépendance entre modules
Cohésion	Degré de liaison entre éléments d'un module
Composant	Morceau de code réutilisable
Design Pattern	Recette pour résoudre un problème courant
Framework	Boîte à outils de développement
Microservices	Architecture avec plein de petits services indépendants
Monolithe	Application d'un seul bloc
Repository	Gestionnaire d'accès aux données
Scalabilité	Capacité à gérer plus de charge

Merci pour votre attention !

Questions ?