Université Nord Américaine privée
Institut International Technologie

Génie Logiciel Avancée
Architecture logicielle évoluée : Framework Spring

04- Architecture logicielle

Mohamed ZAYANI

2025/2026

Plan

- Définition
- Intérêt
- Avantages
- Logique métier et logique de présentation
- Catégories et exemples
- Observations

C’est quoi une architecture d’application informatique ?

❑ Définition

- L’architecture d’application désigne la manière dont les différentes parties d’un logiciel sont **organisées** et **interconnectées** pour former un système cohérent.

- Elle définit **les couches**, **les composants**, **leurs rôles** et **leurs modes de communication.**

- Elle décrit d’une façon **symbolique** et **abstraite** (sans entrer dans les détails d’implémentation) **la structure générale** de l’application (les composants et leurs interactions) afin de répondre à des besoins fonctionnels (ce que fait l’application) et non fonctionnels (performance, sécurité, maintenabilité…).

Pourquoi avons nous besoin d'une architecture d'application?

❑ Intérêt

- **Bien structurer le code:** pour éviter le désordre et les dépendances non maîtrisées.
- **Séparer les responsabilités:** (présentation, logique métier, accès aux données…).
- **Réduire le couplage et augmenter la cohésion**
- **Favoriser l'évolution:** anticiper et prévoir de nouvelles extensions et donner plus de flexibilité pour l'application.
- **Faciliter la maintenance**: localiser les modifications et réduire les risques des effets indésirables (Les corrections ou évolutions se font vite, sans provoquer de nouveaux bugs, et l'équipe peut intervenir sans passer des heures à actualiser le code)
- **Améliorer la collaboration entre les membres de l'équipe:** (développeurs, designers et testeurs,…).
- **Réutiliser des composants existants dans d'autres projets**.

Quels sont les atouts d'une architecture d'application?

❑ Avantages

- **Clarté et organisation** : Chaque partie du code a une fonction précise (Pas de logique métier mélangée avec l'interface graphique)
- **Maintenance simplifiée** : séparation des couches (ex: MVC, MVVM) et indépendance des modules
- **Réutilisabilité** : Les modules peuvent être repris dans d'autres applications.
- **Testabilité** : Plus facile d'écrire et d'exécuter des tests unitaires et fonctionnels.
- **Évolutivité** : L'architecture peut s'adapter à des besoins ou charges plus importants.
- **Interopérabilité** : Possibilité de combiner différentes technologies ou de communiquer avec d'autres systèmes (définir des API pour consommer et exposer des services..)

Logique métier vs Logique de présentation

Logique métier	Logique de présentation
• C'est la partie de l'application qui crée, stocke et modifie les données et qui leur donne du sens. • Elle est spécifique du domaine d'application et indépendante des problèmes d'interaction avec l'utilisateur. • On y retrouve les classes du domaine et les règles métier qui régissent les données. → C'est le cœur fonctionnel du système (ce que fait l'application): le QUOI	• Elle désigne tout type de logique qui spécifie comment l'interface graphique réagit aux interactions avec l'utilisateur ou aux changements induits dans le modèle de domaine. • Elle concerne l'affichage des données • Elle s'occupe de la cinématique des écrans → Comment on montre le QUOI à l'utilisateur

- Pour une conception architecturale, on a intérêt à découper et séparer la logique métier de la logique de présentation!!!
- Les modèles d'architectures proposent diverses versions pour ce découpage et comment les deux logiques interagissent.
- Chaque patron d'architecture possède sa façon de partitionner une application et d'y répartir les responsabilités.

Exemple: Gestion des étudiants

Logique métier (Business Logic)	Logique de présentation (UI/Controllers)
• Elle définit les règles métier : ▪ Ajouter un étudiant. ▪ Calculer sa moyenne. ▪ Vérifier si l'étudiant est admis ou ajourné. ▪ Ici, il n'y a aucun affichage (Ne sait rien sur l'interface console, Swing, Web,..) ▪ C'est du métier pur : calculer une moyenne et dire si l'étudiant est admis. ▪ La logique métier est indépendante : elle peut être testée seule.	• Ici, on décide comment montrer les résultats à l'utilisateur. • On peut choisir Swing ou Console ou Web. • Ne calcule rien : elle demande tout au modèle. • On peut changer le style d'affichage (Console, Swing ,Web) sans changer la logique métier. • La présentation s'appuie toujours sur le métier, jamais l'inverse.

Exemple de séparation des responsabilités avec Java

- **Logique « métier »**

- Une classe qui modélise l'entité « **Etudiant** »:

```
    // LOGIQUE METIER
public class Etudiant
{
   private String nom;
   private double note1;
   private double note2;

   public Etudiant (String nom,
   double note1, double note2) {
   this.nom = nom;
   this.note1 = note1;
   this.note2 = note2;
   }
```

```
// Règle métier 1 : calculer la moyenne
  public double calculerMoyenne() {
  return (note1 + note2) / 2.0;
  }
  // Règle métier 2 : déterminer si
  // l'étudiant est admis
  public boolean estAdmis() {
  return calculerMoyenne() >= 10;
  }

  public String getNom() {
  return nom;
  }
}
```

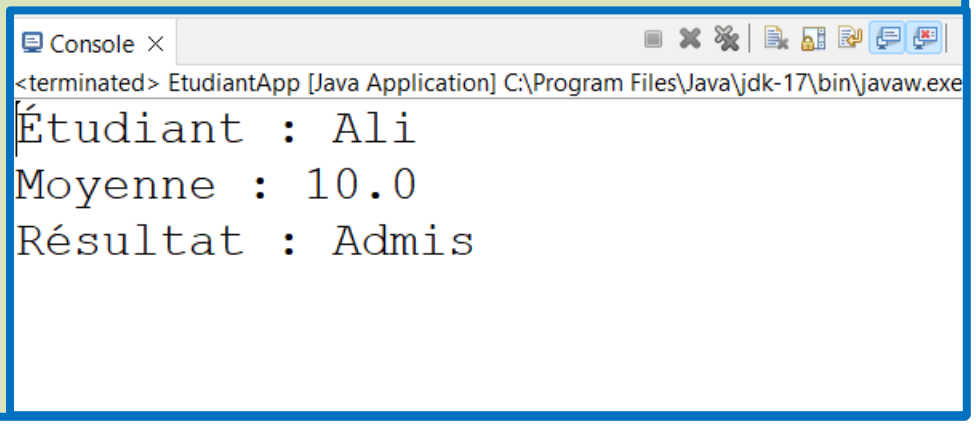
Exemple de séparation des responsabilités avec Java

- **Logique de présentation**

- En mode «**Console**»

```
public class EtudiantApp {
    public static void main(String[] args) {
    // Créer un étudiant (métier)
    Etudiant e = new Etudiant("Ali", 12, 8);
    // Présentation : afficher les résultats
    System.out.println("Étudiant : " + e.getNom());
    System.out.println("Moyenne : " + e.calculerMoyenne());

    if (e.estAdmis()) {
    System.out.println("Résultat : Admis ");
    } else {
    System.out.println("Résultat : Ajourné ");
    }
    }
}
```



Exemple de séparation des responsabilités avec Java

- **Logique de présentation**

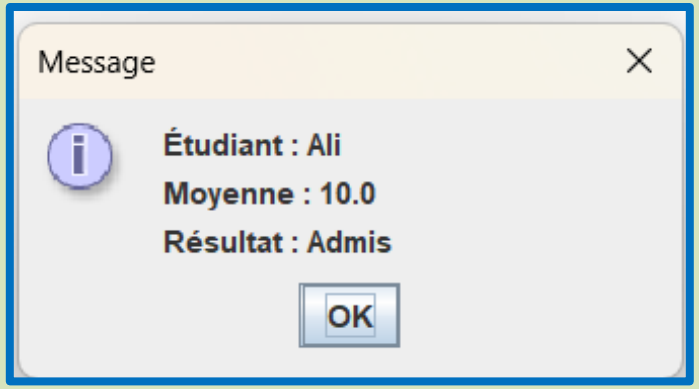
- En mode «**Swing**»

```
import javax.swing.*;

public class EtudiantSwing
{
   public static void main(String[] args) {
   Etudiant e = new Etudiant("Ali", 12, 8);

   String message = "Étudiant : " + e.getNom() +
   "\n    Moyenne : " + e.calculerMoyenne() +
   "\n    Résultat : " + (e.estAdmis() ? "Admis " : "Ajourné ");

   JOptionPane.showMessageDialog(null, message);
   }
}
```



Quelques exemples d'architectures d'application?

- **MVC (Model–View–Controller)**: Séparation entre données, interface et logique de contrôle.
- **MVP (Model–View–Presenter)** : Variante du MVC, avec un Presenter (***Présentateur***) qui gère la logique de présentation. (la vue est passive et ne voit pas le modèle)
- **MVVM (Model–View–ViewModel)** : Adaptée aux interfaces réactives avec liaison de données (***data binding***).
- **CBA (Component-Based Architecture)**: composée de petits blocs indépendants (composants) qui gèrent leur propre état et logique.
- **Multi couches**: style d'organisation d'application où le code est divisé en **couches logiques distinctes**, chacune ayant un rôle bien défini. Chaque couche **ne communique qu'avec la couche directement inférieure ou supérieure**.
- **Clean Architecture**: souvent représentée sous forme d'**anneaux concentriques** (entités, cas d'utilisation, interface, frameworks)
- **SOA (Service-Oriented Architecture)** :Application composée de services indépendants qui communiquent via un réseau.
- **Microservices** : Évolution moderne du SOA, avec des services plus petits, autonomes et déployables séparément.
- **MOA (Message-Oriented Architecture)** : C'est un modèle où les composants d'une application communiquent en s'échangeant des messages via un middleware de messagerie (ex. JMS en Java).

MVC Model-View-Controller

❑ Principe

Séparation en 3 couches : données (Model), interface (View), logique de contrôle (Controller)

❑ Exemples de frameworks

- **Java** : Spring MVC, Struts 2, JavaFX, Swing (Java)
- **.NET** : ASP.NET MVC
- **PHP** : Laravel, Symfony
- **JavaScript** : AngularJS (version 1.x, inspirée de MVC)
- **Ruby** : Ruby on Rails
- **Python**: Django (utilise MVC ou bien MVT: Model View Template)

MVP Model–View–Presenter

❑ Principe

Le Presenter sert d'intermédiaire unique entre la View et le Model.

la View est **passive** (elle ne contient de logique)

❑ Exemples de frameworks

- **Android** : Moxy, Mosby
- **C# / WinForms** : Implémentations MVP personnalisées
- **Google**: GWT

MVVM Model–View–ViewModel

❑ Principe

Utilise le data-binding pour synchroniser automatiquement l'interface et les données via le ViewModel.

❑ Exemples de frameworks

- **.NET** : WPF, Xamarin.Forms, MAUI
- **Web** : Vue.js (JavaScript), Angular(TypeScript)
- **Mobile** : Flutter (Dart)
- **Java** : JavaFx (avec la technique de Binding en utilisant le type Property)

CBA Component-Based-Architecture

❑ Principe

Composée de petits blocs indépendants (composants) qui gèrent leur propre état et logique.

Très proche du MVVM mais centrée sur l'encapsulation des composants UI

❑ Exemples de frameworks

- **Web** : React (JavaScript/TypeScript)
- **Mobile** : React Native, Flutter

Architecture multi couches

❑ Principe

Organisation en couches:

- Présentation: UI Layer
- Métier: Business Layer
- Accès aux données: Data Access Layer

Chaque couche a un rôle précis et communique uniquement avec la couche voisine.

Ce qui renforce la séparation des responsabilités

Les couches peuvent être divisées d'avantages (4couches, 5 couches,..)

❑ Exemples de frameworks

- **Java :** Spring Boot (Controllers → Services → Repositories).
- **.NET :** ASP.NET Core (Controllers → Services → Data Access).

Clean Architecture

❑ Principe

Elle vise à séparer clairement les responsabilités et à rendre l'application indépendante des frameworks, de l'UI, de la base de données ou de tout autre outil externe.

Le cœur de l'application (la logique métier) ne doit pas dépendre des détails techniques, mais l'inverse.

La Clean Architecture est souvent représentée sous forme d'anneaux **concentriques**

❑ Exemples de frameworks

- En réalité, la **Clean Architecture** n'est pas intégrée "par défaut" dans un framework précis,
- C'est plutôt un **style architectural** que les développeurs appliquent volontairement.
- Mais certains frameworks ou écosystèmes se prêtent particulièrement bien à son adoption grâce à leur modularité et leurs outils. (ex: Spring Boot, ASP.NET, Angular, NestJS,..)

SOA Service-Oriented Architecture

❑ Principe

Application composée de services indépendants, qui communiquent via un protocole standard (SOAP, REST...).

❑ Exemples de frameworks

- **.NET** : WCF (Windows Communication Foundation)
- **Node.js** : Express.js (pour exposer des services REST)
- **Java** : Apache CXF, Spring Web Services

MOA Message-Oriented Architecture

❑ Principe

Composants qui communiquent via un système de messagerie asynchrone.

❑ Exemples de frameworks

- **.NET** : NServiceBus, MassTransit
- **Node.js** : amqplib (RabbitMQ), kafka-node
- **Java** : JMS (Java Message Service) avec ActiveMQ, RabbitMQ, Apache Kafka
- **Cloud** : AWS SQS, Azure Service Bus, Google Pub/Sub

Observations

- **MVC**, **MVP MVVM** et **CBA** concernent surtout **la structure interne** des applications côté code et interface (surtout le niveau présentation)
- **Clean Architecture** et **Architecture mutli couche** donnent une vision **globale** et **technique** de l'application.
- **SOA , micro services** et **MOA** concernent surtout la communication et l'intégration entre différents composants ou services (programmation **distribuée**)