جامعة مولاي إسماعيل
UNIVERSITÉ MOULAY ISMAÏL

ENSAM
المدرسة الوطنية العليا للفنون و المهن
ÉCOLE NATIONALE SUPÉRIEURE D'ARTS ET MÉTIERS

JOBINTECH

Architecture logicielle

Ahmed Laatabi
a•laatabi{at}umi•ac•ma
ENSAM - Meknès
2025-2026

Logiciel

- Un **logiciel** (*software*) est un ensemble de **programme**s, de **donnée**s, et de **règle**s qui permettent à un appareil informatique (ordinateur, smartphone, …) de **fonctionner** et d'**exécuter** des **tâches** spécifiques. Il se compose d'une suite d'**instructions** (code), écrites dans un **langage de programmation**, qui implémentent un ou plusieurs **algorithme**s.

- Le **matériel** (*hardware*) est l'ensemble des composants physiques et électroniques d'un système informatique (carte mère, disque dur, …) qui sert de support et permet l'**exécution des logiciels**.

Système informatique = *Software* (immatériel) + *Hardware* (matériel).

- Le système informatique est l'une des composantes principales du **système d'information** (**SI**).
- Le **SI** est l'ensemble organisé de ressources matérielles, logicielles, humaines et organisationnelles permettant la collecte, le stockage, le traitement et la diffusion de l'**information** (les données) nécessaire au fonctionnement et à la prise de décision au sein d'une organisation.

Architecture logicielle

- L'**architecture logicielle** décrit, schématise et documente l'ensemble des éléments (ou composantes) d'un système informatique ainsi que leurs interactions (ou relations) en termes d'échanges et d'entrées/sorties.

- Son objectif est de définir la structure, les modèles, et les solutions (technologies) nécessaires pour répondre aux besoins du client et assurer la cohérence, la fiabilité, et l'évolutivité du système.

- Une bonne architecture logicielle doit garantir les **qualités non fonctionnelles** du système :
  - **Maintenabilité** : facilité de corriger, modifier, ou faire évoluer le logiciel.
  - **Performance** : rapidité et efficacité d'exécution.
  - **Scalabilité** : capacité à s'adapter à une charge croissante de données et de trafic.
  - **Sécurité** : protection des données et des processus.

- Les architectures logicielles modernes tendent à adopter une **séparation** en modules (ou couches) : 1) **interface utilisateur** (couche de présentation); 2) **processus métiers** (couche logique); 3) **persistance des données** (couche d'accès aux données).

Architecte logiciel

- L'**architecte logiciel** est le concepteur de haut niveau du système. Son rôle consiste à :
  - Définir la structure globale et les principes de conception du logiciel :
  - Choisir les technologies, les styles d'architecture (monolithique, microservices, ...), les patterns (modèles de conception) et les normes à suivre.
  - Coordonner entre les équipes non techniques (produit, direction) et les équipes techniques (développement, test, déploiement).
  - Garantir l'intégration et la cohérence entre les différents modules du logiciel, assurant ainsi qu'il réponde aux besoins clients et au cahier des charges.

- **Architecte logiciel** : conçoit le plan stratégique du logiciel avant sa construction. Il produit des documents et des diagrammes qui répondent au "*quoi ?*" (structure et règles).
- **Ingénieur logiciel / Développeur** : construit et implémente le logiciel en suivant ce plan. Il se concentre sur le "*comment ?*" (algorithmes, codes, et implémentation).

Principes fondamentaux

- **Principes structurels** : qualités non fonctionnelles à maximiser.
  - Séparation des préoccupations (*separation of concerns*) : le système est divisé en **modules**, chacun ayant une responsabilité unique et bien définie. Les composantes du même module doivent être fortement liées et cohérentes (*high cohesion*) → **modularité** et **maintenabilité**.
  - Modularité : les modules qui composent le système sont indépendants et peuvent être développés, testés et déployés de manière autonome et parallèle. Chaque module doit encapsuler sa complexité, qui doit être abstraite aux interactions → **évolutivité**, **interopérabilité** et **réduction des coûts**.
  - Couplage Faible (*low coupling*) : les dépendances entre les modules doivent être minimales pour éviter les erreurs inter-modules. chaque module doit pouvoir évoluer indépendamment afin de faciliter les corrections ou l'ajout de nouvelles fonctionnalités → **interopérabilité** et **scalabilité**.
- **Principes de conception** (*design principles*) : règles à suivre pour répondre aux besoins fonctionnels.
  - **KISS** (*Keep It Simple & Stupid*) : toujours privilégier la **solution la plus simple** qui **fonctionne**.
  - **DRY** (*Don't Repeat Yourself*) : les **données** et les **logiques** ne doivent **pas être dupliquées**.
  - **YAGNI** (*You Aren't Gonna Need It*) : éviter d'implémenter des **données** ou **logiques inutiles**.

SOLID

SOLID regroupe cinq principes de conception à suivre pour produire de bonnes architectures logicielles (code compréhensible, flexible et maintenable), notamment en **programmation orienté objet** :

- ***S**ingle responsibility* (responsabilité unique) : une **fonction** ou une **classe** ne doit avoir qu'**une seule responsabilité** (un seul rôle, objectif).
- ***O**pen/closed* (ouvert/fermé) : une fonction ou une classe doit être **fermée à la modification** mais **ouverte à l'extension** : ajout de nouvelles fonctionnalités sans modifier le code existant.
- ***L**iskov substitution* (substitution de Liskov) : une **instance de type de base** doit pouvoir **être remplacée** par **une instance de l'un de ses sous-types** sans altérer le **bon fonctionnement du programme**. Les sous-classes peuvent donc être utilisées de manière interchangeable avec leurs classes parentes.
- ***I**nterface segregation* (ségrégation des interfaces) : préférer la définition de plusieurs **interfaces spécifiques** plutôt qu'une **seule interface générale**. Ainsi, les classes ne dépendent que des méthodes dont elles ont besoin, ce qui réduit les couplages inutiles.
- ***D**ependency inversion* (inversion des dépendances) : il faut dépendre des **abstractions**, pas des **implémentations** (les modules de haut niveau ne doivent pas dépendre des modules de bas niveau, mais tous deux doivent dépendre d'abstractions).

Responsabilité unique

"*A class should have only one reason to change*"

- Une fonction permet de récupérer (depuis le clavier) les informations d'un étudiant (CNE, nom complet, date de naissance) puis de les enregistrer dans une BD.
- Cette fonction a deux raisons de changer (car elle a deux responsabilités distinctes) :
  - Si l'on souhaite récupérer une donnée supplémentaire (lieu de naissance).
  - Si l'on souhaite modifier le mécanisme d'enregistrement des données :
    - MySQL → MongoDB.

```
def saisir_et_enregistrer_etudiant():
    cne = input("CNE : ")
    nom_complet = input("Nom complet : ")
    date_naissance = input("Date de naissance : ")

    print("Enregistrement dans MySQL...")
    print(f"INSERT INTO etudiant VALUES ('{cne}',",
          f"'{nom_complet}', '{date_naissance}');")

saisir_et_enregistrer_etudiant()
```

```
def saisir_etudiant():
    cne = input("CNE : ")
    nom_complet = input("Nom complet : ")
    date_naissance = input("Date de naissance : ")
    return cne, nom_complet, date_naissance

def enregistrer_etudiant(cne, nom_complet, date_naissance):
    print("Enregistrement dans MySQL...")
    print(f"INSERT INTO etudiant VALUES ('{cne}',"
          f"'{nom_complet}', '{date_naissance}');")

cne, nom_complet, date_naissance = saisir_etudiant()
enregistrer_etudiant(cne, nom_complet, date_naissance)
```

?

Responsabilité unique

- Le principe de responsabilité unique (**SRP**) exige qu'un changement de nature n'affecte qu'une seule composante.
- Ici, les deux fonctions dépendent directement de la même structure de données (cne, nom_complet, date_naissance, ...).
- Si la structure des données change → il faut modifier les deux fonctions (violation du SRP).
- L'utilisation des **fonctions** simples, **couplées fortement** aux détails de la **structure de données**, limite l'implémentation correcte du principe de responsabilité unique.
- Il faut que les deux fonctions dépendent d'une interface ou d'abstraction commune (un objet), et non directement de ses champs (structure interne).

```python
class Etudiant:
    def __init__(self, cne, nom_complet, date_naissance):
        self.cne = cne
        self.nom_complet = nom_complet
        self.date_naissance = date_naissance

    # objet --> dictionnaire
    def to_dict(self):
        return {
            "cne": self.cne,
            "nom_complet": self.nom_complet,
            "date_naissance": self.date_naissance
        }


def saisir_etudiant():
    cne = input("CNE : ")
    nom_complet = input("Nom complet : ")
    date_naissance = input("Date de naissance : ")
    return Etudiant(cne, nom_complet, date_naissance)


def enregistrer_etudiant(etudiant):
    data = etudiant.to_dict()
    print("Enregistrement dans MySQL...")
    print(f"INSERT INTO etudiant VALUES ("
          f"{', '.join([repr(v) for v in data.values()])});")


etu = saisir_etudiant()
enregistrer_etudiant(etu)
```

?

Ouvert/fermé

- Une fois qu'une classe ou une fonction a été testée et validée, elle ne doit plus être modifiée, mais seulement étendue pour ajouter de nouvelles fonctionnalités.

```python
class Etudiant:
    def __init__(self, cne, nom_complet, date_naissance):
        self.cne = cne
        self.nom_complet = nom_complet
        self.date_naissance = date_naissance

    # objet --> dictionnaire
    def to_dict(self):
        return {
            "cne": self.cne,
            "nom_complet": self.nom_complet,
            "date_naissance": self.date_naissance
        }

def saisir_etudiant():
    cne = input("CNE : ")
    nom_complet = input("Nom complet : ")
    date_naissance = input("Date de naissance : ")
    return Etudiant(cne, nom_complet, date_naissance)


def enregistrer_etudiant(etudiant):
    data = etudiant.to_dict()
    print("Enregistrement dans MySQL...")
    print(f"INSERT INTO etudiant VALUES ("
          f"{', '.join([repr(v) for v in data.values()])});")
```

```python
class Etudiant2(Etudiant):
    def __init__(self, cne, nom_complet, date_naissance,
                 lieu_naissance):
        super().__init__(cne, nom_complet, date_naissance)
        self.lieu_naissance = lieu_naissance

    def to_dict(self):
        data = super().to_dict()
        data["lieu_naissance"] = self.lieu_naissance
        return data

def saisir_etudiant2():
    etud = saisir_etudiant()
    lieu_naissance = input("Lieu de naissance : ")
    return Etudiant2(etud.cne, etud.nom_complet, etud.date_naissance,
                     lieu_naissance)

etu = saisir_etudiant2()
enregistrer_etudiant(etu)
```

Substitution de Liskov

- Bonne utilisation de l'héritage : si ***G*** est un sous-type de ***T***, alors tout objet de type ***T*** peut être remplacé par un objet de type ***G*** sans altérer les propriétés désirables du programme.

  → les classes dérivées ***G*** ne doivent pas casser le code qui utilise les classes de base ***T***.

```python
class Etudiant:
    def __init__(self, cne, nom_complet, date_naissance):
        self.cne = cne
        self.nom_complet = nom_complet
        self.date_naissance = date_naissance

    # objet --> dictionnaire
    def to_dict(self):
        return {
            "cne": self.cne,
            "nom_complet": self.nom_complet,
            "date_naissance": self.date_naissance
        }

def saisir_etudiant():
    cne = input("CNE : ")
    nom_complet = input("Nom complet : ")
    date_naissance = input("Date de naissance : ")
    return Etudiant(cne, nom_complet, date_naissance)


def enregistrer_etudiant(etudiant):
    data = etudiant.to_dict()
    print("Enregistrement dans MySQL...")
    print(f"INSERT INTO etudiant VALUES ("
          f"{', '.join([repr(v) for v in data.values()])});")
```

```python
class Etudiant2(Etudiant):
    def __init__(self, cne, nom_complet, date_naissance,
                 lieu_naissance):
        super().__init__(cne, nom_complet, date_naissance)
        self.lieu_naissance = lieu_naissance

    def to_dict(self):
        data = super().to_dict()
        data["lieu_naissance"] = self.lieu_naissance
        return data

def saisir_etudiant2():
    etud = saisir_etudiant()
    lieu_naissance = input("Lieu de naissance : ")
    return Etudiant2(etud.cne, etud.nom_complet, etud.date_naissance,
                     lieu_naissance)

etu = saisir_etudiant2()
enregistrer_etudiant(etu)
```

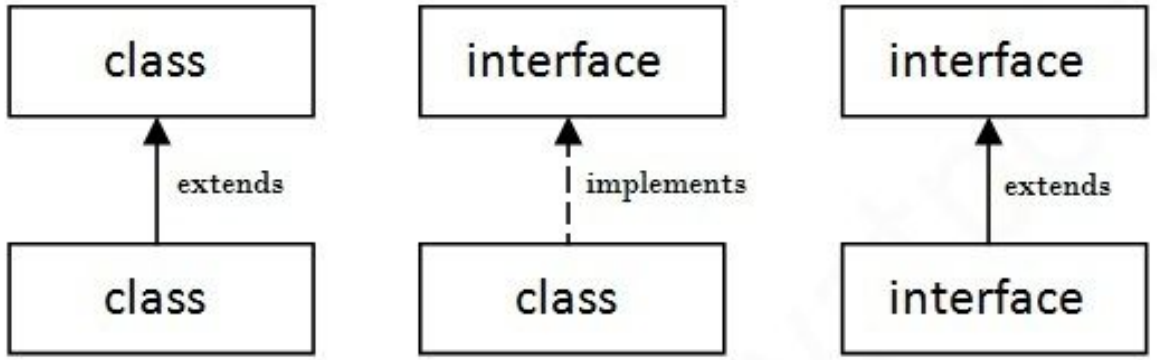
POO

- L'**héritage** permet à une classe (classe fille, classe dérivée, sous-classe) d'acquérir et réutiliser les propriétés (attributs) et les comportements (méthodes) d'une autre classe (classe mère, classe de base, super-classe), tout en ajoutant ou modifiant des fonctionnalités : class Etudiant2 (Etudiant).
- Le **polymorphisme** (multiples formes) permet de traiter des objets de types différents via une interface unique : une même méthode peut se comporter différemment selon l'objet sur lequel elle est appelée.

```
class Etudiant:
    def __init__(self, cne, nom_complet, date_naissance):
        self.cne = cne
        self.nom_complet = nom_complet
        self.date_naissance = date_naissance

    def afficher(self):
        print(f"CNE : {self.cne}, Nom : {self.nom_complet},"
              f"Date de naissance : {self.date_naissance}")

class Etudiant2(Etudiant):
    def __init__(self, cne, nom_complet, date_naissance,
                 lieu_naissance):
        super().__init__(cne, nom_complet, date_naissance)
        self.lieu_naissance = lieu_naissance

    def afficher(self):
        print(f"CNE : {self.cne}, Nom : {self.nom_complet}, "
              f"Date de naissance : {self.date_naissance}, "
              f"Lieu : {self.lieu_naissance}")

etudiants = [
    Etudiant("123", "Moha Sage", "01/01/79"),
    Etudiant2("456", "Moha Fou", "12/05/88", "Maroc")
]

for etu in etudiants:
    etu.afficher()
```

Interface

- L'**interface** est un concept fondamental de la programmation orientée objet (POO).
- Elle définit un ensemble de **méthodes publiques** (et parfois de constantes) qu'une **classe doit implémenter**.
- Toute classe qui implémente cette interface doit **fournir une définition** pour chacune de **ces méthodes**.
- C'est un moyen d'abstraction : on se concentre sur ce qu'une classe **doit faire** (le comportement), plutôt que sur **la manière dont elle le fait** (l'implémentation).




```
class EnregistreurEtudiant:
    #Enregistrer un étudiant, quelle que soit la BD
    def enregistrer(self, etudiant):
        raise NotImplementedError(
        "La méthode enregistrer()doit être implémentée!")

class EnregistreurMySQL(EnregistreurEtudiant):
    def enregistrer(self, etudiant):
        data = etudiant.to_dict()
        print(f"Enregistrement dans MySQL...")
        print(f"INSERT INTO etudiant VALUES ("
                f"{', '.join([repr(v) for v in data.values()])});")
```

Ségrégation des interfaces

- Un objet ne doit **pas dépendre** de méthodes qu'il **n'utilise pas**.
- Il est préférable de diviser une **interface générale** (monolithique) en **plusieurs interfaces spécifiques** et ciblées.
- Chaque objet n'implémente et n'accède qu'aux méthodes qui le concernent, évitant les dépendances inutiles.
- La classe **EnregistreurMySQL** doit implémenter la méthode **afficher()** dont il n'aura jamais besoin !

```
class EnregistreurEtudiant(ABC):
    @abstractmethod
    def enregistrer(self, etudiant):
        pass

    @abstractmethod
    def afficher(self, etudiant):
        pass

class EnregistreurMySQL(EnregistreurEtudiant):
    def enregistrer(self, etudiant):
        data = etudiant.to_dict()
        print(f"Enregistrement dans MySQL...")
        print(f"INSERT INTO etudiant VALUES ("
              f"{', '.join([repr(v) for v in data.values()])});")

    def afficher(self, etudiant):
        print(f"{etudiant.nom_complet} ({etudiant.cne}) "
              f"- Né le {etudiant.date_naissance}")

def enregistrer_etud(etudiant: Etudiant,
                     enregistreur: EnregistreurEtudiant):
    enregistreur.enregistrer(etudiant)

etud = Etudiant("123", "Franz Kafka", "01/01/1900")
enregistrer_etud(etud, EnregistreurMySQL())
```

Inversion des dépendances

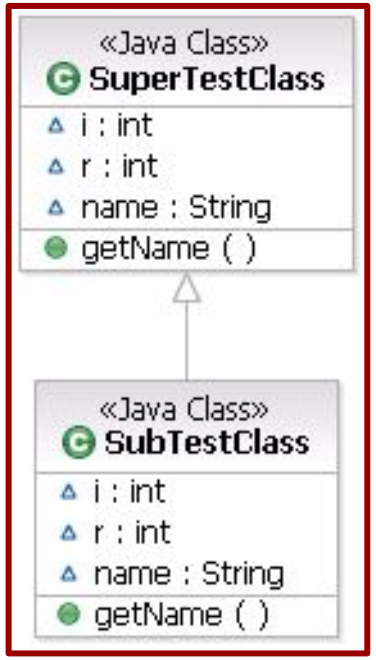
- Les **modules de haut niveau** ne doivent pas dépendre des **modules de bas niveau**. Les deux doivent dépendre d'**abstractions**.
- Les **abstractions** ne doivent pas dépendre des **détails**, mais l'inverse.

- La fonction d'enregistrement d'un étudiant (haut niveau) dépend directement de la logique spécifique à MySQL (bas niveau) → changer de BD obligerait à modifier cette fonction.
- **Solution** : inverser la dépendance pour que le code bas niveau (MySQL, MongoDB, …) dépende d'une abstraction définie au niveau supérieur (enregistrement d'un étudiant).

```
class EnregistreurEtudiant:
    #Enregistrer un étudiant, quelle que soit la BD
    def enregistrer(self, etudiant):
        raise NotImplementedError(
        "La méthode enregistrer()doit être implémentée!")

class EnregistreurMySQL(EnregistreurEtudiant):
    def enregistrer(self, etudiant):
        data = etudiant.to_dict()
        print(f"Enregistrement dans MySQL...")
        print(f"INSERT INTO etudiant VALUES ("
              f"{', '.join([repr(v) for v in data.values()])});")
```

```
class EnregistreurMongoDB(EnregistreurEtudiant):
    def enregistrer(self, etudiant):
        data = etudiant.to_dict()
        print(f"Enregistrement dans MongoDB...")
        print(f"db.etudiants.insert_one({data})")

def enregistrer_etud(etudiant: Etudiant,
                     enregistreur: EnregistreurEtudiant):
    enregistreur.enregistrer(etudiant)

etud = Etudiant("123", "Franz Kafka", "01/01/1900")
enregistrer_etud(etud, EnregistreurMySQL())
enregistrer_etud(etud, EnregistreurMongoDB())
```

UML/JAVA



SuperTestClass.java:

```
public class SuperTestClass {
	int i = 3;
	int r = 5;
	String name ="myName";

	public void getName(){
	};

}
```

SubTestClass.java:

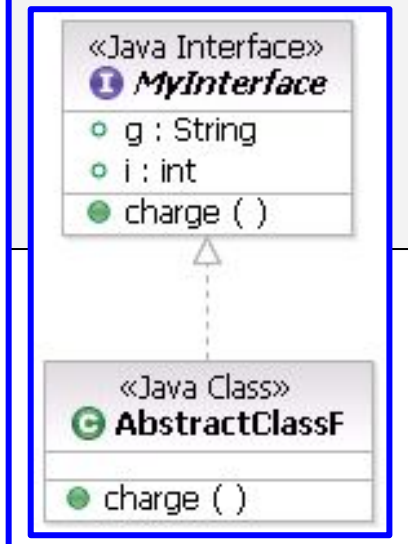
```
public class SubTestClasss
	 extends SuperTestClass {
	int i = 2;
	int r = 3;
	String name ="myName";

	public void getName(){
	};

}
```

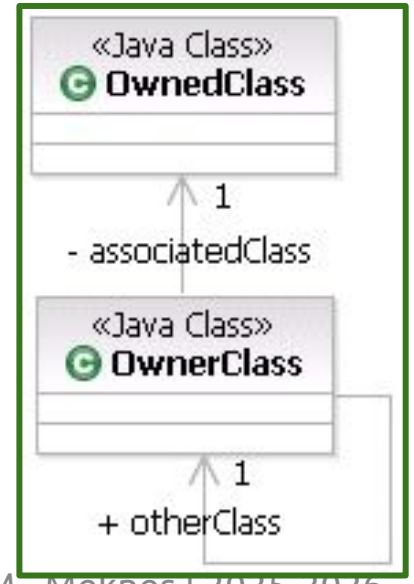
MyInterface.java:

```
public interface MyInterface {
	String g= "";
	int i= 0;

	public void charge (int x);

}
```

AbstractClassF.java:

```
public class AbstractClassF
		implements MyInterface {

	public void charge(int x){
	};

}
```





OwnedClass.Java:

```
public class OwnedClass {

	// <<class body>>
}
```

OwnerClass.Java:

```
public class OwnerClass {
private OwnedClass associatedClass;
public OwnerClass otherClass;

	// <<class body>>
}
```

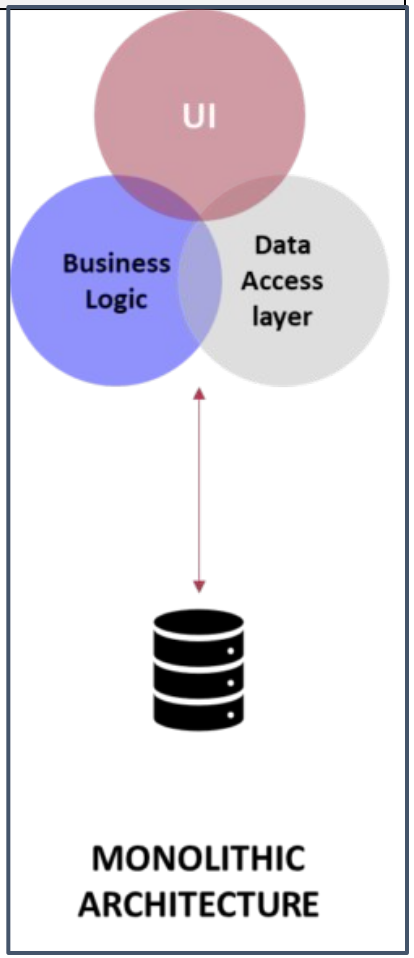
Application

Réaliser un petit projet Java qui respecte les principes **SOLID**

Nous souhaitons modéliser une entité **Etudiant** et son sous-type **EtudiantUMI**, qui introduit un attribut supplémentaire. Le programme doit permettre d'afficher et de sauvegarder (*simulé simplement par un affichage à l'écran, pour le moment!*) les informations des étudiants vers des différentes BD (MySQL, MongoDB).

Architecture monolithique

- L'**architecture monolithique** est le modèle traditionnel où toutes les fonctionnalités d'une application sont regroupées dans une seule unité:
  - Un seul exécutable, un seul répertoire de code source, une seule BD.
  - Les composantes sont étroitement couplées.
  - Application autonome et indépendante.
- Facile à prendre en main, rapide à développer (au début), simple à déployer.
- Un changement de code → reconstruire et redéployer toute l'application.
- Complexité de mises à jour et d'ajout de nouvelles fonctionnalités, en particulier avec des applications volumineuses.
- Difficile de faire évoluer une seule fonctionnalité indépendamment.
- Les architectures moderne → décomposition en services / fonctionnalités spécialisés et faiblement couplés → agilité, flexibilité, et évolutivité.



Architecture monolithique

Une application e-commerce est déployée comme une seule application monolithique. Les trois fonctionnalités métiers sont :

- Prise de commandes.
- Vérification de l'inventaire (stock) et du crédit disponible.
- Expédition des commandes clients.

Toutes les composantes, y compris l'**interface utilisateur** (StoreFrontUI) et les services **backend** (gestion du crédit, inventaire, expédition) sont regroupées dans un même projet.

→ *Par exemple, une **application Java** peut être déployée dans **un seul fichier WAR** sur un serveur **Tomcat**.*