UML — Associations, Agrégation & Composition

Cours CDA • Module UML 2.5

« Pensez comme un architecte, pas comme un codeur. »

Style inspiré de David J. Malan — CS50, Harvard

Chapitre 0 — Pourquoi l'UML ?

Imaginez que vous construisez une maison. Avant de poser la première brique, vous dessinez un plan. L'UML, c'est le plan de votre logiciel.

UML signifie Unified Modeling Language. Ce n'est pas un langage de programmation — vous ne l'exécutez pas. C'est un langage visuel qui vous permet de décrire comment les différentes pièces de votre programme se connectent entre elles.

> 🎯 *Objectif du cours : comprendre les 4 types de relations entre classes UML : Association, Dépendance, Agrégation et Composition.*

Chapitre 1 — La Classe UML, la brique de base

Avant de parler de relations, revenons sur ce qu'est une classe UML. C'est une boîte rectangulaire divisée en 3 zones :

- Le nom de la classe (en haut, en gras)
- Les attributs — les données que la classe stocke
- Les méthodes — les actions que la classe peut faire

Exemple — une classe Voiture :

Voiture
`- marque : String`
`- vitesse : int`
`- nbRoues : int`
`+ demarrer() : void`
`+ accelerer(v: int) : void`
`+ arreter() : void`

Les symboles + et - indiquent la visibilité :

- + public : accessible de partout
- - private : accessible seulement dans la classe
- # protected : accessible dans la classe et ses sous-classes

> 💡 *Analogie CS50 : pensez à une classe comme à un formulaire administratif. Les attributs sont les champs vierges (nom, prénom, âge), et les méthodes sont les actions qu'on peut faire avec ce formulaire (soumettre, imprimer, archiver).*

Chapitre 2 — Les Relations entre Classes

Maintenant, la vraie question : comment les classes se parlent-elles entre elles ? Il existe 4 types principaux de relations en UML 2.5.

2.1 L'Association — « utilise »

Une association dit : « Cette classe connaît cette autre classe. » C'est la relation la plus générale.

> 🔗 *Représentation UML : une ligne simple avec une flèche. Parfois annotée d'un nom de rôle et d'une multiplicité.*

Exemple du tableau : Personne possède une Voiture.

```
Personne ──────────────────> Voiture
           possède
         0..2        0..5
   (une personne peut avoir 0 à 2 voitures)
   (une voiture peut appartenir à 0 à 5 personnes)
```

En Java, ça se traduit comme ça :

```
public class Personne {
    private List<Voiture> voitures; // 0 à 2 voitures

    public void ajouterVoiture(Voiture v) {
        this.voitures.add(v);
    }
}
```

> ⚠️ *Notez bien : dans la version du bas du tableau, Personne stocke ses voitures dans un ArrayList<Voiture>. C'est exactement ce que représente la ligne dans le diagramme.*

2.2 La Dépendance — « dépend de »

Une dépendance est plus faible qu'une association. Elle dit : « Cette classe utilise temporairement cette autre classe, mais ne la stocke pas. »

> 🔗 *Représentation UML : une ligne pointillée avec une flèche ouverte (- - - - ->).*

Exemple : une classe Moteur dépend d'une classe Carburant pour démarrer, mais ne la stocke pas.

```
public class Moteur {
    // Pas d'attribut Carburant — c'est une DÉPENDANCE

    public void demarrer(Carburant c) { // c est juste un paramètre
        c.consommer(10);
    }
}
```

> 💡 *Analogie : c'est la différence entre posséder une voiture (association) et louer une voiture pour un week-end (dépendance). Vous l'utilisez, mais vous ne la gardez pas.*

2.3 L'Agrégation — « contient (mais peut vivre sans) »

L'agrégation est une relation de type tout/partie FAIBLE. Le tout contient des parties, mais les parties peuvent exister indépendamment du tout.

> 🔗 *Représentation UML : une ligne avec un losange VIDE ( ) du côté du « tout ».*

Exemple : une Équipe de football contient des Joueurs. Si l'équipe est dissoute, les joueurs existent encore !

```
public class Equipe {
    private List<Joueur> joueurs; // agrégation

    public void ajouterJoueur(Joueur j) {
        joueurs.add(j);
    }
    // Si Equipe est détruite, les Joueur existent encore
}
```

2.4 La Composition — « contient (et contrôle la vie) »

La composition est une relation de type tout/partie FORTE. Si le tout est détruit, les parties sont détruites aussi.

🔗 *Représentation UML : une ligne avec un losange PLEIN ( ) du côté du « tout ».*

Exemple : une Voiture contient un Moteur. Si la Voiture est détruite à la casse, le Moteur n'a plus de sens d'existence.

```
public class Voiture {
    private Moteur moteur; // composition

    public Voiture() {
        this.moteur = new Moteur(); // Voiture crée le Moteur
    }
    // Si Voiture est détruite → Moteur est détruit aussi
}
```

💡 *La règle du MAX du tableau : • Si MAX vaut 1 → Composition (agrégation forte) • Si MAX est plus grand que 1 → Agrégation (agrégation faible)*

Chapitre 3 — Héritage et Relations de Type

3.1 L'Héritage (Généralisation)

L'héritage dit : « B est un type de A ». B hérite de tous les attributs et méthodes de A.

🔗 *Représentation UML : une ligne avec une flèche TRIANGULAIRE VIDE ( ) du côté du parent.*

```
// A est la classe parent
public class A {
    protected int valeur;
    public void methodeA() { ... }
}

// B hérite de A (B est une sorte de A)
public class B extends A {
    // B a automatiquement 'valeur' et 'methodeA()'
    public void methodeB() { ... }
}
```

3.2 La Réalisation (Interface)

La réalisation dit : « B implémente le contrat défini par A ». C'est la relation entre une interface et sa classe concrète.

🔗 *Représentation UML : une ligne pointillée avec une flèche triangulaire vide (- - - ◁).*

```java
// IA est une interface (contrat)
public interface IA {
    void methodeContrat();
}

// B réalise l'interface IA
public class B implements IA {
    @Override
    public void methodeContrat() {
        // implémentation concrète
    }
}
```

Chapitre 4 — Les Multiplicités

Les multiplicités indiquent combien d'instances d'une classe peuvent être liées à une instance de l'autre. C'est comme les contraintes de votre base de données, mais en visuel.

Notation	Signification	Exemple Java
`1`	Exactement 1	`Voiture v;`
`0..1`	0 ou 1	`Optional<Voiture> v;`
`*` ou `0..*`	0 à infini	`List<Voiture> voitures;`
`1..*`	1 à infini (au moins 1)	`List<Voiture> voitures; // non vide`
`0..2`	0, 1 ou 2	`List<Voiture> v; // max 2`
`2`	Exactement 2	`Voiture[] v = new Voiture[2];`

Chapitre 5 — Le Grand Récapitulatif

Voici comment mémoriser les 4 relations + l'héritage avec des analogies du quotidien :

Relation	Symbole UML	Mot-clé	Analogie
Dépendance	`- - - ->`	*utilise temporairement*	Louer une voiture le week-end
Association	——————>	*connaît / utilise*	Avoir un numéro de téléphone enregistré
Agrégation	◇——————	*contient (faible)*	Une playlist contient des chansons
Composition	◆——————	*contient (forte)*	Un corps contient un cœur

Héritage	————△	*est un type de*	Un chien EST un animal
Réalisation	- - - -△	*implémente le contrat*	Un pompier RÉALISE le contrat de sauveteur

Chapitre 6 — Cas Pratique : Système de Location de Voitures

Mettons tout ensemble. On va modéliser un système de location de voitures. En lisant l'énoncé, on identifie les relations :

- Un Client peut louer 0 à 3 Voitures → Association avec multiplicité
- Une Voiture est composée d'un Moteur → Composition (le moteur ne vit pas sans la voiture)
- Une Agence agrège plusieurs Voitures → Agrégation (les voitures peuvent exister dans d'autres agences)
- Une VoitureElectrique EST une Voiture → Héritage
- Une Voiture utilise temporairement un Service de GPS pour naviguer → Dépendance

```
// COMPOSITION : Moteur est créé et détruit avec Voiture
public class Voiture {
    private Moteur moteur;           // composition ◆
    private String marque;
    private int nbRoues;

    public Voiture(String marque) {
        this.marque = marque;
        this.moteur = new Moteur();   // Voiture crée le Moteur
    }
}

// HÉRITAGE : VoitureElectrique est une Voiture
public class VoitureElectrique extends Voiture {
    private int autonomieKm;
}

// AGRÉGATION : Agence contient des Voitures mais ne les crée pas
public class Agence {
    private List<Voiture> voitures;  // agrégation ◇

    public void ajouterVoiture(Voiture v) {
        voitures.add(v);              // Voiture existe déjà
    }
}

// ASSOCIATION : Client possède une référence vers Voitures
public class Client {
    private List<Voiture> voituresLouees; // 0..3
```

```
}

// DÉPENDANCE : Voiture utilise ServiceGPS temporairement
public class Voiture {
    public void naviguer(ServiceGPS gps) { // dépendance - - ->
        gps.calculerItineraire();
    }
}
```

🎓 *Exercice : à votre tour ! Modélisez un système de bibliothèque. Une Bibliothèque contient des Livres (agrégation ou composition ?). Un Livre peut être emprunté par un Adhérent (quelle relation ?). Un Ebook EST un Livre (quelle relation ?).*

That's it for today's lecture. This was UML.

See you next time. 🎓