Développement Orienté Objets
Notations UML

Petru Valicov
petru.valicov@umontpellier.fr

https://gitlabinfo.iutmontp.univ-montp2.fr/dev-objets

2021-2022

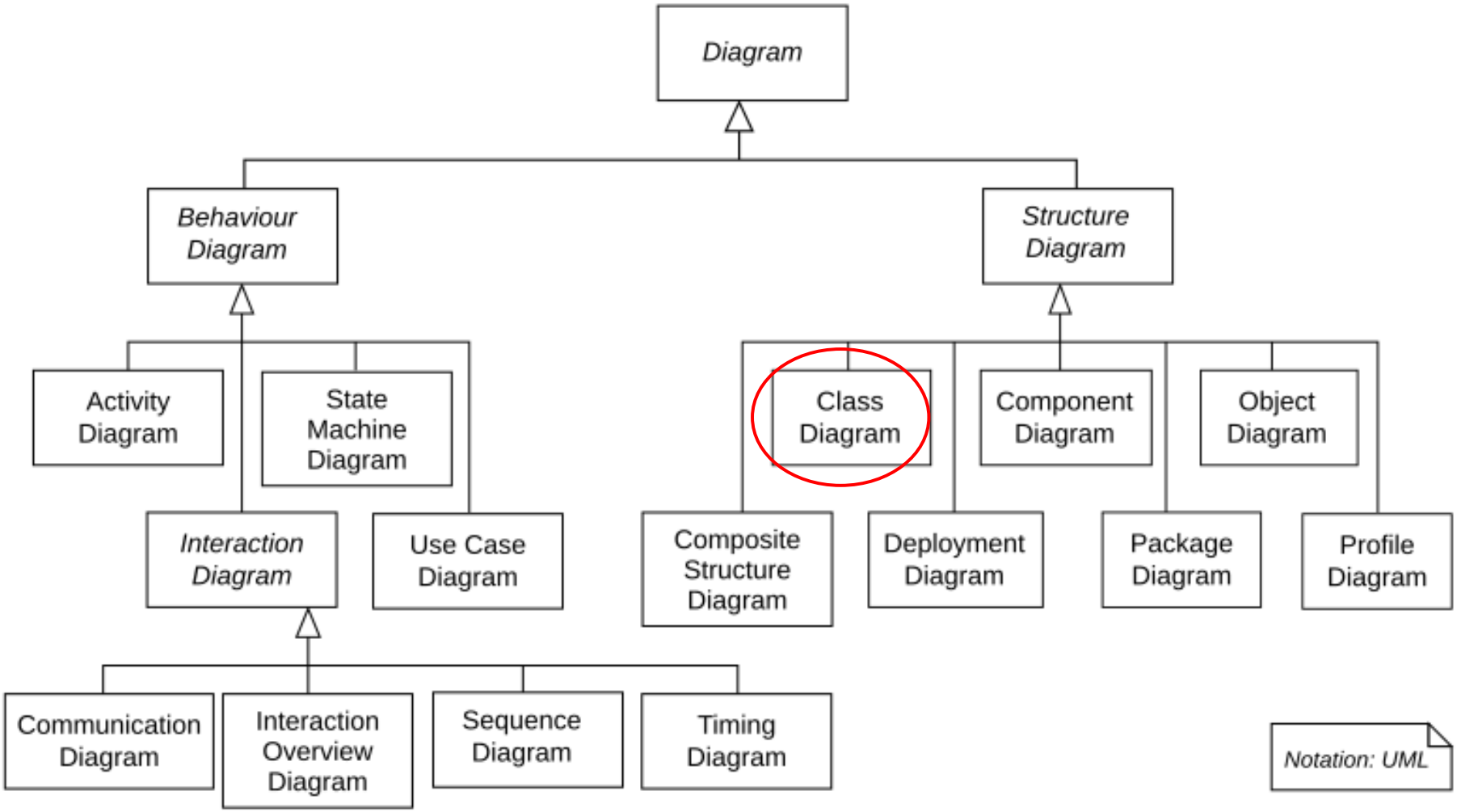
UML - Unified Modeling Language

Définition

UML est un **langage de modélisation** orienté objet qui permet de représenter (de manière graphique) et de communiquer les divers aspects d'un système informatique.

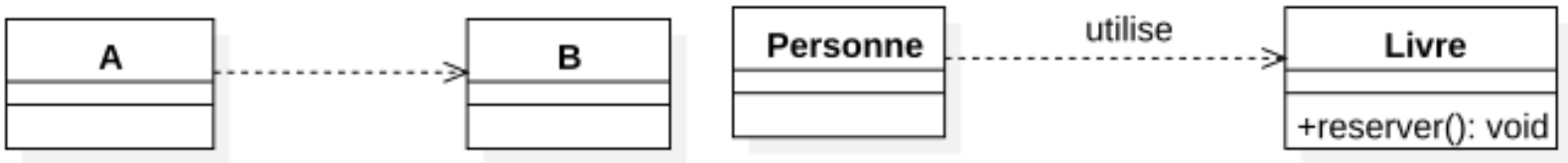
- Apparu au milieu des années '90
- Version actuelle : UML 2.x (standard ISO adopté par l'OMG)
- **langage de modélisation** $\neq$ **langage de programmation**
- C'est juste un ensemble de notations ayant comme base la notion d'*objet*

Les diagrammes UML



Dépendance

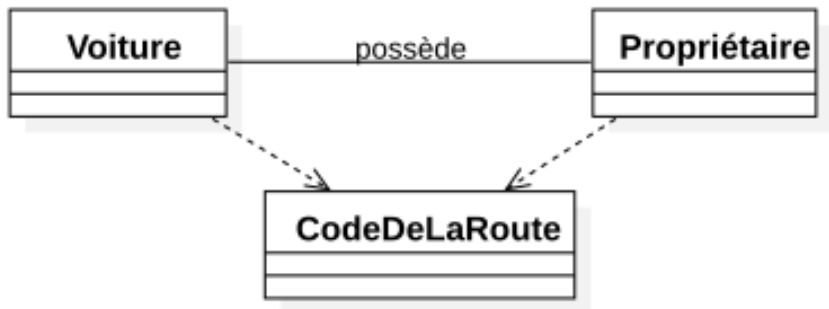
- Relation unidirectionnelle exprimant une dépendance sémantique entre deux classes
- Représenté par un trait discontinu orienté



- Généralement `A` dépend de `B` (on dit aussi `A` *utilise* `B`) si :
  - `A` utilise `B` comme argument dans la signature d'une méthode
  - `A` utilise `B` comme variable locale d'une méthode

Exemple : la modification du code de la route a un impact sur

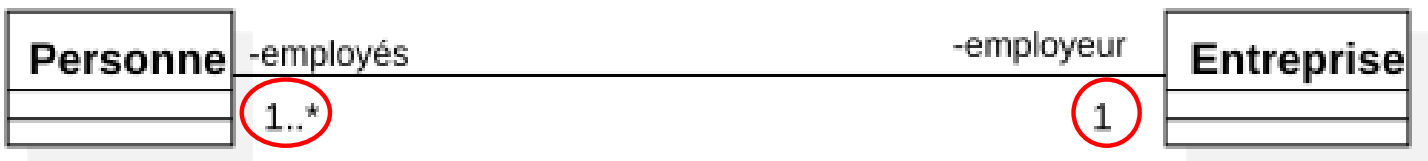
- l'attitude du conducteur
- des caractéristiques des voitures



Relation très générale : par définition toutes les relations possibles entre les classes sont des dépendances

Relations entre classes : association

- relation sémantique entre les **objets** d'une classe
- possède un *rôle* à chaque extrémité
  - décrit comment une classe voit une autre classe à travers l'association
  - **devient le nom d'un attribut en** `Java`
- multiplicité ou cardinalité



Une possible implémentation en `Java` :

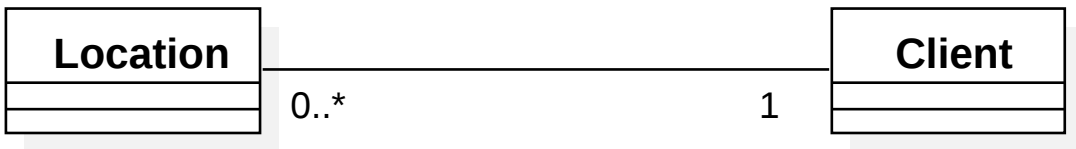
```java
class Personne {
    private Entreprise employeur;
}
```

```java
class Entreprise {
    // un tableau d'employés
    private ArrayList<Personne> employés;
}
```

Multiplicités des associations

La notion de **multiplicité** (ou **cardinalité**) permet de contraindre le nombre d'objets intervenant dans les instanciations des associations.

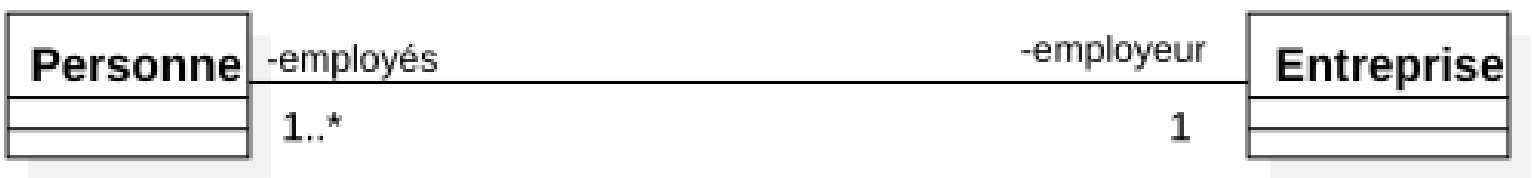
Exemple : *une location est payée par un et un seul client, alors que le client peut réserver plusieurs locations.*



La syntaxe de m

- **1** : toujours un et un seul (dès la création de l'objet)
- **0..1** : zéro ou un
- `m..n` : de m à n (entiers > 0)
- ***** ou ***..0** : de zéro à plusieurs
- **1..*** : au moins un

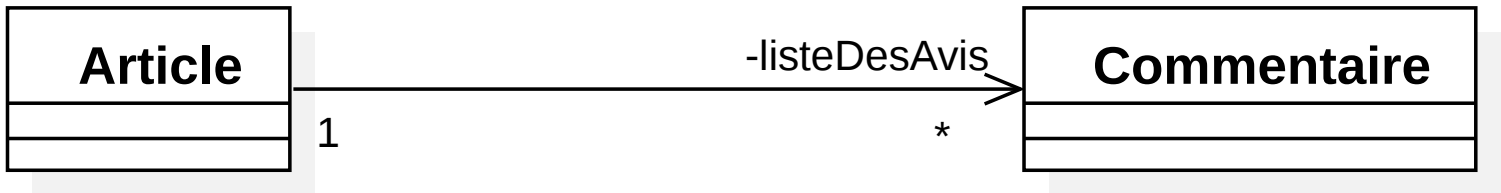
} tableau/liste en Java



Navigabilité d'une association

- La **navigabilité** permet de spécifier dans quel(s) sens il est possible de traverser l'association à l'exécution.

- On restreint la navigabilité d'une association à un seul sens à l'aide d'une flèche.



Exemple : Connaissant un article on connaît les commentaires, mais pas l'inverse. En `Java`, `listeDesAvis` pourrait être un tableau/liste/collection de références vers des objets de type `Commentaire`.

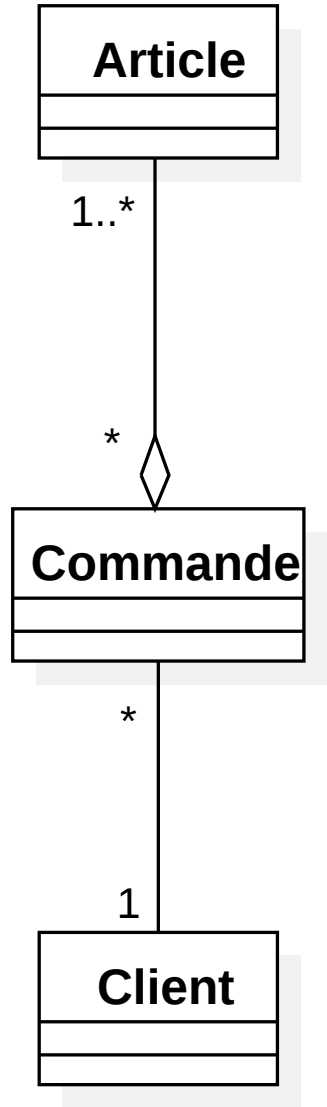
Une possible implémentation en `Java` :

```
public class Artcile {
    private ArrayList<Commentaire> listeDesAvis;
}
```

```
public class Commentaire {

}
```

Associations spéciales : agrégation

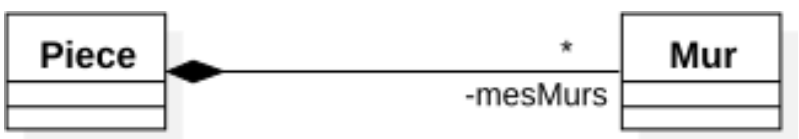
- Une **agrégation** est une forme d'association plus forte que l'association simple
- Représente la relation d'**inclusion faible** d'un élément dans un ensemble
- On représente l'agrégation par l'ajout d'un losange vide du côté de l'agrégat
- On utilise souvent le terme **composition faible**



Associations spéciales : composition

- L'association la plus "forte" : un losange plein
- Décrit une **contenance** structurelle entre instances
- Cardinalité maximum de 1 obligatoire

La **creation**/**destruction** du composant dépend entièrement de l'objet composite.

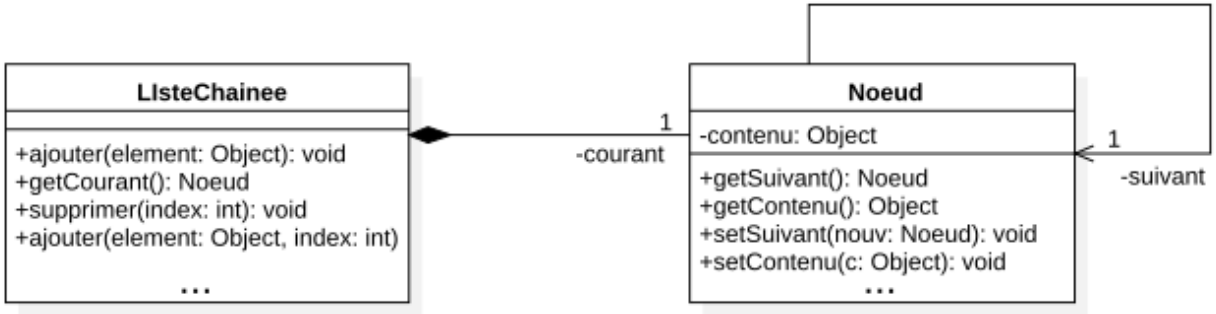


En Java :

```java
public class Piece {
    private ArrayList<Mur> mesMurs;

    public void construire() {
        Mur m = new Mur(); // instanciation des murs
        mesMurs.add(m);
    }
}
```

```java
public class Mur {
}
```

Relations entre classes : association




```java
public class ListeChainee {
   private Noeud courant;

   public void ajouter(Object element) {
       Noeud nouveau = new Noeud(element);
       if (courant == null)
           courant = nouveau;
       else {
           Noeud tmp = courant;
           while (tmp.getSuivant() != null)
               tmp = tmp.getSuivant();
           tmp.setSuivant(nouveau);
       }
   }

   public Noeud getCourant() { return courant; }

   public void supprimer(int index) { ... }

   public void ajouter(Object element, int index) { ... }

   /* d'autres attributs et méthodes */
}
```

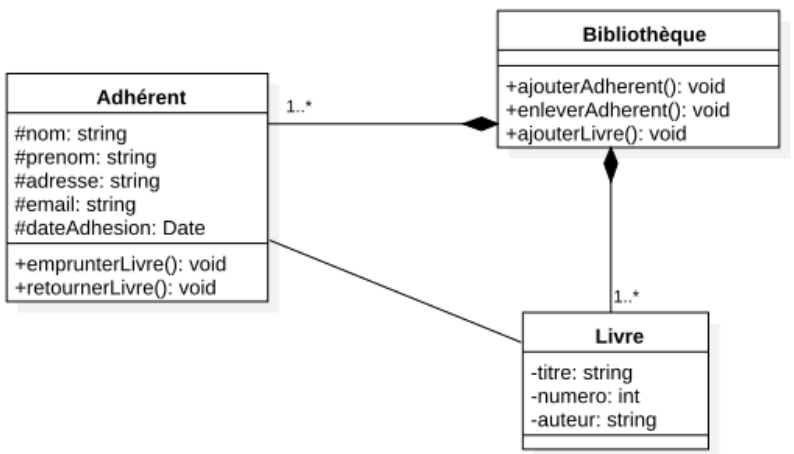
```java
public class Noeud {
   private Noeud suivant;
   private Object contenu;

   // constructeur
   public Noeud(Object contenu) {
      this.contenu = contenu;
   }

   public Noeud getSuivant() { return suivant; }

   public Object getContenu() { return contenu; }

   public void setSuivant(Noeud nouv) {
       suivant = nouv;
   }

   public void setContenu(Object c) {
       contenu = c;
   }

   /* d'autres attributs et méthodes */
}
```

Associations spéciales : composition vs agrégation

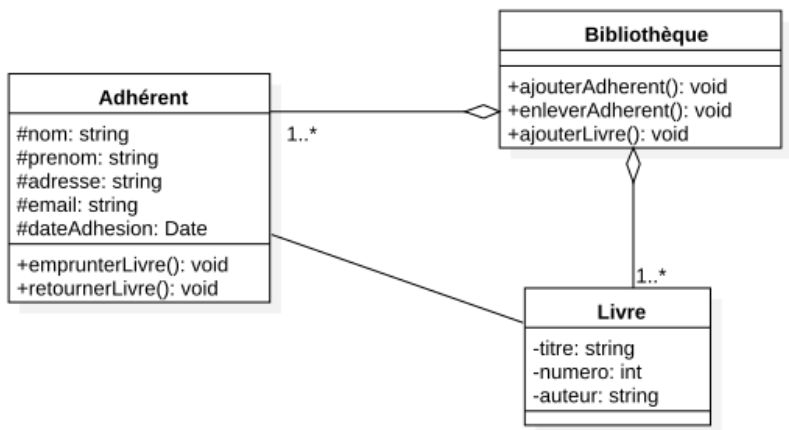
- Dès que il y a la notion de contenance on utilise une agrégation ou une composition
- La composition est aussi dite **agrégation forte**

Comment décider entre la composition et l'agrégation ?

Si les composants ont une autonomie vis-à-vis du composite alors préférez l'agrégation. Mais tout dépend de l'application que vous développez...



vs



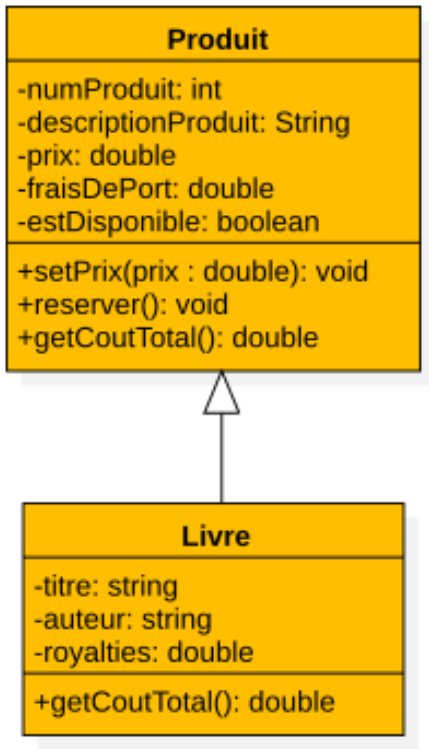
Relation d'héritage

- "Héritage" des propriétés des classes parents
  - La classe **enfant** est la classe spécialisée (ici *Livre*)
  - La classe **parent** est la classe générale (ici *Produit*)

- La classe enfant n'a pas accès aux propriétés privées

- La classe enfant peut redéfinir des méthodes de la classe mère : **polymorphisme**

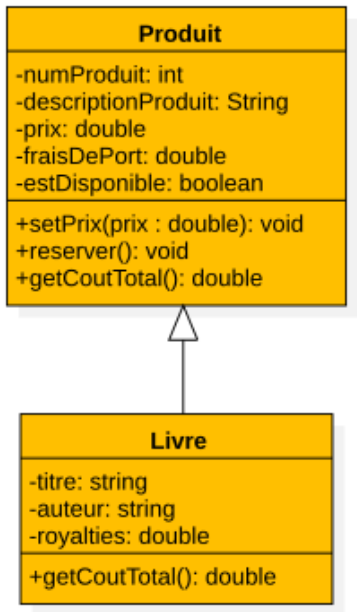
- **Principe de substitution** - toute opération acceptant un objet de type `Produit` doit accepter un objet de type `Livre`.



Attention : TOUTES les propriétés (publiques/privés/protégées) sont héritées dans les classes enfants.

Relation d'héritage : exemple

```java
public class Produit {
    private int numProduit;
    private double prix;
    private boolean estDispo = false;

    public Produit(int numProduit, double prix) {
        this.numProduit = numProduit; this.prix = prix;
    }

    public void setPrix(double prix) { this.prix = prix; }

    public void reserver() { estDispo = false; }

    public double getCoutTotal() { return prix; }
}
```

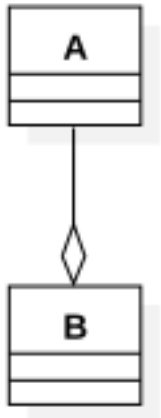



```java
public class Livre extends Produit {
    private String titre, auteur;
    private double royalties;

    public Livre(int numero, double prix, String titre, String auteur, double royalties) {
        super(numero, prix); // appel au constructeur de la classe mère (obligatoire)

        this.titre = titre; this.auteur = auteur; this.royalties = royalties;
    }

    @Override
    public double getCoutTotal() { // redéfinition de la méthode de la classe mère
        return super.getCoutTotal() + royalties;
    }
}
```

Les relations entre les classes

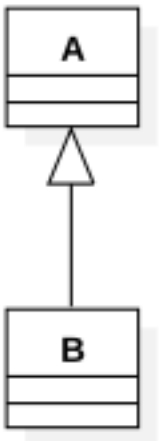
association bidirectionnelle

association unidirectionnelle



agrégation (composition faible)

composition (composition forte)



héritage

réalisation d'interface

dépendance