Analyse et Conception avec UML

# De l'analyse à la conception détaillée : - partie 1 - Enrichissement du diagramme de classe

blay@unice.fr

IUT Nice-Sophia Antipolis

# Rappels sur les concepts d'objets

# Objets

- Un objet est une entité identifiable du monde réel. Il peut avoir une existence physique (un cheval, un livre) ou ne pas en avoir (un texte de loi). Identifiable signifie que l'objet peut être désigné.

- Exemple :
  - Ma jument Jorphée
  - Mon livre sur UML
  - L'article 293B du code des impôts

# Classe

- Classe : Patron pour créer des objets, réservation espace mémoire

*Gaufrier*

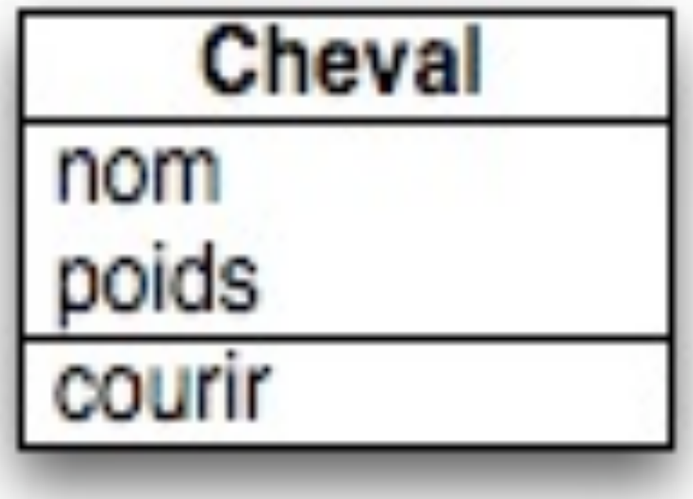
- Objet/Instance

*Gaufre*

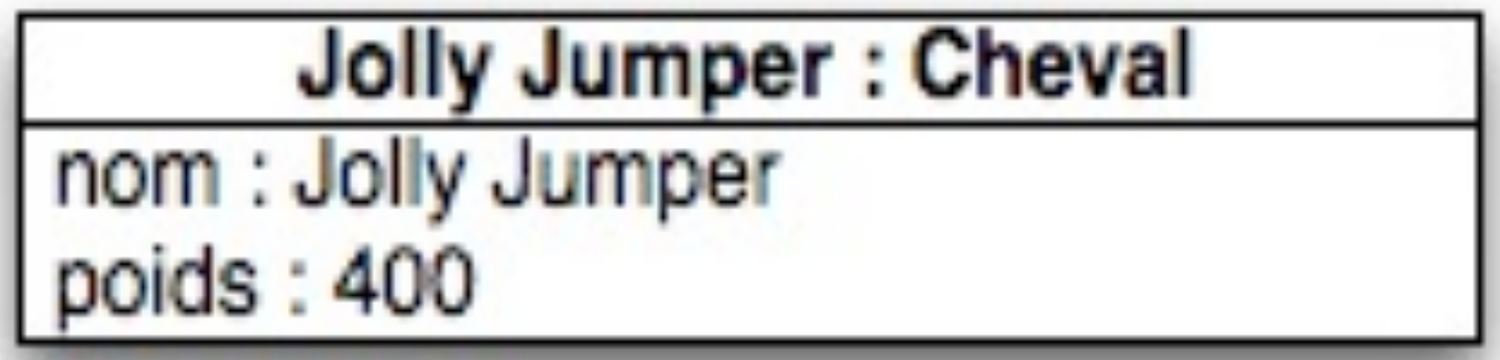
# Vous avez dit classe ?

- Un ensemble
- Un type (abstrait)
- Une structure de données

# Classe/objet




instance de Cheval



# Objets versus Classe

- Il n'est pas possible de créer une classe dynamiquement
  - Mais on peut créer des objets instances d'une classe
  - Mais il n'est pas possible à un objet de changer de classe

- Une classe représente «ses objets»
- Un objet est «instance» d'une seule classe : un seul moule!

# On parle de modélisation par objets

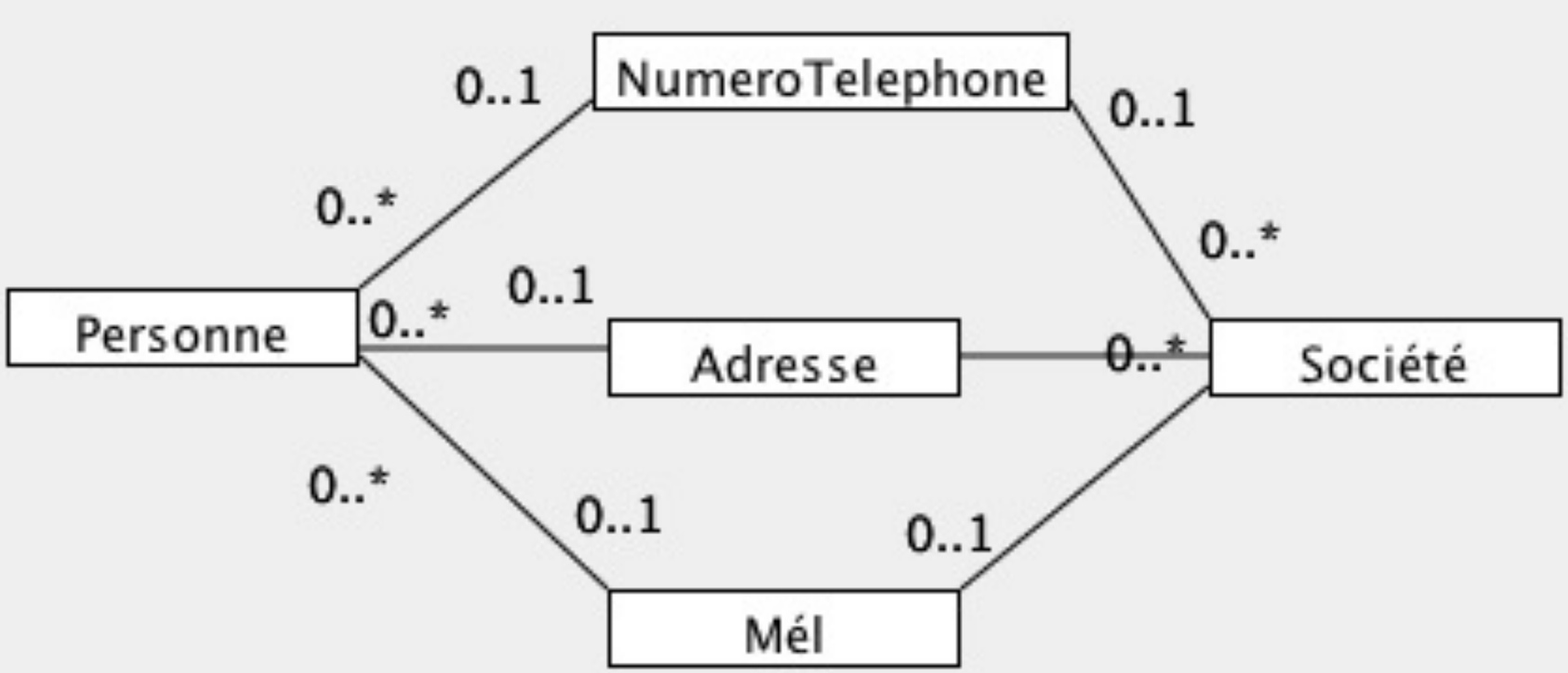
# mais on modélise des classes

# Principes objets

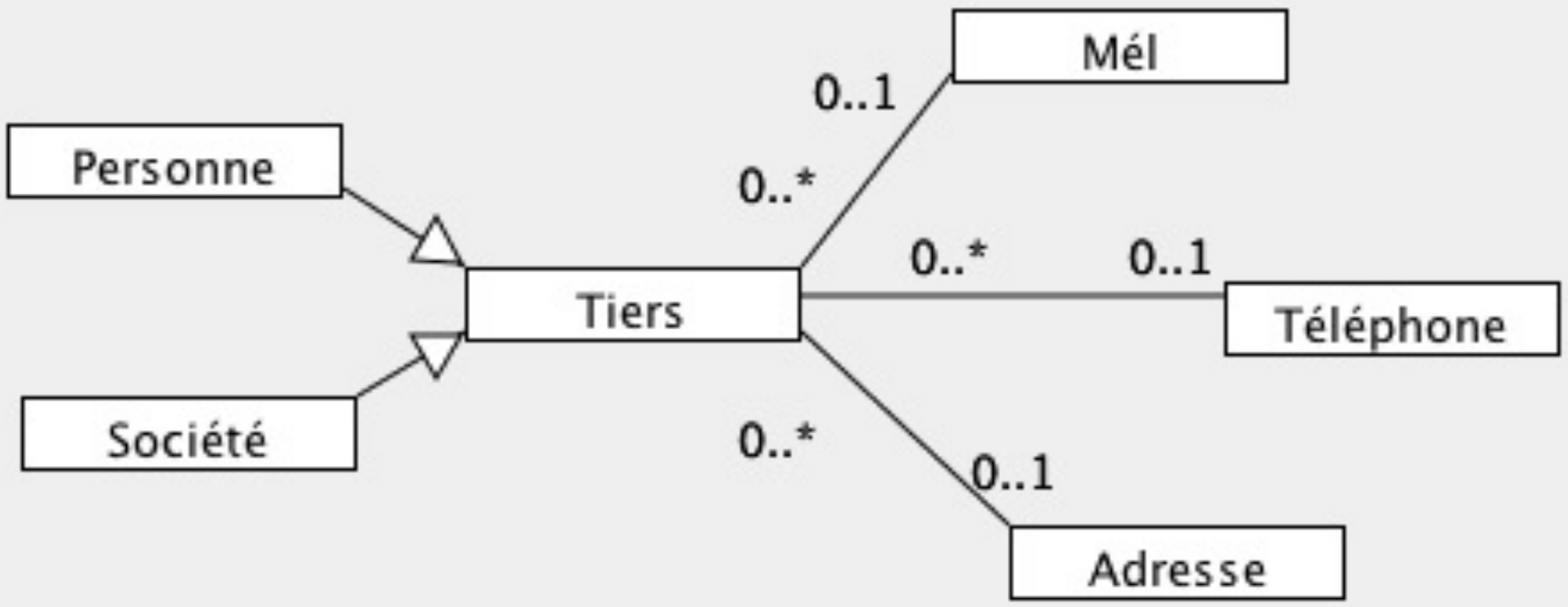
- Encapsulation : regroupement des informations d'état et de comportement sous un nom unique
- Masquage d'information : on ne connaît pas la structure de l'information
- Interface : seuls les services publics (offerts à l'extérieur) sont utilisables
- Envoi de messages : les objets communiquent par messages

Et le Polymorphisme

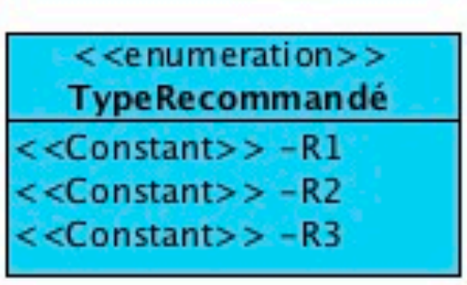
# Polymorphisme : pourquoi faire ?

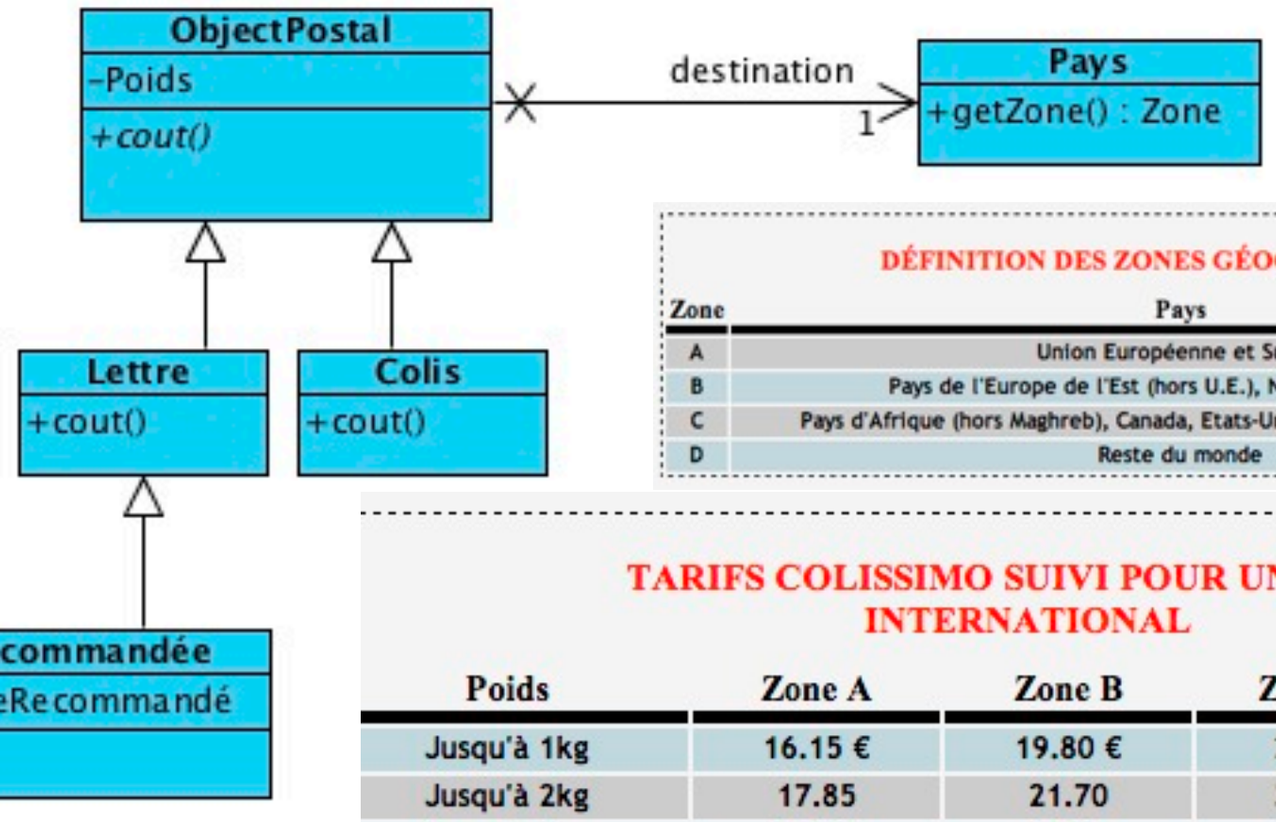


# Généralisation



# Polymorphisme




**DÉFINITION DES ZONES GÉOGRAPHIQUES**

| Zone | Pays |
|---|---|
| A | Union Européenne et Suisse |
| B | Pays de l'Europe de l'Est (hors U.E.), Norvège et Maghreb |
| C | Pays d'Afrique (hors Maghreb), Canada, Etats-Unis, Proche et Moyen-Orient |
| D | Reste du monde |

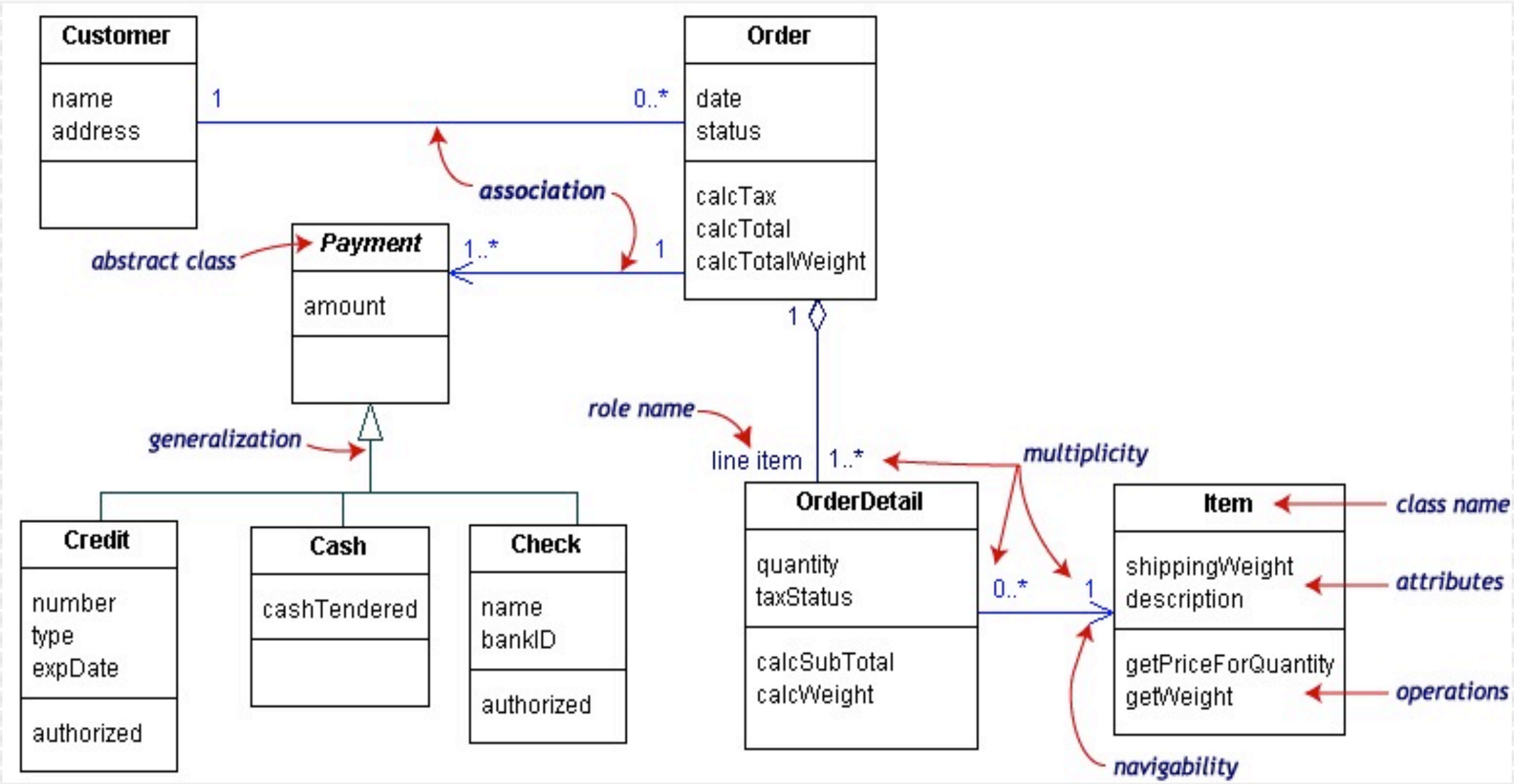
**TARIFS POSTAUX : COURRIER (LETTRE) POUR UN ENVOI EN FRANCE (EN 2014)**

| Poids | Tarif lettre prioritaire (en 2014) | Recommandé R1 (en 2014) | Recommandé R2 (en 2014) | Recommandé R3 (en 2014) |
|---|---|---|---|---|
| Jusqu'à 20g | 0.66 € | +2.84€ | +3.49€ | +4.44€ |
| Jusqu'à 50g | 1.10 € | +2.85€ | +3.50€ | +4.40€ |
| Jusqu'à 100g | 1.65 € | | +3.45€ | +4.35€ |
| Jusqu'à 250g | 2.65 € | | +3.50€ | +4.40€ |
| Jusqu'à 500g | 3.55 € | | +3.45€ | |
| Jusqu'à 1kg | 4.65 € | +2.75€ | +3.40€ | +4.30€ |
| Jusqu'à 2kg | 6.00 € | +2.80€ | | +4.35€ |
| Jusqu'à 3kg | 7.00 € | | | |

**TARIFS COLISSIMO SUIVI POUR UN ENVOI INTERNATIONAL**

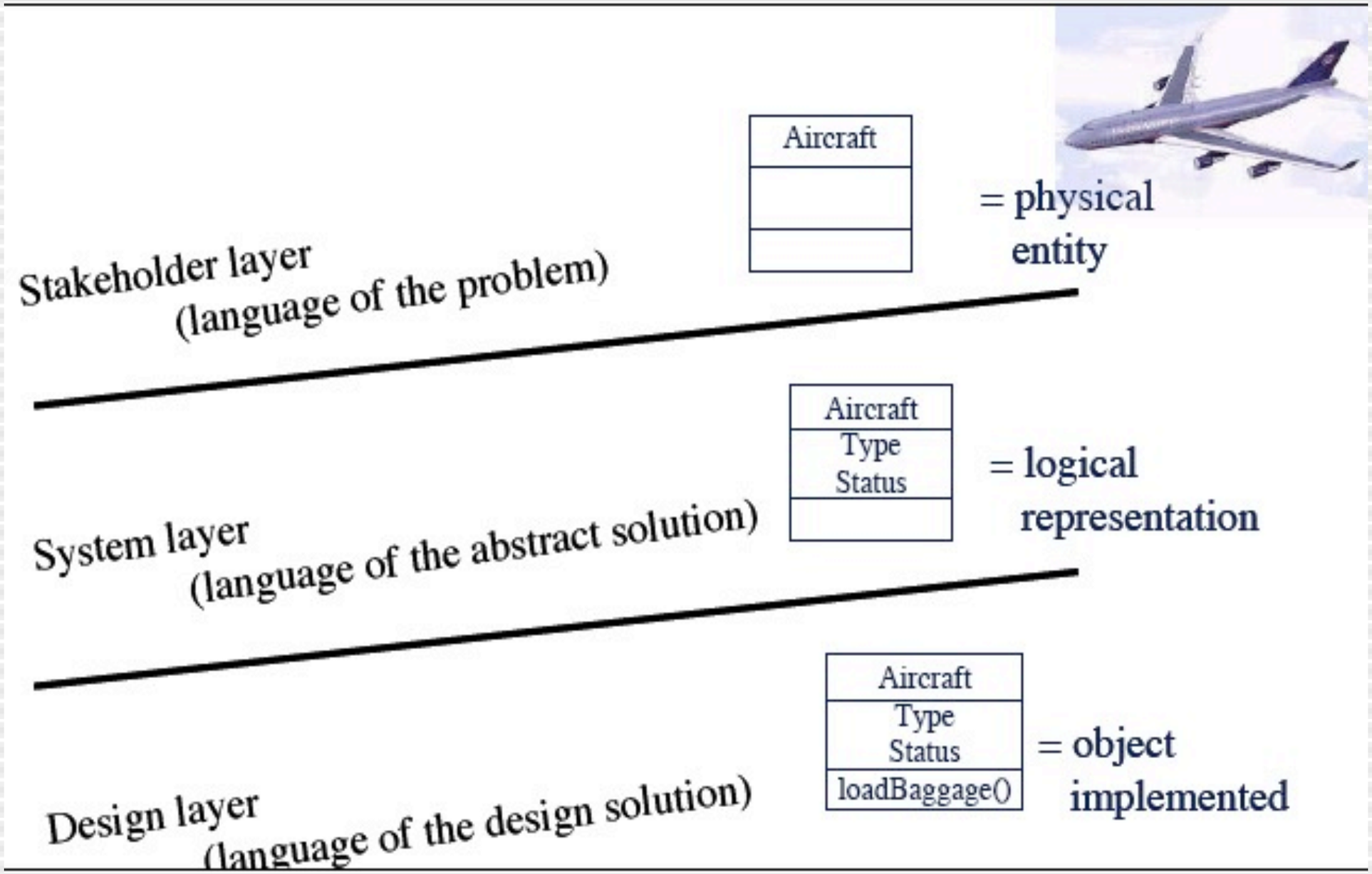
| Poids | Zone A | Zone B | Zone C | Zone D |
|---|---|---|---|---|
| Jusqu'à 1kg | 16.15 € | 19.80 € | 23.20 | 26.40 |
| Jusqu'à 2kg | 17.85 | 21.70 | 31.10 | 39.70 |
| Jusqu'à 3kg | 21.55 | 26.25 | 40.90 | 52.90 |
| Jusqu'à 4kg | 25.25 | 30.80 | 50.70 | 66.10 |
| Jusqu'à 5kg | 28.95 | 35.35 | 60.50 | 79.30 |
| Jusqu'à 6kg | 32.65 | 39.90 | 70.30 | 92.50 |
| Jusqu'à 7kg | 36.35 | 44.45 | 80.10 | 105.70 |
| Jusqu'à 8kg | 40.05 | 49.00 | 89.90 | 118.90 |
| Jusqu'à 9kg | 43.75 | 53.55 | 99.70 | 132.10 |
| Jusqu'à 10kg | 47.45 | 58.10 | 109.50 | 145.30 |
| Jusqu'à 15kg | 54.65 | 68.50 | 133.60 | 171.30 |
| Jusqu'à 20kg | 61.85 | 78.90 | 157.70 | 197.30 |
| Jusqu'à 25kg | 69.05 | - | - | - |
| Jusqu'à 30kg | 76.25 | - | - | - |

# Résumé de notations



# De l'analyse à la conception des classes : Principes

# De l'analyse à la conception des classes



# Différents niveaux

- Une classe peut être spécifiée à différents niveaux :
  - niveau application : classe métier ou classe d'analyse
  - niveau implémentation :
    - traduction informatique d'une classe métier
    - insertion de classes dédiées (par ex. les conteneurs ou les collections)
    - mapping sur un modèle physique pas forcément objet (base de données, fichiers, XML, WSDL, ...)

# Exemple

**Analysis**

| Order |
| --- |
| Placement Date<br>Delivery Date<br>Order Number |
| Calculate Total<br>Calculate Taxes |

**Design**

| Order |
| --- |
| - deliveryDate: Date<br>- orderNumber: int<br>- placementDate: Date<br>- taxes: Currency<br>- total: Currency |
| # calculateTaxes(Country, State): Currency<br># calculateTotal(): Currency<br>getTaxEngine() |

# Analyse/Conception

Diagramme d‘analyse≠diagramme de conception

- Typage des méthodes et des résultats
- Sens de navigation des relations
- Rajout de détails
- Ajout de classes « utilitaires »
- Prise en compte de contraintes d’implémentation

# De l'analyse à la conception

- Des diag. de séquences aux Diag. de classes

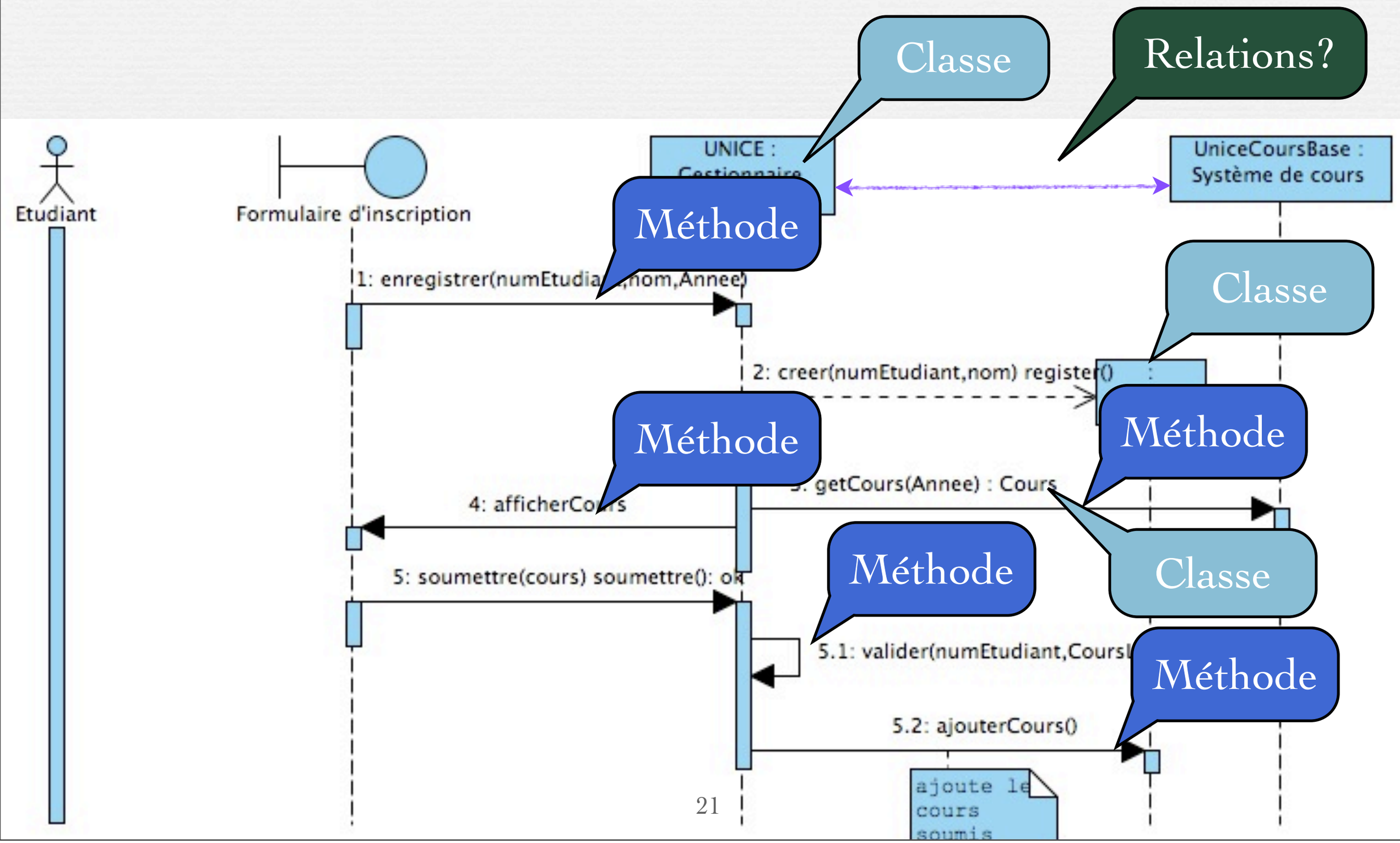
- Structuration en package
- Structuration en package et réutilisation

- Choix des itérations
- Diag. de séquence en conception
- Architecture
- Diagrammes de classes en conception

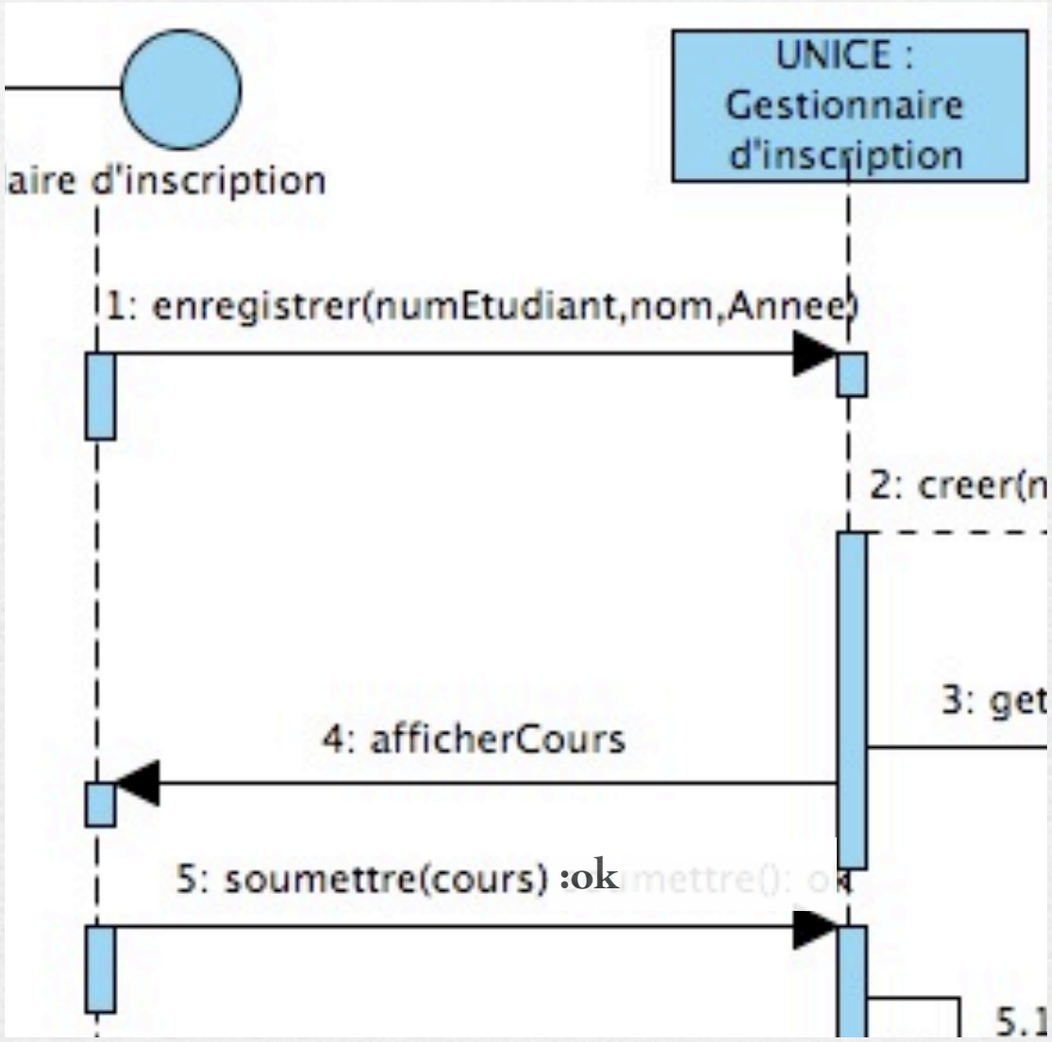
# Des diagrammes de séquence

# aux diagrammes de classes

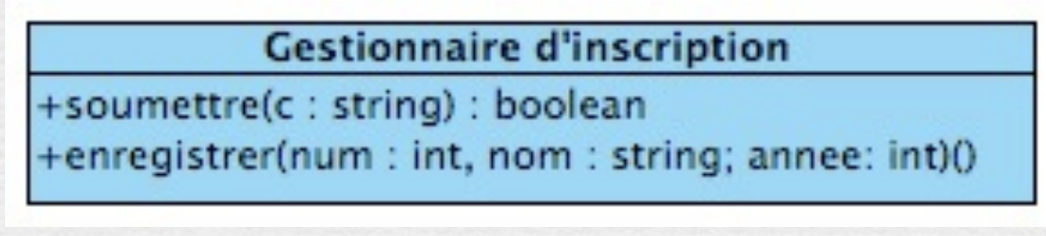
# Des diagrammes de séquence aux classes



# Opération

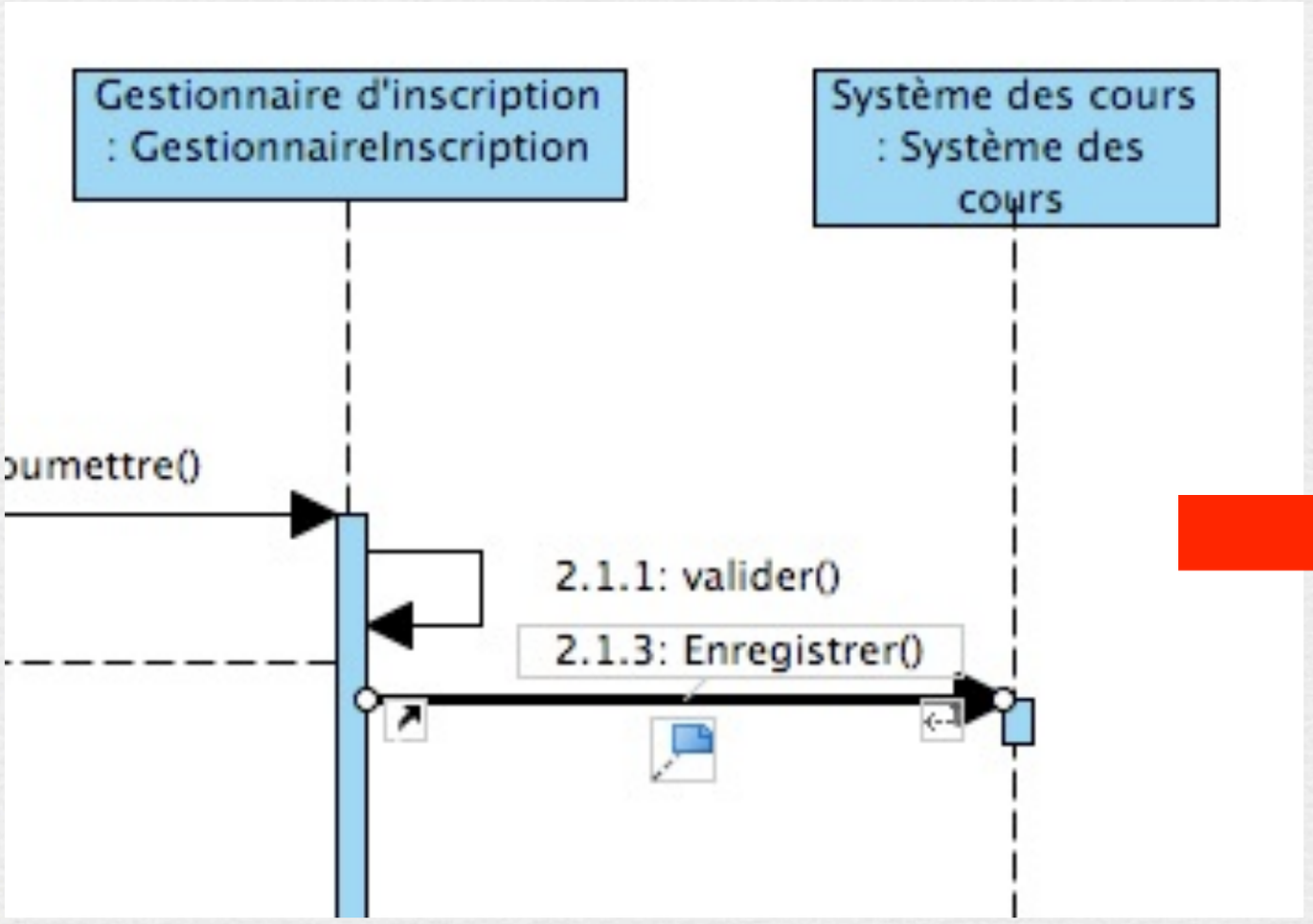


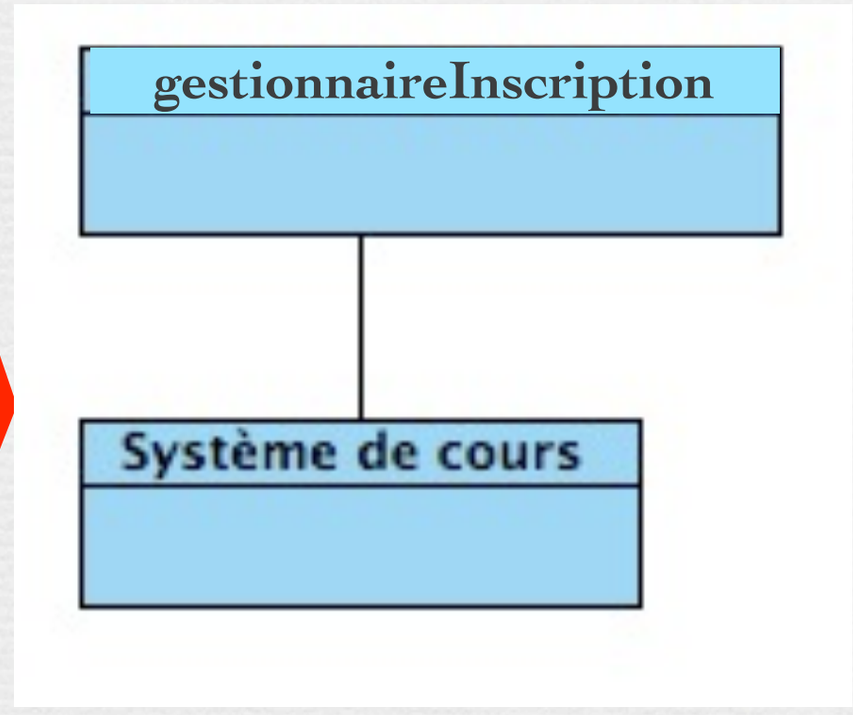

- Le comportement d'une classe est constitué de ses opérations
- On identifie les opérations en examinant les diagrammes de séquences



# Relations

- On identifie les relations en examinant les diagrammes de séquence
  - Si deux objets doivent communiquer, il doit exister un chemin entre eux



# Structuration en packages

# Structuration en packages


## Cohérence et Indépendance

- Minimiser les dépendances
- Eviter les dépendances mutuelles

A **package** in the Unified Modeling Language is used "to group elements, and to provide a namespace for the grouped elements".
A package may contain other packages, thus providing for a hierarchical organization of packages.


# Structuration en packages


# Structuration en packages

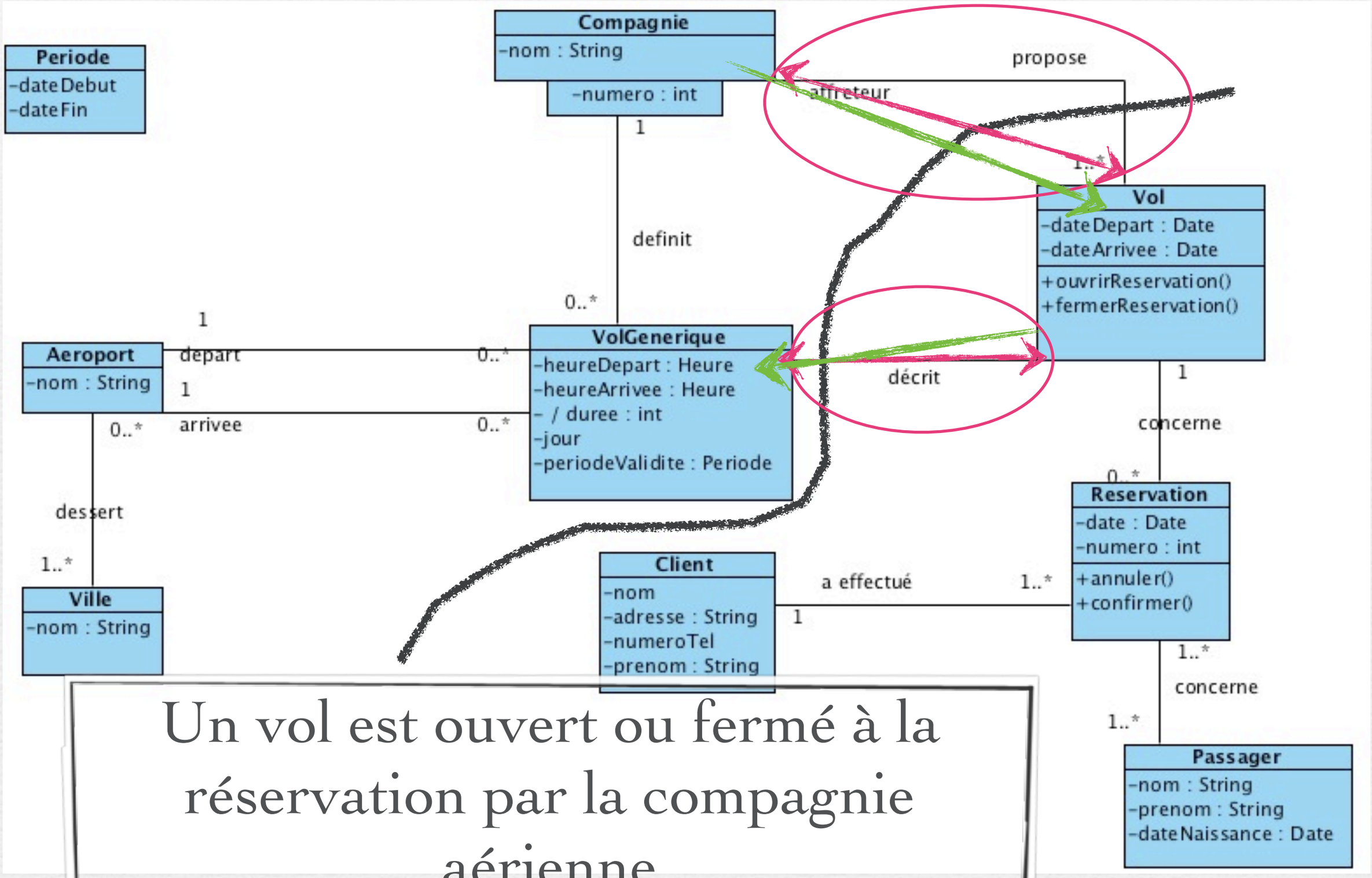

# Structuration en packages


# Structuration en packages


# Structuration en packages

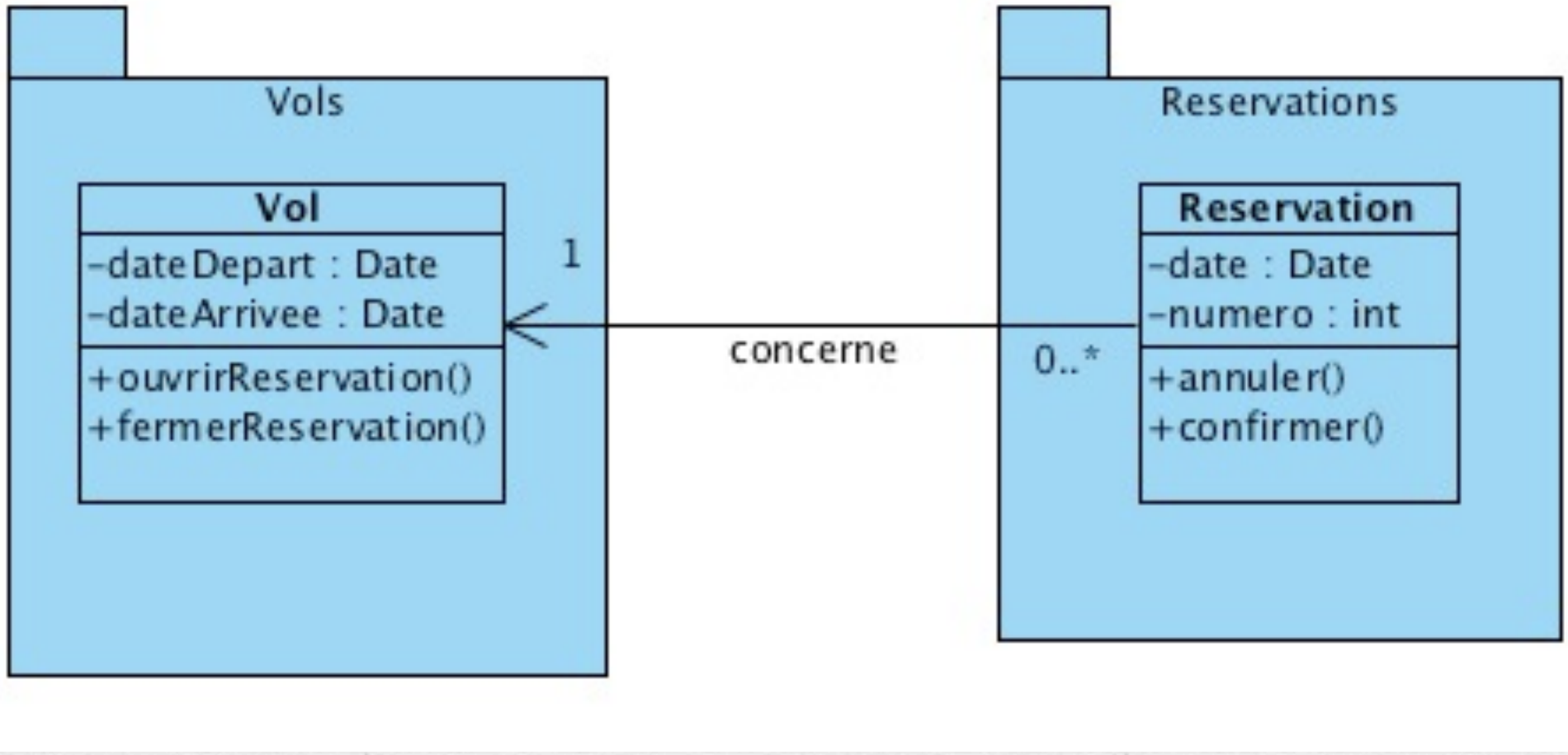

# Structuration en packages


# Structuration en packages

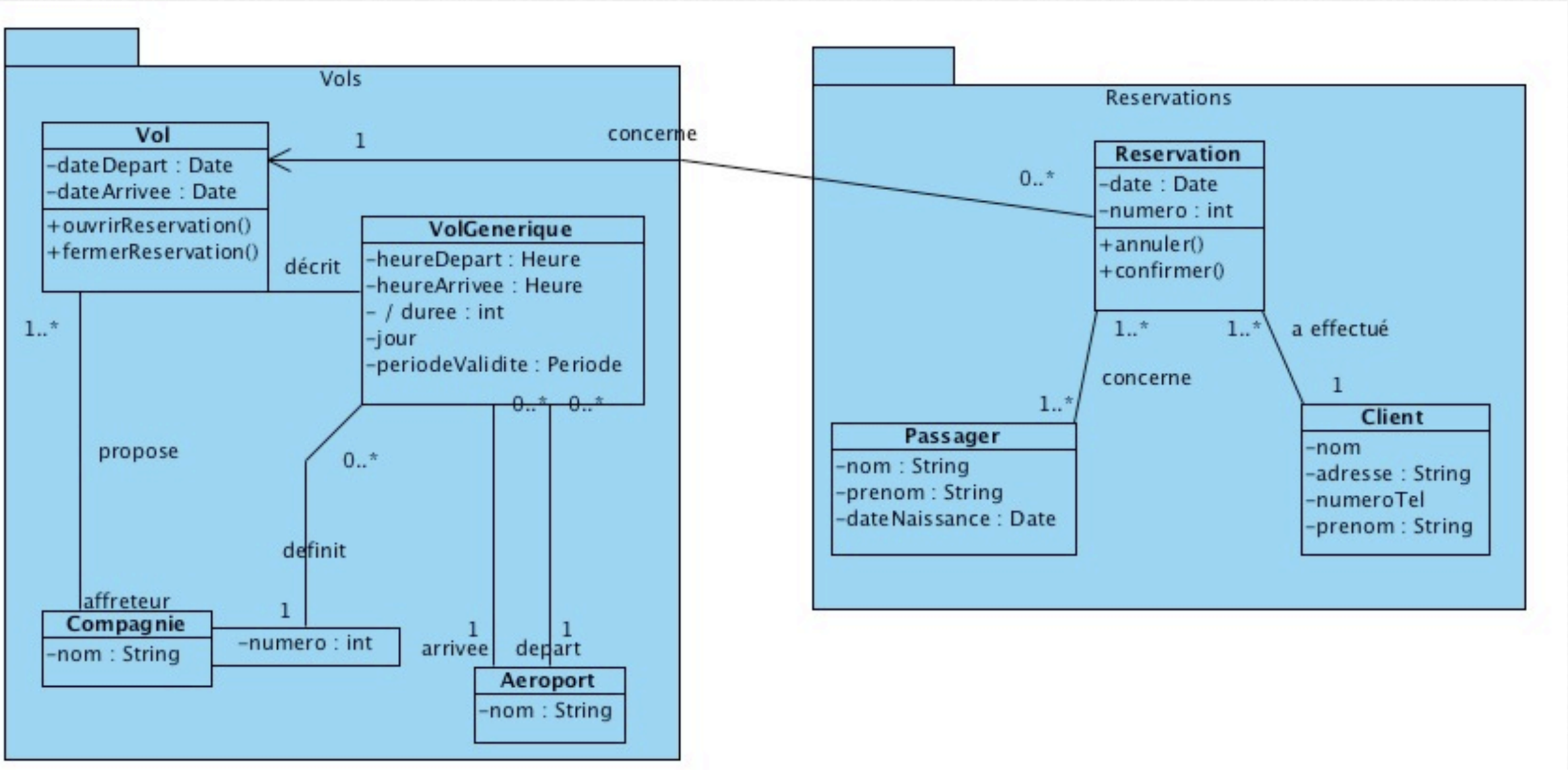



Un vol est ouvert ou fermé à la réservation par la compagnie aérienne


# Structuration en packages



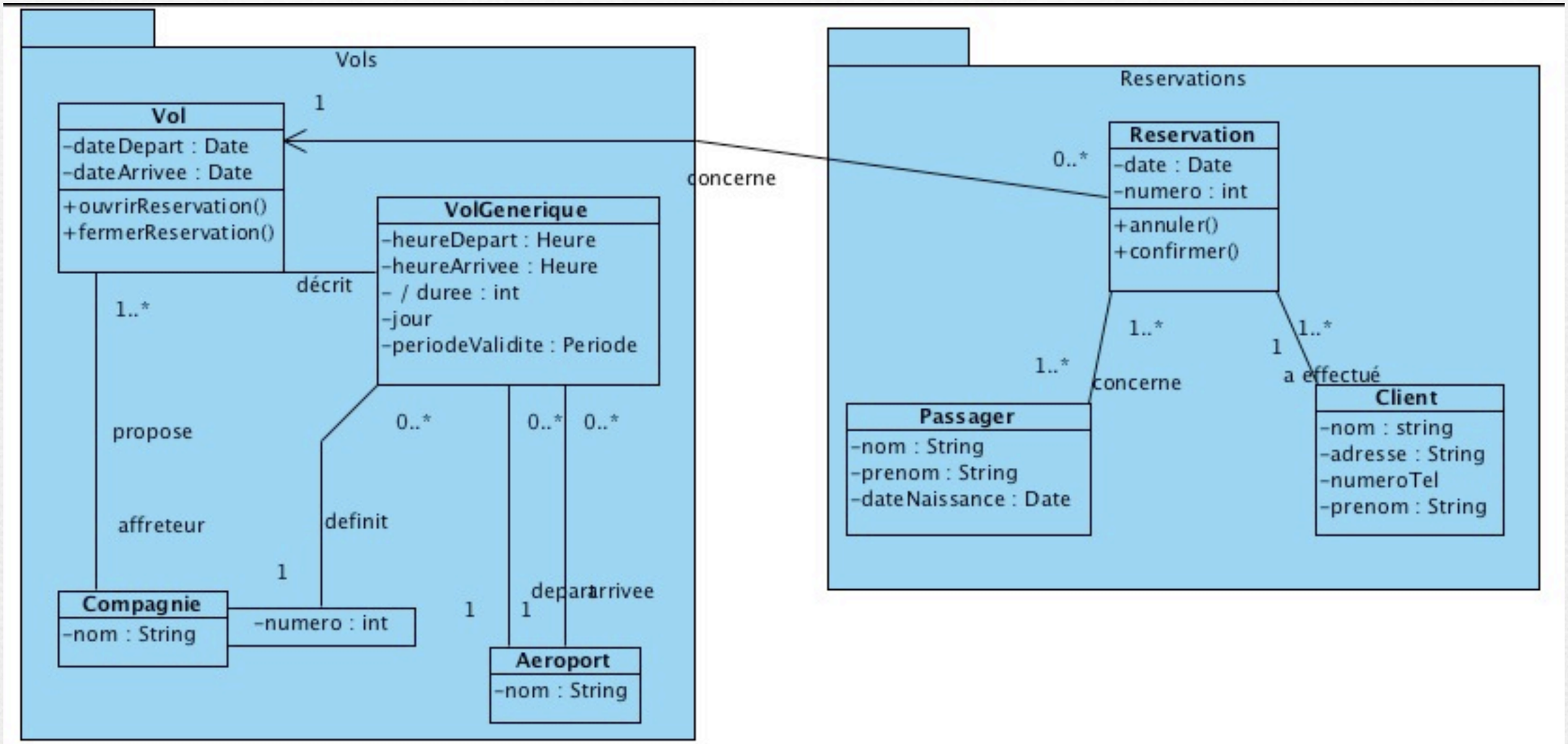

# Structuration en packages



# Structuration des classes en packages et réutilisation

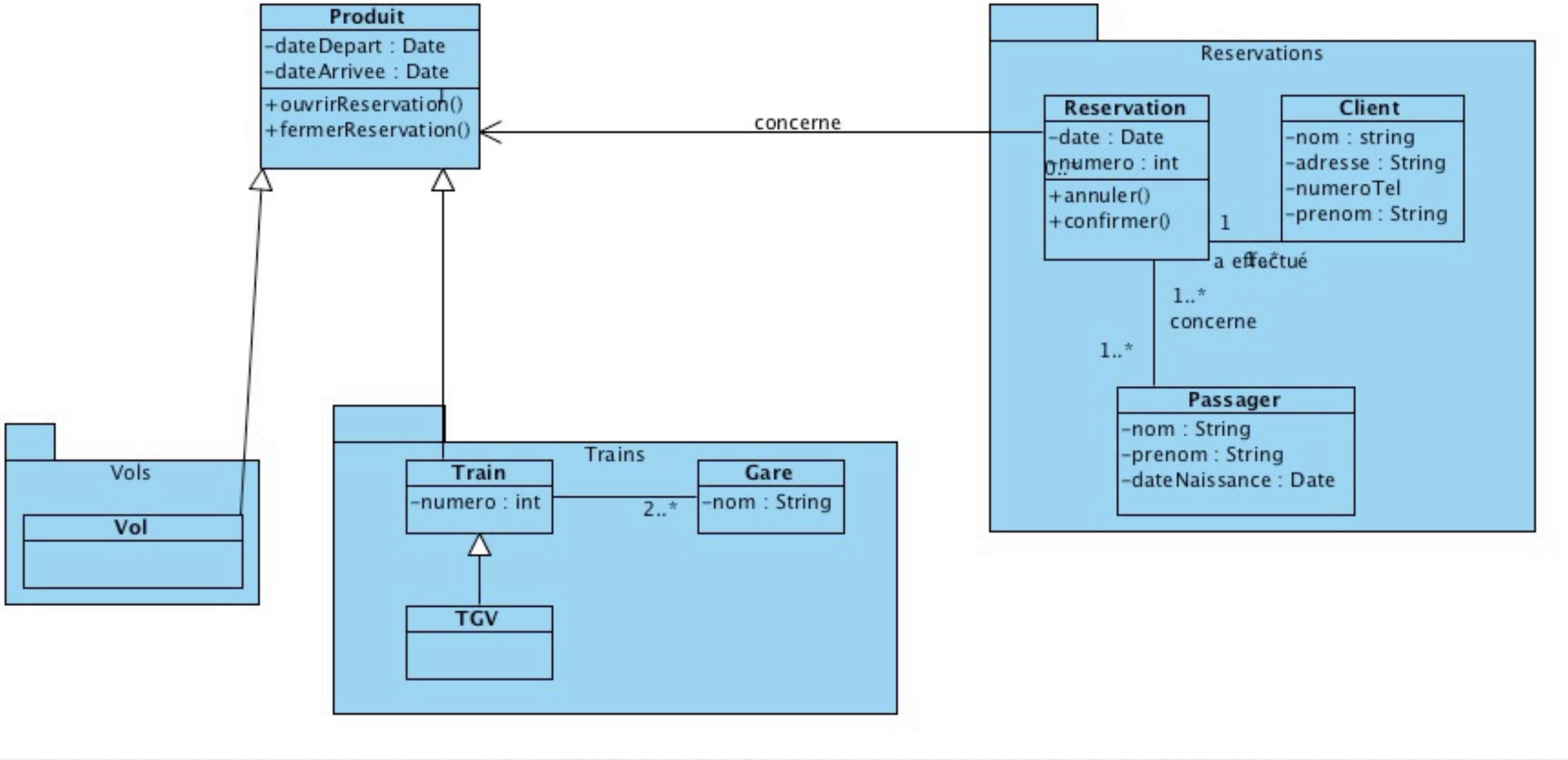
# Spécialisation

- La société qui prend en charge les réservations de vols, veut prendre en charge des réservations de train.



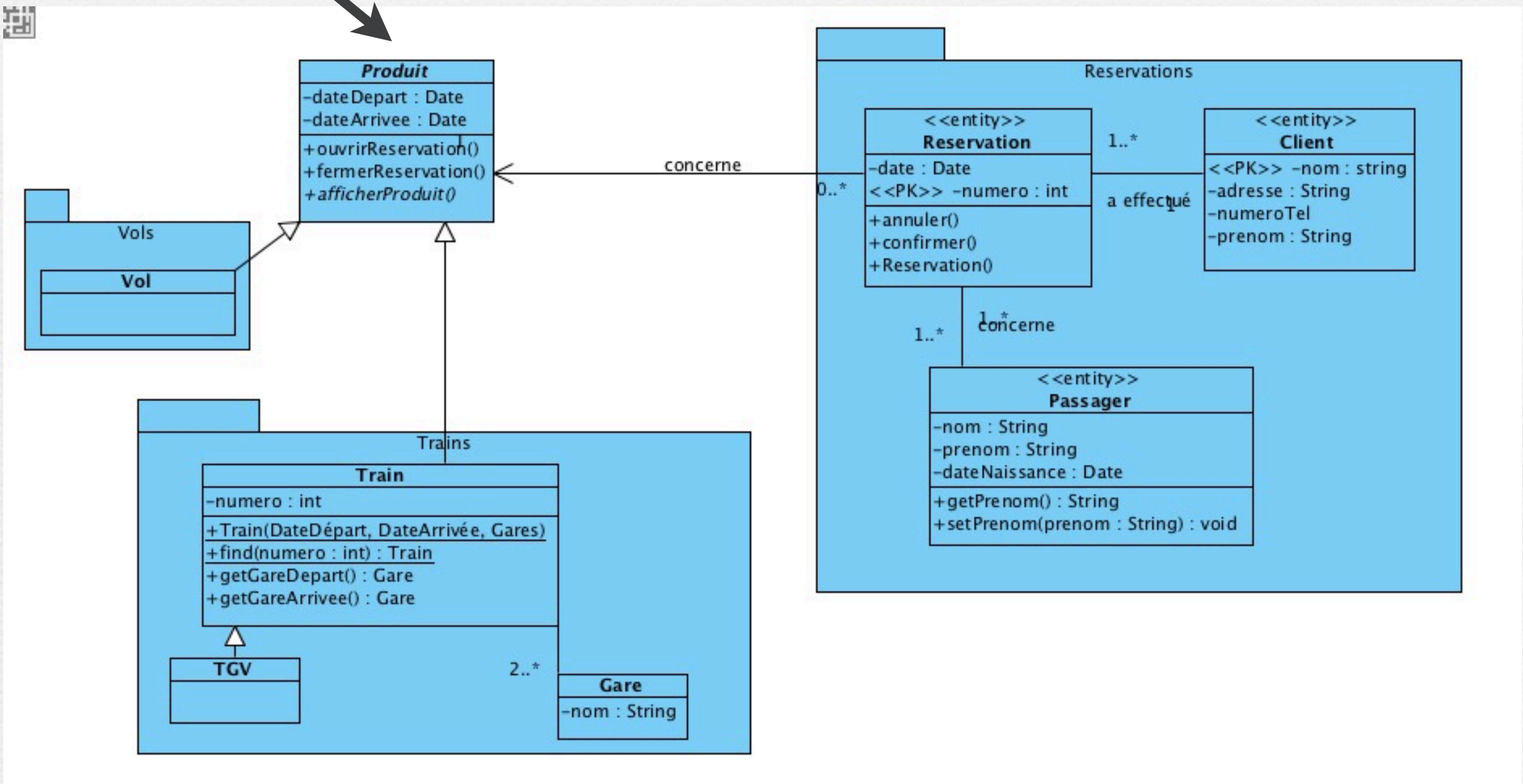
# Spécialisation

- La société qui prend en charge les réservations de vols, veut prendre en charge des réservations de train.



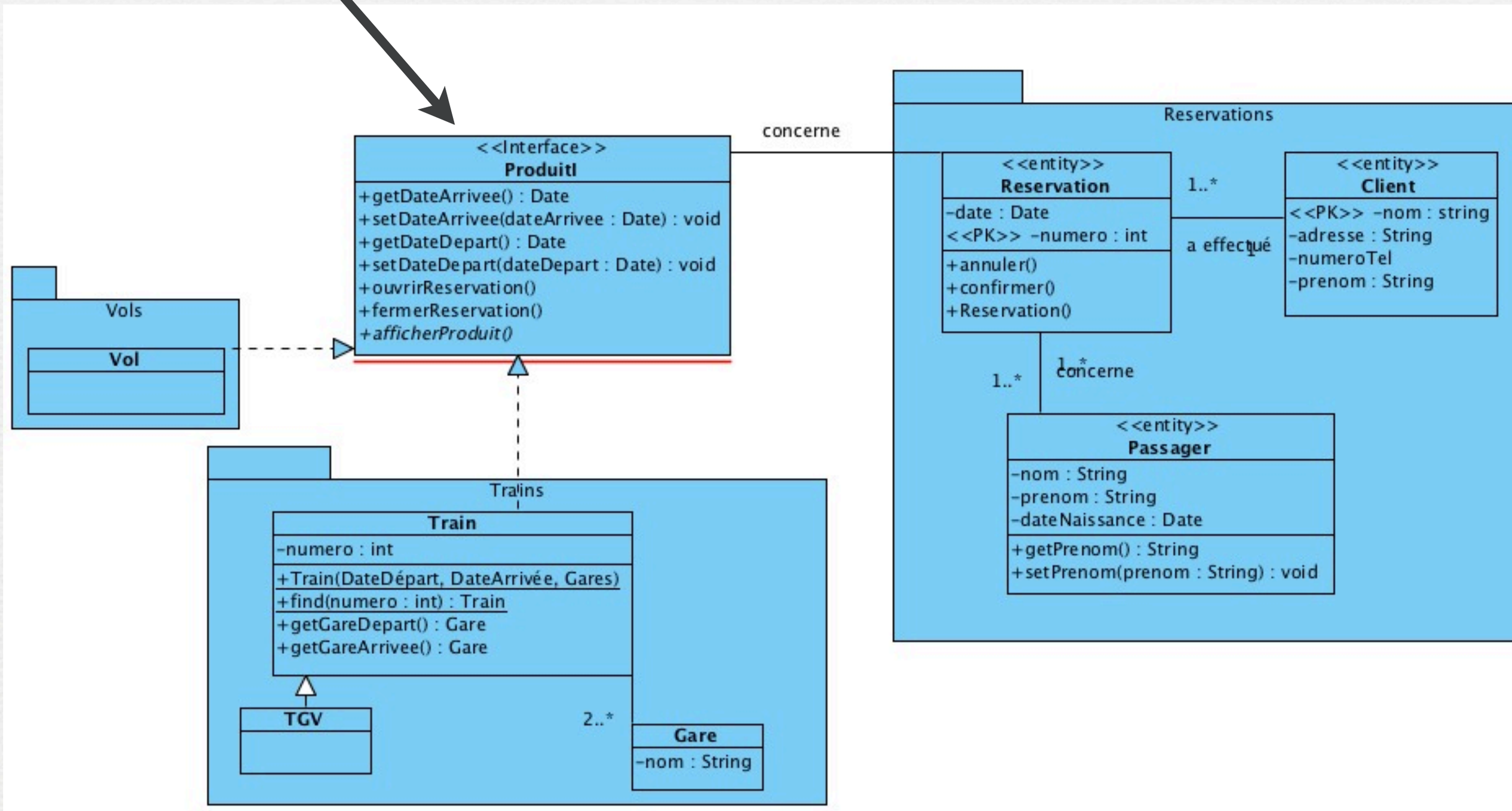
# Spécialisation

Abstraite ?



# Réalisation

Interface ?



# Compléments sur les classes : vers la mise en oeuvre

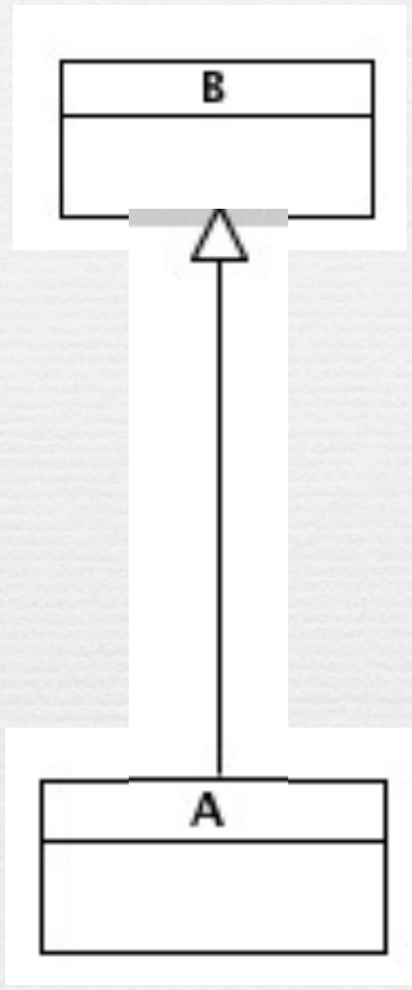
# Vers la mise en oeuvre des classes

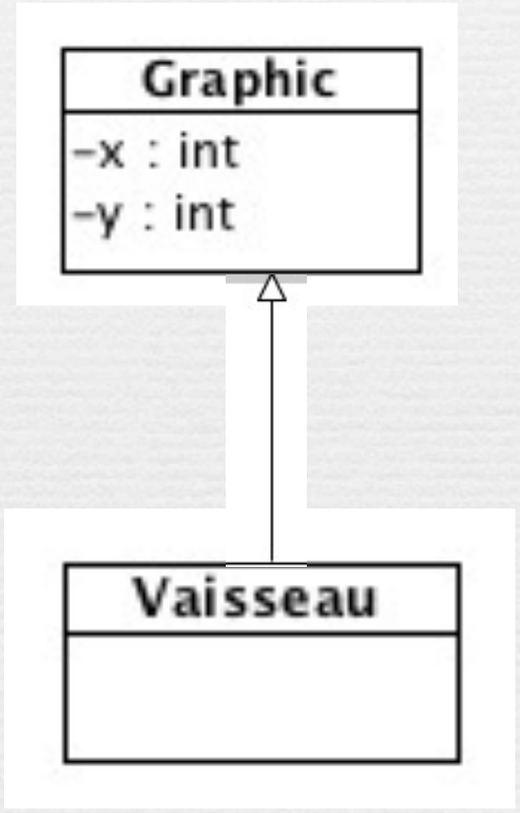
- Visibilité
- Abstraction
- Généralisation
- Packages
- Attributs et Opérations* de Classes
- Transformations des associations
- Anti-Patterns

Opération : terme générique désignant le plus souvent des méthodes

# Généralisation

Quand A hérite de B,
les objets instances de A
possèdent les propriétés de B

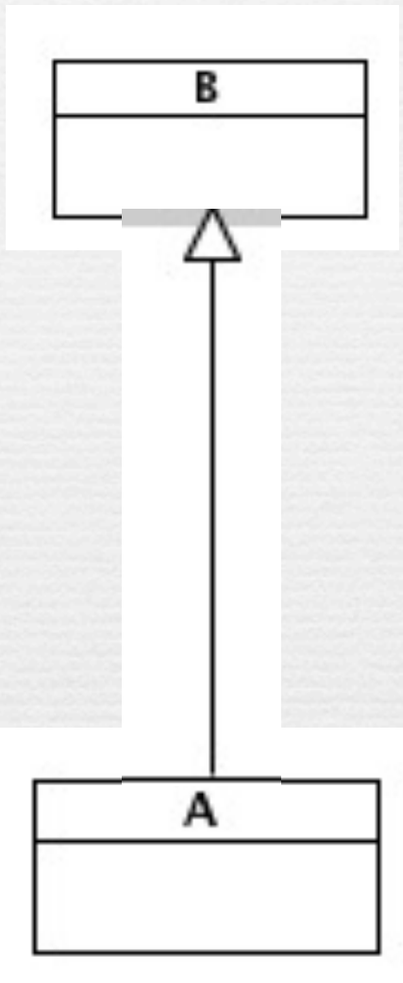


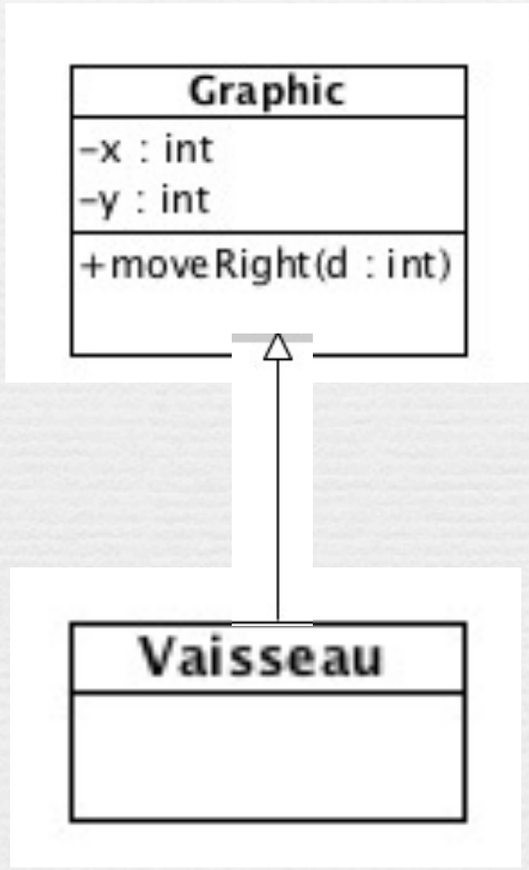




Inspiré de Xavier Blanc : https://www.youtube.com/watch?v=bLge8v-czPg

# Généralisation

Quand A hérite de B, les objets instances de A savent réaliser les méthodes définies par B




Code ?

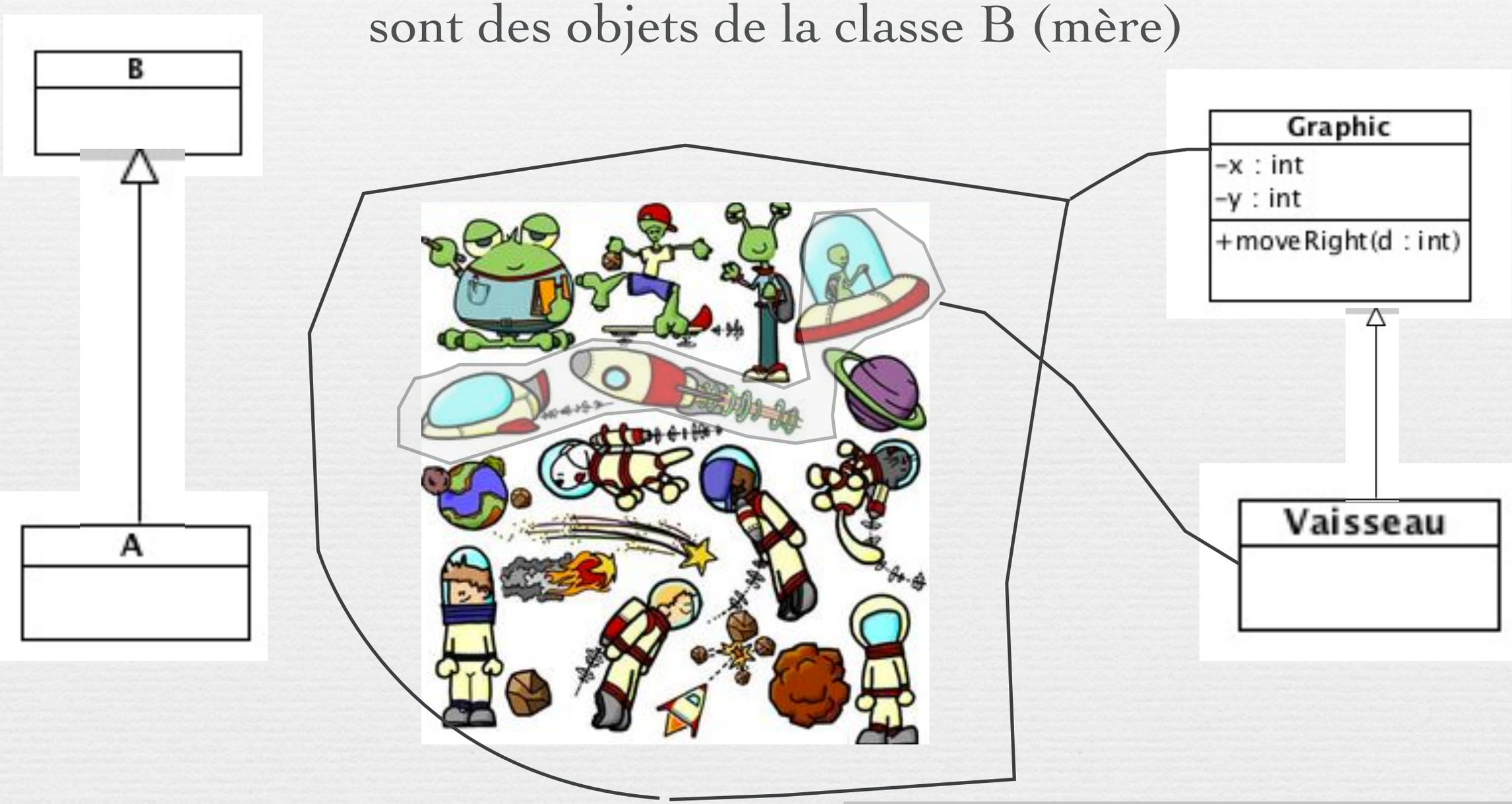
Que fait :

```
Vaisseau v = new Vaisseau();
v.moveRight(10);
```

Xavier Blanc : https://www.youtube.com/watch?v=bLge8v-czPg

# Généralisation : Signification ensembliste

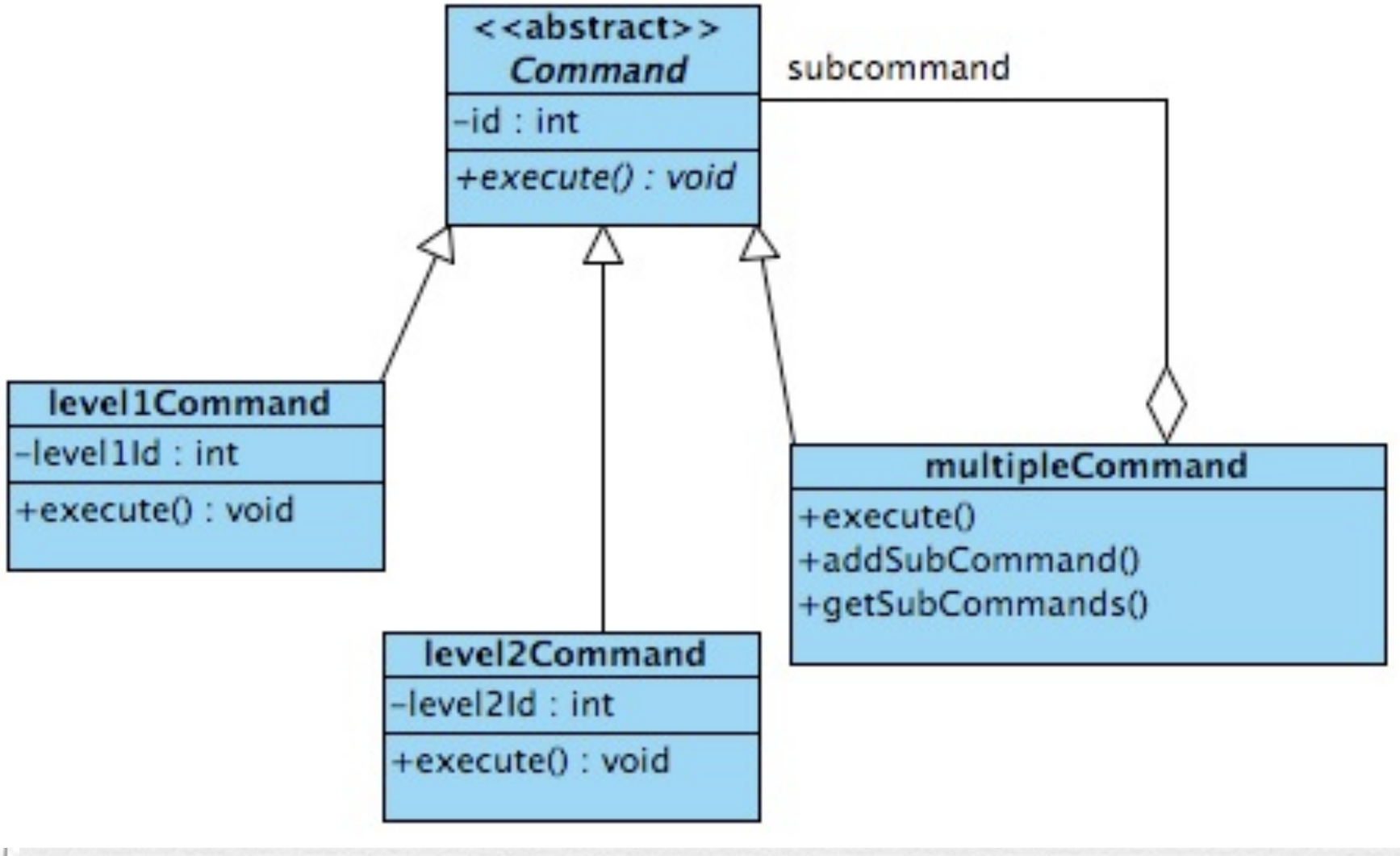
Les objets de classe A (fille)
sont des objets de la classe B (mère)

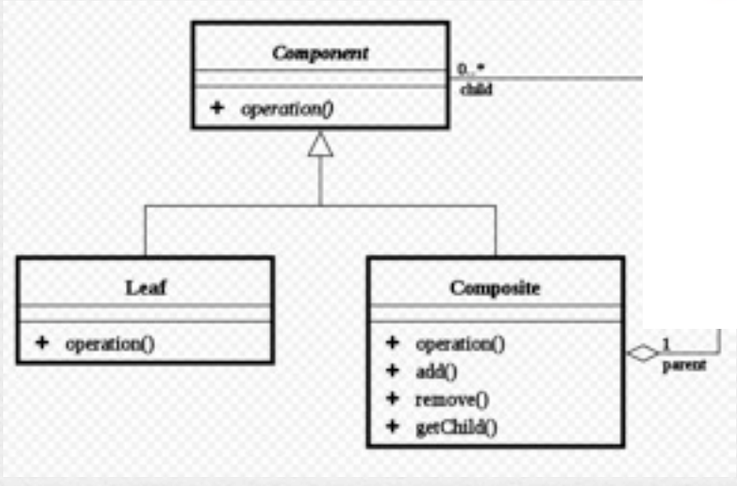



```
Graphic g  = new Vaisseau();
g.moveRight(10);
```

Xavier Blanc : https://www.youtube.com/watch?v=bLge8v-czPg

# Pattern Composite...



# Vers la mise en oeuvre des classes

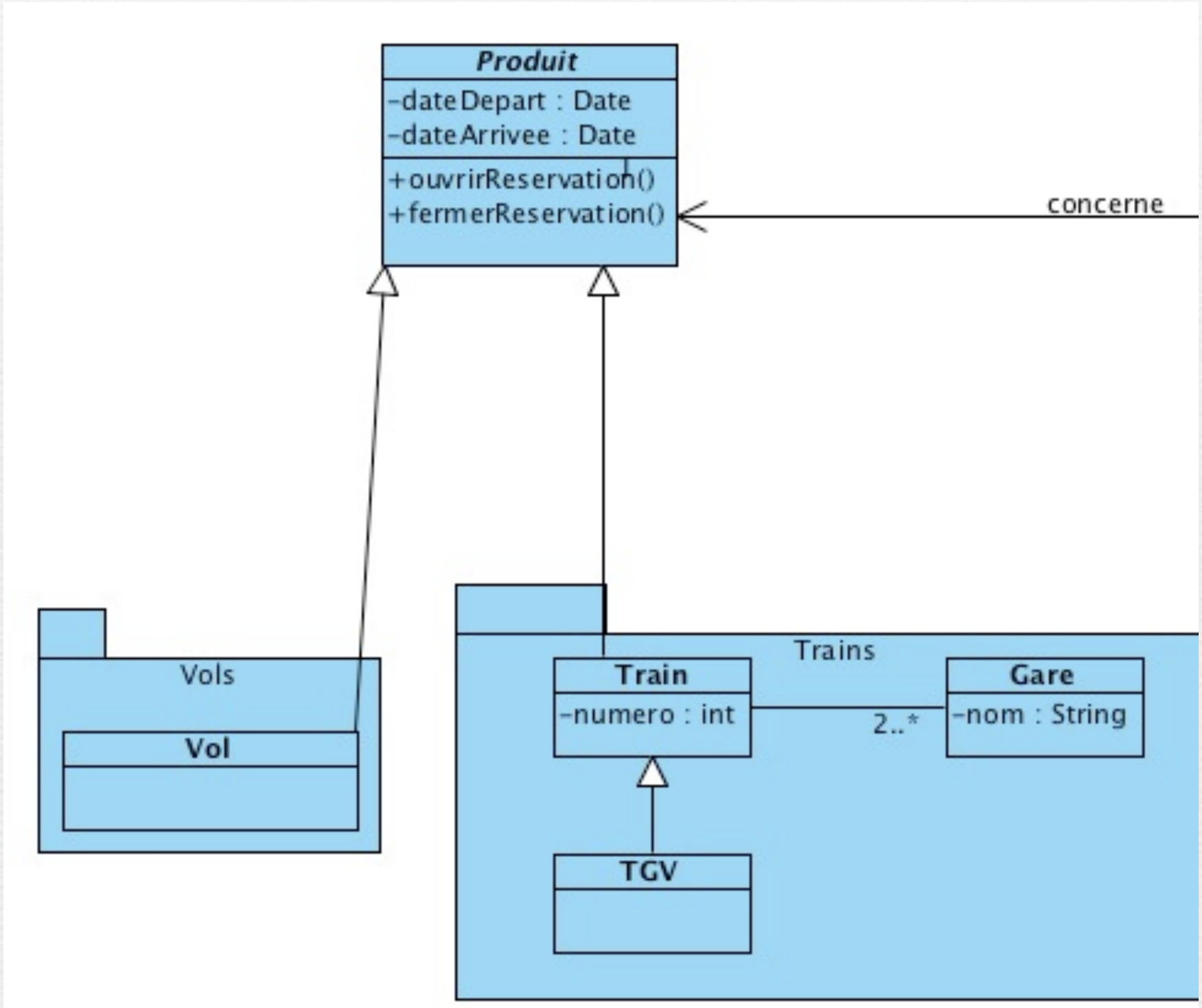
- Visibilité
- Abstraction
- Généralisation
- Packages
- Attributs et Opérations* de Classes
- Transformations des associations
- Anti-Patterns

Opération : terme générique désignant le plus souvent des méthodes

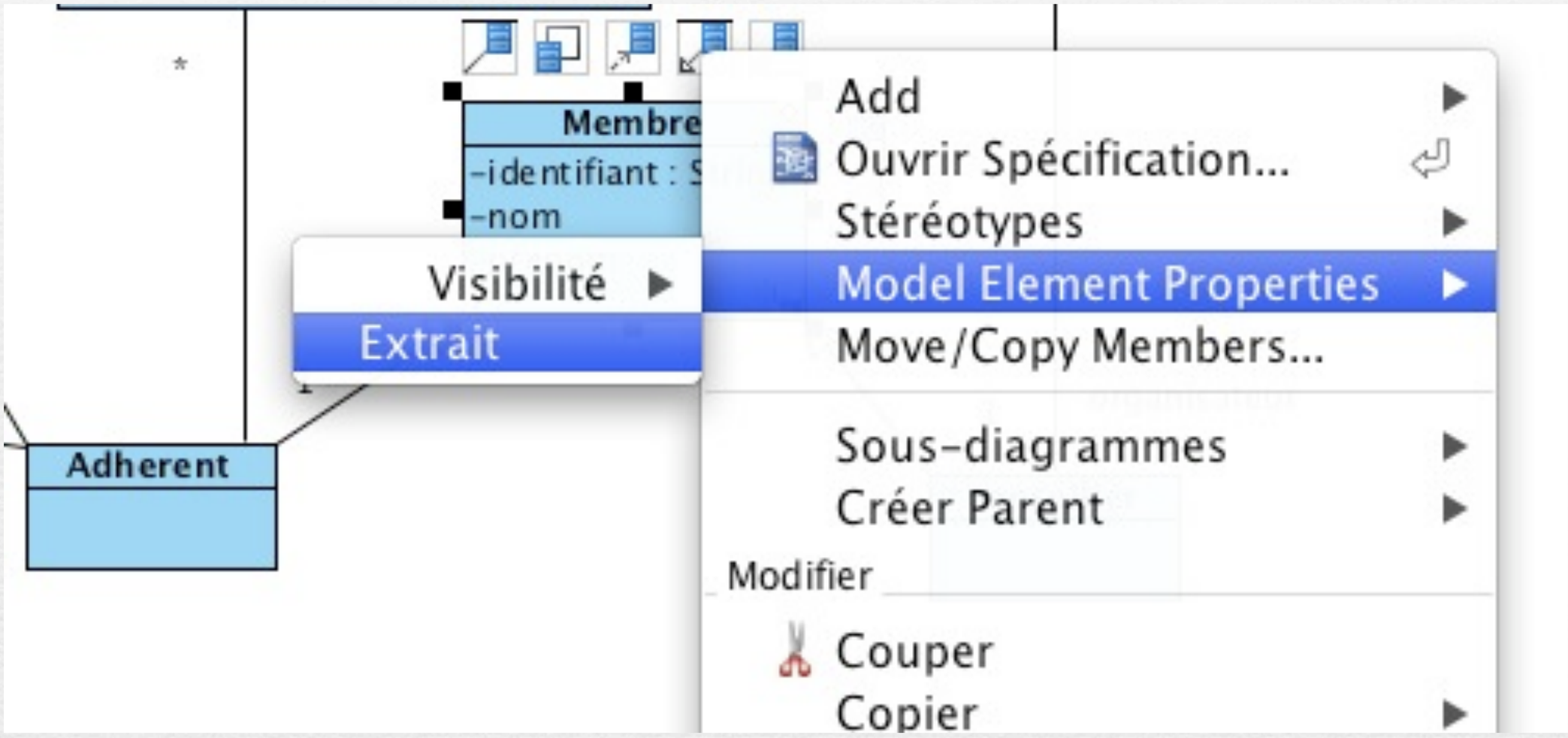
# Classes et Opérations abstraites

- Une *classe abstraite* est une classe non instanciable, c'est à dire qu'elle n'admet pas d'instances directes.
- Une classe abstraite est une description d'objets destinée à être « héritée » par des classes plus spécialisées.
- Pour être utile, une classe abstraite doit admettre des classes descendantes *concrètes*.
- La factorisation optimale des propriétés communes à plusieurs classes par généralisation nécessite le plus souvent l'utilisation de classes abstraites.

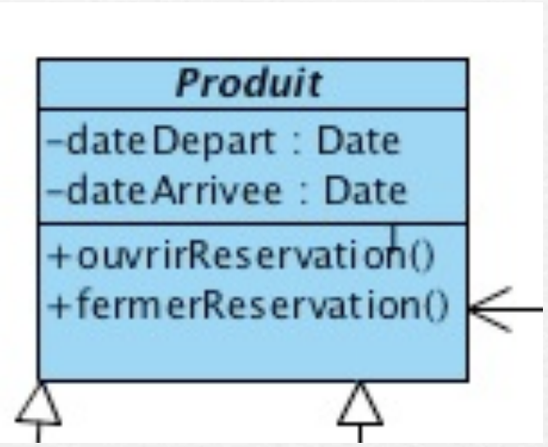
# Représentation de classes abstraites



# Classes et Opérations abstraites



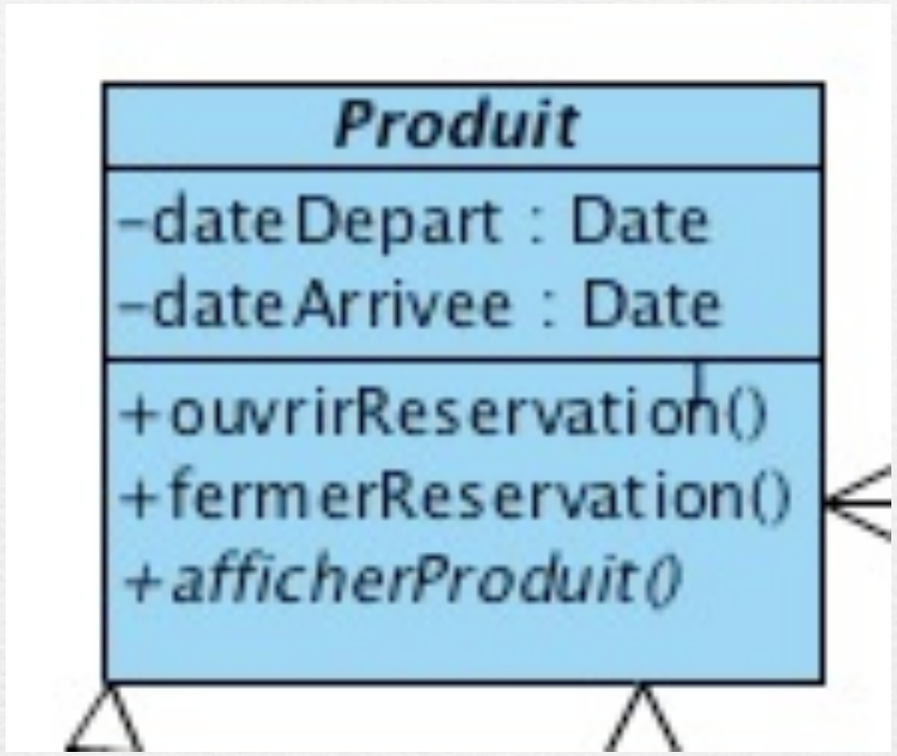
# Représentation de classes abstraites




Attention des choix de mises en oeuvre non explicités au niveau du modèle apparaissent dans ce code.

```
package produitPK;

import java.util.Date;

public abstract class Produit {
 private Date dateDepart ;
 private Date dateArrivée ;

 //Choix de mise en oeuvre
 private boolean open = false;

 //Choix de mise en oeuvre
public void setDateDepart(Date dateDepart) {
    this.dateDepart = dateDepart;
}
//Choix de mise en oeuvre
public Date getDateDepart() {
    return dateDepart;
}
//Choix de mise en oeuvre
public void setDateArrivée(Date dateArrivée) {
    this.dateArrivée = dateArrivée;
}
//Choix de mise en oeuvre
public Date getDateArrivée() {
    return dateArrivée;
}
```

# Opérations abstraites



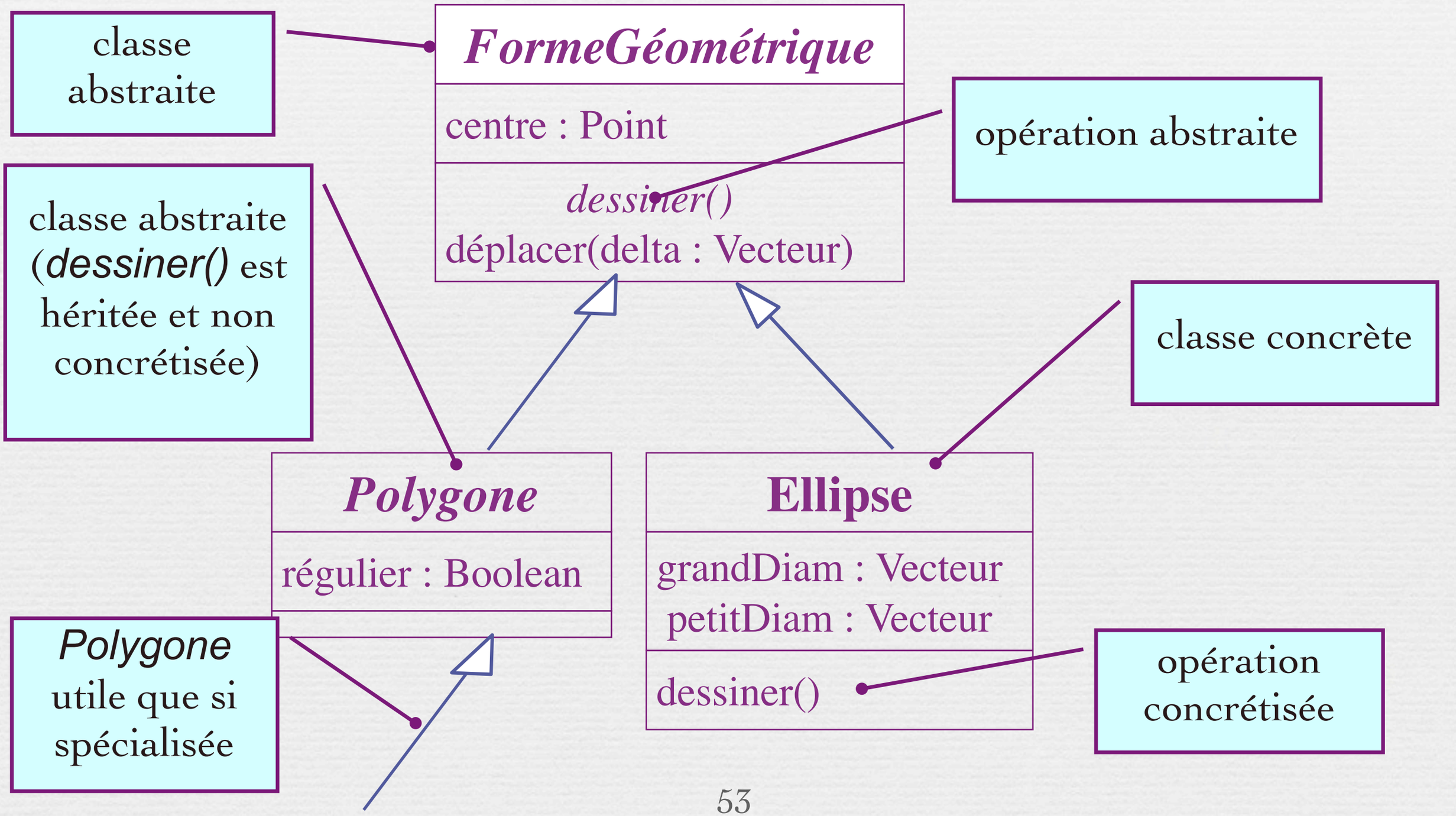

Attention des choix de mises en oeuvre non explicités au niveau du modèle apparaissent dans ce code.

```
public abstract class Produit {
 private Date dateDepart ;
 private Date dateArrivée ;

 //Choix de mise en oeuvre
 private boolean open = false;

 //Choix de mise en oeuvre
public void setDateDepart(Date dateDepart) {
    this.dateDepart = dateDepart;
}
//Choix de mise en oeuvre
public Date getDateDepart() {
    return dateDepart;
}
//Choix de mise en oeuvre
public void setDateArrivée(Date dateArrivée) {
    this.dateArrivée = dateArrivée;
}
//Choix de mise en oeuvre
public Date getDateArrivée() {
    return dateArrivée;
}

public void ouvrirReservation() {
    open = true;
}

public void fermerReservation() {
    open = false;
}

public abstract void afficherProduit();
```

# Opérations abstraites

- Une *opération abstraite* est une opération n'admettant pas d'implémentation : au niveau de la classe dans laquelle est déclarée, on ne peut pas dire comment la réaliser.
- Les opérations abstraites sont particulièrement utiles pour mettre en œuvre le polymorphisme.
- Toute classe concrète sous-classe d'une classe abstraite doit “concrétiser” toutes les opérations abstraites de cette dernière.

# *Classes abstraites*

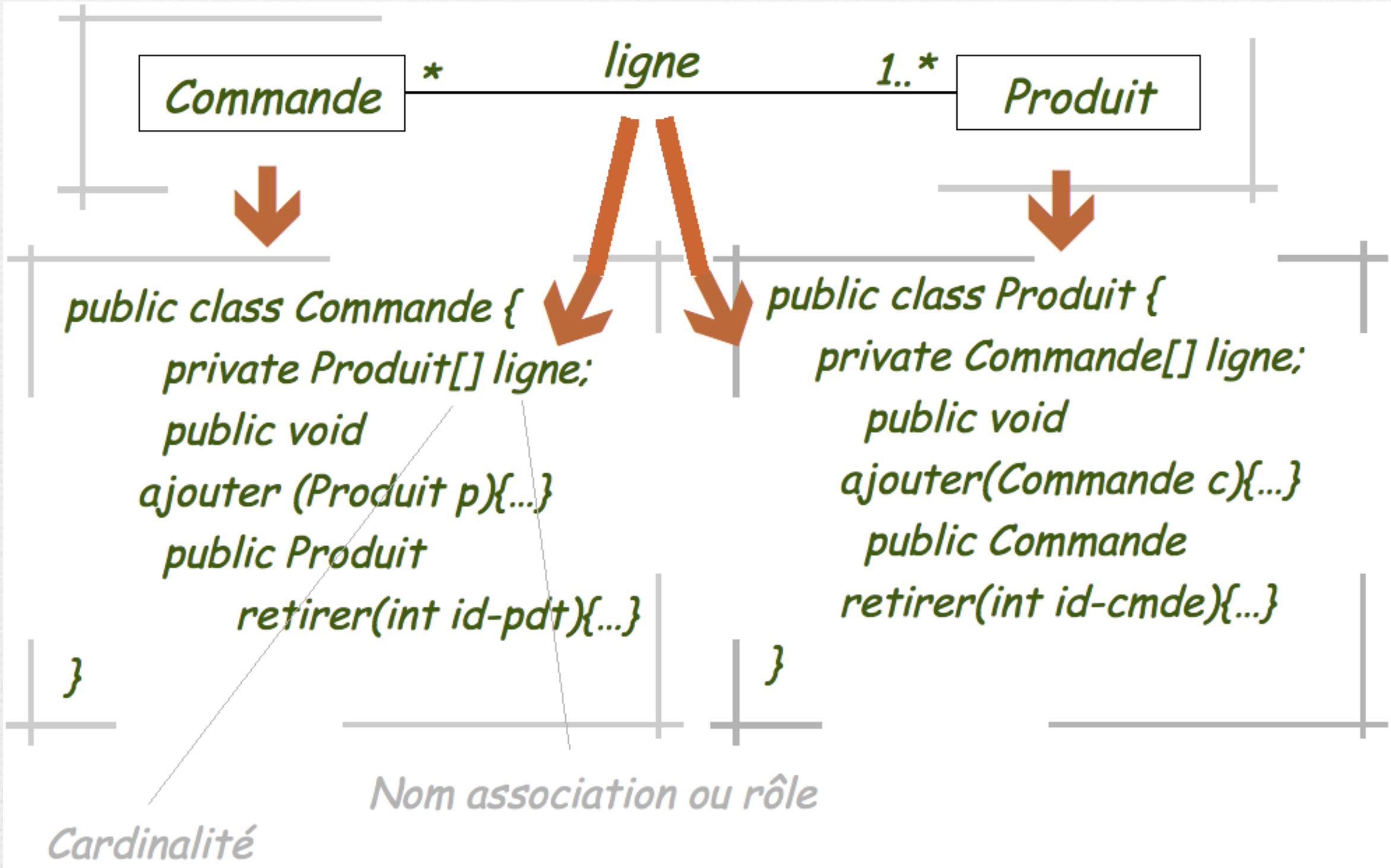


# Vers la mise en oeuvre des classes

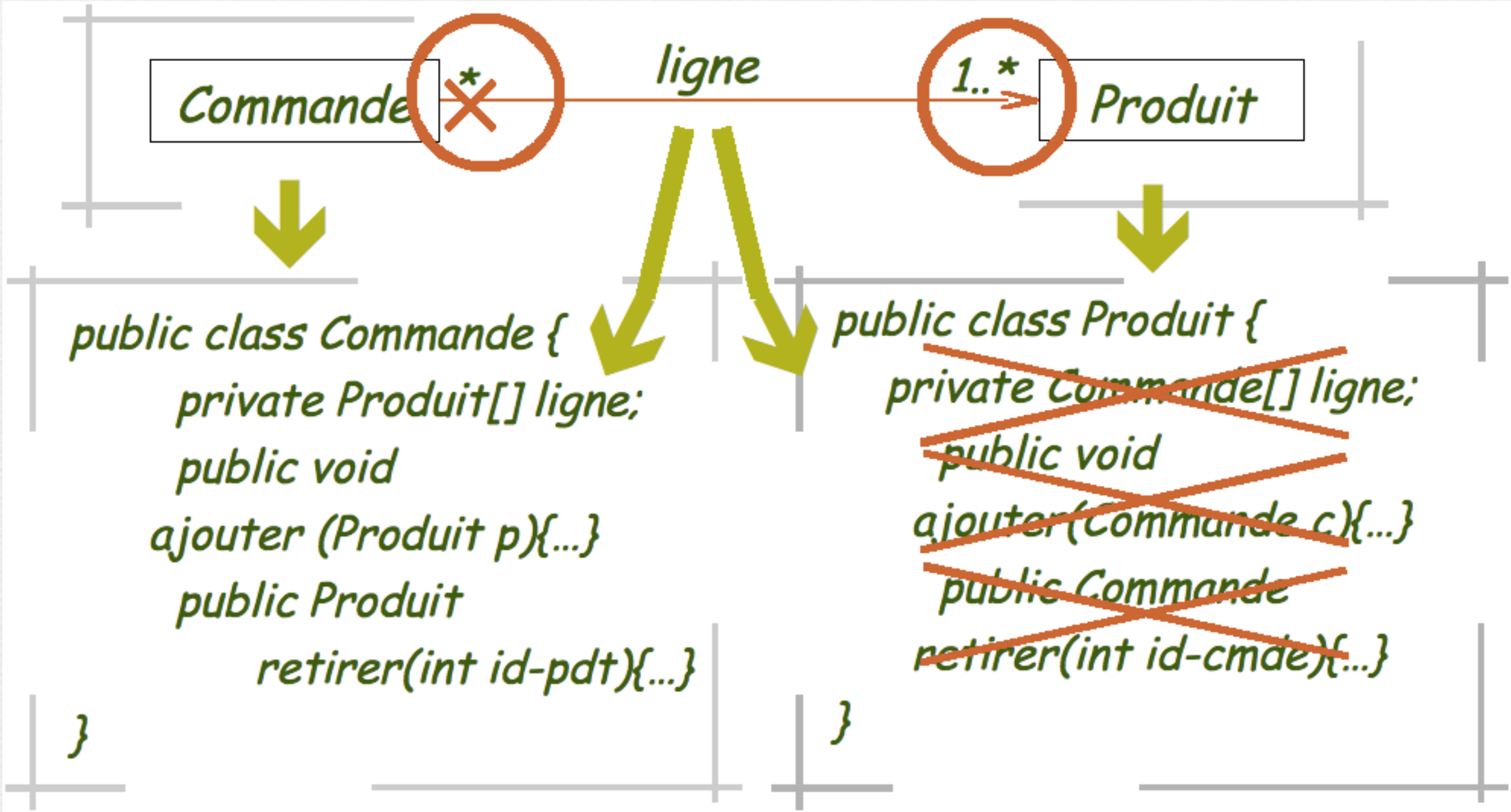
- Visibilité
- Abstraction
- Attributs et Opérations* de Classes
- Généralisation
- Packages
- Transformations des associations
- Anti-Patterns

Opération : terme générique désignant le plus souvent des méthodes

# Association...



# Association...



# Association:
# De la conception à l'implémentation


```
public class Commande {
        private Produit[] ProduitsCommandés;
        public void
      ajouter (Produit p){…}
        public Produit
            retirer(int id-pdt){…}
}
```

# Association:
# De la conception à l'implémentation


```
public class Commande {
        private Produit[] ProduitsCommandés;
        public void
```

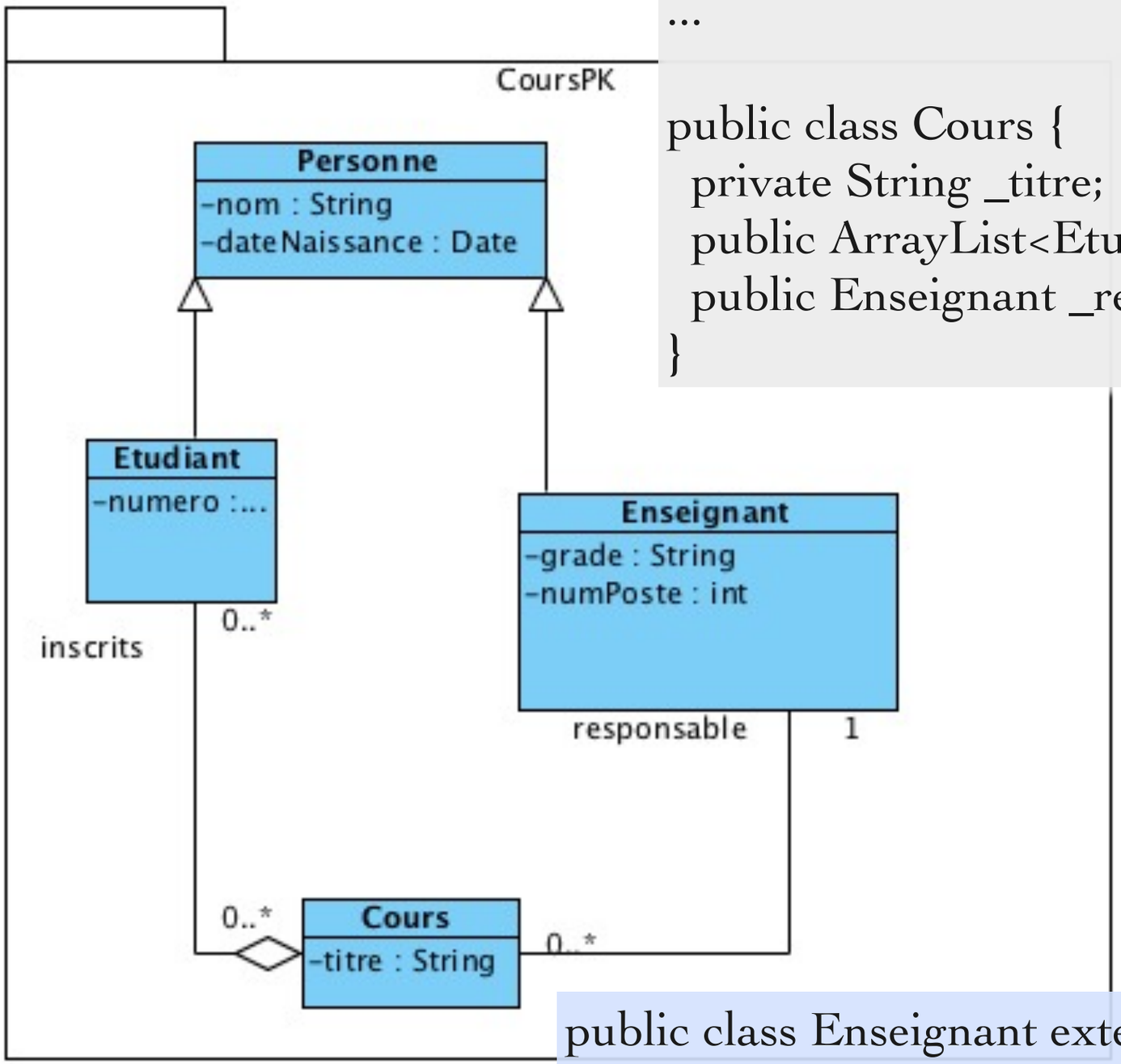
```
public Commande (Produit[] c) throws Exception {
     if (c.length != 0)
          lignes = c;
     else
          throw new Exception("Un produit au moins est requis");
}
```

# Association:
# De la conception à l'implémentation


```
public Commande (Produit[] c) throws Exception {
    if (c.length != 0)
        lignes = c;
    else
        throw new Exception("Un produit au moins est
requis");
    }
```

```
    public boolean oterProduit(Course c) {
        if (lignes.length==1)
            return false;
        ...
    }
```

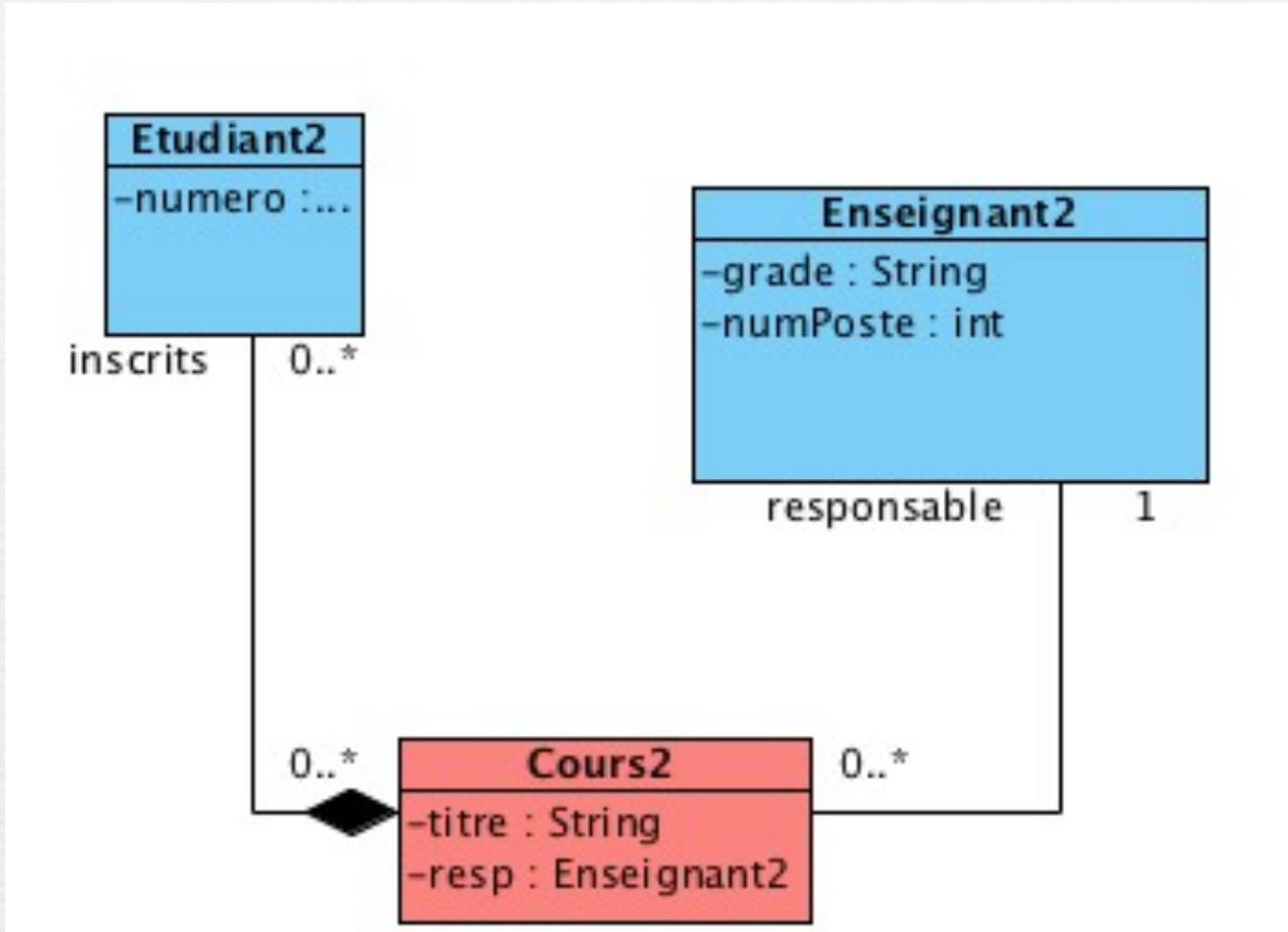
# Association et générations de code




```
package code_generation.CoursPK;
...

public class Cours {
  private String _titre;
  public ArrayList<Etudiant> _inscrits = new ArrayList<Etudiant>()
  public Enseignant _responsable;
}
```

```
public class Enseignant extends Personne {
        private String _grade;
        private int _numPoste;
        public ArrayList<Cours> _unnamed_Cours_ = new ArrayList<Cours
```

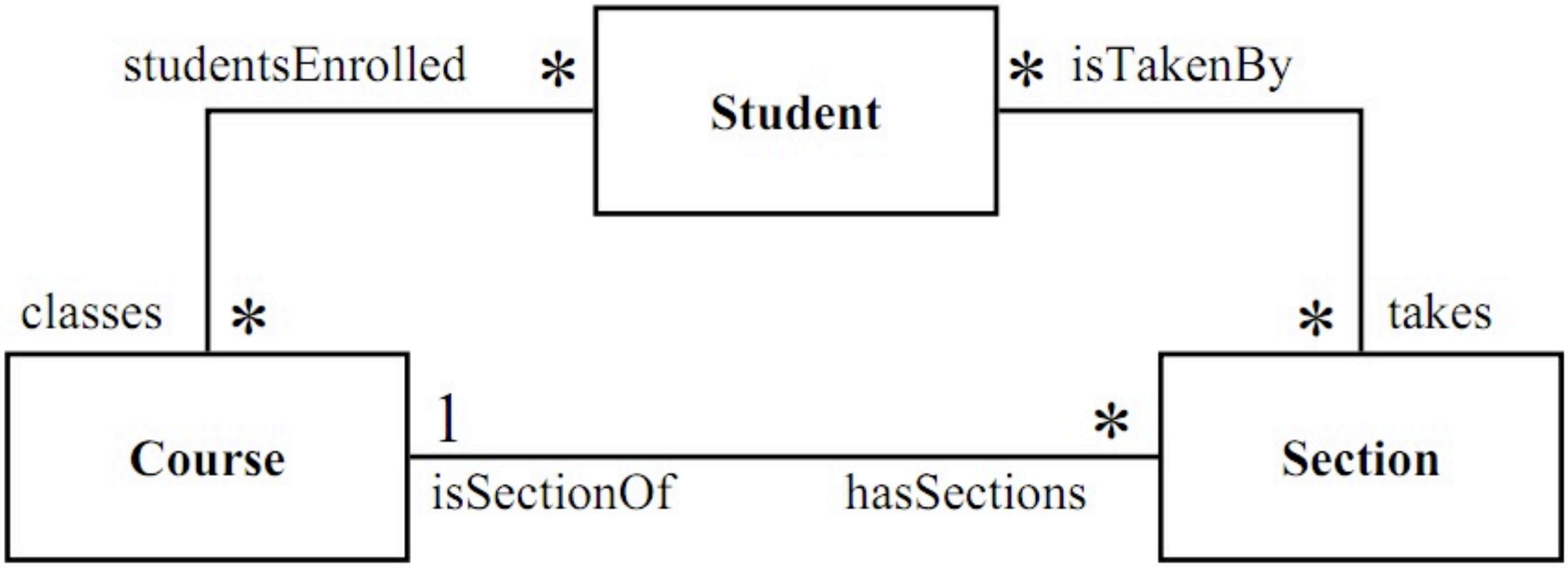
# Association et générations de code



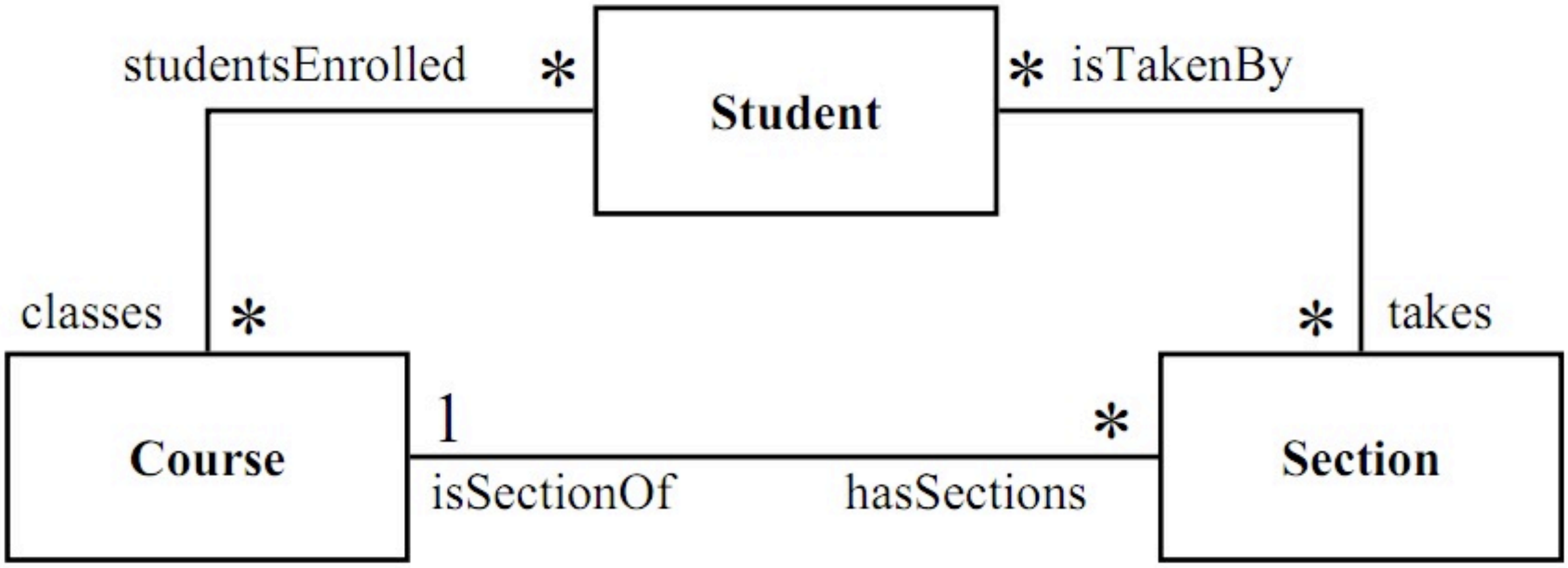

Des problèmes ?

```
public class Cours2 {
	private String _titre;
	private Enseignant2 _resp;
	public ArrayList<Etudiant2> _inscrits = new ArrayList<Etudiant2>();
	public Enseignant2 _responsable;
}
```

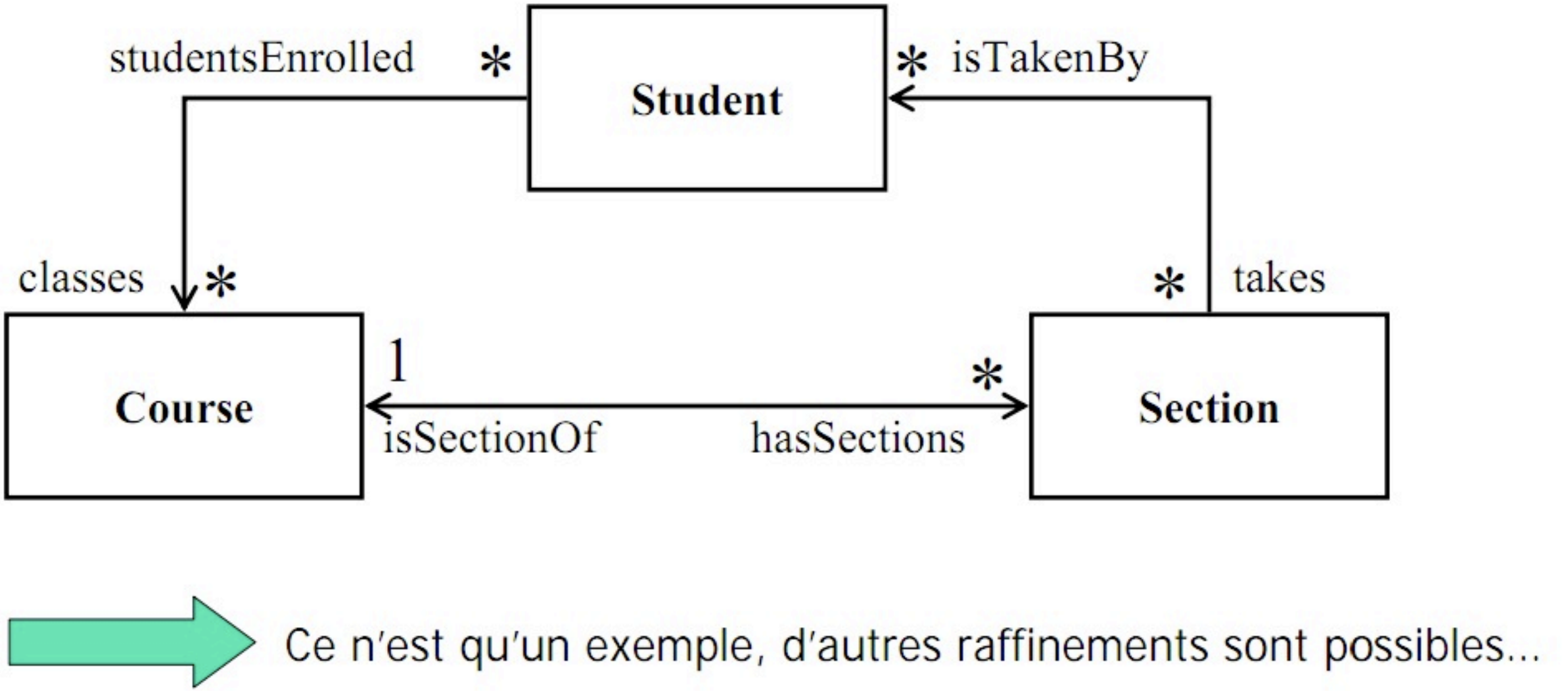
# Gestion des associations



# Associations & Navigations



# Exemple de Raffinement

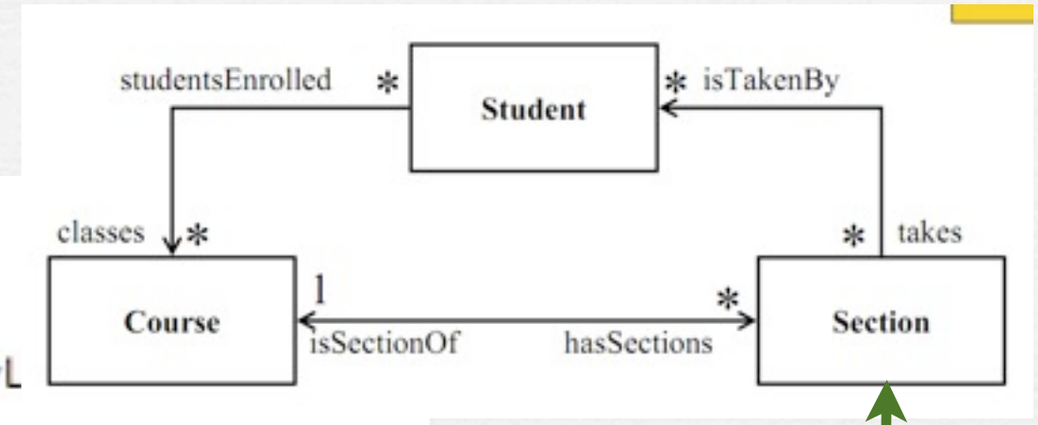



Ce n'est qu'un exemple, d'autres raffinements sont possibles...

# Principes d'implémentation

- Extrémité d'association 1
  - Rôle en Attribut avec type de l'extrémité
  - *Type* get*Role*()
- Extrémité d'association *
  - Rôle (pluriel) en collection
  - Type de l'extrémité en élément de collection
  - Collection get*Roles*()
  - // Collection<TypeExtrémité>//

# Implémentation




```
public class Section {
    private String name;
    private Course isSectionOf;
    private  Collection<Student> isTakenBy = new ArrayL
    public String toString() {
```

```
    public Section(String name, Course isSectionOf) {
        this.name = name;
        //ATTENTION ...
        setIsSectionOf(isSectionOf) ;
    }
```

```
    public void setIsSectionOf(Course c) {
        isSectionOf = c;
        c.addHasSections(this);
    }
```

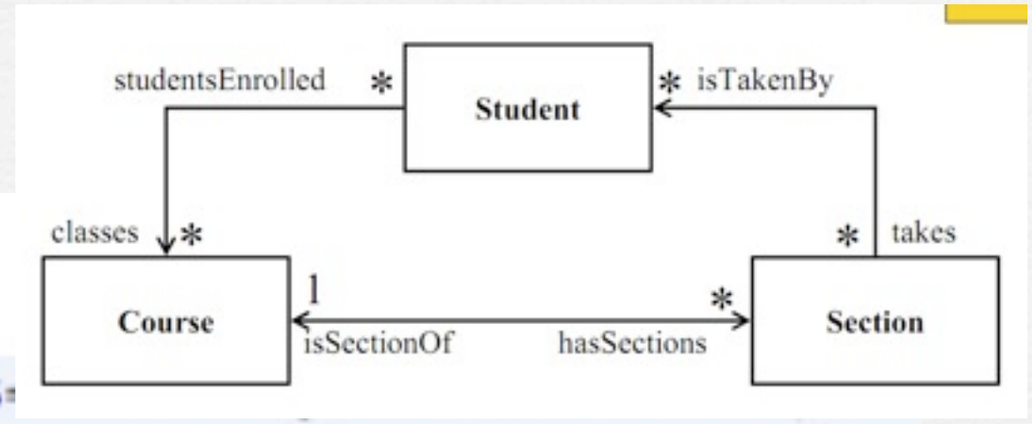
```
    public Course getIsSectionOf() { return isSectionOf;}

    public Collection<Student> getIsTakenBy() {
        return isTakenBy;
    }
```

```
    public void addStudent(Student s) {
        isTakenBy.add(s);
        if (!(s.getClasses().contains(isSectionOf)) )
            s.addClass(isSectionOf);
    }
```

A la construction

Prise de responsabilités

# Implémentation



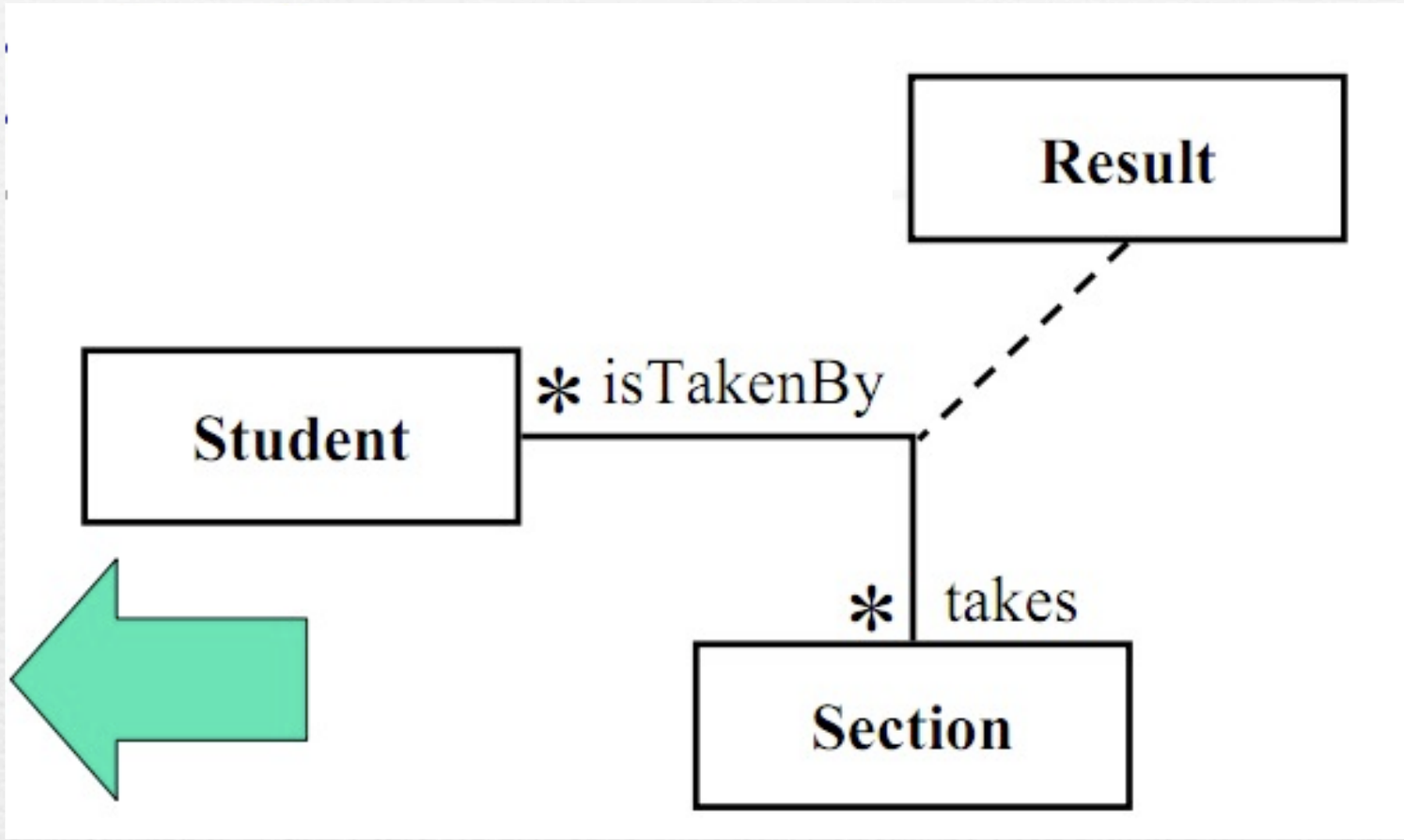

```
public class Student {
        private String name;
        private Collection<Course> classes=

        public Student(String name) {..
        public String toString() {..
        public Collection<Course> getClasses() {
                return classes;
        }
        protected void addClass(Course c){
                classes.add(c);
```

```
public class Course {
    private String name;
    private Collection<Section>  hasSections = new ArrayList<Section>();

    public Course(String name) {..
    public String toString() {..
    public Collection<Section> getHasSections() {
        return hasSections;}
    protected void addHasSections(Section s){
            hasSections.add(s);
        }
```

Définition des responsabilités
Ne jamais appeler ***addHasSections*** ou ***addClass*** directement !

# Implémentation

```
class Student {
   Result getResult(Section s)
  }
class Section {
  Result getResult(Student s)
 }
class Result {
  Student getStudent()
  Section getSection()
}
```

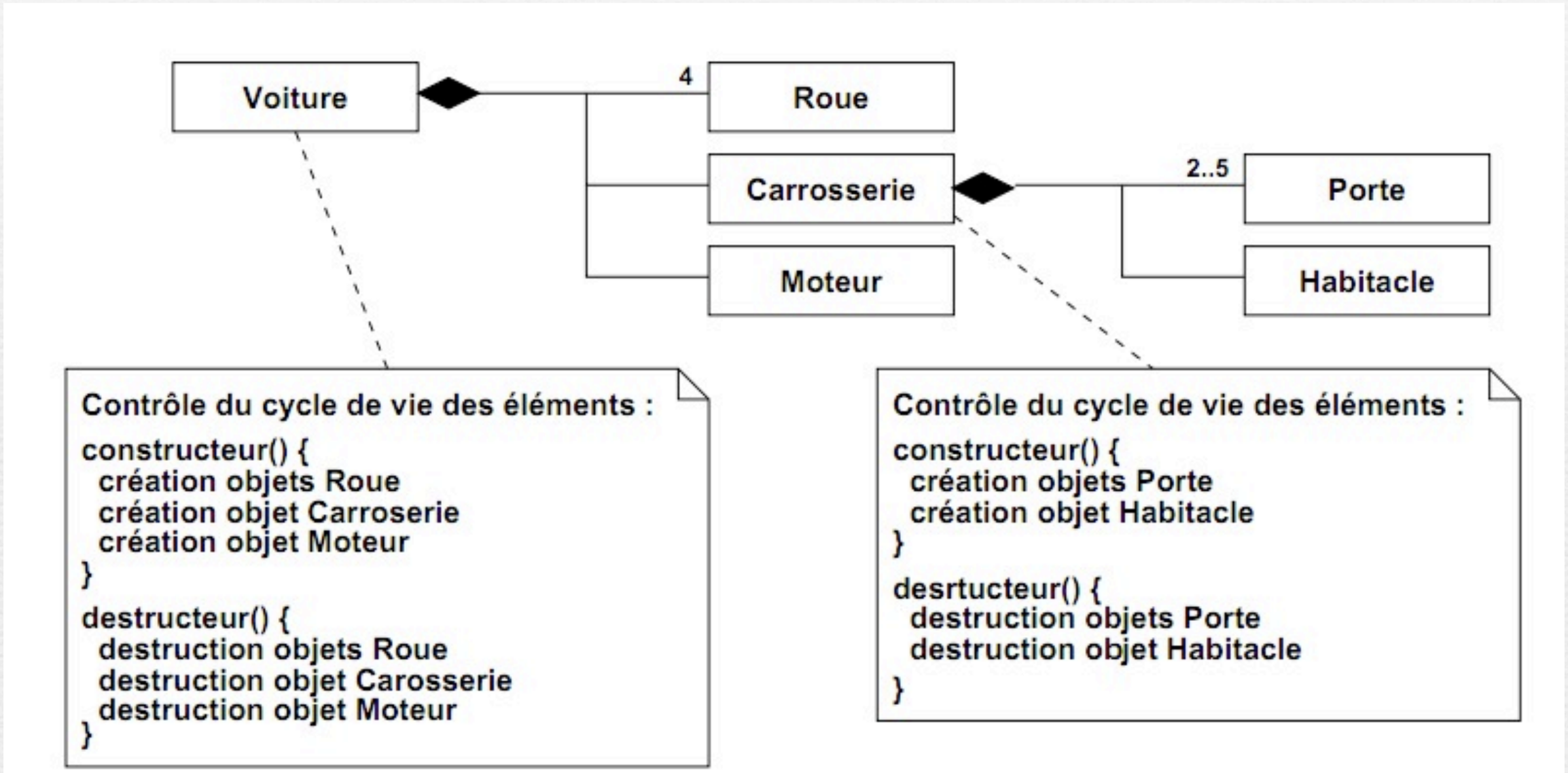


# En résumé : Traduction des associations en attributs

- Autant d'attributs que de classes auxquelles elle est reliée (navigable)
- Association unidirectionnelle = pas d'attribut du côté de la flèche
- Nom de l'attribut = nom du rôle ou forme nominale du nom de l'association
- Attribut du type référence sur un objet de la classe à l'autre extrémité de l'association
  - Référence notée « @ »
- Traduction des multiplicités
  - 1 =⇒ @Classe
  - * =⇒ Collection @Classe
  - 0..N =⇒ Tableau[N] Classe
- Multiplicité avec tri = Collection ordonnée @Classe

En principe...

# Compositions



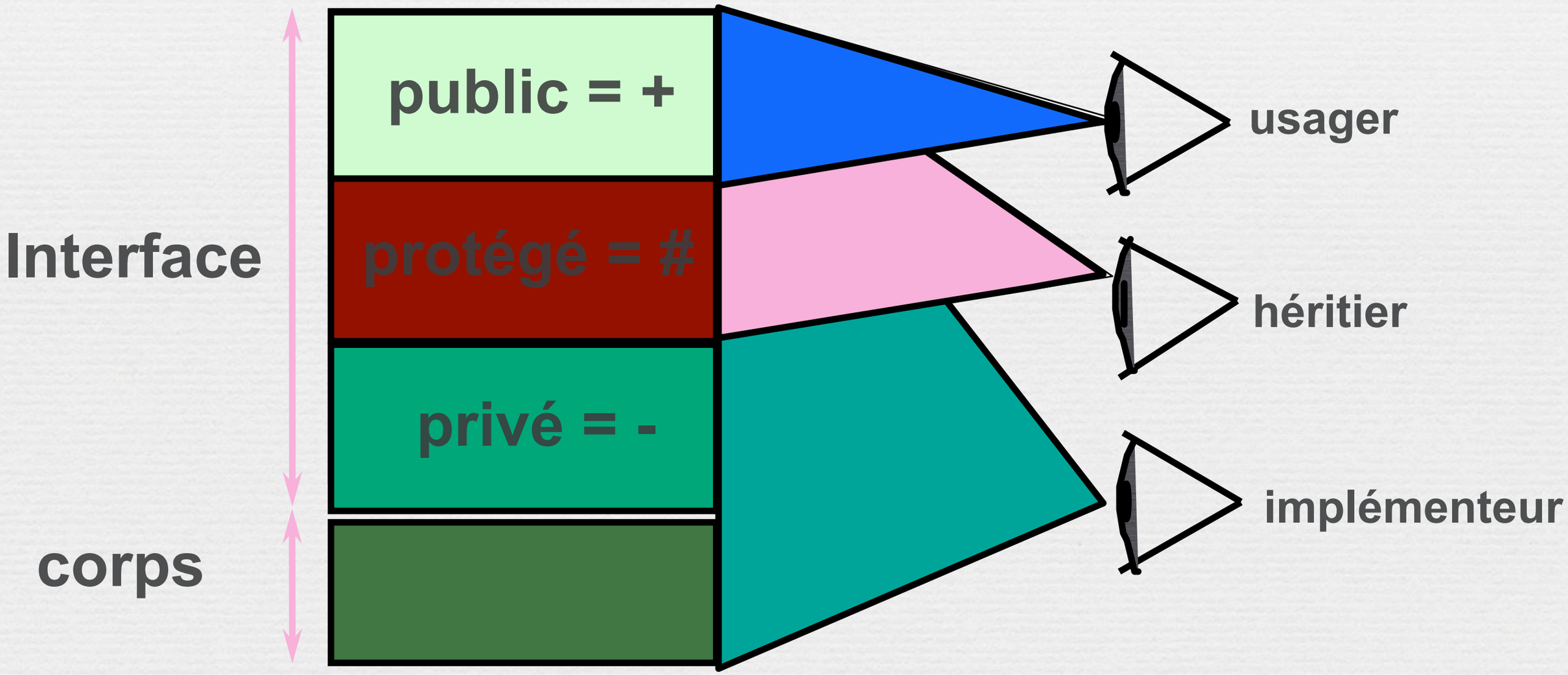
# Vers la mise en oeuvre des classes

- Visibilité
- Abstraction
- Généralisation
- Packages
- Attributs et Opérations* de Classes
- Transformations des associations
- Anti-Patterns

Opération : terme générique désignant le plus souvent des méthodes

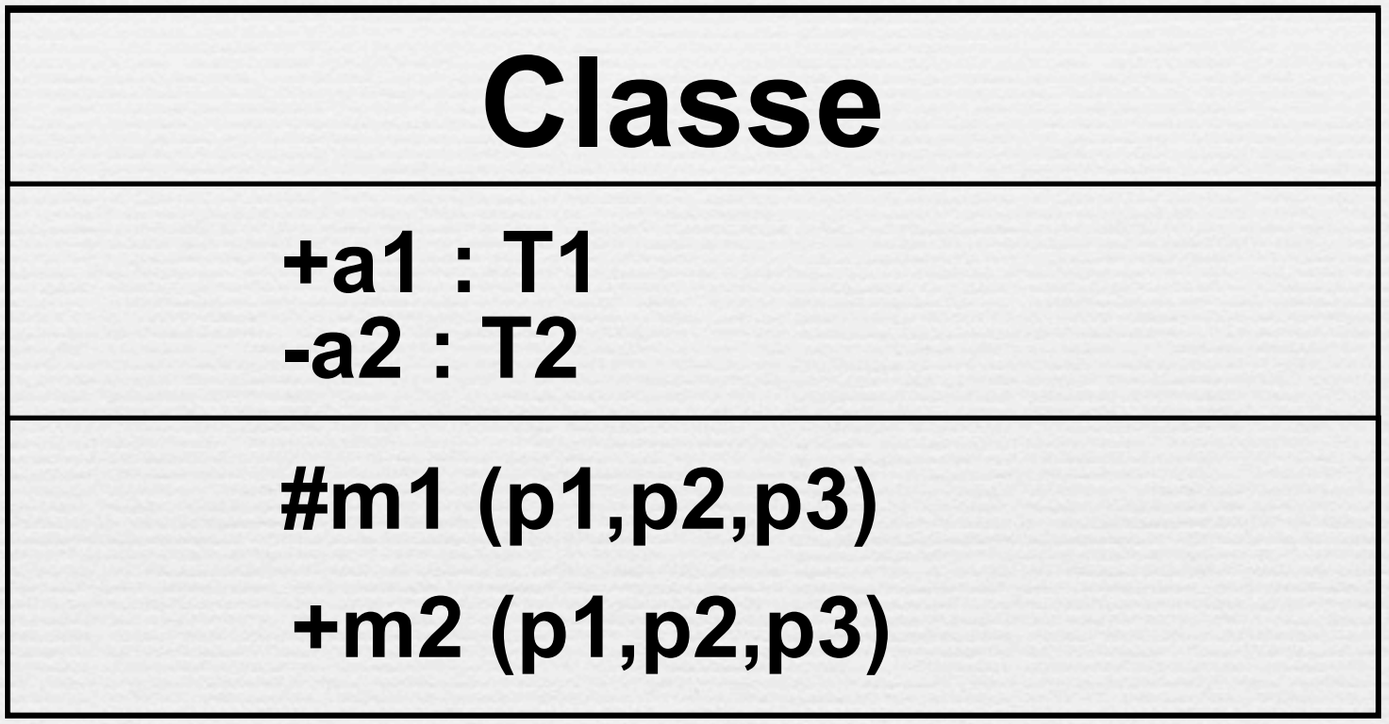
# Visibilité

- Différentes visibilités des membres d'une classe



# Visibilité

- Représentation




- **Pas de sens en analyse...**

# Encapsulation : accessibilité des attributs et des opérations

- Peut-on accéder à tous les attributs ou à toutes les méthodes d'un objet ? Non
  - La classe définit ce qui est accessible
  - C'est le principe de l'encapsulation
  - Un objet complexe ne peut être utilisé qu'au travers de ce qui est accessible
- *Principes :*
  - *Il n'est possible d'utiliser une voiture qu'à travers son volant, son frein, son accélérateur, etc.*
  - *L'accès au carburateur est impossible sauf par les méthodes qui le font de manière cohérente (méthode accélérer de l'accélérateur)*

# Encapsulation avec le concept de Visibilité

- Les attributs sont en général inaccessibles (secrets). Ils sont alors qualifiés de :
  - « private » : notation UML « – »
  - Lecture ou modification possible au travers des opérations (p.ex. les ***accesseurs*** : setAdresse(), getAdresse())

- Les opérations sont en général accessibles par toutes les classes. Elles sont alors qualifiées de :
  - « public » : notation UML « + »

# Encapsulation avec le concept de Visibilité

- Certains attributs/opérations doivent être accessibles par les sous-classes ou aux classes d'un même package et inaccessibles aux autres classes. Ils sont alors qualifiés de :
  - « protected » : notation UML « # »

- Certaines opérations peuvent cependant
  - être privées (factorisation interne de traitements) et
  - certains attributs peuvent être publics (non souhaitable / principe d'encapsulation)

# Vers la mise en oeuvre des classes

- Visibilité
- Abstraction
- Généralisation
- Packages
- Attributs et Opérations* de Classes
- Transformations des associations
- Anti-Patterns

Opération : terme générique désignant le plus souvent des méthodes

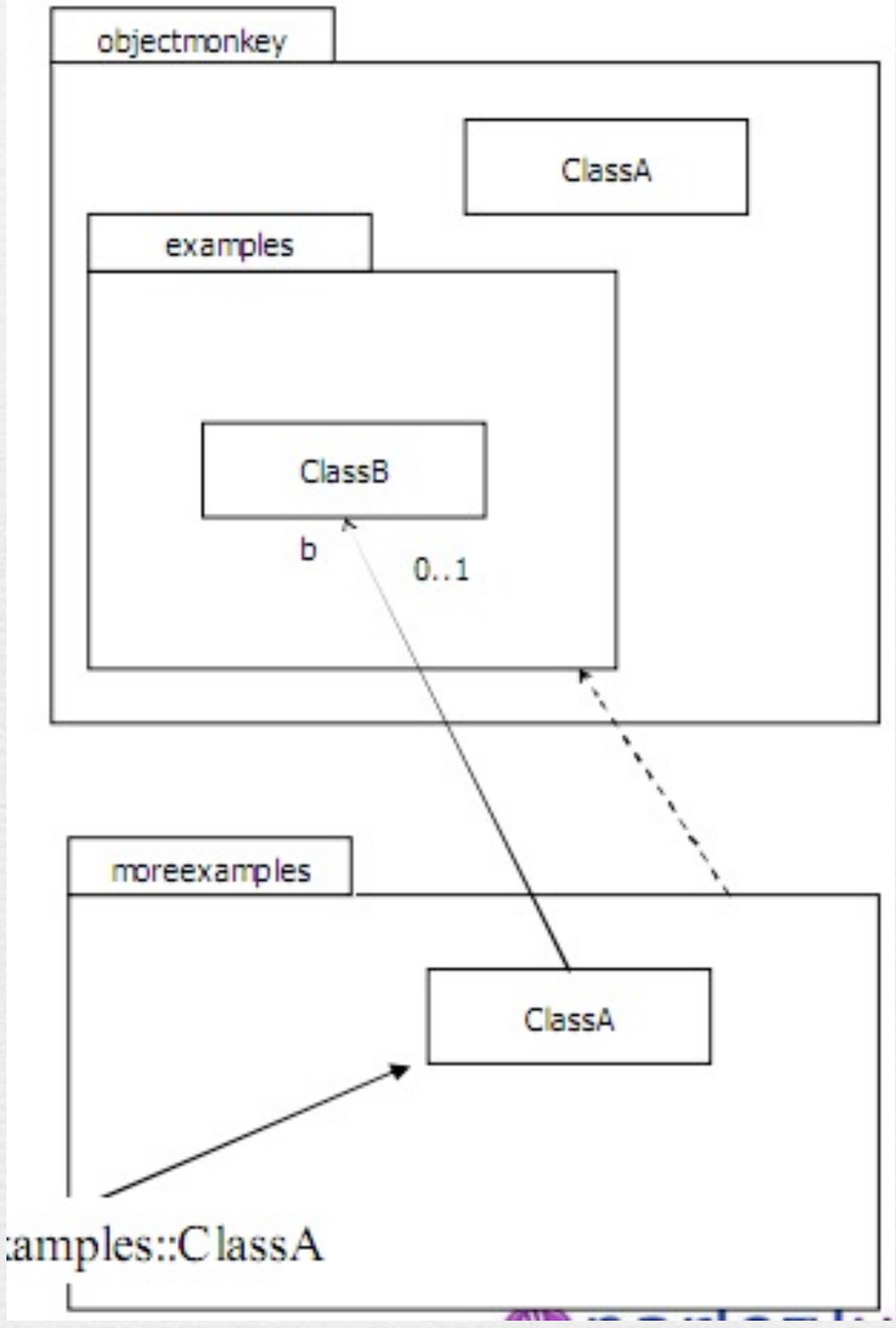
# Package

ICI

```
package objectmonkey;


class ClassA
{
}
```

```
package objectmonkey.examples;


class ClassB
{
}
```

```
package moreexamples;

import objectmonkey.examples.*;

class ClassA
{
                private ClassB b;
}
```



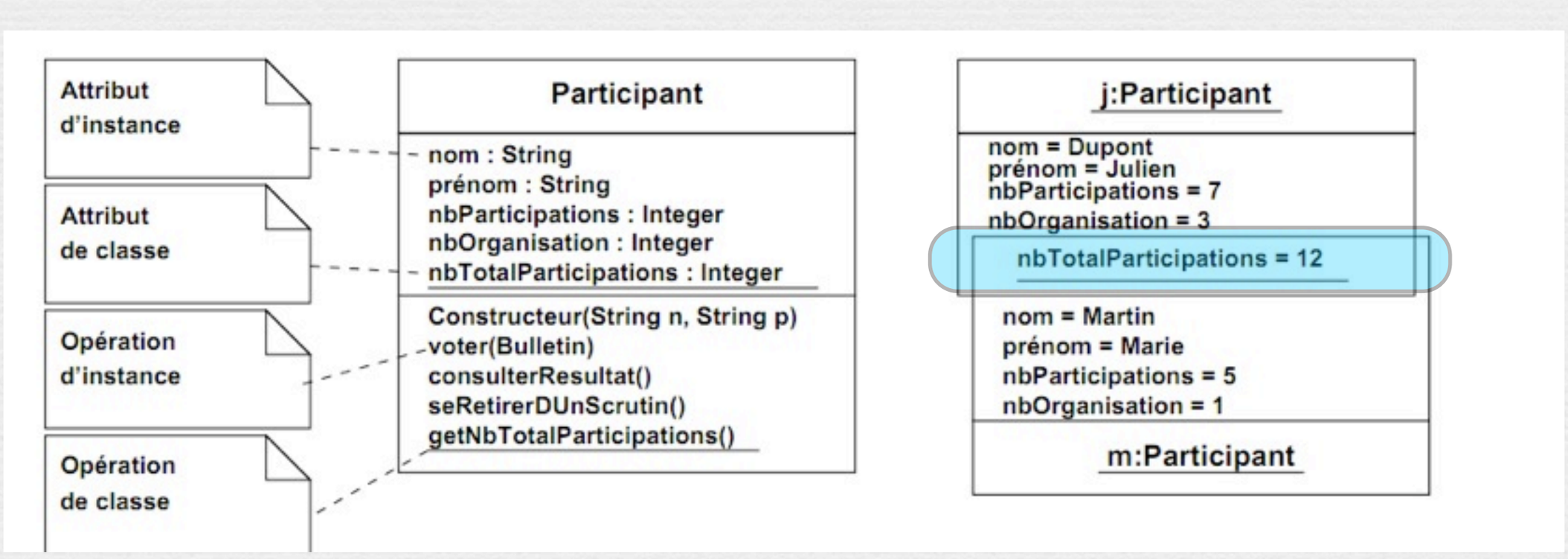
# Vers la mise en oeuvre des classes

- Visibilité
- Abstraction
- Attributs et Opérations* de Classes
- Généralisation
- Packages
- Transformations des associations
- Anti-Patterns

Opération : terme générique désignant le plus souvent des méthodes

# Attributs et opérations de classe

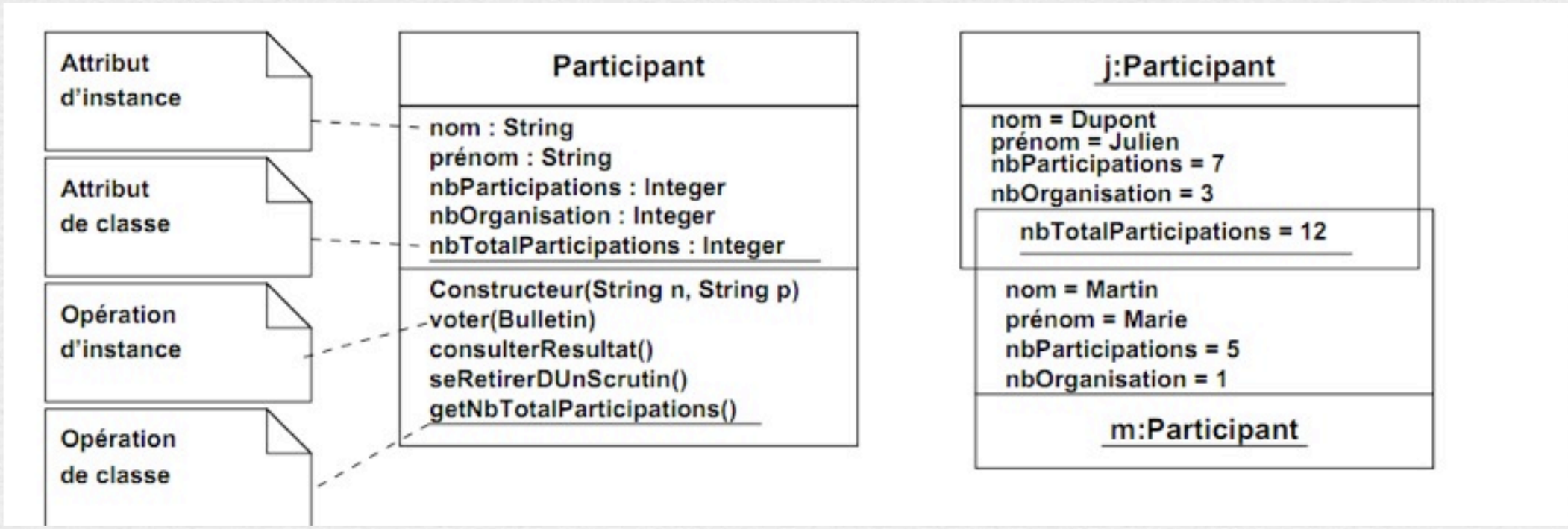
On gère des participants. Un participant peut avoir participé à plusieurs évènements. On veut connaître le nombre total de participations à des évènements.



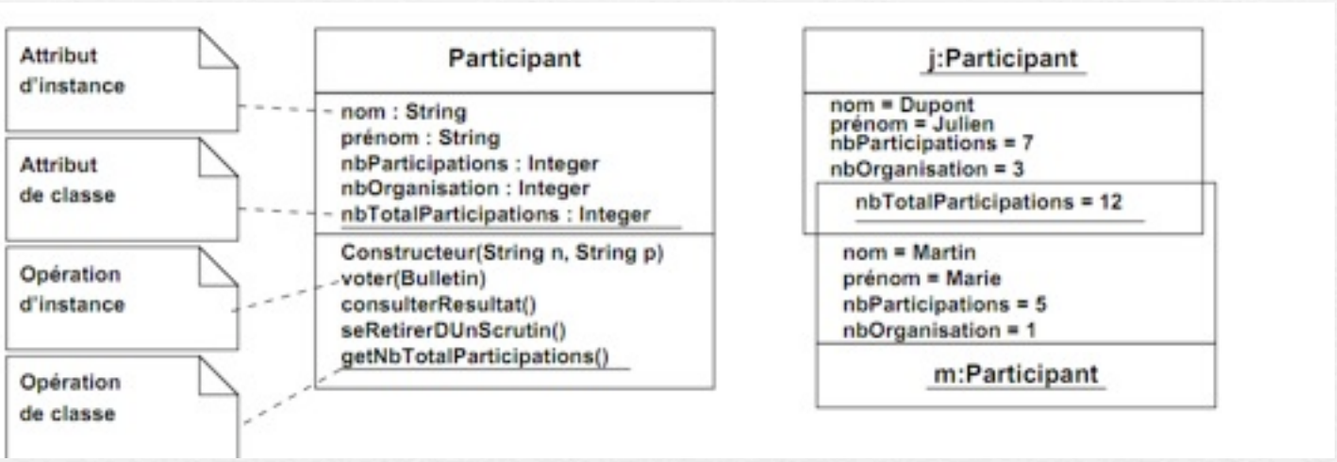

D'après Introduction au langage de modélisation UML, Denis Conan, Chantal Taconet, Christian Bac, Telecom Sud Paris

# Attributs et opérations de classe

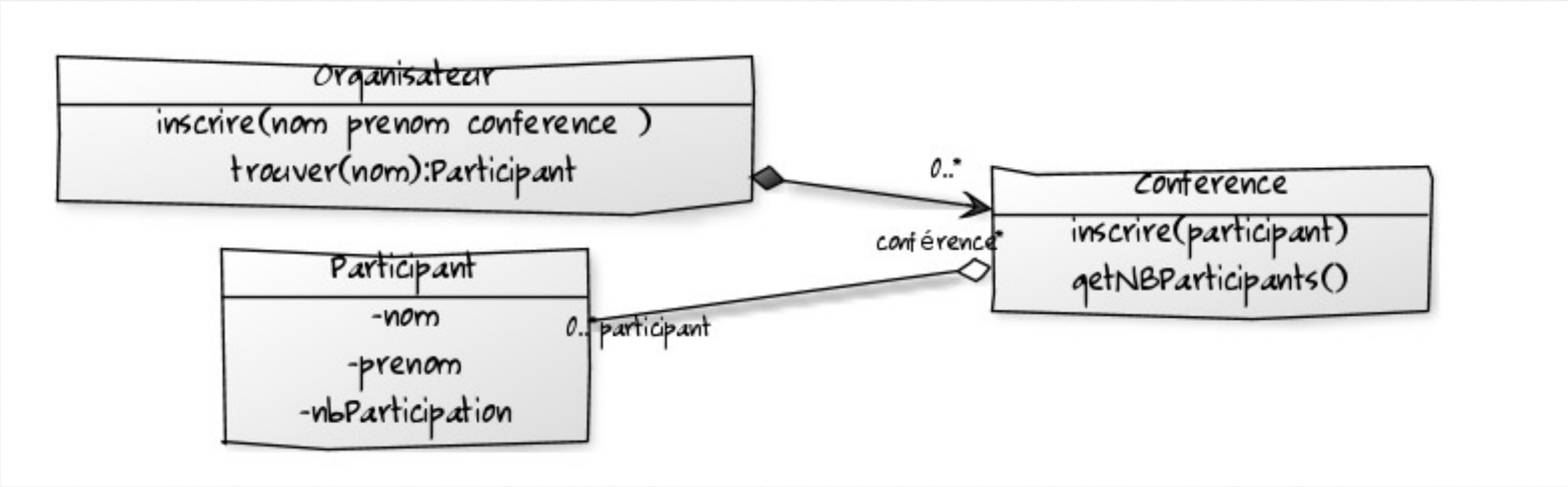
- Le nombre total de participations est une caractéristique des Participant (classe), pas d'un seul participant.
- L'opération **getNbTotalParticipations()** utilise la valeur de l'attribut **nbTotalParticipations** connue par la classe
  - Cette opération peut être appliquée directement à la classe Participant et aussi aux objets / instances de Participant



# Attributs et opérations de classe versus «fabrique»

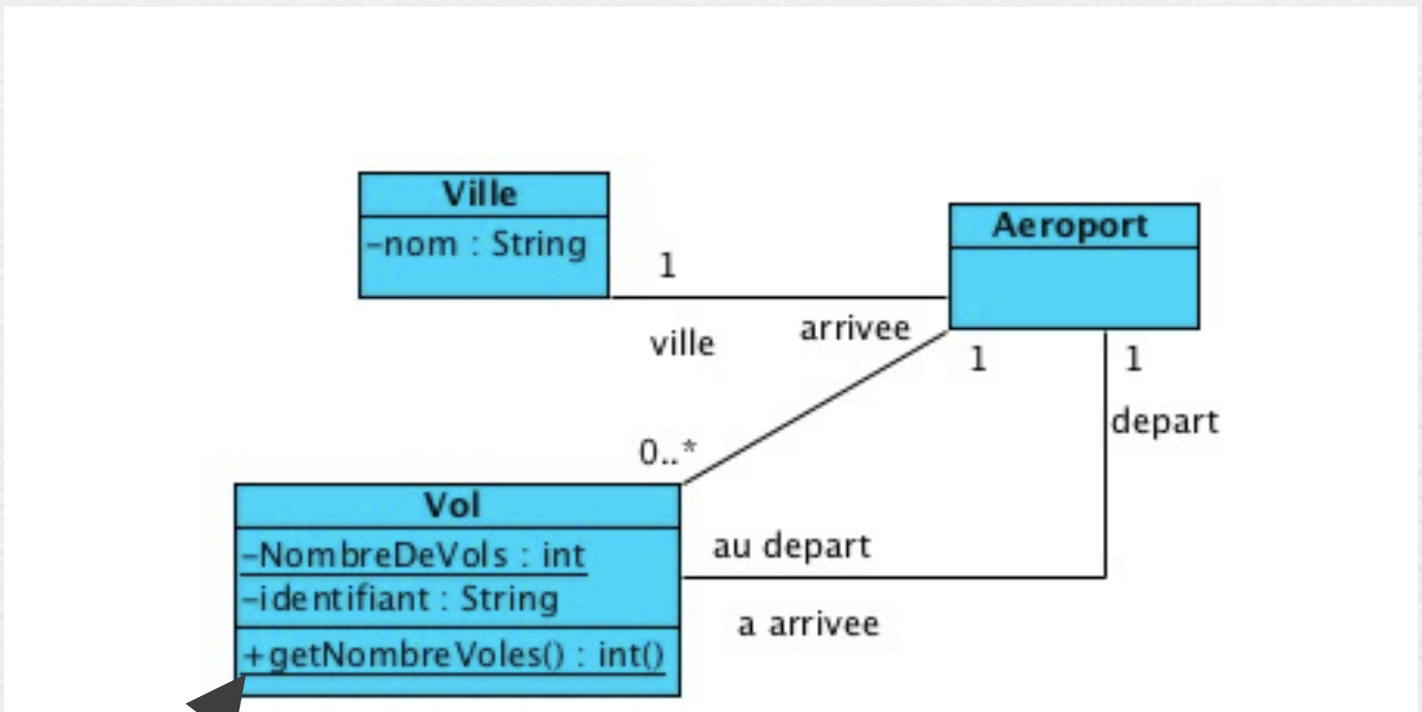



A PREFERER EN CONCEPTION.

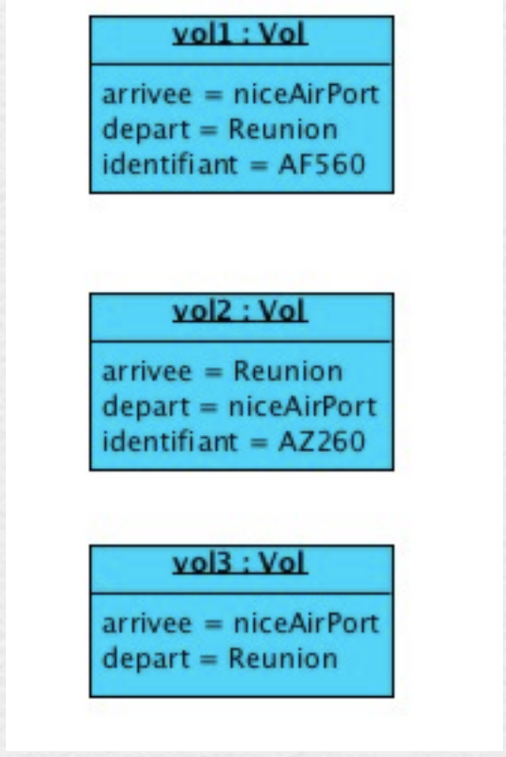


# Attributs et opérations de classe

On gère des Vols. On veut mémoriser le nombre de vols créés.




NombreDeVols=3




Variable de classe / static en java !

# Attributs et opérations de classe

| Person |
| --- |
| - numberOfPeople : int<br>- name : string |
| + createPerson(name : string) : Person<br>+ getName() : string<br>+ getNumberOfPeople() : int<br>- Person(name : string) |

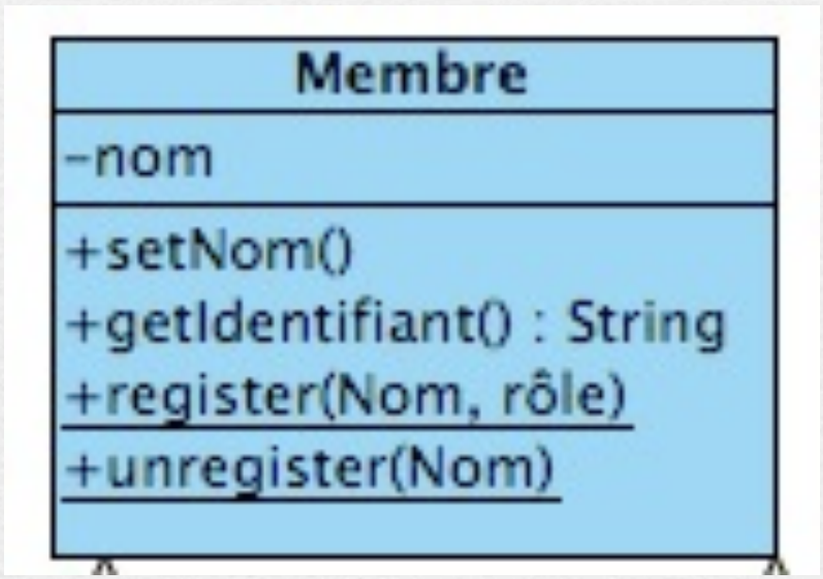
```
int noOfPeople = Person.getNumberOfPeople();
Person p = Person.createPerson("Jason Gorman");
```

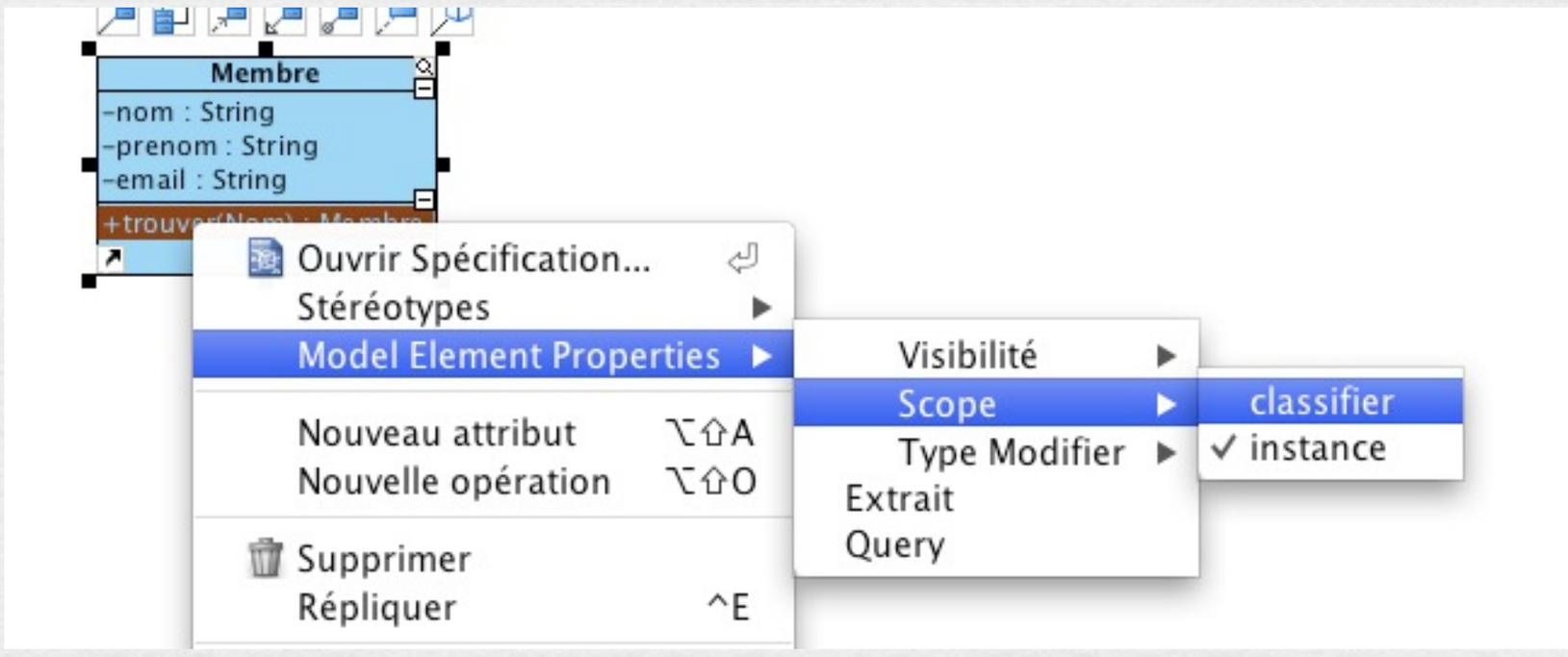


```
class Person
{
        private static int numberOfPeople = 0;
        private String name;

        private Person(string name)
        {
                this.name = name;
                numberOfPeople++;
        }

        public static Person createPerson(string name)
        {
                return new Person(name);
        }

        public string getName()
        {
                return this.name;
        }

        public static int getNumberOfPeople()
        {
                return numberOfPeople;
        }
```

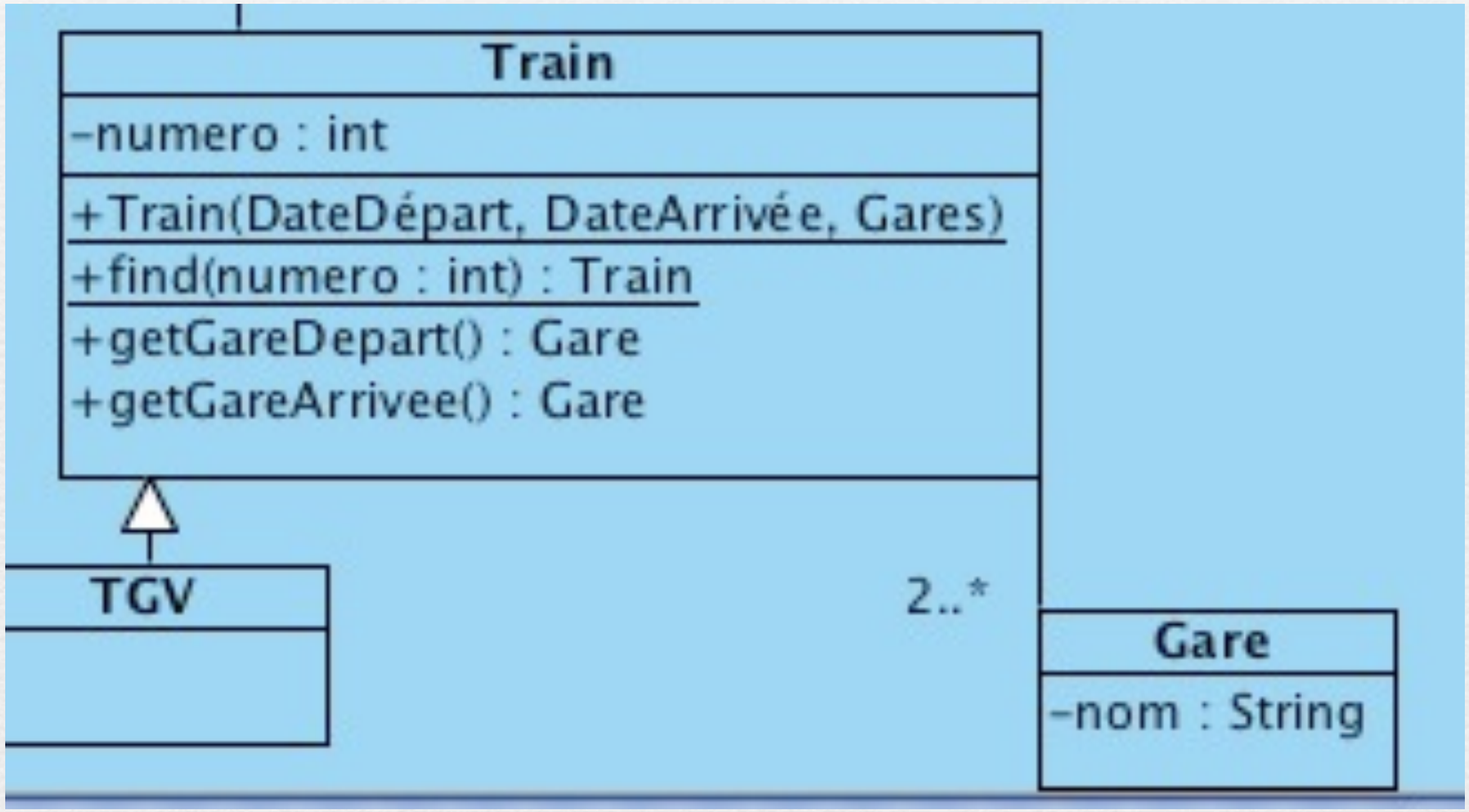
# Opérations du niveau de la classe : *Static*

- Une opération «de niveau classe» est soulignée.

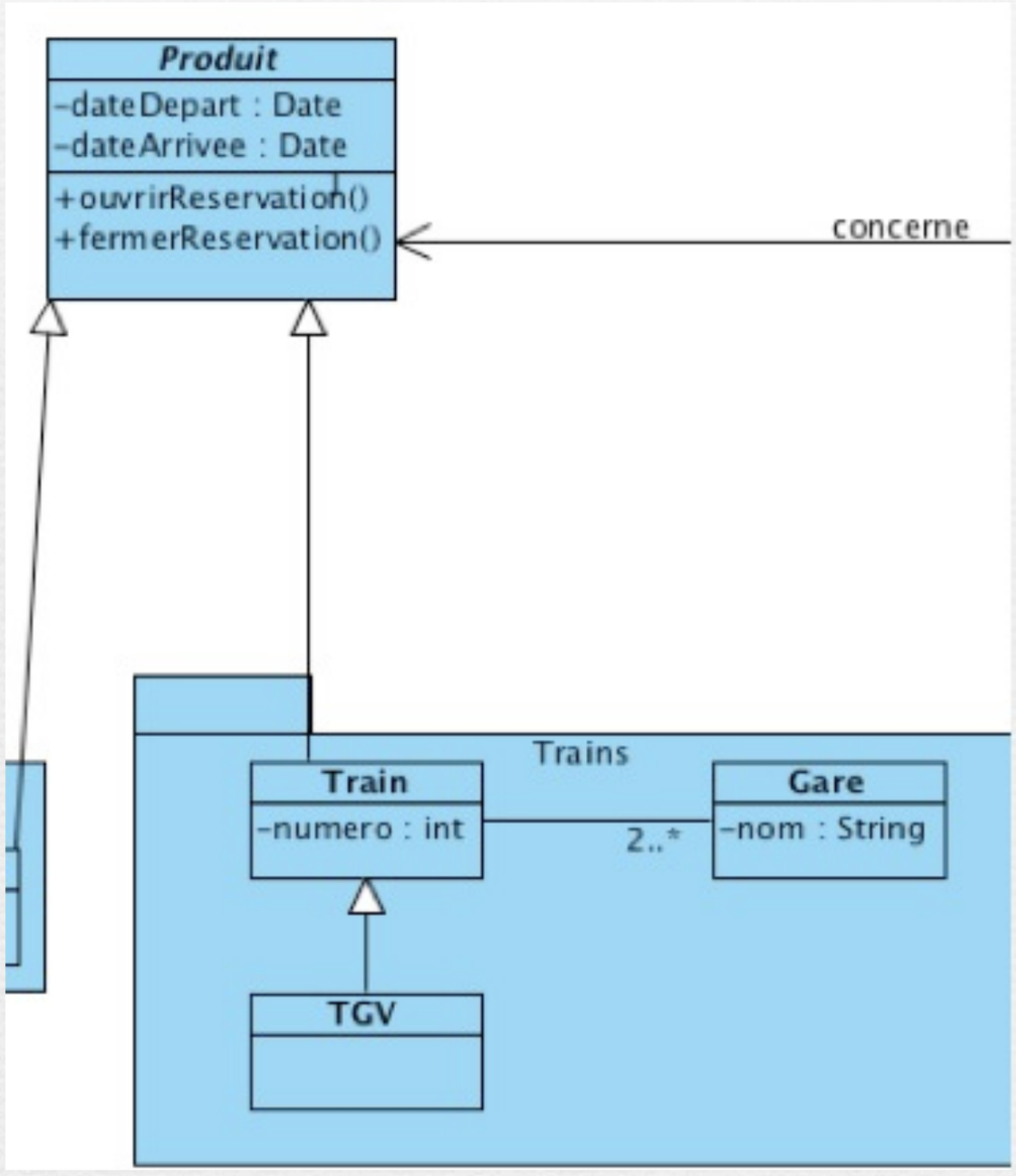


# Opérations du niveau de la classe : *Static*



# Opérations du niveau de la classe : *Static*



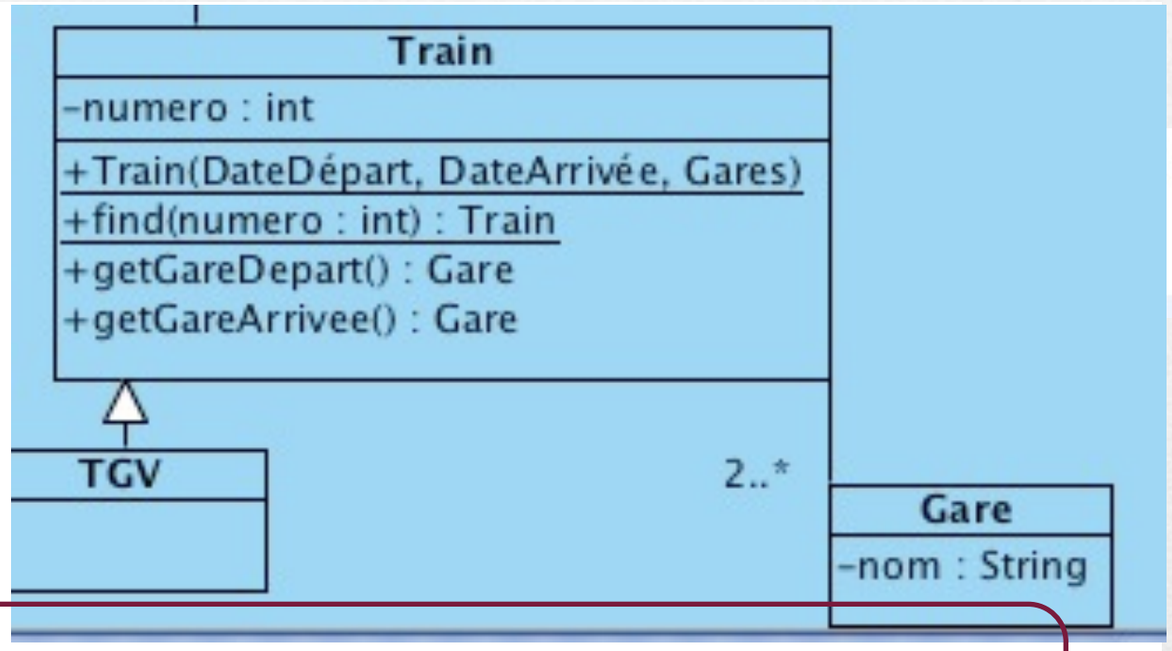

## Dans la classe Produit

```
protected Produit(Date dateDepart, Date dateArrivée) {
    this.dateDepart = dateDepart;
    this.dateArrivée = dateArrivée;
}
```

```
package trainPK;

public class Gare {
   String nom;

public String getNom() {
    return nom;
}

public void setNom(String name) {
    this.nom = name;
}

}
```

# Opérations du niveau de la classe : *Static*

```java
package trainPK;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Hashtable;

import produitPK.Produit;

public class Train extends Produit{
 int numero;
 Gare[] parcours;

 static private int NombreTrains = 0;
 static private Hashtable<Integer, Train> ListeDesTrains = new Hashtable<Integer, Train>();

 //Constructeur
 public Train(Date DateDepart, Date DateArrivee, Gare[] parcours){..

 //Obligatoire
public void afficherProduit() {..

public Gare getGareDepart(){..
public Gare getGareArrivee(){..
public static Train FIND(int numero){..

}
```



# Opérations du niveau de la classe : *Static*

```
package trainPK;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Hashtable;

import produitPK.Produit;

public class Train extends Produit{
 int numero;
 Gare[] parcours;

 static private int NombreTrains = 0;
 static private Hashtable<Integer, Train> ListeDesTrains = new Hashtable<Integer, Train>()


 //Constructeur
 public Train(Date DateDepart, Date DateArrivee, Gare[] parcours){
     super(DateDepart,DateArrivee);
     this.parcours = parcours;
     NombreTrains++;
     numero = NombreTrains;
     ListeDesTrains.put(numero,this);
 }
}
```

# Opérations du niveau de la classe : *Static*

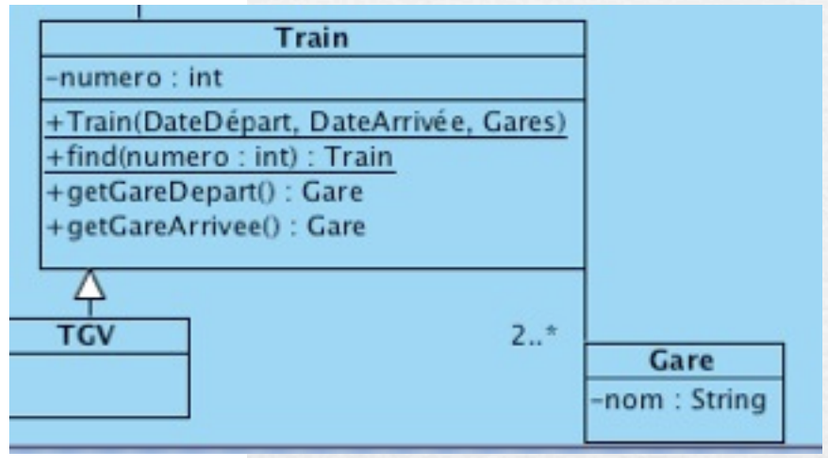
```
package trainPK;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Hashtable;

import produitPK.Produit;

public class Train extends Produit{
 int numero;
 Gare[] parcours;

 static private int NombreTrains = 0;
 static private Hashtable<Integer, Train> ListeDesTrains = new Hashtable<Integer, Train>()


 public static Train FIND(int numero){
     return ListeDesTrains.get(numero);
 }

    ListeDesTrains.put(numero,this);
 }
}
```

# Opérations du niveau de la classe : *Static*

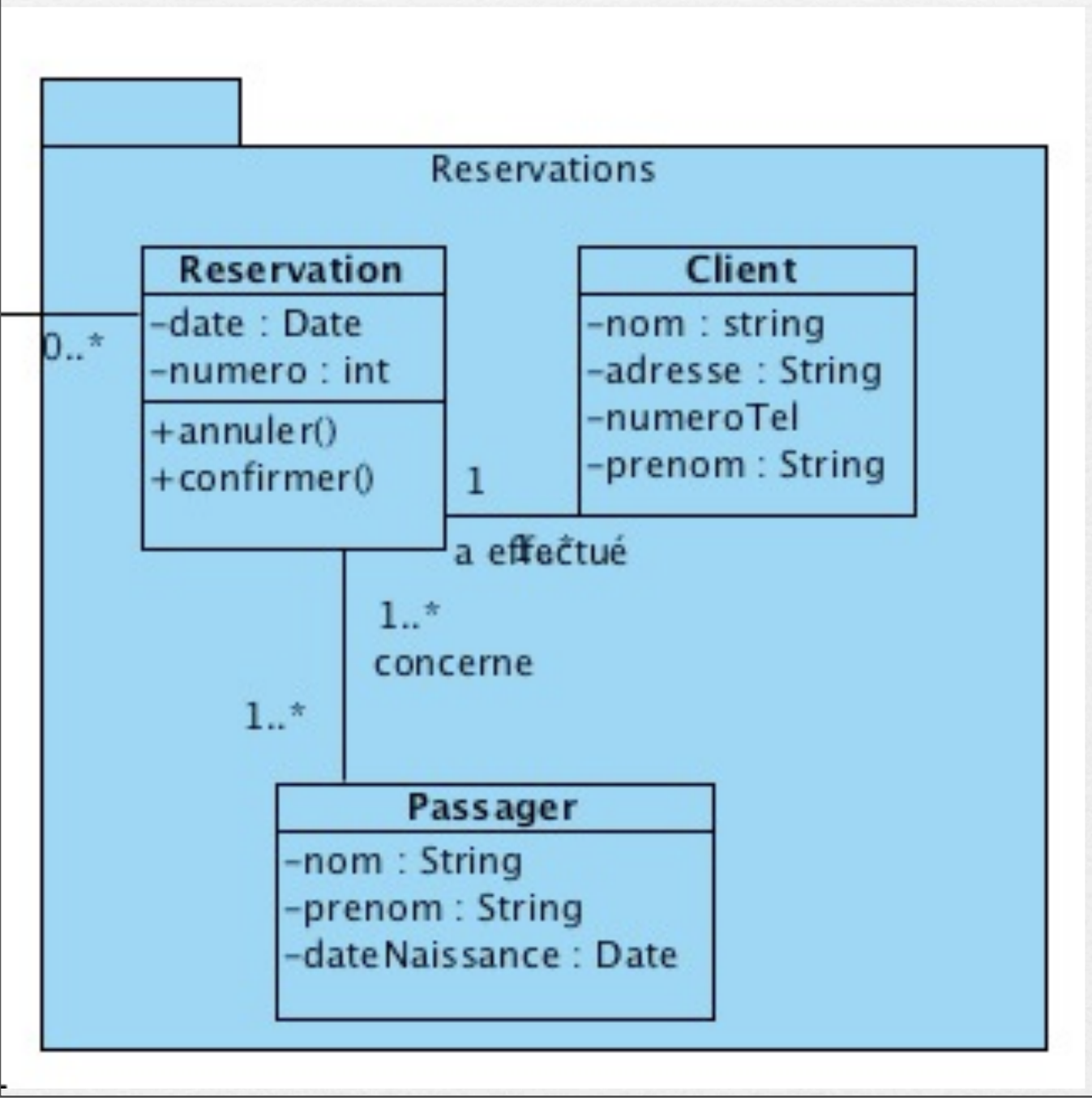
```
package trainPK;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Hashtable;

import produitPK.Produit;

public class Train extends Produit{
 int numero;
 Gare[] parcours;
  //Obligatoire
@Override
public void afficherProduit() {
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd:hh:mm");
    System.out.print("train " + numero + " de ");
    System.out.print(parcours[0].getNom());
    System.out.println( " a " + parcours[parcours.length-1].getNom());
    System.out.println( dateFormat.format(this.getDateDepart()) + " -- " +
                        dateFormat.format(this.getDateArrivee()) );
}

public Gare getGareDepart(){
    return parcours[0];
}
public Gare getGareArrivee(){
    return parcours[parcours.length-1];
```

# Opérations du niveau de la classe : *Utilisation*

```java
public class TestTrains {

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception{
        //Pas joli : il faudrait un constructeur
        Gare nice = new Gare();
        nice.setNom("Nice");
        Gare antibes = new Gare();
        antibes.setNom("antibes");

        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd:hh:mm");

        // date to string
        Date depart = dateFormat.parse("2011-03-12:08:00");
        System.out.println("Depart :  "+dateFormat.format(depart));
        Date arrivee = dateFormat.parse("2011-03-12:08:35");
        Train tMatin = new Train(depart, arrivee, new Gare[]{nice,antibes});
        tMatin.afficherProduit();

        Train tSoir = new Train(dateFormat.parse("2011-03-12:19:00"),
                dateFormat.parse("2011-03-12:19:40"), new Gare[]{antibes,nice});
        tSoir.afficherProduit();

        System.out.println("----------------- ");
        System.out.println("Train du matin ");
        Train.FIND(1).afficherProduit();
        System.out.println("Train du soir ") ;
        Train.FIND(2).afficherProduit();
    }
```

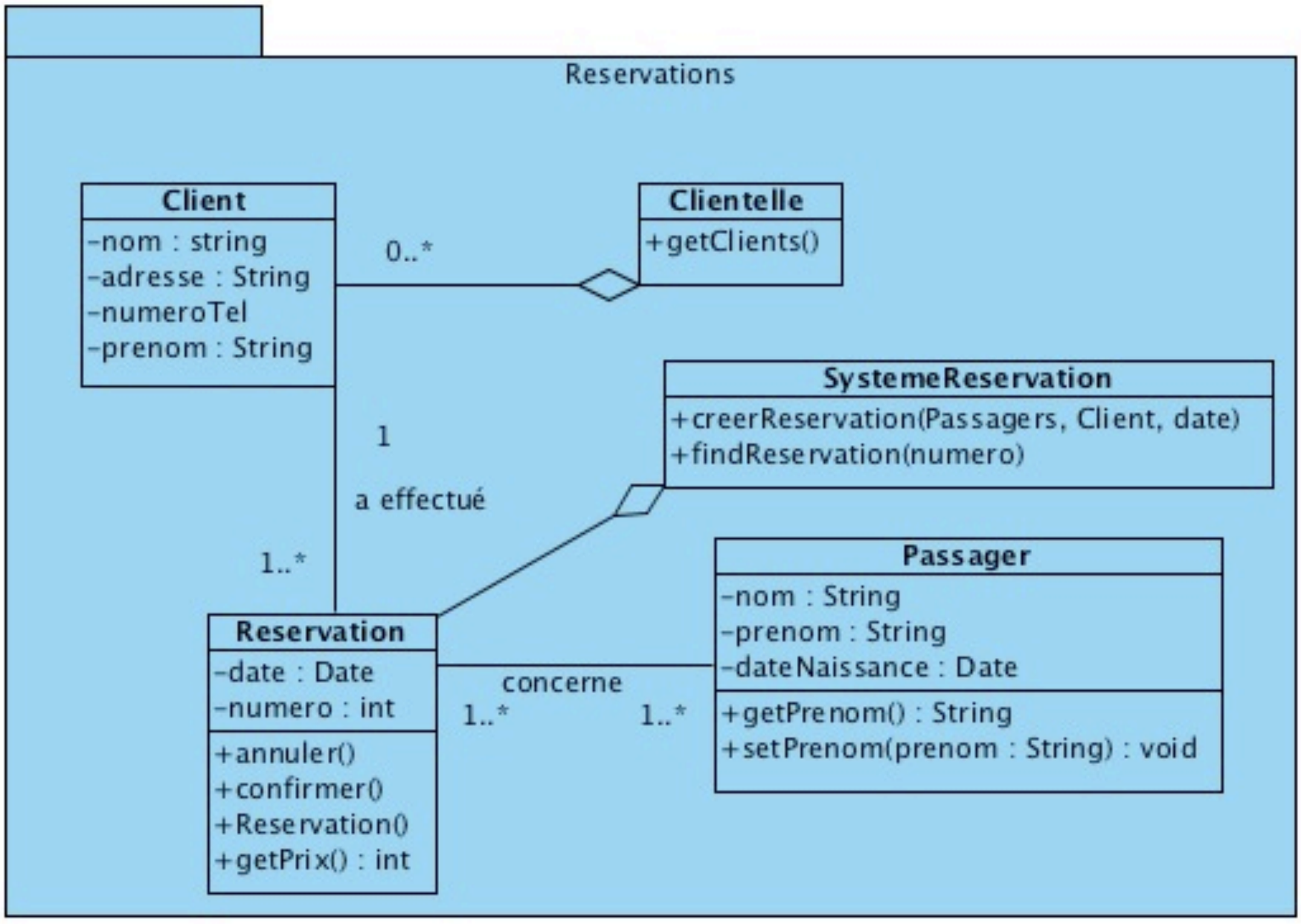


# Opérations du niveau de la classe : *Static*




1. Obtenir la liste des clients
2. Modifier la date d'une réservation
3. Créer une réservation
4. Modifier le prénom d'un passager
5. Calculer le prix d'une réservation

# Propriété Statique ou Classe dédiée (Factory)



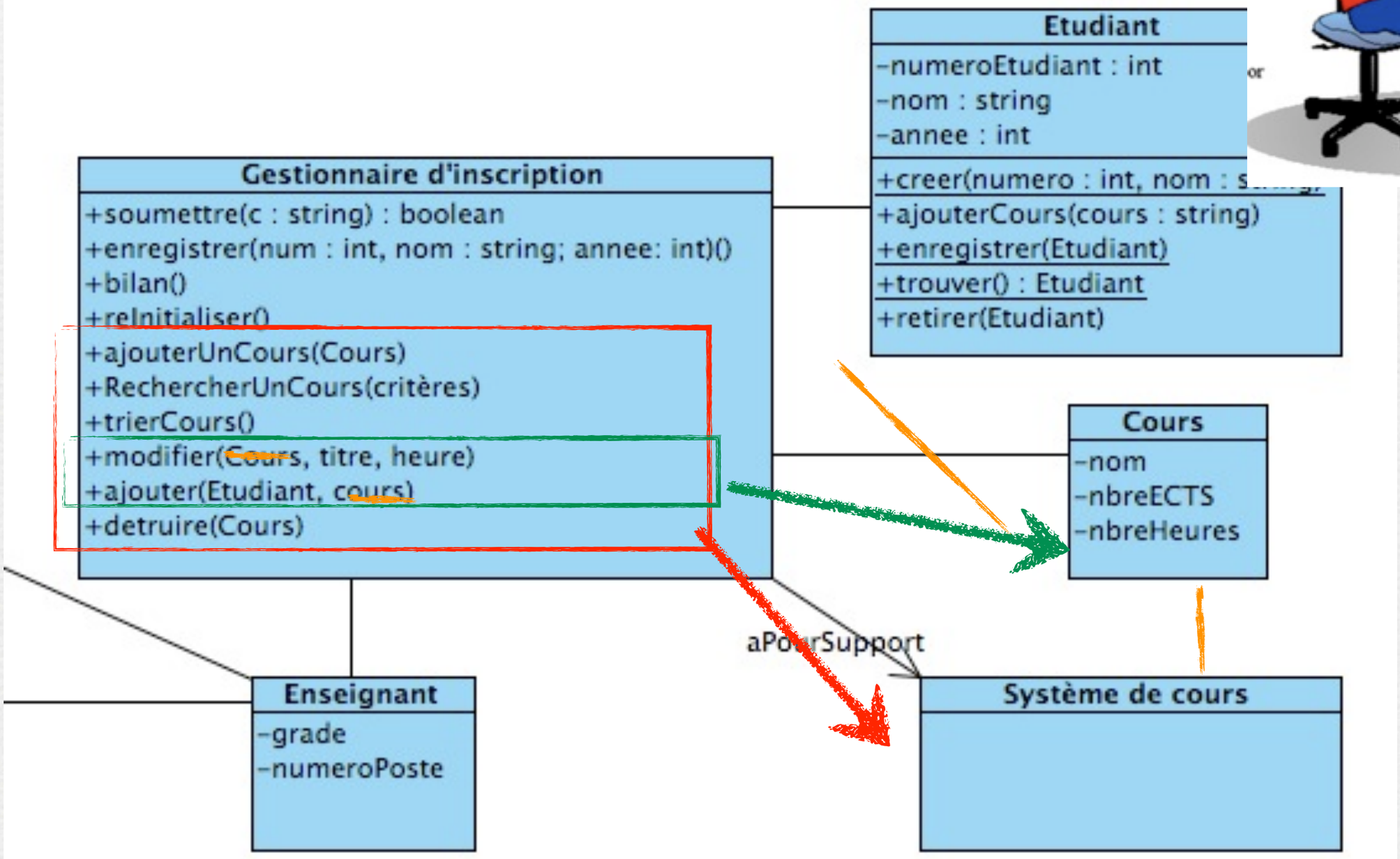
# Vers la mise en oeuvre des classes

- Visibilité
- Abstraction
- Attributs et Opérations* de Classes
- Généralisation
- Packages
- Transformations des associations
- Anti-Patterns

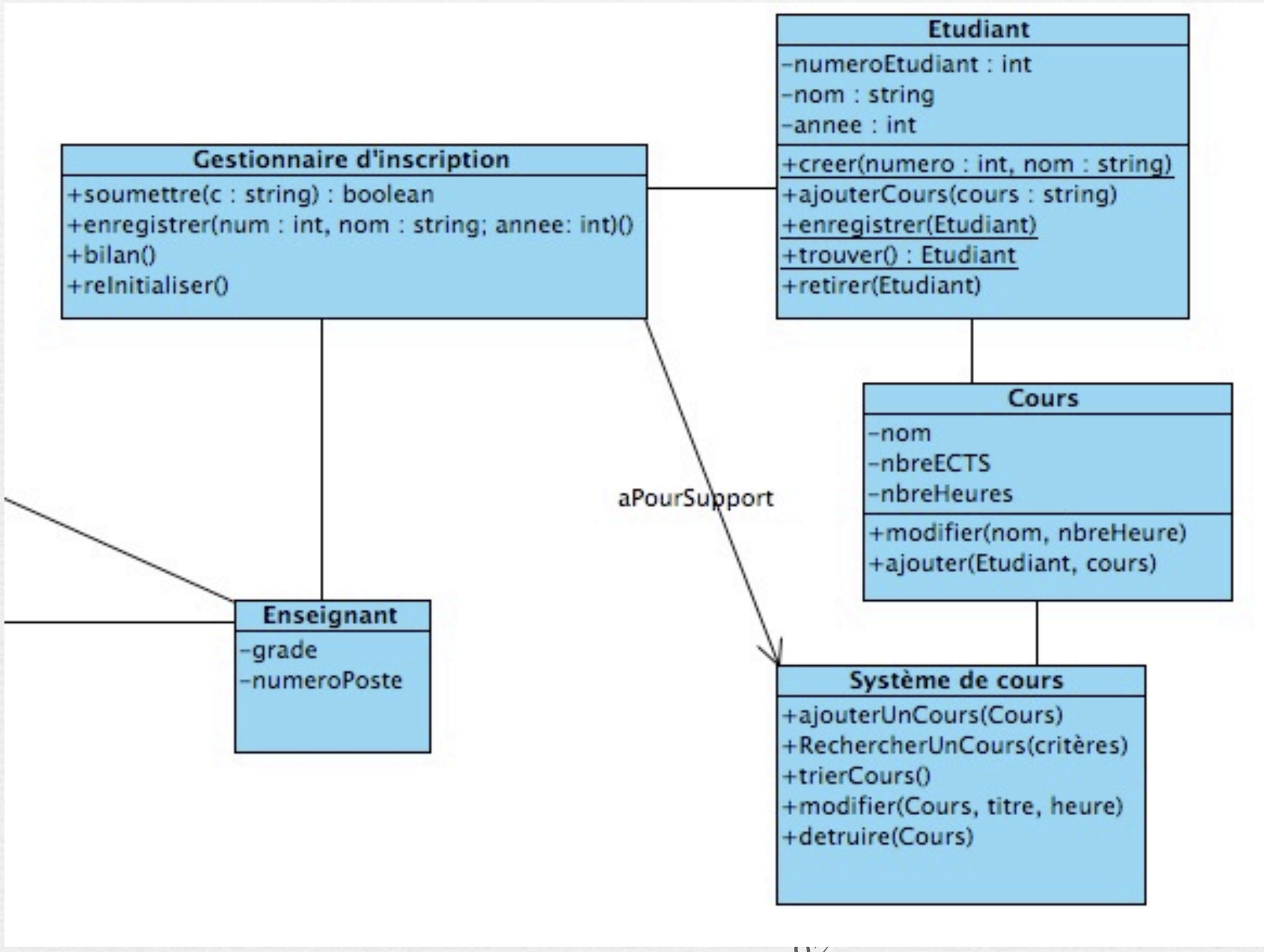
Opération : terme générique désignant le plus souvent des méthodes

# Anti-Patterns

## Blob ou la classe Dieu



# Anti-Patterns




A
Améliorer
encore...