De l'analyse à la conception détaillée

Mireille Blay-Fornarino
S2T
blay@unice.fr

Rappels sur les concepts d’Objets

Objets & Classes

- Il n’est pas possible de créer une classe dynamiquement
  - Mais on peut créer des objets « instances d’une classe » dynamiquement
- 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!

Principes objets

- **Encapsulation** : regroupement des informations d'état et de comportement sous un nom unique

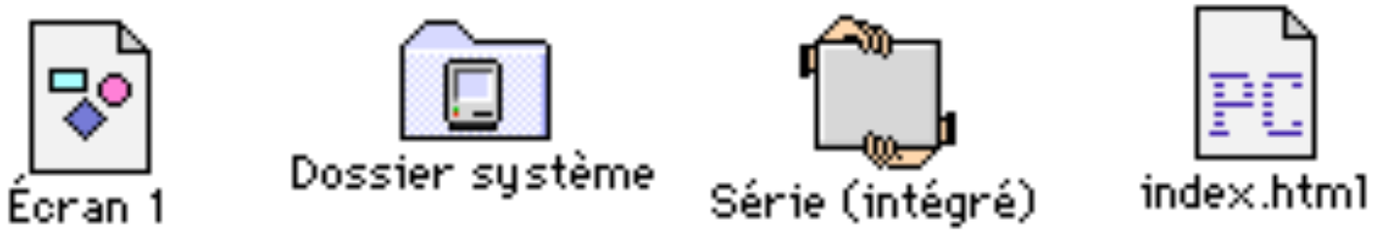
- **Masquage d'information** : on ne connaît pas la structure de l'information

- **Interface** : seuls les services publics (offerts à l'extérieur) sont utilisables

- **Envoi de messages** : les objets communiquent par messages

- Et le **Polymorphisme**

Polymorphisme



Supprimer, Déplacer, Ouvrir ....

Q1 à Q6

De l’analyse à la conception

Quelques principes

Stakeholder layer
(language of the problem)

Aircraft

= physical entity

System layer
(language of the abstract solution)

Aircraft
Type Status

= logical representation

Design layer
(language of the design solution)

Aircraft
Type Status
loadBaggage()

= object implemented

De l'analyse à la conception des classes

Différents niveaux de modélisation

- 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, ...)

Différents niveaux de modélisation : exemple

Analysis

Order
Placement Date Delivery Date Order Number
Calculate Total Calculate Taxes

Design

Order
- deliveryDate: Date - orderNumber: int - placementDate: Date - taxes: Currency - total: Currency
calculateTaxes(Country, State): Currency # calculateTotal(): Currency 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

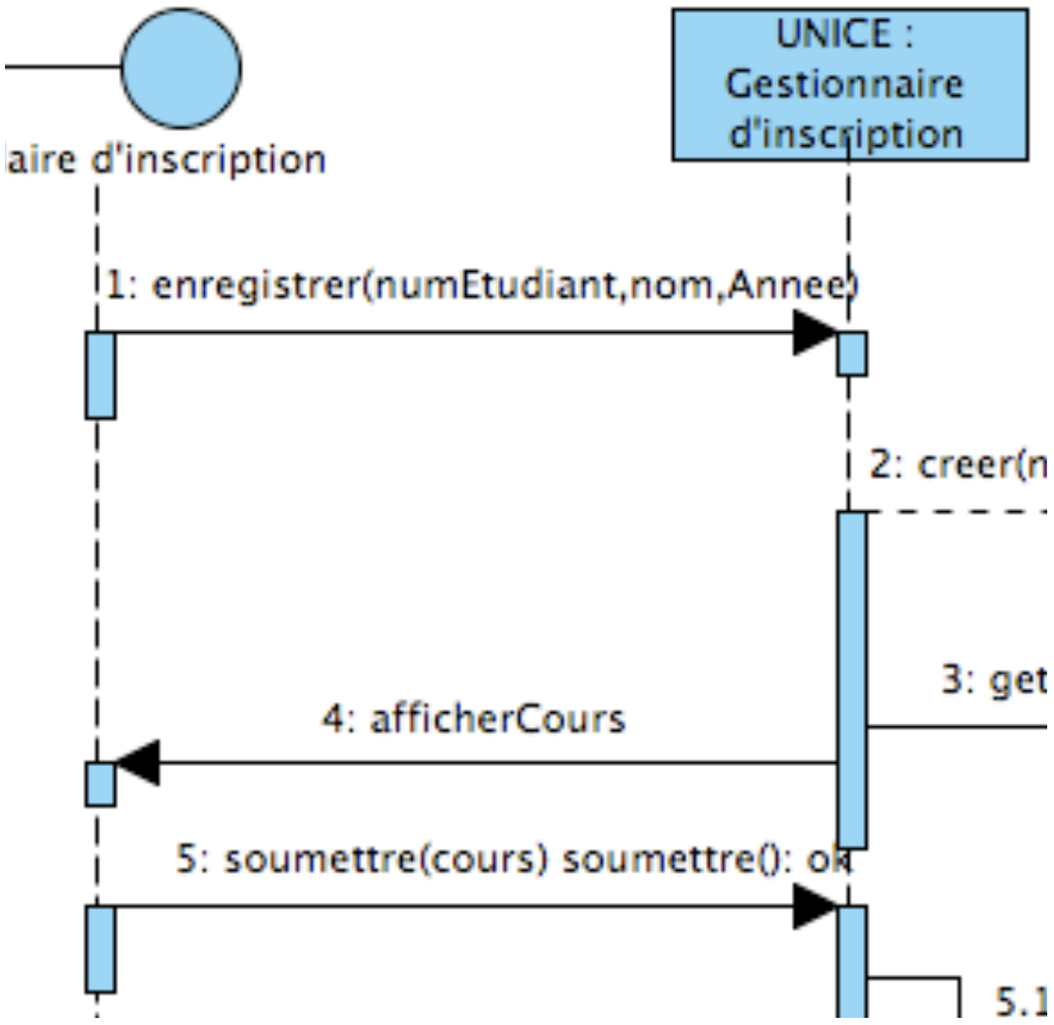
De l’analyse à la conception

- Des diag. de séquences aux Diag. de classes
- Diagrammes de classes en conception
- Anti-Patterns

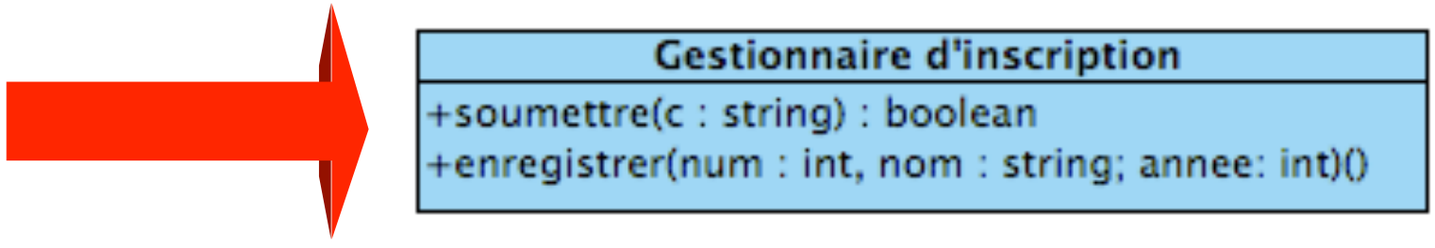
➡ Diag. de séquence en conception

Des diagrammes de séquence au diagrammes de classes

Des diagrammes de séquence aux classes : opérations

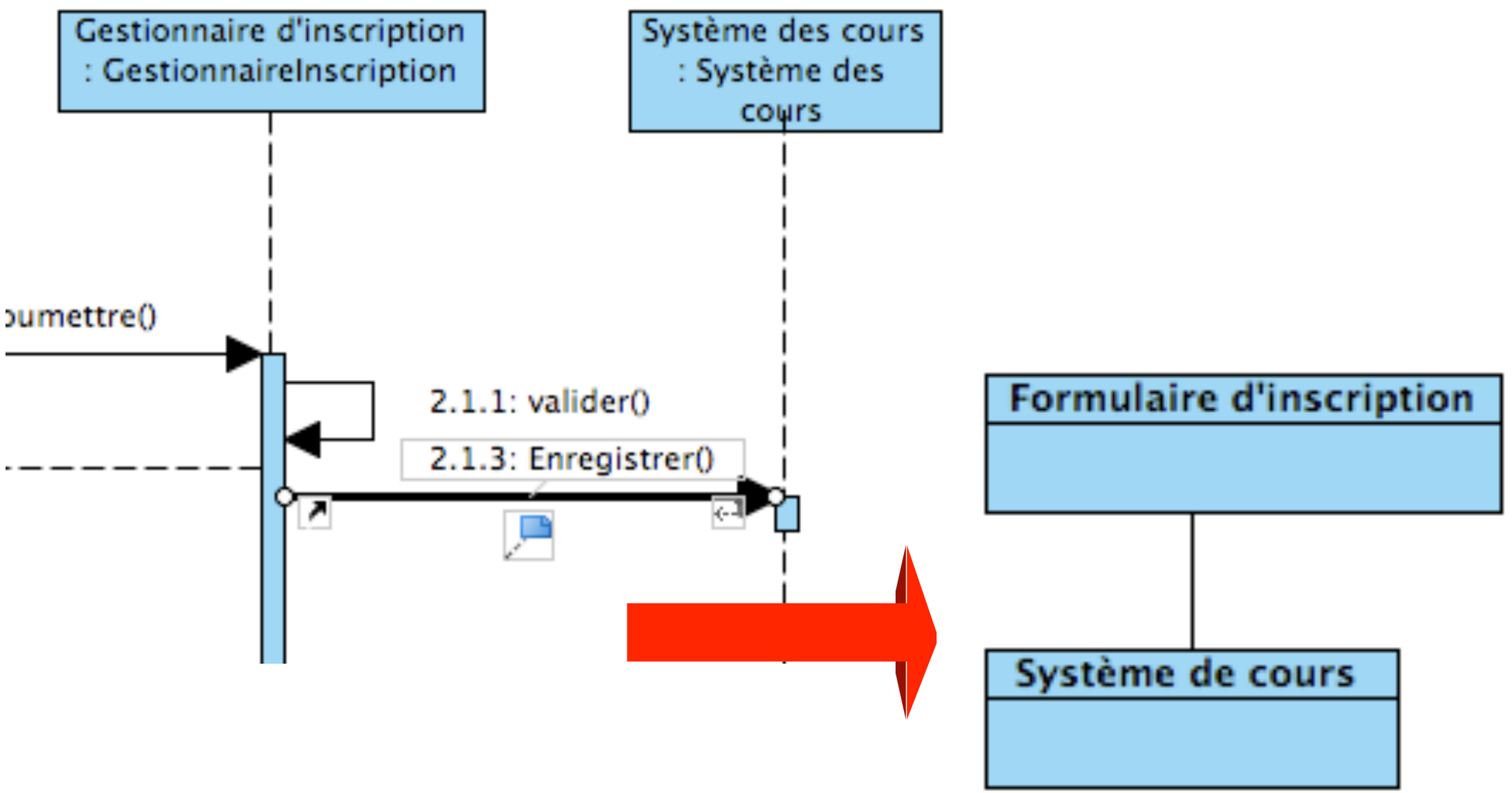


- Le comportement d'une classe est constitué de ses opérations
- On identifie les opérations en examinant les diagrammes de séquences

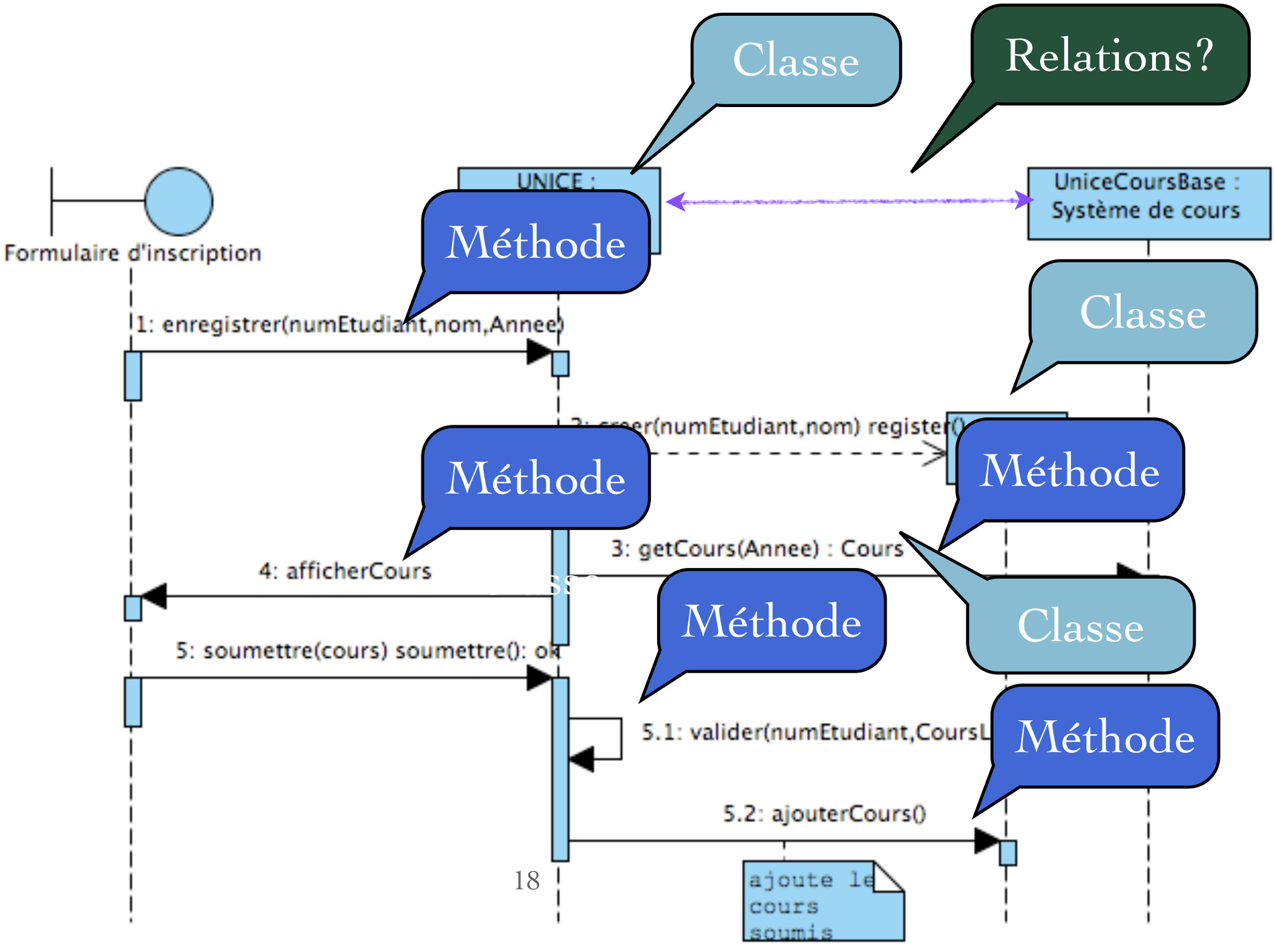


Des diagrammes de séquence aux classes : opérations

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



Des diagrammes de séquence aux classes



Diagrammes de classes en conception

On veut se rapprocher du code sans s'y perdre :
La phase de conception doit permettre

d'identifier l'architecture du système *(les objets et leur mode de communication pour vous)*,
&
de déterminer et réduire les dépendances
(comprendre comment cela fonctionnera, un programme ce n'est pas de la MAGIE! S'il ne fonctionne pas c'est de votre FAUTE, pas celle de la machine…) ,

Le langage reste encore accessoire.

Des diagrammes de classes en conception

Packages

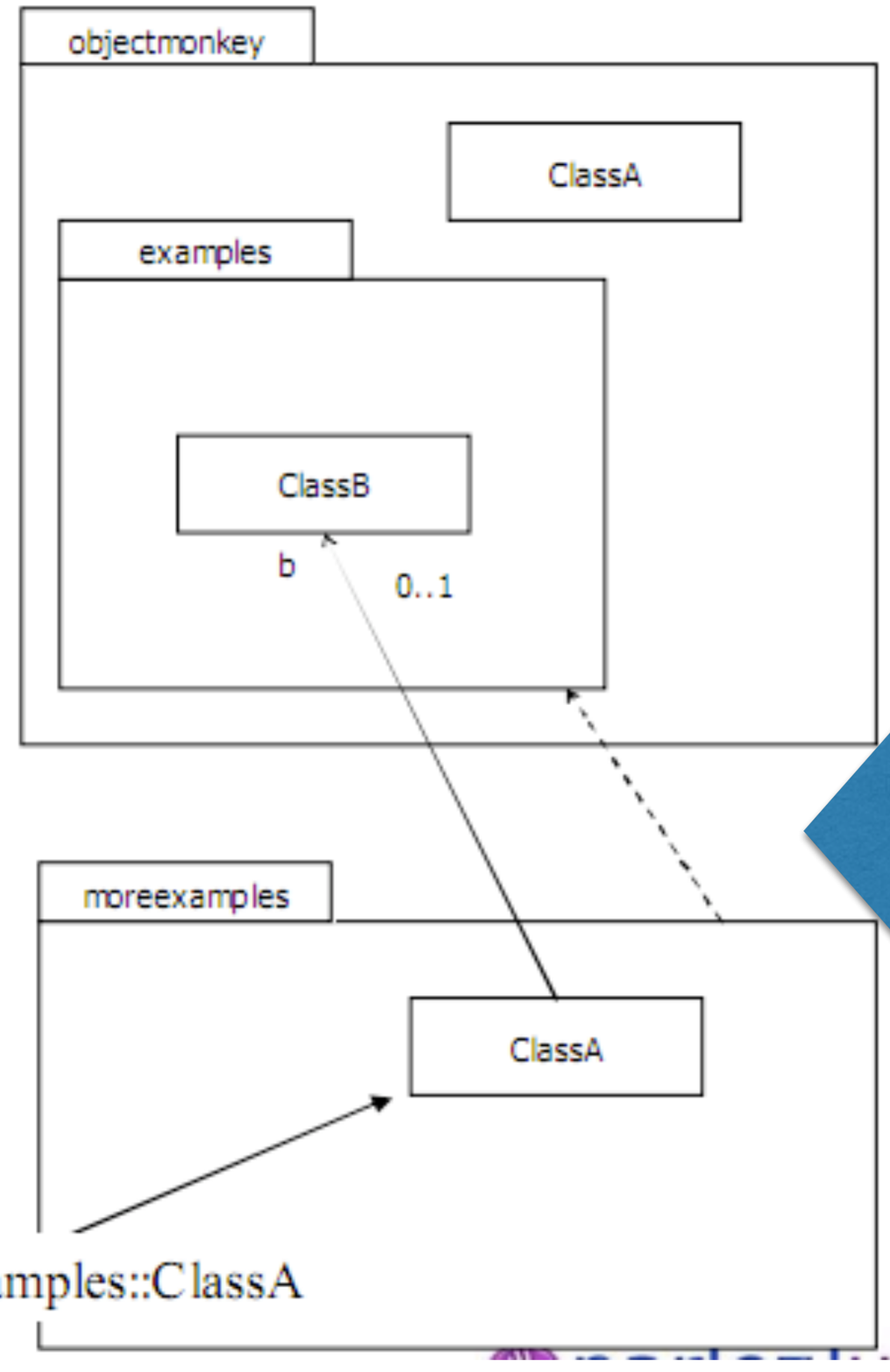
Structuration en packages

A **package** in the UML 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.

Cohérence et Indépendance

- Minimiser les dépendances
- Eviter les dépendances mutuelles

Packages

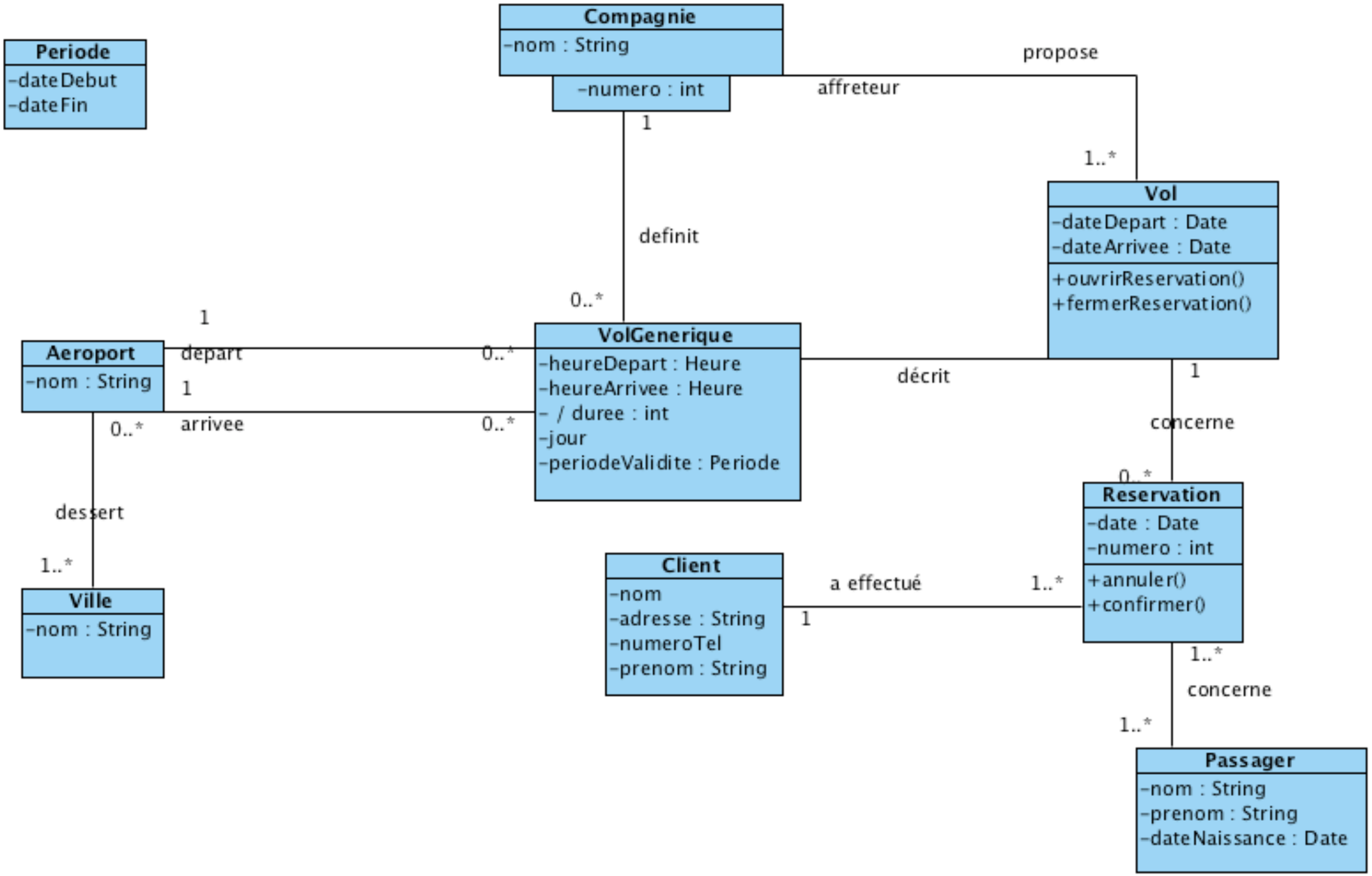



```
package objectmonkey;


class ClassA
{
}
```

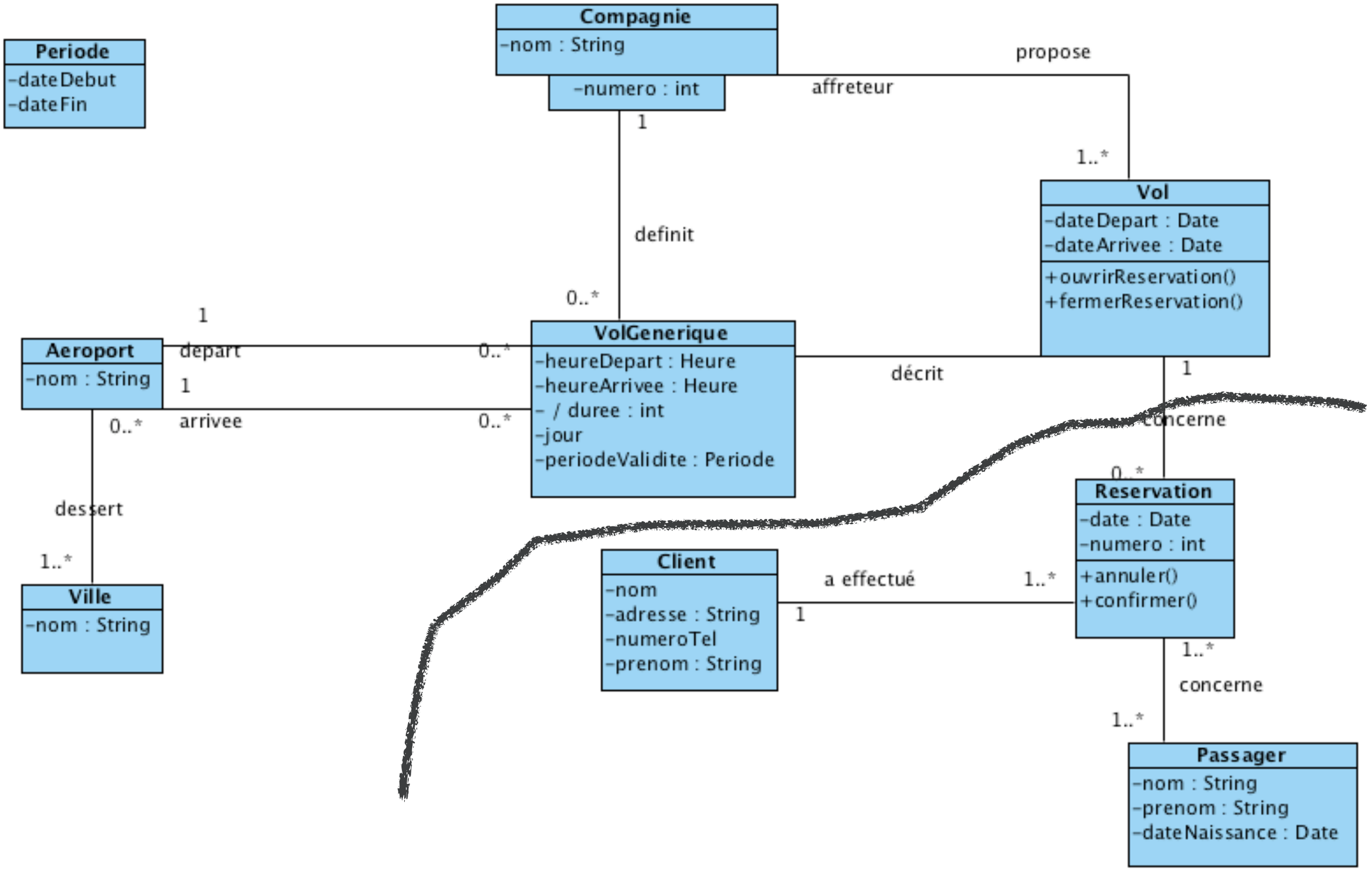
```
package objectmonkey.examples;


class ClassB
{
}
```

```
package moreexamples;

import objectmonkey.examples.*;


class ClassA
{
                private ClassB b;
}
```

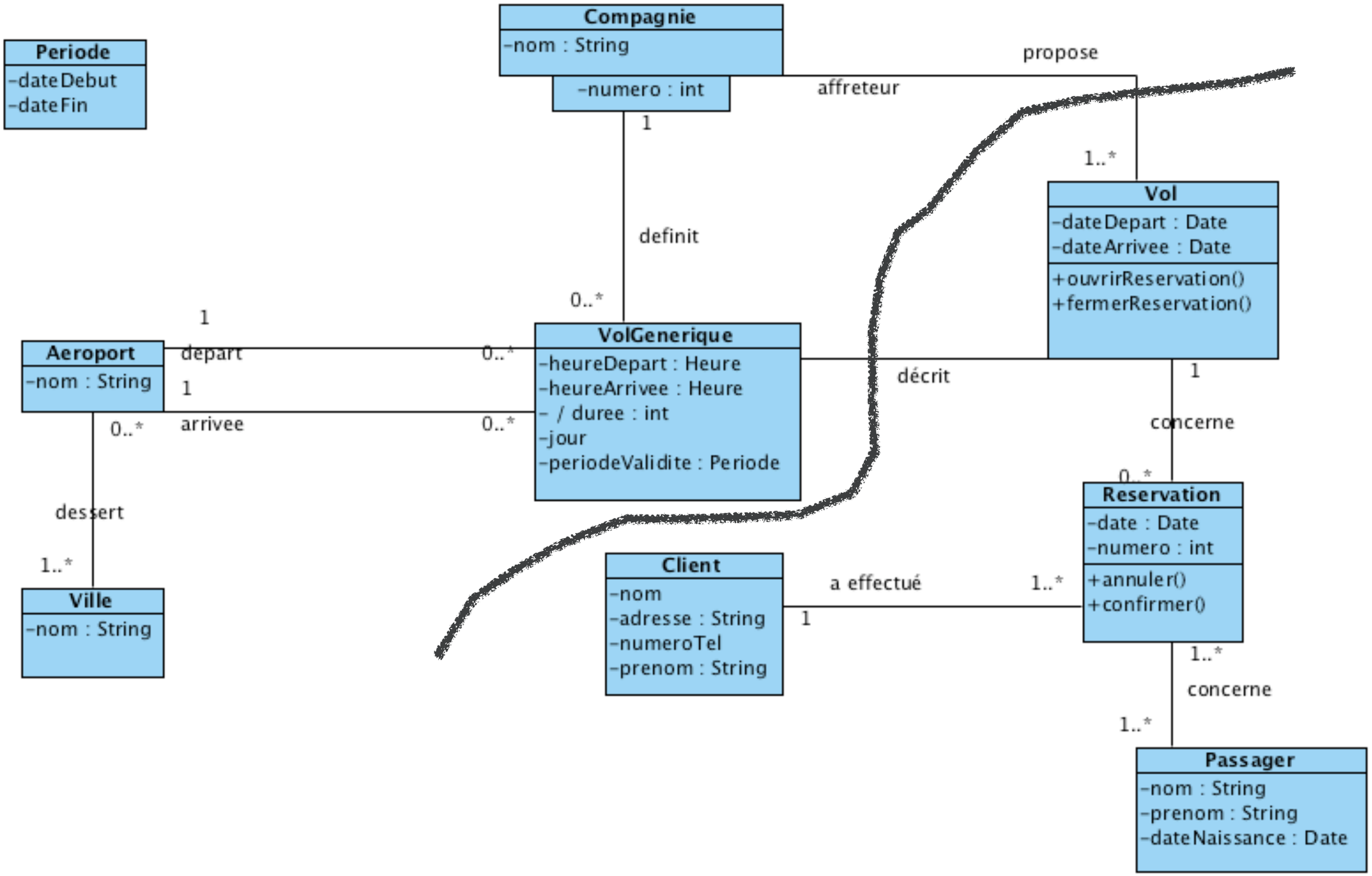
Structuration en packages



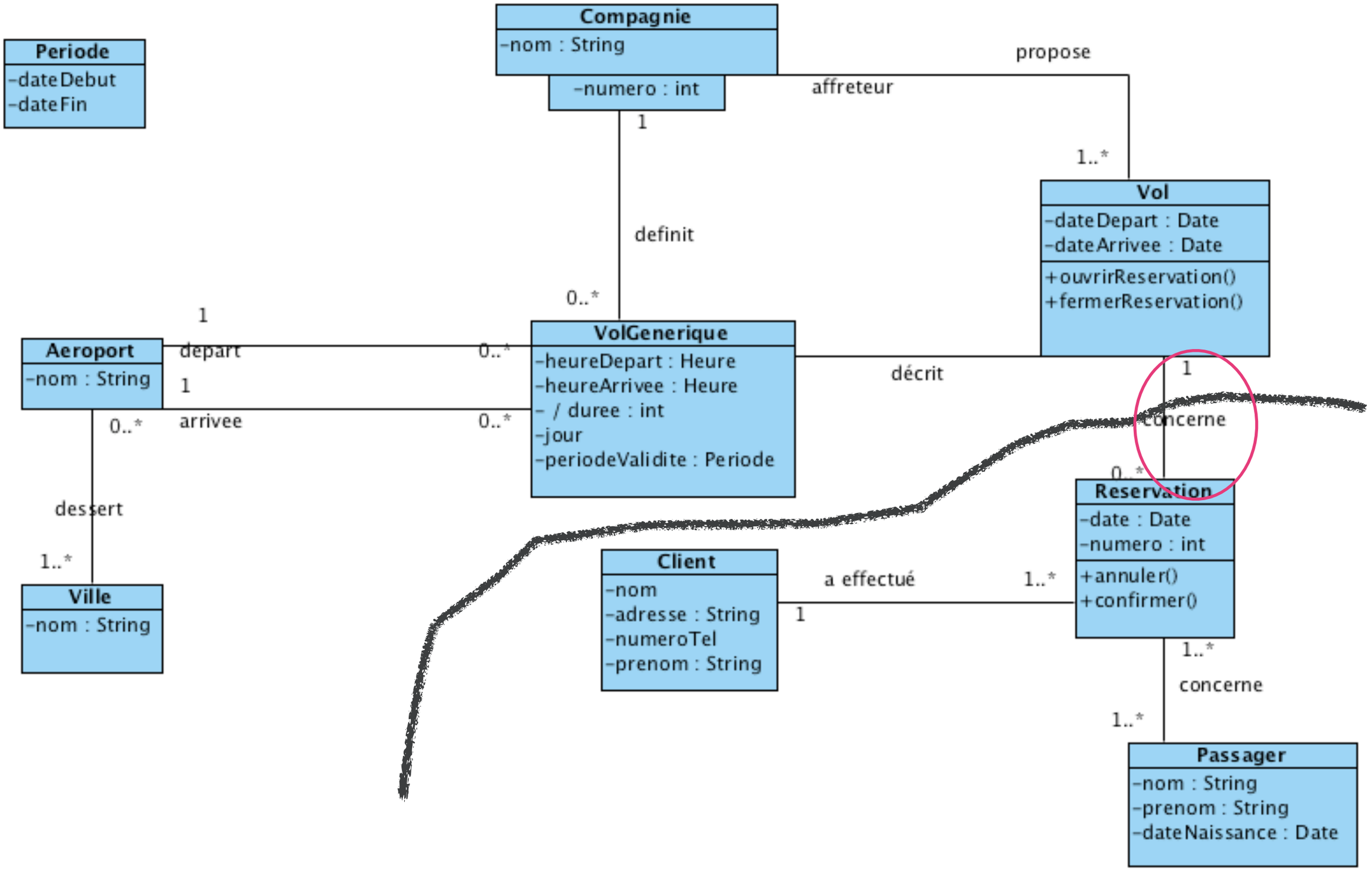
Structuration en packages



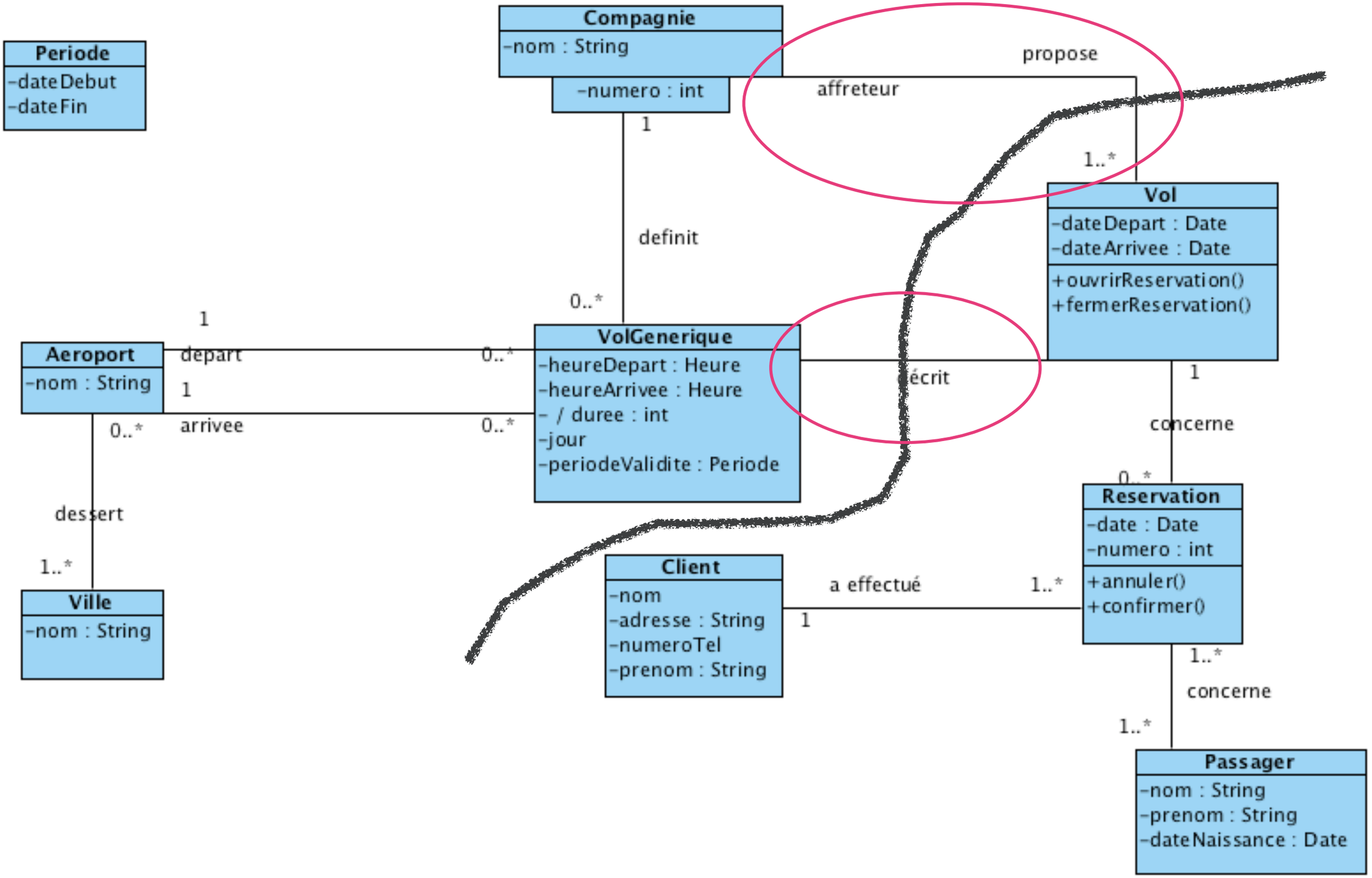
Structuration en packages



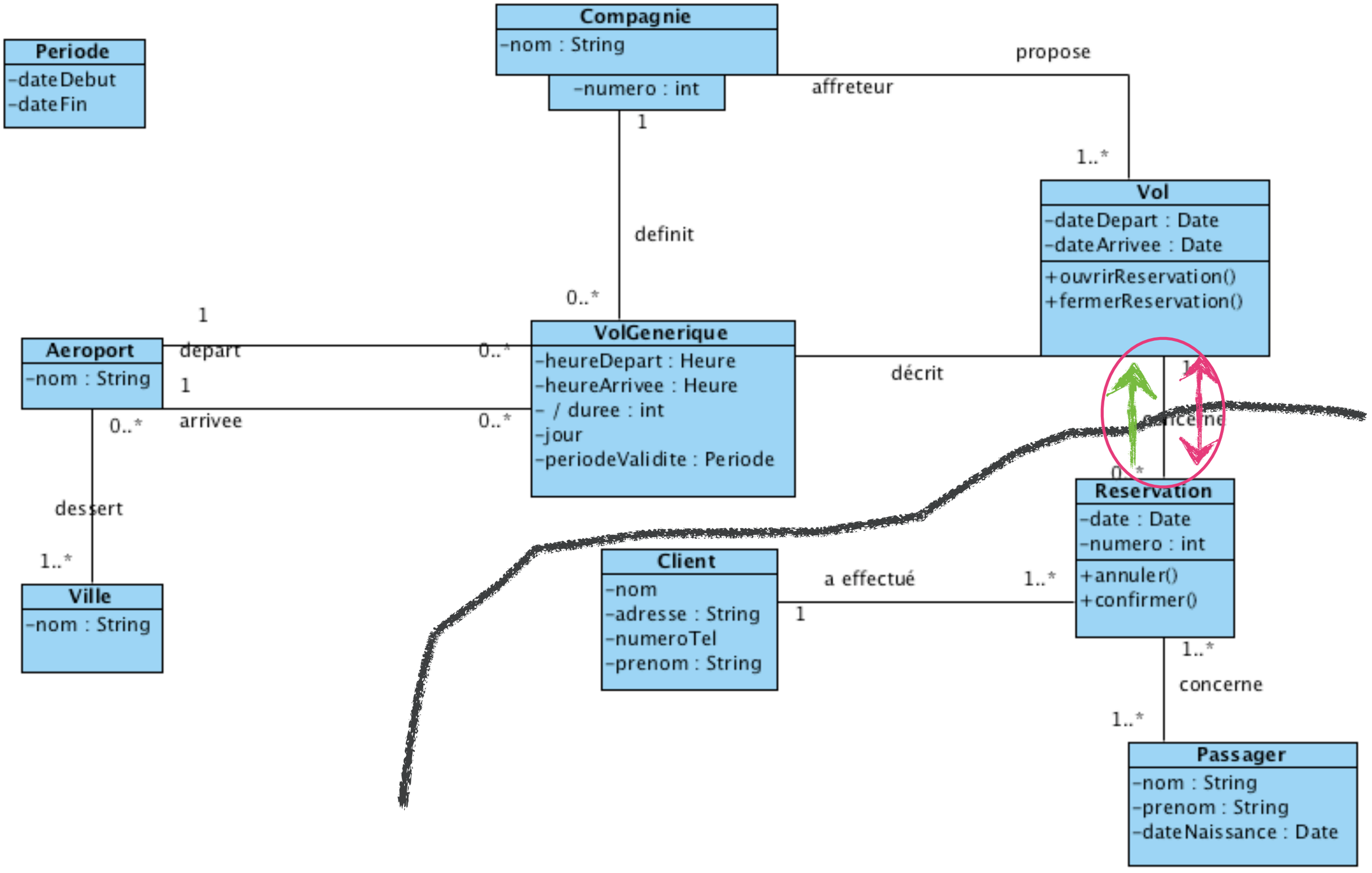
Structuration en packages



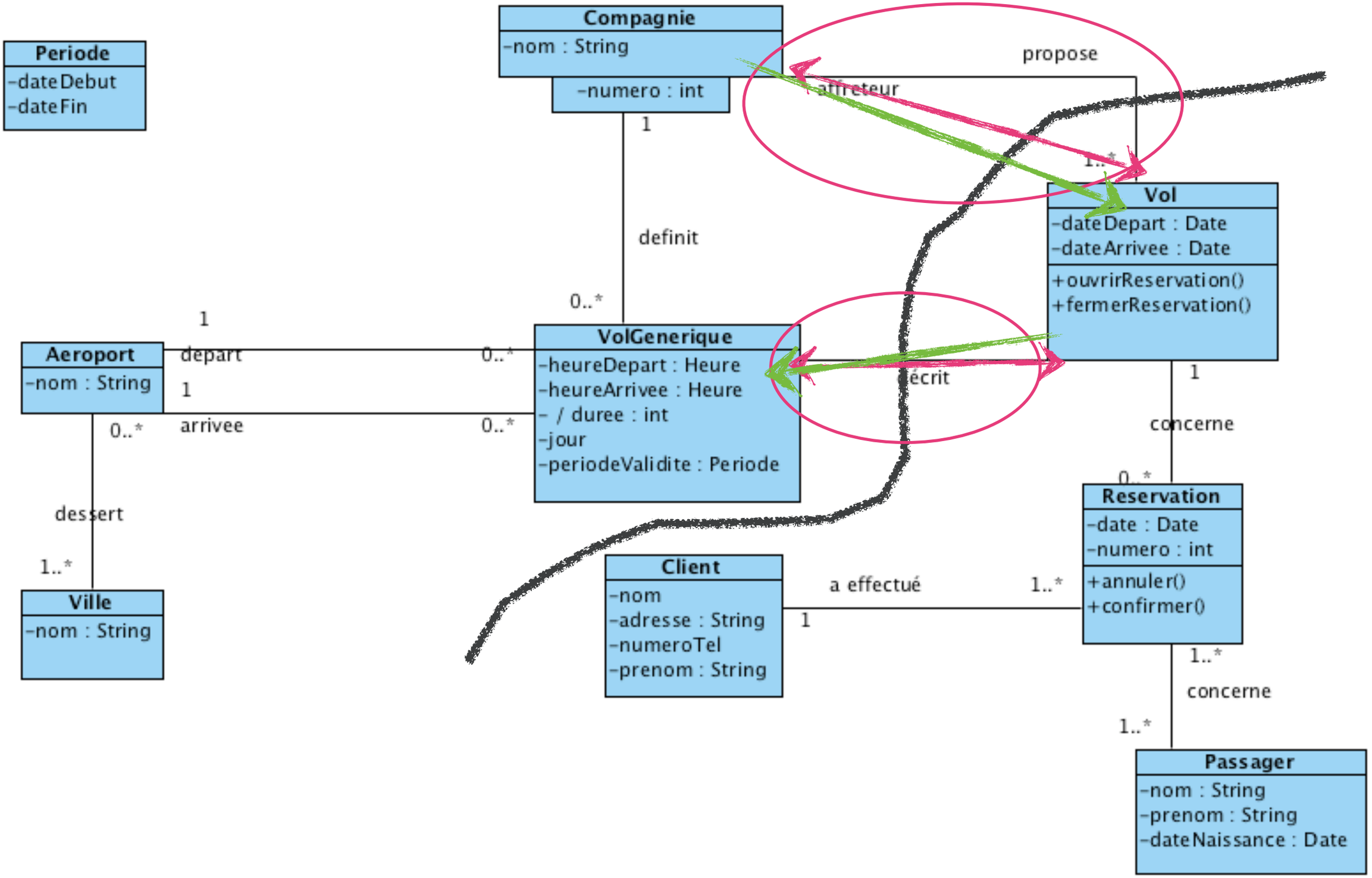
Structuration en packages



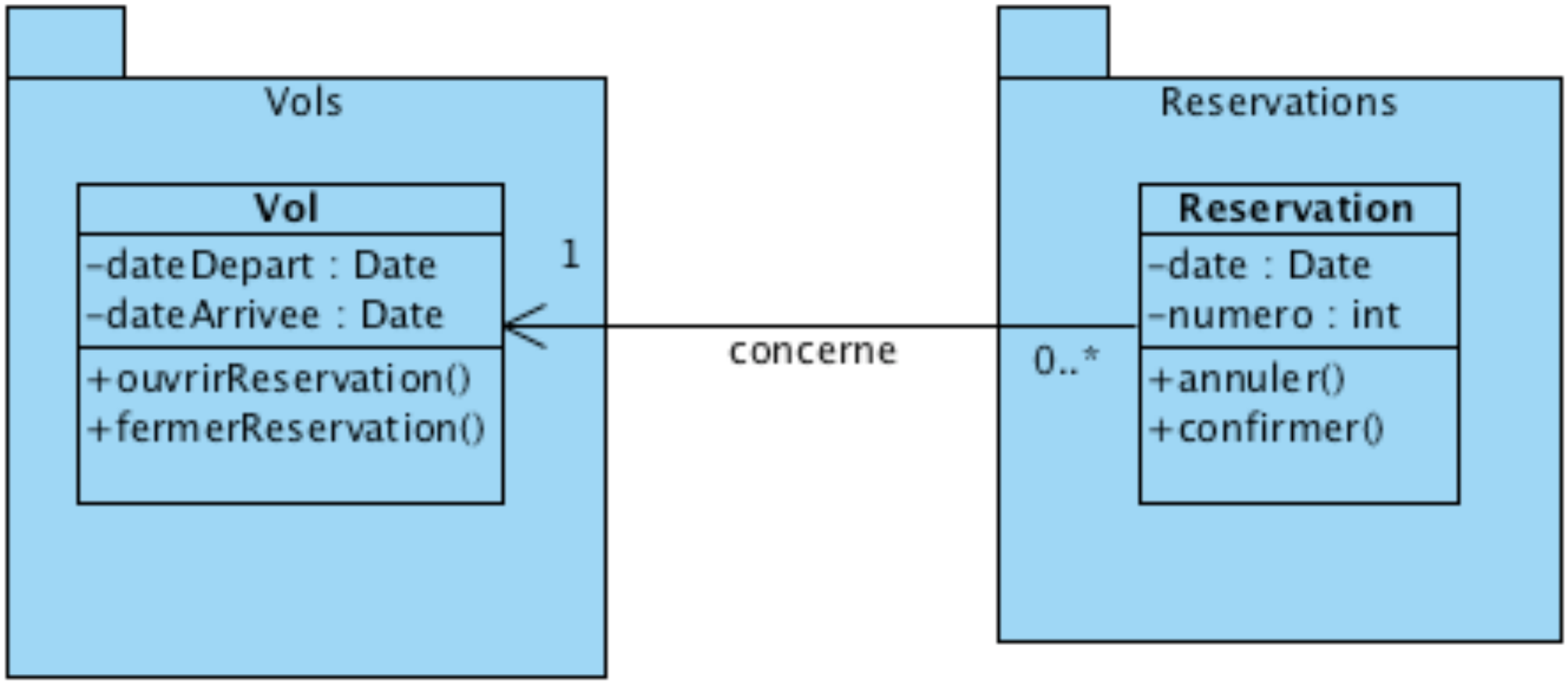
Structuration en packages



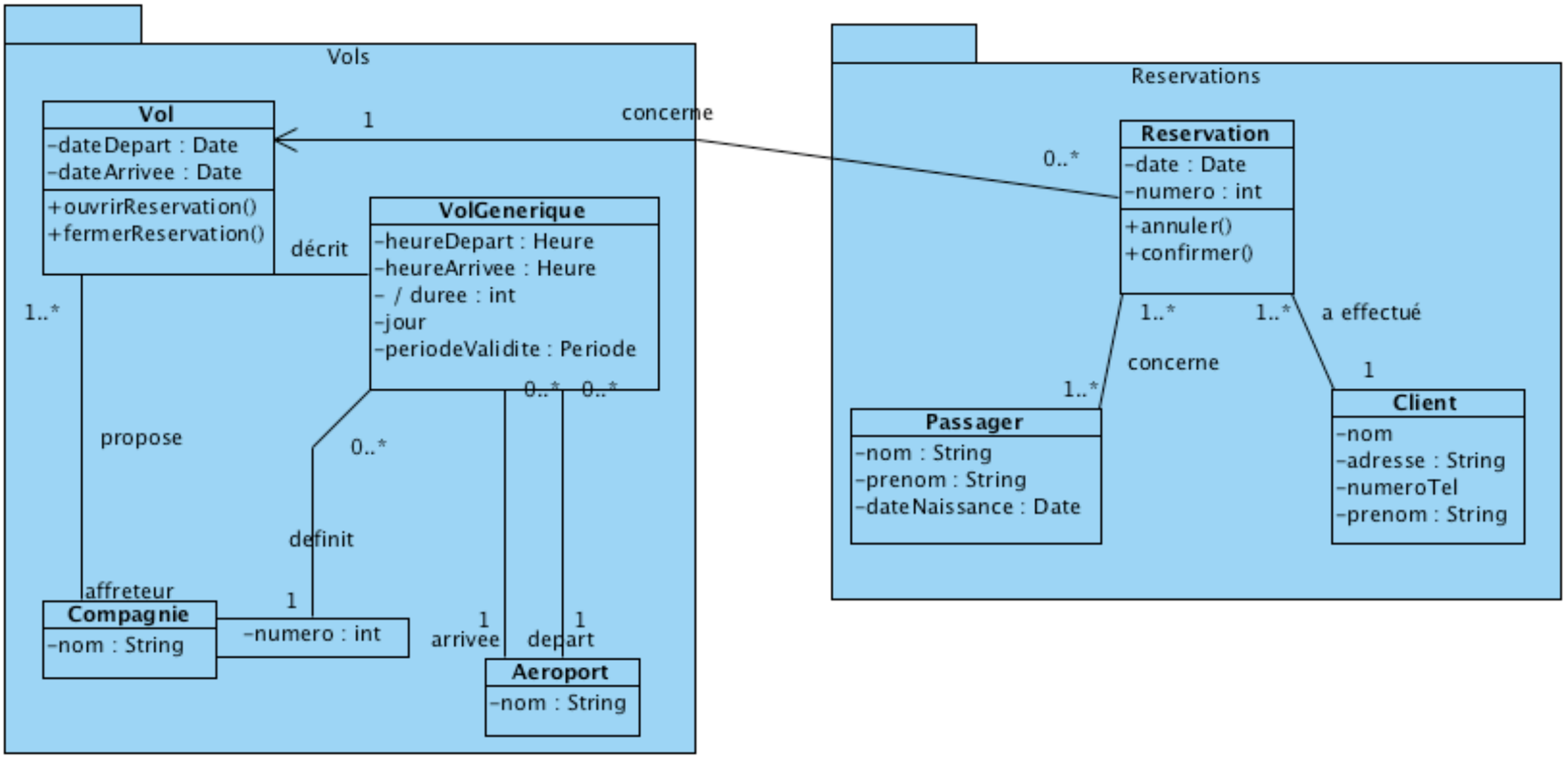
Structuration en packages



Structuration en packages

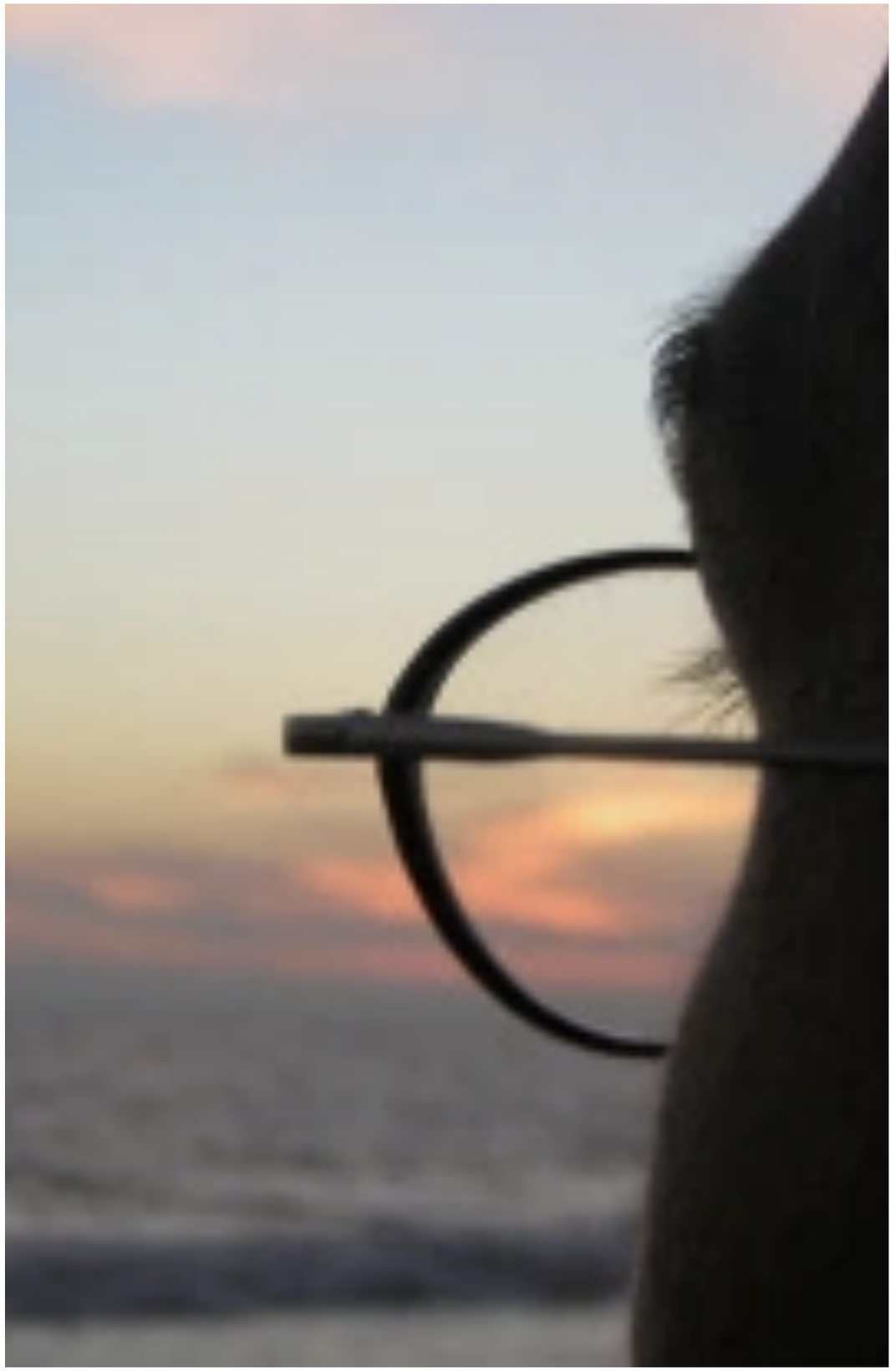


Structuration en packages

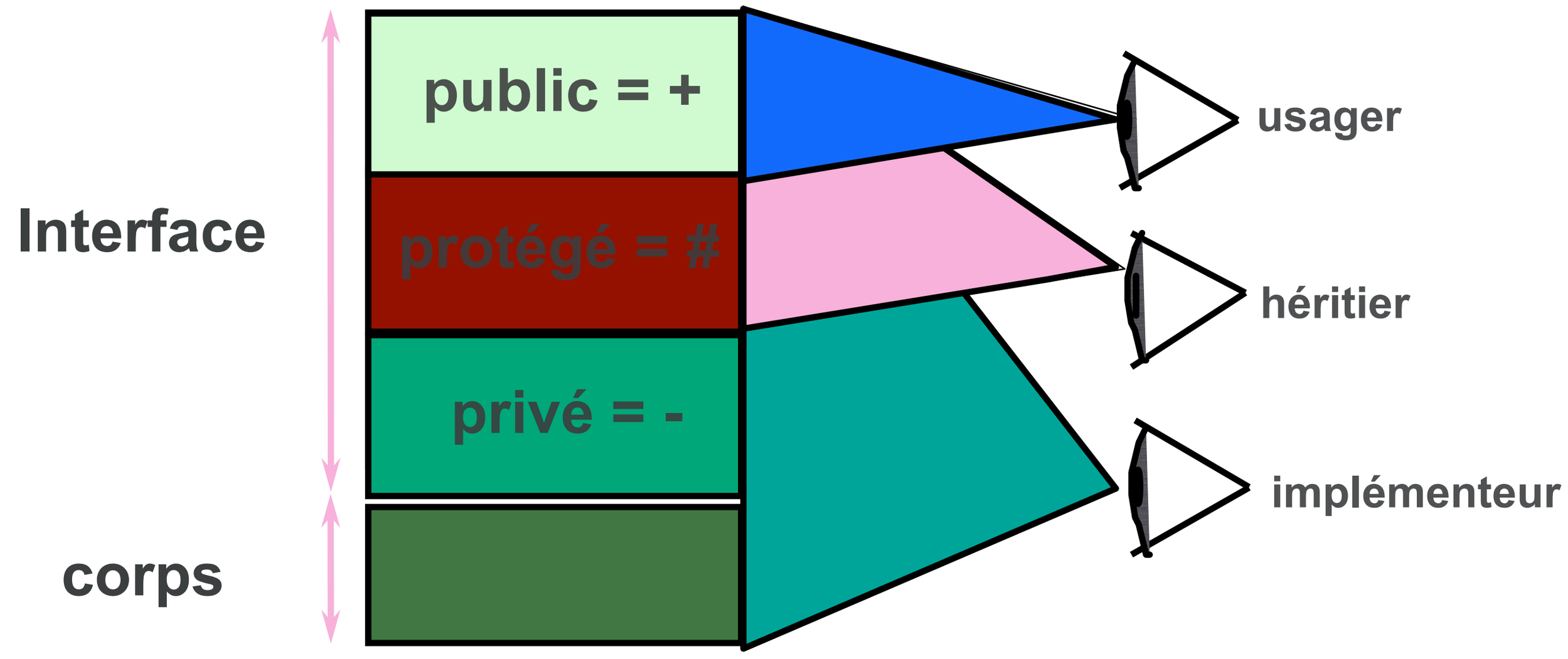


Des diagrammes de classes en conception

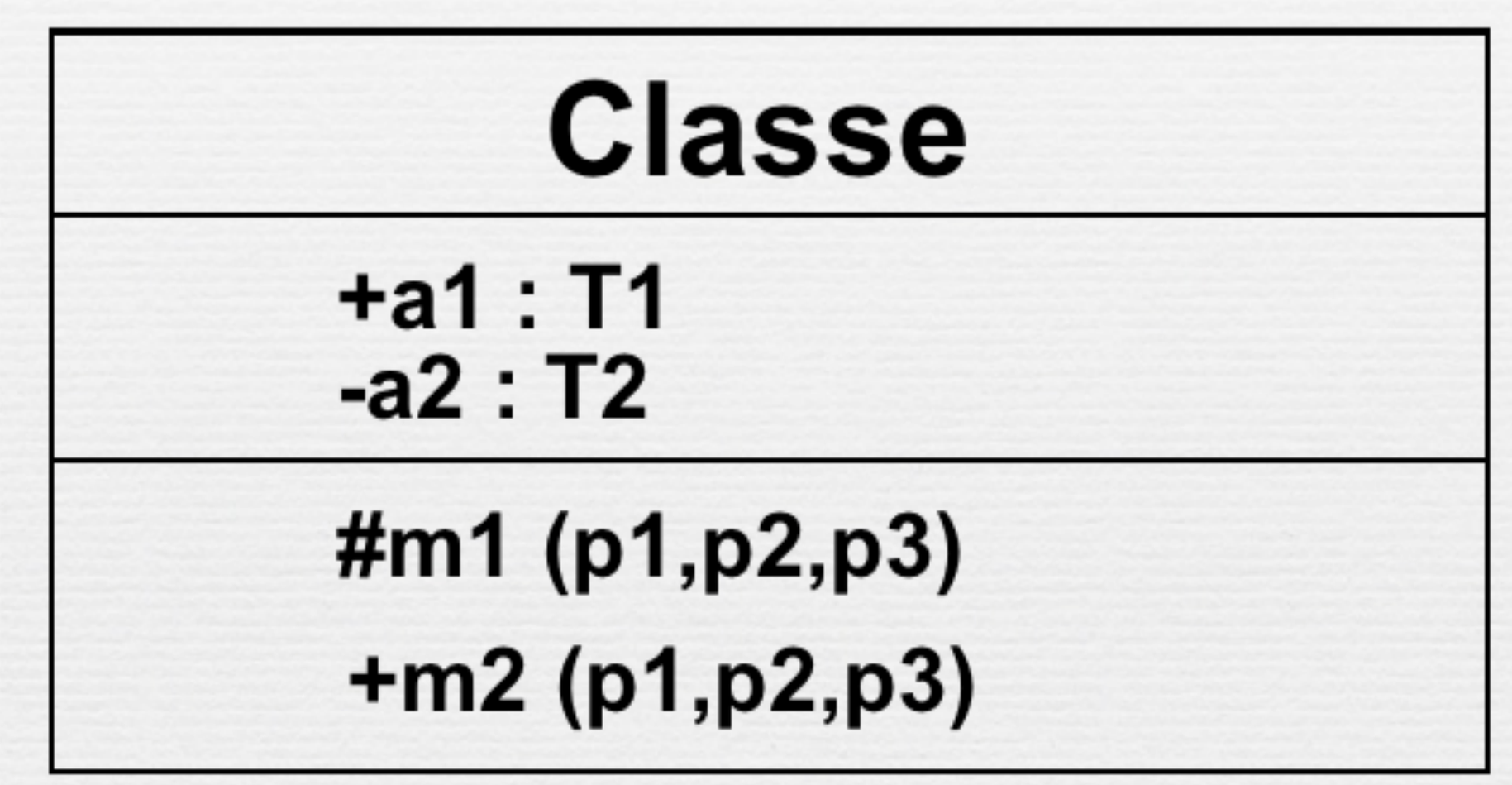
Visibilité



Visibilités des membres d'une classe



Visibilités des membres d'une classe : représentation



Pas de sens en analyse...mais en conception.

Visibilités des membres d'une classe

- 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

Métaphore :
=> 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)

Visibilités des membres d'une classe

- 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())

Visibilités des membres d'une classe

- Les opérations sont en général **accessibles** par toutes les classes. Elles sont alors qualifiées de :

- « **public** » : notation UML « + »

Visibilités des membres d'une classe

- 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 « # »

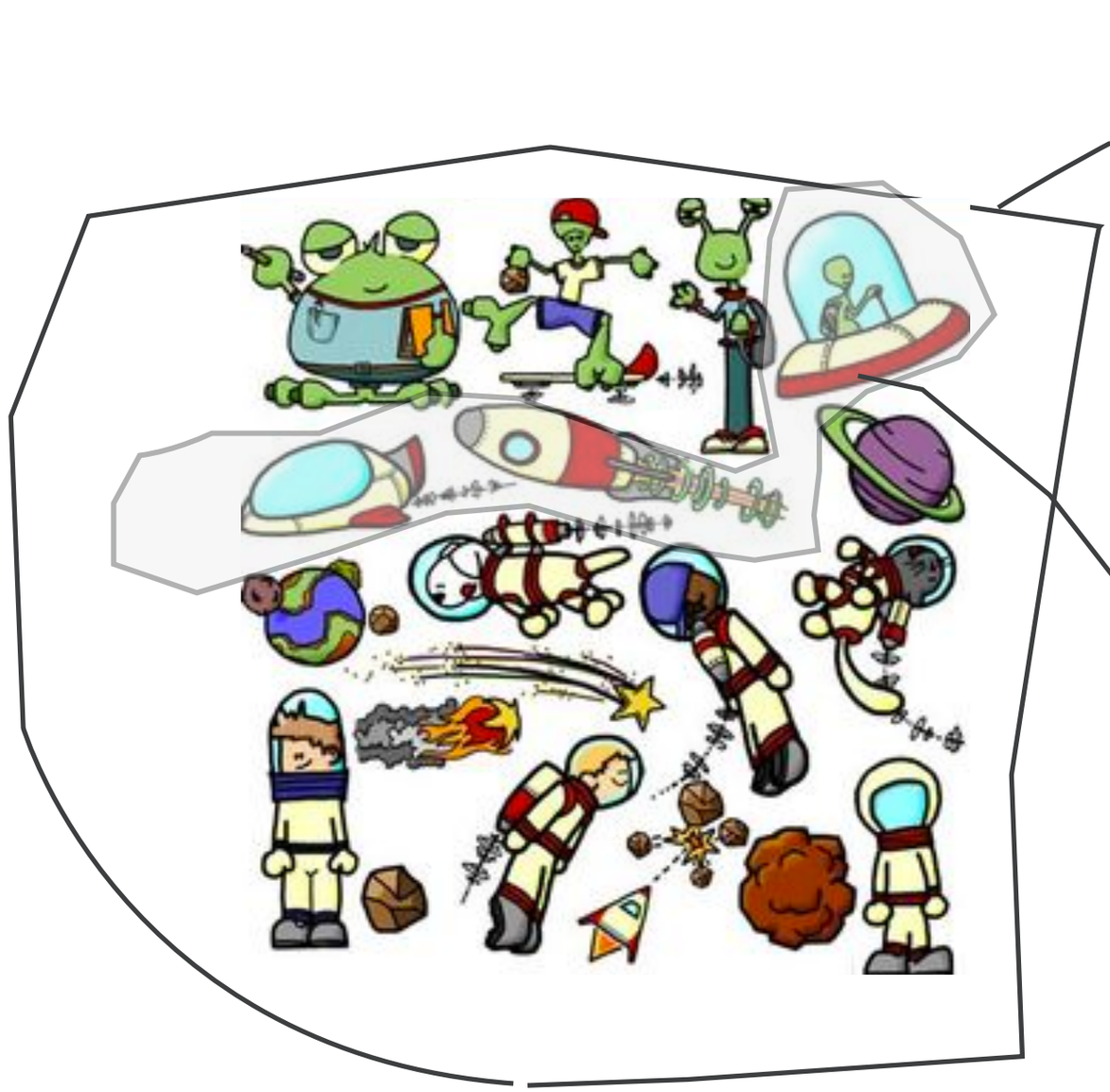
Visibilités des membres d'une classe

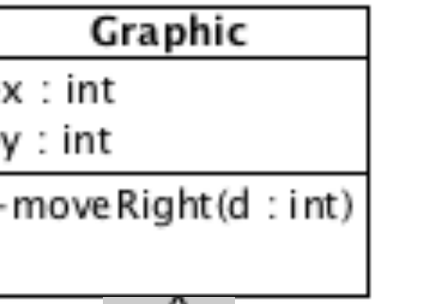
- Certaines opérations peuvent cependant
  - être privées (factorisation interne de traitements) et
  - certains attributs peuvent être publics (non souhaitable, cf. principe d’encapsulation)

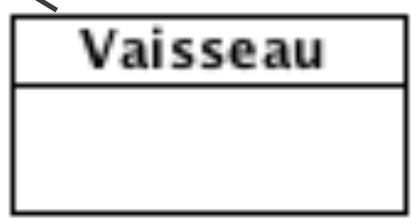
Des diagrammes de classes en conception

Généralisation/ Spécialisation

Généralisation : Signification ensembliste





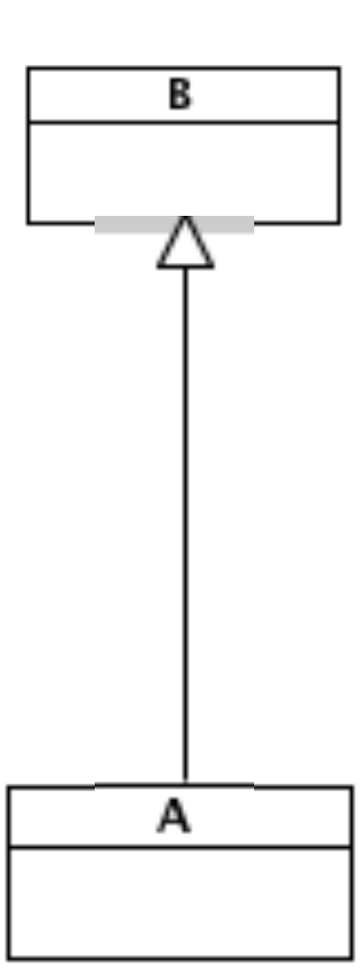


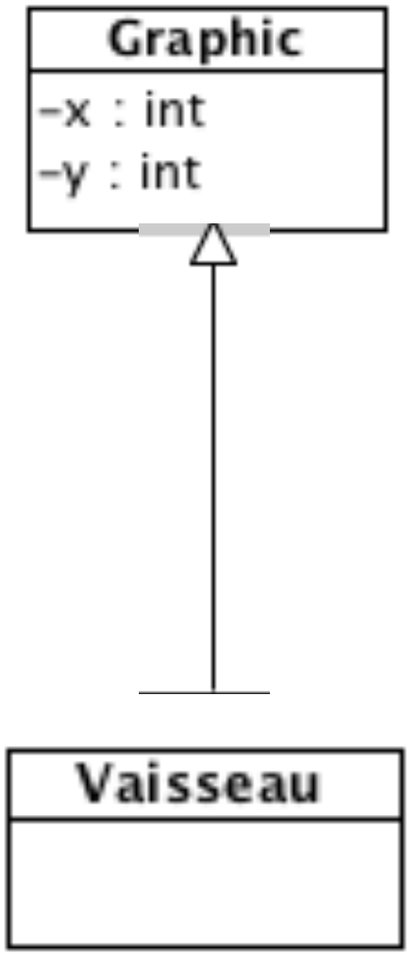

```
Graphic g  = new Vaisseau();
g.moveRight(10);
```

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

Généralisation

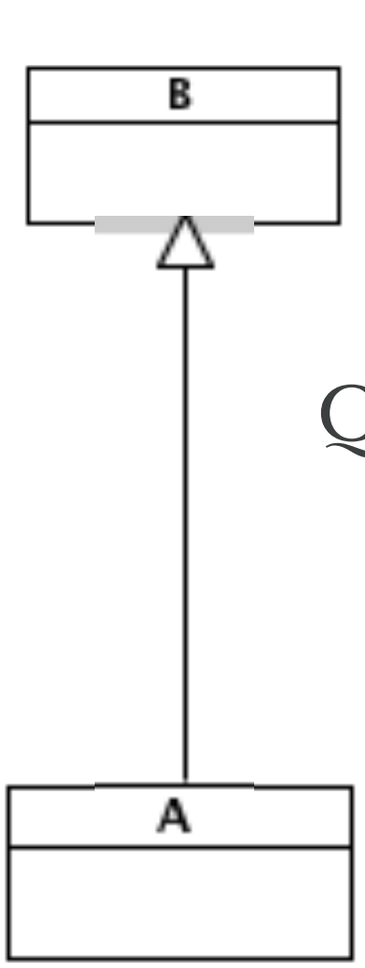
Quand A hérite de B,
les objets instances de A
possèdent les propriétés de B





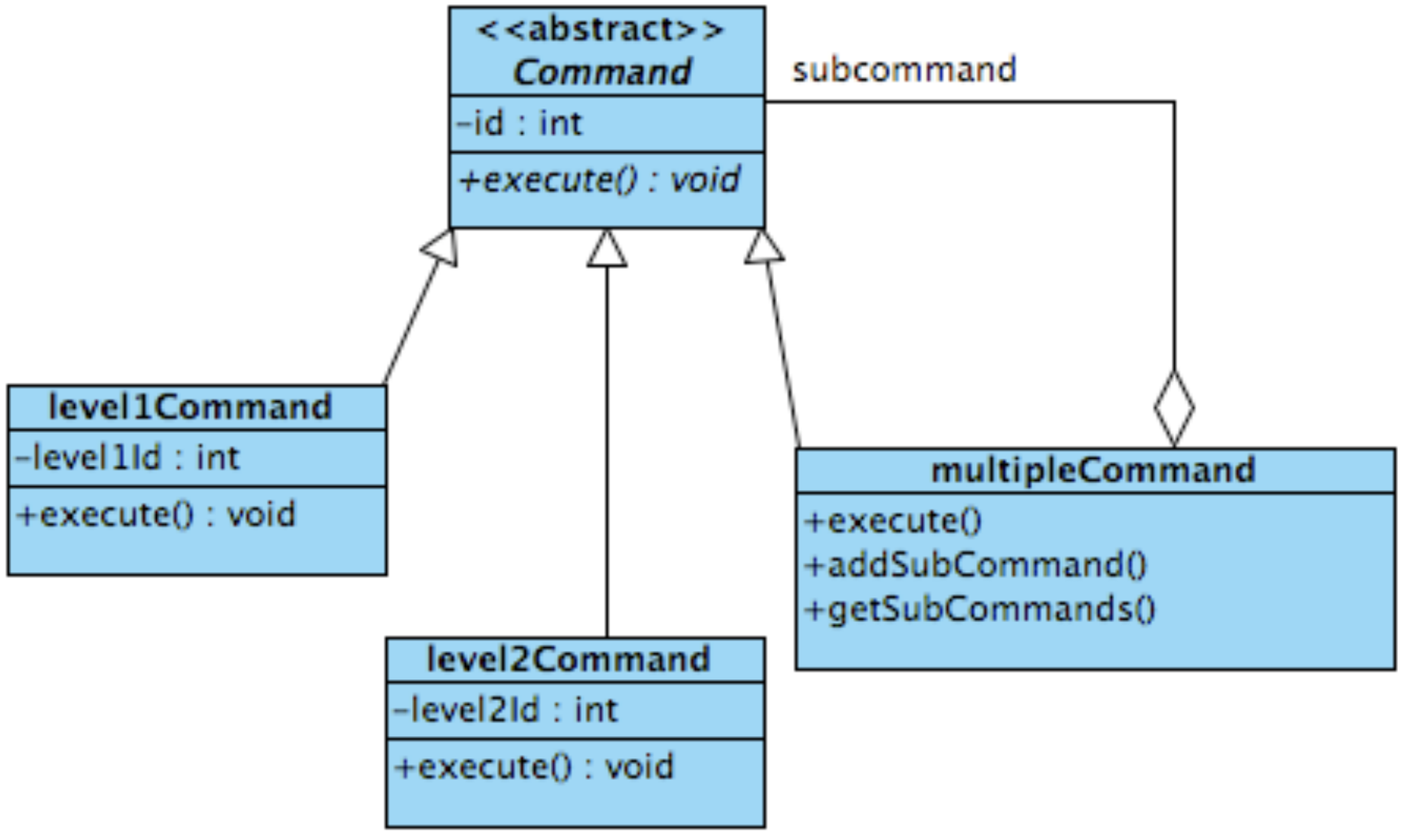
Généralisation

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

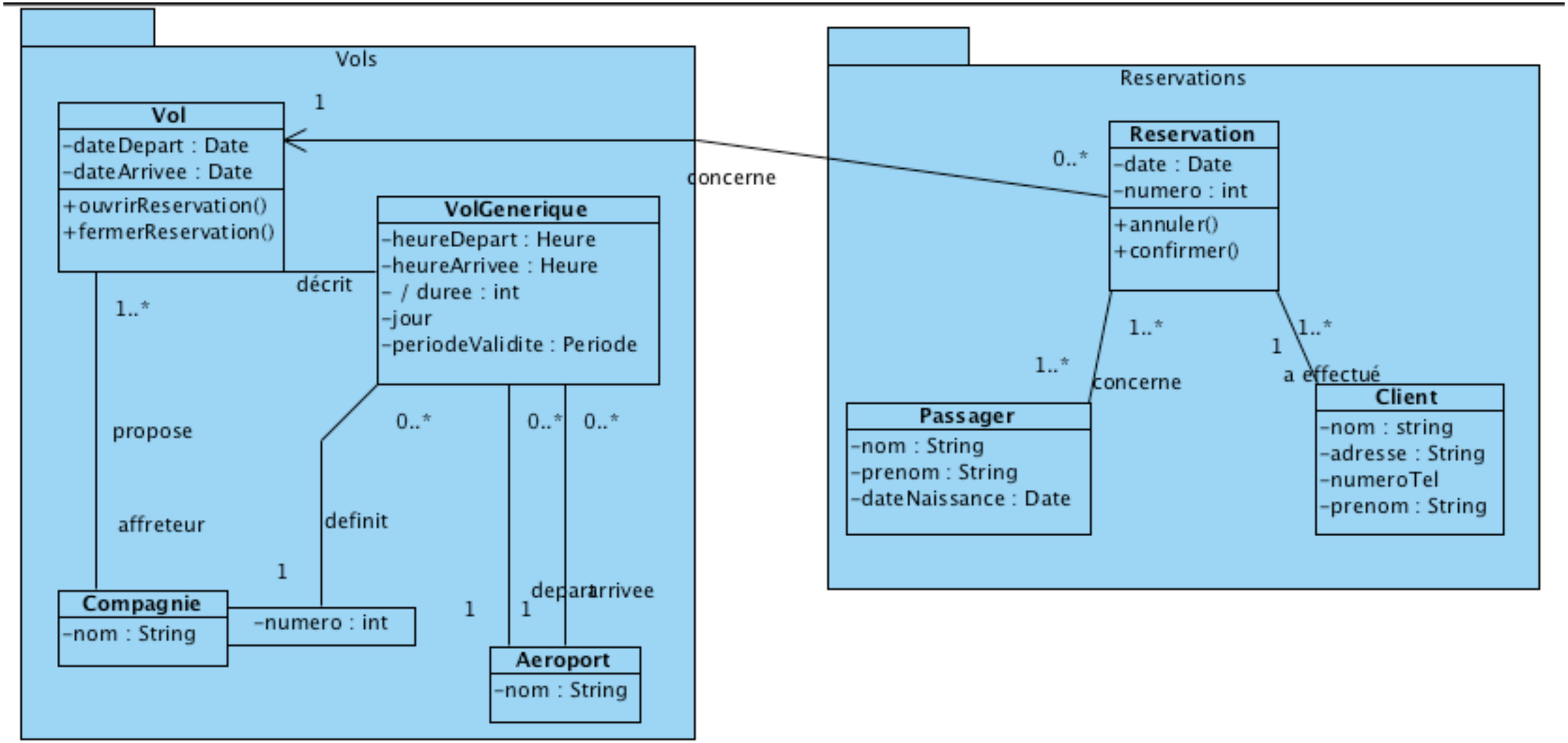


Quand A hérite de B, les objets instances de A
savent réaliser les méthodes définies par B

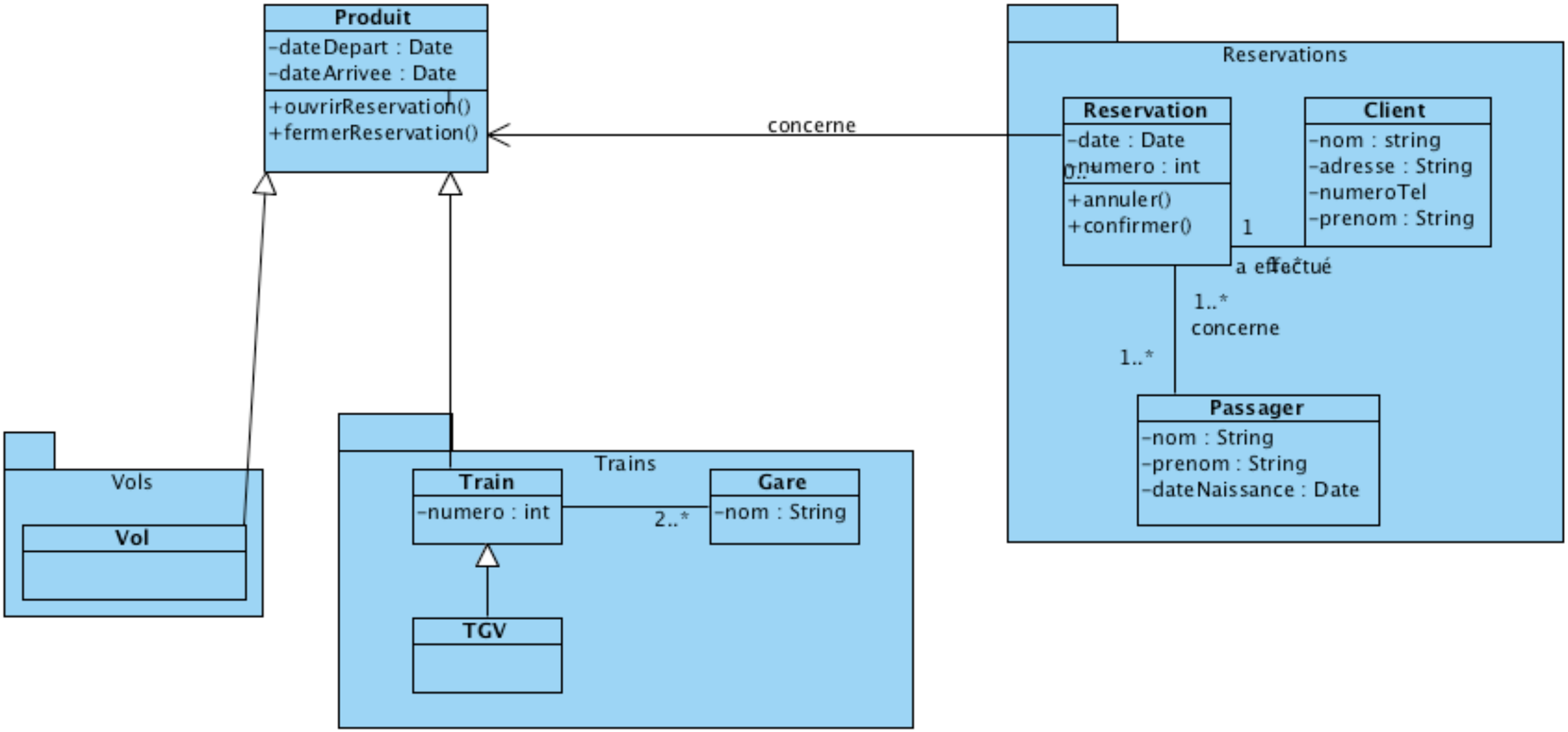
Généralisation : exemple d'usage
Pattern composite



La société qui prend en charge les réservations de vols, veut prendre en charge des réservations de train. **Que devez-vous modifier ?**

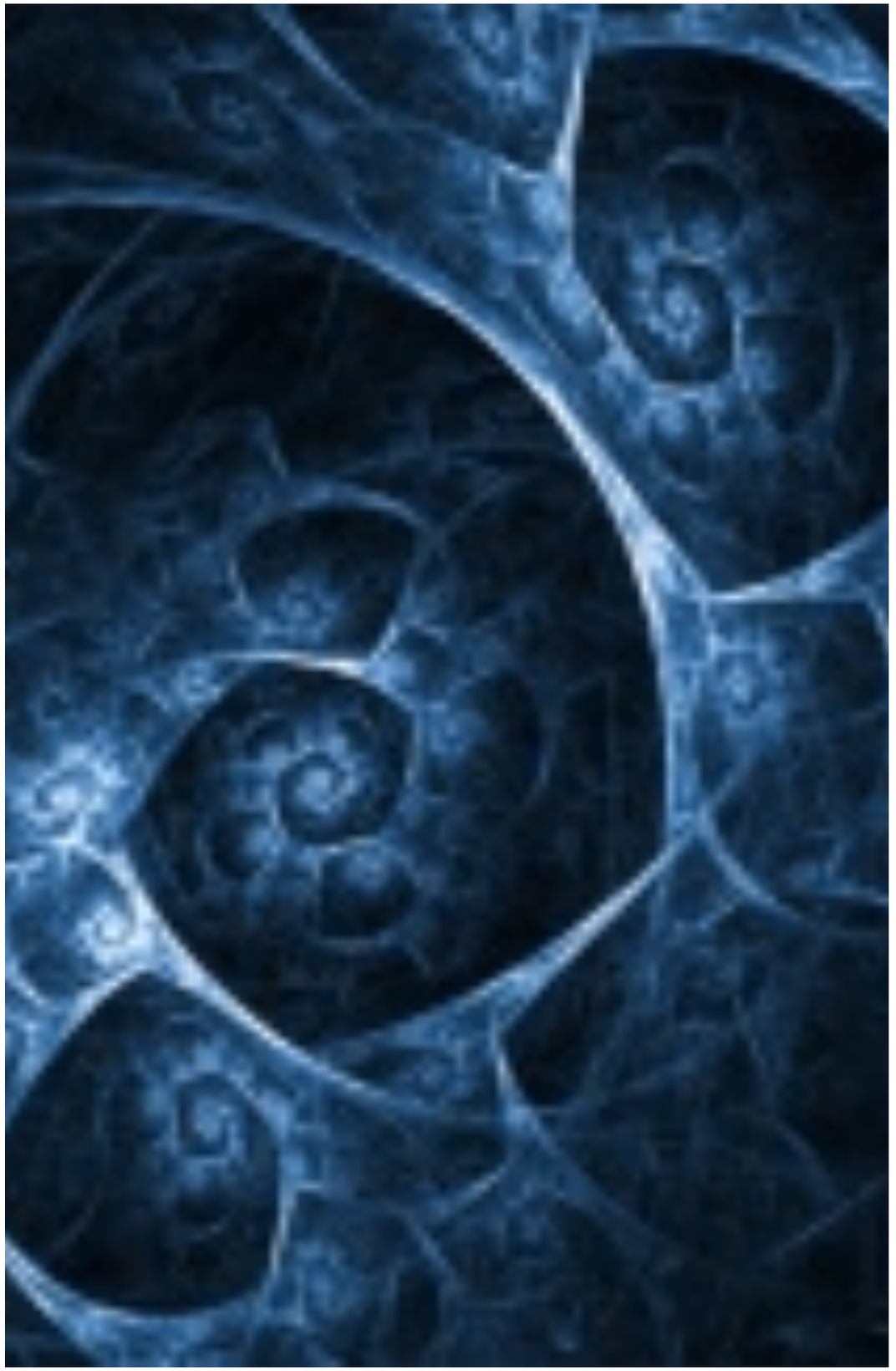


La société qui prend en charge les réservations de vols, veut prendre en charge des réservations de train. **Proposition : Généralisation**



Des diagrammes
de classes
en conception

Abstraction



Abstraite

Je dois exprimer que dans le jeu les véhicules se déplacent en fonction des mouvements de la souris…

mais dans le jeu il n'y a pas des véhicules, mais des avions ou des voitures….une voiture ou un avion ne se déplacent pas de la même manière…

Classe Abstraite

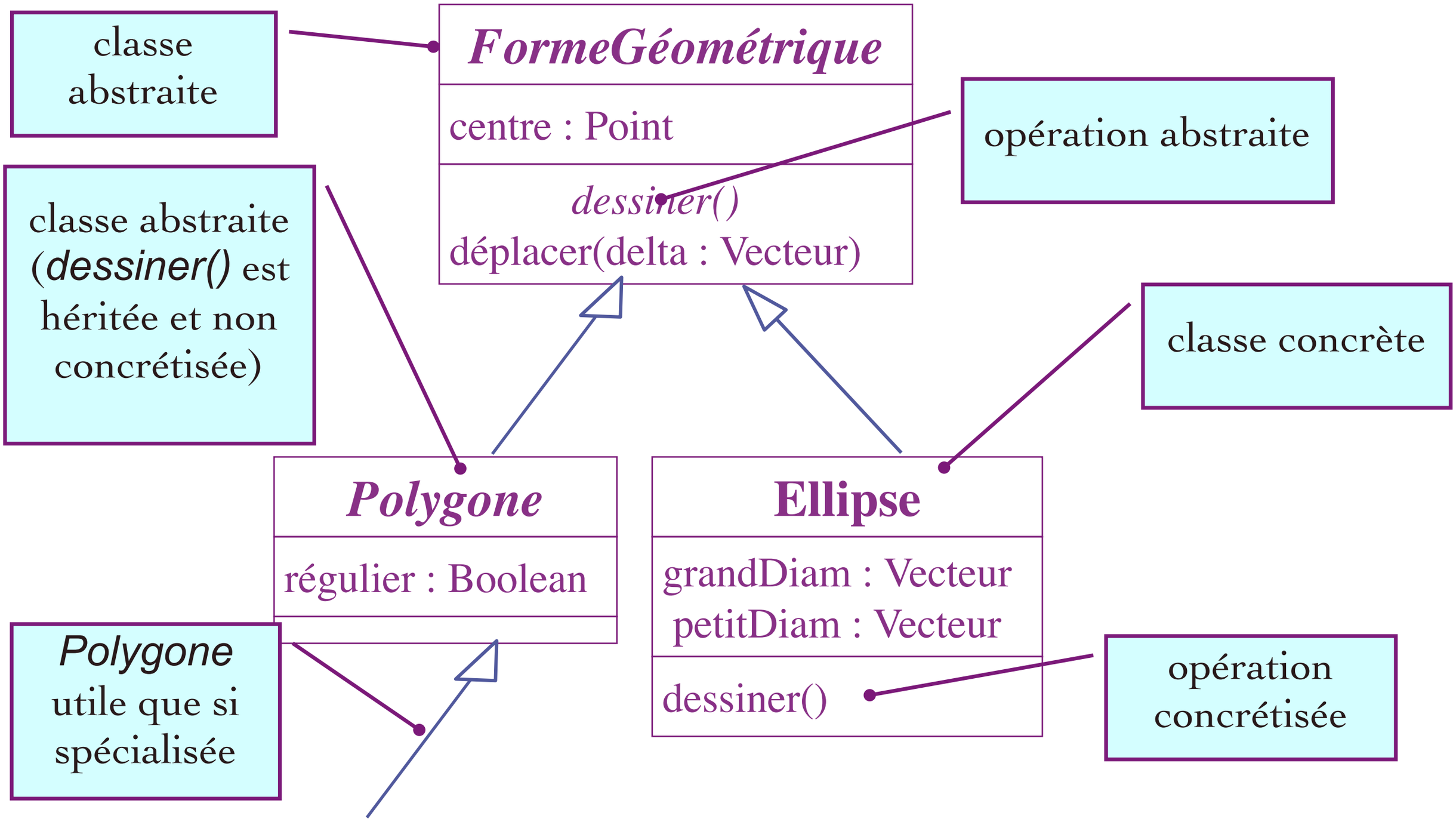
- Une **classe abstraite** est une classe non instanciable, c'est à dire qu'elle n'admet pas d'instances directes.

- Une classe abstraite est 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.

Classe abstraite



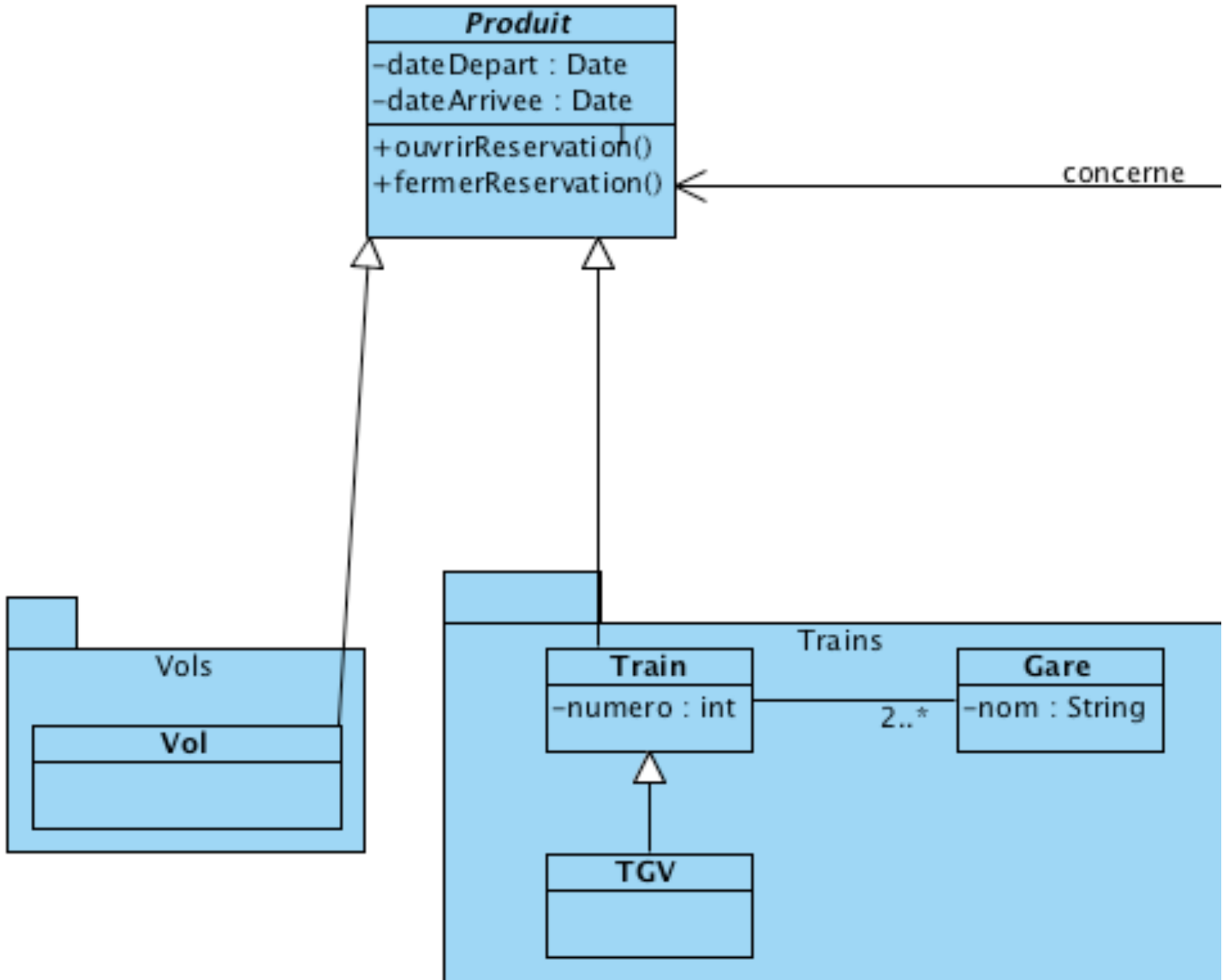
Opération Abstraite

- 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.

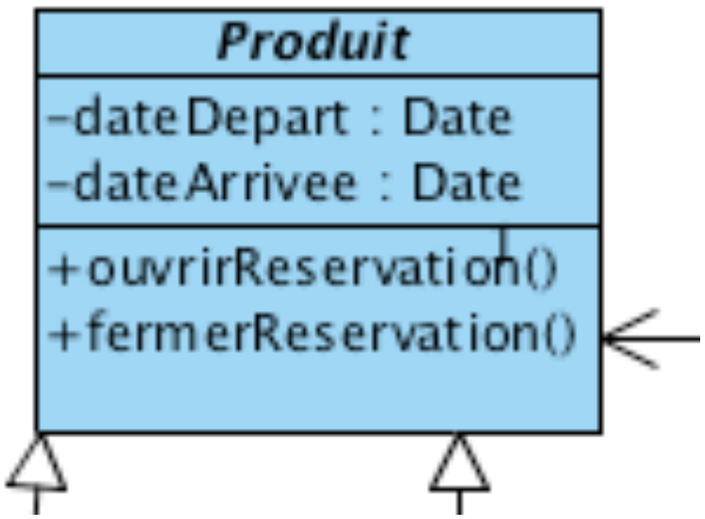
- Les opérations abstraites sont particulièrement utiles pour mettre en œuvre le *polymorphisme*.

- Toute classe concrète sous-classe d'une classe abstraite doit “concrétiser” toutes les opérations abstraites de cette dernière.

La société qui prend en charge les réservations de vols, veut prendre en charge des réservations de train. **Proposition : Généralisation & abstraction**



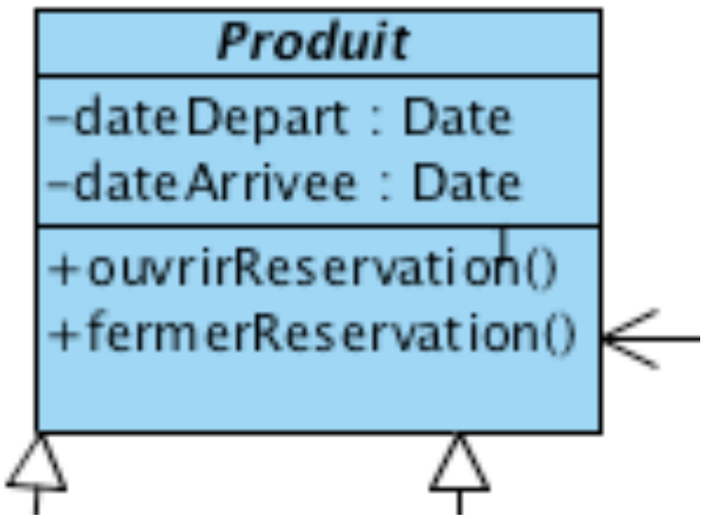
Représentation de classes abstraites



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

```java
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;
}

public void ouvrirReservation() {
    open = true;
}

public void fermerReservation() {
    open = false;
}
```

Représentation de opérations abstraites

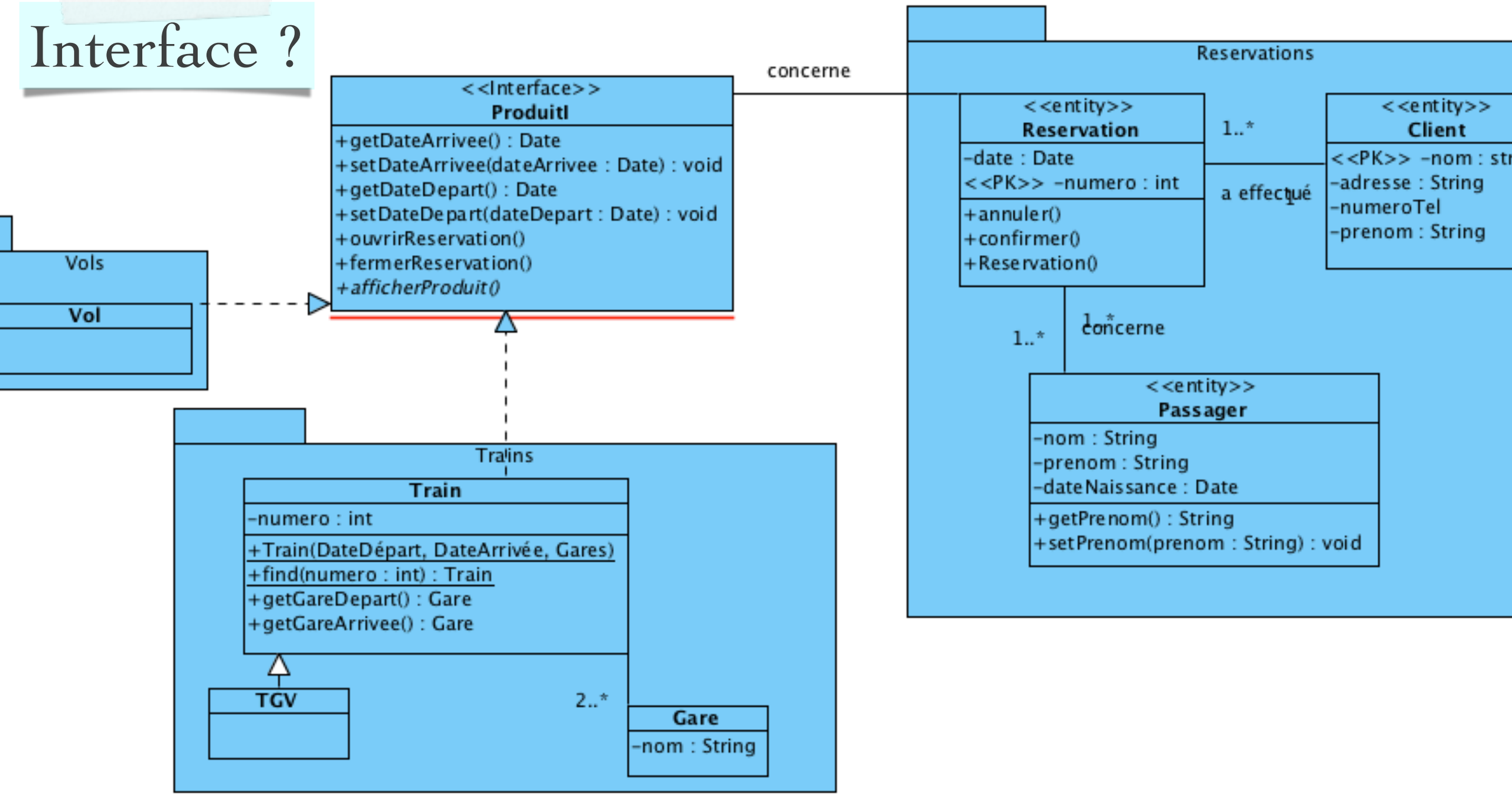


Attention des choix de mises en oeuvre non explicités au niveau du modèle apparaissent dans ce code.

```java
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();
```

La société qui prend en charge les réservations de vols, veut prendre en charge des réservations de train. **Proposition : Généralisation & abstraction**

Interface ?

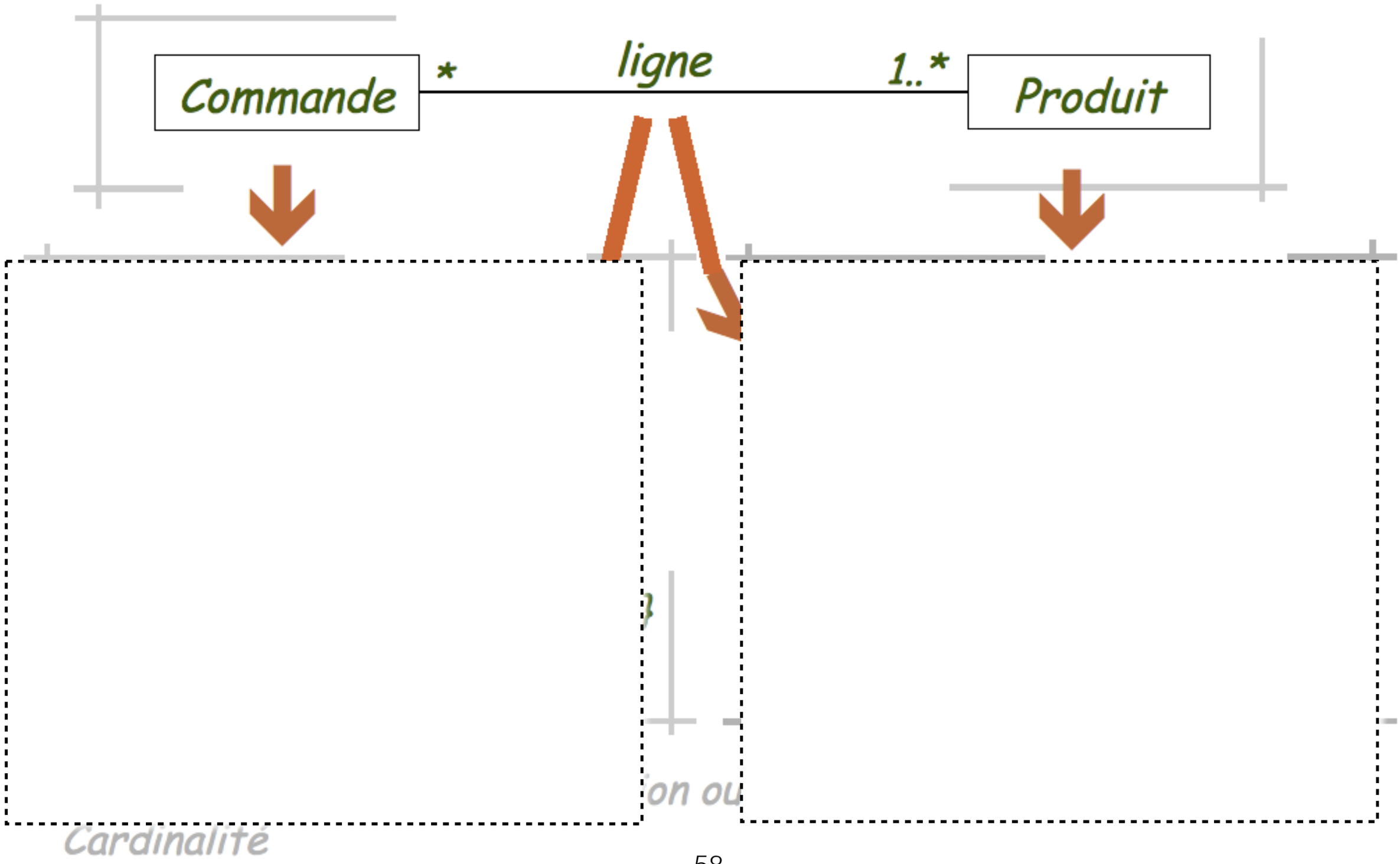


Q6 à Q9

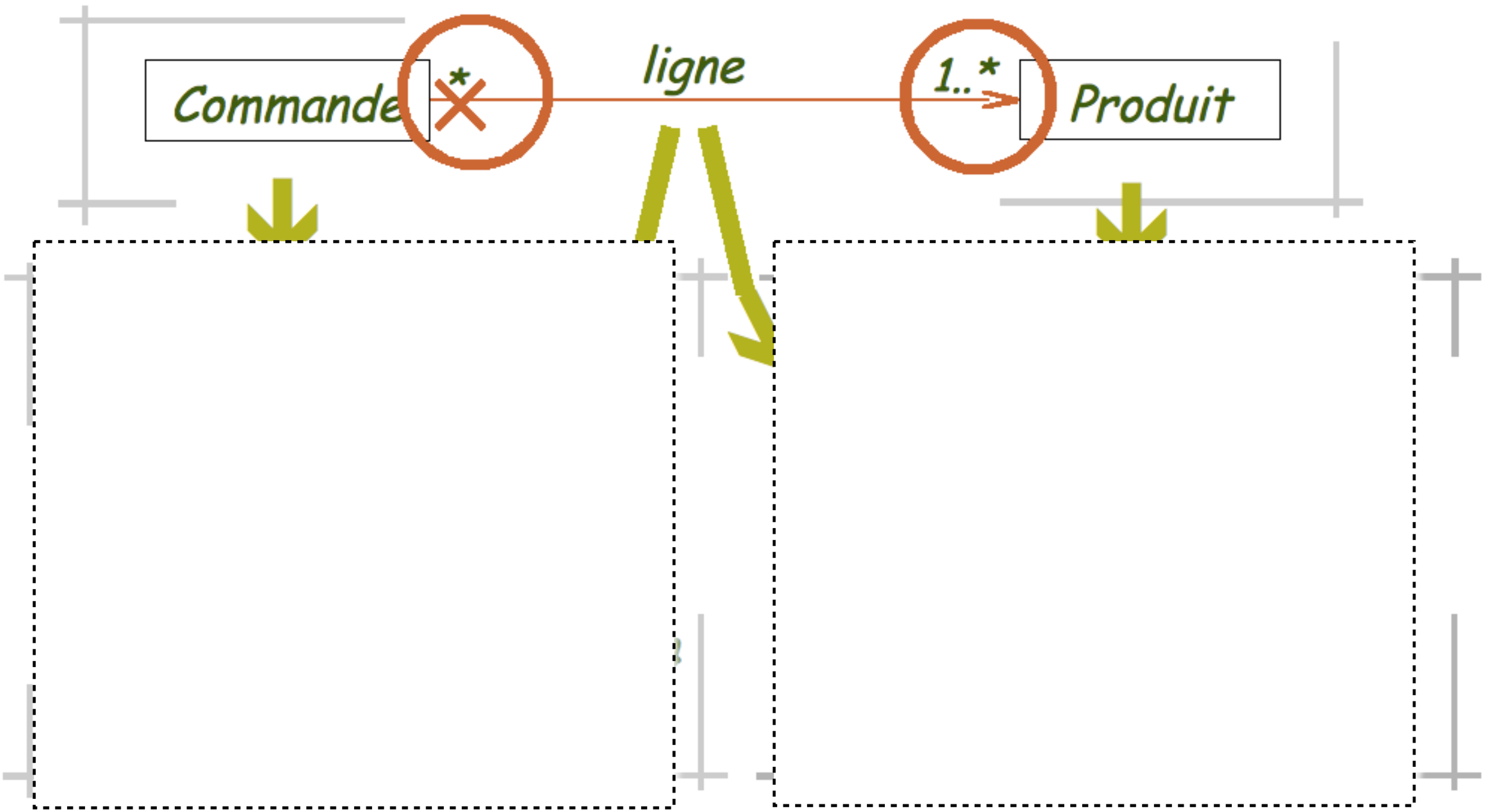
Des diagrammes
de classes
en conception

Transformation des associations

Association...



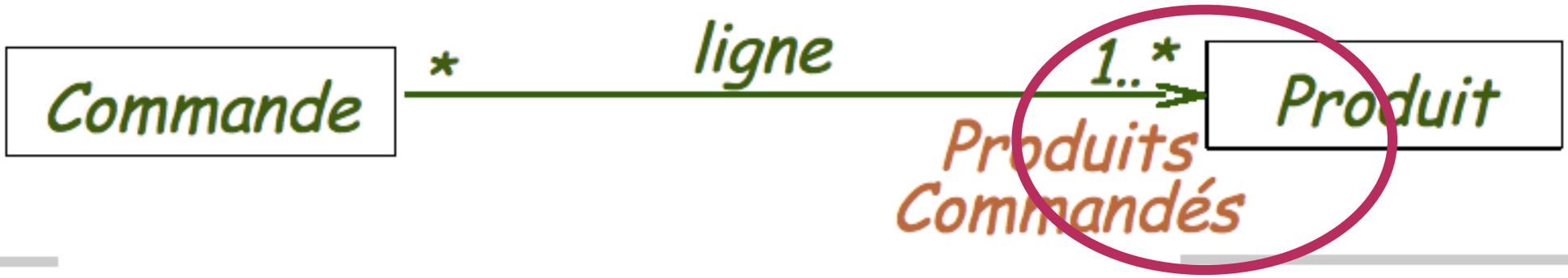
Association...



Association...


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

Association…

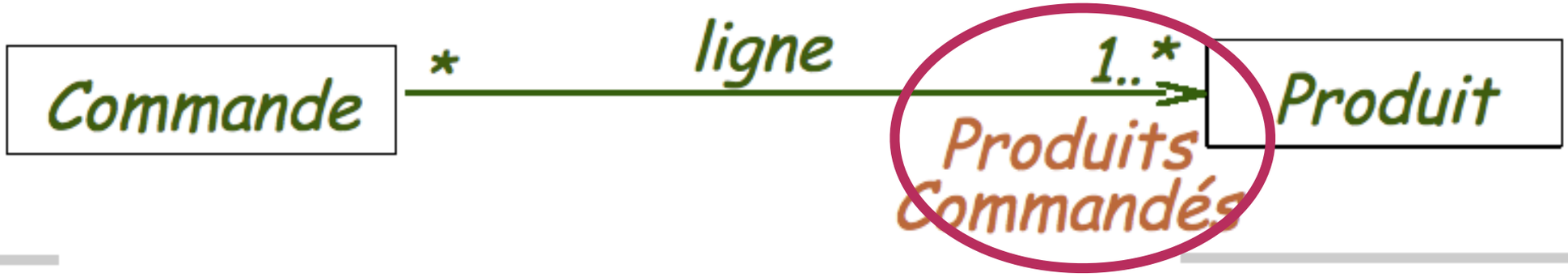



```
public class Commande {
        private Produit[] ProduitsCommandés;
        public void
```

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

lignes-> produitsCommandes… pas de majuscules…pas d'accent… en anglais ???

Association...

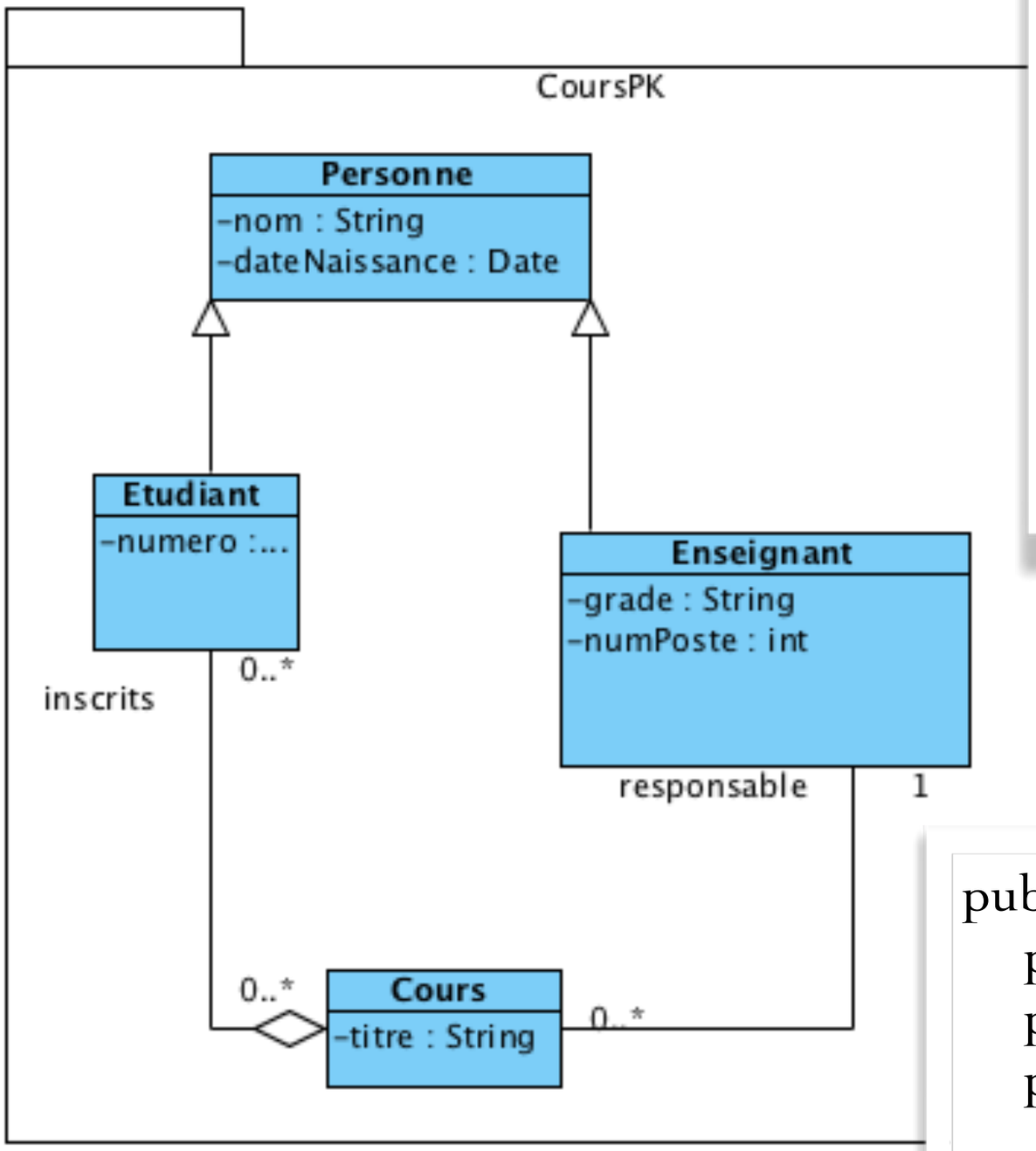



```
public Commande (Produit[] c) throws Exception {
  if (c.length != 0)
    lignes = c;
  else
    throw new Exception("Un produit au moins est requis");
}
```

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

lignes-> produitsCommandes…

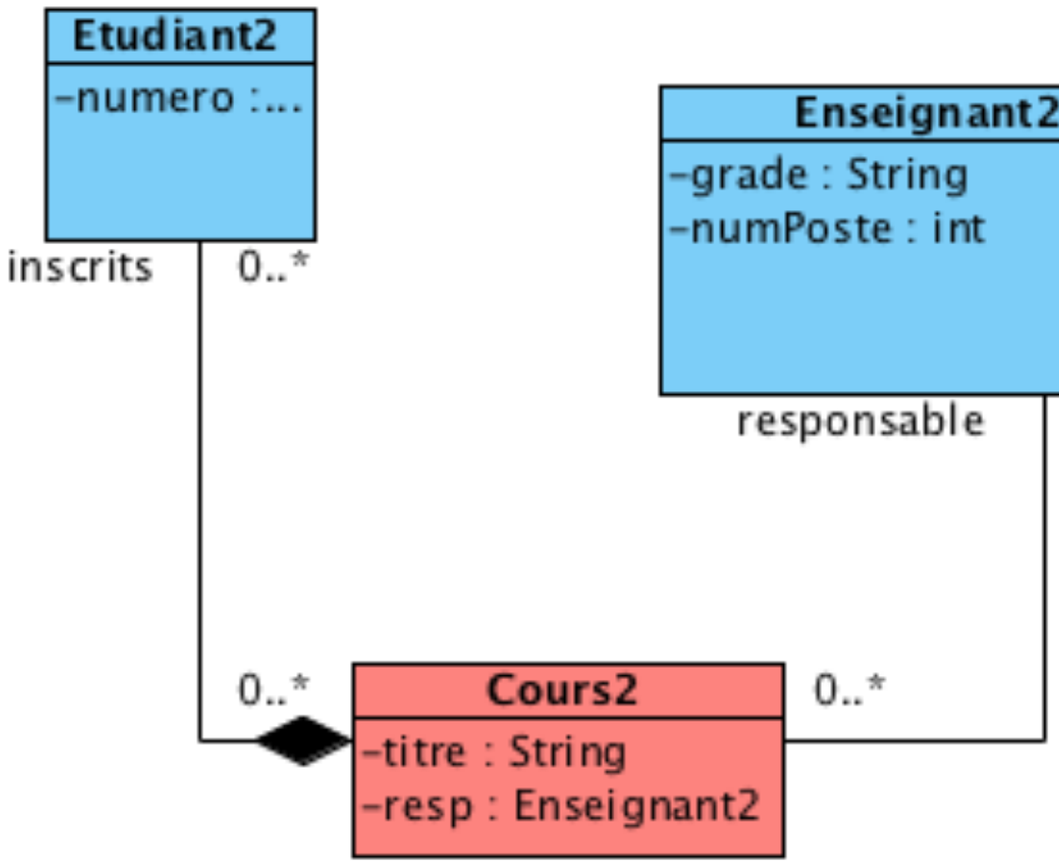
Association et générations de code




```
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>();
}
```

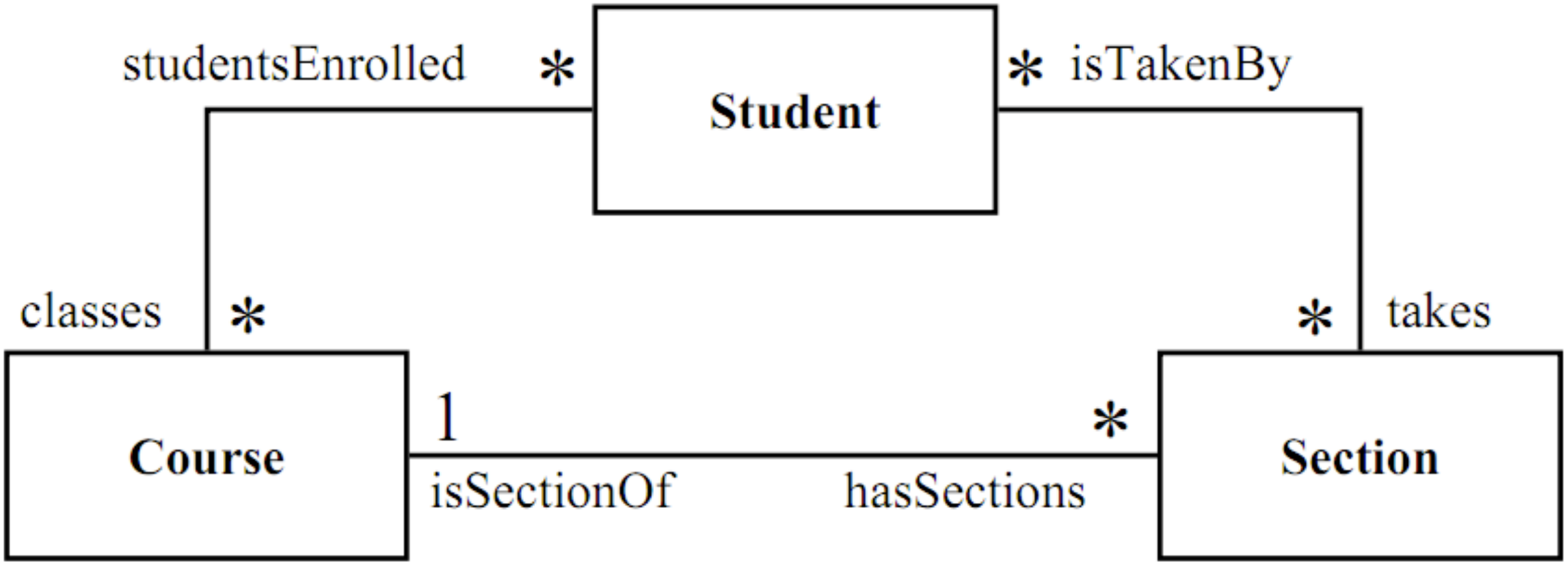
Association et générations de code



Des problèmes ?

```
public class Cours2 {
    private String titre;
    private Enseignant2 resp;
    public ArrayList<Etudiant2> inscrits = new ArrayList<Etudiant2>();
    public Enseignant2 responsable;
}
```

Gestion des associations bidirectionnelles

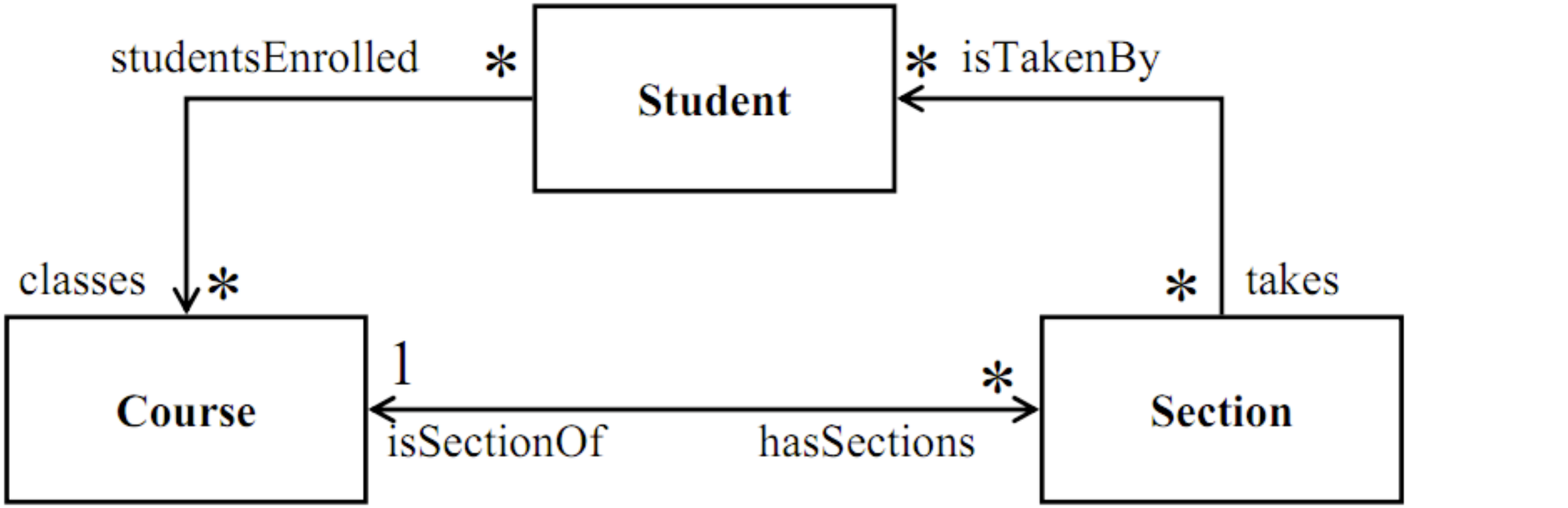


Bi-directionnalité :

Si j'ajoute un étudiant dans un cours, comment m'assurer que le cours fait bien référence à cet étudiant?

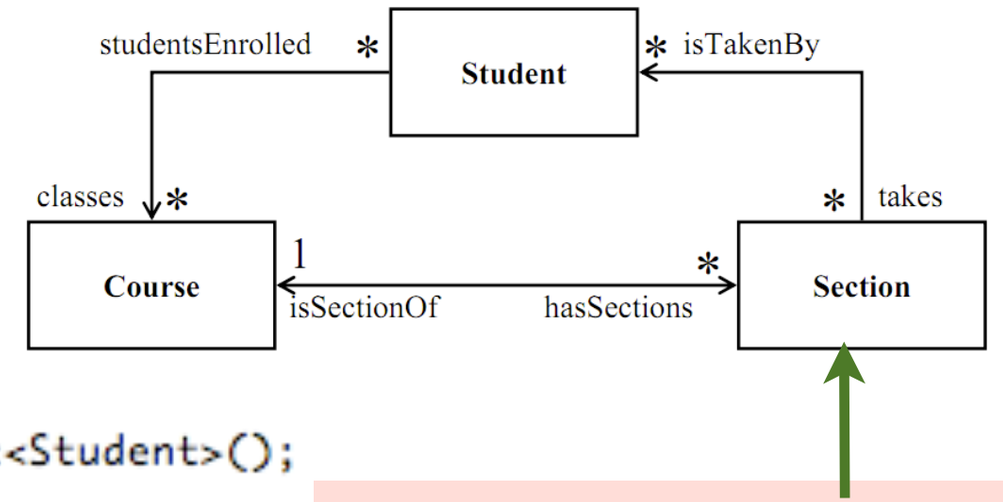
etc.

Gestion des associations : un choix



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

Implémentation



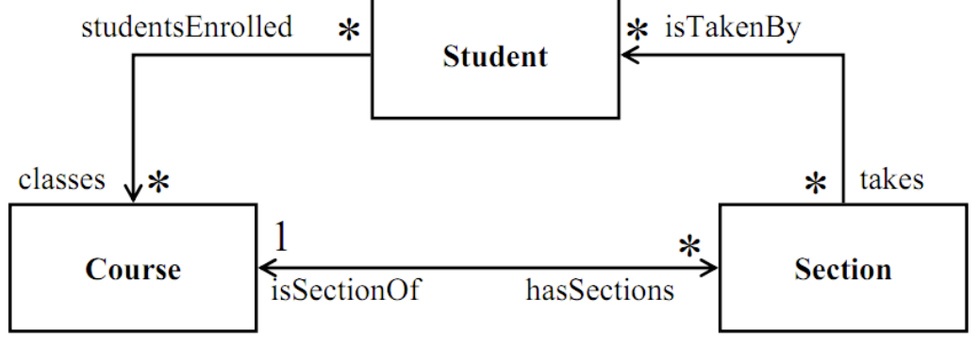
A la construction

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

Prise de responsabilités(Course)

Prise de responsabilités (Student)

Implémentation




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

        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 !

Association et générations de code

- 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

Des diagrammes
de classes
en conception

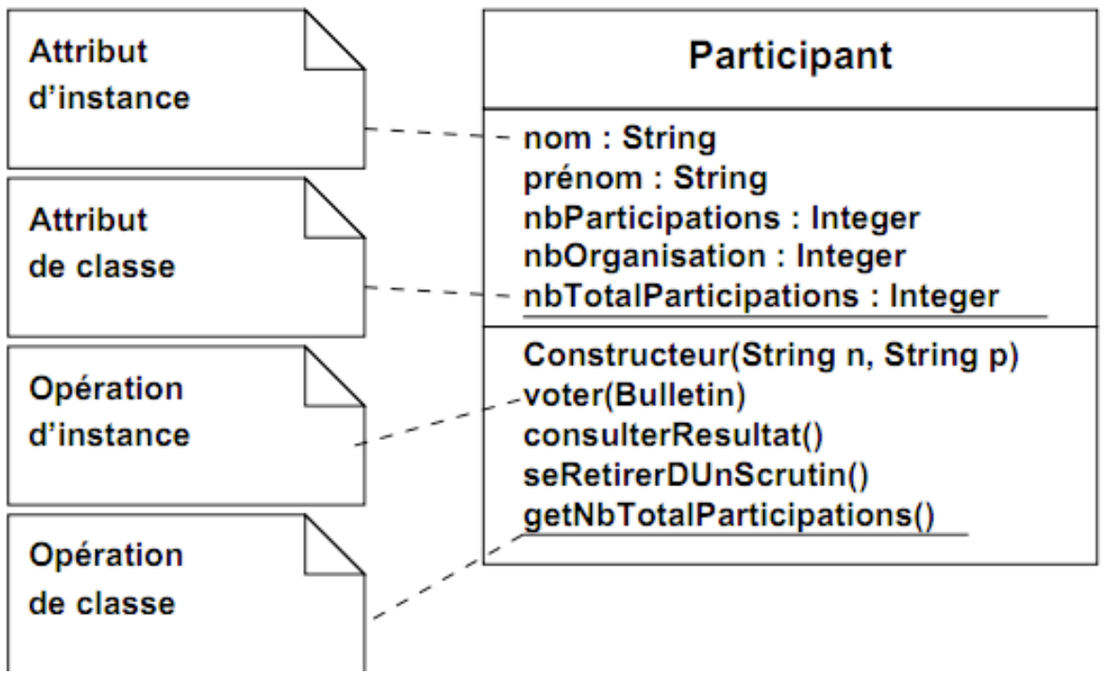
Attributs &
Méthodes de
classes

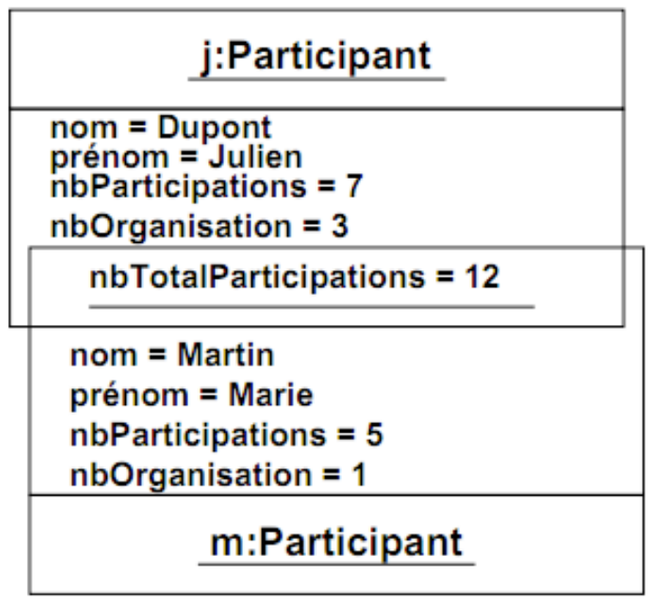
...

ou pas

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 de tous les participants.



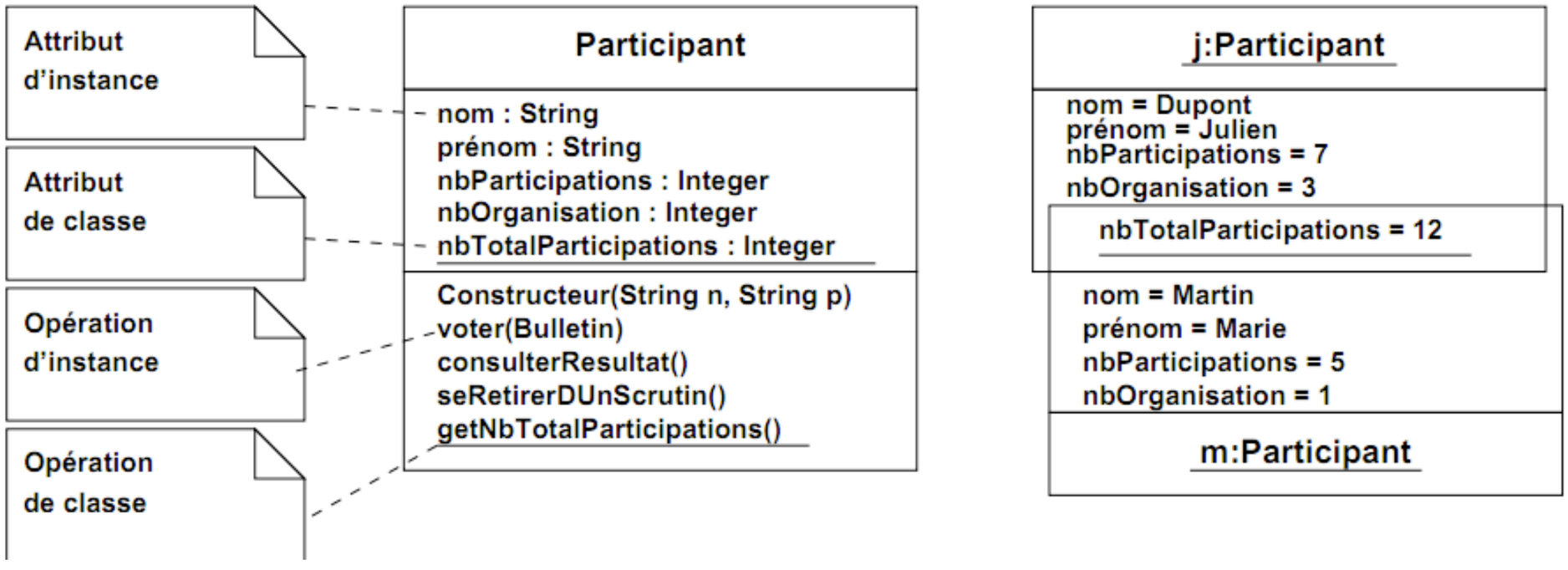


Attributs et opérations de classe

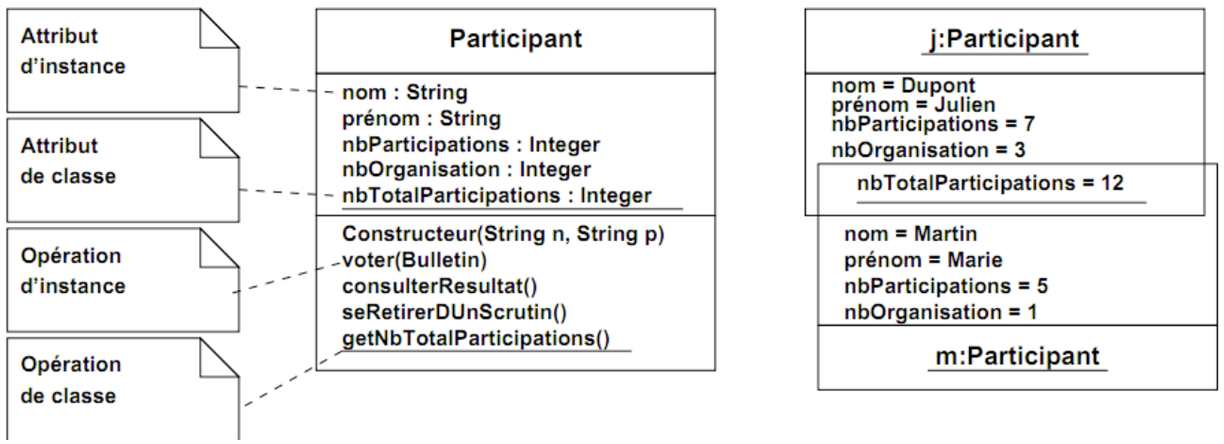
Le nombre total de participations est une caractéristique des Participants (classe), pas d'un seul participant.

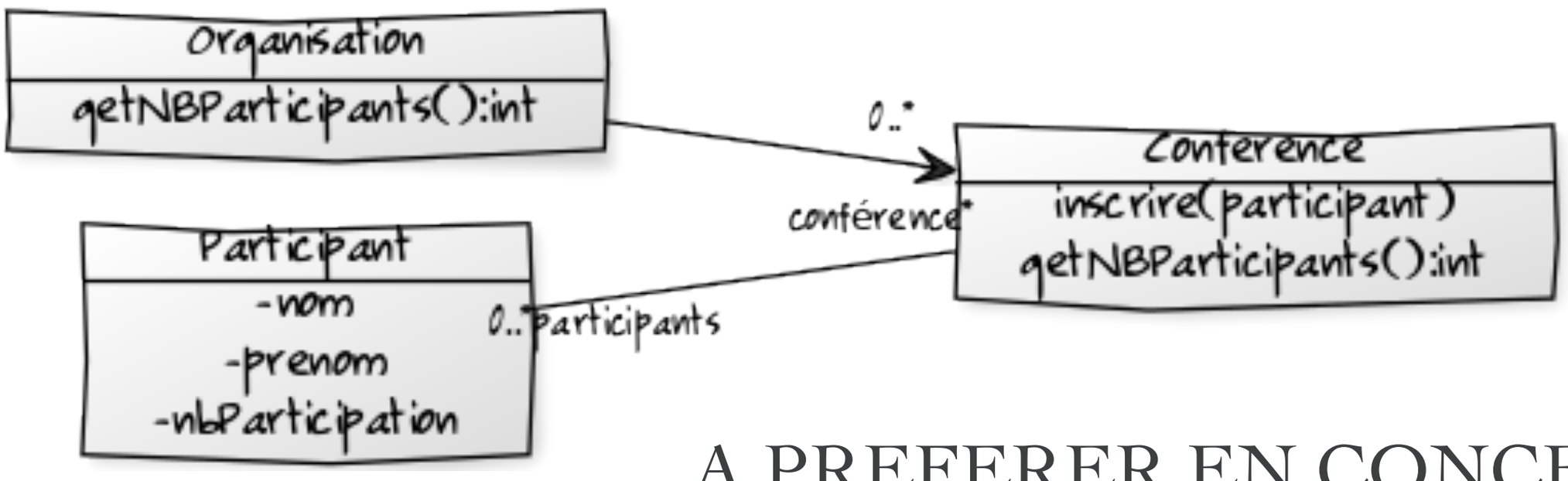
L'opération **getNbTotalParticipations()** utilise la valeur de l'attribut **nbTotalParticipations** connue par la classe

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

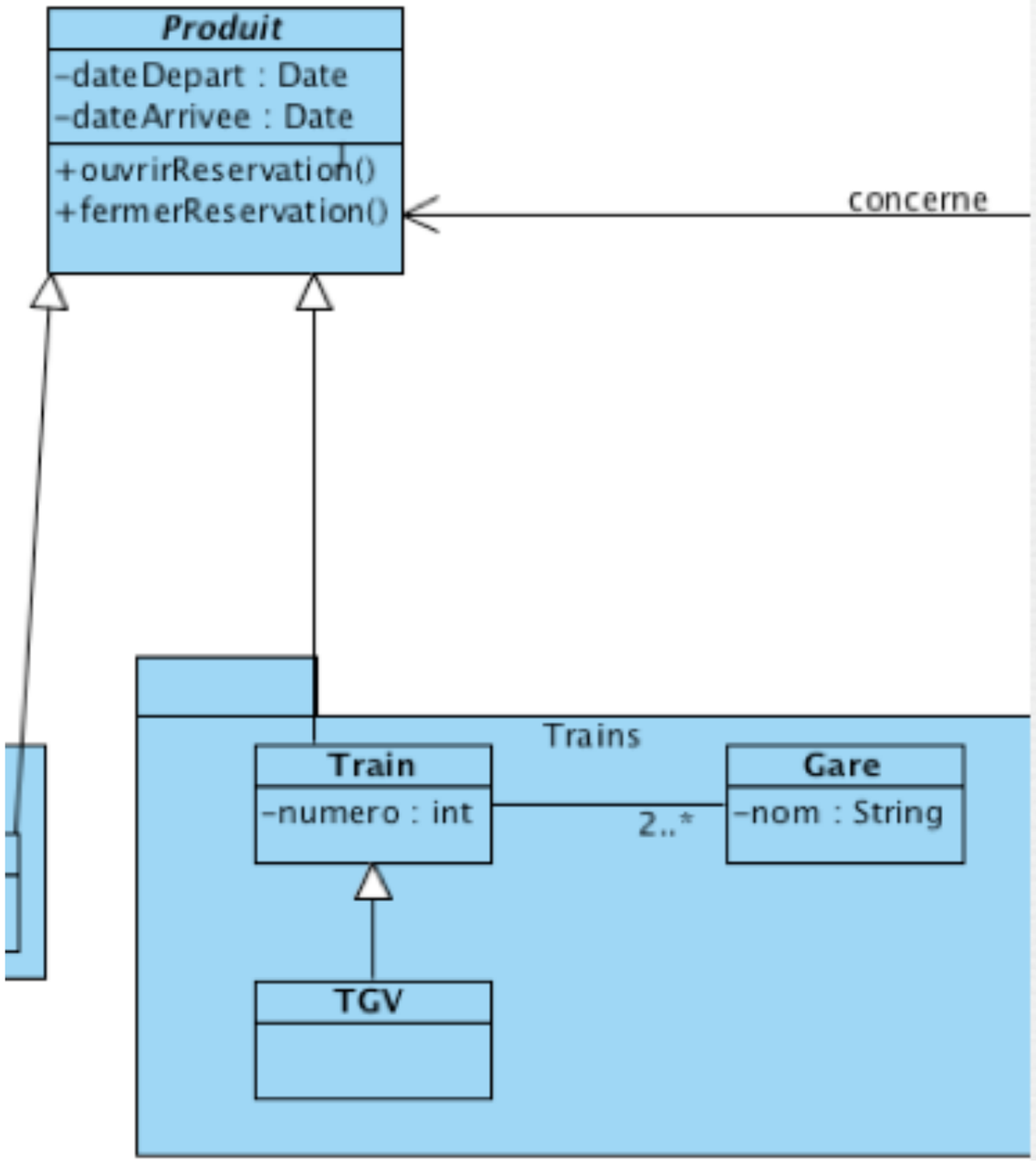
Person
- numberOfPeople : int - name : string
+ createPerson(name : string) : Person + getName() : string + getNumberOfPeople() : int - Person(name : string)

```
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



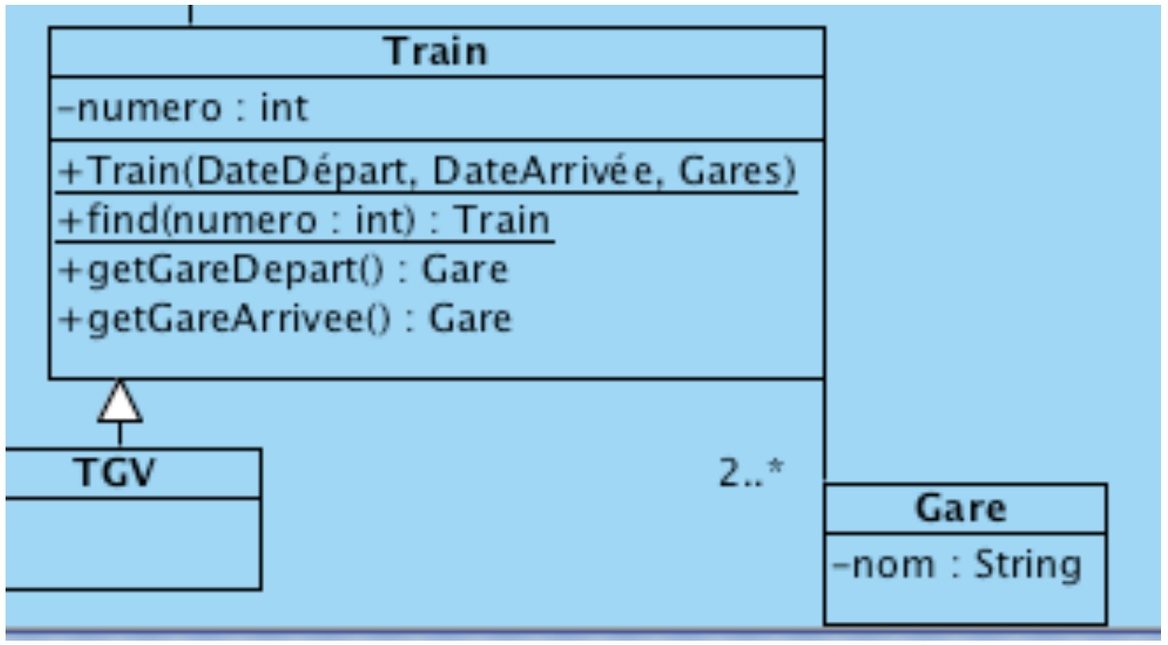
Dans la classe Produit

```
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){..

 //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

 //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);
}

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;

 //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

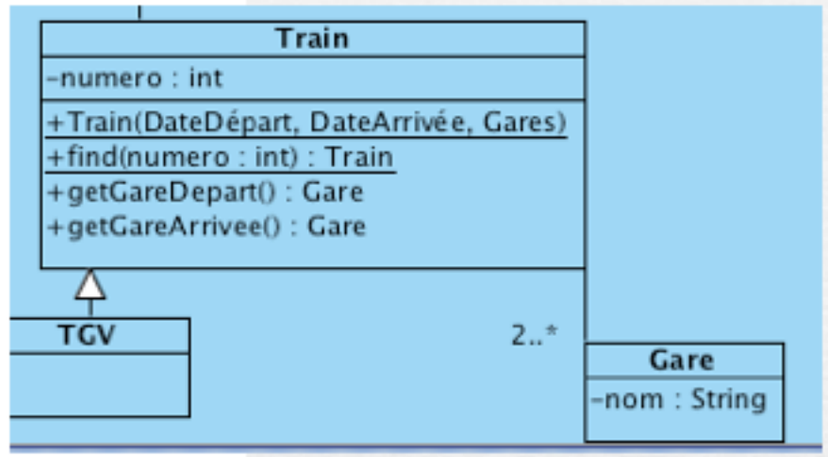
```
public class TestTrains {

    /**
     * @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*

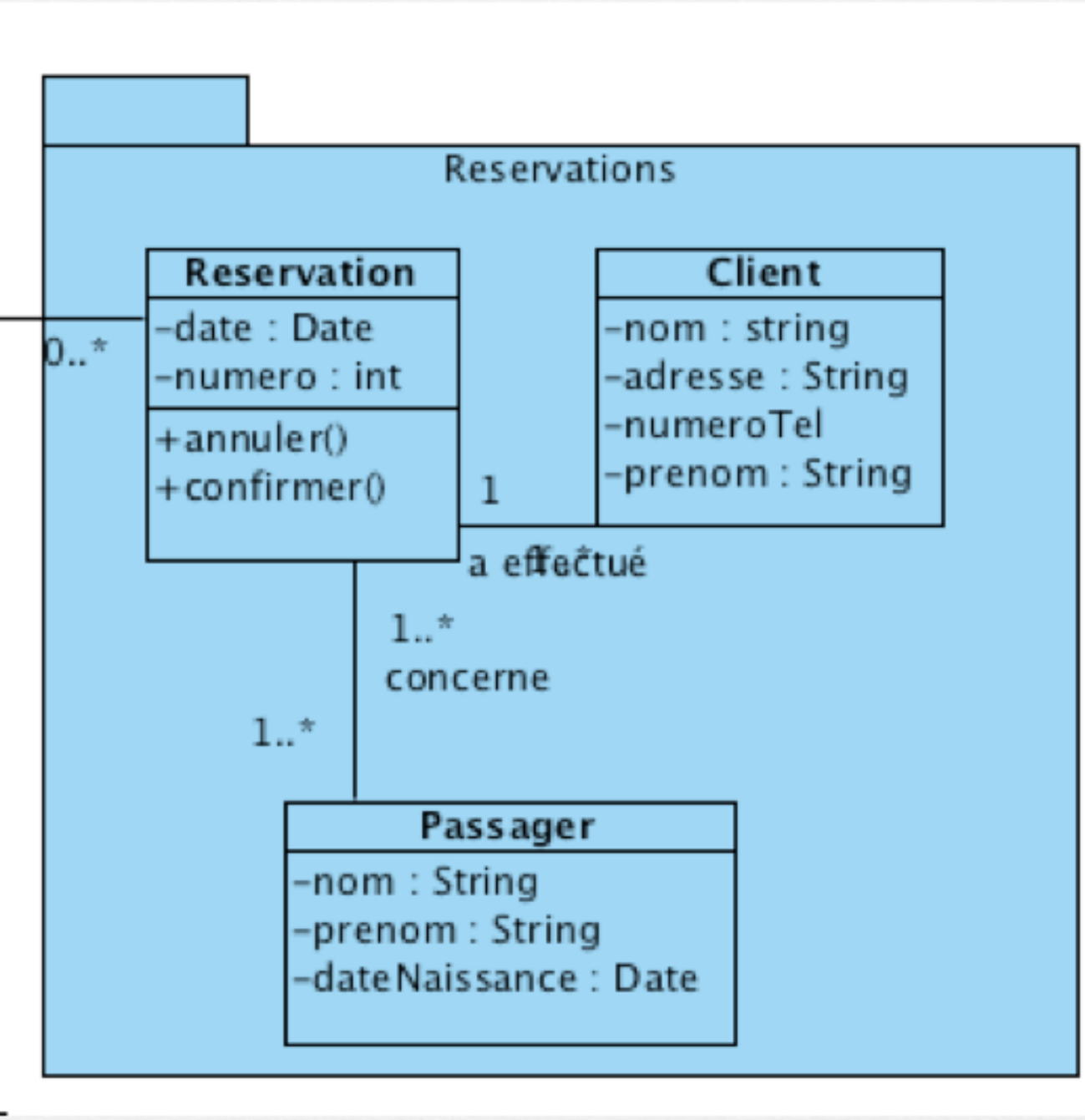
```java
package trainPK;

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

import produitPK.Produit;

public class Train extends Produit{
int numero;
Gare[] parcours;
 //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();
    }
```

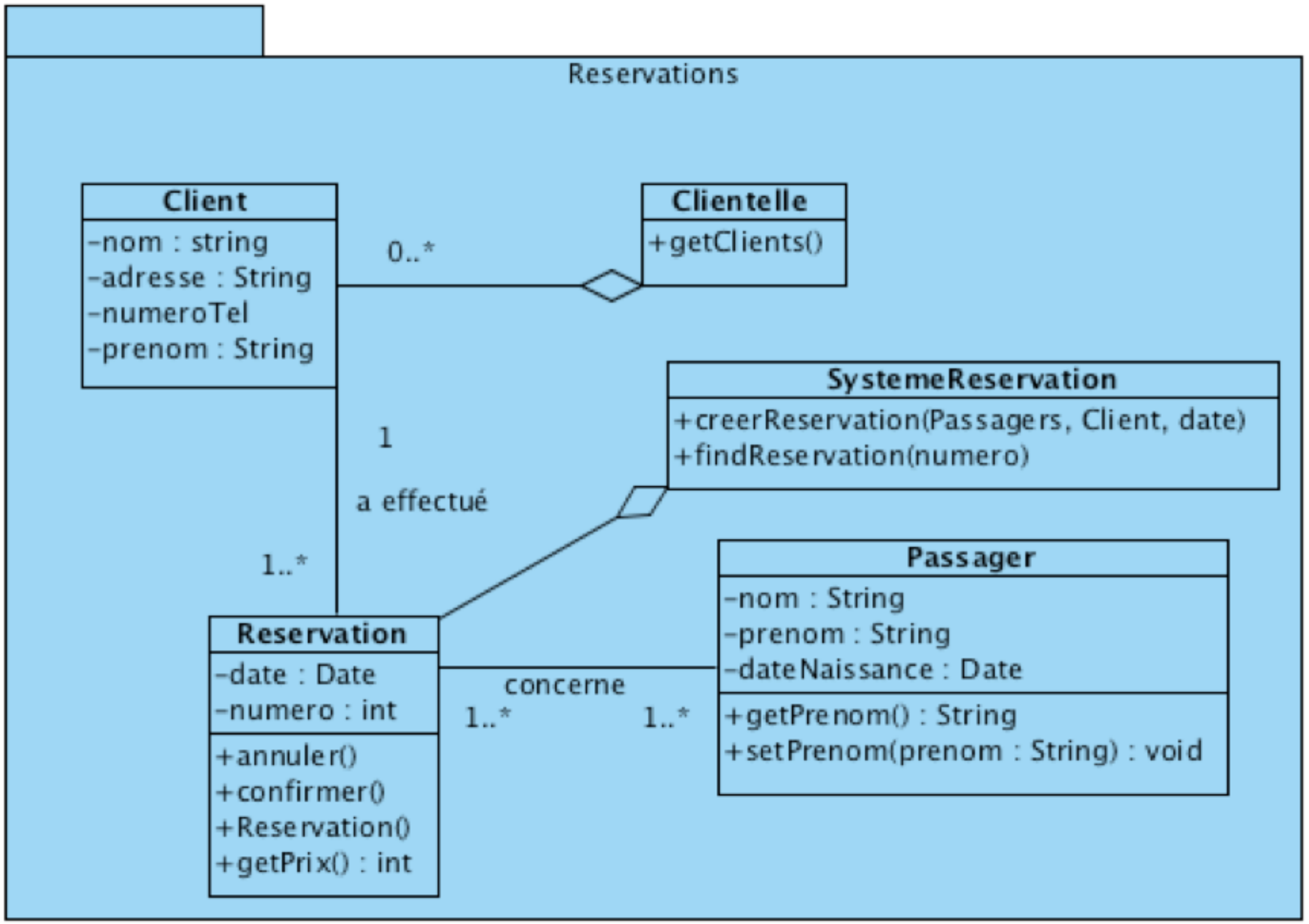


Opérations du niveau de la classe : *Static*



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

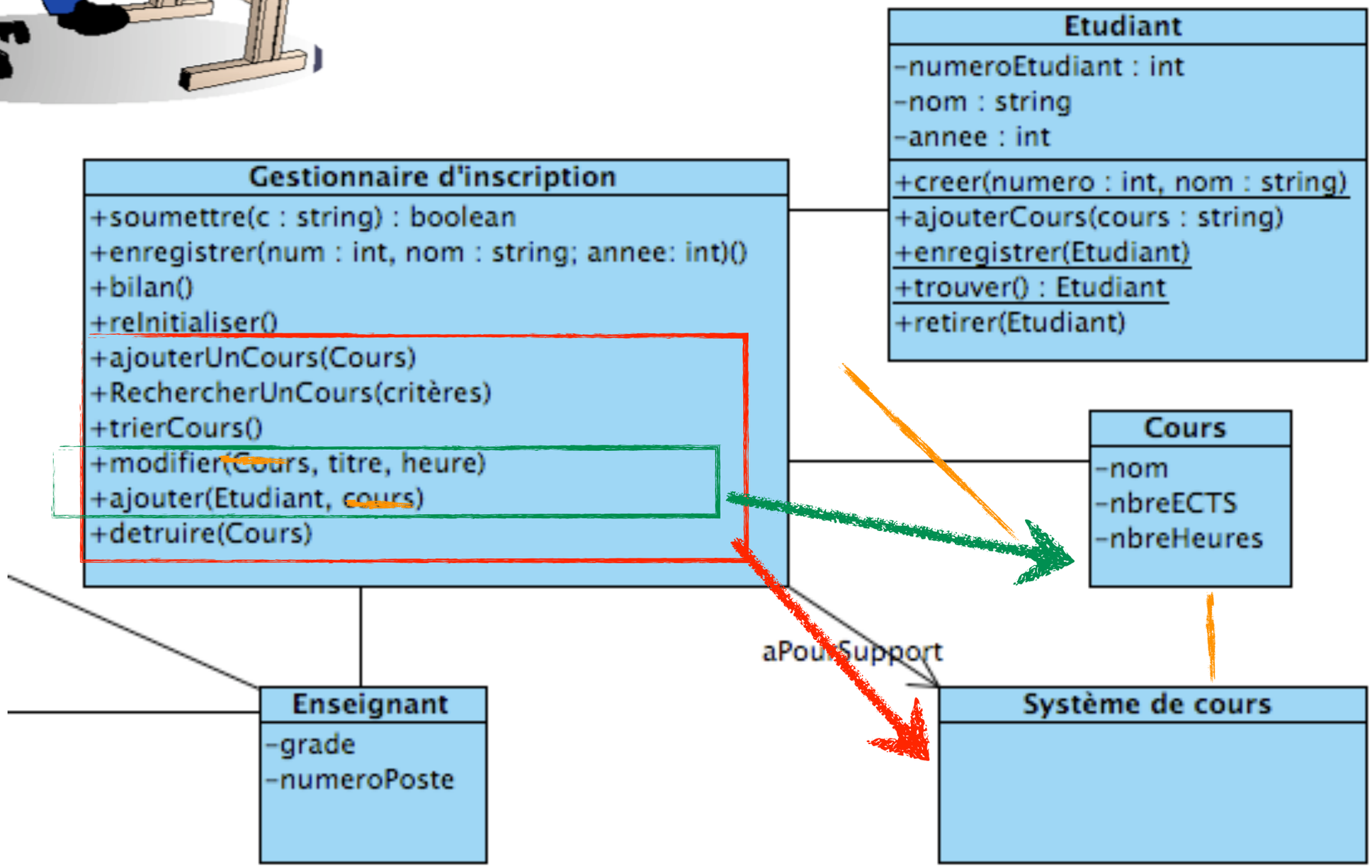
Propriété Statique ou Classe dédiée (Factory)



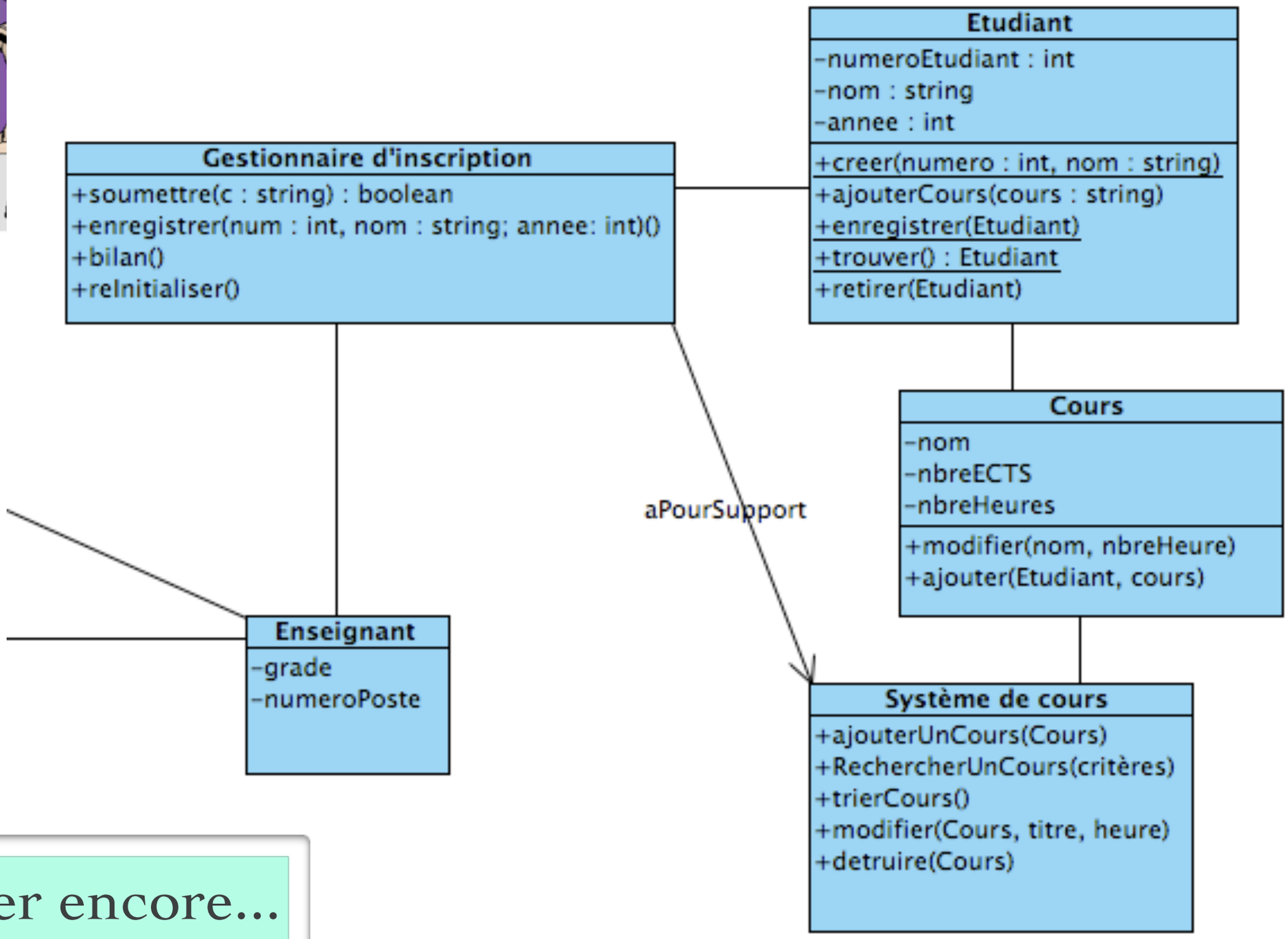
Des diagrammes de classes en conception

Anti-patterns

Blob ou la classe Dieu



Amélioration : prise de responsabilité



A Améliorer encore...