

MODÉLISATION DES APPLICATIONS

Manuele Kirsch Pinheiro

Maître de Conférences – Université Paris 1

mkirschpin@univ-paris1.fr / kirschpm@gmail.com

<http://mkirschp.free.fr>

2

Objectifs et Planning

- **Objectifs :**
 - Sensibiliser à la modélisation d'applications
 - Introduire / réviser le langage UML
 - Introduire le passage UML → code Java
- **Planning :**
 - 10h (CM / TP) en 4 séances
 - Séance 1 : Introduction à la modélisation
 - Séance 2 : Diagramme de classes UML
 - Séance 3 : Diagramme de séquence UML
 - Séance 4 : Passage UML → code
- **Evaluation**
 - Examen final

Références

- Bibliographie

- B. Charroux, A. Osmani, Y. Thierry-Mieg, « UML2 : Pratique de la modélisation », 2e édition, Pearson Education
- T. Weilkiens, B. Oestereich, « UML2 Certification guide : Fundamentals & intermediate examens », Morgan Kaufmann Publishers / Elsevier

- Sites Web

- <http://lgl.isnetne.ch/uml/>
- http://www.omg.org/gettingstarted/what_is_uml.htm
- <http://laurent-audibert.developpez.com/Cours-UML/>
- http://www.lamsade.dauphine.fr/~manouvri/UML/CoursUML_MM.html

Pourquoi Modéliser ?

- Pourquoi modéliser ?

- Mieux **appréhender** un système **complexe**
- Mieux **comprendre** le système qu'on conçoit / développe
- Mieux **maitriser** la complexité et assurer la **cohérence**

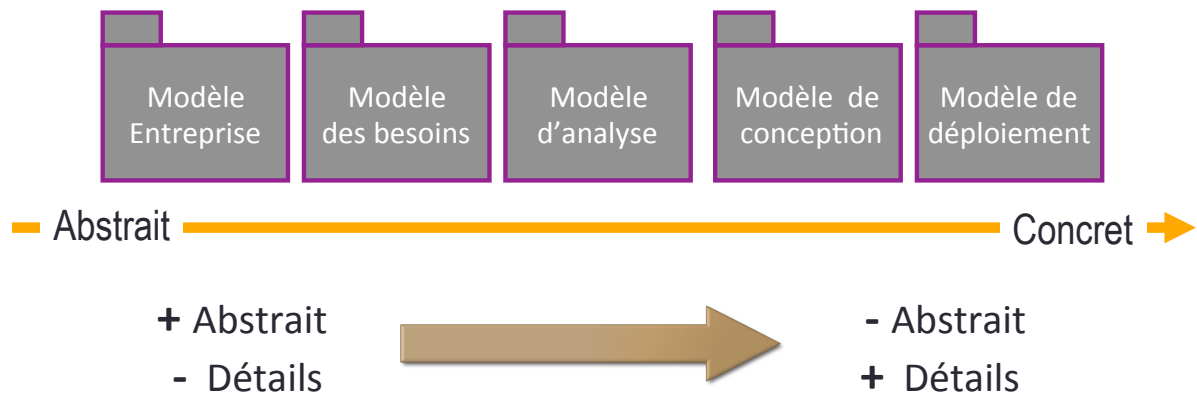
- Un modèle est ...

- Une vision simplifiée de la réalité
- Un outil de communication pour les acteurs engagés

Réduire la complexité pour mieux comprendre

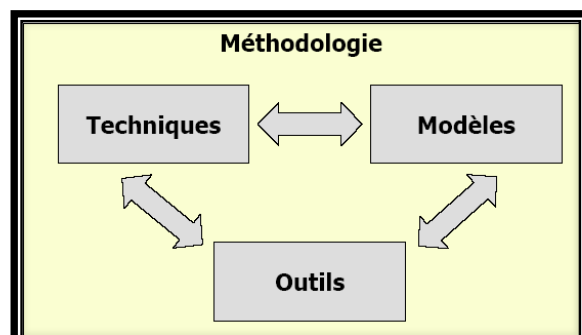
Modélisation

- La réalisation d'un système impose la création de modèles successifs
 - ◆ Chaque modèle représente le système à ***un certain niveau d'abstraction***



Méthodologies de développement

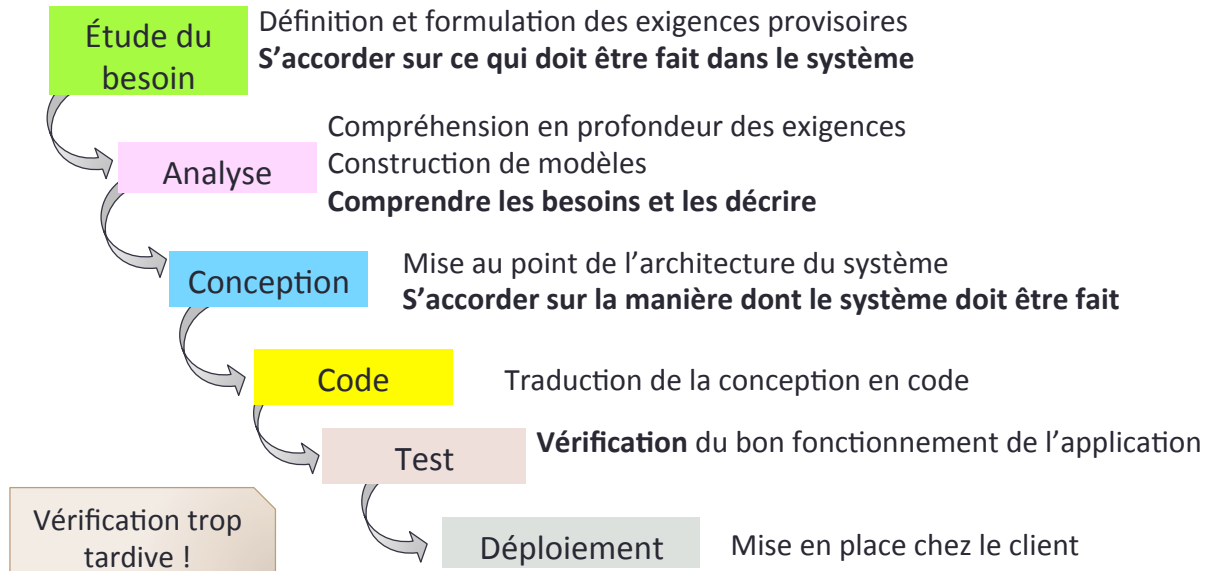
- **Méthodologie**
 - Démarche reproductible pour obtenir un résultat
- But : **Organiser la démarche de travail**
 - Procédés
 - Artefacts
 - Intervenants



Méthodologies de développement

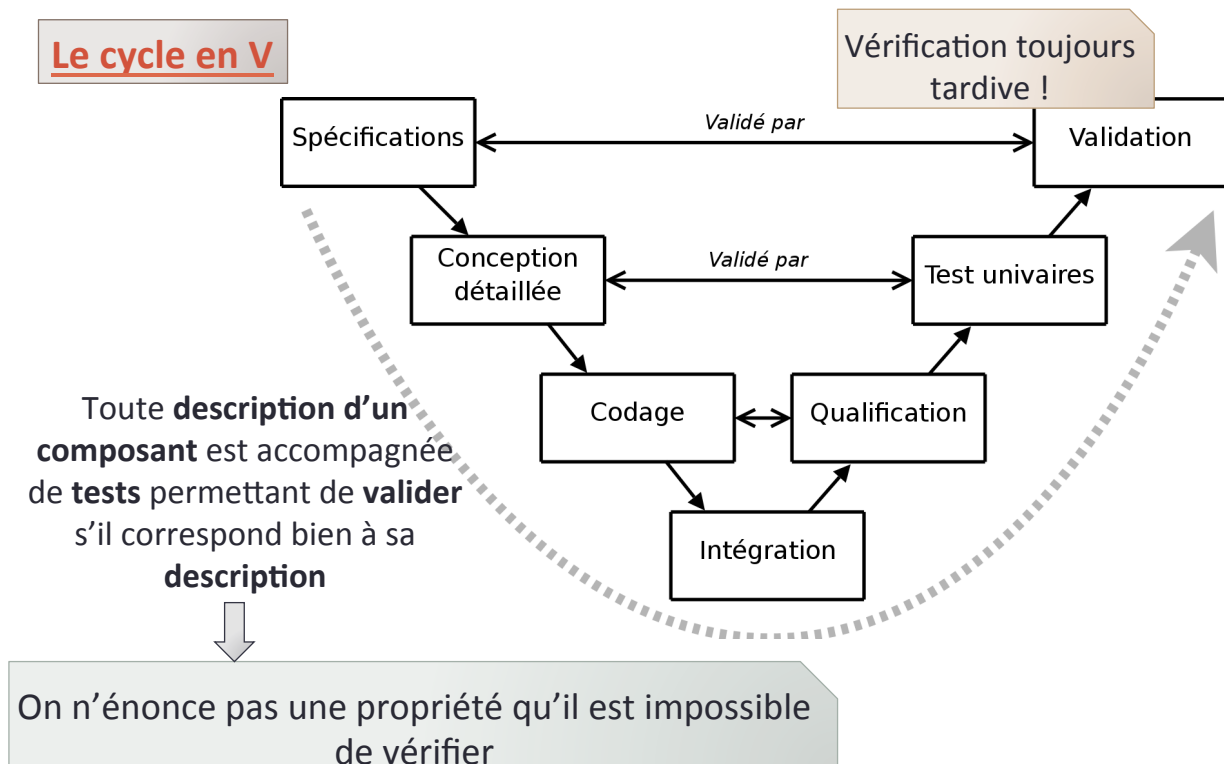
Cycle de vie en cascade

Étapes d'un projet informatique



Méthodologies de développement

Le cycle en V



Méthodologies de développement

- **Méthodes agiles**

- « Nouvelle » génération de méthodologies de développement
- **Agilité** : capacité de s'adapter et à réagir à l'environnement
- Priorité à la *satisfaction du client*

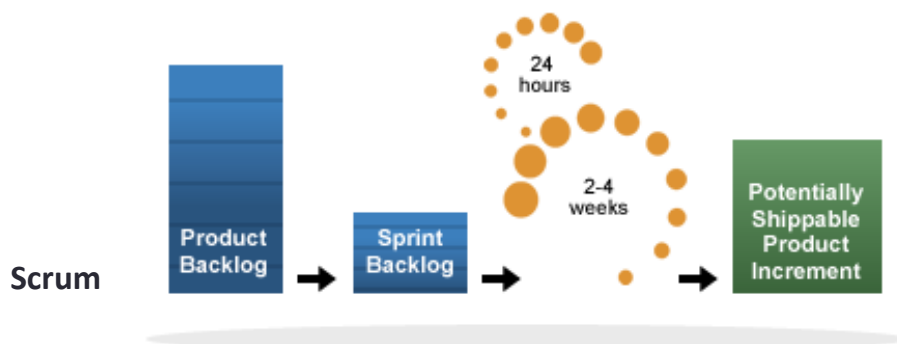
- **Manifeste Agile**

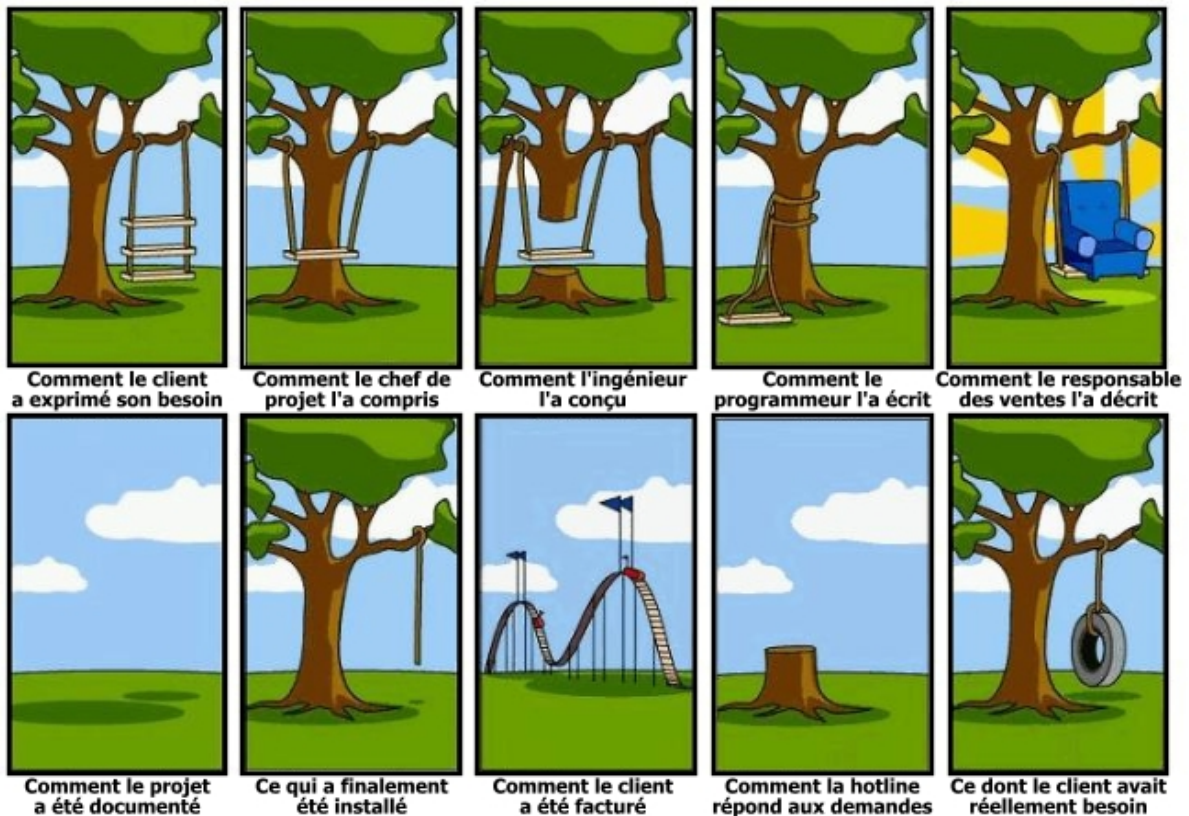
- **Individus** et **interactions** plus que processus et outils
- **Logiciels fonctionnel** plus que documentation abondante
- **Collaboration** avec le client plus que négociation contractuelle
- **Adaptation** au changement plus que suivi d'un plan

Méthodes agiles

- **Principes sous-jacents de l'agilité**

- **Intégration du client** au processus de développement
- **Livraison rapide et régulière** de fonctionnalités à forte valeur ajoutée
- **Acceptation du changement** dans les besoins
- **Travail d'équipe**
- **Simplicité et qualité**
- Attention continue à **l'excellence technique** et à une **bonne conception**





Réussite d'un Projet

- Difficile équilibre : **Qualité – Coût – Délai**
 - *Rapide et pas cher* → **Mauvaise qualité**
 - *Rapide et de bonne qualité* → **Cher**
 - *Bonne qualité et pas cher* → **Lent**
 - *Lent, de mauvaise qualité, et cher* → **Catastrophe !!**
- **Chaos Report** by Standish Group 2009
 - Seulement **32%** des projets **réussissent** (temps, budget et *features*)
 - **44%** **ne respectent pas** les délais, les coûts et/ou les besoins énoncés
 - **24%** des projets **n'aboutissent pas**
- **Facteurs de réussite**
 - *Implication de l'utilisateur*
 - *Exigences et spécifications claires*



UML : quoi ?!



UNIFIED MODELING LANGUAGE™

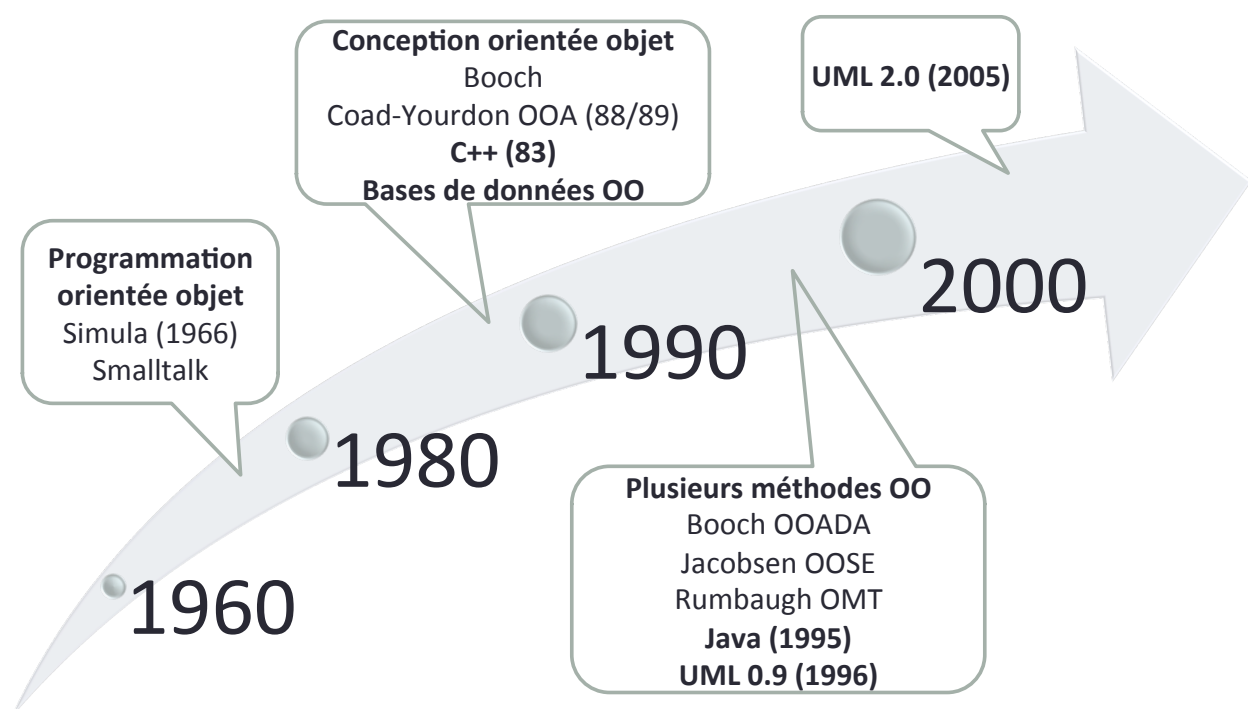


- UML, c'est quoi ?
 - UML est une **notation**, un **formalisme** permettant la modélisation
 - *UML n'est pas une méthode !*
- UML est un langage de **modélisation objets** conçu pour
 - *visualiser*
 - *spécifier*
 - *construire*
 - *documenter*

les artefacts d'un système à forte composante logicielle

- **Standard OMG** depuis 1997

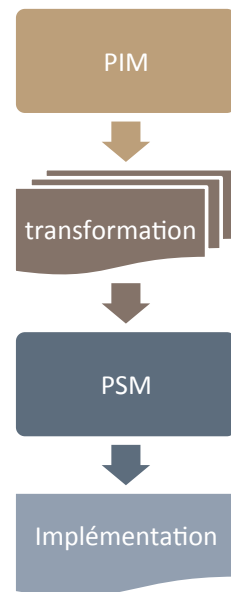
Historique



UML & MDA

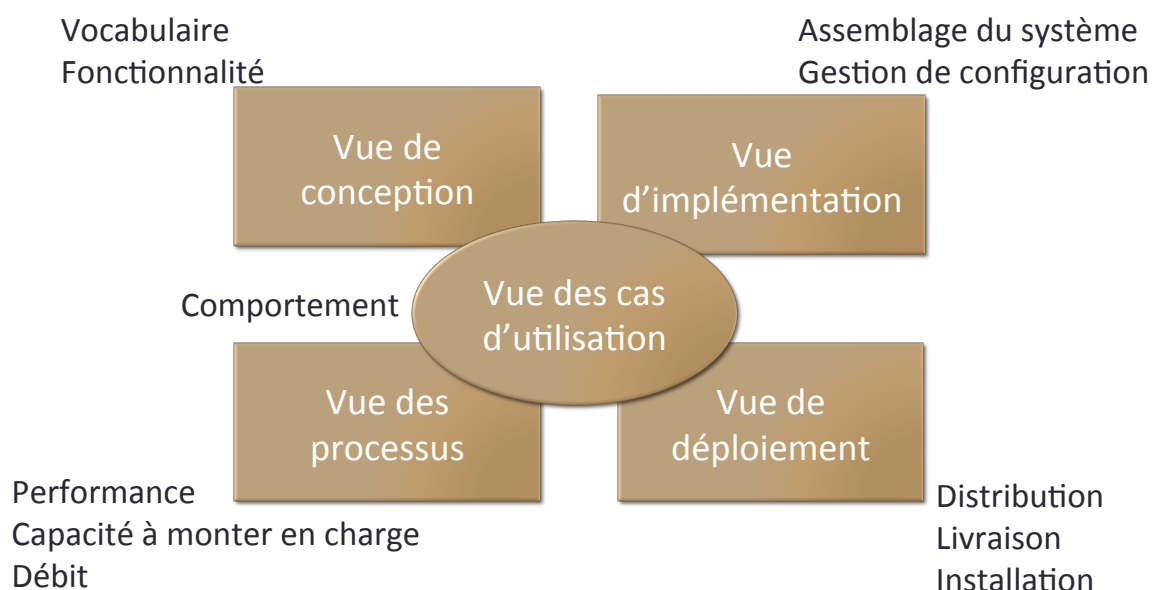
- **MDA (Model Driven Architecture)**

- Démarche d'**ingénierie dirigée par les modèles (IDM)**
- Développement de systèmes par l'**élaboration** et la **transformation** successives de **différents modèles**, jusqu'à l'**implémentation**



UML : 5 vues

Différents diagrammes pour différentes vues



Les diagrammes d'UML

Diagrammes structurels vues statiques

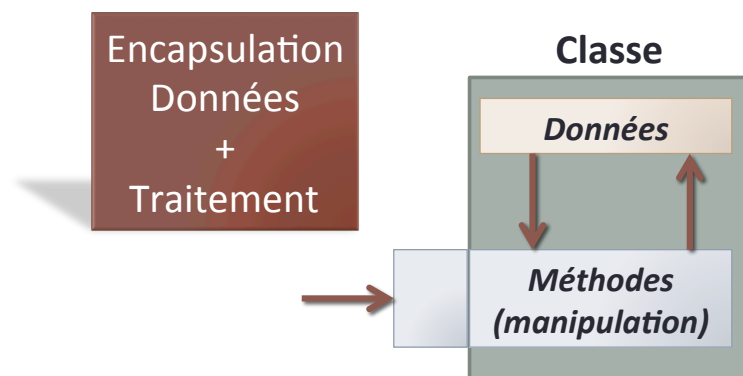
- **Diagrammes de classes**
- **Diagrammes d'objets**
- Diagrammes de composants
- Diagrammes de déploiement
- **Diagrammes de paquetage**

Diagrammes comportementaux vues dynamiques

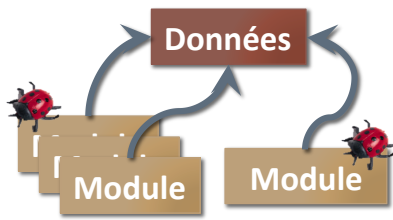
- Diagrammes de cas d'utilisation
- **Diagrammes de séquence**
- Diagrammes d'activités
- Diagrammes de communication (collaboration)
- Diagrammes d'états
- Timing diagram
- Interaction overview diagram

Orientation à Objets

- Concepts clés de l'orientation à objets
 - Classe & objets
 - Encapsulation
 - Héritage

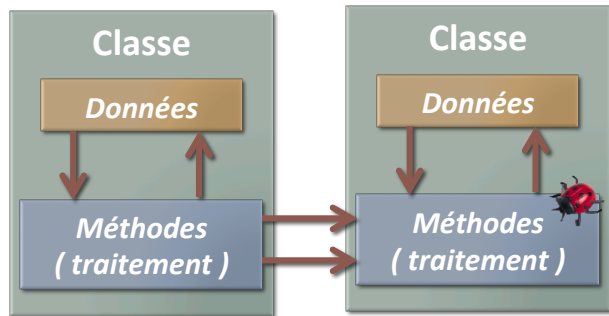


Pourquoi l'orientation à objets ?



Programmation Modulaire

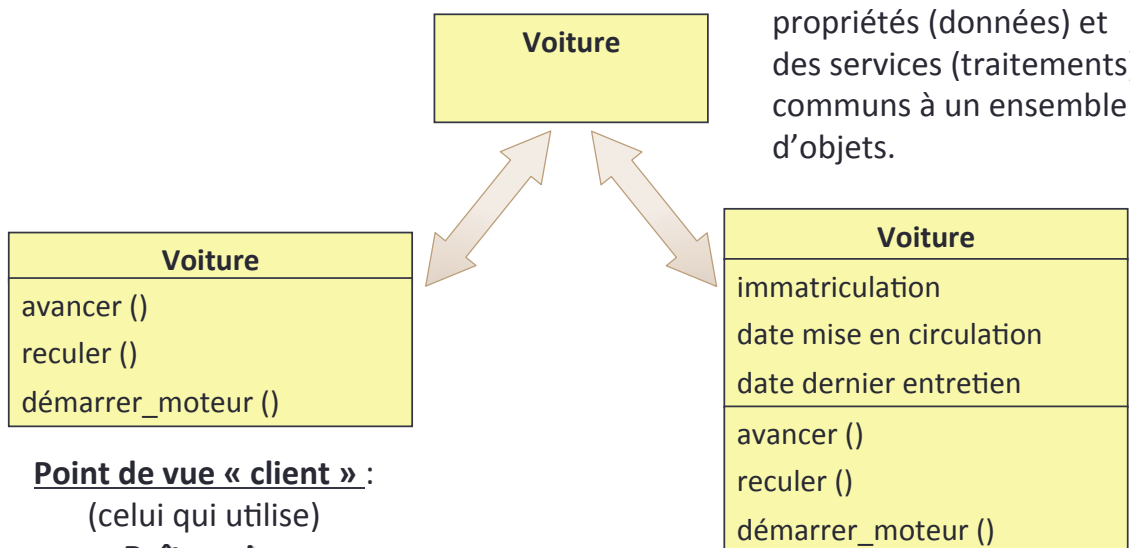
- Plusieurs modules manipulent les données
- Difficile à déboguer / maintenir
- Difficile à faire évoluer



Programmation Orientée à Objets

- Traitement des données isolé dans la classe
- ⊕ facile à réutiliser
- ⊕ facile à maintenir / déboguer
- ⊕ facile à faire évoluer

Réutilisation



Point de vue « client » :

(celui qui utilise)

Boîte noire

Peu importe l'implémentation, tant qu'elle offre les services

Classe : abstraction des propriétés (données) et des services (traitements) communs à un ensemble d'objets.

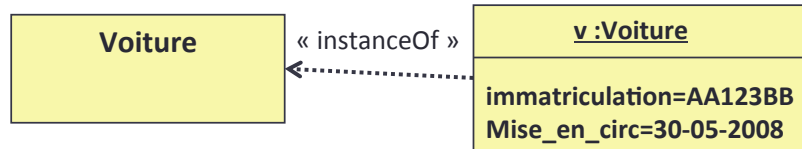
Point de vue « concepteur » :

(celui qui conçoit / implémente)

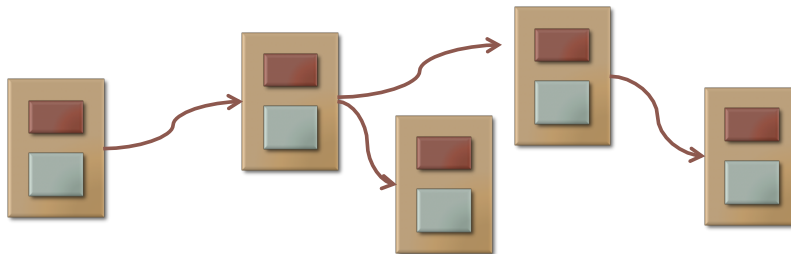
Boîte blanche

Prise en charge de l'implémentation

Réutilisation

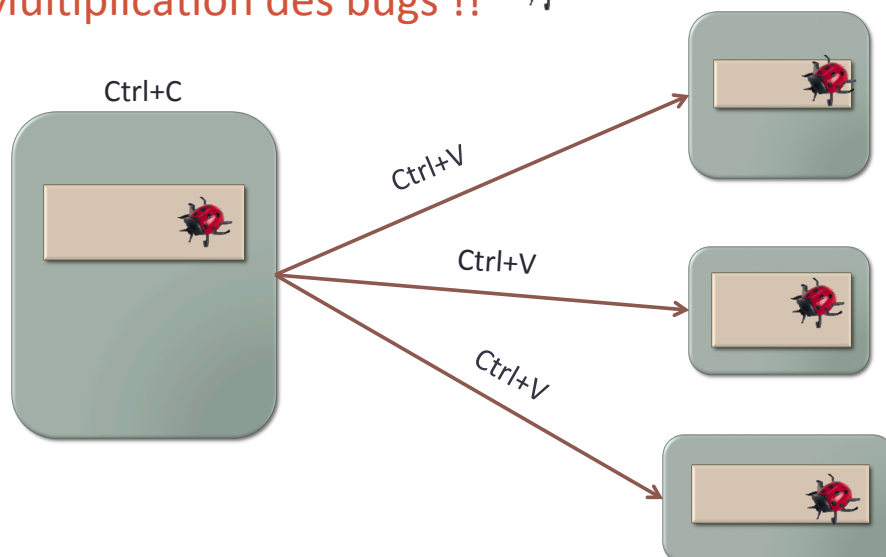


- Dans la POO, les modules logiciels réutilisables sont les classes
 - *On réutilise les classes*
 - Une **classe** détermine les **propriétés** qui peut avoir un objet et son **comportement**
 - Un **objet** est l'**instance** d'une classe. Il a un **état** (valeurs attribuées aux propriétés à un moment t), mais son **comportement** est régi par la classe
 - Un programme est constitué par un **ensemble interconnecté de classes**, mais son **exécution** s'opère sur les **objets** (instances)
- **Complexité d'une application → décomposition en modules**



Concevoir un code de qualité

- Problème de la stratégie du « copier-coller »
 - **Multiplication des bugs !!** 



Concevoir un code de qualité

- Critères de qualité dans l'orientation à objets
 - **Modularité** :
 - **forte cohésion** dans la classe, **faible couplage** entre les classes
 - **Robustesse** :
 - capacité d'un programme à **bien fonctionner**, sans bugs
 - **Extensibilité** :
 - possibilité d'**étendre** facilement les fonctionnalités d'un programme, **sans compromettre son intégrité**
 - **Evolutivité** :
 - possibilité de faire concevoir un logiciel de manière **incrémentale**
 - **Réutilisabilité** :
 - possibilité de réutiliser **sans modification** une classe

Anti-exemples

- **Modularité**
 - Cette classe est-elle bien définie ?

```
public class Personne {
    String noom, permis, carteLecteur;
    Float salaire;

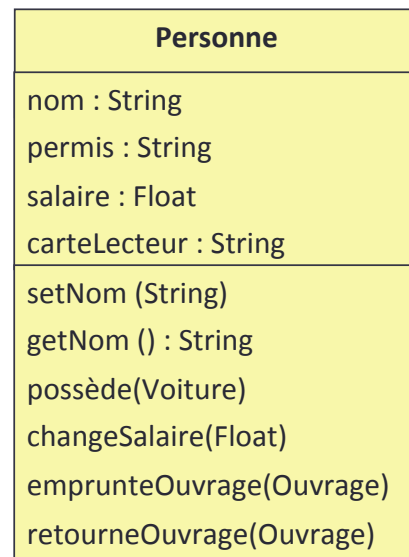
    public void setNom (String s) { ... }
    public String getNom () { ... }
    public void possede (Voiture v) { ... }
    public void changeSalaire (Float s) { ... }
    public void emprunteOuvrage (Ouvrage o) { ... }
    public void retourneOuvrage (Ouvrage o) { ... }
}
```

Personne
nom : String permis : String salaire : Float carteLecteur : String
setNom (String) getNom () : String possède(Voiture) changeSalaire(Float) emprunteOuvrage(Ouvrage) retourneOuvrage(Ouvrage)

Anti-exemples

• Modularité

- Cette classe est-elle bien définie ?
- **Non !!**
 - Personne → nom
 - Employé → salaire
 - Conducteur → permis, voiture
 - Lecteur → carte lecteur, ouvrage



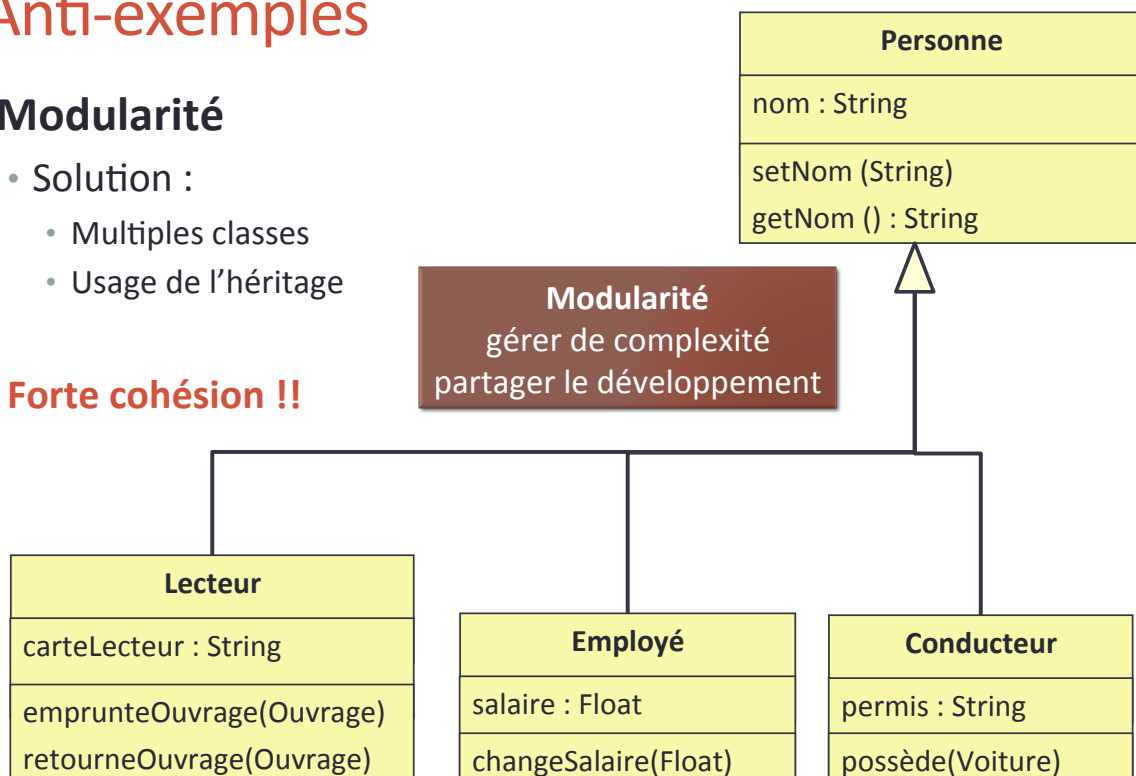
Faible cohésion !!

Anti-exemples

• Modularité

- Solution :
 - Multiples classes
 - Usage de l'héritage

Forte cohésion !!



Anti-exemples

• Réutilisabilité

- Cette classe est-elle réutilisable ?
- **Non !!**
 - Et si on voulait lire la température à partir du clavier ? ou d'une interface graphique ??
 - **Nouvelle classe !! ☹**

Convertisseur
main (args [0..*] : String)

```

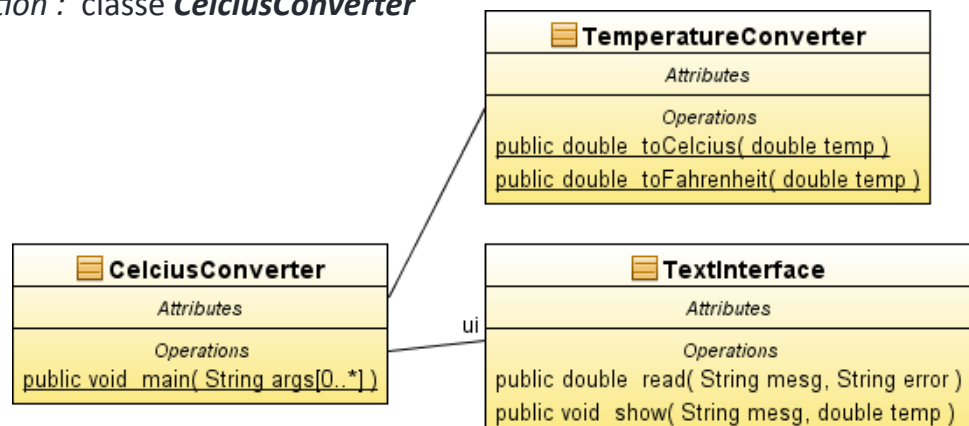
7  +  /**...*/
11  public class Convertisseur {
12  -    public static void main(String[] args) {
13      double temp = Double.parseDouble(args[0]);
14      double celc = ((temp-32)*5)/9;
15      System.out.println("Temperature en C° :"+celc);
16  }
17  }

```

Anti-exemples

• Réutilisabilité

- Solution : Séparer les responsabilités
 - **1 classe = 1 responsabilité**
 - Conversion de température : classe **TemperatureConverter**
 - Interface avec l'utilisateur : classe **TextInterface**
 - Application : classe **CelciusConverter**



Anti-exemples

- **Extensibilité / évolutivité**

- Développer un calculateur
 - La direction métier affirme n'avoir besoin que de deux opérateurs : '+' et '-'
 - C'est très urgent !

Anti-exemples

- **Extensibilité / évolutivité**

- Développer un calculateur
 - La direction métier affirme n'avoir besoin que de deux opérateurs : '+' et '-'
 - C'est très urgent !

Calculateur
a: int
b: int
op: char
init ()
calc() : int

```

11 public class Calculateur {
12
13     int a, b; //operands
14     int op; //opérateur
15
16     public Calculateur () {...}
21
22     public void init() {...}
31
32     public int calc() {
33         int r = 0;
34
35         switch (this.op) {
36             case '+':
37                 r = a + b;
38                 break;
39             case '-':
40                 r = a - b;
41                 break;
42             default:
43                 System.out.println("Opérateur inconnu");
44         }
45         return r;
46     }
47
48     public static void main (String args[]) {...}
53 }

```

Anti-exemples

- **Extens**

- Déve

- La direction met
- opérateurs : '+' e
- C'est très urgent

Après la mise en production, la direction métier demande l'ajout de nouvelles fonctionnalités :

deux nouvelles opérations * et /

Calculateur
a: int
b: int
op: char
init ()
calc(): int

```

31
32 public int calc() {
33     int r = 0;
34
35     switch (this.op) {
36         case '+':
37             r = a + b;
38             break;
39         case '-':
40             r = a - b;
41             break;
42         default:
43             System.out.println("Opérateur inconnu");
44     }
45     return r;
46 }
47
48 public static void main (String args[]) { ... }
53

```

Anti-exemples

- **Extens**

- Déve

- La direction met
- opérateurs : '+' e
- C'est très urgent

Après la mise en production, la direction métier demande l'ajout de nouvelles fonctionnalités :

deux nouvelles opérations * et /

Calculateur
a: int
b: int
op: char
init ()
calc(): int

```

31
32 public int calc() {
33     int r = 0;
34
35     switch (this.op) {
36         case '+':
37             r = a + b;
38             break;
39         case '-':
40             r = a - b;
41             break;
42         default:
43             System.out.println("Opérateur inconnu");
44     }
45     return r;
46 }
47
48 public static void main (String args[]) { ... }
53

```

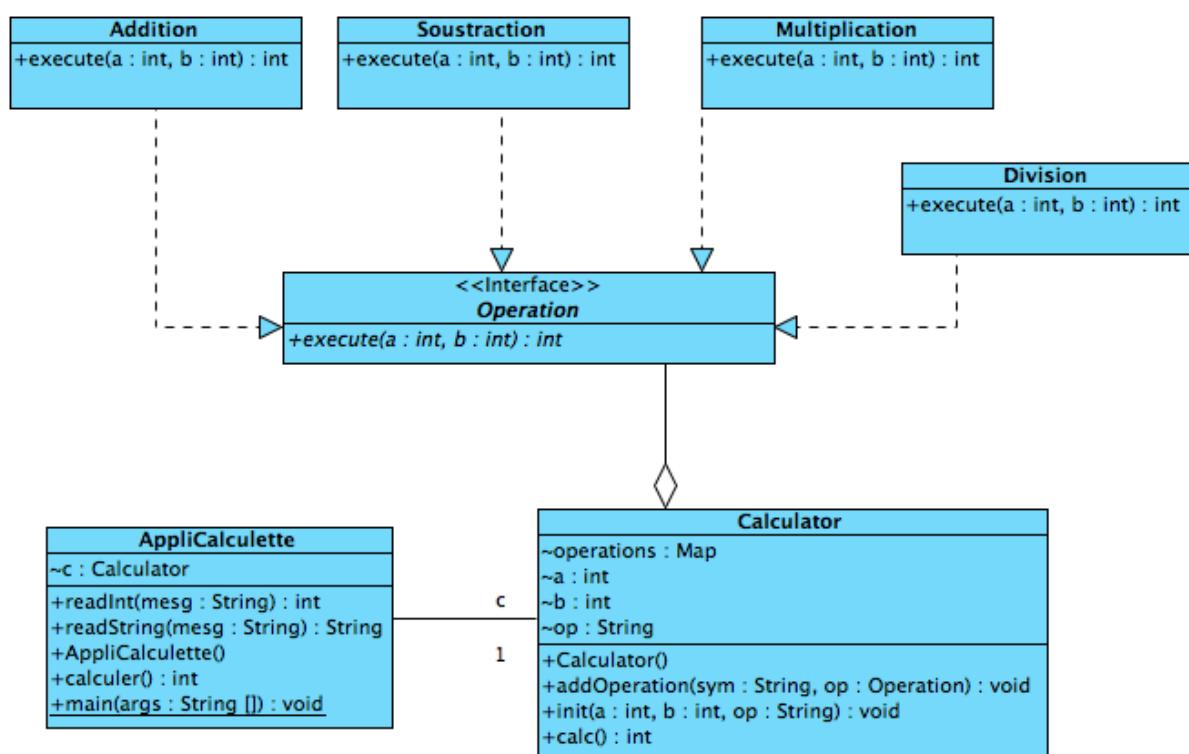
Réutilisation difficile !
Classe non-extensible

Anti-exemples

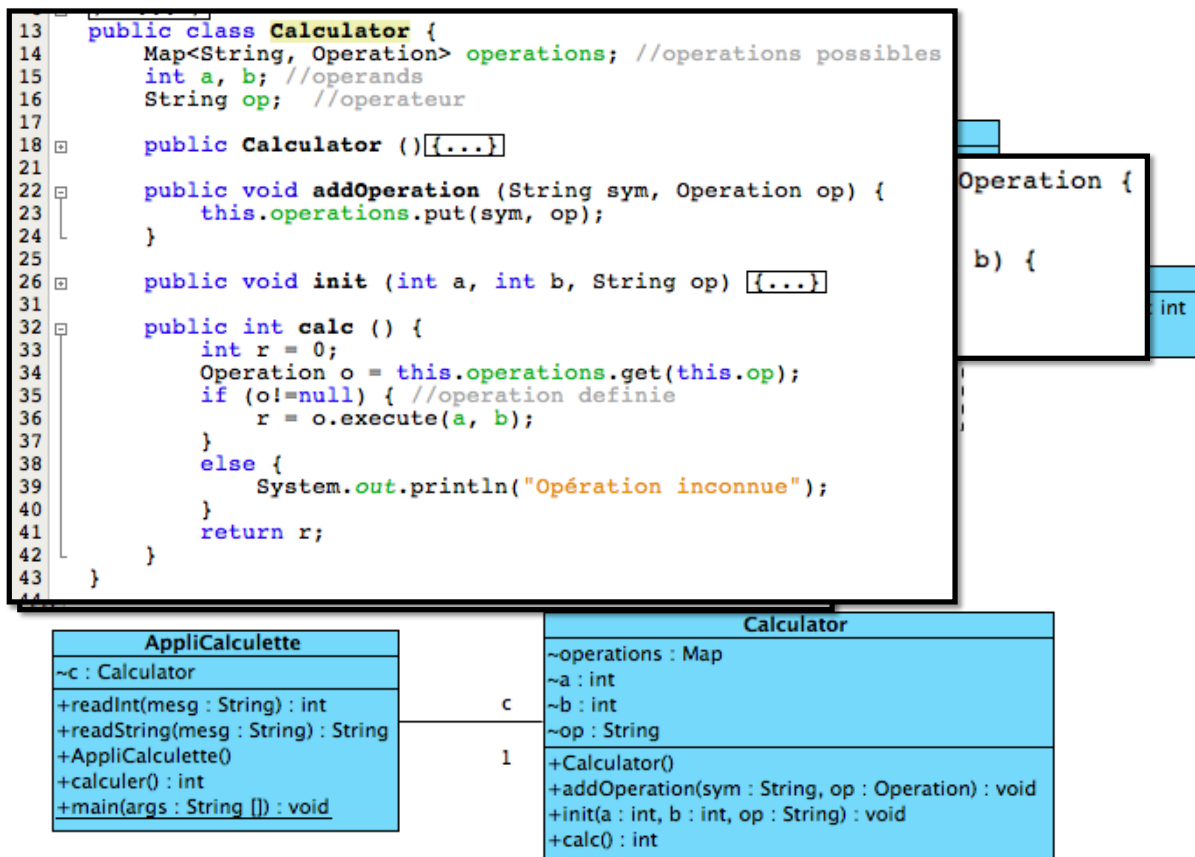
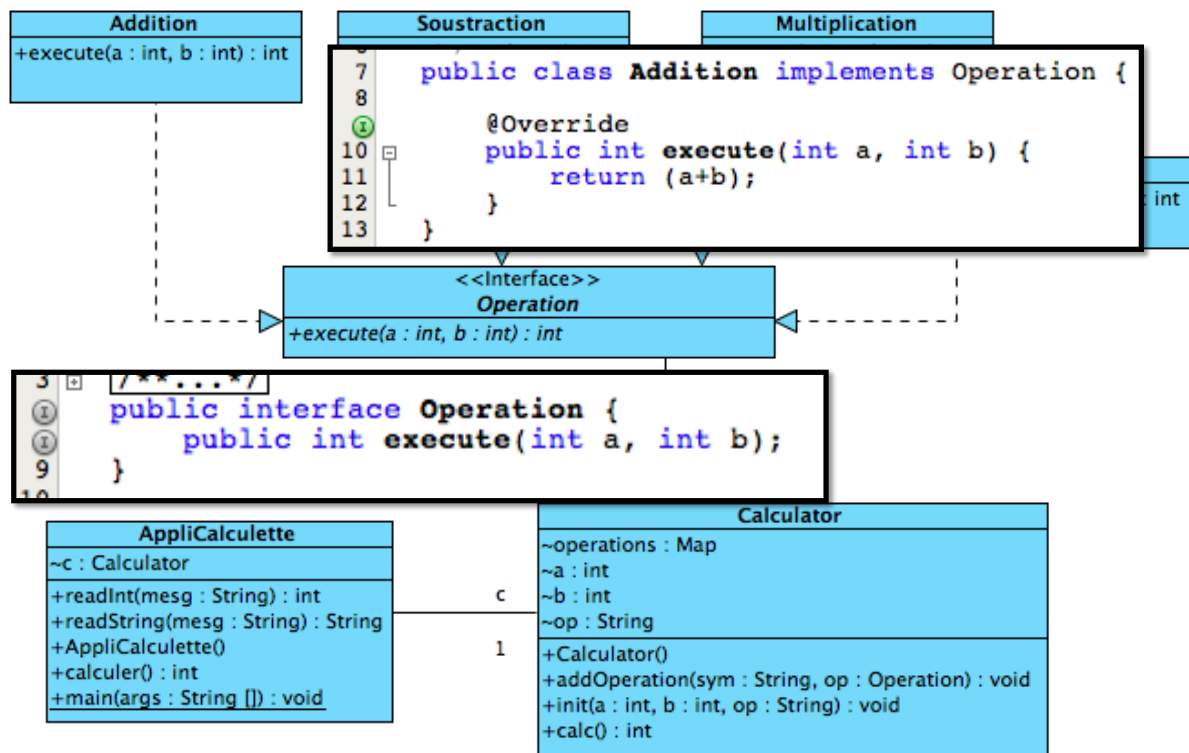
- **Extensibilité / évolutivité**

- Comment pouvoir ajouter de nouvelles opérations sans affecter le code existant ??
 - **Factoriser les responsabilités !!**
 - **Opération** : définition abstraite d'une opération
 - **Calculateur** : exécution des opérations, quelque soit l'opération
 - **Application principale** : interaction utilisateur, définition des opérations
- Design pattern **Strategy**

Anti-exemples



Anti-exemples



```

13 public class Calculator {
14     Map<String, Operation> operations; //operations possibles
15     int a, b; //operands
16     String op; //opérateur
17
18     public Calculator () {
19         //initialisation
20     }
21
22     public void addOperation (String sym, Operation o) {
23         this.operations.put(sym, o);
24     }
25
26     public void init (int a, int b, String op) {
27         this.a = a;
28         this.b = b;
29         this.op = op;
30     }
31
32     public int calc () {
33         int r = 0;
34         Operation o = this.operations.get(this.op);
35         if (o != null) {
36             r = o.execute(a, b);
37         }
38         else {
39             System.out.println("Opérateur non reconnu");
40         }
41         return r;
42     }
43 }

```

```

11 public class AppliCalculette {
12     /**...*/
13     public int readInt (String msg) {...}
14
15     public String readString (String msg) {...}
16
17     Calculator c;
18
19     public AppliCalculette() {
20         c = new Calculator();
21         c.addOperation("+", new Addition());
22         c.addOperation("-", new Soustraction());
23         c.addOperation("*", new Multiplication());
24         c.addOperation("/", new Division());
25     }
26
27     public int calculer () {
28         int a = readInt("a ? ");
29         int b = readInt("b ? ");
30         String op = readString("op ?");
31
32         c.init(a, b, op);
33         return c.calc();
34     }
35
36     public static void main (String args[]) {...}
37 }

```

AppliCalculette
~c : Calculator
+readInt(msg : String) : int
+readString(msg : String) : String
+AppliCalculette()
+calculer() : int
+main(args : String []) : void

Calculator
+Calculator()
+addOperation(sym : String, op : Operation) : void
+init(a : int, b : int, op : String) : void
+calc() : int

Références

- <http://www.uml.org/>
- <http://www.omg.org/mda/index.htm>
- <http://epi.univ-paris1.fr/ufr06m1info>
- http://www1.standishgroup.com/newsroom/chaos_2009.php
- <http://www.projectsmart.co.uk/docs/chaos-report.pdf>
- <http://www.geek-directeur-technique.com/2009/07/10/le-triangle-qualite-cout-delai>
- http://www.scrumalliance.org/pages/what_is_scrum
- <http://agilemanifesto.org/iso/fr/manifesto.html>

MODÉLISATION DES APPLICATIONS

Manuele Kirsch Pinheiro

Maître de Conférences – Université Paris 1

mkirschpin@univ-paris1.fr / kirschpm@gmail.com

<http://mkirschp.free.fr>

40

Objectifs et Planning

- **Objectifs :**
 - Sensibiliser à la modélisation d'applications
 - Introduire / réviser le langage UML
 - Introduire le passage UML → code Java
- **Planning :**
 - 10h (CM / TP) en 4 séances
 - ✓ Séance 1 : Introduction à la modélisation
 - **Séance 2 : Diagramme de classes UML**
 - Séance 3 : Diagramme de séquence UML
 - Séance 4 : Passage UML → code
- **Evaluation**
 - Examen final

Les diagrammes d'UML

Diagrammes structurels vues statiques

- **Diagrammes de classes**
- **Diagrammes d'objets**
- Diagrammes de composants
- Diagrammes de déploiement
- **Diagrammes de paquetage**

Diagrammes comportementaux vues dynamiques

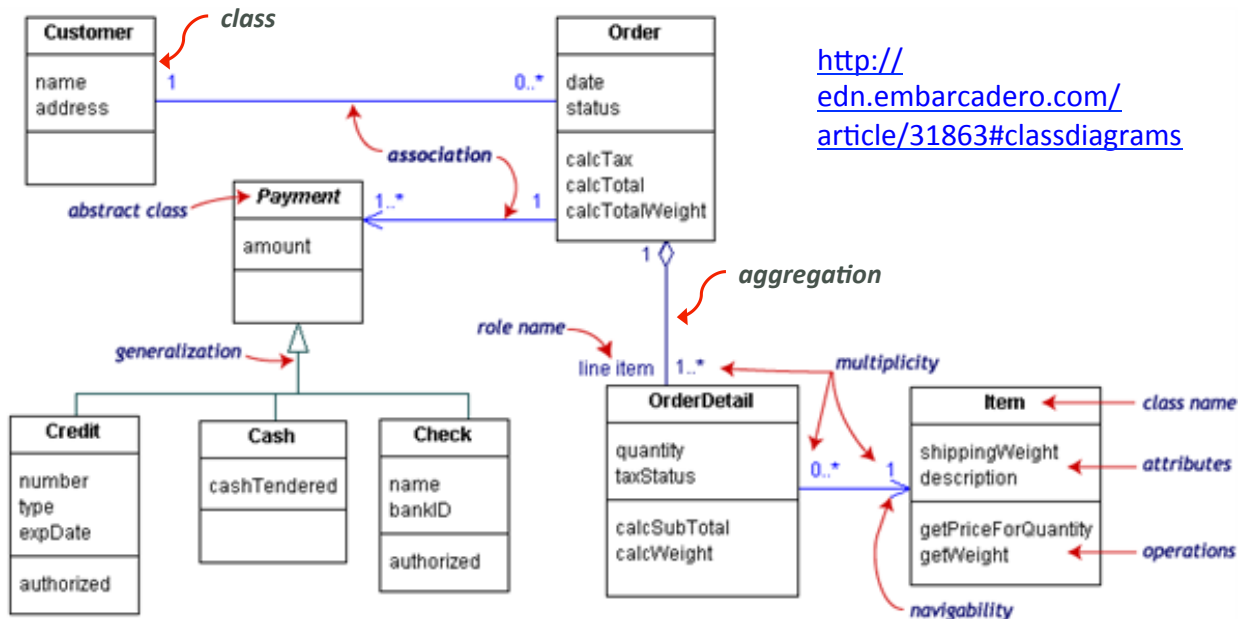
- Diagrammes de cas d'utilisation
- **Diagrammes de séquence**
- Diagrammes d'activités
- Diagrammes de communication (collaboration)
- Diagrammes d'états
- Timing diagram
- Interaction overview diagram

Diagramme de classes

- **Diagramme de classe**
 - Principal diagramme de l'approche objets
 - Description des classes du monde réel et de leurs relations
 - Montrer la **structure statique** du système
- Un diagramme de classe décrit les classes et les relations, leur regroupement en paquetage, les interfaces...
 - Définir les **classes** et leurs **responsabilités**
 - Définir les **relations** (associations, agrégations, héritage...) entre ces classes
 - Les **paquetages** organisant ces classes

Diagramme de classes

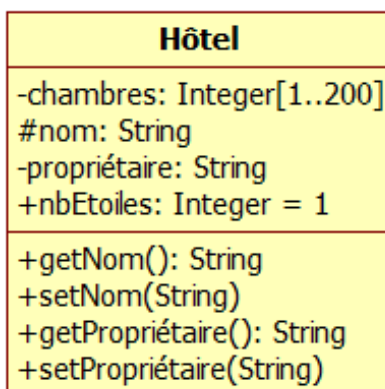
R é s u m é



<http://edn.embarcadero.com/article/31863#classdiagrams>

Diagramme de classes : classe

- **Classe**
 - Description formelle d'un ensemble d'objets ayant une sémantique et des caractéristiques communes
 - Trois compartiments : **nom** (obligatoire), **attributs**, **méthodes**



Nom : il doit être significatif et commencer par majuscule

Liste d'attributs: les attributs définissent des informations d'une classe ou d'un objet

Liste de méthodes: les méthodes définissent le comportement d'une classe

Diagramme de classes : classe

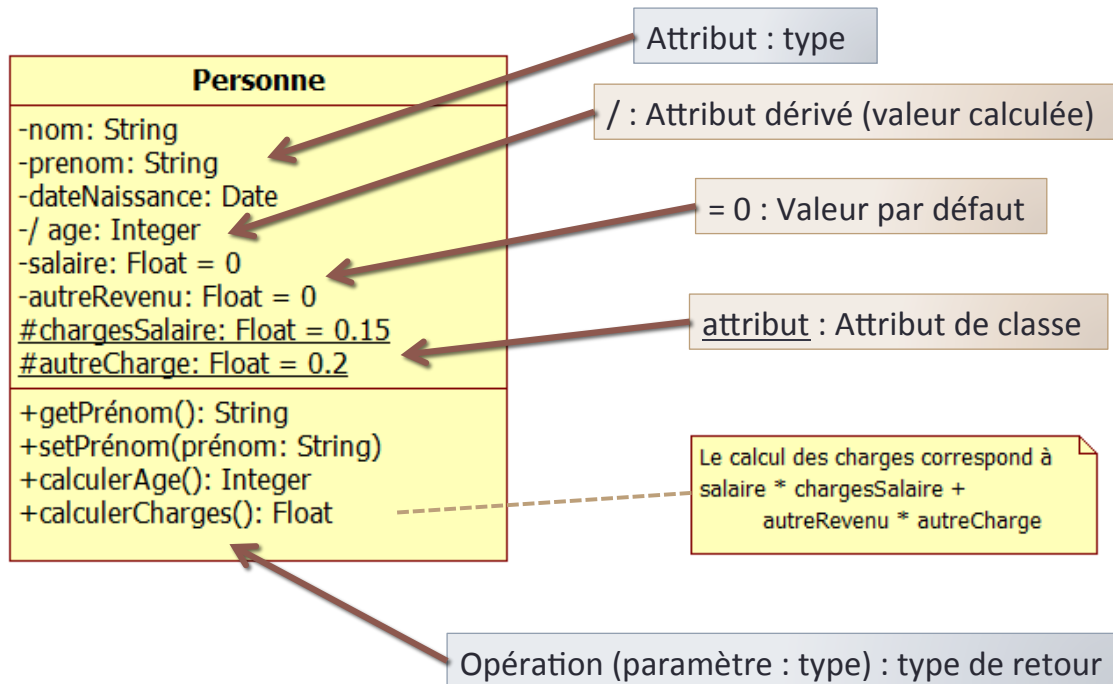


Diagramme de classes : classe

- **Visibilité** : UML prévoit **4 niveaux** de visibilité
 - **Private** (-) : visible **uniquement** à l'intérieur de l'**objet**
 - **Protected** (#) : visible à l'intérieur de la **classe** et de ses **sous-classes**
 - **Package** (~) : visible à l'intérieur du **paquetage**
 - **Publique** (+) : visible à tout le **monde**



Application du principe de l'encapsulation !!

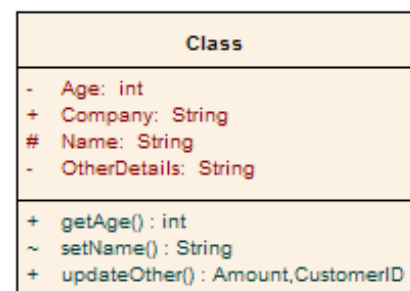


Diagramme de classes : classe

- **Opérations**

- Services offerts par une classe
- Une **opération** peut être mise en œuvre par **plusieurs méthodes**
 - **Polymorphisme & surcharge**

- **Envoi de message**

- Appel d'un service offert par une opération
- Un **objet o_1** invoque une **opération op** offerte par un **objet o_2**



Diagramme de classes : classe

- Méthode « **constructeur** » :
 - **Opération** spéciale responsable de la **création d'une nouvelle instance (objet)**.
 - Définition de l'état initial de l'objet
 - Transmission de tous les renseignements nécessaires à la nouvelle instance
 - **Car (puiss : Float, coul : String, portes : Integer)**

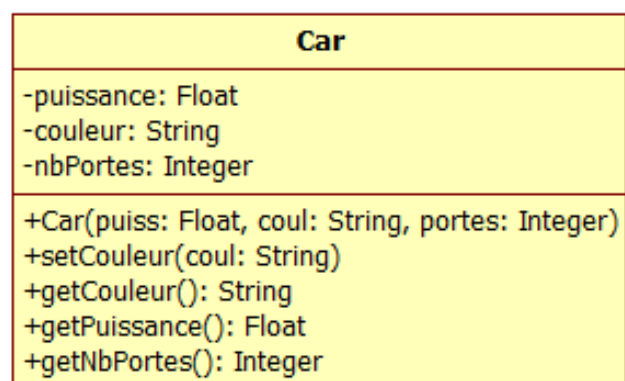


Diagramme de classes : héritage

- **Héritage** (généralisation / spécialisation)
 - **Réutilisation** d'une classe pour la création d'une nouvelle
 - **Relation de classification** entre un élément \oplus général et un élément \oplus spécifique
 - Les **sous-classes héritent** tous les attributs et les opérations de la **superclasse**
 - Toutes les **associations** de la classe mère **s'appliquent** aux sous-classes

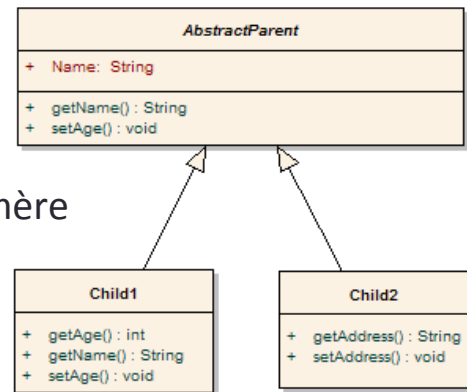
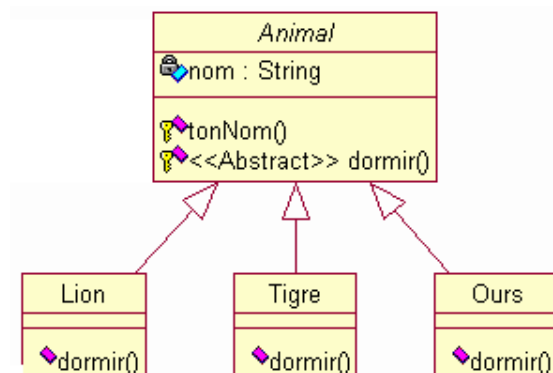


Diagramme de classes : héritage

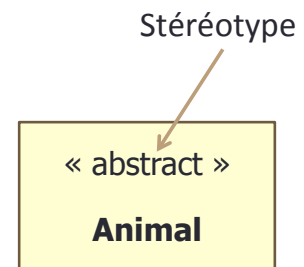
- **Classes abstraites**
 - Les classes abstraites représentent une abstraction afin de factoriser des **propriétés/opérations** communes
 - Spécifications générales à un ensemble de sous-classes
 - Généralisation d'un concept abstrait commun à plusieurs classes concrètes
 - Classes non instantiables
 - Pas d'implémentation complète
 - Stéréotype « **abstract** »



UML : Stéréotypes

• Stéréotype

- Un stéréotype ajoute une **description sémantique** à un élément du modèle
- Un stéréotype permet la **classification** des possibles usages d'un élément du modèle
- Il s'agit d'un mécanisme d'extension propre à UML



• Stéréotypes prédéfinis en UML

- « Actor »
- « Instantiate »
- « Implement »
- « Call »
- ...

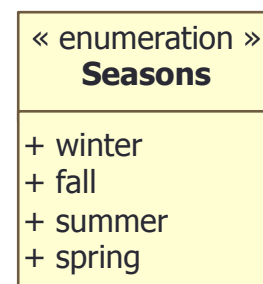


Diagramme de classes : associations

• Associations

- *Connexion sémantique entre les classes*
- Les associations représentent des **relations structurelles** entre les classes

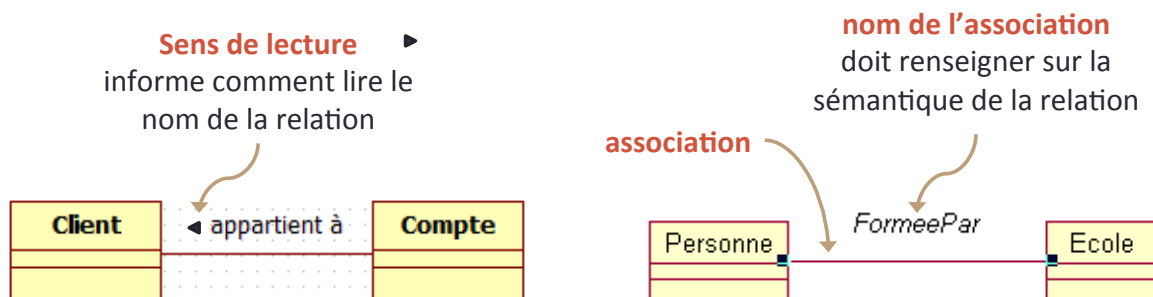


Diagramme de classes : associations

• Rôle

- Le **rôle de la classe** au sein d'une **association**
 - Les rôles agissent comme une propriété (un attribut)
 - Ils peuvent donc avoir une visibilité



• Multiplicité

- Nombre d'objets** qui peuvent être **liés à un seul objet de l'autre classe**
- A combien d'objets** du côté opposé **un objet peut être lié**
- Indiqué par un intervalle **min..max**
 - Ex.: Une personne possède plusieurs réservations, une réservation ne concerne qu'une seule personne

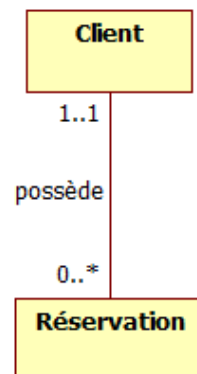


Diagramme de classes : associations

• Multiplicité

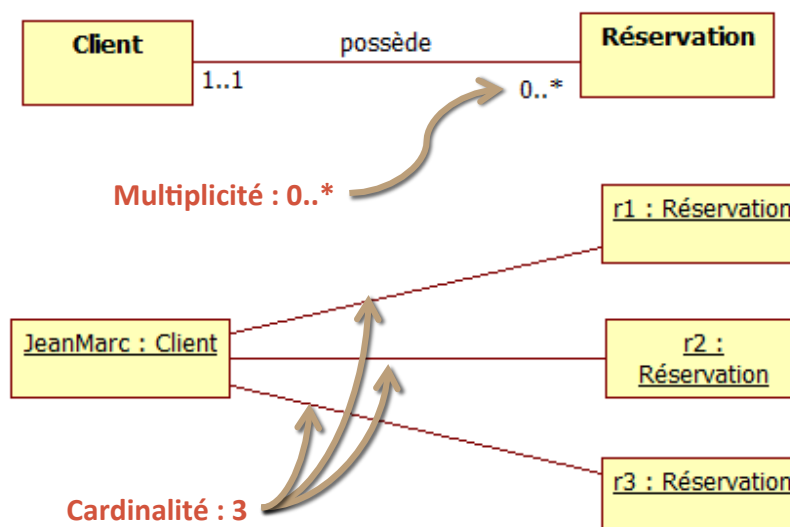


Diagramme de classes : associations

• Navigabilité

- Sens de circulation de l'information
- La navigabilité indique **si un objet o1** peut **accéder** à un **objet o2** dans l'autre extrémité du lien
 - Ex.: Un utilisateur peut connaître un mot de passe, mais le mot de passe ne connaît pas l'utilisateur
- À ne pas confondre avec le sens de lecture !!

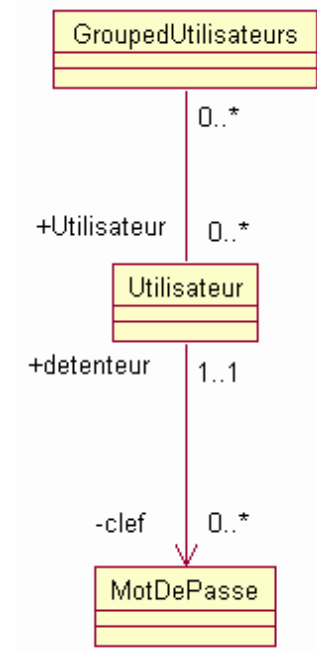
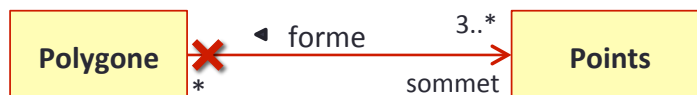


Diagramme de classes : associations

• Associations binaires et n-aires

- **Binaires** : relie 2 classes entre elles



- **N-aires** : relie plus de 2 classes entre elles

- Plus rares , peu utilisées
- Plus difficiles à comprendre

• Multiplicité ?!

- Pour **< Traveler X, Seat S >**, combien d'objets **Train** ?
- Pour **< Train T, Seat S >**, combien d'objets **Traveller** ?
- Pour **< Traveler X, Train T >** combien d'objets **Seat** ?

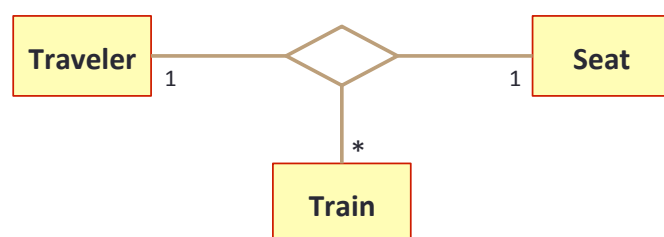
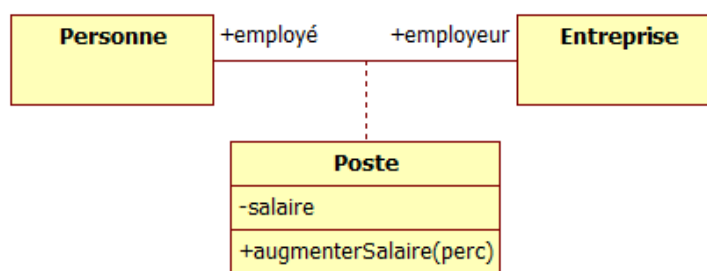


Diagramme de classes : classe-association

• Classe-association

- Lorsqu'une **association possède des attributs** qui lui sont propres, l'association devient une classe-association
- Il s'agit d'une **association promue** au rang de **classe**
 - On peut lui attribuer des **attributs** et des **opérations** comme n'importe quelle classe



Un **poste** n'existe que s'il existe une **personne** et une **entreprise**

Diagramme de classes : agrégations

• Agrégation

- Un '**tout**' qui est une **agrégation** de plusieurs '**parties**'
- Une '**partie**' peut **participer** à plusieurs '**tout**'

• Composition

- Type particulier d'**agrégation**
- Les '**parties**' n'**existent** que dans un seul '**tout**'
- Si on supprime le '**tout**', les '**parties**' aussi sont **supprimées**

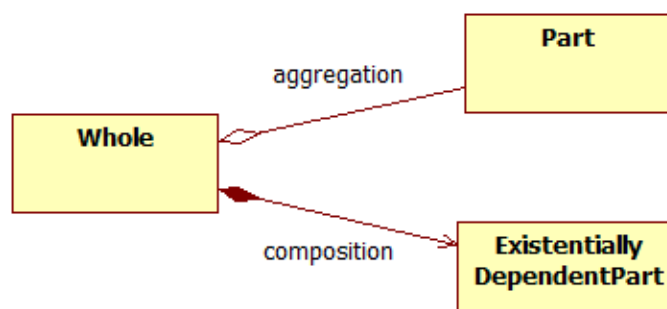


Diagramme de classes : agrégations

- **Pattern Composite**

- Ex.: Une arbre qui contient des branches, qui elles-aussi contiennent d'autres branches, jusqu'aux feuilles

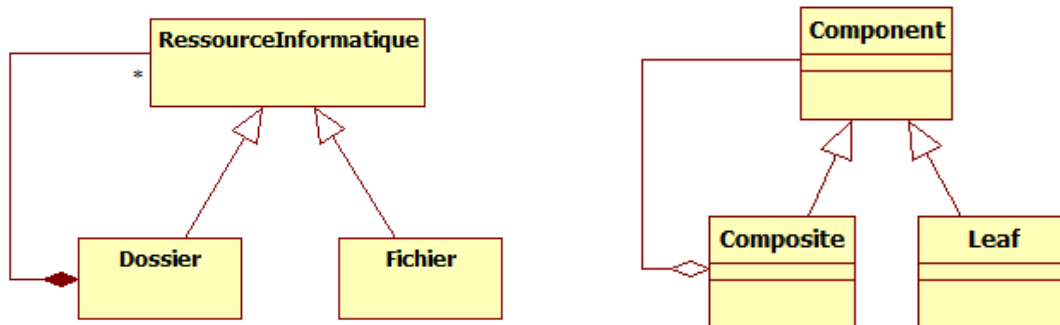


Diagramme de classes : dépendances

- **Dépendances**

- **Dépendance sémantique** entre éléments de la modélisation
- Un **changement** au niveau de la **cible** implique un **changement** au niveau de la **source**
- Les **stéréotypes** sont utilisés pour préciser la nature de la dépendance (« **call** », « **use** », « **instantiate** »...)

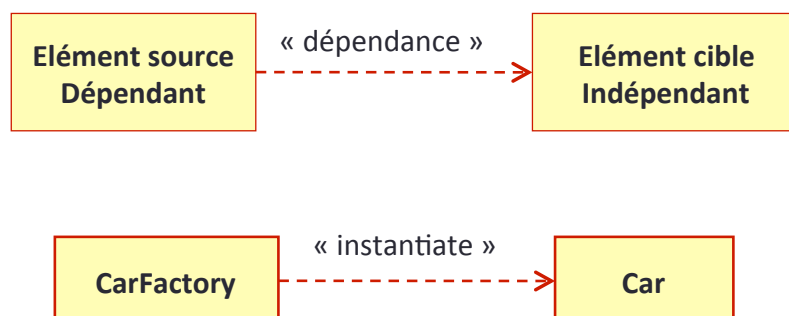


Diagramme de classes : contraintes

• Contraintes

- **Condition** qui doit être **vérifiée** par les **éléments** d'un modèle
- **Expression** qui **limite** la sémantique d'un **élément** et doit être **toujours respectée** afin de garantir la **validité** du **système** modélisé
- On peut associer des contraintes à n'importe quel élément du modèle
 - Attribut, méthodes, associations...

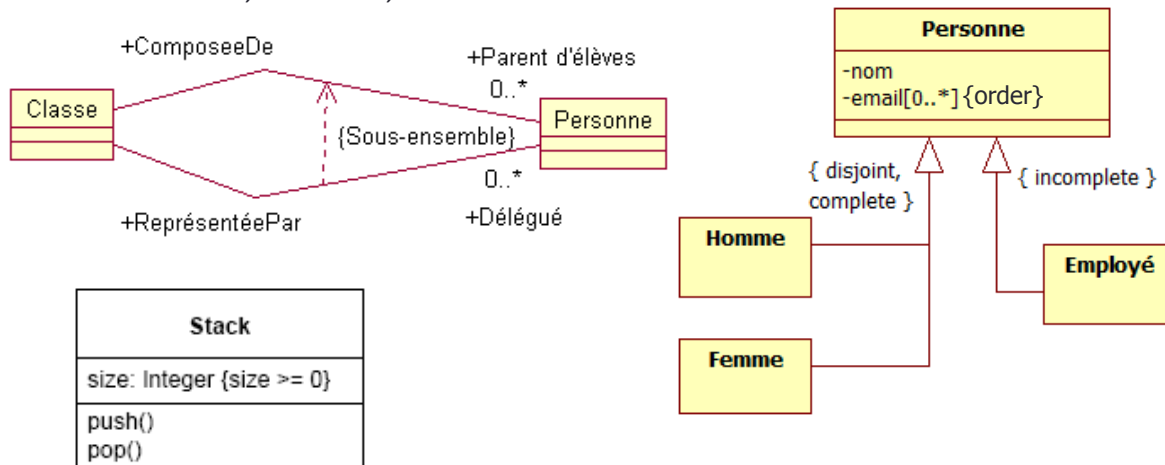


Diagramme de classes : contraintes

• Contraintes

- **Condition** qui doit être **vérifiée** par les **éléments** d'un modèle
- **Expression** qui **limite** la sémantique d'un **élément** et doit être **toujours respectée** afin de garantir la **validité** du **système** modélisé
- On peut associer des contraintes à n'importe quel élément du modèle
 - Attribut, méthodes, associations...

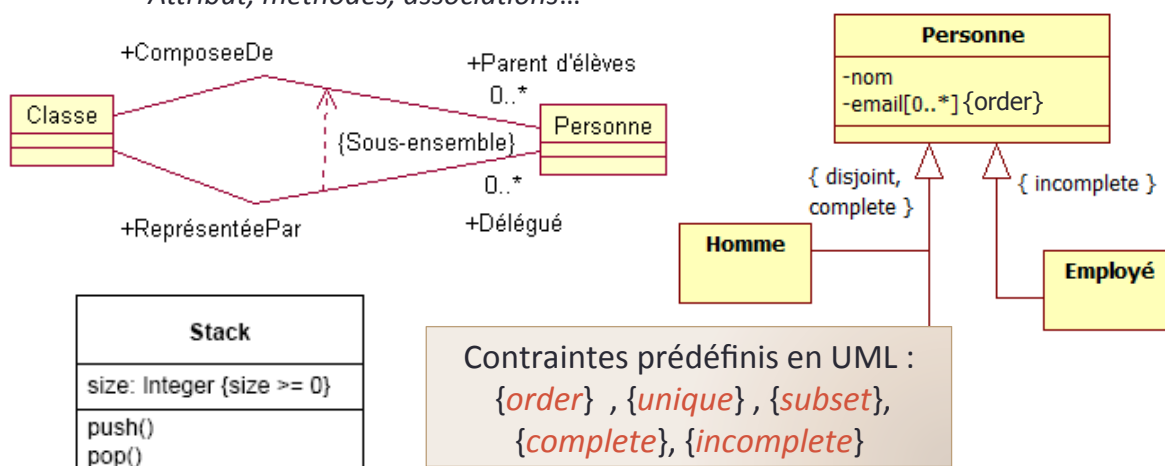


Diagramme de classes : contraintes

• Contraintes

- Les contraintes peuvent être écrites de manière plus ou moins formelle
 - Plus formelle : langage OCL
 - Moins formelle : langage naturel

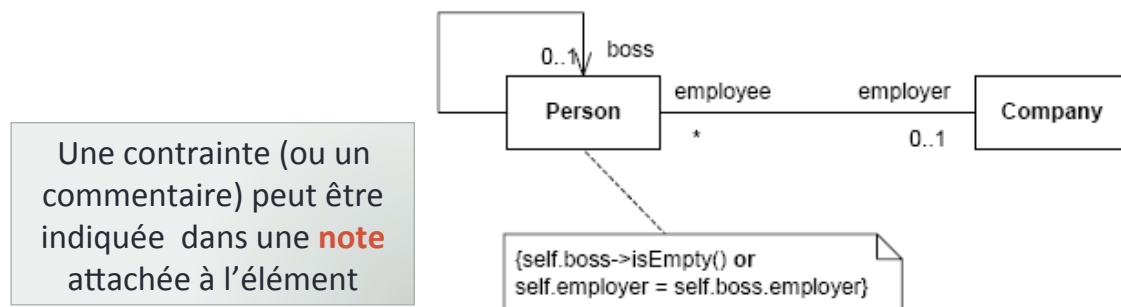


Diagramme de classes : interfaces

• Interface

- **Ensemble d'opérations** assurant un **service cohérent** offert par une (ou plusieurs) classe(s)
- Représentation d'un **comportement visible** à l'extérieur
- Une interface spécifie les opérations **sans en définir** la **structure interne**
- La **classe** réalisant l'interface fournit **l'implémentation** de **toutes les opérations**

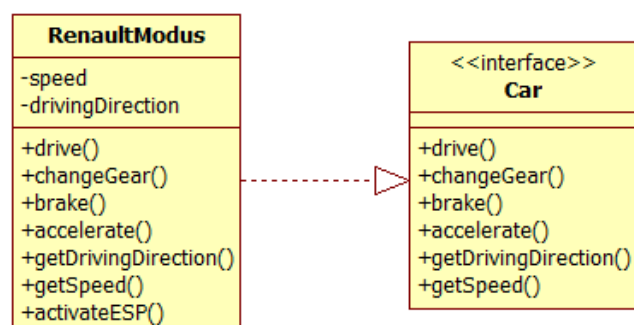
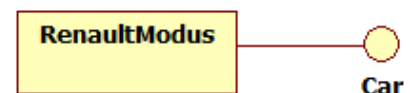


Diagramme de classes : interfaces

• Interface

- Une classe peut utiliser les services d'une interface
 - Dépendance « use »

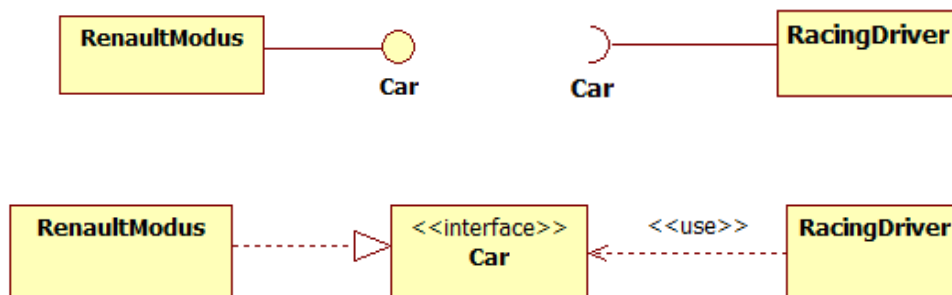


Diagramme de classes : paquetages

• Paquetage

- Mécanisme permettant le **regroupement** des éléments de modélisation
- Les paquetages permettent de...
 - Organiser les classes par **ensemble fonctionnel**
- Un paquetage définit **un espace de nommage**
 - Banque::Client
 - Commerce::Client

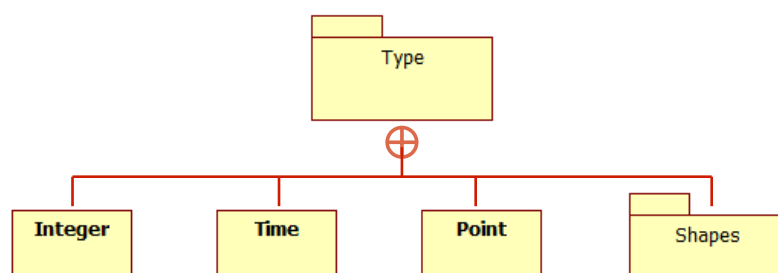
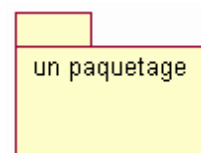


Diagramme de paquetages

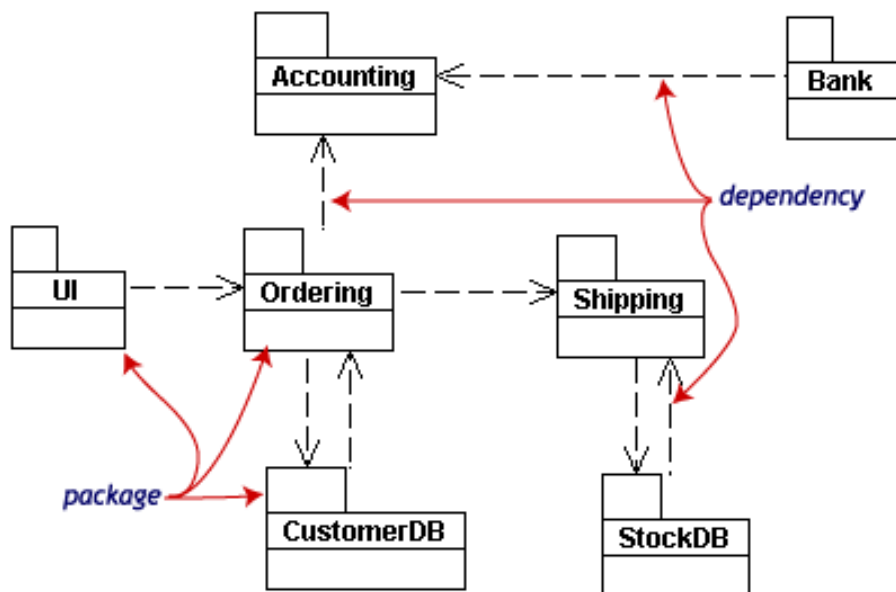
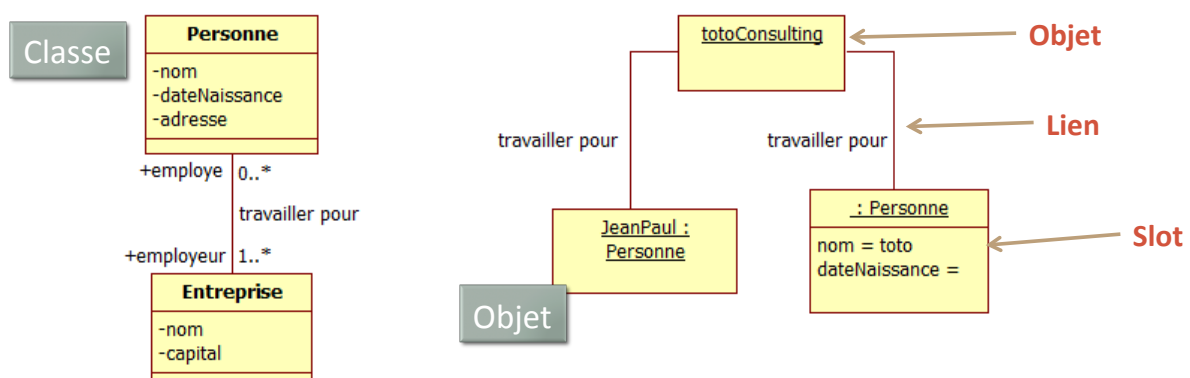


Diagramme objets

- Représentation des **instances** des **classes** et des **associations**
 - Représentation de l'**état** des objets
 - **Slot** : indication des **valeurs des attributs** à un instant t
 - **Snapshot du système** en modélisation
 - **Illustration** du diagramme de classes, **d'une situation précise**



MODÉLISATION DES APPLICATIONS

Manuele Kirsch Pinheiro

Maître de Conférences – Université Paris 1

mkirschpin@univ-paris1.fr / kirschpm@gmail.com

<http://mkirschp.free.fr>

70

Objectifs et Planning

- **Objectifs :**
 - Sensibiliser à la modélisation d'applications
 - Introduire / réviser le langage UML
 - Introduire le passage UML → code Java
- **Planning :**
 - 10h (CM / TP) en 4 séances
 - ✓ Séance 1 : Introduction à la modélisation
 - ✓ Séance 2 : Diagramme de classes UML
 - **Séance 3 : Diagramme de séquence UML**
 - Séance 4 : Passage UML → code
- **Evaluation**
 - Examen final

Les diagrammes d'UML

Diagrammes structurels vues statiques

- **Diagrammes de classes**
- **Diagrammes d'objets**
- Diagrammes de composants
- Diagrammes de déploiement
- **Diagrammes de paquetage**

Diagrammes comportementaux vues dynamiques

- Diagrammes de cas d'utilisation
- **Diagrammes de séquence**
- Diagrammes d'activités
- Diagrammes de communication (collaboration)
- Diagrammes d'états
- Timing diagram
- Interaction overview diagram

Diagramme de séquence

- Les **diagrammes de séquence** permettent la modélisation des **interactions** entre les instances
 - Illustration de **l'aspect temporel**
 - **L'échange des messages** entre les instances dans le temps
 - **Communication** entre participants
 - Échange de données, appel de méthodes

Le diagramme de classes ne dit pas comment les méthodes sont utilisées, dans quel ordre...

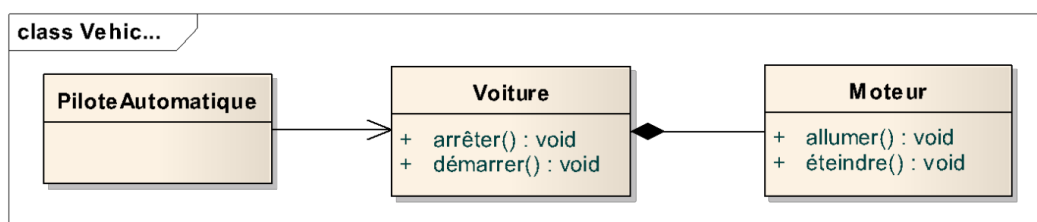


Diagramme de séquence

Le diagramme de séquence montre les interactions sous un angle temporel

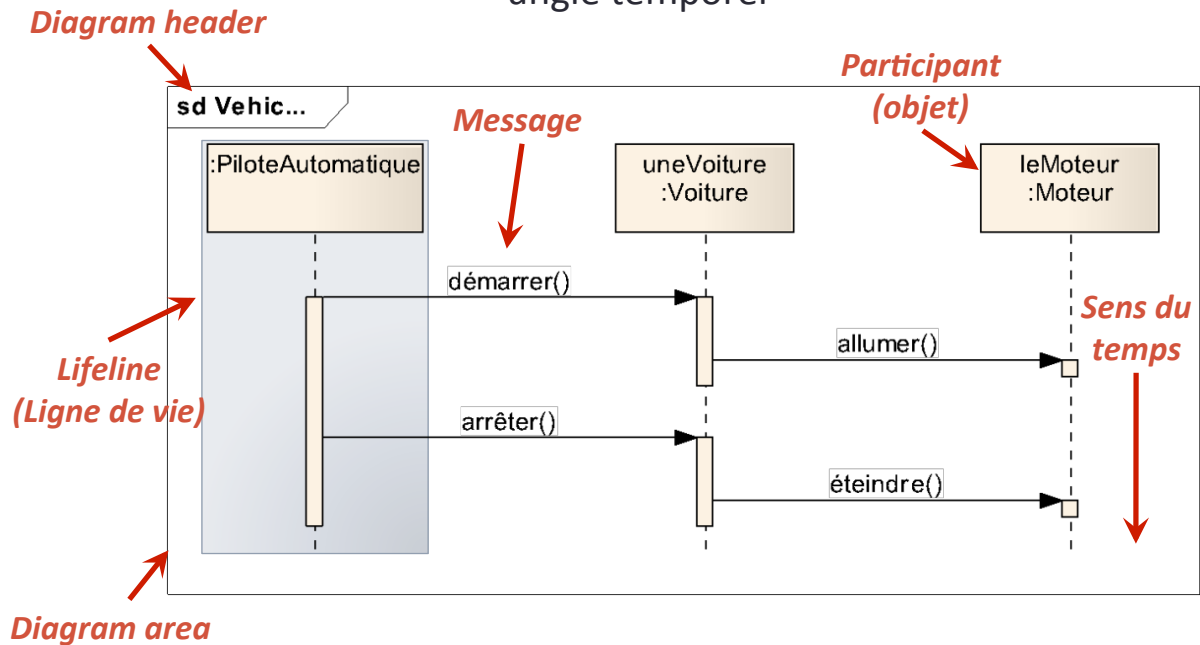


Diagramme de séquence

- Un SD illustre un **scénario d'exécution**

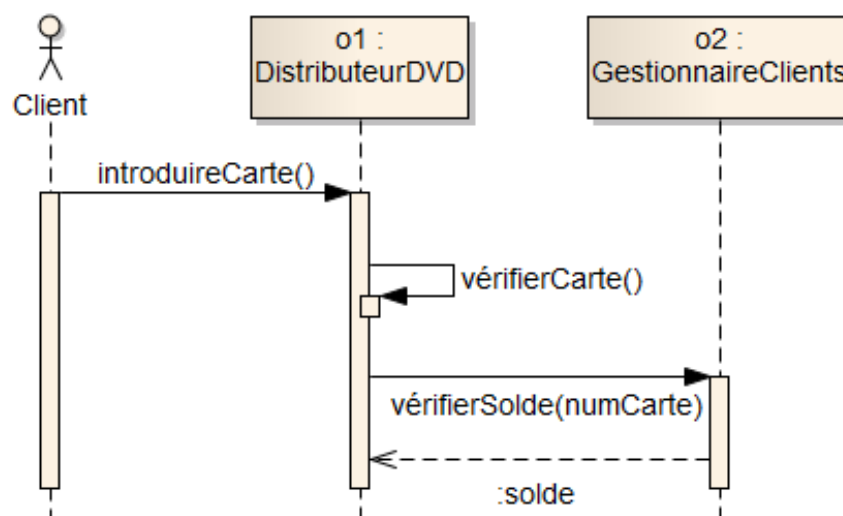


Diagramme de séquence

- Un SD illustre un **scénario d'exécution**

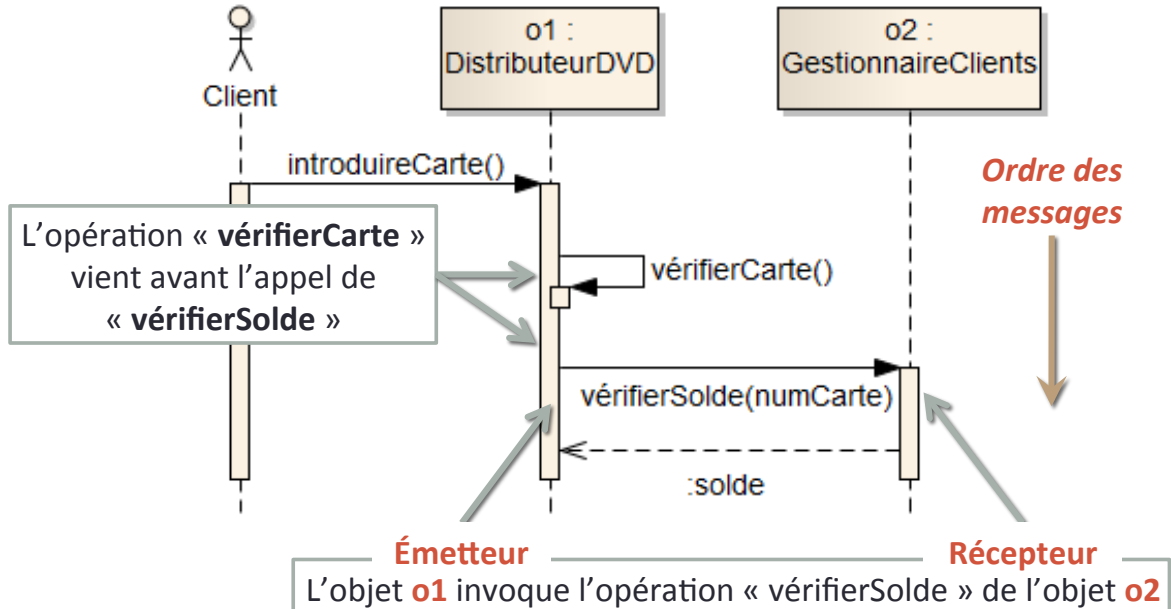


Diagramme de séquence

• Messages

• **Synchrone**

- L'émetteur reste bloqué le temps que dure l'invocation

• **Asynchrone**

- L'émetteur n'attend pas la fin de l'invocation, il ne reste pas bloqué

• **Retour (reply)**

- Un message peut donner lieu à un retour

• **Création**

- Création d'une nouvelle instance

• **Destruction**

- Destruction d'une instance

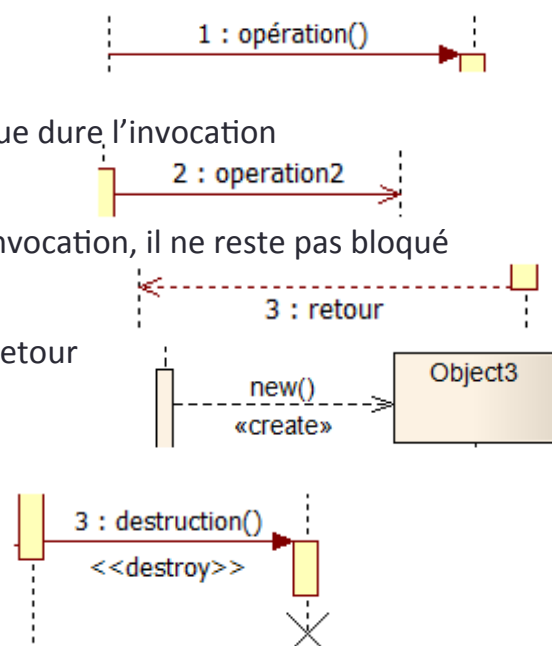


Diagramme de séquence

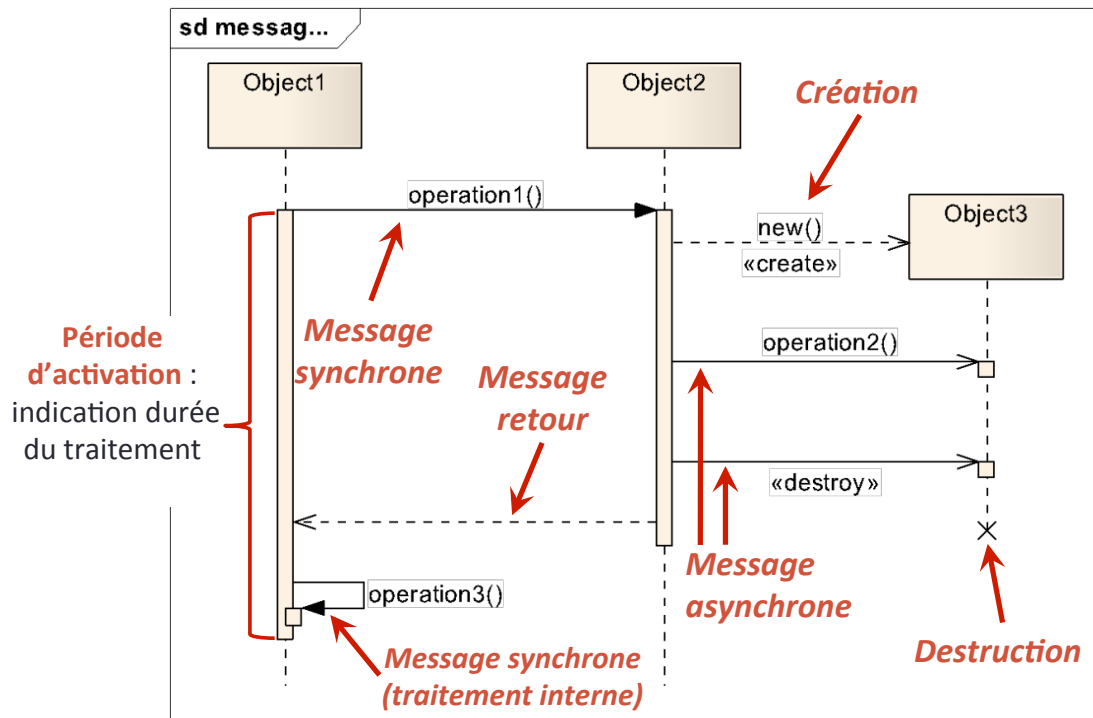


Diagramme de séquence

- **Messages asynchrones**

- L'ordre de réception des messages, dans un scénario précis, peut être indiqué

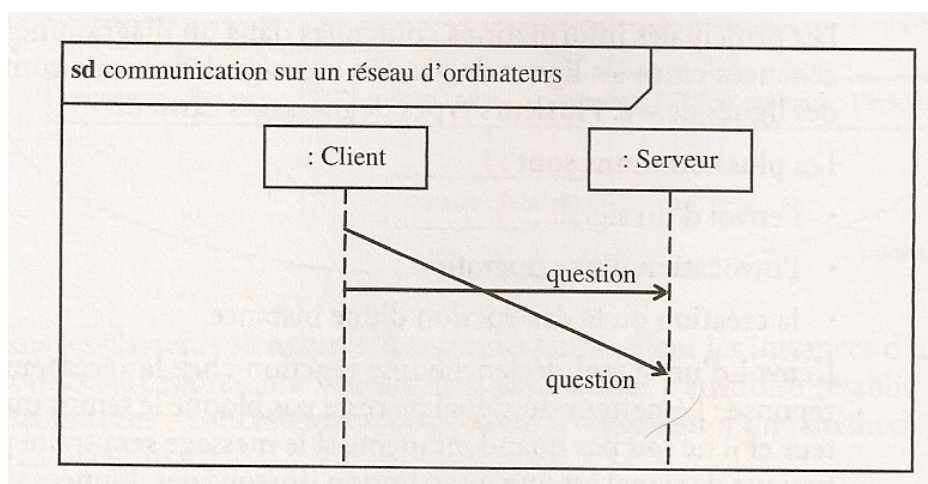


Diagramme de séquence

• Messages perdus

- UML permet d'indiquer la perte d'un message, ou encore la réception d'un message inattendu

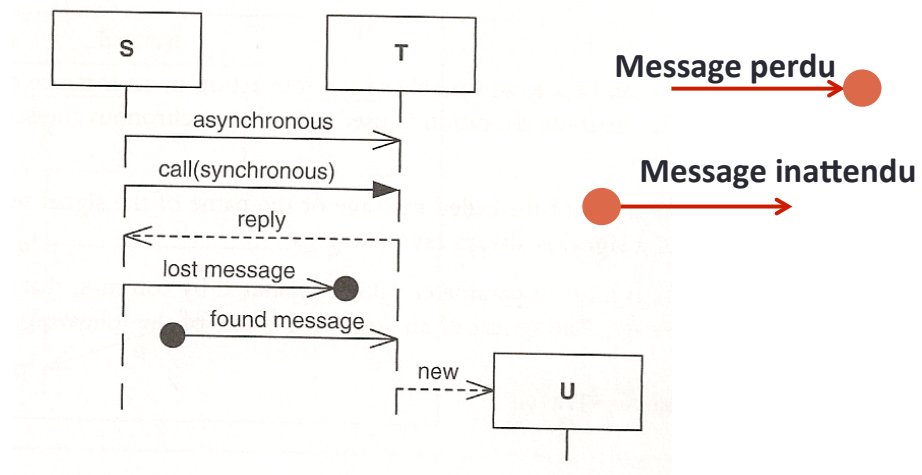
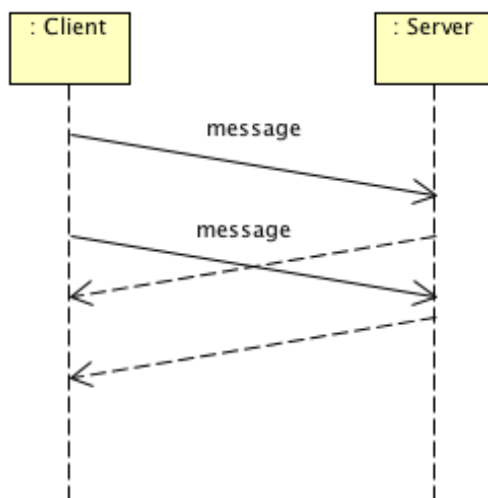


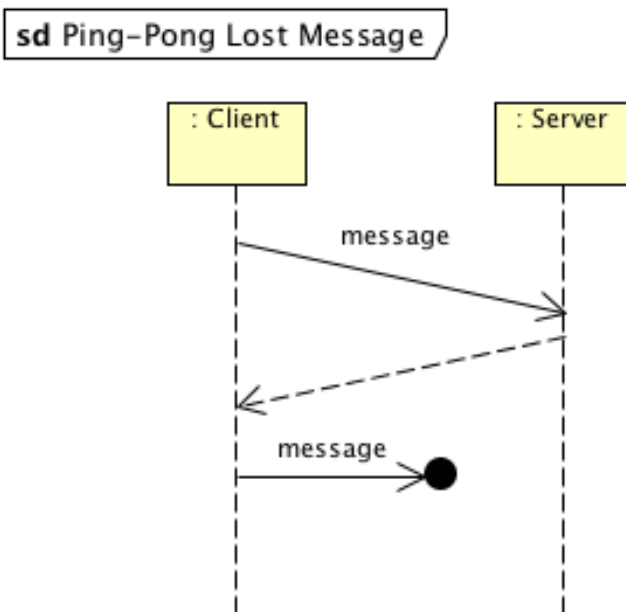
Diagramme de séquence

sd Ping Pong C/S



Les diagrammes de séquence sont utiles pour illustrer les protocoles de communication réseau.

Diagramme de séquence



Les diagrammes de séquence sont utiles pour illustrer les protocoles de communication réseau.

Diagramme de séquence

• Scénario :

- Un client ouvre une connexion avec un serveur
- Le serveur, lorsqu'il reçoit une demande d'ouverture de connexion de la part d'un client, va créer un nouveau *thread* qui s'occupera de cette connexion
- Le client envoie alors le message à travers la nouvelle connexion
- Le *thread* dédié reçoit le message et le traite. Il envoie ensuite la réponse au client, qui l'attend
- Le client ferme ensuite la connexion auprès du serveur, qui détruit le *thread*

Diagramme de séquence

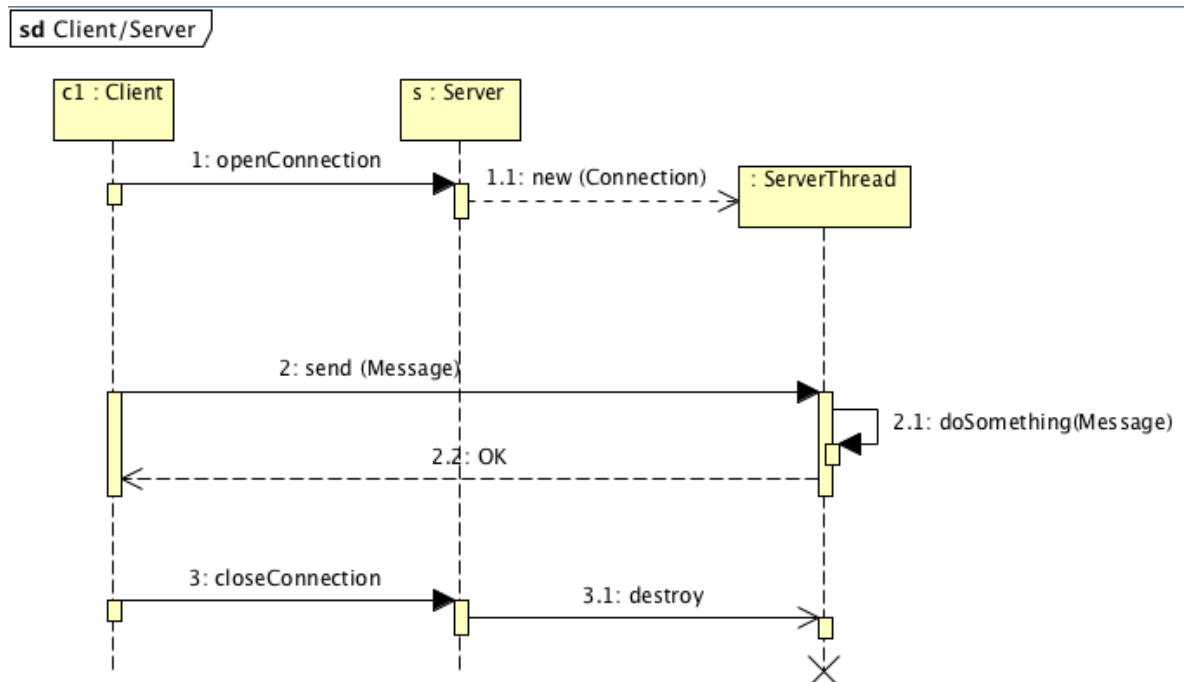
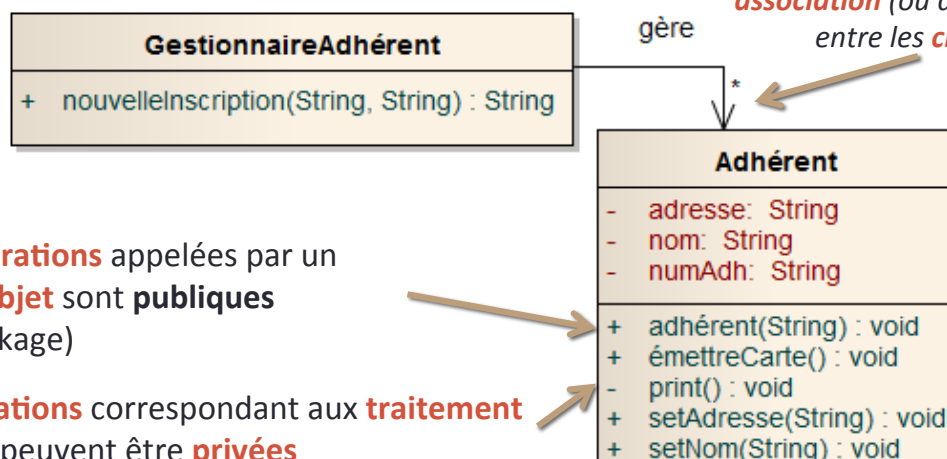


Diagramme de séquence

- Relation **diagramme de séquence** ↔ **diagramme de classe**
 - Les messages dans un diagramme de séquence correspondent aux opérations dans le diagramme de classes



Souvent un **message** entre 2 **objets** indique une **association** (ou dépendance) entre les **classes**

Les **opérations** appelées par un **autre objet** sont **publiques** (ou package)

Les **opérations** correspondant aux **traitement internes** peuvent être **privées**

Diagramme de séquence

• Fragment d'interaction

- **Regroupement de messages** à l'intérieur d'un diagramme de séquence
- Les fragments permettent de représenter une situation plus complexe à partir d'un opérateur
 - Fragments optionnels, alternatifs, parallèles, boucles...

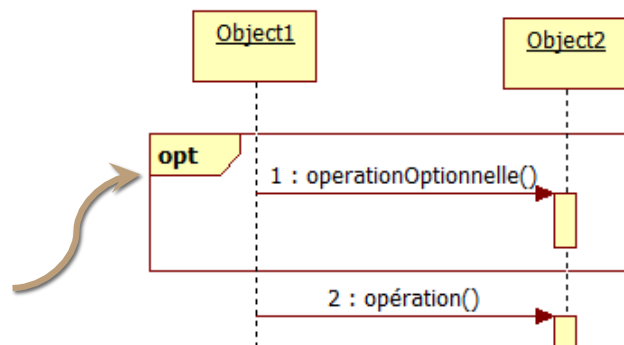


Diagramme de séquence

• Fragment d'interaction

- **Opt** indique qu'un groupe de messages est optionnel
- **Alt** indique des groupes de messages alternatifs (un choix)

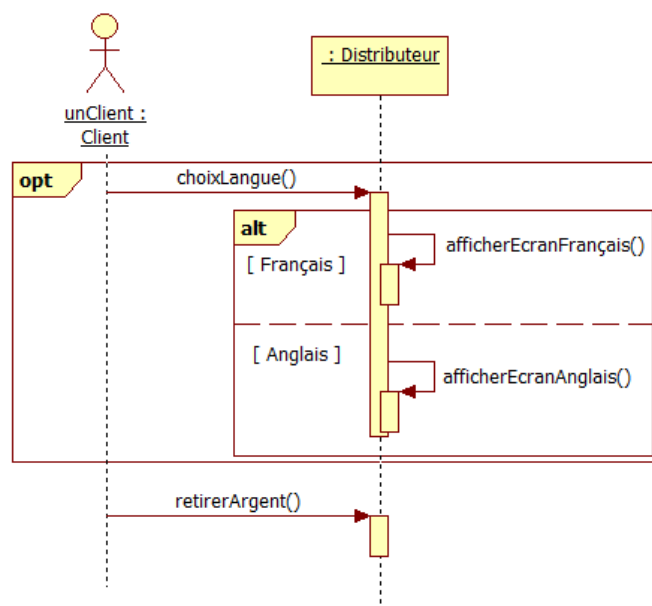


Diagramme de séquence

• Fragment d'interaction

- **Par** indique que l'envoi des messages se déroule en parallèle

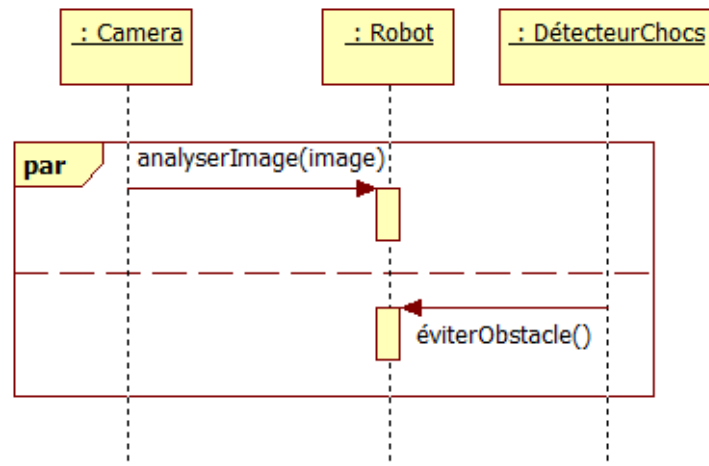


Diagramme de séquence

• Fragment d'interaction

- **loop (min, max)** : boucle se répète au moins min fois, jusqu'à max, ou tant que la condition est vraie

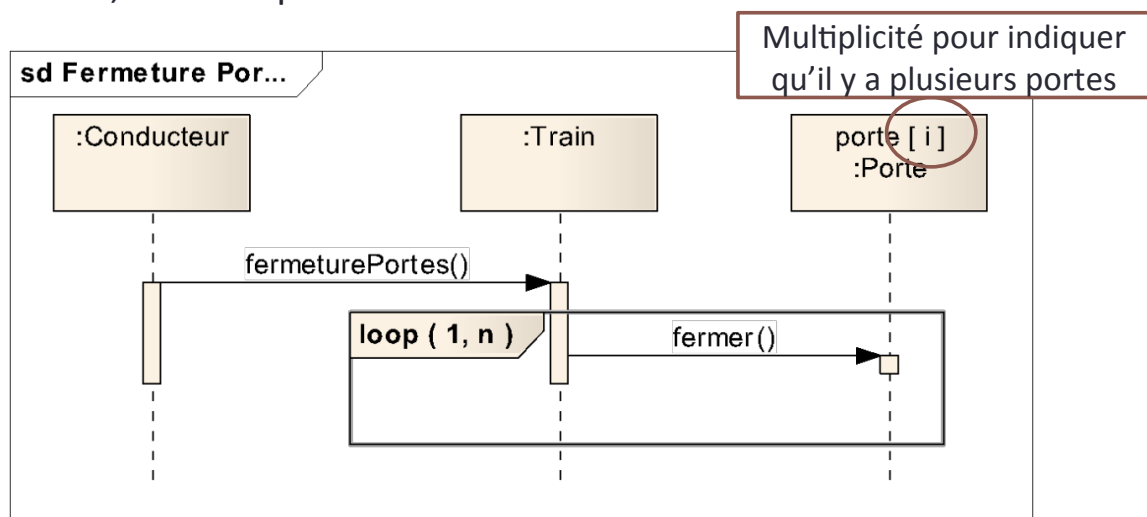


Diagramme de séquence

- **Fragment d'interaction**

- **Critical** indique que le fragment est atomique, qu'il doit être complètement traversé avant que de nouveaux messages soient acceptés
- **Ref** : lorsqu'une interaction est trop complexe, on peut la décomposer en plusieurs diagrammes

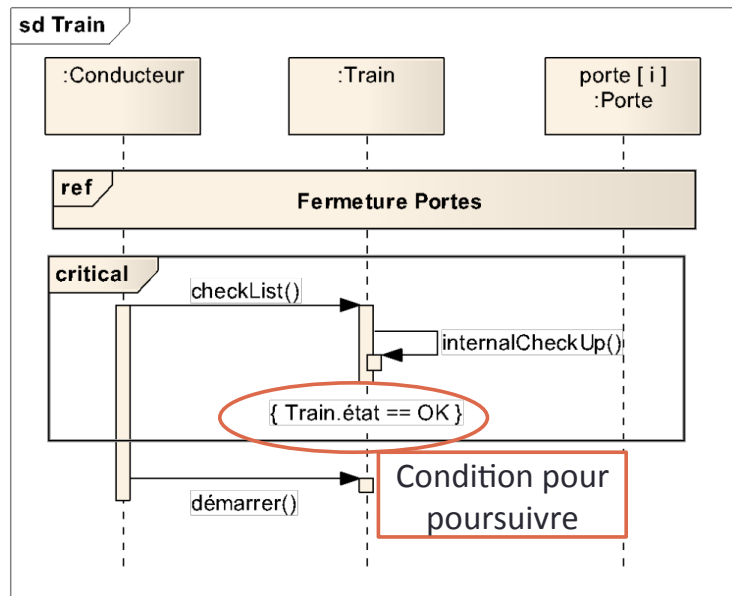
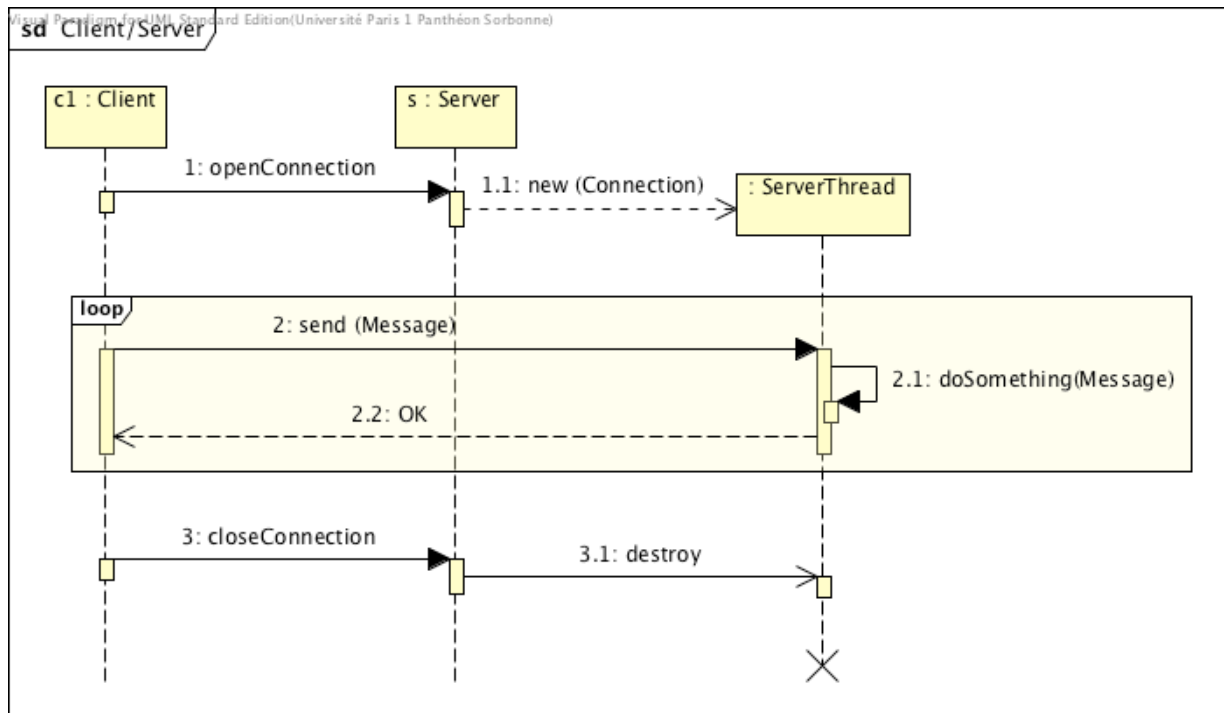


Diagramme de séquence

- **Scénario :**

- Un client ouvre une connexion avec un serveur
- Le serveur, lorsqu'il reçoit une demande d'ouverture de connexion de la part d'un client, va créer un nouveau *thread* qui s'occupera de cette connexion
- Le client envoie alors **une ou plusieurs messages** à travers la nouvelle connexion
- Le *thread* dédié reçoit **chaque message et le traite**. Il envoie ensuite la réponse au client, qui **attend à chaque message**
- Une fois **tous les messages envoyés**, le client ferme la connexion auprès du serveur, qui détruit le *thread*

Diagramme de séquence

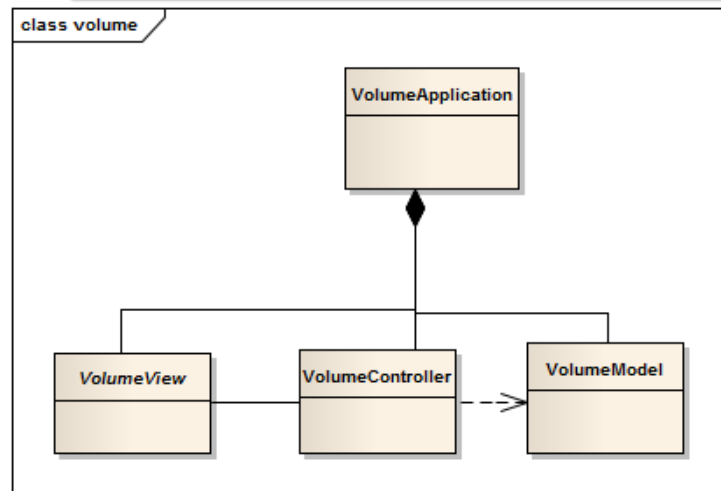


Interaction Diagramme de séquence - Diagramme de classes

- Les diagrammes de classes (CD) et de séquence (SD) sont étroitement liés
 - Les SD illustrent l'interaction entre objets d'une ou plusieurs classes
 - Les messages envoyés dans un SD correspondent à d'appels de méthodes que les classes doivent fournir
- Grâce aux diagrammes de séquence, on peut enrichir les diagrammes de classes
 - Méthodes, associations, dépendances oubliés

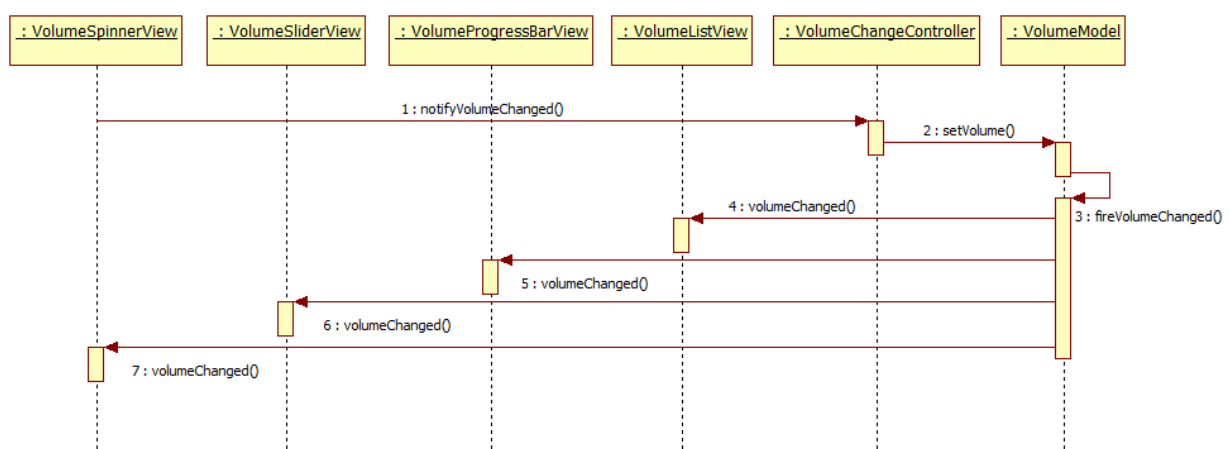
Interaction Diagramme de séquence - Diagramme de classes

Exemple : Application de gestion de volume



Interaction Diagramme de séquence - Diagramme de classes

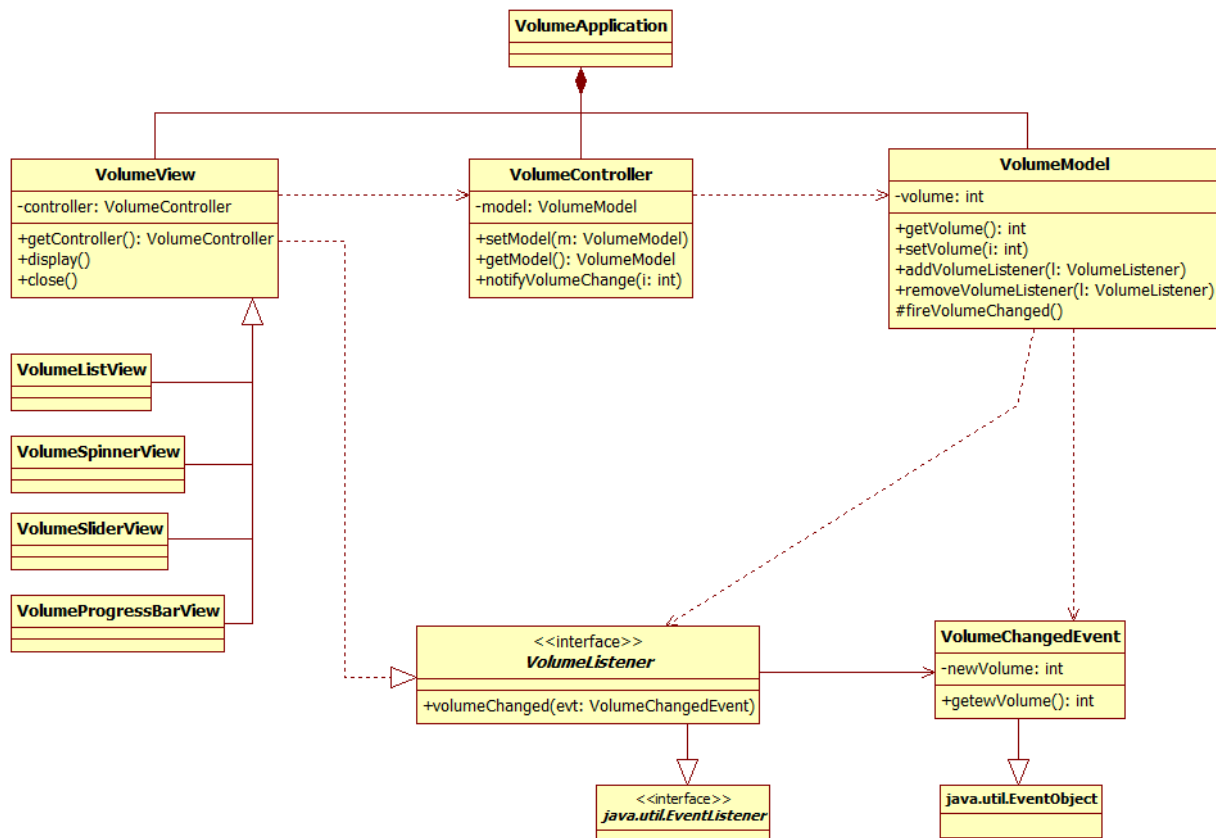
Exemple : Application de gestion de volume



Quelles classes participent à cette interaction ?

Quelles opérations proposent-elles ?

Avons-nous des interfaces communes à ces classes ?



MODÉLISATION DES APPLICATIONS

Manuele Kirsch Pinheiro

Maître de Conférences – Université Paris 1

mkirschpin@univ-paris1.fr / kirschpm@gmail.com

<http://mkirschp.free.fr>

Objectifs et Planning

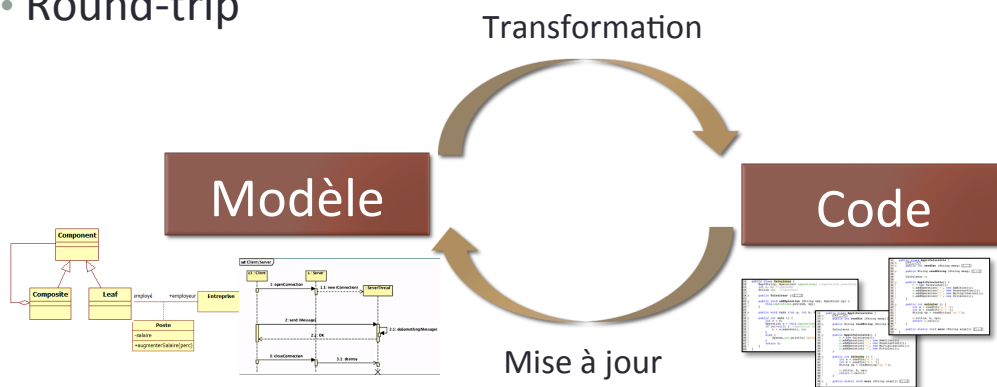
- Objectifs :
 - Sensibiliser à la modélisation d'applications
 - Introduire / réviser le langage UML
 - Introduire le passage UML → code Java
- Planning :
 - 10h (CM / TP) en 4 séances
 - Séance 1 : Introduction à la modélisation
 - Séance 2 : Diagramme de classes UML
 - Séance 3 : Diagramme de séquence UML
 - **Séance 4 : Passage UML → code**
- Evaluation
 - Examen final

Modélisation des applications

- Développement de qualité
 - **Penser qualité**
 - Modularité / réutilisation
 - Évolutivité / extensibilité
 - Robustesse
 - **Vision globale**
 - Solution à court terme X solution à long terme
- **Besoin de modélisation !**
- ✧ **Correspondance modèle ↔ code généré**

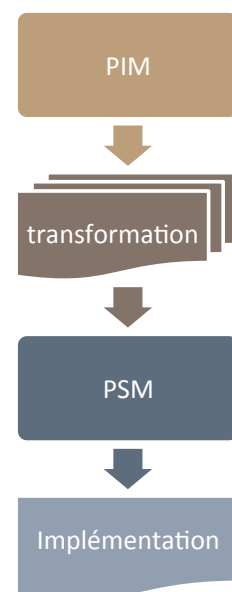
Modélisation des applications

- Correspondance modèle \leftrightarrow code
 - Bien modéliser ne suffit pas si le code ne correspond pas au modèle
 - **Traçabilité**
- Round-trip



UML & MDA

- **MDA (Model Driven Architecture)**
 - Démarche d'**ingénierie dirigée par les modèles (IDM)**
 - **Élaboration** de **modèles** successifs
 - **Transformation** d'un modèle d'un niveau d'abstraction supérieur vers un modèle moins abstrait
 - À chaque modèle, on rajoute des détails
 - Jusqu'à un modèle spécifique à une plateforme et à la génération du code (**implémentation**)



Passage UML → code Java

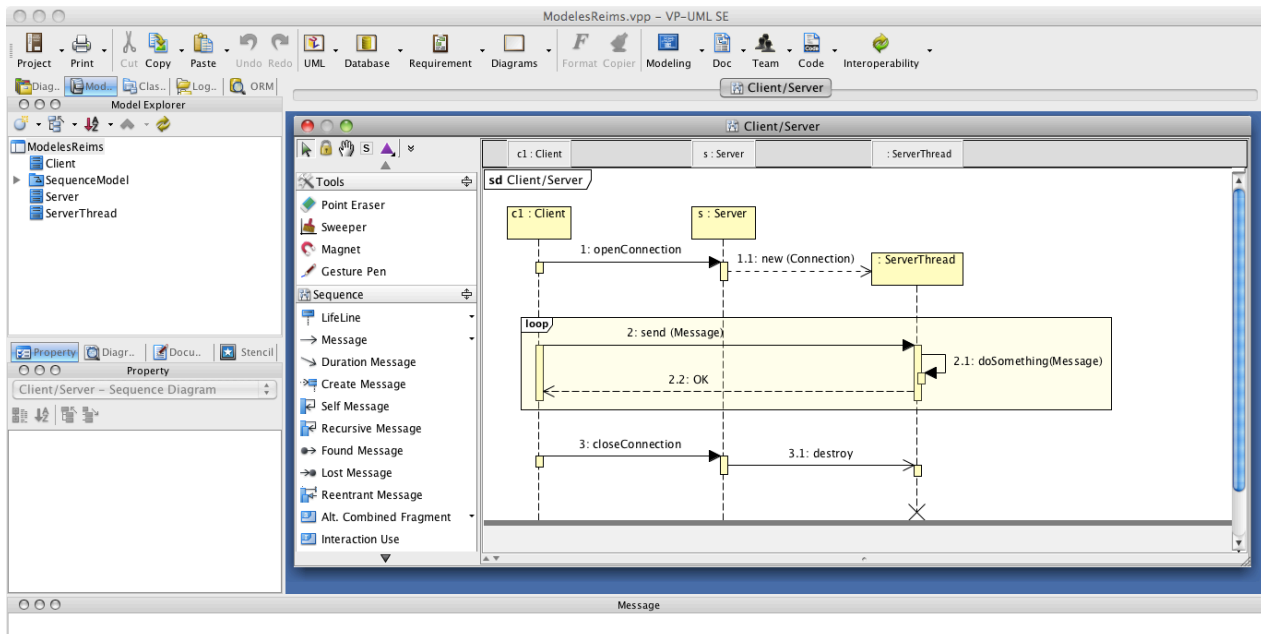
- Plusieurs applications proposent la génération automatique de code
 - Passage modèles UML → code Java, C++...
 - Souvent à partir du **diagramme de classes**
 - Intégré dans une **démarche de modélisation**
- Au-delà des applications, comment se traduit-il un diagramme de classes en code Java ??
 - Quelques « patterns » peuvent être observés

Applications génération de code

- Exemple d'application de génération de code
- **Visual Paradigm**
 - Outil de modélisation complet
 - Modélisation & gestion de projet
 - Plugins NetBeans et Eclipse disponibles

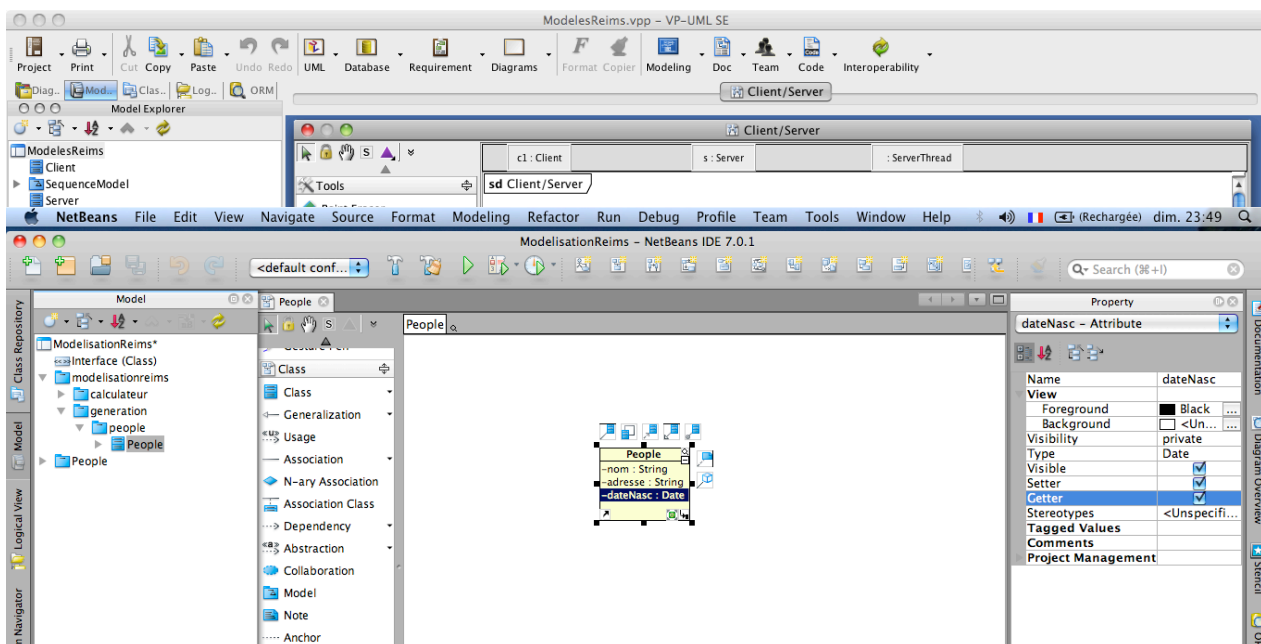
Applications génération de code

- Exemple d'application de génération de code



Applications génération de code

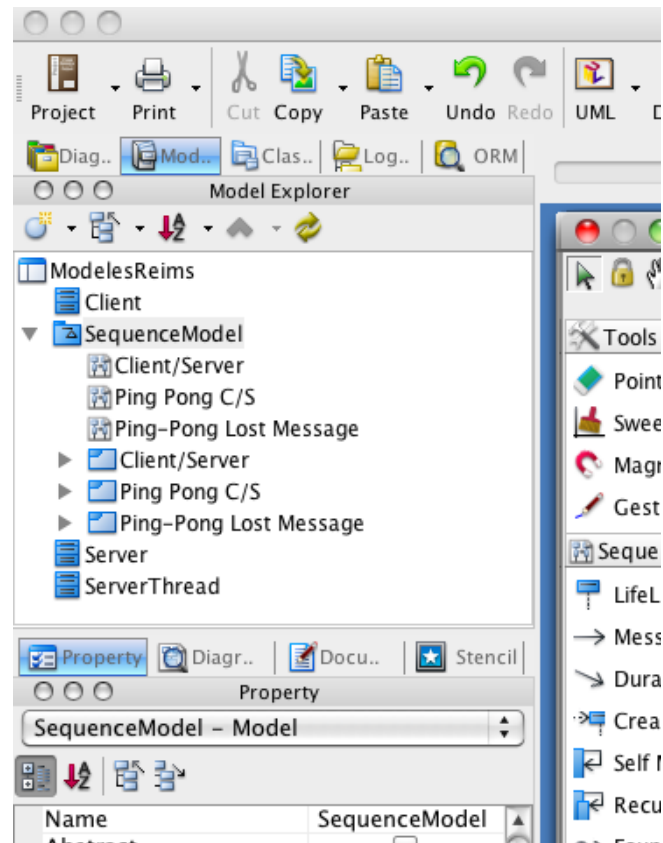
- Exemple d'application de génération de code



Applications

• Visual Paradigm

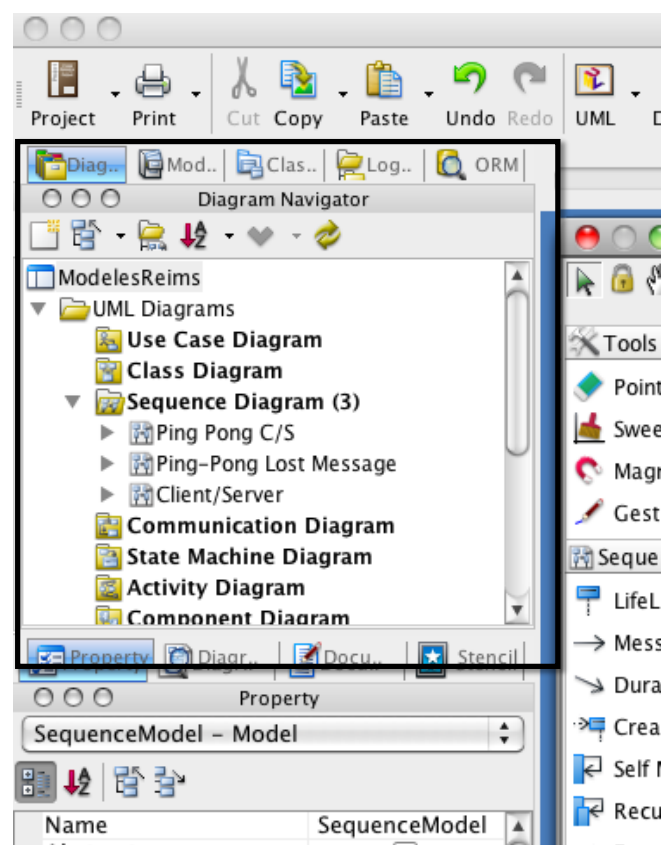
- Support complet à UML 2
- Vue par modèle ou par diagramme
- Plusieurs diagrammes disponibles



Applications

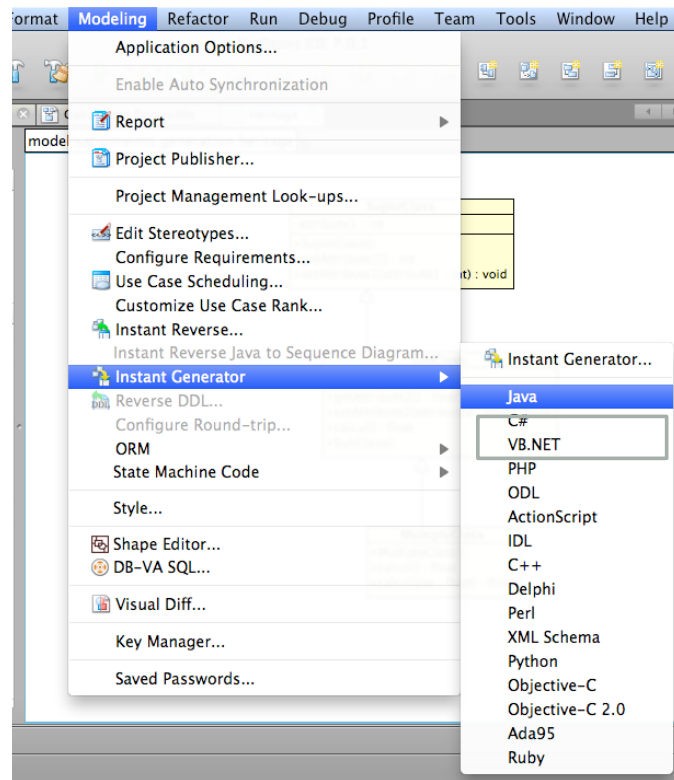
• Visual Paradigm

- Support complet à UML 2
- Vue par modèle ou par diagramme
- Plusieurs diagrammes disponibles



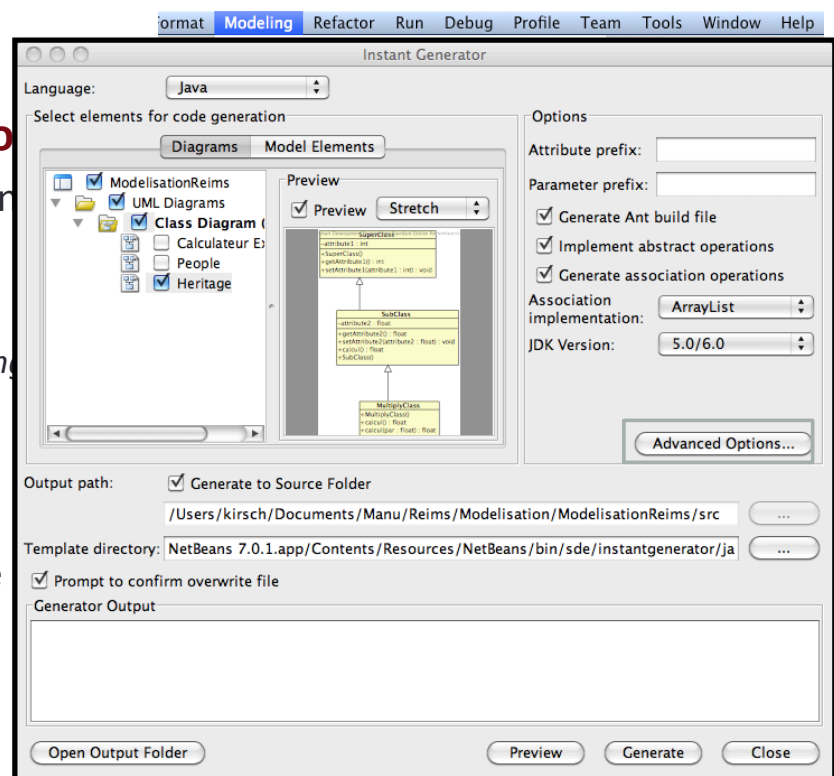
Applications

- **Génération de code**
 - Génération à un instant t
 - *Instant generator*
- **Rétroconception**
 - *Reverse engineering*
 - *Instant reverse*
- **Round-trip**
 - Java & C++
 - Version Entreprise
 - Java -> SD



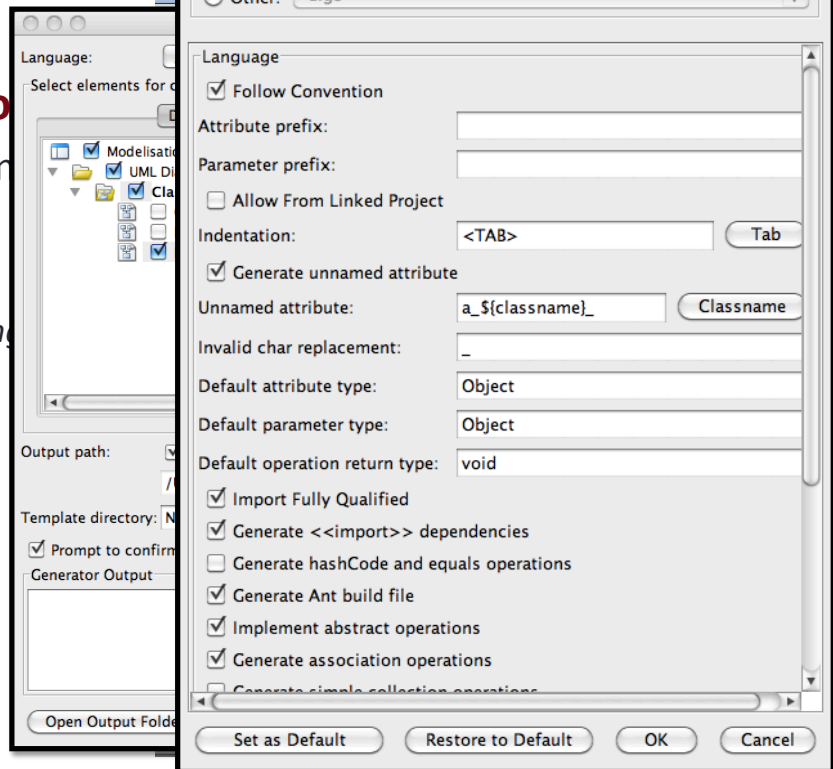
Applications

- **Génération de code**
 - Génération à un instant t
 - *Instant generator*
- **Rétroconception**
 - *Reverse engineering*
 - *Instant reverse*
- **Round-trip**
 - Java & C++
 - Version Entreprise
 - Java -> SD



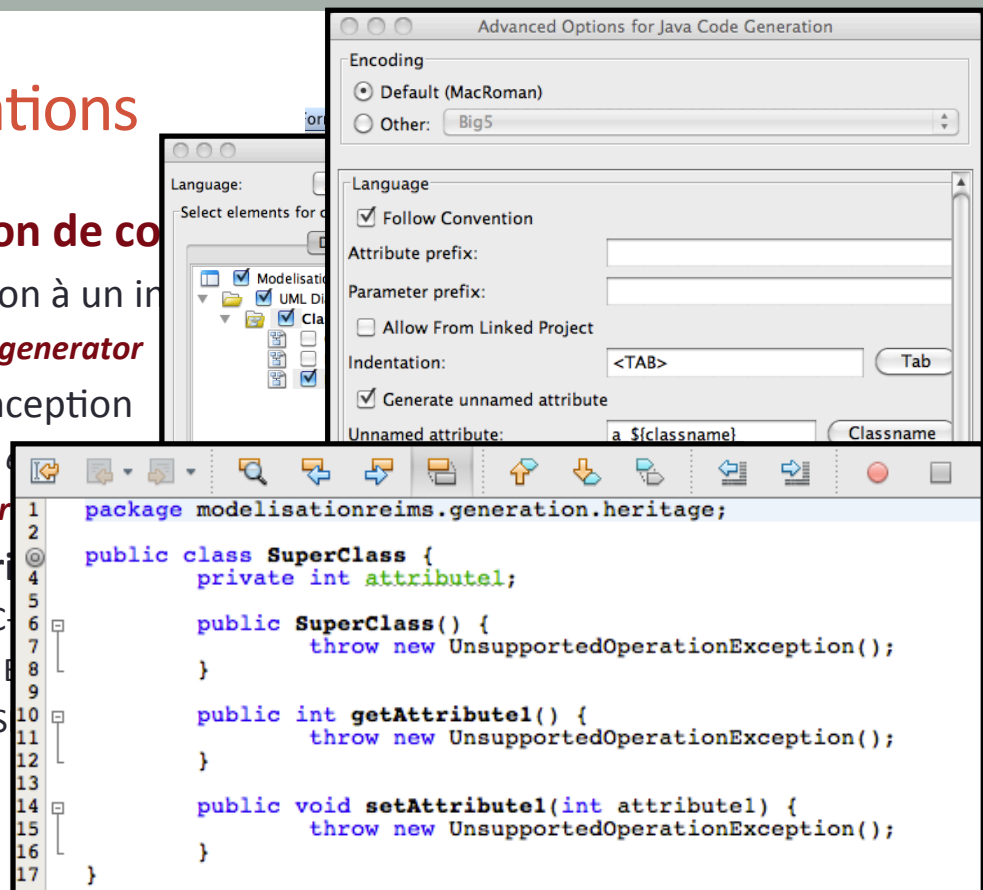
Applications

- **Génération de code**
 - Génération à un instantané
 - *Instant generator*
 - Rétroconception
 - *Reverse engineering*
 - *Instant reverse*
- **Round-trip**
 - Java & C++
 - Version Entreprise
 - Java -> SD



Applications

- **Génération de code**
 - Génération à un instantané
 - *Instant generator*
 - Rétroconception
 - *Reverse engineering*
 - *Instant reverse*
- **Round-trip**
 - Java & C++
 - Version Entreprise
 - Java -> SD

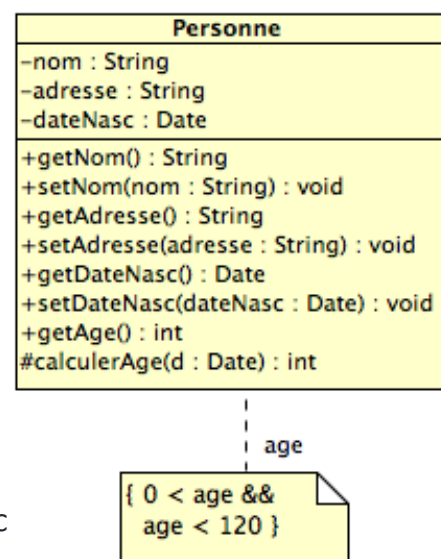


Applications génération de code

- Beaucoup d'autres applications de ce type sont disponibles
 - Yatta **UML-Lab**
 - Outil « round-trip » basé sur Eclipse
 - Uniquement diagramme de classes
 - Sparks Systems **Enterprise Architect**
 - Outil de modélisation complet
 - IBM **Rational Rose**
 - Borland **Together**
 - **Poseidon**
 - ...
- Applications souvent payantes

Passage UML → code Java

- **Classe**
 - Correspondance directe avec les *class* Java
 - **Attributs et méthodes**
 - Attributs dérivés
 - **Multiplicité des attributs** : **collections** ou *arrays*
 - **Visibilité** :
 - Package (~) valeur par défaut en Java
 - private (-), public (+), protected (#)
- **Bonnes pratiques**
 - Getter / Setter pour chaque attribut
 - Documentation à l'aide de commentaires Javadoc
- Attention au respect des **contraintes** !!



```

1 package modelisationreims.generation.people;
2
3 import java.util.Calendar;
4 import java.util.Date;
5
6 /**...*/
7 public class Personne {
8
9     private String nom;
10    private String adresse;
11    private Date dateNasc;
12
13    /**...*/
14    public String getNom() {
15        return this.nom;
16    }
17
18    /**...*/
19    public void setNom(String nom) {
20        this.nom = nom;
21    }
22
23    /**...*/
24    public String getAdresse() {
25        return this.adresse;
26    }
27
28    /**...*/
29    public void setAdresse(String adresse) {...}
30
31    /**...*/
32    public Date getDateNasc() {...}
33
34    /**...*/
35    public void setDateNasc(Date dateNasc) {...}
36
37    /**...*/
38    public int getAge() {

```

Personne
-nom : String -adresse : String -dateNasc : Date
+getNom() : String +setNom(nom : String) : void +getAdresse() : String +setAdresse(adresse : String) : void +getDateNasc() : Date +setDateNasc(dateNasc : Date) : void +getAge() : int #calculerAge(d : Date) : int

```

39 package modelisationreims.generation.people;
40
41 /**...*/
42 public void setAdresse(String adresse) {
43     this.adresse = adresse;
44 }
45
46 /**...*/
47 public Date getDateNasc() {
48     return this.dateNasc;
49 }
50
51 /**...*/
52 public void setDateNasc(Date dateNasc) {...}
53
54 /**
55  * La méthode <i>getAge</i> calcule l'âge de l'individu à partir de sa
56  * date de naissance.
57  * @return nombre entier représentant l'âge de l'individu
58  */
59 public int getAge() {
60     return this.calculerAge(this.dateNasc);
61 }
62
63 /**
64  * La méthode <i>calculerAge</i> calcule l'âge à partir d'une date d
65  * arbitraire.
66  * @param d date à partir de laquelle on calcule l'âge
67  * @return nombre entier représentant l'âge
68  */
69 protected int calculerAge(Date d) {
70     Calendar today = Calendar.getInstance();
71     Calendar birth = Calendar.getInstance();
72     birth.setTime(d);
73     int y = birth.get(Calendar.YEAR) - today.get(Calendar.YEAR);
74     if ((birth.get(Calendar.MONTH) - today.get(Calendar.MONTH)) > 0) //pas e
75     {
76         y--;
77     }
78 }

```

Javadoc

```

39 // classe modèle contenant les données personnelles
40 /**...*/
44 public void setAdresse(String adresse) {
45     this.adresse = adresse;
46 }
47
48 /**...*/
52 public Date getDateNasc() {
53     return this.dateNasc;
54 }
55
56 /**...*/
62 public void setDateNasc(Date dateNasc) {...}
66
67 /**
68  * La méthode <i>getAge</i> calcule l'âge de l'i
69  * date de naissance.
70  * @return nombre entier représentant l'âge de l
71  */
72 public int getAge() {
73     return this.calculerAge(this.dateNasc);
74 }
75
76 /**
77  * <p>La méthode <i>setDateNasc</i> permet de mettre à jour la date de naissanc
78  * d'un individu.</p>
79  * <p><b>Important: </b> l'âge de l'individu doit être comprise entre 0 et 120
80  * @param dateNasc
81  */
82 public void setDateNasc(Date dateNasc) {
83     //Traitement de la contrainte : { 0 < age && age < 120 }
84     int age = this.calculerAge(dateNasc);
85     if (0 < age && age < 120) { //contrainte OK, valeur repris
86         this.dateNasc = dateNasc;
87     }
88 }
89
90 }

```

Personne	
-nom : String	
-adresse : String	
-dateNasc : Date	
+getNom() : String	
+setNom(nom : String) : void	
+getAdresse() : String	
+setAdresse(adresse : String) : void	
+getDateNasc() : Date	
+setDateNasc(dateNasc : Date) : void	
+getAge() : int	
#calculerAge(d : Date) : int	

age

{ 0 < age && age < 120 }

Passage UML → code Java

- **Classe : héritage**
 - Mot-clé **extends**
 - Pas d'*héritage multiple* !!
- Attention à la **visibilité**
 - La sous-classe hérite tous les attributs et les méthodes
 - L'accès se limite aux attributs/méthodes **public** et **protected**
 - Pas d'accès aux attributs **private**
- **Polymorphisme**
 - Usage de **super** et **this**
 - Annotations **@Override**

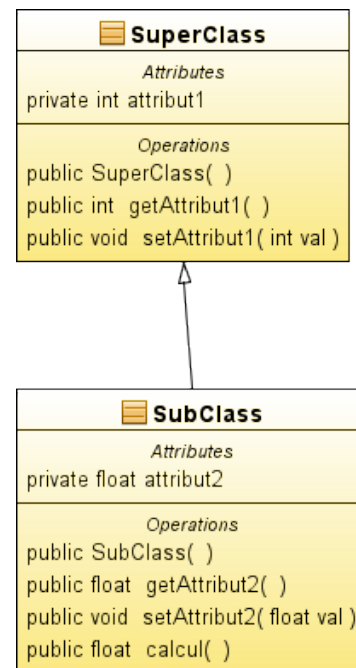
Passage UML → code Java

• Classe : héritage

```

3  public class SuperClass {
4
5      private int attribut1;
6
7      + public SuperClass () {...}
8
9
10     - public int getAttribut1 () {
11         return attribut1;
12     }
13
14     - public void setAttribut1 (int val) {
15         this.attribut1 = val;
16     }
17
18 }

```



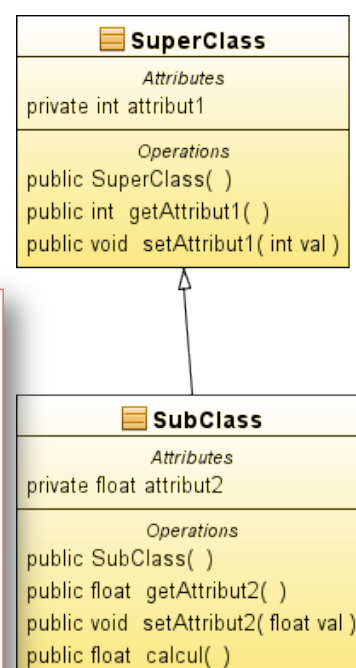
Passage UML → code Java

• Classe : héritage

```

3  public class SuperClass {
4
5      private int attribut1;
6
7      + public SuperClass () {...}
8
9
10     - public int getAttribut1 () {
11         return attribut1;
12     }
13
14     - public void setAttribut1 (int val) {
15         this.attribut1 = val;
16     }
17
18 }
19
20 public class SubClass extends SuperClass {
21
22     private float attribut2;
23
24     - public SubClass () {
25     }
26
27     + public float getAttribut2 () {...}
28
29     + public void setAttribut2 (float val) {...}
30
31     - public float calcul () {
32         return this.attribut2 + (float)this.getAttribut1();
33     }
34 }

```



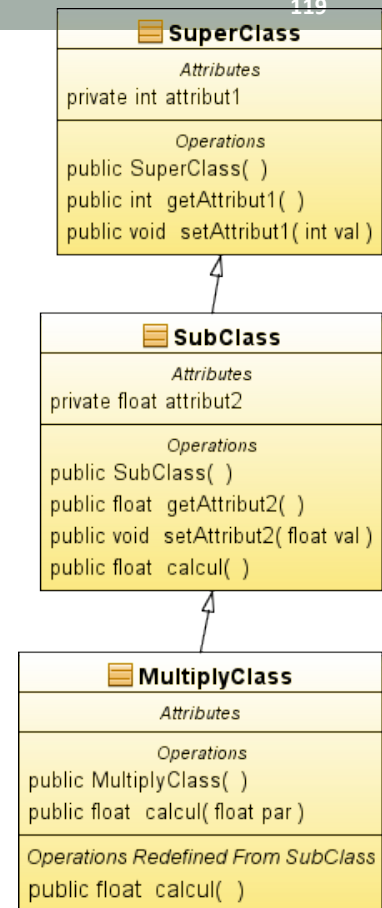
Passage UML → code Java

• Classe : héritage

- Exemple de **redéfinition & surcharge** : classe **MultiplyClass**
 - Redéfinition** : méthode **calcul ()**
 - Surcharge** : méthode **calcul(float)**

➤ Polimorphisme

- float c = **super**.calcul();
- float c = **this**.calcul();



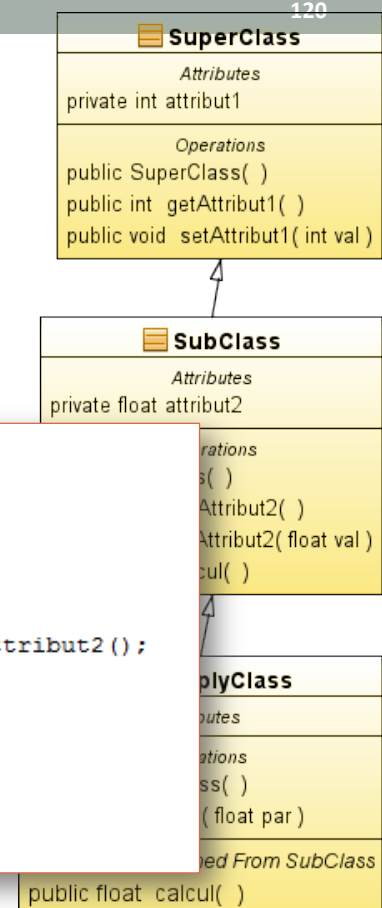
Passage UML → code Java

• Classe : héritage

- Exemple de **redéfinition & surcharge** : classe **MultiplyClass**
 - Redéfinition** : méthode **calcul ()**

```

3  public class MultiplyClass extends SubClass {
4
5  +   public MultiplyClass () { ... }
7
8  @Override
9  -   public float calcul () {
10     return super.getAttribut1() * super.getAttribut2();
11   }
12
13  -   public float calcul (float par) {
14     return par * super.calcul();
15   }
16 }
  
```



Passage UML → code Java

• Interface

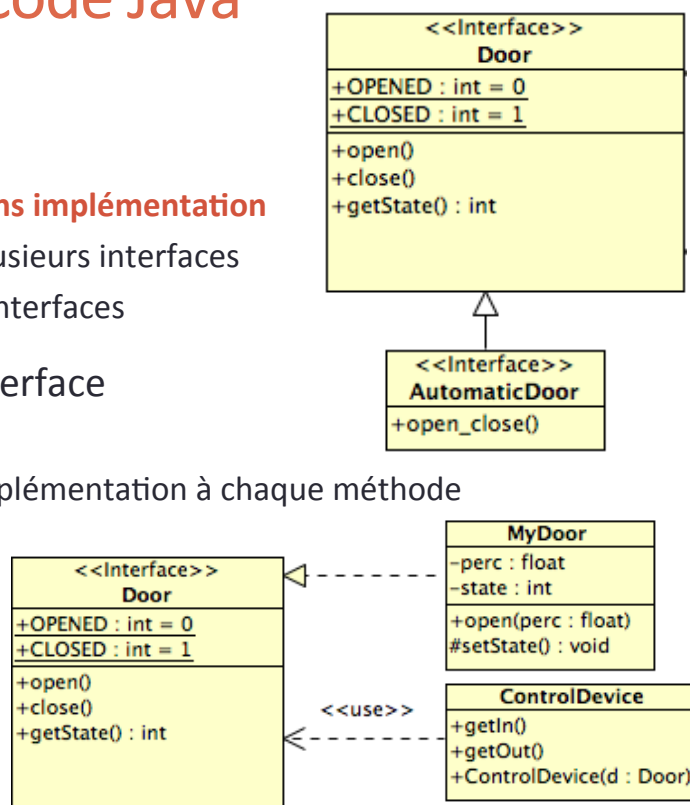
- Mot-clé **interface**
- Définition des **méthodes**, **sans implémentation**
- Possibilité d'implémenter plusieurs interfaces
- Possibilité d'**héritage** entre interfaces

• Implémentation d'une interface

- Mot-clé **implements**
- Obligation de fournir une implémentation à chaque méthode

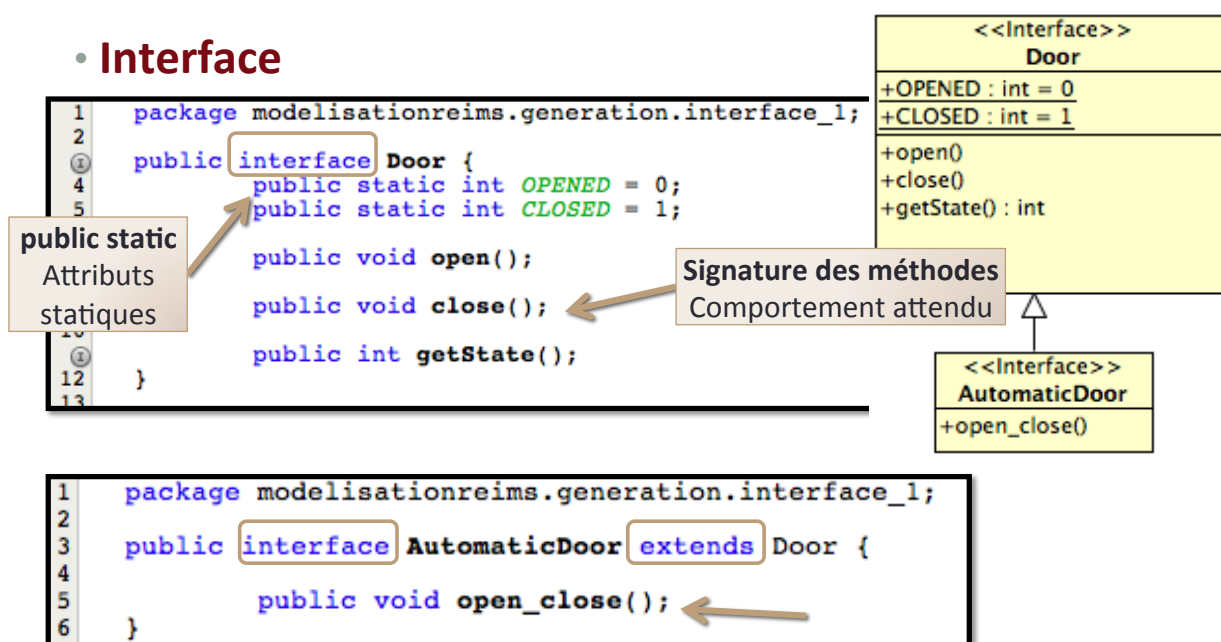
• Usage

- Faible couplage
- Stéréotype « use »

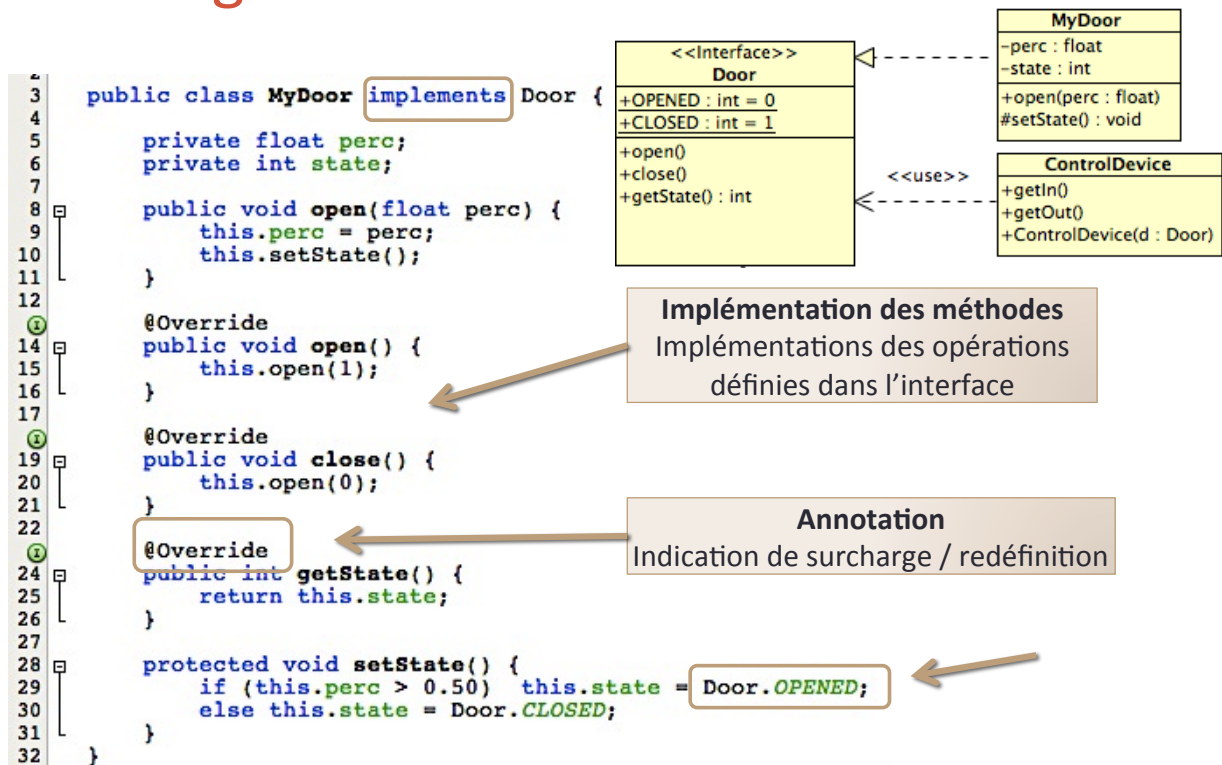


Passage UML → code Java

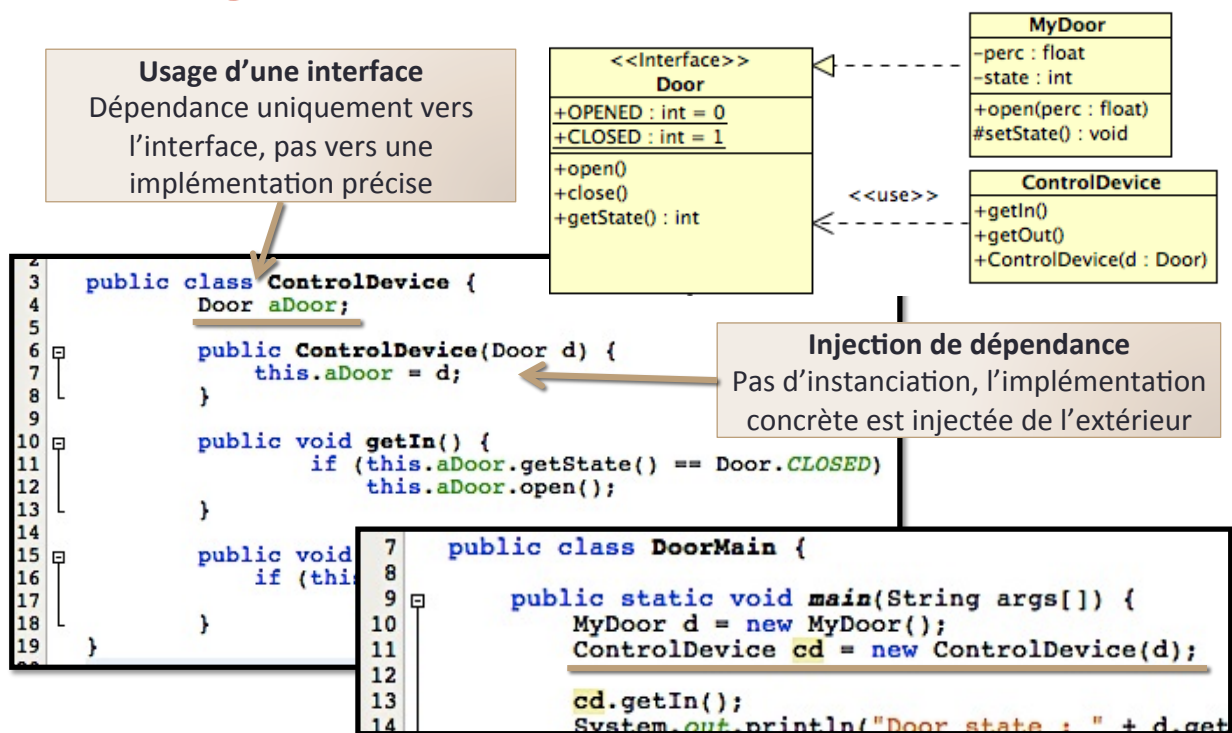
• Interface



Passage UML → code Java



Passage UML → code Java

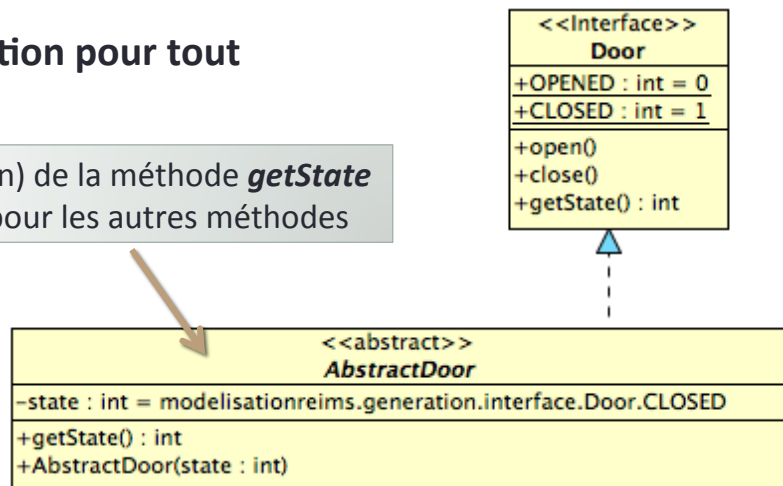


Passage UML → code Java

• Classes abstraites

- Stéréotype « **abstract** » (optionnel)
- Caractéristiques (attributs/méthodes) communes aux sous-classes
- Pas d'implémentation pour tout

Définition (implémentation) de la méthode **getState**
Pas d'implémentation pour les autres méthodes



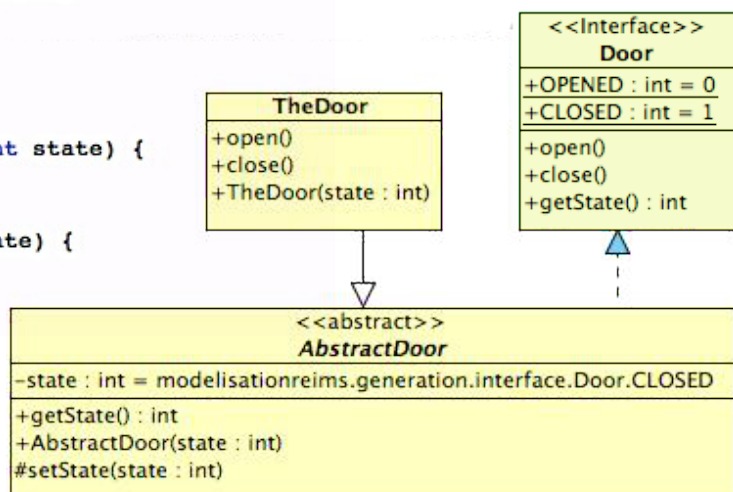
Passage UML → code Java

• Classes abstraites

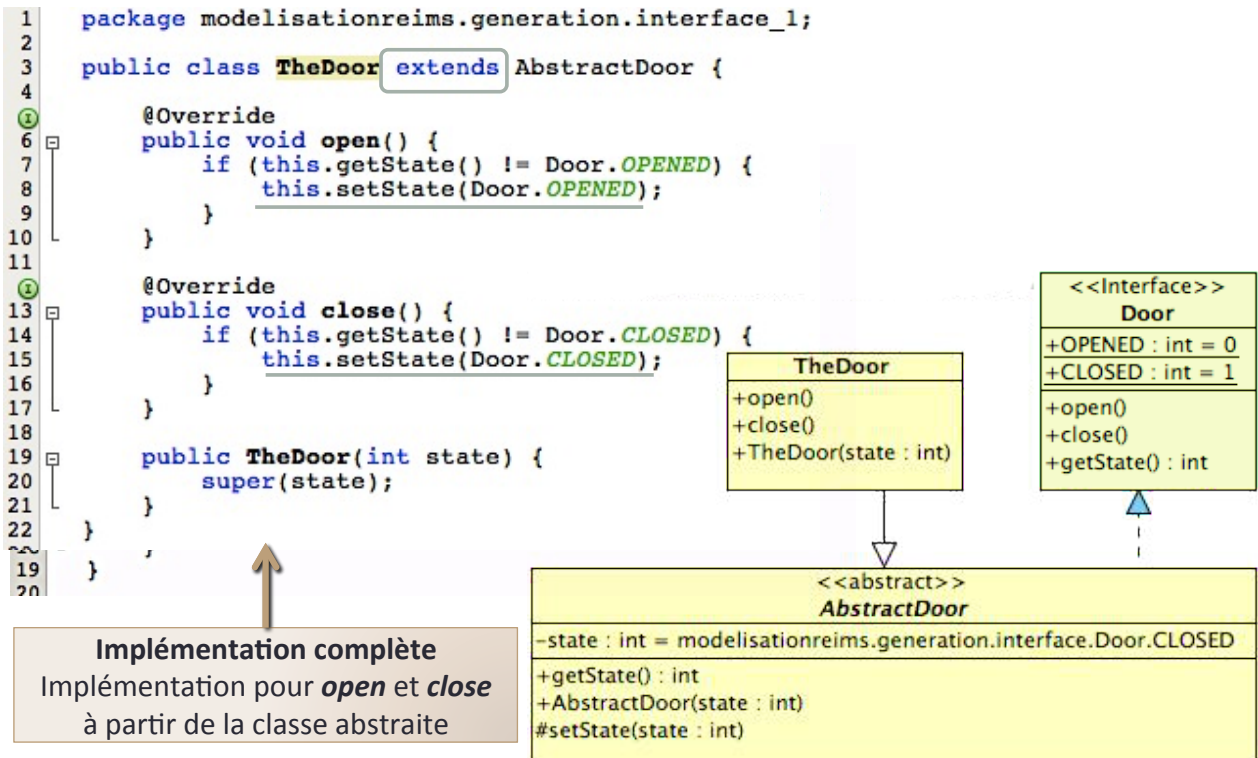
```

1 package modelisationreims.generation.interface_1;
2
3 public abstract class AbstractDoor implements Door {
4     private int state = Door.CLOSED;
5
6     @Override
7     public int getState() {
8         return this.state;
9     }
10
11     protected void setState (int state) {
12         this.state = state;
13     }
14
15     public AbstractDoor(int state) {
16         this.state = state;
17     }
18 }
19
20
  
```

Implémentation partielle
Implémentation de quelques
méthodes, *pas* pour *open* et *close*



Passage UML → code Java



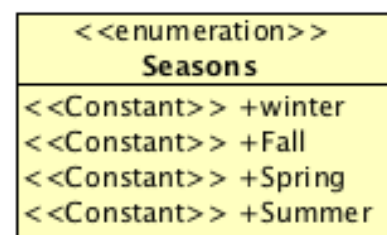
Passage UML → code Java

- Certains **stéréotypes** peuvent se traduire facilement
 - Profiles** spécifiques pour/par un langage
 - Profile pour Enterprise Java Beans : « EJBEntityBean »...
 - Profile pour .NET : « NetComponent »...
- Exemple : stéréotype « **enumeration** »

```

2
3 public enum Seasons {
4     winter, Fall, Spring, Summer;
5 }

```



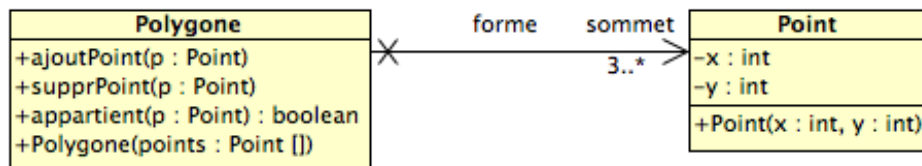
Passage UML → code Java

• Associations

- Liens de **dépendance** entre les classes
- Observation des **rôles**
 - Les rôles deviennent des **attributs** représentant la classe opposée
- Attention à la **navigabilité**
 - Association non navigable → pas d'attribut

• Multiplicités

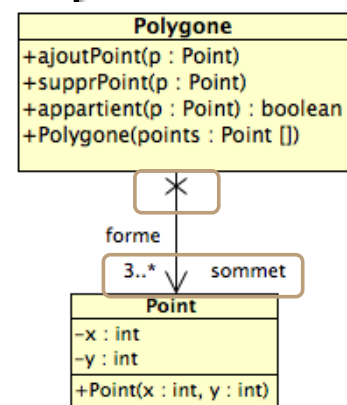
- Usage des **collections** pour traduire **0..***



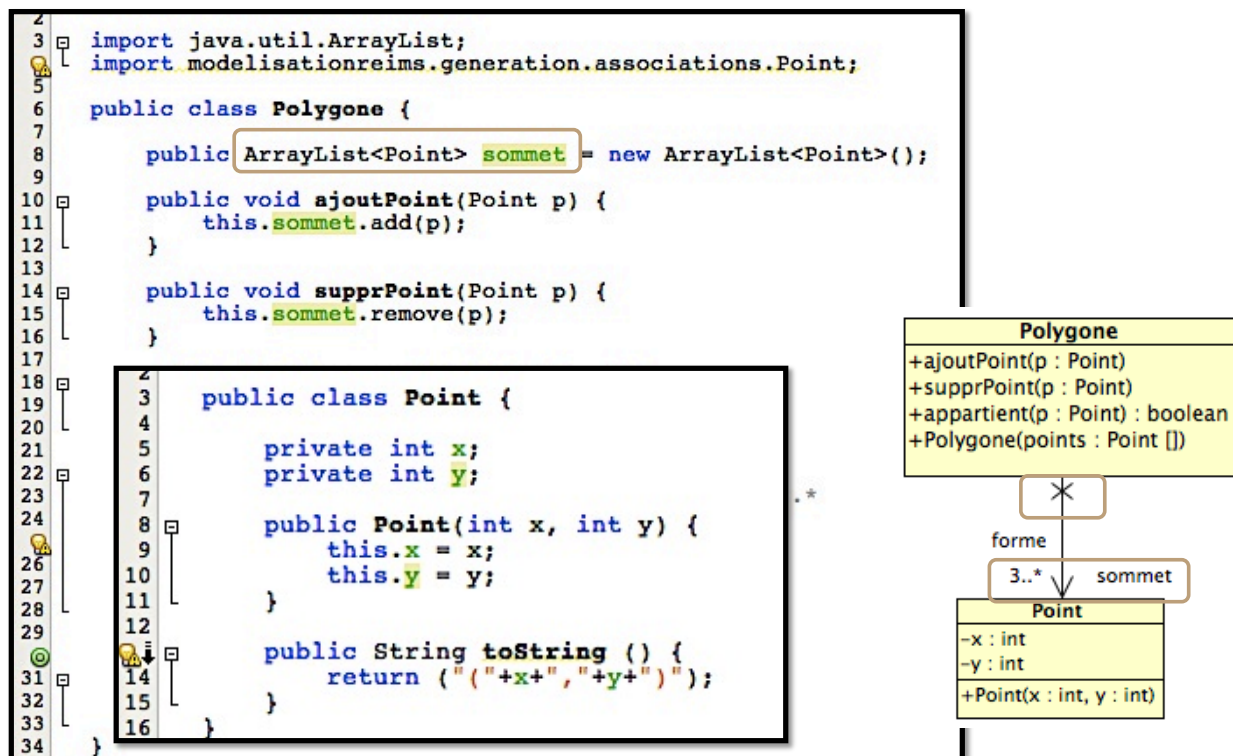
Passage UML → code Java

```

2
3 import java.util.ArrayList;
4 import modelisationreims.generation.associations.Point;
5
6 public class Polygone {
7     public ArrayList<Point> sommet = new ArrayList<Point>();
8
9     public void ajoutPoint(Point p) {
10         this.sommet.add(p);
11     }
12
13     public void supprPoint(Point p) {
14         this.sommet.remove(p);
15     }
16
17     public boolean appartient(Point p) {
18         return (this.sommet.contains(p));
19     }
20
21     public Polygone(Point[] points) {
22         //MISSING : traitement multiplicité min 3..*
23         //points.length >= 3 ou exception
24         for (Point p : points) {
25             sommet.add(p);
26         }
27     }
28
29     @Override
30     public String toString() {
31         return this.sommet.toString();
32     }
33 }
34
  
```



Passage UML → code Java



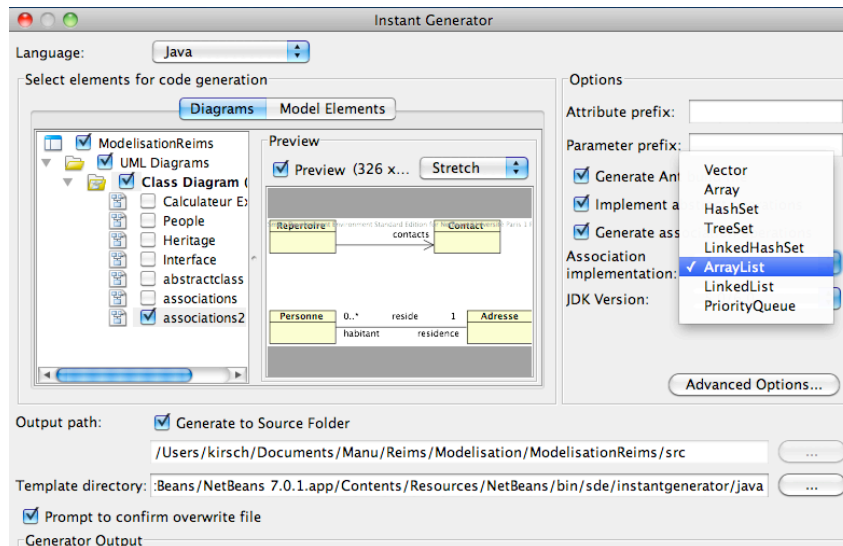
Passage UML → code Java

- **Multiplicité** : choix de la **collection**
 - Le choix de la collection dépend aussi des contraintes
 - *Order, unique...*
 - Exemples :
 - **Unique** : chaque élément est unique → **Set**
 - `private HashSet<Contact> contacts ;`
 - **Order, unique** : en plus d'être uniques, les éléments sont ordonnés
 - `private TreeSet<Contact> contacts;`



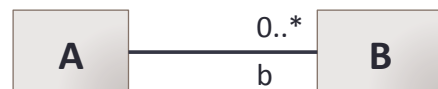
Passage UML → code Java

- **Multiplicité** : choix de la **collection**
 - Sur **VisualParadigm** : choix à charge du développeur



Passage UML → code Java

- **Multiplicité** : quelques conseils ...
 - **0..*** : usage des collections recommandée
 - Instanciation de la collection au constructeur
 - **La collection peut être vide**
 - **1..*** : usage des collections recommandée
 - **Un objet A est toujours associé à, au moins, un objet B**
 - On peut garantir la présence d'un objet B par le constructeur : A(B)
 - **0..1** : usage d'un attribut (rôle b)
 - **Le rôle b peut être null**
 - Gestion du null afin d'éviter les **NullPointerException**
 - **1..1** : usage d'un attribut (rôle b)
 - Un objet A est associé à **un et un seul objet B**
 - La valeur de B doit être indiqué dans le **constructeur**
 - Soit par instanciation **B = new (B)**, soit par **paramètre A(B)**
 - **M..N** : array ou collection
 - Les méthodes de A doivent **assurer le respect** de la multiplicité (**min M, max N**)

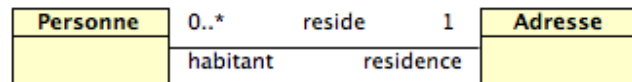


Passage UML → code Java

- **Navigabilité** : attention à la **cohérence**
 - Dans les **associations bidirectionnelles** (navigables dans les 2 sens), la **cohérence** doit être **assurée**
 - Toute modification sur une extrémité du lien doit être répercutée sur l'autre extrémité

- Exemple :

- *Si la résidence change, la liste d'habitants change aussi*



```

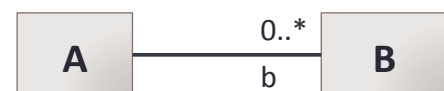
public class Personne {
    public Adresse residence;
    public void setAdresse(Adresse a) {
        if (this.residence != null)
            this.residence.removeHabitant(this);
        this.residence = a;
        this.residence.addHabitant(this);
    } ...
}
  
```

```

public class Adresse {
    public ArrayList<Personne> habitant =
        new ArrayList<Personne>();
    public void addHabitant (Personne p) { ... }
    public void removeHabitant (Personne p)
        { this.habitant.remove(p); }
    ... }
  
```

Passage UML → code Java

- **Associations** : **rôle** permanent ou variable ?
 - L'extrémité d'une association peut-elle changer ?
 - L'objet référencé par le rôle b peut-il changer ?



- **Rôle changeable**

- Un objet A peut être lié à **différents objets B** au cours de son cycle de vie
- Il faut alors prévoir les méthodes nécessaires sur la classe A
 - *getB / setB, addB / removeB ...*

- **Rôle permanent**

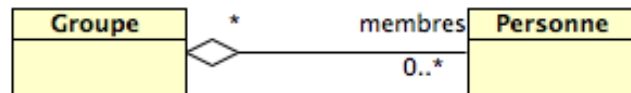
- Une fois lié à un objet B, la valeur du rôle b sur un objet A **ne change plus**
- Considérer l'usage du constructeur pour attribuer une valeur au rôle b
- Pas besoin de méthode pour modifier la valeur de b (pas de setB)

Passage UML → code Java

• Associations : agrégation & composition

- Sémantique **conteneur-contenu** différente
- Gestion des parties (**contenu**) différente

Une personne peut participer à plusieurs groupes



• Agrégation

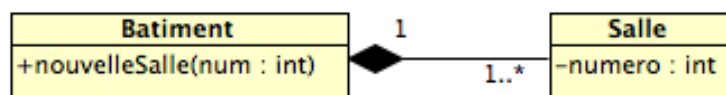
- Les objets contenu (Personne) peuvent être partagés entre plusieurs conteneurs (Groupe)
- Usage des collections similaire aux associations 0..* ou n..*
- Méthodes de **gestion de la collection** : addXXX, removeXXX
 - *addPersonne (Personne p), removePersonne(Personne p), List members()...*

Passage UML → code Java

• Composition

- Les objets contenu (Salle) ne sont **pas** partagés

Une salle n'appartient qu'à un seul bâtiment



- Le **cycle de vie** des objets **contenu** (Salle) est souvent **géré** par le **conteneur** (Bâtiment)
 - *new Salle (...)* dans la classe Batiment

```

5 public class Batiment {
6     public HashMap<Integer,Salle> unnamed_Salle_ = new HashMap<Integer,Salle>();
7     public void nouvelleSalle(int num) {
8         Salle s = new Salle(num);
9         unnamed_Salle_.put(num, s);
10    }

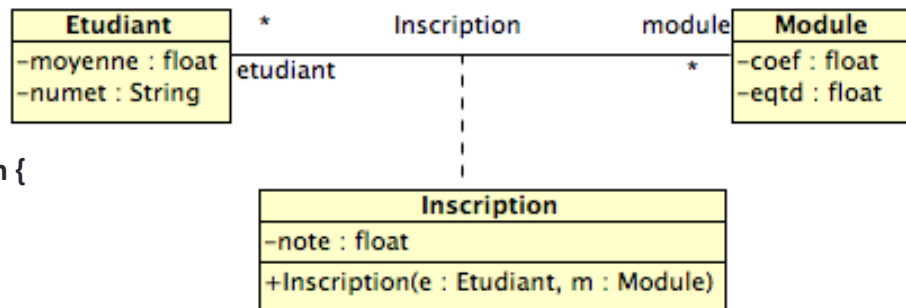
```

- L'objet **contenu** peut ne **pas être externalisé** par le conteneur
- Les instances de **contenu** ne sont **accessibles qu'au conteneur**
 - *Pas d'objets Salle en paramètre ou en retour*

Passage UML → code Java

• Classe-Associations

- **Sémantique** : un objet Inscription n'existe que s'il y a un étudiant inscrit à un module
- **Constructeur** peut assurer ce lien : *Inscription (Etudiant, Module)*



```

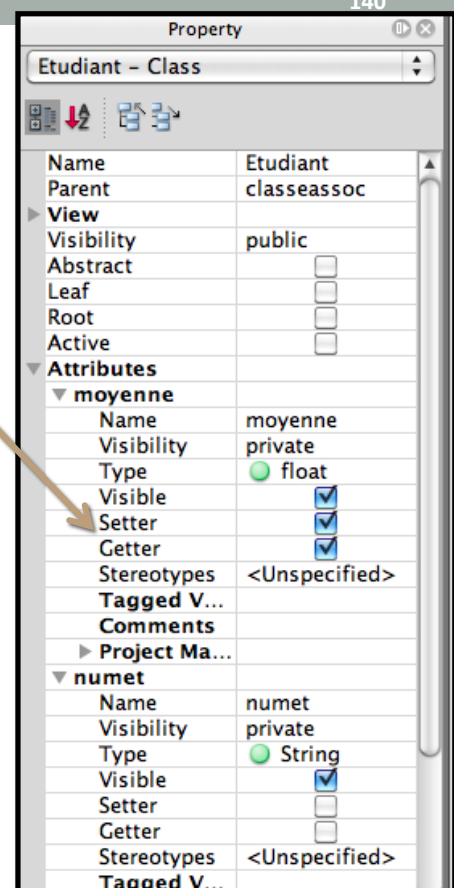
public class Inscription {
    ...
    Etudiant etudiant;
    Module module;
    public Inscription(Etudiant e,
                      Module m) {
        this.etudiant = e;
        this.module = m; ... }
    ...
  
```

Passage UML → code Java

• Classe-Associations

- Génération de code sur VisualParadigm

Propriétés des extrémités
Plusieurs propriétés disponibles, dont la présence des get/set

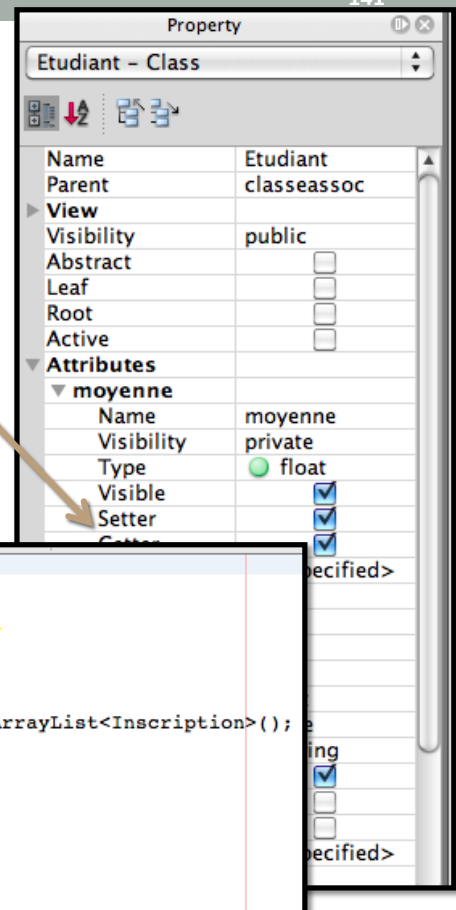


Passage UML → code Java

• Classe-Associations

- Génération de code sur VisualParadigm

Propriétés des extrémités
Plusieurs propriétés disponibles, dont
la présence des get/set



```

1 package modelisationreims.generation.classeassoc;
2
3 import java.util.ArrayList;
4 import modelisationreims.generation.classeassoc.Inscription;
5
6 public class Etudiant {
7     private float moyenne;
8     private String numet;
9     public ArrayList<Inscription> a_Inscription_ = new ArrayList<Inscription>();
10
11     public void setMoyenne(float moyenne) {
12         this.moyenne = moyenne;
13     }
14
15     public float getMoyenne() {
16         return this.moyenne;
17     }
18 }

```

Passage UML → code Java

Rôle changeable ou pas ?
VP génère automatiquement les
getter/setter pour les rôles

```

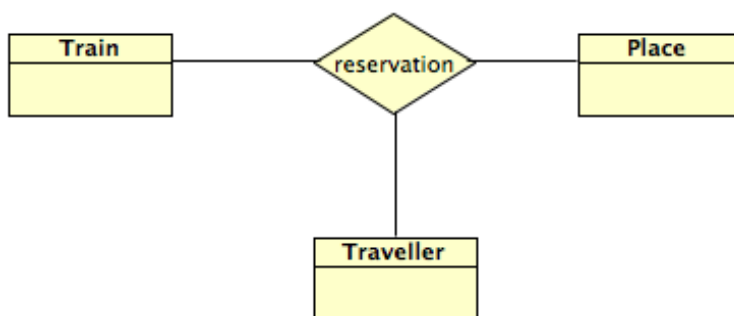
1 package modelisationreims.generation.classeassoc;
2
3 public class Inscription {
4     private float note;
5     public Etudiant etudiant;
6     public Module module;
7
8     public Inscription(Etudiant e, Module m) {
9         throw new UnsupportedOperationException();
10    }
11
12    public void setEtudiant(Etudiant etudiant) {
13        this.etudiant = etudiant;
14    }
15
16    public Etudiant getEtudiant() {
17        return this.etudiant;
18    }
19
20    public void setModule(Module module) {
21        this.module = module;
22    }
23
24    public Module getModule() {
25        return this.module;
26    }
27 }

```

Passage UML → code Java

• Associations n-aires

- Bien souvent, les associations n-aires vont être traitées comme une classe-association
 - Création d'une **classe** lui correspondant
 - Extrémités établies par le **constructeur** ou par **getter/setter**



```

public class reservation {
    Train a_Train_;
    Place a_Place_;
    Traveller a_Traveller_;
    public reservation (Train t, Place p,
                        Traveller tr) {
        this.a_Place_ = p;
        this.a_Train_ = t;
        this.a_Traveller_ = tr;
    }
    ...
}

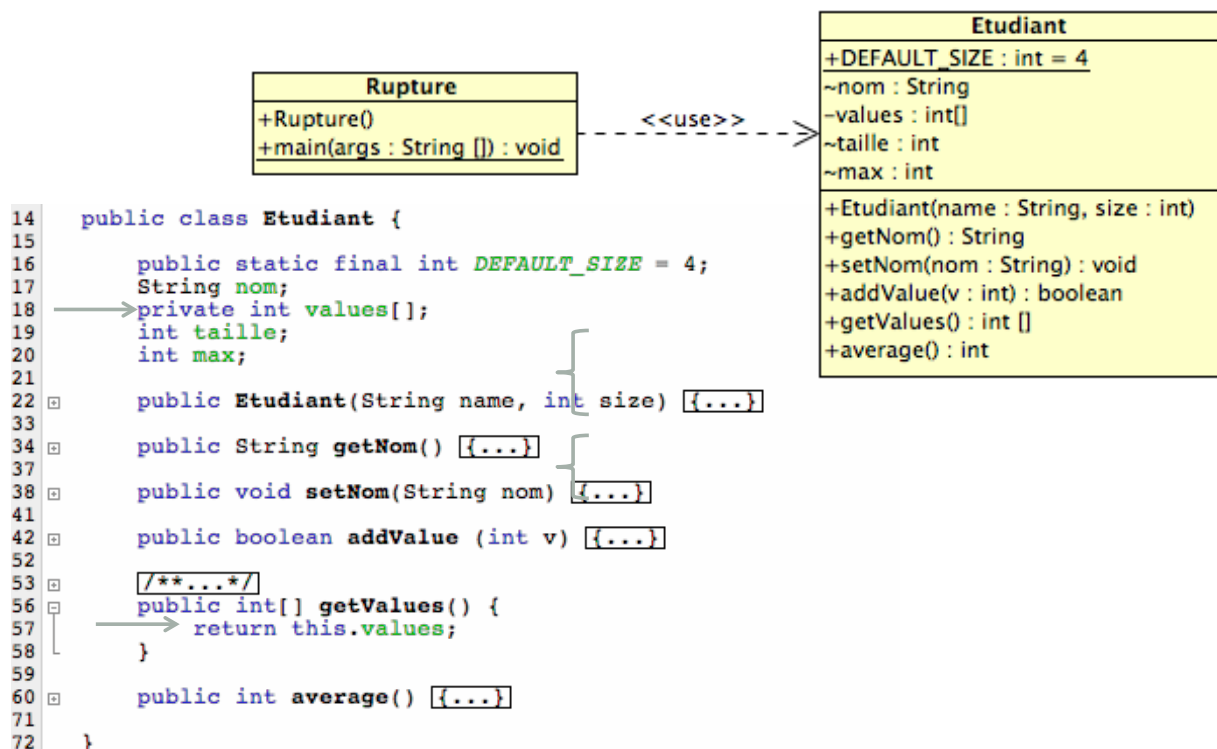
```

Passage UML → code Java

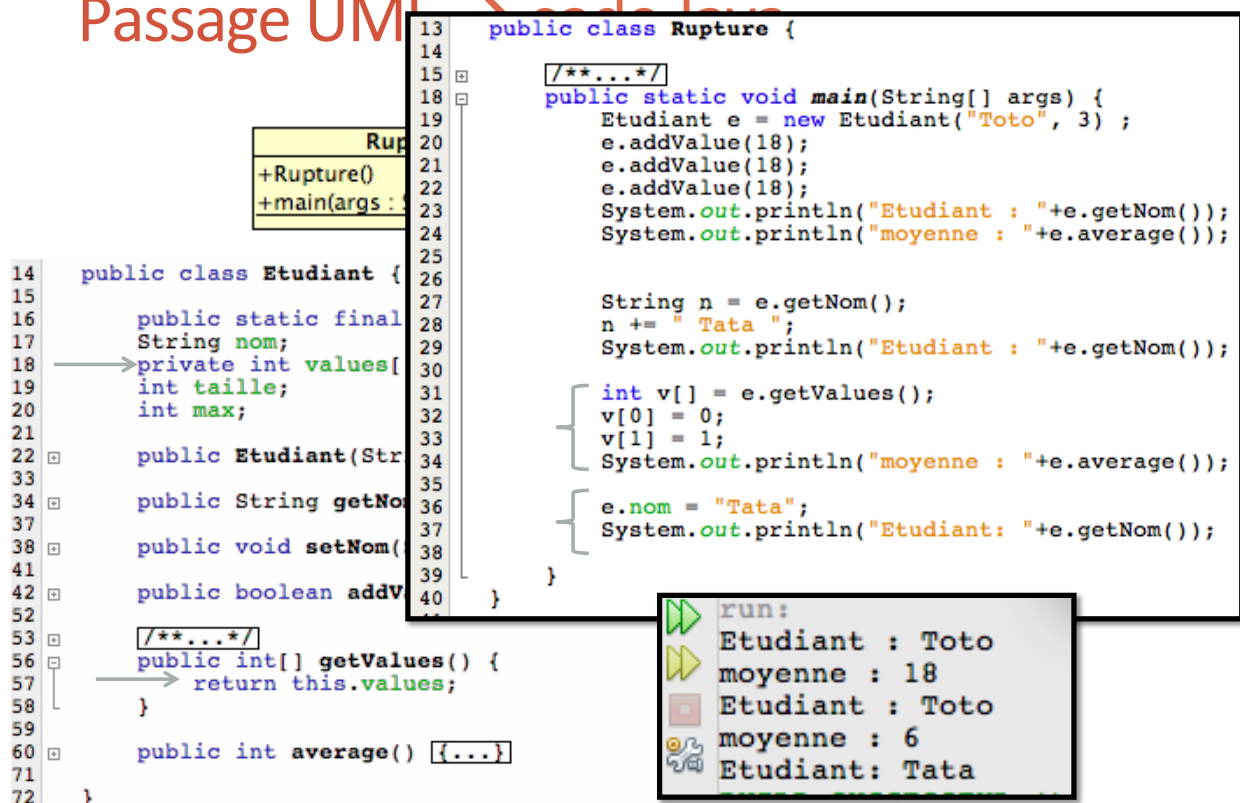
• Quelques détails à ne pas oublier...

- **Envoie de message** : Attention à l'**encapsulation** !
 - Attention à ne pas **exposer les structures internes** par les retours des méthodes
 - Retour d'un objet/array : **passage par référence**
 - Classe **String** : non modifiable
- Attention aux attributs **protected** (et "package")

Passage UML → code Java



Passage UML → code Java



Passage UML → code Java

- **Quelques détails à ne pas oublier...**
- **Instanciation** : Attention où on instancie
 - La création d'une instance d'une autre classe entraîne la création d'une **dépendance forte** entre classes
 - **Couplage fort** → plus difficile à réutiliser et à faire évoluer
 - Exemple : utilisation d'une sous-classe....
 - Usage du principe de l'**injection de dépendance**
 - Introduction de la nouvelle instance par paramètre ou par getter/setter
 - Exemple : interface Door, classe ControlDevice