

MODÉLISATION DES APPLICATIONS

Manuele Kirsch Pinheiro

Maître de Conférences – Université Paris 1

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

<http://mkirschp.free.fr>

97

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

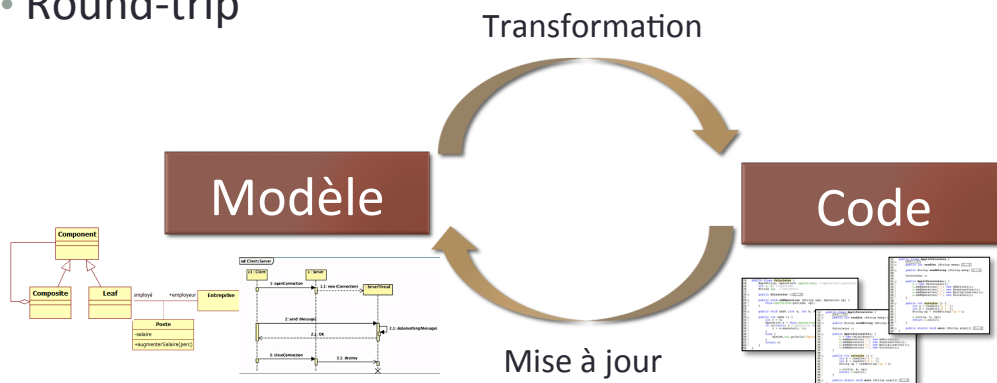
- Développement de qualité
 - **Penser qualité**
 - Modularité / réutilisation
 - Évolutivité / extensibilité
 - Robustesse
 - **Vision globale**
 - Solution à court terme X solution à long terme

➤ **Besoin de modélisation !**

✧ **Correspondance modèle ↔ code généré**

Modélisation des applications

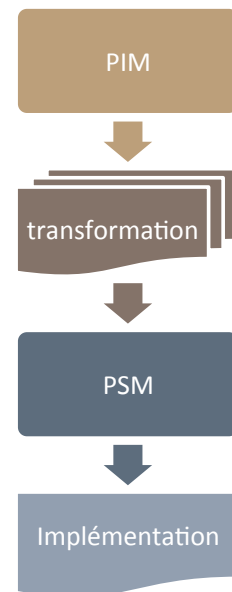
- Correspondance modèle ↔ code
 - Bien modéliser ne suffit pas si le code ne correspond pas au modèle
 - **Traçabilité**
- Round-trip



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

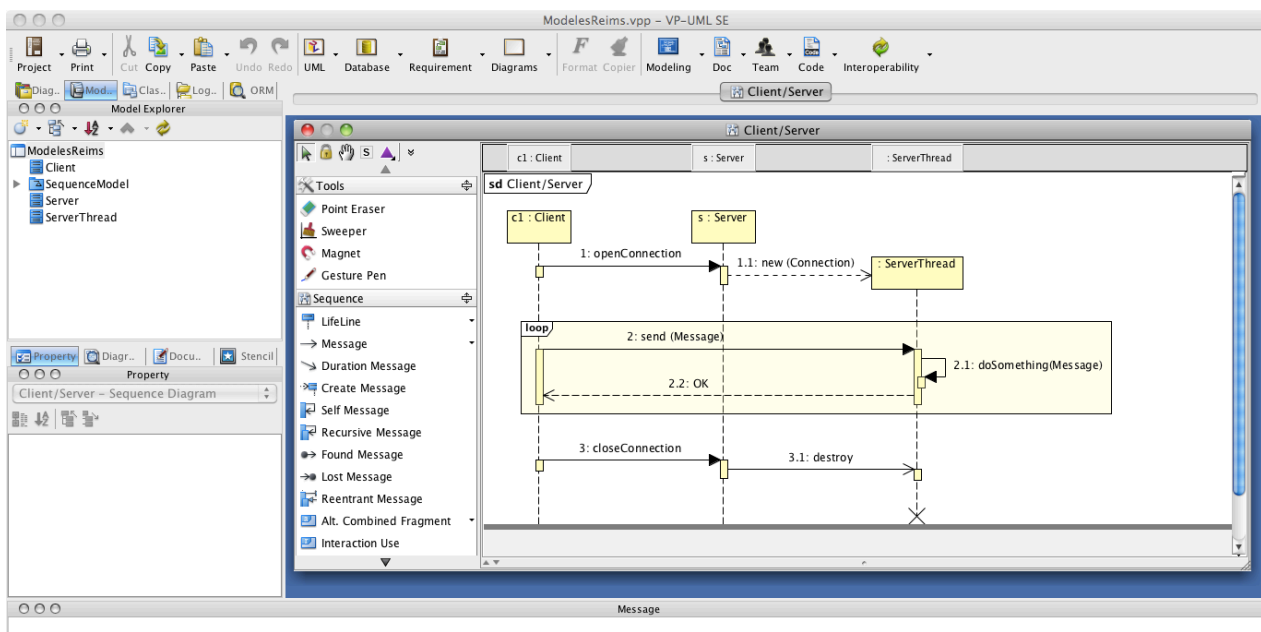
- Plusieurs applications proposent la génération automatique de code
 - Passage modèles UML → code Java, C++...
 - Souvent à partir du **diagramme de classes**
 - Intégré dans une **démarche de modélisation**
- Au-delà des applications, comment se traduit-il un diagramme de classes en code Java ??
 -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

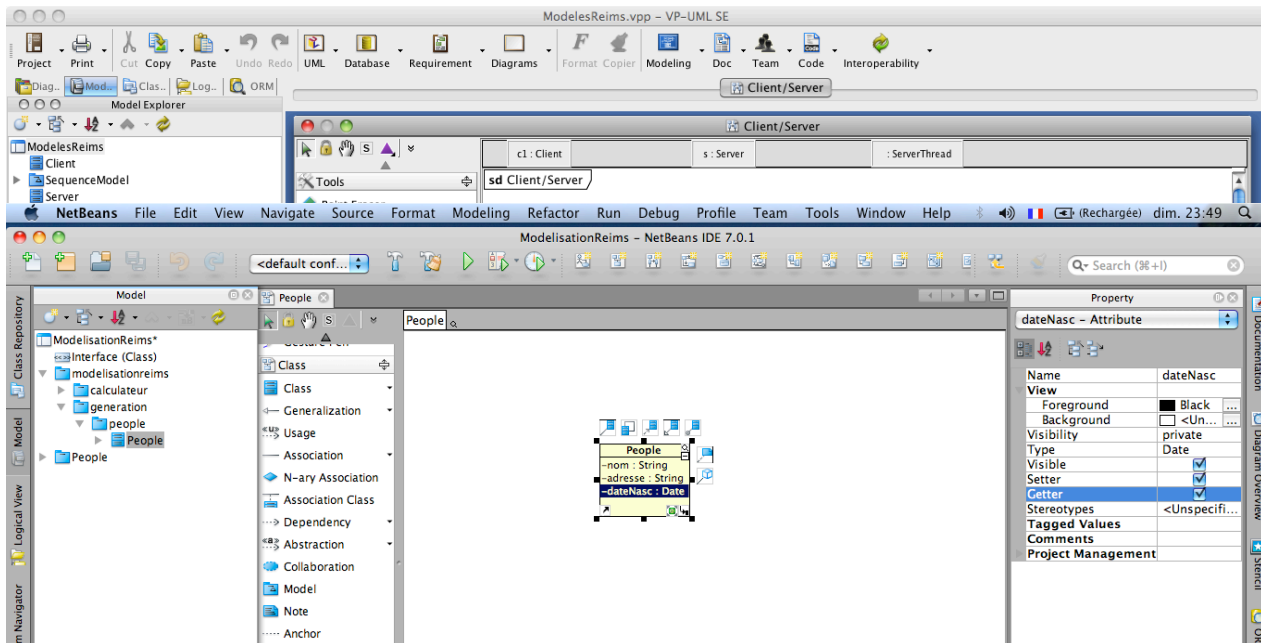
Applications génération de code

- Exemple d'application de génération de code



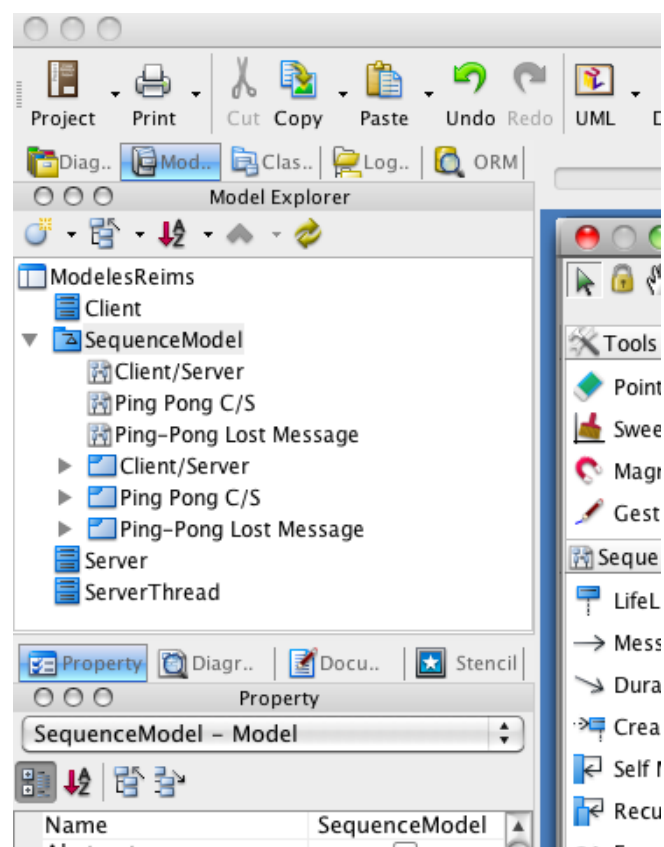
Applications génération de code

- Exemple d'application de génération de code



Applications

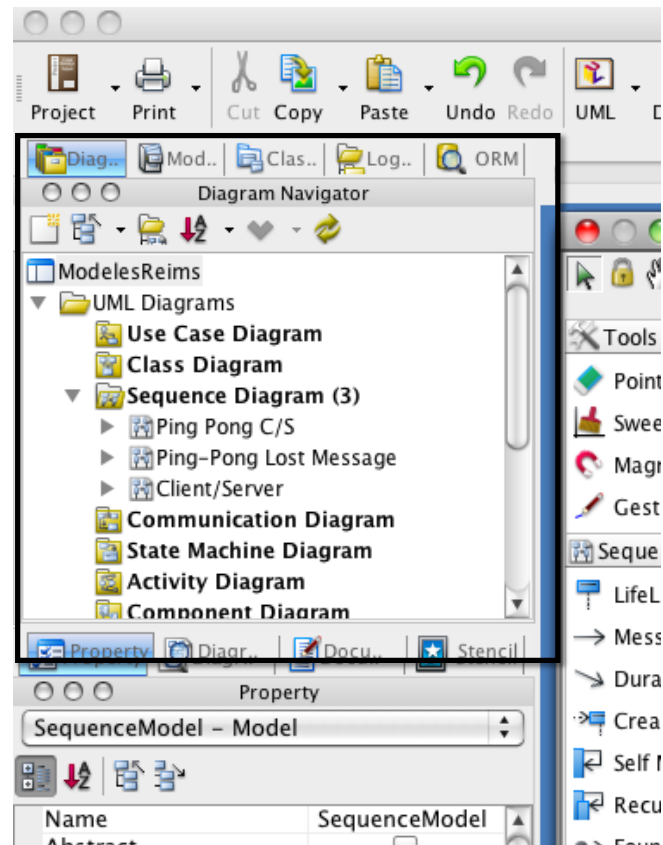
- **Visual Paradigm**
 - Support complet à UML 2
 - Vue par modèle ou par diagramme
 - Plusieurs diagrammes disponibles



Applications

- **Visual Paradigm**

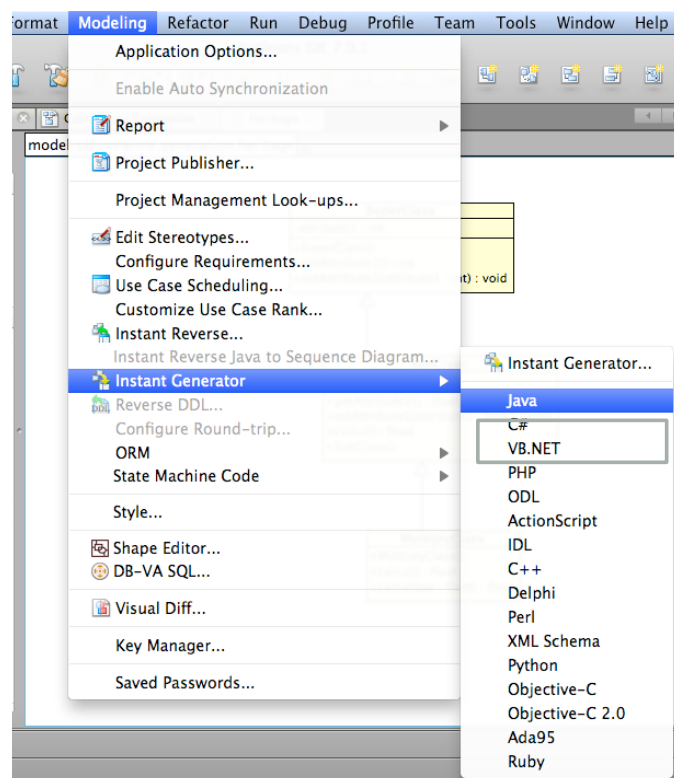
- Support complet à UML 2
- Vue par modèle ou par diagramme
- Plusieurs diagrammes disponibles



Applications

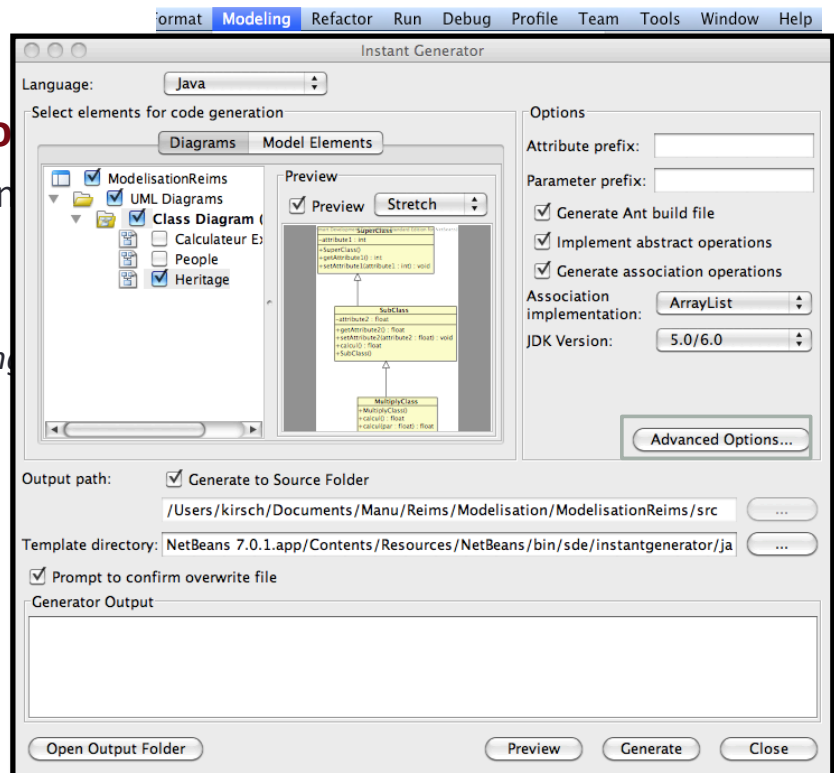
- **Génération de code**

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



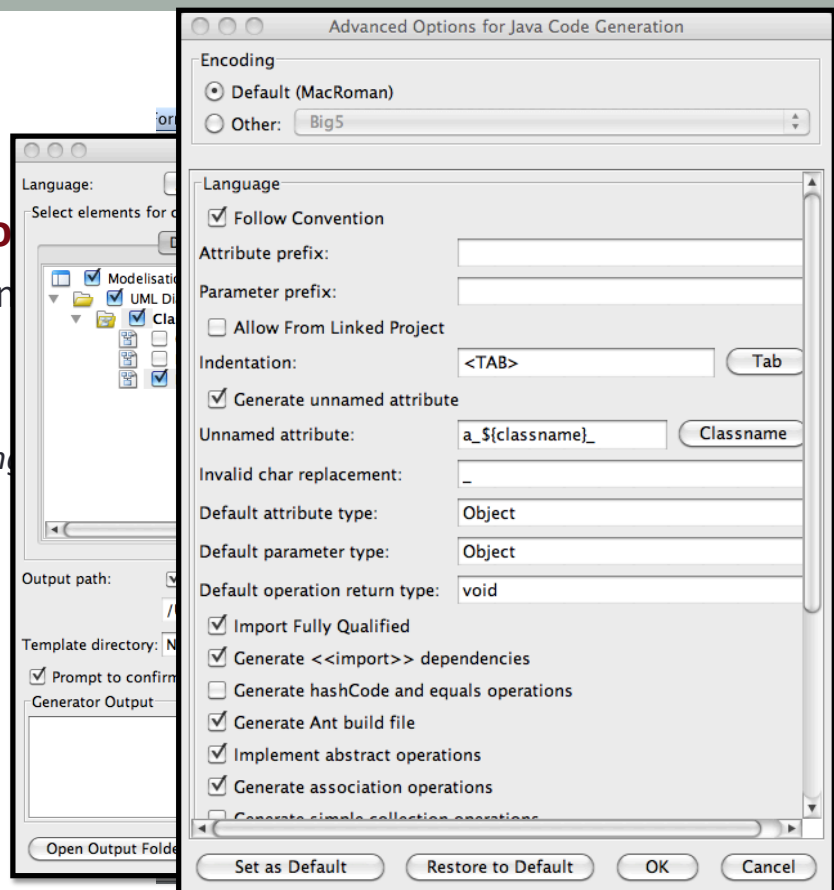
Applications

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



Applications

- **Génération de code**
 - Génération à un instant
 - *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

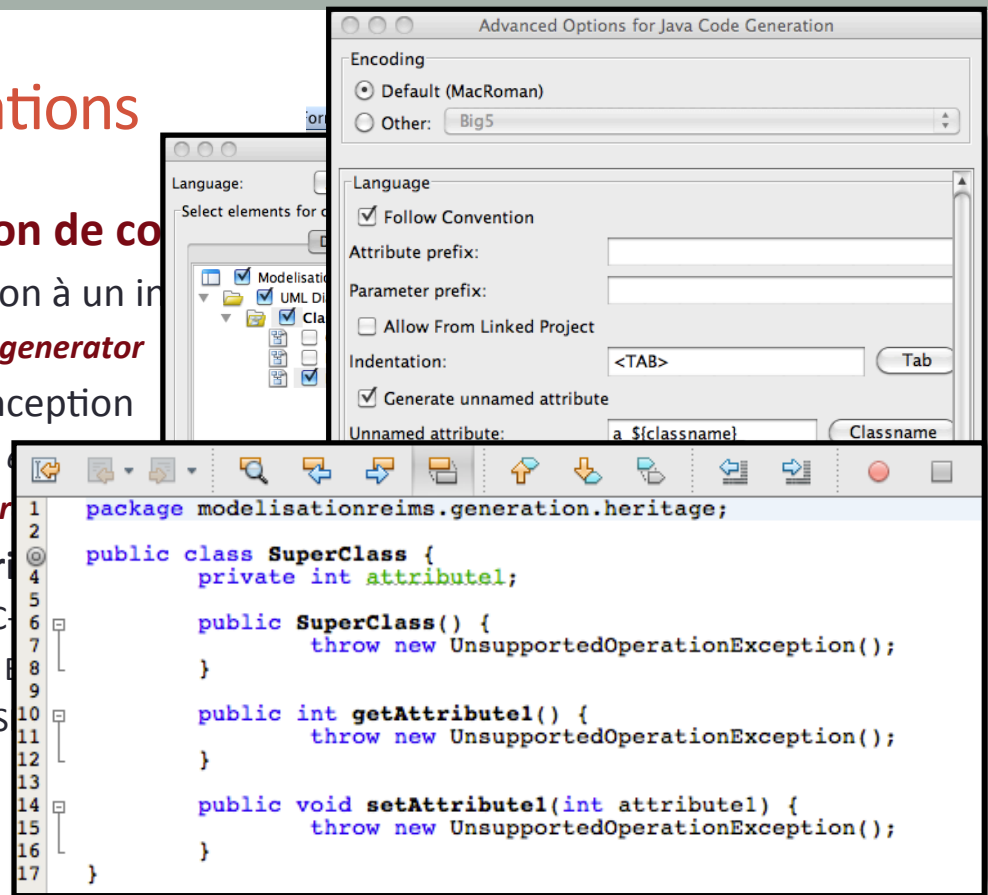
- **Instant**

- **Round-trip**

- Java & C

- Version B

- Java -> S



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

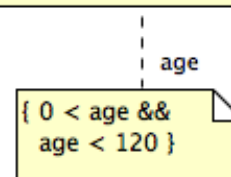
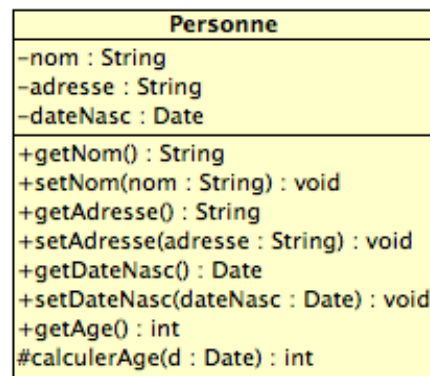
- Correspondance directe avec les *class* Java
- **Attributs et méthodes**
 - Attributs dérivés
 - **Multiplicité des attributs** : **collections** ou **arrays**

• Visibilité :

- Package (~) valeur par défaut en Java
- private (-), public (+), protected (#)

• Bonnes pratiques

- Getter / Setter pour chaque attribut
- Documentation à l'aide de commentaires Javadoc

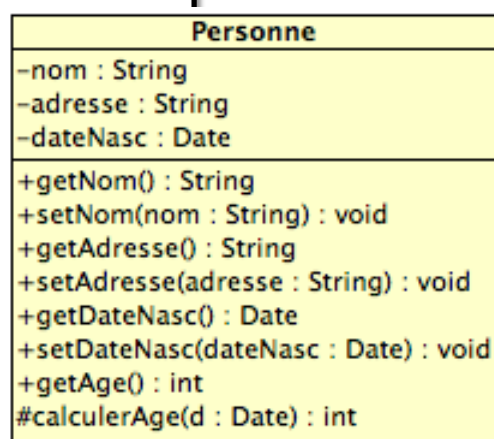


- Attention au respect des **contraintes** !!

```

1 package modelisationreims.generation.people;
2
3 import java.util.Calendar;
4 import java.util.Date;
5
6 /**...*/
10 public class Personne {
11
12     private String nom;
13     private String adresse;
14     private Date dateNasc;
15
16     /**...*/
20     public String getNom() {
21         return this.nom;
22     }
23
24     /**...*/
28     public void setNom(String nom) {
29         this.nom = nom;
30     }
31
32     /**...*/
36     public String getAdresse() {
37         return this.adresse;
38     }
39
40     /**...*/
44     public void setAdresse(String adresse) {...}
47
48     /**...*/
52     public Date getDateNasc() {...}
55
56     /**...*/
62     public void setDateNasc(Date dateNasc) {...}
66
67     /**...*/
72     public int getAge() {

```



```

39
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'individu à partir de sa
69   * date de naissance.
70   * @return nombre entier représentant l'âge de l'individu
71   */
72  public int getAge() {
73      return this.calculerAge(this.dateNasc);
74  }
75
76  /**
77   * La méthode <i>calculerAge</i> calcule l'âge à partir d'une date d
78   * arbitraire.
79   * @param d date à partir de laquelle on calcule l'âge
80   * @return nombre entier représentant l'âge
81   */
82  protected int calculerAge(Date d) {
83      Calendar today = Calendar.getInstance();
84      Calendar birth = Calendar.getInstance();
85      birth.setTime(d);
86      int y = birth.get(Calendar.YEAR) - today.get(Calendar.YEAR);
87      if ((birth.get(Calendar.MONTH) - today.get(Calendar.MONTH)) > 0) //pas e
88      {
89          y--;

```

Javadoc

```

39
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  }

```

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 }

```

55
56  /**
57   * <p>La méthode <i>setDateNasc</i> permet de mettre à jour la date de naissanc
58   * d'un individu.</p>
59   * <p><b>Important:</b> l'âge de l'individu doit être comprise entre 0 et 120
60   * @param dateNasc
61   */
62  public void setDateNasc(Date dateNasc) {
63      //Traitement de la contrainte : { 0 < age && age < 120 }
64      int age = this.calculerAge(dateNasc);
65      if (0 < age && age < 120) { //contrainte OK, valeur repris
66          this.dateNasc = dateNasc;
67      }
68  }
69
70

```

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**

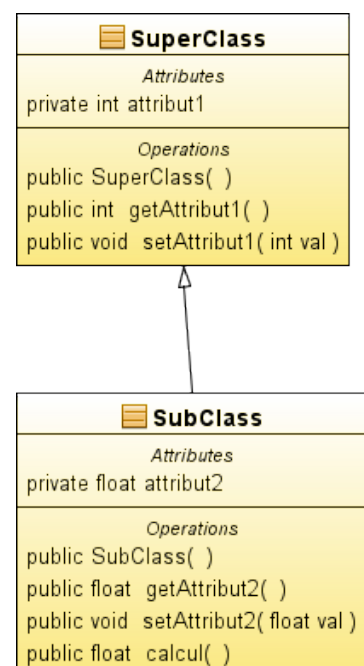
Passage UML → code Java

- **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

```

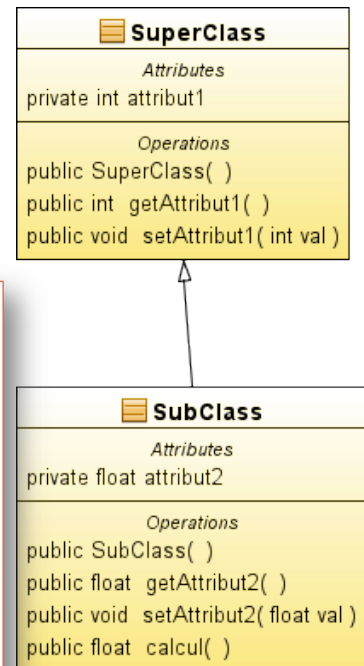
3  public class SuperClass {
4
5      private int attribut1;
6
7      public SuperClass () {...}
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22

```

```

4  public class SubClass extends SuperClass {
5
6      private float attribut2;
7
8      public SubClass () {
9      }
10
11      public float getAttribut2 () {...}
12
13      public void setAttribut2 (float val) {...}
14
15      public float calcul () {
16          return this.attribut2 + (float)this.getAttribut1();
17      }
18
19
20
21
22

```



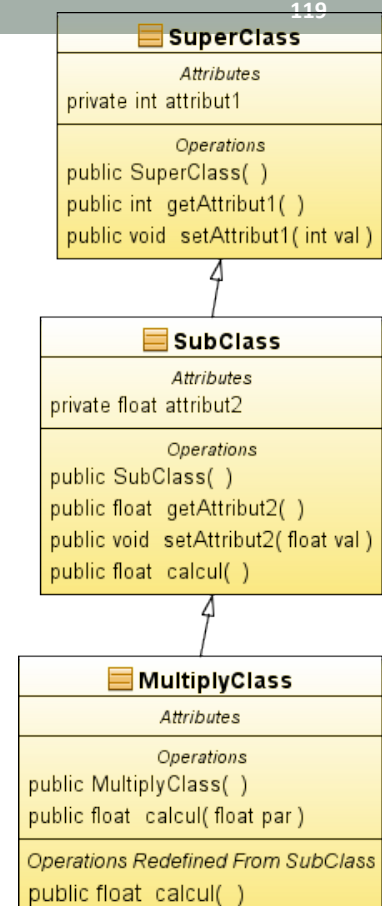
Passage UML → code Java

• 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

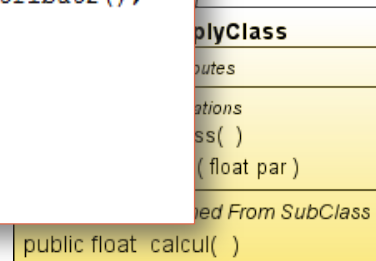
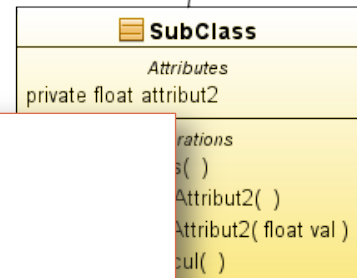
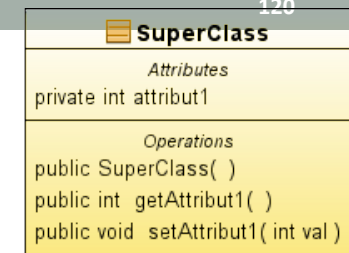
- Exemple de **redéfinition & surcharge** : classe **MultiplyClass**

- Redéfinition** : méthode **calcul ()**

```

3 public class MultiplyClass extends SubClass {
4
5     public MultiplyClass () { ... }
6
7     @Override
8     public float calcul () {
9         return super.getAttribut1() * super.getAttribut2();
10    }
11
12    public float calcul (float par) {
13        return par * super.calcul();
14    }
15
16 }

```



Passage UML → code Java

• Interface

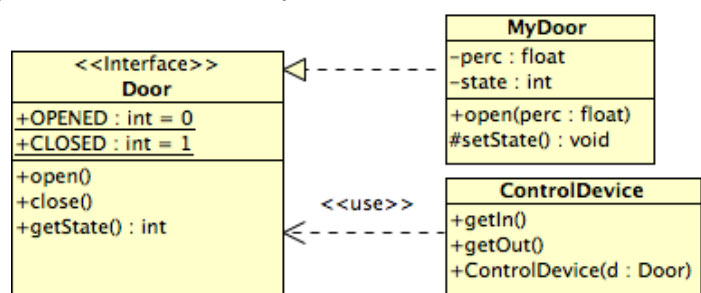
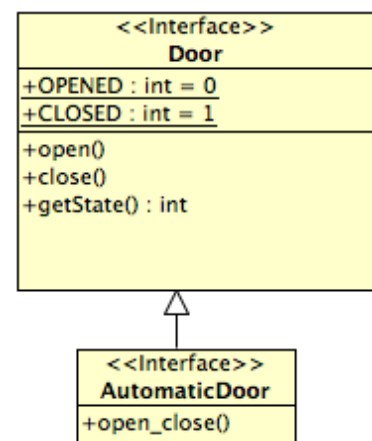
- Mot-clé **interface**
- Définition des **méthodes, sans implémentation**
- Possibilité d'implémenter plusieurs interfaces
- Possibilité d'**héritage** entre interfaces

• Implémentation d'une interface

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

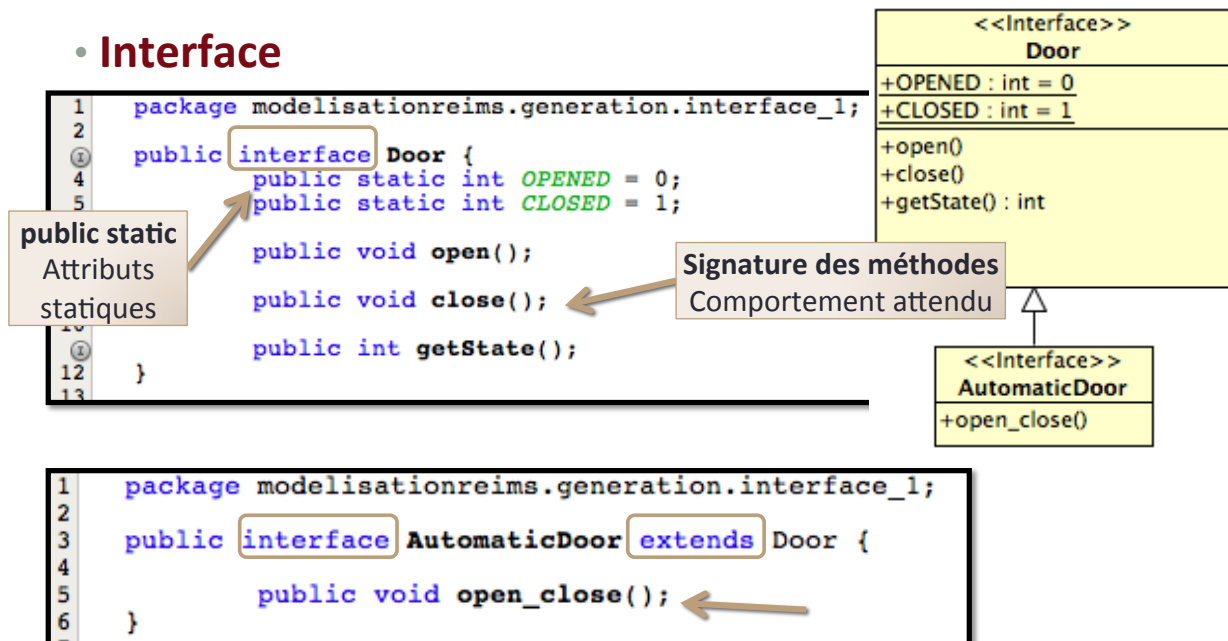
• Usage

- Faible couplage
- Stéréotype « use »

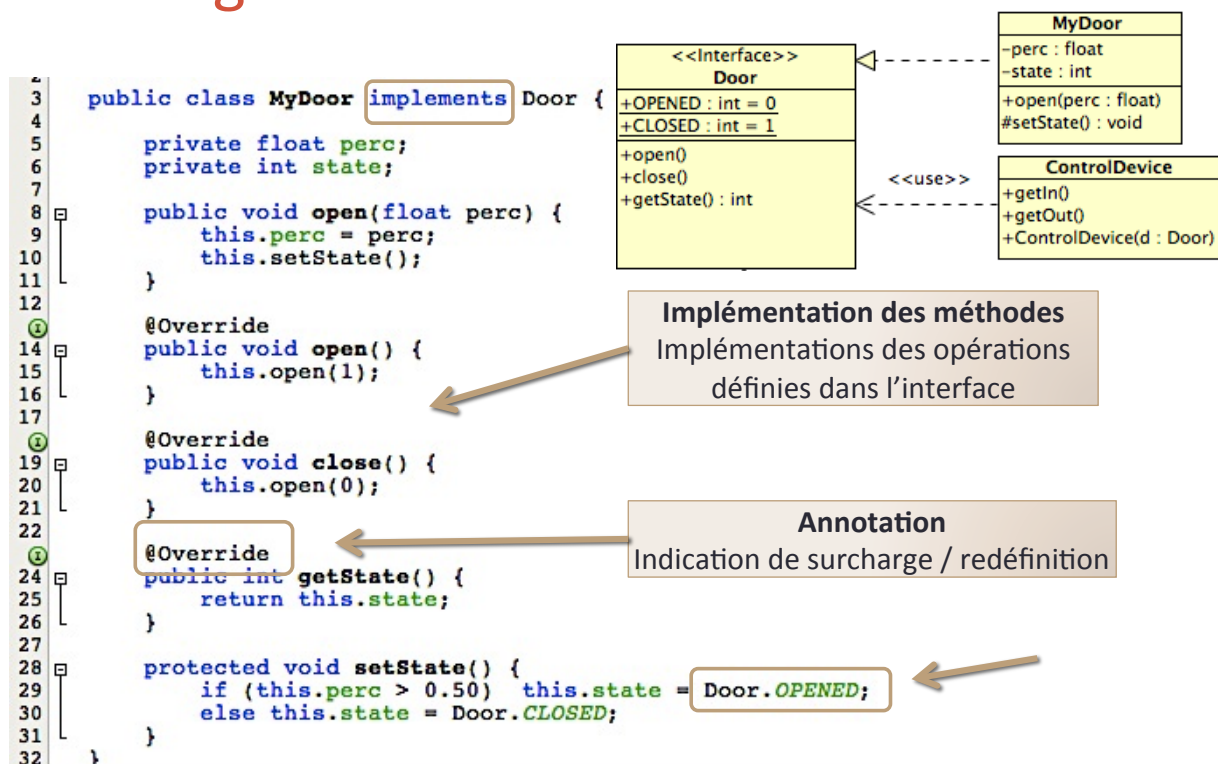


Passage UML → code Java

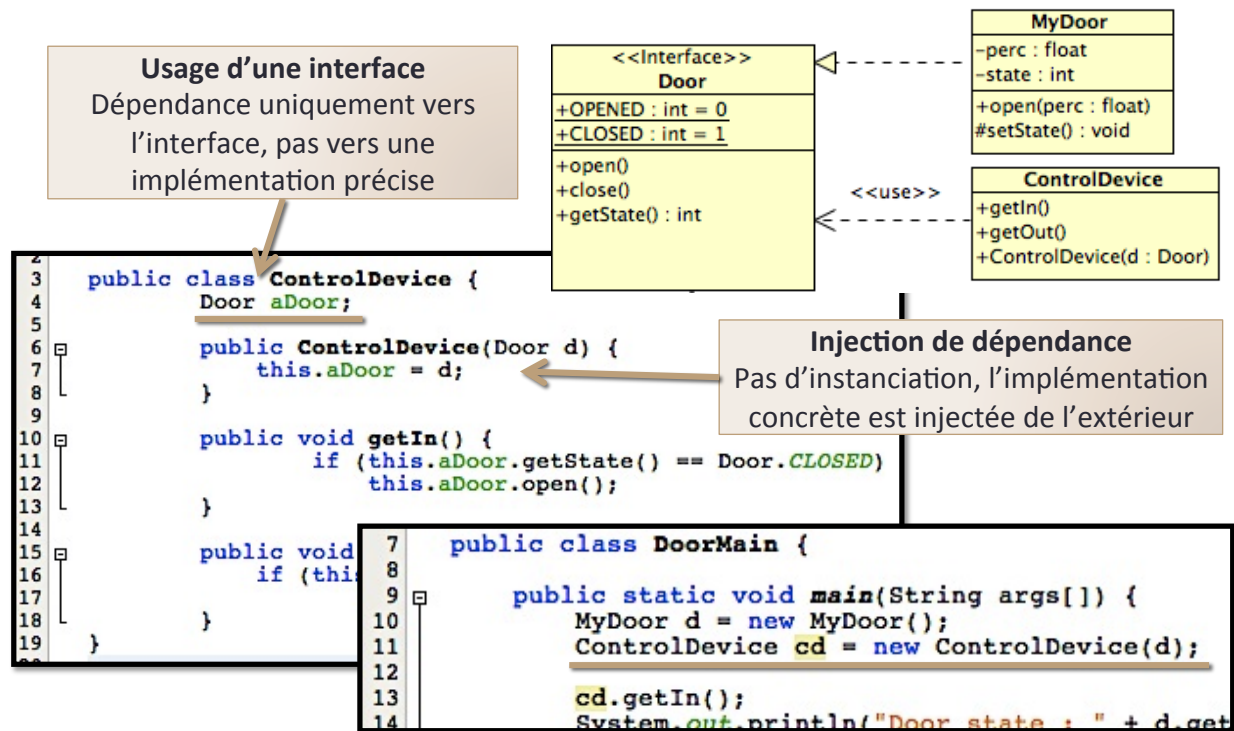
Interface



Passage UML → code Java



Passage UML → code Java

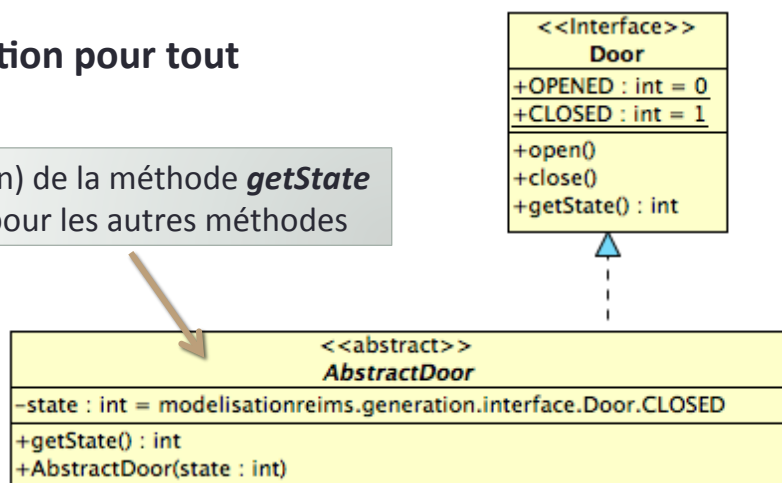


Passage UML → code Java

• Classes abstraites

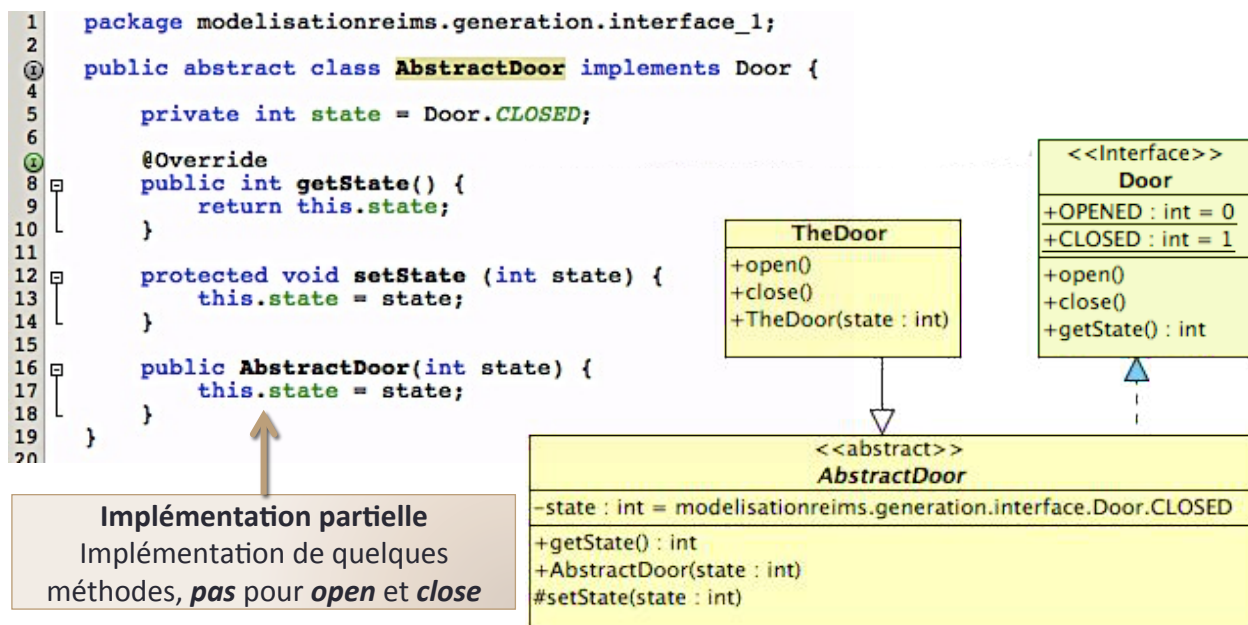
- Stéréotype « **abstract** » (optionnel)
- Caractéristiques (attributs/méthodes) communes aux sous-classes
- **Pas d'implémentation pour tout**

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

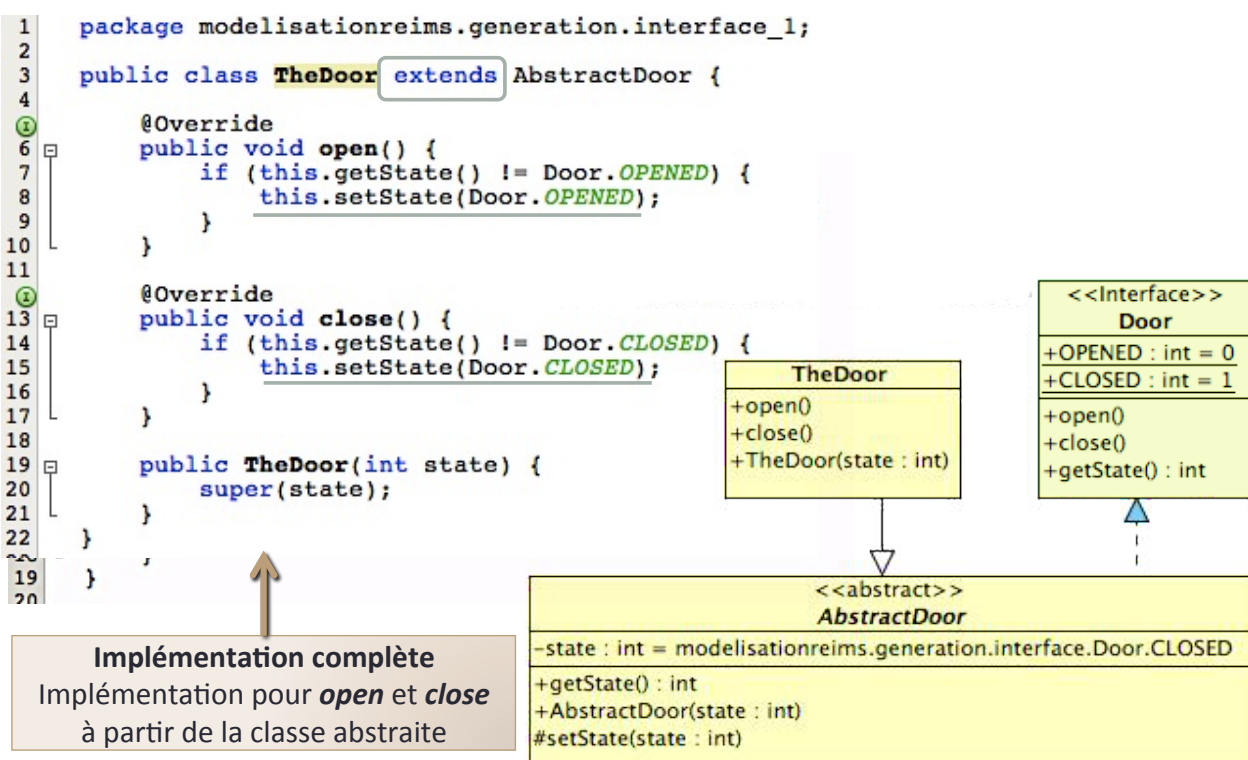


Passage UML → code Java

• Classes abstraites



Passage UML → code Java



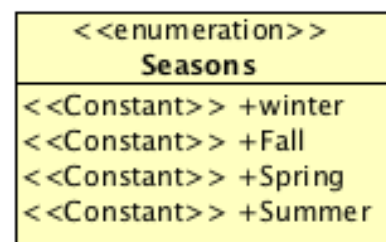
Passage UML → code Java

- 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** »

```

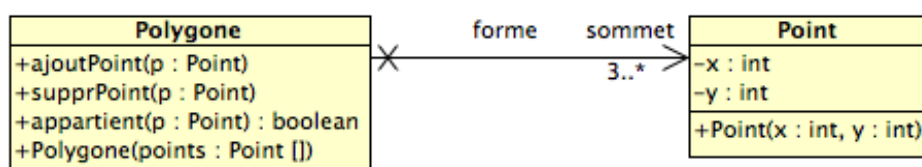
2
3 public enum Seasons {
4     winter, Fall, Spring, Summer;
5 }

```

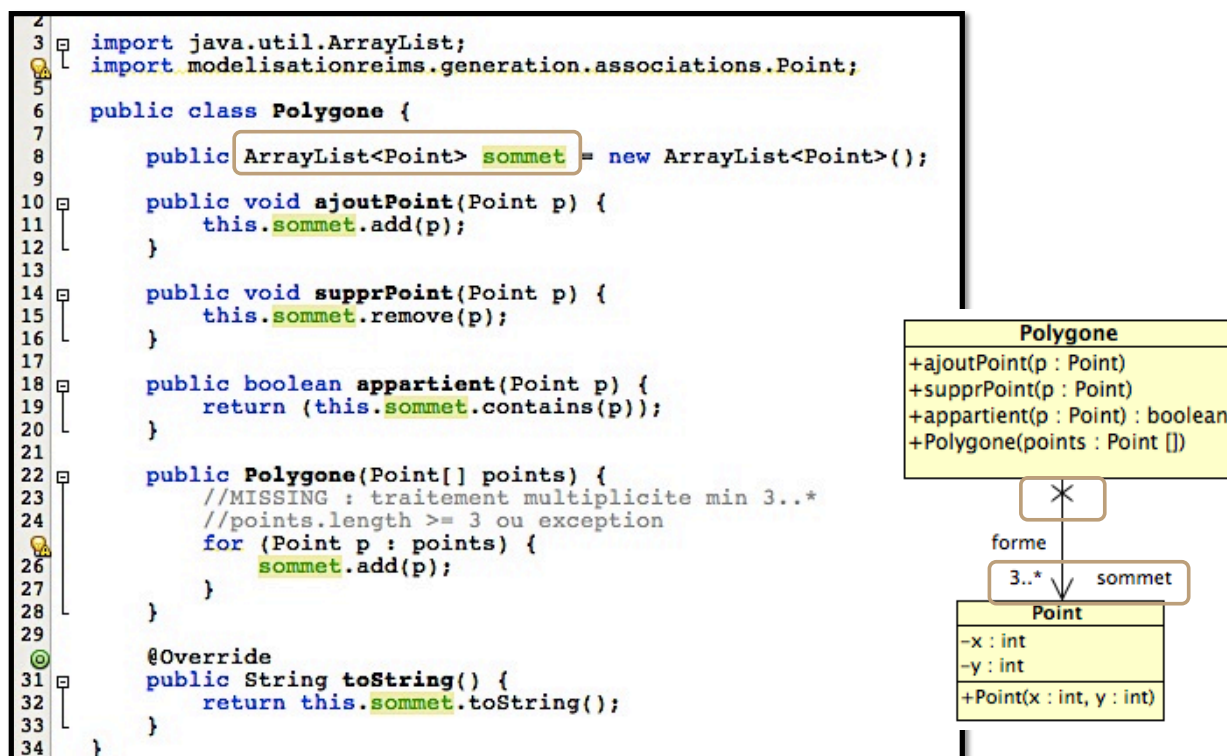


Passage UML → code Java

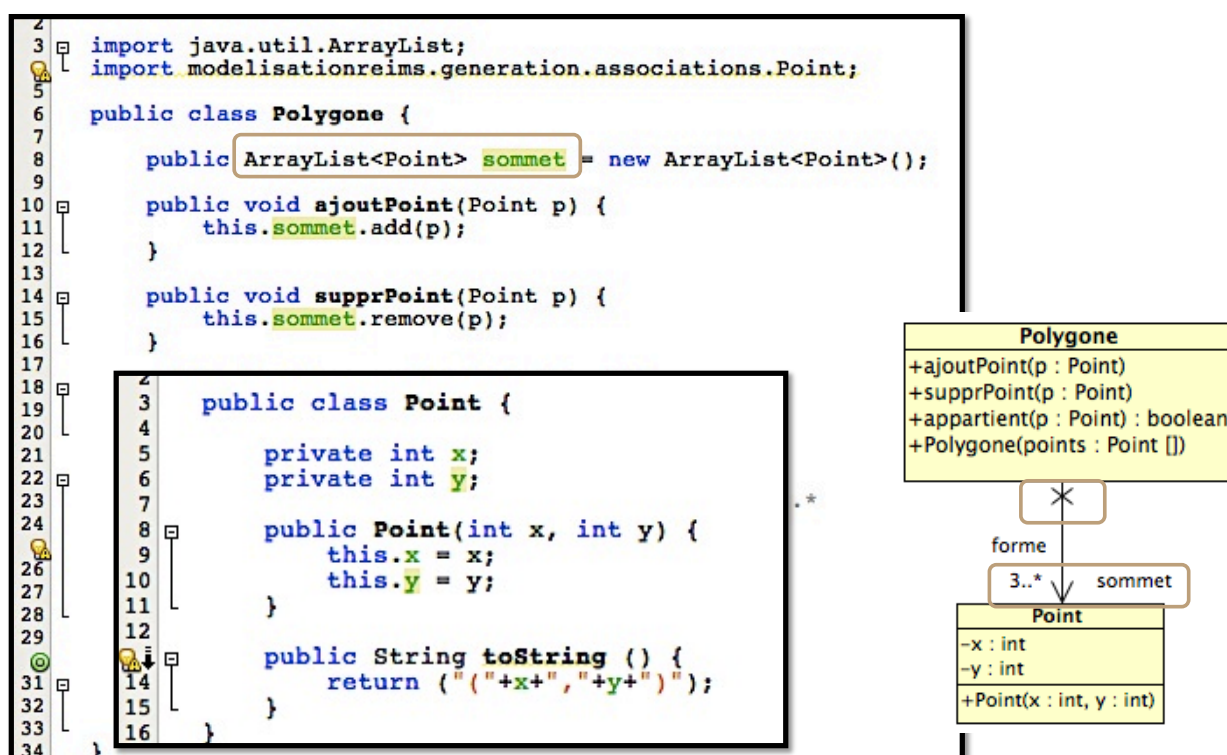
- **Associations**
 - Liens de **dépendance** entre les classes
 - Observation des **rôles**
 - Les rôles deviennent des **attributs** représentant la classe opposée
 - Attention à la **navigabilité**
 - Association non navigable → pas d'attribut
- **Multiplicités**
 - Usage des **collections** pour traduire 0..*



Passage UML → code Java

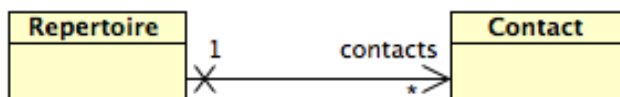


Passage UML → code Java



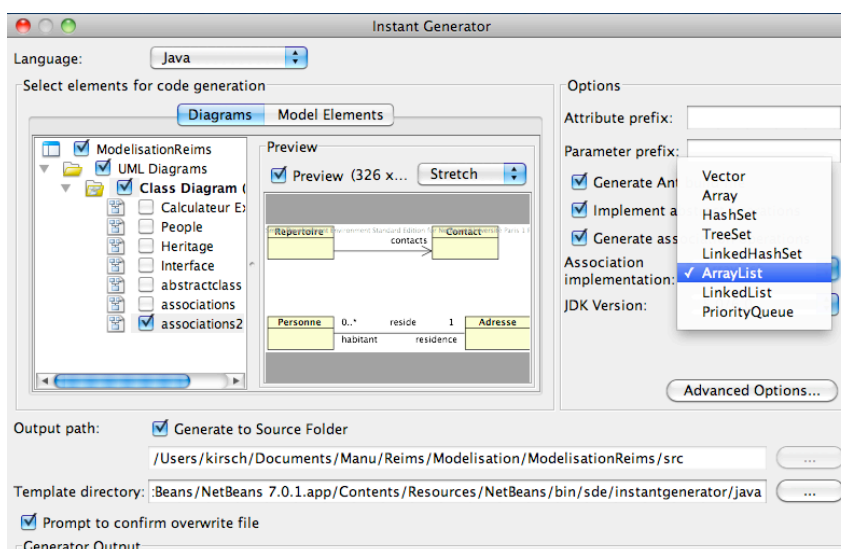
Passage UML → code Java

- **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)



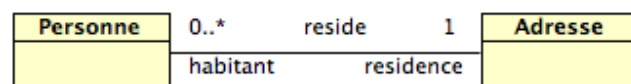
Passage UML → code Java

- **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 ?



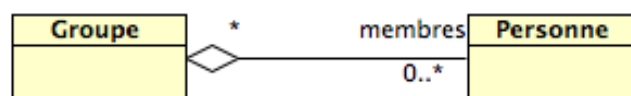
- **Rôle changeable**
 - Un objet A peut être lié à **différents objets B** au cours de son cycle de vie
 - Il faut alors prévoir les méthodes nécessaires sur la classe A
 - *getB / setB, addB / removeB ...*
- **Rôle permanent**
 - Une fois lié à un objet B, la valeur du rôle b sur un objet A **ne change 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**

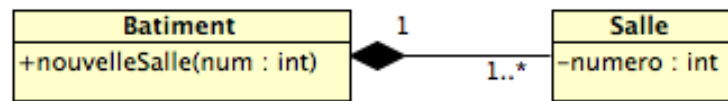
- Sémantique **conteneur-contenu** différente
- Gestion des parties (**contenu**) différente

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

```

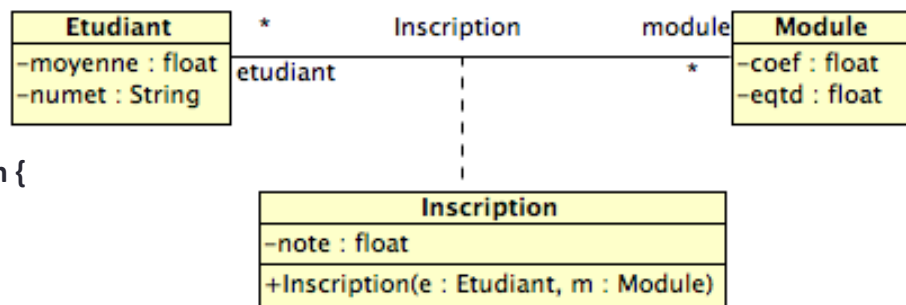
5 public class Batiment {
6     public HashMap<Integer,Salle> unnamed_Salle_ = new HashMap<Integer,Salle>();
7     public void nouvelleSalle(int num) {
8         Salle s = new Salle(num);
9         unnamed_Salle_.put(num, s);
10    }
  
```

- 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)*



```

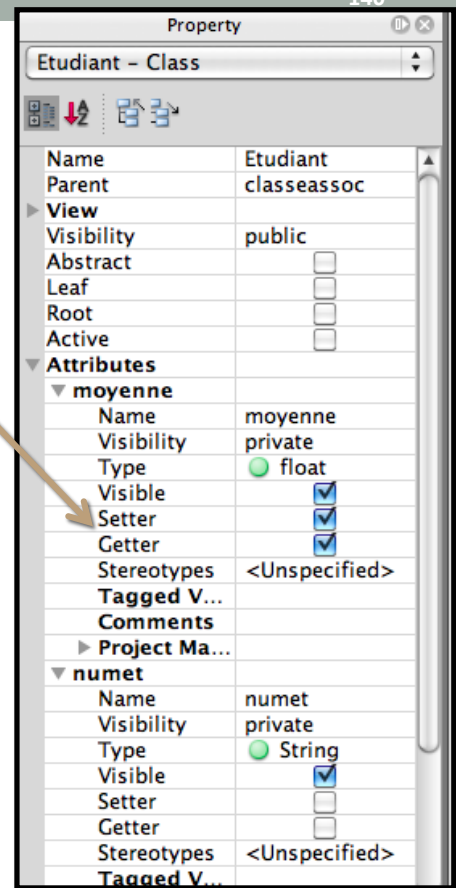
public class Inscription {
    ...
    Etudiant etudiant;
    Module module;
    public Inscription(Etudiant e,
                      Module m) {
        this.etudiant = e;
        this.module = m; ... }
    ...
  
```

Passage UML → code Java

• Classe-Associations

- Génération de code sur VisualParadigm

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

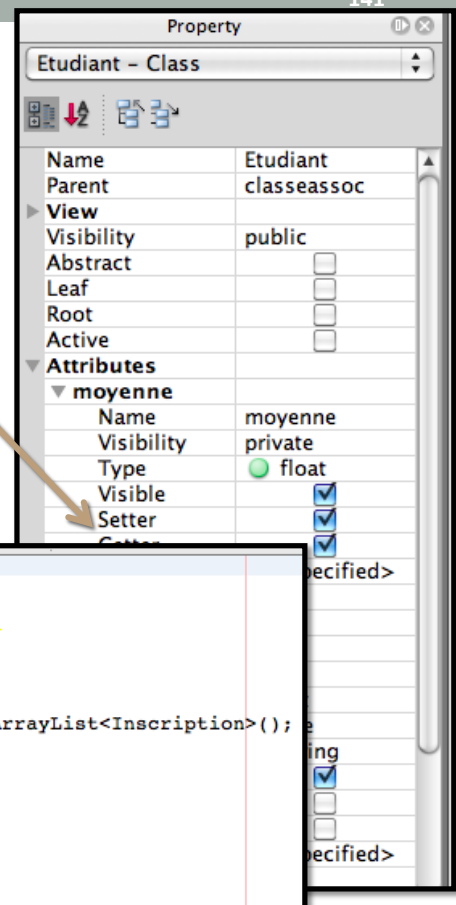


Passage UML → code Java

• 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

```

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 }

```

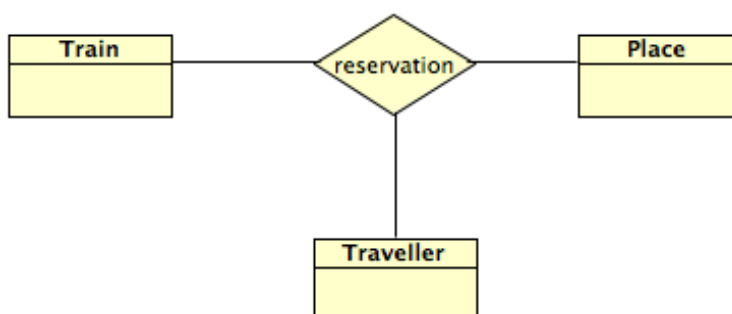
Rôle changeable ou pas ?
 VP génère automatiquement les
 getter/setter pour les rôles

Property
 Etudiant - Class
 Name Etudiant
 Assoc

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;
    }
}

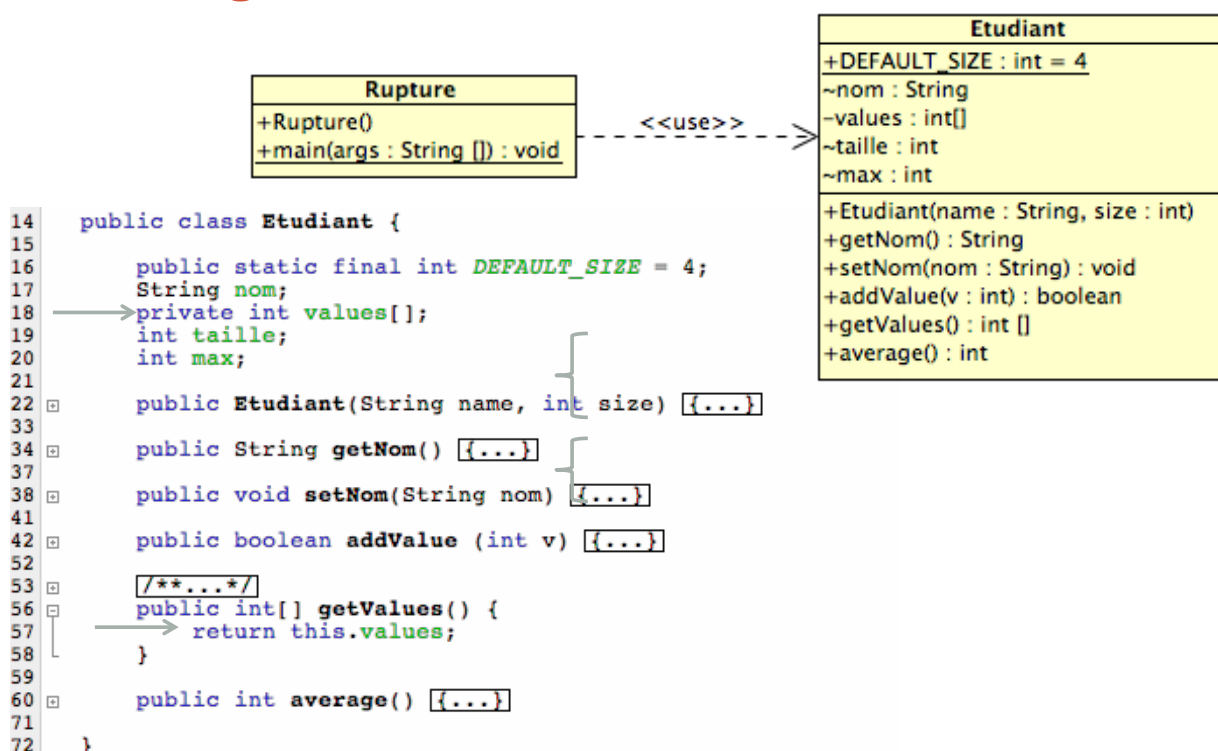
```

...

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

```

13 public class Rupture {
14
15     /**...*/
16     public static void main(String[] args) {
17         Etudiant e = new Etudiant("Toto", 3);
18         e.addValue(18);
19         e.addValue(18);
20         e.addValue(18);
21         System.out.println("Etudiant : "+e.getNom());
22         System.out.println("moyenne : "+e.average());
23
24         String n = e.getNom();
25         n += " Tata ";
26         System.out.println("Etudiant : "+e.getNom());
27
28         int v[] = e.getValues();
29         v[0] = 0;
30         v[1] = 1;
31         System.out.println("moyenne : "+e.average());
32
33         e.nom = "Tata";
34         System.out.println("Etudiant: "+e.getNom());
35     }
36 }
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72

```

```

14 public class Etudiant {
15
16     public static final
17     String nom;
18     private int values[];
19     int taille;
20     int max;
21
22     public Etudiant(String nom, int taille) {
23         this.nom = nom;
24         this.taille = taille;
25     }
26
27     public String getNom() {
28         return nom;
29     }
30
31     public void setNom(String nom) {
32         this.nom = nom;
33     }
34
35     public boolean addValue(int value) {
36         if (value < max) {
37             values[0] = value;
38         }
39     }
40
41     /**...*/
42     public int[] getValues() {
43         return this.values;
44     }
45
46     public int average() {
47         // ...
48     }
49 }

```

```

run:
Etudiant : Toto
moyenne : 18
Etudiant : Toto
moyenne : 6
Etudiant: Tata

```

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