MODÉLISATION DES APPLICATIONS

Manuele Kirsch Pinheiro
Maître de Conférences – Université Paris 1
mkirschpin@univ-paris1.fr / kirschpm@gmail.com
http://www.kirschpm.fr

Objectifs et planning

- Objectifs :
  - Introduction à la modélisation d'applications en UML
- Planning :
  - Introduction à la modélisation
  - Diagrammes de cas d'utilisation
  - Diagramme de classes
  - Passage UML → code (Java)
  - Diagramme de séquence
- Evaluation
  - Exercices / Participation & Devoir maison & Examen final

Références

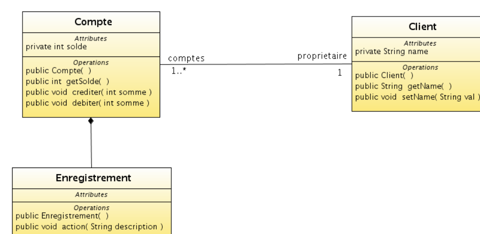
- Bibliographie
  - B. Charroux, A. Osmani, Y. Thierry-Mieg, « UML2 : Pratique de la modélisation », 2e édition, Pearson Education
  - T. Weilkiens, B. Oestereich, « UML2 Certification guide : Fundamentals & intermediate examens », Morgan Kaufmann Publishers / Elsevier

- 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

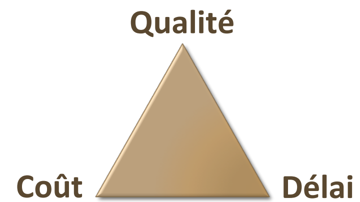
Pourquoi modéliser ?

- Modéliser pour mieux **comprendre**
  - **Maitriser** la complexité et assurer la **cohérence**
  - **Communiquer** autour du système

- Modéliser pour **voir plus loin**
  - **Vue d'ensemble**
  - Visualiser les **interactions** entre les éléments
  - **Évolution** et **maintenance**



Pourquoi modéliser ?



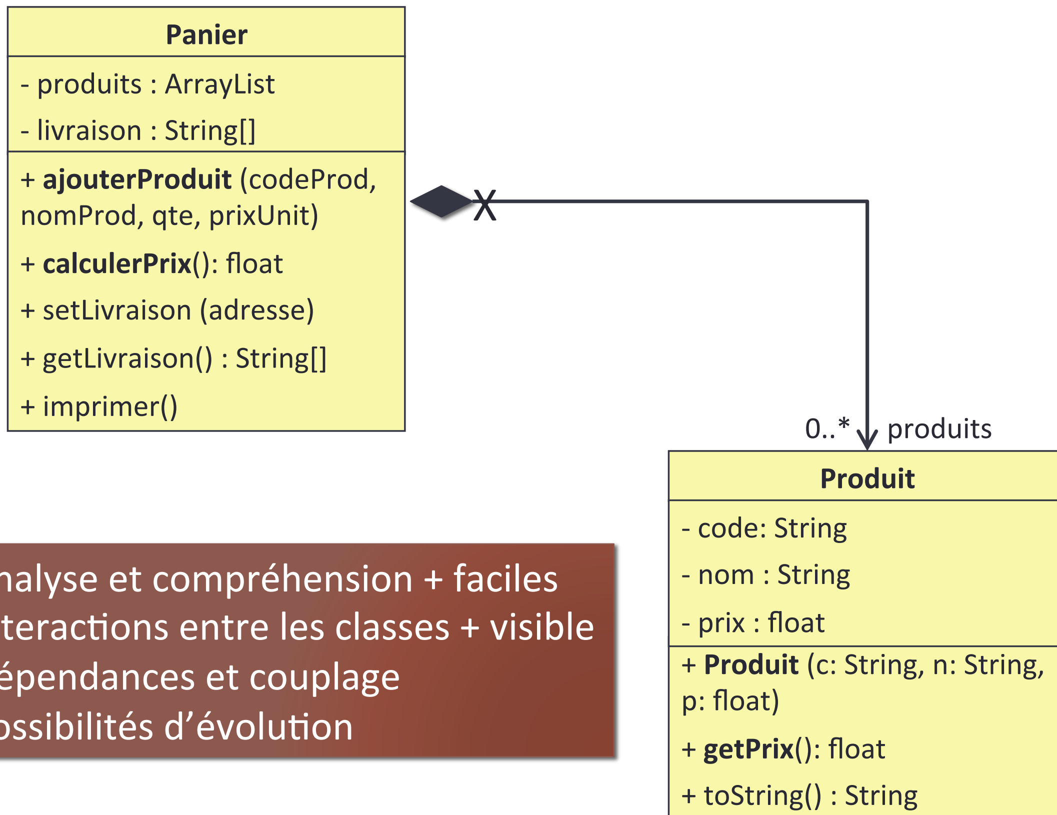
- Équilibre difficile : **Qualité – Coût – Délai**
- **Compromis** entre **performance** et **maintenabilité**
  - **Maintenance** = **80% coût** d'un logiciel

- **Chaos Repport** by Standish Group 2009
  - Seulement **32%** des projets **réussissent** (temps, budget et *features*)
- **Facteurs de réussite**
  - *Implication de l'utilisateur*
  - *Exigences et spécifications claires*

```
public class Panier {
    private ArrayList produits = new ArrayList();
    private String[] livraison = null;
    public void ajouterProduit ( String codeProd, String nomProd, int qte,
                                 float prixUnit) {
        this.produits.add(new Produit(codeProd, nomProd, prixUnit*qte));
    }
    public float calculerPrix() {
        float prix = 0;
        for (int i=0; i<produits.size()
            Produit p = (Produit) prod
            prix += p.getPrix();
        }
        return prix;
    }
    …
```

```
public class Produit {
    private String code;
    private String nom;
    private float prix;
    public Produit(String c, String n, float p) {
        code = c;
        nom = n;
        prix = p;
    }
    public float getPrix() {  return this.prix;  }
    public String toString () {
        return code + " " + nom + " " + prix + "€";  }
}
```

Analyse ?
Compréhension ?
Possibilités d'évolution ?

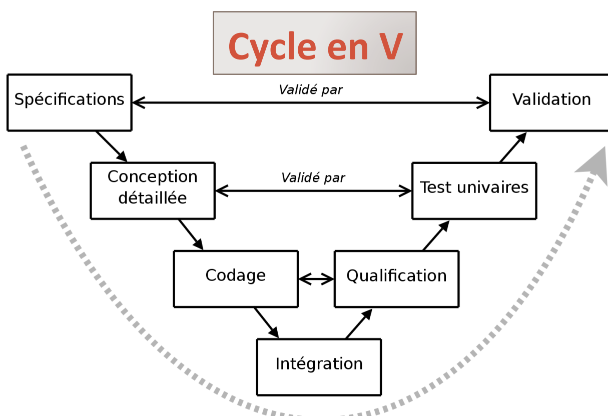


UML

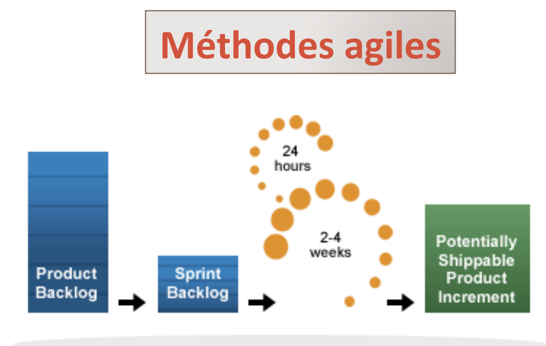
UNIFIED MODELING LANGUAGE™

- **UML** est un langage de **modélisation objets**
  - Une **notation standard** permettant la modélisation d'un système
  - **Description** et **spécification** d'artefacts à forte composante logiciels
- UML n'est **pas une méthode** !
  - UML apporte une aide aux différentes méthodes de conception

Cycle en V



Méthodes agiles



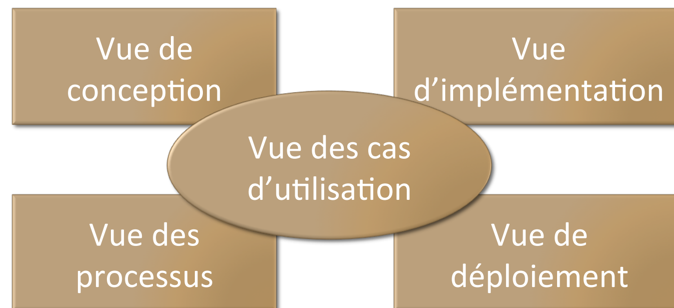
UML

UNIFIED MODELING LANGUAGE ™

- UML offre de multiples diagrammes…

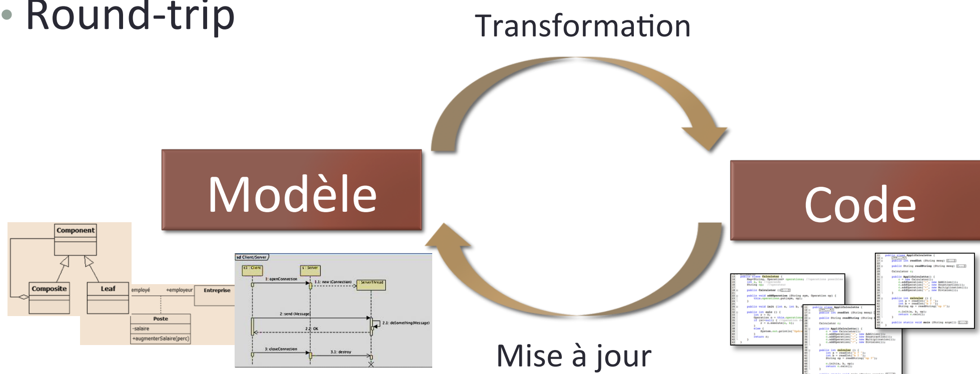
Diagramme de classes	**Diagramme objets**	**Diagramme de cas d'utilisation**	**Diagramme de séquence**
Diagramme de déploiement	Diagramme de composants	Diagramme d'états	Diagramme d'activités

- … offrant différentes vues d'un même système.



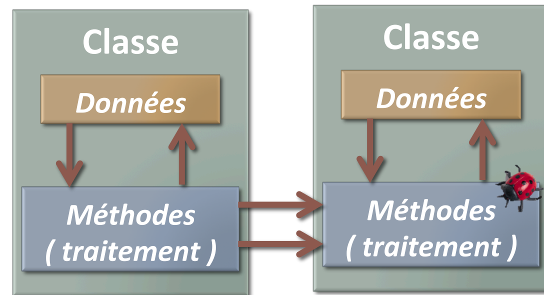
UML et le développement d'applications

- Correspondance modèle ←→ code
  - Génération automatique de code à partir des modèles
  - Les modèles en tant que documentation du code
  - **Traçabilité**
- Round-trip

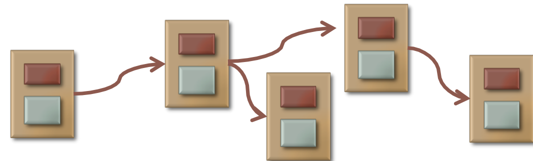


UML et l'orientation à objets

- UML est un langage orienté objets
  - Concepts clés
    - Classes & objets
    - Héritage
  - Modélisation architectures Orientée Objets
    - Interaction entre classes
    - **Complexité d'une application ➔ décomposition en classes réutilisables**



Modéliser pour avoir une Vue d'ensemble



Références

- **http://www.uml.org/**
- http://www.uml-diagrams.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

Objectifs et planning

- Objectifs :
  - Introduction à la modélisation d'applications en UML
- Planning :
  - Introduction à la modélisation
  - **Diagrammes de cas d'utilisation**
  - Diagramme de classes
  - Passage UML → code (Java)
  - Diagramme de séquence

Diagramme de cas d'utilisation

- **Objectifs**
  - Identification des **fonctionnalités** nécessaires au système
  - Passage **des besoins aux fonctionnalités** (souvent suite au cahier de charges)

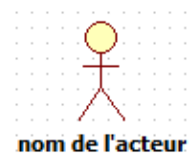
- Un diagramme de cas d'utilisation décrit
  - Ce qui **doit faire** le système, **sans dire comment** le faire
  - Ce qui attend l'utilisateur

➢ **Point de vue utilisateur**

Diagramme de cas d'utilisation

- Eléments clés : les **acteurs** et les **cas d'utilisation**

- **Acteurs**
  - **Élément extérieur** au système qui interagit avec celui-ci
  - Élément extérieur qui **sollicite** le système ou **est sollicité** par lui



« actor »
Nom de l'acteur

- **Cas d'utilisation (use case)**
  - **Fonctionnalité** que le système doit accomplir
  - **Ensemble d'actions** réalisées par le système
  - **Service rendu** par le système

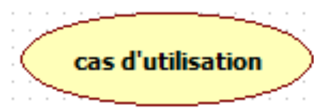


Diagramme de cas d'utilisation

- Exemple

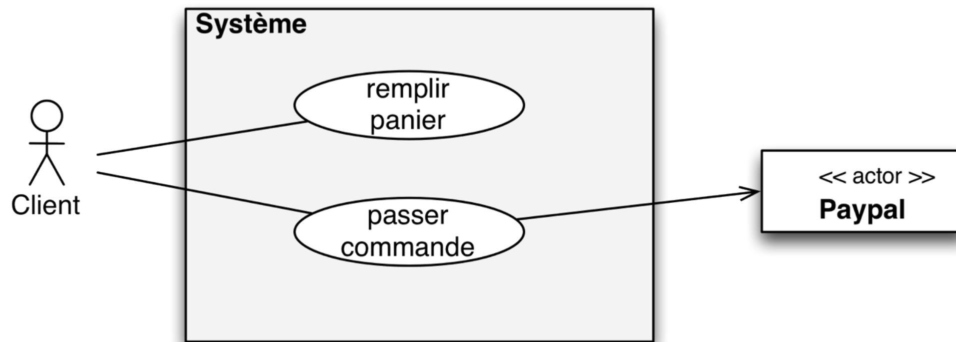


Diagramme de cas d'utilisation

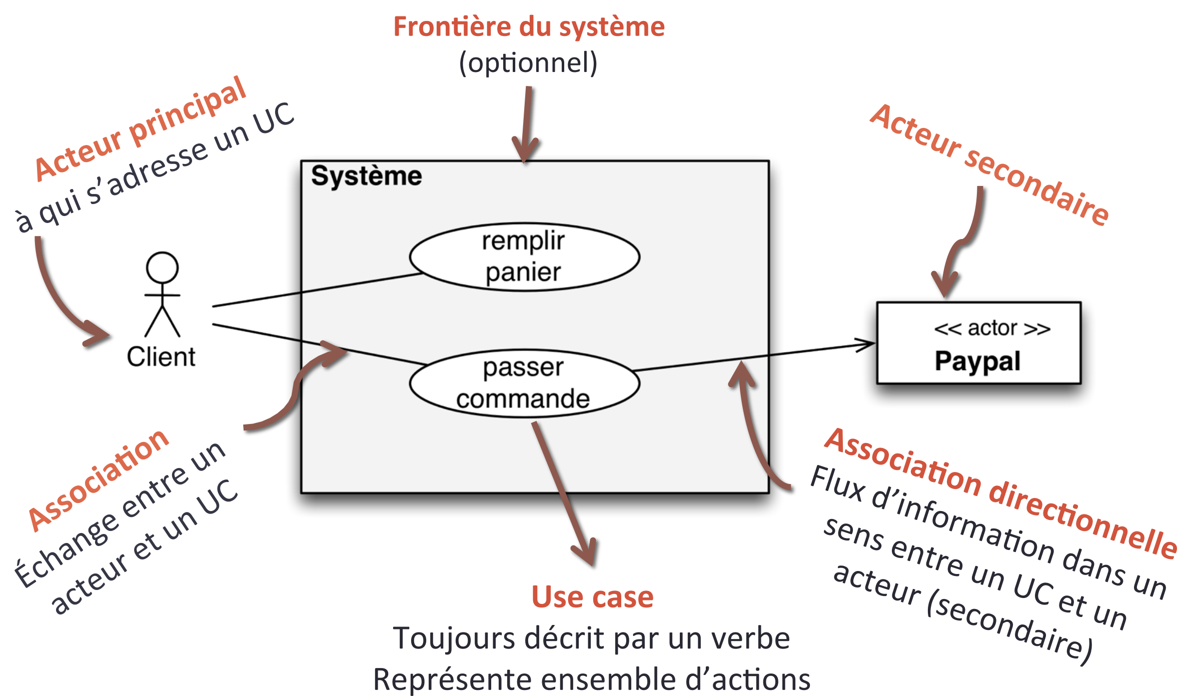


Diagramme de cas d'utilisation

- Certaines fonctionnalités nécessitent la réalisation d'autres fonctionnalités pour bien fonctionner
- **Dépendance de type « includes »**
  - Un UC a besoin d'un autre UC pour réussir
  - **Pas d'ordre temporel entre les UC**

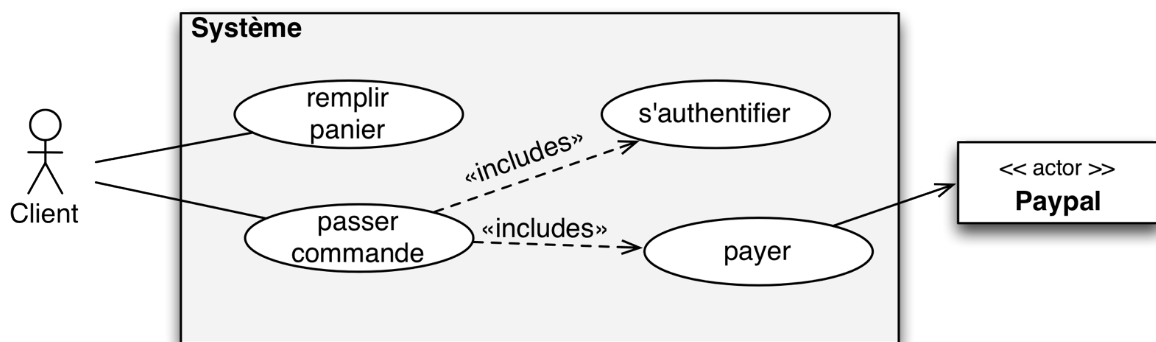


Diagramme de cas d'utilisation

- Certaines fonctionnalités peuvent être étendues par d'autres
- **Dépendance de type « extends »**
  - Un UC étend les fonctionnalités d'un autre UC
  - Définition d'un *point d'extension*
  - **Comportement optionnel**

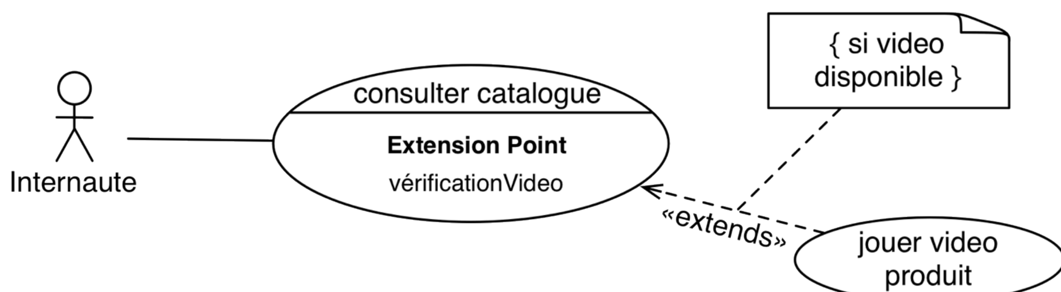
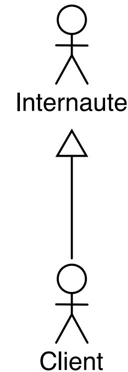


Diagramme de cas d'utilisation

- Les acteurs et les cas d'utilisation peuvent également être **généralisés**
- **Héritage**
  - **Un acteur qui spécialise un autre acteur**
    - Un client **est un** internaute
    - Tous les UC associés à un internaute sont disponibles pour un client



  - **Un UC qui spécialise un autre UC**
    - Deux **modes différents** de réaliser une fonctionnalité

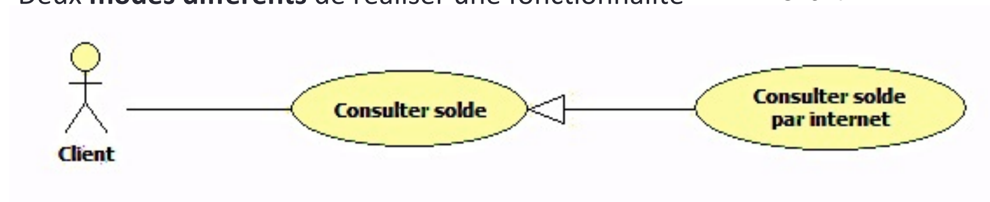


Diagramme de cas d'utilisation

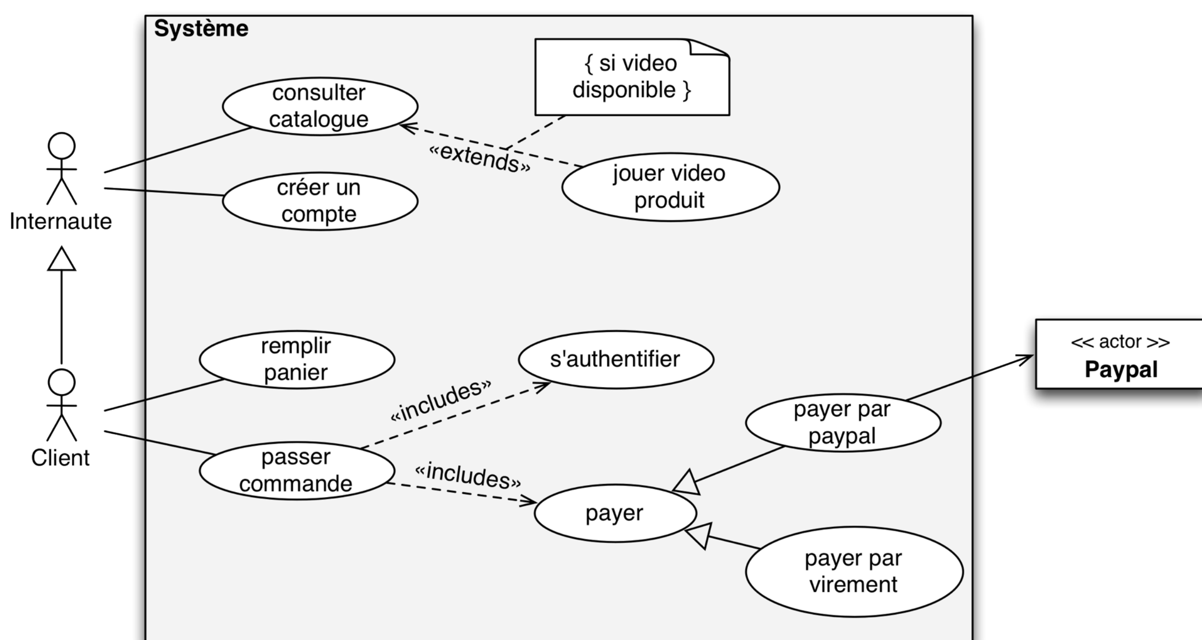
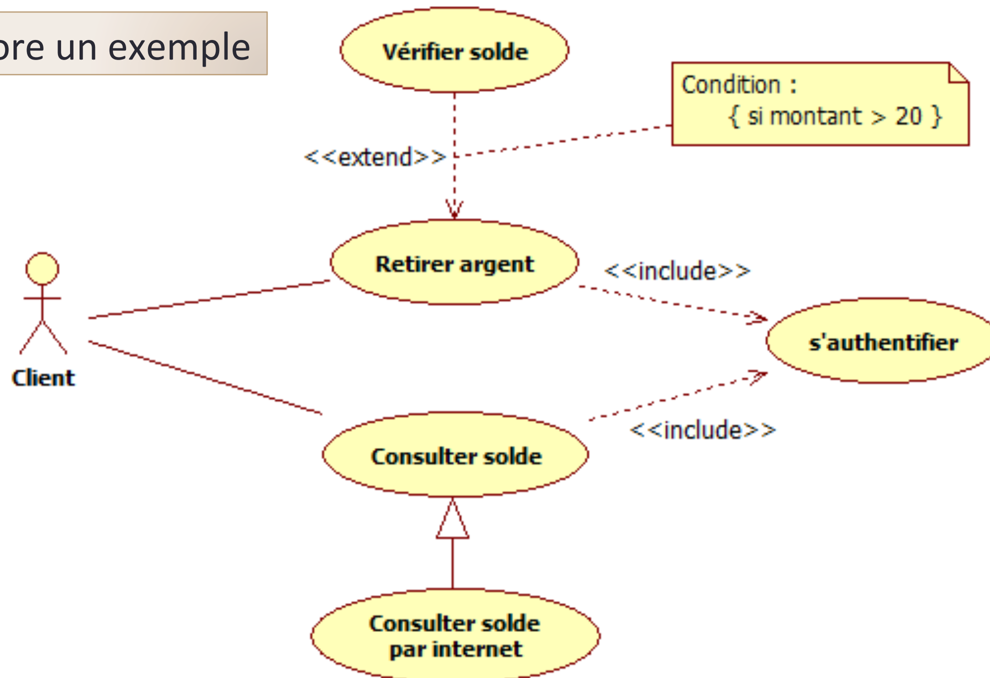


Diagramme de cas d'utilisation

Encore un exemple



Diagrammes cas d'utilisation

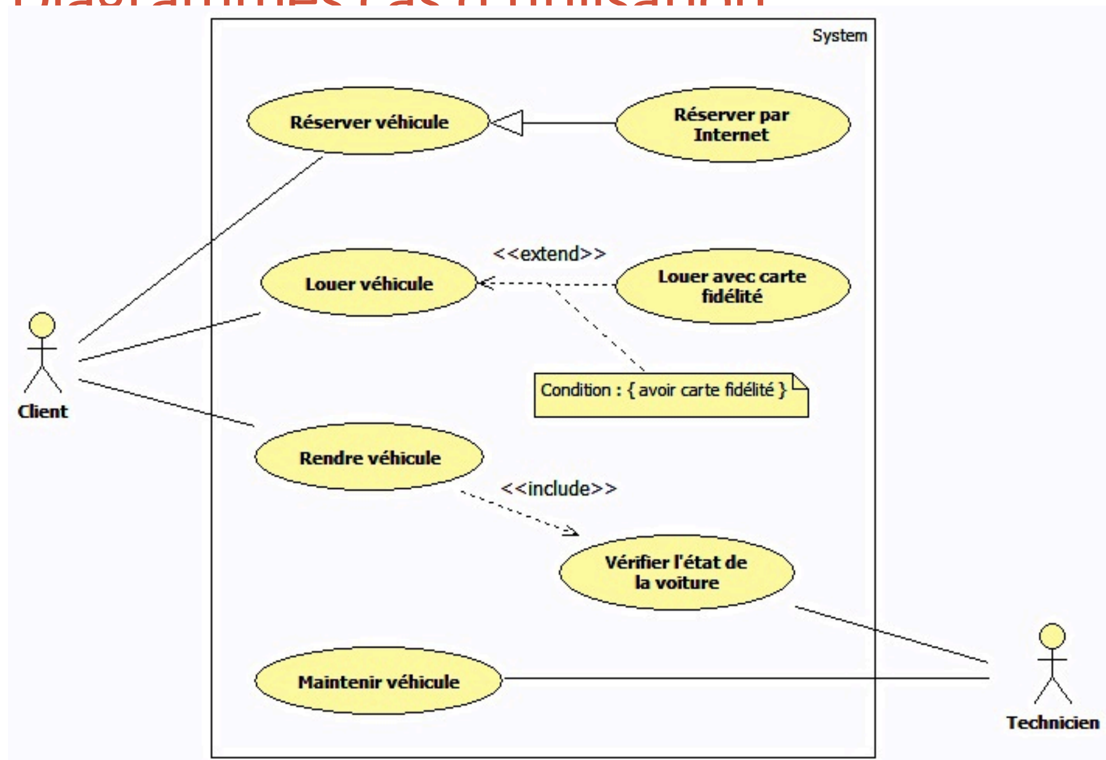
- Scénario : Agence location de voiture

  « Un client arrive à l'agence de location de voiture. Il peut y louer un véhicule. Il peut aussi rendre un véhicule loué ou encore réserver un véhicule pour une future location. Le client peut également réserver par internet son véhicule.

  Au moment de louer le véhicule, si le client possède une carte de fidélité de l'agence, il aura des avantages clients qui ne sont pas disponibles aux clients sans cette carte.

  Lors que le client rend le véhicule, celui-ci est vérifié par le technicien de l'agence. Le technicien de l'agence s'assure également la maintenance des voitures louées. »

Diagrammes cas d'utilisation



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 :
  - Introduction à la modélisation d'applications en UML

- Planning :
  - ✓ Introduction à la modélisation
  - ✓ Diagrammes de cas d'utilisation
  - **Diagramme de classes**
  - Passage UML → code (Java)
  - Diagramme de séquence

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

Résumé

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

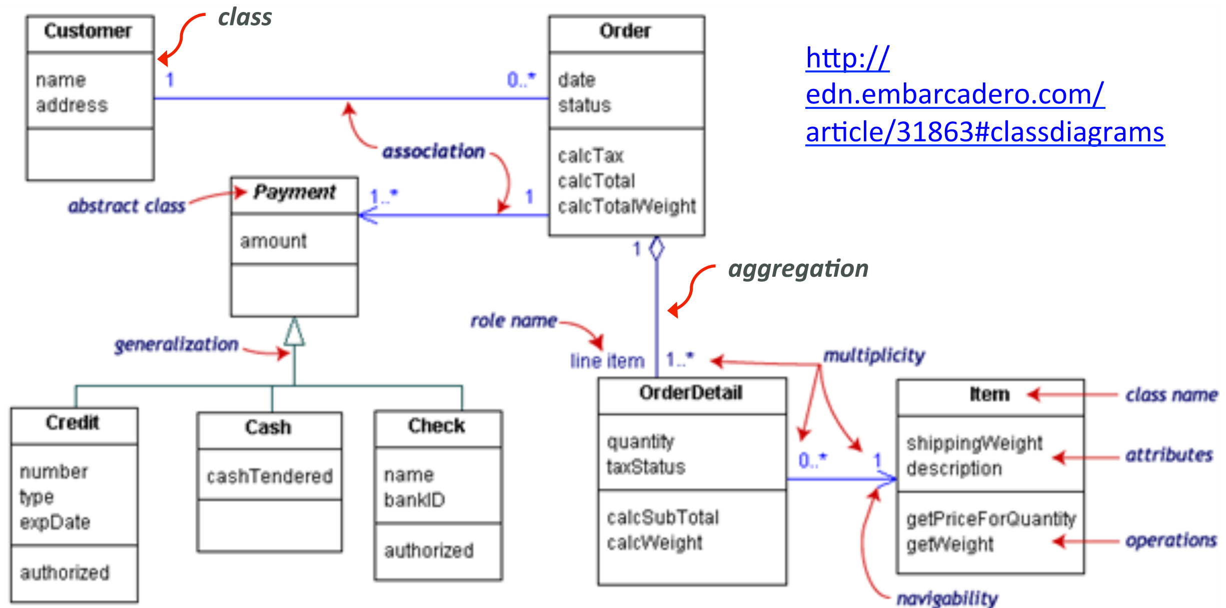


Diagramme de classes : classe

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

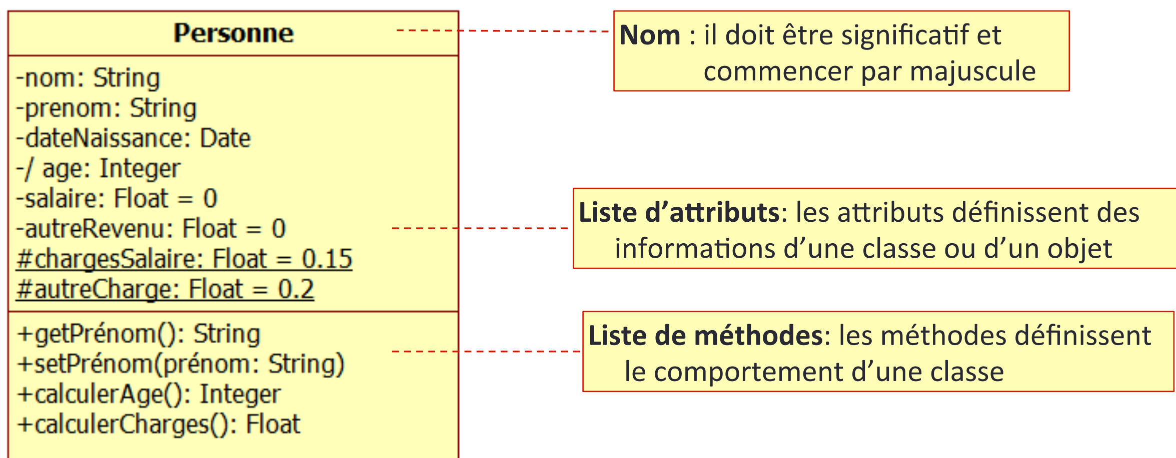


Diagramme de classes : classe

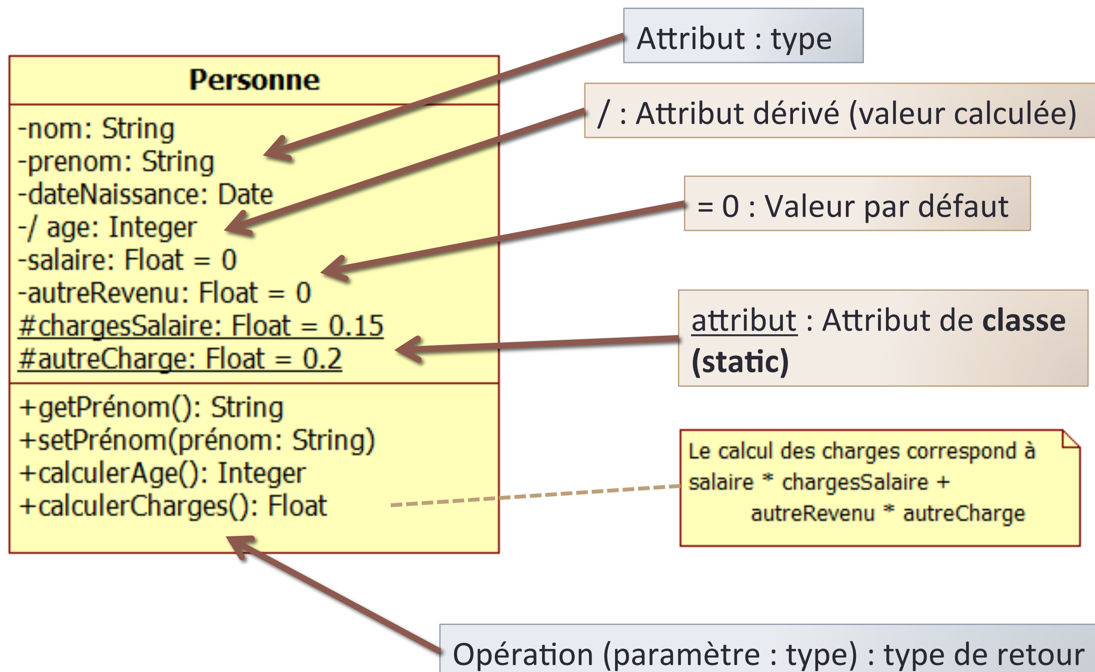


Diagramme de classes : classe

- **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

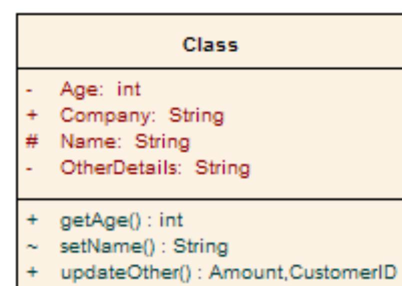


Diagramme de classes : classe

- **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)***

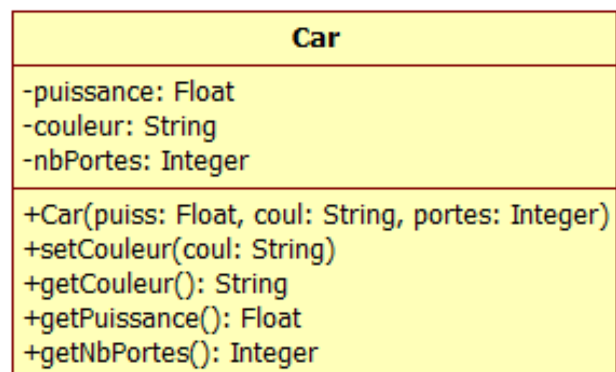


Diagramme de classes : héritage

- **Héritage** (généralisation / spécialisation)
  - **Réutilisation** d'une classe pour la création d'une nouvelle
  - **Relation de classification** entre un élément ⊕ général et un élément ⊕ 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

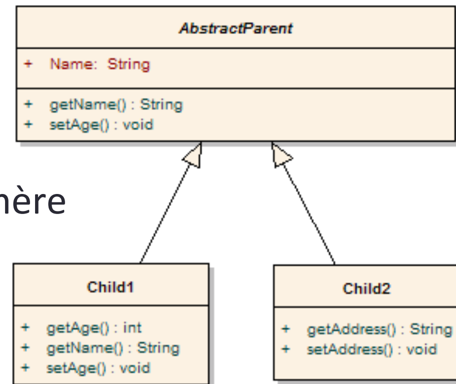


Diagramme de classes : associations

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

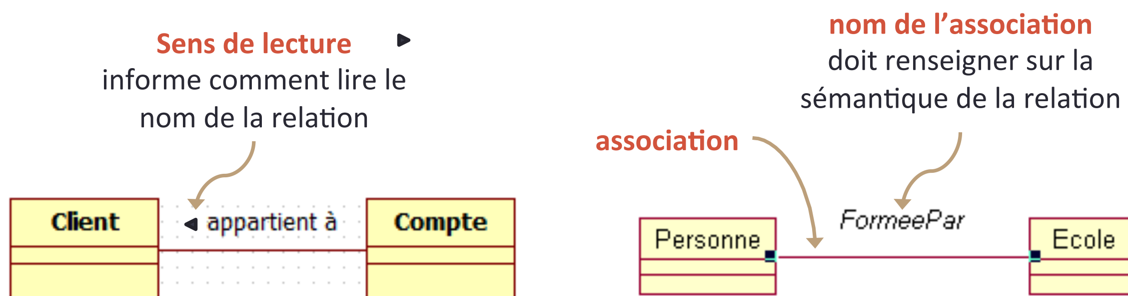


Diagramme de classes : associations

- **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

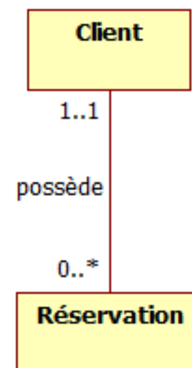


Diagramme de classes : associations

- **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 polygone connaît ses sommets, mais les points ne connaissent pas s'il appartiennent ou pas à un polygone
  - À ne pas confondre avec le sens de lecture !!

Diagramme de classes : associations

- **Associations binaires et n-aires**
  - **Binaires** : relient 2 classes entre elles

  - **N-aires** : relient plus de 2 classes entre elles
    - Plus rares , peu utilisées
    - Plus difficiles à mettre en œuvre
    - **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*** ?

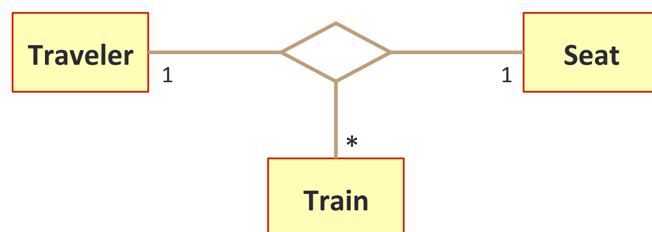
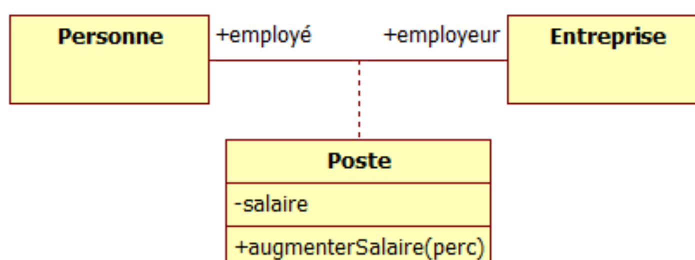


Diagramme de classes : classe-association

- **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***

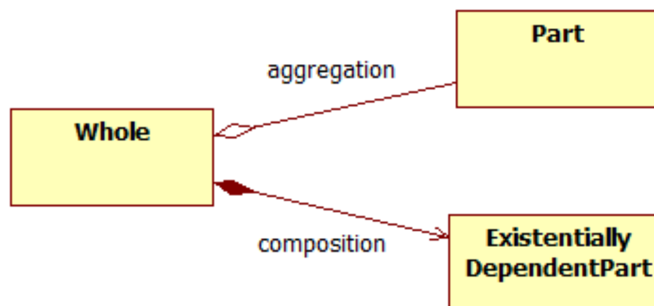


Diagramme de classes : agrégations

- *Pattern* **Composite**
  - Ex.: Une arbre qui contient des branches, qui elles-aussi contiennent d’autres branches, jusqu’aux feuilles

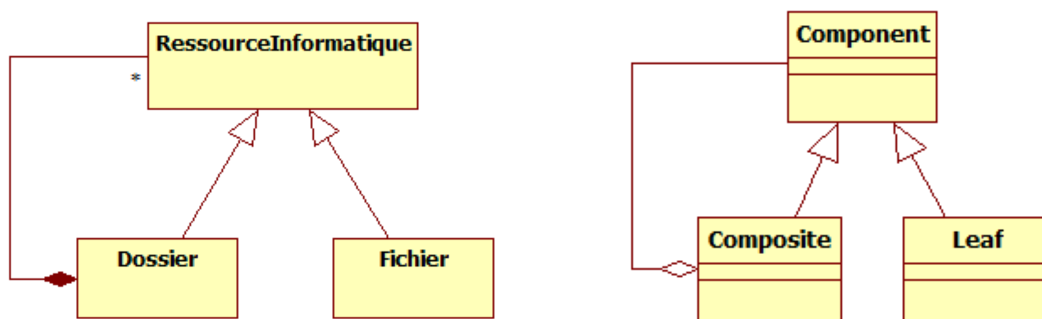
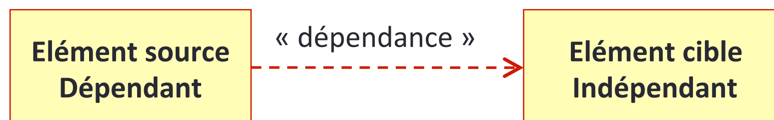


Diagramme de classes : dépendances

- **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 »***…)



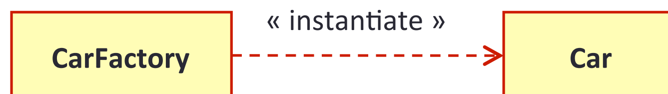


Diagramme de classes : contraintes

- **Contraintes**
  - ***Condition*** qui doit être ***vérifiée*** par les ***éléments*** d'un modèle
  - On peut associer des contraintes à n'importe quel élément du modèle
    - *Attribut, méthodes, associations…*
  - Contraintes prédéfinis en UML
    - {*order*} , {*unique*} , {*subset*}, {*complete*}, {*incomplete*}

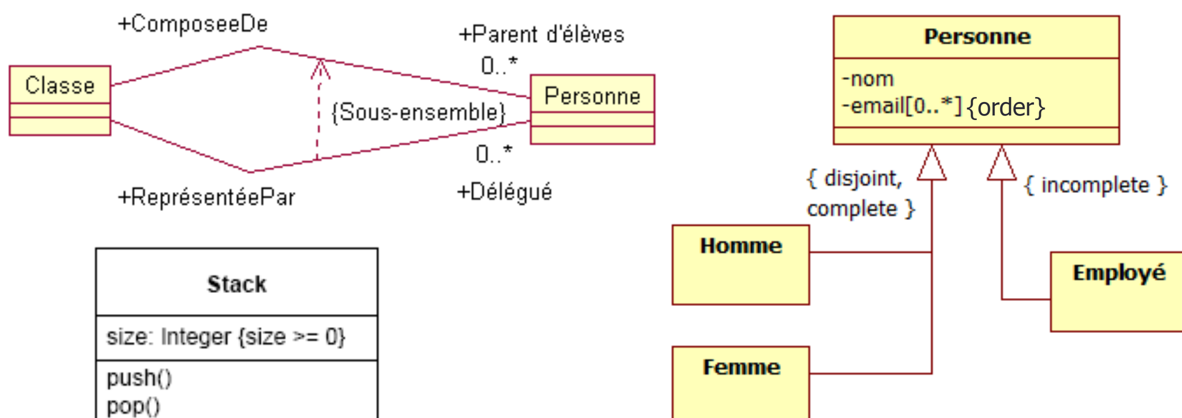
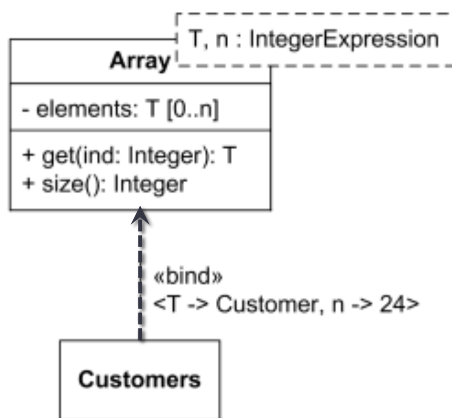


Diagramme de classes : classes paramétrables

- **Classes paramétrables**
  - Une classe paramétrable (ou générique) permet le remplacement d'un (ou plusieurs) type(s) générique(s) par un type concret lord de son utilisation



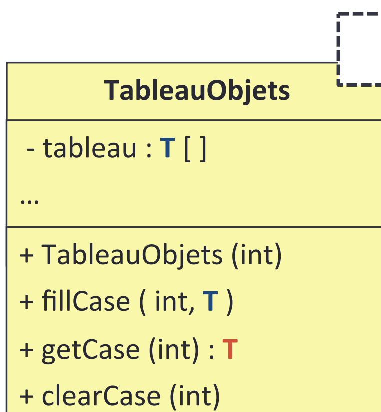
- Spécification de « **paramètres** » qui seront spécifiés lors de **l'usage** de la classe

- Exemple:
  - Un Array de type « T » et de taille « n »
  - Pour la classe « Customers », « T » équivaut à « Costumer » et n = 24

Diagramme de classes : classes paramétrables

- Classe paramétrables en Java

Code + **lisible** avec
– de **transtypage**



Bind <T = Integer >

Vérification pendant la **compilation**

TableauObjets**<Integer>** tab =
new TabeauObjets**<Integer>**(x);

```
public class TableauObjetParametre<T>
…
    private T[] tableau;
…
    this.tableau = (T[]) new Object[this.taille];
```

tab.fillCase (0, new Integer(x));

tab.fillCase (1, "String");

Impossible !!
Erreur de compilation

Diagramme de classes : classes abstraites

- **Classes abstraites**
  - Les classes abstraites représente 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 »**

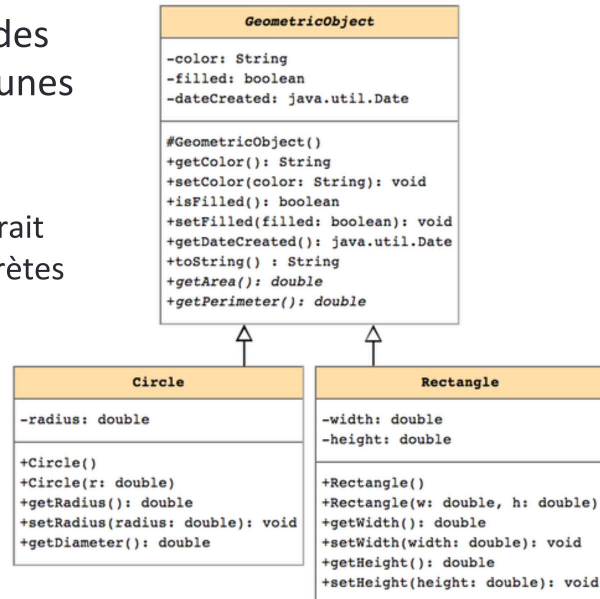
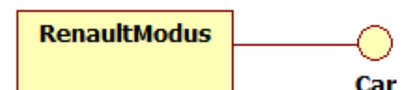


Diagramme de classes : interfaces

- **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**



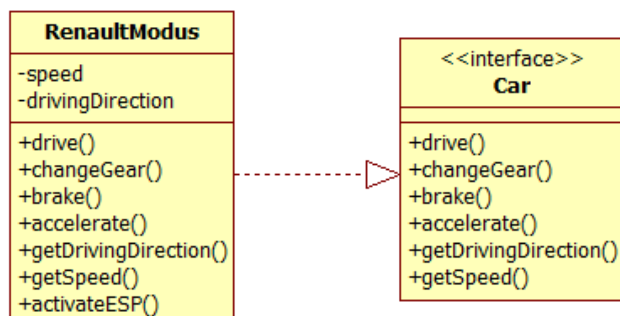


Diagramme de classes : interfaces

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

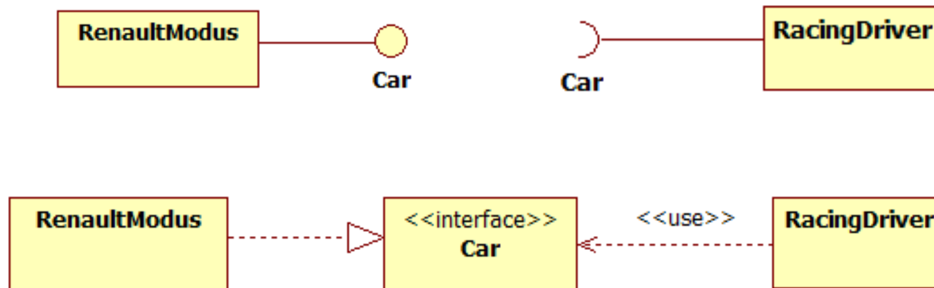
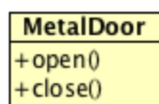


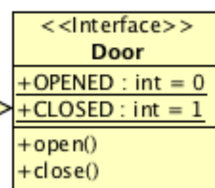
Diagramme de classes : interfaces

- **Interface & réutilisation**
  - Une interface introduit un **point de variation**
    - On peut changer les clients (ceux qui utilisent l'interface) aussi bien que les fournisseurs (ceux qui l'implémentent)
  - **Faible couplage** entre les classes

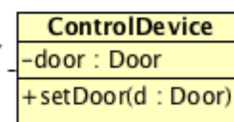
Pas de connaissances sur le **client** : on peut le remplacer sans impacter la classe

Référence uniquement à l'**interface** : on peut **changer l'implémentation sans impact** sur la classe





<<use>>




```
public class MetaDoor
            implements Door {
   public void open() { … }
   public void close() { … }
… }
```

```
public class ControlDevice {
   private Door door ;
   public void setDoor(Door d) {… }
… }
```

Diagramme de classes : interfaces

- **Interface & réutilisation**
  - **Faible couplage**
    - Implémentation réelle « **injectée** » de l'extérieur

```
public class ControlDevice {...
  public void setDoor(Door d) {... }
}
public class MetaDoor
          implements Door { ... }
```

```
public class Main {
  private ControlDevice cd ;
  ...
  cd.setDoor (new MetalDoor() );
...
```

  - **Héritage**
    - L'héritage entre interfaces reste possible
      - Nouvelles opérations
      - Surcharge

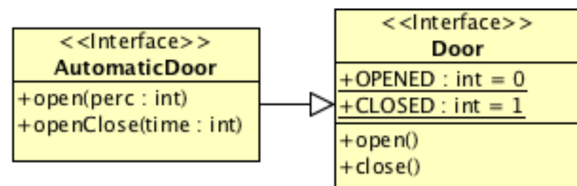
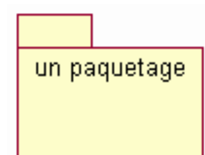


Diagramme de classes : paquetages

- **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

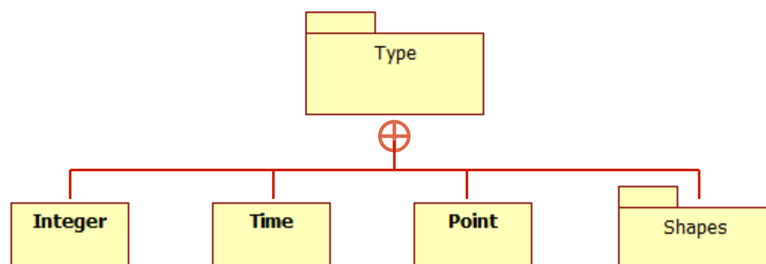


Diagramme de paquetages

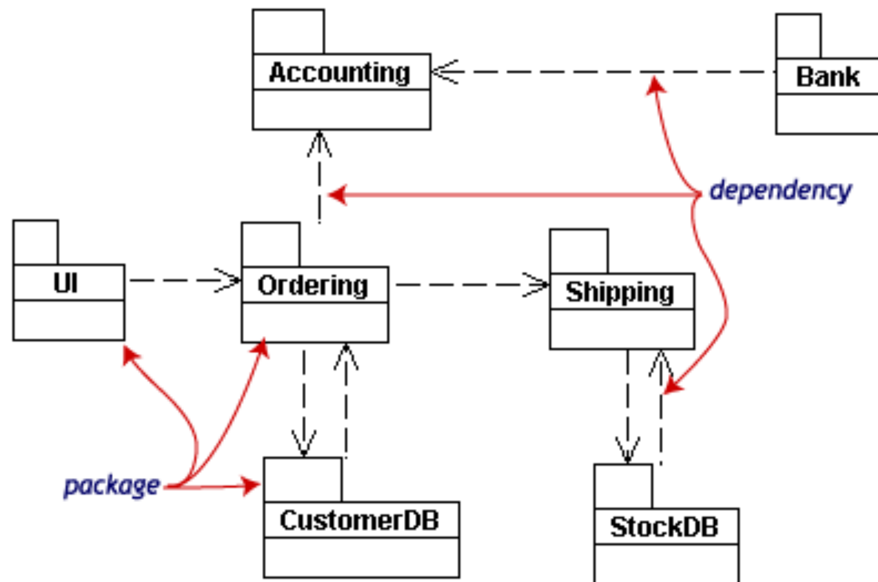
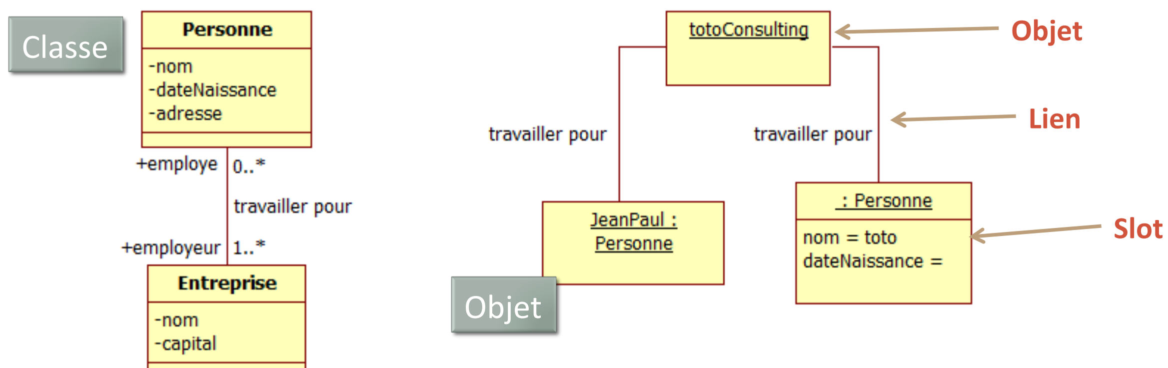


Diagramme objets

- 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

Objectifs et planning

- Objectifs :
  - Introduction à la modélisation d'applications en UML
- Planning :
  - ✓Introduction à la modélisation
  - ✓Diagrammes de cas d'utilisation
  - ✓Diagramme de classes
  - ✓Passage UML → code (Java)
  - **Diagramme de séquence**

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…

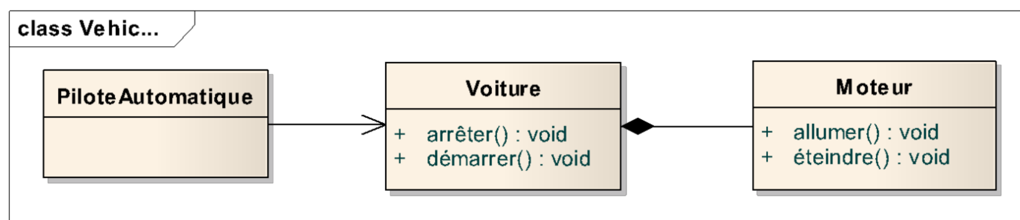


Diagramme de séquence

Le diagramme de séquence montre les interactions sous un angle temporel

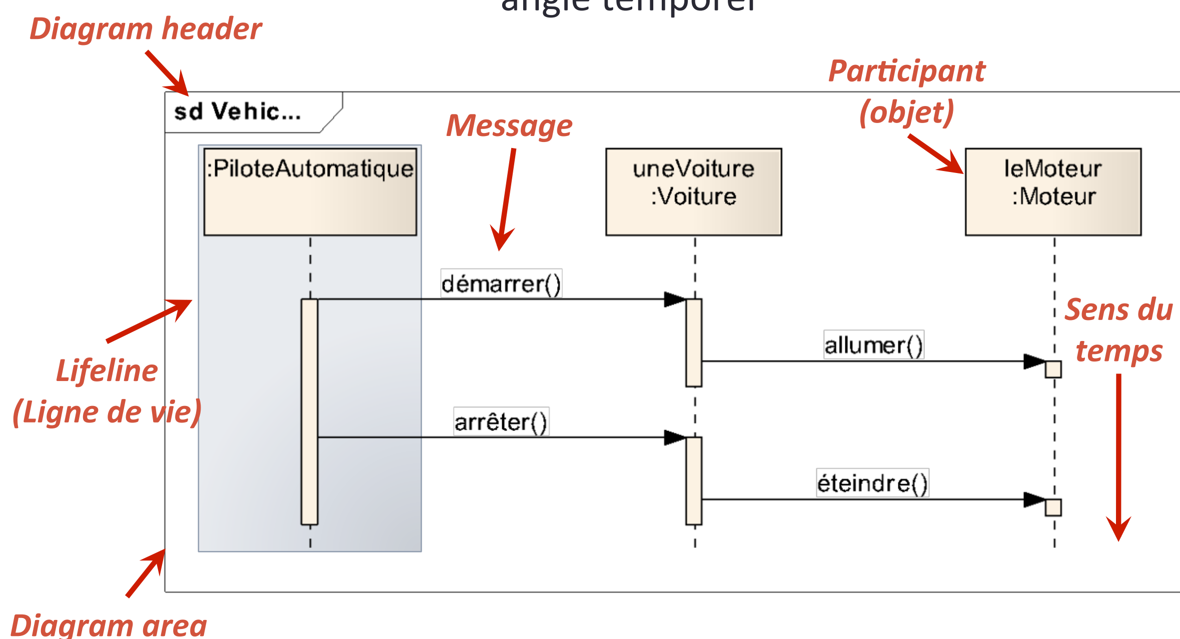


Diagramme de séquence

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

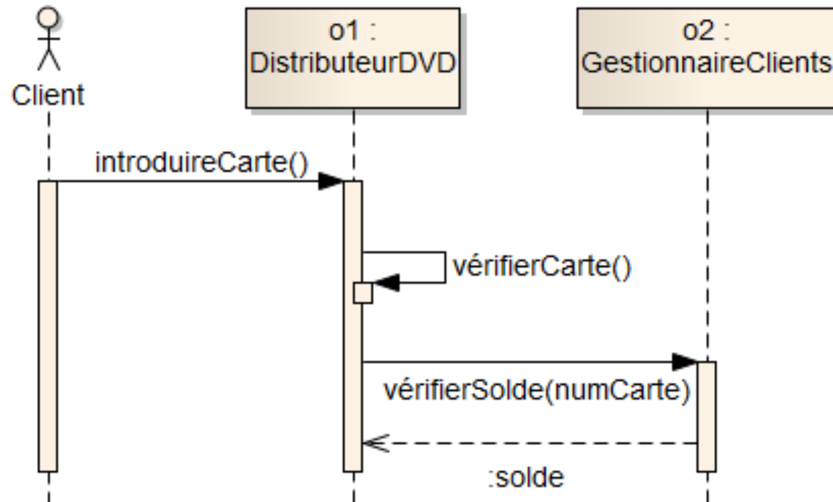


Diagramme de séquence

On déroule un scénario

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

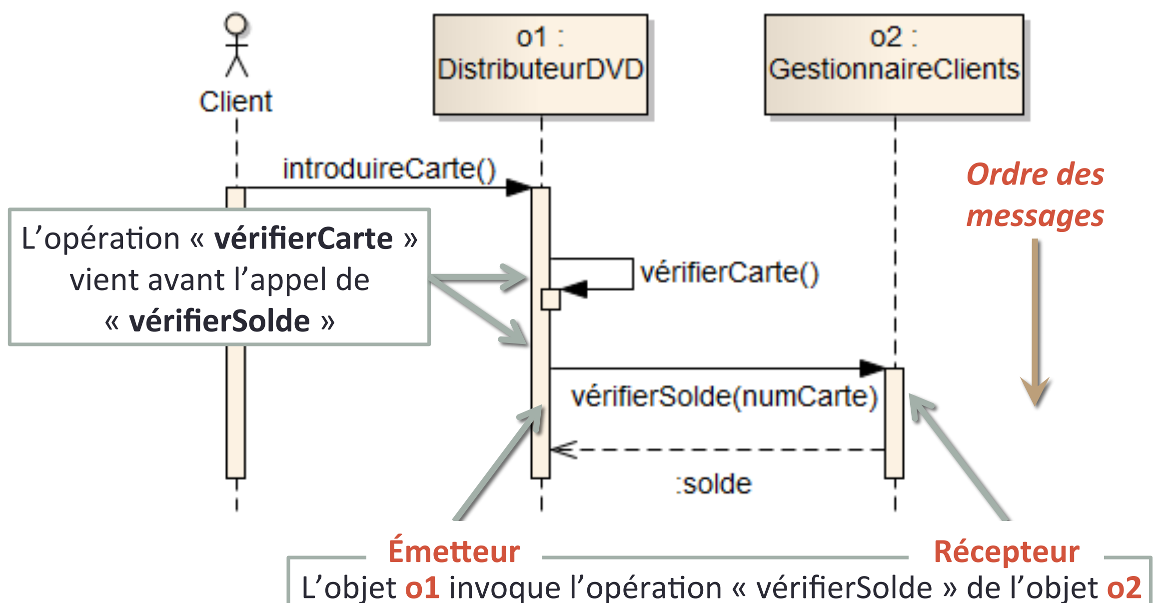


Diagramme de séquence

- 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

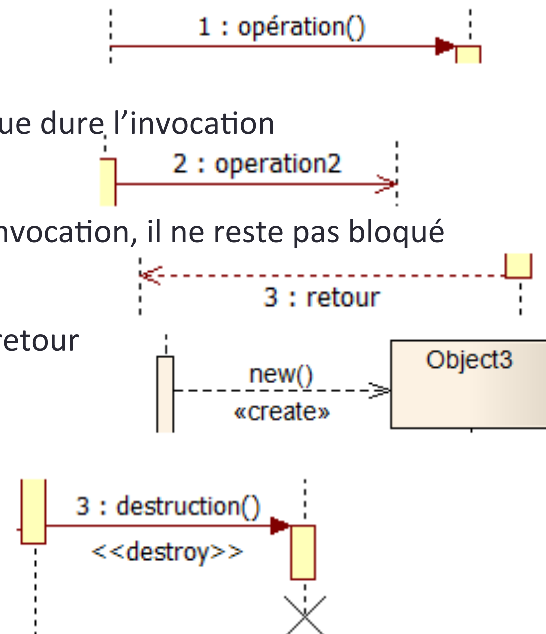


Diagramme de séquence

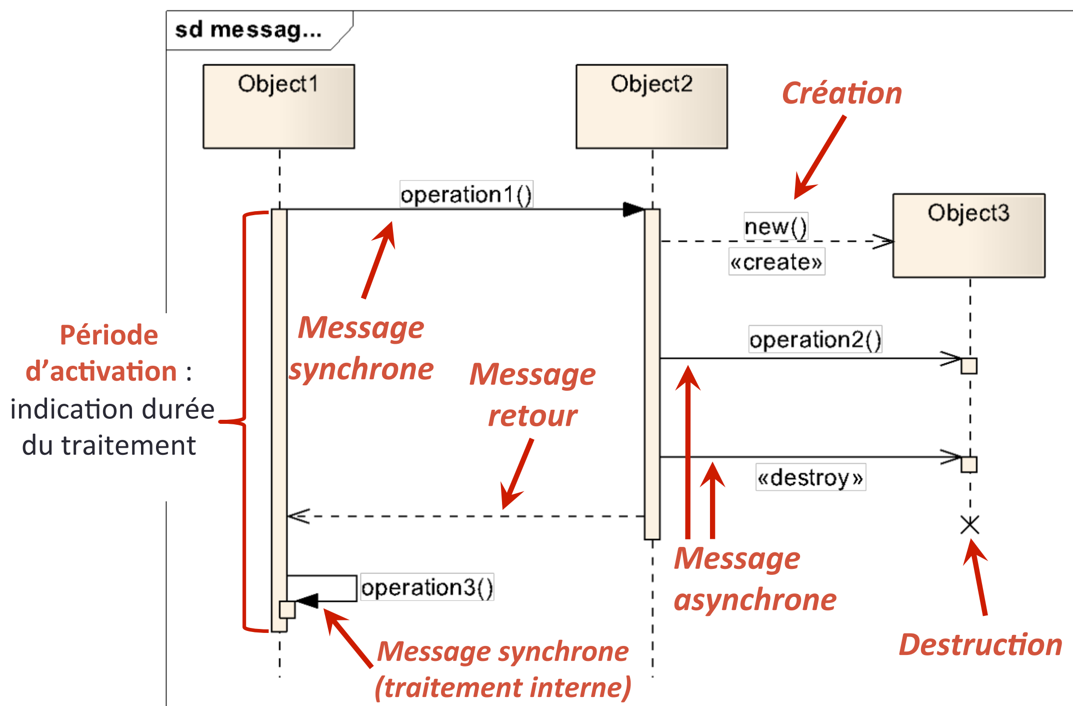


Diagramme de séquence

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

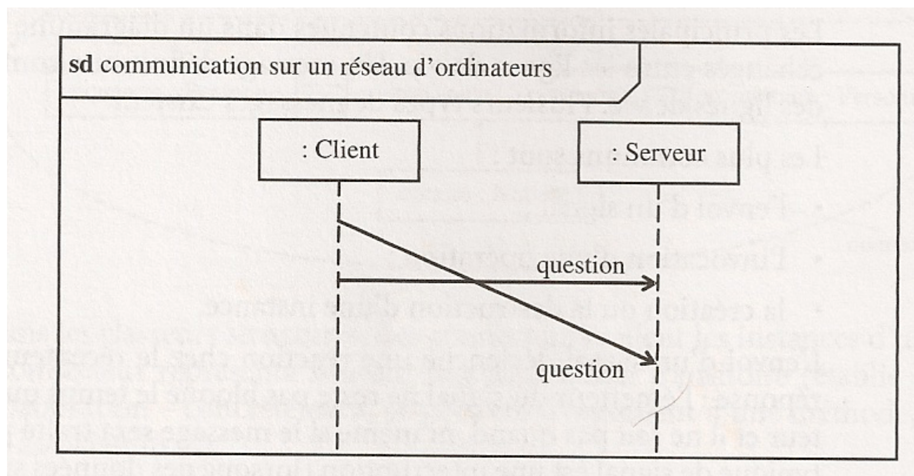
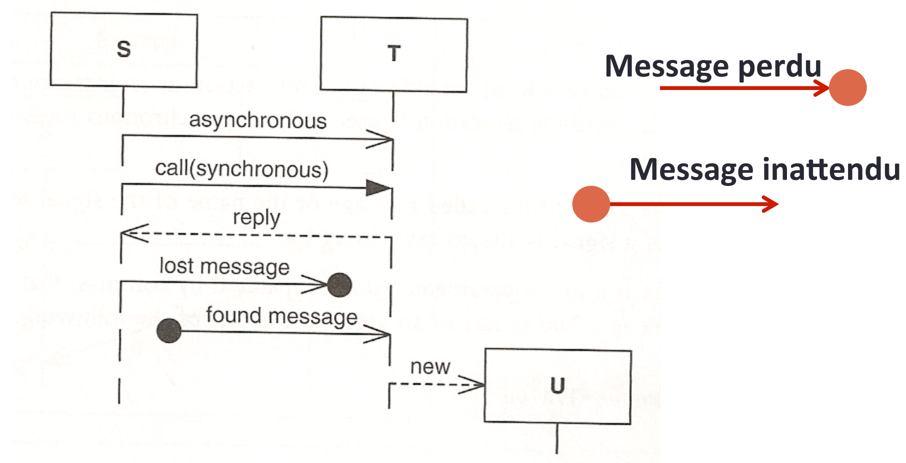


Diagramme de séquence

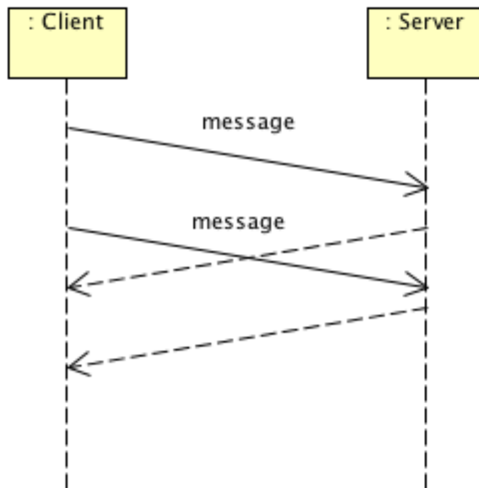
- **Messages perdus**
  - UML permet d'indiquer la perte d'un message, ou encore la réception d'un message inattendu



Message perdu

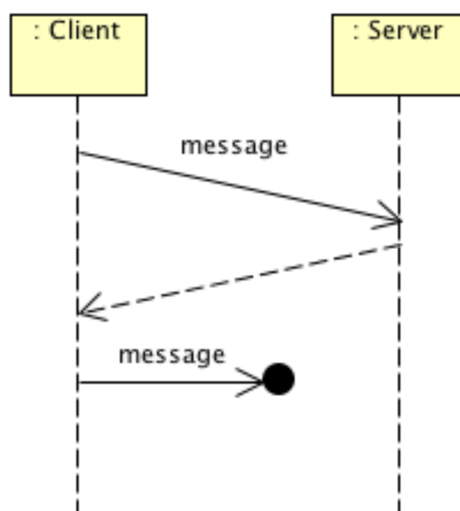
Message inattendu

Diagramme de séquence



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

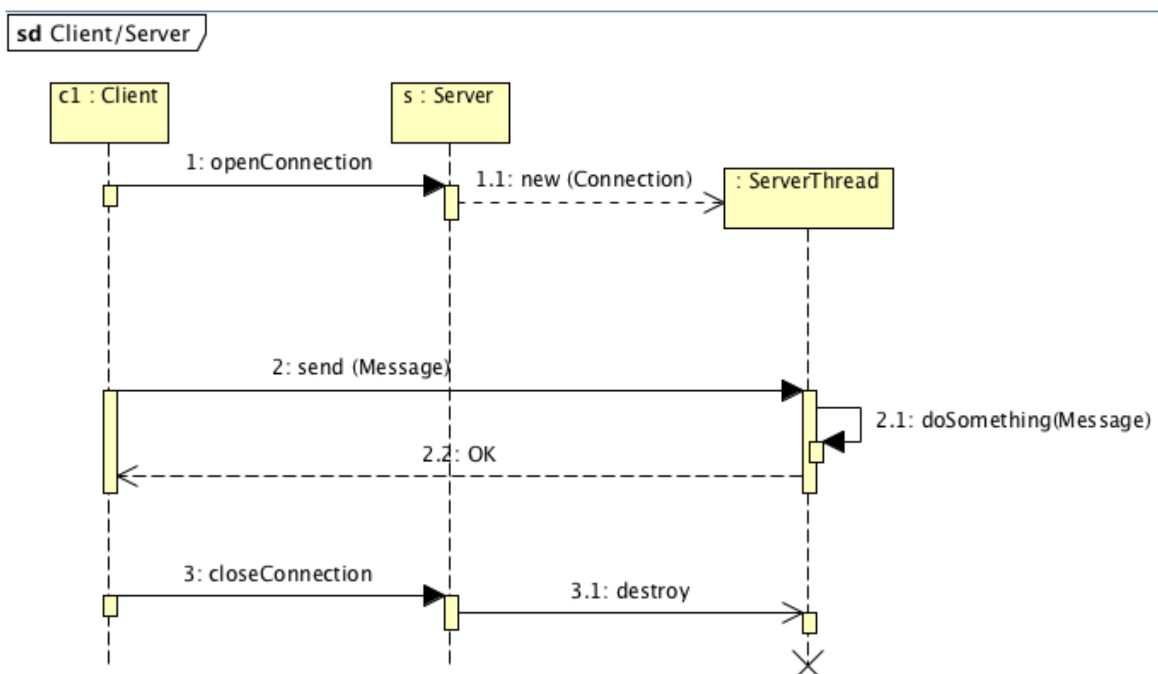


Diagramme de séquence

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

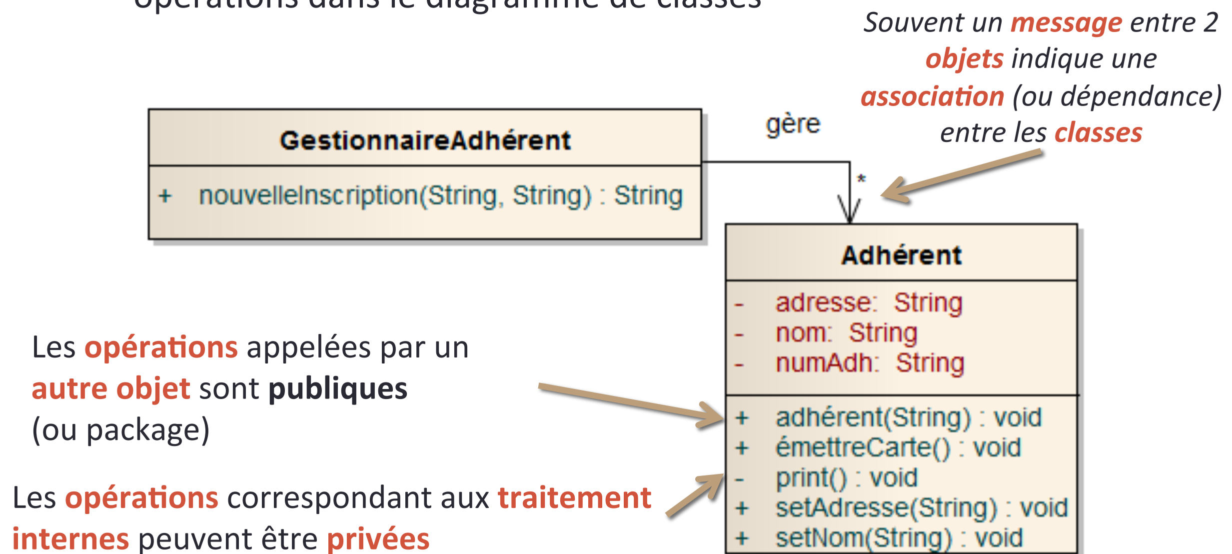


Diagramme de séquence

- **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…

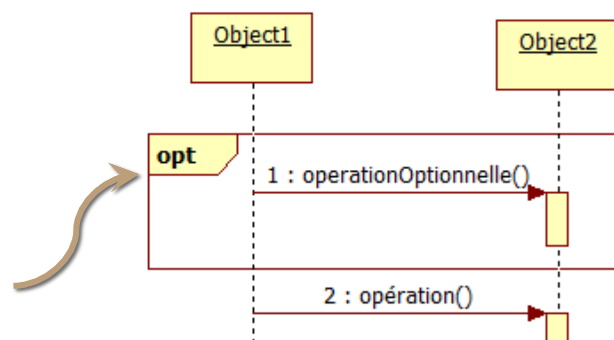


Diagramme de séquence

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

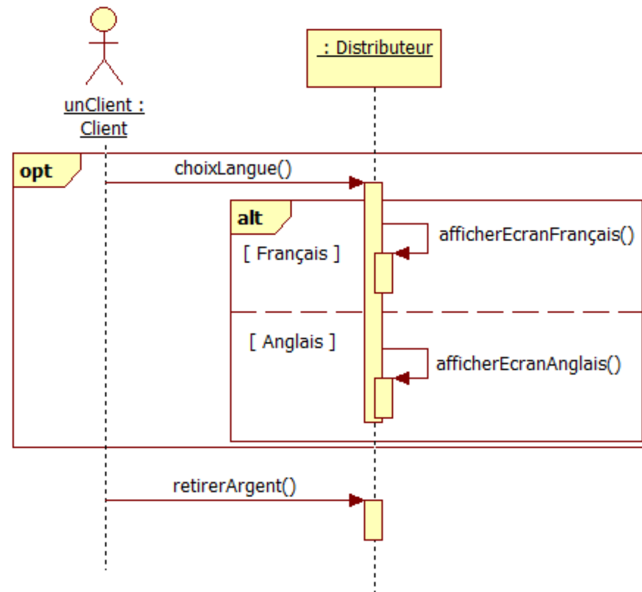


Diagramme de séquence

- **Fragment d'interaction**
  - **Par** indique que l'envoie des messages se déroule en parallèle

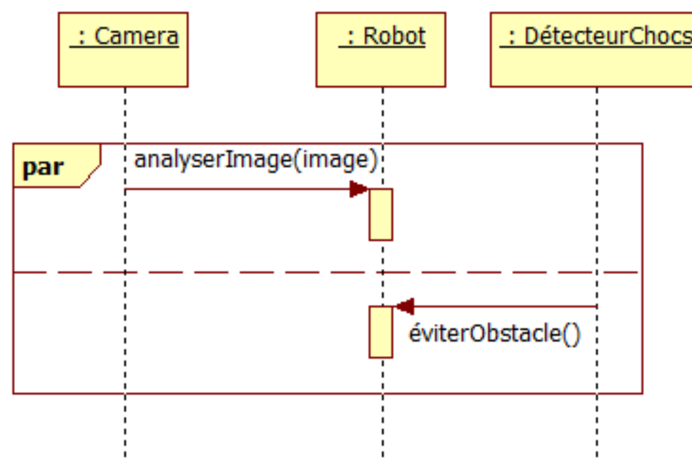


Diagramme de séquence

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

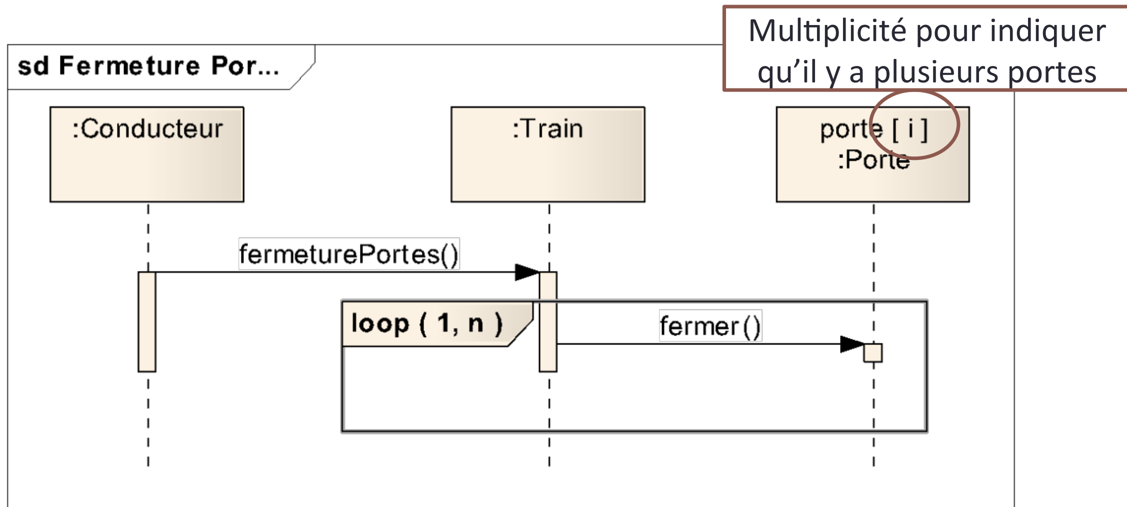


Diagramme de séquence

- **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

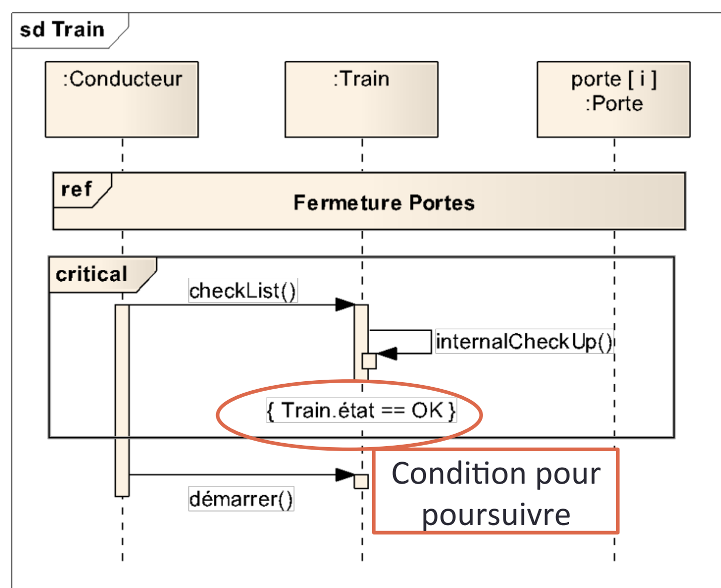
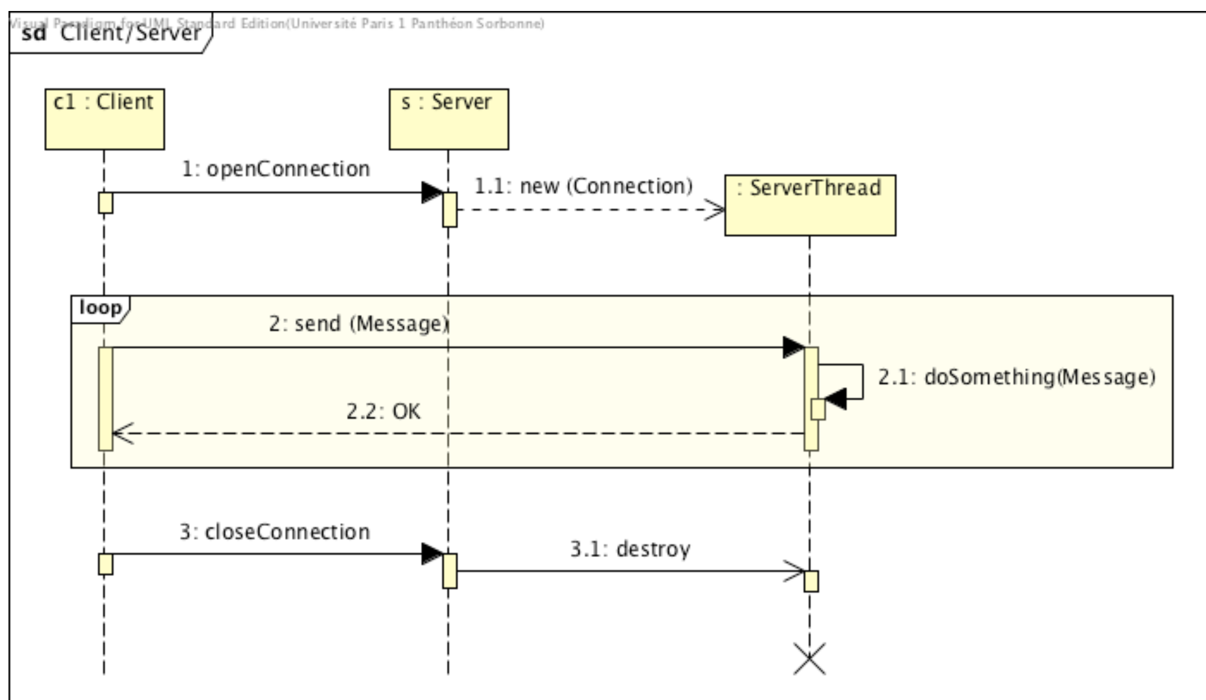


Diagramme de séquence

- **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



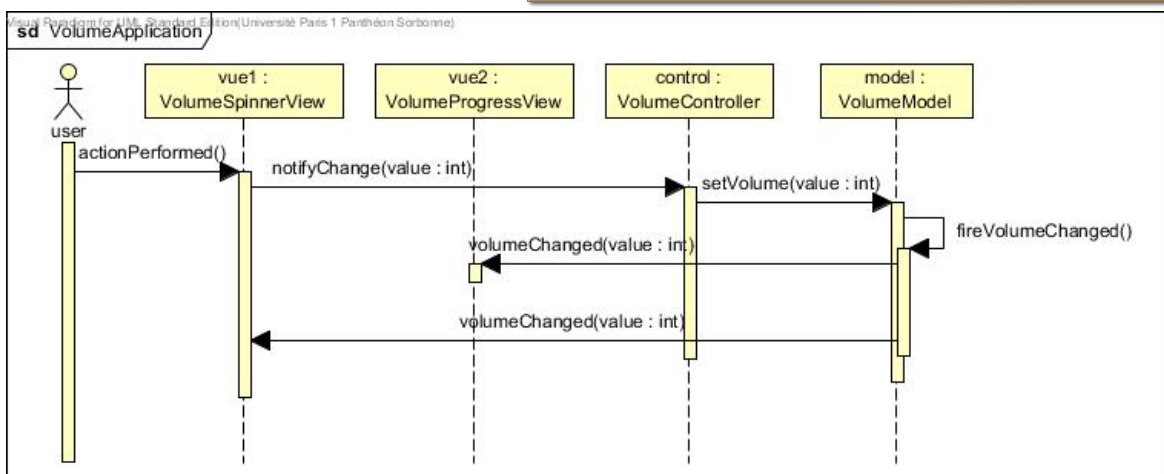
Interaction Diagramme de séquence - Diagramme de classes

- 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

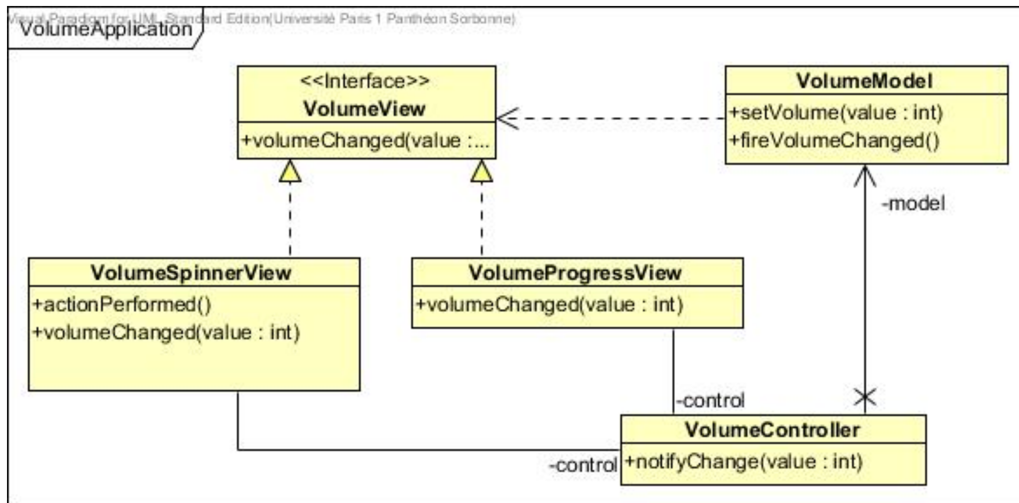
Exemple : Application de gestion de volume

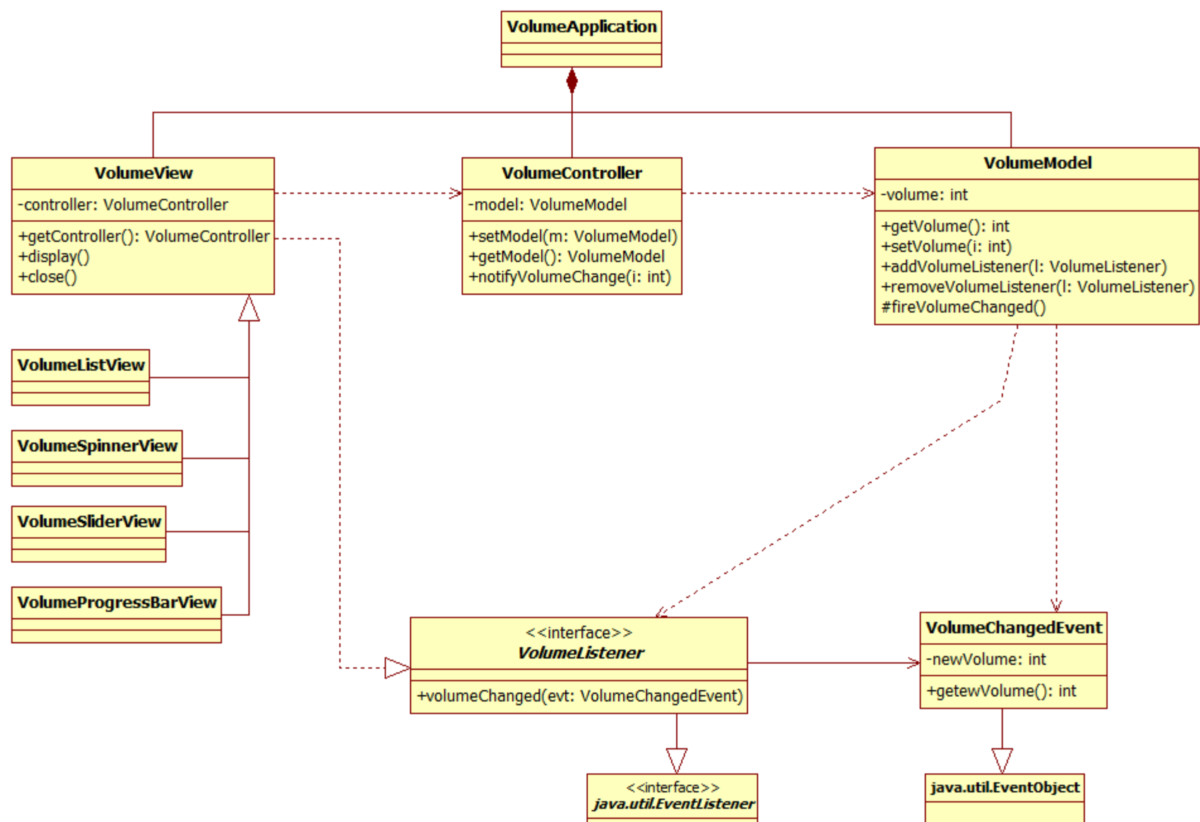


Quelles classes participent à cette interaction ?
Quelles opérations proposent-elles ?
Pouvons-nous définir des interfaces communes à ces classes ?

Interaction Diagramme de séquence - Diagramme de classes

Exemple : Application de gestion de volume





MODÉLISATION DES APPLICATIONS

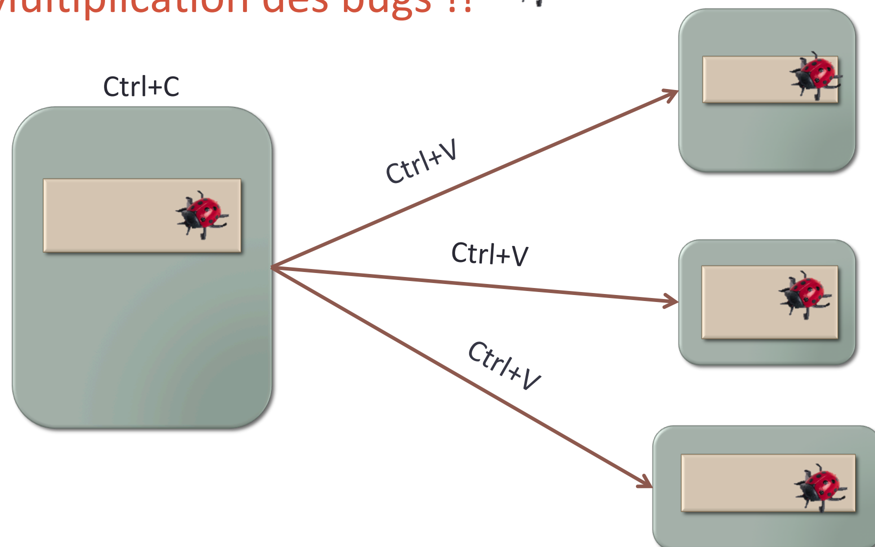
Manuele Kirsch Pinheiro
Maître de Conférences – Université Paris 1
mkirschpin@univ-paris1.fr / kirschpm@gmail.com
http://mkirschp.free.fr

Objectifs et planning

- Objectifs :
  - Introduction à la modélisation d'applications en UML
- Planning :
  - 10h (CM / TP) en 5 séances
    - ✓ Séance 1 : Introduction à la modélisation
    - ✓ Séance 2 : Diagrammes de cas d'utilisation
    - ✓ Séance 3 : Diagramme de classes
    - **Séance 4 : Passage UML → code (Java)**
    - Séance 5 : Diagramme de séquence
- Evaluation
  - Exercices / Participation & Devoir maison & Examen final

Concevoir un code de qualité

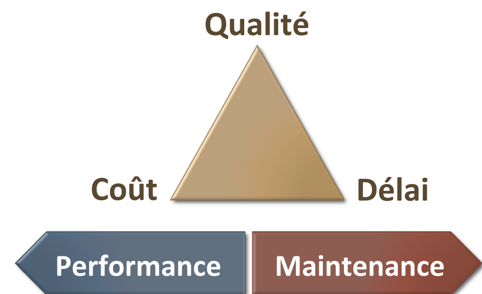
- Problème de la stratégie du « copier-coller »
  - ➢ Multiplication des bugs !!



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

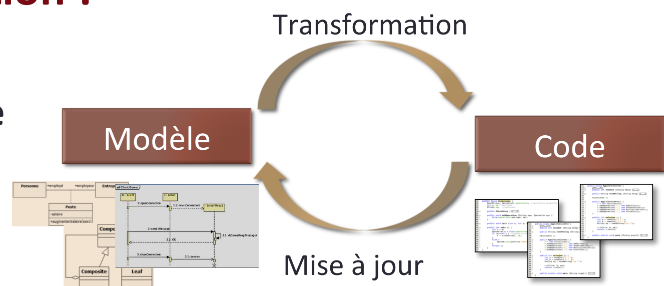
Modélisation des applications



- Développement de qualité
  - **Penser qualité**
  - **Vision globale**
    - Solution à court terme X solution à long terme

➢ **Besoin de modélisation !**

✧**Correspondance modèle ←→ code généré**



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**

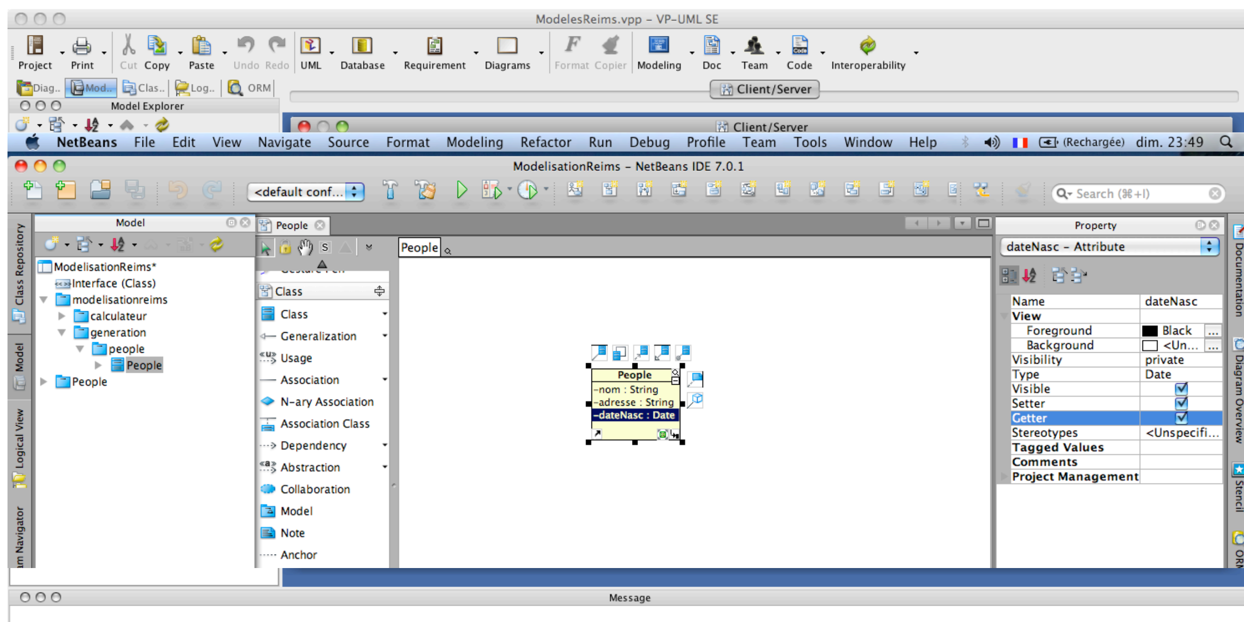
- Au-delà des applications, comment se traduit-il un diagramme de classes en code Java ??
  - Quelques « patterns » peuvent être observés

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

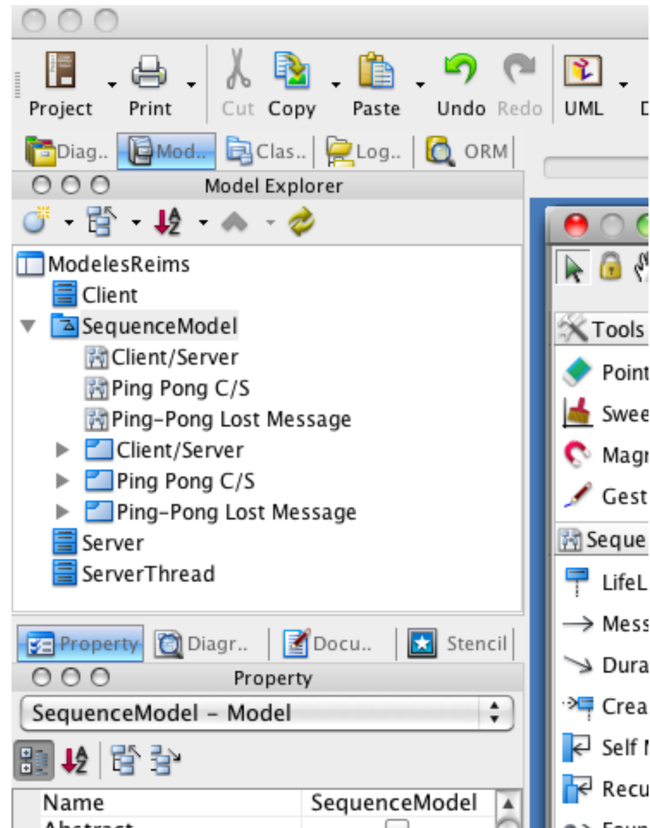
Applications génération de code

- Exemple d'application de génération de code



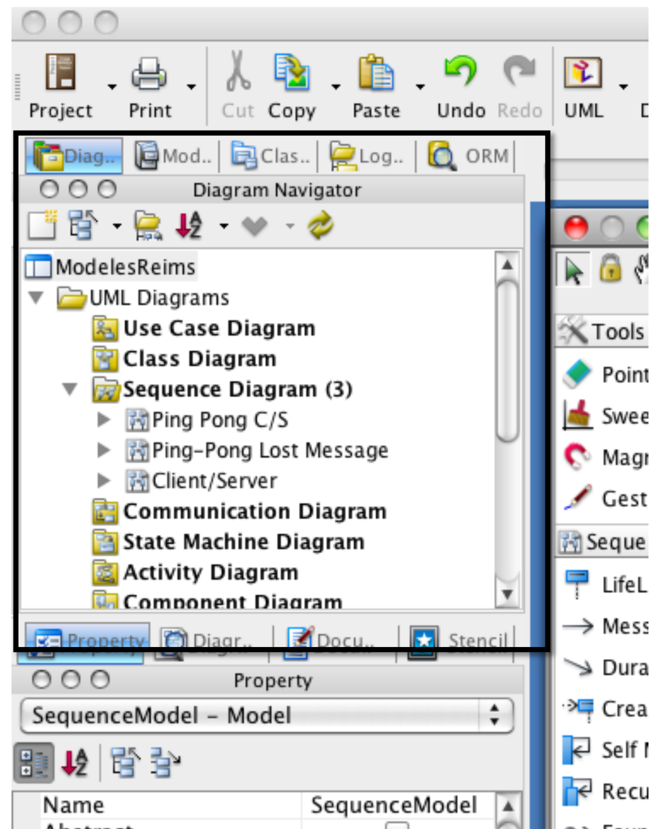
Applications

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



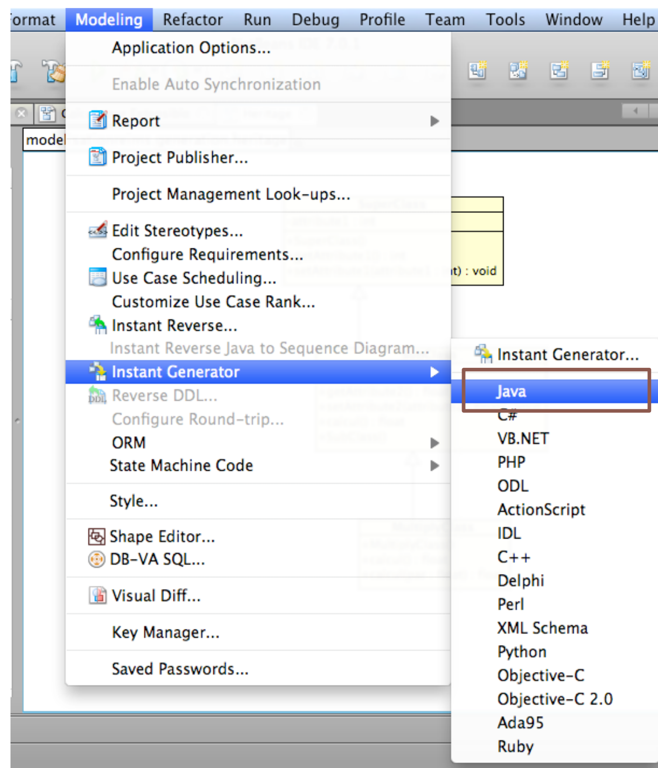
Applications

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



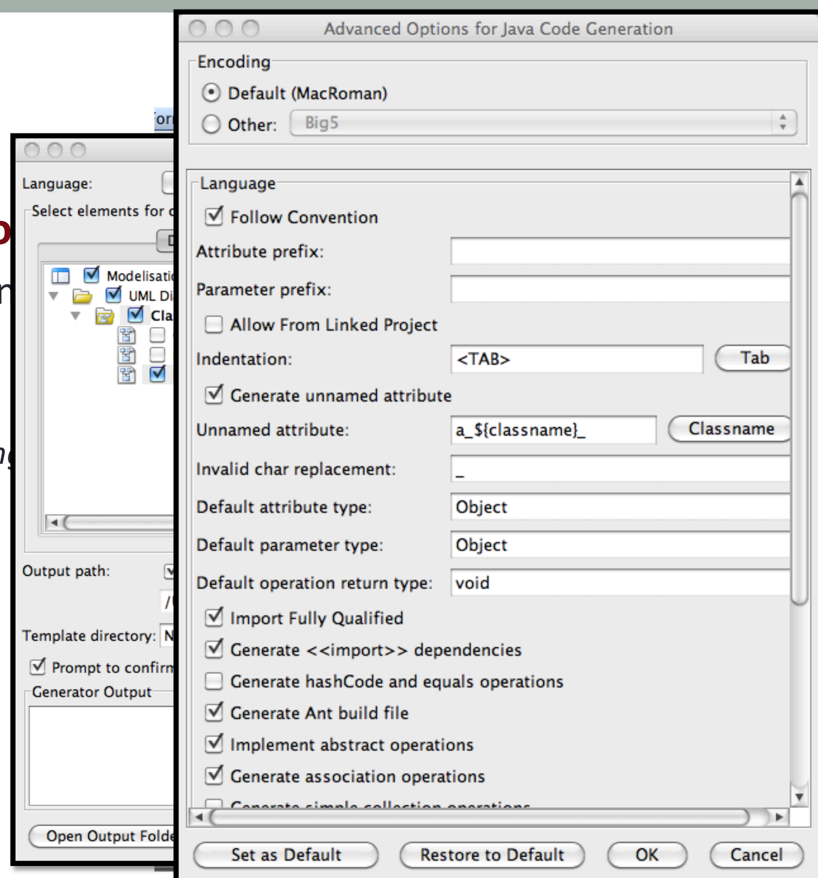
Applications

- **Génération de code**
  - Génération à un instant *t*
    - ***Instant generator***
  - Rétro conception
    - *Reverse engineering*
    - ***Instant reverse***
  - **Round-trip**
    - Java & C++



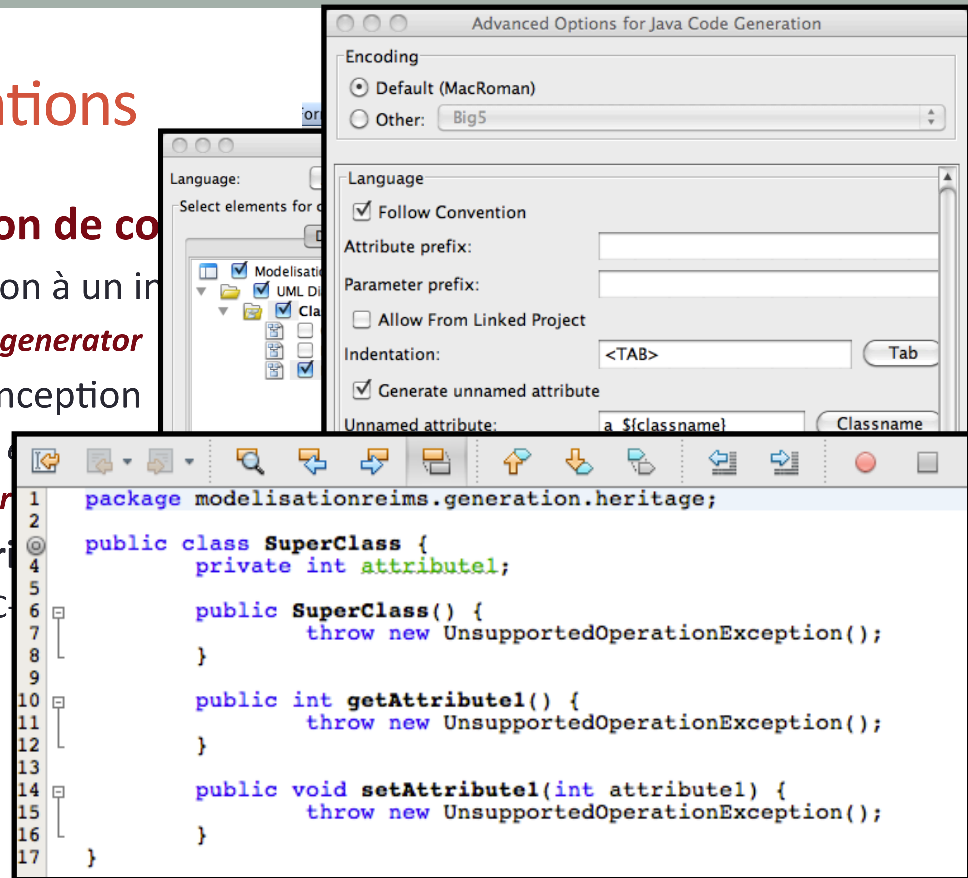
Applications

- **Génération de code**
  - Génération à un instant *t*
    - ***Instant generator***
  - Rétro conception
    - *Reverse engineering*
    - ***Instant reverse***
  - **Round-trip**
    - Java & C++



Applications

- **Génération de co**
  - Génération à un in
    - ***Instant generator***
  - Rétro conception
    - *Reverse e*
    - ***Instant r***
  - **Round-tri**
    - Java & C

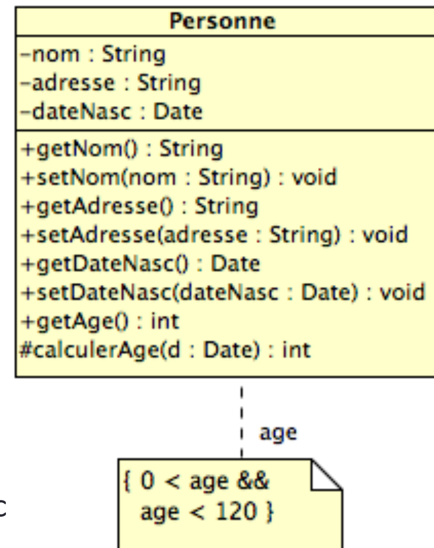


Applications génération de code

- Beaucoup d’autres applications de ce type sont disponibles
  - Yatta **UML-Lab**
    - Outil « round-trip » basé sur Eclipse
    - Uniquement diagramme de classes
  - Sparks Systems **Entreprise Architect**
    - Outil de modélisation complet
  - **Modelio**
  - **Poseidon**
  - …
- Applications souvent payantes

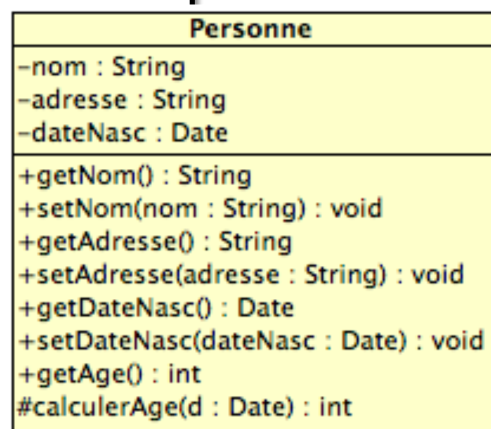
Passage UML → code Java

- **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 }

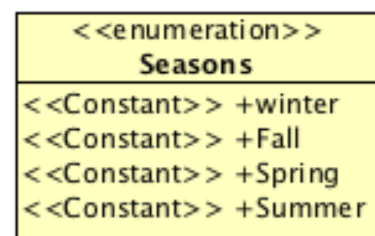
```
56  /**
57   * <p>La méthode <i>setDateNasc</i> permet de mettre à jour la date de naissanc
58   * d'un individu.</p>
59   * <p><b>Important: </b> l'âge de l'individu doit être comprise entre 0 et 120
60   * @param dateNasc
61   */
62  public void setDateNasc(Date dateNasc) {
63      //Traitement 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

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

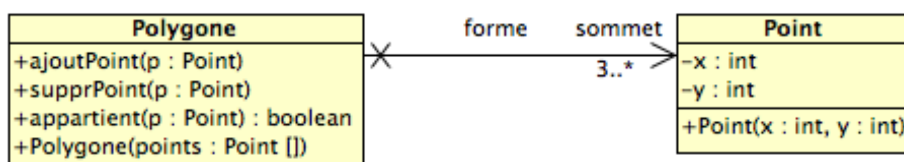
- Exemple : stéréotype « **enumeration** »

```
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

```java
import java.util.ArrayList;
import modelisationreims.generation.associations.Point;

public class Polygone {

    public ArrayList<Point> sommet = new ArrayList<Point>();

    public void ajoutPoint(Point p) {
        this.sommet.add(p);
    }

    public void supprPoint(Point p) {
        this.sommet.remove(p);
    }

    public boolean appartient(Point p) {
        return (this.sommet.contains(p));
    }

    public Polygone(Point[] points) {
        //MISSING : traitement multiplicite min 3..*
        //points.length >= 3 ou exception
        for (Point p : points) {
            sommet.add(p);
        }
    }

    @Override
    public String toString() {
        return this.sommet.toString();
    }
}
```

Polygone
+ajoutPoint(p : Point)
+supprPoint(p : Point)
+appartient(p : Point) : boolean
+Polygone(points : Point [])

forme — 3..* sommet

Point
-x : int
-y : int
+Point(x : int, y : int)

Passage UML → code Java

```java
import java.util.ArrayList;
import modelisationreims.generation.associations.Point;

public class Polygone {

    public ArrayList<Point> sommet = new ArrayList<Point>();

    public void ajoutPoint(Point p) {
        this.sommet.add(p);
    }

    public void supprPoint(Point p) {
        this.sommet.remove(p);
    }
```

```java
public class Point {

    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String toString () {
        return ("("+x+","+y+")");
    }
}
```

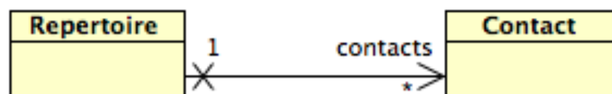
Polygone
+ajoutPoint(p : Point)
+supprPoint(p : Point)
+appartient(p : Point) : boolean
+Polygone(points : Point [])

forme — 3..* sommet

Point
-x : int
-y : int
+Point(x : int, y : int)

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;***

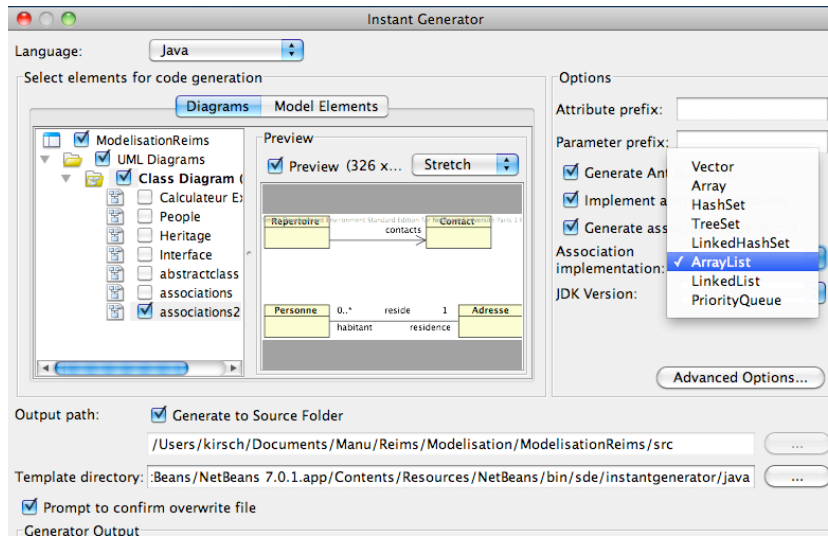


Passage UML → code Java

- **Multiplicité** : 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**)

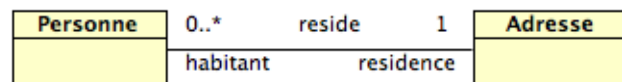
Passage UML → code Java

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



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

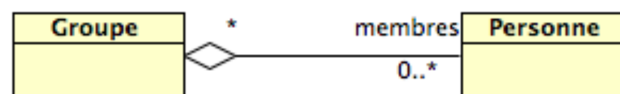
- **Associations** : **rôle** permanent ou variable ?
  - **L'extrémité d'une association peut-elle changer ?**
    - **L'objet référencé par le rôle b peut-il changer ?**

- **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

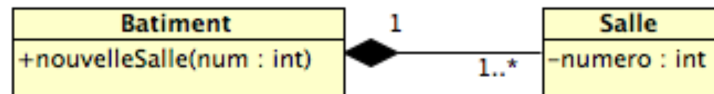
- **Associations** : **agrégation** & **composition**
  - Sémantique ***conteneur-contenu*** différente
  - Gestion des parties (***contenu***) différente

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

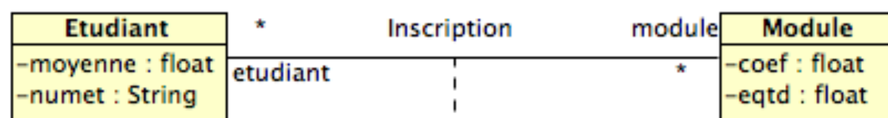
  - Le **cycle de vie** des objets **contenu** (Salle) est souvent **géré** par le **conteneur** (Bâtiment)
    - *new Salle (…)   dans la classe Batiment*

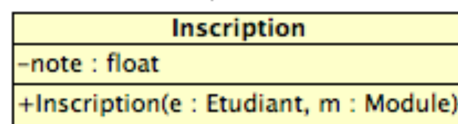
```
5   public class Batiment {
6           public HashMap<Integer,Salle> unnamed_Salle_ = new HashMap<Integer,Salle>();
7           public void nouvelleSalle(int num) {
8                   Salle s = new Salle(num);
9                   unnamed_Salle_.put(num, s);
10          }
```

  - L'objet **contenu** n'est **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

Property
Etudiant – Class

Name	Etudiant
Parent	classeassoc
View	
Visibility	public
Abstract	☐
Leaf	☐
Root	☐
Active	☐
Attributes	
moyenne	
Name	moyenne
Visibility	private
Type	float
Visible	☑
Setter	☑
Getter	☑
Stereotypes	<Unspecified>
Tagged V...	
Comments	
Project Ma...	
numet	
Name	numet
Visibility	private
Type	String
Visible	☑
Setter	☐
Getter	☐
Stereotypes	<Unspecified>
Tagged V...	

Passage UML → code Java

- **Classe-Associations**
  - Génération de code sur VisualParadigm

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

Property
Etudiant – Class

Name	Etudiant
Parent	classeassoc
View	
Visibility	public
Abstract	☐
Leaf	☐
Root	☐
Active	☐
Attributes	
moyenne	
Name	moyenne
Visibility	private
Type	float
Visible	☑
Setter	☑

```java
package modelisationreims.generation.classeassoc;

import java.util.ArrayList;
import modelisationreims.generation.classeassoc.Inscription;

public class Etudiant {
        private float moyenne;
        private String numet;
        public ArrayList<Inscription> a_Inscription_ = new ArrayList<Inscription>();

        public void setMoyenne(float moyenne) {
                this.moyenne = moyenne;
        }

        public float getMoyenne() {
                return this.moyenne;
        }
}
```

Passage UML → code Java

Property

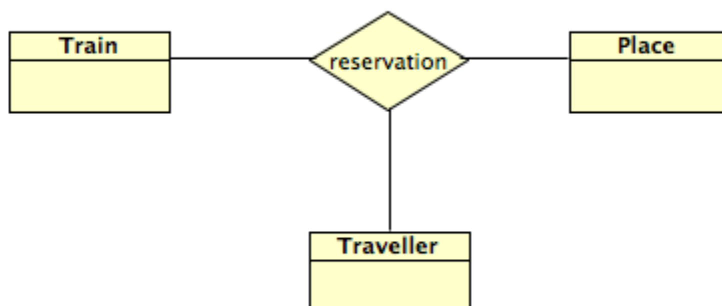
Etudiant – Class

```
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

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**



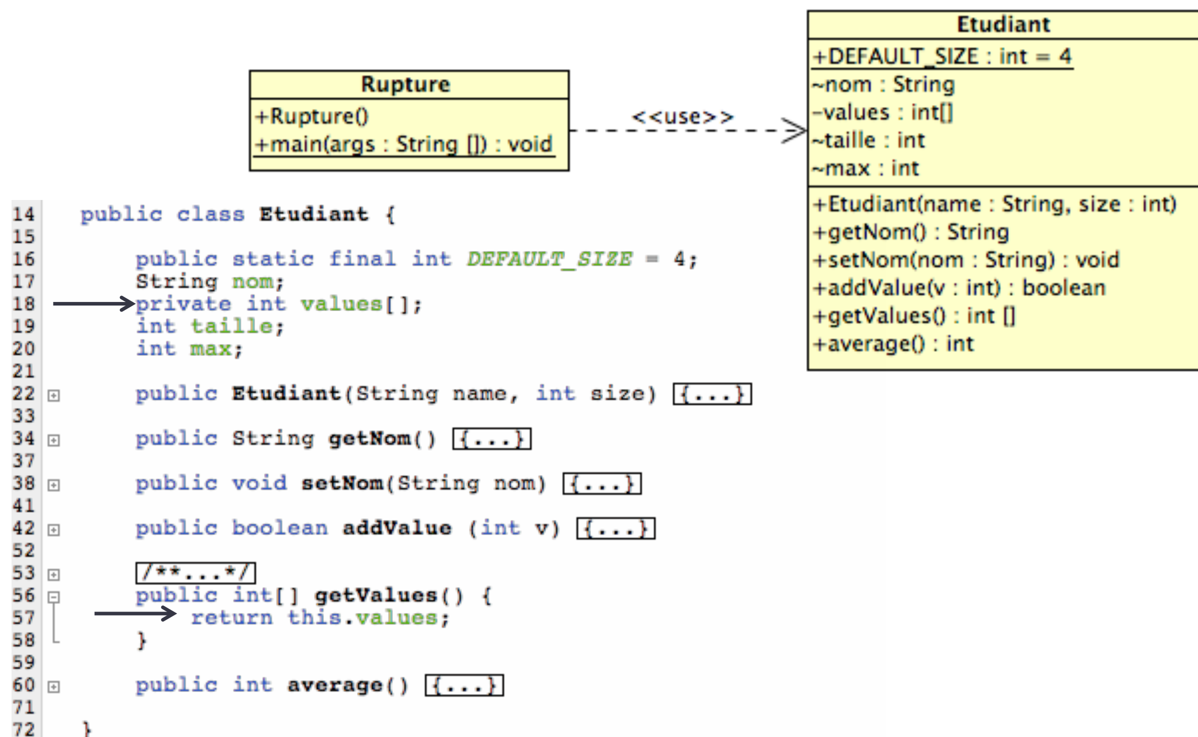

```
public class reservation {
    Train a_Train_;
    Place a_Place_;
    Traveller a_Traveller_;
    public reservation (Train t, Place p,
                        Traveller tr) {
        this.a_Place_ = p;
        this.a_Train_ = t;
        this.a_Traveller_ = tr;
    }
…
```

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

Rup…
+Rupture()
+main(args : …

```
13   public class Rupture {
14
15 ⊞     /**...*/
18 ⊟     public static void main(String[] args) {
19         Etudiant e = new Etudiant("Toto", 3) ;
20         e.addValue(18);
21         e.addValue(18);
22         e.addValue(18);
23         System.out.println("Etudiant : "+e.getNom());
24         System.out.println("moyenne : "+e.average());
25
26
27         String n = e.getNom();
28         n += " Tata ";
29         System.out.println("Etudiant : "+e.getNom());
30
31         int v[] = e.getValues();
32         v[0] = 0;
33         v[1] = 1;
34         System.out.println("moyenne : "+e.average());
35
36         e.nom = "Tata";
37         System.out.println("Etudiant: "+e.getNom());
38
39     }
40   }
```

```
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(Str
33
34 ⊞     public String getNo
37
38 ⊞     public void setNom(
41
42 ⊞     public boolean addV
52
53 ⊞     /**...*/
56 ⊟     public int[] getValues() {
57  →        return this.values;
58       }
59
60 ⊞     public int average() {...}
71
72   }
```

```
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’un 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

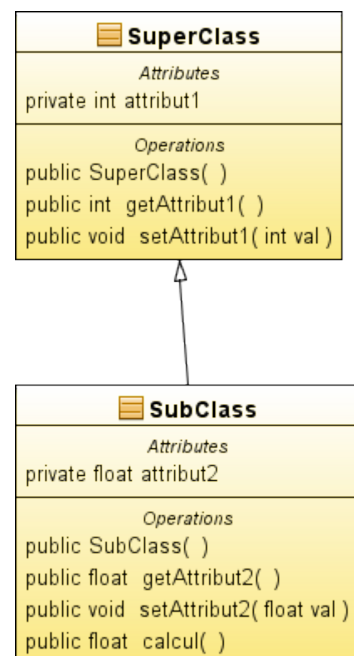
Passage UML → code Java

- **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 () {...}
9
10      public int getAttribut1 () {
11          return attribut1;
12      }
13
14      public void setAttribut1 (int val) {
15          this.attribut1 = val;
16      }
17
18  }
```

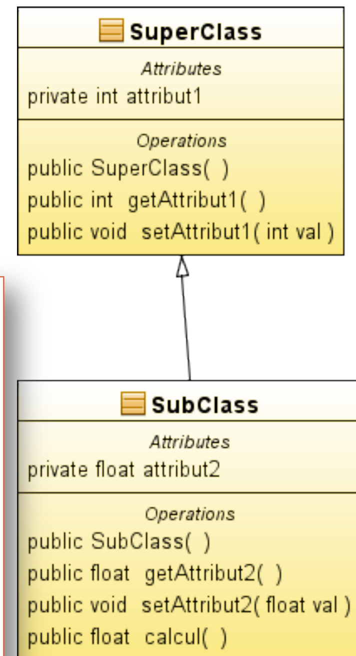


Passage UML → code Java

- **Classe : héritage**

```
3   public class SuperClass {
4
5       private int attribut1;
6
7 +     public SuperClass () {...}
```

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

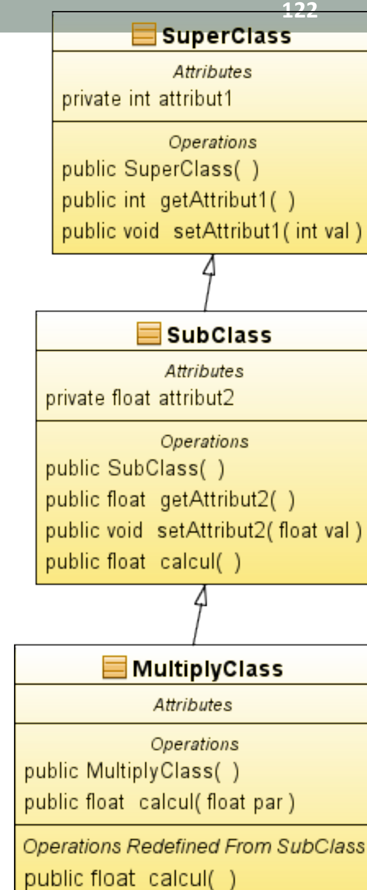


Passage UML → code Java

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

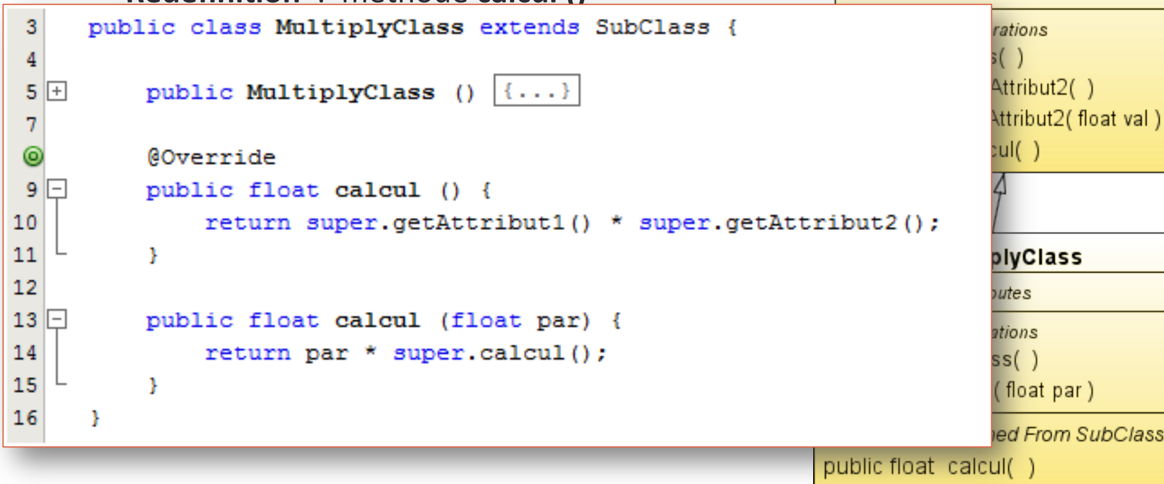
➢ **Polimorphisme**

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



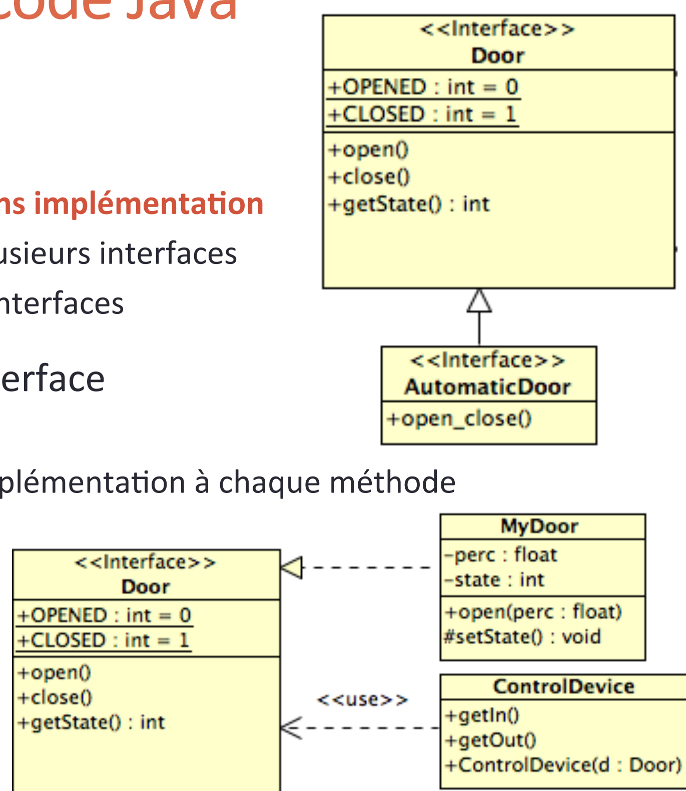
Passage UML → code Java

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



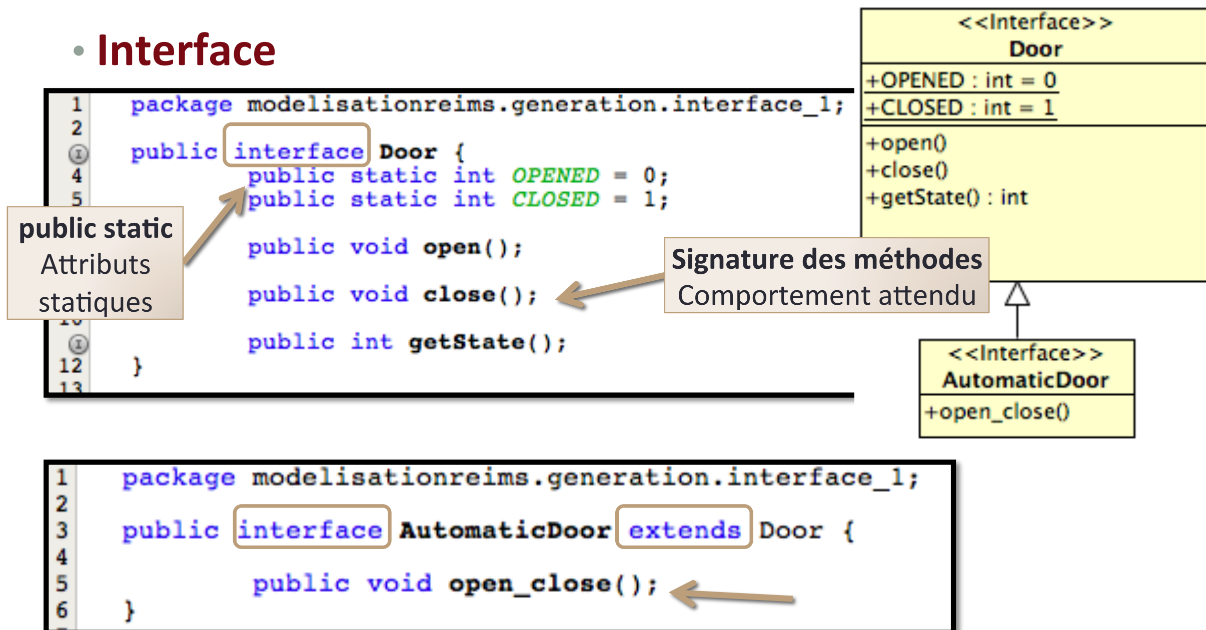
Passage UML → code Java

- **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 »**

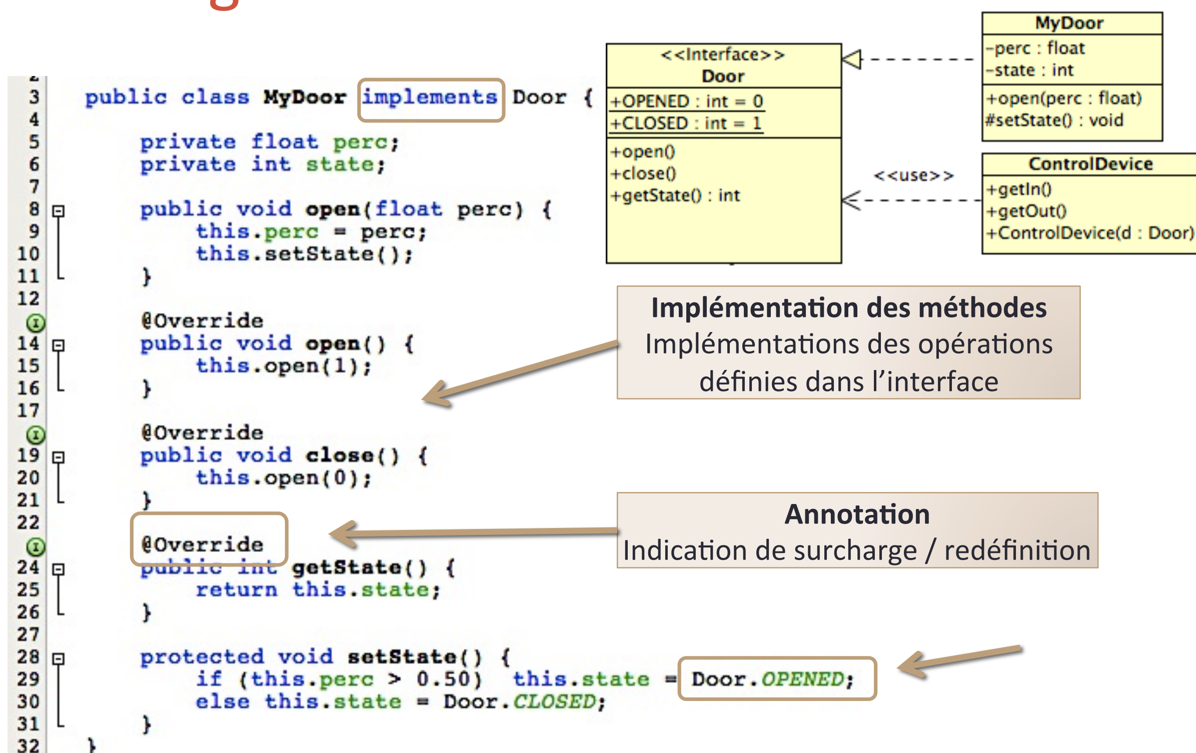


Passage UML → code Java

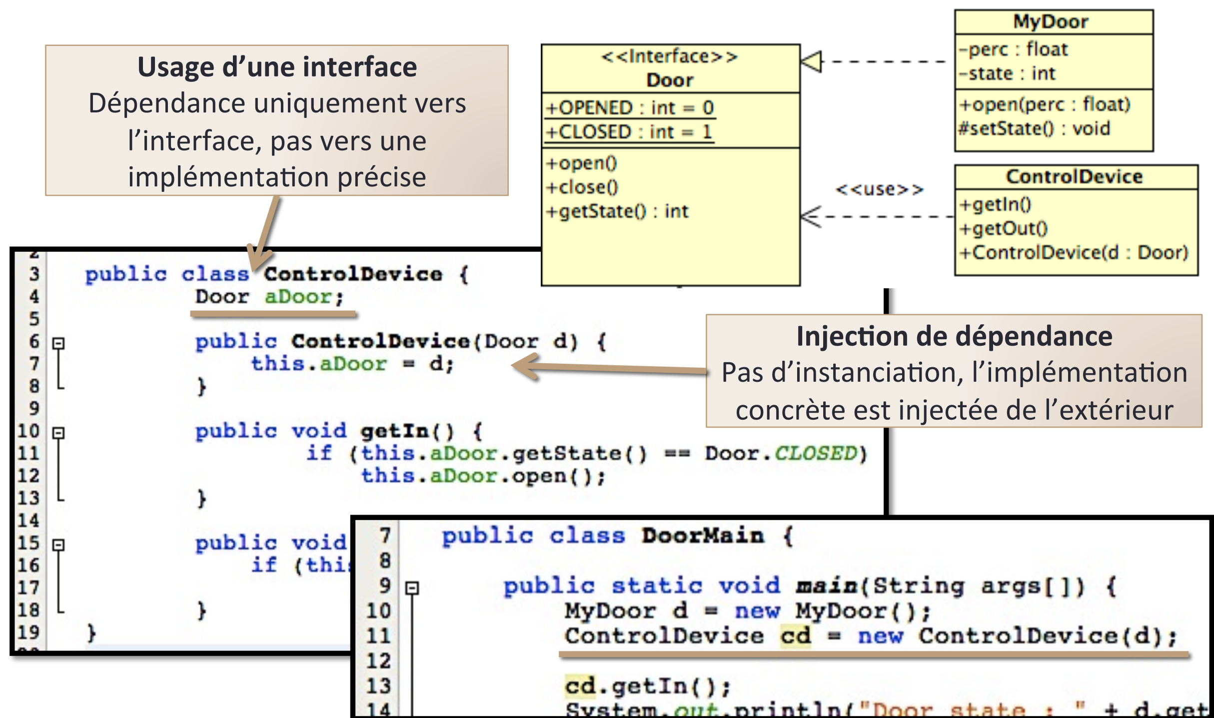
- **Interface**



Passage UML → code Java



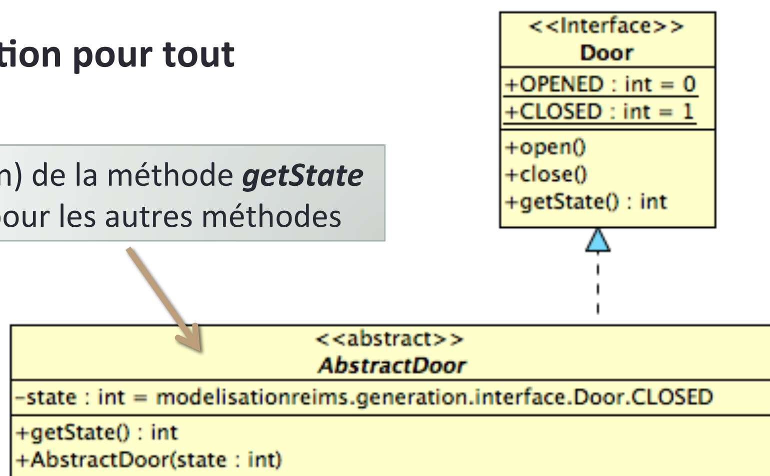
Passage UML → code Java



Passage UML → code Java

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

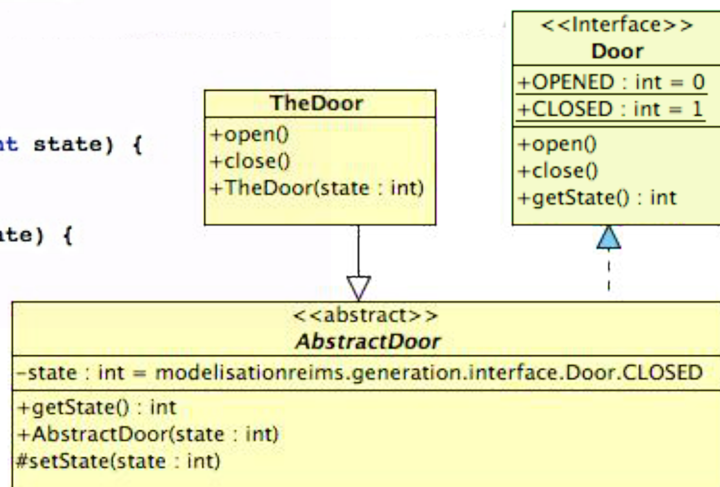
Définition (implémentation) de la méthode ***getState***
Pas d'implémentation pour les autres méthodes



Passage UML → code Java

- **Classes abstraites**

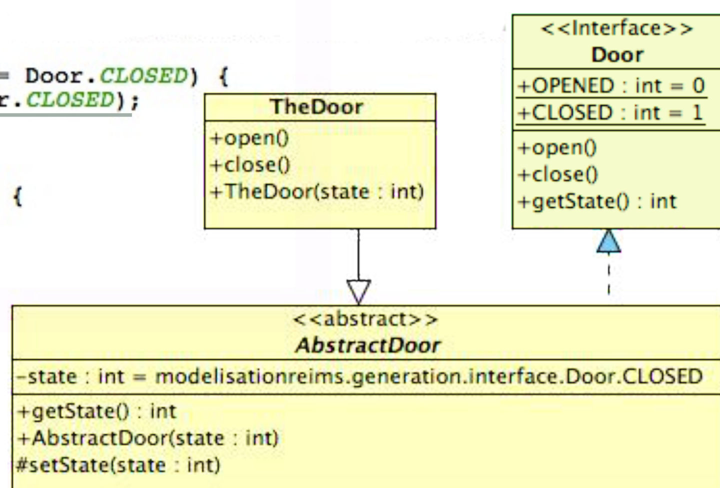
```java
package modelisationreims.generation.interface_1;

public abstract class AbstractDoor implements Door {

    private int state = Door.CLOSED;

    @Override
    public int getState() {
        return this.state;
    }

    protected void setState (int state) {
        this.state = state;
    }

    public AbstractDoor(int state) {
        this.state = state;
    }
}
```



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

Passage UML → code Java

```java
package modelisationreims.generation.interface_1;

public class TheDoor extends AbstractDoor {

    @Override
    public void open() {
        if (this.getState() != Door.OPENED) {
            this.setState(Door.OPENED);
        }
    }

    @Override
    public void close() {
        if (this.getState() != Door.CLOSED) {
            this.setState(Door.CLOSED);
        }
    }

    public TheDoor(int state) {
        super(state);
    }
}
```



Implémentation complète
Implémentation pour ***open*** et ***close*** à partir de la classe abstraite