

MODÉLISATION DES APPLICATIONS

Manuele Kirsch Pinheiro

Maître de Conférences – Université Paris 1

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

<http://mkirschp.free.fr>

2

Objectifs et Planning

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

Références

- Bibliographie

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

- Sites Web

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

Pourquoi Modéliser ?

- Pourquoi modéliser ?

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

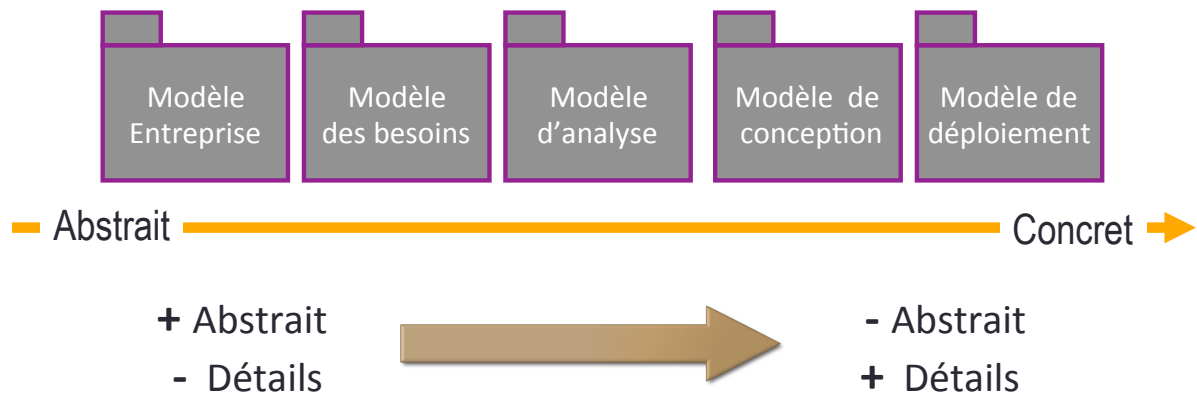
- Un modèle est ...

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

Réduire la complexité pour mieux comprendre

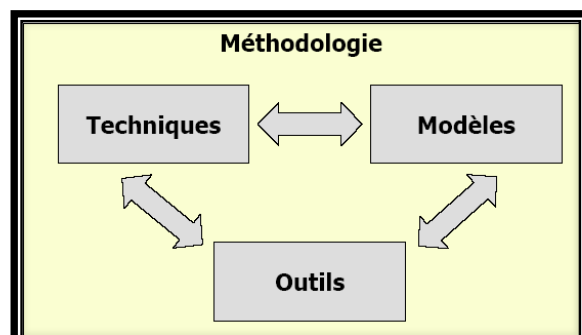
Modélisation

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



Méthodologies de développement

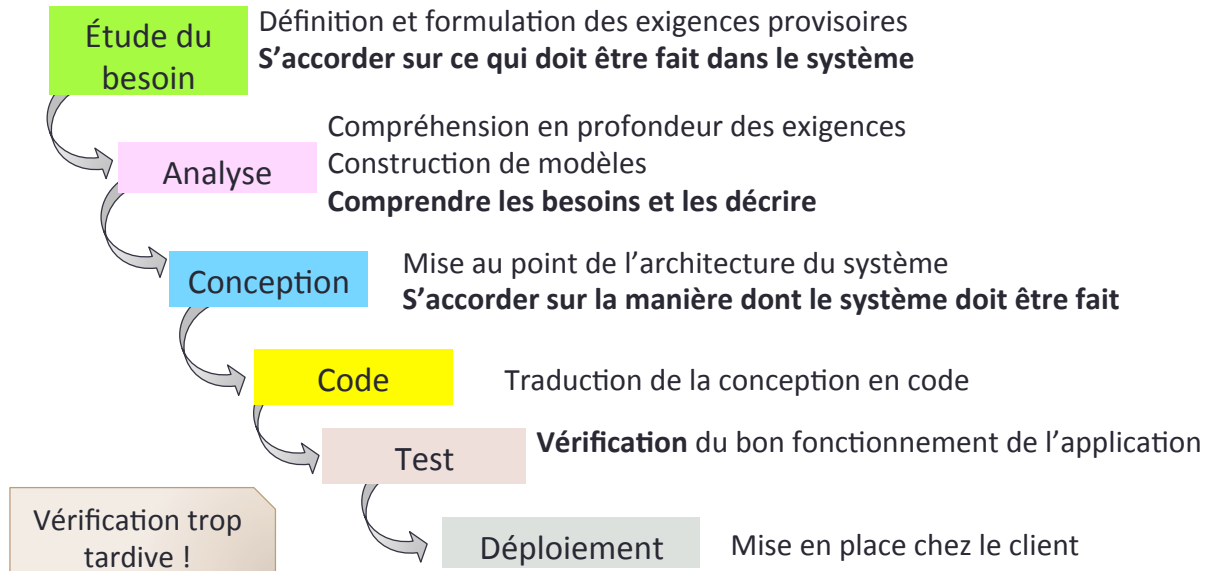
- **Méthodologie**
 - Démarche reproductible pour obtenir un résultat
- But : **Organiser la démarche de travail**
 - Procédés
 - Artefacts
 - Intervenants



Méthodologies de développement

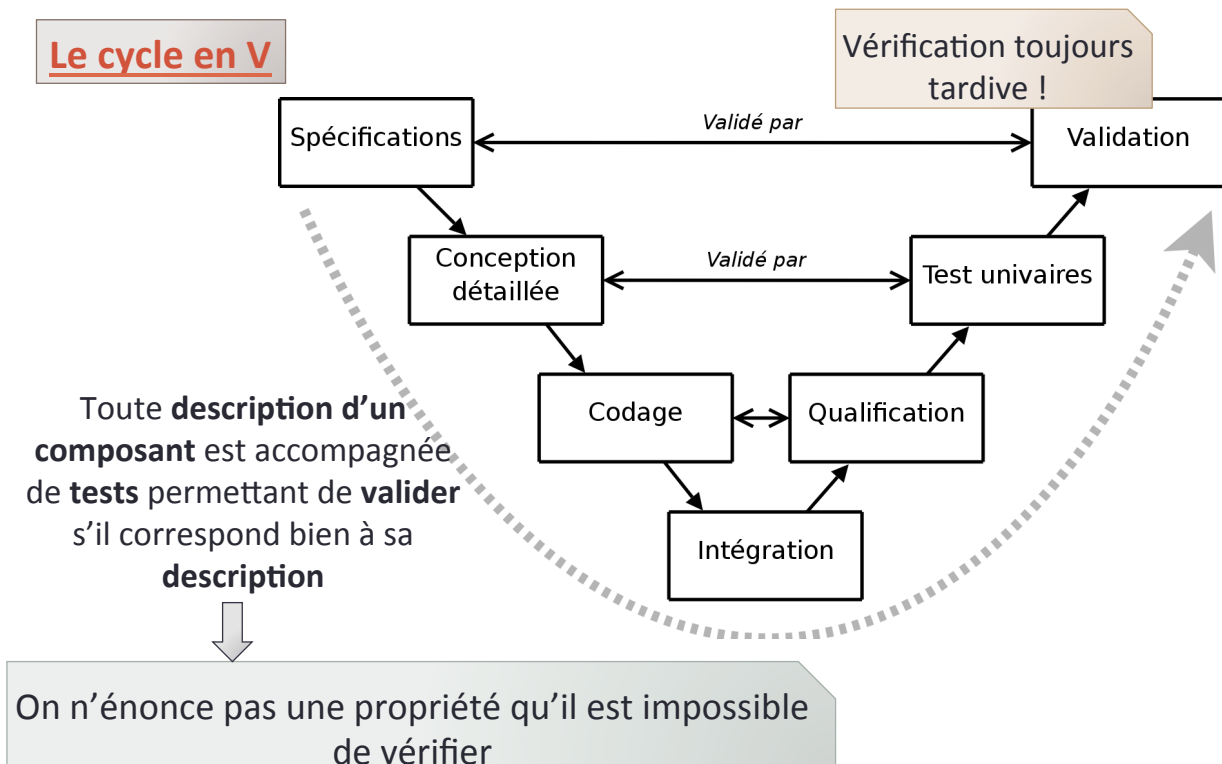
Cycle de vie en cascade

Étapes d'un projet informatique



Méthodologies de développement

Le cycle en V



Méthodologies de développement

- **Méthodes agiles**

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

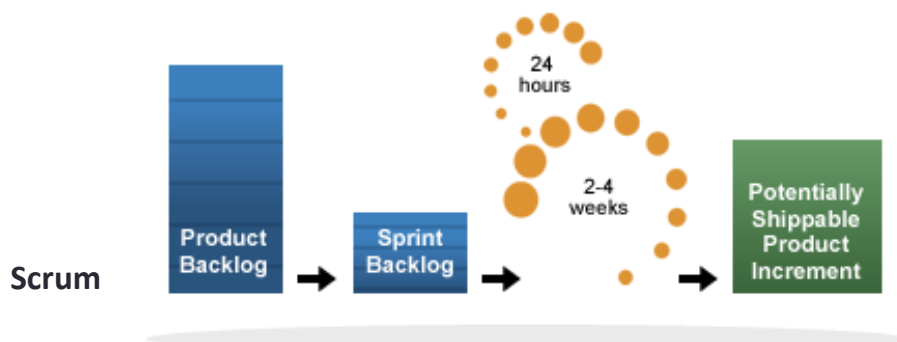
- **Manifeste Agile**

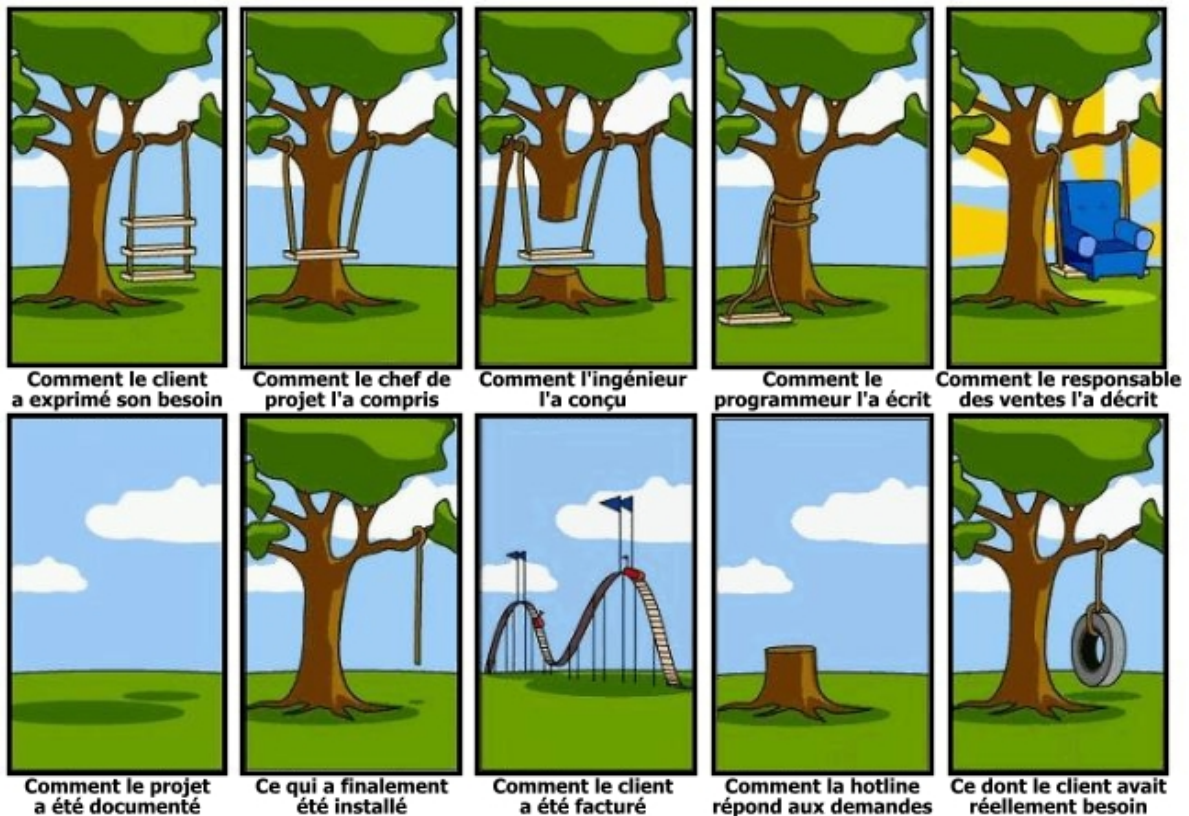
- **Individus** et **interactions** plus que processus et outils
- **Logiciels fonctionnel** plus que documentation abondante
- **Collaboration** avec le client plus que négociation contractuelle
- **Adaptation** au changement plus que suivi d'un plan

Méthodes agiles

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

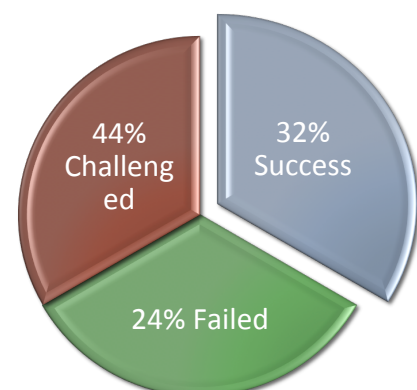
- **Intégration du client** au processus de développement
- **Livraison rapide et régulière** de fonctionnalités à forte valeur ajoutée
- **Acceptation du changement** dans les besoins
- **Travail d'équipe**
- **Simplicité et qualité**
- Attention continue à **l'excellence technique** et à une **bonne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





Réussite d'un Projet

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



UML : quoi ?!



UNIFIED MODELING LANGUAGE™

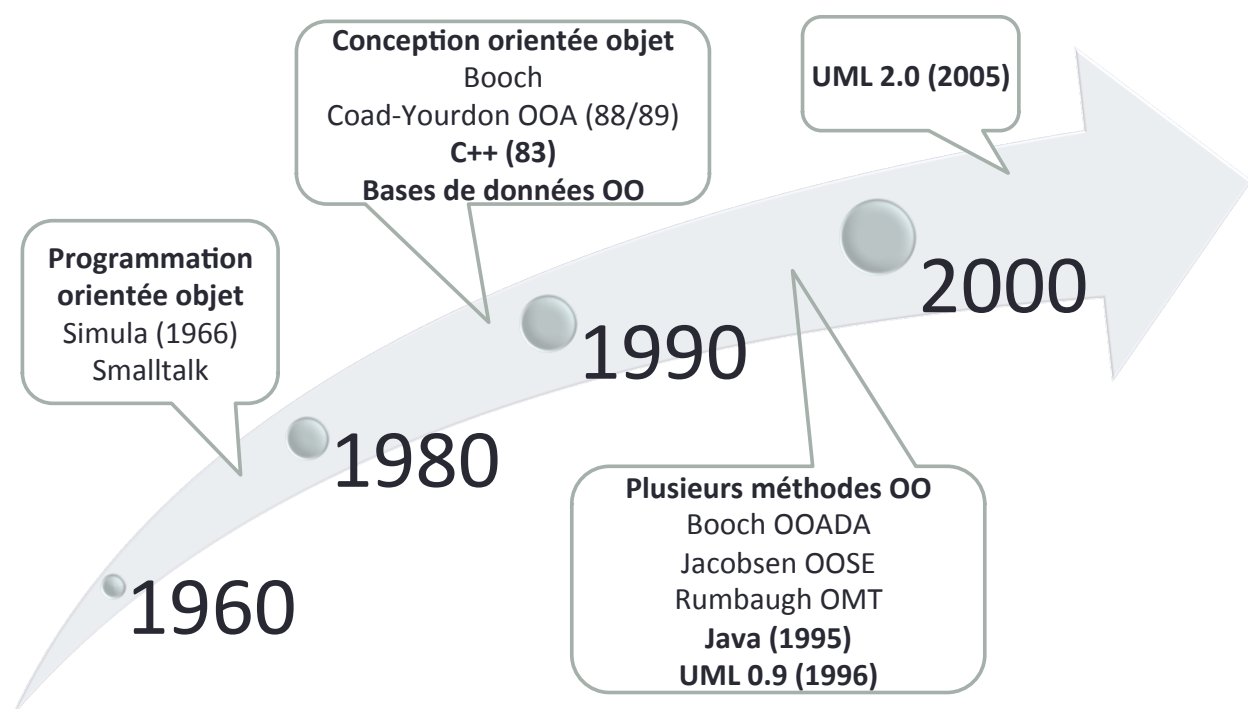


- UML, c'est quoi ?
 - UML est une **notation**, un **formalisme** permettant la modélisation
 - *UML n'est pas une méthode !*
- UML est un langage de **modélisation objets** conçu pour
 - *visualiser*
 - *spécifier*
 - *construire*
 - *documenter*

les artefacts d'un système à forte composante logicielle

- **Standard OMG** depuis 1997

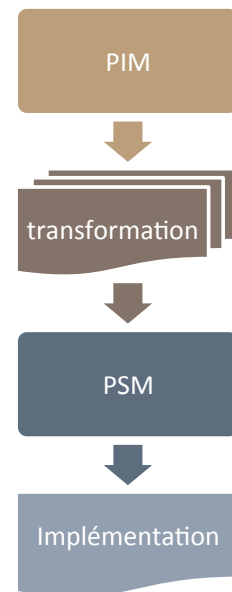
Historique



UML & MDA

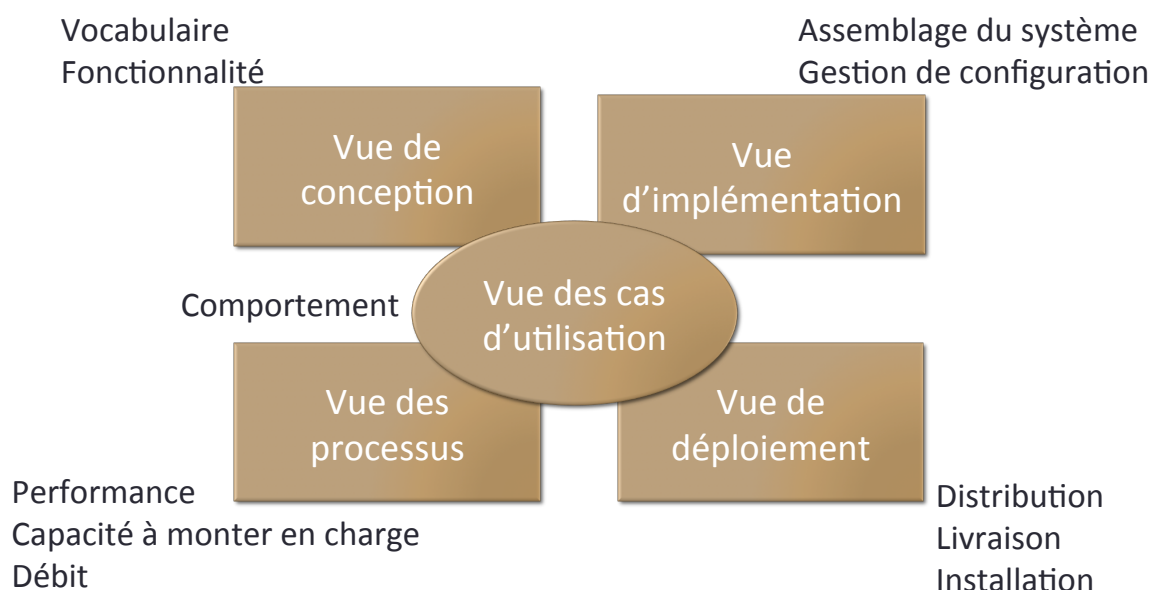
- **MDA (Model Driven Architecture)**

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



UML : 5 vues

Différents diagrammes pour différentes vues



Les diagrammes d'UML

Diagrammes structurels vues statiques

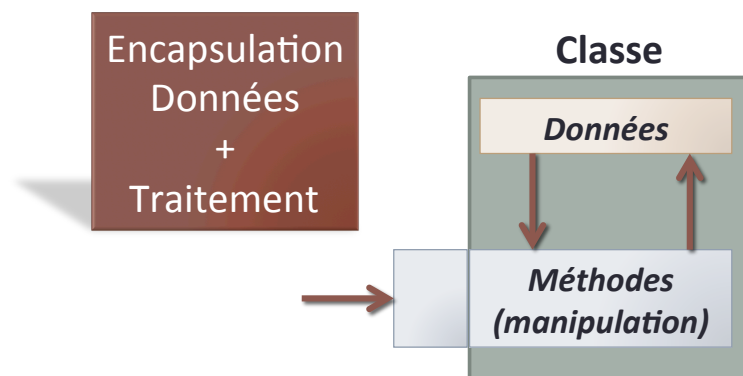
- *Diagrammes de classes*
- *Diagrammes d'objets*
- Diagrammes de composants
- Diagrammes de déploiement
- *Diagrammes de paquetage*

Diagrammes comportementaux vues dynamiques

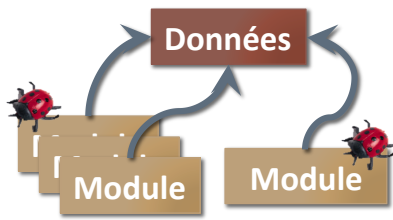
- Diagrammes de cas d'utilisation
- *Diagrammes de séquence*
- Diagrammes d'activités
- Diagrammes de communication (collaboration)
- Diagrammes d'états
- Timing diagram
- Interaction overview diagram

Orientation à Objets

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

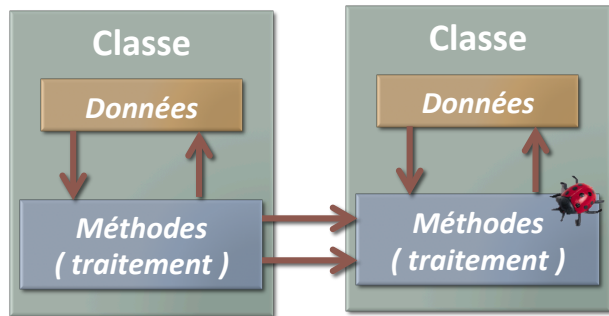


Pourquoi l'orientation à objets ?



Programmation Modulaire

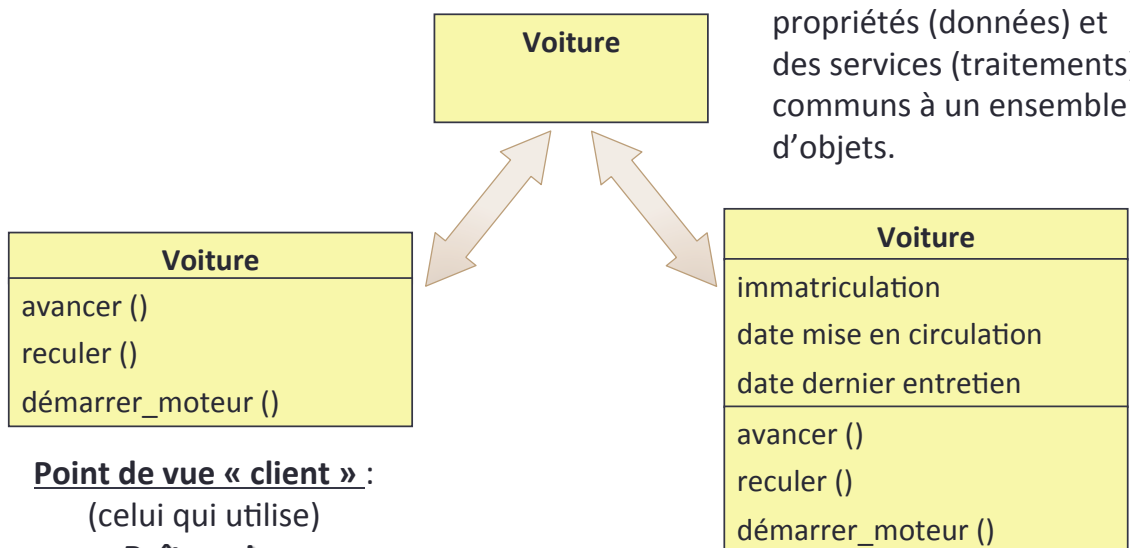
- Plusieurs modules manipulent les données
- Difficile à déboguer / maintenir
- Difficile à faire évoluer



Programmation Orientée à Objets

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

Réutilisation



Point de vue « client » :

(celui qui utilise)

Boîte noire

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

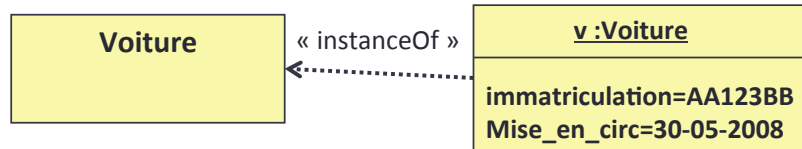
Point de vue « concepteur » :

(celui qui conçoit / implémente)

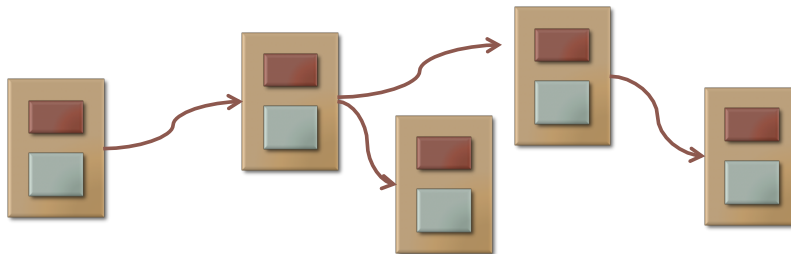
Boîte blanche

Prise en charge de l'implémentation

Réutilisation

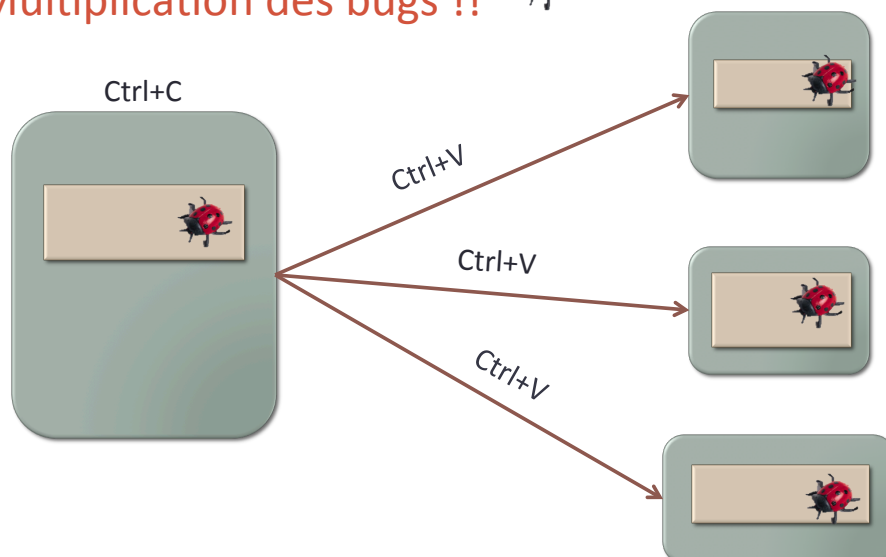


- Dans la POO, les modules logiciels réutilisables sont les classes
 - *On réutilise les classes*
 - Une **classe** détermine les **propriétés** qui peut avoir un objet et son **comportement**
 - Un **objet** est l'**instance** d'une classe. Il a un **état** (valeurs attribuées aux propriétés à un moment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



Concevoir un code de qualité

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



Concevoir un code de qualité

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

Anti-exemples

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

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

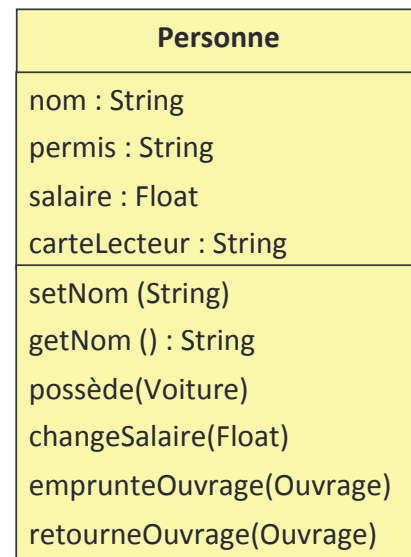
    public void setNom (String s) { ... }
    public String getNom () { ... }
    public void possede (Voiture v) { ... }
    public void changeSalaire (Float s) { ... }
    public void emprunteOuvrage (Ouvrage o) { ... }
    public void retourneOuvrage (Ouvrage o) { ... }
}
```

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

Anti-exemples

• Modularité

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



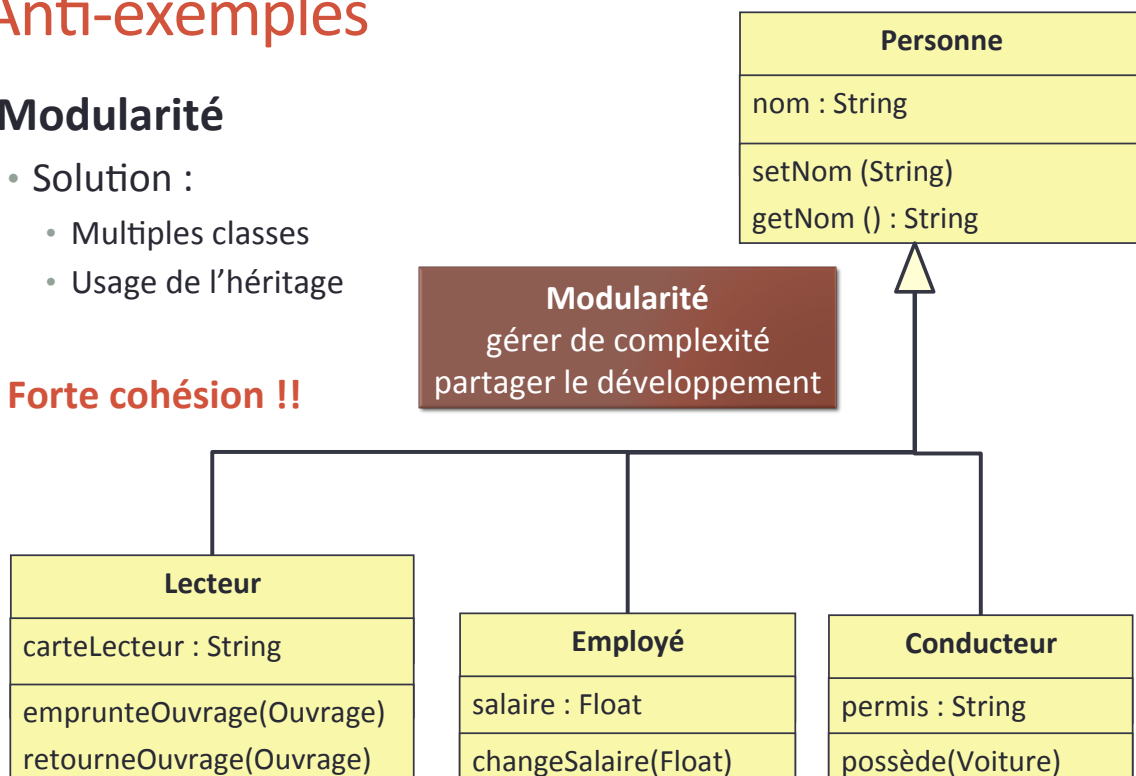
Faible cohésion !!

Anti-exemples

• Modularité

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

Forte cohésion !!



Anti-exemples

• Réutilisabilité

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

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

```

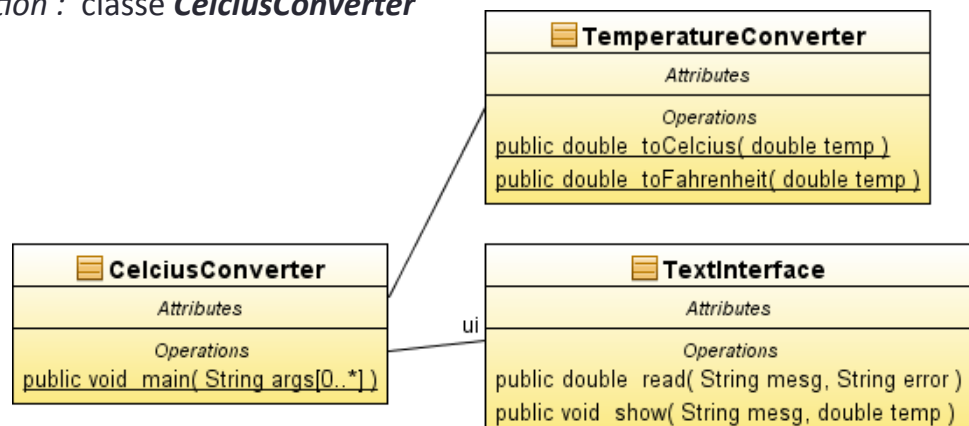
7  +  /**...*/
11  public class Convertisseur {
12  -    public static void main(String[] args) {
13      double temp = Double.parseDouble(args[0]);
14      double celc = ((temp-32)*5)/9;
15      System.out.println("Temperature en C° :"+celc);
16  }
17  }

```

Anti-exemples

• Réutilisabilité

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



Anti-exemples

- **Extensibilité / évolutivité**

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

Anti-exemples

- **Extensibilité / évolutivité**

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

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

```

11 public class Calculateur {
12
13     int a, b; //operands
14     int op; //opérateur
15
16     public Calculateur () {...}
17
18     public void init() {...}
19
20     public int calc() {
21         int r = 0;
22
23         switch (this.op) {
24             case '+':
25                 r = a + b;
26                 break;
27             case '-':
28                 r = a - b;
29                 break;
30             default:
31                 System.out.println("Opérateur inconnu");
32         }
33         return r;
34     }
35
36     public static void main (String args[]) {...}
37 }

```

Anti-exemples

- **Extens**

- Déve

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

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

deux nouvelles opérations * et /

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

```

31
32 public int calc() {
33     int r = 0;
34
35     switch (this.op) {
36         case '+':
37             r = a + b;
38             break;
39         case '-':
40             r = a - b;
41             break;
42         default:
43             System.out.println("Opérateur inconnu");
44     }
45     return r;
46 }
47
48 public static void main (String args[]) { ... }
49
50
51
52
53

```

Anti-exemples

- **Extens**

- Déve

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

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

deux nouvelles opérations * et /

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

```

31
32 public int calc() {
33     int r = 0;
34
35     switch (this.op) {
36         case '+':
37             r = a + b;
38             break;
39         case '-':
40             r = a - b;
41             break;
42         default:
43             System.out.println("Opérateur inconnu");
44     }
45     return r;
46 }
47
48 public static void main (String args[]) { ... }
49
50
51
52
53

```

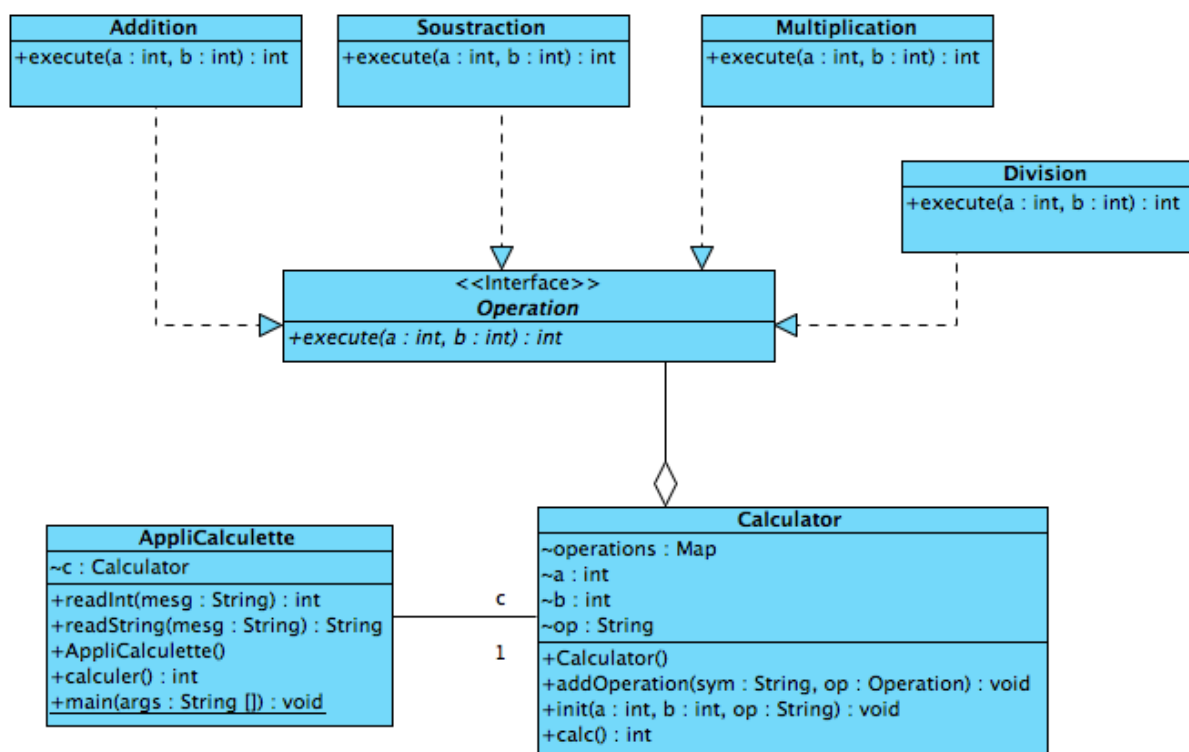
Réutilisation difficile !
Classe non-extensible

Anti-exemples

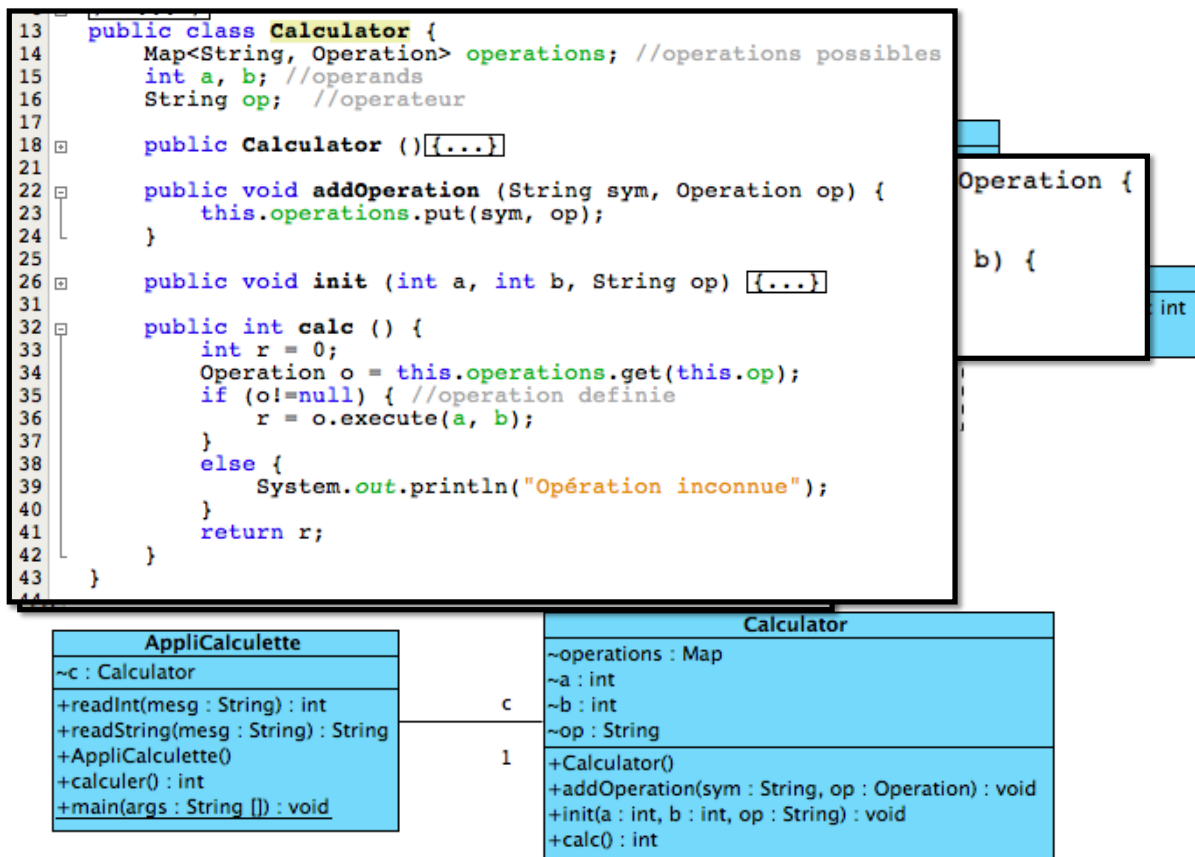
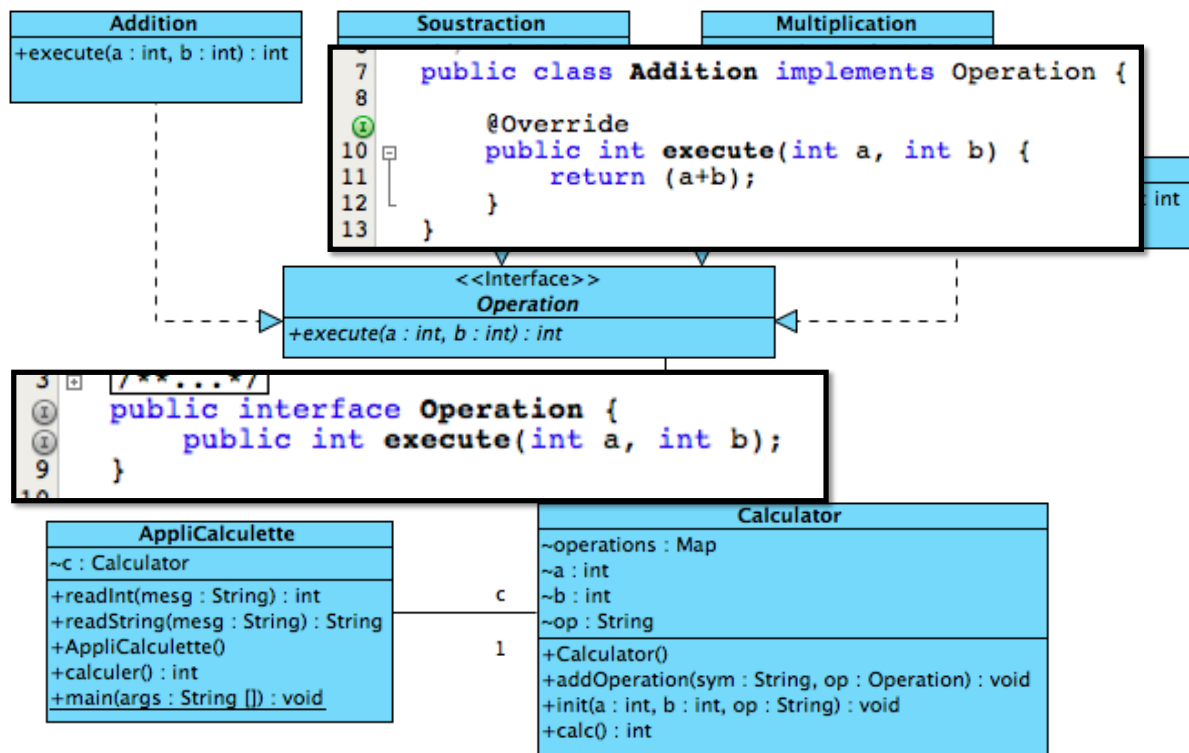
- **Extensibilité / évolutivité**

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

Anti-exemples



Anti-exemples



```

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

```

```

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

```

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

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

Références

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

MODÉLISATION DES APPLICATIONS

Manuele Kirsch Pinheiro

Maître de Conférences – Université Paris 1

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

<http://mkirschp.free.fr>

40

Objectifs et Planning

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

Les diagrammes d'UML

Diagrammes structurels vues statiques

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

Diagrammes comportementaux vues dynamiques

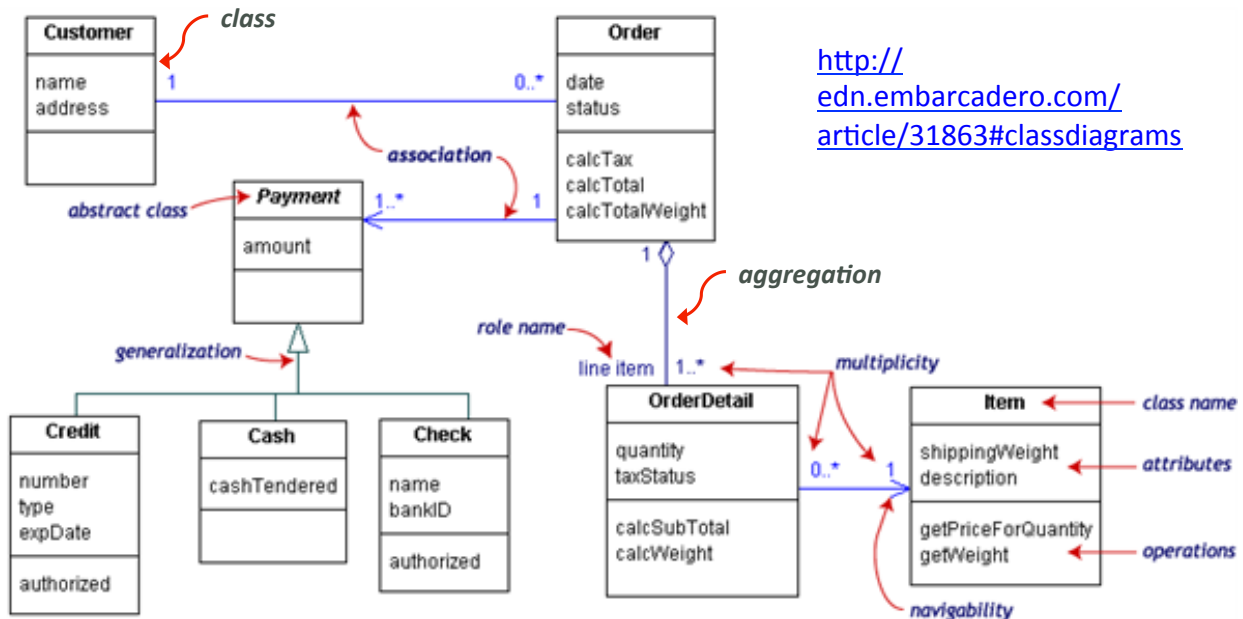
- Diagrammes de cas d'utilisation
- **Diagrammes de séquence**
- Diagrammes d'activités
- Diagrammes de communication (collaboration)
- Diagrammes d'états
- Timing diagram
- Interaction overview diagram

Diagramme de classes

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

Diagramme de classes

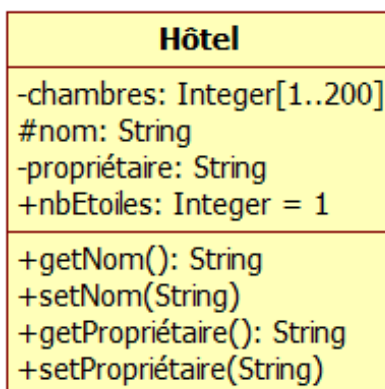
R é s u m é



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

Diagramme de classes : classe

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



Nom : il doit être significatif et commencer par majuscule

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

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

Diagramme de classes : classe

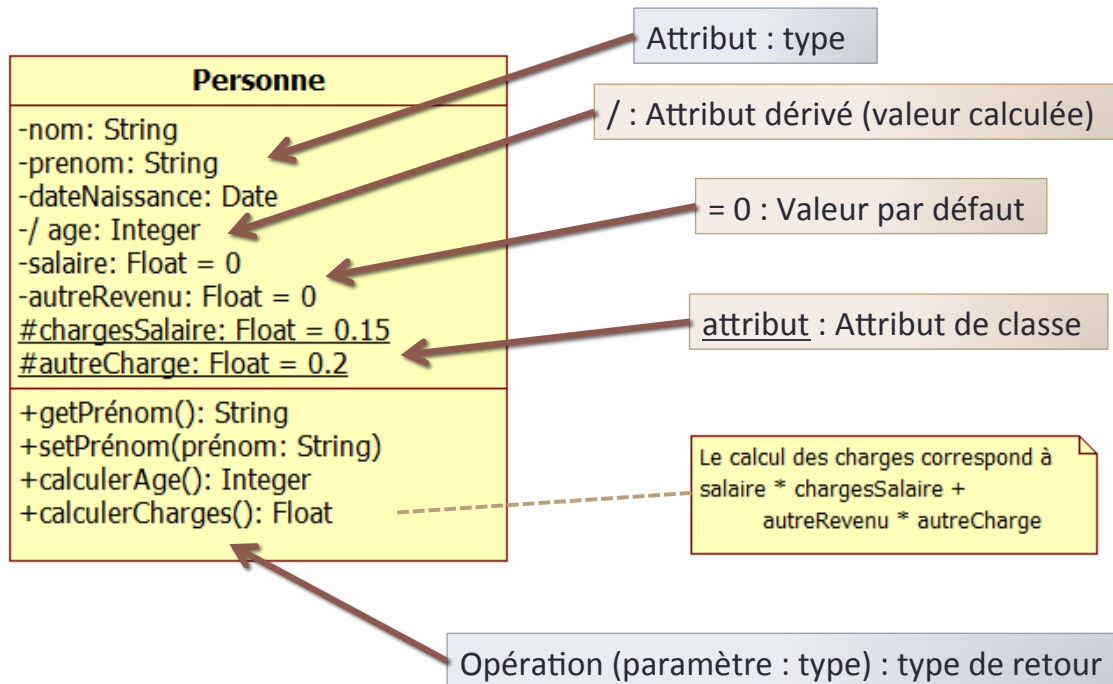


Diagramme de classes : classe

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



Application du principe de l'encapsulation !!

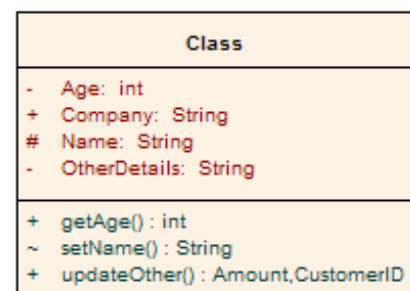


Diagramme de classes : classe

- **Opérations**

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

- **Envoi de message**

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



Diagramme de classes : classe

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

Car
-puissance: Float -couleur: String -nbPortes: Integer
+Car(puiss: Float, coul: String, portes: Integer) +setCouleur(coul: String) +getCouleur(): String +getPuissance(): Float +getNbPortes(): Integer

Diagramme de classes : héritage

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

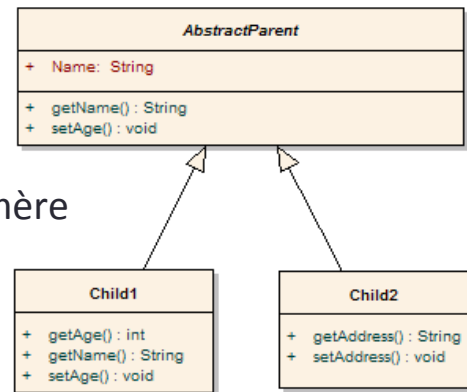
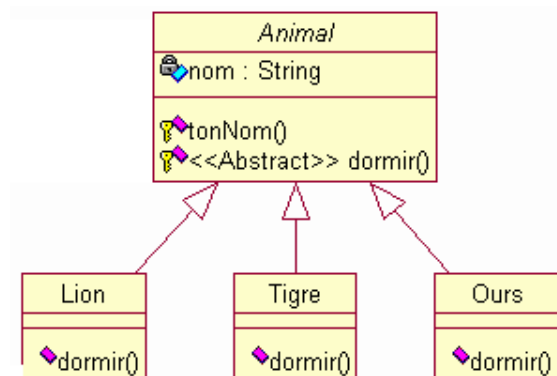


Diagramme de classes : héritage

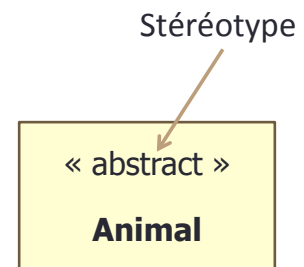
- **Classes abstraites**
 - Les classes abstraites représentent une abstraction afin de factoriser des **propriétés/opérations** communes
 - Spécifications générales à un ensemble de sous-classes
 - Généralisation d'un concept abstrait commun à plusieurs classes concrètes
 - Classes non instantiables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



UML : Stéréotypes

• Stéréotype

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



• Stéréotypes prédéfinis en UML

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

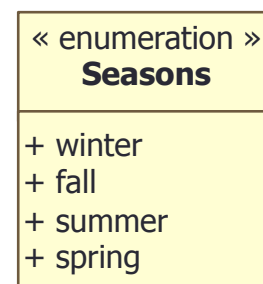


Diagramme de classes : associations

• Associations

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

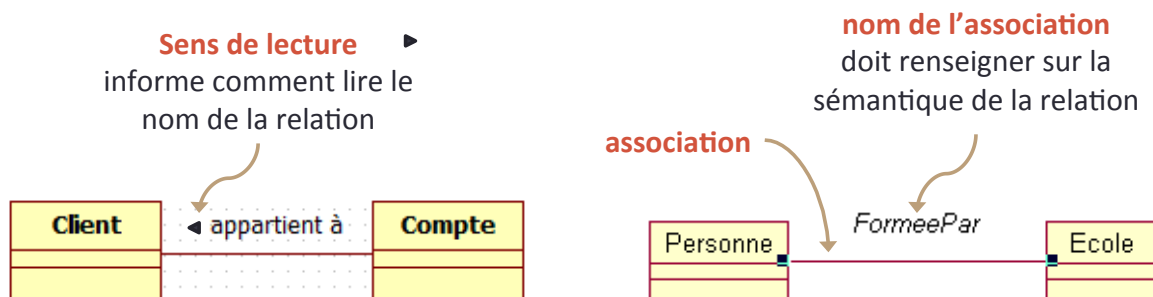


Diagramme de classes : associations

• Rôle

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



• Multiplicité

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

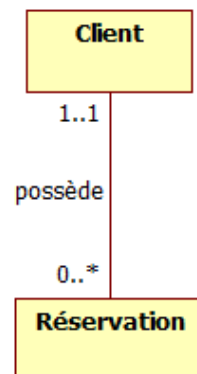


Diagramme de classes : associations

• Multiplicité

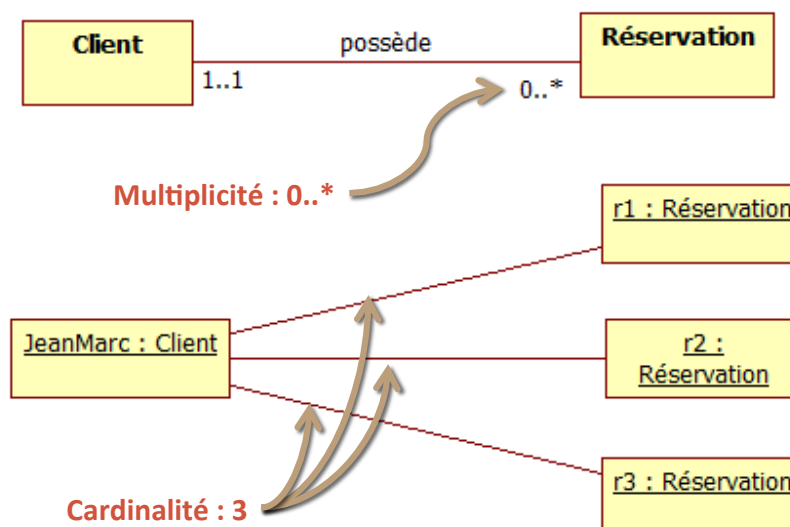


Diagramme de classes : associations

• Navigabilité

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

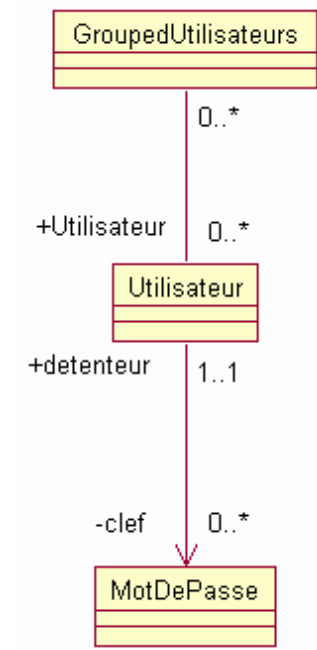
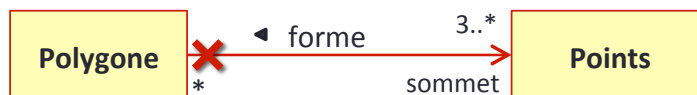


Diagramme de classes : associations

• Associations binaires et n-aires

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



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

- Plus rares , peu utilisées
- Plus difficiles à comprendre

• Multiplicité ?!

- Pour $\langle \text{Traveler } X, \text{Seat } S \rangle$, combien d'objets **Train** ?
- Pour $\langle \text{Train } T, \text{Seat } S \rangle$, combien d'objets **Traveller** ?
- Pour $\langle \text{Traveler } X, \text{Train } T \rangle$ combien d'objets **Seat** ?

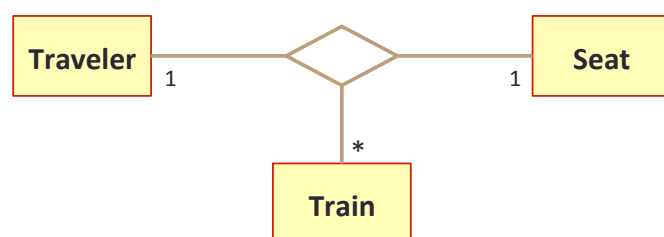
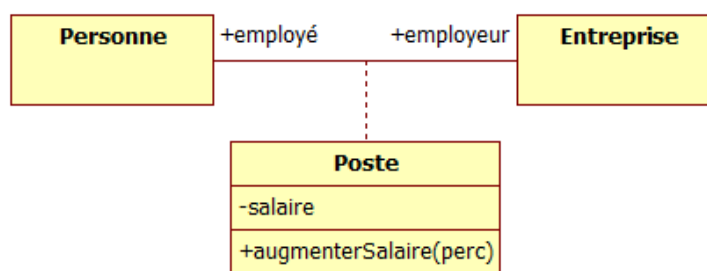


Diagramme de classes : classe-association

• Classe-association

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



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

Diagramme de classes : agrégations

• Agrégation

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

• Composition

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

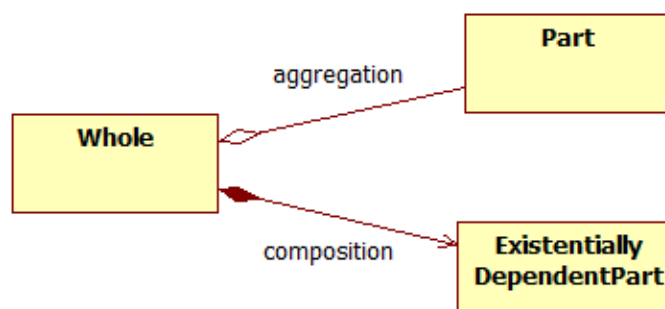


Diagramme de classes : agrégations

- **Pattern Composite**

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

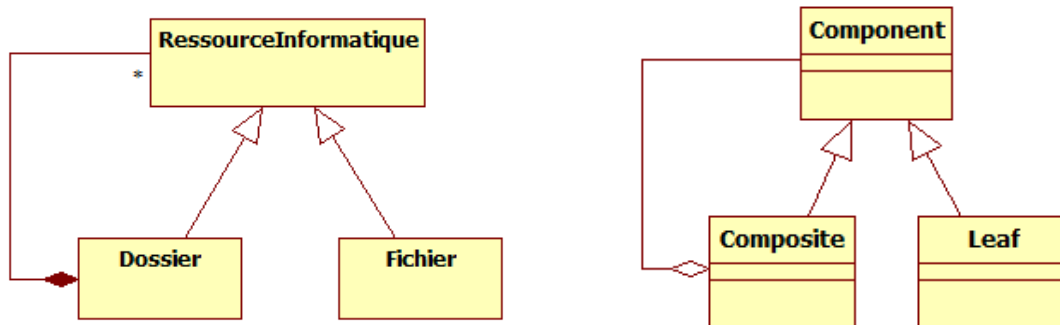


Diagramme de classes : dépendances

- **Dépendances**

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

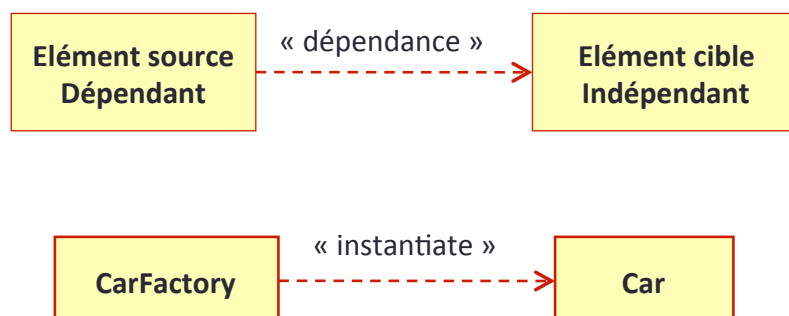


Diagramme de classes : contraintes

• Contraintes

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

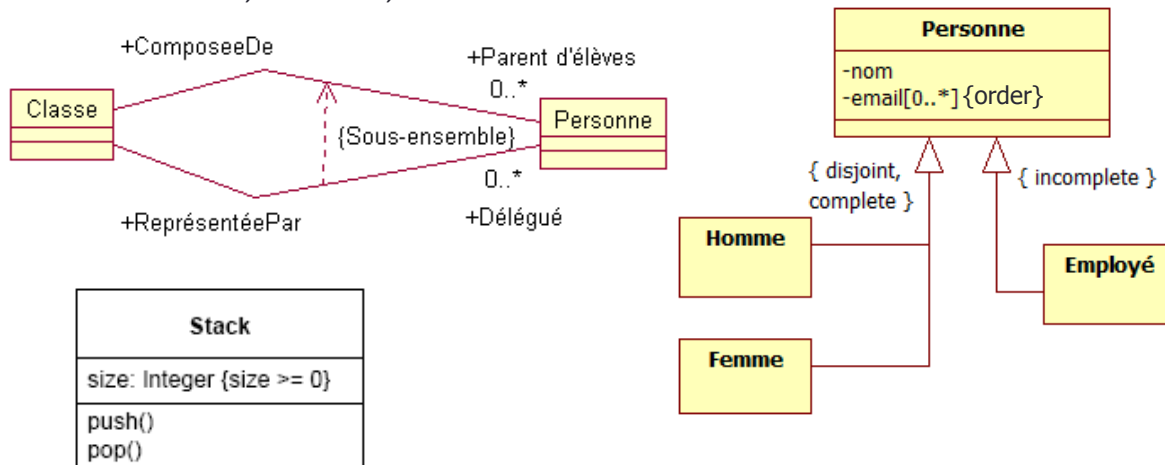


Diagramme de classes : contraintes

• Contraintes

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

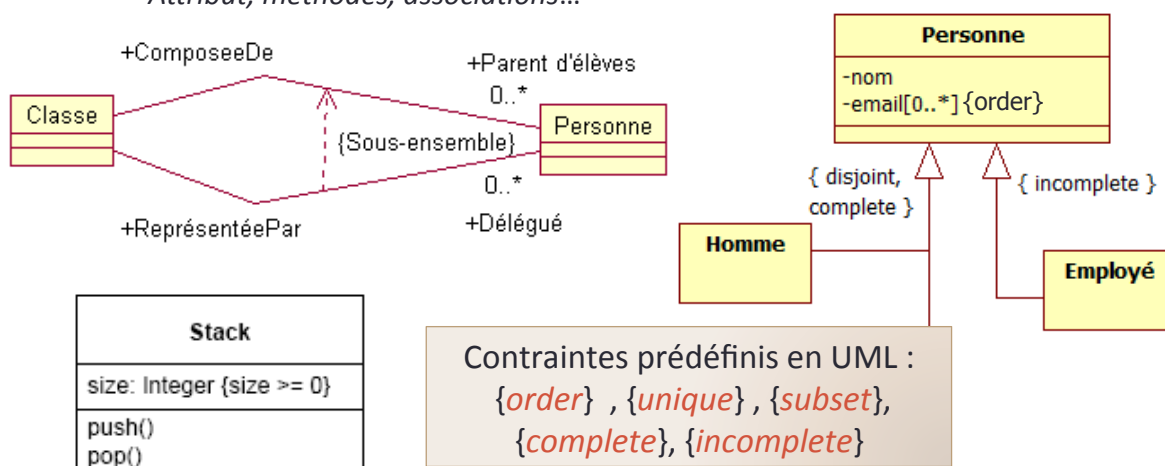


Diagramme de classes : contraintes

• Contraintes

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

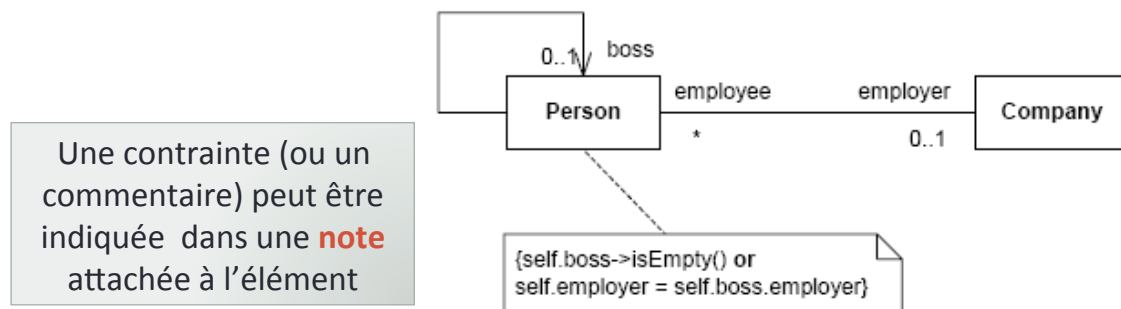


Diagramme de classes : interfaces

• Interface

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

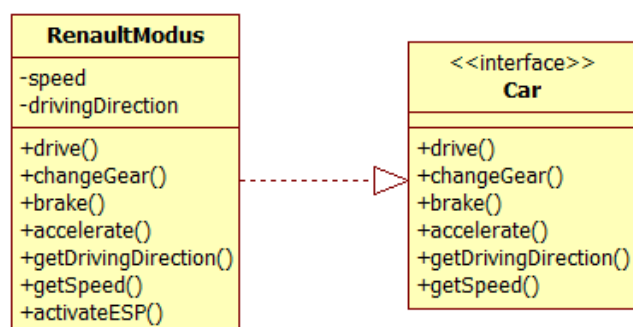
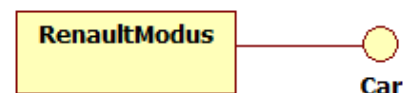


Diagramme de classes : interfaces

• Interface

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

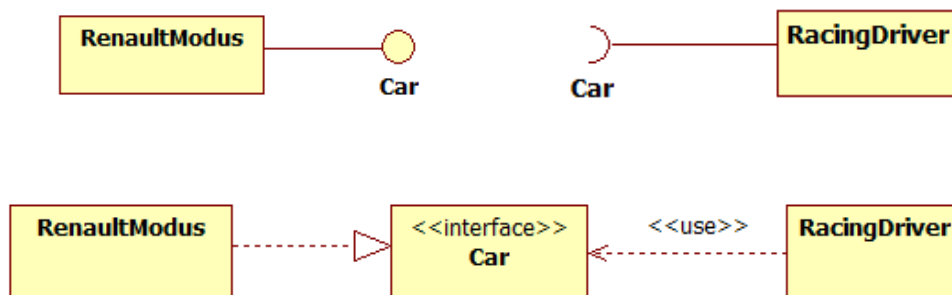


Diagramme de classes : paquetages

• Paquetage

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

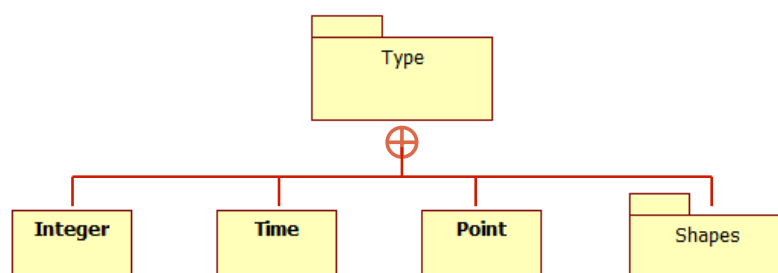
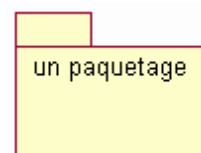


Diagramme de paquetages

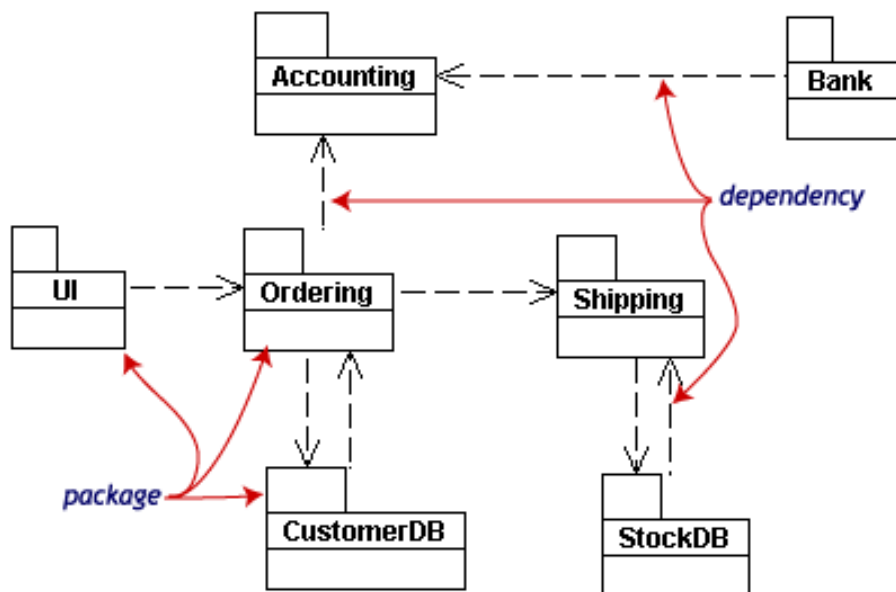
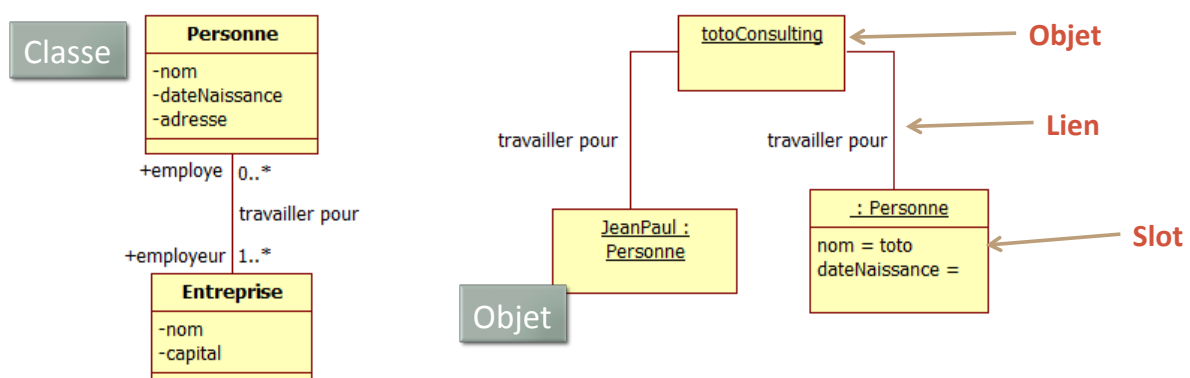


Diagramme objets

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



MODÉLISATION DES APPLICATIONS

Manuele Kirsch Pinheiro

Maître de Conférences – Université Paris 1

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

<http://mkirschp.free.fr>

70

Objectifs et Planning

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

Les diagrammes d'UML

Diagrammes structurels vues statiques

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

Diagrammes comportementaux vues dynamiques

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

Diagramme de séquence

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

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

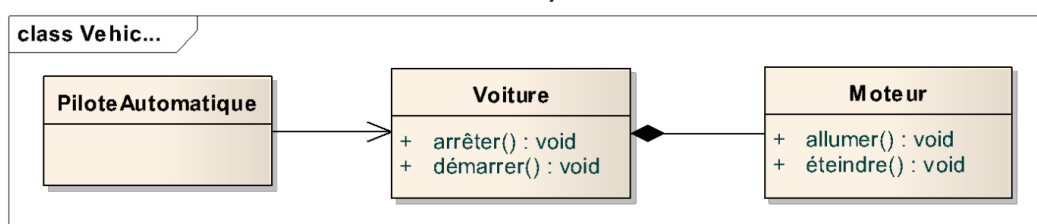


Diagramme de séquence

Le diagramme de séquence montre les interactions sous un angle temporel

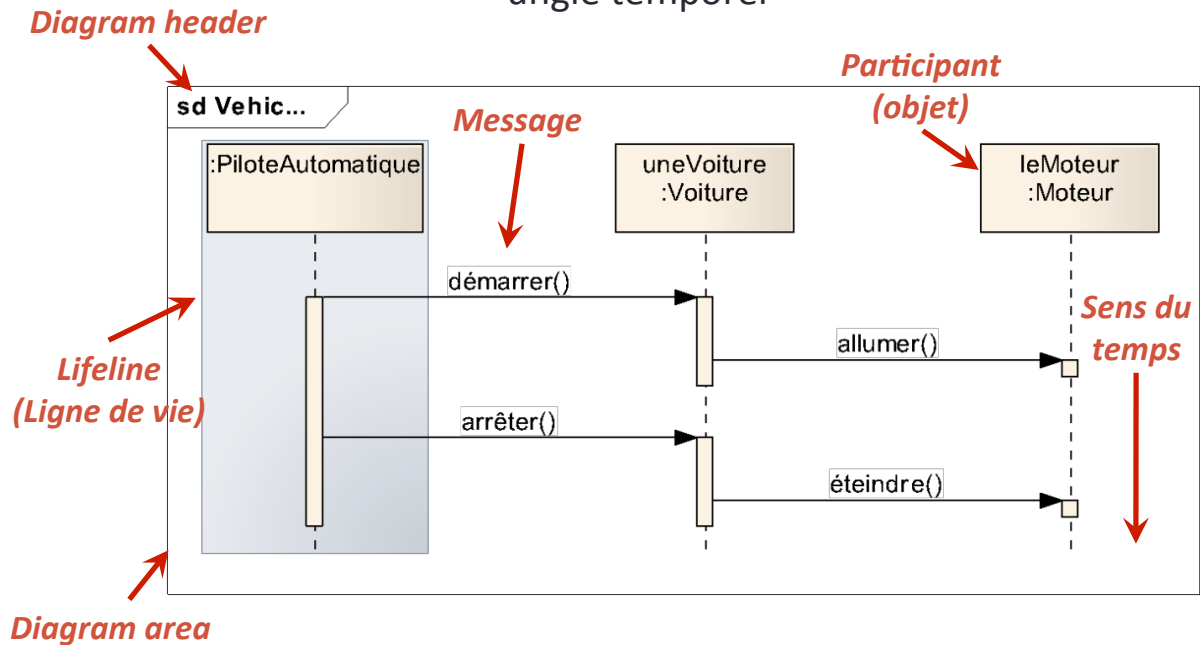


Diagramme de séquence

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

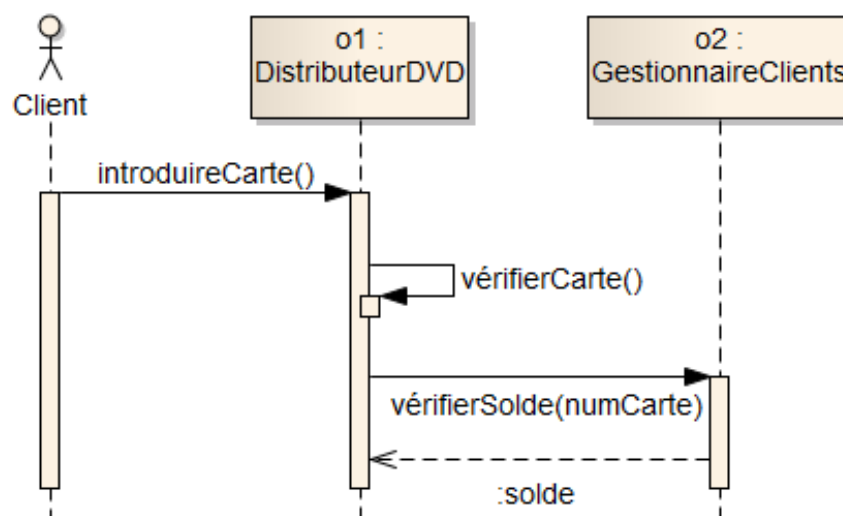


Diagramme de séquence

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

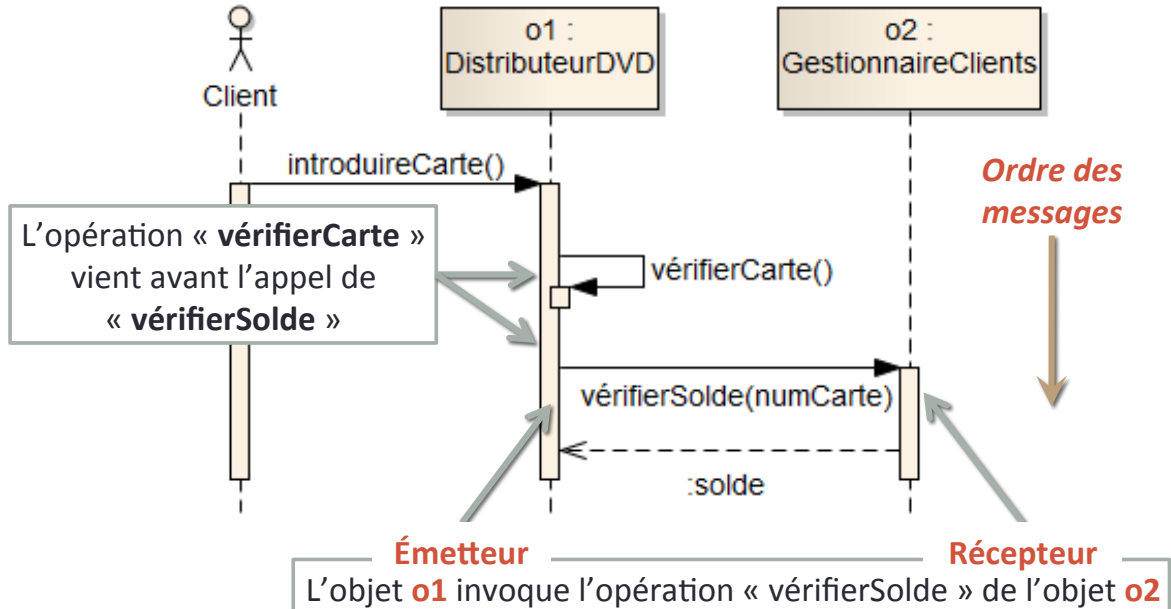


Diagramme de séquence

• Messages

• **Synchrone**

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

• **Asynchrone**

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

• **Retour (reply)**

- Un message peut donner lieu à un retour

• **Création**

- Création d'une nouvelle instance

• **Destruction**

- Destruction d'une instance

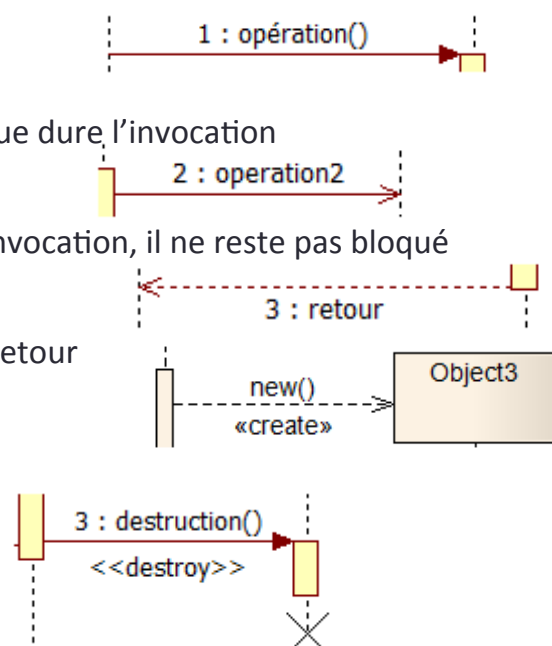


Diagramme de séquence

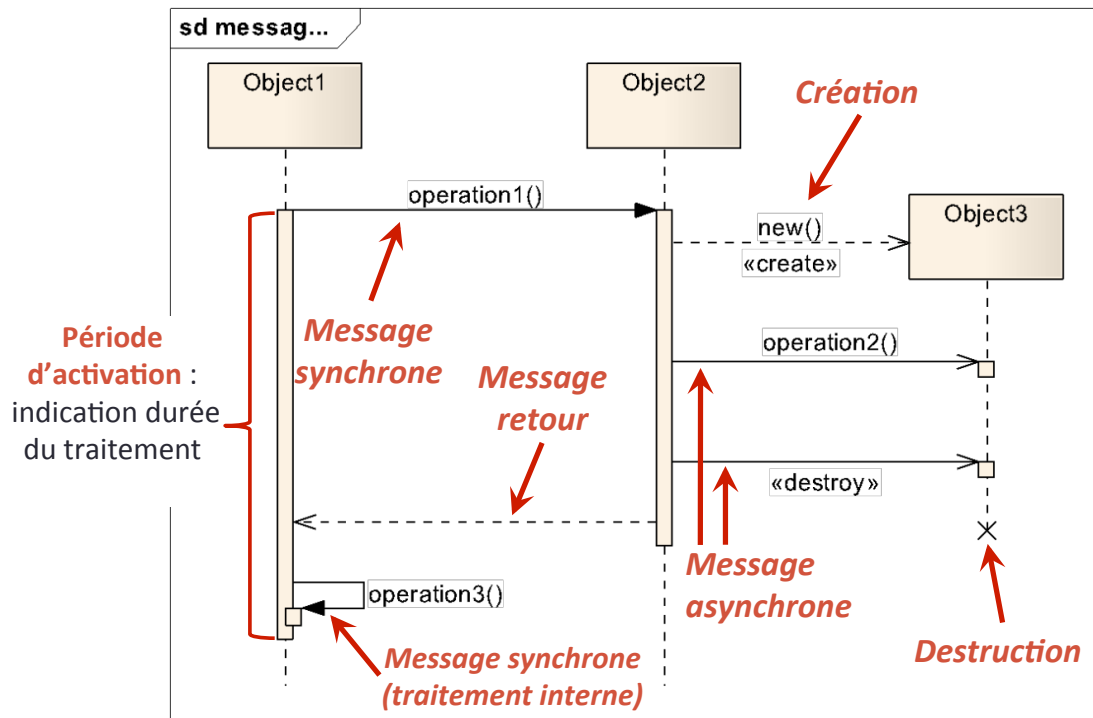


Diagramme de séquence

- **Messages asynchrones**

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

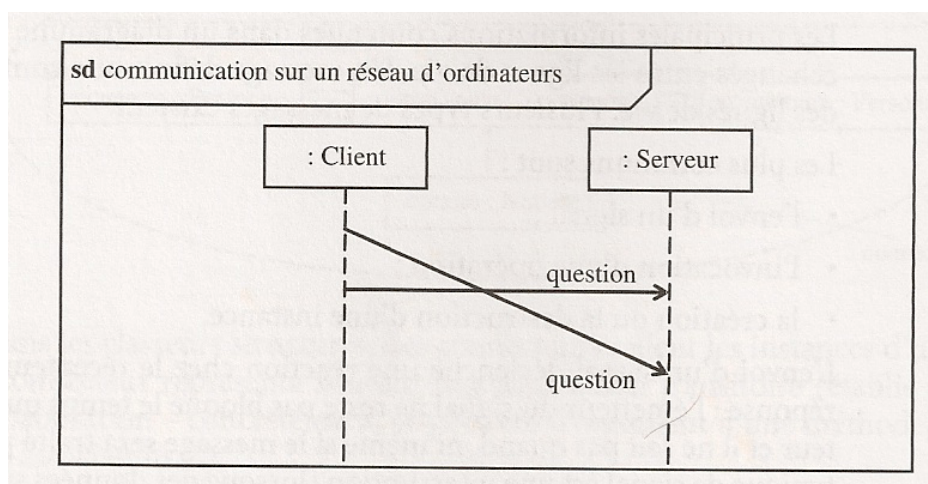


Diagramme de séquence

• Messages perdus

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

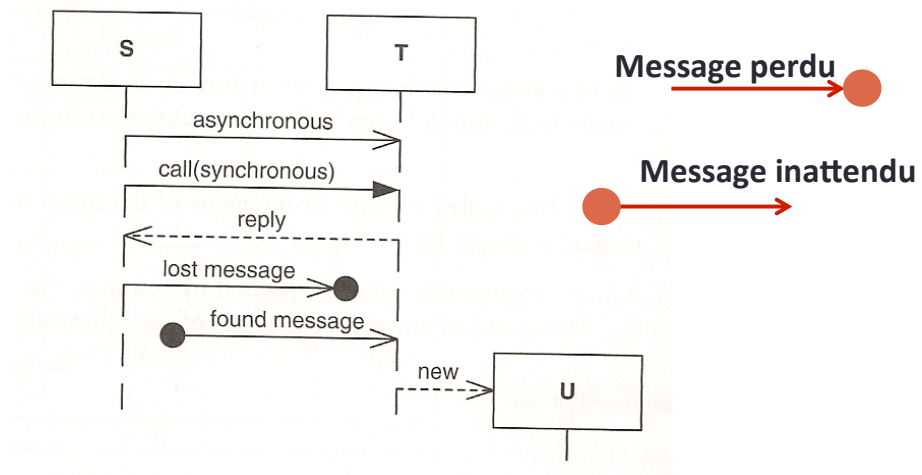
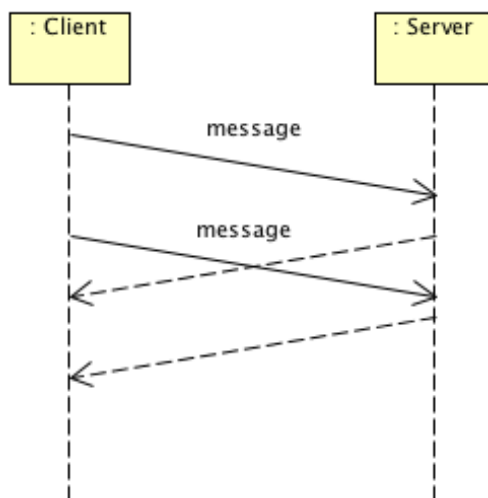


Diagramme de séquence

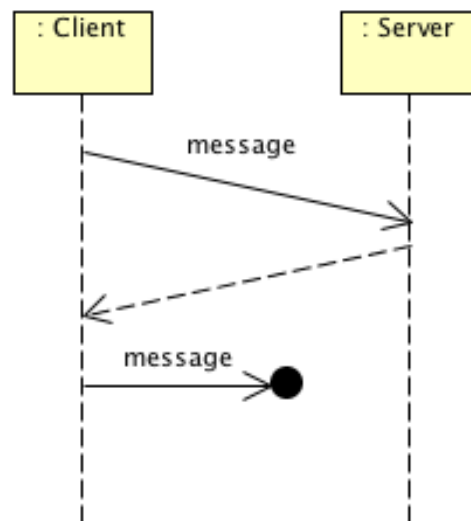
sd Ping Pong C/S



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

Diagramme de séquence

sd Ping-Pong Lost Message



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

Diagramme de séquence

• Scénario :

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

Diagramme de séquence

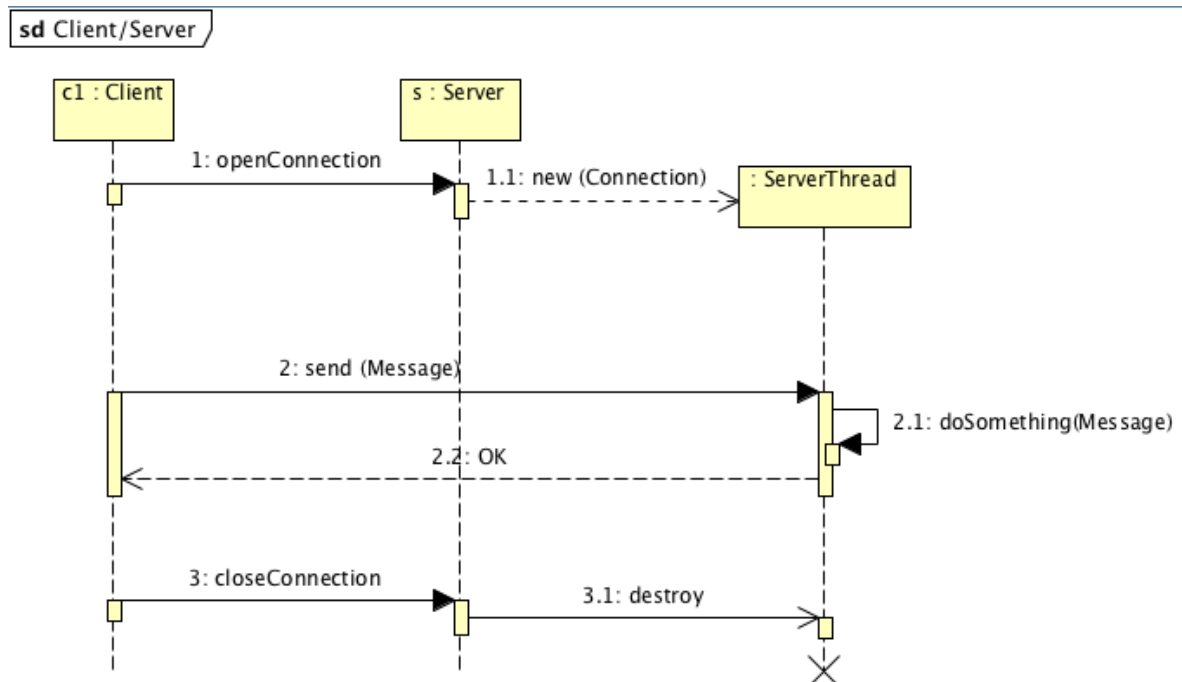
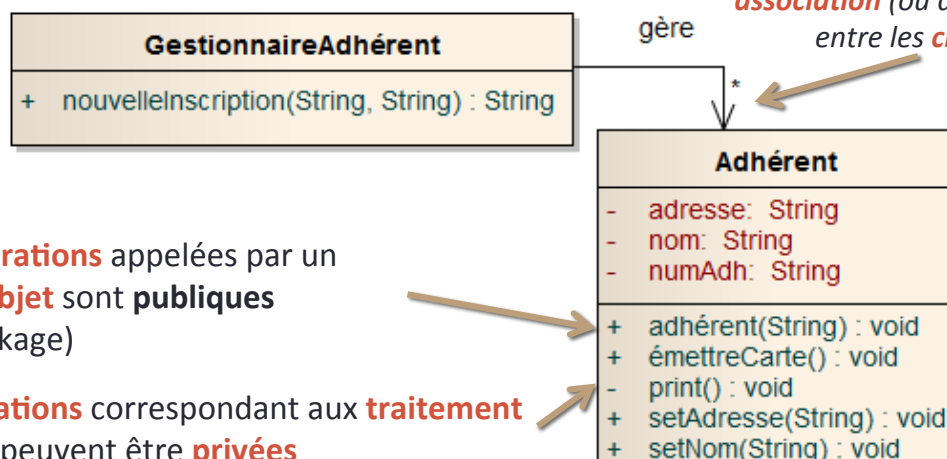


Diagramme de séquence

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



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

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

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

Diagramme de séquence

• Fragment d'interaction

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

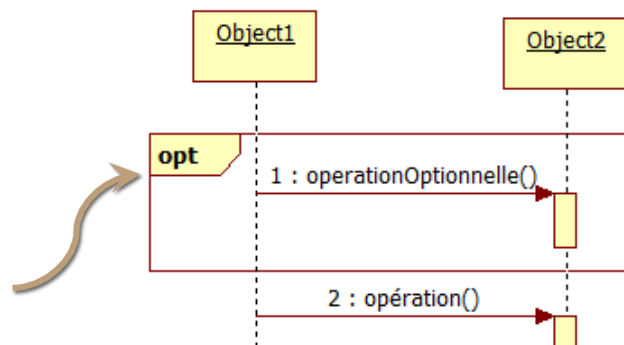


Diagramme de séquence

• Fragment d'interaction

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

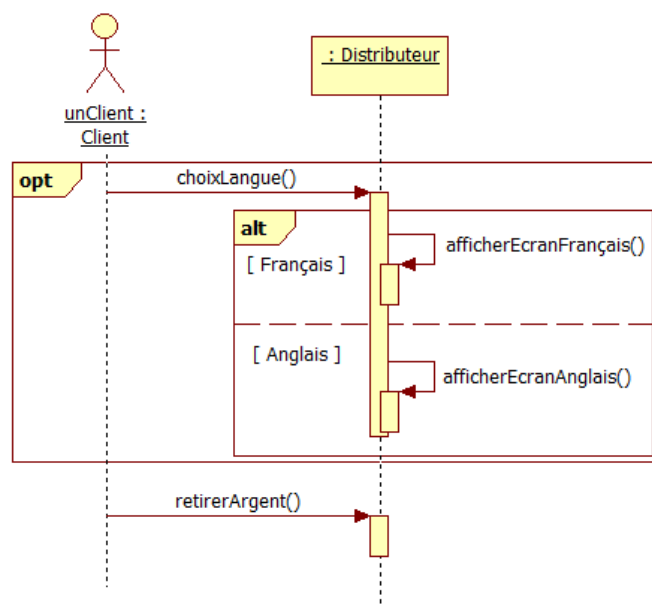


Diagramme de séquence

• Fragment d'interaction

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

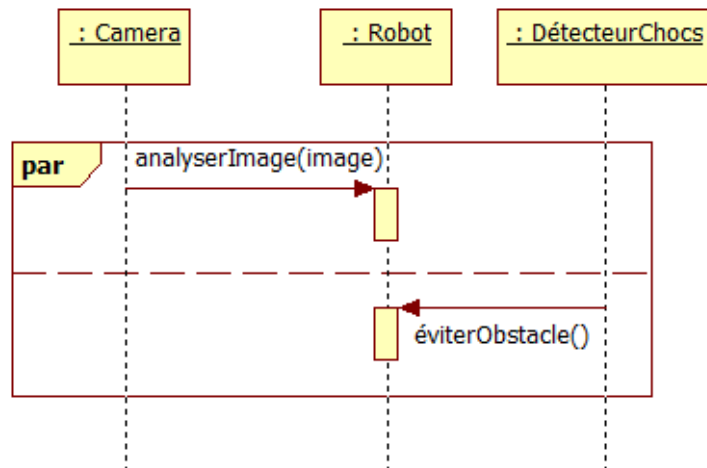


Diagramme de séquence

• Fragment d'interaction

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

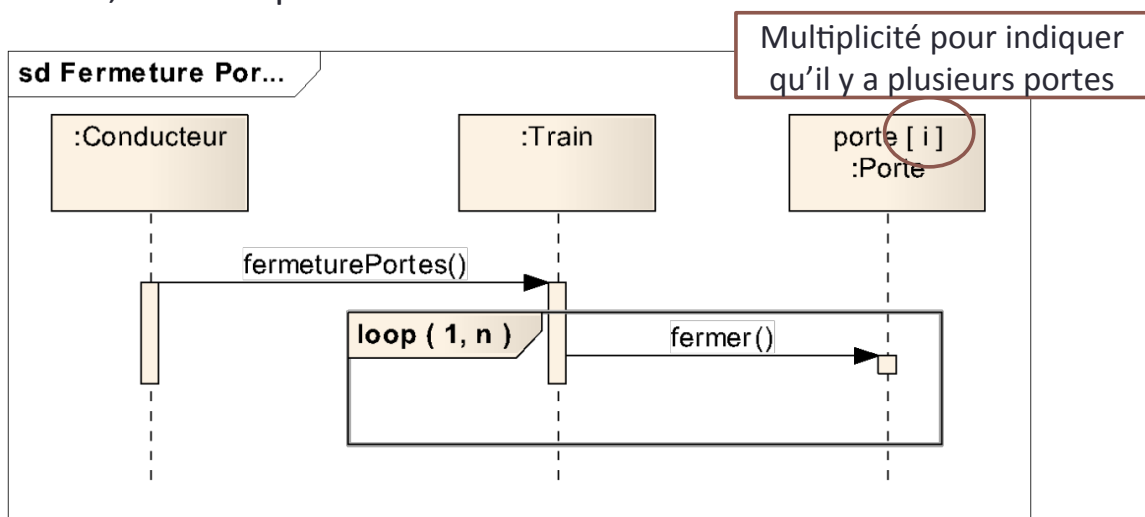


Diagramme de séquence

• Fragment d'interaction

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

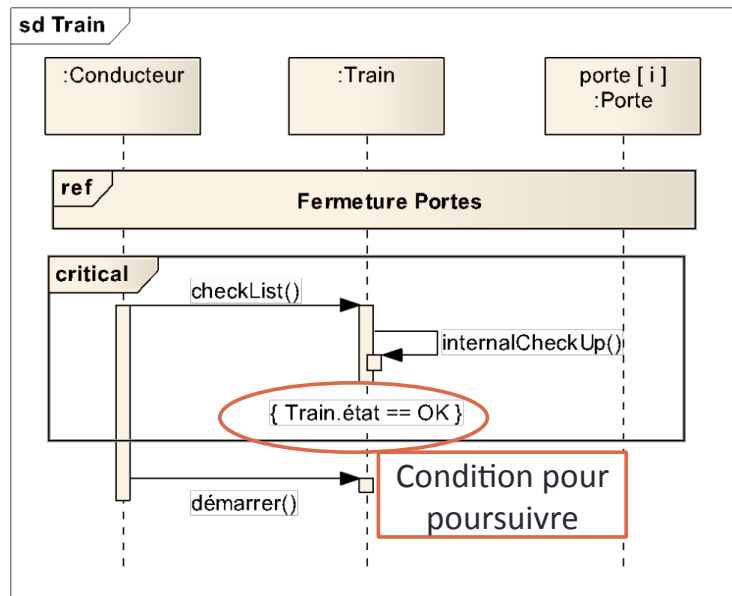
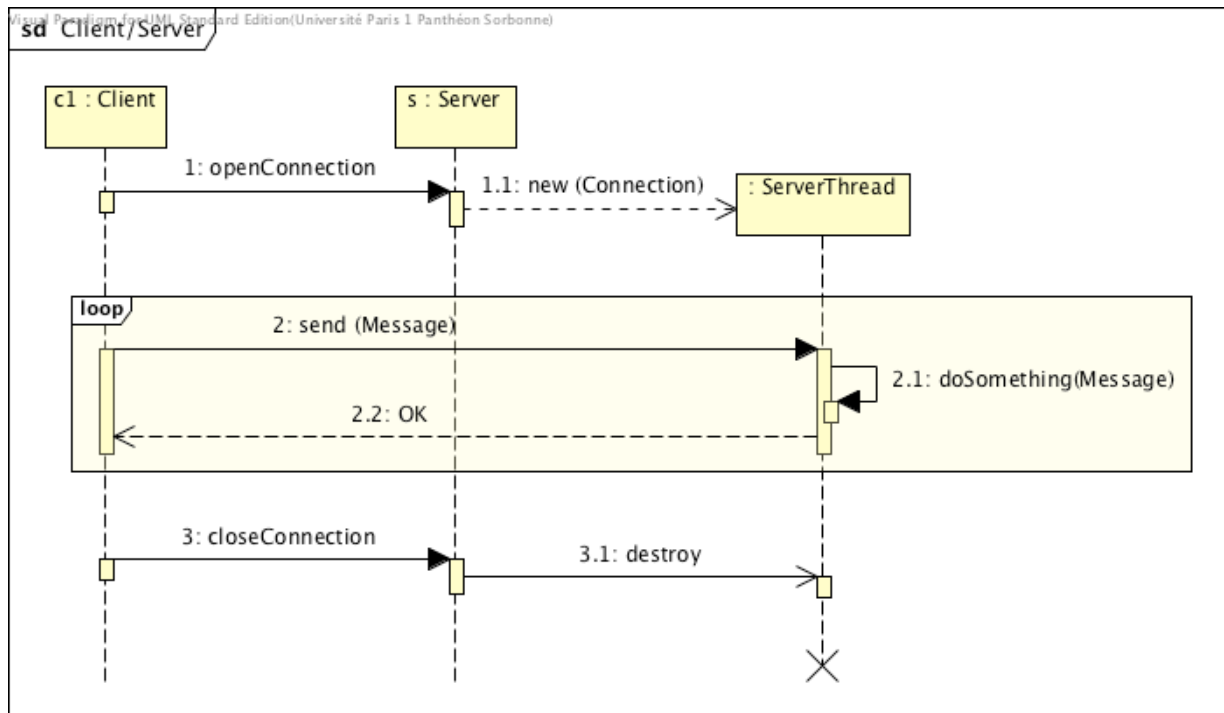


Diagramme de séquence

• Scénario :

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

Diagramme de séquence

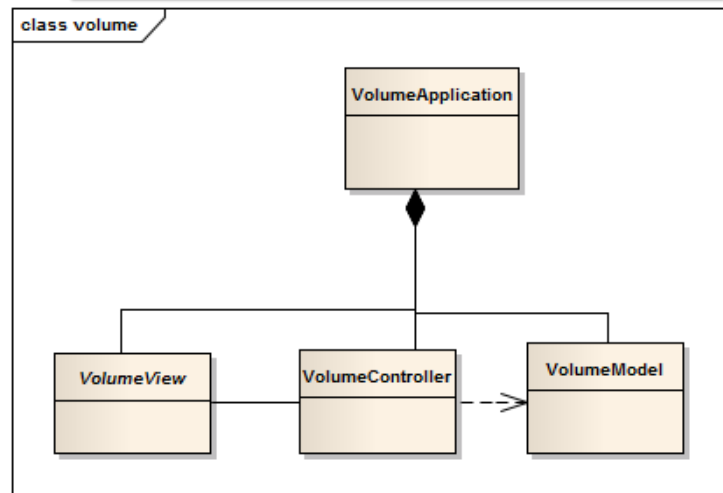


Interaction Diagramme de séquence - Diagramme de classes

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

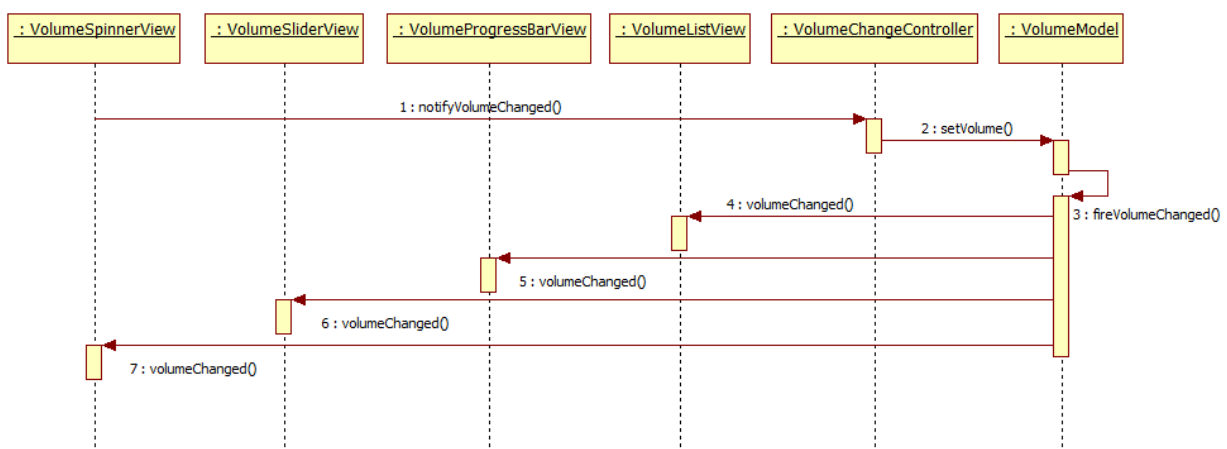
Interaction Diagramme de séquence - Diagramme de classes

Exemple : Application de gestion de volume



Interaction Diagramme de séquence - Diagramme de classes

Exemple : Application de gestion de volume



Quelles classes participent à cette interaction ?

Quelles opérations proposent-elles ?

Avons-nous des interfaces communes à ces classes ?

