ULB UNIVERSITÉ LIBRE DE BRUXELLES

Introduction à l'Orienté Objet: Introduction

Hugues Bersini
CoDE-IRIDIA, ULB

Plan

- Intro à l'Orienté Objet
- Intro à l'UML
- Réutilisation : Modularité, Flexibilité, Design Patterns

Concepts de programmation

- Variables
- Expressions
- Instructions / Assignation
- Pointeurs / Références / Adresses
- Fonctions / Procédures
- Structures de contrôle : tests, boucles

ULB UNIVERSITÉ LIBRE DE BRUXELLES

CoDe

Introduction à l'OO

Concepts de base

Questions au public

- Que savez-vous de l'Orienté Objet ?
- Quels mots-clés connaissez-vous ?
- Comment définir l'Orienté Objet ?

Les raisons du succès de l'OO

- Les **langages**: tout commence avec les langages: simula, smalltalk,eiffel,C++,delphi,java,C#,Python
  - Aussi vieux que l'approche procédurale ou l'approche IA

- La **modélisation**: un nouveau standard UML, l'outil d'analyse qui transforme programmation en modélisation

De la programmation fonctionnelle à la POO
Introduction à la POO

- Les éléments clefs sont les **classes** et les **objets**. Toute application est éclatée entre les différentes classes intervenant dans le problème.
- La classe est le **type** de l'objet, l'objet est **l'instance** physiquement réalisée de la classe, qui n'a d'existence que pendant l'exécution du programme.
- Classe: CompteEnBanque - Objet : Compte_210_024....
  : Vélo - : Le préféré de Merckx
- Un POO est une société de classes qui collaborent entre elles en se chargeant mutuellement de certaines tâches. Elles s'échangent des messages. Seuls ces échanges entre classes peuvent avoir lieu
  ⇨ le programme ne se réduit qu'à cela.

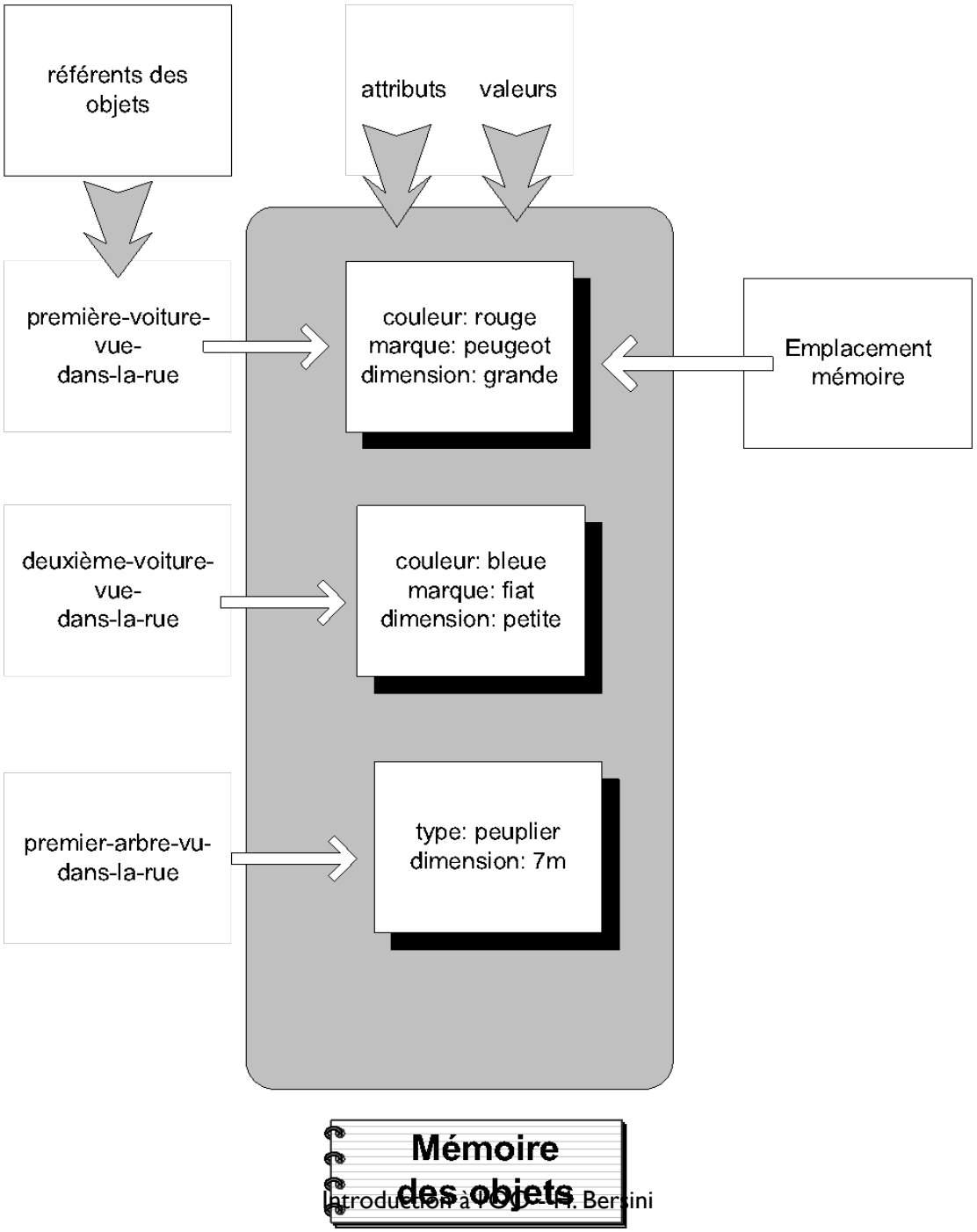
Classes et Objets

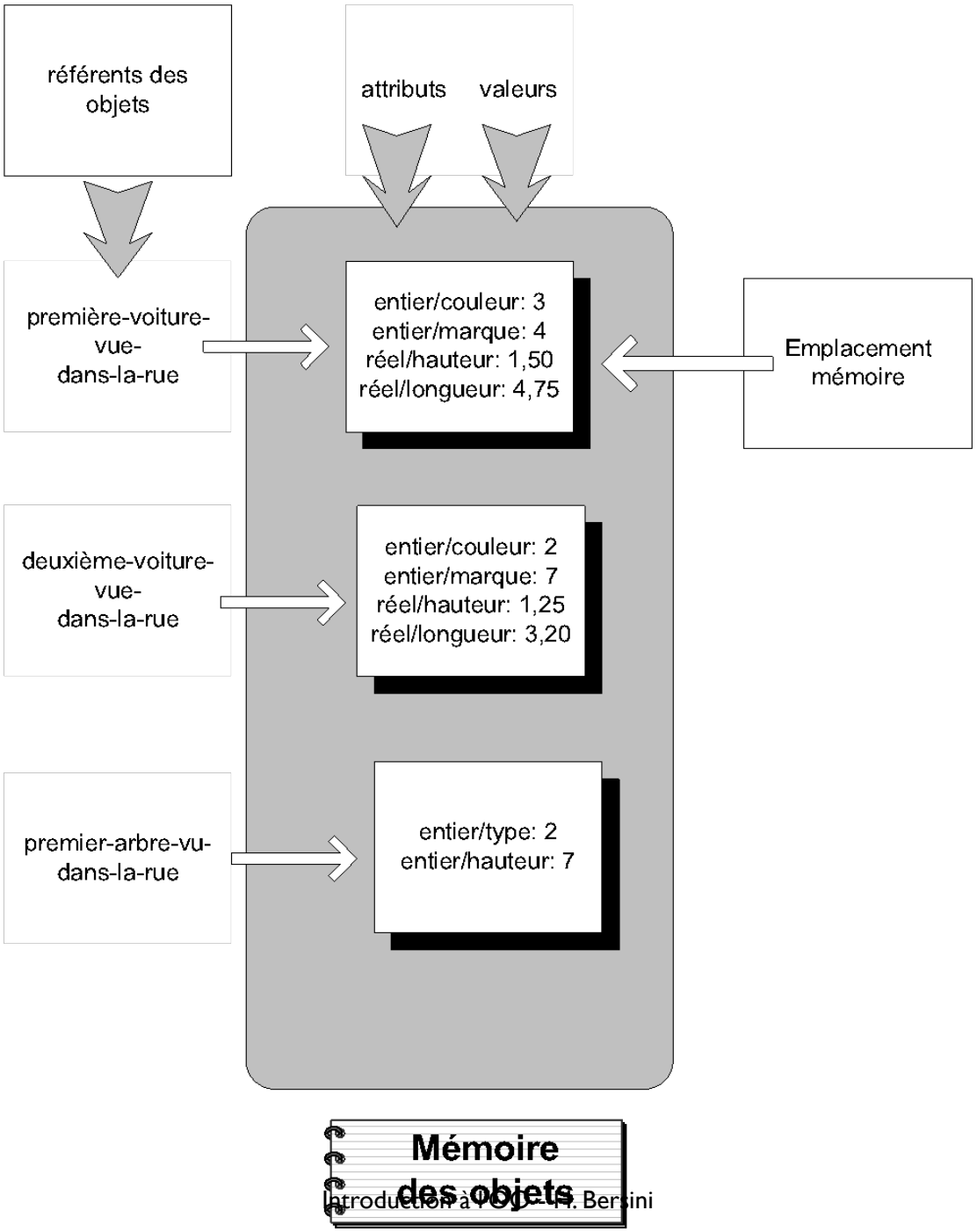
- Une **classe** peut être vue comme un nouveau type de donnée, un modèle, qui possède :
  - ses propres **variables** (attributs)
  - son propre **comportement** (méthodes ou fonctions de classe).
- Par exemple : une personne, un vélo, …

- Un **objet** est une **instance** d'une classe qui a une existence propre.
- Par exemple : la personne Jean Dupont, le vélo vert de Frédéric, …

Le trio <entité, attribut, valeur>

- Un objet = une entité = un ensemble d'attributs avec leur valeur
- Les objets sont stockés en mémoire et référés
- Les attributs sont de type primitif: réel, entier, booléens,…
- Cela ressemble à des enregistrements dans des bases de données relationnelles
- Regardez dans la rue ….


référents des objets
attributs
valeurs
première-voiture-vue-dans-la-rue
couleur: rouge
marque: peugeot
dimension: grande
Emplacement mémoire
deuxième-voiture-vue-dans-la-rue
couleur: bleue
marque: fiat
dimension: petite
premier-arbre-vu-dans-la-rue
type: peuplier
dimension: 7m
Mémoire des objets

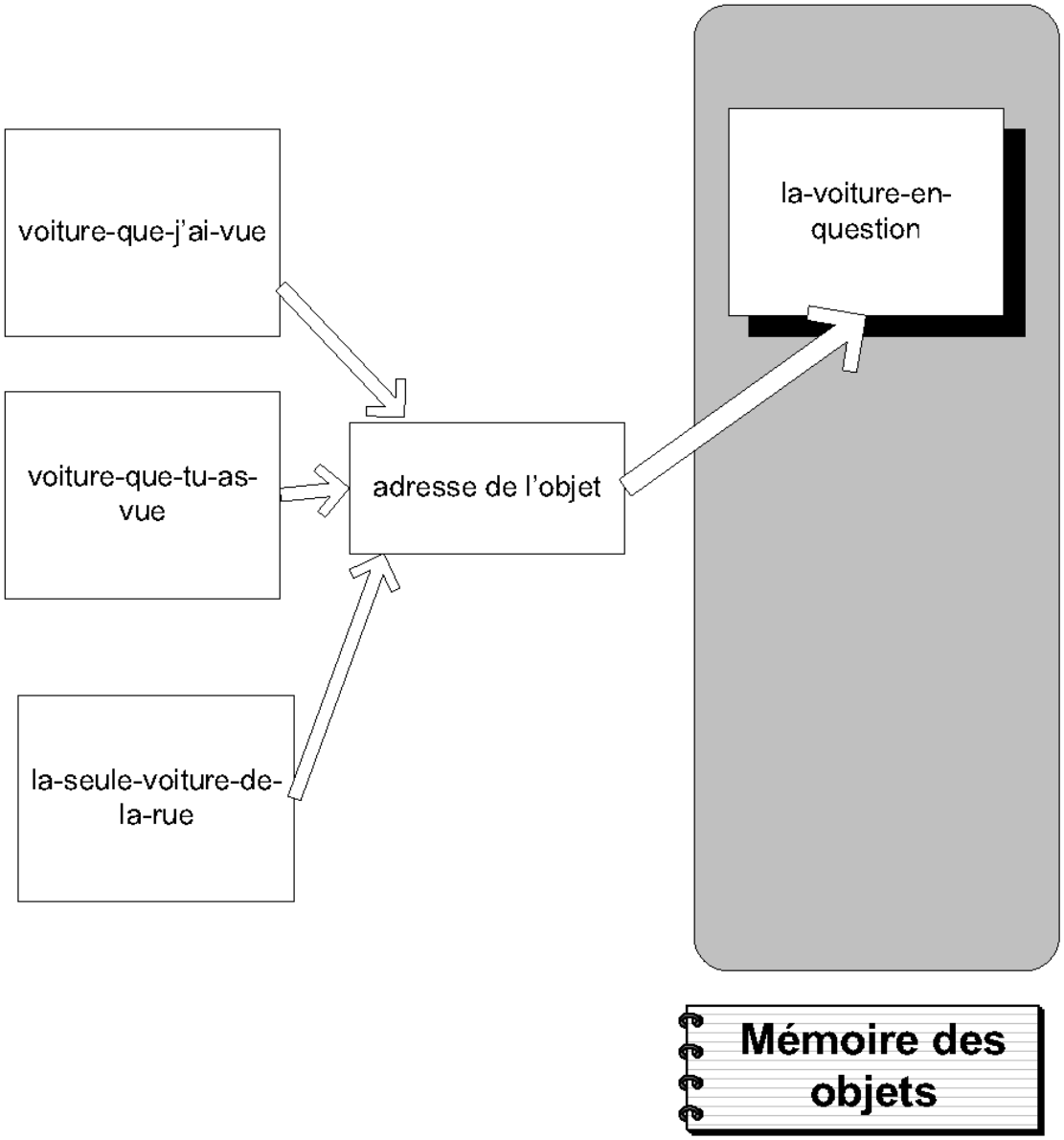

référents des objets
attributs
valeurs
première-voiture-vue-dans-la-rue
entier/couleur: 3
entier/marque: 4
réel/hauteur: 1,50
réel/longueur: 4,75
Emplacement mémoire
deuxième-voiture-vue-dans-la-rue
entier/couleur: 2
entier/marque: 7
réel/hauteur: 1,25
réel/longueur: 3,20
premier-arbre-vu-dans-la-rue
entier/type: 2
entier/hauteur: 7
Mémoire des objets

Marque	Modele	Serie	Numero
Renault	18	RL	4698 SJ 45
Renault	Kangoo	RL	4568 HD 16
Renault	Kangoo	RL	6576 VE 38
Peugeot	106	KID	7845 ZS 83
Peugeot	309	chorus	7647 ABY 82
Ford	Escort	Match	8562 EV 23

Table d'une base de données relationnelles

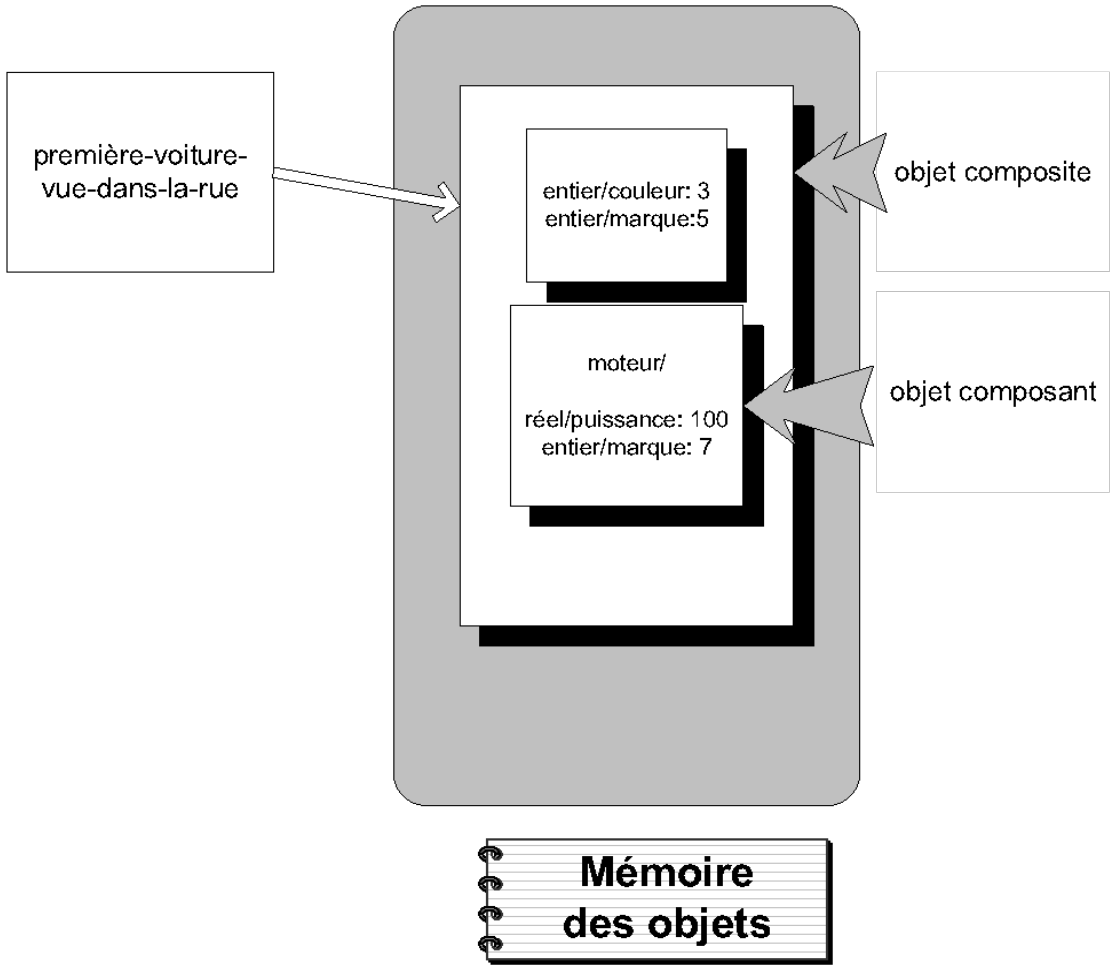
Le référent d'un objet

- C'est le nom et son moyen d'accès: son adresse physique
- Plusieurs référents peuvent référer un même objet: adressage indirect


voiture-que-j'ai-vue
voiture-que-tu-as-vue
la-seule-voiture-de-la-rue
adresse de l'objet
la-voiture-en-question
Mémoire des objets

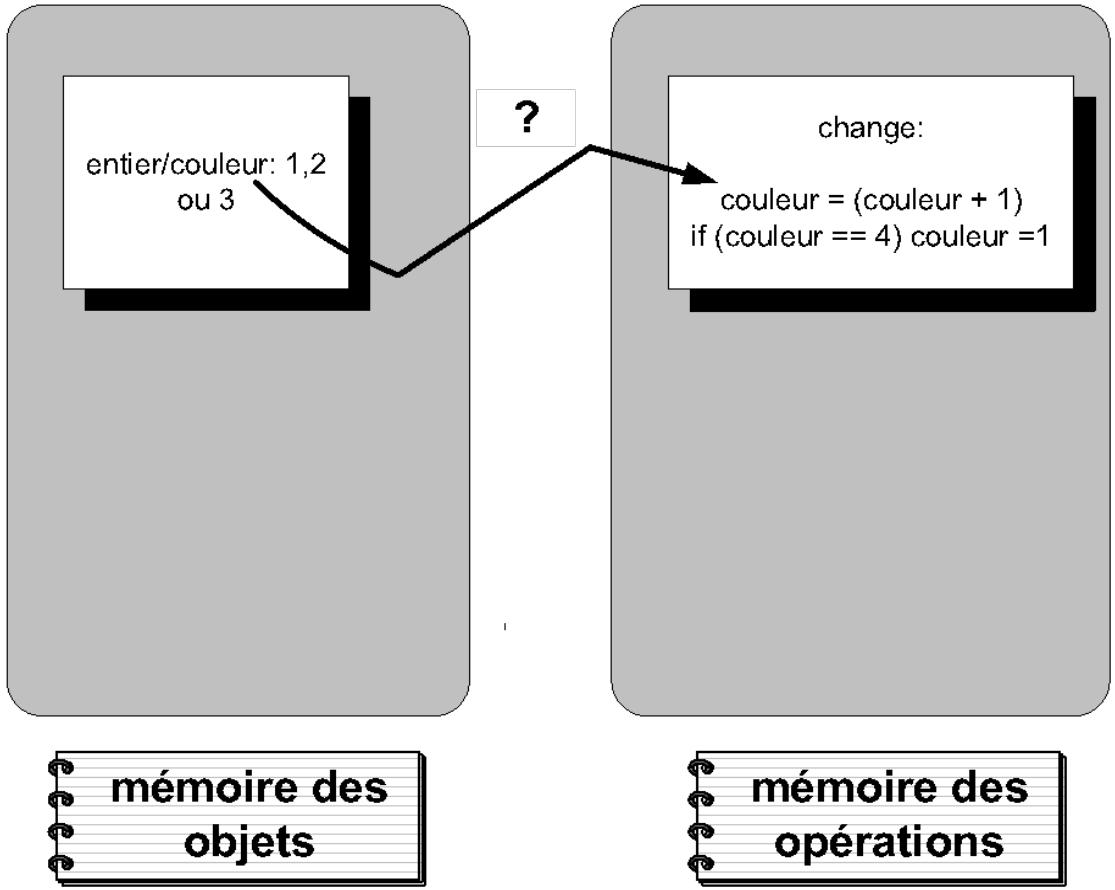
L'objet dans sa version passive

- L'objet et ses constituants:
  - L'objet = un ensemble d'attributs/valeurs
- L'objet peut être un composite d'objets


première-voiture-vue-dans-la-rue
entier/couleur: 3
entier/marque:5
moteur/
réel/puissance: 100
entier/marque: 7
objet composite
objet composant
Mémoire des objets

L'objet dans sa version active

- Les objets changent d'état
- Le cycle de vie d'un objet se limite à une succession de changements d'états jusqu'à disparaître de la RAM
- Mais qui est la cause des changement d'états ?
  - LES METHODES
- Soit l'objet feu de signalisation avec sa couleur et ses trois valeurs


entier/couleur: 1,2
ou 3
?
change:
couleur = (couleur + 1)
if (couleur == 4) couleur =1
mémoire des objets
mémoire des opérations

Mais comment relier l'opération « change » à l'attribut couleur ??

LA CLASSE

La classe

- La classe unit les méthodes aux attributs:

```
class Feu-de-signalisation {
  int couleur ;
  change() {
    couleur = couleur + 1 ;
    if (couleur ==4) couleur = 1 ;
  }
}
```

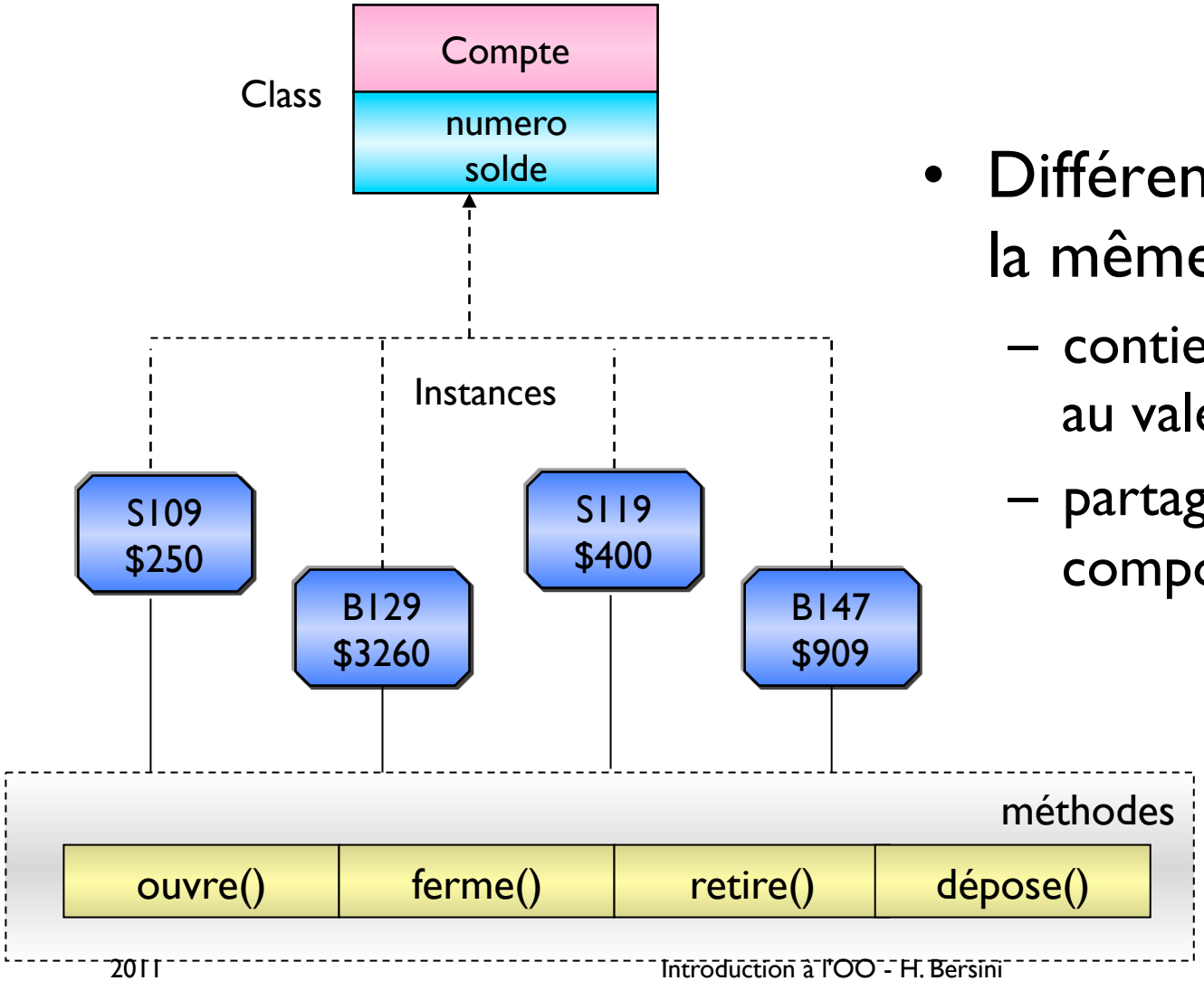
Sur quel objet précis s'exerce la méthode

```
feu-de-signalisation-en-question.change()
```

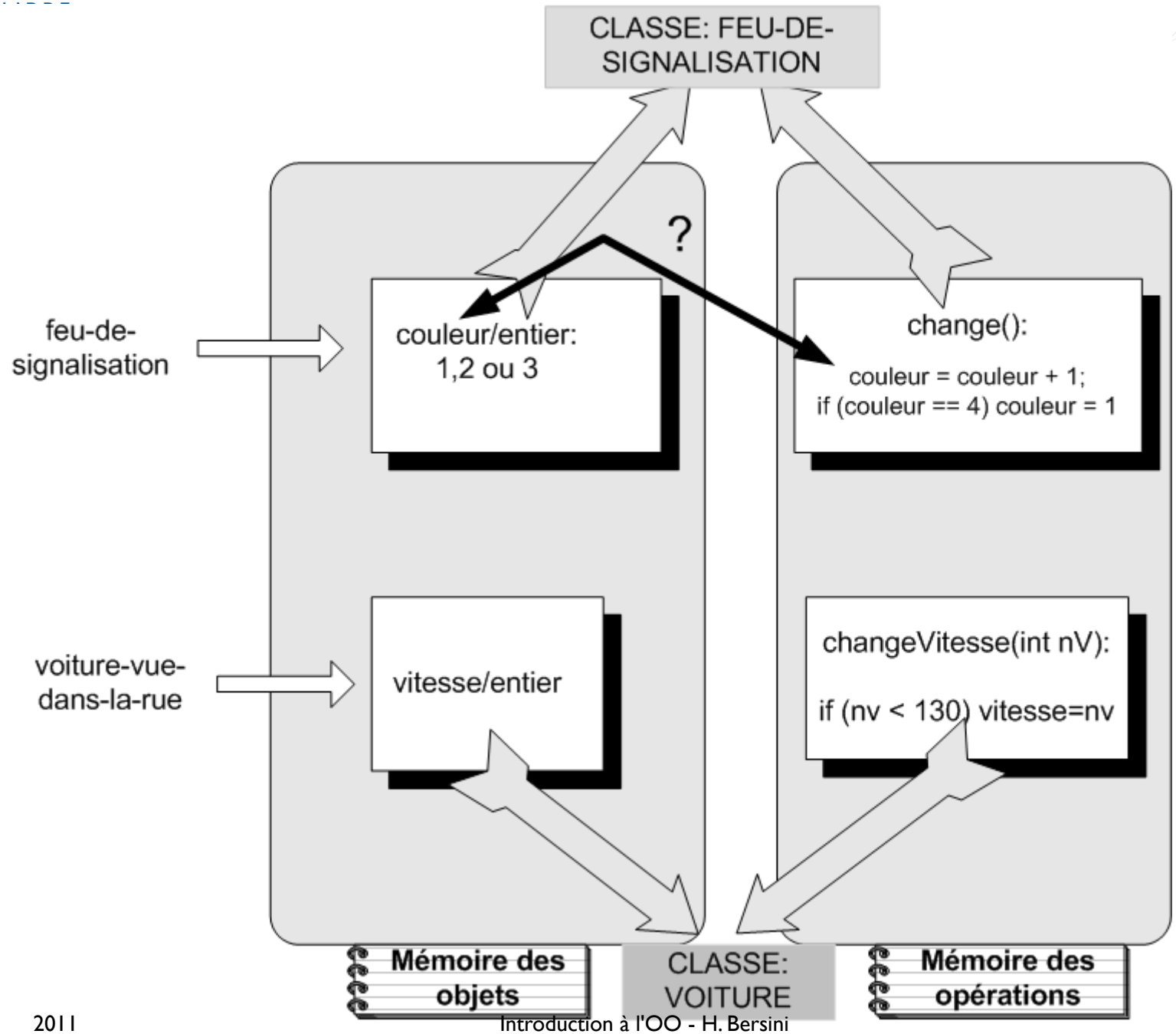
On lie la méthode f(x) à l'objet « a » par l'instruction:

a.f(x)

Les objets ne diffèrent que par leurs attributs

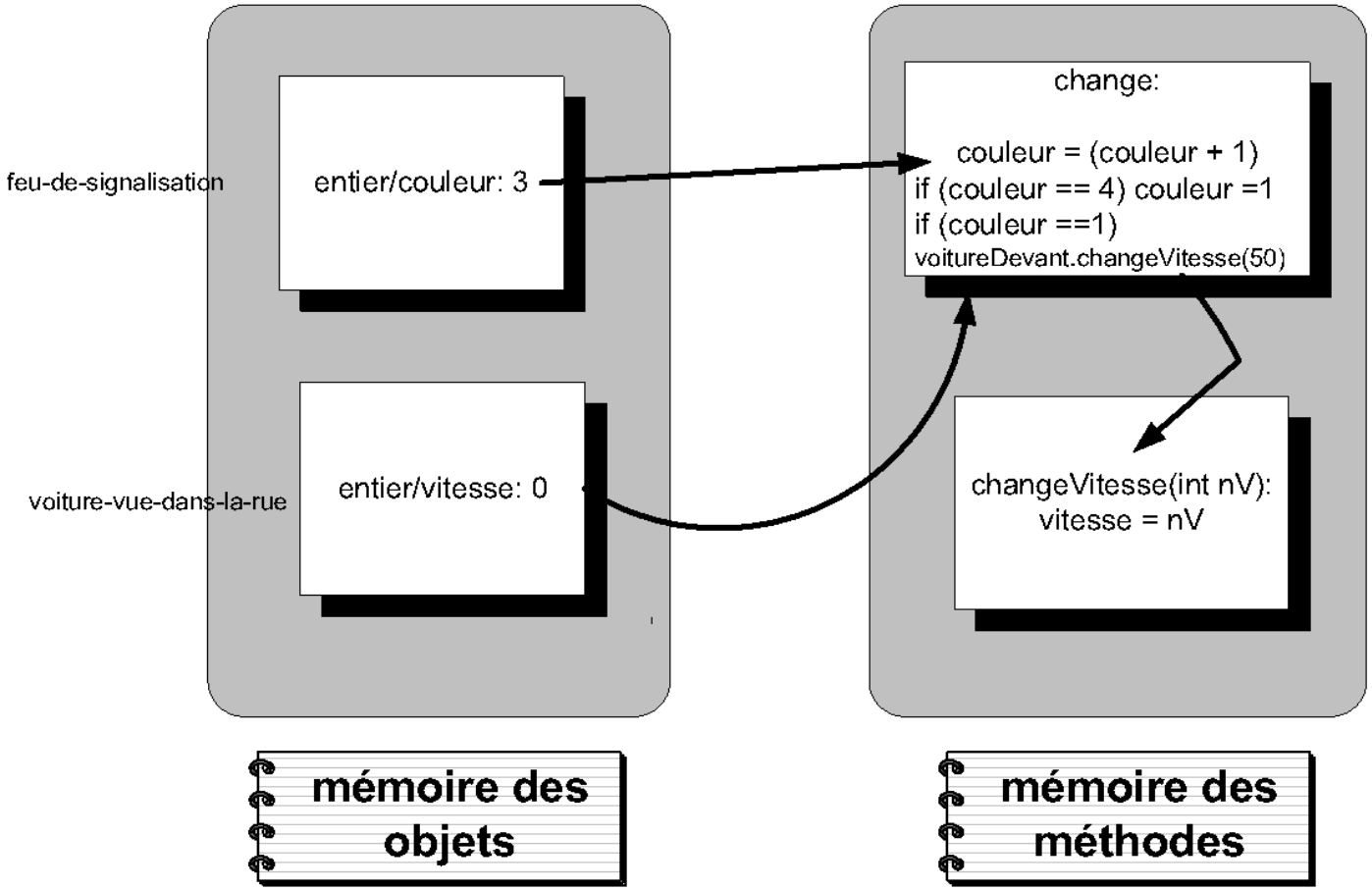


- Différentes instances de la même classe
  - contiennent des attributs au valeur différente
  - partagent un même comportement


CLASSE: FEU-DE-SIGNALISATION
?
feu-de-signalisation
couleur/entier:
1,2 ou 3
change():
couleur = couleur + 1;
if (couleur == 4) couleur = 1
voiture-vue-dans-la-rue
vitesse/entier
changeVitesse(int nV):
if (nv < 130) vitesse=nv
Mémoire des objets
CLASSE: VOITURE
Mémoire des opérations

Des objets en interaction

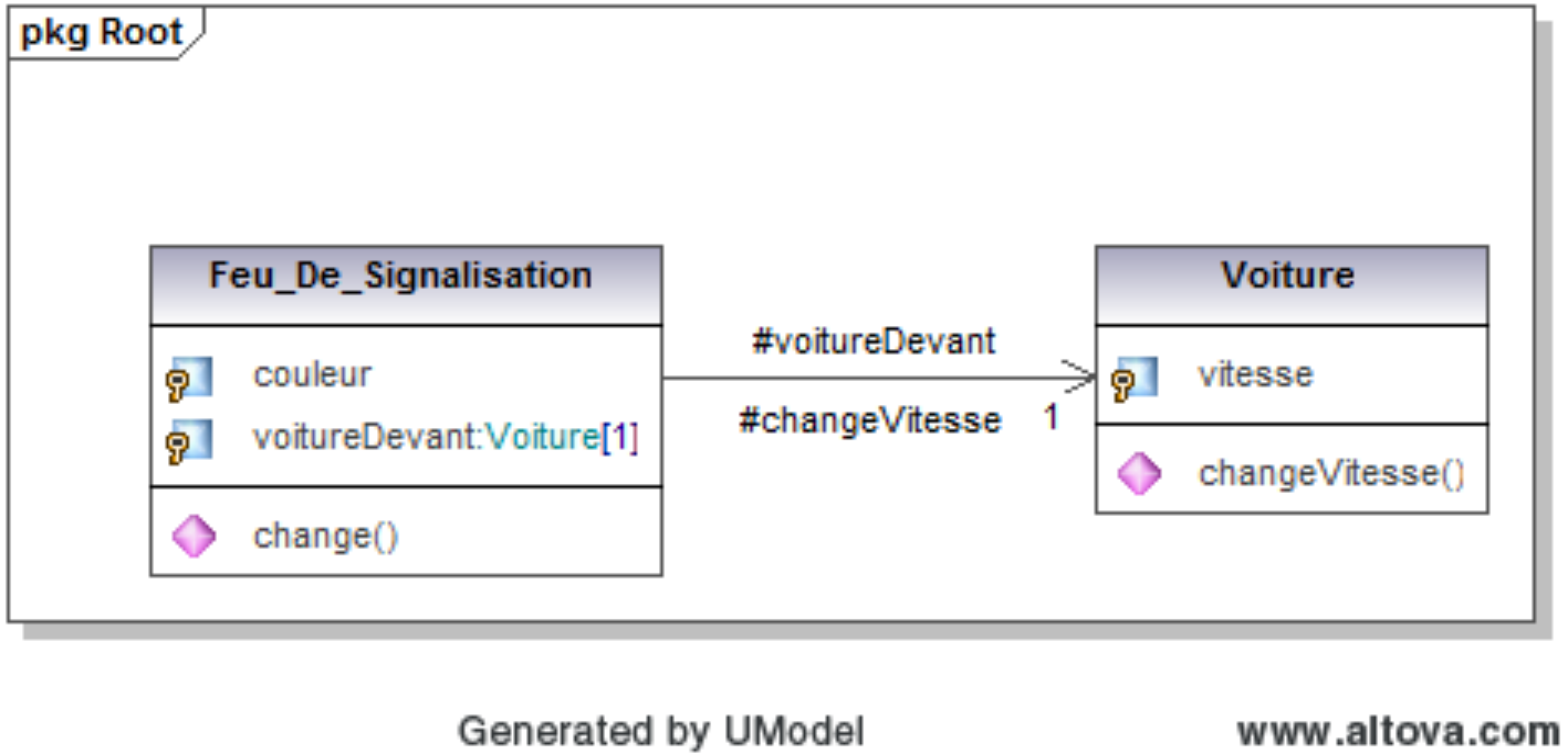
- La base de l' informatique OO: communication entre les objets
- Supposons que le feu fasse ralentir ou accélérer la voiture


feu-de-signalisation
entier/couleur: 3
voiture-vue-dans-la-rue
entier/vitesse: 0
change:
couleur = (couleur + 1)
if (couleur == 4) couleur =1
if (couleur ==1)
voitureDevant.changeVitesse(50)
changeVitesse(int nV):
vitesse = nV
mémoire des objets
mémoire des méthodes

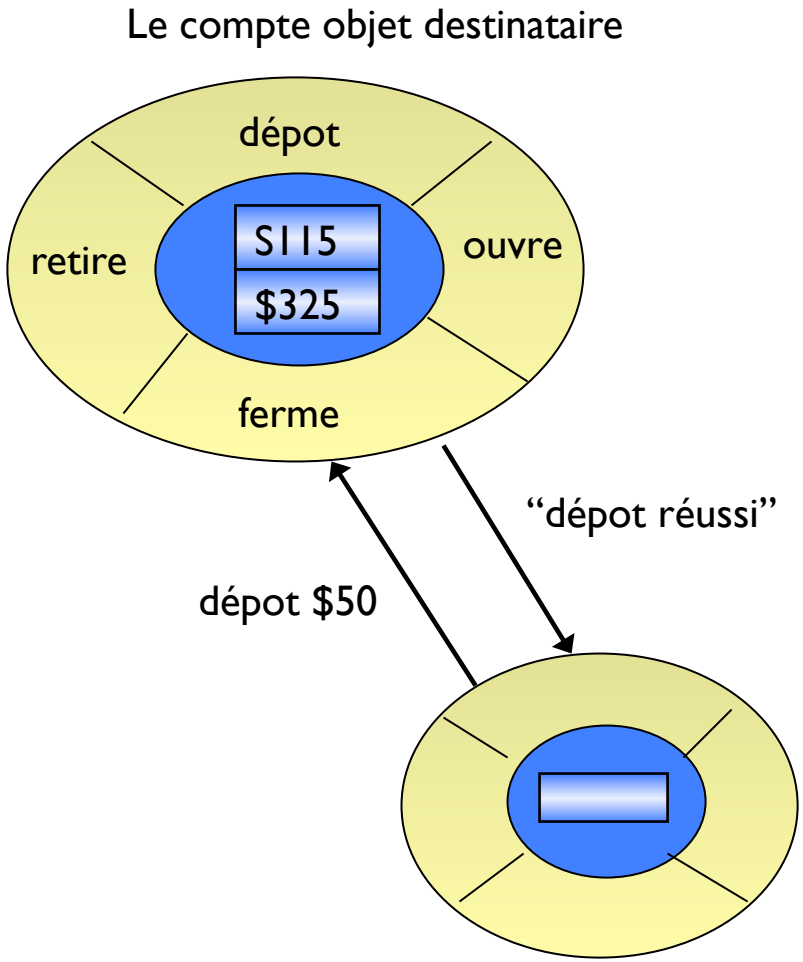
- Les objets communiquent par envois de messages
- Quand un objet demande à un autre d'exécuter une méthode qui est propre à cet autre.

```
class Feu-de-signalisation {
  int couleur ;
  Voiture voitureDevant;
  change() {
    couleur = couleur + 1 ;
    if (couleur ==4) couleur = 1 ;
    if (couleur ==1)
  voitureDevant.changeVitesse(50) ;
  }
}
```

UML envoi de message



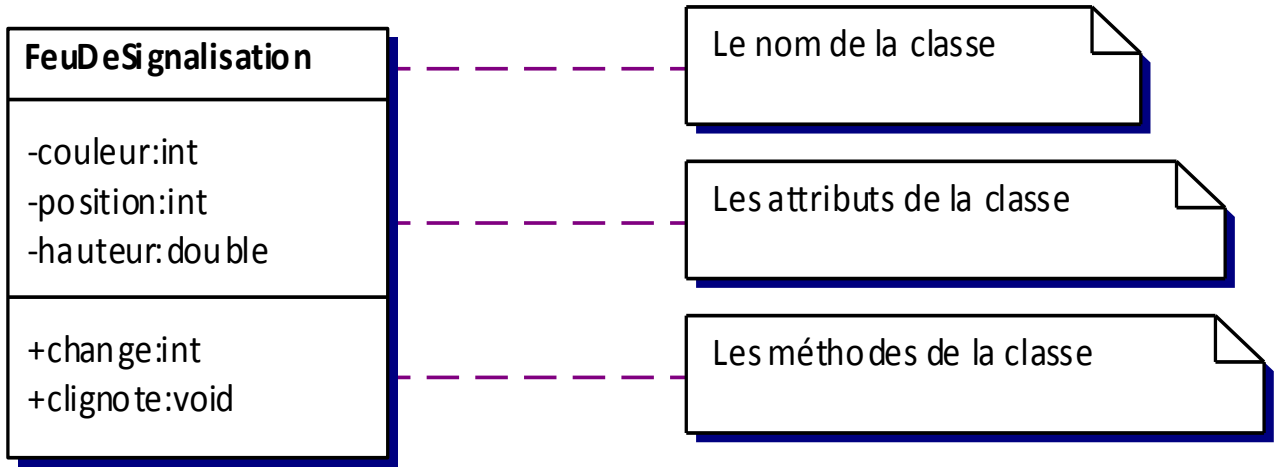
Appel de méthode à distance = “envoi de message”



- Les objets collaborent en s’envoyant des messages
- On parle d’envoi de message car il y a expéditeur et destinataire

Un objet sans classe n'a pas de classe

- Constitution d'une classe d'objet



Définition d'une méthode

```
int change() {
   couleur = couleur + 1 ;
   if couleur == 4 couleur = 1;
     return couleur ; /* la méthode
 retourne un entier */
 }
```

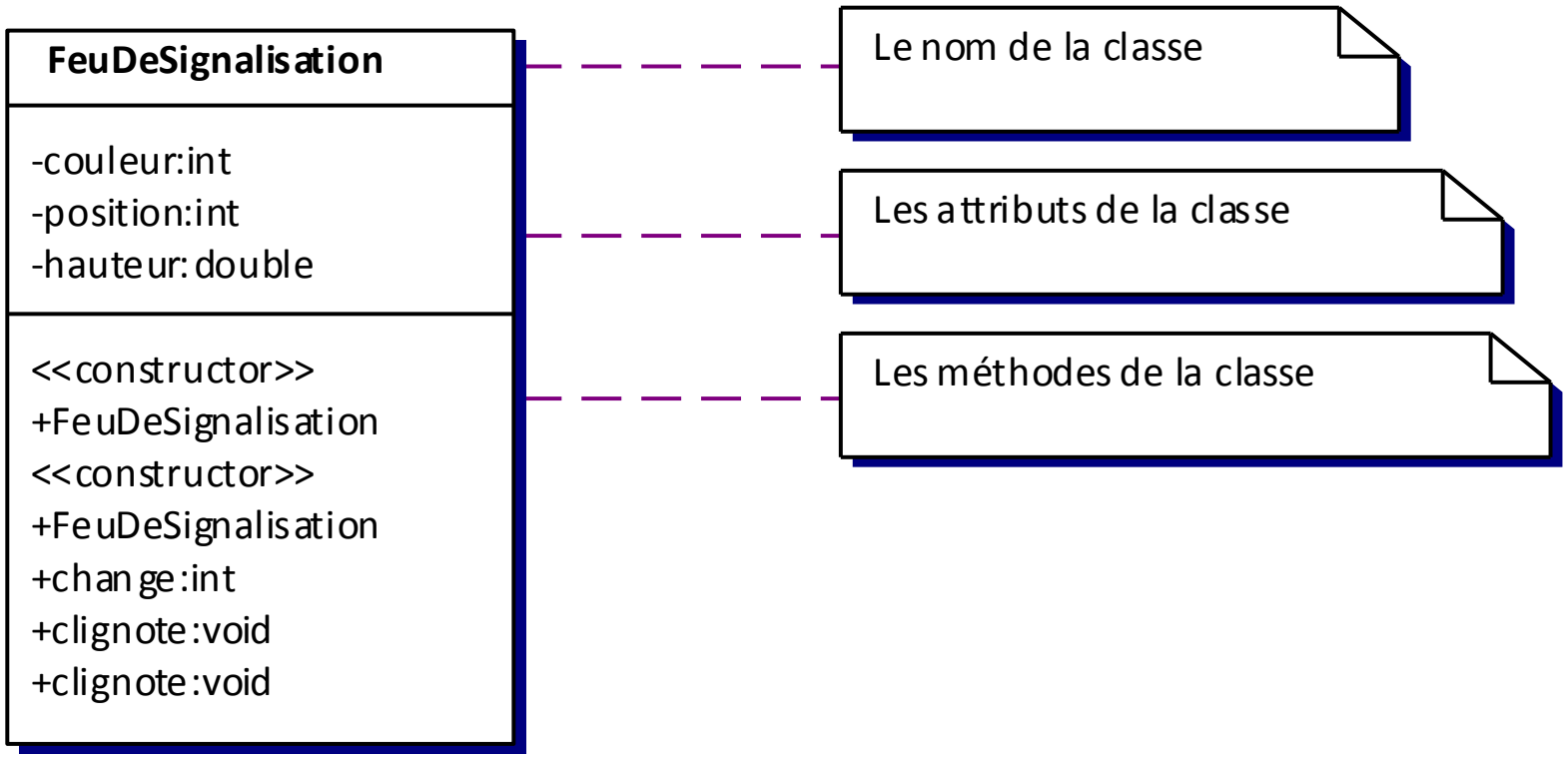
Signature et surcharge de méthodes

```
class FeuDeSignalisation {
  void clignote() {
    System.out.println("premiere manière de
  clignoter");
    for(int i=0; i<2; i++) {
       for (int j=0; j<3; j++)
         System.out.println("je suis eteint");
       for (int j=0; j<3; j++)
         System.out.println("je suis allume");
    }
  }
```

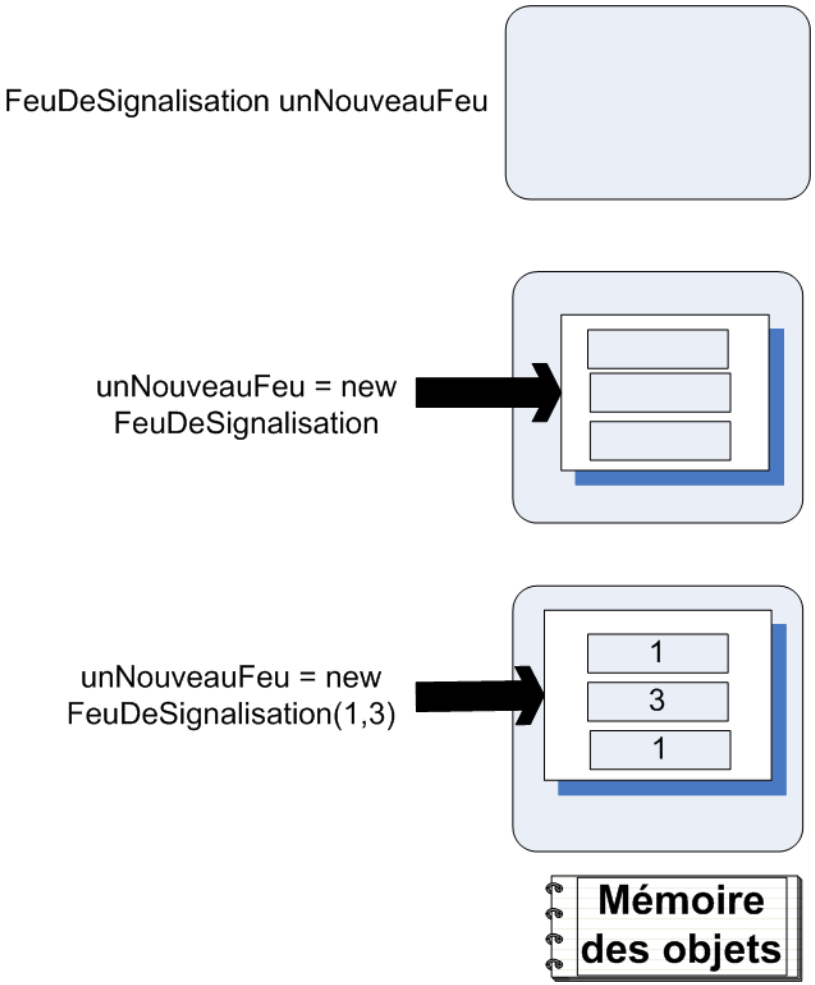
```
void clignote(int a) {
  System.out.println("deuxieme maniere de
clignoter"); /* Affichage de texte à l'écran */
  for(int i=0; i<2; i++) {
    for (int j=0; j<a; j++)  /* on retrouve le
"a" de l'argument */
      System.out.println("je suis eteint");
    for (int j=0; j<a; j++)
      System.out.println("je suis allume");
  }
}
```

```
int clignote(int a, int b) {
    System.out.println("troisieme maniere de
  clignoter");
    for(int i=0; i<2; i++) {
      for (int j=0; j<a; j++)
          System.out.println("je suis eteint");
      for (int j=0; j<b; j++)
          System.out.println("je suis allume");
    }
    return b;
  }
```

Le constructeur



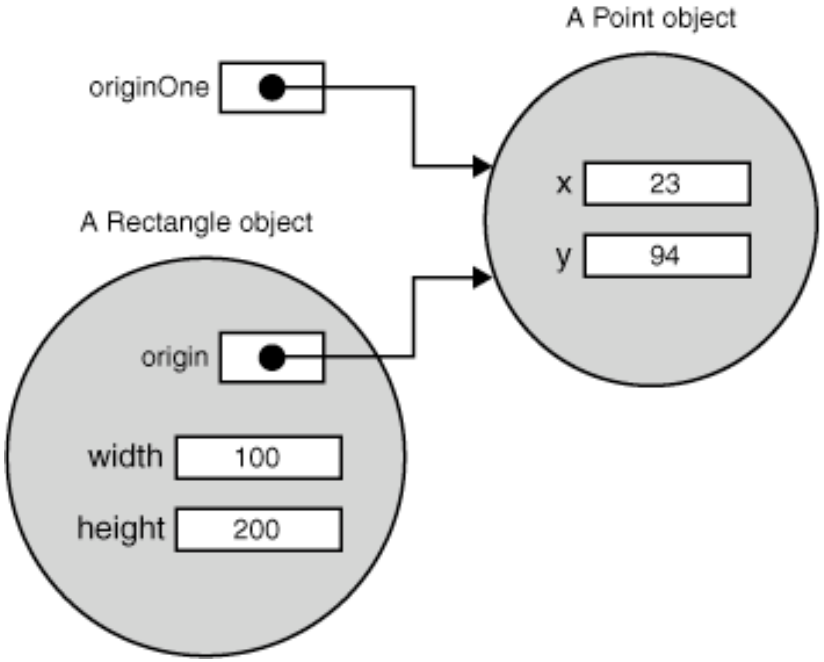
```
FeuDeSignalisation(int positionInit, double
  hauteurInit) {
    /* pas de retour pour le constructeur */
    position = positionInit ;
    hauteur = hauteurInit ;
    couleur = 1 ;
  }

FeuDeSignalisation unNouveauFeu = new
  FeuDeSignalisation(1, 3) ;
```


FeuDeSignalisation unNouveauFeu
unNouveauFeu = new
FeuDeSignalisation
unNouveauFeu = new
FeuDeSignalisation(1,3)
1
3
1
Mémoire
des objets

Objets et références

- Les objets sont toujours utilisés par références.
- Une instance = un `new`.

```
Point originOne = new Point(23, 94);
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

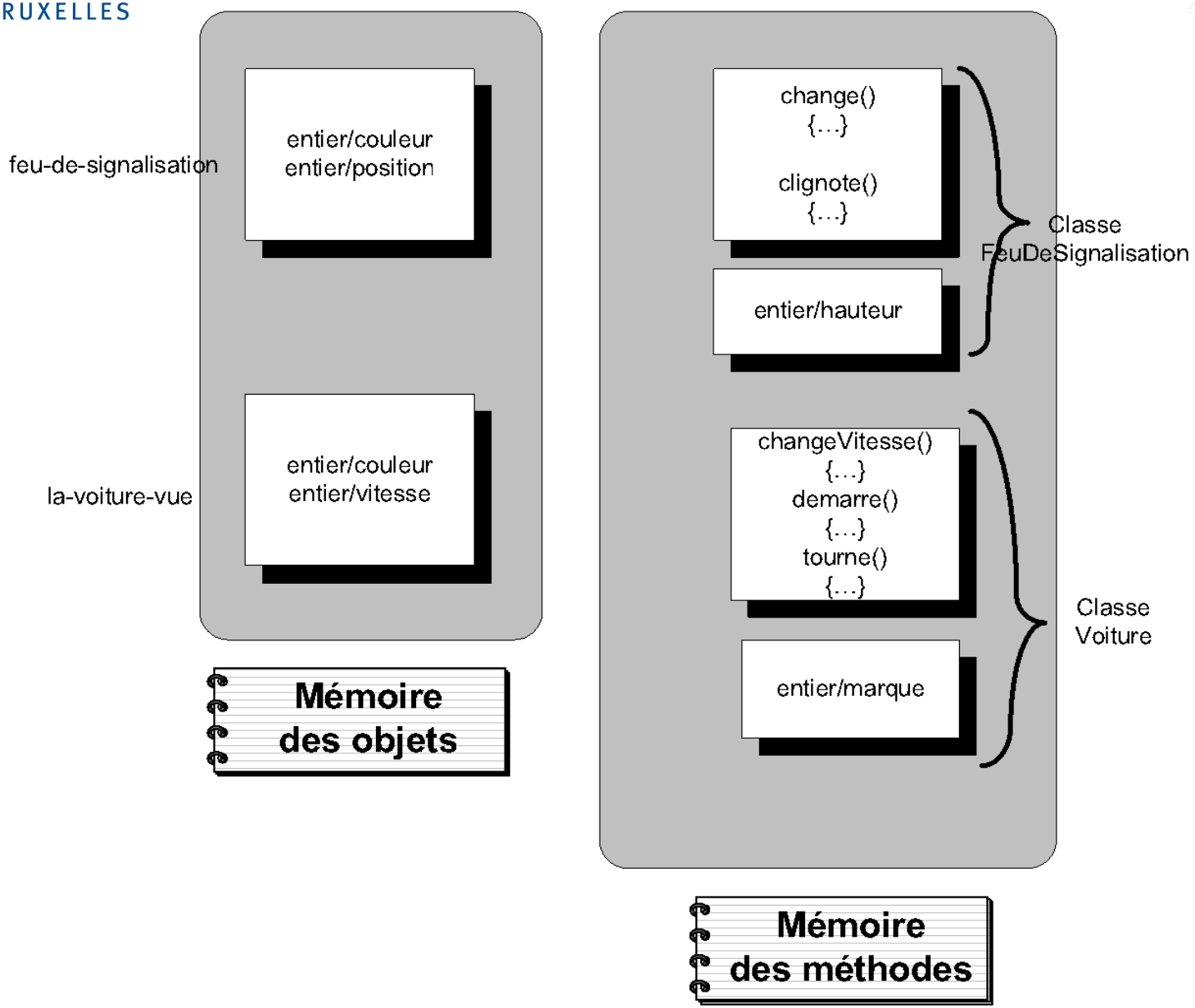


- La classe vérifie que l'on ne fait avec les objets que ce qu'elle autorise
- Les langages OO sont fortement typés.
- Classe et compilateur font bon ménage
- Des attributs et des méthodes peuvent être de classe: On dit alors qu'ils sont statiques

Attributs et méthodes statiques

- Un attribut ou une méthode statique **se réfère à une classe** et non à une instance particulière.

- Exemples :
  - `Math.PI` : pas besoin d'instancier un objet de type Math
  - `Math.max(int i, int j)`
  - `static void main(…)` : méthode dans laquelle on va créer les premiers objets.


feu-de-signalisation
entier/couleur
entier/position
la-voiture-vue
entier/couleur
entier/vitesse
Mémoire des objets
change()
{…}
clignote()
{…}
entier/hauteur
Classe FeuDeSignalisation
changeVitesse()
{…}
demarre()
{…}
tourne()
{…}
entier/marque
Classe Voiture
Mémoire des méthodes

Rappels

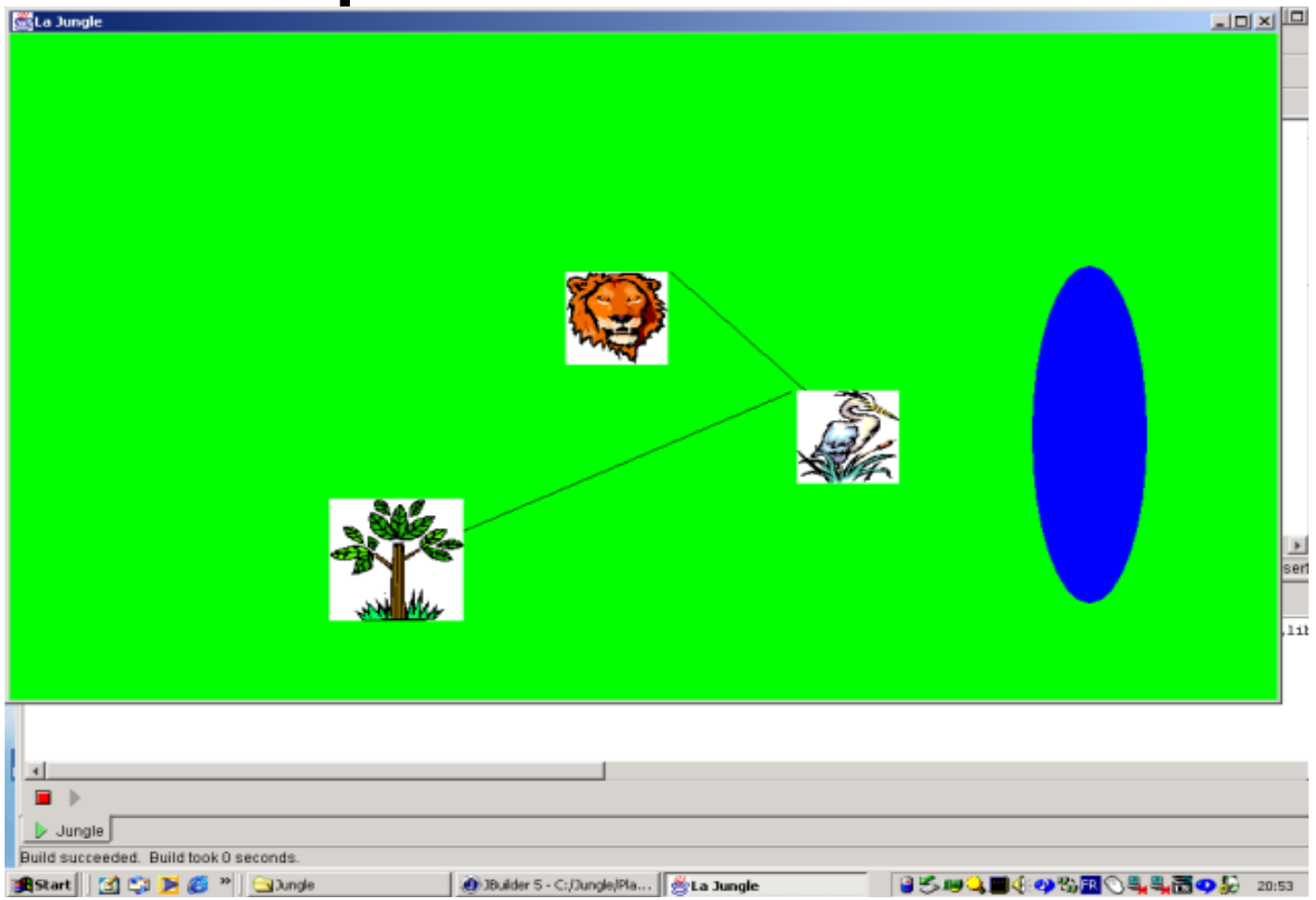
- Classe
- Objet
- Référent
- Attributs
- Méthodes
- Messages
- Signature et surcharge
- Constructeur et new

Classe = mécanisme de modularisation

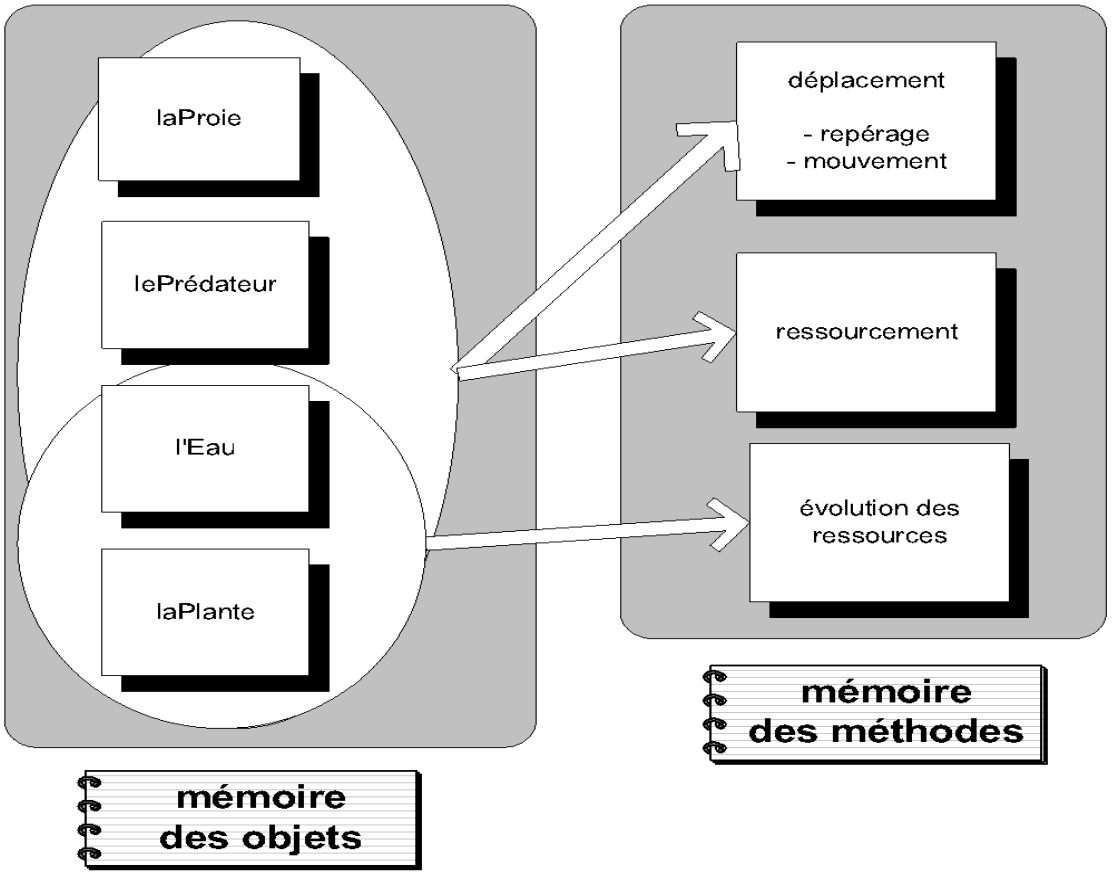
- Classes = fichiers
- Assemblages = répertoires
- La classe = le type et le module à la fois.

Comparaison procédural / OO

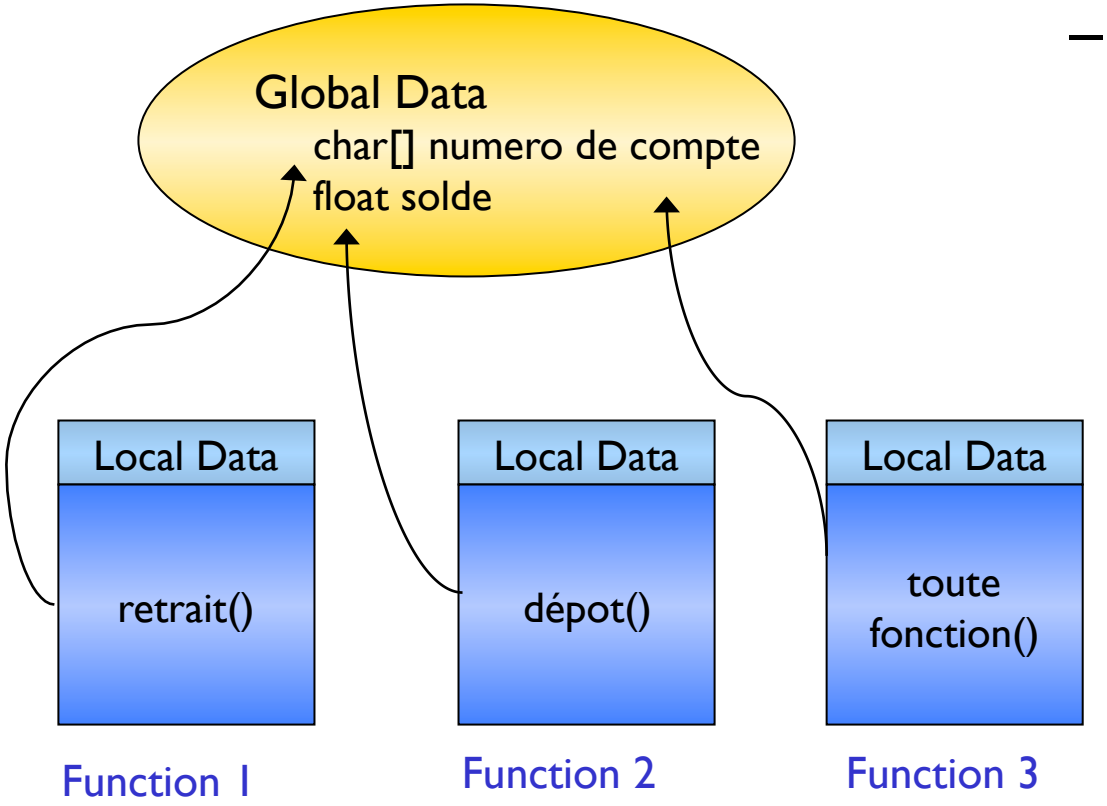
Du procédural à l'OO



Conception procédurale

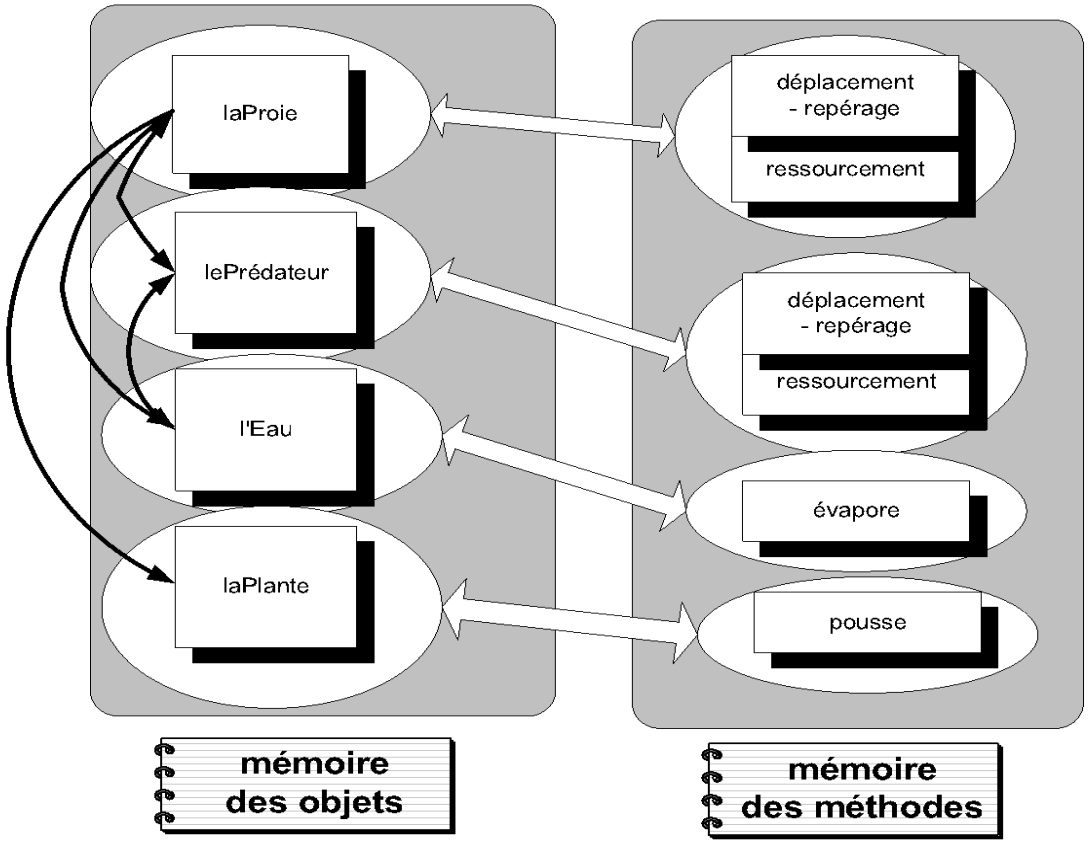


Problème de la programmation procédurale



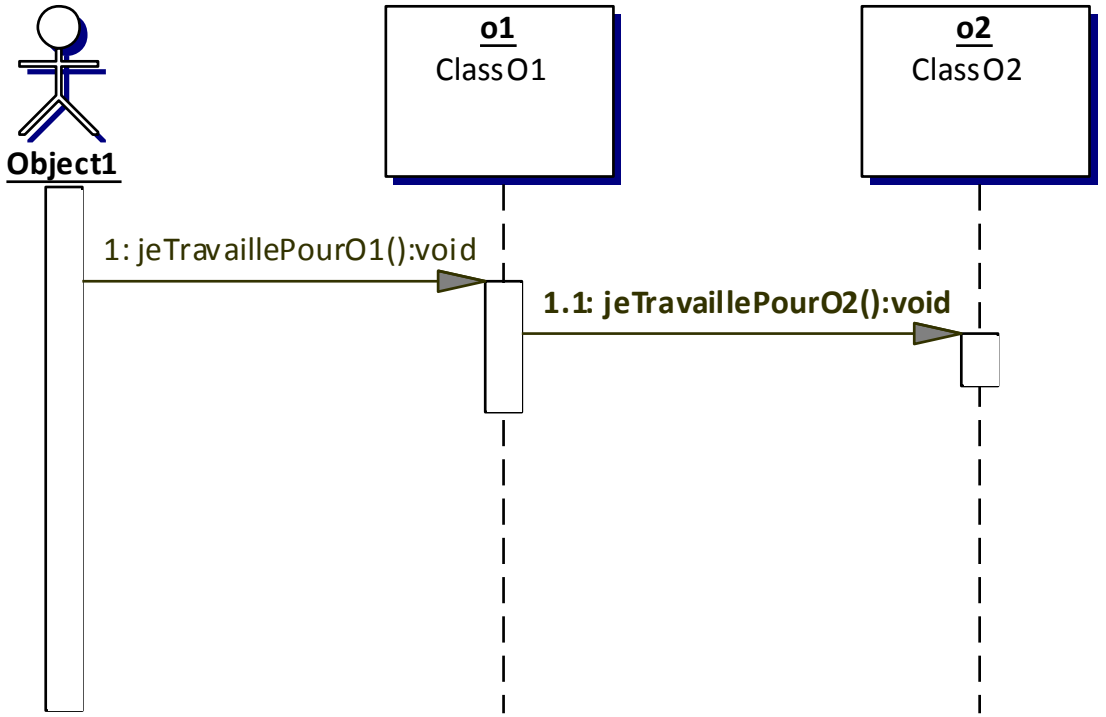
- Accès sans restriction
  - Plusieurs modules fonctionnels auront accès aux mêmes données créant une grande sensibilité aux changements

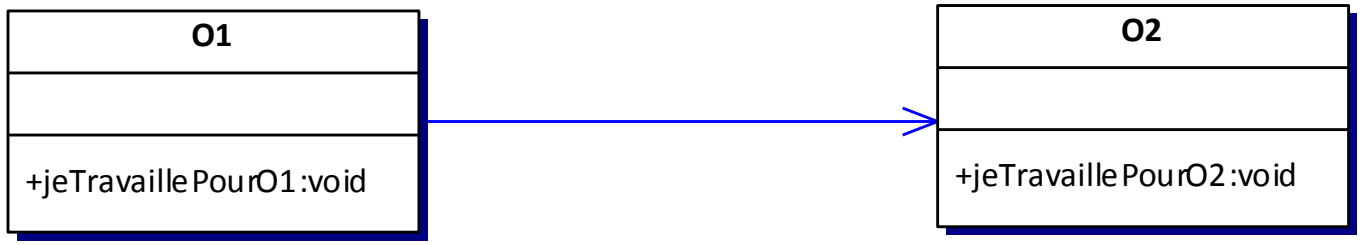
Conception OO



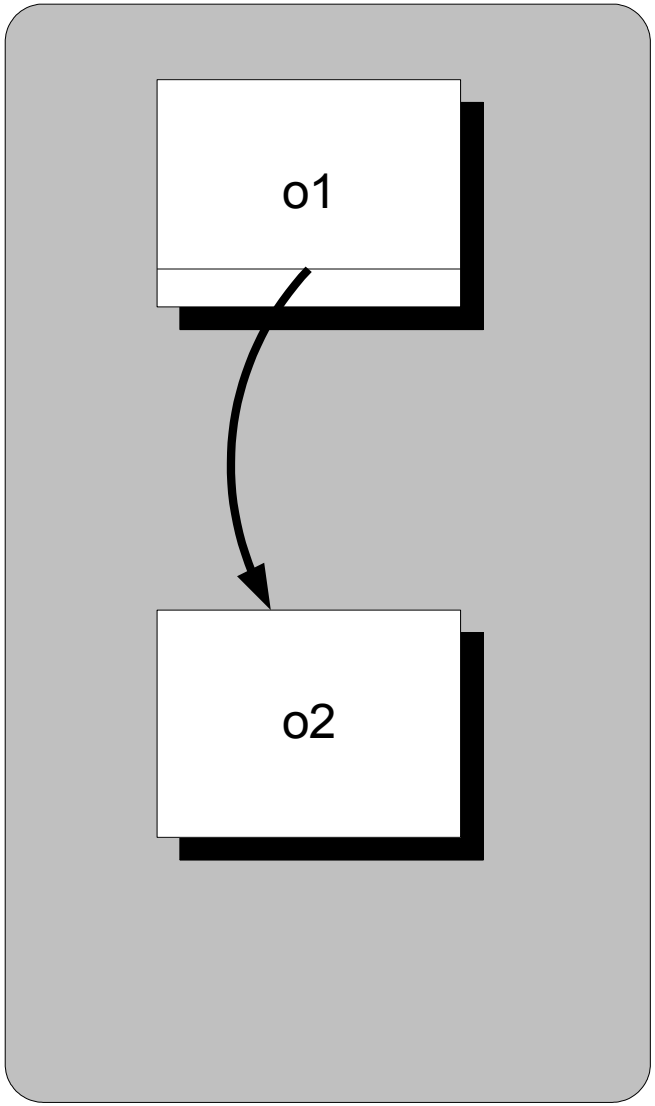
Envoi de messages et diagramme UML de séquence

Les objets parlent aux objets

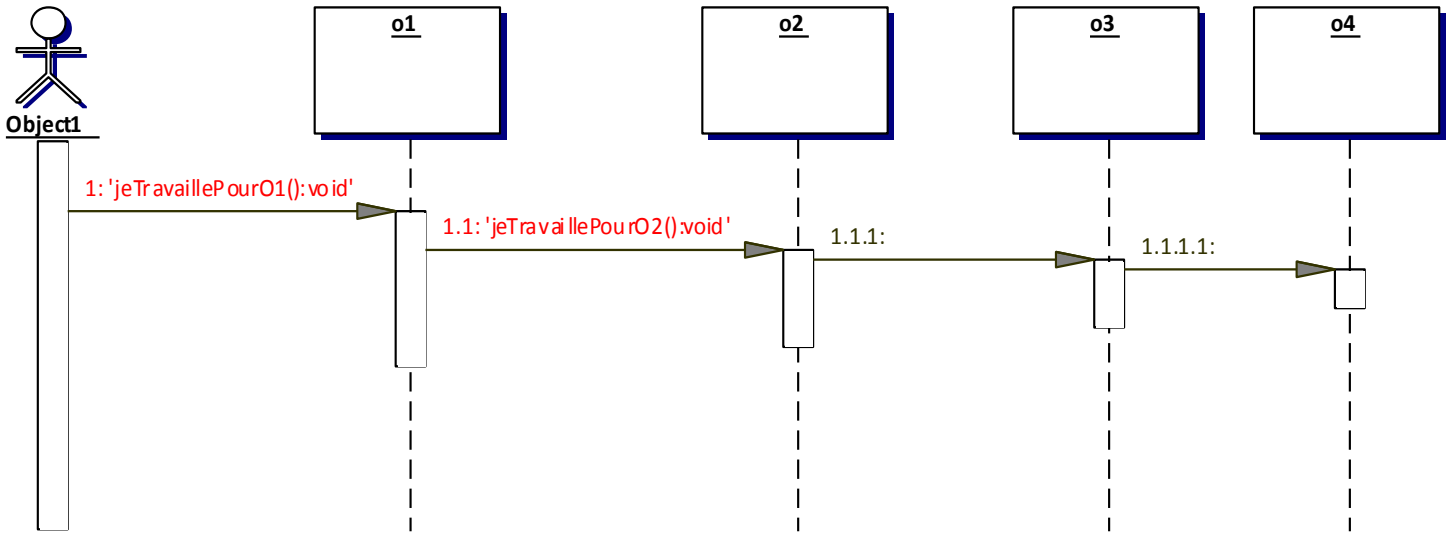





```
class O1 {
  O2   lienO2 ; /*la classe O2 type un
attribut de la classe O1 */
  void  jeTravaillePourO1() {
    lienO2.jeTravaillePourO2() ;
  }
}
```


o1
o2

Réaction en chaîne



Encapsulation

Encapsulation

- **L'encapsulation** est un **concept OO** qui consiste à **protéger** l'information contenue dans un objet.

- Les attributs et méthodes peuvent être :
  - `public` : accessibles de partout (intérieur, extérieur)
  - `private` : accessibles seulement à l'intérieur de la classe
  - D'autres niveaux de protection plus poussés existent.

- ***Une classe doit être le vigile de l'intégrité de ses objets.***
  Pour ce faire, on va généralement déclarer tous ses attributs en `private` et gérer leur intégrité via des **accesseurs**.
  Attention, il ne faut faire des accesseurs que si c'est nécessaire.

Encapsulation : Accesseurs

- Un **accesseur** est une méthode pour accéder, en lecture ou en écriture aux attributs d'une classe.
- Un accesseur en lecture renverra l'attribut.
- Un accesseur en écriture est généralement `void`, prend un paramètre et modifie l'attribut, après avoir vérifié que cette modification ne portait pas atteinte à l'intégrité de l'objet.
- Exemple :

```
public double getSalary(){
  return salary;
}

public void setSalary(double salary){
  if(salary >= 0)
    this.salary = salary;
}
```

Encapsulation

- Attributs et méthodes private ou public
- Private = accès restreint à la seule classe
- Public = accès permis à toutes les classes
- Attributs private: préserver l'intégrité des objets et cloisonner le traitement

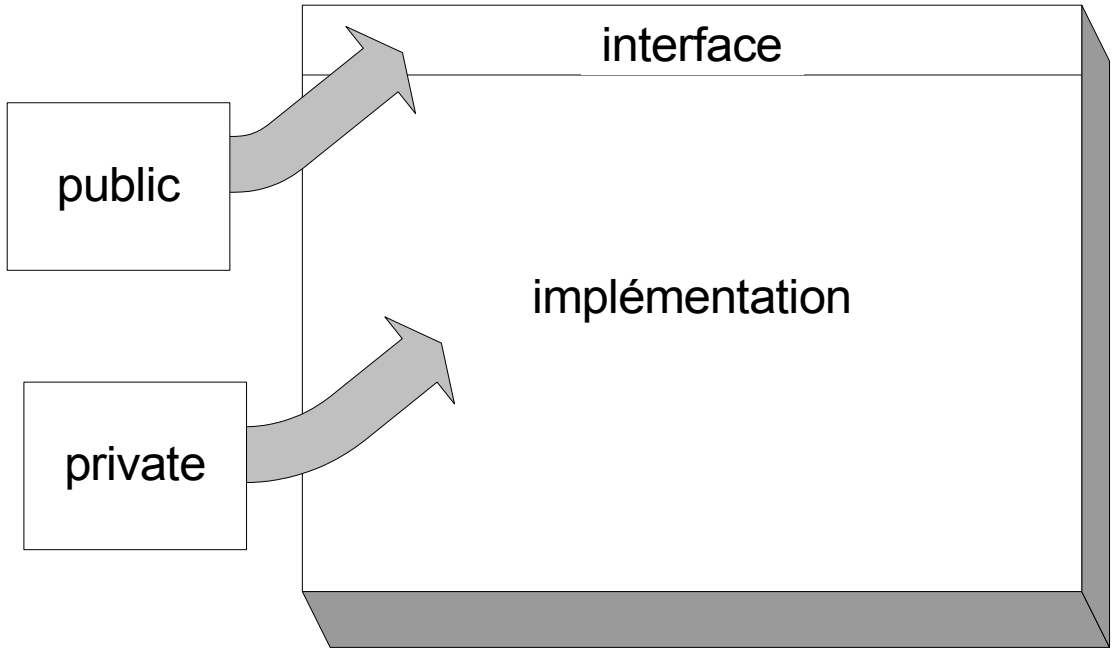
```
class Feu-de-signalisation {
  private int couleur;
  private Voiture voitureDevant;

  public void changeCouleur(int nouvelleCouleur)
  {
    if (nouvelleCouleur >= 1) && (nouvelleCouleur
  <=3) /* intégrité assurée */
      couleur = nouvelleCouleur ;
  }
}
```

```
class Voiture {

    private int vitesse ;

    public int changeVitesse(int
nouvelleVitesse) {

      if (nouvelleVitesse >= 0) &&
(nouvelleVitesse <=130) /* intégrité assurée
*/

        vitesse = nouvelleVitesse ;

      return vitesse ;

    }

}
```

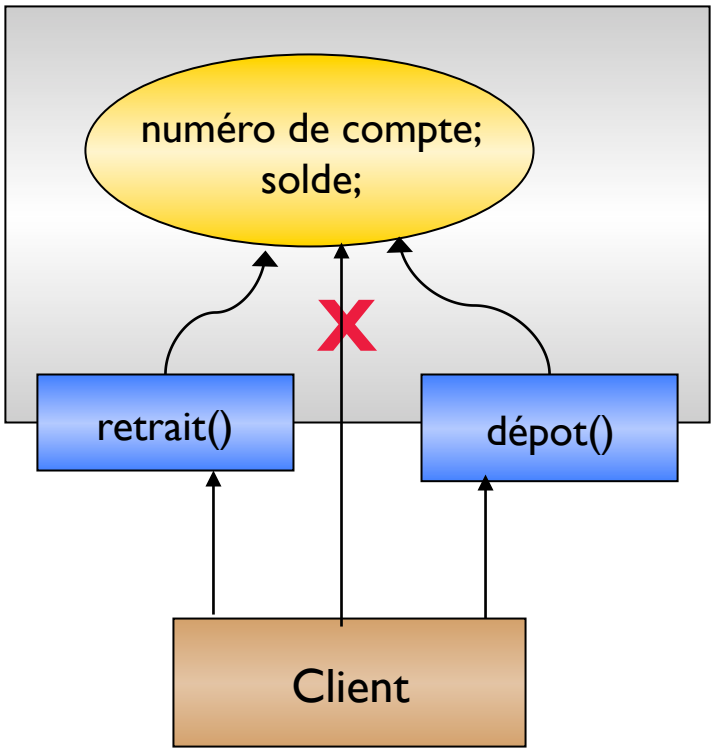
```
class Acheteur {
  private Voiture voitureInteressante ;

  public int négociePrix() {
    int prixPropose = 0 ;
     if (voitureInteressante.dateFabrication <
  19970101) /* accès possible à l'attribut date
  */
      prixPropose =
  voitureInteressante.getPrixDeBase() - 10000;
  }
}
```

Encapsulation des méthodes

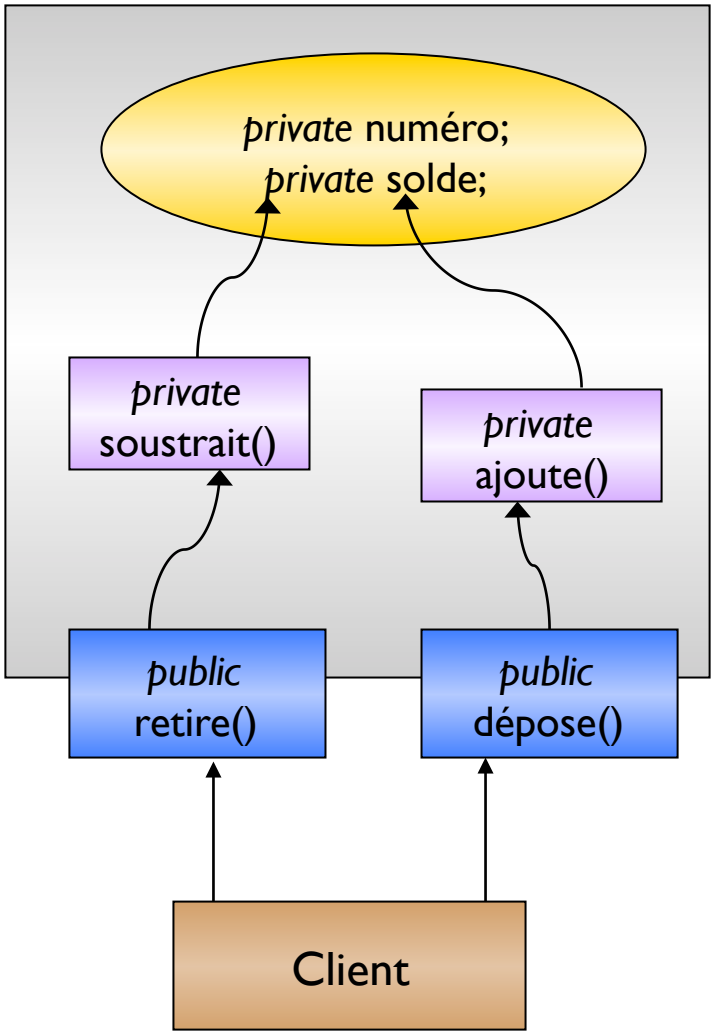
- Stabilisation des développements
- Dépendances à l'exécution mais indépendance au développement
- Interface et implémentation
- Signature d'une classe = son interface = liste des signatures des méthodes disponibles
- **Carte de visite de l'objet**


interface
public
implémentation
private

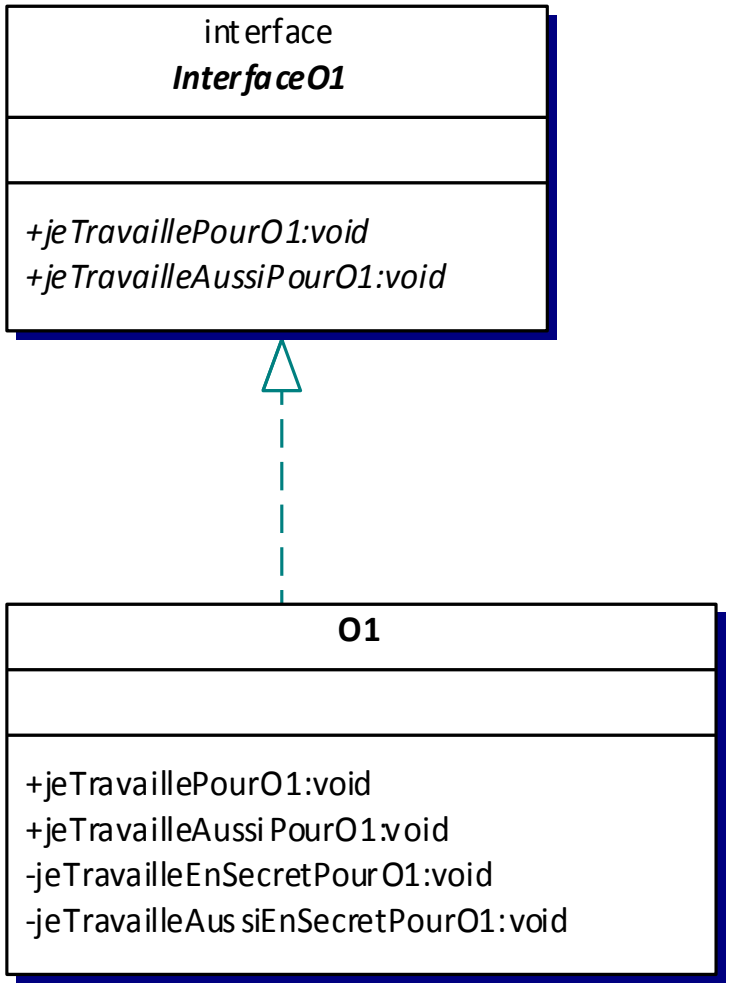
Il faut forcer l'encapsulation



- Les objets restreignent leur accès qu'aux méthodes de leur classe
  - Ils protègent leurs attributs
- Cela vous permet d'avoir un contrôle sur tous les accès.
- Cela permet de stabiliser les développements



- Garder tous les détails de l'implémentation privés, y compris certaines méthodes
- D'où le recours aux interfaces qui ne reprennent que la partie publique


interface
InterfaceO1
+jeTravaillePourO1:void
+jeTravailleAussiPourO1:void
O1
+jeTravaillePourO1:void
+jeTravailleAussiPourO1:void
-jeTravailleEnSecretPourO1:void
-jeTravailleAussiEnSecretPourO1:void

L'héritage

Héritage

- **L'héritage** est un **concept OO** qui consiste à spécialiser des classes en réutilisant les attributs et le comportement d'une autre classe.

- Vocabulaire : Une classe (*sous-classe, classe fille, classe dérivée*) peut être dérivée d'une autre classe (*super-classe, classe mère*).

- L'héritage est un des concept principal de l'OO : **réutiliser du code**.

Héritage

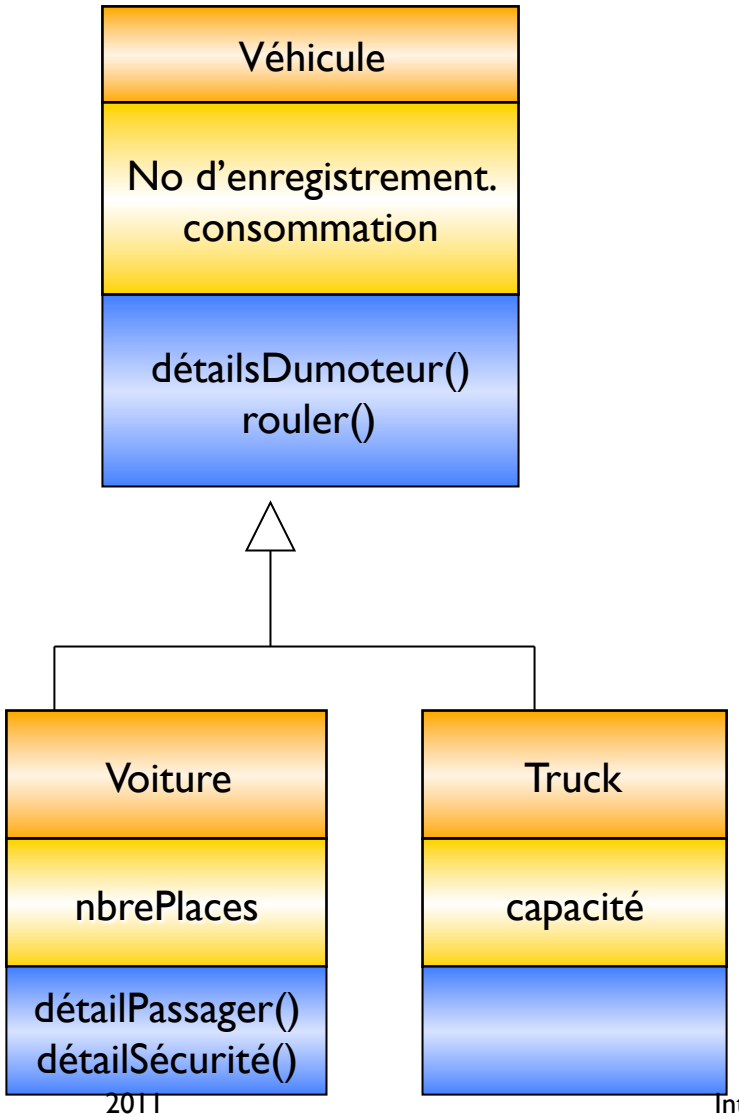
- La classe B **hérite** de la classe A. C'est-à-dire que *B est une spécialisation de la classe A*.
- La classe B hérite des méthodes et propriétés de la classe A.
- On peut y ajouter des propriétés et méthodes et redéfinir des méthodes.

- Ex : `Voiture : Véhicule` `Moto : Véhicule`
  - Méthode `nombreDeRoues()` dans `Véhicule`.

L'héritage

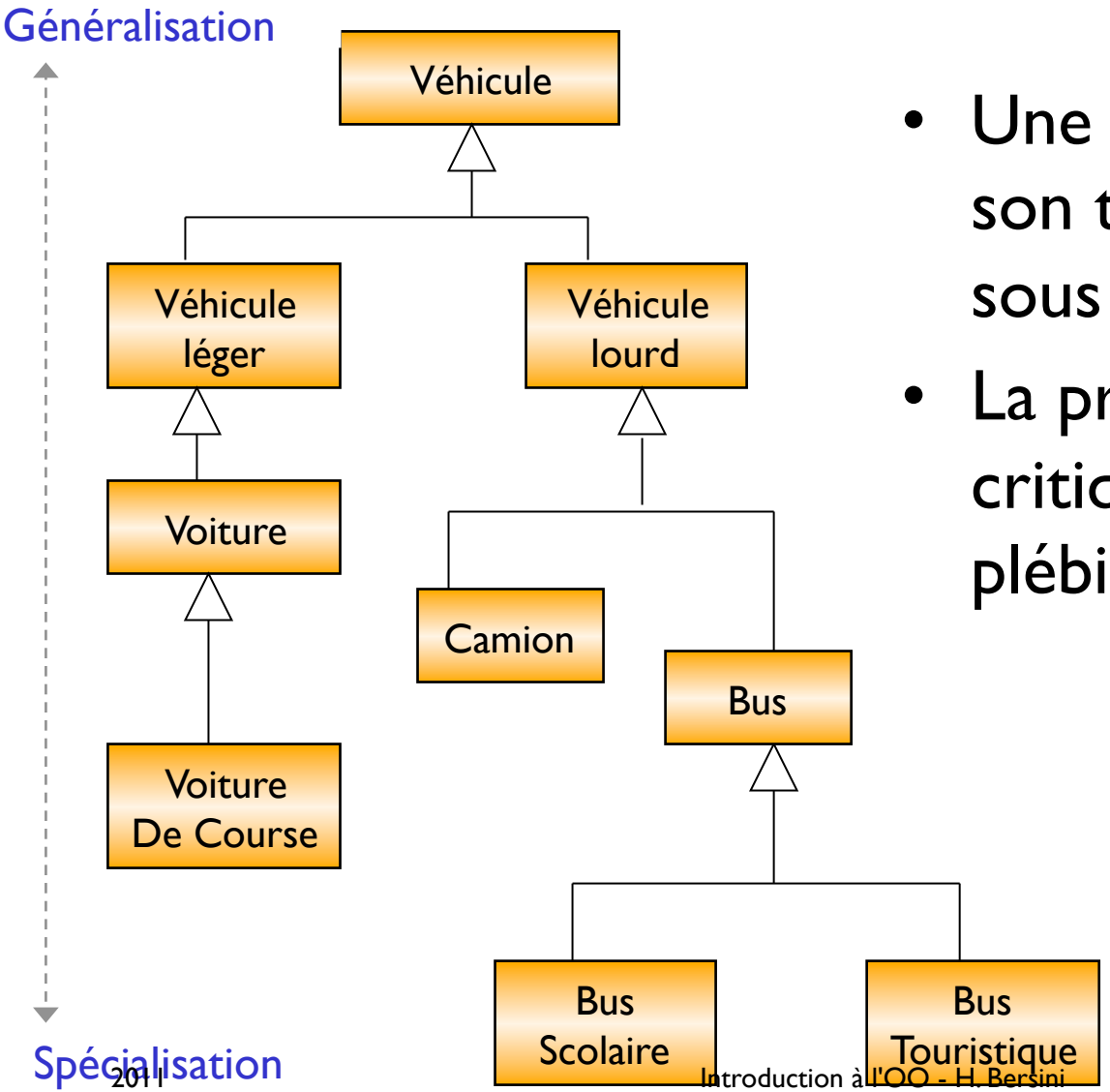
- Regrouper les classes en super-classes en factorisant et spécialisant
- Dans l'écosystème: La classe Faune regroupe les animaux.
- La classe Ressource regroupe l'eau et la plante
- La sous-classe hérite des attributs et méthodes et peut en rajouter de nouveau

Super classes et Sous classes



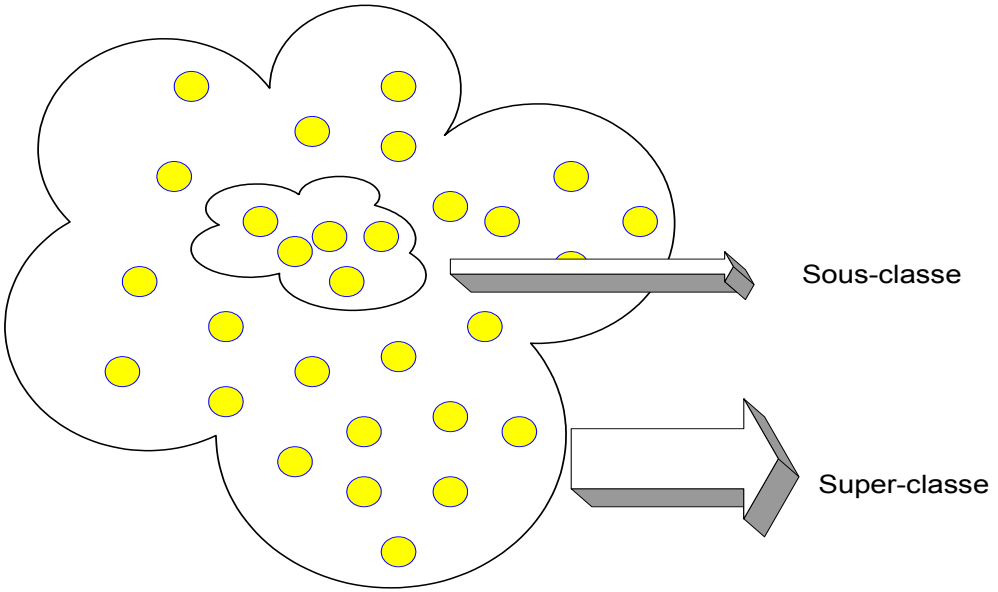
- Super classe
  - reprend les attributs et les méthodes communes aux sous classes
- Sous classe
  - héritent de tout ce qui constitue la super classe
  - peut ajouter de nouveaux attributs
  - peut ajouter ou redéfinir des méthodes

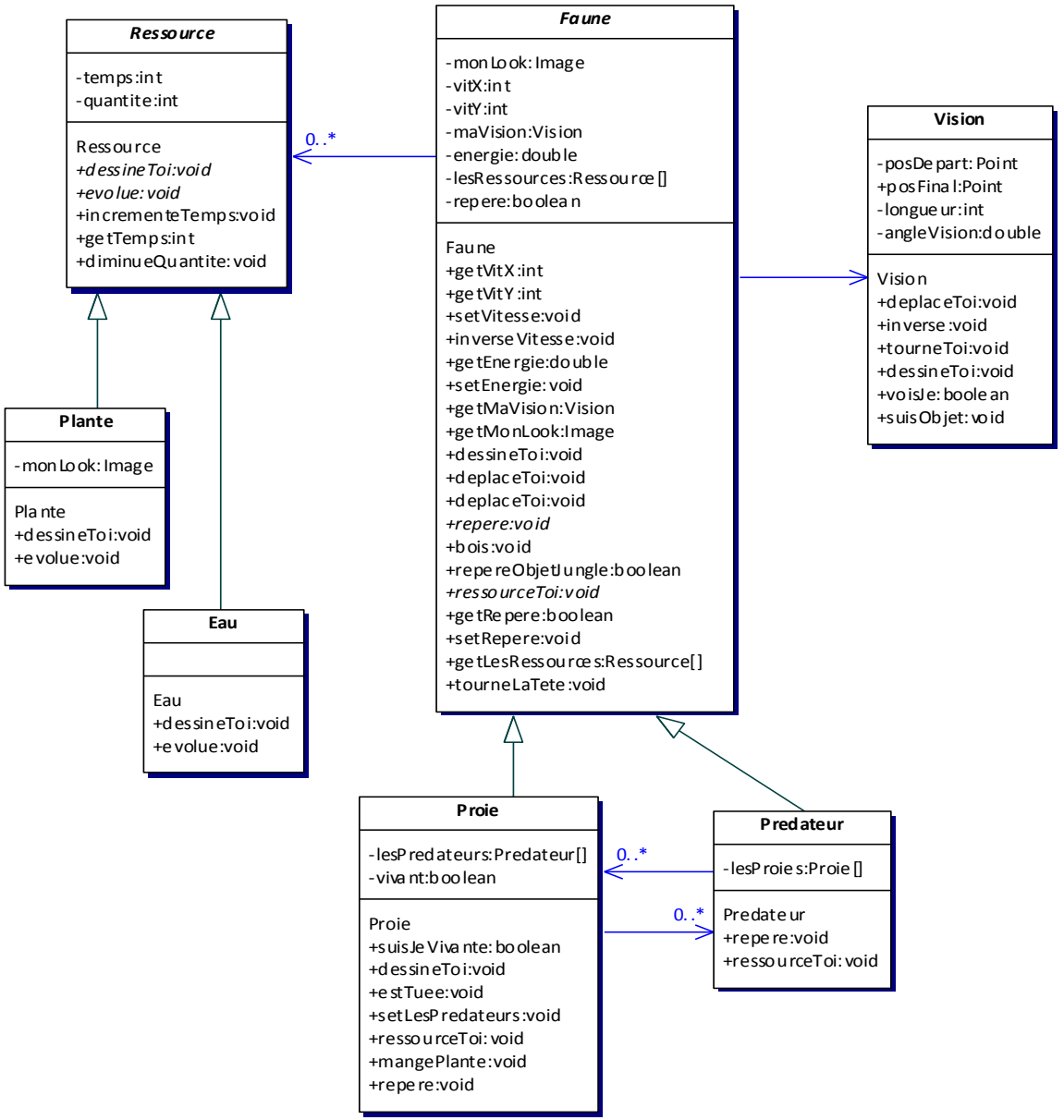
Héritage à plusieurs étages



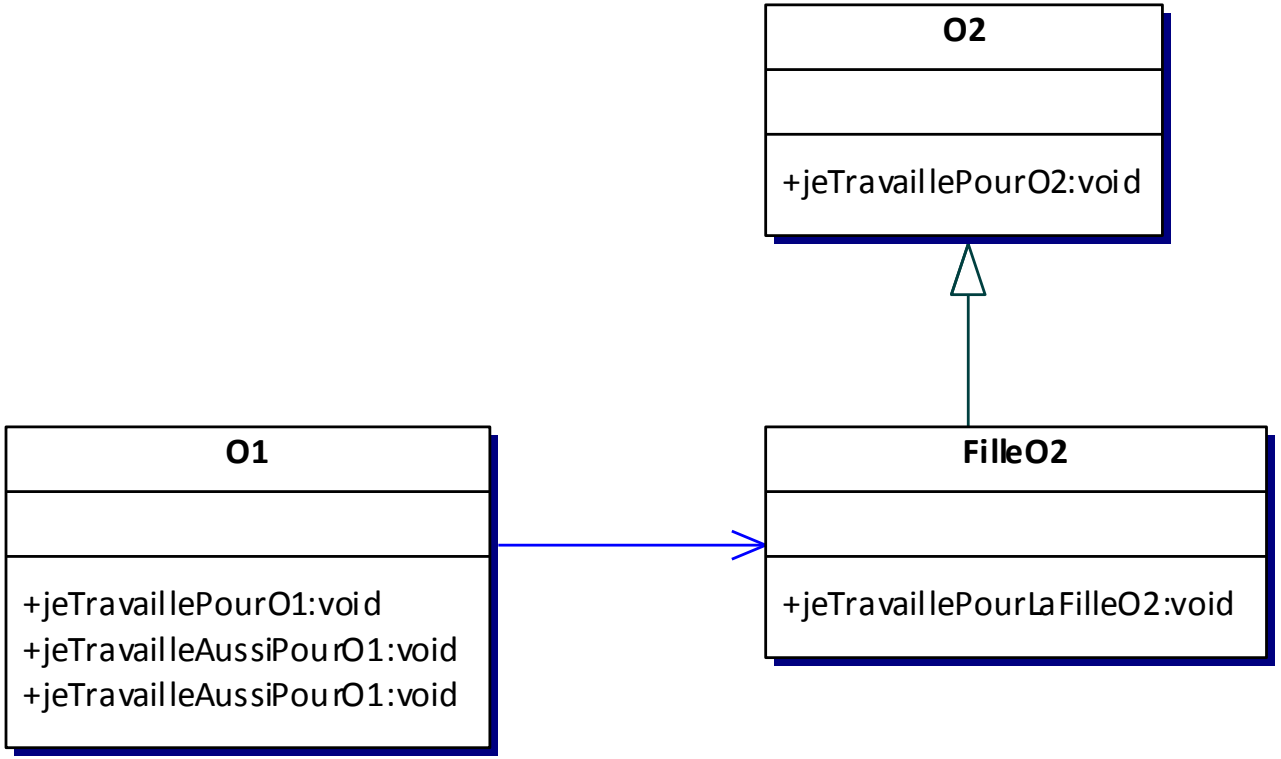
- Une super classe peut à son tour devenir une sous classe …
- La profondeur est parfois critiquée parfois plébisicitée…

- L'addition des propriétés permet l'application du principe de subsitution et de l'inteprétation ensembliste:

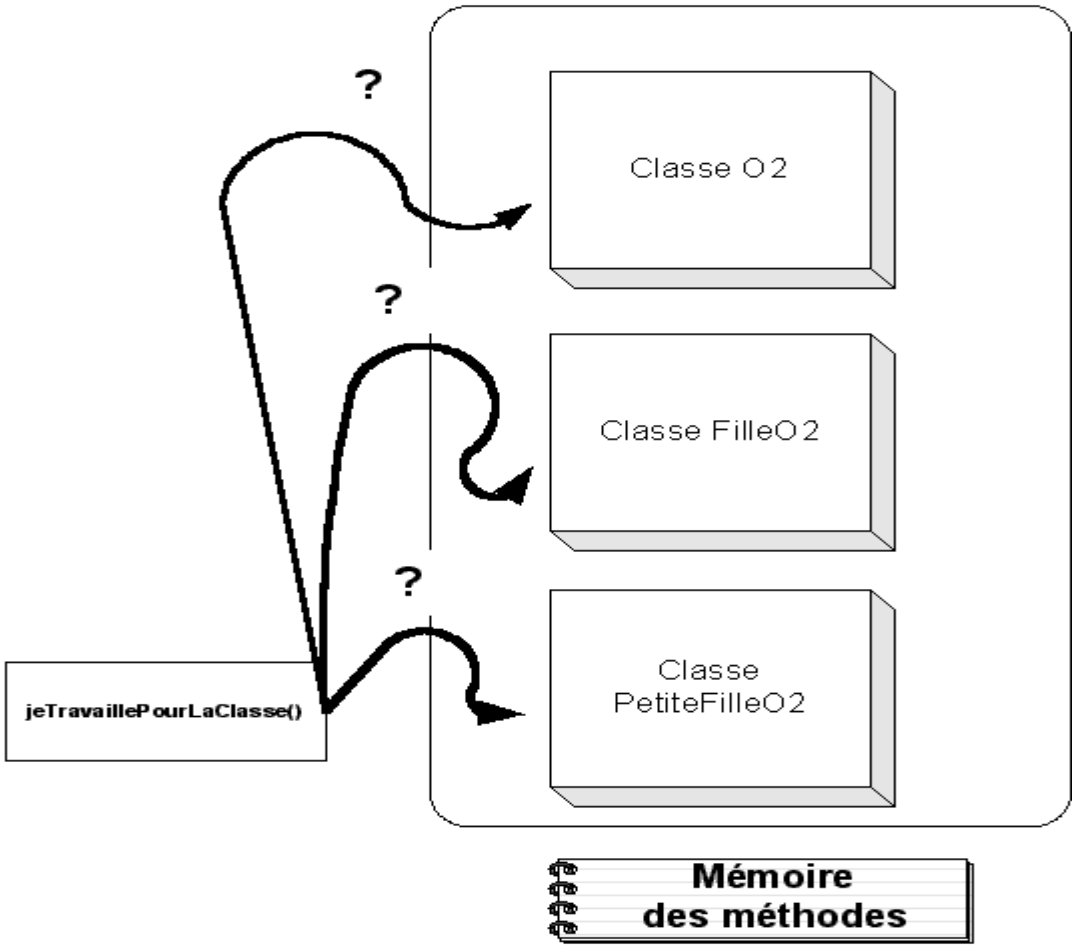



Ressource
-temps:int
-quantite:int
Ressource
+dessineToi:void
+evolue:void
+incrementeTemps:void
+getTemps:int
+diminueQuantite:void
0..*
Faune
-monLook:Image
-vitX:int
-vitY:int
-maVision:Vision
-energie:double
-lesRessources:Ressource[]
-repere:boolean
Faune
+getVitX:int
+getVitY:int
+setVitesse:void
+inverseVitesse:void
+getEnergie:double
+setEnergie:void
+getMaVision:Vision
+getMonLook:Image
+dessineToi:void
+deplaceToi:void
+deplaceToi:void
+repere:void
+bois:void
+repereObjetJungle:boolean
+ressourceToi:void
+getRepere:boolean
+setRepere:void
+getLesRessources:Ressource[]
+tourneLaTete:void
Vision
-posDepart:Point
+posFinal:Point
-longueur:int
-angleVision:double
Vision
+deplaceToi:void
+inverse:void
+tourneToi:void
+dessineToi:void
+voisJe:boolean
+suisObjet:void
Plante
-monLook:Image
Plante
+dessineToi:void
+evolue:void
Eau
Eau
+dessineToi:void
+evolue:void
Proie
-lesPredateurs:Predateur[]
-vivant:boolean
Proie
+suisJeVivante:boolean
+dessineToi:void
+estTuee:void
+setLesPredateurs:void
+ressourceToi:void
+mangePlante:void
+repere:void
0..*
0..*
Predateur
-lesProies:Proie[]
Predateur
+repere:void
+ressourceToi:void

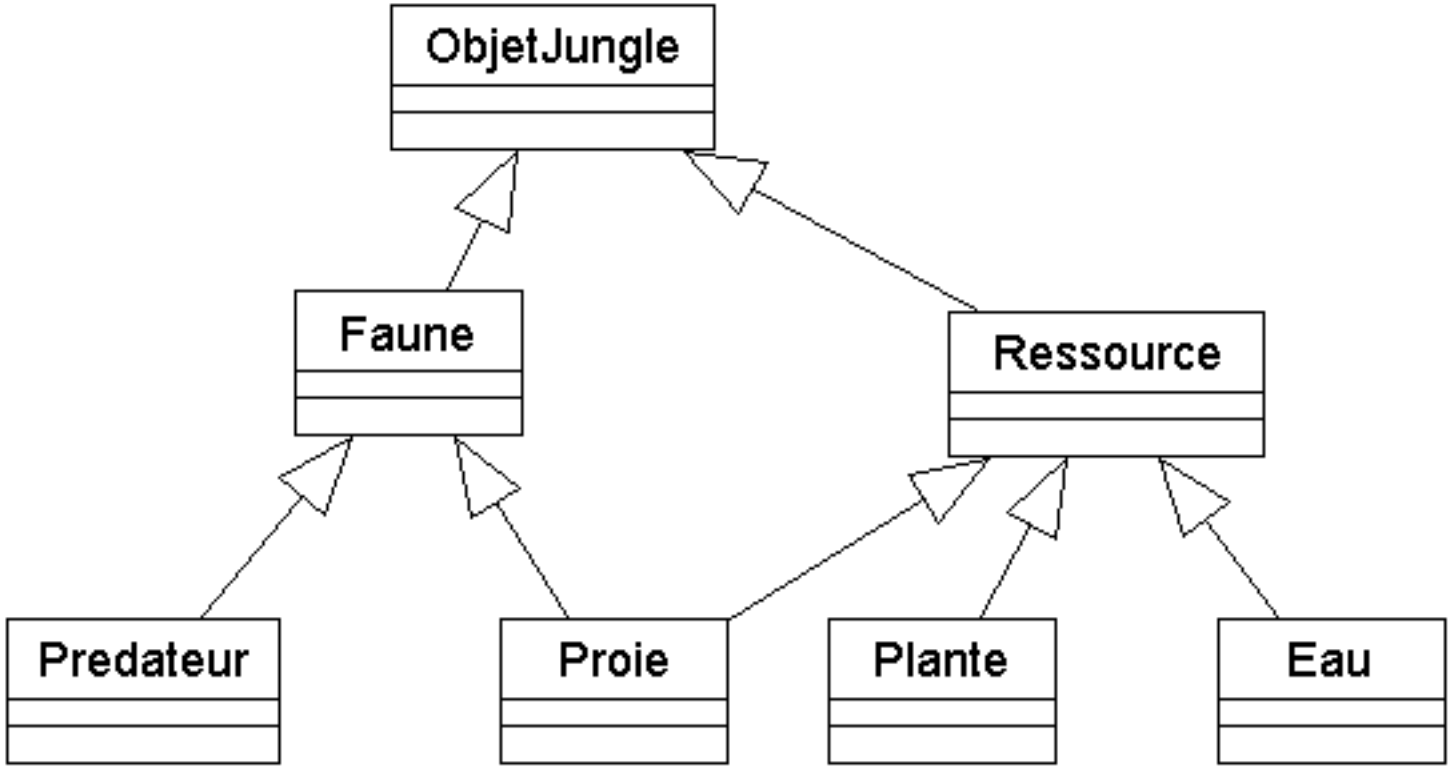
L'héritage des méthodes:



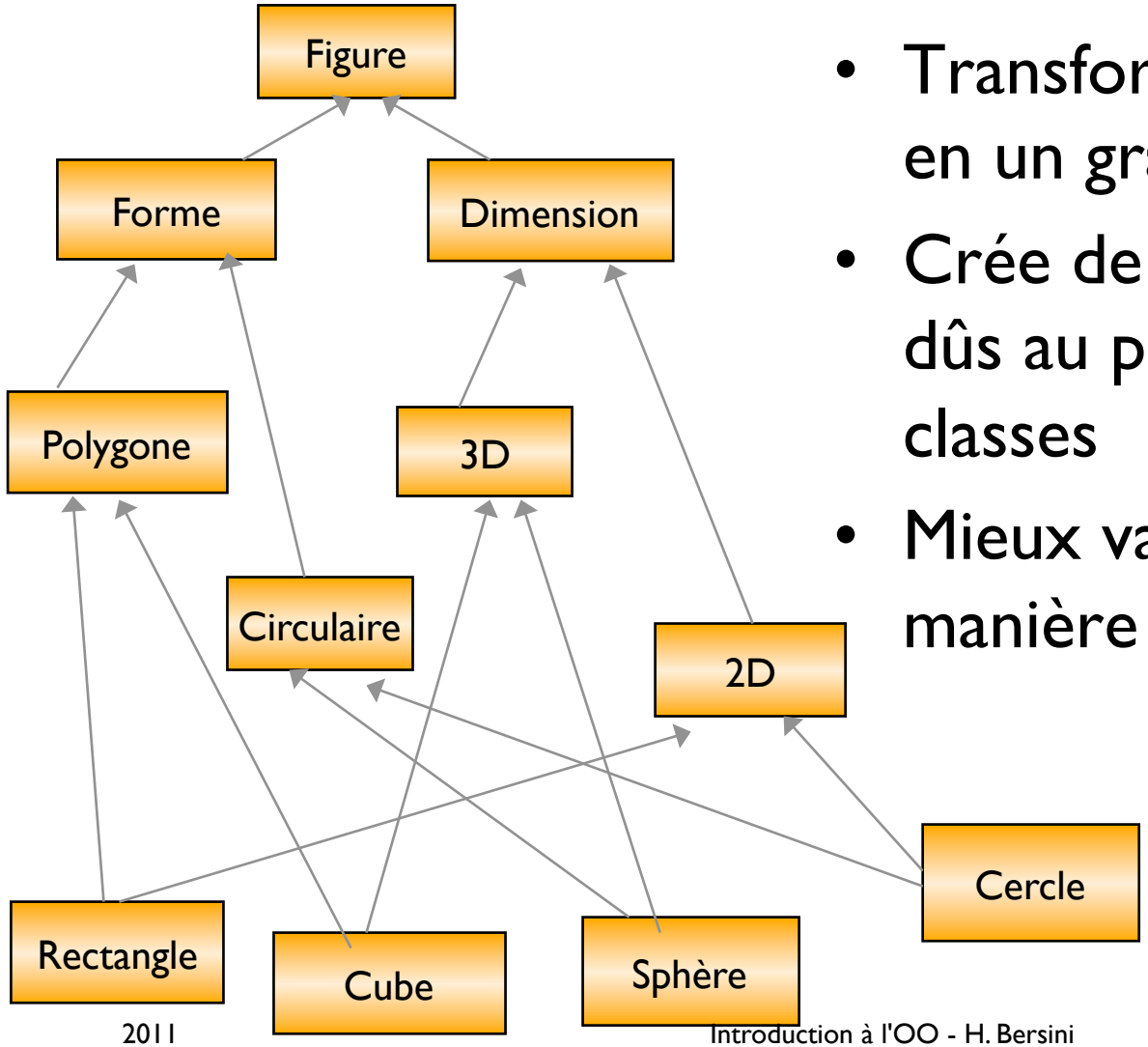
La recherche des méthodes



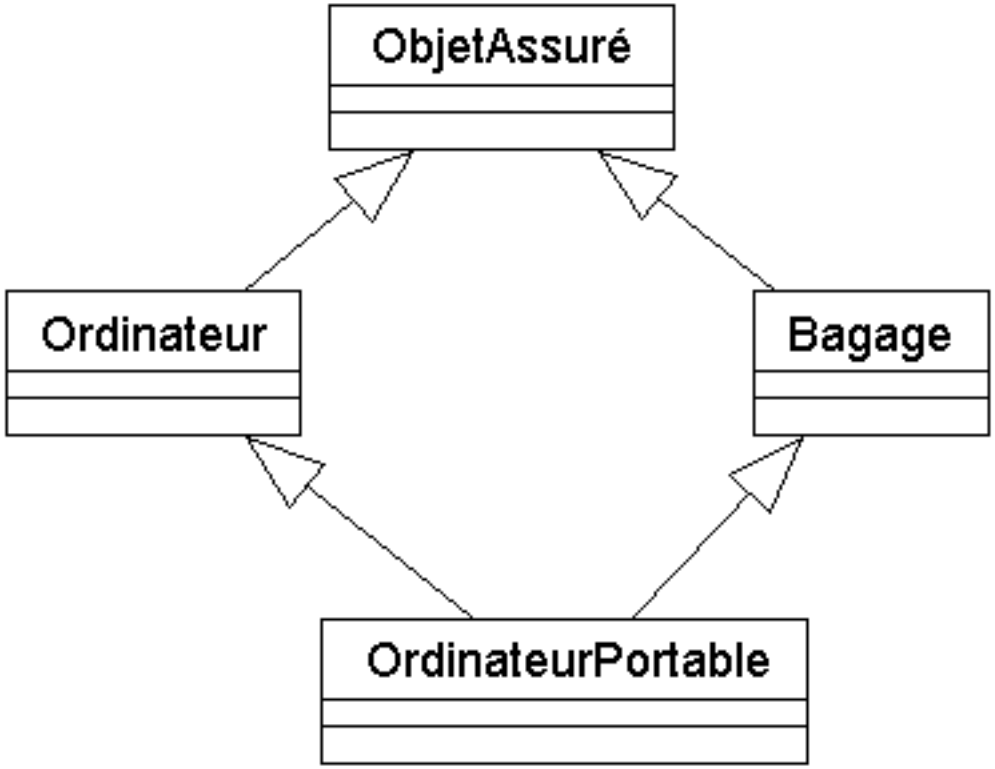
La problématique du multihéritage



Héritage multiple



- Transforme l'arbre d'héritage en un graphe
- Crée de nouveaux problèmes dûs au partage des super classes
- Mieux vaut l'éviter de toute manière

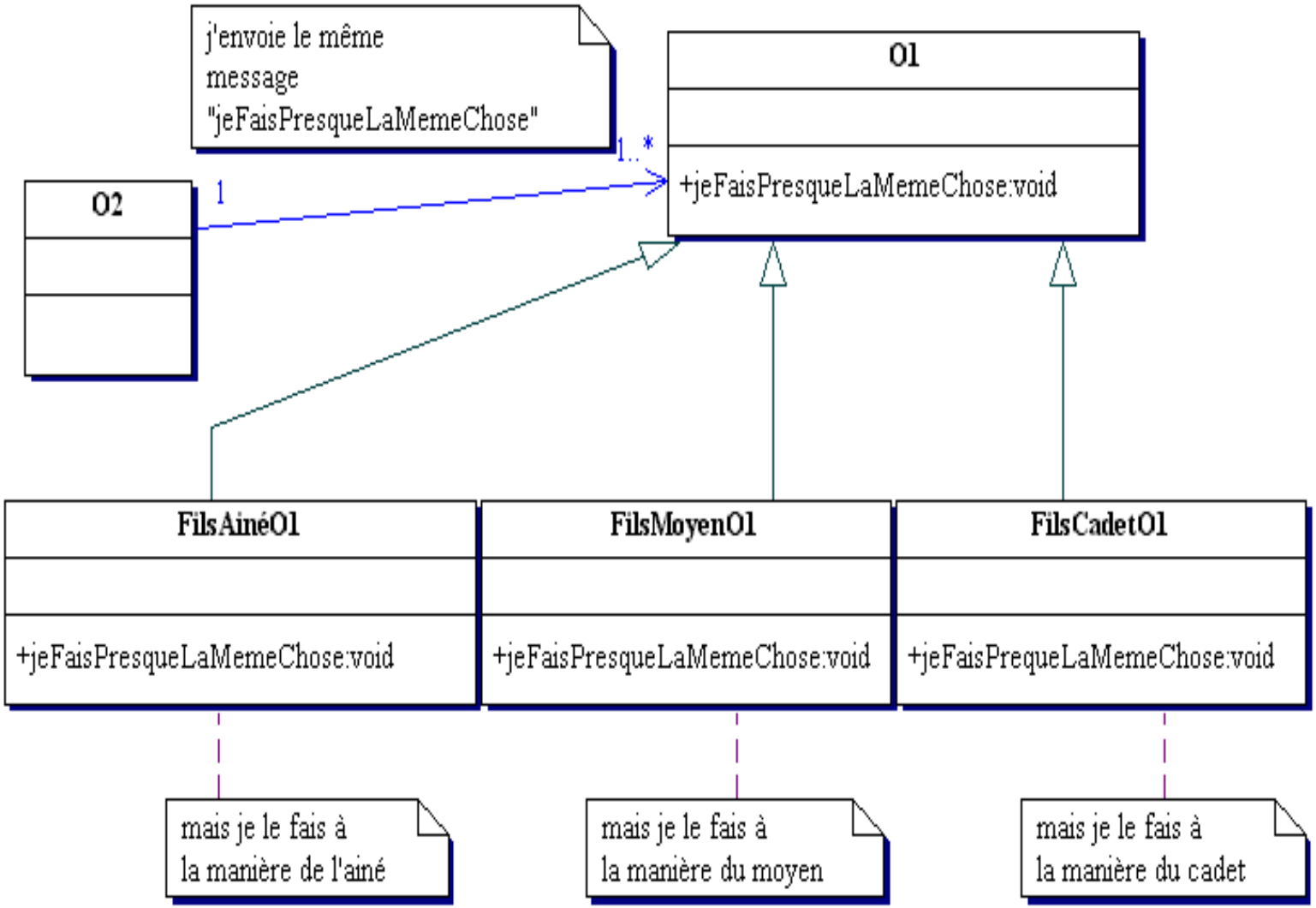

ObjetAssuré
Ordinateur
Bagage
OrdinateurPortable

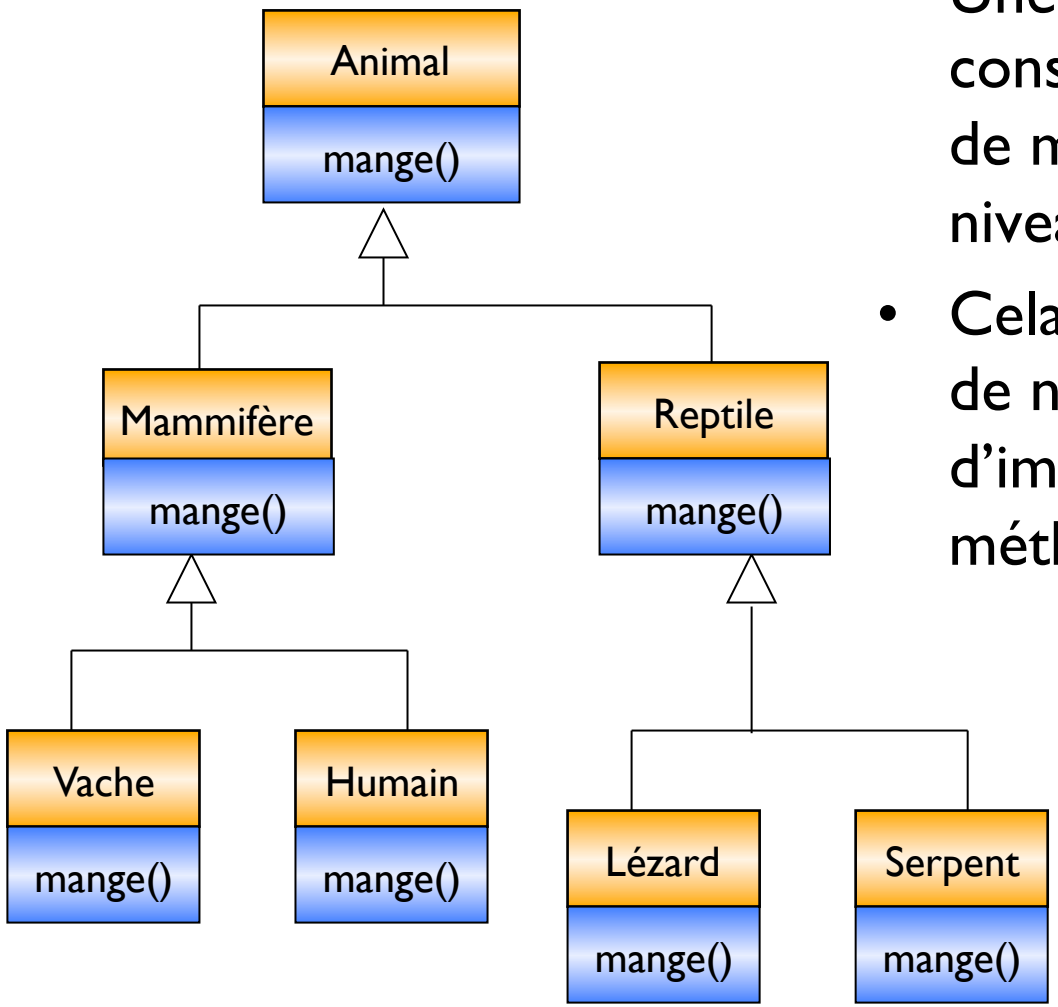
Polymorphisme

- Le **polymorphisme** est un **concept OO** dont l'idée est d'autoriser le même code à être utilisé avec différents types, ce qui permet des implémentations plus abstraites et générales.

Polymorphisme

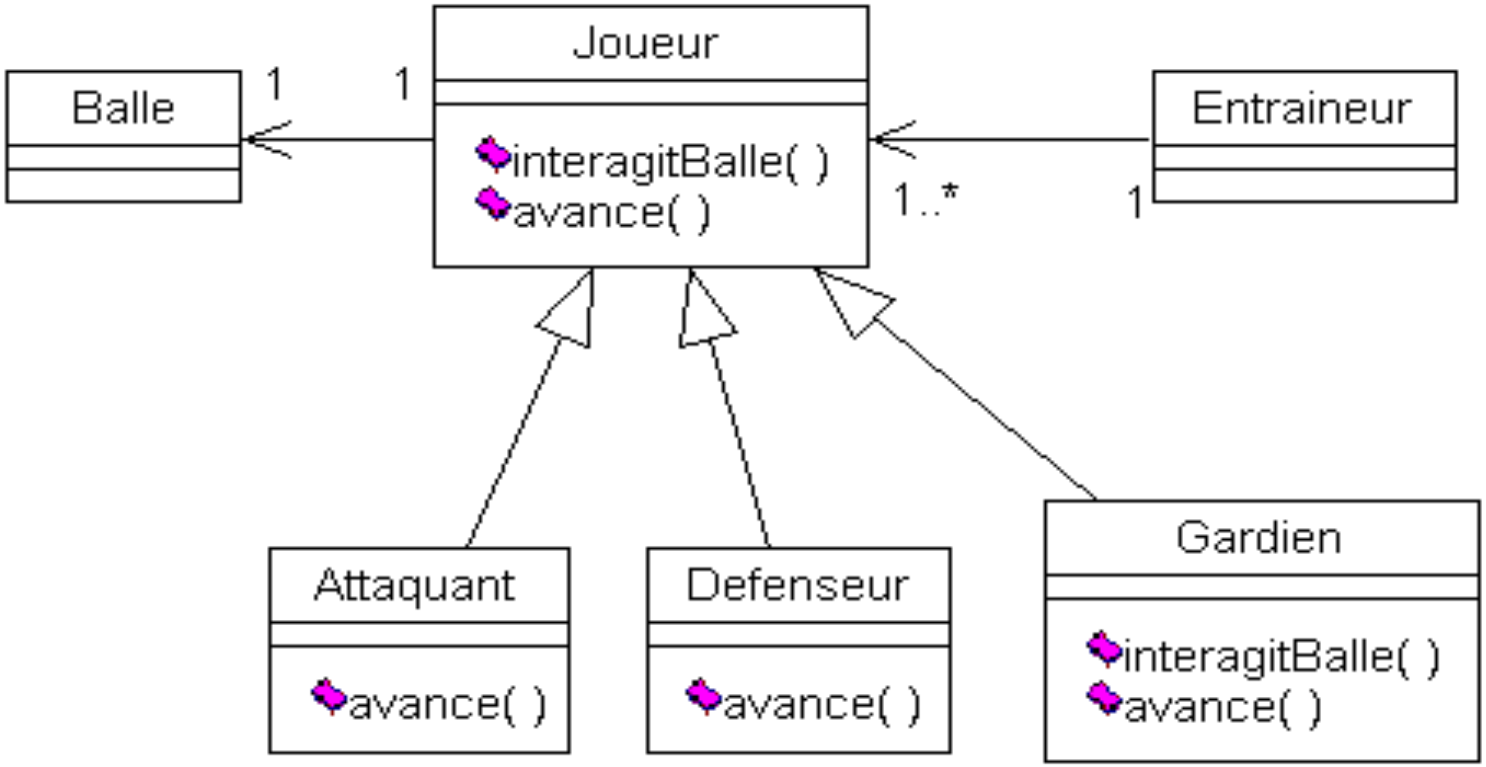
- Basé sur la redéfinition des méthodes
- Permet à une tierce classe de traiter un ensemble de classes sans connaître leur nature ultime.
- Permet de factoriser des dénominations d'activité mais pas les activités elles-mêmes.

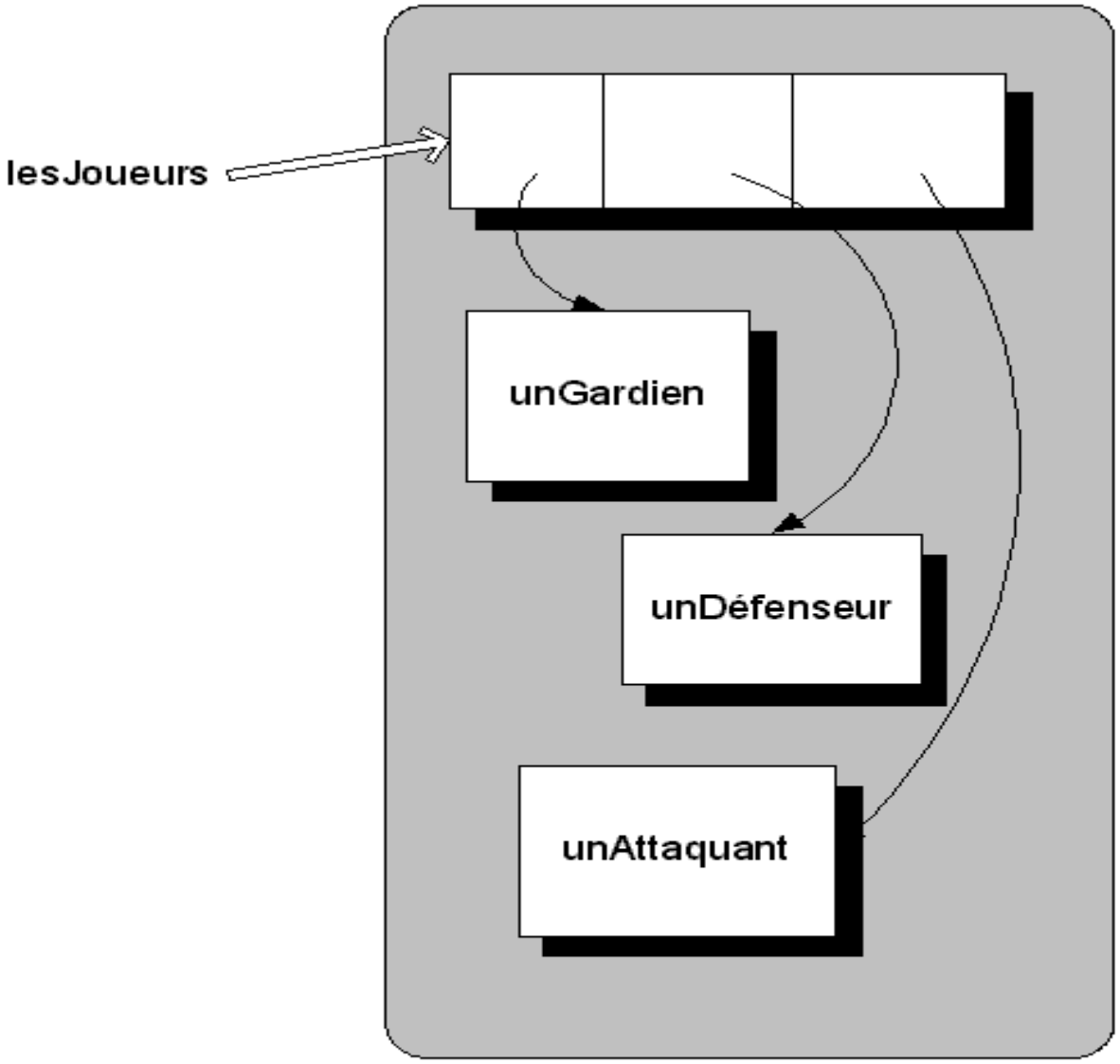

j'envoie le même
message
"jeFaisPresqueLaMemeChose"
1..*
O1
+jeFaisPresqueLaMemeChose:void
O2
1
FilsAinéO1
FilsMoyenO1
FilsCadetO1
+jeFaisPresqueLaMemeChose:void
+jeFaisPresqueLaMemeChose:void
+jeFaisPrequeLaMemeChose:void
mais je le fais à
la manière de l'ainé
mais je le fais à
la manière du moyen
mais je le fais à
la manière du cadet



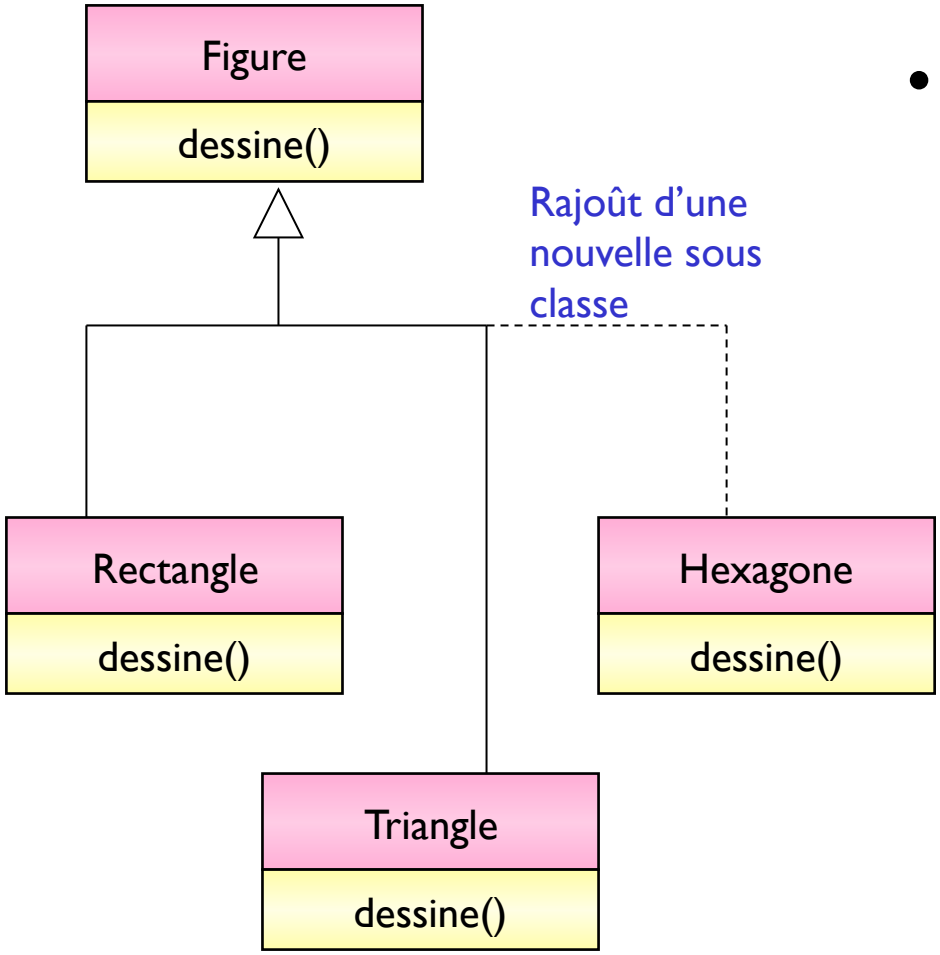
- Une méthode d'appellation constante peut être redéfinie de multiples fois à différents niveaux de la hiérarchie
- Cela permet à une tierce classe de ne pas à connaître les détails d'implémentation de la méthode

Un match de foot polymorphique:




lesJoueurs
unGardien
unDéfenseur
unAttaquant
Mémoire des objets

Polymorphisme: Facilite l'extension



- Les programmes peuvent aisément être étendus
  - De nouvelles sous classes peuvent s'ajouter sans rien modifier à l'existant

Construction d'objet

```
public class A {
  public A(){
    …
  }
}
public class B extends A {
 public B(){
    …
  }
}
public class C extends B {
 public C(){
    …
  }
}
```

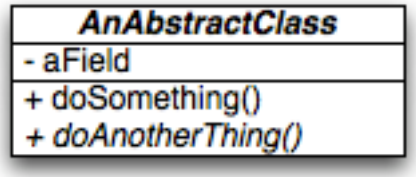
Lors de la construction d'un objet de la classe C :

- le constructeur de C commence par appeler le construteur de B
- le constructeur de B commence par appeler le constructeur de A
- le constructeur de A appelle le constructeur d'Object
- le constructeur d'Object s'exécute
- le constructeur de A se termine
- le constructeur de B se termine
- le constructeur de C se termine

Classes abstraites et interfaces

Classes abstraites

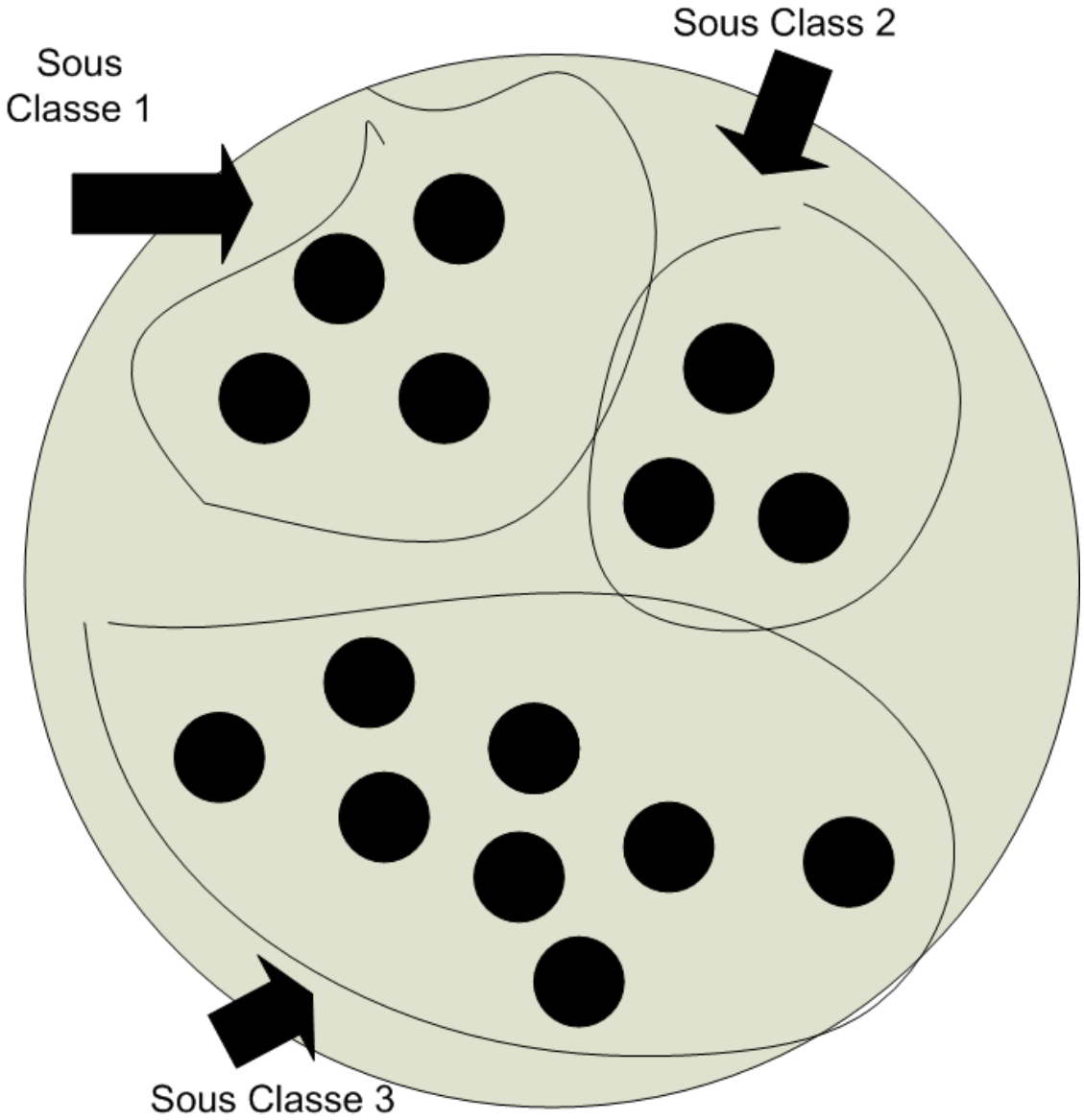
- Une **classe abstraite** ne peut être instanciée mais peut être *sous-classée*. Elle peut contenir ou non des méthodes abstraites.
- Uné **méthode abstraite** est une méthode déclarée mais non implémentée.
- Si une classe inclut des méthodes abstraites elle doit être abstraite.
- Une classe qui hérite d'une classe abstraite doit implémenter toutes les méthodes abstraites de la classe parente. Sinon elle doit être déclarée abstraite.




```
public abstract class AnAbstractClass{
  private int aFied;
  ...
  public void doSomething(){ ... }
  public abstract void doAnotherThing();
}
```

Classe abstraite

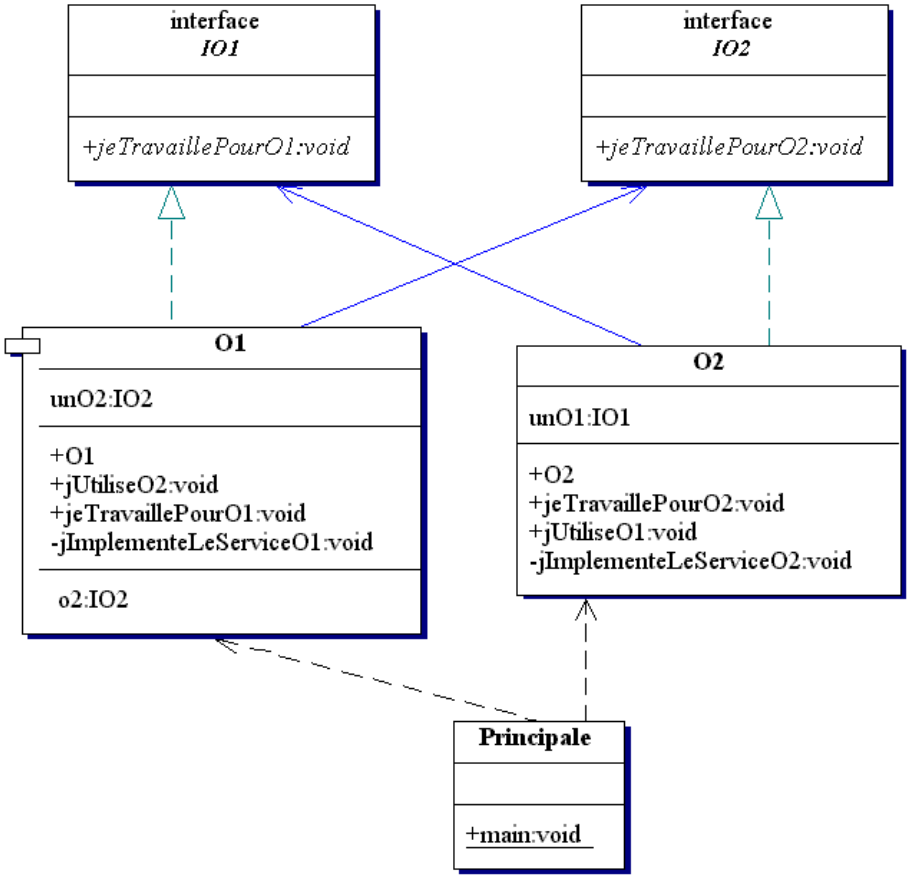
- Classe avec une ou des méthodes abstraites
- Classe sans objet
- Toutes les super classes devraient l'être
- Une interface est une classe abstraite dont toutes les méthodes le sont


Sous Class 2
Sous
Classe 1
Sous Classe 3

Les trois raisons d'être des interfaces

- Forcer la redéfinition
- Permettre le multihéritage
- Jouer la carte de visite de l'objet

Les interfaces



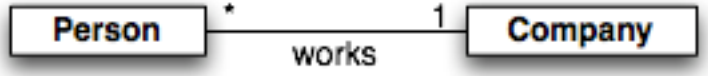
Associations entre objets

Association, composition, agrégation

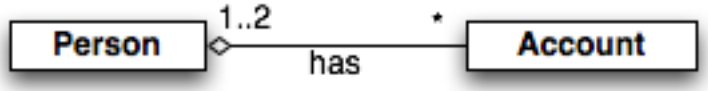
- Une **association** est une relation sémantique entre des classes qui définit un ensemble de liens.
  - Un homme est associé à sa femme
- Une **agrégation** est une association dans laquelle il y a un lien d'appartenance entre les deux objets associés (contenant/contenu, possession, …).
  - Un homme possède un compte en banque
- Une **composition** (ou agrégation forte) est une agrégation dans laquelle la disparition du composite entraine la disparition des composants.
  - Si un arbre meurt, ses feuilles ne servent plus à rien (on ne peut pas les mettre sur un autre arbre, au contraire de roues sur une voiture)
- Il s'agit surtout d'une différence sémantique qui impliquera des changements dans votre implémentation au niveau du cycle de vie des objets.

Association, composition, agrégation

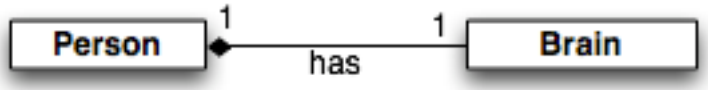
Association : Une personne travaille pour une et une seule compagnie



Agrégation : Une personne possède entre 0 et n comptes



Composition : Une personne a un et un seul cerveau



- Cardinalités :
  - **, n, m..*,* où *n>0* et *m>=0*
  - Par défaut, la cardinalité est *un*.
  - Dans une composition, la cardinalité du côté de l'agrégat ne peut être que 1 ou 0..1
- Le nom de l'association est facultatif

De la programmation fonctionnelle à la programmation OO

De la programmation fonctionnelle à la programmation OO

- il faut **casser le problème** en ses différents **modules**
- on passe d'une vision centralisée à une **vision plus décentralisée** de la programmation
- on ne gère plus le programme, on **délègue les classes** qui se délèguent elles-mêmes
- ne pas se préoccuper de ce que l'on fait avec les structures de données mais plutôt se demander ce que ces structures de données peuvent faire pour vous.
- dans la programmation fonctionnelle, on définit d'abord les données puis on crée une décomposition hiérarchique fonctionnelle. On est contraint de penser le programme de manière hiérarchique, de manière centralisée (toutes les fonctions peuvent agir sur toutes les données) et de manière temporelle

De la programmation fonctionnelle à la programmation OO (suite)

- dans la programmation OO, il n'y a pas vraiment de sommet ni de chronologie stricte, **les données et les fonctions sont éclatées.**
- on se préoccupe beaucoup moins de l'aspect temporel**, on lie les classes par des contraintes logiques et non plus temporelles**. On ne fait pas d'ordonnancement prématuré. Par ex., l'achat d'une maison est conditionné par l'obtention d'un prêt et la découverte d'une maison mais on n'ordonne pas entre elles les différentes opérations.
- on essaie de coller au mieux au problème au point que la programmation se transforme en modélisation
- le programme se pense de manière ascendante et non plus descendante
- les relations explicites entre les classes peuvent restées simples même si l'exécution computationnelle qui en découle peut s'avérer complexe.

Les avantages de la programmation OO

- **les programmes sont plus stables, plus robustes et plus faciles à maintenir car le couplage est faible entre les classes («encapsulation»)**
- elle facilite grandement le ré-emploi des programmes: par petite adaptation, par agrégation ou par héritage.
- émergence des «design patterns»
- il est plus facile de travailler de manière itérée et évolutive car les programmes sont facilement extensibles. On peut donc graduellement réduire le risque plutôt que de laisser la seule évaluation pour la fin.
- l'OO permet de faire une bonne analyse du problème suffisamment détachée de l'étape d'écriture du code - on peut travailler de manière très abstraite  UML
- l'OO colle beaucoup mieux à notre façon de percevoir et de découper le monde

Les avantages de l'OO (suite)

- Tous ces avantages de l'OO se font de plus en plus évident avec le grossissement des projets informatiques et la multiplication des acteurs. Des aspects tels l'encapsulation, l'héritage ou le polymorphisme prennent vraiment tout leur sens. On appréhende mieux les bénéfices de langage plus stable, plus facilement extensible et plus facilement ré-employable.
- JAVA est un langage strictement OO qui a été propulsé sur la scène par sa complémentarité avec Internet mais cette même complémentarité (avec ce réseau complètement ouvert) rend encore plus précieux les aspects de stabilité et de sécurité.

Les avantages en programmation

- Programmation = modélisation
- Stabilisation
- Récupération
- Modularisation
- Programmation évolutive

Les avantages en modélisation

- Idée de délégation entre classes
- Se demander comment les classes collaborent afin de vous servir
- L'héritage est un mécanisme cognitif (inventé par Minsky en IA)
- Modélisation <-> programmation

ULB UNIVERSITÉ LIBRE DE BRUXELLES

CoDe

Introduction à l'UML

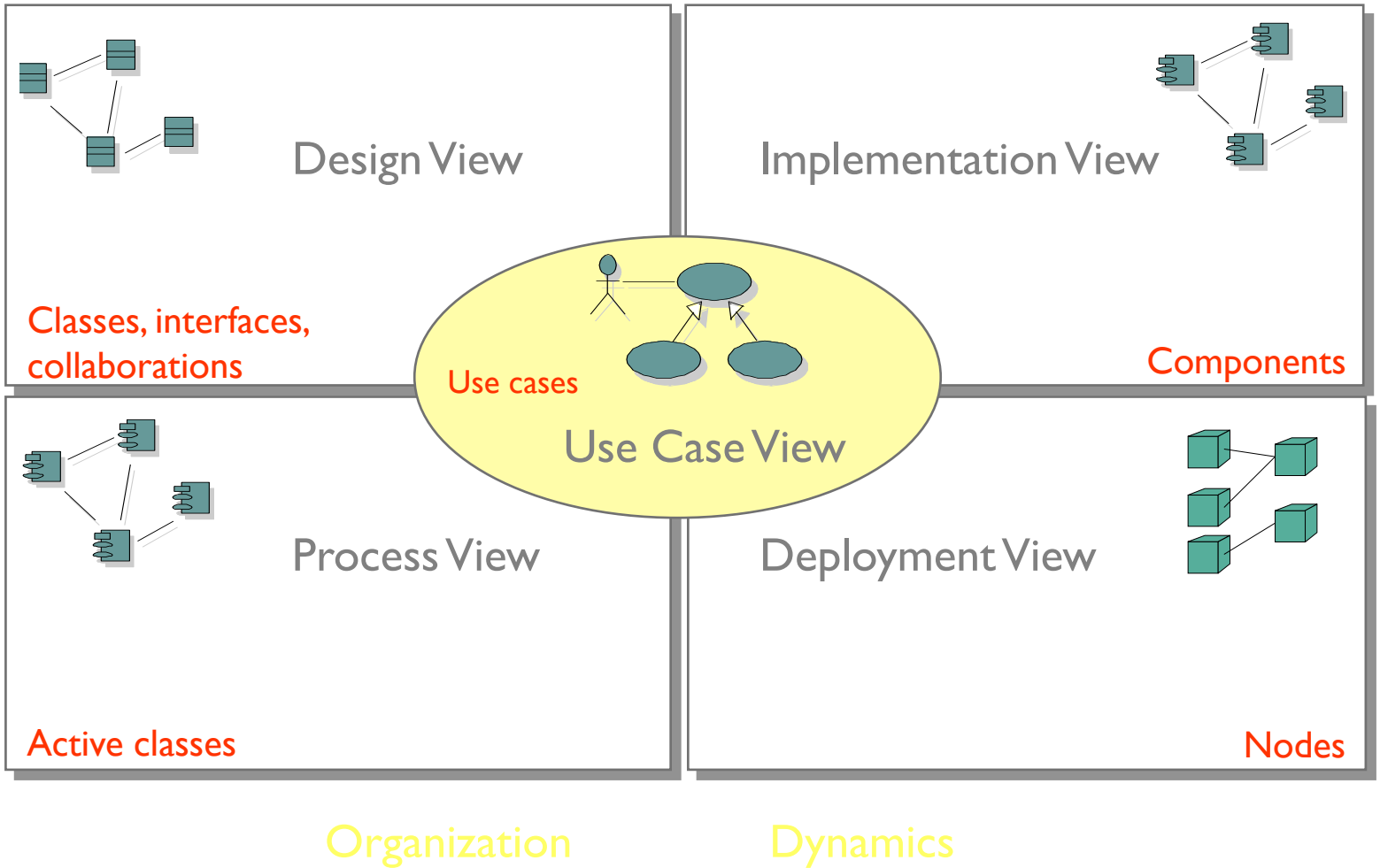
Introduction à UML

- UML (Unified Modeling Language)
- La fusion des travaux de G. Booch, J. Rumbaught et I. Jacobson - dans le prolongement d 'OMT.
- Les trois acteurs se rassemblent en 1995 en vue de standardiser l 'OO par UML.
- Développement de la société Rational, développement de l 'environnement Rational Rose dès 1995
- L 'OMG a imposé UML comme un standard en septembre 97, supporté par les principaux acteurs de l 'informatique: Microsoft, Oracle, HP, Texas, ...
- Actuellement, version UML 2

Introduction à UML (suite)

- UML est un langage objet graphique - un formalisme orienté objet, basé sur des diagrammes (13)
- UML permet de s 'abstraire du code par une représentation des interactions statiques et du déroulement dynamique de l 'application.
- UML prend en compte, le cahier de charge, l 'architecture statique, la dynamique et les aspects implémentation
- Facilite l 'interaction, les gros projets.
- Générateur de squelette de code
- UML est un langage PAS une méthodologie, aucune démarche n'est proposée juste une notation.

Architecture and the UML

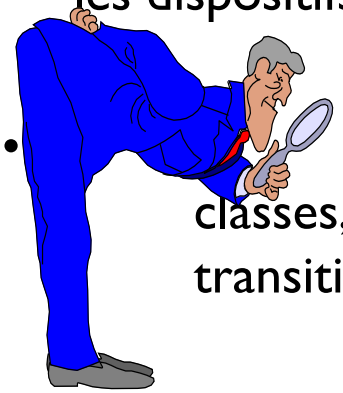


UML-1: 9 diagrammes principaux

- Les diagrammes des cas d 'utilisation: les fonctions du système, du point de vue de l 'utilisateur ou d 'un système extérieur - l 'usage que l 'on en fait.
- Les diagrammes de classes: une description statique des relations entre les classes.
- Les diagrammes d 'objet: une description statique des objets et de leurs relations. Une version « instanciée » du précédent.
- Les diagrammes de séquence: un déroulement temporel des objets et de leurs interactions
- Les diagrammes de collaboration: les objets et leurs interactions en termes d 'envois de message + prise en compte de la séquentialité

Les 9 diagrammes (suite)

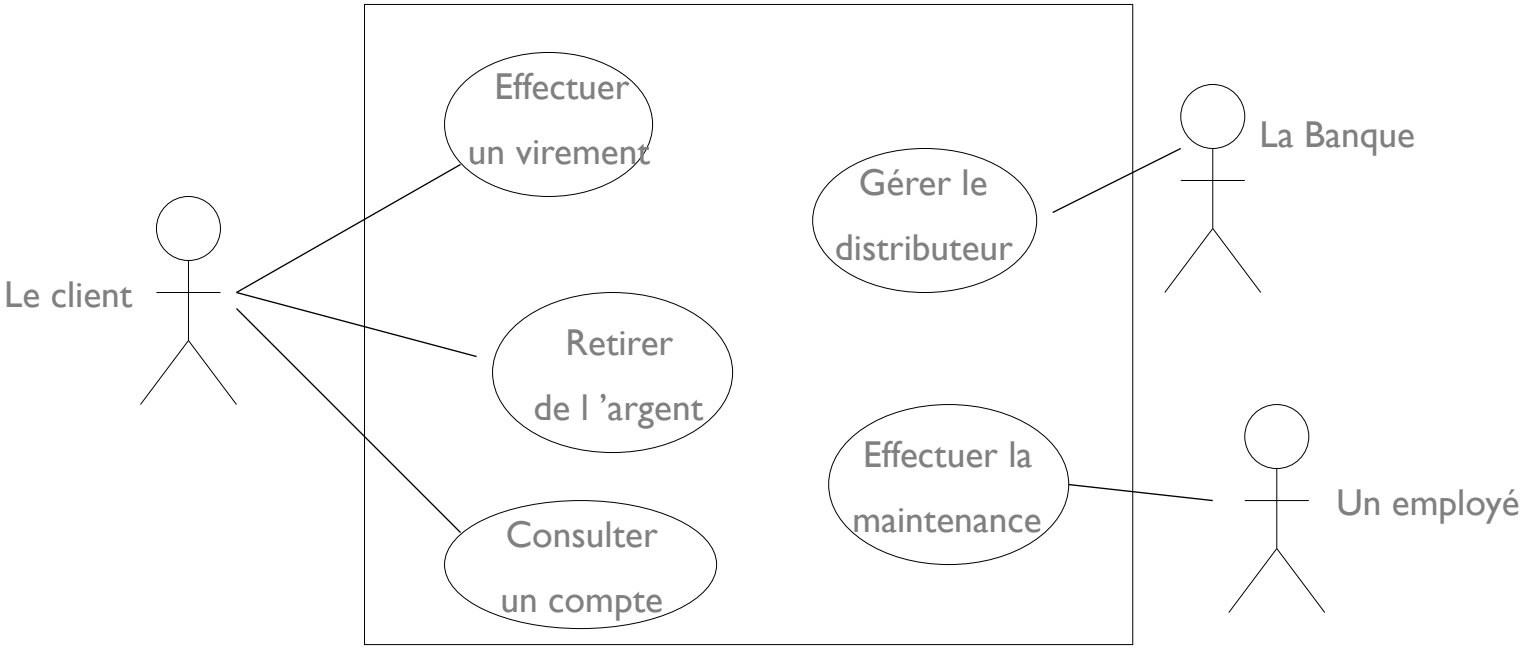
- Les diagrammes d 'états-transitions: scrute les cycles de vie d 'une classe d 'objet, la succession d 'états et les transitions.
- Les diagrammes d 'activité: le comportement des différentes opérations en termes d 'actions.
- Les diagrammes de composants: représente les composants physiques d 'une application.
- Les diagrammes de déploiements: le déploiement des composants sur les dispositifs et les supports matériels.



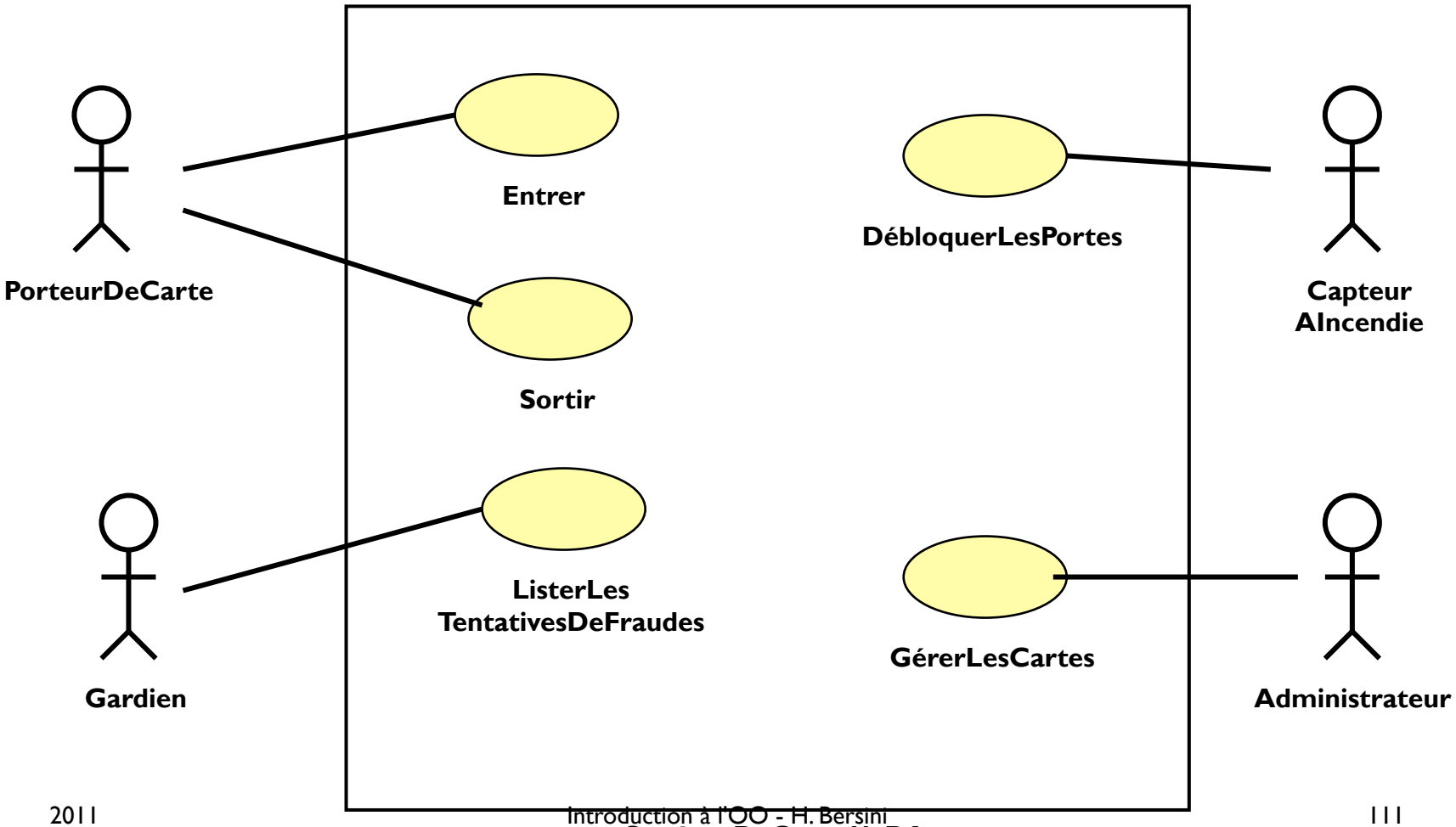
- Les diagrammes: de cas d 'utilisation, de classes, de collaboration, de séquences et d 'états-transitions.

Le diagramme des cas d'utilisation (use cases)

- Cela répond aux spécifications du système. Ses fonctionnalités, son utilisation, les attentes de l 'utilisateur. Par. Ex: le mister Cash.



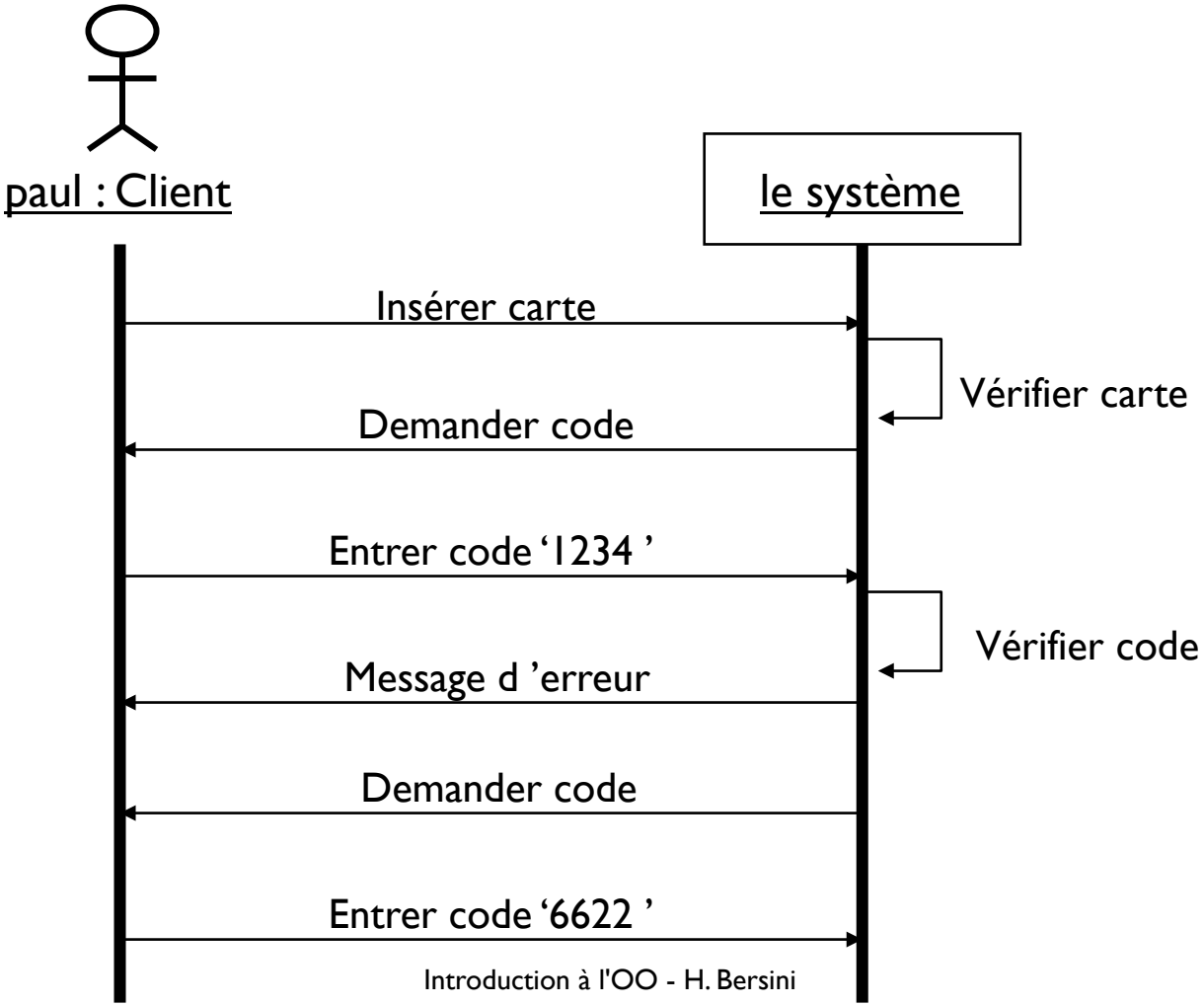
Diagrammes des cas d 'utilisation



Les cas d 'utilisation (suite)

- L 'acteur délimite le système, il peut être humain, logiciel ou matériel. Il est l 'utilisateur du système. Les acteurs peuvent être nombreux.
- Les cas d 'utilisation peuvent également se décrire comme des scénarios.
- Le diagramme de ces cas indiquent les acteurs et les différents types d'utilisation.
- Il faut donc indiquer l'acteur ainsi que l'utilisation qu'il fait du système.
- En déroulant ces cas, faisant des scénarios plus précis, on arrive aux diagrammes de séquences.

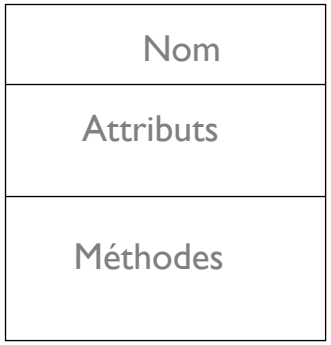
Exemple de scénario
Cas d 'utilisation détaillé

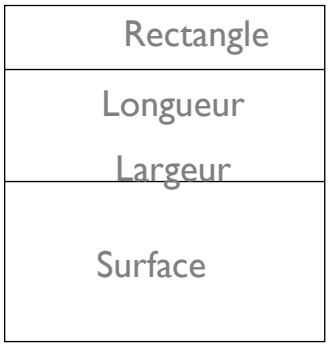


...

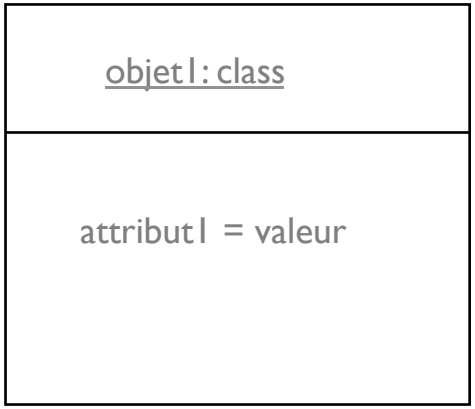
Les diagrammes de classe

- Un réseau de classes et d 'associations
- Les associations peuvent être quelconques:
  mari et femme, employé et entreprise,…
- Ou plus classique: généralisation, aggrégation
- Mais d 'abord comment représente-t-on une classe en UML ?



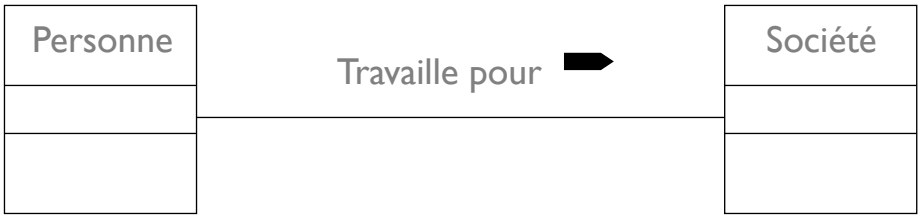


Les objets



Les diagrammes de classe (suite)

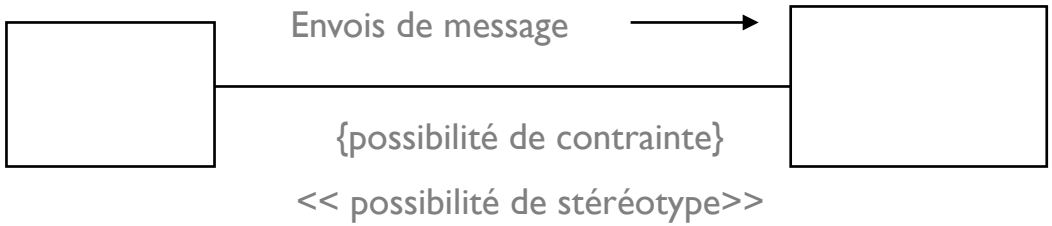
- L 'association entre classes



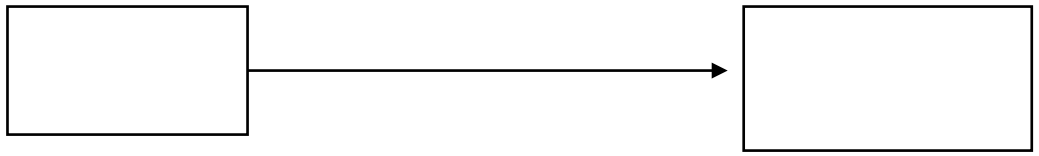
- Multiplicité des associations

L'assocation

- Il y a association entre classes quand il y a transmission de messages entre classes.

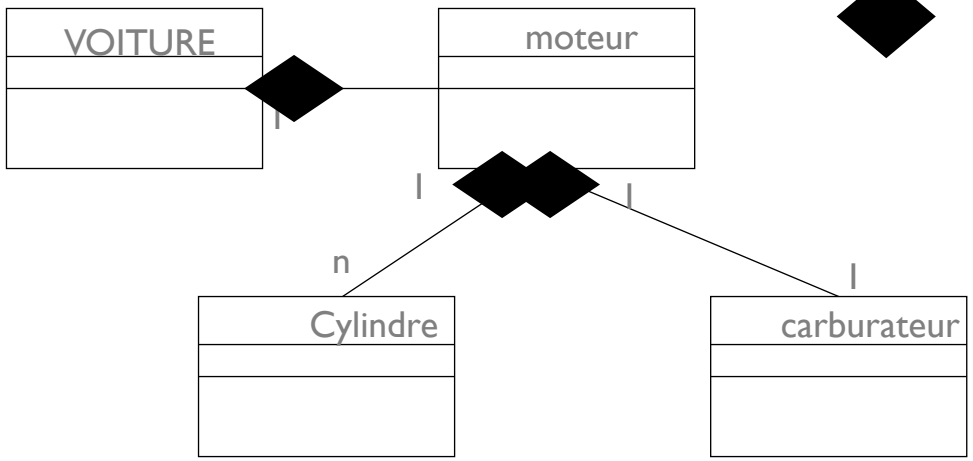


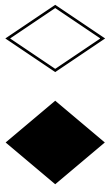
- L'association peut être directionnelle:



Les diagrammes de classe (suite)

- Un type d 'association particulier:
  - l 'aggrégation
- L 'aggrégation:

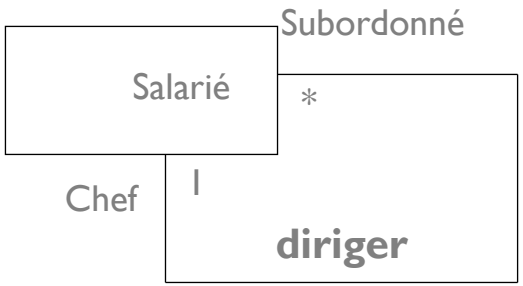


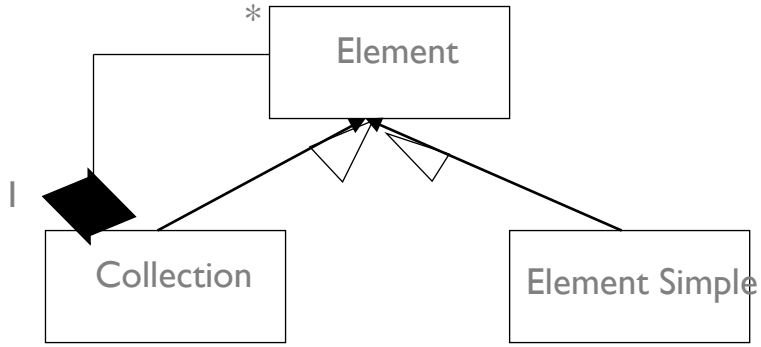


aggrégation

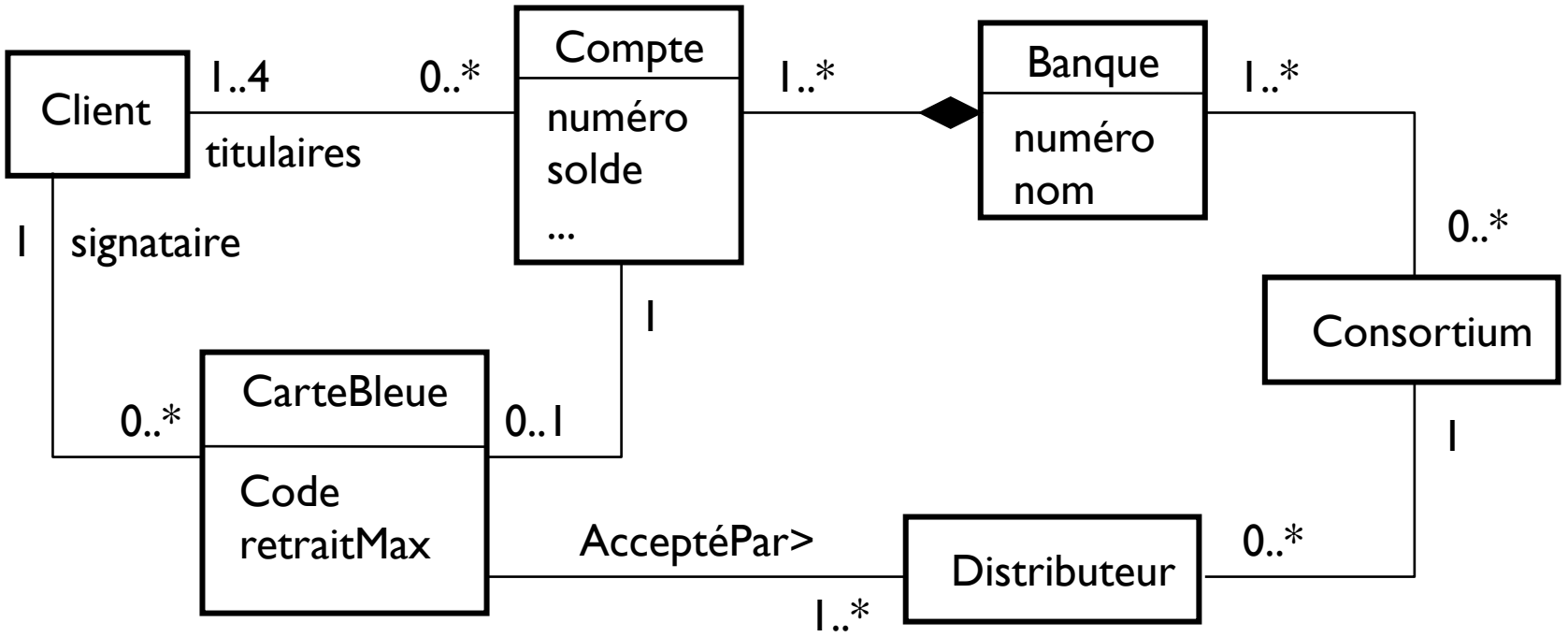
Composition: l'existence des parties est conditionnée par l'existence du contenant. En C++ cela permet de choisir entre pointeur et value

Diagrammes de classes (suite)

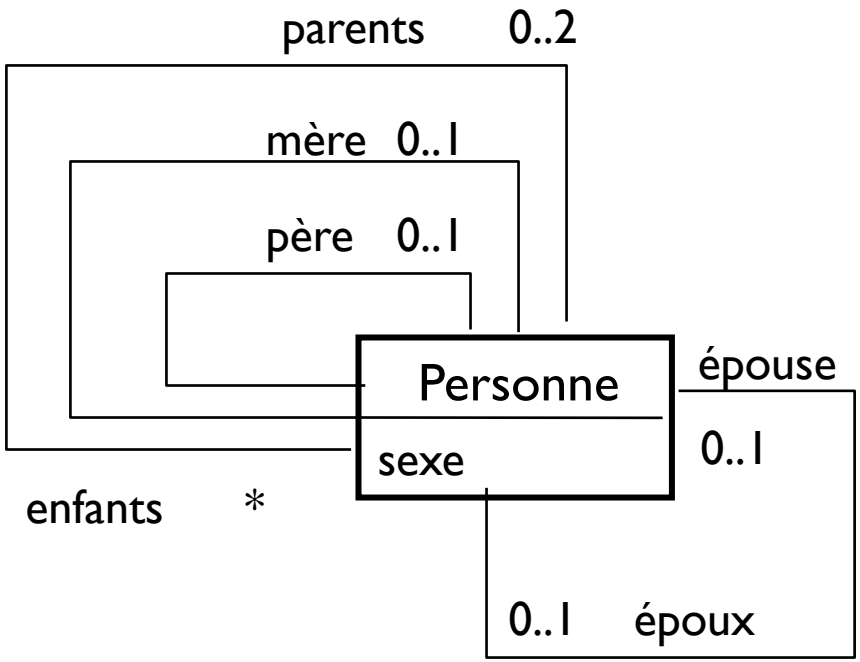


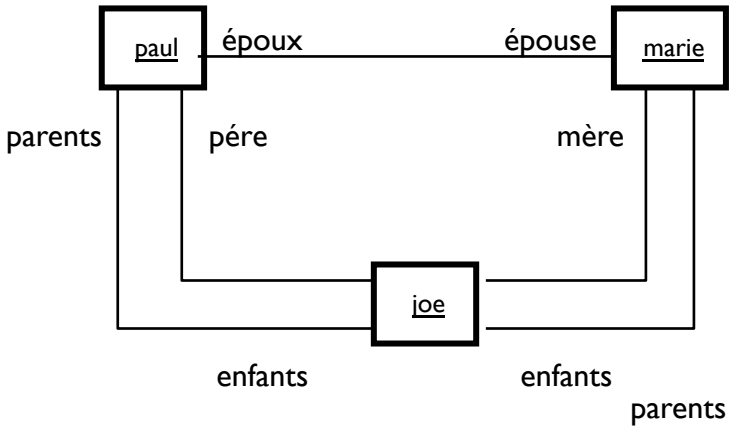


Diagrammes de classes

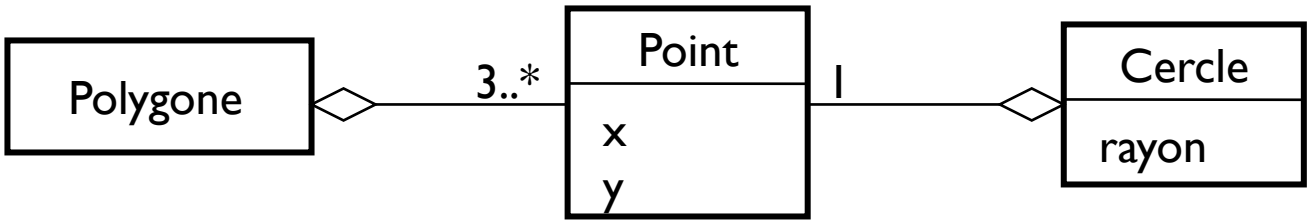


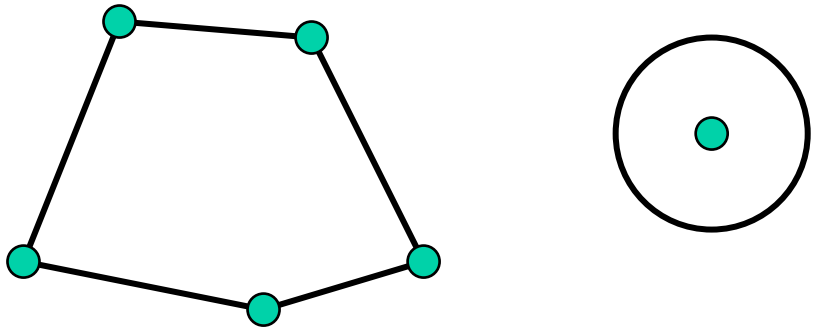
Exemple


parents
0..2
mère
0..1
père
0..1
Personne
sexe
épouse
0..1
enfants
*
0..1
époux


paul
époux
épouse
marie
parents
père
mère
joe
enfants
enfants
parents

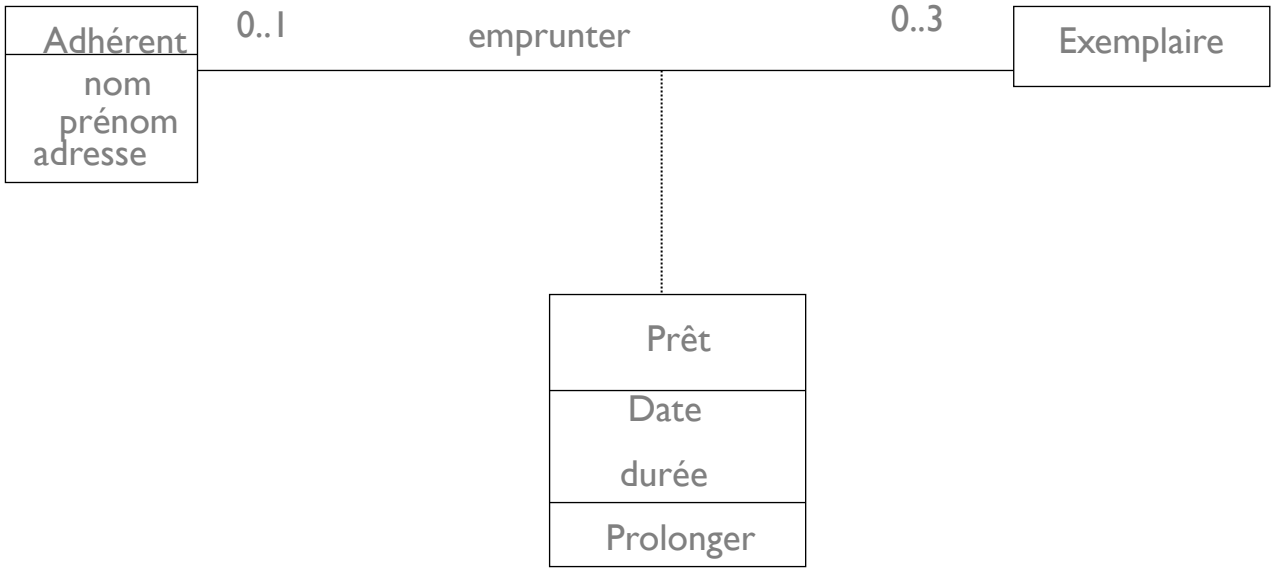
Composition





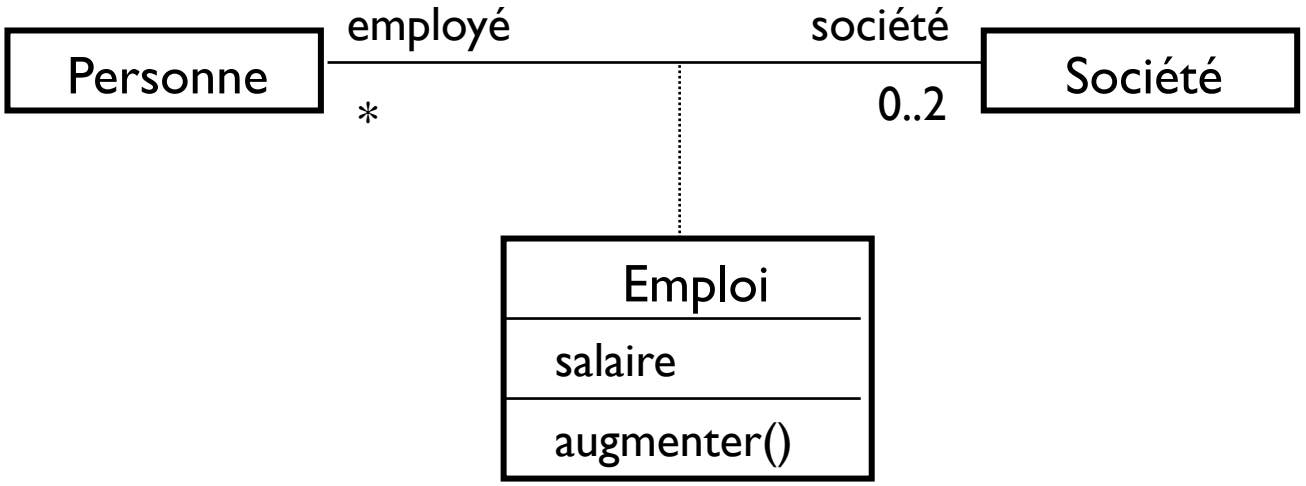
Diagrammes de classe (suite)

- Dans certains cas, l 'association peut elle-même devenir une classe.

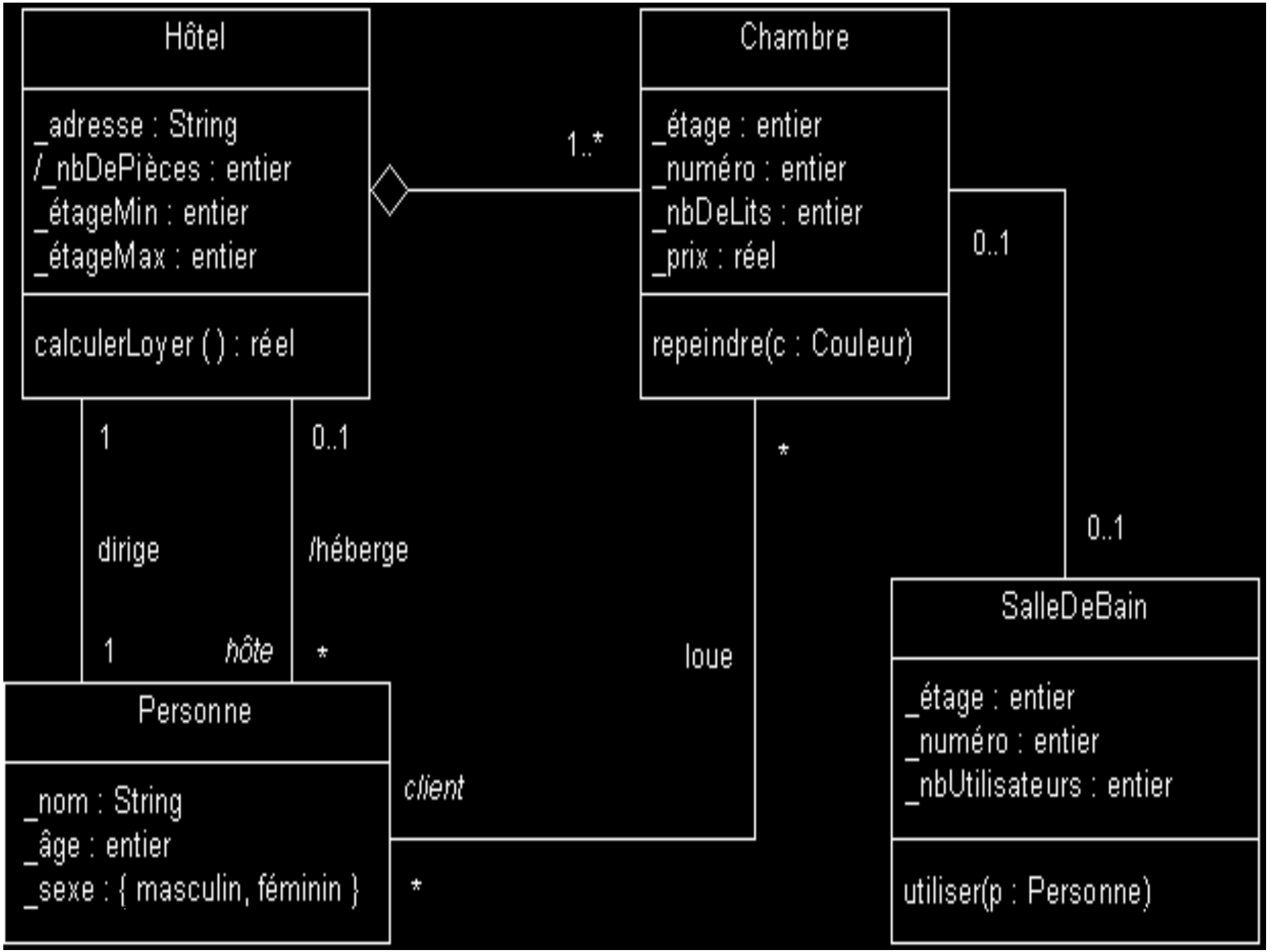


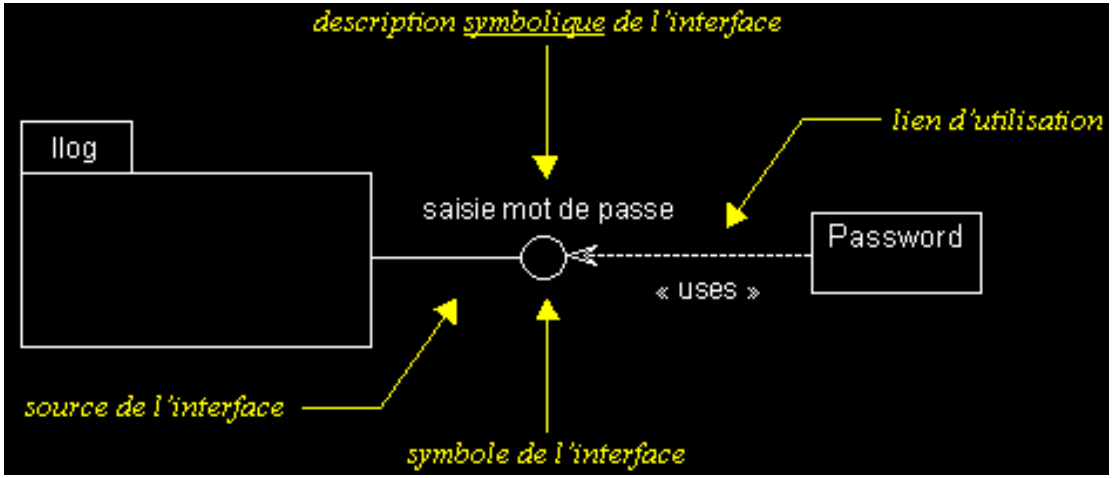
Classes associatives

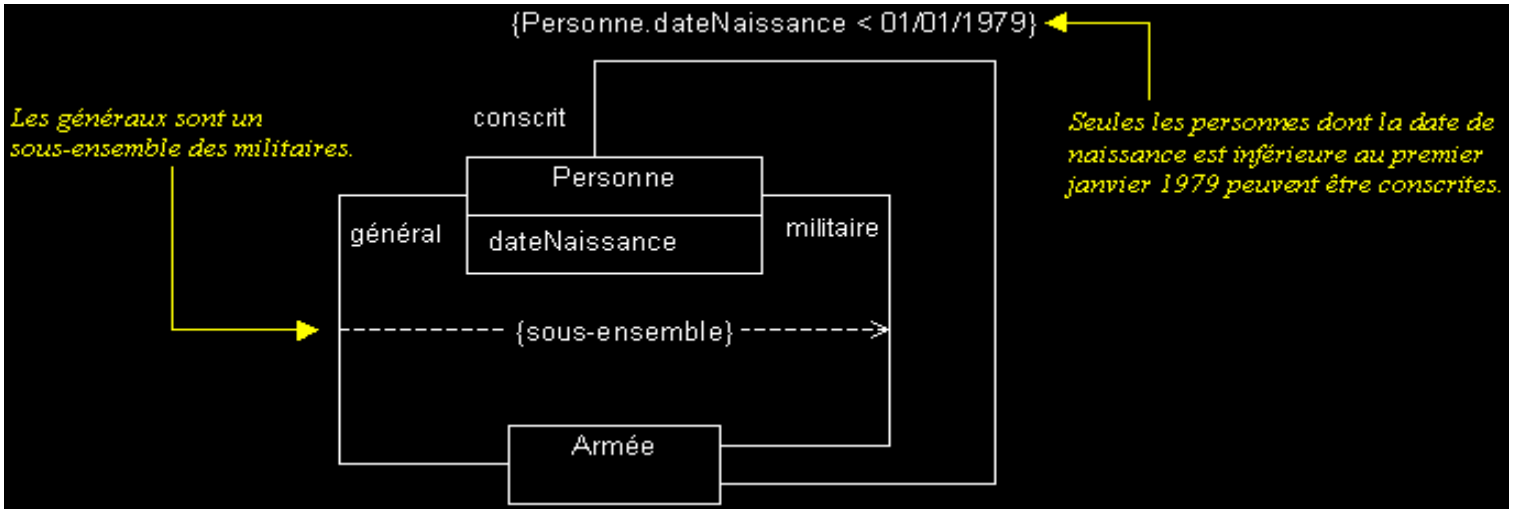
il est utile d'associer des attributs et/ou des méthodes aux associations => classes associatives



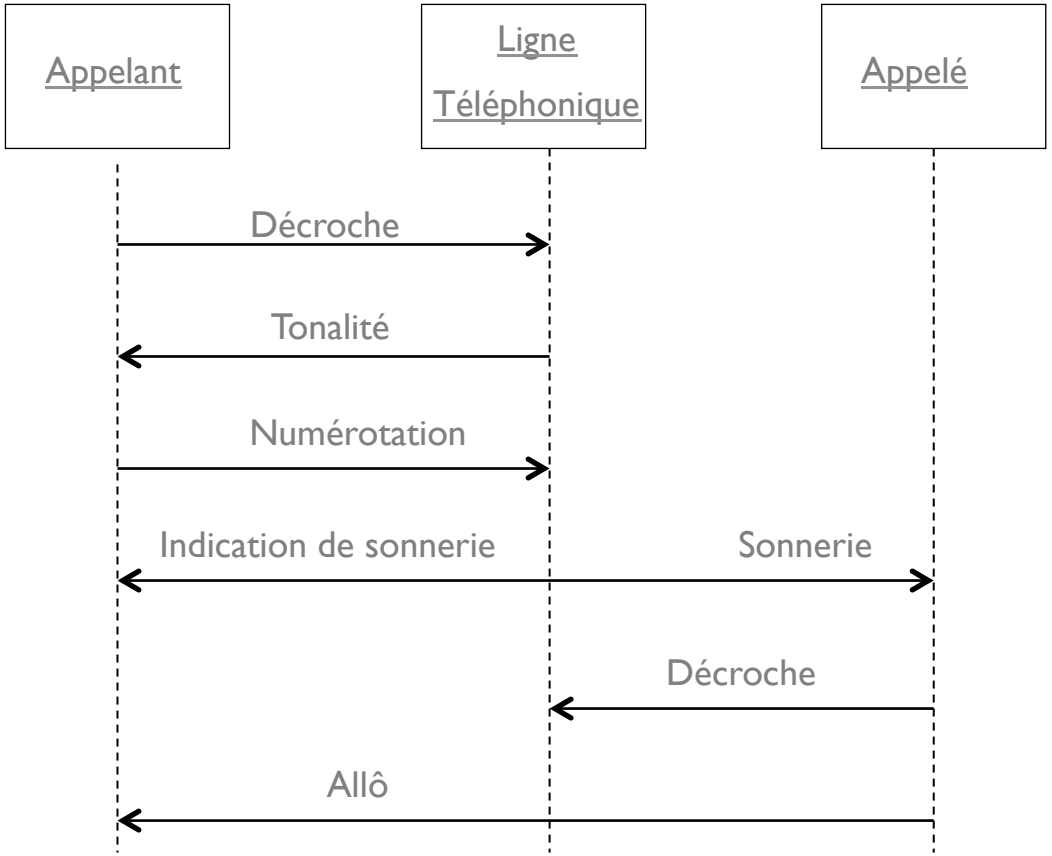
Le nom de la classe correspond au nom de l'association


Hôtel
_adresse : String
/_nbDePièces : entier
_étageMin : entier
_étageMax : entier
calculerLoyer ( ) : réel
Chambre
_étage : entier
_numéro : entier
_nbDeLits : entier
_prix : réel
repeindre(c : Couleur)
1..*
0..1
1
dirige
1
hôte
0..1
/héberge
*
client
*
*
loue
0..1
SalleDeBain
_étage : entier
_numéro : entier
_nbUtilisateurs : entier
utiliser(p : Personne)
Personne
_nom : String
_âge : entier
_sexe : { masculin, féminin }


description symbolique de l'interface
lien d'utilisation
llog
saisie mot de passe
Password
« uses »
source de l'interface
symbole de l'interface


{Personne.dateNaissance < 01/01/1979}
Les généraux sont un sous-ensemble des militaires.
conscrit
Seules les personnes dont la date de naissance est inférieure au premier janvier 1979 peuvent être conscrites.
Personne
général
dateNaissance
militaire
{sous-ensemble}
Armée

Diagrammes de séquence (suite)



Diagrammes de séquence

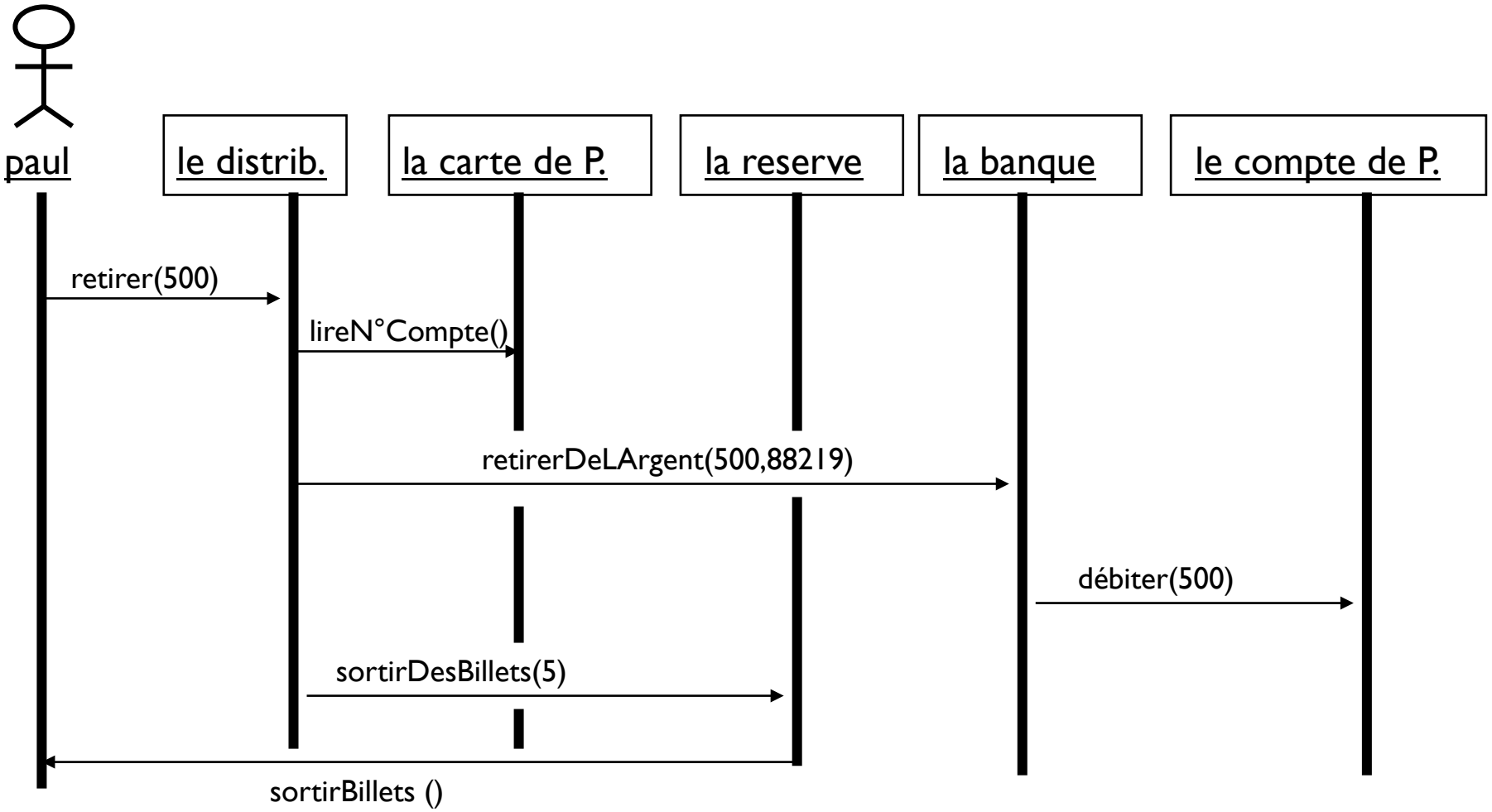
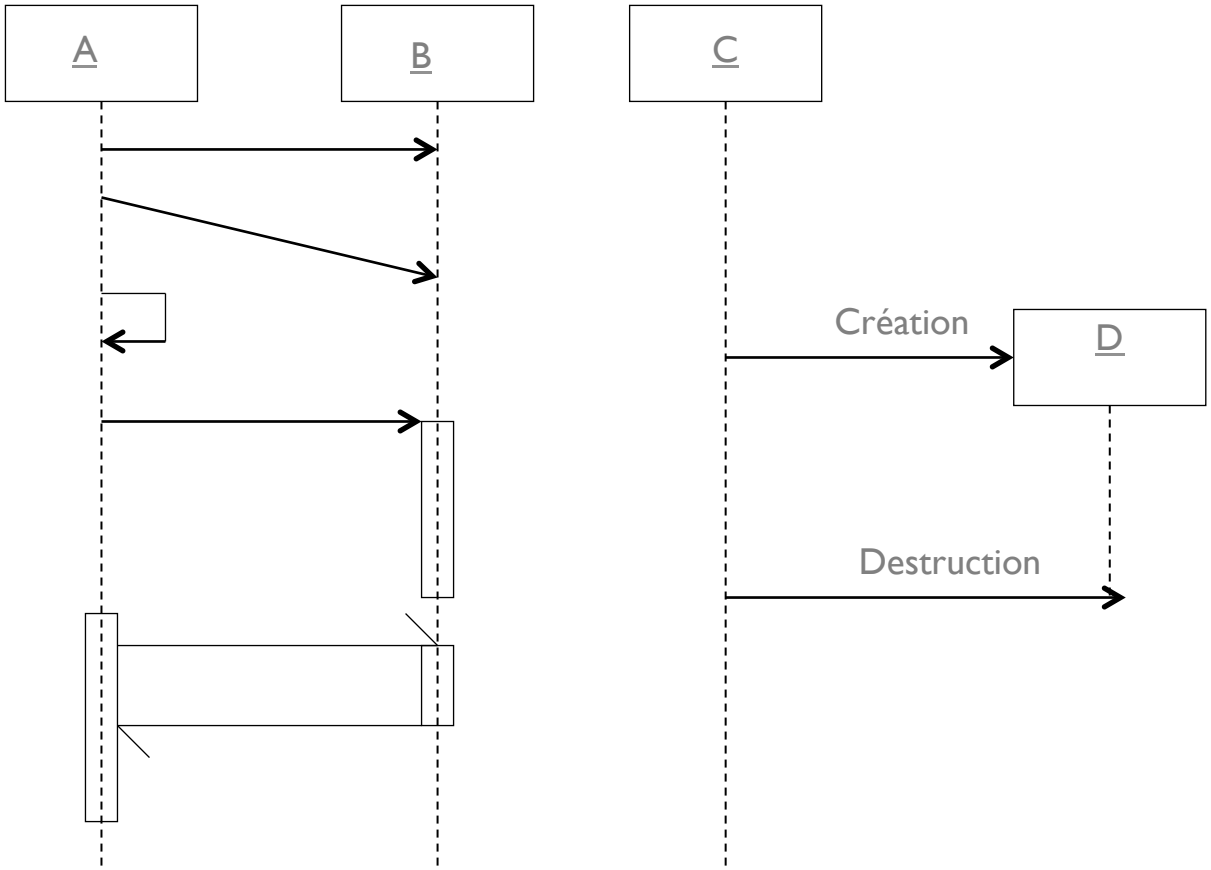
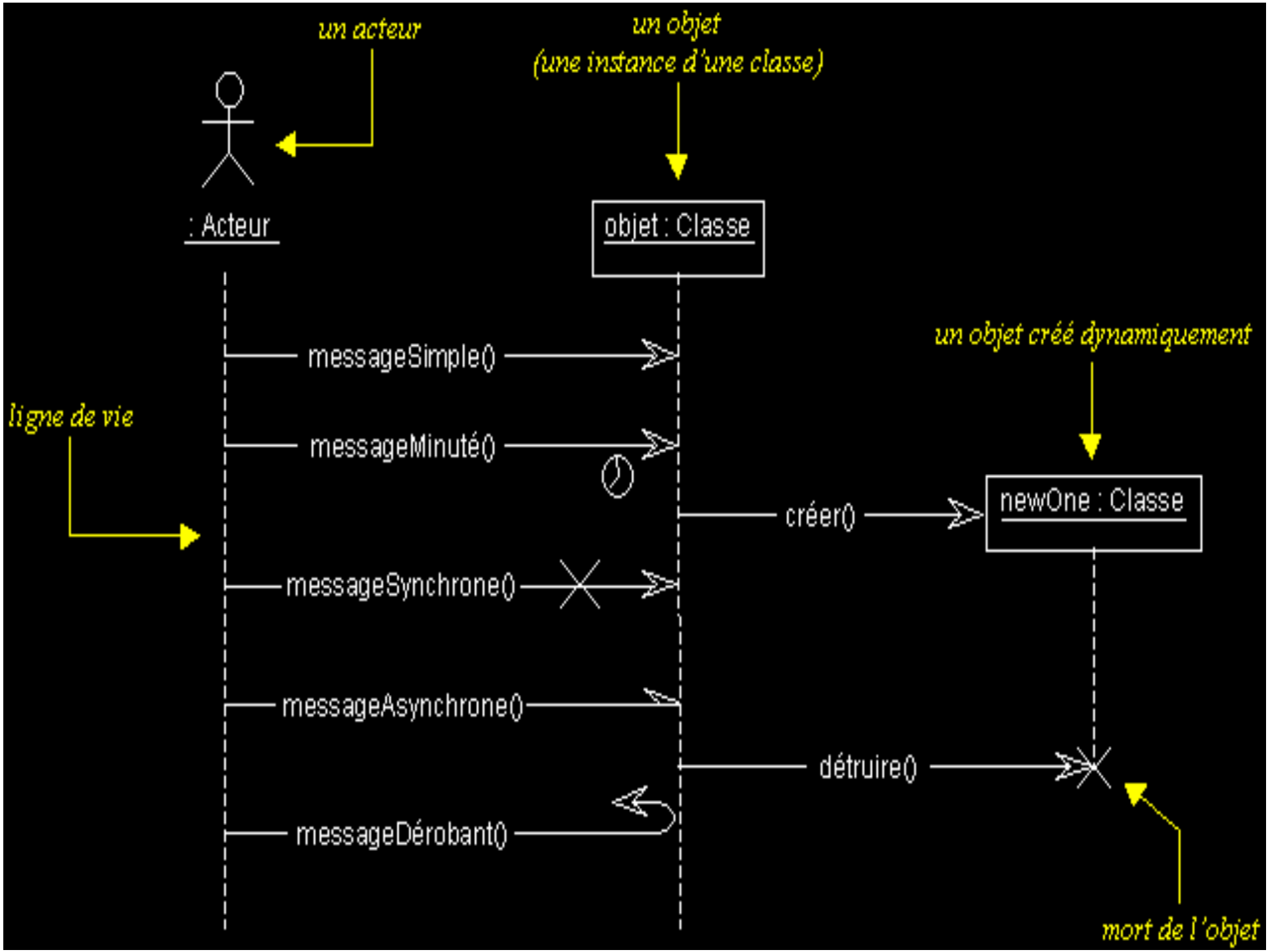
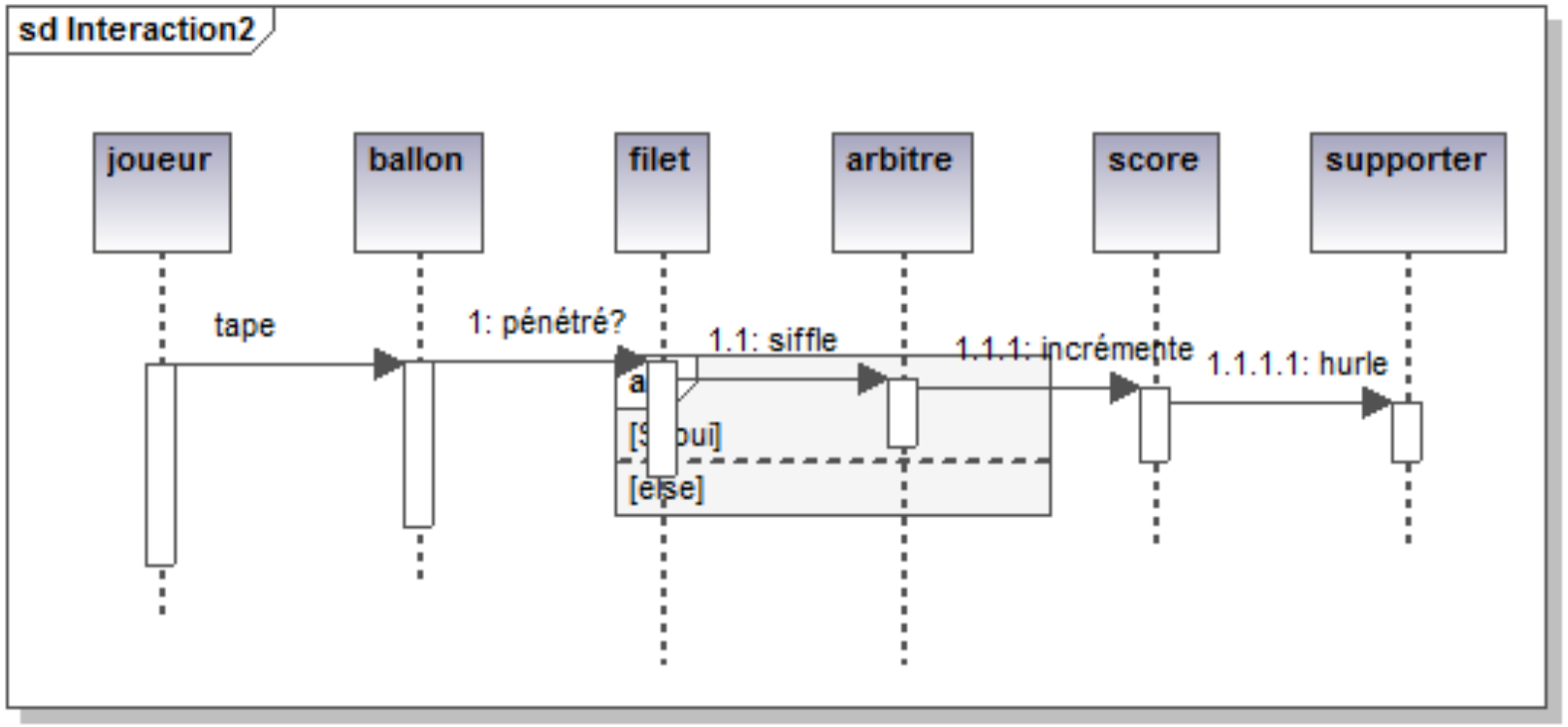


Diagramme de séquence (suite)
Quelques exemples




un acteur
un objet
(une instance d'une classe)
: Acteur
objet : Classe
un objet créé dynamiquement
ligne de vie
messageSimple()
messageMinuté()
créer()
newOne : Classe
messageSynchrone()
messageAsynchrone()
détruire()
messageDérobant()
mort de l'objet


sd Interaction2
joueur
ballon
filet
arbitre
score
supporter
tape
1: pénétré?
1.1: siffle
1.1.1: incrémente
1.1.1.1: hurle
[else]
Generated by UModel
www.altova.com

ULB UNIVERSITÉ LIBRE DE BRUXELLES

CoDG

Introduction aux Design Patterns

Bonnes pratiques OO

- "*Controlling complexity is the essence of computer programming.*"(Brian Kernighan)

- Bonnes pratiques :
  - **Low Coupling** (modifier une classe sans toucher aux autres)
  - **High Cohesion** (idéalement, toutes les méthodes d'une classe utilisent toutes ses donnes)
  - **Expert**
  - **Don't talk to strangers**
  - **Polymorphism**

Patrons de conception

- Concept de **génie logiciel** visant à résoudre des **problèmes récurrents** d'architecture et de conception logicielle suivant le paradigme objet.
- Formalisation de **bonnes pratiques.**
- Capitalisation de l'expérience appliquéee à la conception logicielle.

- Réduire les interactions entre les objets, diminer le temps de développement
  - → gagner des sous ;-)

- Aussi appellé **design pattern** ou motif de conception

Patrons de conception

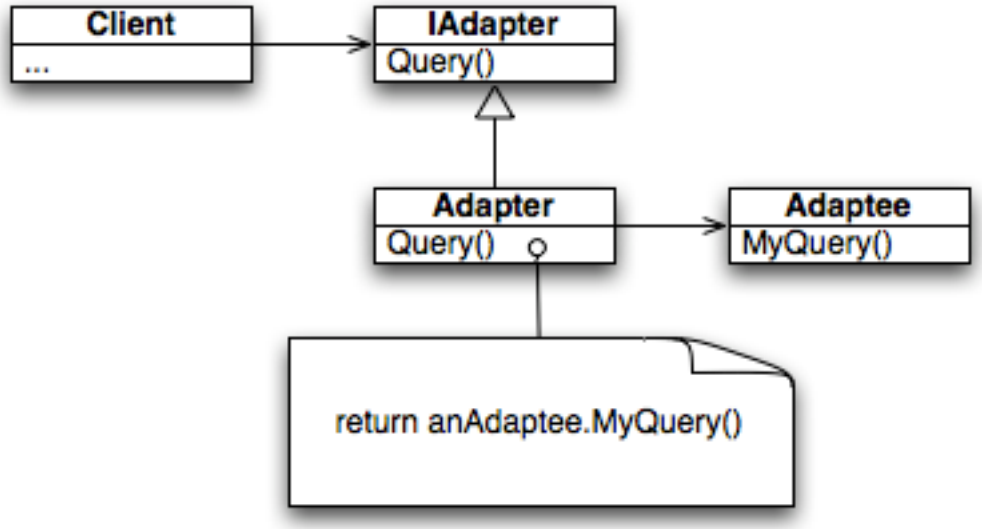
- Formalisés en 1995 par le « Gang of Four »
- Trois grandes familles :
  - **Patrons de construction** : ***instancier*** et configurer des classes et des objets
  - **Patrons structuraux** : organiser les classes d'un programme en séparant l'***interface*** de l'implémentation
  - **Patrons comportementaux** : organiser les objets pour que ceux-ci ***collaborent***

Quelques patrons structuraux

Organiser les classes d'un programme en séparant l'**interface** de l'implémentation

Adapter

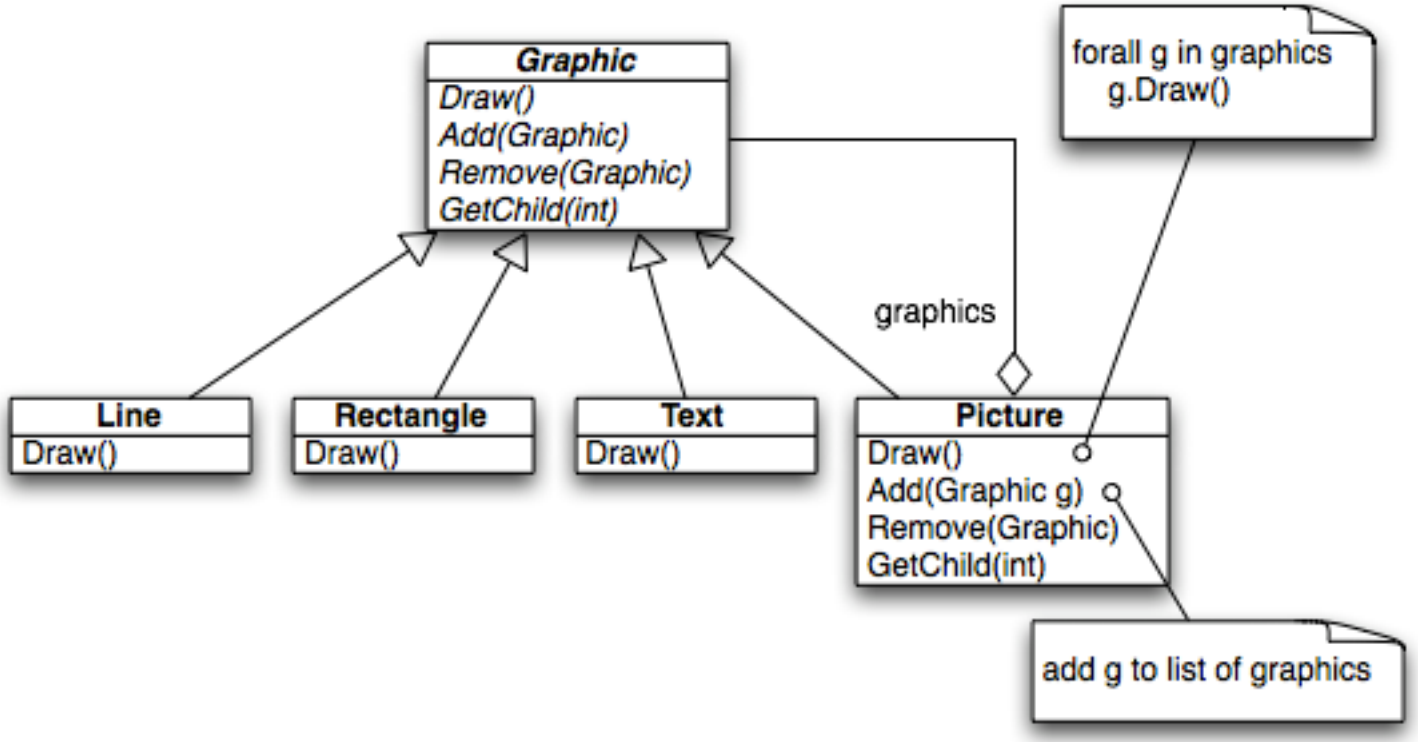
- Convertir **l'interface** d'une classe en une autre interface que le client attend



Composite

- Un objet composite est constitué d'un ou de **plusieurs objets similaires** (ayant des fonctionnalités similaires)
- L'idée est de **manipuler un groupe** d'objets de la même façon que s'il s'agissait **d'un seul objet**
- Les objets regroupés doivent avoir des **méthodes communes**.
- Souvent utilisé à l'aide du polymorphisme.

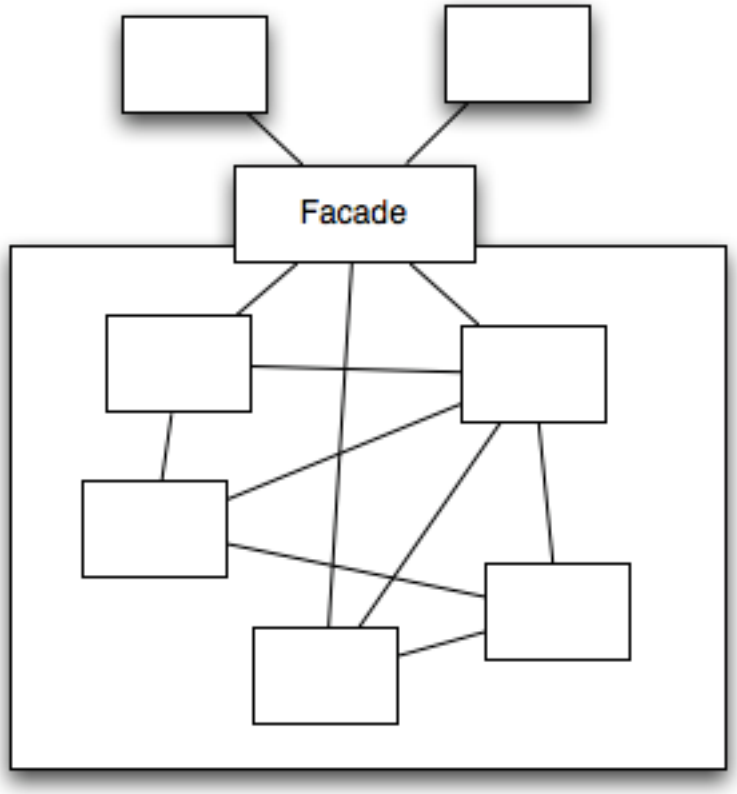
Composite


Graphic
Draw()
Add(Graphic)
Remove(Graphic)
GetChild(int)
forall g in graphics
g.Draw()
graphics
Line
Draw()
Rectangle
Draw()
Text
Draw()
Picture
Draw()
Add(Graphic g)
Remove(Graphic)
GetChild(int)
add g to list of graphics

Facade

- But : **cacher une conception** et une interface complexe difficile à comprendre

- La façade permet de **simplifier** cette complexité en fournissant une interface simple du sous-système.



ULB UNIVERSITÉ LIBRE DE BRUXELLES

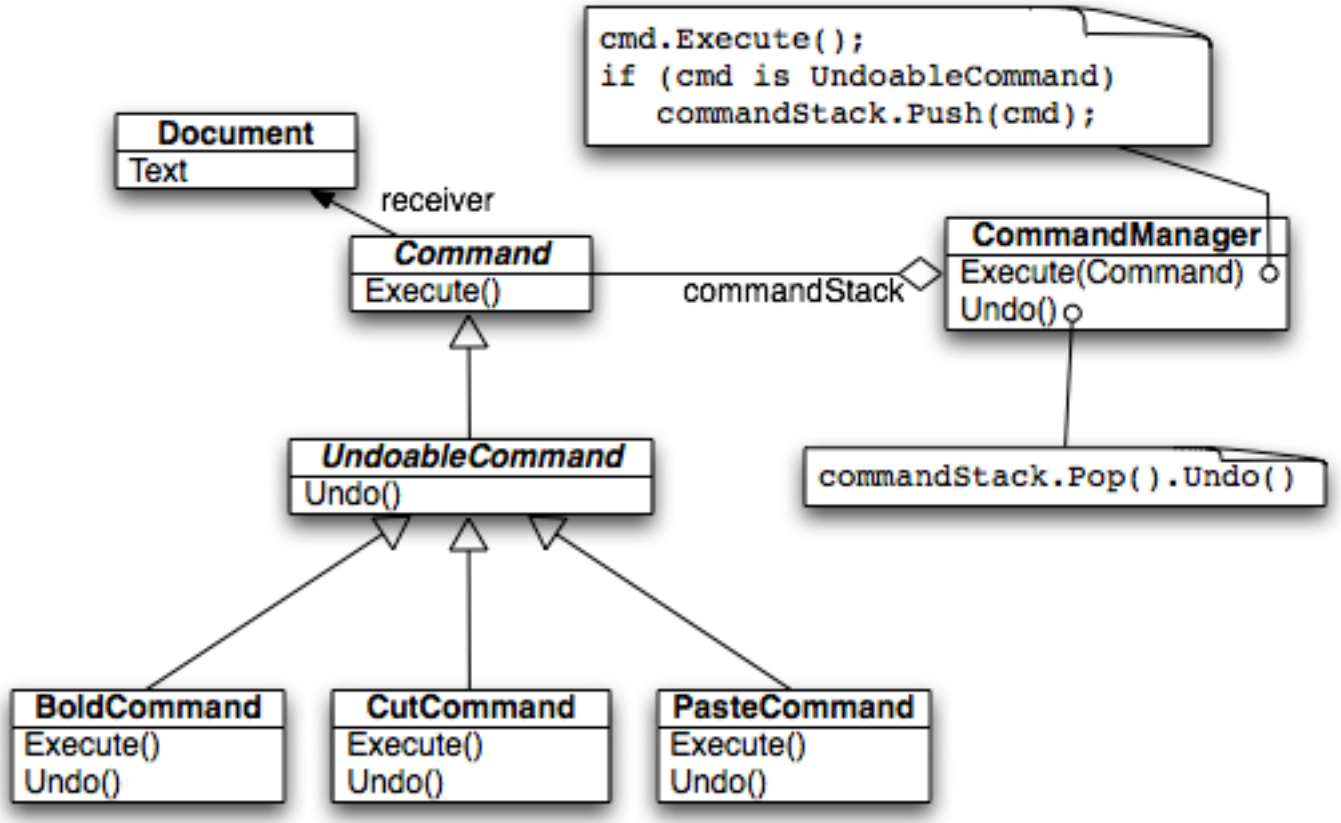
CoDe

Quelques patrons comportementaux

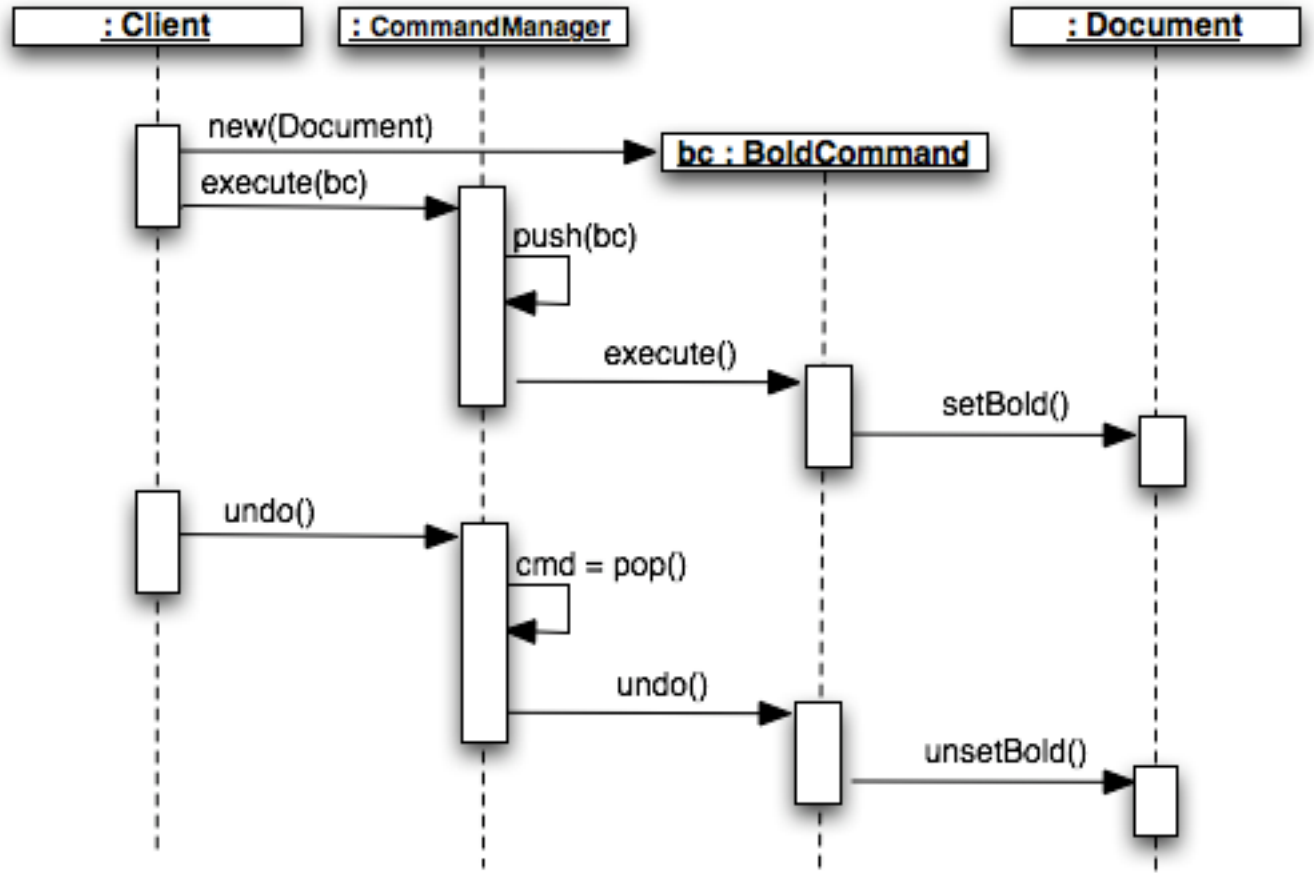
Organiser les objets pour que ceux-ci
collaborent

Command

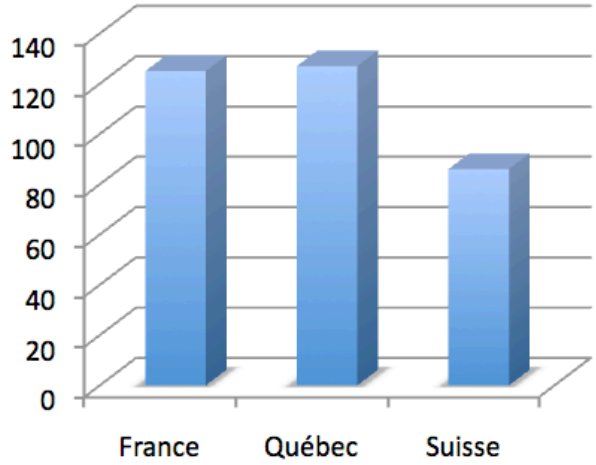
- **Encapsuler une requête/commande comme un objet.**
- Ex :
  - Annuler une commande
  - undo()

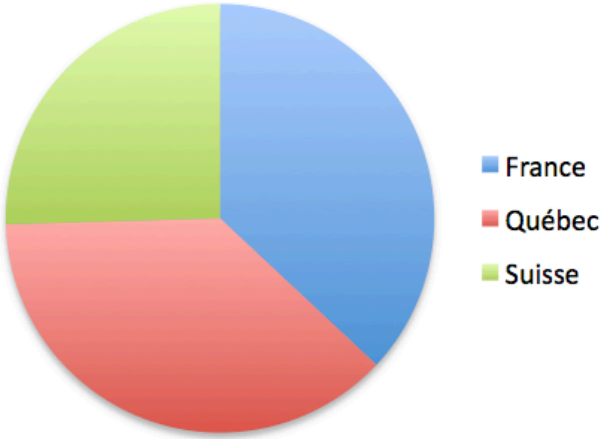


Command


: Client
: CommandManager
: Document
new(Document)
bc : BoldCommand
execute(bc)
push(bc)
execute()
setBold()
undo()
cmd = pop()
undo()
unsetBold()

Patron de conception Observer





France	125
Québec	127
Suisse	86

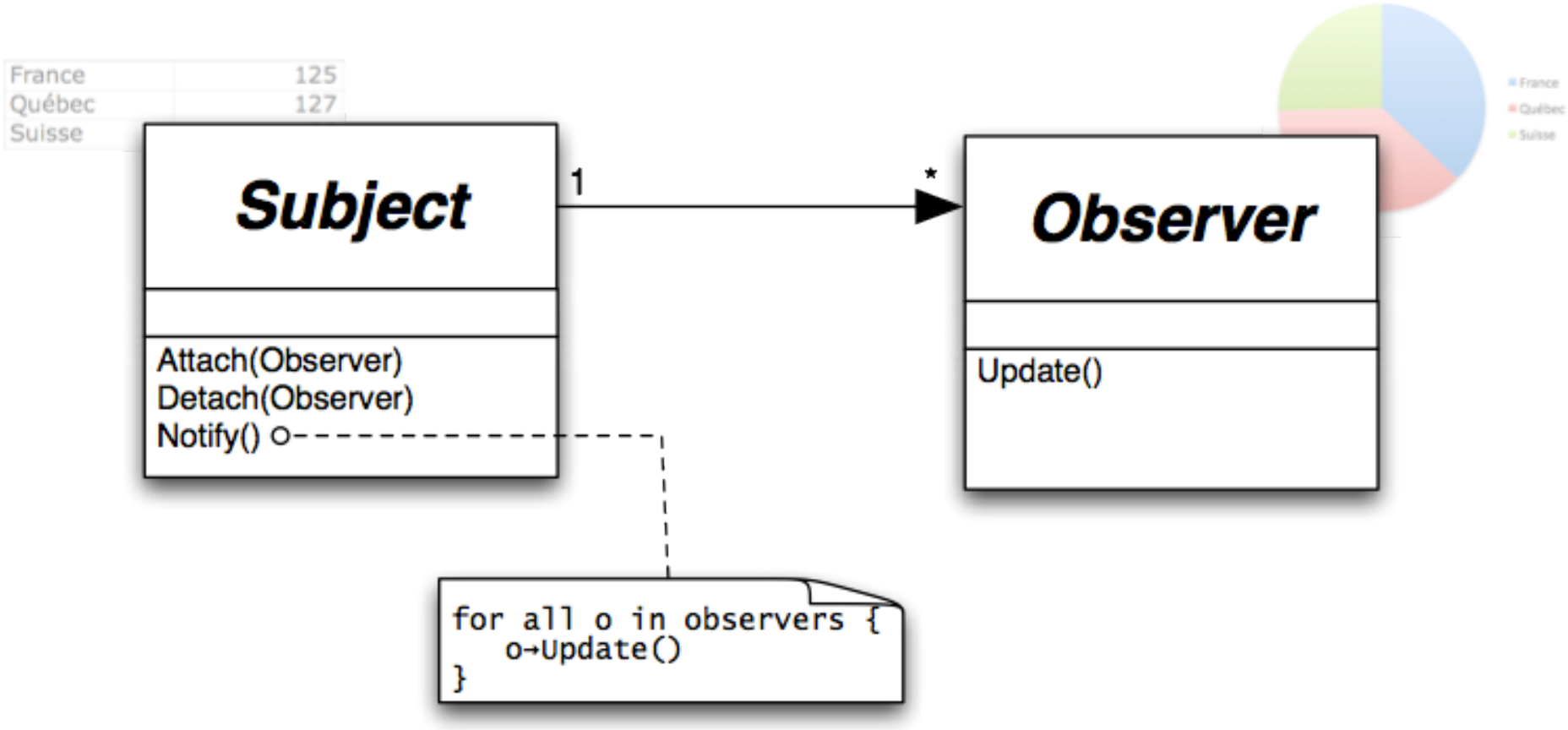
Patron de conception Observer

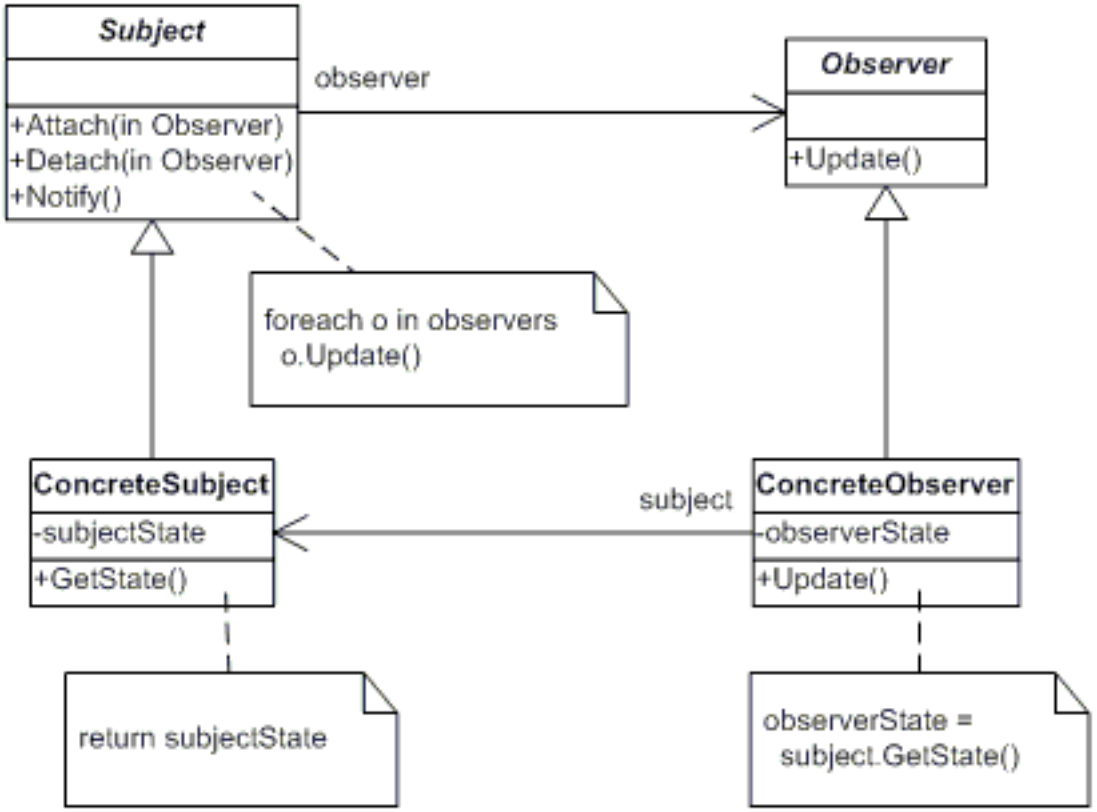
- Un *observateur* observe des *observables*.

- En cas de **notification**, les observateurs effectuent une action en fonction des informations qui viennent des observables.
- La notion d'**observateur/observable** permet de coupler des modules de façon à réduire les dépendances aux seuls phénomènes observés.

- Notions **d'événements (voir cours sur les événements)**

Patron de conception Observer



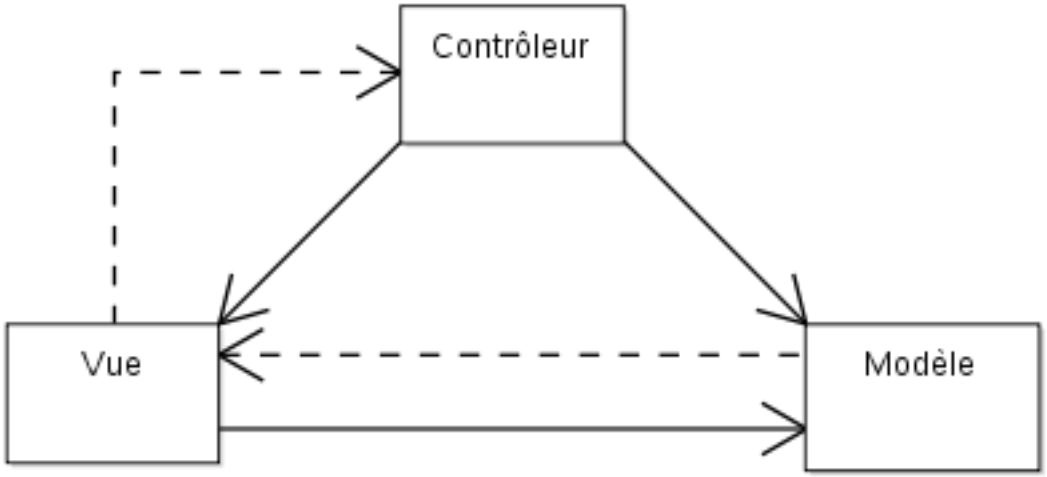


Exemple :

- Le sujet concret est *un cours de bourse*
- L'observer concret et un *Investisseur (Jean)*

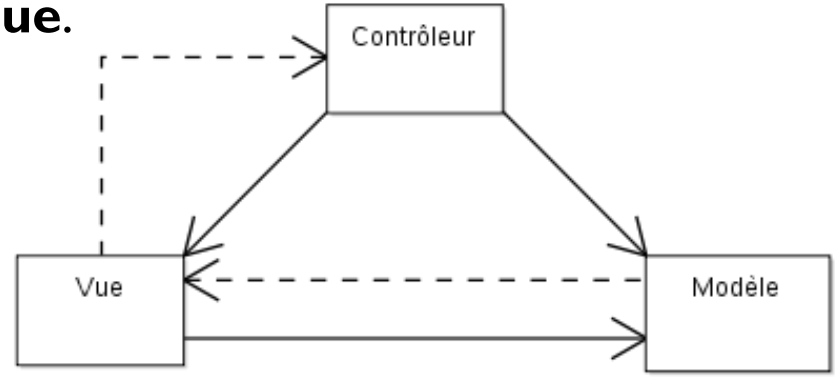
L'investisseur (ConcreteObserver) veut être notifié à chaque fois que le cours d'IBM (ConcreteSubject) change

Le patron Model-View-Controller



Model-View-Controller

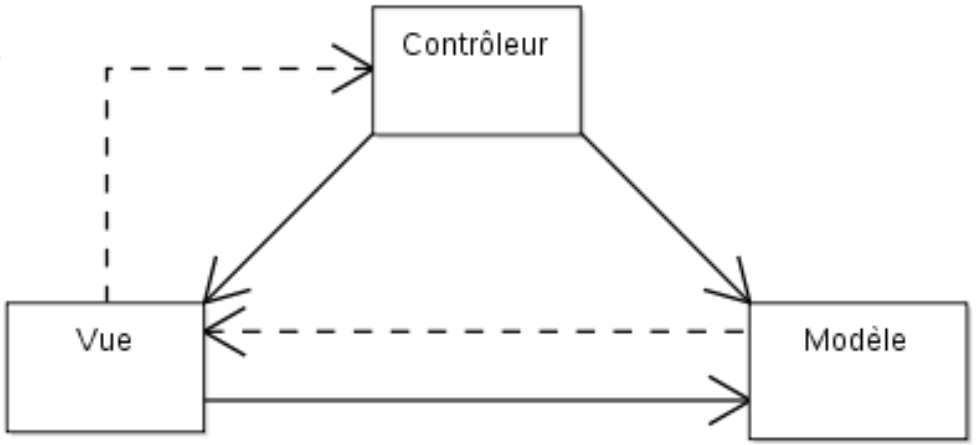
- Méthode de conception pour organiser **l'architecture d'une application avec interface graphique**.



- On divise l'application entre :
  - **Modèle** (de données, connection avec la DB, etc, notification de la vue)
  - **Vue** (présentation, interface, notification du controleur, répond aux notifications du modèle)
  - **Contrôleur** (logique de contrôle, synchronisation, répond aux notifications de la vue)

Modèle

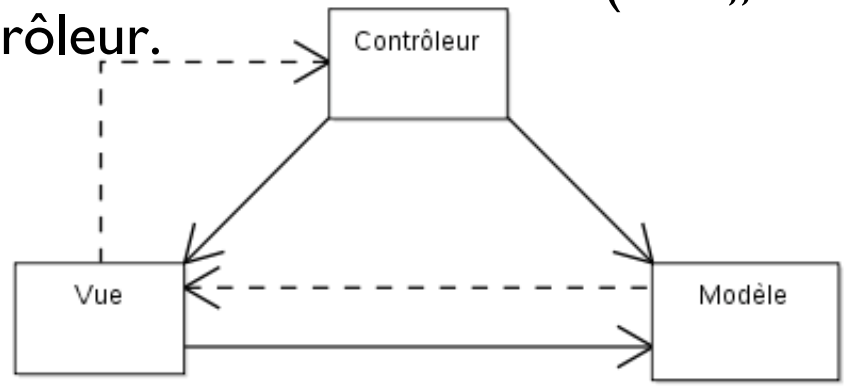
- **Comportement** de l'application :
  - traitement des **données**
  - interaction avec la base de données
  - garanti l'intégrité des données
  - signale à la vue que les données ont changé (via un événement)



- Le modèle est le **seul à modifier** les données, il est le garant de ces données.

Vue

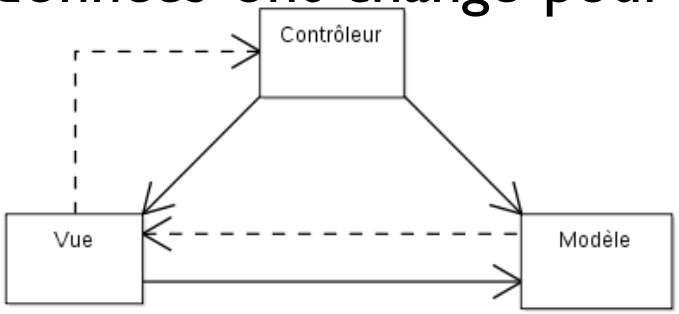
- **Interface** avec laquelle l'utilisateur interagit
- Deux rôles :
  - **Présenter les résultats** renvoyés par le modèle
  - Recevoir les **actions des utilisateurs** (click,, …) et transmettre au contrôleur. (via un événement)



- N'effectue **aucun traitement**, ne fait qu'afficher les données du modèle.

Contrôleur

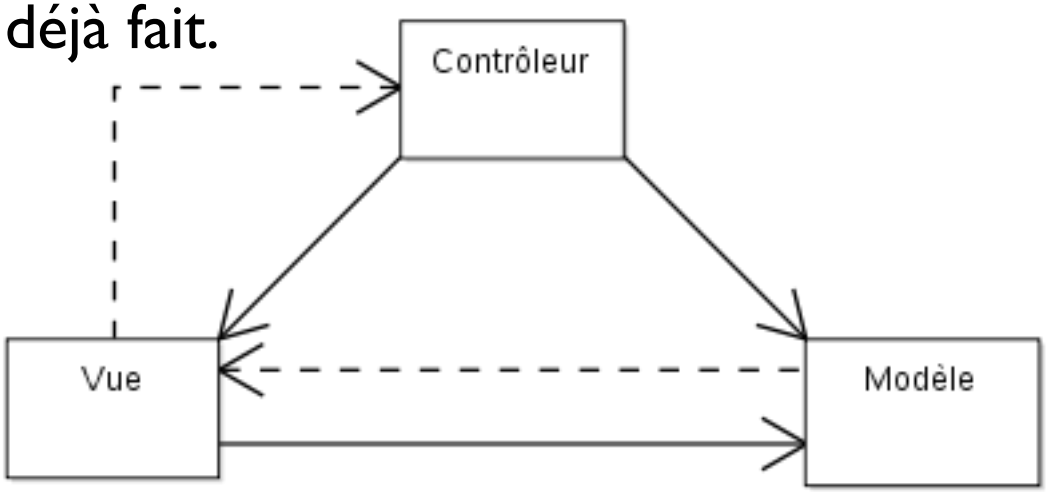
- **Gestion des évements** pour mettre à jour la vue ou le modèle.
- Synchronise la vue et le modèle.
- Reçoit les notifications de la vue, met à jour le modèle au besoin et avertit la vue que les données ont changé pour que celle-ci se mette à jour.



- **Le contrôleur ne modifie aucune donnée.**
- **Le contrôleur est simplement les méthodes de gestion d'événement (EventHandler)**

En résumé

- L'utilisateur clique sur quelque chose (->notification)
- La requête est analysée par le *controleur*
- Le *contrôleur* demande au *modèle* de faire ces changements
- Le *contrôleur* renvoie la *vue* adaptée, si le *modèle* ne l'a pas déjà fait.



Pointillés = notifications
Plein = association

Règles de base

- **Le modèle ne peut pas connaître la vue.**
- **La vue peut connaître le modèle (et l'utilise en lecture seule).**