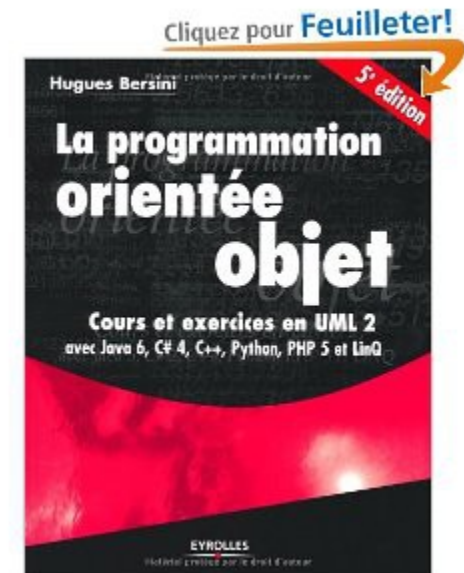


Introduction à l'Orienté Objet:

Hugues Bersini
CoDE-IRIDIA, ULB



Plan

- Intro à l'Orienté Objet
- Intro à l'UML
- Utilisation de Java/Python/UML
- UML et Modélisation Objet
- Stockage des objets
- Objets distribués

Concepts de programmation considérés connus

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

Introduction à l'OO

Concepts de base

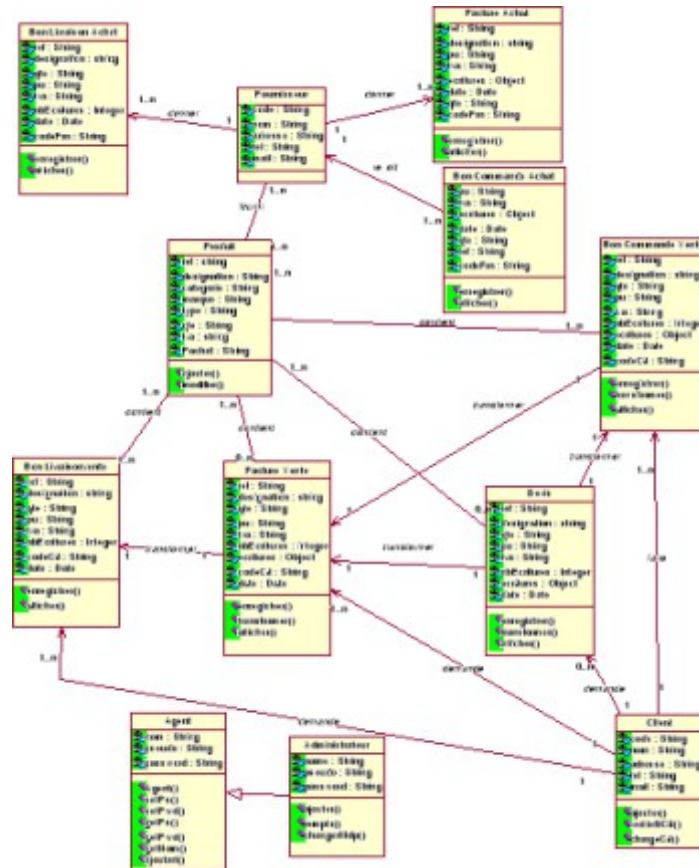
Les raisons du succès de l'OO

- 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
- **Les applications distribuées** sur le Net. Une extension facilitée par rapport aux applications locales: RMI, CORBA, DCOM, Jini, JXTA, .Net

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.

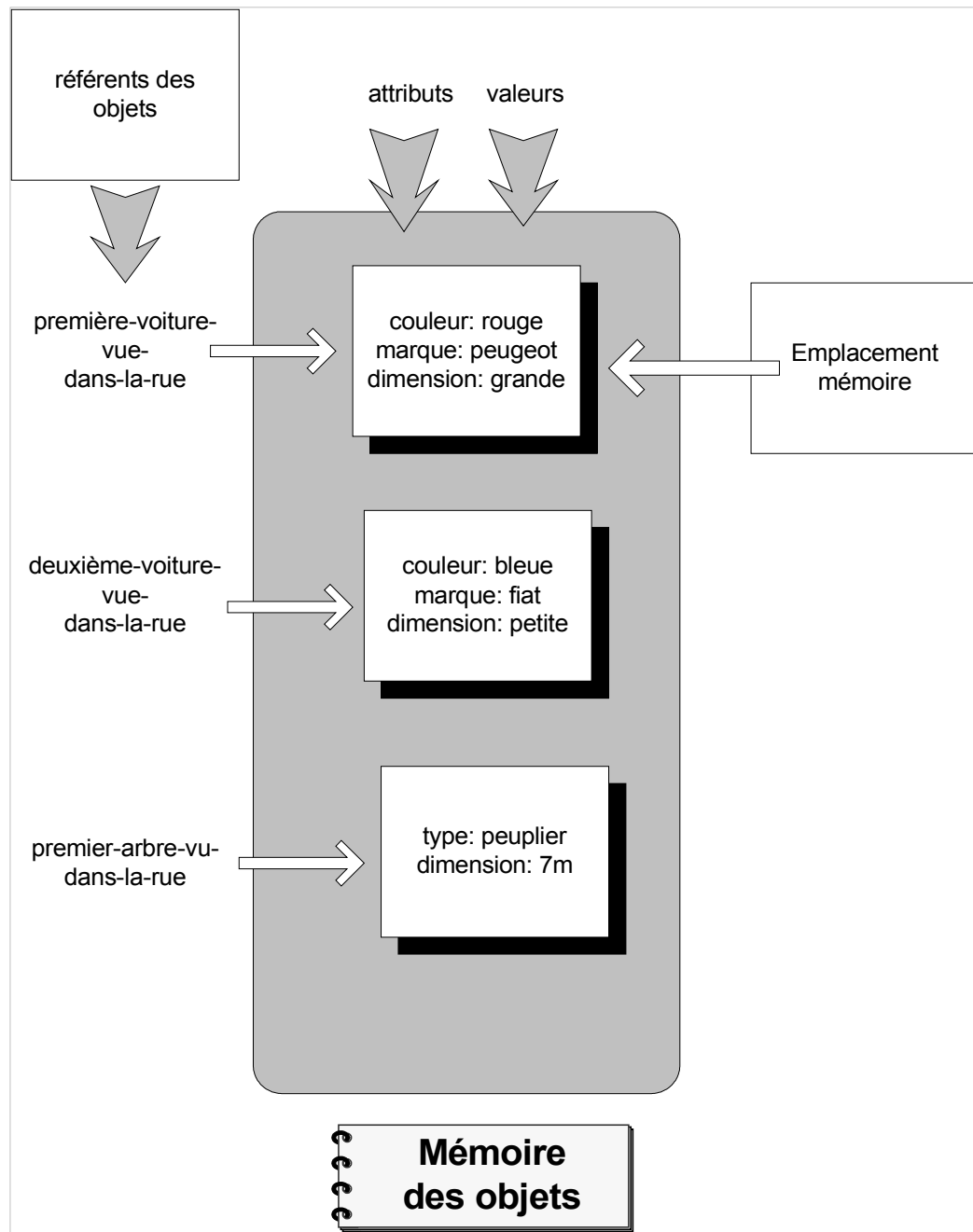


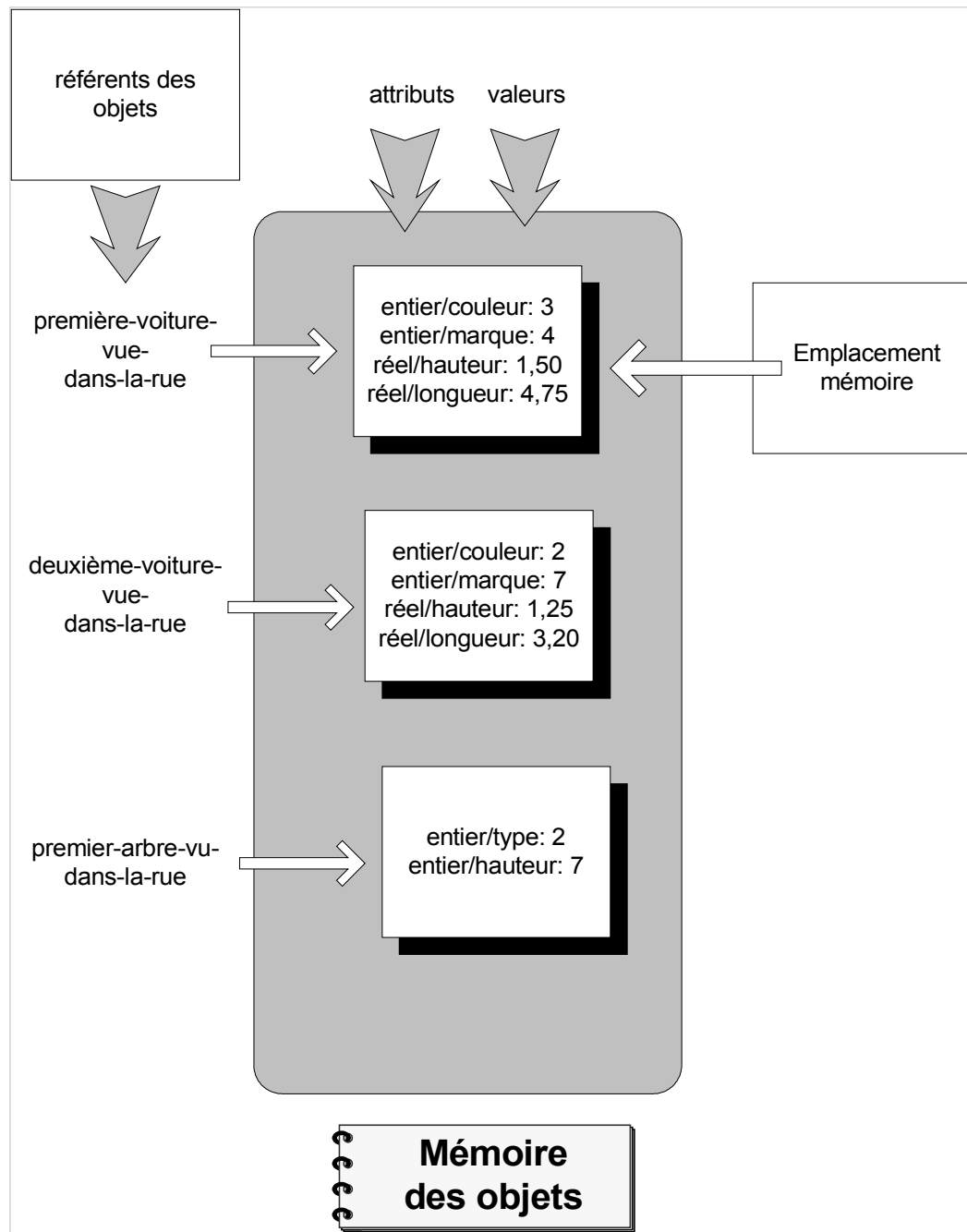
Classes et Objets

- 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



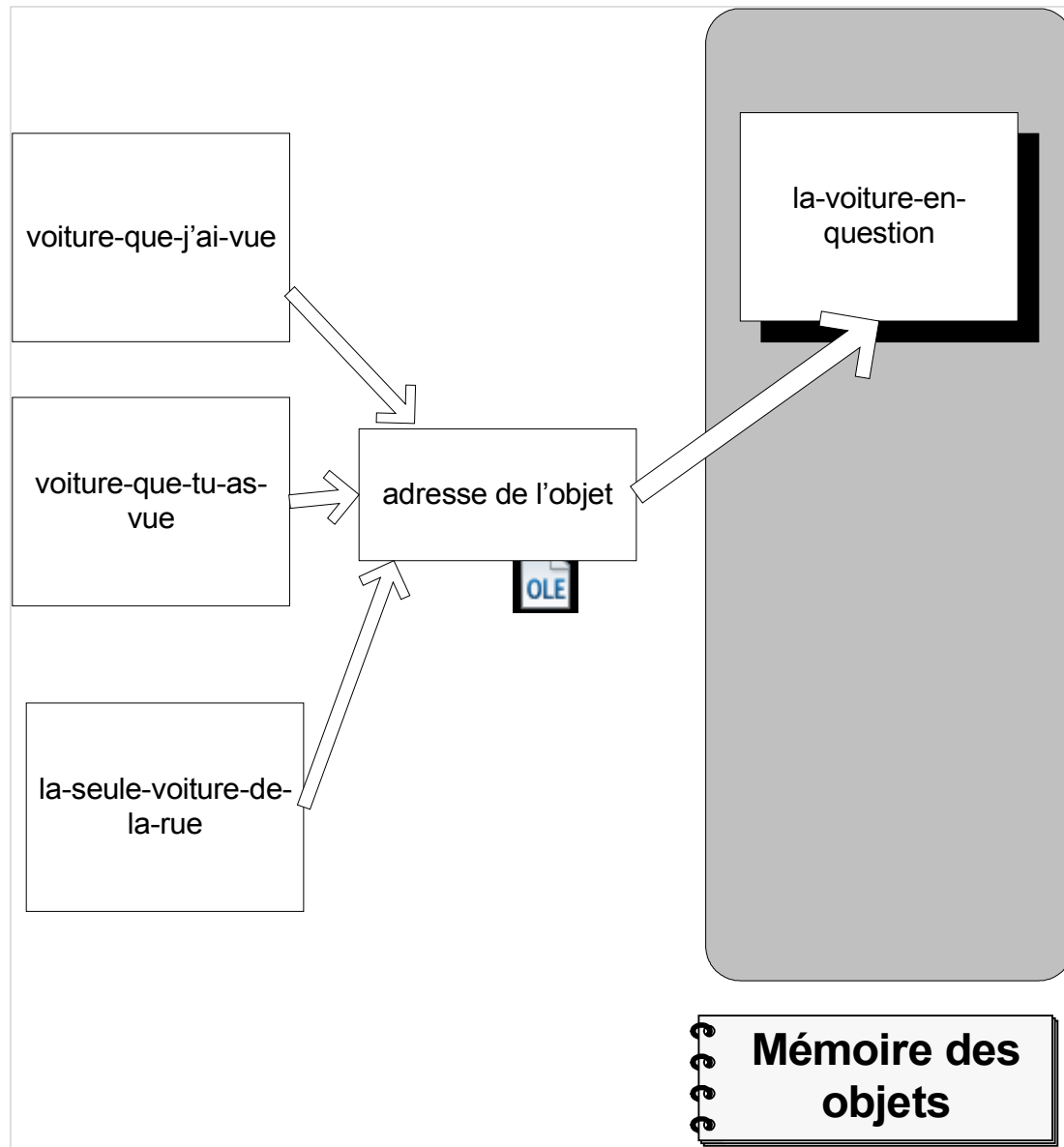


Marque	Modele	Serie	Numero
Renault	18	RL	4698 SJ 45
Renault	Kangoo	RL	4568 HD 16
Renault	Kangoo	RL	6576 VE 38
Peugeot	106	KID	7845 ZS 83
Peugeot	309	chorus	7647 ABY 82
Ford	Escort	Match	8562 EV 23

Table d'une base de données relationnelles

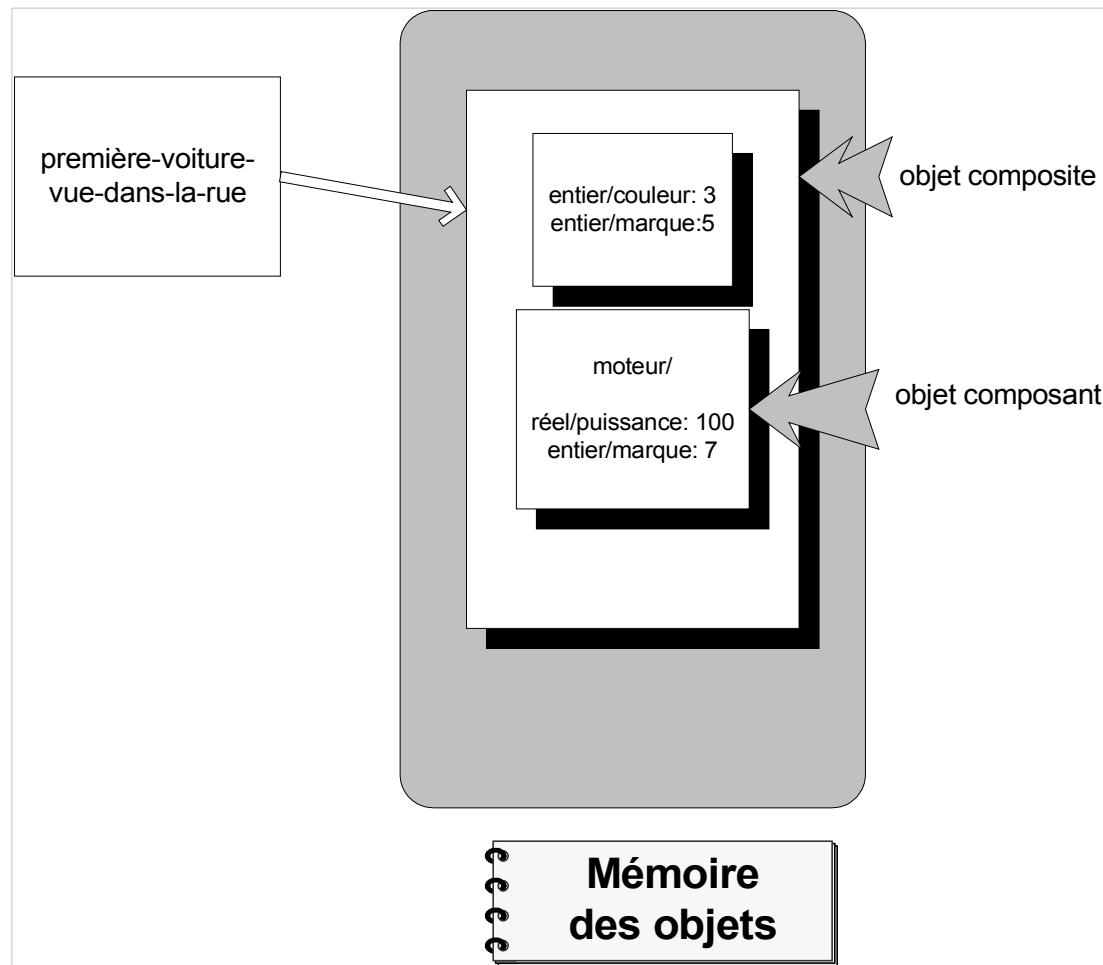
Le référent d'un objet

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



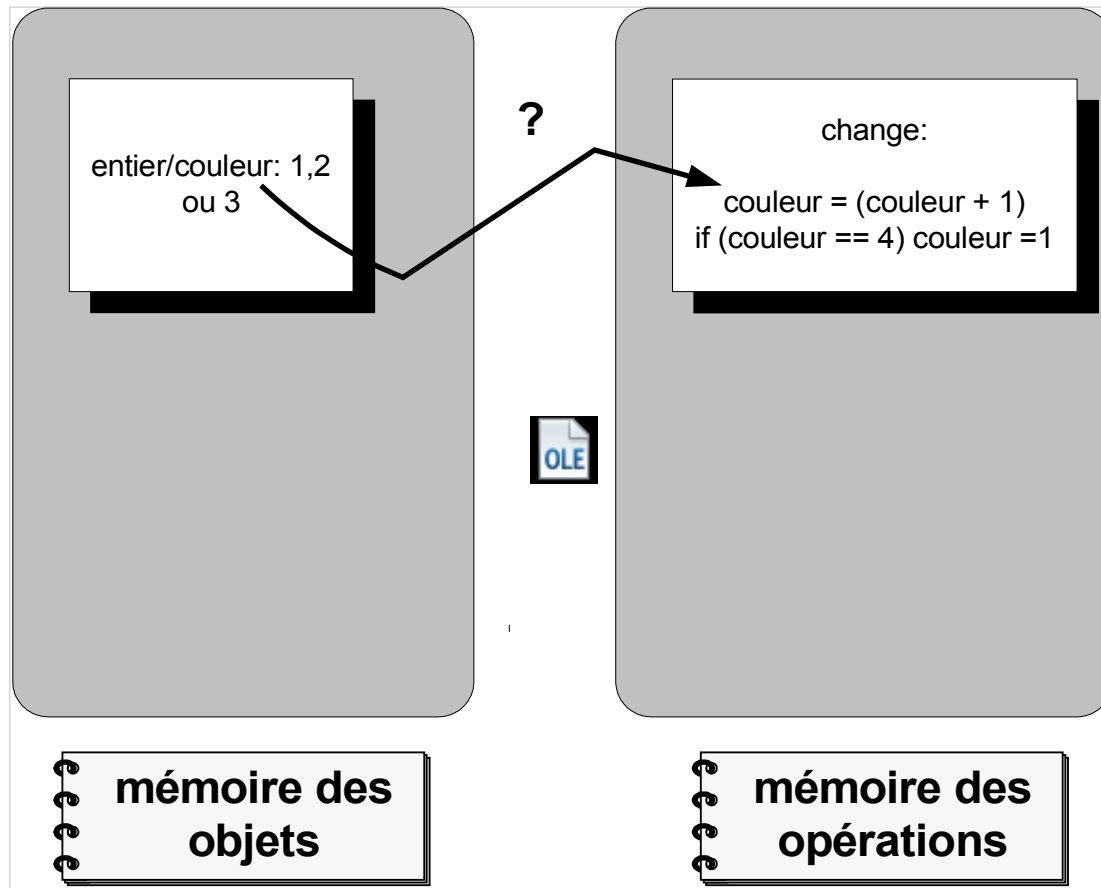
L'objet dans sa version passive

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

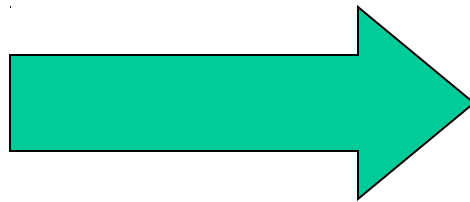


L'objet dans sa version active

- 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



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 ;  
    }  
}
```

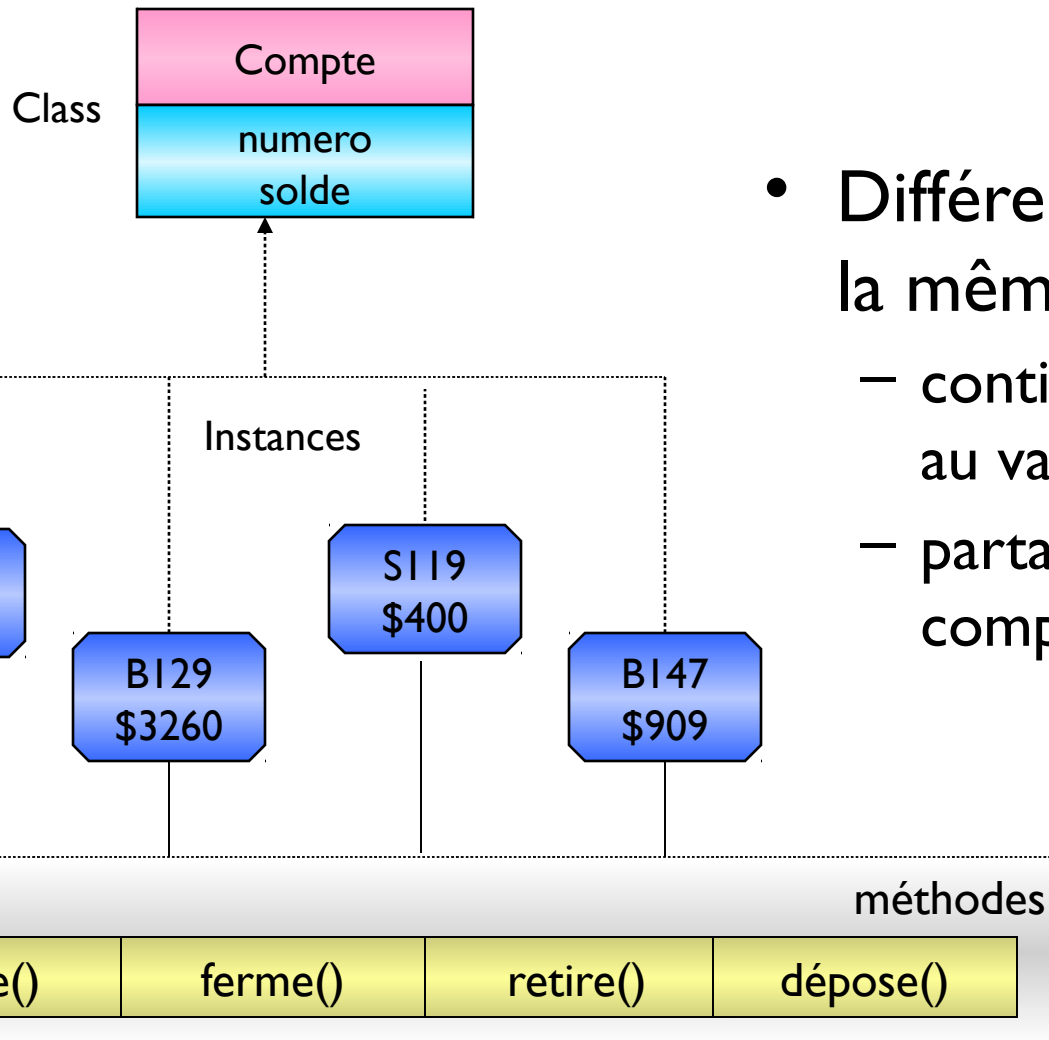
Sur quel objet précis s'exerce la méthode

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

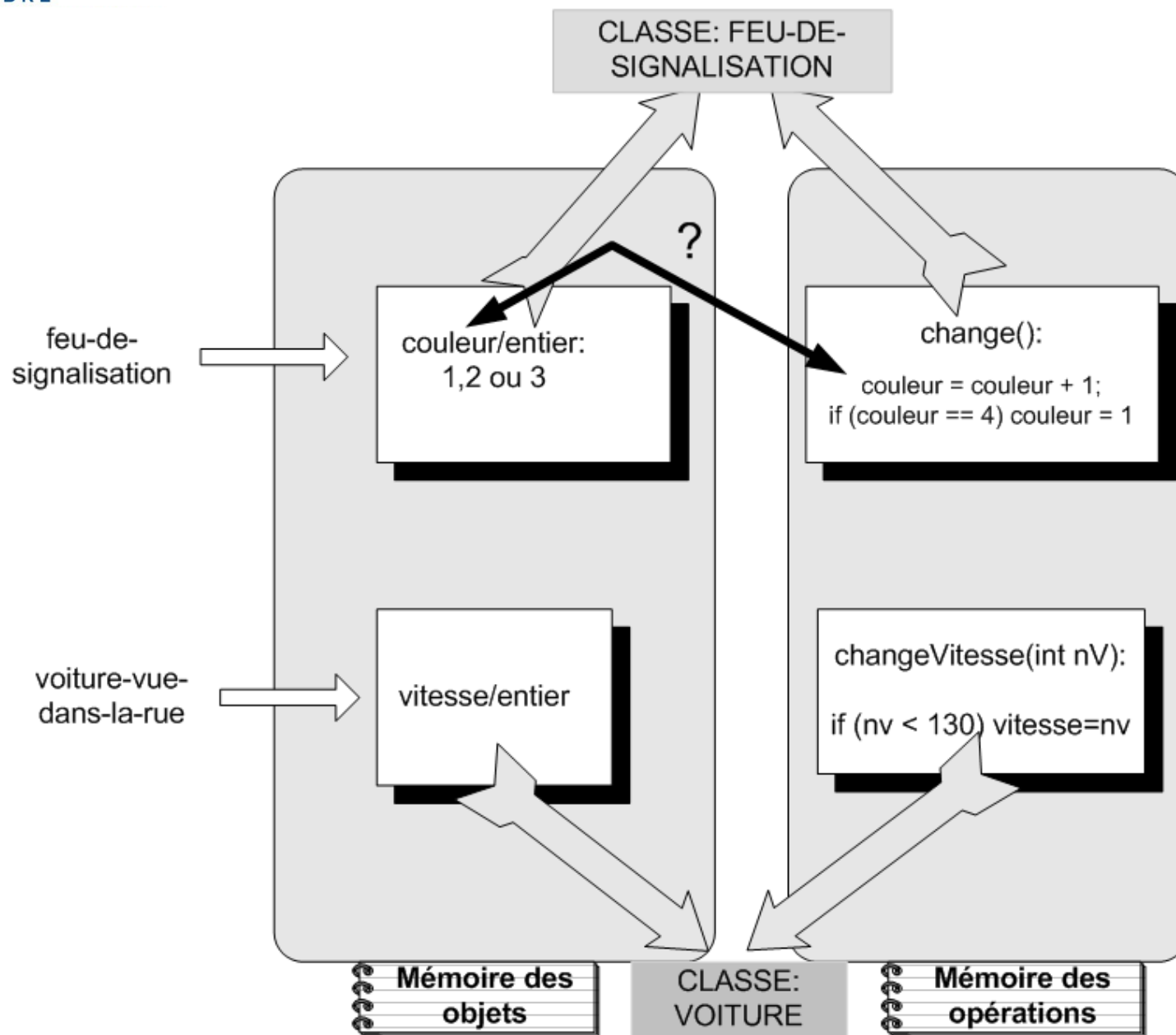
On lie la méthode $f(x)$ à l'objet « a » par l'instruction:

$a.f(x)$

Les objets ne diffèrent que par leurs attributs

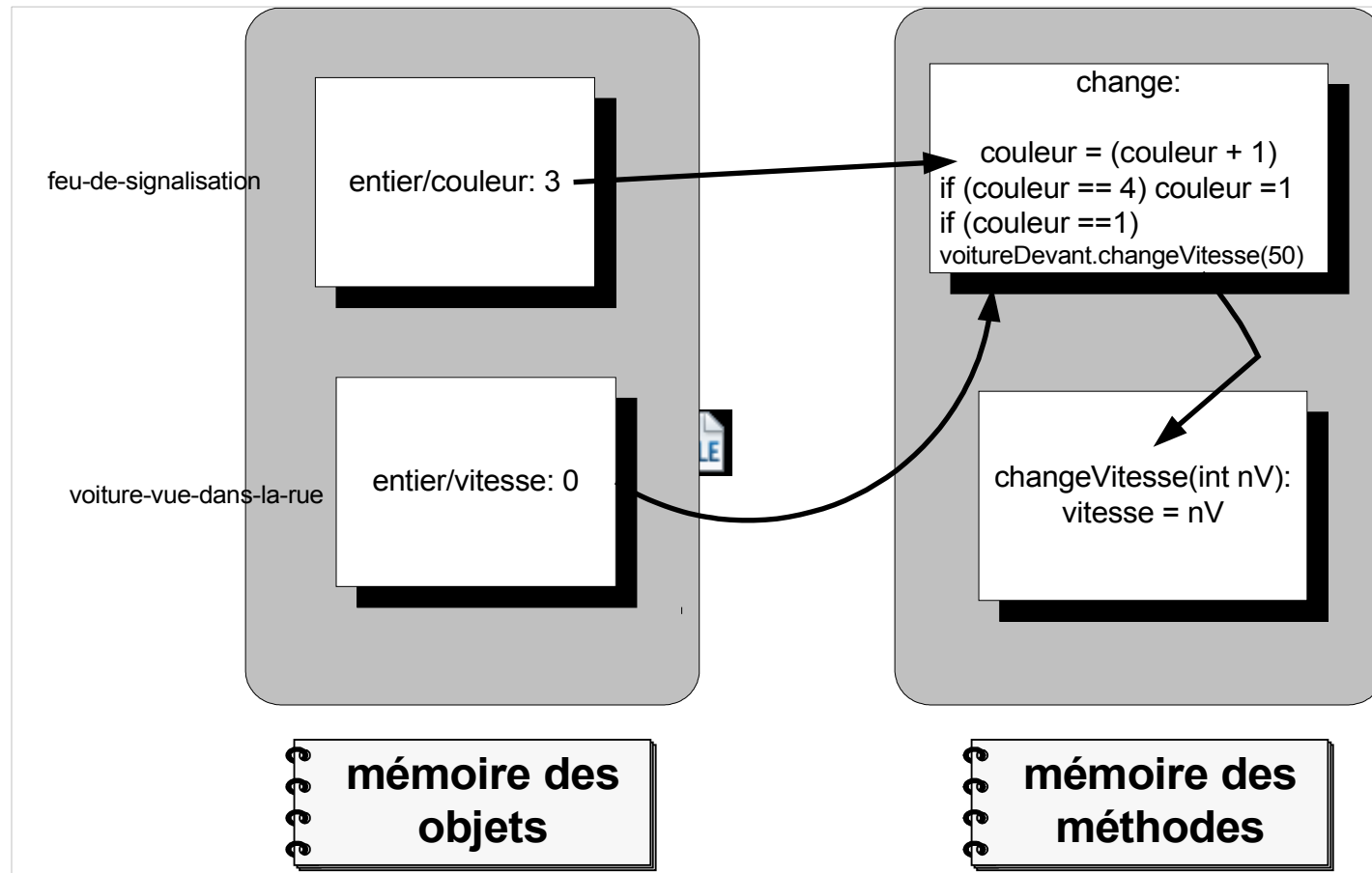


- Différentes instances de la même classe
 - contiennent des attributs à valeur différente
 - partagent un même comportement



Des objets en interaction

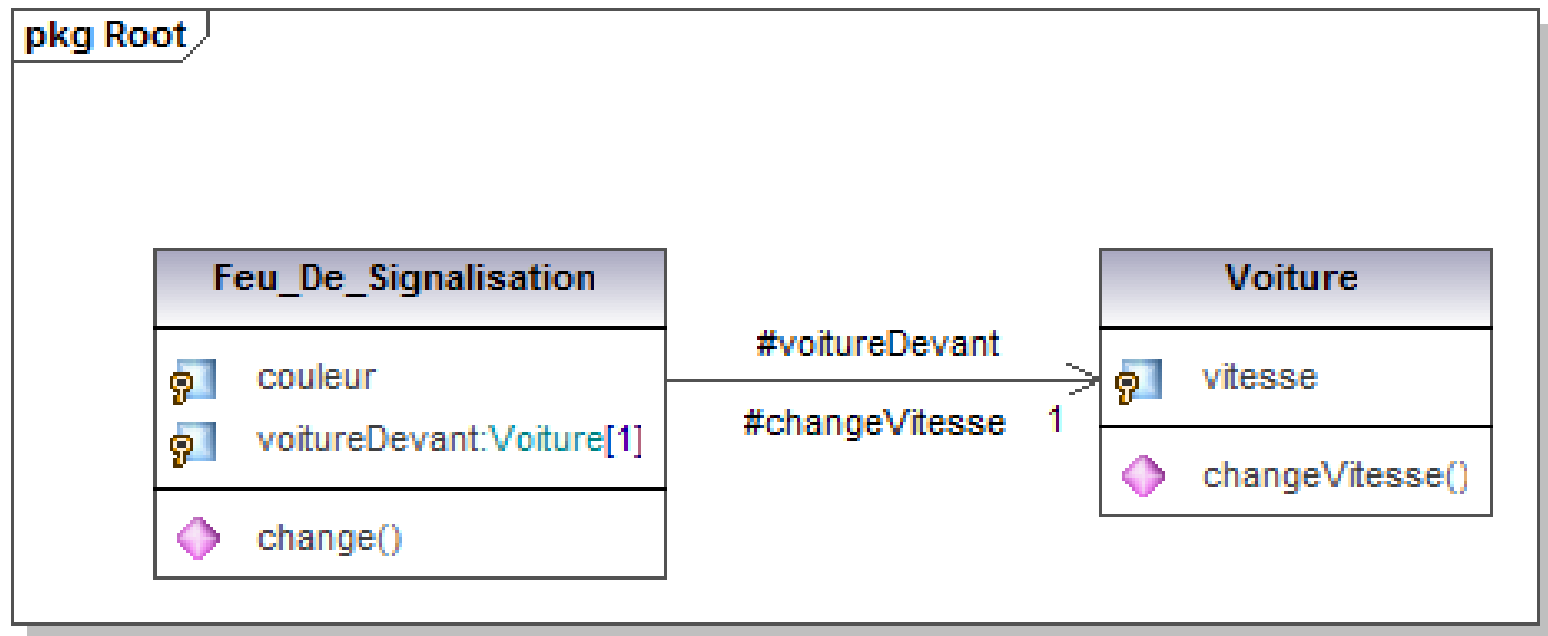
- La base de l' informatique OO:
communication entre les objets
- Supposons que le feu fasse ralentir ou
accélérer la voiture



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

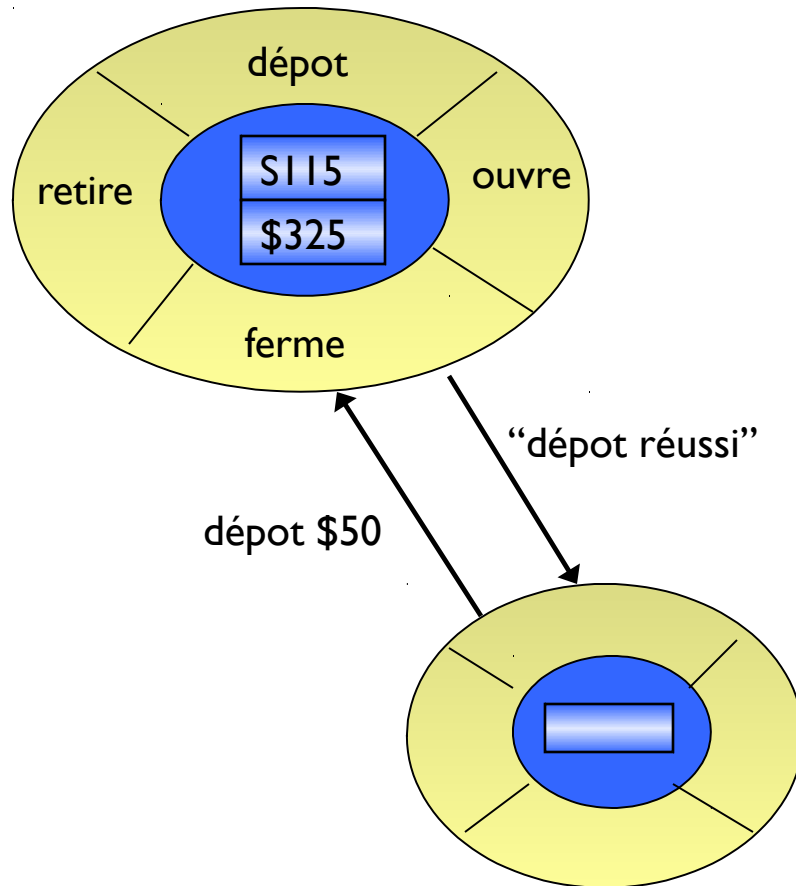
```
class Feu-de-signalisation {  
    int couleur ;  
    Voiture voitureDevant;  
    change() {  
        couleur = couleur + 1 ;  
        if (couleur ==4) couleur = 1 ;  
        if (couleur ==1)  
voitureDevant.changeVitesse(50) ;  
    }  
}
```

UML envoi de message



Appel de méthode à distance = “envoi de message”

Le compte objet destinataire

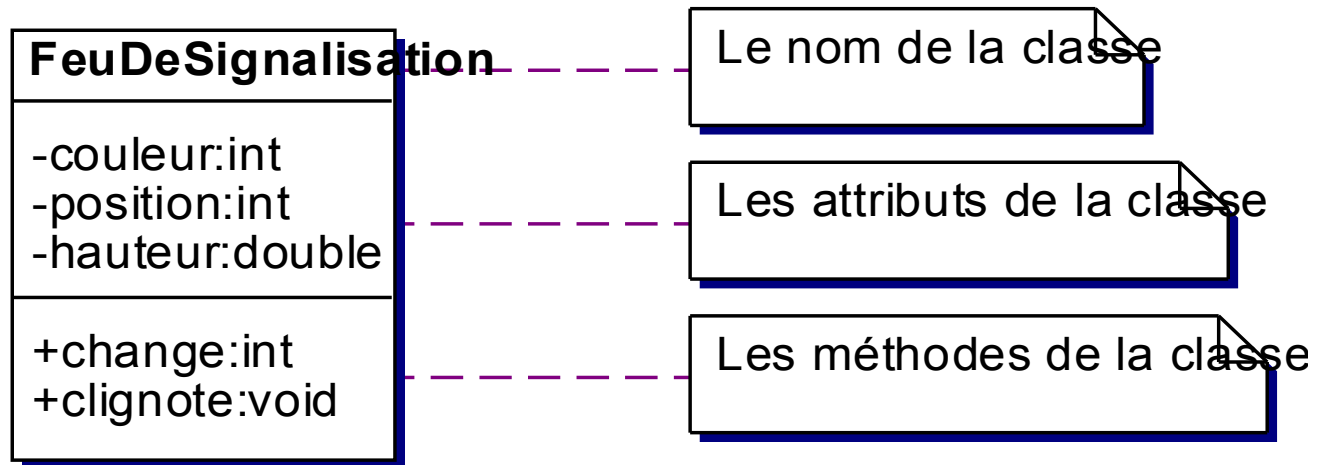


Le client objet expéditeur

- 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 */  
}
```

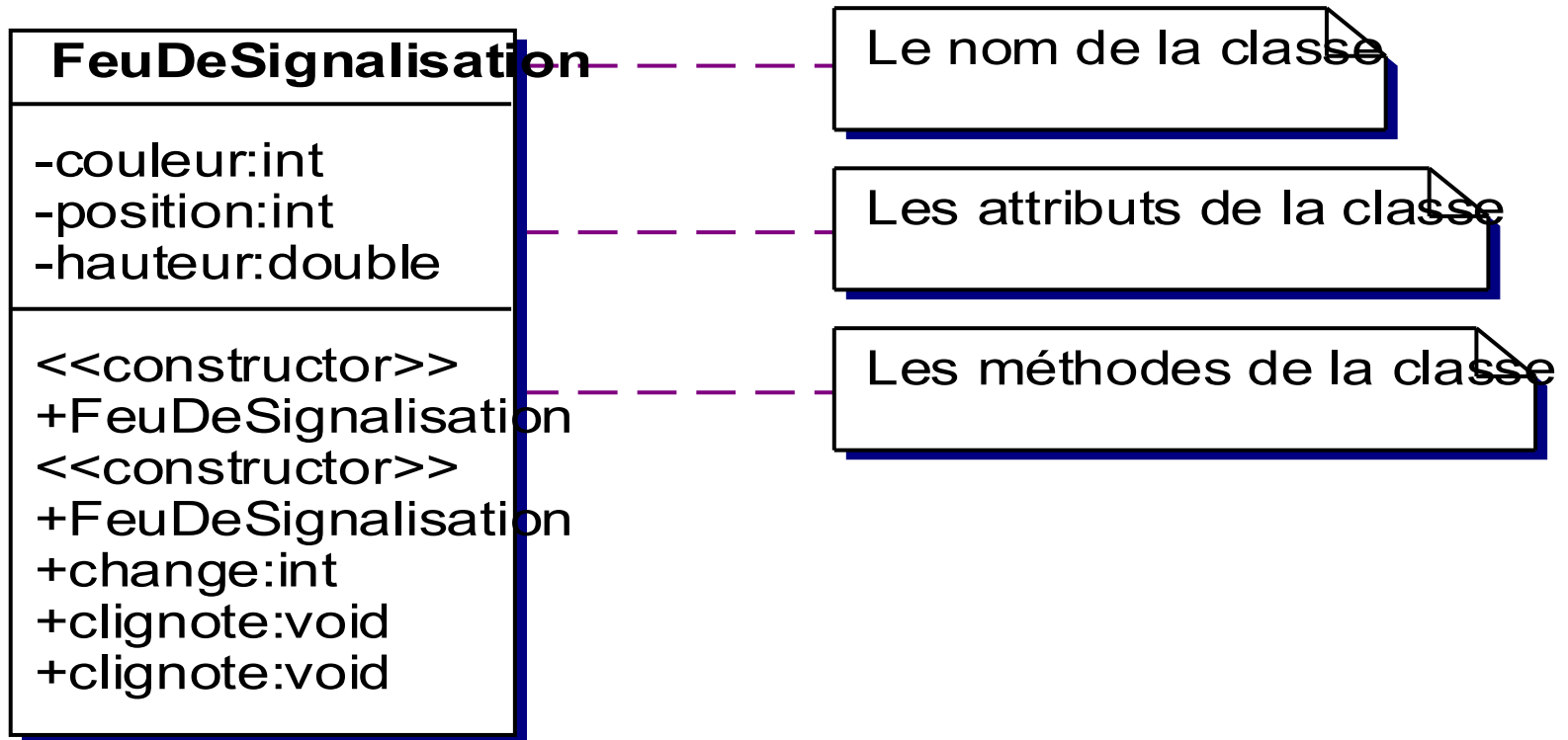

Signature et surcharge de méthodes

```
class FeuDeSignalisation {  
    void clignote() {  
        System.out.println("premiere maniere 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



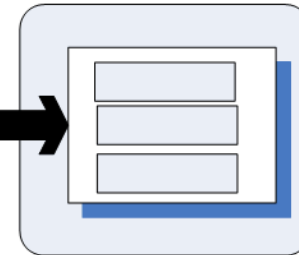
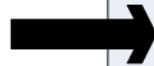
```
FeuDeSignalisation(int positionInit, double  
hauteurInit) {  
    /* pas de retour pour le constructeur */  
    position = positionInit ;  
    hauteur = hauteurInit ;  
    couleur = 1 ;  
}
```

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

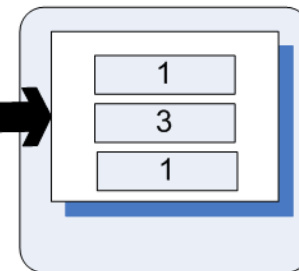
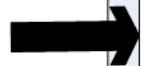
FeuDeSignalisation unNouveauFeu



unNouveauFeu = new
FeuDeSignalisation



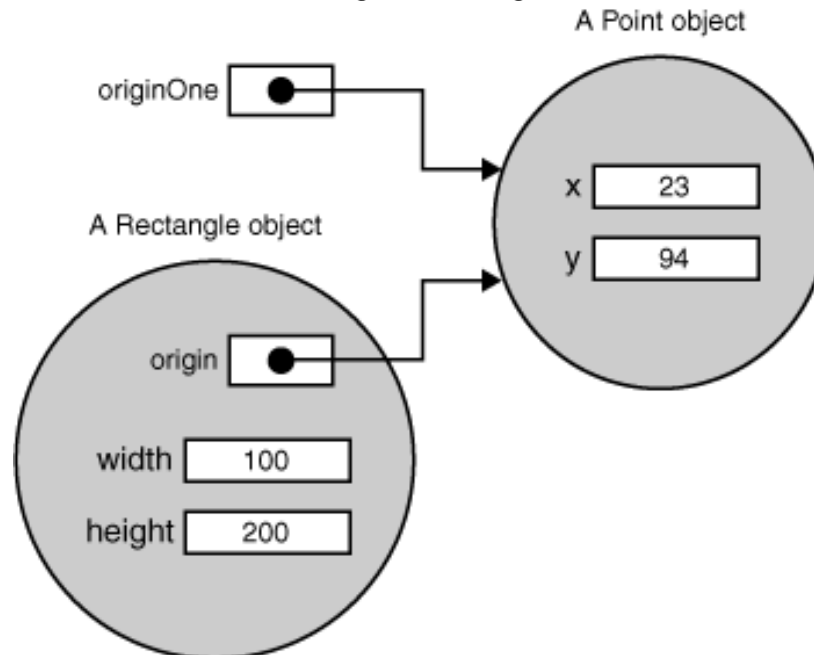
unNouveauFeu = new
FeuDeSignalisation(1,3)



Objets et références

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

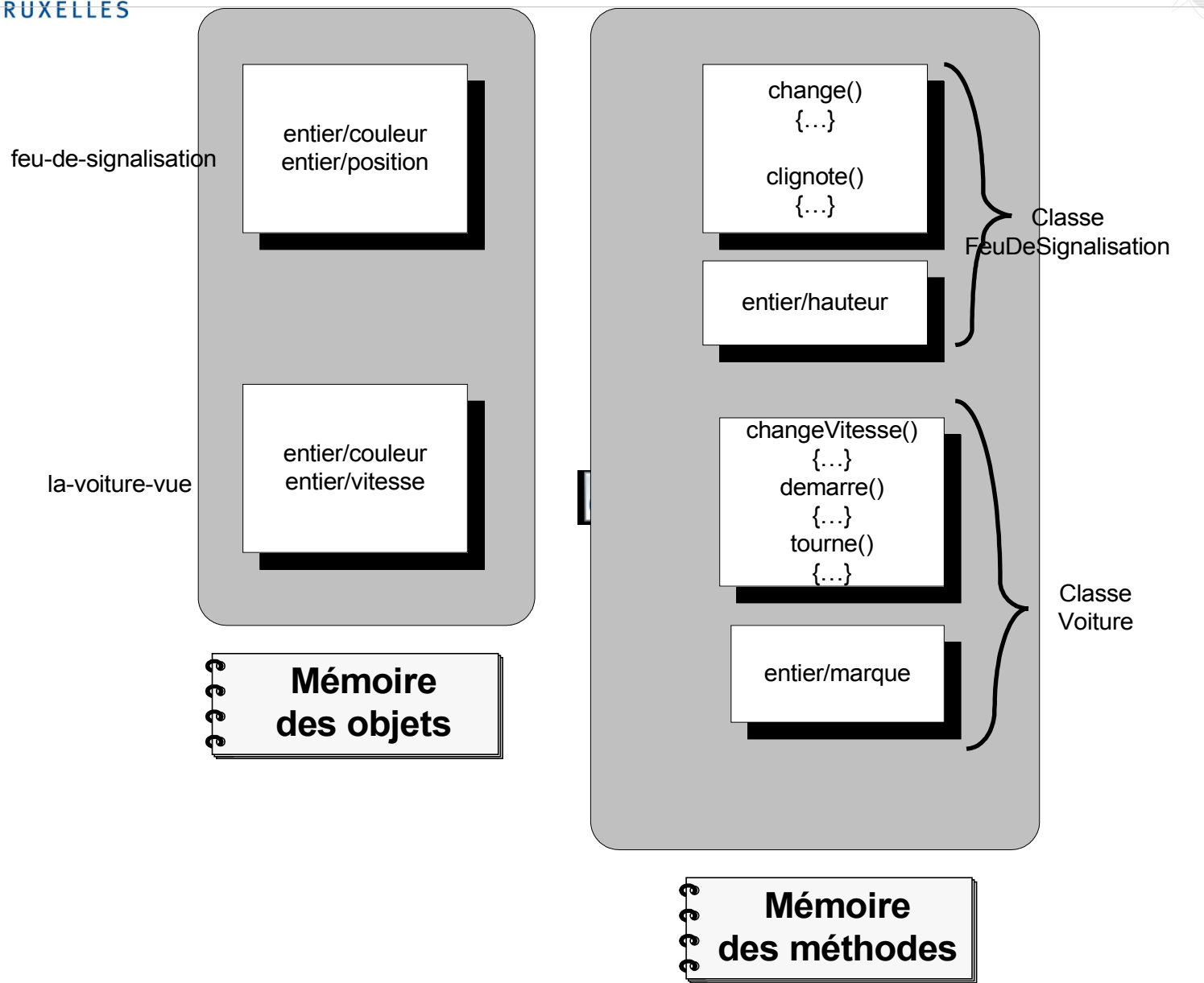
```
Point originOne = new Point(23, 94);  
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```



- 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.



Rappels

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

Exemples Python et Java

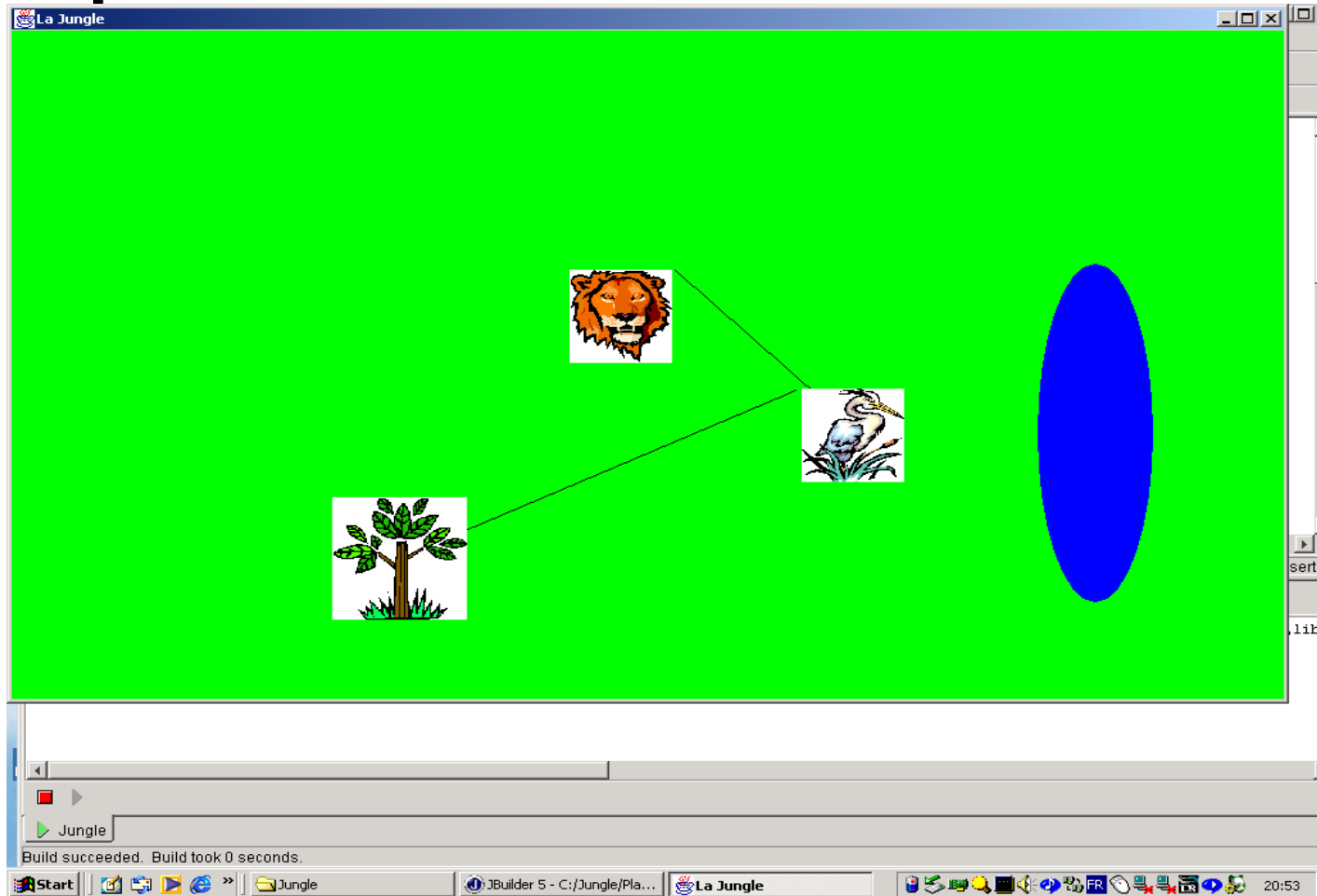
- Voir livre OO, chapitre 2
- Exemple des feux de signalisation.
- Introduction à Java

Classe = mécanisme de modularisation

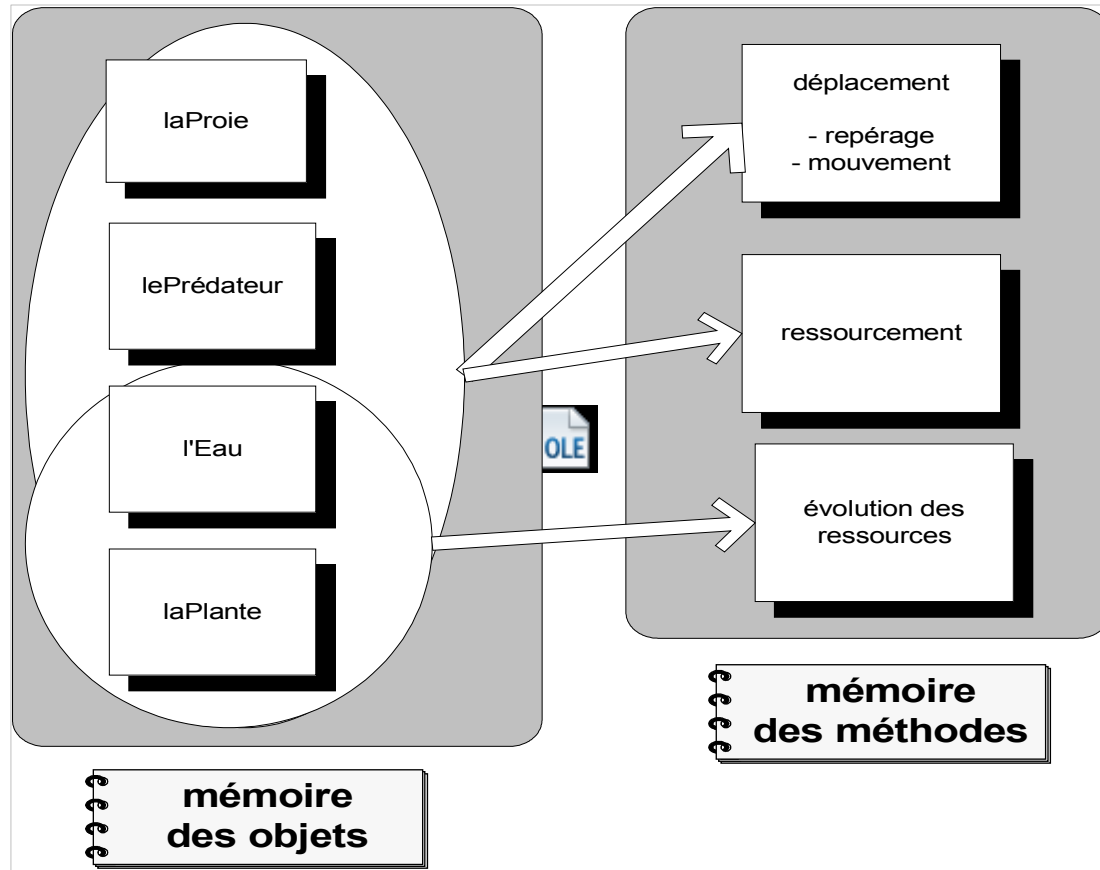
- Classes = fichiers
- Assemblages = répertoires
- La classe = le type et le module à la fois.

Comparaison procédural / OO

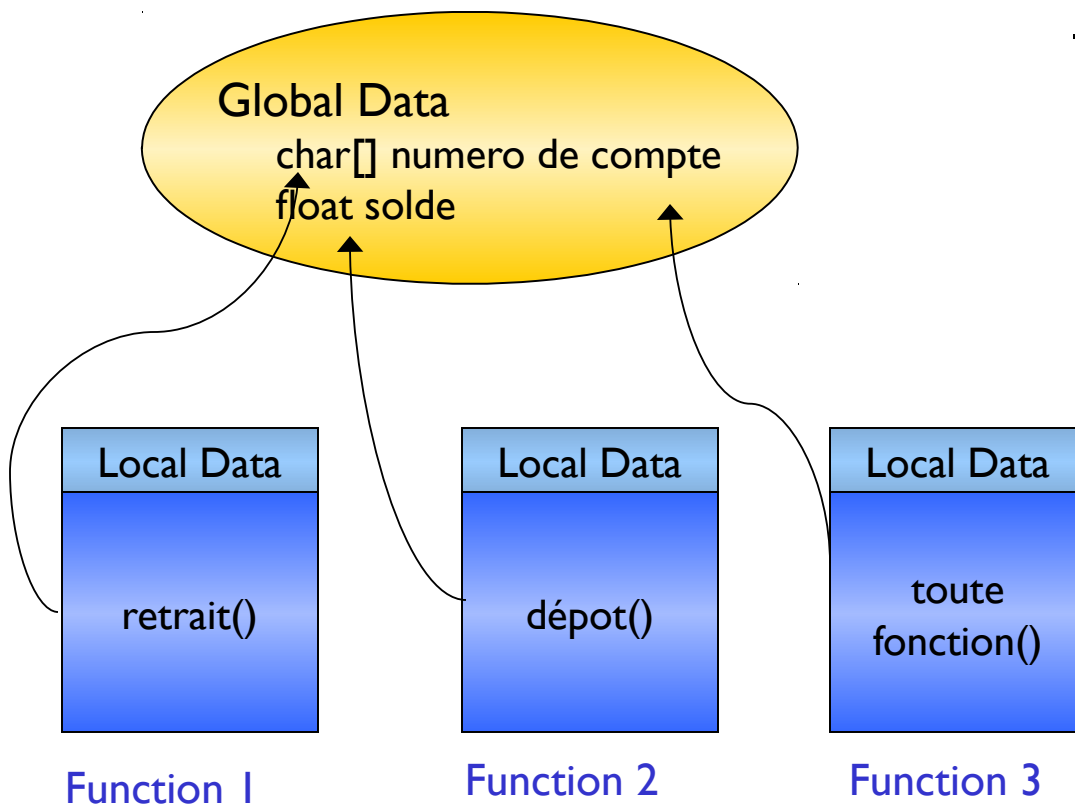
Du procédural à l'OO



Conception procédurale

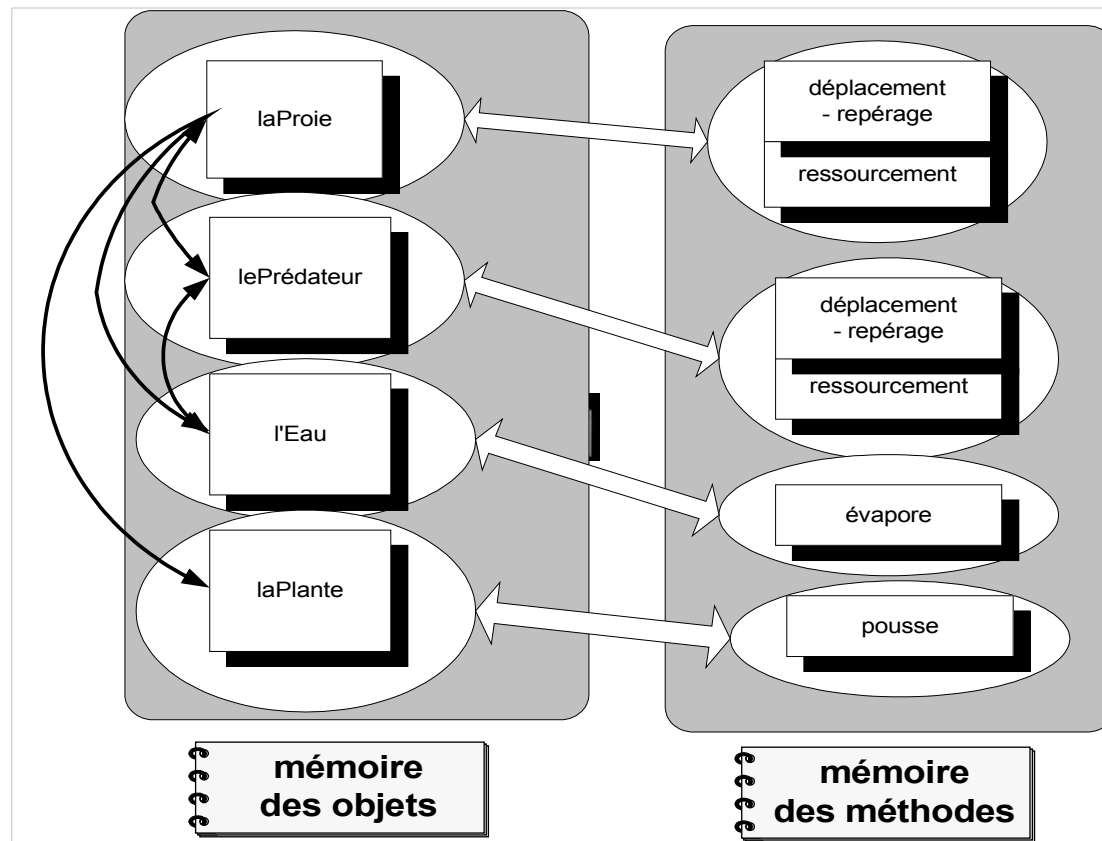


Problème de la programmation procédurale



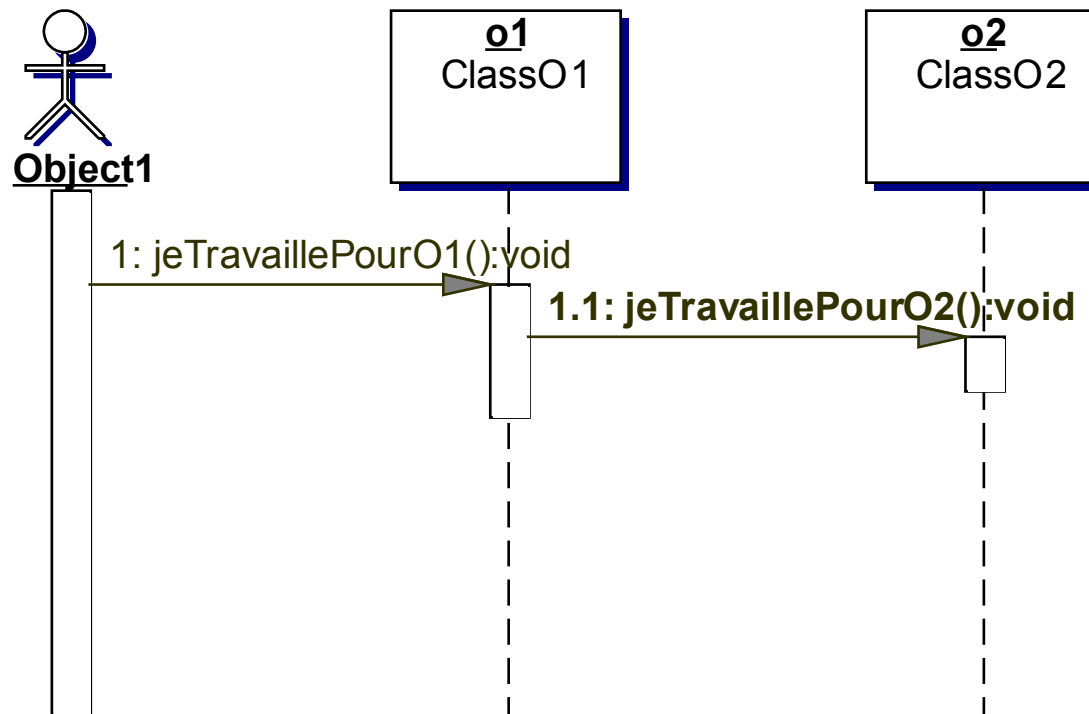
- Accès sans restriction
 - Plusieurs modules fonctionnels auront accès aux mêmes données créant une grande sensibilité aux changements

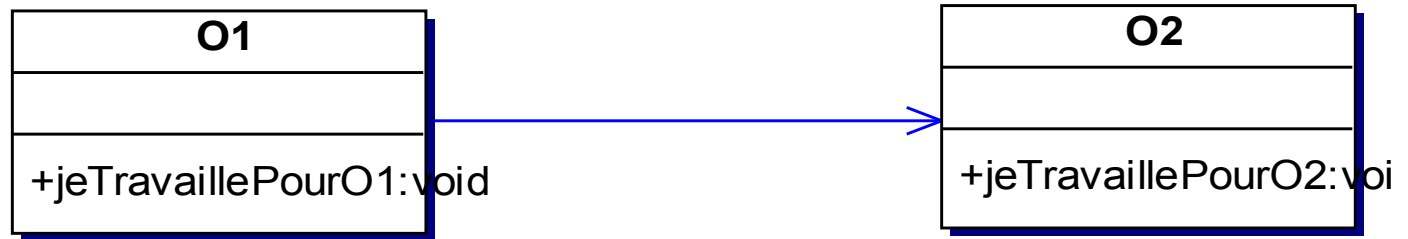
Conception OO



Envoi de messages et diagramme UML de séquence

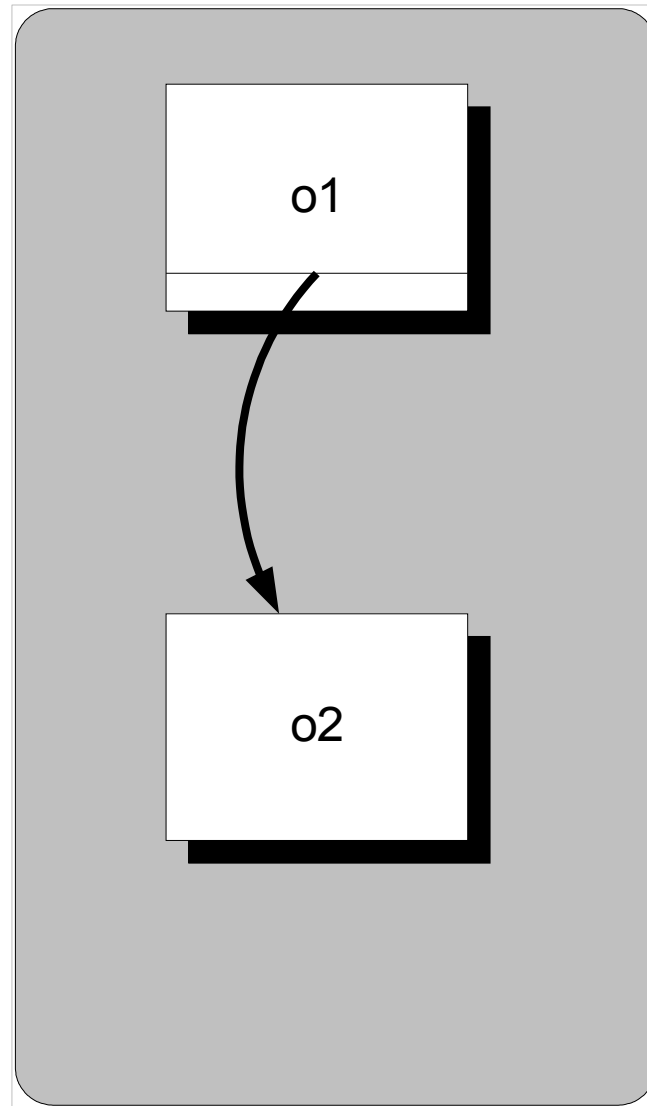
Les objets parlent aux objets



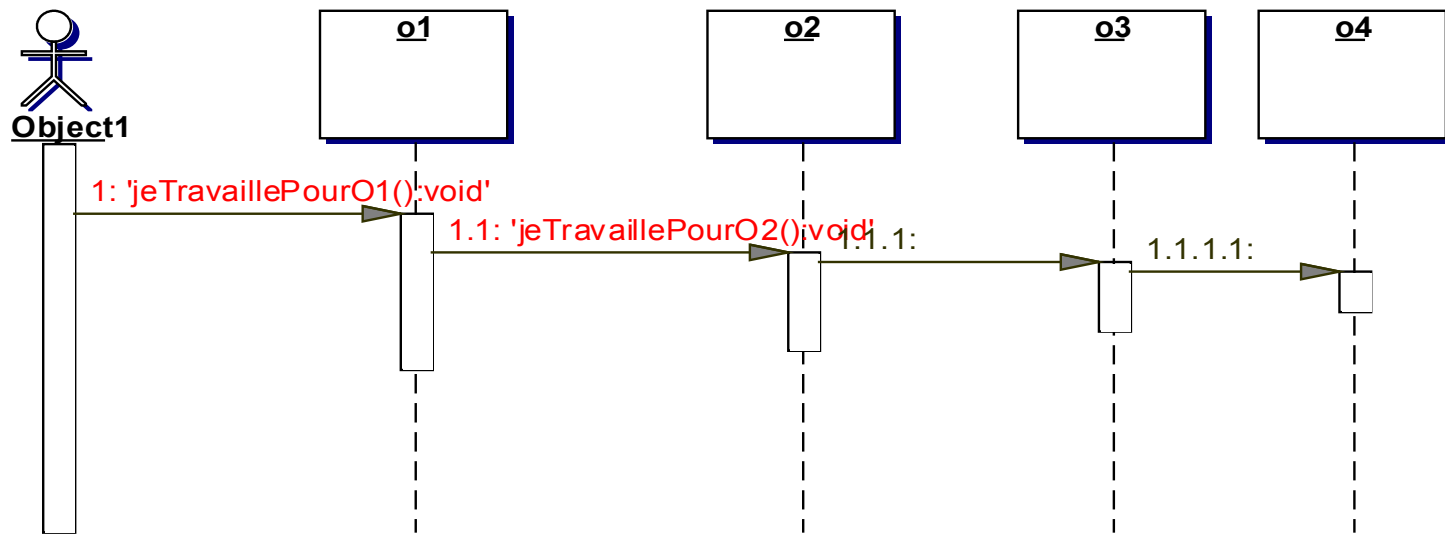


```

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



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

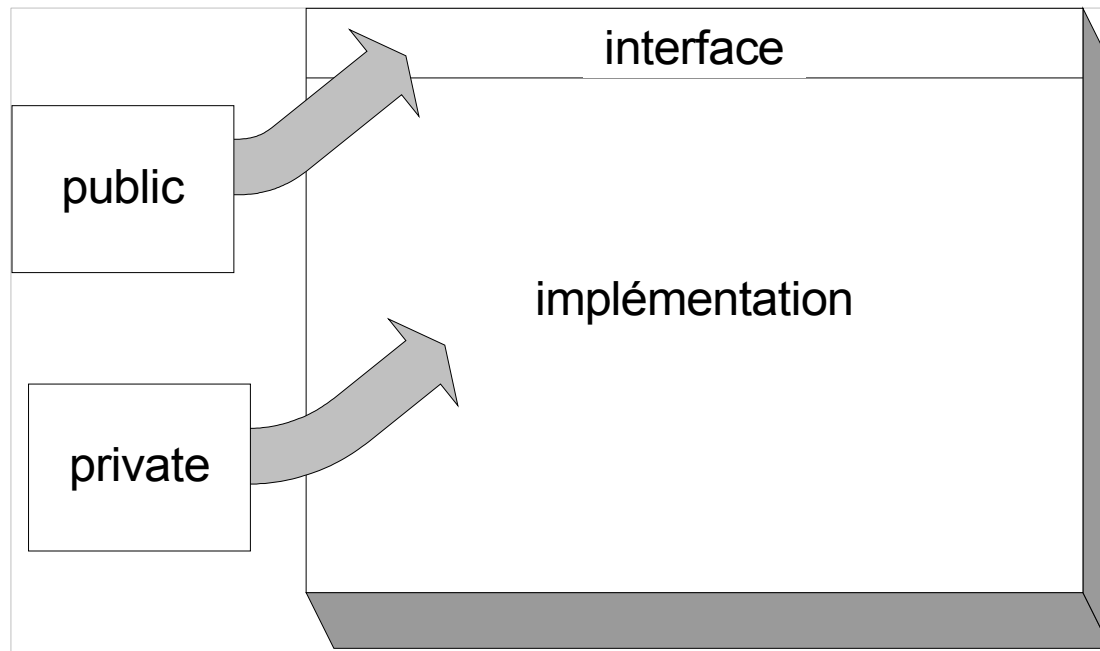
```
class Feu-de-signalisation {  
    private int couleur;  
    private Voiture voitureDevant;  
  
    public void changeCouleur(int nouvelleCouleur)  
    {  
        if (nouvelleCouleur >= 1) && (nouvelleCouleur  
<=3) /* intégrité assurée */  
            couleur = nouvelleCouleur ;  
    }  
}
```

```
class Voiture {  
    private int vitesse ;  
    public int changeVitesse(int  
nouvelleVitesse) {  
        if (nouvelleVitesse >= 0) &&  
(nouvelleVitesse <=130) /* intégrité assurée  
*/  
            vitesse = nouvelleVitesse ;  
        return vitesse ;  
    }  
}
```

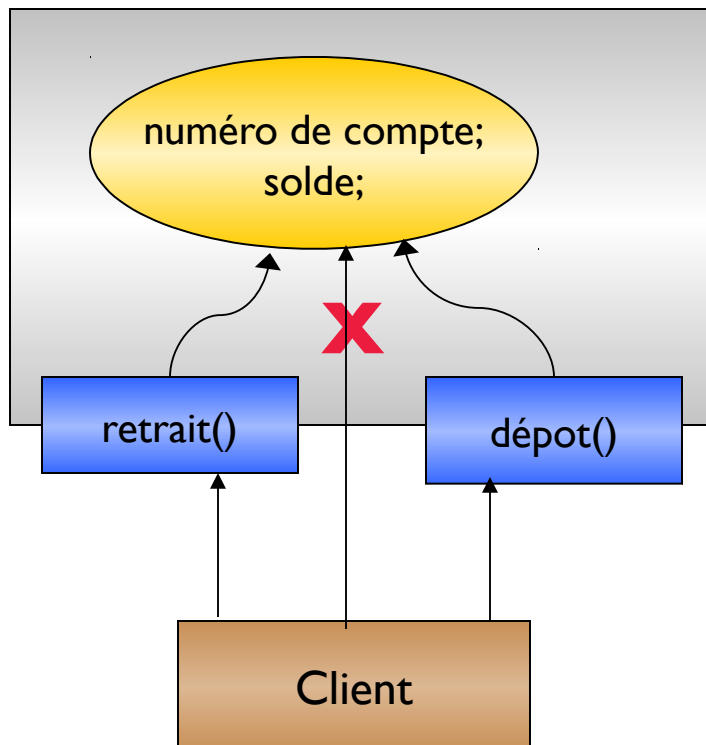
```
class Acheteur {  
    private Voiture voitureInteressante ;  
  
    public int négociePrix() {  
        int prixPropose = 0 ;  
        if (voitureInteressante.dateFabrication <  
19970101) /* accès possible à l'attribut date */  
            prixPropose =  
voitureInteressante.getPrixDeBase()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

Encapsulation des méthodes

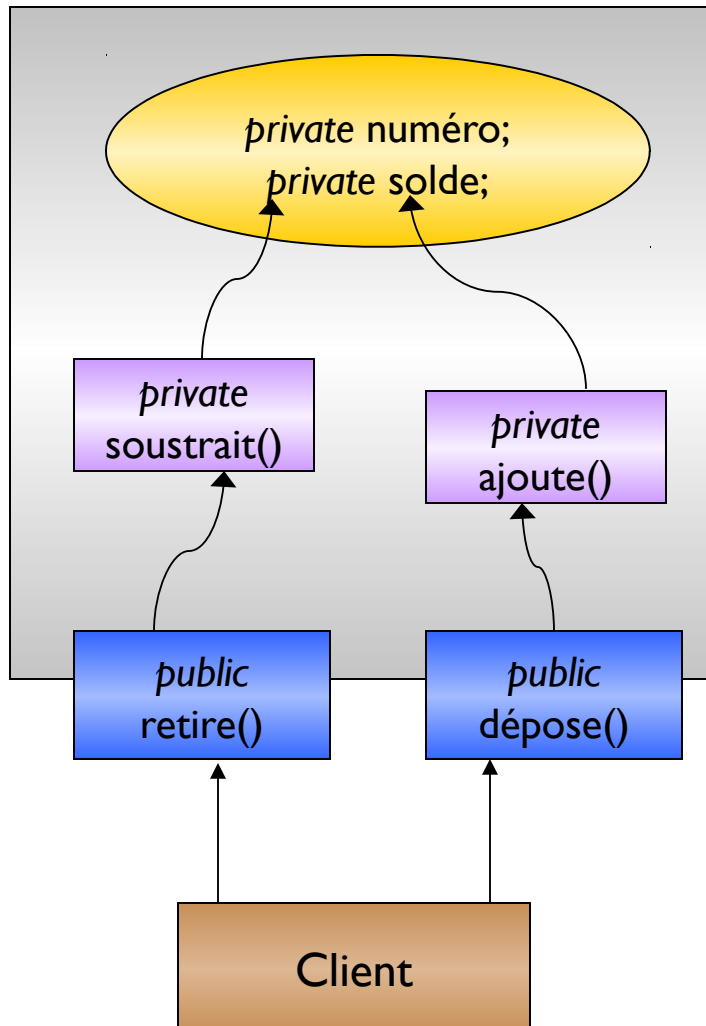
- Stabilisation des développements
- Dépendances à l'exécution mais indépendance au développement
- Interface et implémentation
- Signature d'une classe = son interface = liste des signatures des méthodes disponibles
- **Carte de visite de l'objet**



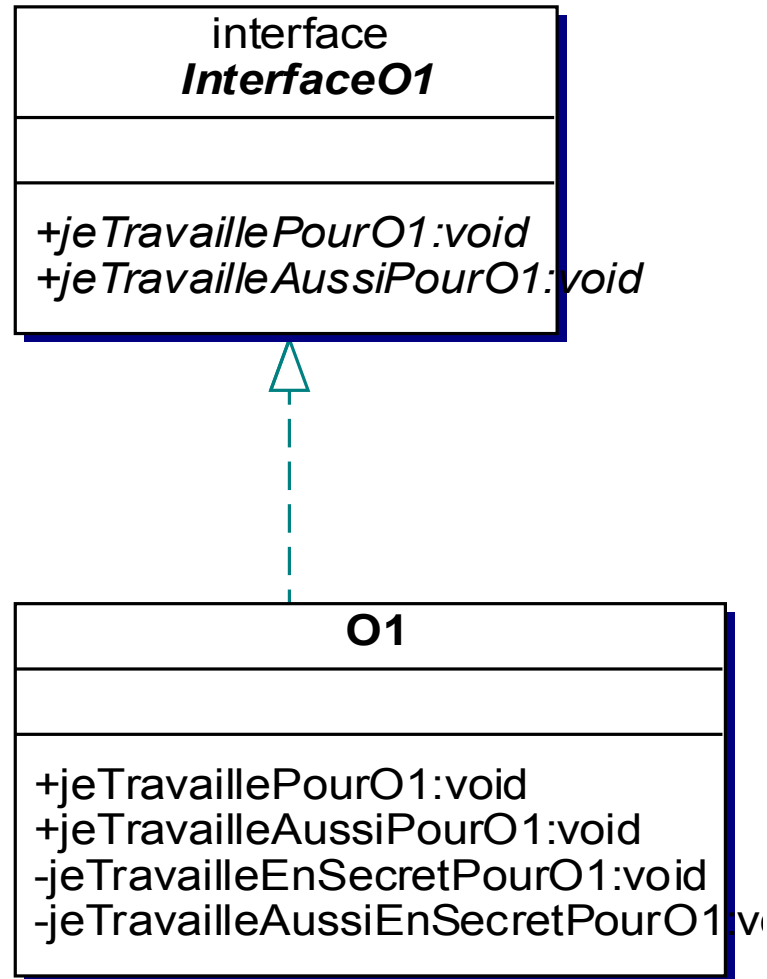
Il faut forcer l'encapsulation



- 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



Exemples Java/Python

- Livre OO
- Envoi de message entre le feu et la voiture

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**.

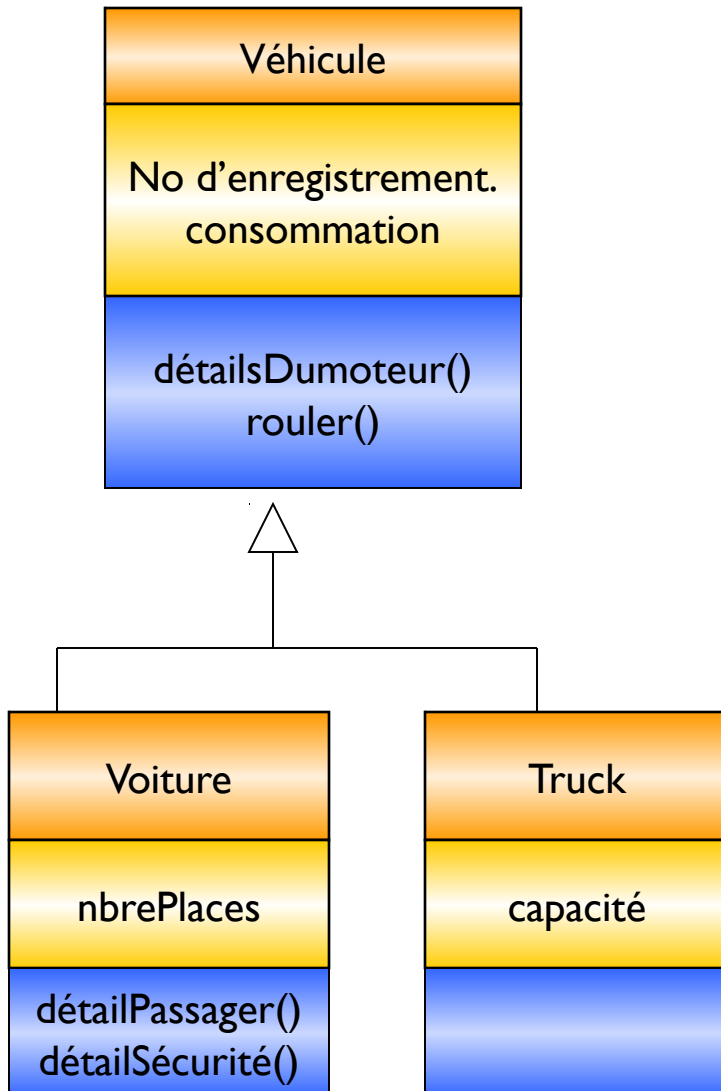
Héritage

- 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

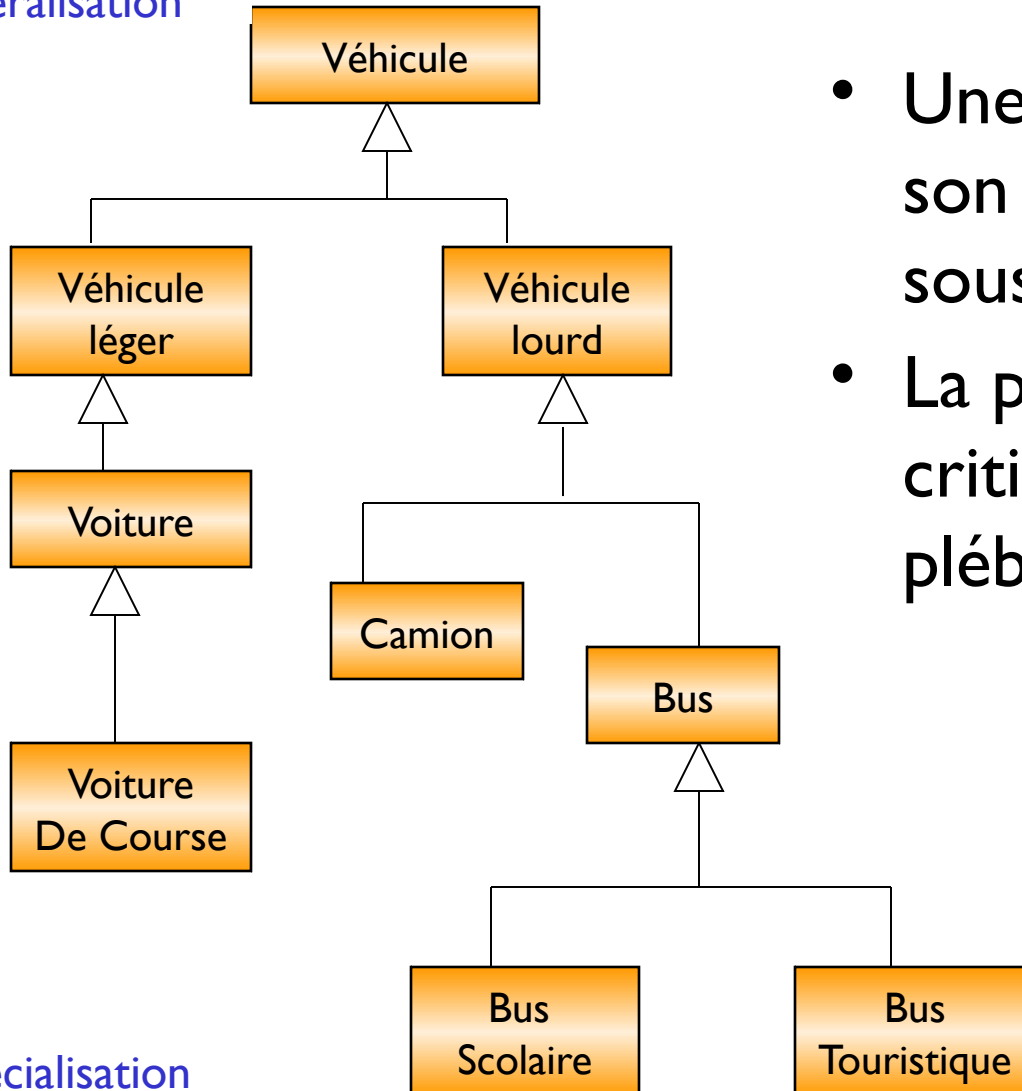
Super classes et Sous classes



- 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

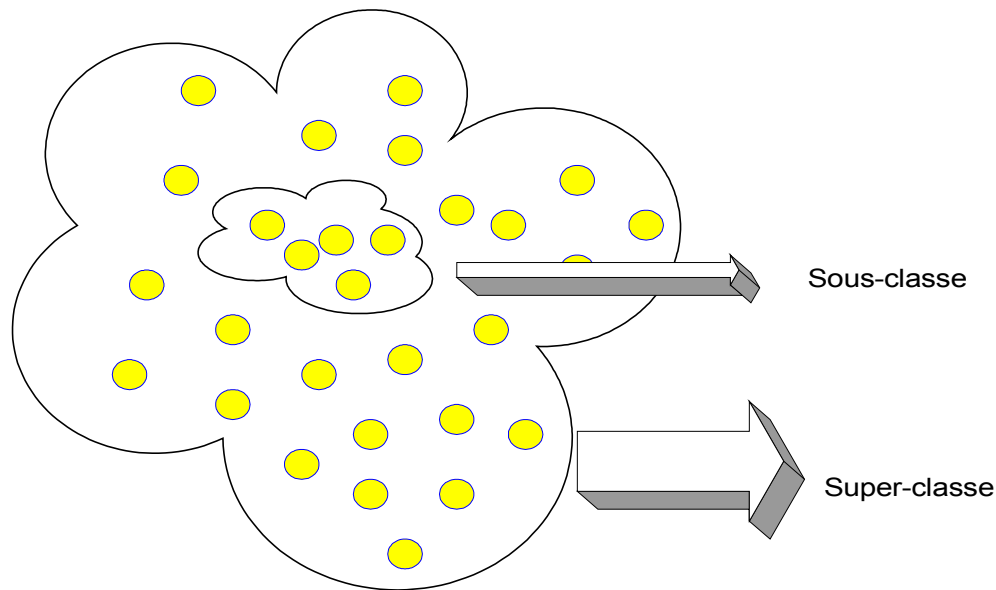
Généralisation

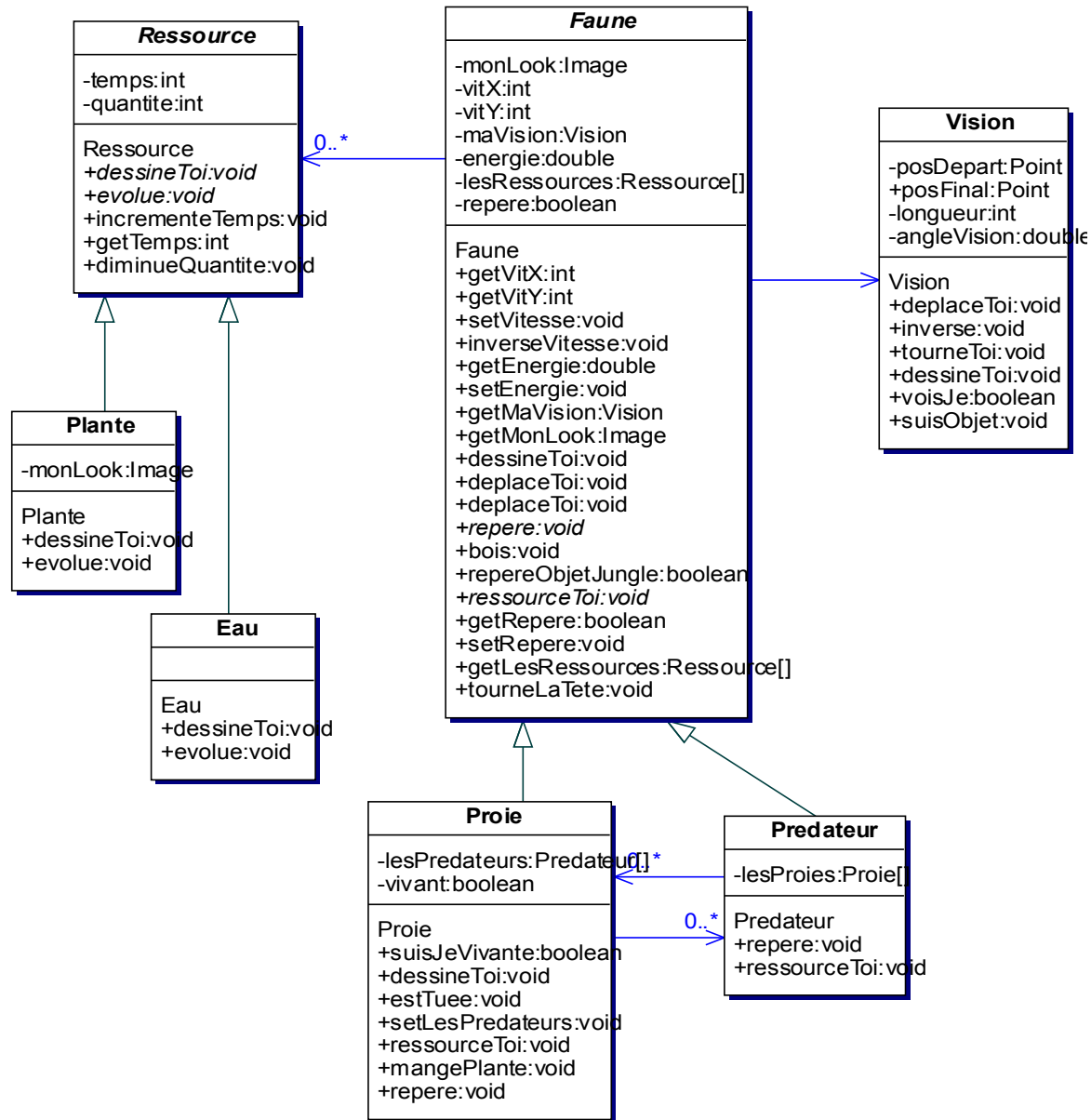


Spécialisation

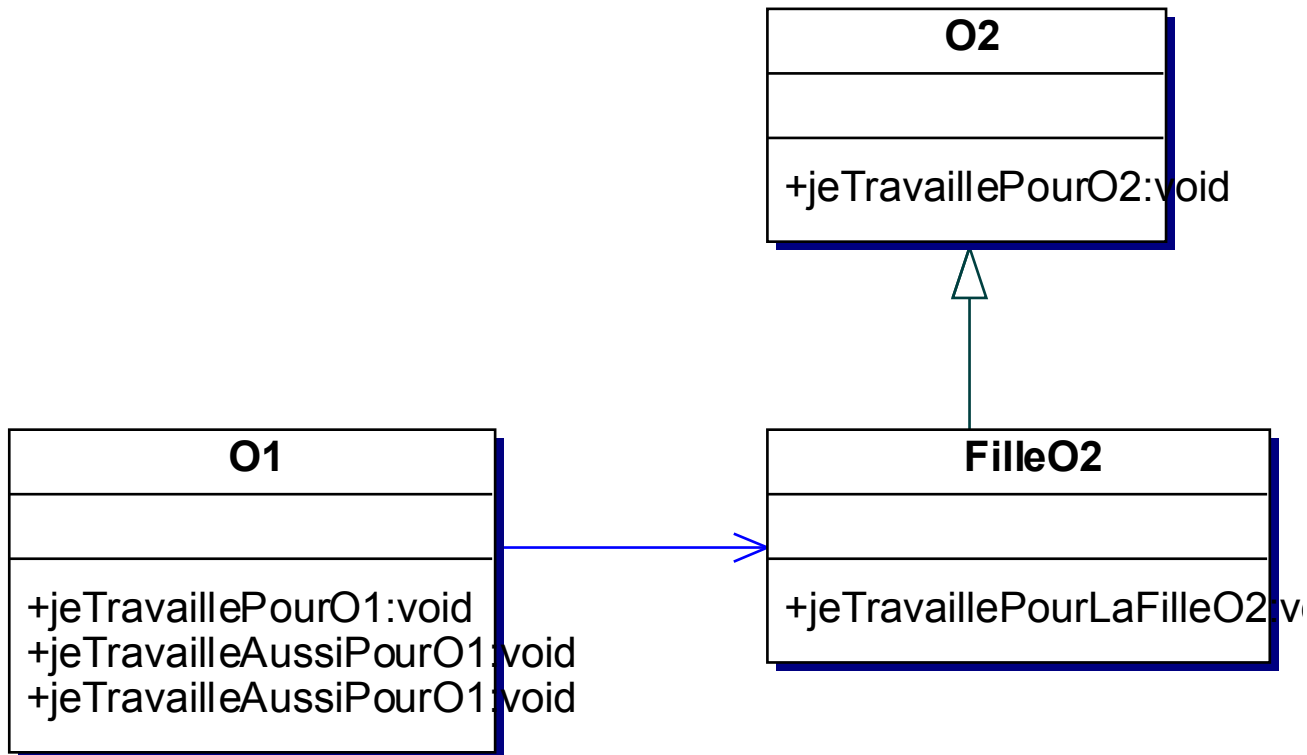
- Une super classe peut à son tour devenir une sous classe ...
- La profondeur est parfois critiquée parfois plébiscitée...

- L'addition des propriétés permet l'application du principe de substitution et de l'interprétation ensembliste:

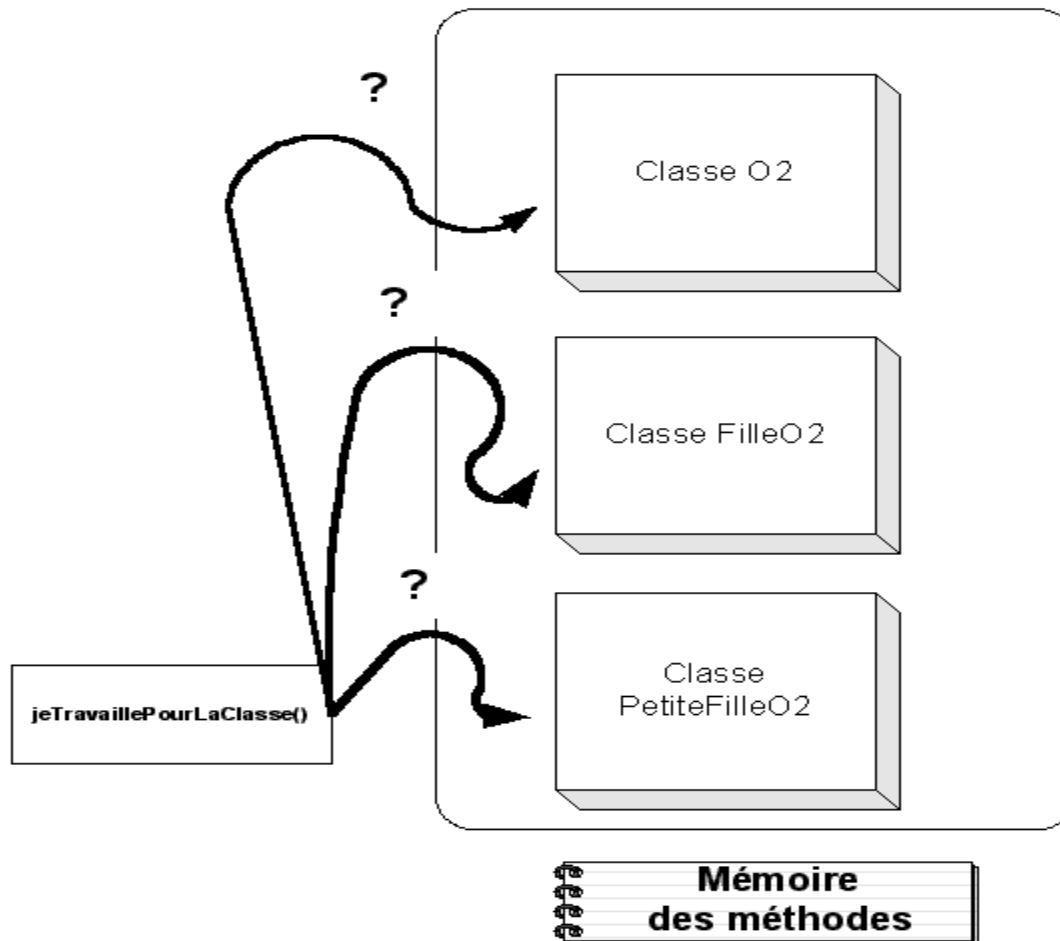




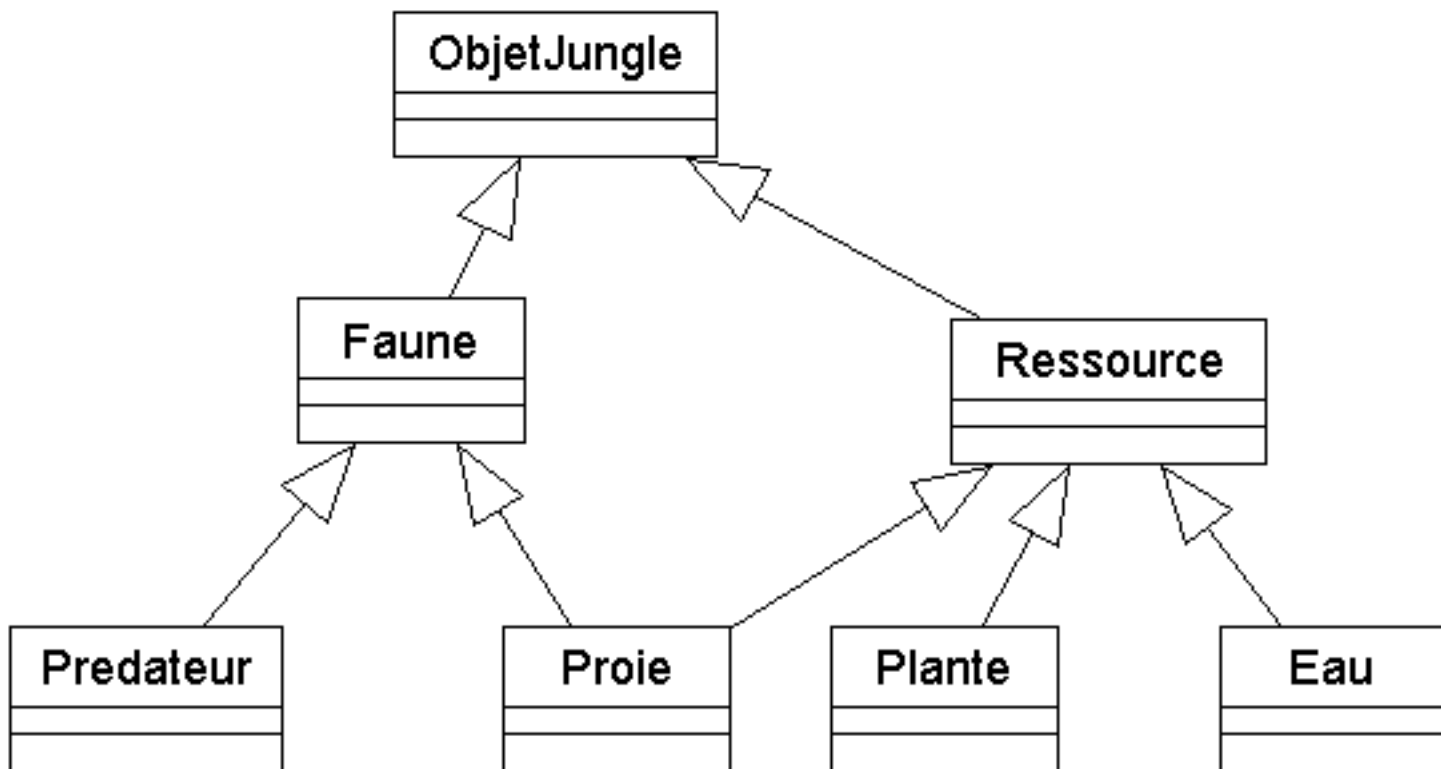
L'héritage des méthodes:



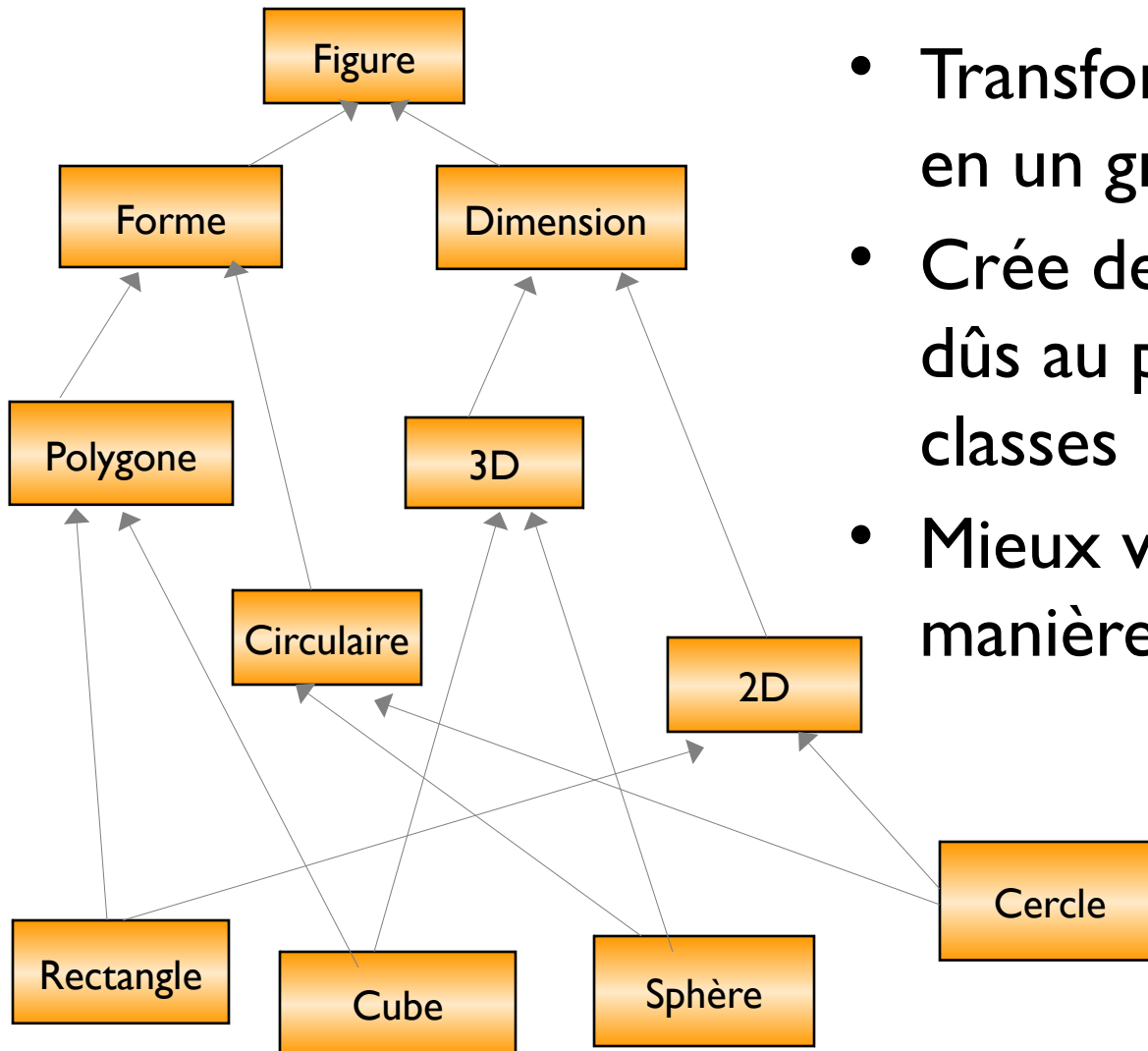
La recherche des méthodes



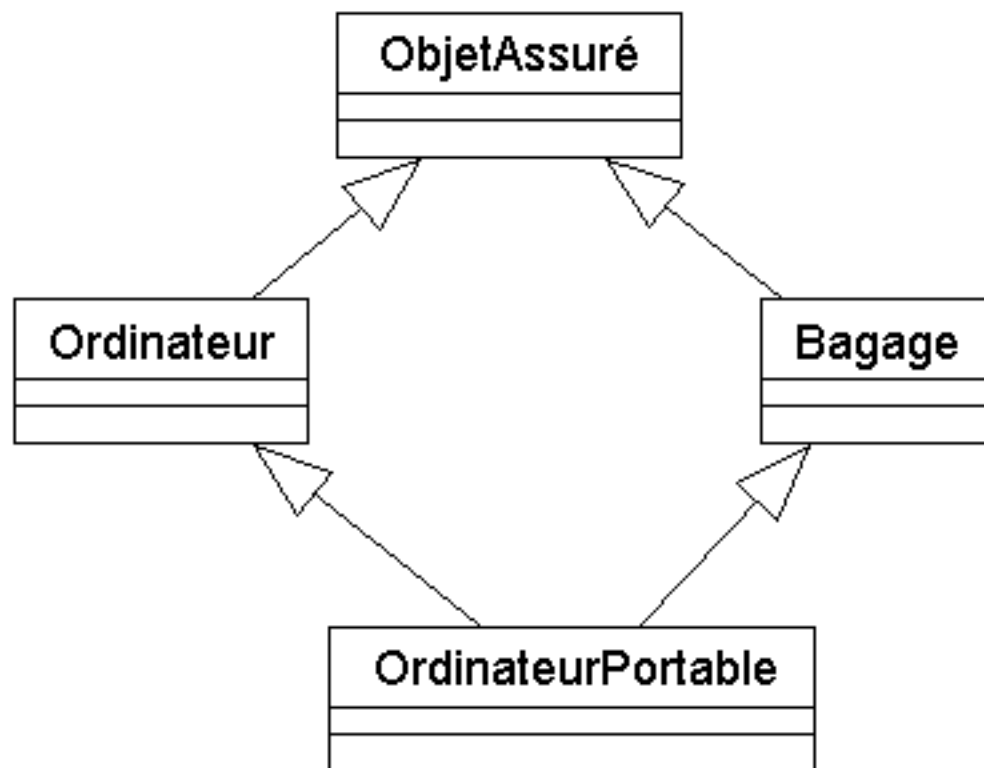
La problématique du multihéritage



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

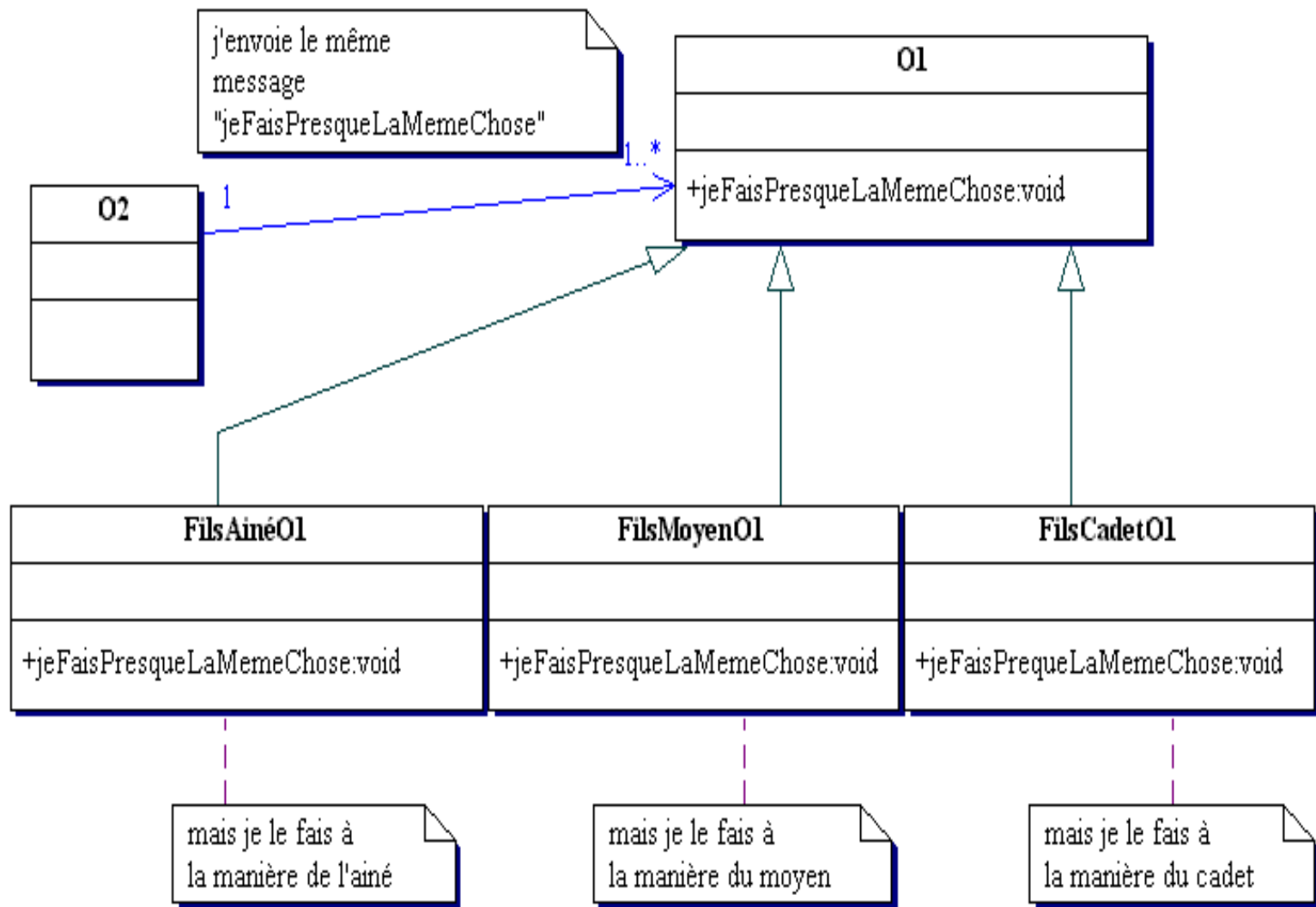


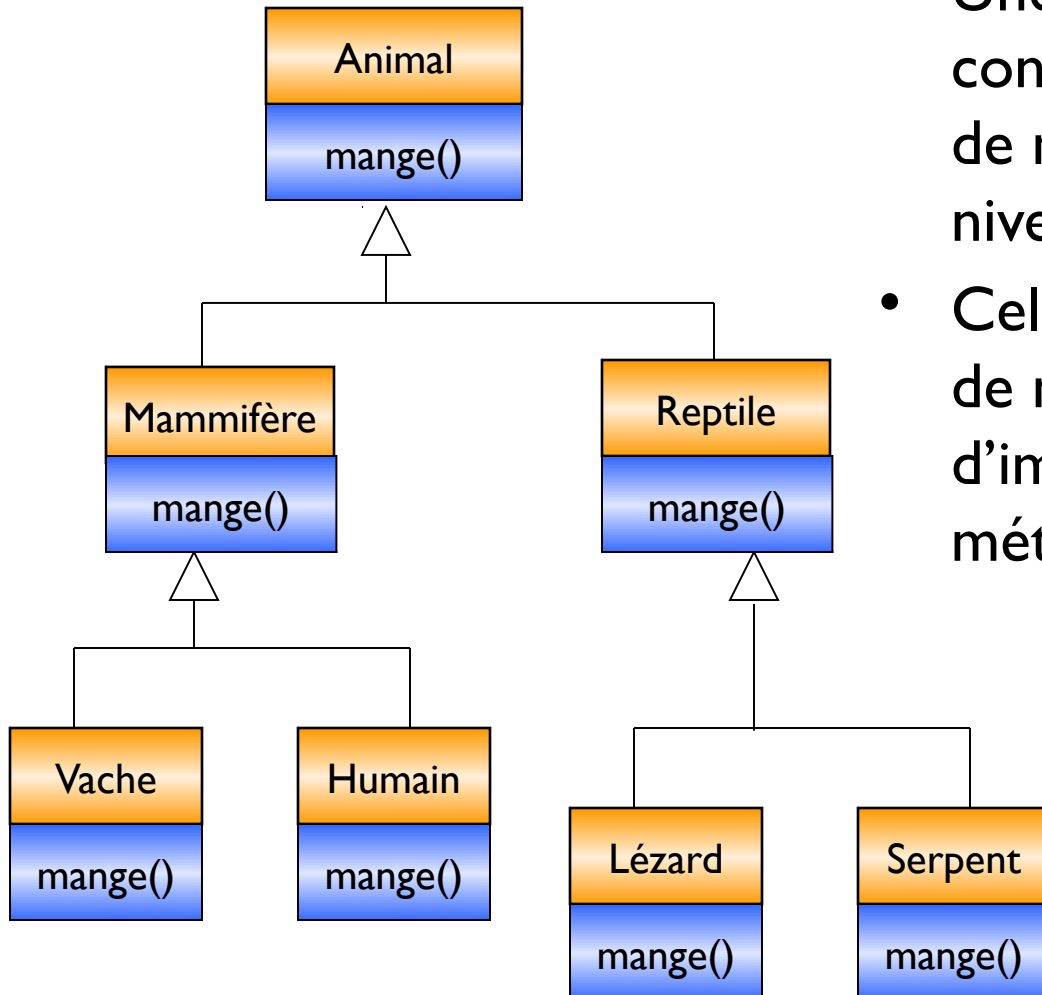
Polymorphisme

- 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

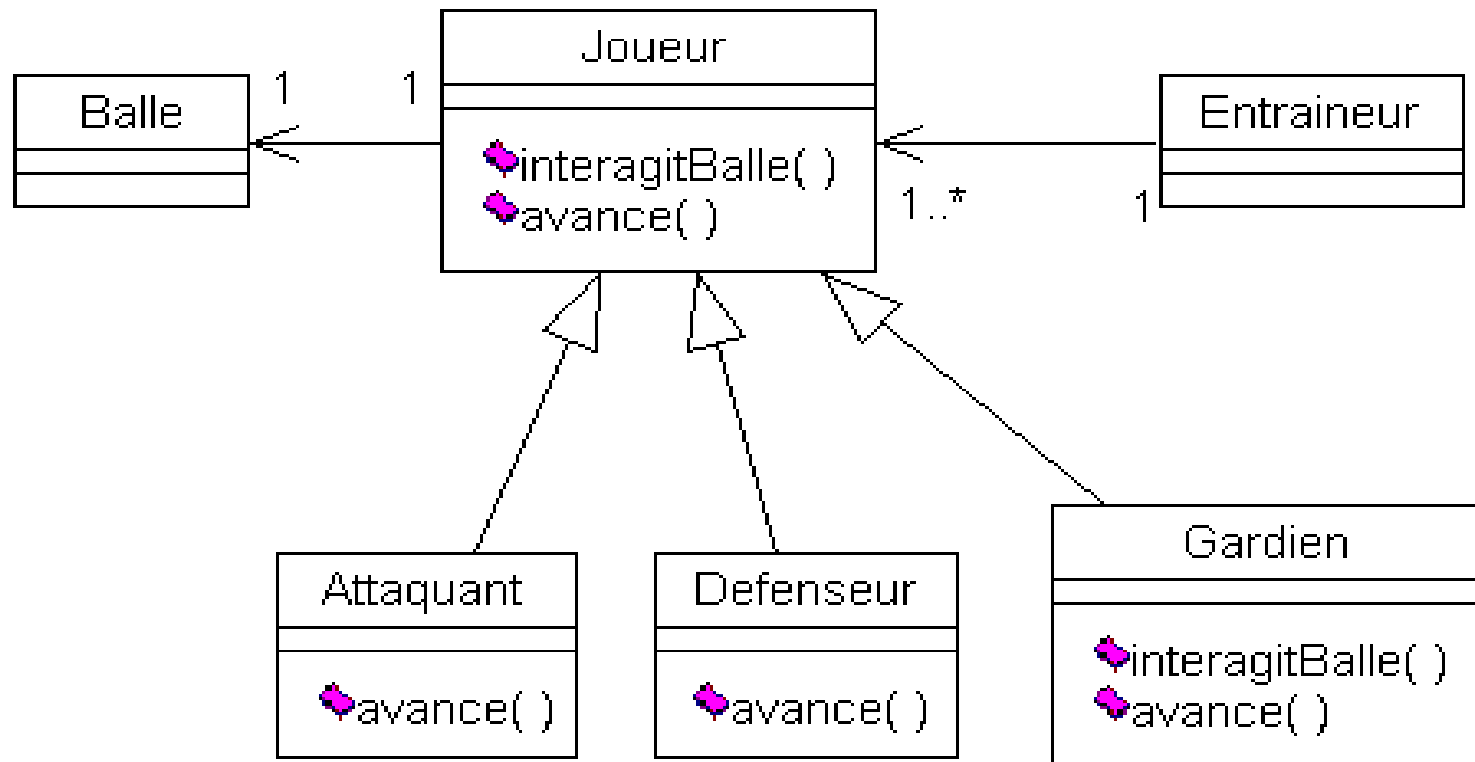
- Basé sur la redéfinition des méthodes
- Permet à une tierce classe de traiter un ensemble de classes sans connaître leur nature ultime.
- Permet de factoriser des dénominations d'activité mais pas les activités elles-mêmes.



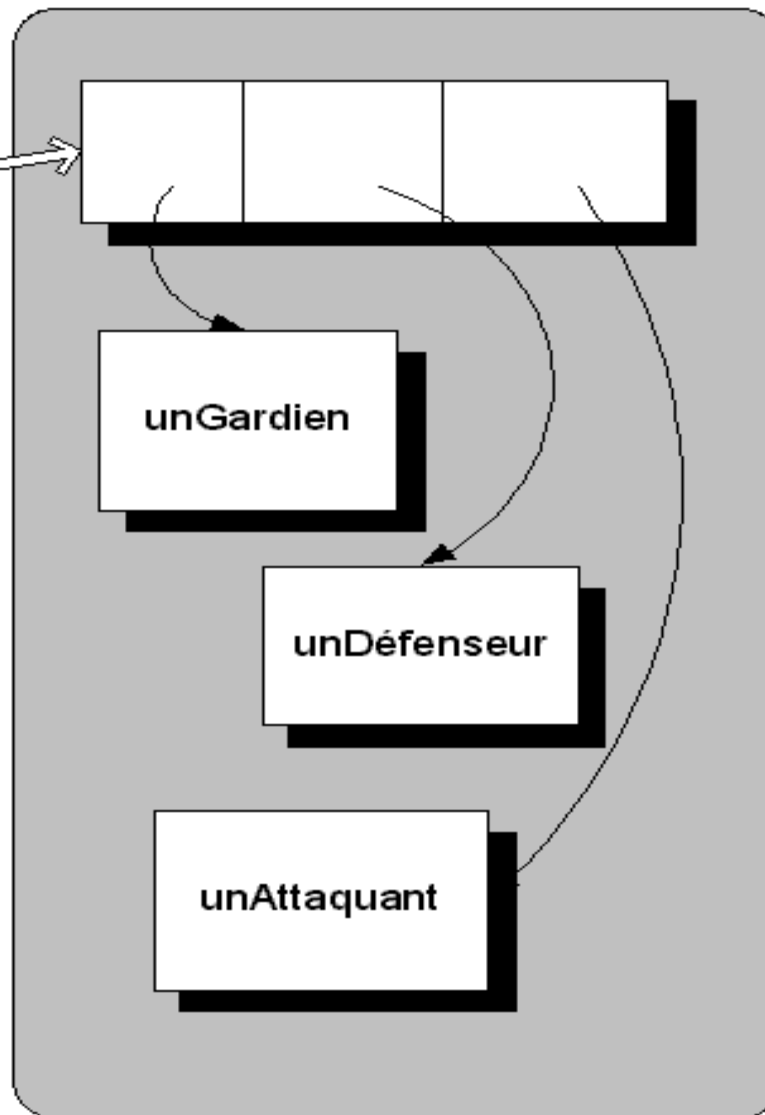


- Une méthode d'appellation constante peut être redéfinie de multiples fois à différents niveaux de la hiérarchie
- Cela permet à une tierce classe de ne pas à connaître les détails d'implémentation de la méthode

Un match de foot polymorphique:

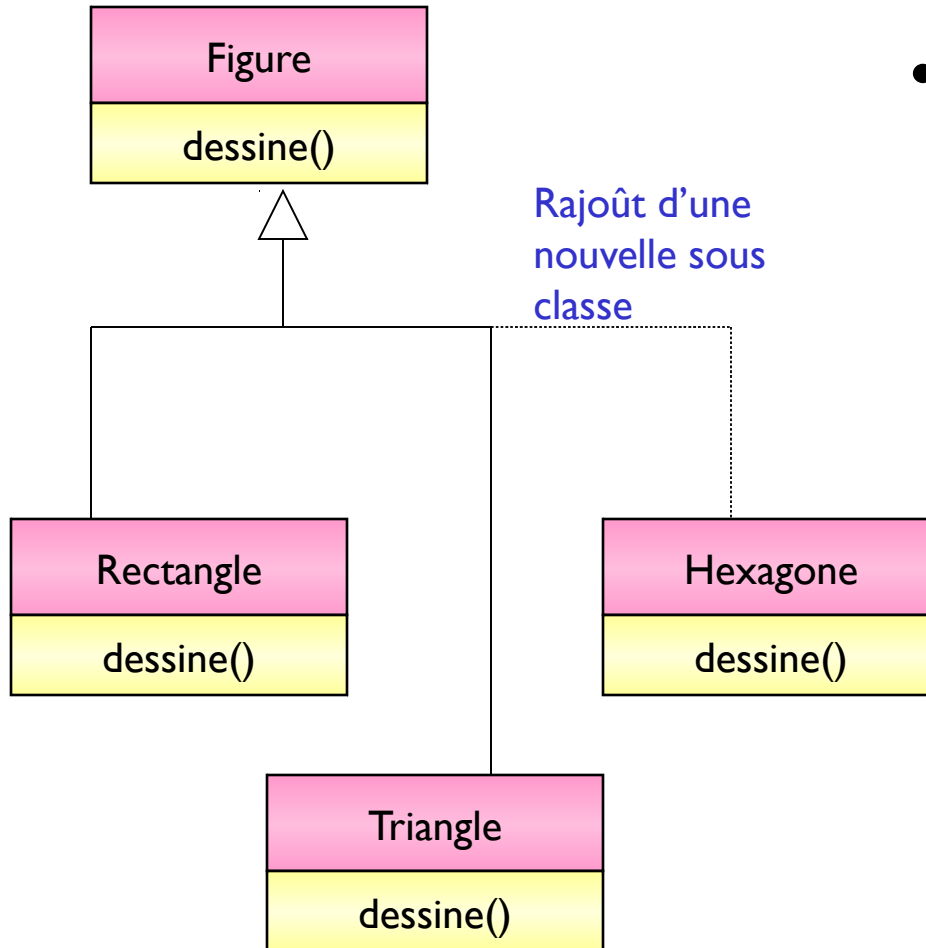


les Joueurs



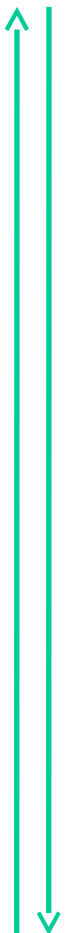
**Mémoire des
objets**

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 constructeur 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.
- Un **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.

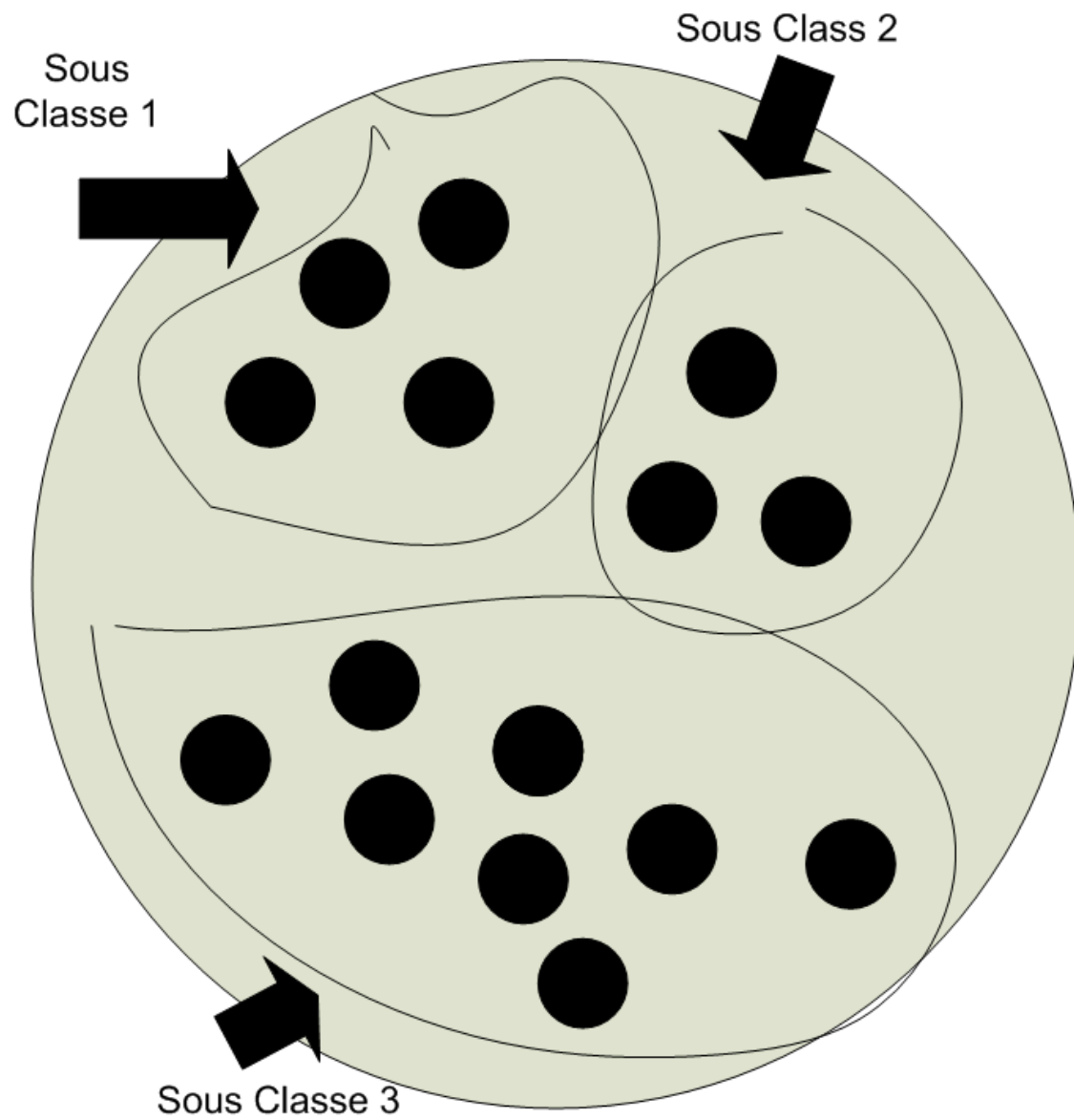
<i>AnAbstractClass</i>
- aField
+ doSomething()
+ doAnotherThing()

```

public abstract class AnAbstractClass{
    private int aField;
    ...
    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



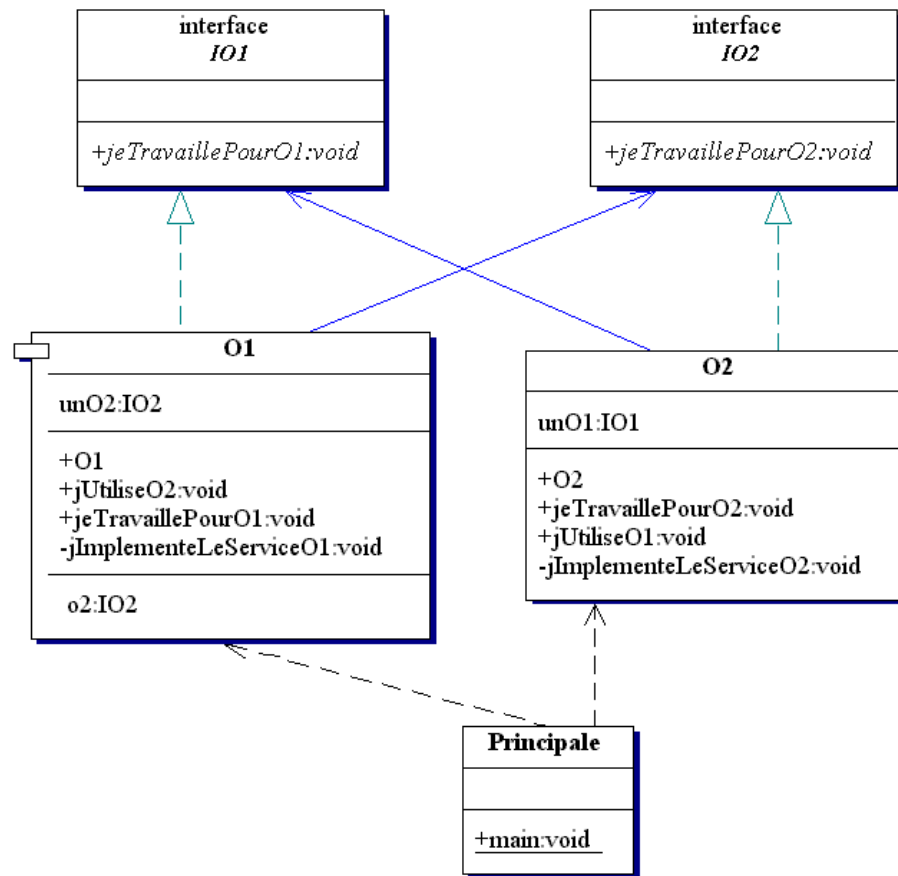
Exemples Python et Java

- Voir livre OO chapitre 12
- Le match de foot polymorphique

Les trois raisons d'être des interfaces

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

Les interfaces



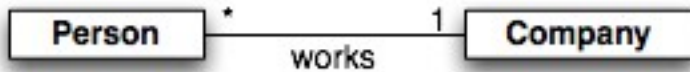
Associations entre objets

Association, composition, agrégation

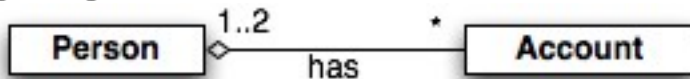
- Une **association** est une relation sémantique entre des classes qui définit un ensemble de liens.
 - Un homme est associé à sa femme
- Une **agrégation** est une association dans laquelle il y a un lien d'appartenance entre les deux objets associés (contenant/contenu, possession, ...).
 - Un homme possède un compte en banque
- Une **composition** (ou agrégation forte) est une agrégation dans laquelle la disparition du composite entraîne 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

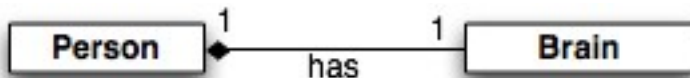
Association : Une personne travaille pour une et une seule compag



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 $m \geq 0$
 - Par défaut, la cardinalité est *un*.
 - Dans une composition, la cardinalité du côté de l'agregat 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
- 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 $\leftrightarrow$ programmation

Introduction à l'UML

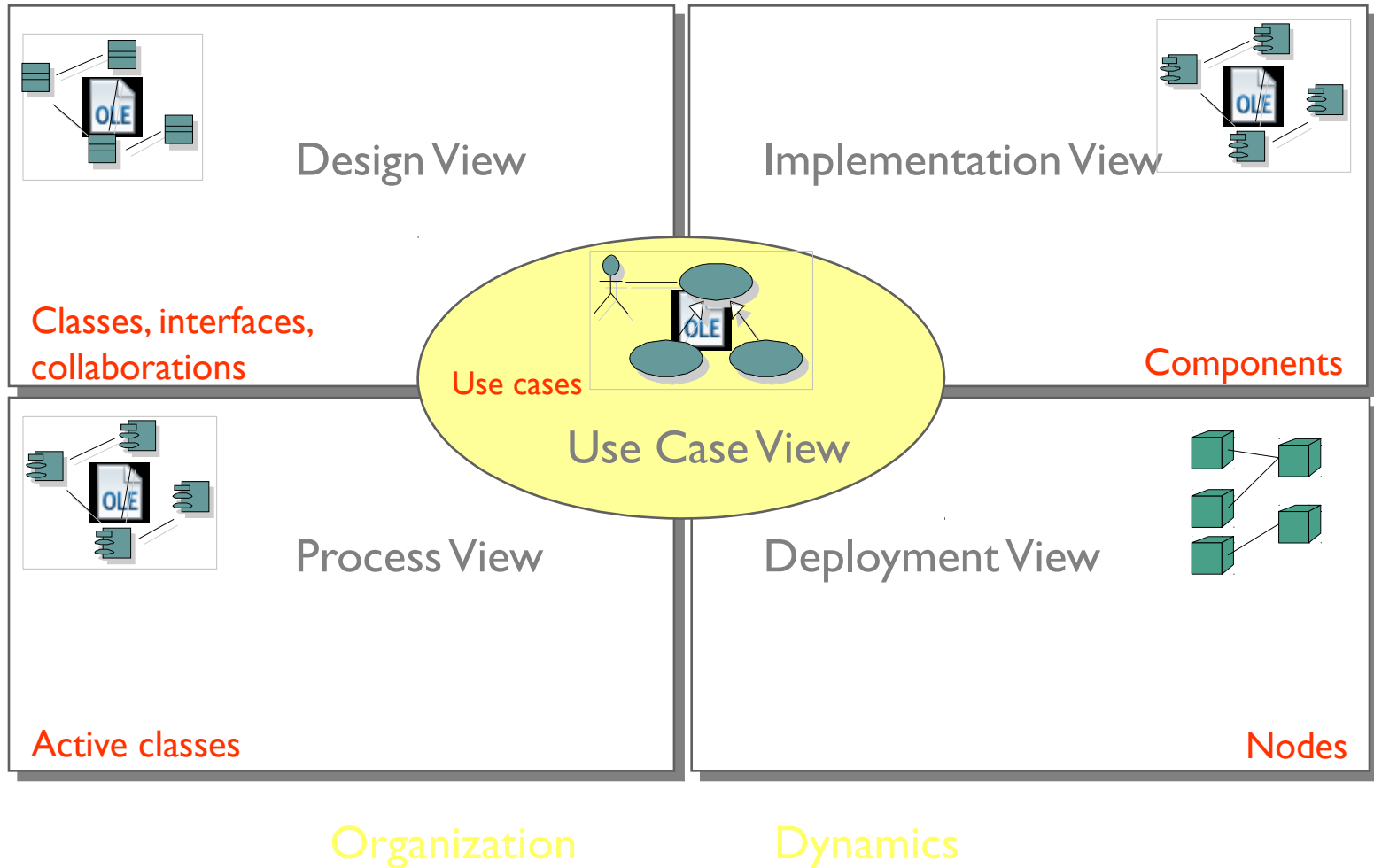
Introduction à UML

- UML (Unified Modeling Language)
- La fusion des travaux de G. Booch, J. Rumbaugh 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-I: 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)

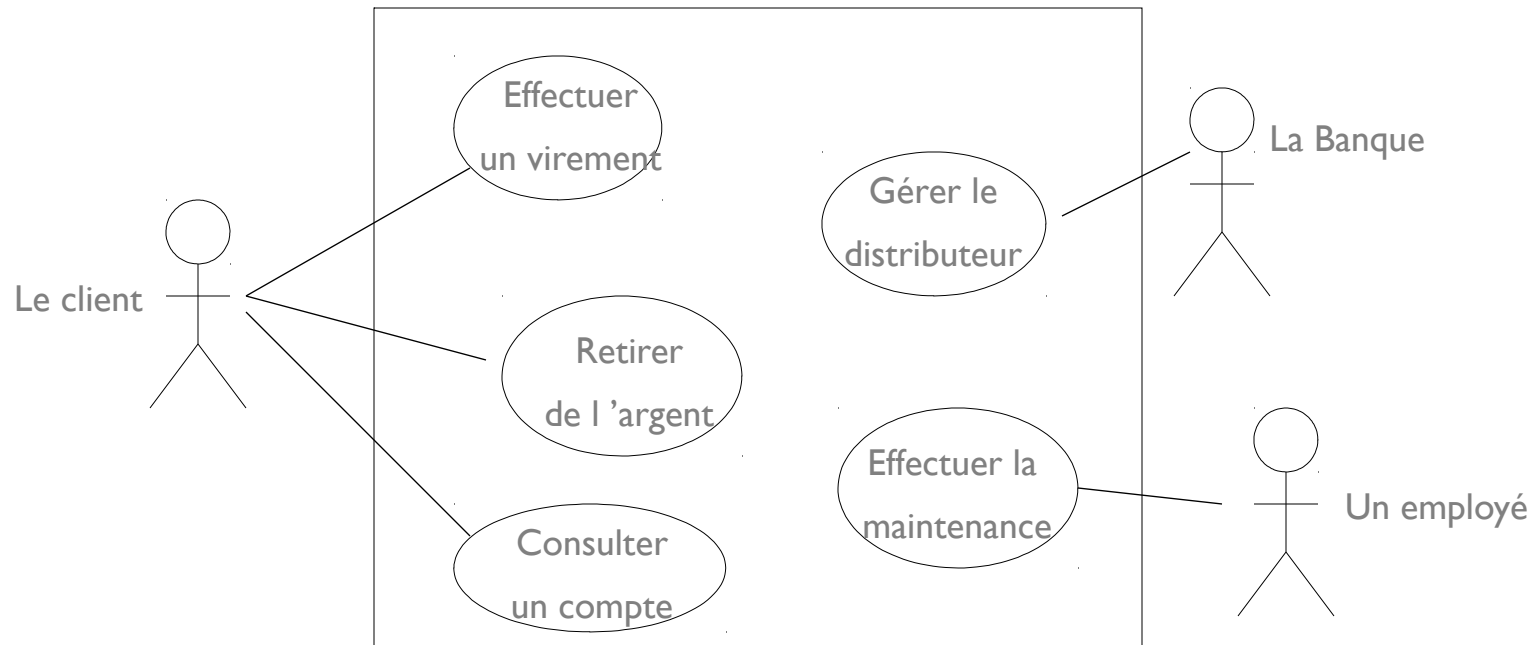
- **Les diagrammes d'états-transitions:** scrute les cycles de vie d'une classe d'objet, la succession d'états et les transitions.
- Les diagrammes d'activité: le comportement des différentes opérations en termes d'actions.
- Les diagrammes de composants: représente les composants physiques d'une application.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



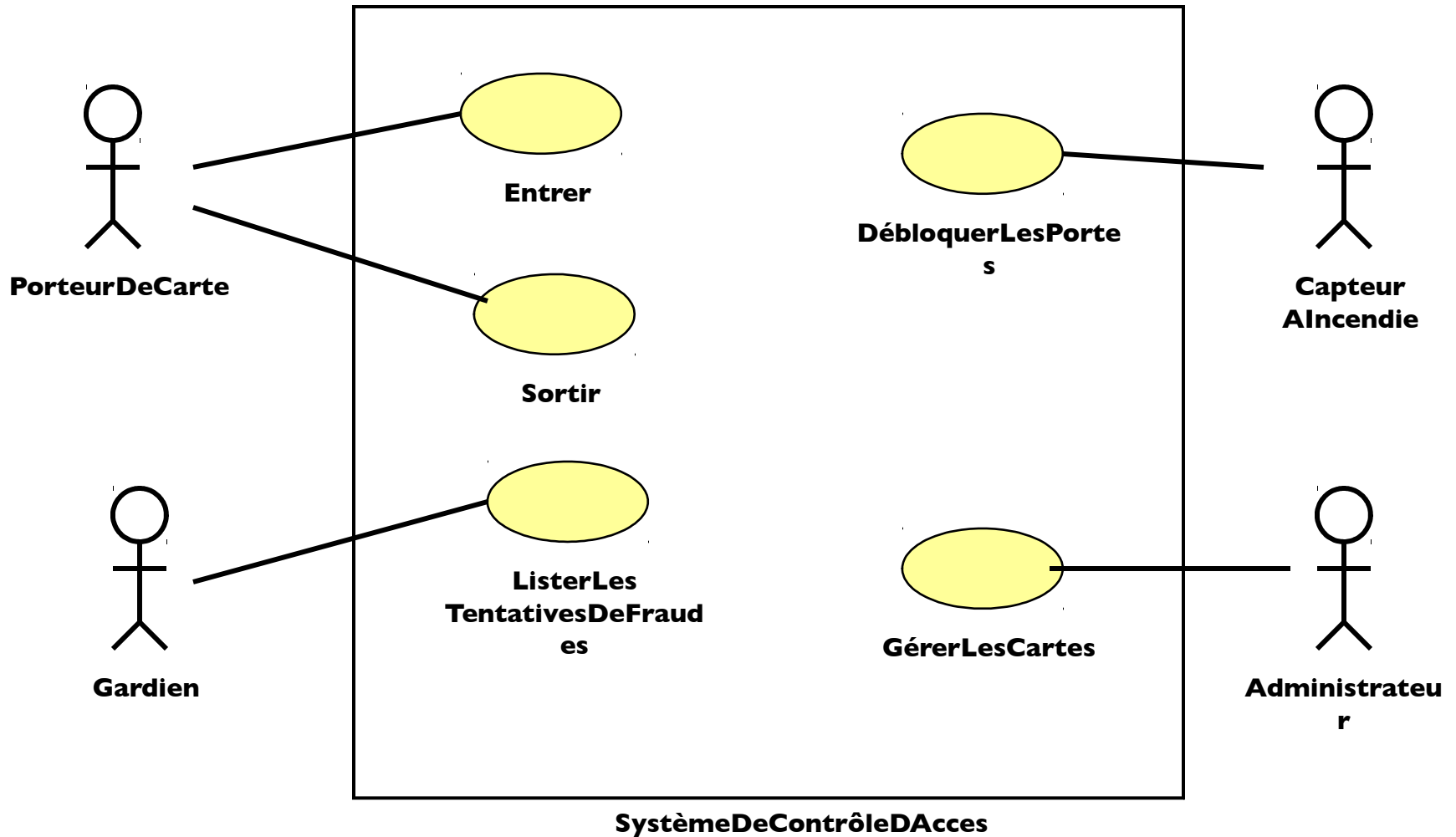
- Les diagrammes: de cas d'utilisation, de classes, de séquences et d'états-transitions.

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

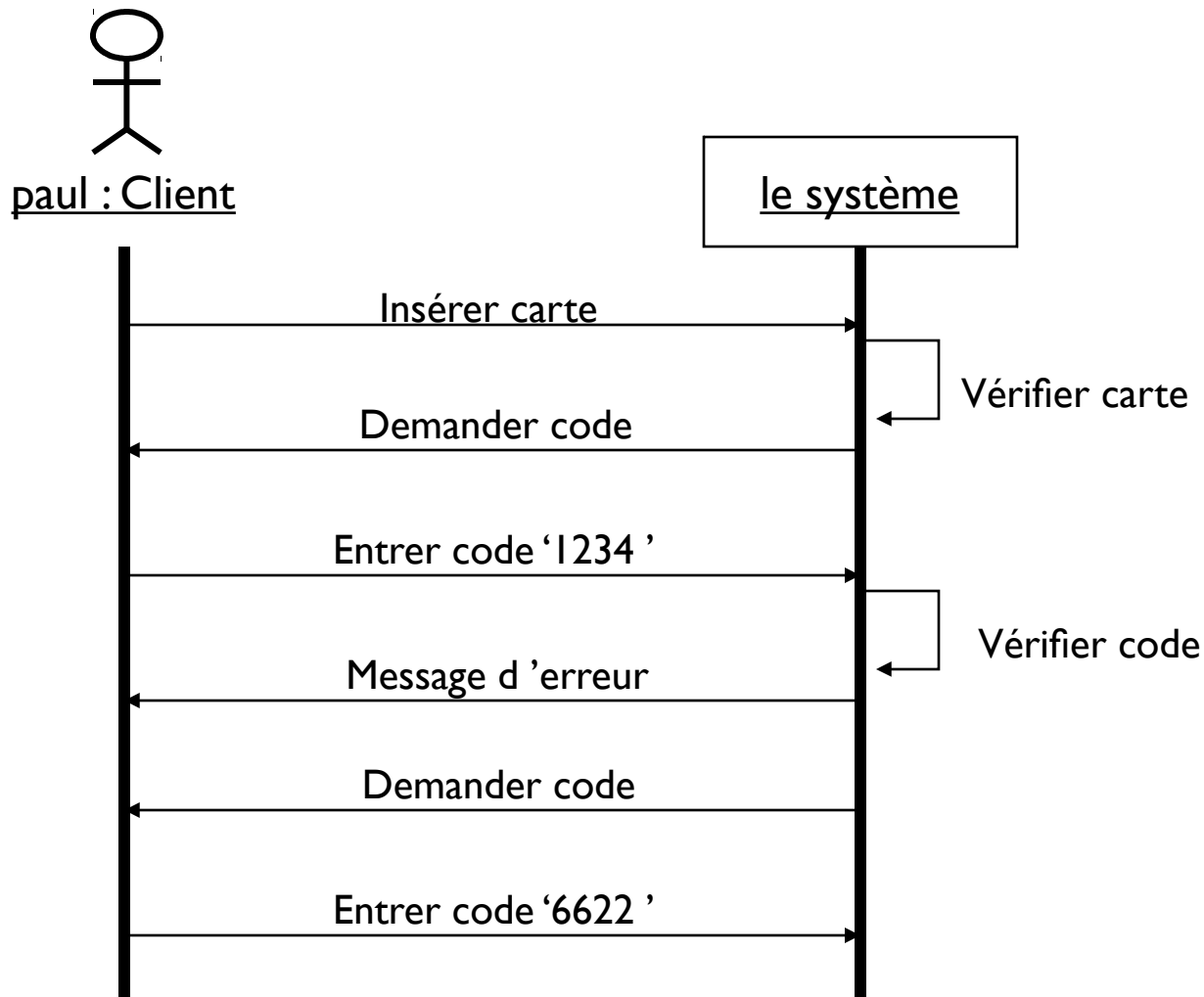


Les cas d'utilisation (suite)

- 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.

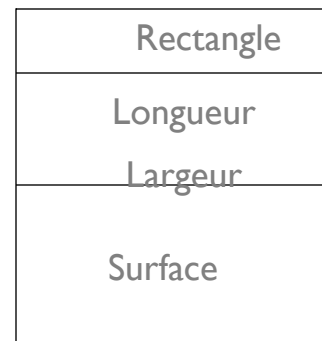
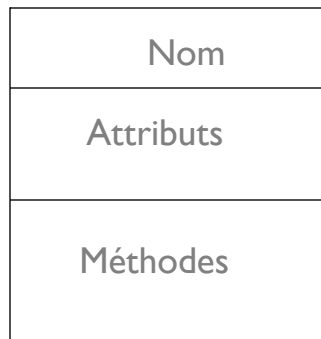
Exemple de scénario

Cas d'utilisation détaillé

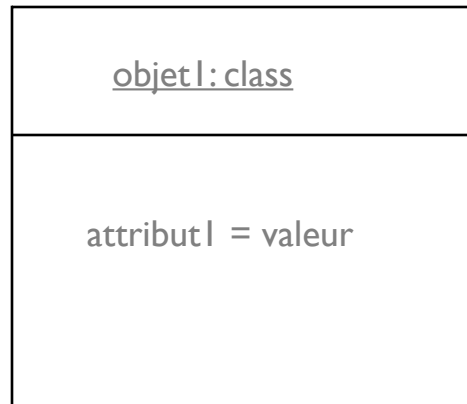


Les diagrammes de classe

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



Les objets

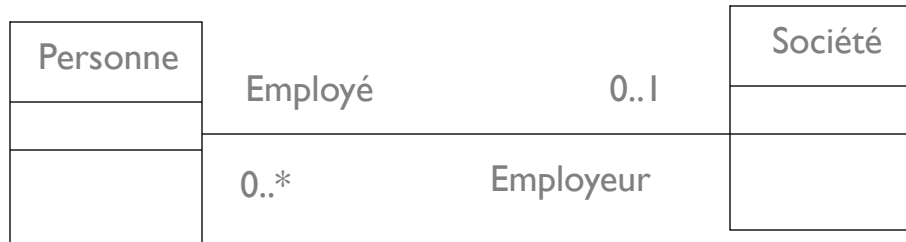


Les diagrammes de classe (suite)

- L'association entre classes

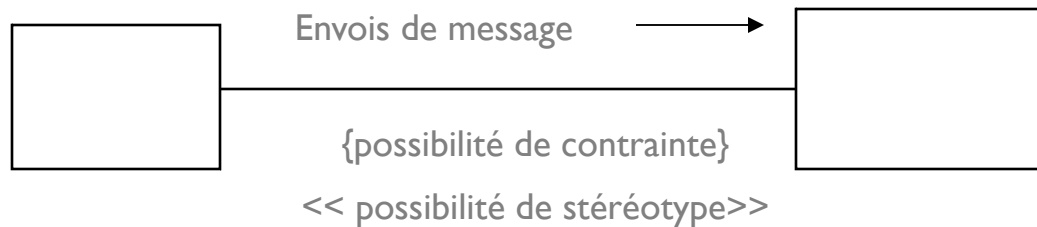


- Multiplicité des associations

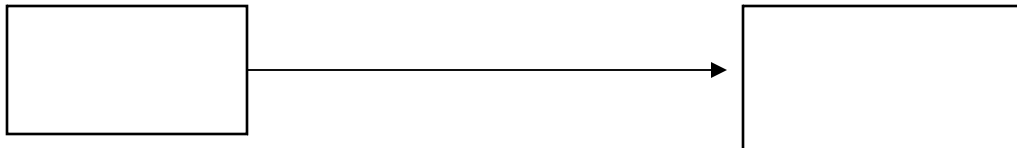


L'association

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

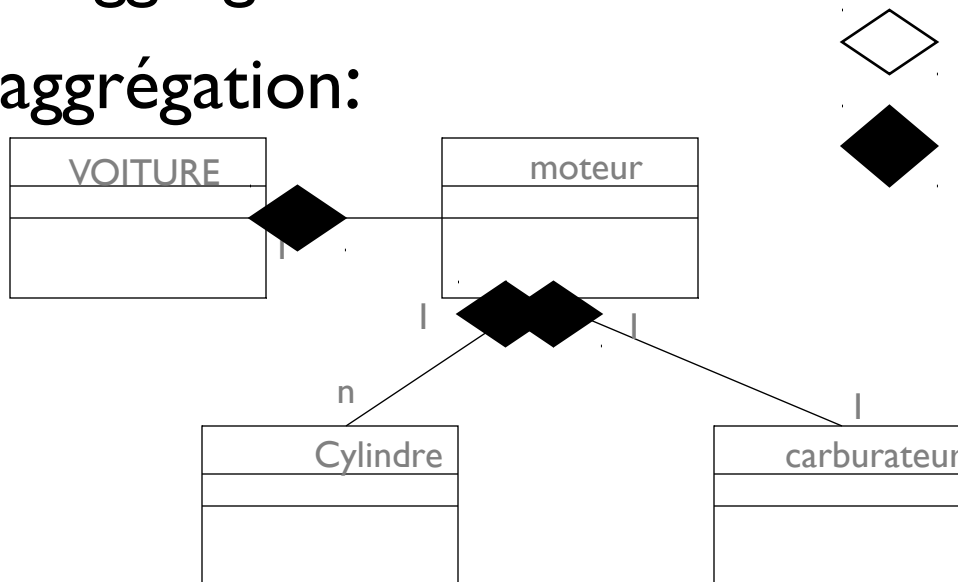


- L'association peut être directionnelle:



Les diagrammes de classe (suite)

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

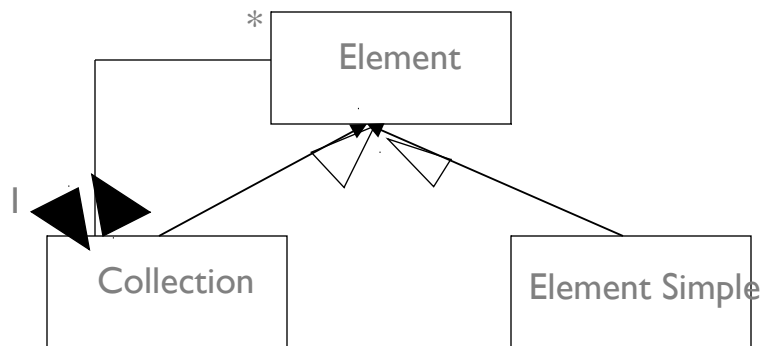
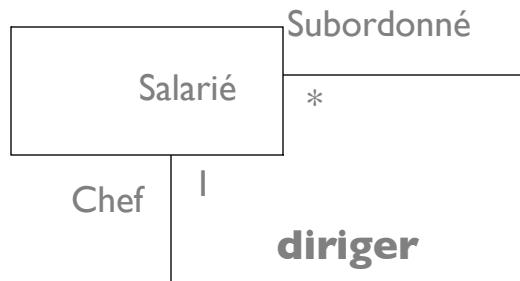


agrégation

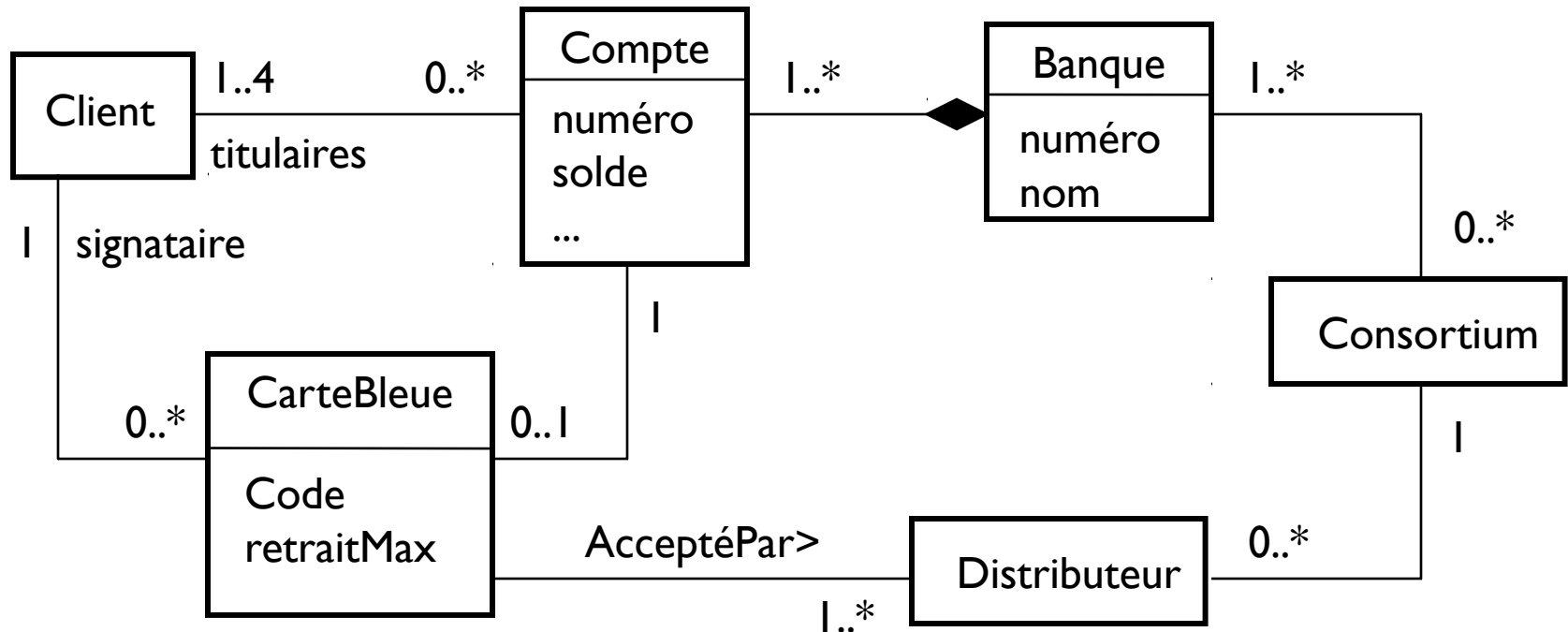


Composition: l'existence des parties est conditionnée par l'existence du contenant.
En C++ cela permet de choisir entre pointeur et valeur

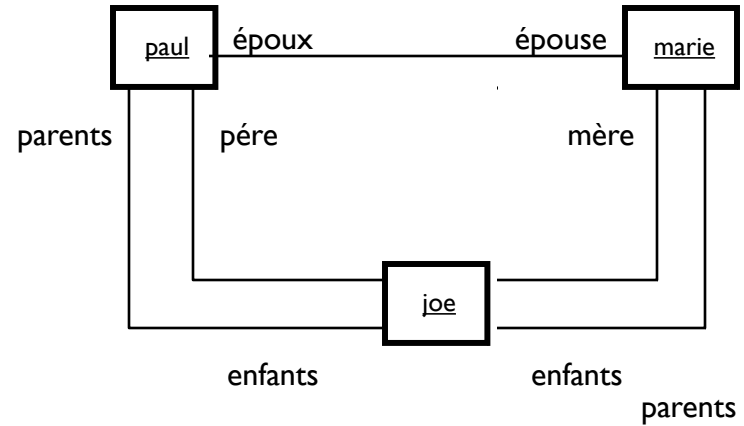
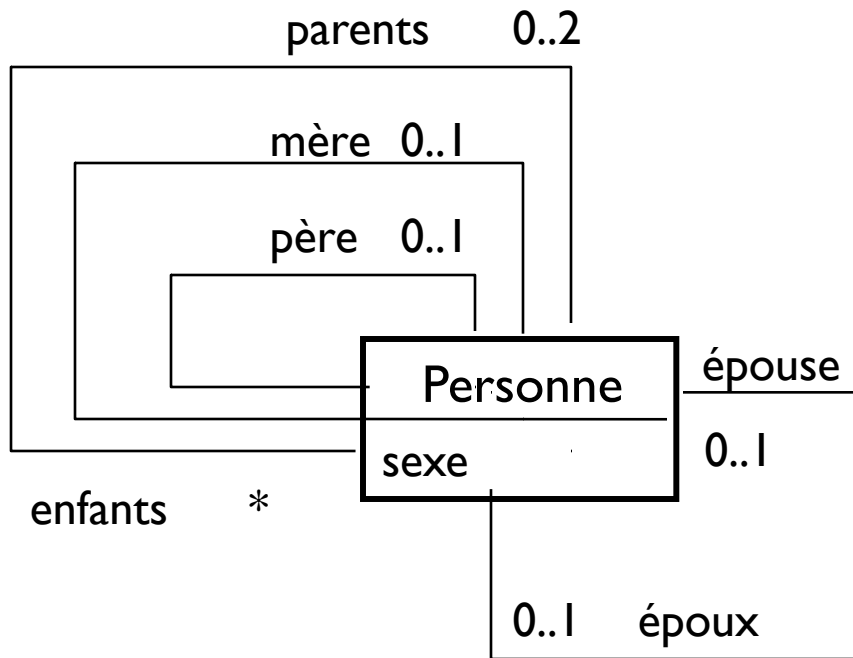
Diagrammes de classes (suite)



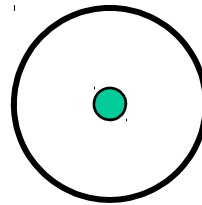
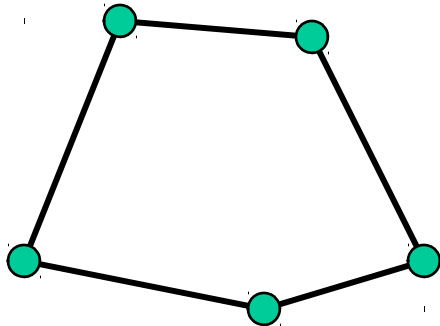
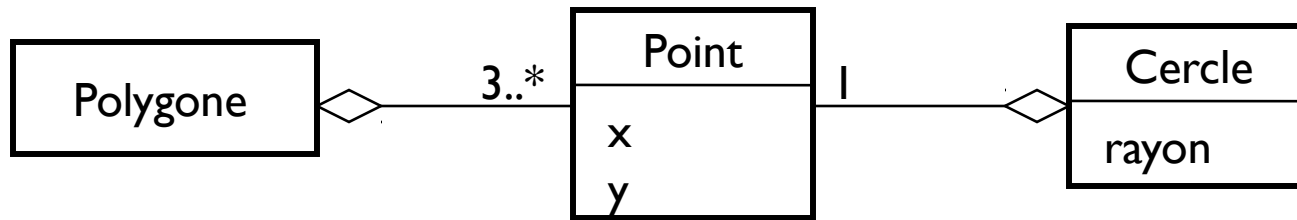
Diagrammes de classes



Exemple

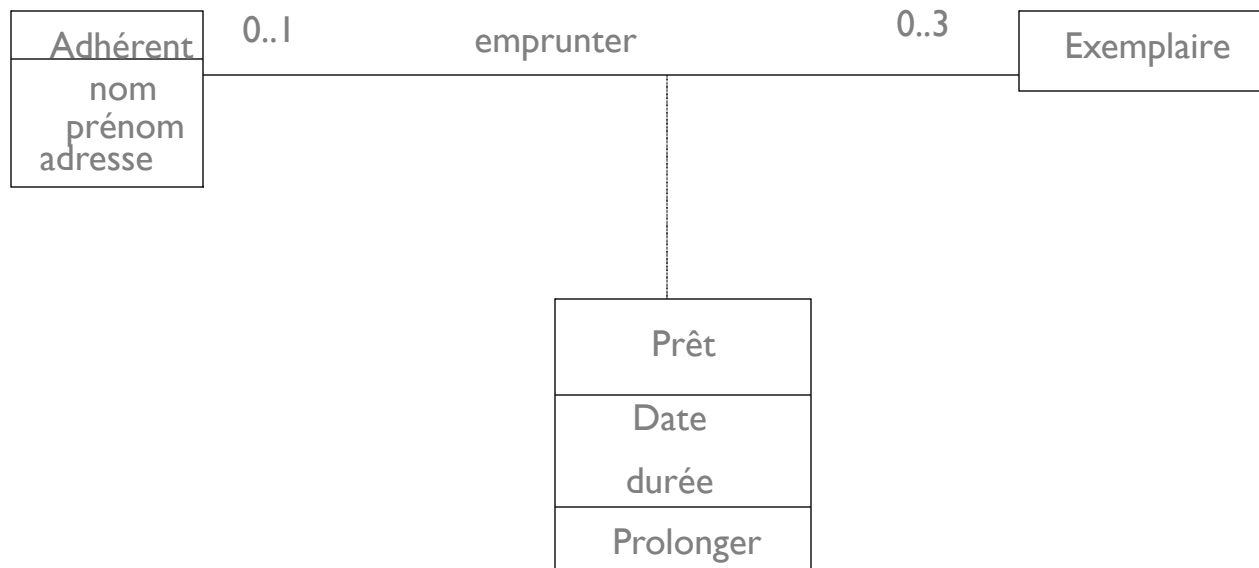


Composition



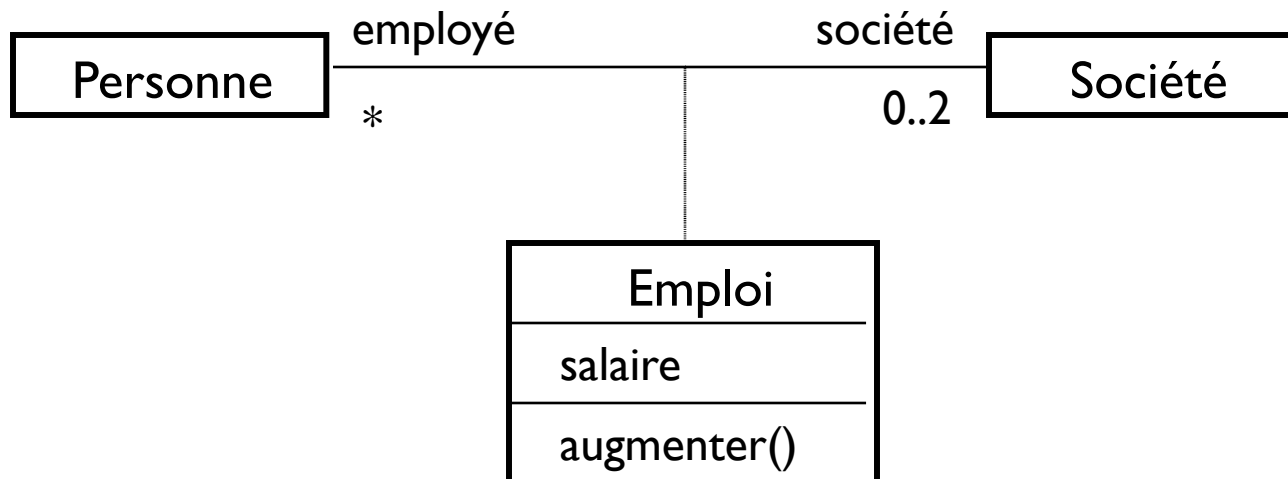
Diagrammes de classe (suite)

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

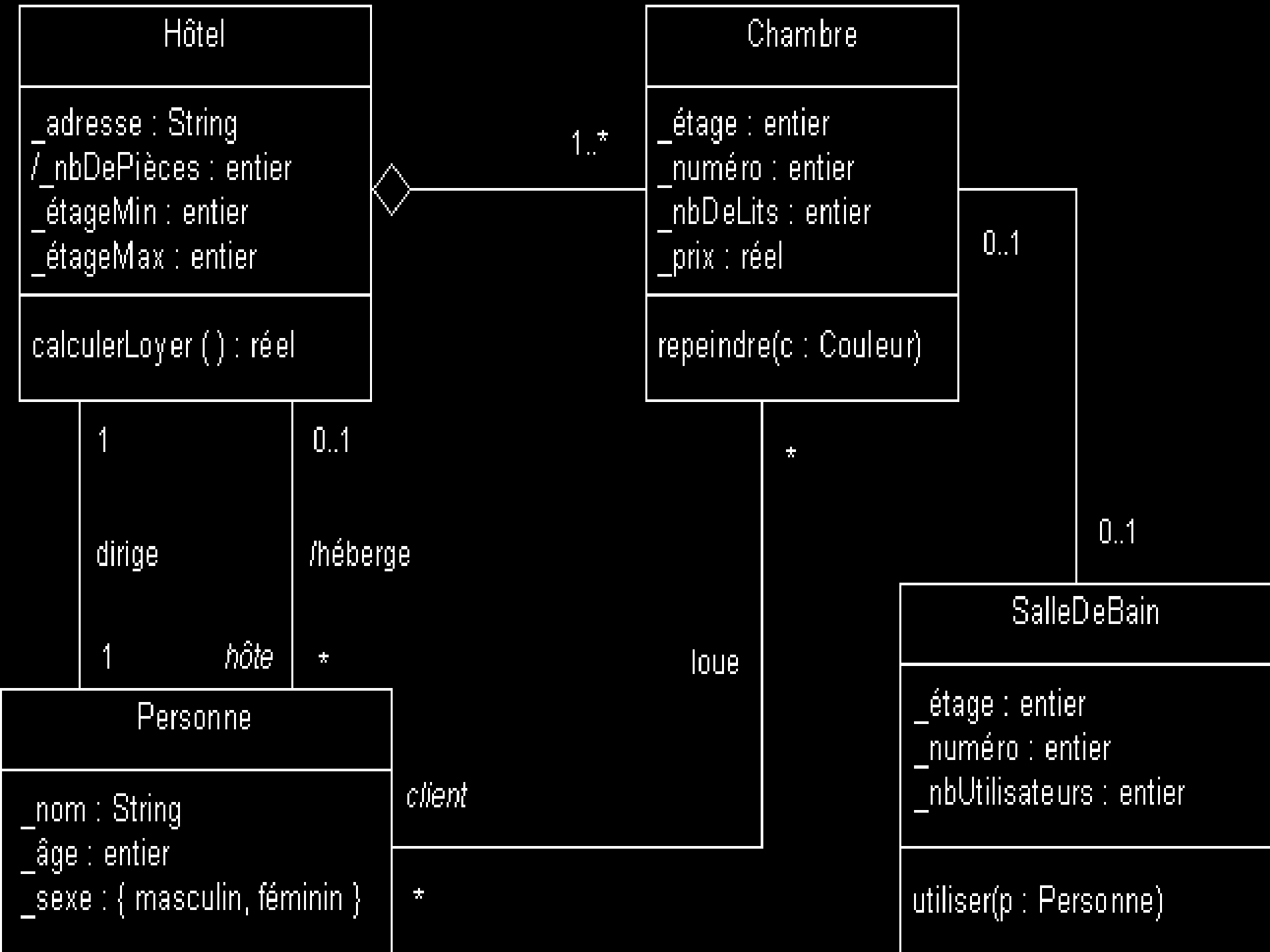


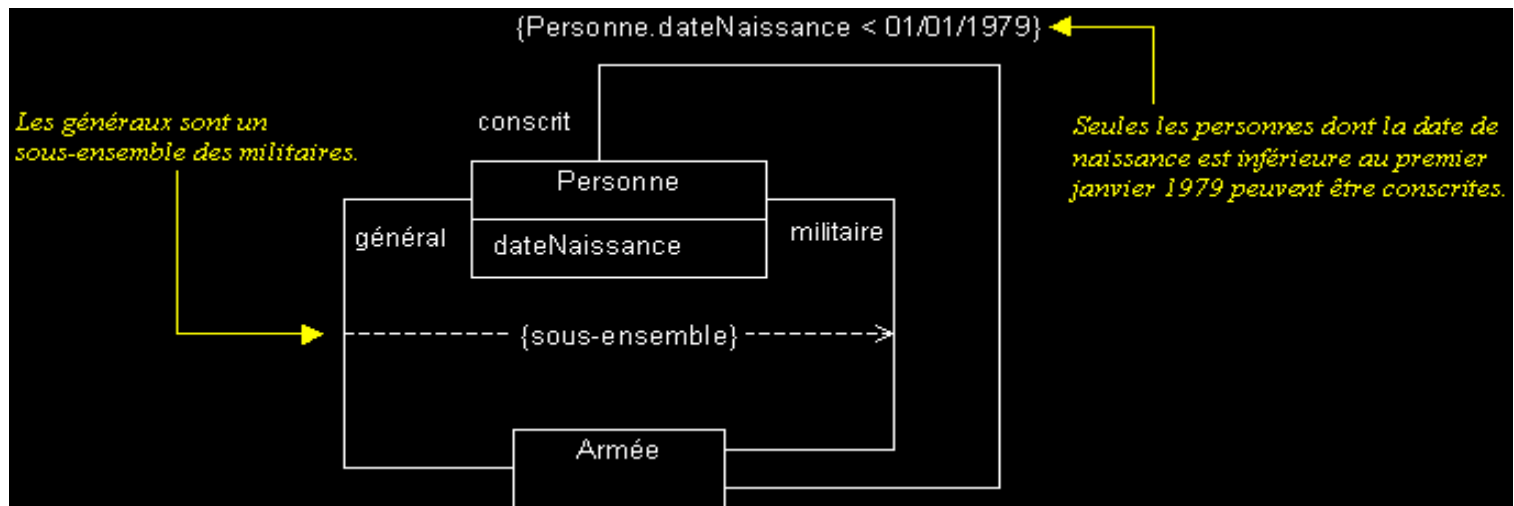
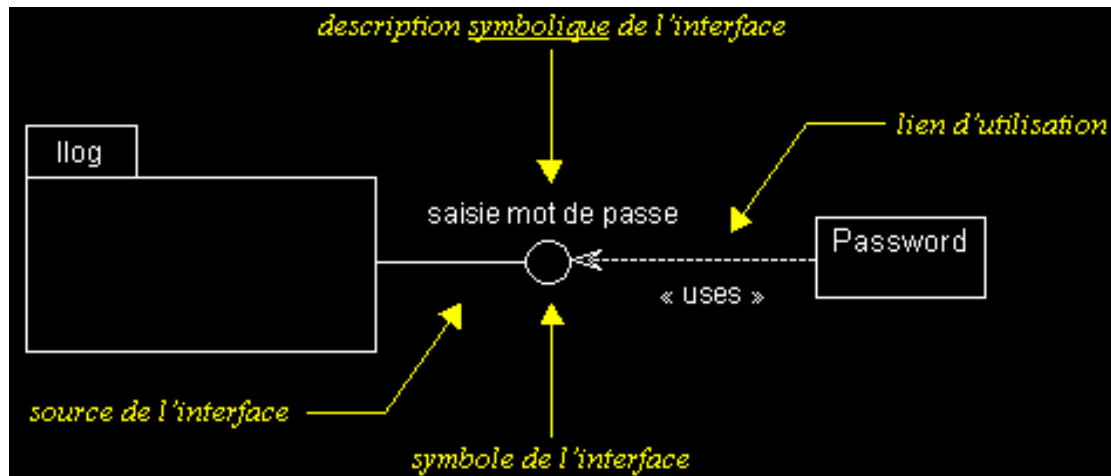
Classes associatives

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

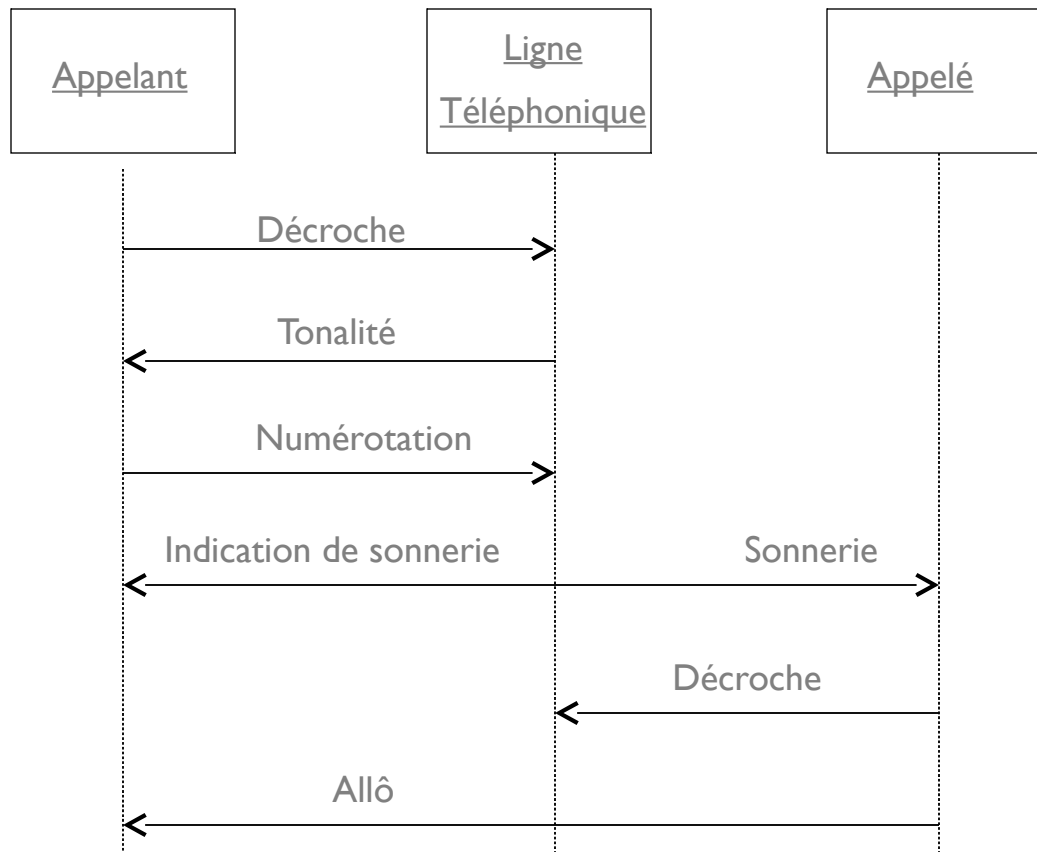


Le nom de la classe correspond au nom de l'association





Diagrammes de séquence (suite)



Diagrammes de séquence

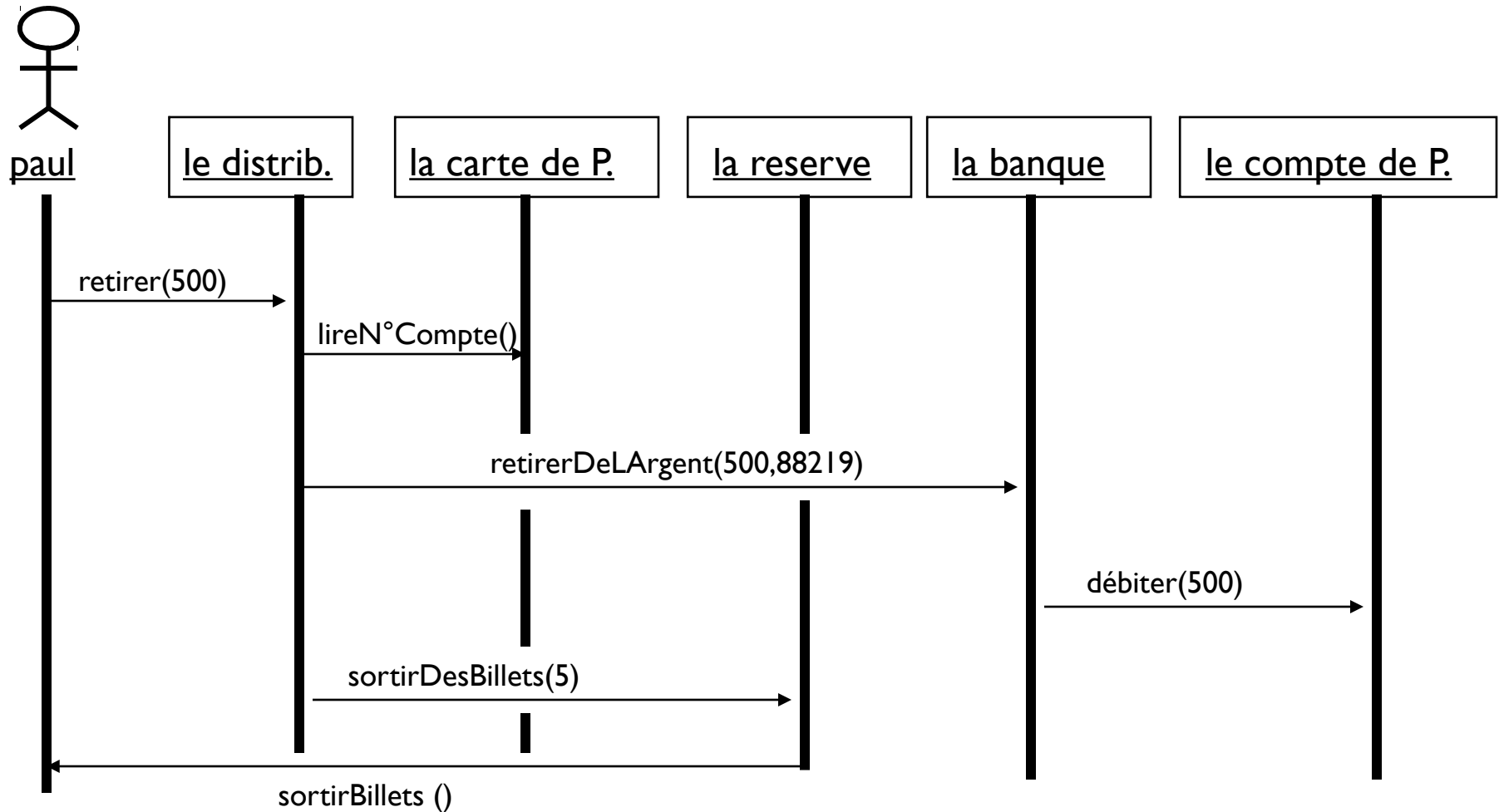
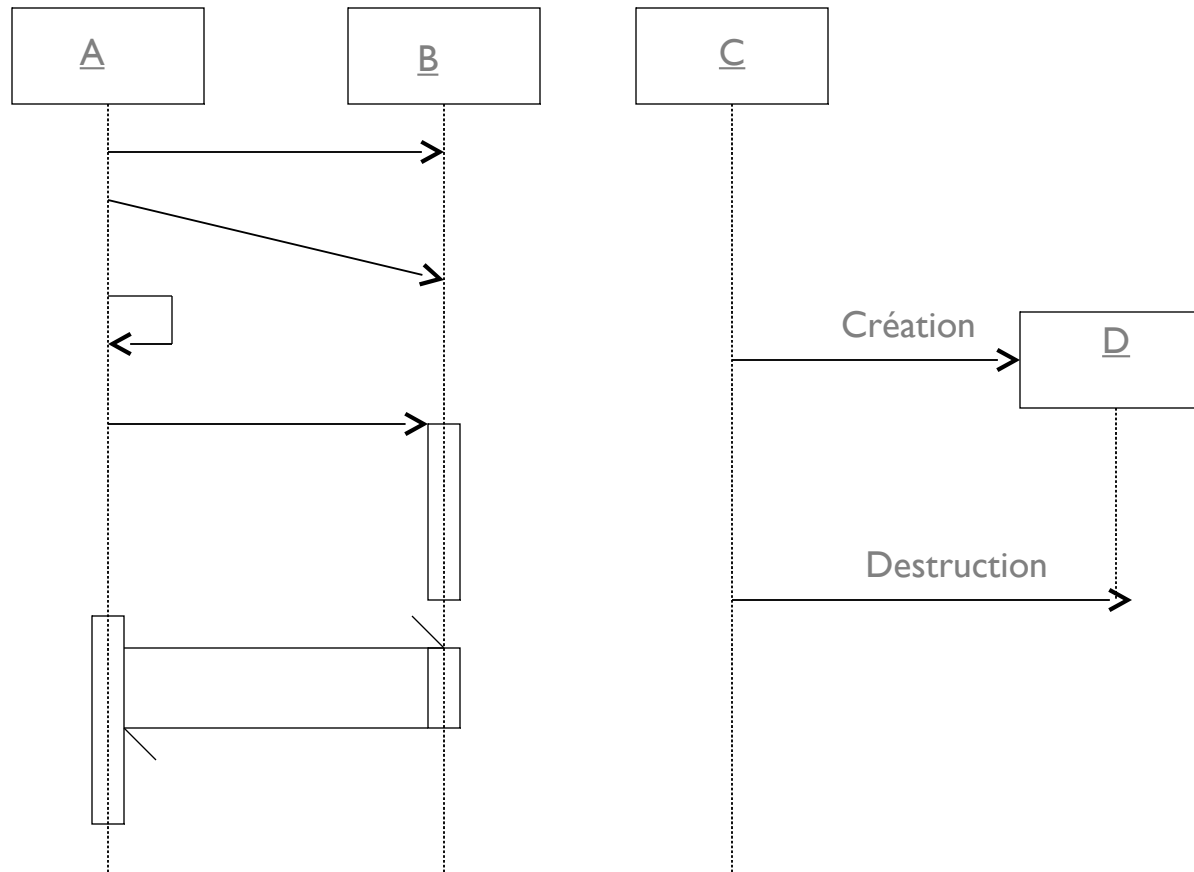
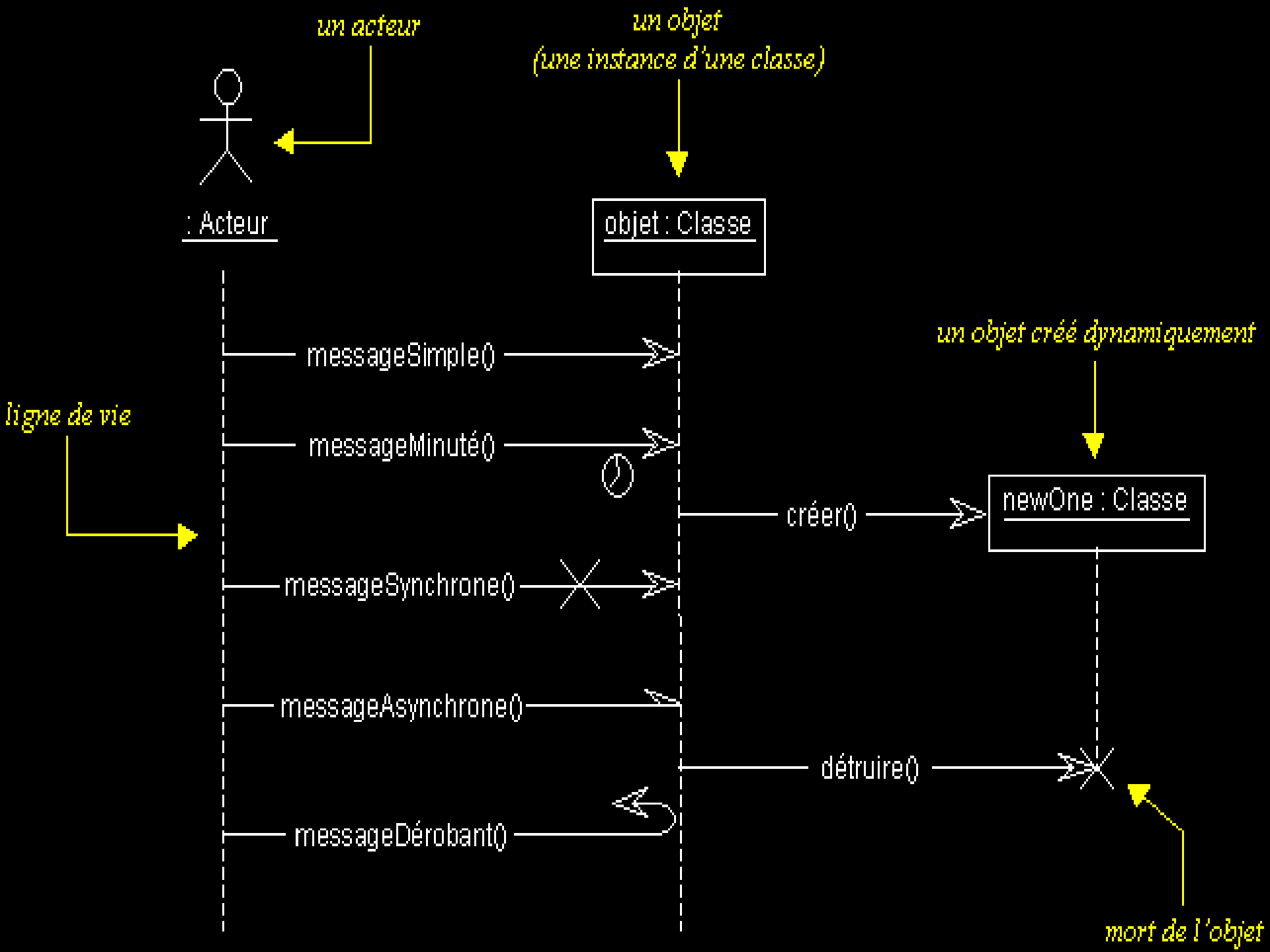


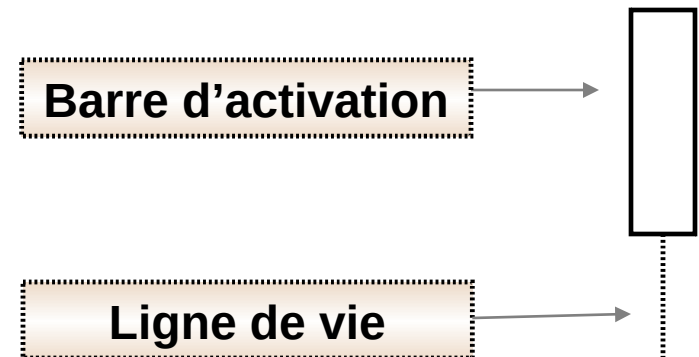
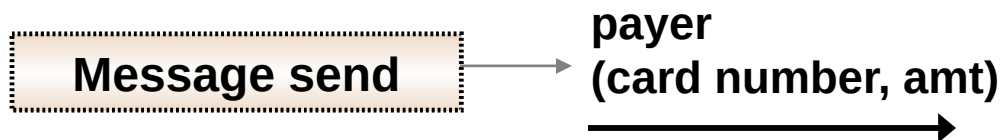
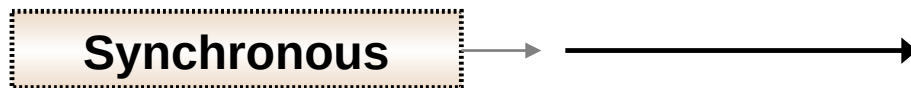
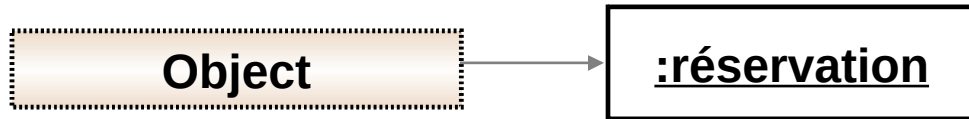
Diagramme de séquence (suite)

Quelques exemples

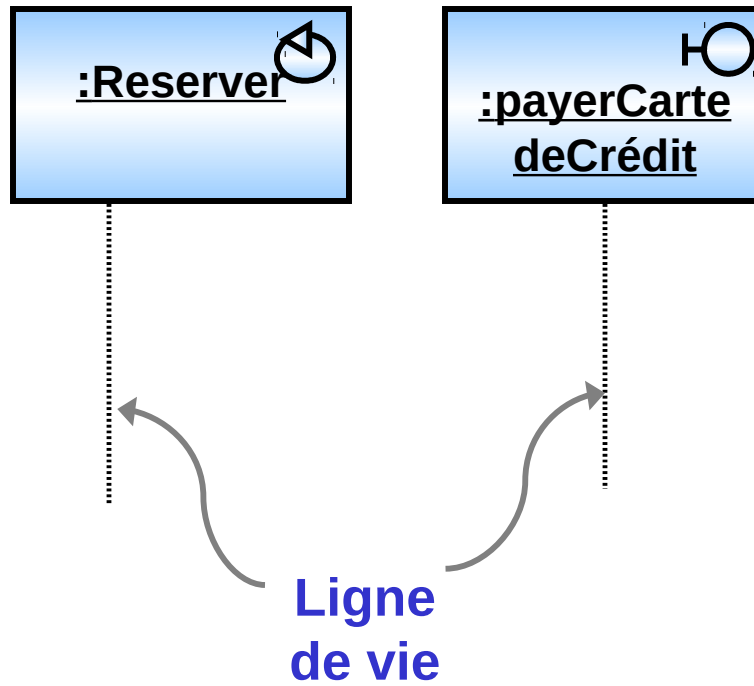




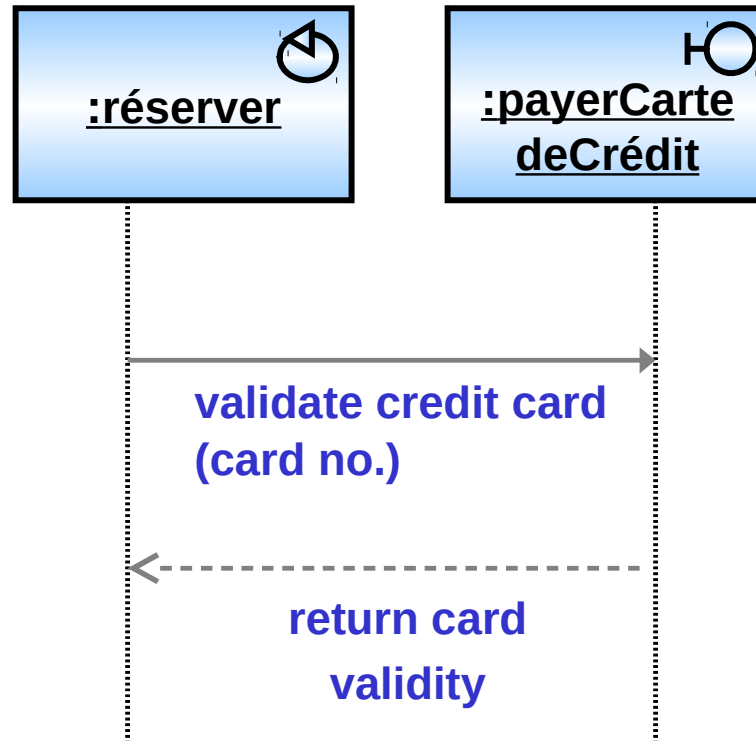
Notations



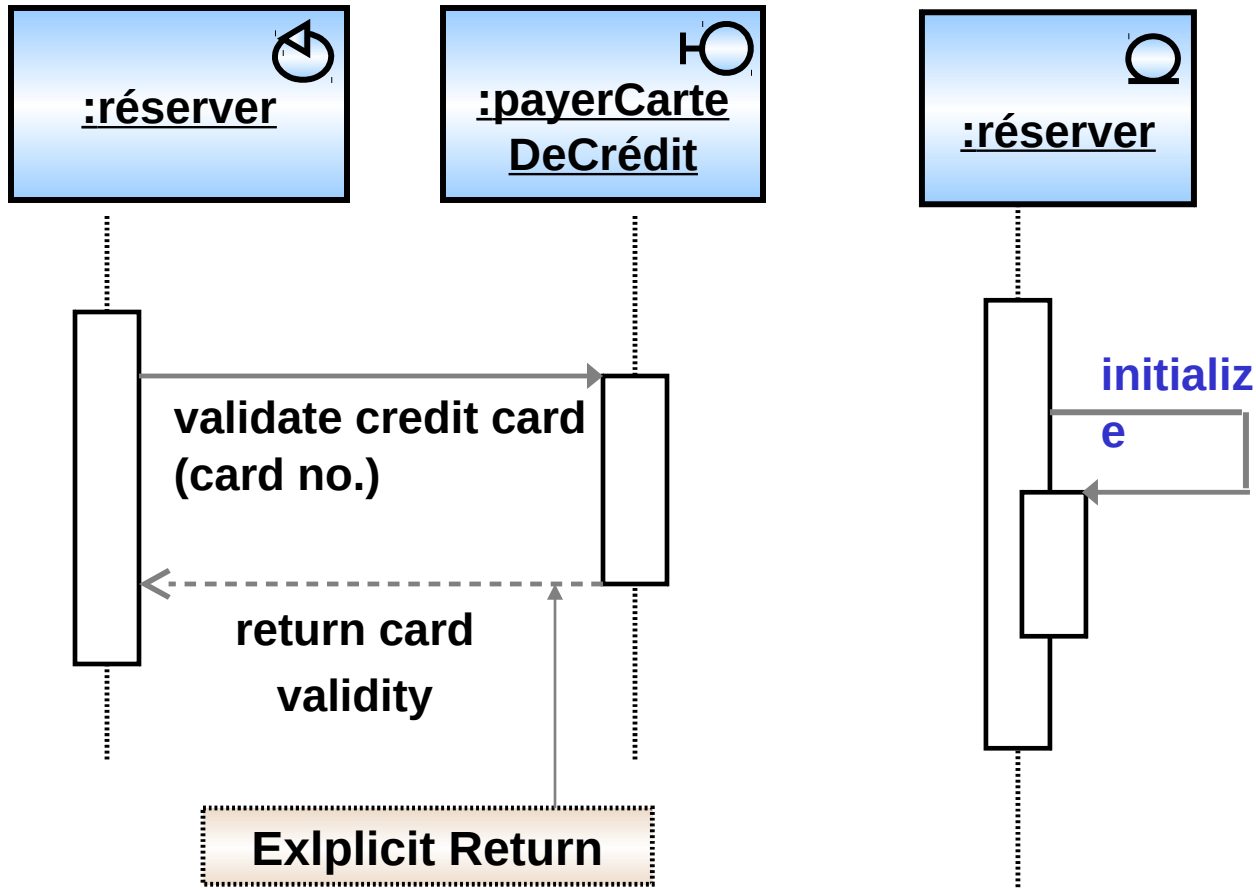
Lignes de vie



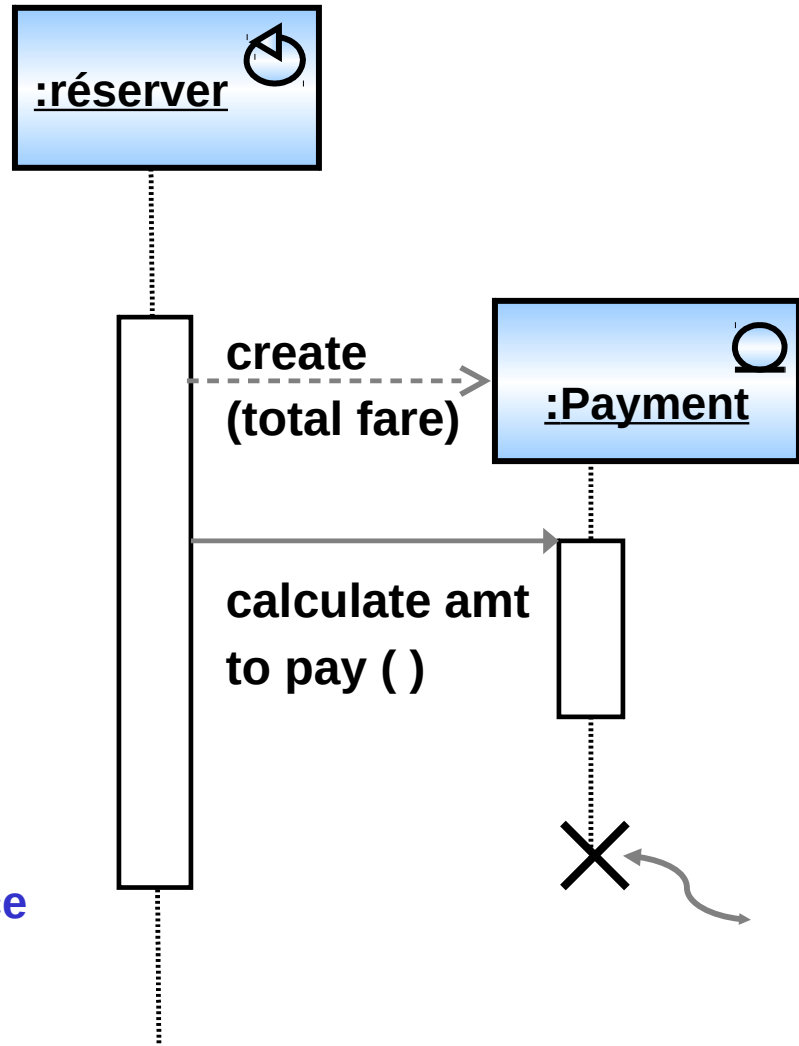
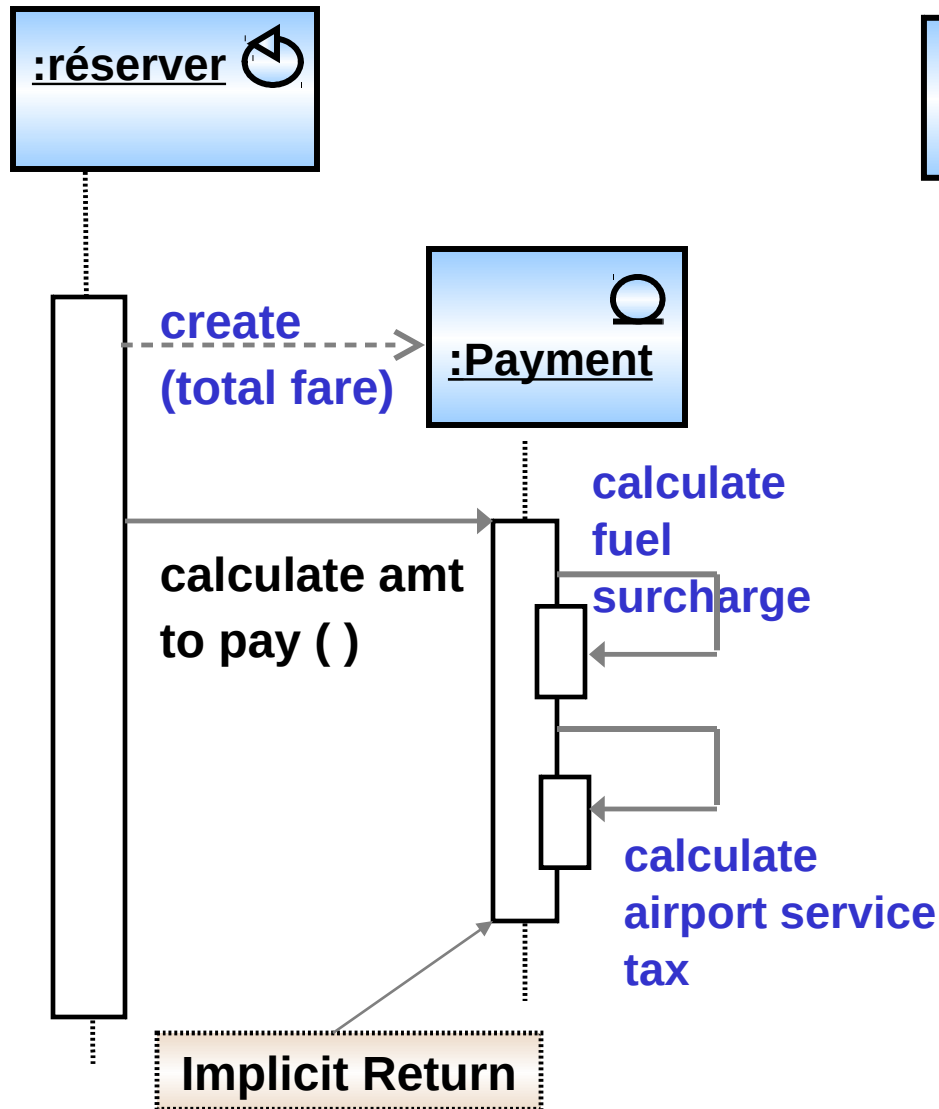
Messages



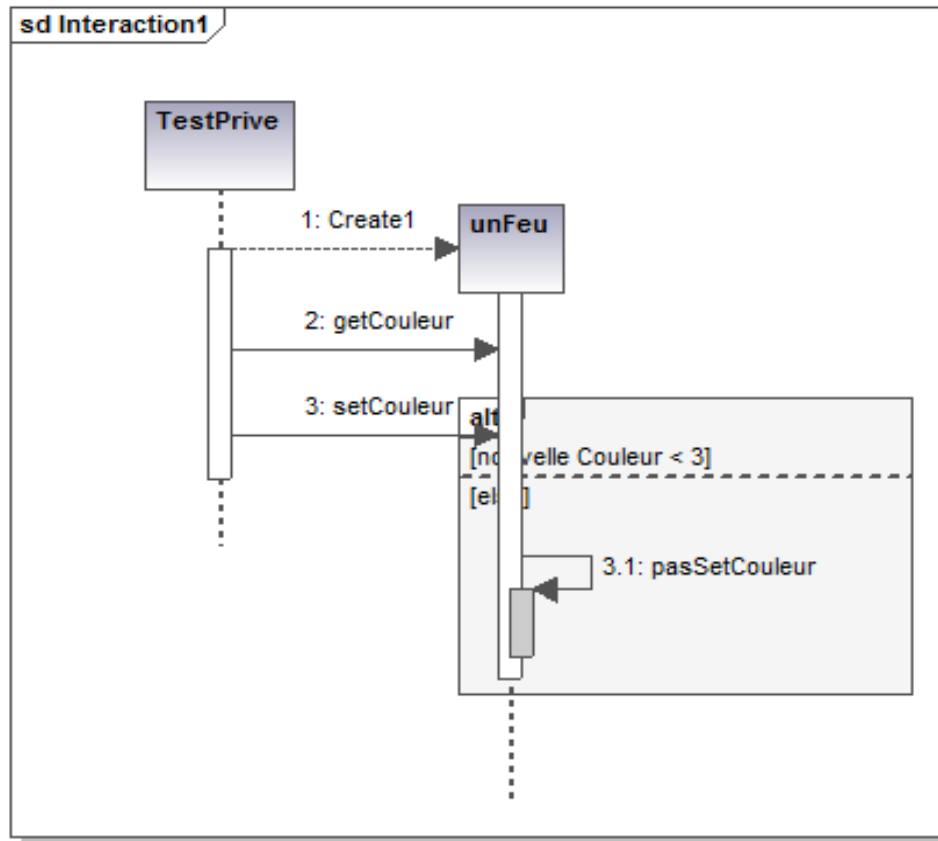
Barre d'activation



Création et destruction d'objets



Effet de l'encapsulation



Liaison avec le code

```

public class O1 {
    private int attributI;
    private O2 lienO2;
    private O3 lienO3;
    public void jeTravaillePourO1(int a) {
        if (a > 0){
            lienO2.jeTravaillePourO2();
        }
        else{
            lienO3.jeTravaillePourO3(a);
        }
    }
}

```

```

class O2 {
    private O3 lienO3;
    public void jeTravaillePourO2() {
        lienO3.jeTravaillePourO3(6);
    }
}

```

```

class O3 {
    private O2 lienO2;
    public void jeTravaillePourO3(int a) {
        if (a > 0){
            a--;
            jeTravaillePourO3(a);
        }
        else {
            lienO2.jeTravaillePourO2();
        }
    }
}

```

Click to edit Master text styles

Second level

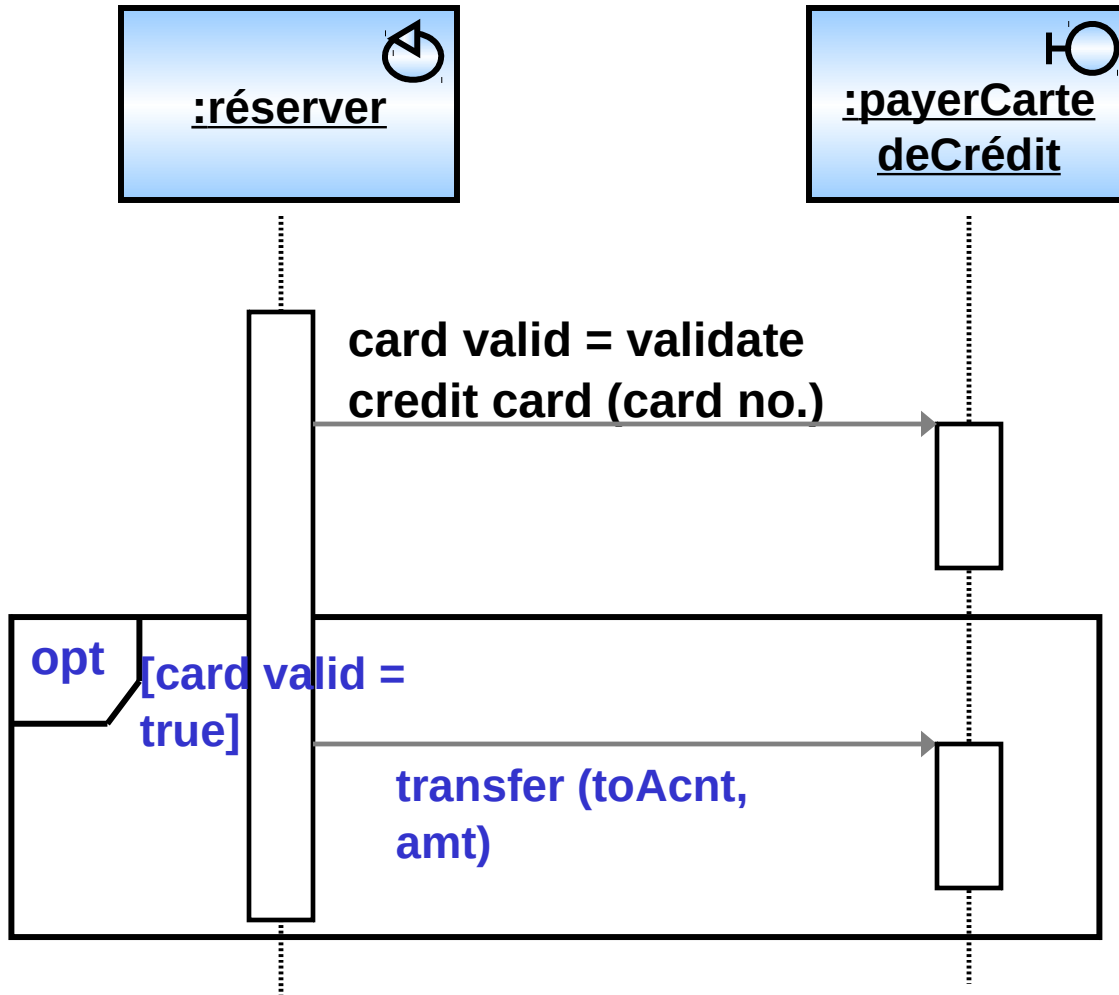
Third level

Fourth level

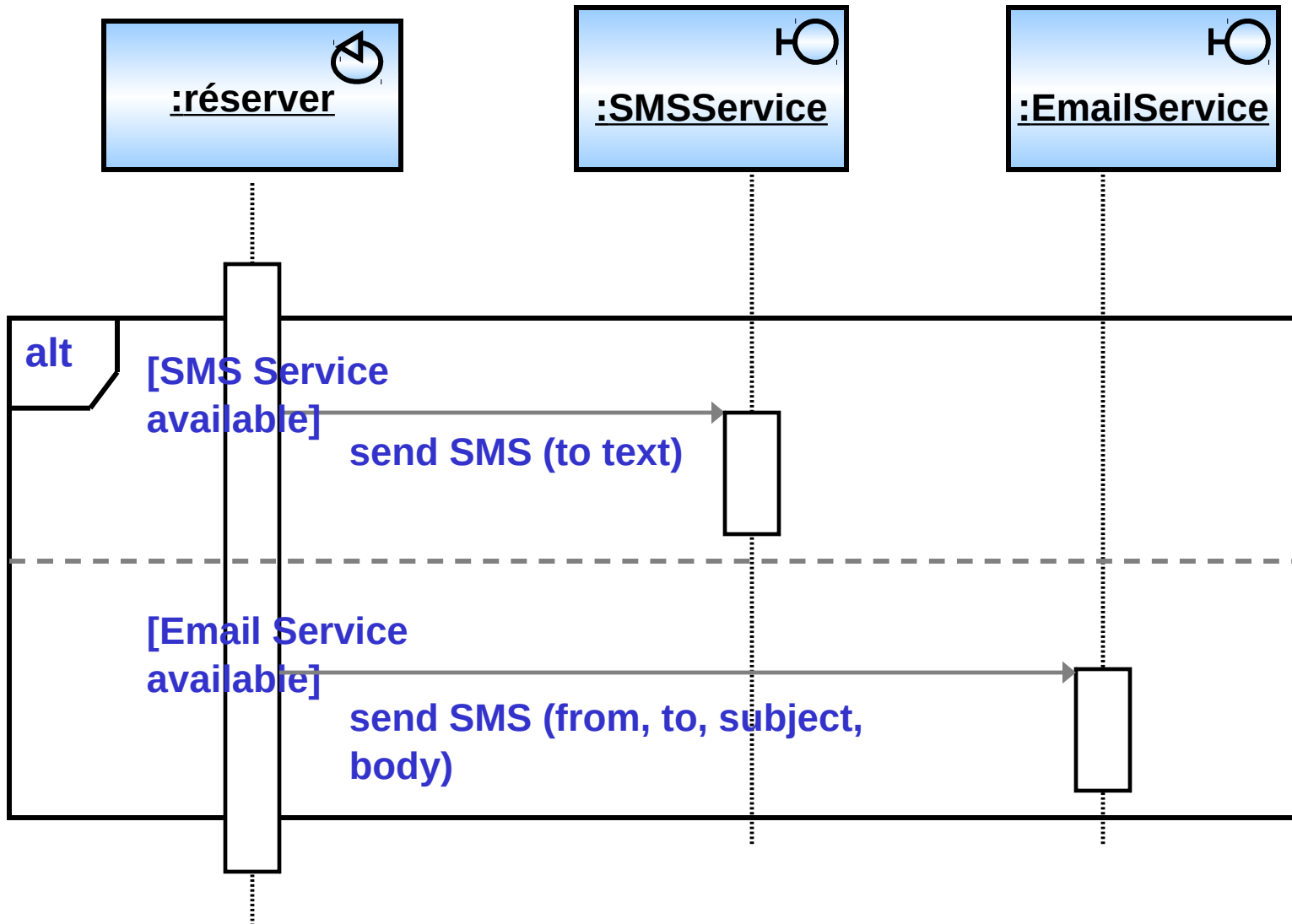
Fifth level

Nouvelle bouture UML2

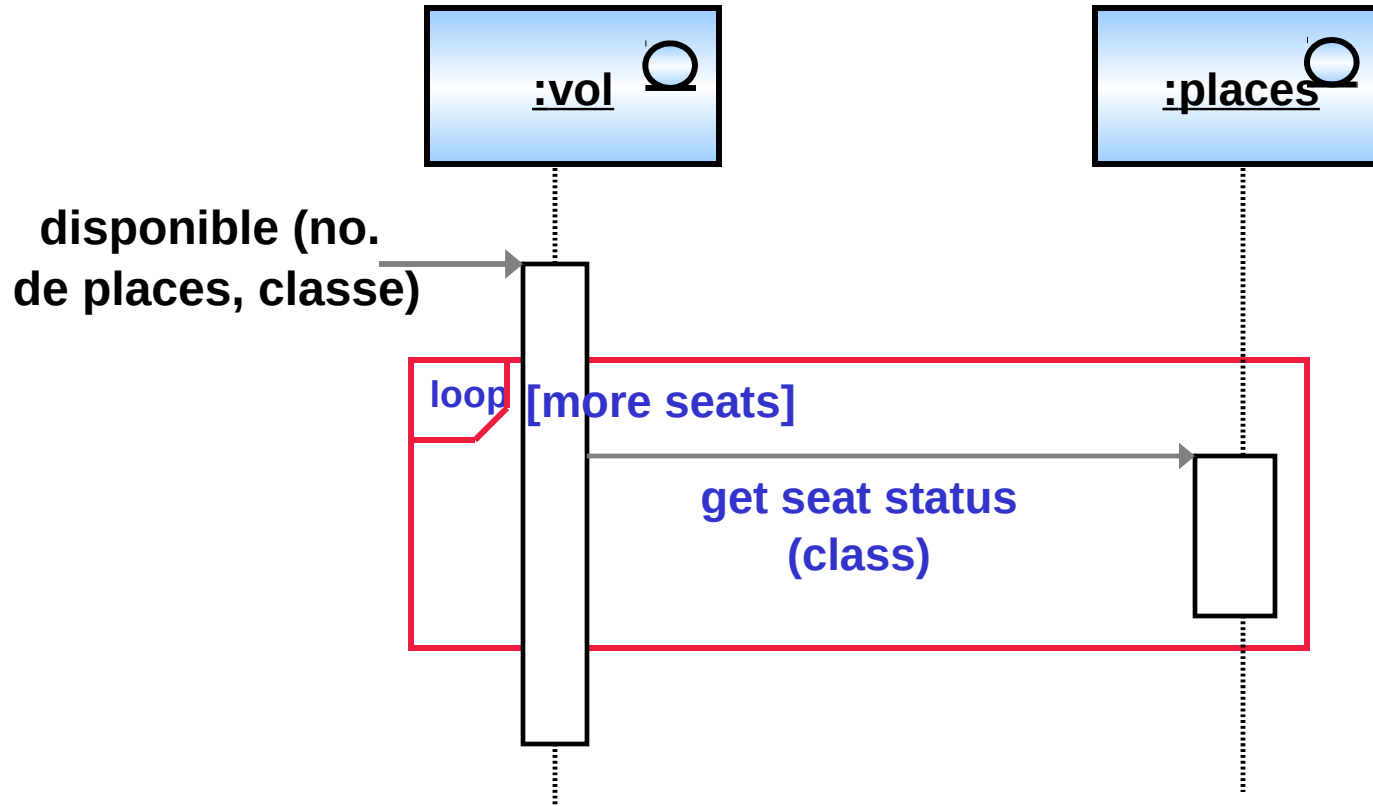
IF

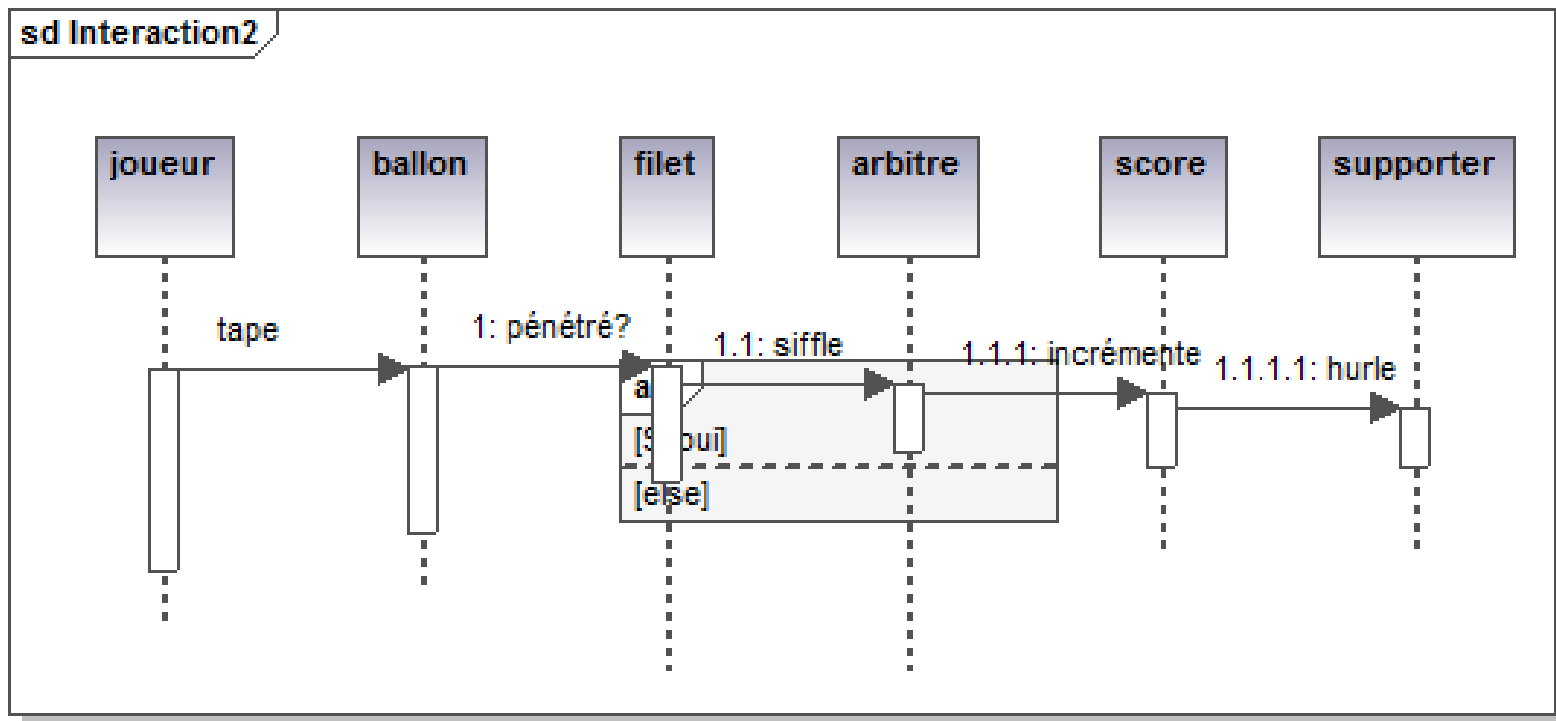


IF THEN ELSE



LOOP



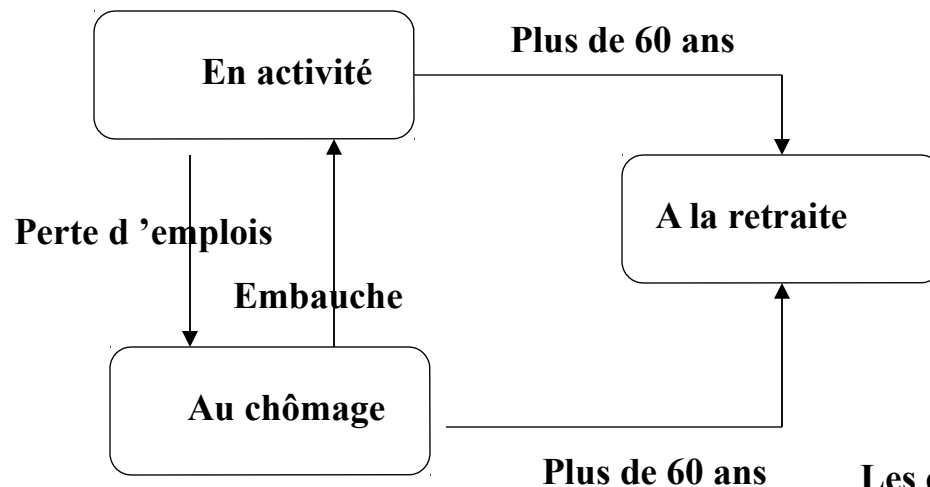


Les diagrammes d'états-transitions

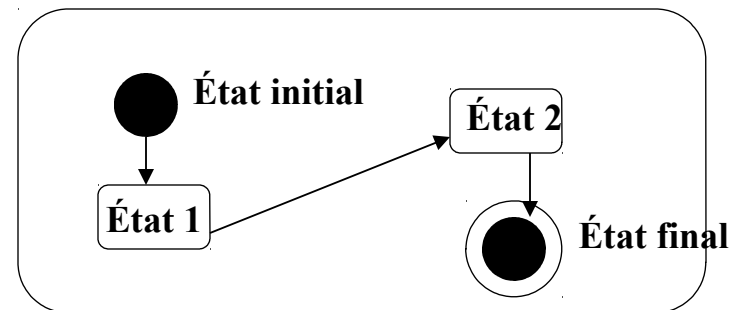
- Ce sont les diagrammes les plus parlant pour les aspects dynamiques du système
- Similaire aux automates à états finis
- Ils ne concernent que le cycle de vie d'une classe d'objets.
- Il s'agit de graphes dont les états sont les nœuds, les flèches sont les transitions, portant des paramètres et des noms d'évènements. Un événement cause une transition.
- L'état d'un objet c'est la valeur de ses attributs et la nature de ses relations avec les autres objets. C'est une durée séparé par deux événements.
- Un message est un événement impliqué dans l'interaction de deux objets.

Les diagrammes d'états (suite)

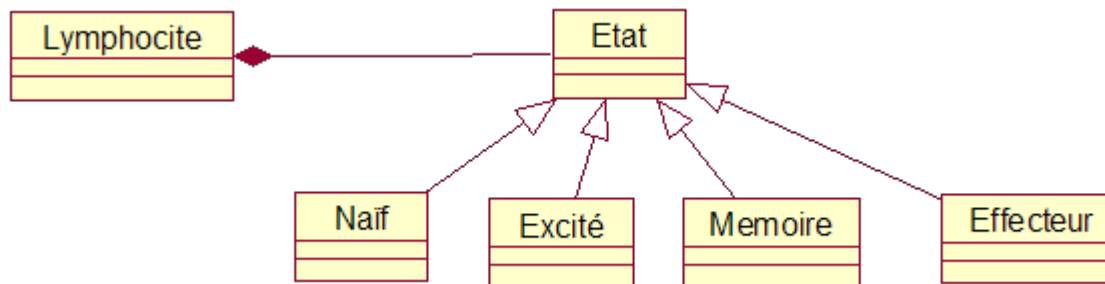
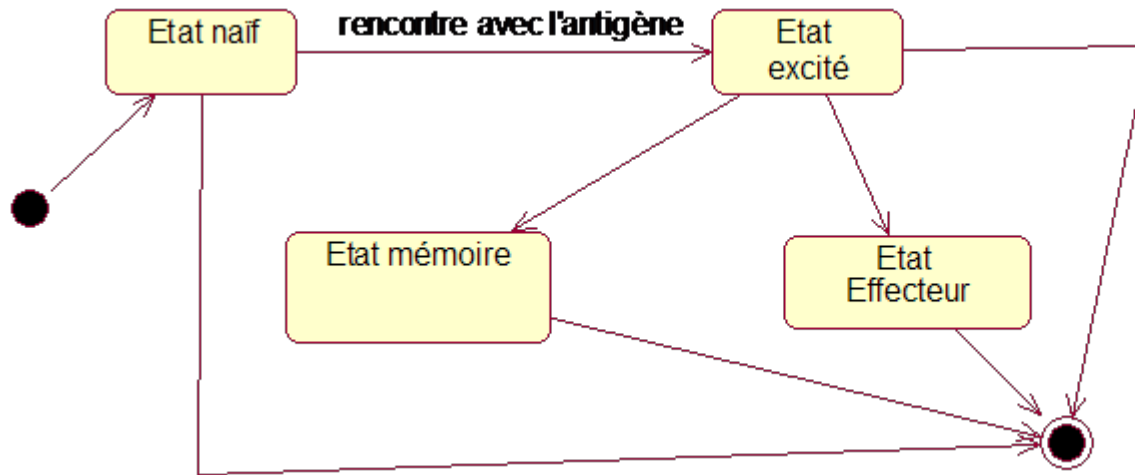
Classe: Personne



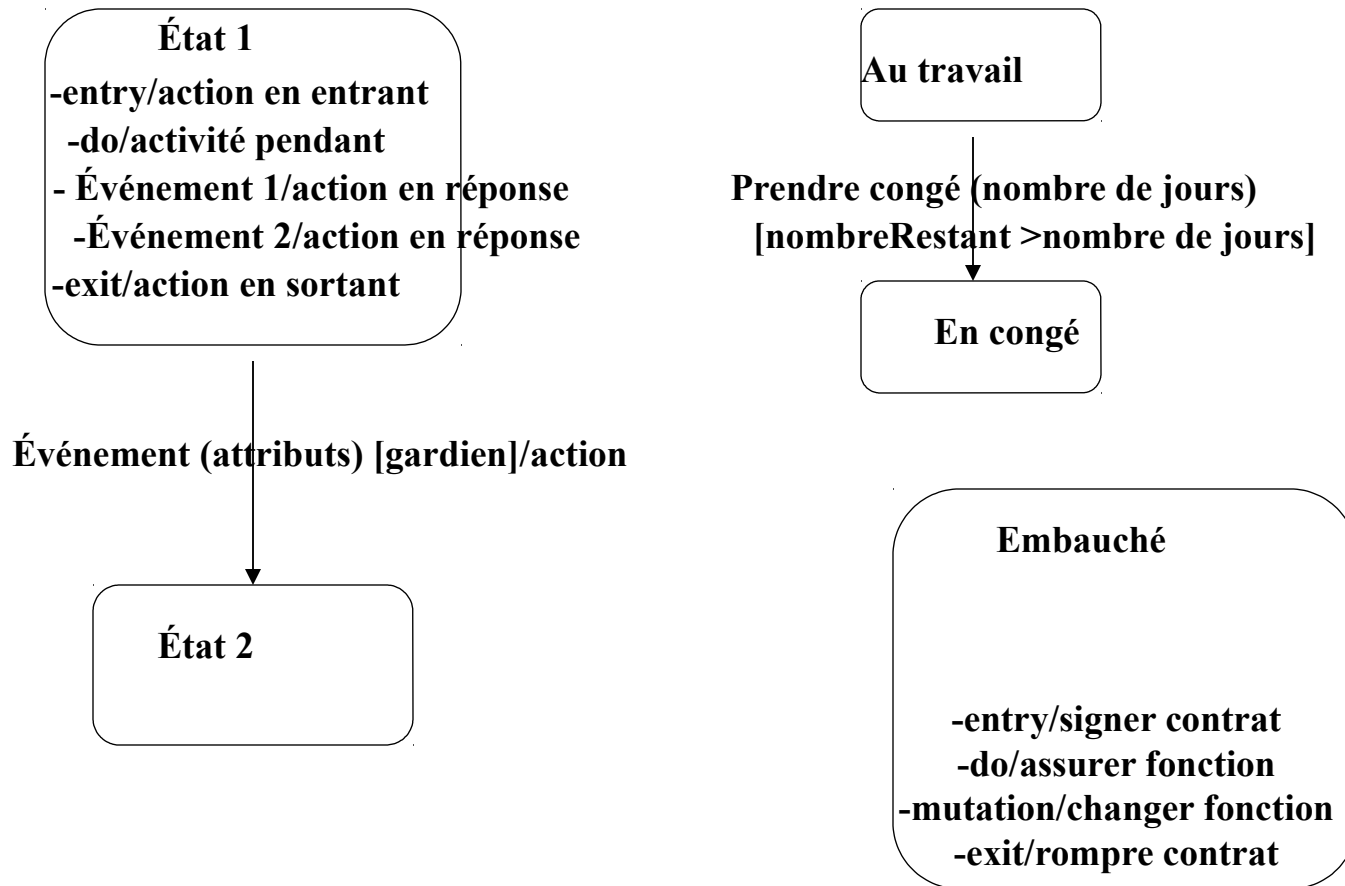
Les états peuvent être imbriqués

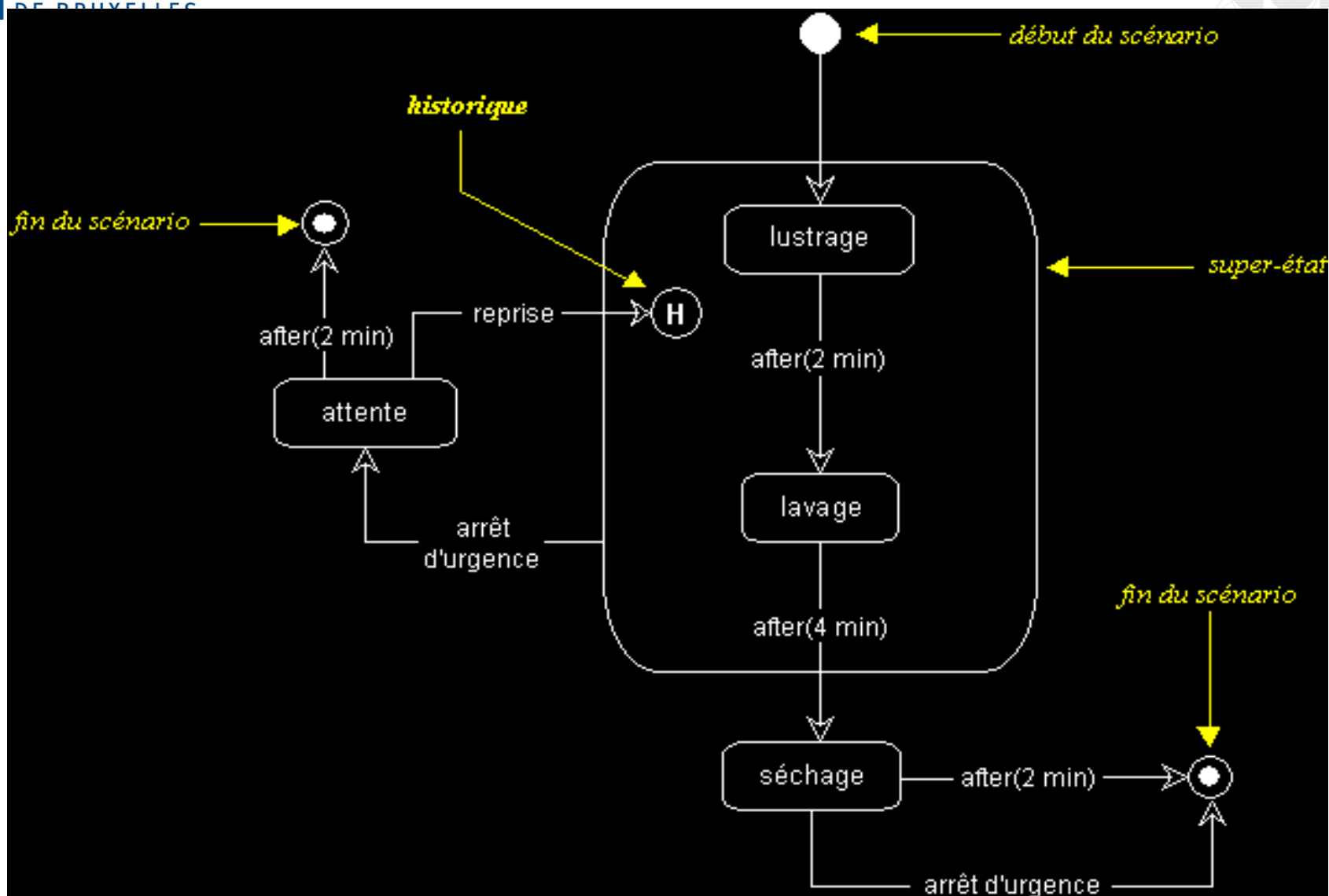


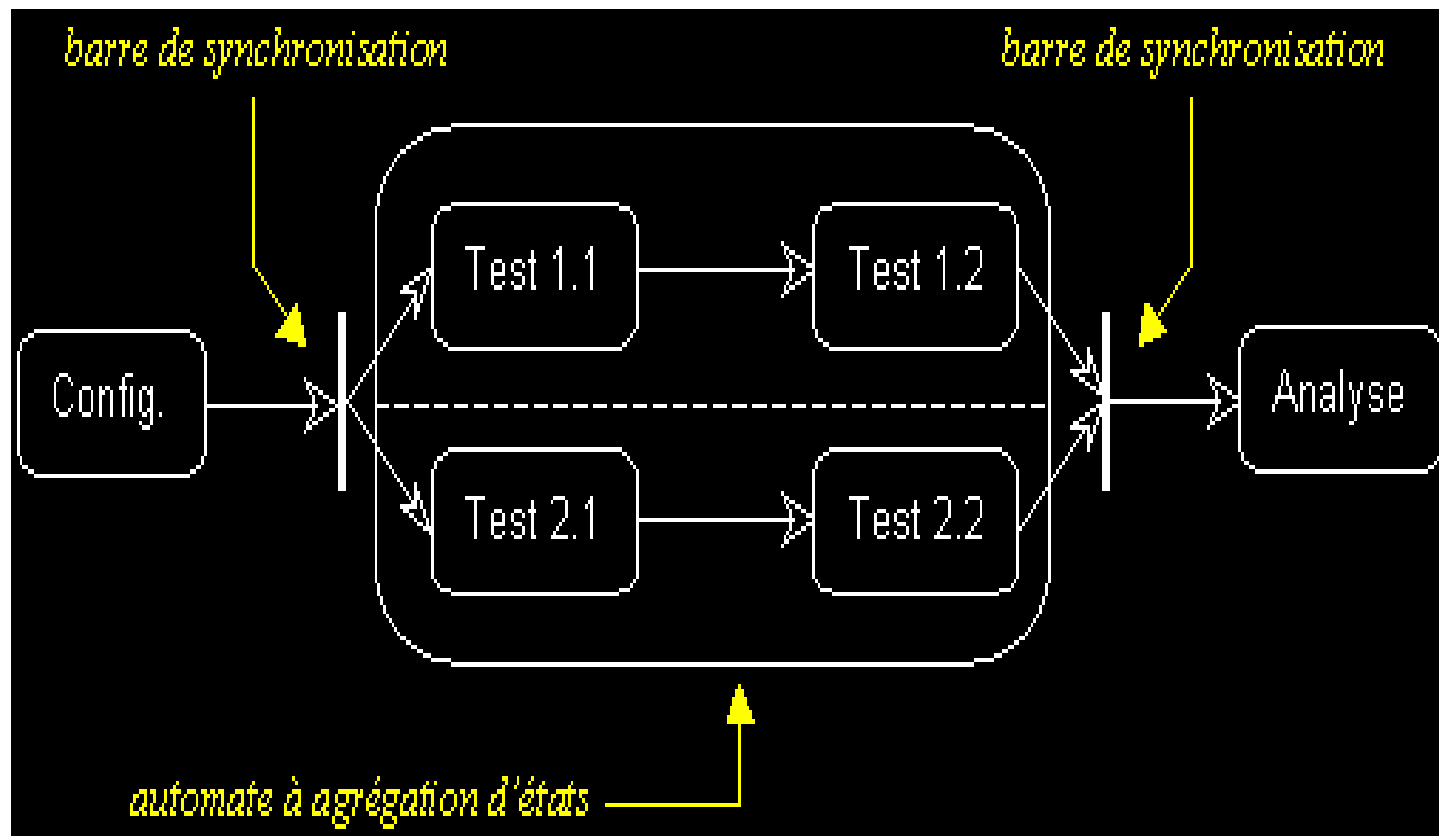
Le design pattern d'état en immunologie



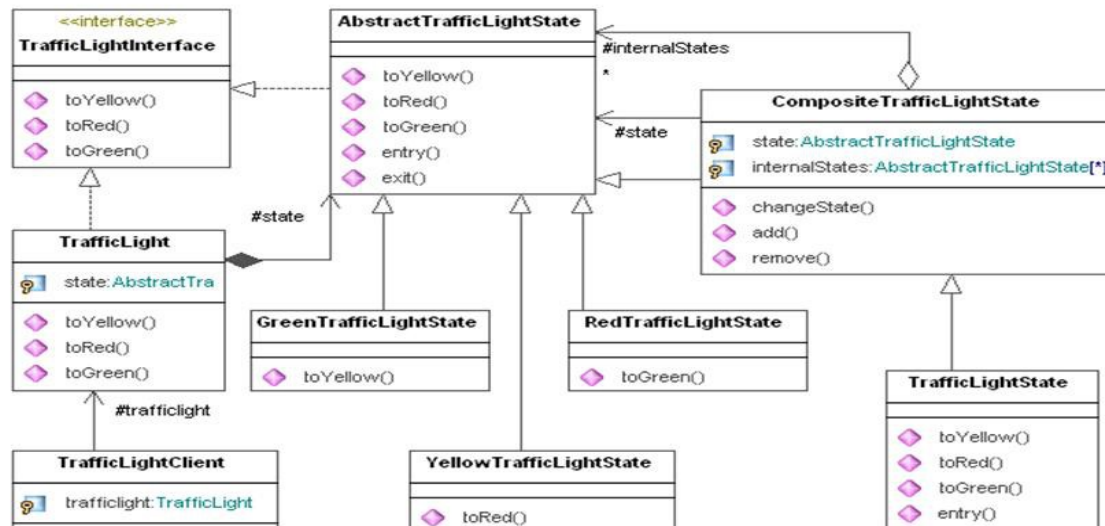
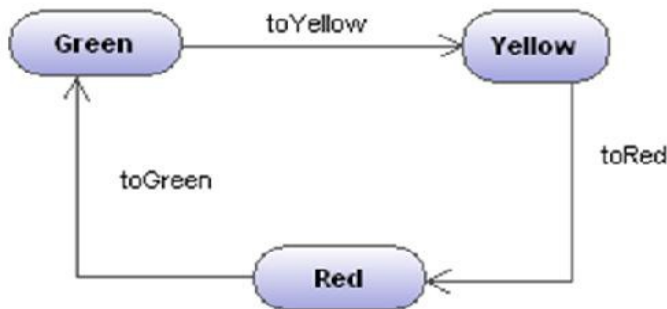
Les diagrammes d'état (suite)



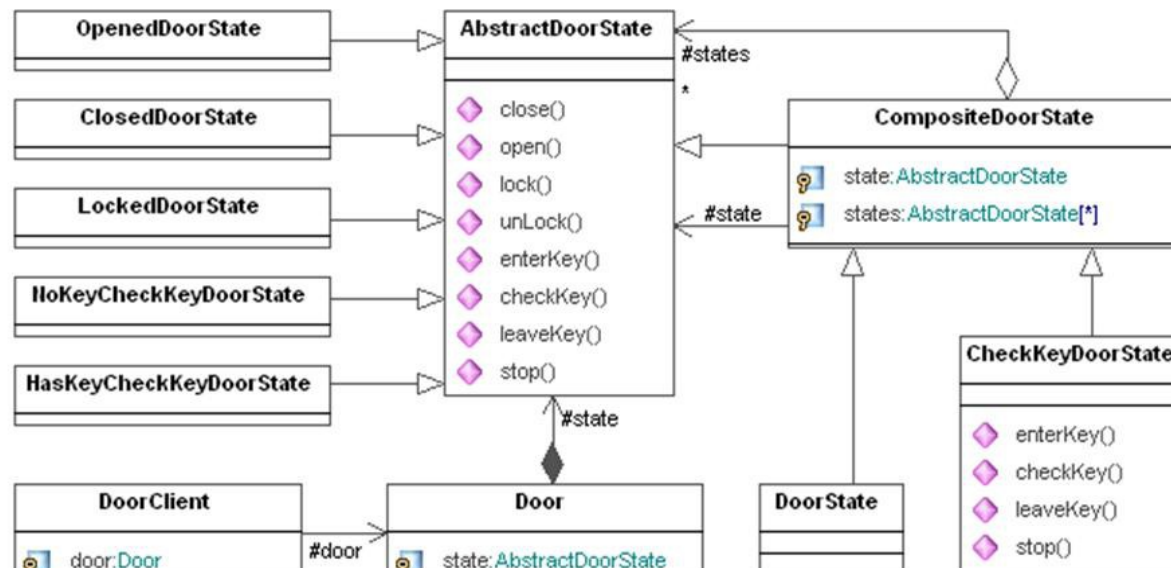
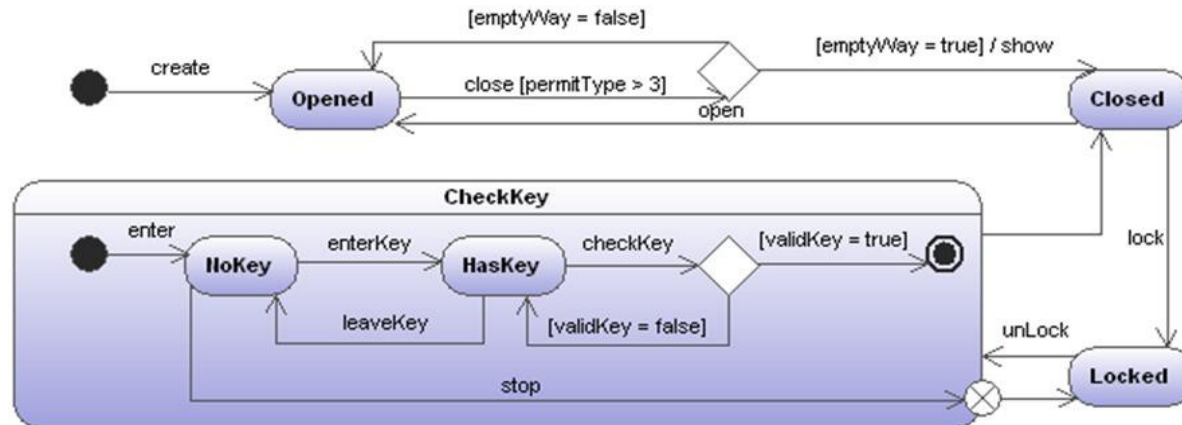




Génération de code (I)



Génération de code (2)



Exercice Java

- Voir livre OO chapitre 20
- Le match de tennis

Stockage des objets

- Les solutions immédiates gardant séparés le monde du relationnel et de l'OO.
 - Soit la persistance dans les langages OO
 - Soit des requêtes SQL dans les programmes OO
- On assiste à une succession d'étapes intermédiaires pour atteindre les bases de données OO
- L'idéal final étant une parfaite intégration entre les langages OO et les aspects solides des bases de données: permanence, intégrité, sécurité, concurrence, requête en langue naturelle, recovery, archivage, ...

Les deux solutions immédiates

- **Persistance:**

```
Date d = new Date();
```

```
FileOutputStream f = new FileOutputStream (« date.ser »);
```

```
ObjectOutputStream s = new ObjectOutputStream (f);
```

```
try {
```

```
    s.writeObject(d);
```

```
    s.close();
```

```
} catch (IOException e) ....
```

```
Date d = null;
```

```
FileInputStream f = new FileInputStream (« date.ser »);
```

```
ObjectInputStream s = new ObjectInputStream (f);
```

```
try{
```

```
    d = (Date)s.readObject ();
```

```
} catch (IOException e)
```

- **Intégration SQL**

```
public class QueryMyTable {  
  
    public static void main(String args[]) {  
  
        try {  
  
            String theURL = ' jdbc:odbc:dbname ' ;  
  
            Connection dbConn = DriverManager.getConnection (theURL, « myLogin », « passwd »)  
  
            Statement sqlStmt = dbConn.createStatement();  
  
            ResultSet rSet = sqlStmt.executeQuery (« Select xInt,Ystring FROM myTable »);  
  
            while (rSet.next()) {  
  
                int xIntVal = rset.getInt (« xInt »);  
  
                String yStrVal = rset.getString (« yStr »);  
  
                ..... }  
  
            sqlStmt.close();  
  
            dbConn.close();  
  
        }  
    }  
}
```

ObjectStore

Class Example {

public static void main (String args[]) {

ObjectStore.initialize();

Database db = Database.create ("C:\Employee.odb",

Database.allRead | Database.allWrite);

Transaction t = Transaction.begin(Transaction.update);

Employee e1 = new Employee ("John Smith", 50, 70000, null, null);

Employee subordinates[] = {e1};

Employee e2 = new Employee ("Jane Edwards", 40, 90000, subordinates, null);

e1.manager = e2;

db.createRoot("theBoss", e2);

t.commit();

db.close();

}

```
Transaction tr = Transaction.begin(ObjectStore.Update);
```

```
Employee b = aCollection.query("Employee", "name == Jane Edwards");
```

```
System.out.println("TheBoss is " + b.name + ", earns " + b.salary + ", has"  
                  + b.subordinates.length)
```

```
for (int i=0; i< b.subordinates.length; i++) {  
    b.subordinates[i].salary = Math.random();  
}
```

```
tr.commit();  
}
```


Le pilote “JDBC”

- Intégration SQL

```

public class QueryMyTable {

    public static void main(String args[]) {

        try {

            String theURL = 'jdbc:obdc:dbname ';

            Connection dbConn = DriverManager.getConnection (theURL, « myLogin », « passwd »)

            Statement sqlStmt = dbConn.createStatement();

            ResultSet rSet = sqlStmt.executeQuery (« Select xInt,Ystring FROM myTable »);

            while (rSet.next()) {

                int xIntVal = rset.getInt (« xInt »);

                String yStrVal = rset.getString (« yStr »);

                ..... }

            sqlStmt.close();

            dbConn.close();

        }
    
```

JDBC

- C'est un niveau d'abstraction qui permet une interaction indifférenciée avec la grande majorité des bases de données relationnelles
- JDBC est compatible ANSI-SQL2 et très proche de ODBC
- JDBD nécessite un pilote de communication avec la BD qui réalise la communication, l'envoi de requêtes SQL et la réception de "ResultSet".
- Il faut d'abord installer le pilote:
 - `Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");`
- Puis lancer la connection: `DriverManager.getConnection (url)`

JDBC (suite)

Application
JDBC Driver Manager
JDBC-ODBC Bridge 
ODBC Driver Manager
ODBC Driver Libraries



- Ensuite on crée un “statement”:
 - `Statement stmt = con.createStatement();`
- Sur lequel on exécute soit un “executeUpdate” soit un “query”
 - `ResultSet rs = stmt.executeQuery("Select * From Client");`
- Le “ResultSet” pointe à la première ligne et par “next” circule dans le vecteur. “get” accède aux colonnes.
- `getDouble, getObject, getShort, getByte, getInt, ...`

Quelques questions méthodologiques

- Doit-on séparer le comportement propre à l'application des aspects bases de données ??
- Les classes doivent-elles contenir tous les détails qui permettent d'interagir avec la base de données ou alors ne rien savoir et laisser cela à un module externe ?
- Dans le premier cas, la classe possède les méthodes nécessaires à la sauvegarde des données. Mais les différences entre l'OO et le relationnel rendent cette solution difficile. Dans l'autre cas, les deux aspects sont bien tenus séparés.

Les bases de données OO



L 'OO et le schéma relationnel ??

- Une équivalence attirante mais inexacte est: Classe = Table et Objet = enregistrement.
- Cela devient d 'autant plus faux que l 'on s 'interesse aux aspects relationnels.
- Et cela même si on se maintient aux données statiques, sans parler de méthodes, héritage, polymorphisme
- Les relations sont plus explicites et plus riches en OO. Les pointeurs jouent un rôle capital. Un join est différent d 'un attribut d 'une classe.
- L 'objet préserve une identité malgré des changements d 'attributs.
- En cours: développement d 'un ODBMS standardisé qui intègre les langages OO et les systèmes de BD.



Table Personne

<u>nom</u>	<u>Id</u>	<u>IdPers</u>
	15	15
	29	29
		15

Table adresse

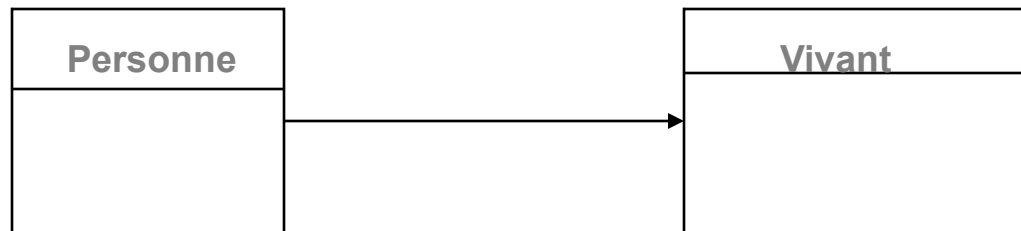
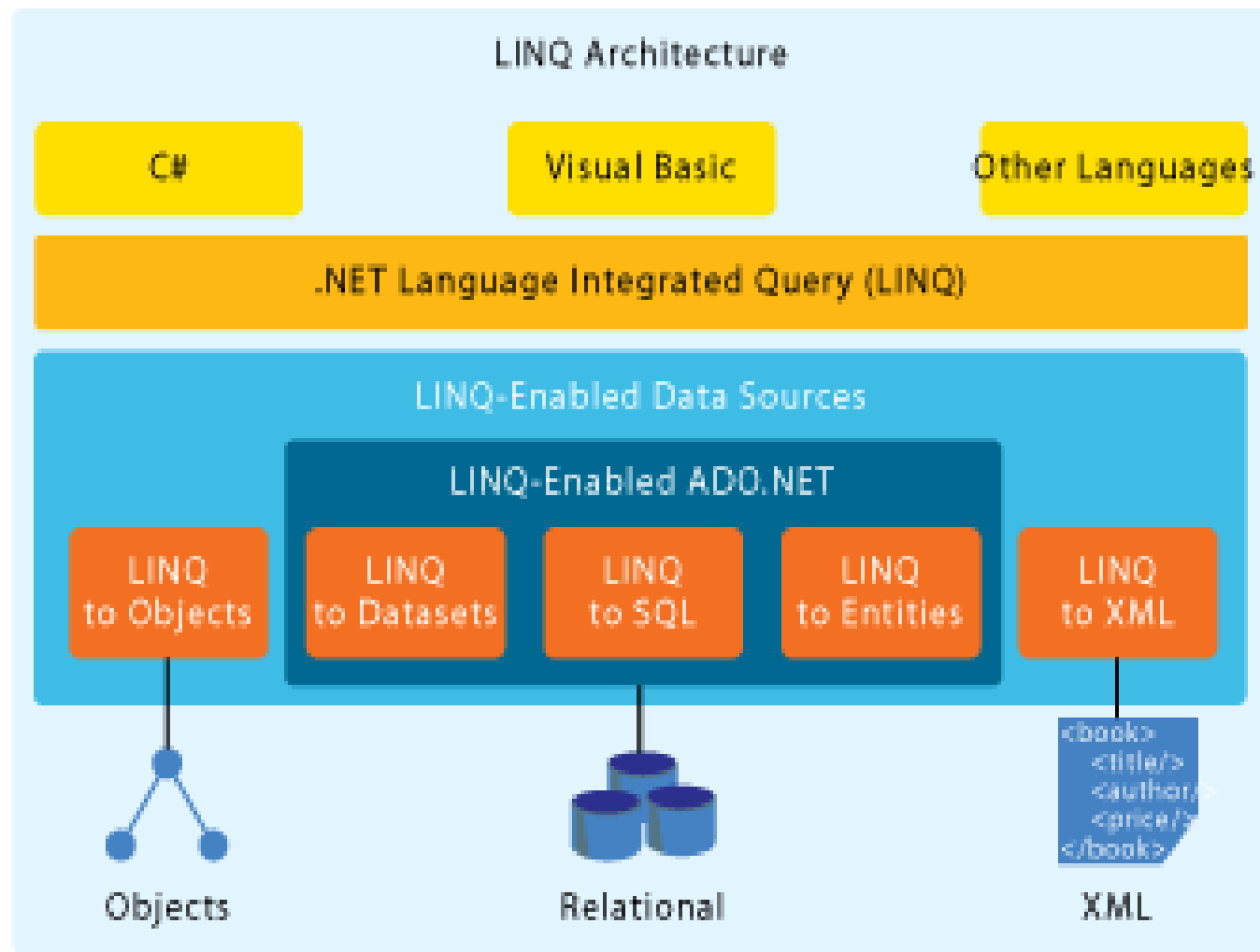


Table Personne

<u>nom</u>	<u>Id</u>	<u>IdPers</u>	<u>age</u>
	15	15	
	29	29	
		31	

- L 'ODBMS intégrerait dans les langages OO la persistance, le contrôle concurrent, le « data recovery », les requêtes type SQL, et autres...
- Il y a donc un recentrage des bases de données sur les langages OO
- Mais l 'OODBMS reste une technologie émergente.
- Aujourd 'hui, les règles d 'intégrité, l 'optimisation de l 'accès, l 'archivage, le backup et les « recovery » ne sont pas « automatisés »
- La standardisation se fait par l 'ODMG mais la situation reste confuse (voir C.J. Date contre l 'ODMG).



LINQ → rapprocher les lambda expression de SQL → Query expression

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

class app {
    static void Main() {
        string[] names = {"Burke","Connor","Frank","Everet","Albert","George",
            "Harris","David"};

        var test = from s in names
                    where s.Length == 5 // On filtre avec WHERE
                    orderby s // On ordonne avec ORDERBY
                    select s.ToUpper(); // On projette avec SELECT

        /* SQL à l'envers pour les « complétions » automatiques */
        foreach (var item in test) Console.WriteLine(item); /* C'EST AU MOMENT DU FOREACH
        QUE SE PASSE L'EXECUTION DE LA REQUÊTE LINQ*/
    }
}
  
```

```
class Etudiant {  
    public Etudiant (String prenom, String nom, int age) {  
        Prenom = prenom;  
        Nom = nom;  
        Age = age; }  
    public int Age {  
        get; set; // admirez le nouveau raccourci de syntaxe C#3  
    }  
    public String Nom {  
        get ; set;  
    }  
    public String Prenom {  
        get; set;  
    }  
}
```

```
class app {  
    static void Main() {  
        List<Etudiant> lesEtudiants = new List<Etudiant> ();  
        lesEtudiants.Add(new Etudiant("Paul","Lenders",25));  
        lesEtudiants.Add(new Etudiant("Pierre", "Bertrand", 22));  
        lesEtudiants.Add(new Etudiant("Pascale", "Aimée", 28));  
        lesEtudiants.Add(new Etudiant("Marcel", "VanDamme", 18));  
        var test = from s in lesEtudiants  
                    where s.Age > 20  
                    orderby s.Nom  
                    select new {s.Prenom, s.Nom}; /*D'où le besoin du "var" ;*/  
        foreach (var item in test) Console.WriteLine(item.Prenom);  
    }  
}
```

Résultat

Pascale
Pierre
Paul

```
static void Main() {
    string[] names = {"Burke", "Connor", "Frank", "Everet", "Albert", "George",
        "Harris", "David"};

    IEnumerable<string> test1 = Enumerable.Where(names, s=> s.Length == 5);

    //Vient de: static IEnumerable<TSource> Where<TSource>
    //      (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
    // Extension method !! IEnumerable<string> f = names.Where(s=> s.Length == 5);

    foreach (string item in test1) Console.WriteLine(item);

    IEnumerable<String> test2 = from s in names
        where s.Length == 5
        orderby s
        select s.ToUpper(); //Syntaxe « Query comprehension » !!

    foreach (string item in test2) Console.WriteLine(item);

    var test3 = from s in names
        where s.Length == 5
        orderby s
        select s.ToUpper(); // Utilisation de var !!

    foreach (string item in test3) Console.WriteLine(item);
}
```

Réécriture du compilateur

```
private static void test () {  
    IEnumerable<string> selection = from word in list  
                                   where word.Contains(« * »)  
                                   select word;  
}
```

DEVIENT

```
private static void test () {  
    IEnumerable<string> selection =  
        list.Where(word=>word.Contains(« * »));  
}
```

Linq to SQL

```
using System.Linq;
using System.Data;
using System.Data.Linq;
using System.Data.Linq.Mapping;
using System.Data.OleDb;

[Table] public class compteEnBanque
{
    [Column(IsPrimaryKey=true)] public int numero;
    [Column] public string type;
    [Column] public int solde;
    [Column] public int taux_interet;
    [Column] public string titulaire;
}

class SomeDataContext : DataContext
{
    public SomeDataContext(IDbConnection connection) : base (connection) {}
}
```



```
class Test
{
    static void Main() {
        string strConnection = "Provider=Microsoft.Jet.OleDb.4.0;";
        strConnection += "Data
Source=C:\\C#\\CodeC#\\C#\\compteEnBanque.mdb";
        OleDbConnection connexion = new
OleDbConnection(strConnection);
        SomeDataContext dataContext = new SomeDataContext(connexion);
        Table<compteEnBanque> comptes =
dataContext.GetTable<compteEnBanque>();

        IQueryable<string> query = from c in comptes
            where c.solde > 0
            select c.titulaire;

        foreach (string titulaire in query) Console.WriteLine (titulaire);
    }
}
```

- `dataContext.comptes.InsertOnSubmit(p) ;`
 `//p serait un nouveau compte`
- `dataContext.SubmitChanges() ;` `//`
 exécution de l'insertion
- `dataContext.comptes.DeleteOnSubmit(p) ;` `//`

`p` serait l'objet à supprimer
- `dataContext.SubmitChanges() ;` `//exécution de`
 la suppression

Exemple Java

- Petite application bancaire
- Les comptes en banques et les clients sont dans une base de données relationnelle.

Les Objets Distribués: RMI et CORBA

- Traitement réparti sur le réseau internet
- OMG ---> CORBA
- CORBA se veut indépendant des plateformes, des OS et des langages
- Microsoft: OLE---> ActiveX
- Java RMI uniquement java mais indépendant également des plateformes et des OS
- La grande différence avec CORBA est donc la flexibilité additionnelle du langage.
- Les objets s'envoient des messages à travers le réseau

RMI - Un exemple - 3 fichiers

- L'interface serveur:

```
public interface Hello extends java.rmi.Remote {  
    String sayHello() throws java.rmi.RemoteException;  
}
```

- L'implémentation serveur:

```
import java.rmi.*;
```

```
import java.rmi.server.UnicastRemoteObject;
```

```
public class HelloImpl extends UnicastRemoteObject implements Hello  
{
```

```
    private String name;
```

```
    public HelloImpl(String s) throws RemoteException {
```

```
        super ();
```

```
        name = s;
```

```
    }
```

```
public String sayHello() throws RemoteException  
{  
    return "Hello World";  
}
```

```
public static void main(String args[])  
{  
    try {  
        System.out.println ("Je commence");  
        HelloImpl obj = new HelloImpl("HelloServer");  
        Naming.rebind("HelloServer", obj);  
        System.out.println("HelloServer enregistré dans le registry");  
    } catch (Exception e) {System.out.println("HelloImpl err: " + e.getMessage());  
        e.printStackTrace();} } }
```

L'implémentation client:

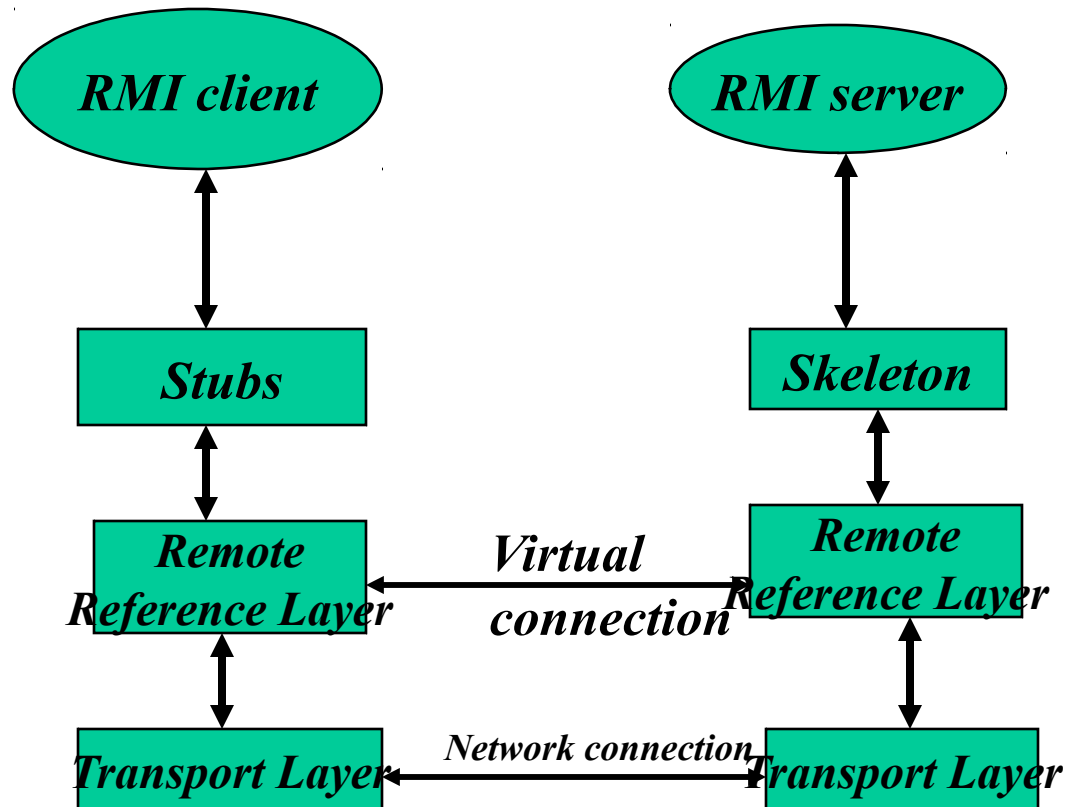
```
import java.rmi.*;
import java.awt.*;
import java.net.*;
import java.io.*;

public class HelloClient {
    static String message = "";
    public static void main (String args[])
    {
        try { System.out.println ("Hello je teste");
            Hello obj = (Hello)Naming.lookup("rmi://localhost/HelloServer");
            message = obj.sayHello();
        } catch (Exception e) {System.out.println ("HelloClient exception: " + e.getMessage());
            e.printStackTrace();}
        System.out.println(message); }}
```


RMI (suite)

- L'objet serveur doit s'enregistrer dans un "registry" (Naming.rebind ...) avec un certain nom.
- Le client obtient une référence à l'objet et communique avec lui via cette référence.
- Protocole hiérarchisé
- Le Remote Reference Layer s'occupe des références objets.
- Le Stub est un proxy pour le client représentant le serveur
- Le skeleton agit comme un interface entre le reference layer et l'objet serveur.

L'architecture RMI



RMI: la séquence d'instruction

- Il faut compiler le serveur
- Créer le stub et le skeleton avec RMIC
- Deux nouveaux fichiers .class sont créés
- Lancer le RMI Registry
- exécuter l'application serveur
- Compiler et exécuter l'application client

CORBA (Common Object Request Broker Architecture) - Bus d'objets répartis

- Norme d'applications distribuées prenant en compte: la communication, l'hétérogénéité, l'intégration et l'interopérabilité
- Intégration: il est possible de récupérer des applications "legacy" par l'encapsulation de celles-ci dans des objets ré-utilisables
- Plus universel mais plus complexe que RMI
- Défini et standardisé par l'OMG
- Middleware: vaste gamme de services logiciels (tous objets) pour construire des applications distribuées

CORBA (suite)

- Rendre l'appel distant à des méthodes semblable à l'appel local.
- Un stub client (sert de proxy serveur pour le client), il “emballe” l'appel et reçoit la réponse.
- Un skeleton serveur déballe l'appel, appelle la méthode locale et renvoie les résultats.
- Le stub et le skeleton sont générés automatiquement (comme pour RMI) à partir d'une définition IDL (Interface Definition Language) de l'objet serveur.
- La notion d'interface (existant en java) est fondamentale: les opérations effectuées par le serveur.

CORBA (suite)

- Le langage IDL est indépendant de tout langage de programmation et peut être projeté dans chacun de ces langages. Il contient des définitions de:
 - module = package de java
 - interfaces qui peuvent être hérités entre eux
 - les opérations (méthodes): nom, type de retour, le nombre de paramètres + le type et le mode de passage (in, inout, out)
 - les attributs
 - les exceptions = java et C++
 - les constantes
 - les types: élémentaires, composés, types dynamiques (pour les templates du C++)

CORBA (suite)

- Architecture globale:
 - le bus d'objets répartis: ORB
 - les services objet communs: Common Object Services: nommage, persistance, cycle de vie, transactions, sécurité
 - utilitaires communs: CORBA facilities: logiciels de haut niveau pour l'interface utilisateur, la gestion de l'information, administration du système
 - les interfaces de domaine: pour des segments d'activité tels que la santé, la finance, les télécoms ou les échanges commerciaux = objets métiers

CORBA (suite)

- Le bus d'objets répartis: Object Request Broker:
 - les invocations sont transparentes: quelles soient locales ou distantes. Indépendantes des langages ou des OS
 - Il y a deux types d'invocation:
 - statique (très proche de RMI)
 - dynamique: plus complexe car il n'y a pas de stub capturant la fonctionnalité de l'objet serveur. On peut découvrir sa fonctionnalité pendant le cours même de l'exécution - grâce à un référentiel des interfaces. A partir des informations extraites, un programme peut construire dynamiquement les invocations. C'est parfait quand les objets changent souvent ---> plus de flexibilité
 - Différentes stratégies d'activation automatique des objets: un processus par objet, un processus partagé, ou un processus par exécution de méthode. Communication standard entre différents bus

CORBA (suite)

- L'architecture de l'ORB:
 - Le noyau de communication
 - L'interface d'invocation statique: le Stub
 - L'interface d'invocation dynamique: construit les requêtes dynamiquement
 - le référentiel des interfaces: auto-description des définitions de l'IDL
 - l'interface du bus: initialisation de CORBA et paramétrage
 - l'interface de squelette statique: le Skeleton
 - l'interface de squelette dynamique
 - l'adaptateur d'objets; exécution des requêtes: plusieurs supports d'objets: processus, DB - BOA (Basic Object Adaptor)
 - Le référentiel des implantations: les exécutable , les droits d'accès

CORBA (suite)

- Les services d'objet communs sont codés en IDL:
 - Cycle de vie: création, copie, destruction des objets du bus
 - Nommage: désigne les objets par des noms symboliques
 - Persistance
 - Concurrence d'accès: mécanismes de verrous
 - Interrogations: SQL3
 - Propriétés: associés des propriétés aux objets (comme aux fichiers)
 - Sécurité: authentifier les clients des objets
 - Vendeur: comme les pages jaunes
 - Changements : suit l'évolution des objets

Un exemple CORBA

```
Module ExampleI { // installé dans le fichier ExampleI.idl
  const long Happy = 1;

  const long SAD = 2;

  typedef long mood;

  exception VeryBadMood {
    long codeErreur;
    string messageErreur;
  };

  struct Job2{
    string identity;
    string message;
  }
```

```
Interface Job {  
    readonly attribute string identity;  
    attribute string message;  
};  
  
Interface ObjetDistant {  
    string getTime (in long tps l);  
    string sayHello (inout mood theMood) raises (VeryBAdMood);  
    Job2 sayPhrase (in long tps l);  
    Job sayPhraseBis(in long tps l);  
};  
};
```

- Compilation idl ---> java
- idltojava ExampleI.idl

- **19 fichiers sont générés**
- ObjetDistant.java, _ObjetDistantImplBase.java, _ObjetDistantStub.java,
- ObjetDistantHelper.java, ObjetDistantHolder.java
- Job.java, _JobImplBase.java,
- HAPPY.java, SAD.java
- Job2.java, Job2Helper.java, ...
- VeryBadMood.java, ...
- VeryBAdMoodHelper.java,

```
class ObjetDistantServant extends _ObjetDistantImplBase
{
    private ORB orb;

    public objetDistantServant (ORB orb) {
        this.orb = orb;
    }

    public string SayHello(org.omg.CORBA.IntHolder theMood) throws
    ExempleI.VeryBadMood{ if (theMood.value ....) theMood.value =....
    String s = new String(...)
    return s;
}
```

```

public ExempleI.Job2 sayPhrase (long tpsI)....
{
    ExempleI.Job2 job2 = new ExempleI.Job2();
    job2.identity = ...
    job2.message = .....
    Return job2;
}

public ExempleI.Job sayPhraseBis (long tpsI) {
    org.omg.CORBA.Object objRef = orb.resolve.....("NameService");
    NamingContext ncref = NamingContextHelper.narrow(objRef);
    NameComponent nc = new NameComponent ("Job");
    NameComponent path[] = {nc};
    Job jobRef = JobHelper.narrow(ncRef.resolve(path));}
    return jobRef;}

```



```
public class CorbaExServeur {  
    private static ORB orb;  
    public static void main(String args[])  
    {  
        orb = ORB.init(...);  
        ObjetDistantServant odref = new ObjetDistantServant (orb);  
        orb.connect(odref);  
        org.omg.CORBA.Object objRef = orb.resolve.....("NameService"); NamingContext ncref =  
NamingContextHelper.narrow(objRef);  
        NameComponent nc = new NameComponent ("OD");  
        NameComponent path[] = {nc};  
        ncref.rebind(path,odref);  
  
        java.lang.Object sync = new java.lang.Object ();  
        synchronized(sync) {  
            sync.wait(); }  
    }  
}
```

```

public class CORBAExClient
{
    private static ObjetDistant odref;

    public static void main(String args[]) {
        ORB orb = ORB.init(...);
        org.omg.CORBA.Object objRef = orb.resolve.....("NameService"); NamingContext ncref =
NamingContextHelper.narrow(objRef);
        NameComponent nc = new NameComponent ("OD");
        NameComponent path[] = {nc};
        odref = ObjectDistantHelper.narrow(ncref.resolve(path));
        .....
        IntHolder moody;
        if (moody ...)
            String sbis = odref.sayHello(moody);
        if (moody ...);
    }
}

```

RMI vs CORBA

- RMI
 - tout java (+ langage Natif)
 - passage par référence pour des objets IDL
 - possible surcharge
 - supportent l'héritage
 - tous les types d'argument Java
 - le système de nommage = registry
 - RMI est entièrement polymorphique (téméchargement des classes)
 - certains services sont intégrés dans les API java.
 - Une unique distribution JAVA
- CORBA
 - toute plateforme, tout langage
 - passage par référence pour des objets IDL
 - pas de surcharge
 - supportent l'héritage (multiple sans conflit de noms)
 - tous les types d'argument C++
 - le système de nommage
 - CORBA n'est pas polymorphique
 - CORBA PROPOSE DES SERVICES
 - De multiples distributions CORBA avec des implémentations variées des services

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 données)
 - **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 appliquée à la conception logicielle.
- Réduire les interactions entre les objets, diminuer le temps de développement
 - → gagner des sous ;-)
- Aussi appelé **design pattern** ou motif de conception

Patrons de conception

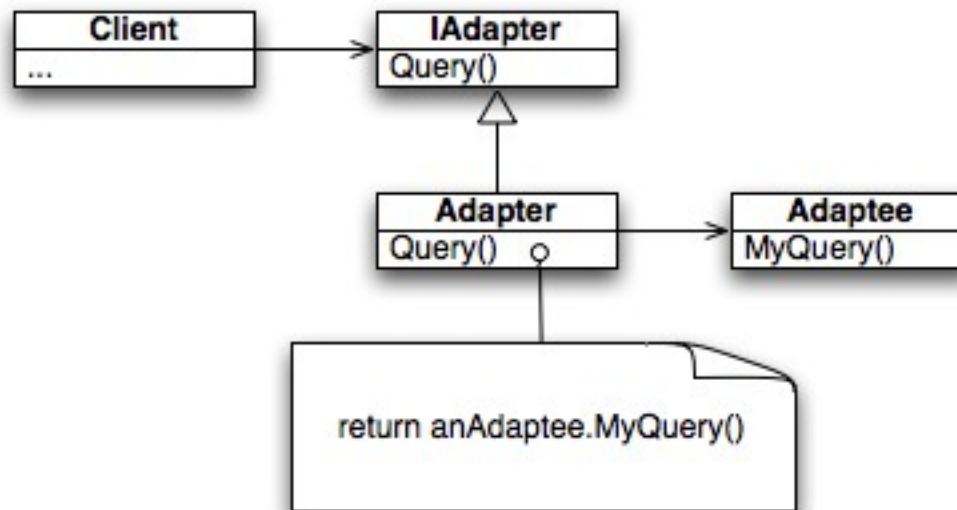
- Formalisés en 1995 par le « Gang of Four »
- Trois grandes familles :
 - **Patrons de construction** : *instancier* et configurer des classes et des objets
 - **Patrons structuraux** : organiser les classes d'un programme en séparant l'*interface* de l'implémentation
 - **Patrons comportementaux**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

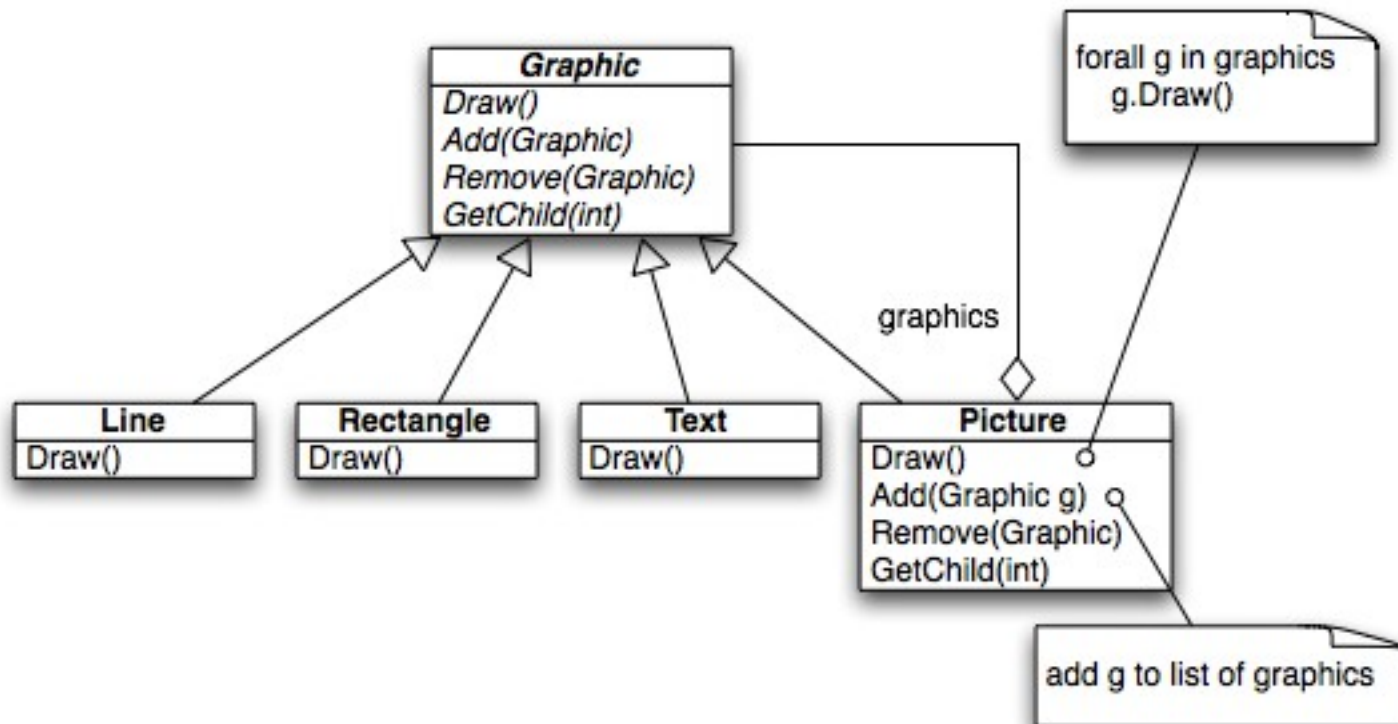
- Convertir **l'interface** d'une classe en une autre interface que le client attend



Composite

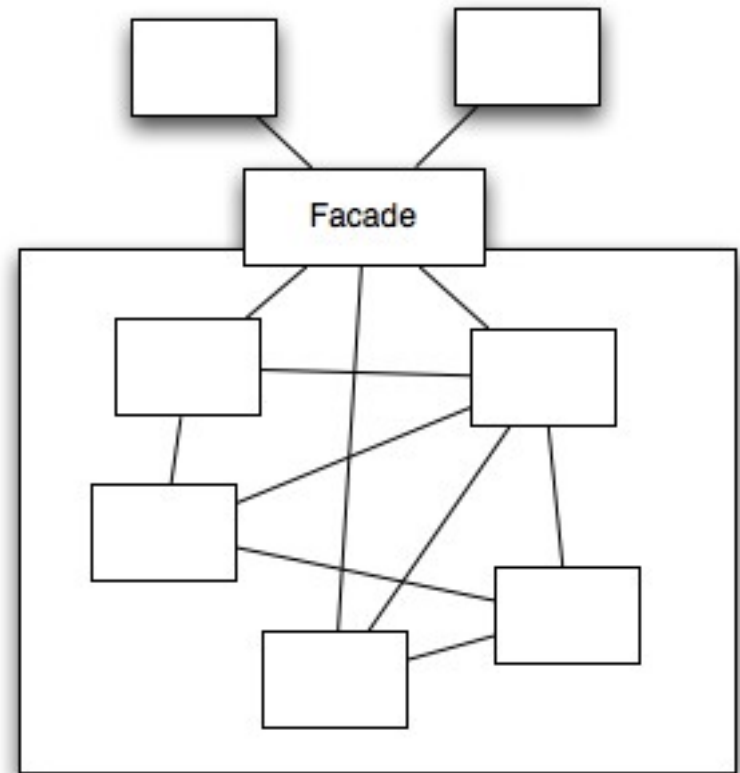
- Un objet composite est constitué d'un ou de **plusieurs objets similaires** (ayant des fonctionnalités similaires)
- L'idée est de **manipuler un groupe** d'objets de la même façon que s'il s'agissait **d'un seul objet**
- Les objets regroupés doivent avoir des **méthodes communes**.
- Souvent utilisé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

Composite



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.

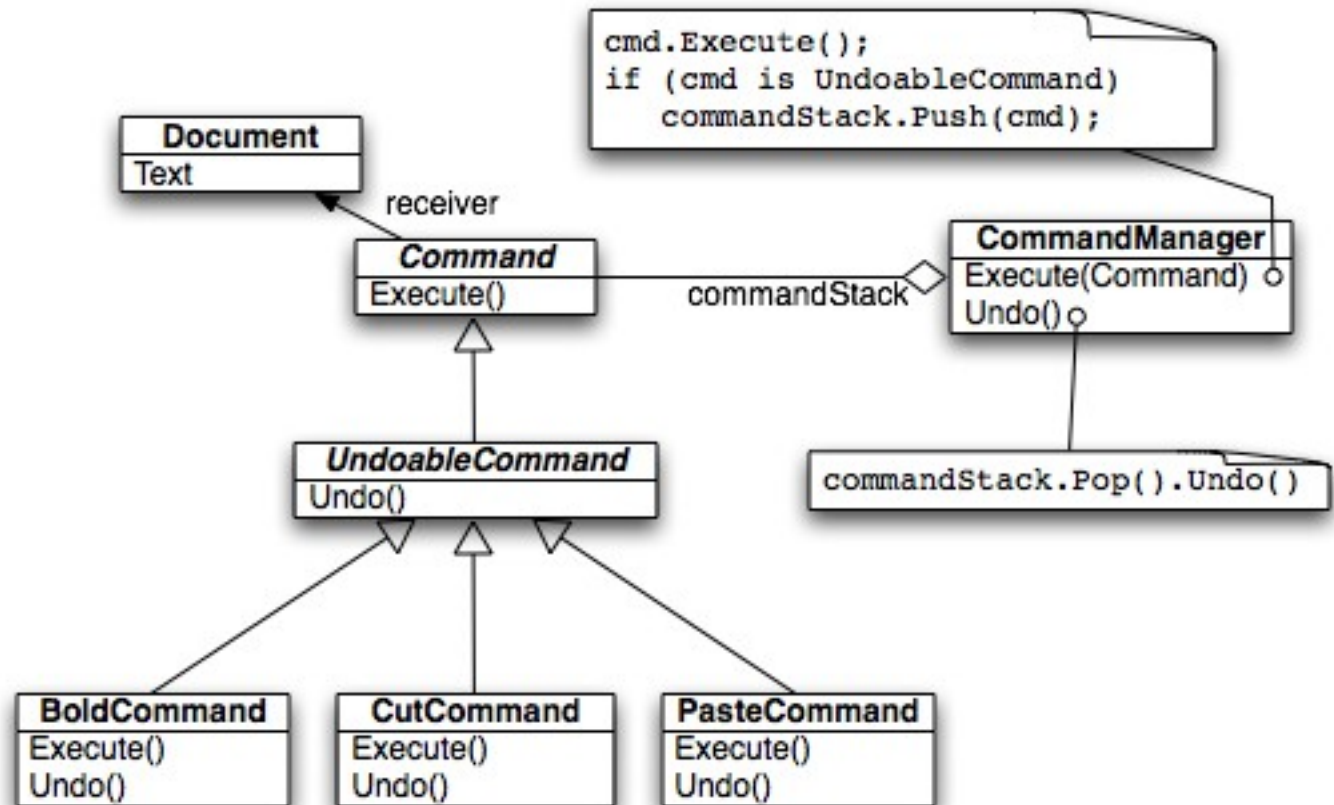


Quelques patrons comportementaux

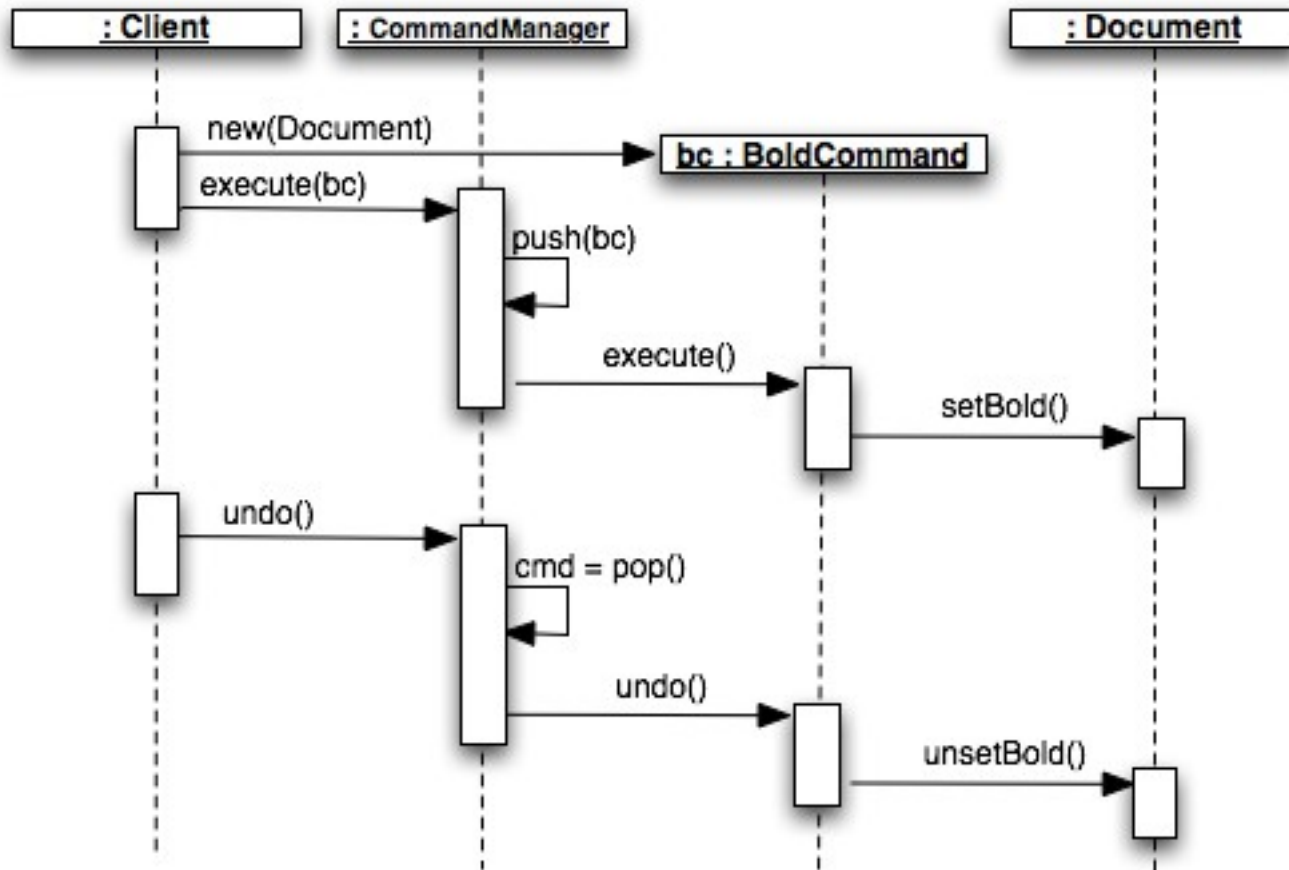
Organiser les objets pour que ceux-ci
collaborent

Command

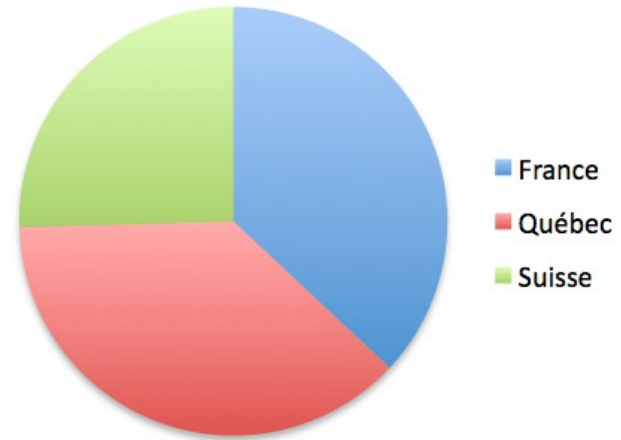
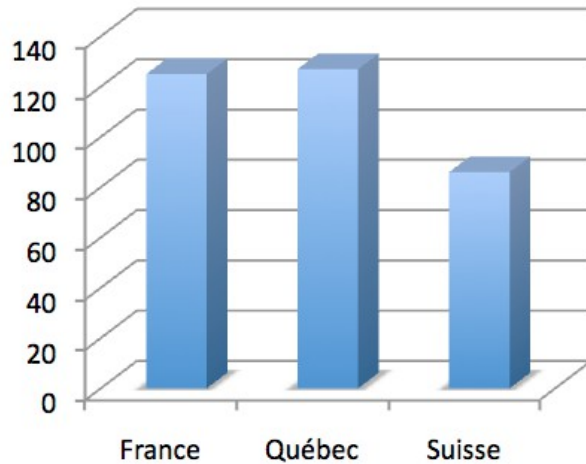
- **Encapsuler une requête/commande comme un objet.**
- Ex :
 - Annuler une commande
 - undo()



Command



Patron de conception Observer



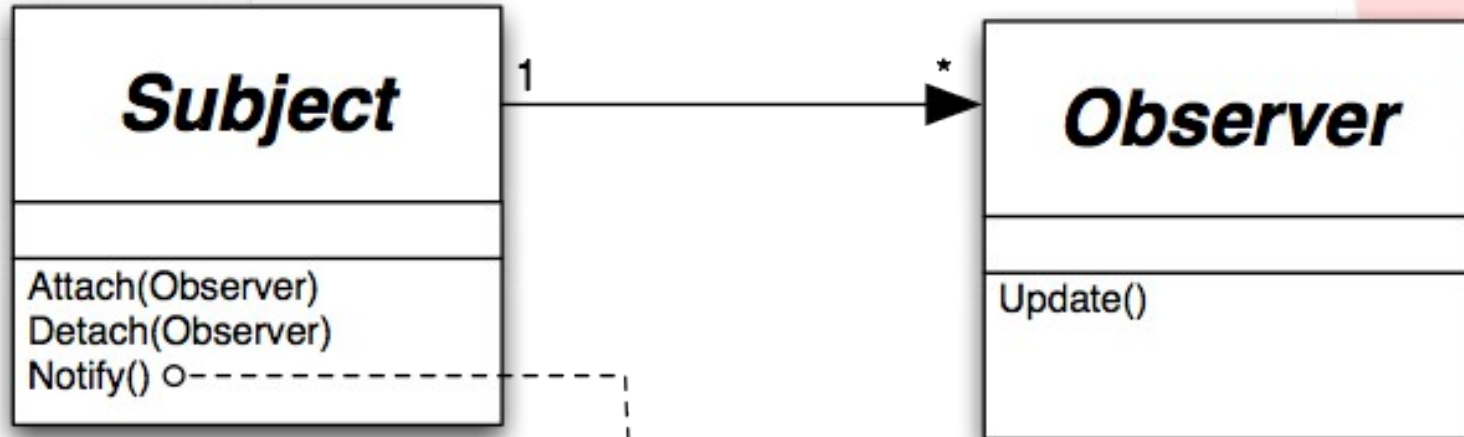
France	125
Québec	127
Suisse	86

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

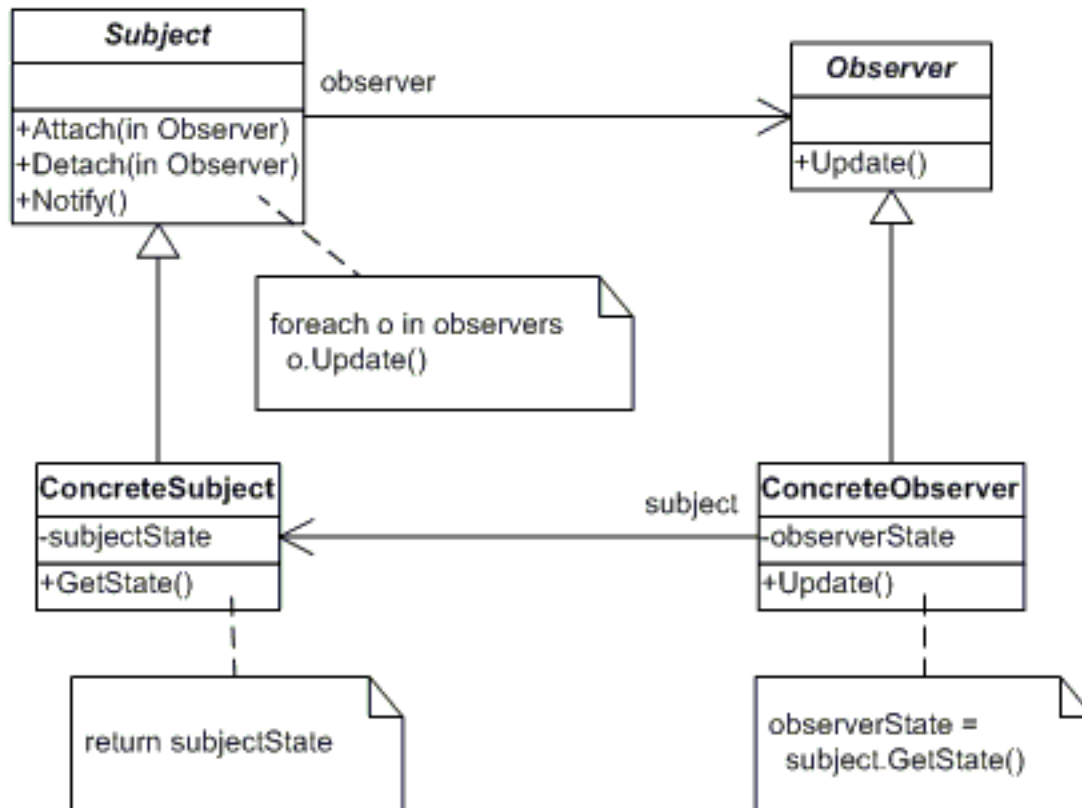
France	125
Québec	127
Suisse	



```

for all o in observers {
  o->Update()
}
  
```



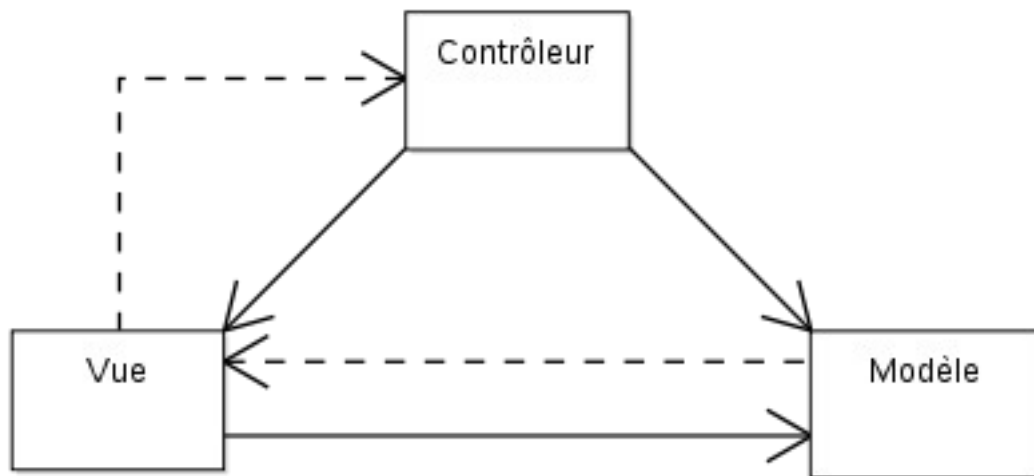


Exemple :

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

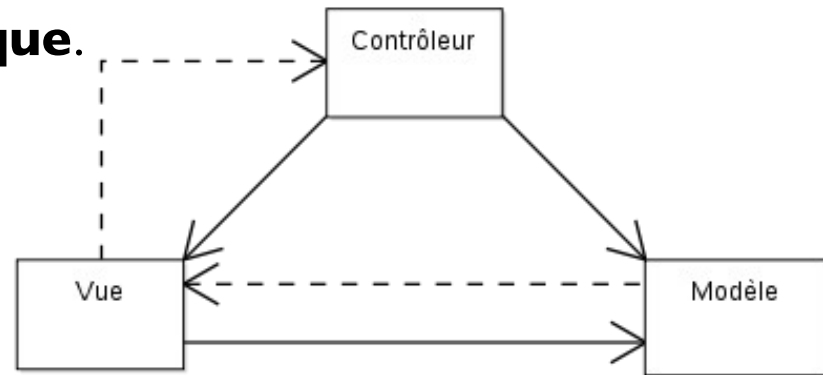
L'investisseur (ConcreteObserver) veut être notifié à chaque fois que le cours d'IBM (ConcreteSubject) change

Le patron Model-View-Controller



Model-View-Controller

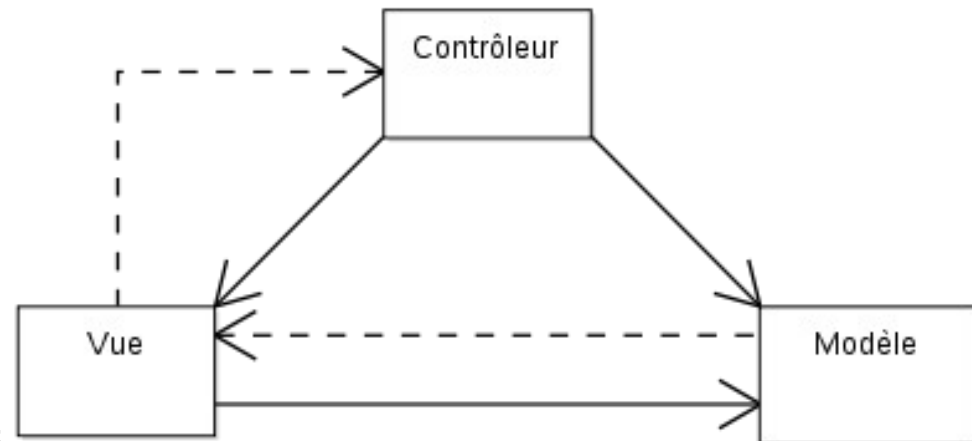
- Méthode de conception pour organiser **l'architecture d'une application avec interface graphique.**



- 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)

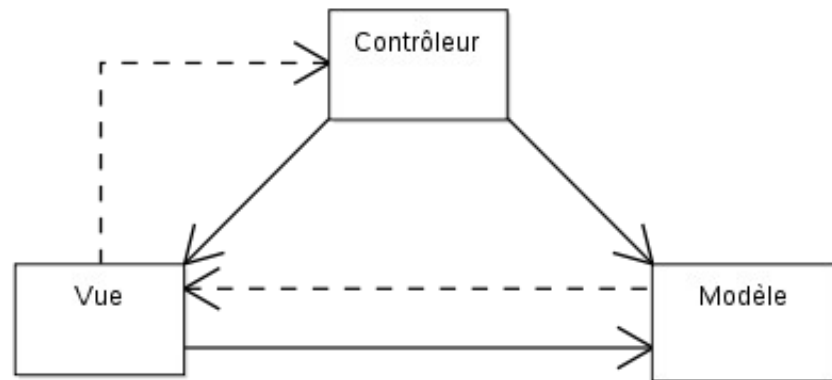
Modèle

- **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)

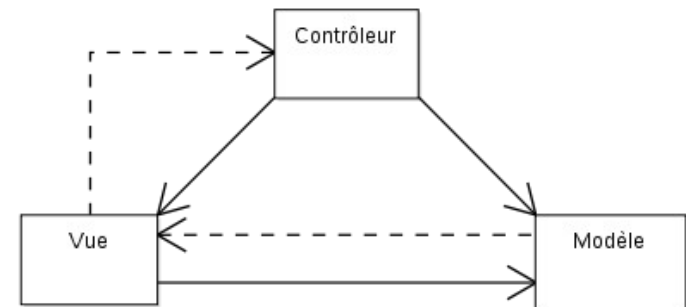


- N'effectue **aucun traitement**, ne fait qu'afficher les données du modèle.

Contrôleur

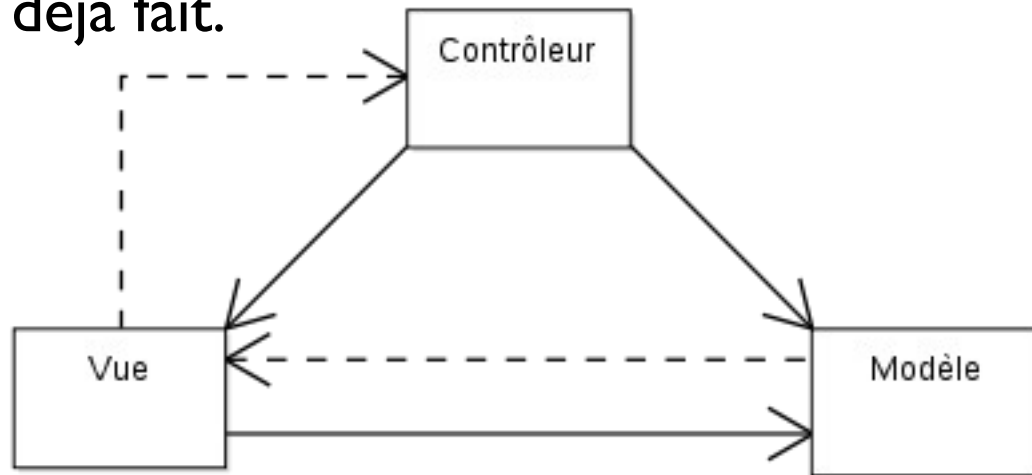
- **Gestion des événements** pour mettre à jour la vue ou le modèle.
- Synchronise la vue et le modèle.
- Reçoit les notifications de la vue, met à jour le modèle au besoin et avertit la vue que les données ont changé pour que celle-ci se mette à jour.

- **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 *contrôleur*
- 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).**