


# La modélisation UML



- Introduction à cette formation
  - Votre formateur ... Et Vous 
  - Le matériel
    - Le support de cours
  - L'organisation – horaires
    - Formation de 4 jours
  - La forme :
    - Un mélange de concepts avec application directe par un exemple simple
    - Des exercices
    - Une évaluation à la fin de la formation.

- Des liens utiles

- <https://lipn.univ-paris13.fr/~gerard/docs/cours/uml-cours-support.pdf>  
\*\*\*
- <https://openclassrooms.com/fr/courses/5647281-appliquez-le-principe-d-u-domain-driven-design-a-votre-application>
- <http://uml.free.fr/index-cours.html>
- <http://laurent-audibert.developpez.com/Cours-UML/>
- <http://staruml.io> pour l'outil StarUML2
- Visual Paradigm Community pour un outil graphique de saisie de diagrammes <https://www.visual-paradigm.com/download/community.jsp>

- Situer l'analyse et la conception dans le processus de développement
- Un support à la modélisation d'un projet : UML
  - Besoin des utilisateurs
  - Vue logique : conception orientée objet (un sous sommaire sera présenté)
  - Vue des processus
  - Vue des composants
  - Vue de déploiement

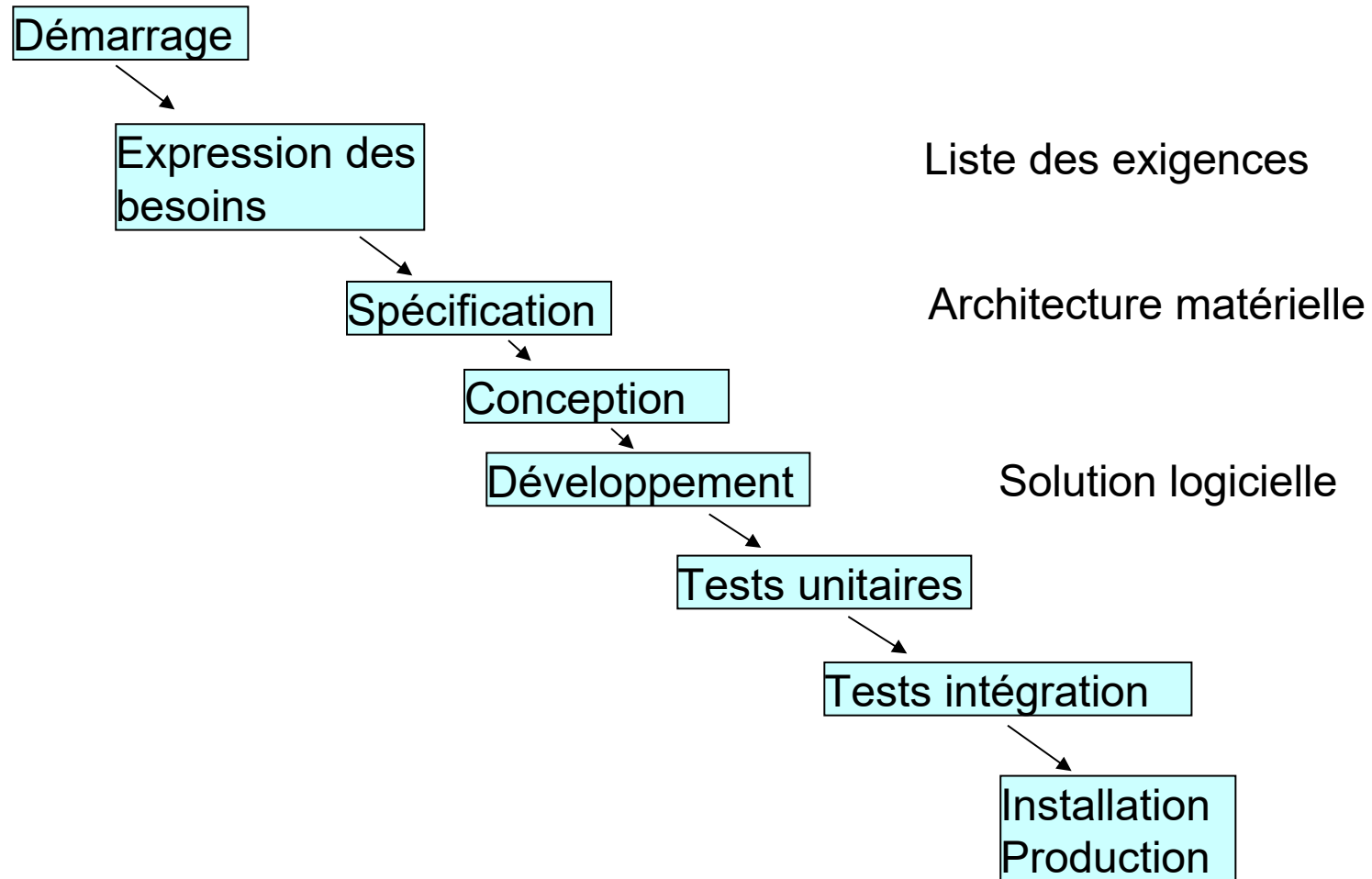
- Pour tout type de projet, les phases d'analyse et de conception sont indispensables



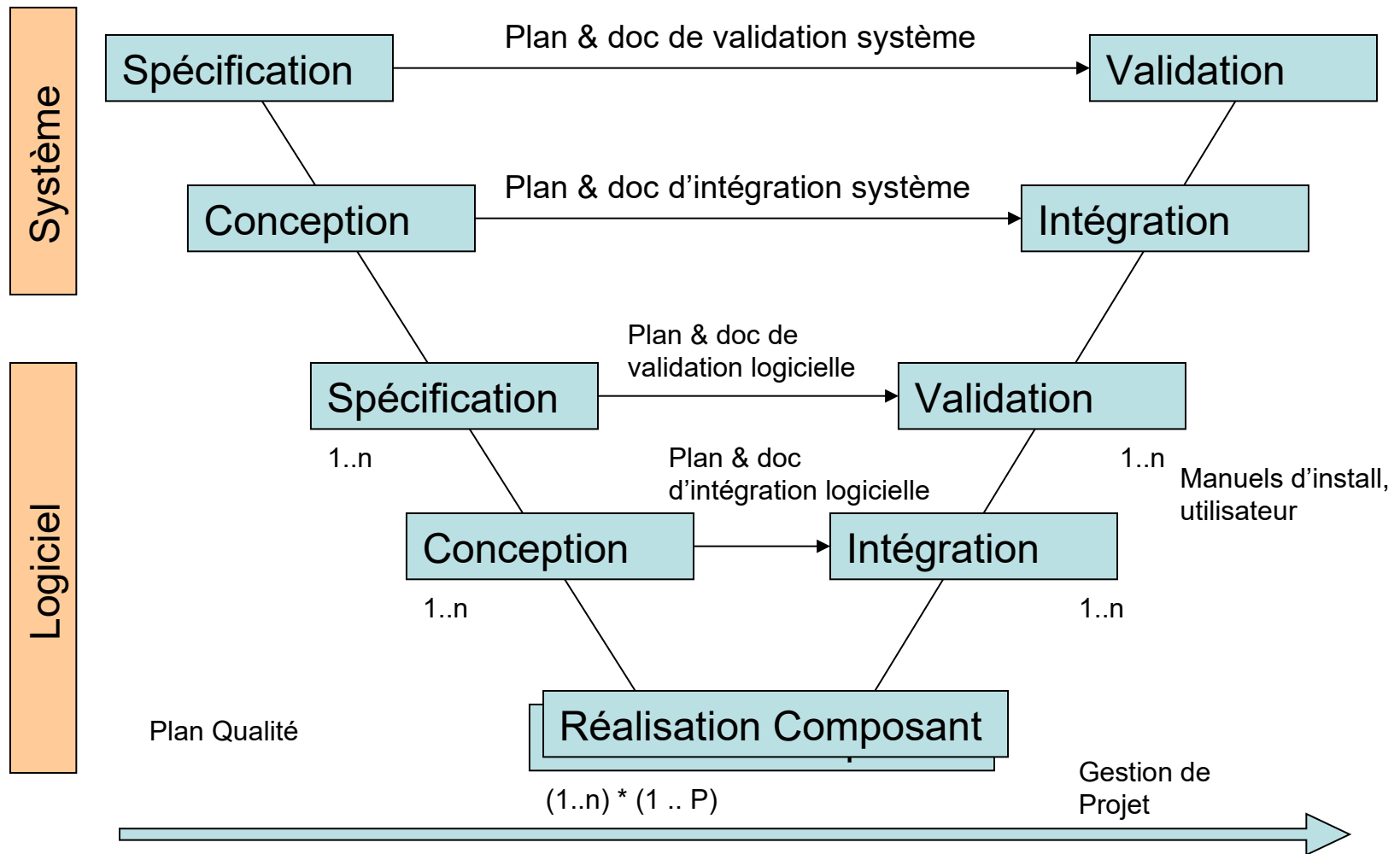
- Il faut pouvoir s'échanger des informations précises entre les divers intervenants.
  - Entre architecte et client : plans simplifiés, maquettes numériques 3D ..
  - Entre architecte et divers corps de métiers : plans BTP, plans électriques, plomberie, plans d'aménagement intérieurs et extérieurs ...
- Ces informations sont élaborées de façon itérative jusqu'à la satisfaction de toutes les parties – surtout le client

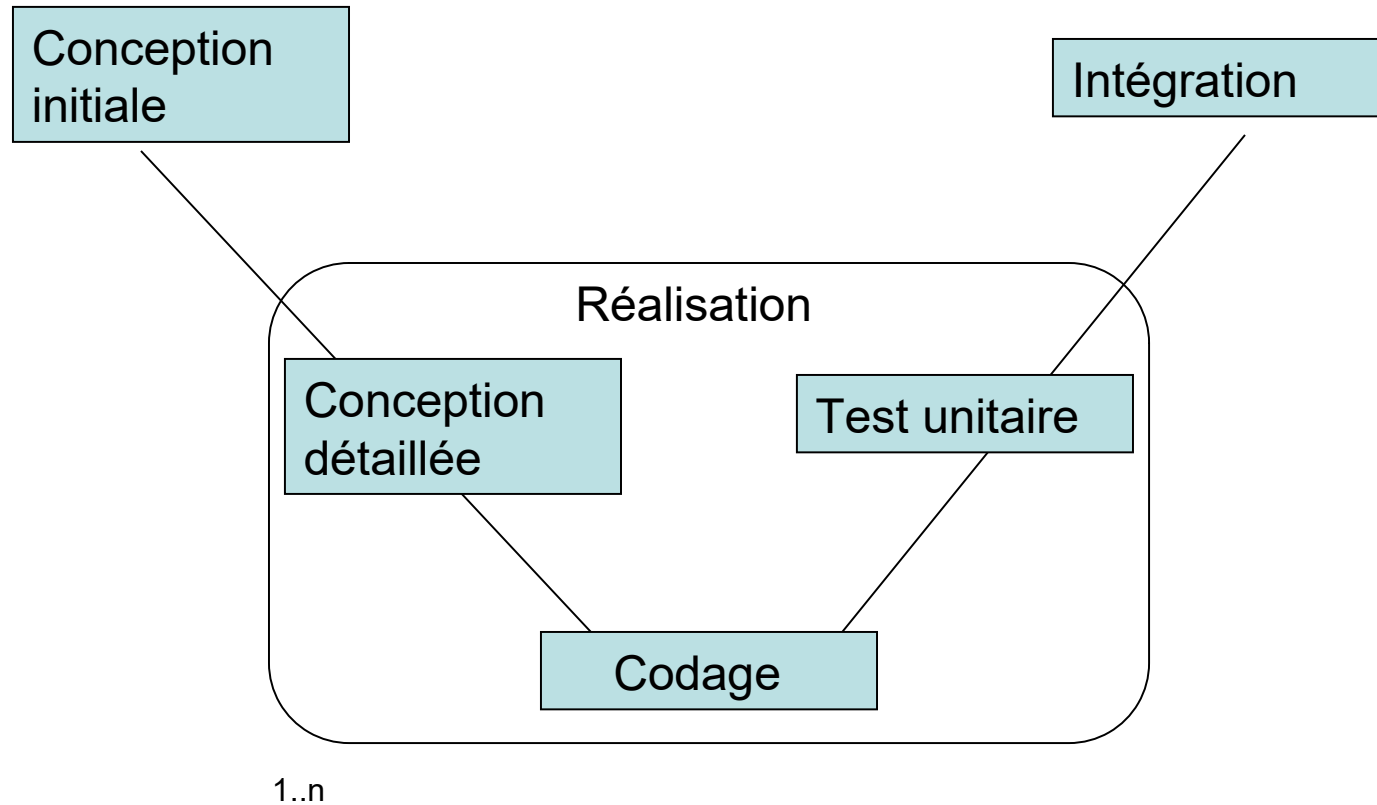


- Pour tout projet informatique, les phases d'analyse et de conception sont primordiales.
- Nous les replaçons d'abord dans leur contexte parmi quelques modèles de développement :
  - Le modèle cascade
  - Le modèle en V
  - Le modèle incrémental

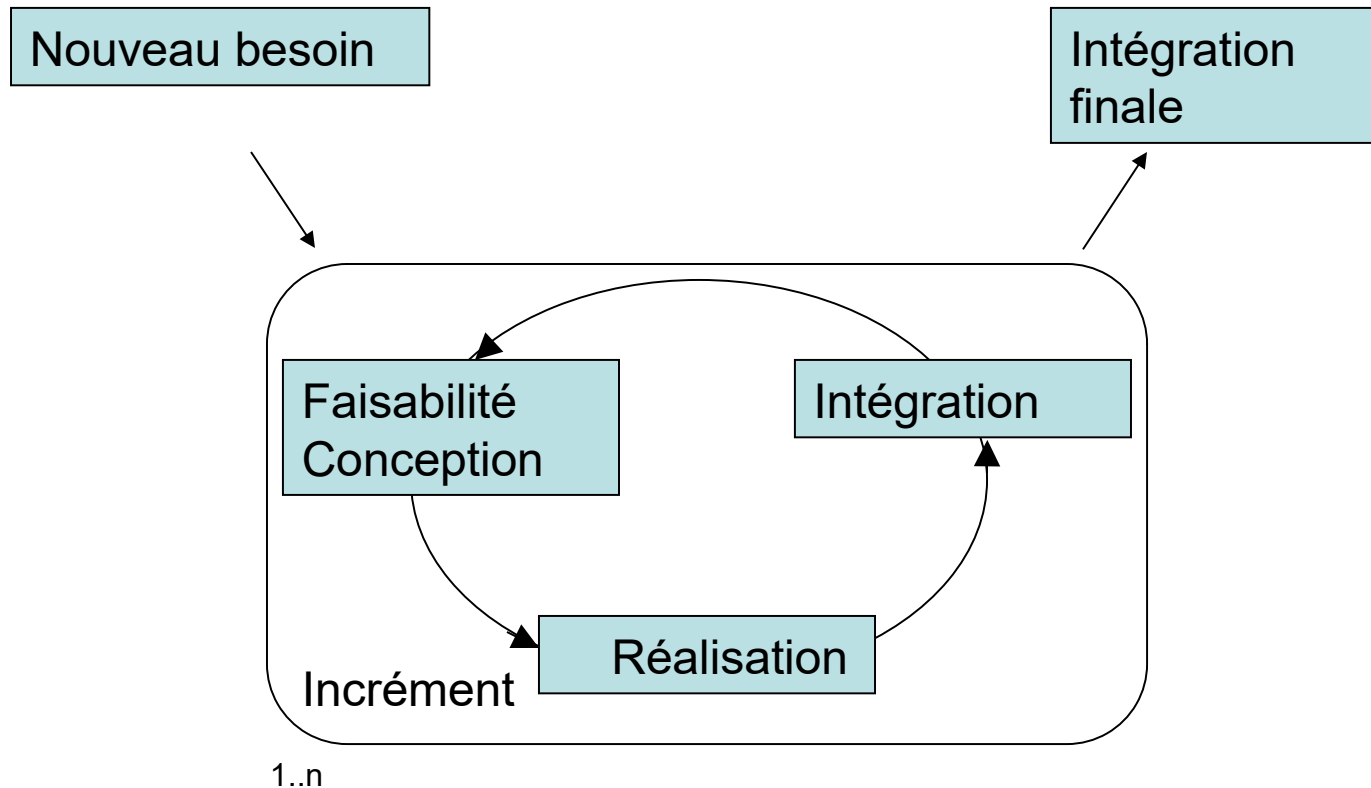








- Développement incrémental, type Agile



- La phase d'analyse/spécification comporte :
  - L'analyse du besoin des utilisateurs : quel usage, quelles fonctions, quels contextes ...
  - Après validation du besoin vient l'analyse de la solution

Le QUI et le QUOI

- La phase de conception vient ensuite et affine, décrit, clarifie l'analyse de la solution.

Le COMMENT

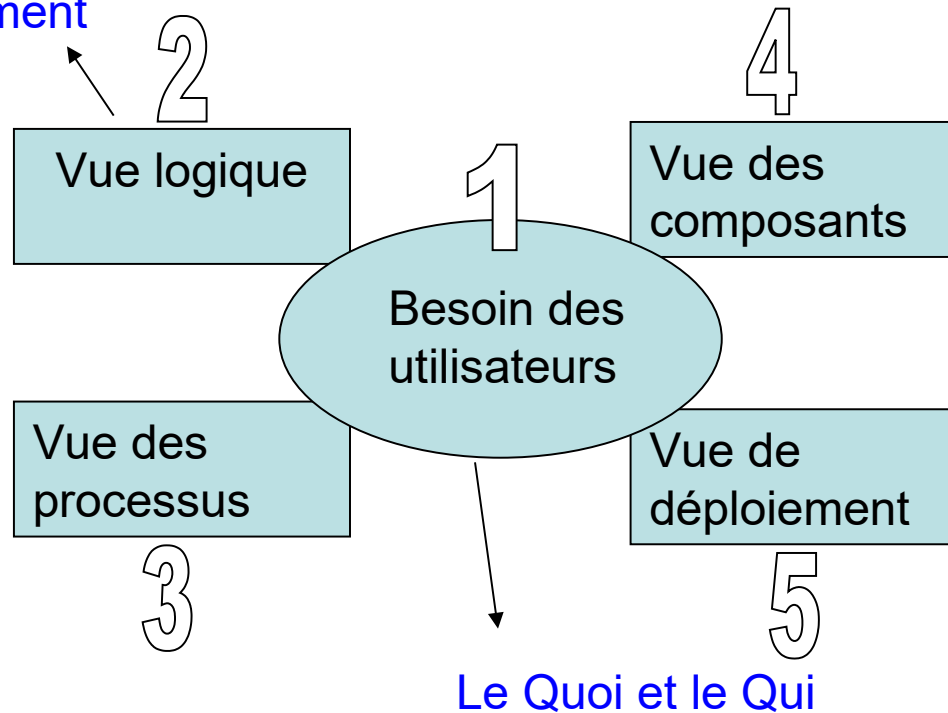
- Afin d'être clair et compris d'un grand nombre d'acteurs, il faut utiliser des méthodes, notations et conventions.  
UML est l'une de ces notations les plus courantes actuellement



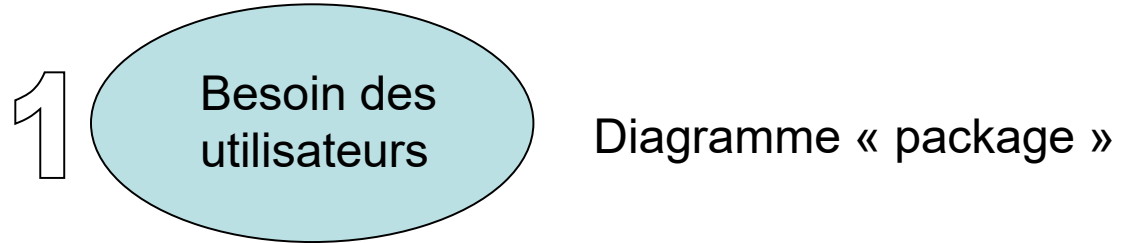
- UML est constitué d'un ensemble de représentations graphiques, nommées diagrammes
- L'équipe projet décide les types de diagrammes – parmi 13 – utiles au projet
- La démarche est pilotée par les besoins des utilisateurs et du client.
- La démarche est plus orientée 'Objet' que fonctionnelle.
- UML :
  - ➔ Un dessin vaut mieux qu'un long discours
  - ➔ Connaître, utiliser et comprendre une même façon de s'exprimer entre les divers intervenants d'un projet.

- La démarche UML apporte une aide au choix de l'architecture logicielle , par le biais d'angles de vues différents

Le Comment



- Dans les pages qui suivent, l'ordre de présentation des diagrammes n'est pas l'ordre réel d'un projet :
  - L'équipe projet définit et choisit les diagrammes utiles au projet
  - Certains diagrammes sont complémentaires
  - D'autres représentent différemment les mêmes idées
  - Leur contenu est **itératif** au cours du projet
  - Tout texte aidant à la compréhension ou mentionnant un détail peut être ajouté – le formalisme est ouvert.




- Permet une 1<sup>ère</sup> décomposition d'un système en parties – des « packages » - pour séparer des problématiques différentes.

D'un système complexe et pour l'instant inconnu on essaie de définir des sous-parties un peu moins complexes



Le formalisme :

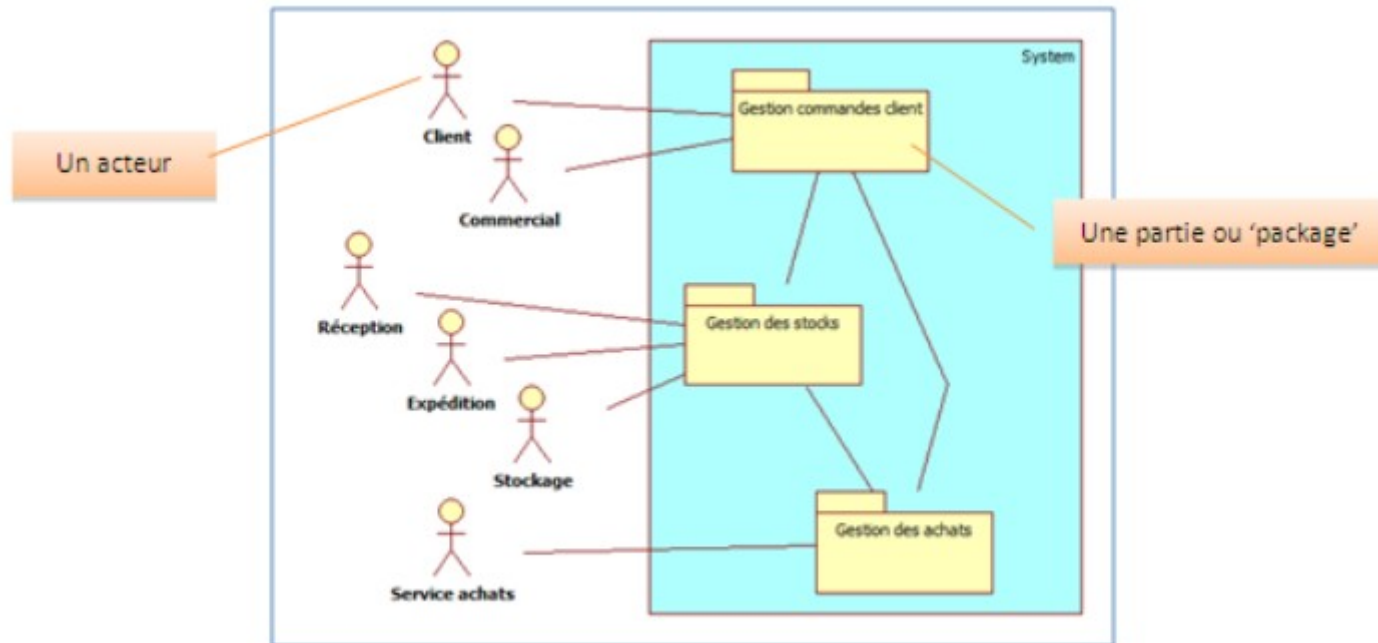
- L'acteur  représente l'élément externe (utilisateur, autre système)
- Les packages (parties) portent un libellé de fonction (ex Gestion Commande) et sont dans un rectangle double

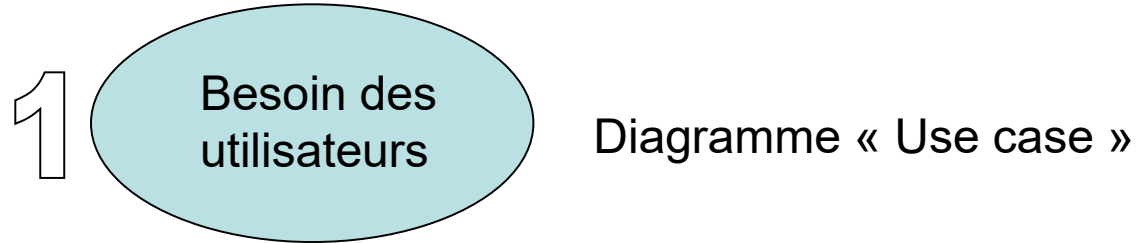


- Les packages sont dans un rectangle avec le nom 'système' s'il s'agit de l'application complète ou un autre nom s'il s'agit d'un package de plus haut niveau.
- Des traits montrent les relations entre les divers éléments



- Exemple (source openclassrooms) d'un site d'achat en ligne





- Un diagramme de cas d'utilisation – un 'use case' - identifie et représente les interactions entre le système et les acteurs extérieurs au système

➔ Le QUOI et le QUI

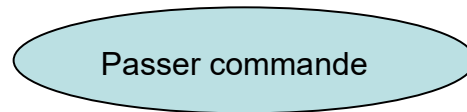
- Formalisme

- L'acteur système)

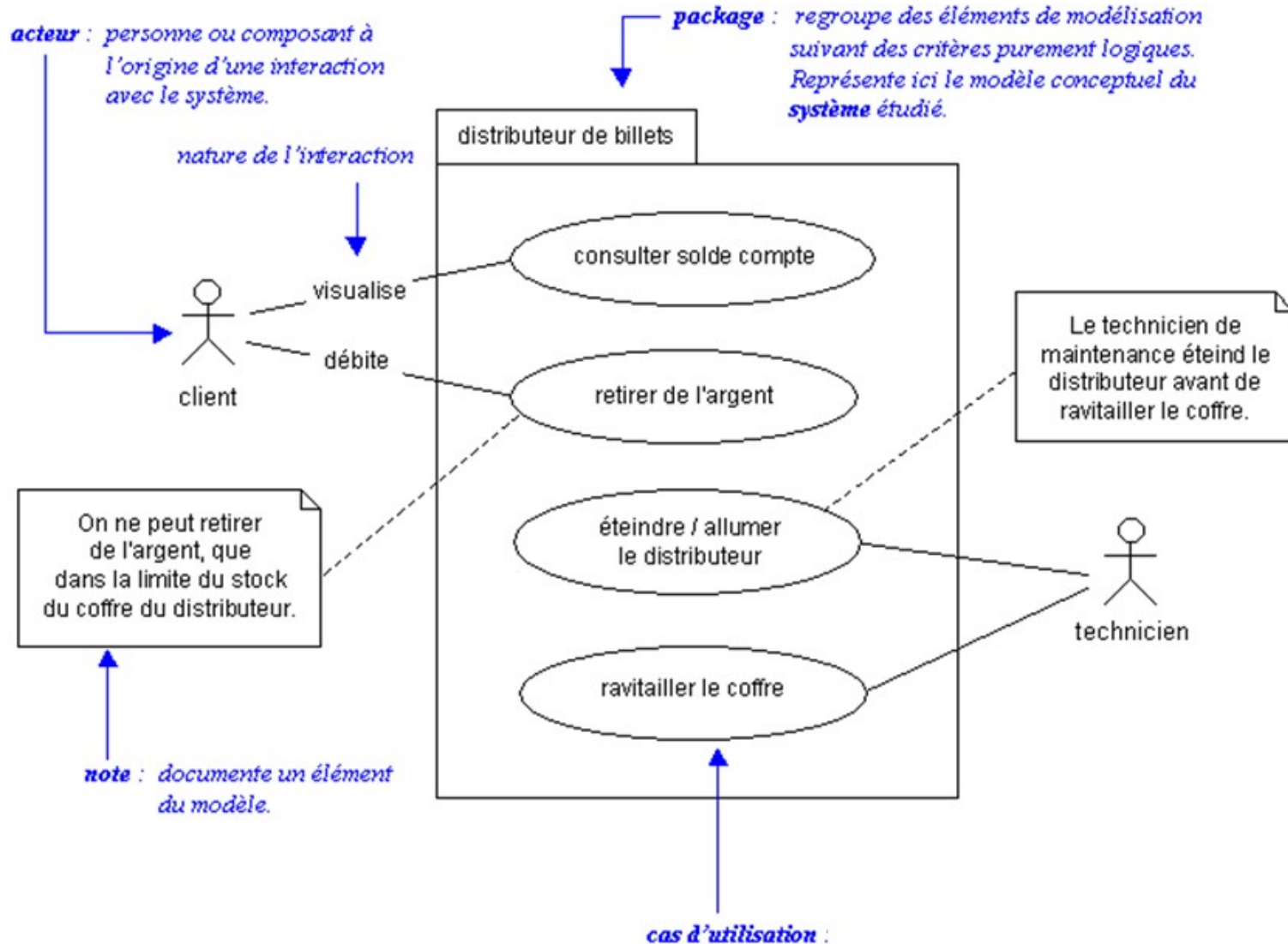


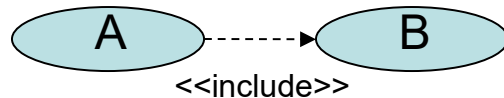
représente l'élément externe (utilisateur, autre

- Le cas d'utilisation se représente dans une ellipse et un verbe.  
Modélise un service rendu par le système

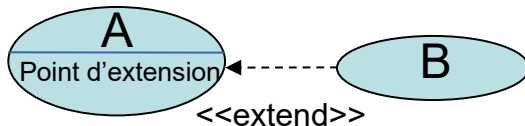


- Un rectangle encadre les ellipses. Un titre figure en haut. Il indique si le diagramme représente le système complet ou un package.



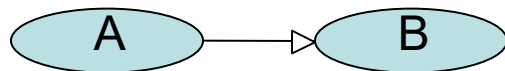


Relation d'**inclusion**. Lorsque A est sollicité, B l'est **obligatoirement**



Relation d'**extension**. Exécuter A peut entraîner l'exécution de B si le « point d'extension » est réalisé

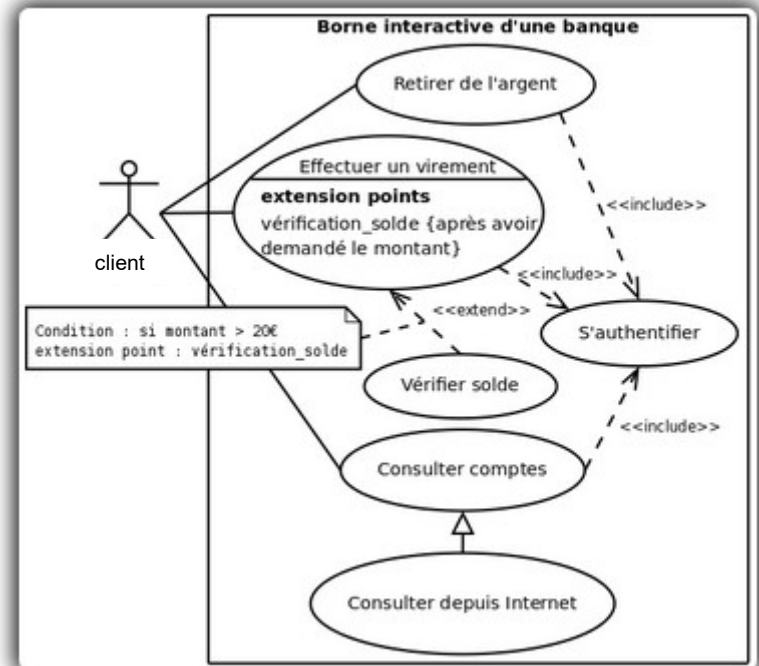
B est une partie **Optionnelle** de A



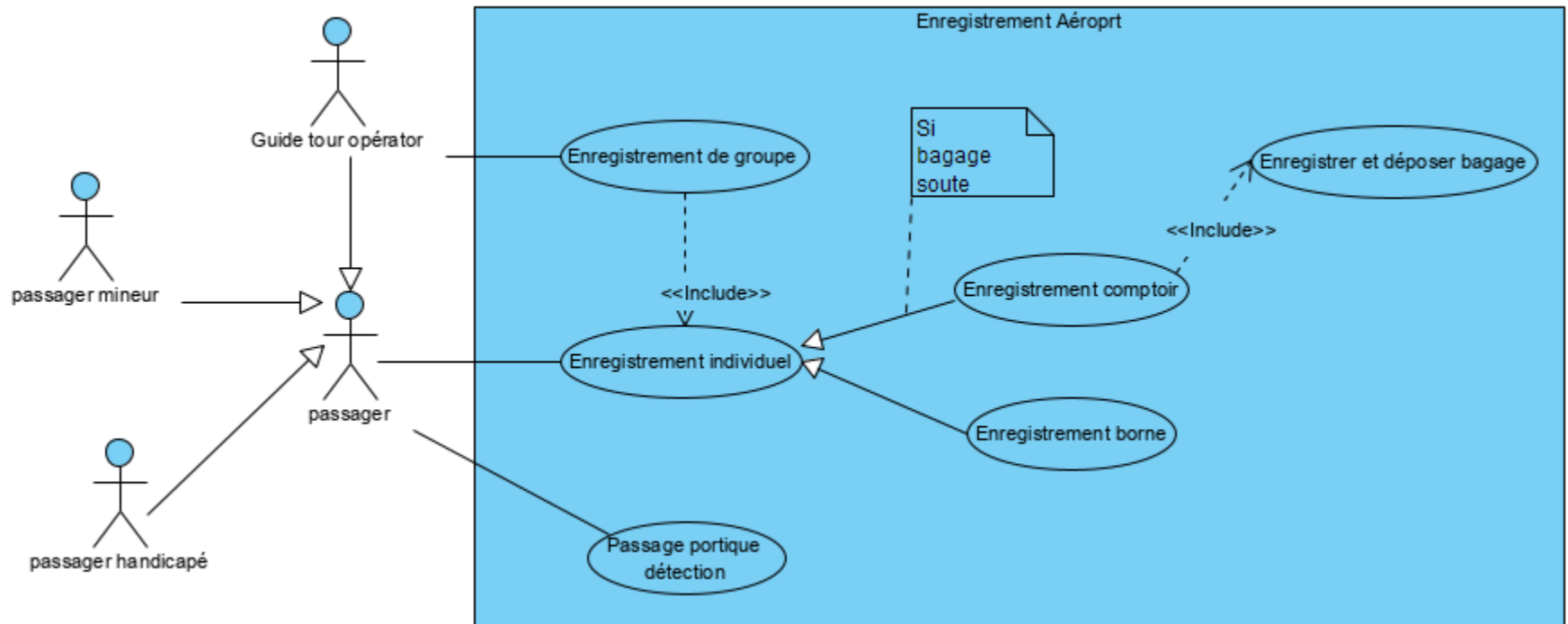
Généralisation.

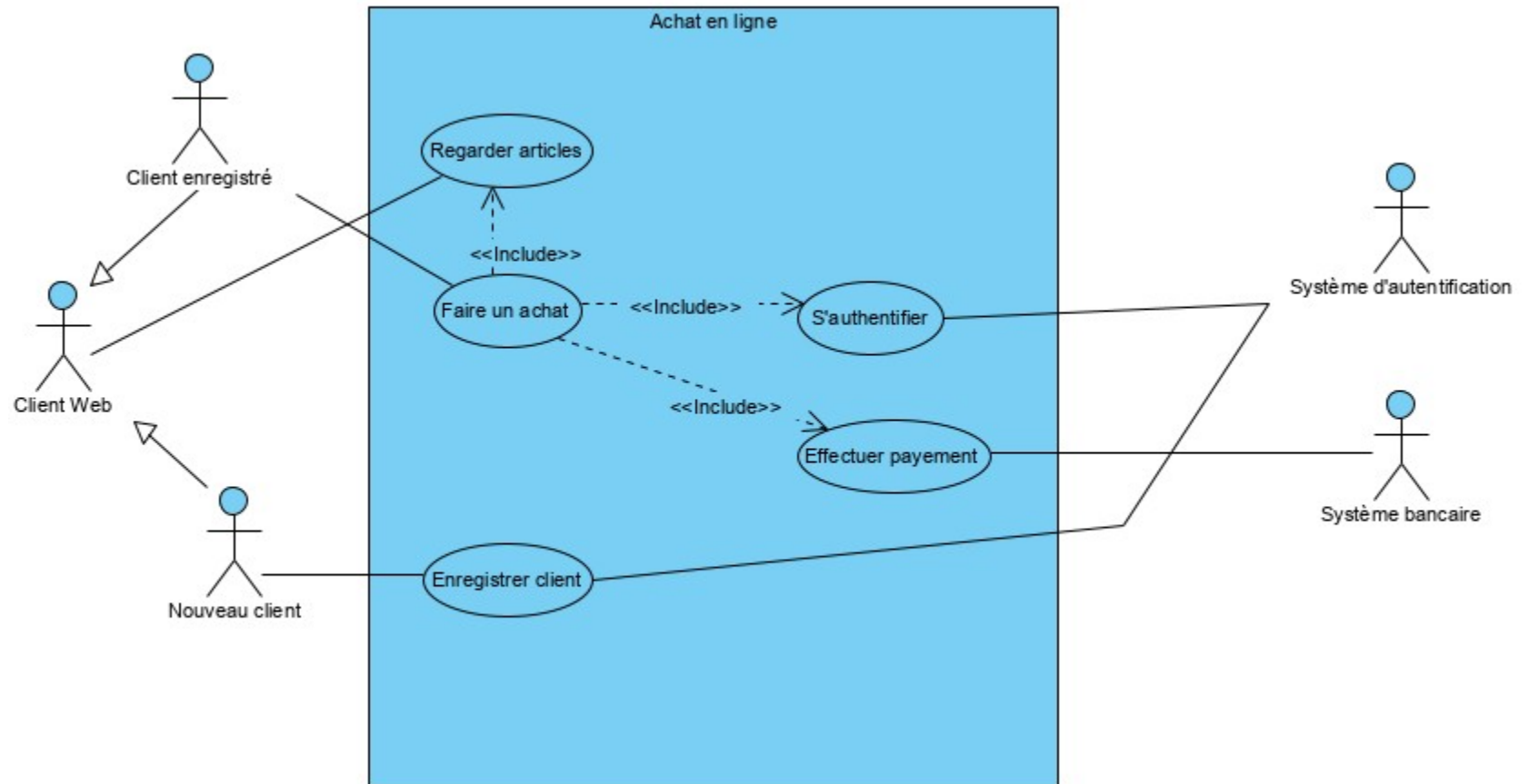
A est un **cas particulier** de B ou

B est une **généralisation** de A

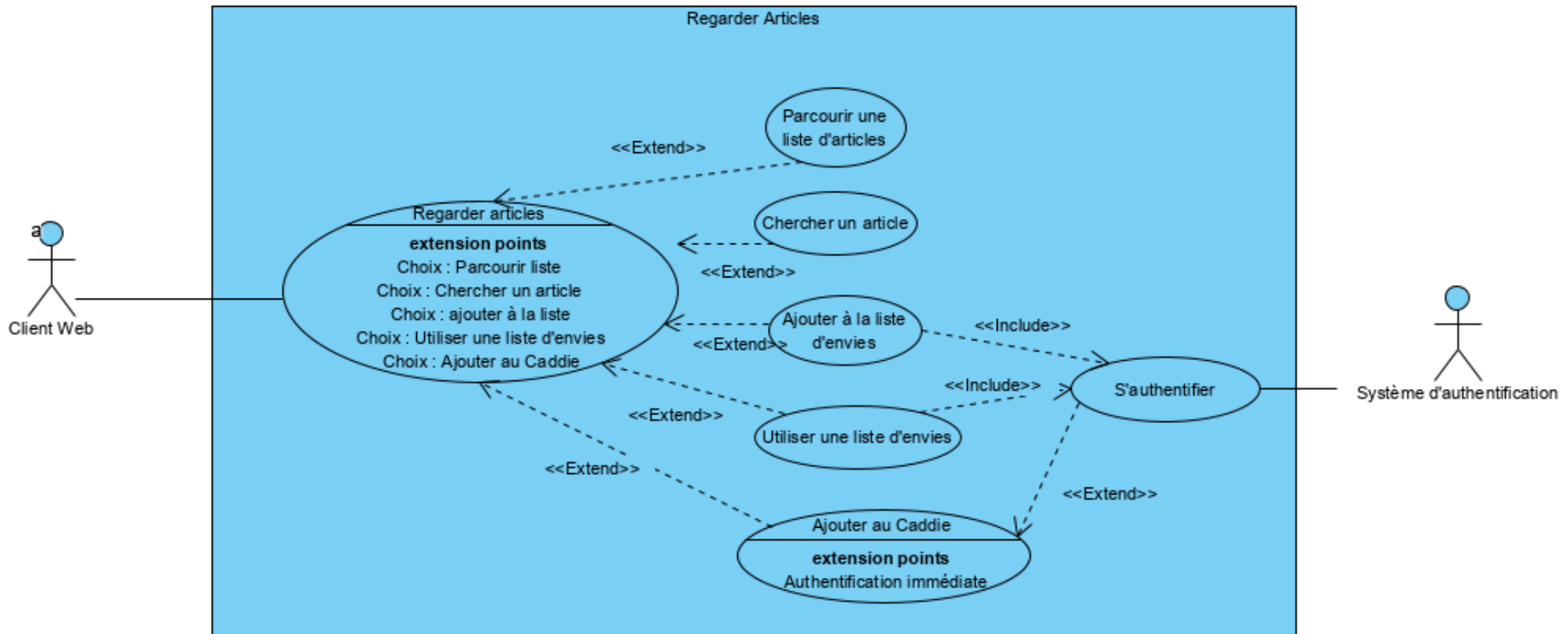


- <https://www.uml-diagrams.org/use-case-diagrams-examples.html>









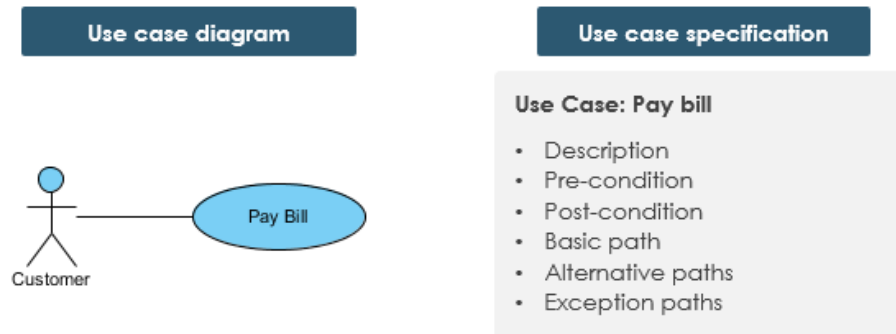
Plus de précision sur ce site :

<https://lipn.univ-paris13.fr/~gerard/uml-s2/uml-cours04.html>

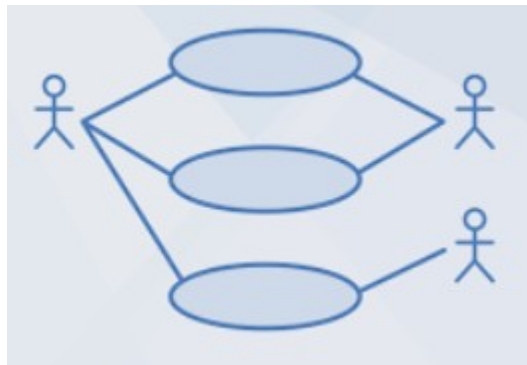


Dans un projet les use cases doivent être commentés. On parle de spécification de cas d'utilisation

<https://www.cybermedian.com/fr/use-case-diagram-vs-use-case-specification/>



- Réaliser les exercices concernant les use cases



2

Vue logique

Le Comment

- On s'intéresse ici à comment va fonctionner le système

UML propose une approche orientée  
Objet

plutôt que

Fonctionnelle.



En support, UML définit plusieurs types de diagrammes :

- Diagrammes d'objets
- Diagrammes de classes

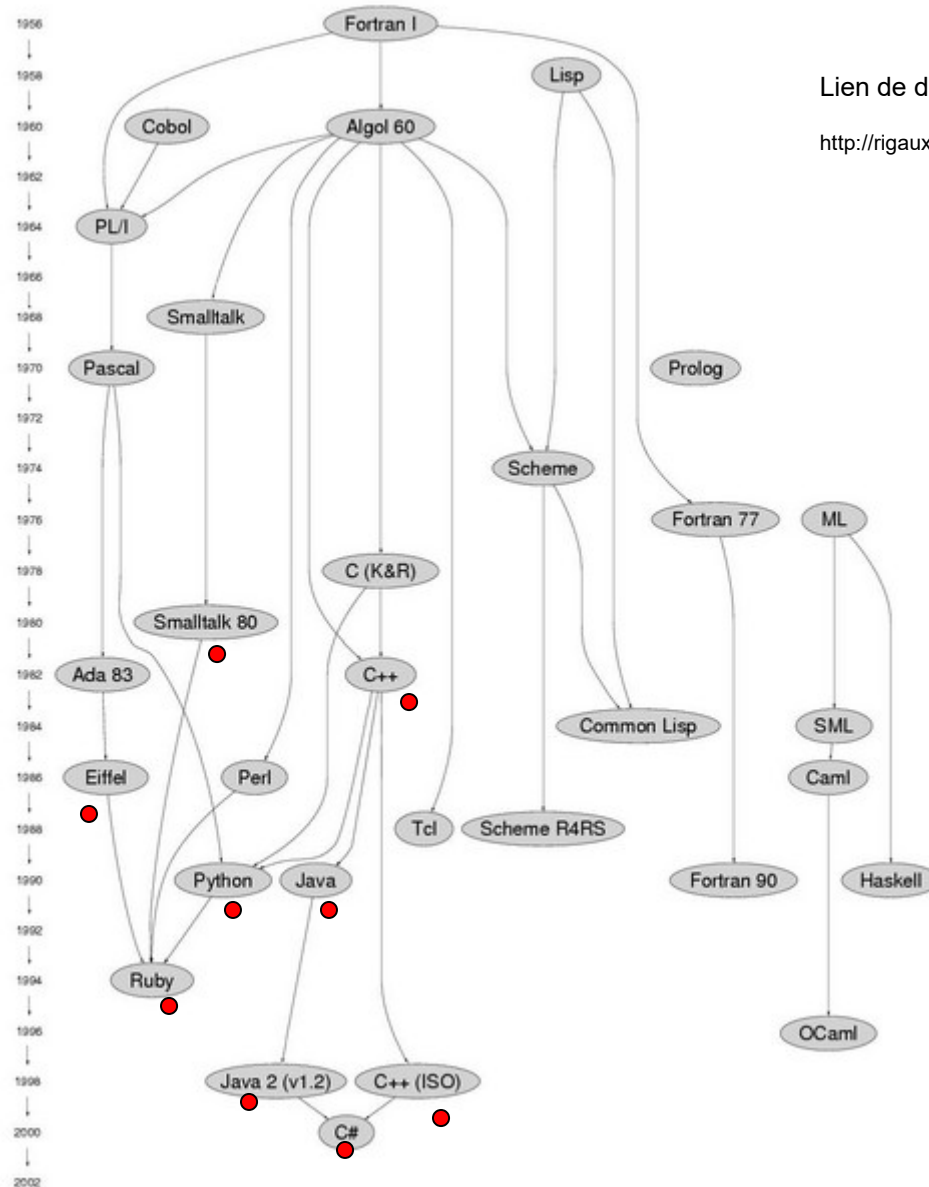
# Conception et Programmation orientée Objet

- Des liens utiles

- <http://uml.free.fr/index-cours.html>
- <http://laurent-audibert.developpez.com/Cours-UML/>
- Staruml.io pour l'outil StarUML2
- <https://www.modelio.org/>



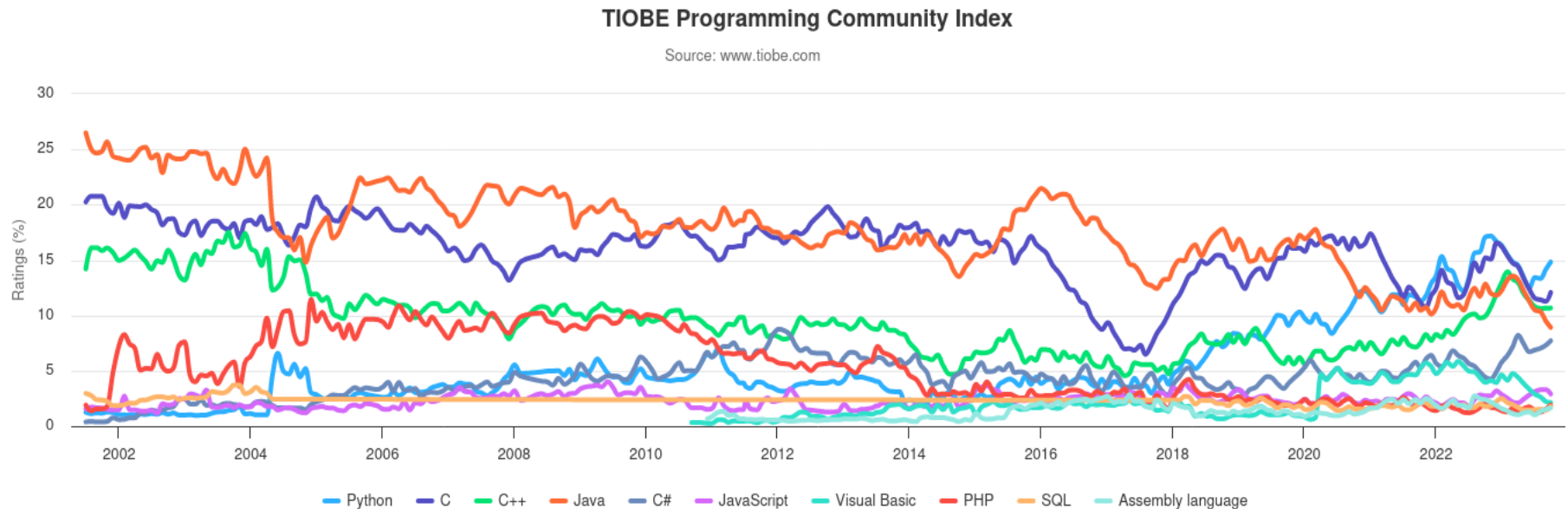
- Un bref historique des langages
- Pourquoi concevoir
- Les bases de l'objet
- Formalisme UML
- Relations entre objets
- Héritage
- Classes et Méthodes abstraites
- Interface
- Diagramme de séquence
- Les design Patterns

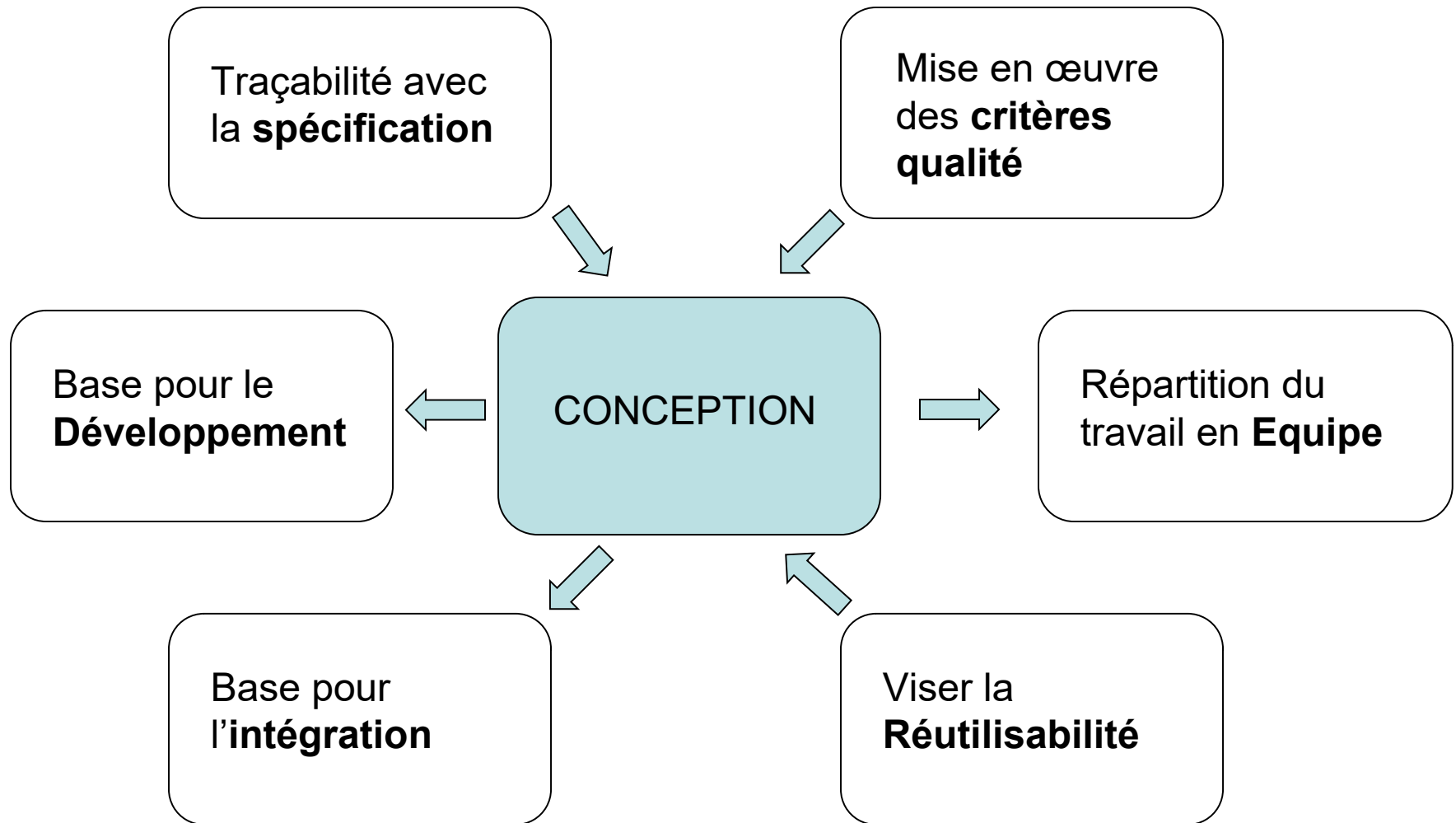


Lien de diagramme plus complet

<http://rigaux.org/language-study/diagram.png>

Popularité des langages (source [https://en.wikipedia.org/wiki/Object-oriented\\_programming](https://en.wikipedia.org/wiki/Object-oriented_programming) )

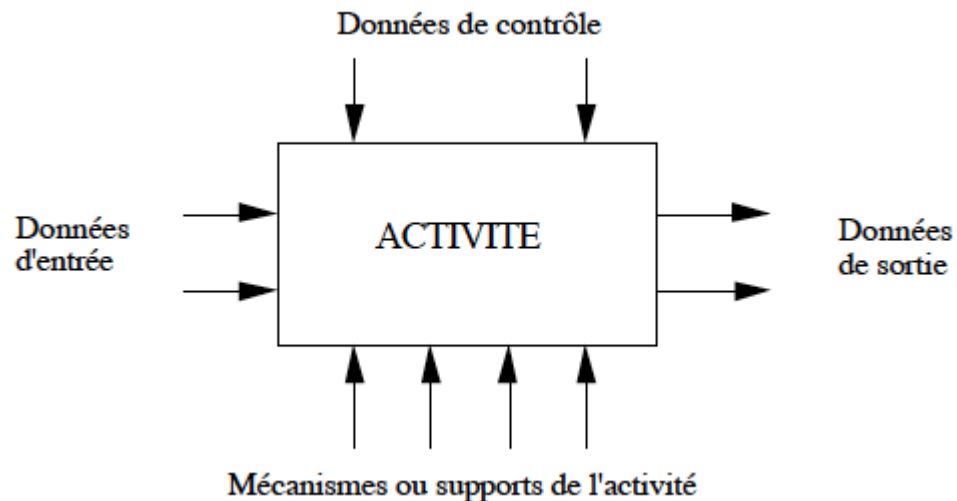






# Conception orientée Objet

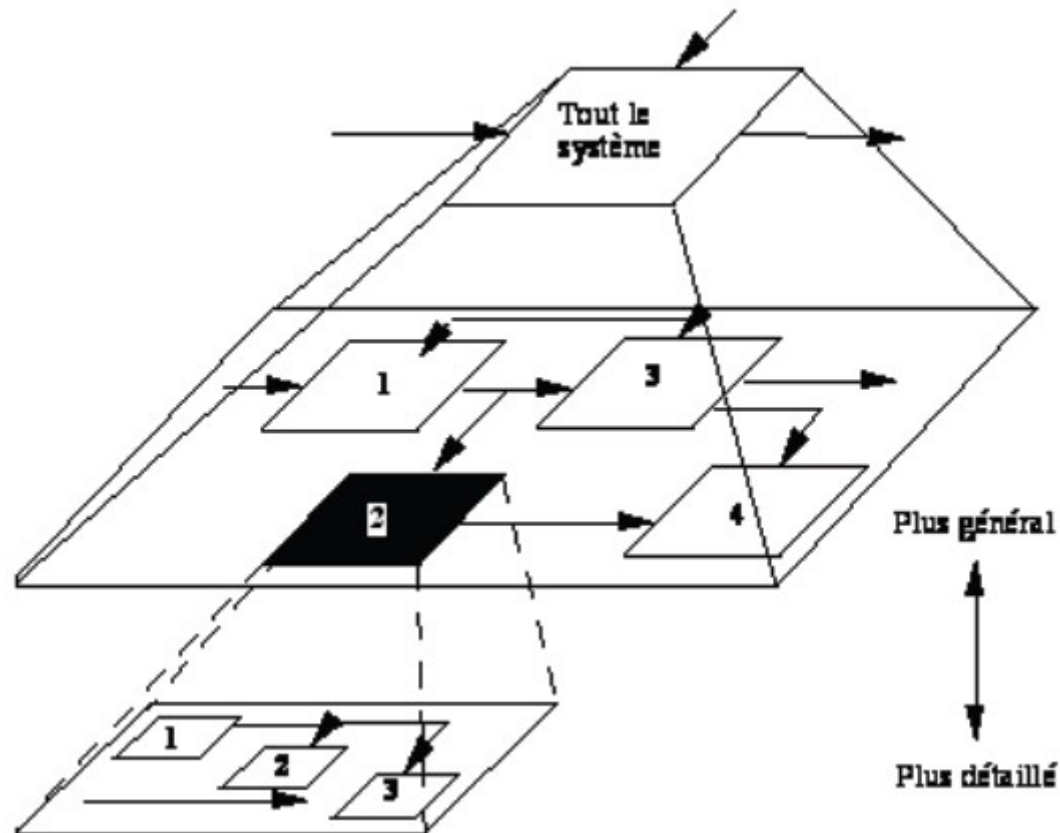
- Travailler avec des outils communs, connus et compréhensibles par l'Equipe
  - Exemple en spécification avec une **méthode fonctionnelle, non Objet** – système ou logicielle -, pour les analyses globales de systèmes complexes : méthode SADT



# Conception orientée Objet

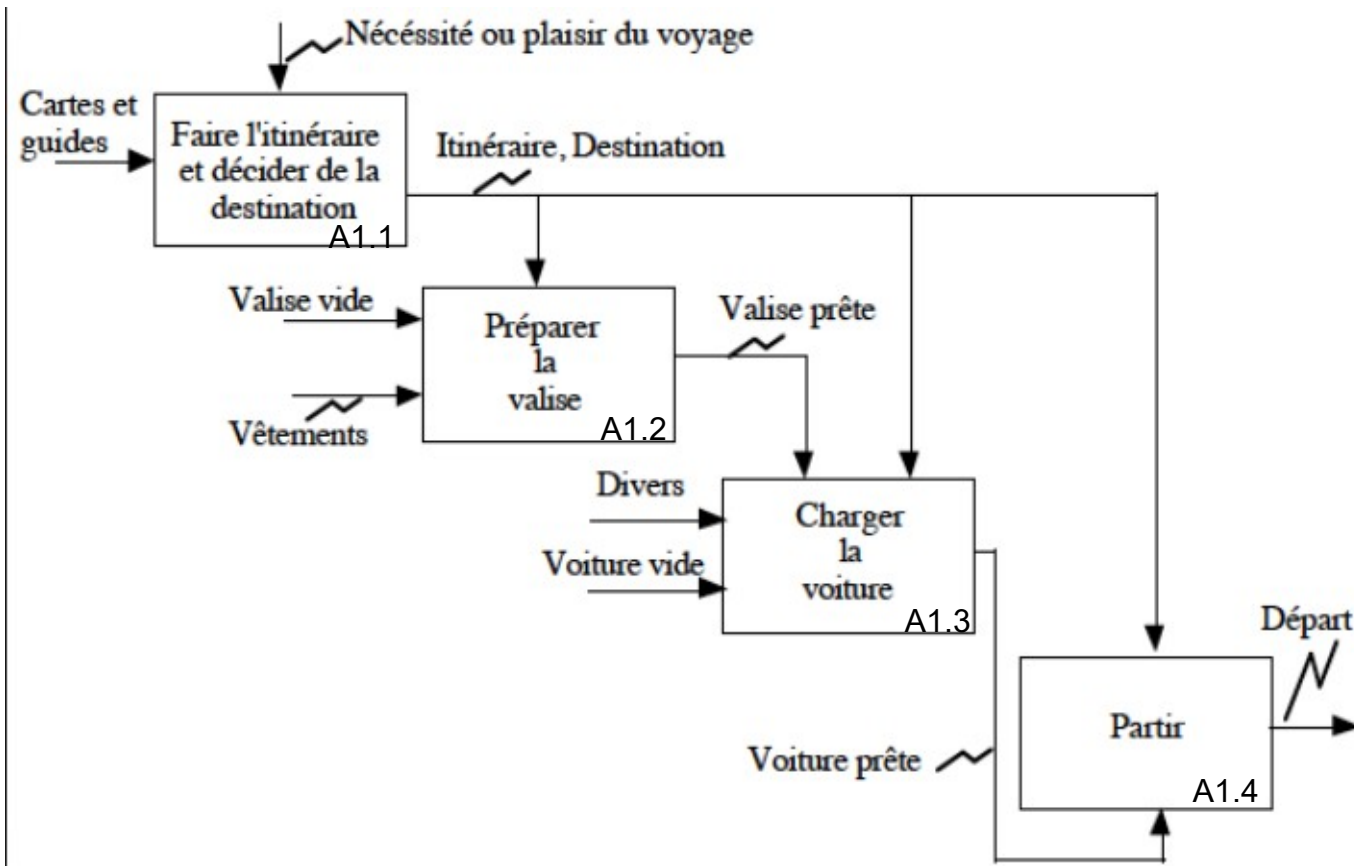
- SADT suite ..

La démarche est de partir d'une page blanche, de définir la boîte de niveau 0, puis de reprendre cette boîte pour la décomposer en sous-niveaux fonctionnels successifs jusqu'à un niveau suffisant.



# Conception orientée Objet

- Exemple **SADT** de la fonction rêver et partir en vacances



- Une méthode unifiée de conception objet : UML
- Il faut d'abord définir la conception objet.

Commençons par une petite histoire ...

- Le Vintage Téléphone et le e-Phone



C'est à l'appelant :

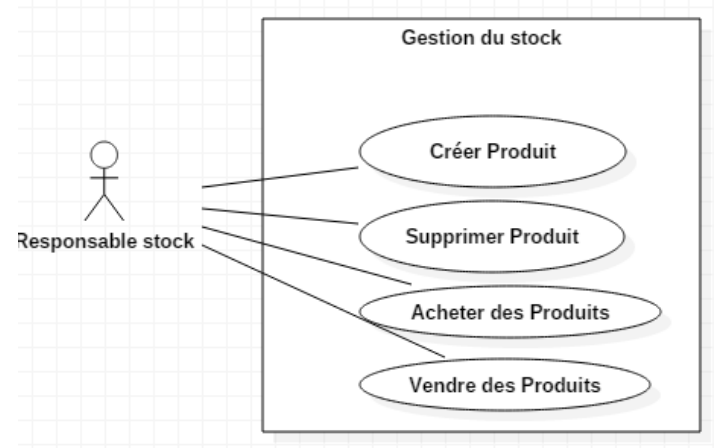
- de connaître le n°
  - de consulter un annuaire
  - de composer le n°
- ➔ Aucune donnée
- ➔ Une seule fonction



C'est l'Objet téléphone

- qui possède les n°
  - qui réalise l'appel à partir d'un nom
  - il est facile d'ajouter une fonction/application
  - .....
- ➔ Contient ses données propres
- ➔ Les fonctions utilisent leurs données propres

- Un petit exemple : Gestion d'un entrepôt de produits :
- Spécification
  - Un Produit est défini à l'aide de
    - Un nom
    - Un prix d'achat
    - Une marge bénéficiaire
    - Une quantité en stock
  - Transactions possibles
    - Création d'un nouveau produit
    - Suppression d'un produit
    - Achat d'une quantité de produits
    - Vente d'une quantité de produits



- En solution non objet (C par ex)  
Regardons le code



- Un Produit est une structure de données
- Les Produits sont dans un tableau global
- Le prog principal invoque les traitements, lesquels accèdent aux var globales
- Critiques :
  - Les données ne sont pas protégées contre des accès sauvages : tout le monde voit tout
  - Une évolution de la structure Produit remet en cause tout le code
  - Le test unitaire revient à tout tester

- L'approche Objet

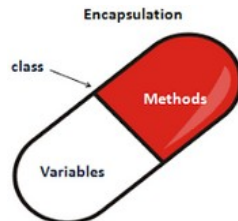
Imaginer des « boîtes » autonomes qui contiennent leurs données et tout ce qu'il faut en interne pour les manipuler, en toute sécurité (intégrité).

Il faut donc imaginer ce que contient la boîte (les données) et ce que cette boîte doit faire avec (les services ou méthodes).

Ce raisonnement se nomme l' « **Abstraction** ».



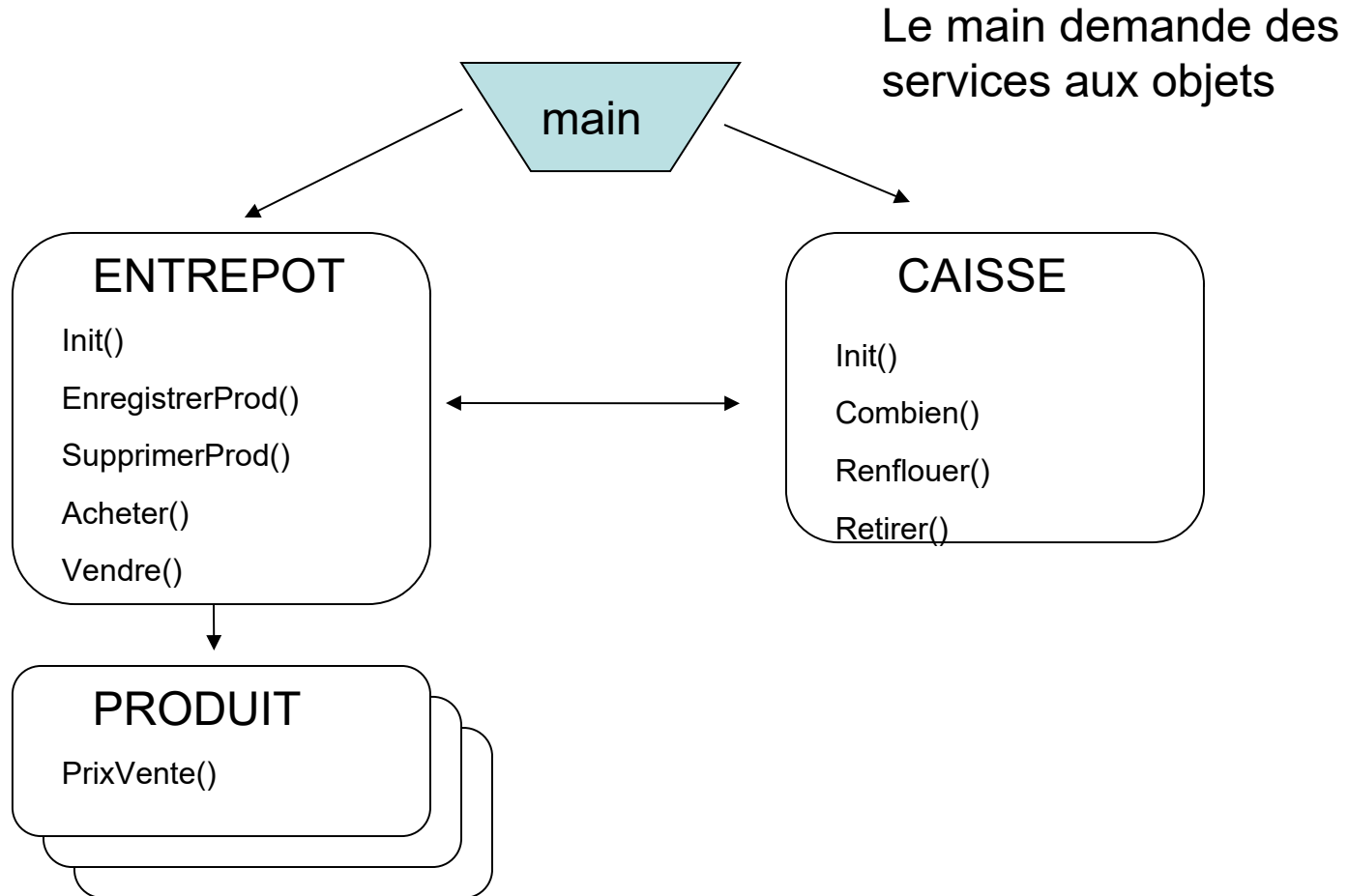
Le fait de mettre en boîte et de cacher/montrer ce qu'il faut se nomme l'« **Encapsulation** »





- Retour sur l'exemple

En imaginant des boîtes (non UML)



- Dans cette application on a l'intuition que :
  - La boîte ENTREPOT est unique
  - La boîte CAISSE est unique
  - Il y aura autant de boîtes PRODUIT que l'on utilisera de produits dans la vie de l'appli. Pour autant les Propriétés de chaque Produit et ce qu'ils doivent faire sont communs à tous les Produits (leurs valeurs sont différentes).

L'idée est de définir la **CLASSE** « Produit » qui est un **moule** sur des futurs Objets de cette classe.

- Une Classe est « une fabrique à Objets »
- Un Objet est une « **instance** » de Classe
- Les services se nomment des **Méthodes** (ou Opérations)
- Les données se nomment des **Attributs** (ou Données Membre, champs, variables d'instance)

- Encapsulation

- Consiste à mettre dans une même boîte le code (Méthodes) et les données (les Attributs)
- Consiste à masquer les Attributs et Méthodes qu'il est dangereux/inutile de montrer
- Consiste à montrer aux autres objets le juste suffisant.

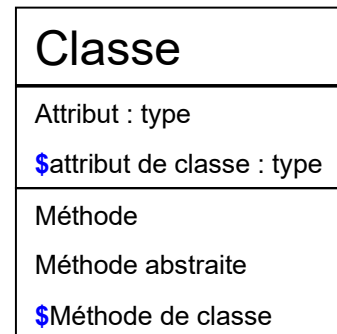
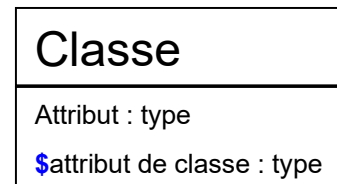
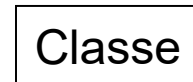
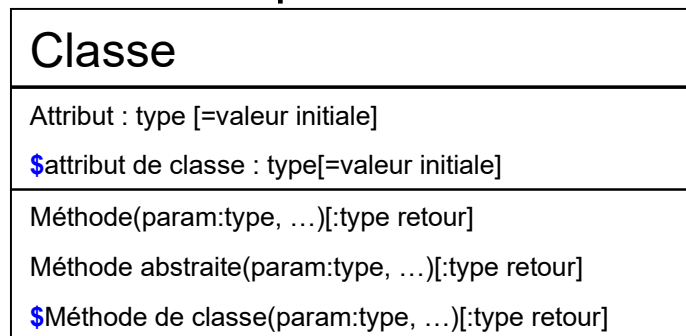


- Comment définir la visibilité/masquage :

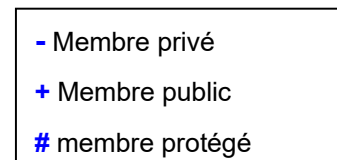
- Public : les Attributs et Méthodes sont accessibles en dehors de l'objet
- Private : les Attributs et Méthodes sont accessibles uniquement dans l'objet
- Protected : les Attributs et Méthodes sont accessibles dans l'objet et tout objet instance d'une classe dérivée

- Formalisme graphique en UML. Plusieurs niveaux possibles :

- Classe
- Classe + Attributs
- Classe + Attributs + Méthodes
- Diagramme complet



Visibilité



- Exemple

Rectangle
<ul style="list-style-type: none"><li>- origine : Point</li><li>- hauteur : int</li><li>- largeur : int</li></ul>
<ul style="list-style-type: none"><li>+ Rectangle(origine:Point,larg:int, haut:int)</li><li>+ Perimetre() : float</li><li>+ Translater(point:Point)</li><li>+ Afficher()</li></ul>

- Une analogie avec des chariots d'atelier !
  - Les chariots de ce type (des objets) ont été créés à partir d'un même plan (la Classe) chez le fabricant de chariot.
  - Ils sont utilisés dans une usine pour assembler la planche de bord d'un véhicule.
  - Les tiroirs du haut contiennent les pièces (les données) utiles pour la planche de bord
  - Les tiroirs du bas contiennent les outils (les méthodes) utiles à l'assemblage.



- Dans l'atelier il y a plusieurs chariots (Objets) de ce type pour construire des planches de bord de différents modèles
- Les outils sont les mêmes
- Les pièces dans les tiroirs du haut sont différentes.



- Ici des chariots pour l'assemblage des feux, d'autres outils, d'autres pièces



- Sans bouger le chariot on est capable d'assembler complètement les pièces

- Certains travaillent « à l'ancienne » (procédural) dans cet atelier en utilisant ce type de chariot :



- Ici le chariot sert à déplacer la pièce à assembler (les données)
- Les outils (les fonctions) sont en fixe sur des établis de l'atelier
- C'est à celui qui pousse (équivalent au main) à savoir vers quel établi se déplacer



- Dans une application un objet d'une classe n'agit pas seul mais dans un ensemble d'objets de plusieurs classes.
- Les Objets interagissent ensemble, ils sont en **Relation**
- Ces Relations se représentent au niveau du diagramme de classe.
  - Dépendance (Dependency)
  - Association
  - Agrégation
  - Composition

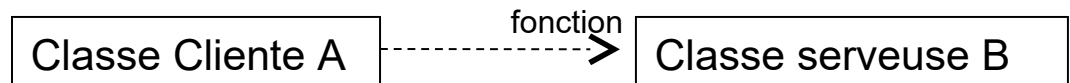
- Dépendance

- Relation limitée dans le temps qui ne nécessite pas forcément de création d'un Attribut

- Exemple

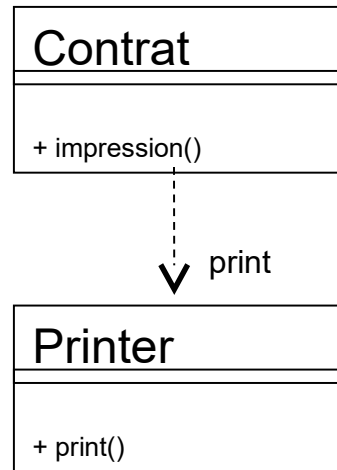
```
class Contrat {  
    ...  
    public void impression()  
    {  
        Printer imprimante = PrinterFactory.getInstance(); // Contrat dépend de Printer  
        ...  
        imprimante.print(client.getName());  
        ...  
    }  
}
```

- Notation UML



La Classe A « **Dépend** » de B. Pas d'attribut de A nécessaire.

## – Exemple



- Association mono directionnelle

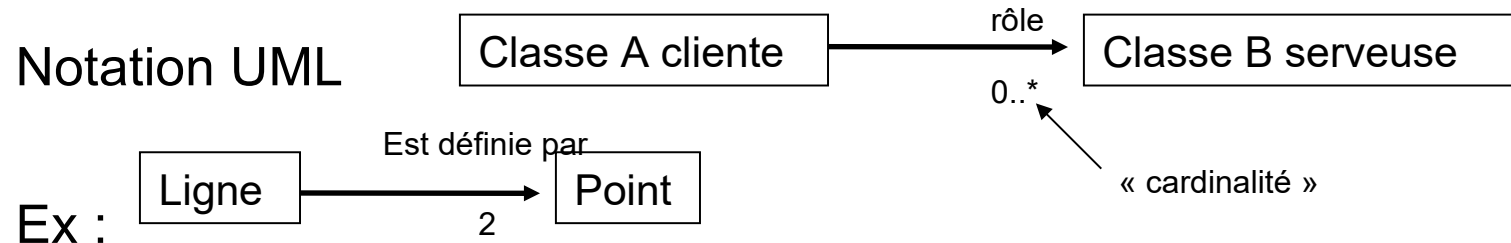
Exprime que tout Objet d'une Classe A **Utilise** (ou peut utiliser) une (des) instance(s) d'une Classe B

L'association permet

- Le changement d'objet associé
- La vie indépendante des Objets associés

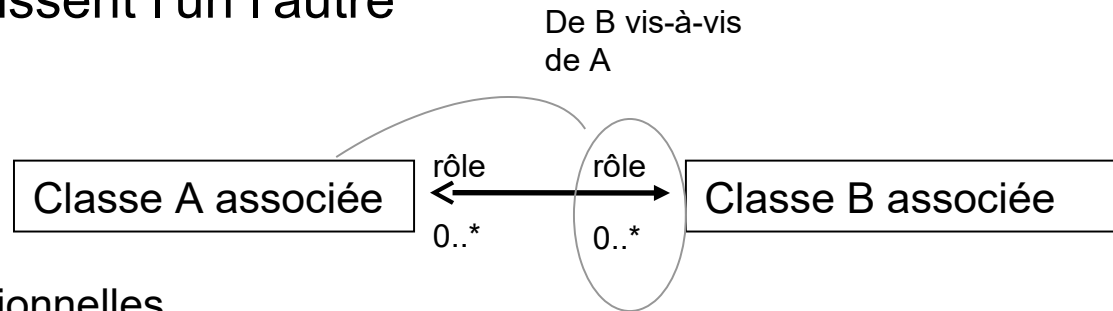
Un Attribut de l'Objet A est du type Classe B (ou List<Classe B>)

A se sert de B et B ne le sait pas

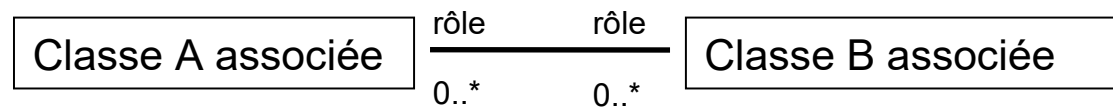


- Association bi directionnelle  
A et B se connaissent l'un l'autre

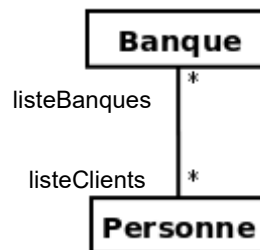
Notation UML



Les flèches sont optionnelles



Ex :



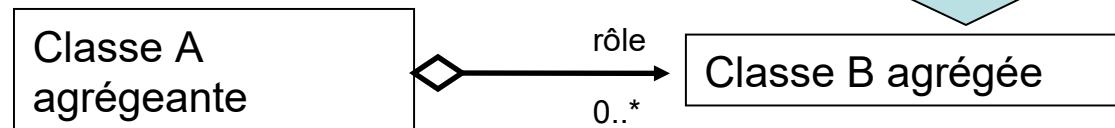
*Aucune des 2 classes n'est plus importante que l'autre*

- Agrégation

Exprime que tout Objet d'une Classe A **Possède** (ou peut posséder) une (des) instance(s) d'une Classe B.

Les Objets A et B ont des vies indépendantes.

Notation UML

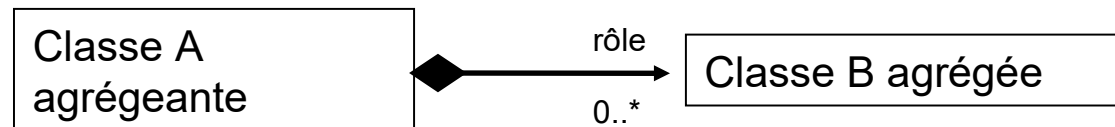


*Modélise une relation tout/partie*

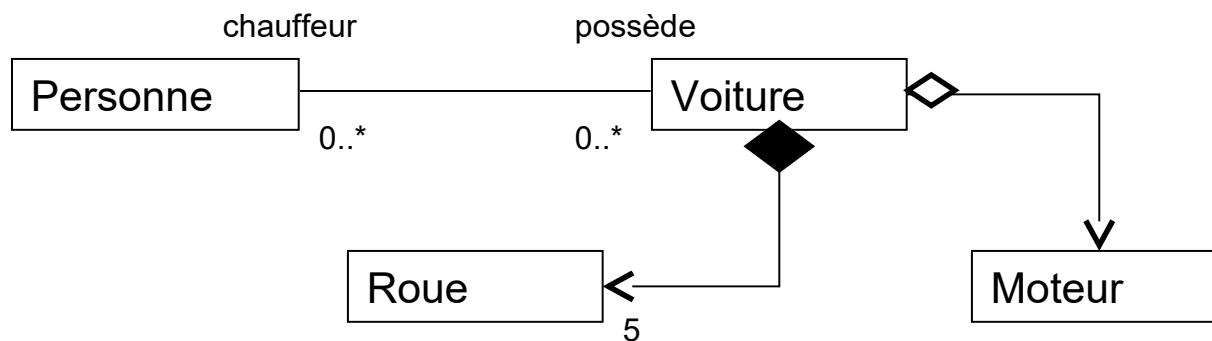
- Composition

Exprime une agrégation pour laquelle la vie des objets agrégés B dépend de la vie de l'objet A

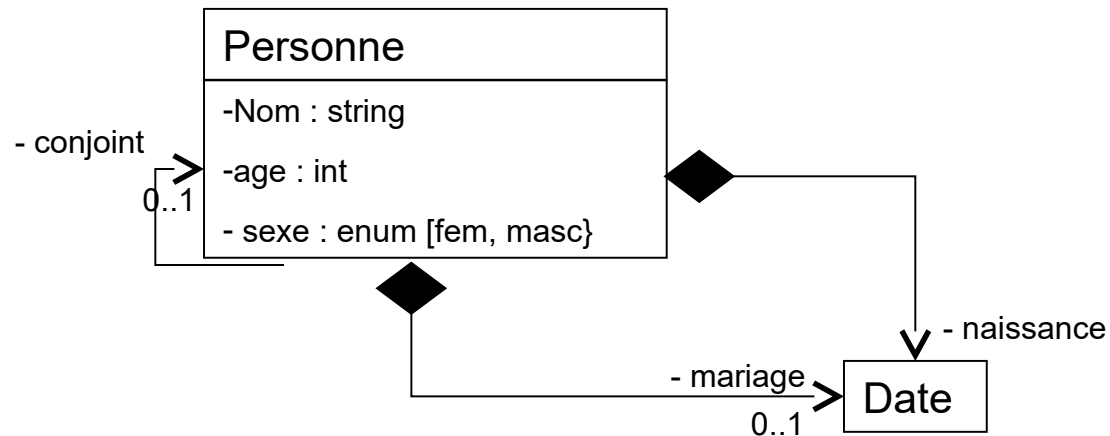
Notation UML



- Exemples
  - Relations entre une Voiture, une Personne, les Roues, le Moteur



- Exemple  
Une Personne a :
  - Un nom
  - Un âge
  - Un sexe {masculin, féminin}
  - Date de naissance
  - Éventuel conjoint
  - Éventuelle date de mariage

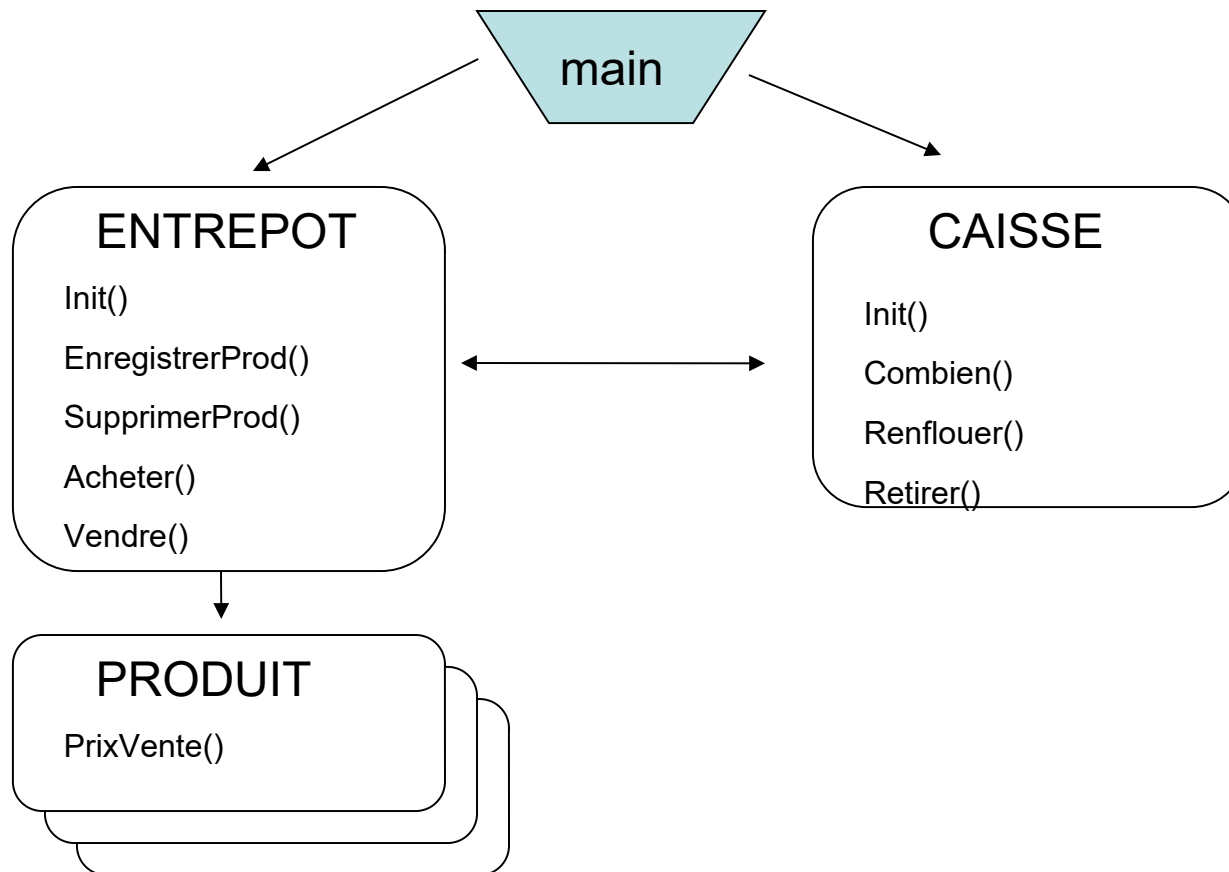


## Exercice Le polygone



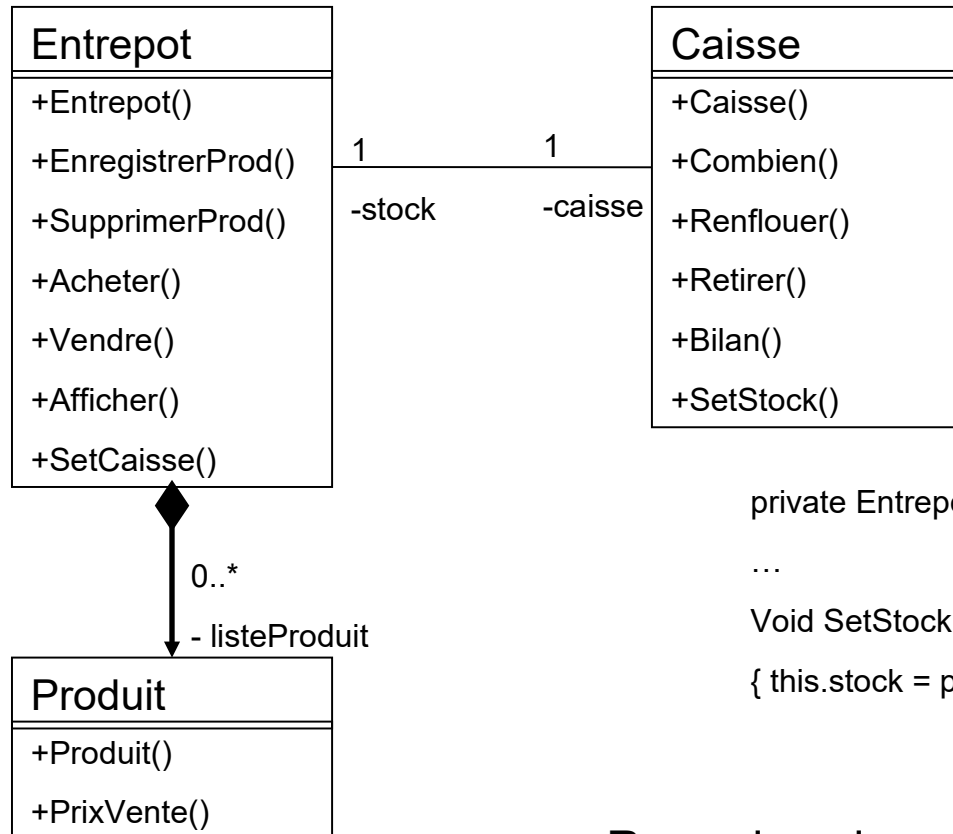
- Exemple

L'entrepôt : transcrire ce schéma en UML



```
private Caisse caisse;
```

```
private List<Produit> listeProduit = new List<Produit>();
```



```
private Entrepot stock;
```

```
...
```

```
Void SetStock(Entrepot pStock)
```

```
{ this.stock = pStock; }
```

Regardons le code C#



- A ce stade :
  - Une Classe définit **un moule** d'Objets. L'opérateur **new** permet de créer un objet, instance de Classe  
Voiture voiture = new **Voiture()**; // appel du constructeur

**Voiture()** est un constructeur de la Classe Voiture.

Cette Méthode permet d'initialiser l'objet.

Le destructeur de Classe rend l'Objet **~Voiture()** en C#, finalize() en java

Toute Méthode, y compris le constructeur peut avoir plusieurs signatures i.e. plusieurs séries de paramètres et retour : la **surcharge**

```
float Ajouter(float val) {};  
int Ajouter(int val) {};  
int Ajouter(int val1, float val2) {};
```

Une méthode est appelée en utilisant généralement le '.'  
voiture.Demarrer();

- A ce stade :
  - Les objets sont reliés entre eux par des **Relations**, visibles dans le diagramme de Classe
    - Dépendance
    - Association, uni directionnelle ou mutuelle
    - Agrégation
    - Composition

- Soit une Classe B héritée de la Classe A.  
Tout Objet de la Classe B possède les mêmes propriétés (Attributs et Méthodes) que les Objets de Classe A, plus des propriétés spécifiques.

Héritage quand correspond à « **est une sorte de** »

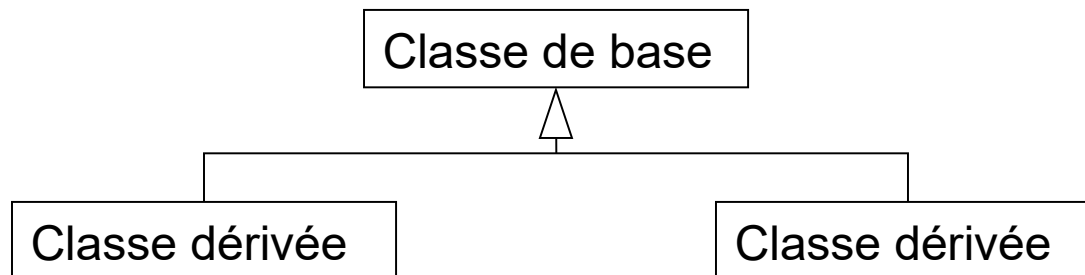
Ex : un Rectangle est une sorte de Polygone

- Classer les mots suivants dans le sens de l'héritage (qui hérite de qui)
  - Banane
  - Animal
  - Légume
  - Poireau
  - Fruit
  - Végétal
  - Corail



- Terminologie  
Une **classe dérivée** hérite d'une **classe de base**  
Une **sous-classe** hérite d'une **super classe**

- Notation UML



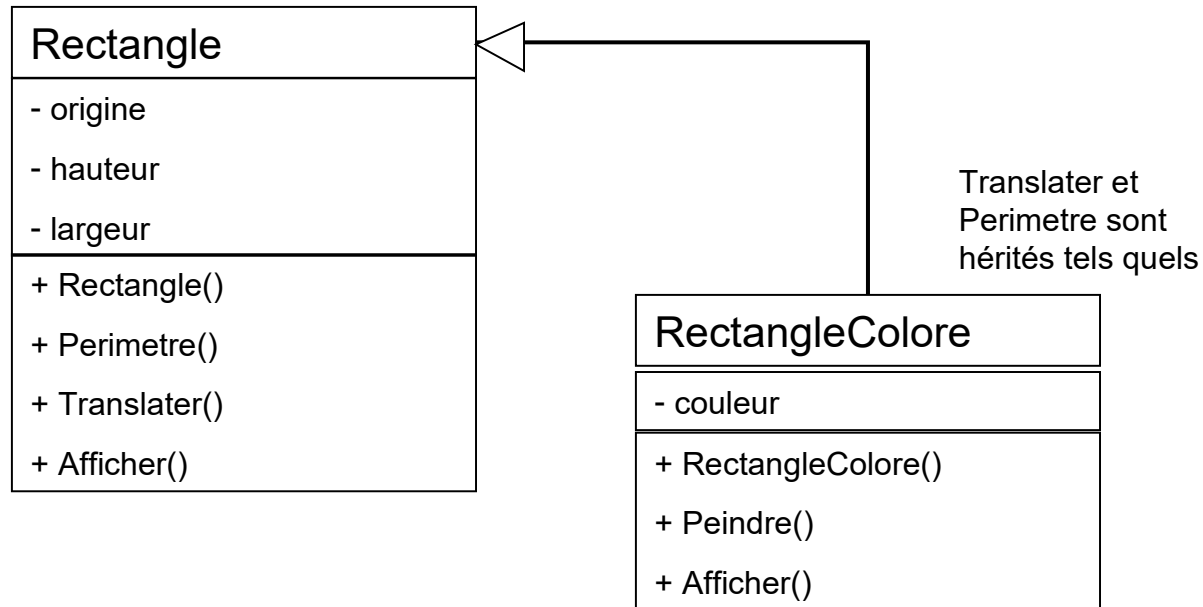
Vu de la classe dérivée, la classe de base est une **Généralisation**

Vu de la classe de base, la classe dérivée est une **Spécialisation**

Permet de **capitaliser le code** dans la classe de base

- Héritage et attributs
  - Impossible de « retrancher » des attributs dans une classe héritée
  - Possible d'ajouter des nouveaux attributs
  - Possible d'ajouter des nouvelles relations
- Héritage et Méthodes
  - Impossible d'enlever des Méthodes dans une classe héritée
  - Possible de créer de nouvelles Méthodes
  - Possible de « surcharger » des Méthodes existantes  
Sous réserve que dans la classe de base la Méthode soit Public ou Protected

- Exemple



C#

```
public class RectangleColore : Rectangle
...
public string Afficher() : base()
{ ... }
```

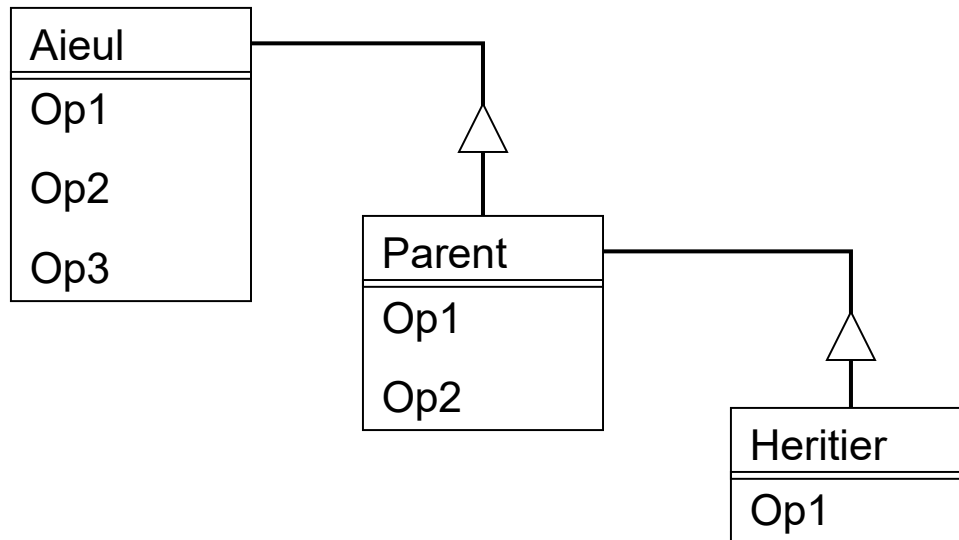
Java

```
public class RectangleColore extends Rectangle
...
public string Afficher()
{ ... super(); ... }
```



Exercice Les figures





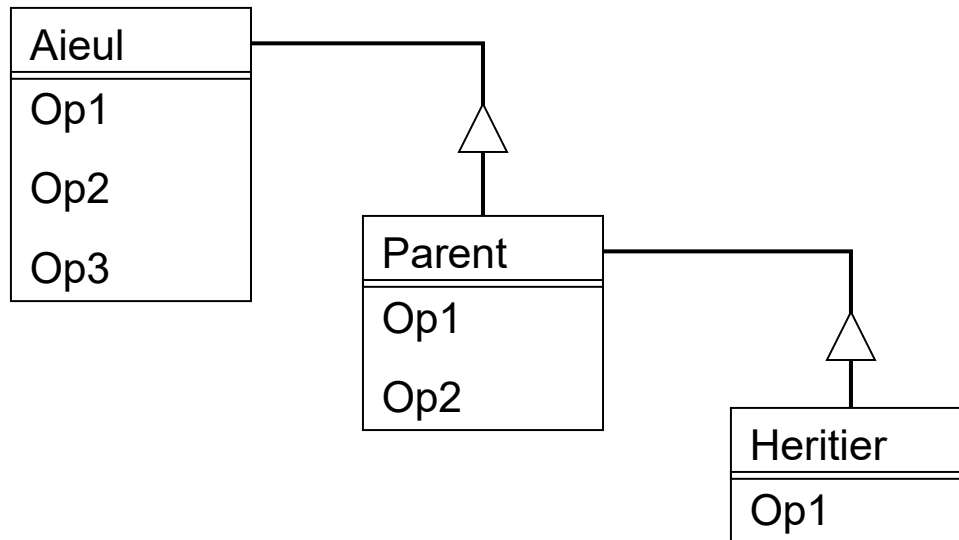
```
Heritier objetH = new Heritier();
```

```
objetH.Op1();           // Op1 défini dans Heritier
objetH.Op2();           // Op2 défini dans Parent
objetH.Op3();           // Op3 défini dans Aieul
```

```
Parent autreObjetP = new Parent();
autreObjetP.Op1();       // Op1 défini dans Parent
```

```
Aieul obj = autreObjetP ;
obj.Op1();               // Op1 défini dans Parent
```

```
Heritier objetH1 = autreObjetP ; // ne compile pas
```



```
Aieul[] tabObjet = new Aieul[3];
```

```
tabObjet[0] = new Aieul();
```

```
tabObjet[1] = new Parent();
```

```
tabObjet[2] = new Heritier();
```

```
For (int i = 0; i < tabObjet.Count; i++)
```

```
    tabObjet[i].Op1(); // Polymorphisme : c'est « le bon » Op1 qui est appelé
```

**IMPORTANT**

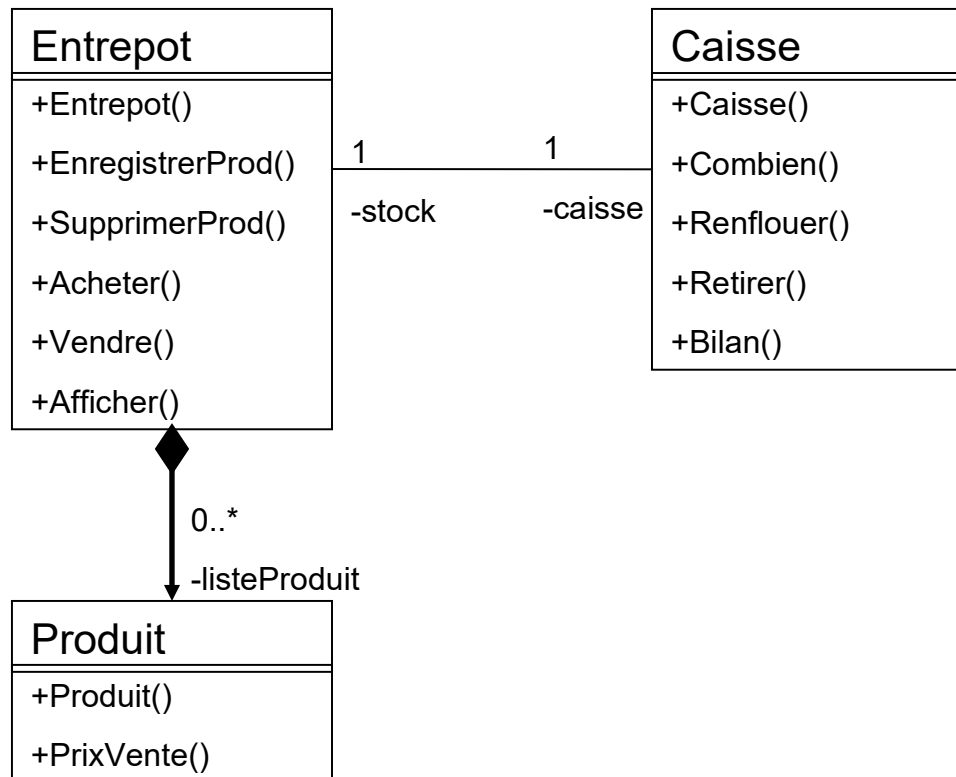


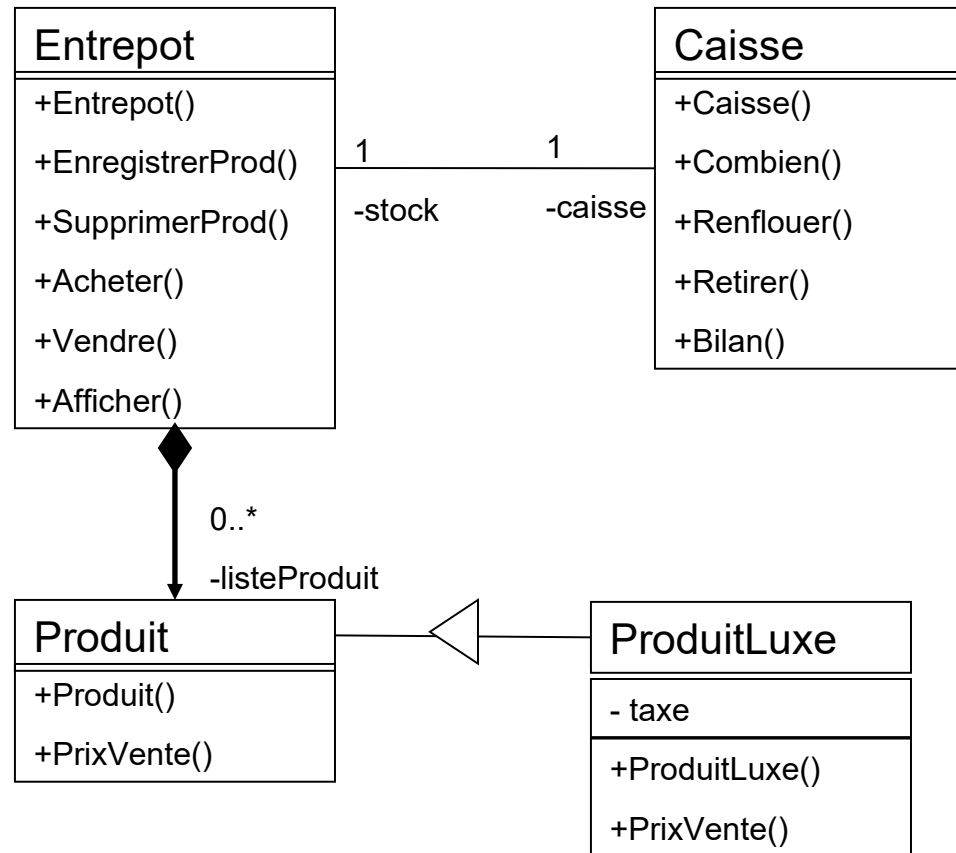
- Exemple avec entrepôt

L'appli évolue pour accepter des produits de luxe.

- Un produit de luxe a une taxe supplémentaire variable d'un produit à l'autre
- La méthode PrixVente est donc modifiée

Compléter le diag UML





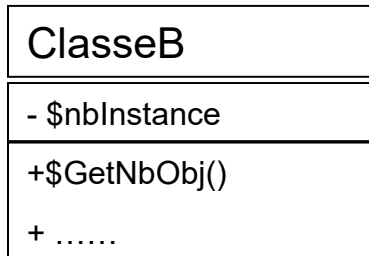
Regardons le code C#



- Jusqu'alors nous avons vu des Attributs et Méthodes qui appartenait à l'Objet

```
ClasseA obj = new ClasseA();  
obj.Attr = valeur;  
obj.Op1();
```

- Il est possible de définir des Attributs et Méthodes globaux à la Classe : utilisation du mot clé **static** et **\$** en UML  
On parle alors **d'Attributs et Méthodes de Classe**



Utilisation chez l'appelant

```
Int nb = ClasseB.GetNbObj();
```

```
public class ClasseB  
{  
    private static int nbInstance;  
  
    public ClasseB()  
    {  
        nbInstance++;  
    }  
  
    ~ClasseB()  
    {  
        nbInstance--;  
    }  
  
    public static int GetNbObj()  
    {  
        return nbInstance;  
    }  
}
```

- Exemple : Classe « Singleton »

Comment s'assurer qu'une classe n'a qu'une instance à l'exécution ?

Moyen :

- Une donnée de classe mémorise la référence de l'unique instance
- Pas de constructeur public
- Une Méthode de Classe rend une référence sur l'unique objet, la crée s'il le faut

Horloge
- \$instance
- valeur
- Horloge() + \$ getInstance() + initialiser()

Utilisation chez l'appelant

```
Horloge clock = Horloge.getInstance();  
clock.Initialiser(0);
```

```
public class Horloge  
{  
    private static Horloge instance;  
    private int valeur = 0;  
  
    private Horloge()  
    { }  
  
    public static Horloge getInstance()  
    {  
        if (instance == null)  
            instance = new Horloge();  
        return instance;  
    }  
    public void Initialiser(int valeur)  
    {  
        this.valeur = valeur;  
    }  
}
```

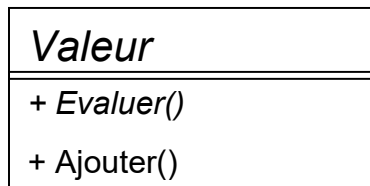
- Réaliser les exercices concernant les diagrammes de classe



- Une classe abstraite ne peut être utilisée qu'en dérivation
- Si une Méthode est abstraite, toute la classe doit être déclarée abstraite
- L'intérêt d'une Méthode abstraite :
  - Seule la signature est définie dans la classe abstraite
  - Les classes dérivées, concrètes, doivent fournir le code de la Méthode
  - Oblige donc une classe dérivée « à ne pas oublier » de définir une Méthode que la classe mère ne peut pas définir.
- L'intérêt d'une classe abstraite :
  - Factoriser au maximum le code dans la classe de base pour ne pas avoir à le définir dans les classes dérivées.
  - Forcer/contraindre les classes dérivées à implémenter une Méthode



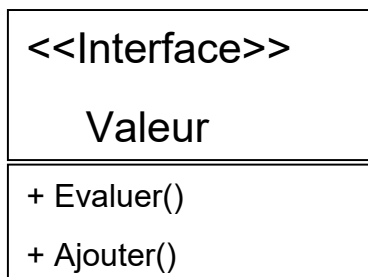
- Côté classe dérivée :
  - Utiliser telles quelles les Méthodes de la classe de base
  - Surcharger des Méthodes de la classe de base avec emploi ou non du code de la classe de base (:base ou super)
  - **Obligation** d'écrire le code des Méthodes abstraites.
- Notation UML : *en italique*



```
public abstract class Valeur
{
    public abstract int Evaluer(); // methode abstraite

    public int Ajouter()
    {
        return 0; // factice
    }
}
```

- Une classe interface est une sorte de classe abstraite :
  - Sans attribut
  - Uniquement avec des méthodes abstraites
- Donc du « Tout abstrait »
- Notation UML : <<Interface>>



C#

```
Public ValeurInt : IValeur
```

...

```
public void Evaluer()
```

```
{ ... }
```

```
public interface IValeur
{
    void Ajouter();

    void Evaluer();
}
```

Java

```
Public ValeurInt implements IValeur
```

...

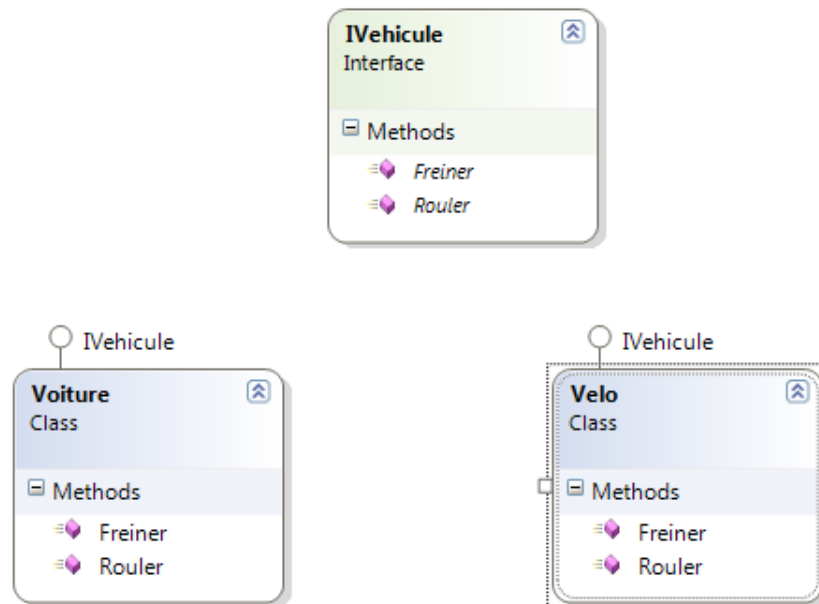
```
public void Evaluer()
```

```
{...}
```

- Une classe peut dériver en général que d'une seule classe (Java, C#) et peut **implémenter** plusieurs interfaces.
- L'interface est à considérer comme un « **Rôle** » que l'on souhaite donner à la classe dérivée.

# Conception orientée Objet    Interface & polymorphisme

- Le polymorphisme fonctionne aussi pour une référence de type interface

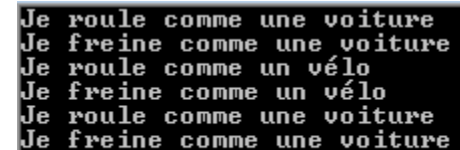


# Conception orientée Objet    Interface & polymorphisme

```
public interface IVehicule
{
    void Rouler(int vitesse);
    void Freiner();
}
public class Voiture : IVehicule
{
    public void Rouler(int vitesse)
    {
        Console.WriteLine("Je roule comme une voiture");
    }
    public void Freiner()
    {
        Console.WriteLine("Je freine comme une voiture");
    }
}
public class Velo : IVehicule
{
    public void Rouler(int vitesse)
    {
        Console.WriteLine("Je roule comme un vélo");
    }
    public void Freiner()
    {
        Console.WriteLine("Je freine comme un vélo");
    }
}
static void Main(string[] args)
{
    List<IVehicule> vehiculeList = new List<IVehicule>();

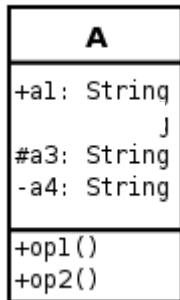
    vehiculeList.Add(new Voiture());
    vehiculeList.Add(new Velo());
    vehiculeList.Add(new Voiture());

    foreach (IVehicule vehi in vehiculeList)
    {
        vehi.Rouler(10);
        vehi.Freiner();
    }
}
```



```
Je roule comme une voiture
Je freine comme une voiture
Je roule comme un vélo
Je freine comme un vélo
Je roule comme une voiture
Je freine comme une voiture
```

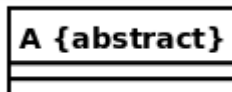
- Principales représentations UML avec exemples en java
  - Classe avec attributs et opérations



```
public class A {
    public String a1;

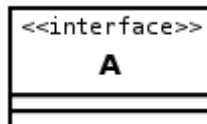
    protected String a3;
    private String a4;
    public void op1() {
        ...
    }
    public void op2() {
        ...
    }
}
```

- Classe abstraite



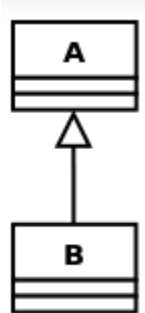
```
public abstract class A {
    ...
}
```

- Interface



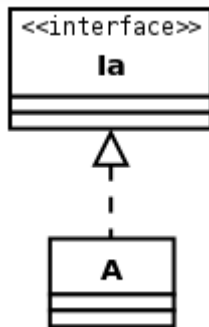
```
public interface A {
    ...
}
```

## – Héritage simple



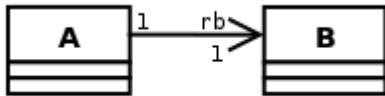
```
public class A {  
    ...  
}  
  
public class B extends A {  
    ...  
}
```

## – Implémentation d'une interface



```
public interface Ia {  
    ...  
}  
  
public class A implements Ia {  
    ...  
}
```

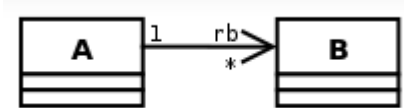
- Association unidirectionnelle 1 vers 1



```
public class A {
    private B rb;
    public void addB( B b ) {
        if( b != null ) {
            this.rb=b;
        }
    }
}

public class B {
    ... // La classe B ne connaît pas l'existence de la classe A
}
```

- Association unidirectionnelle 1 vers n

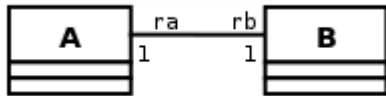


```
public class A {
    private ArrayList <B> rb;
    public A() { rb = new ArrayList<B>(); }
    public void addB(B b) {
        if( !rb.contains( b ) ) {
            rb.add(b);
        }
    }
}

public class B {
    ... // B ne connaît pas l'existence de A
}
```



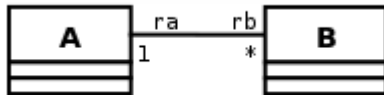
- Association bidirectionnelle 1 vers 1



```
public class A {
    private B rb;
    public void addB( B b ) {
        if( b != null ){
            if ( b.getA() != null ) {    // si b est déjà connecté à un autre A
                b.getA().setB(null);    // cet autre A doit se déconnecter
            }
            this.setB( b );
            b.setA( this );
        }
    }
    public B getB() { return( rb ); }
    public void setB( B b ) { this.rb=b; }
}

public class B {
    private A ra;
    public void addA( A a ) {
        if( a != null ) {
            if ( a.getB() != null ) {    // si a est déjà connecté à un autre B
                a.getB().setA( null );    // cet autre B doit se déconnecter
            }
            this.setA( a );
            a.setB( this );
        }
    }
    public void setA(A a){ this.ra=a; }
    public A getA(){ return(ra); }
}
```

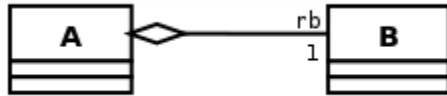
- Association bidirectionnelle 1 vers n



```
public class A {
    private ArrayList <B> rb;
    public A() { rb = new ArrayList<B>(); }
    public ArrayList <B> getArray() {return(rb);}
    public void remove(B b) {rb.remove(b);}
    public void addB(B b) {
        if( !rb.contains(b) ){
            if (b.getA() != null) b.getA().remove(b);
            b.setA(this);
            rb.add(b);
        }
    }
}

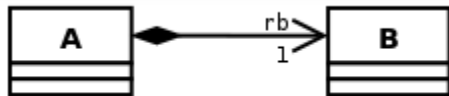
public class B {
    private A ra;
    public B() {}
    public A getA() { return (ra); }
    public void setA(A a){ this.ra=a; }
    public void addA(A a){
        if( a != null ) {
            if( !a.getArray().contains(this)) {
                if (ra != null) ra.remove(this);
                this.setA(a);
                ra.getArray().add(this);
            }
        }
    }
}
```

- Agrégation



S'implémente comme l'association unidirectionnelle 1 vers 1 ou 1 vers n

- Composition



idem avec le fait en plus que c'est A qui crée l'objet B

- Liens <https://lipn.univ-paris13.fr/~gerard/uml-s2/uml-cours01.html>  
et <https://lipn.univ-paris13.fr/~gerard/uml-s2/uml-cours02.html>

# Conception orientée Objet    Classes patron

- Les classes paramétrées ou « patron » de classe
- Certains langages permettent de définir des classes paramétrées :
  - Exemple de ListArray<> en java
  - Exemple de Array<> en java
- Lors de la définition d'une telle classe le type est fourni de façon anonyme.

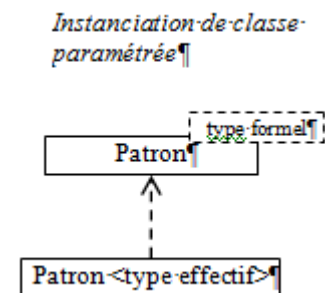
## Définition

```
class Generic1<T> {  
    public static void Echanger(ref T value1, ref T value2){  
        // on échange les références value1 et value2  
        T temp = value2;  
        value2 = value1;  
        value1 = temp;  
    }  
}
```

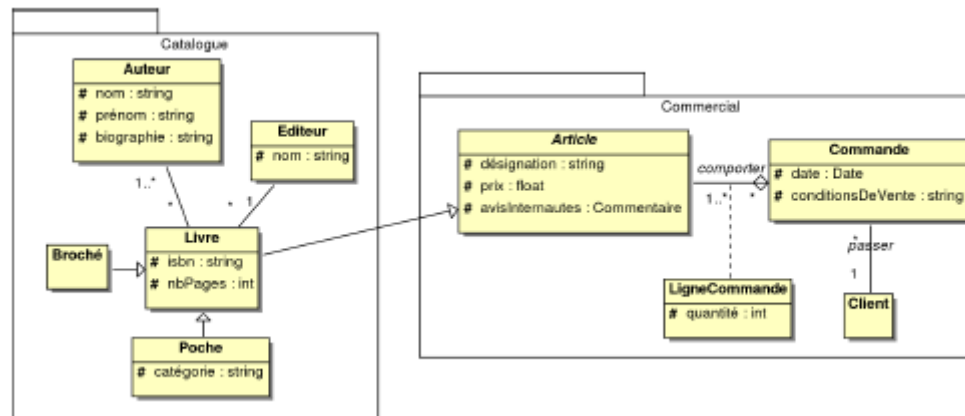
## Emploi

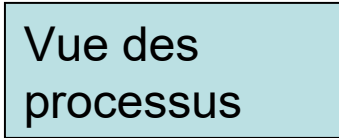
```
// int  
int i1 = 1, i2 = 2;  
Generic1<int>.Echanger(ref i1, ref i2);  
Console.WriteLine("i1={0},i2={1}", i1, i2);  
// string  
string s1 = "s1", s2 = "s2";  
Generic1<string>.Echanger(ref s1, ref s2);  
Console.WriteLine("s1={0},s2={1}", s1, s2);
```

## Représentation UML



- Un diagramme de package peut servir à définir des sous ensembles dans la conception objet
- Le concept se traduit alors en
  - « package » pour java
  - Module ou « package » pour python
  - « Namespace » pour C#
  - ...



- Le diagramme de classe fournit un aspect qualitatif et une vision **statique** des objets mis en jeu.  
Il est insuffisant pour décrire l'existence des objets dans la marche de l'application
- En UML le **diagramme d'objets** permet d'apporter cette précision. Il représente les objets du système à un instant donné.
- Fait partie de  dans la présentation générale UML

3

- Le diagramme de classes modélise des règles et le diagramme d'objets modélise des faits à un instant donné.
- Instance anonyme
- Instance nommée
- multiples

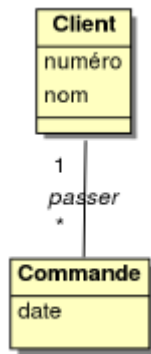
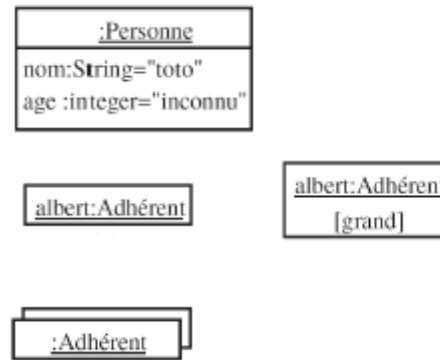


Diagramme de classe

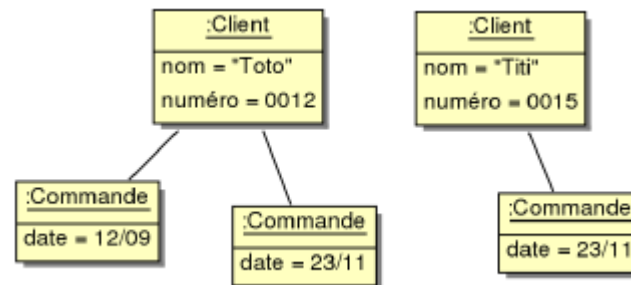


Diagramme d'objets

- Exercices
  - Un avion en détresse
  - Instanciation d'un diagramme de classes

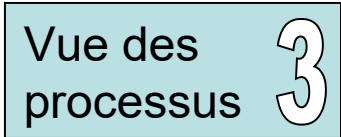




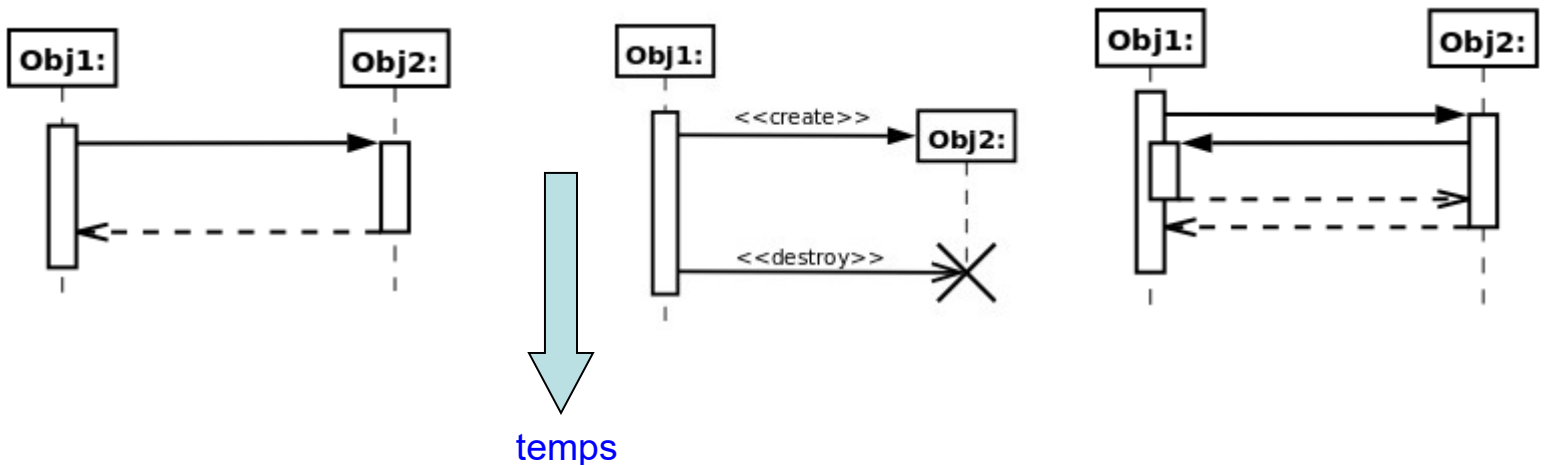
3

Vue des  
processus

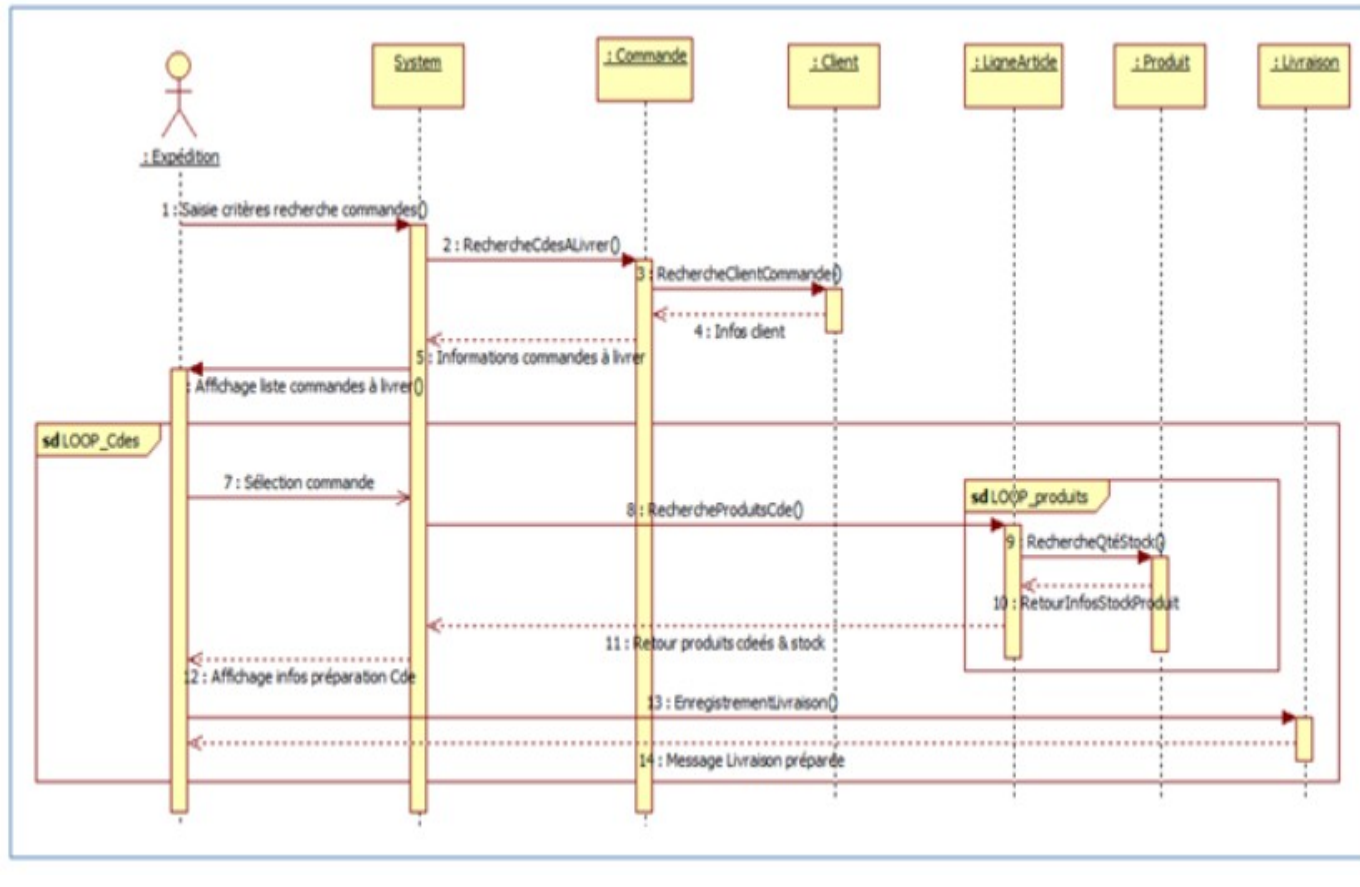
- Cet angle d'analyse décrit :
  - Le découpage du système en processus et actions
  - Les interactions entre processus
- UML propose pour les représenter :
  - Le diagramme de séquence
  - Le diagramme d'activité

- Le diagramme de classe fournit un aspect qualitatif et une vision **statique** des objets mis en jeu.  
Il est insuffisant pour décrire l'enchaînement des méthodes et l'interaction avec l'extérieur
- En UML le **diagramme de séquence** permet d'apporter cette précision. Ajoute une vision **dynamique** du système.
- Fait partie de  dans la présentation générale UML

- Traduit l'interaction des objets dans le temps
  - Chaque diagramme comporte un titre qui qualifie le cas
  - Il y a autant de diagrammes que jugé nécessaire à la compréhension
  - Sont à faire pendant ou après le diagramme de classes



- Exemple d'un diagramme de séquence (source openclassrooms)



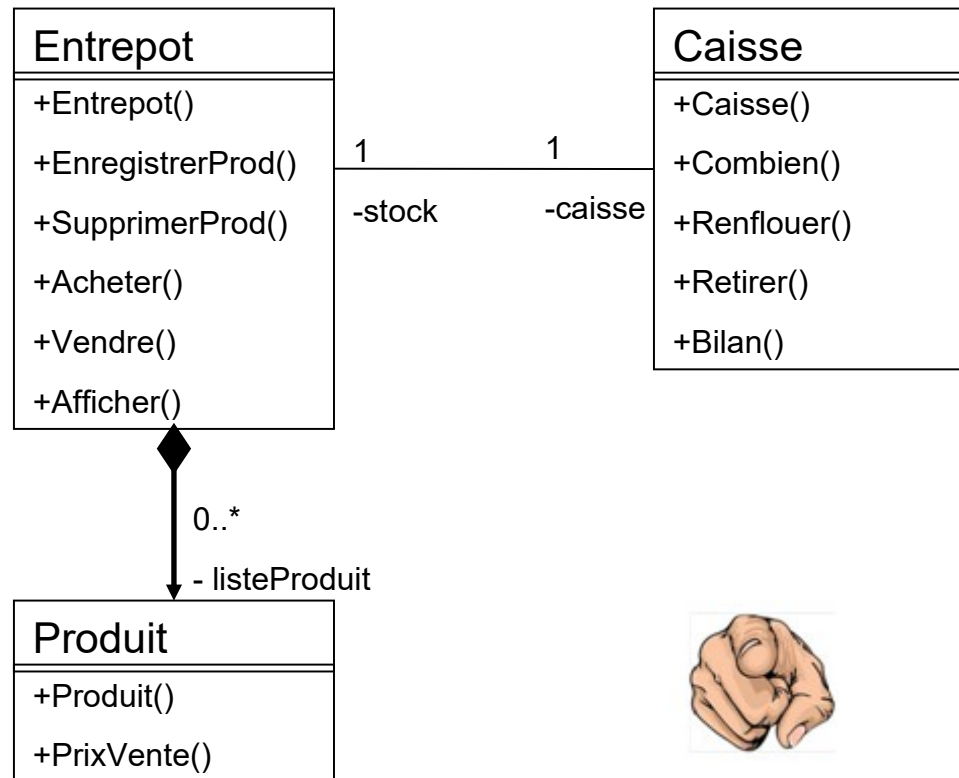
Le rectangle 'LOOP' représente une boucle  
Opt/alt sont possibles.

temps

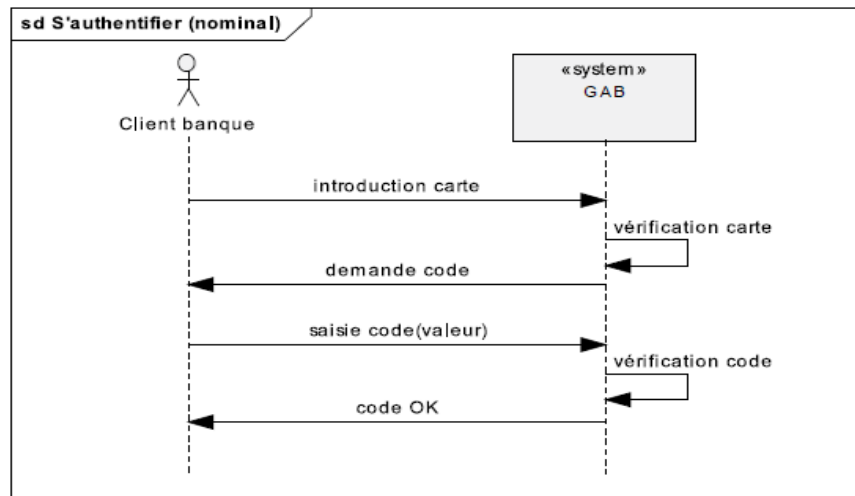
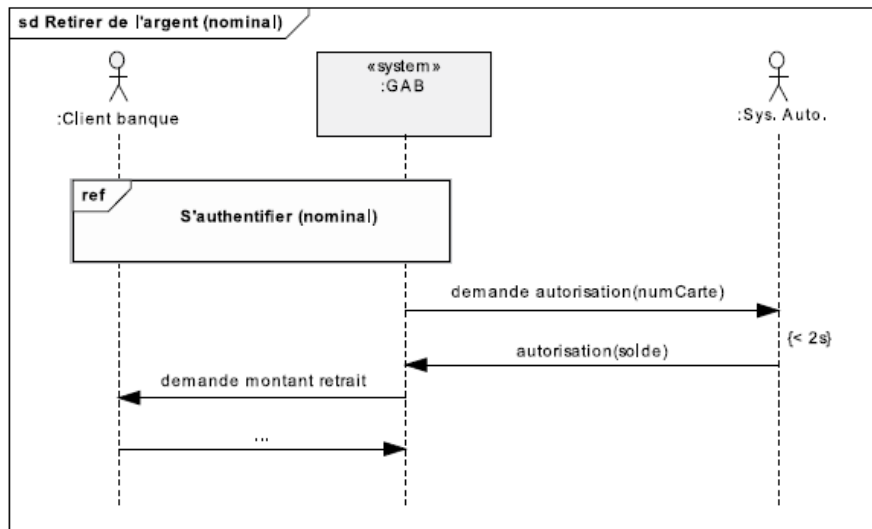
- Pour plus de précision :

<https://lipn.univ-paris13.fr/~gerard/uml-s2/uml-cours05.html>

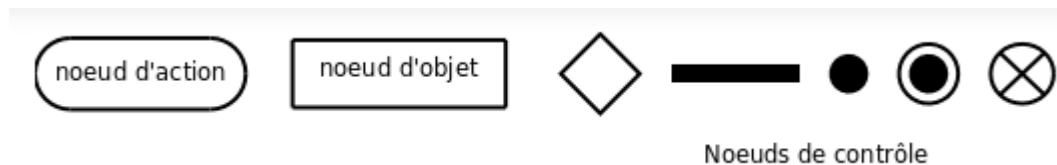
- Pour l'application Entrepôt, réaliser les diagrammes pour l'Init de l'application et enregistrer un produit.



- Un diagramme de séquence peut aider à la compréhension d'un use case, d'un point de vue chronologique.



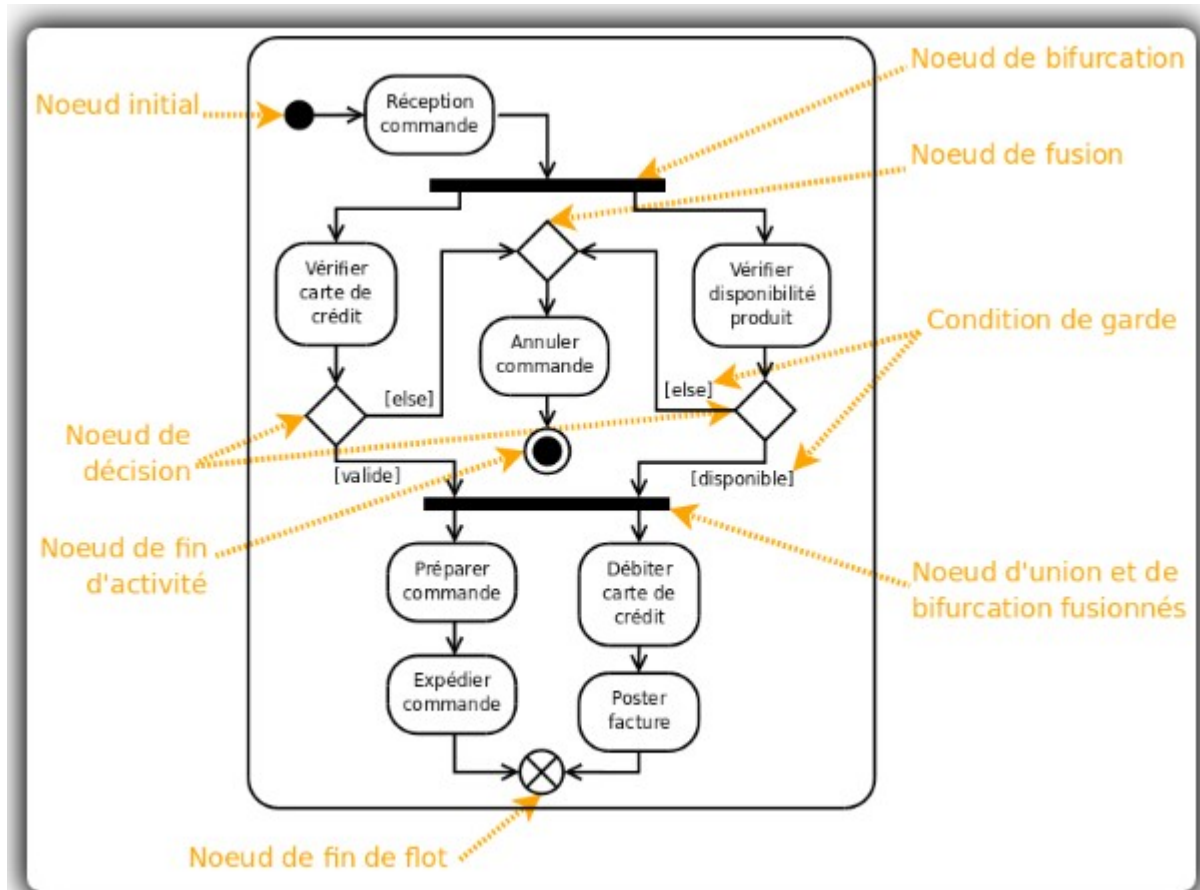
- Le diagramme d'activité
  - Mettent l'accent sur les traitements
  - Permet de représenter autrement un cas d'utilisation



- Éléments graphiques
  - le noeud représentant une action, qui est une variété de noeud exécutable,
  - un noeud objet,
  - un noeud de décision ou de fusion,
  - un noeud de bifurcation ou d'union,
  - un noeud initial,
  - un noeud final,
  - et un noeud final de flot

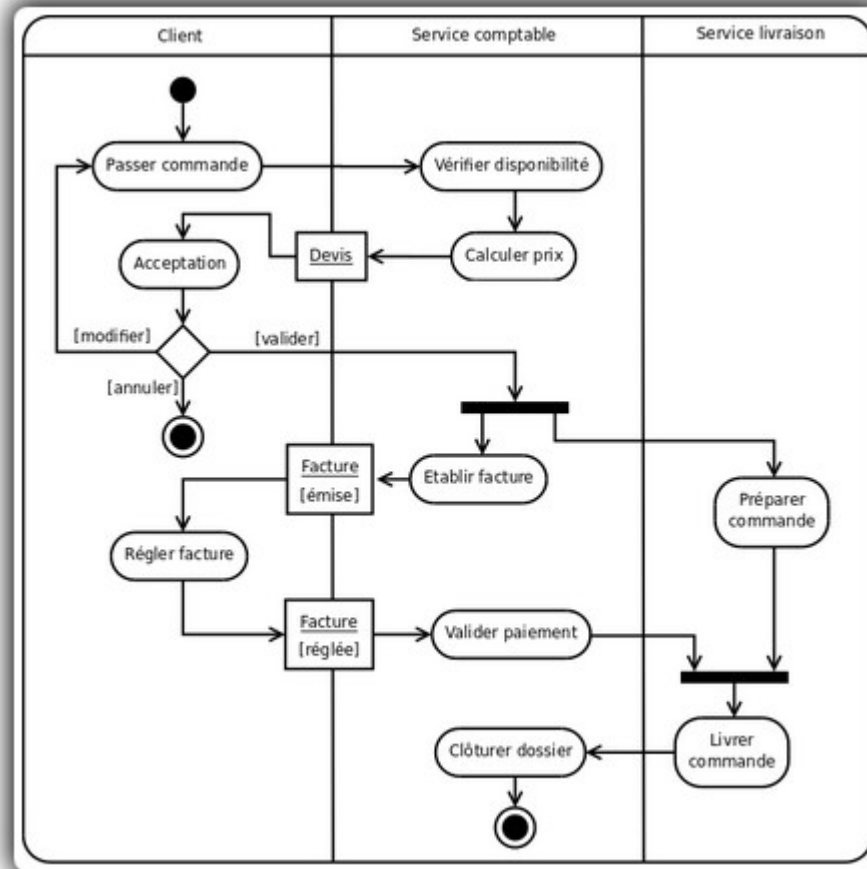
- Exemple de diagramme d'activité

(source <https://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-activites> )





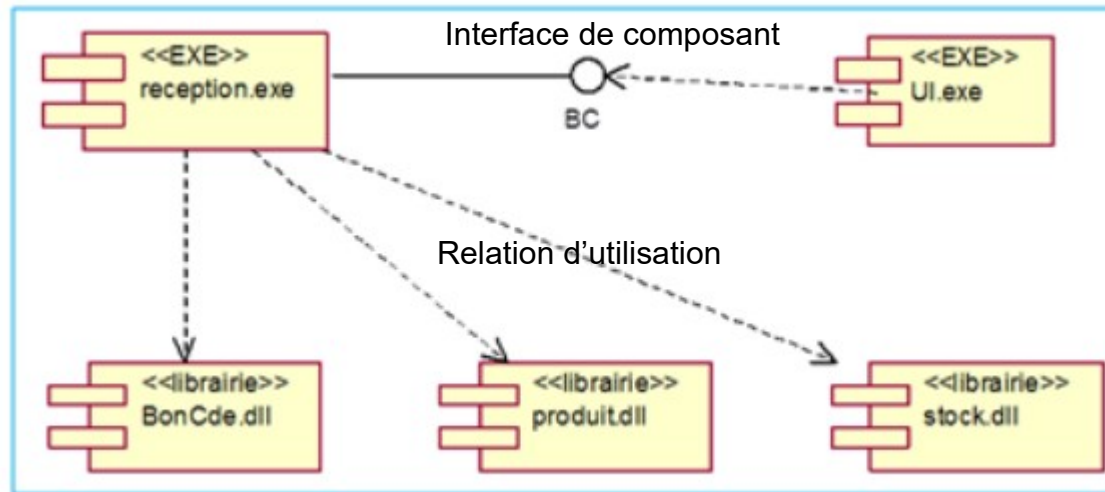
- Exemple de diagramme d'activité, représentation en partition ou en couloir



## 4

## Vue des composants

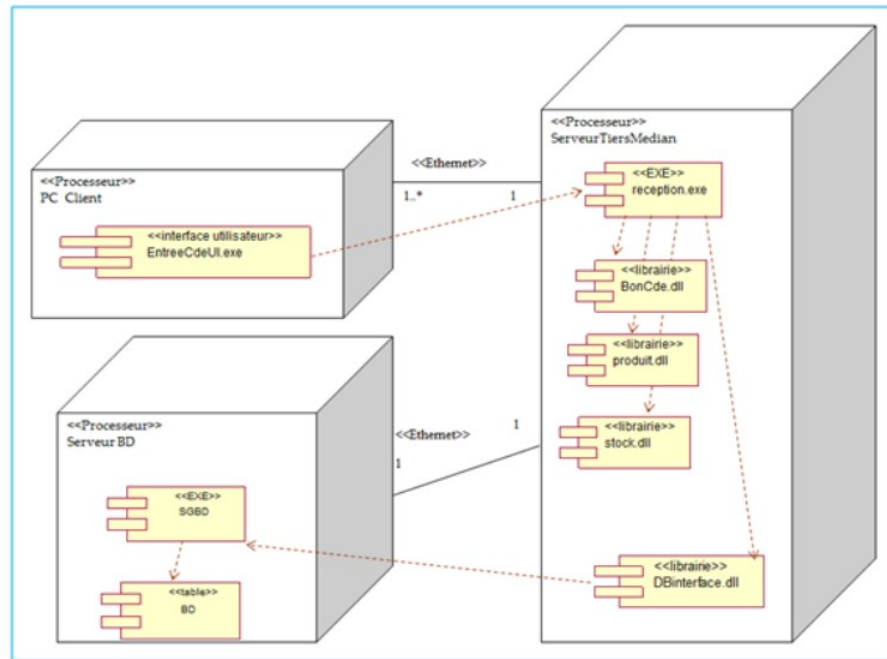
- Cet angle identifie les parties qui composeront le système : fichiers, exécutables, bases de données ... et leurs dépendances
- Le diagramme de composant existe pour cela



## 5

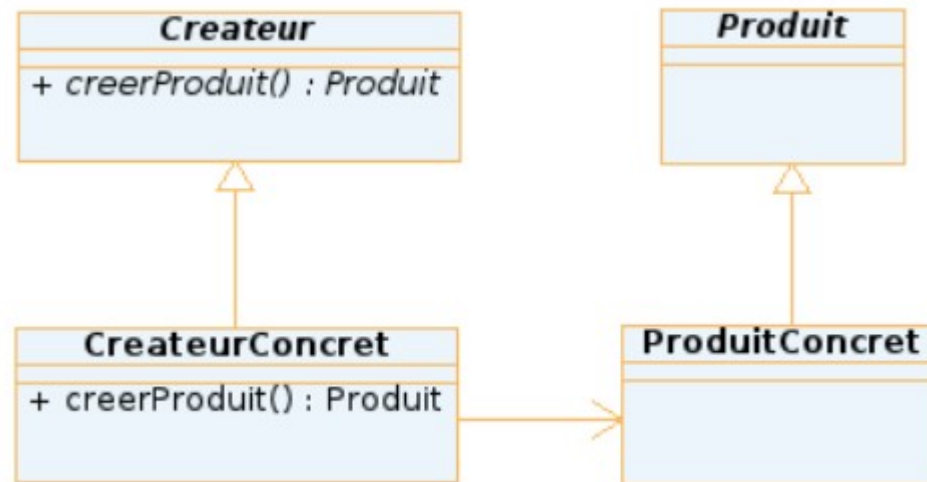
Vue de  
déploiement

- Cet angle de vue décrit la répartition des parties du logiciel sur les ressources matérielles de la cible.
- UML propose le diagramme de déploiement



- Dans le monde des applications informatiques un certain nombre de mécanismes, de façon de résoudre des **problématiques récurrentes** ont été identifiées par les concepteurs.
- Plutôt que de tout réinventer, des documents ont été produits pour **décrire ces cas** et surtout **proposer des solutions 'élégantes'** : qui fonctionnent et auxquelles 'on n'aurait pas pensé'.
- Ceci se nomme des Design Patterns – Patron de conception en français.
- Ceci est à comparer aux ouvrages concernant **les meilleurs coups** au jeu d'échec.

- Exemples
  - Le singleton
  - Factory Method  
<http://design-patterns.fr/fabrique>



- Un Framework est un ensemble de bibliothèques logicielles qui constitue la fondation et une grande partie d'un logiciel applicatif
  - Propose une architecture logicielle
  - Propose des patrons de conception
  - Consiste souvent à créer des classes héritées ou à implémenter des interfaces du framework
- Une bibliothèque offre des Classes « passives » : c'est au codeur de créer une architecture/conception pour organiser les divers objets de classes

Un Framework est « actif » dans la mesure où le cœur crée et appelle déjà les objets/méthodes . Il utilise pleinement le pattern Observer (fonctions de rappel, événements)

- Exemples :
  - Framework .Net chez Microsoft pour applications
  - Java SE pour applications Java
  - Bootstrap, Symfony, Laravel ... en PHP pour applications Web
  - Django en Python pour applications Web
  - NodeJS en PHP pour serveurs Web
  - Xamarin pour applications mobiles
  - Corona pour jeux sur mobiles
  - ....

- Non abordés : diagrammes état/transition
- Pour la conception des bases de données : méthode Merise ou méthode UML  
La méthode UML est proche des diagrammes de classes UML  
MySQLWorkbench.exe crée la structure BD à partir du diagramme.



- Les diagrammes UML sont le support de réflexion pour l'analyse d'un système et sa conception.
- La démarche itérative permet d'approcher une analyse et conception de plus en plus complète au cours du projet

