Conception et programmation orientée objet

Table des matières

I - Fondamentaux orientés "objet

1. Paradigme "objet" (vs "procédural")

L'histoire de l'informatique se confond avec celle de ses langages et des idées /paradigmes associés.

De l'assembleur jusqu'aux langages les plus sophistiqués de l'intelligence artificielle la programmation a évolué vers:

- ***des techniques de plus en plus détachées du matériel***
- une tentative de plus grande ***modularité***
- l'accroissement de la puissance et de ***l'expressivité*** des langages

Le tableau ci-dessous dresse la liste des principaux modes de programmation (qui ne sont pas obligatoirement exclusifs):

Programmation séquentielle et structurée	Un programme (automate) est un ***ensemble d'instructions qui manipulent un ensemble de données***	Cobol, Fortran, C, Pascal, Ada, …
Programmation événementielle	Un programme est une collection de ***morceaux de code*** qui sont automatiquement ***déclenchés quand arrive un certain événement***	VB, POWER-BUILDER, beaucoup d'autres **L4G**
Programmation fonctionnelle	Un programme est le représentant informatique d'une fonction mathématique qui associe des valeurs de sorties à des valeurs d'entrée	**LISP** (avec ses nids de parenthèses et sa récursivité)
Programmation logique	Un programme est un raisonnement et son exécution une preuve de la déductibilité d'une formule logique	**PROLOG** (avec ses prédicats)
Programmation objet	**Un programme est un ensemble de petites entités ayant chacune leur propre état et comportement et qui communiquent par messages.**	**SMALLTALK, C++, JAVA, C#, ...**

Le tableau ci-dessous dresse la liste des principaux langages orientés objets (ceux qui ont marqué l'histoire de l'informatique):

1967→ **SIMULA**	Génèse de la notion d'objet (objet = instuctions+données) Introduit la notion de classe (type) d'objet Langage créé pour résoudre des problèmes de simulation
1972 → **SMALLTALK** (Xerox)	Entièrement orienté objet (un entier est une instance d'une classe). Une classe est elle même une instance d'une méta-classe.
1983 → **C++**	Langage objet compatible avec le C et compilé ➔ bonnes performances. très grande expressivité . grande complexité
1995 → **JAVA** (Sun)	Langage récent indépendant de la plate-forme (UNIX,NT,…) et accompagné d'une imposante bibliothèque de classes prédéfinies. Intégration fine dans le WEB (Applet)
2001 --> **C#** (Microsoft)	Langage objet se voulant un peu un hybride java/c++

2. Concepts objets fondamentaux

Classe	Type d'objet . Tous les objets d'une même classe ont la même structure (attributs / données internes) et le même comportement (méthodes / fonctions membres)
Instance	Objet (exemplaire) créé à partir d'une certaine classe (via new en java)
Héritage	Une classe (dite dérivée) hérite d'une classe (dite de base) lorsqu'elle ajoute des spécificités. [ex: Employe est une sorte de Personne, avec un salaire en plus]
Polymorphisme	Plusieurs variantes possibles pour une même opération générique , sélection automatique en fonction du type exact de l'objet mis en jeu.
Interface	Classe complètement abstraite (sans code mais avec prototypes de fonctions) permettant de préciser un contrat fonctionnel .

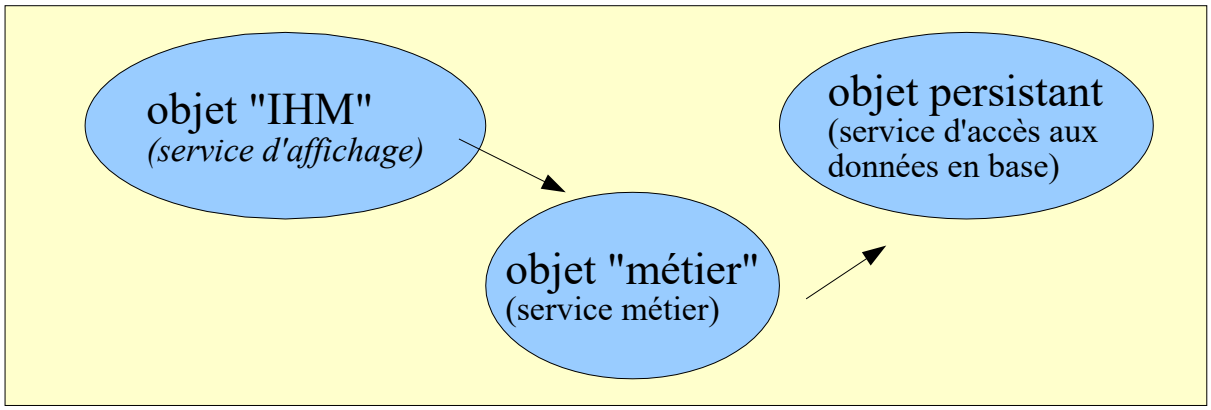
--> approfondir si besoin en cours avec le formateur.
--> approfondir si besoin ensuite via Wikipédia ou "comment ça marche" ou autre.

Avant tout,

Un objet (informatique ou ...) **est une entité qui rend un (ou plusieurs) service(s)** .Sinon, il n'a aucune raison d'être.

Une application modulaire est une collection (assemblage) d'objets spécialisés offrant chacun des services complémentaires.

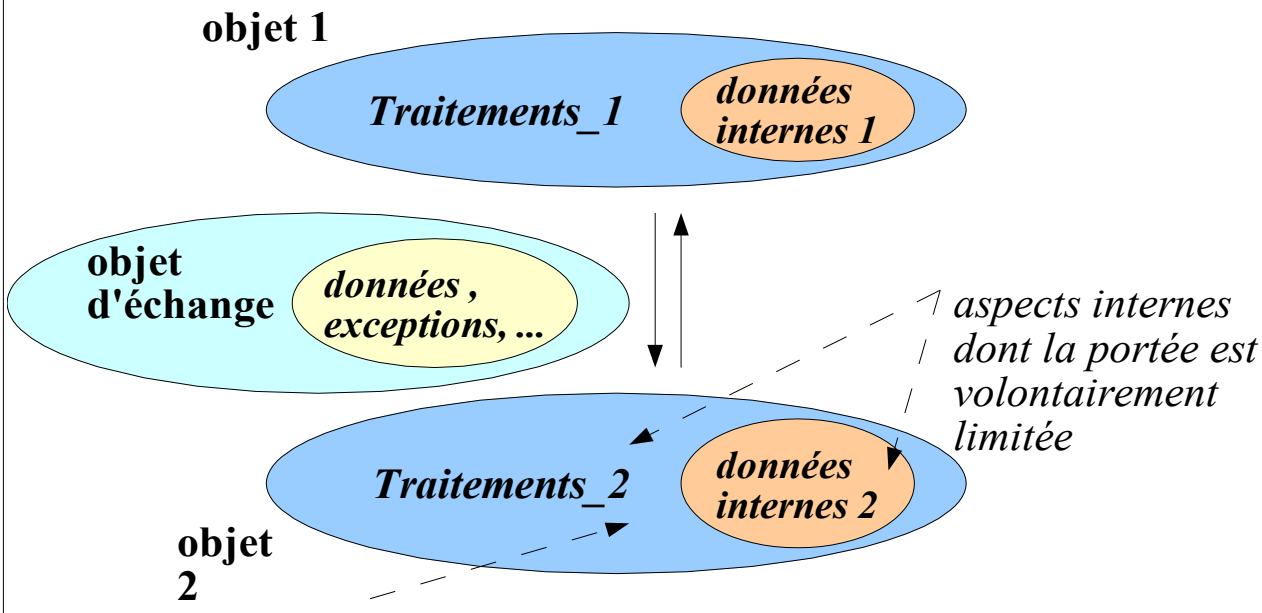
Un objet interagit avec un autre en lui envoyant des messages de façon à solliciter certains services.



Grand traits de l'approche objet:

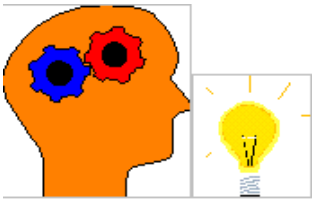
Un programme est un ensemble de petites **entités** . Chacune ayant :
- son propre **état** *(lié au **données internes** et aux inter-relations)*
- son propre **comportement** *(**traitements internes** pour fonctionner)*

Ces entités communiquent entre elles par **messages** (sollicitations).



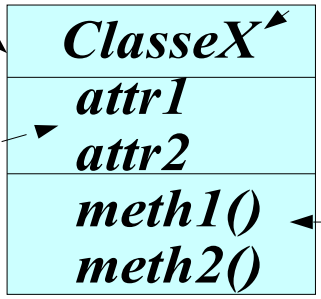
Traitements** en grande partie **locaux** ==> assez bonne **autonomie

Abstraction des données (réalité => types abstraits)



abstraction

De quoi parle-t-on ?



Eléments pertinents ?

Quelle utilité ?
Quels services ?

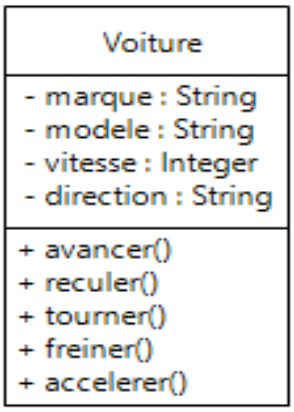
*Le **mécanisme d'abstraction** consiste à **créer ses propres types, appelés types abstraits** (ou classes) de façon à les intégrer dans une modélisation (vue simplifiée) du monde réel .*

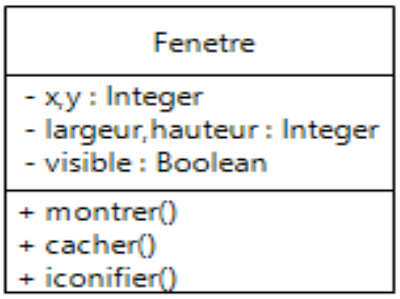
instances

Notion de classe

Pour **regrouper les objets aux caractéristiques et comportements identiques**, on fait appel à la notion de **classe**.
Une **classe** peut être vue comme un **type** *(abstrait ou concret)* **d'objets**.

Une classe est une sorte de moule à partir duquel seront générés les objets que l'on appelle **instances** de la classe. *[Un objet est créé à l'image de sa classe]*



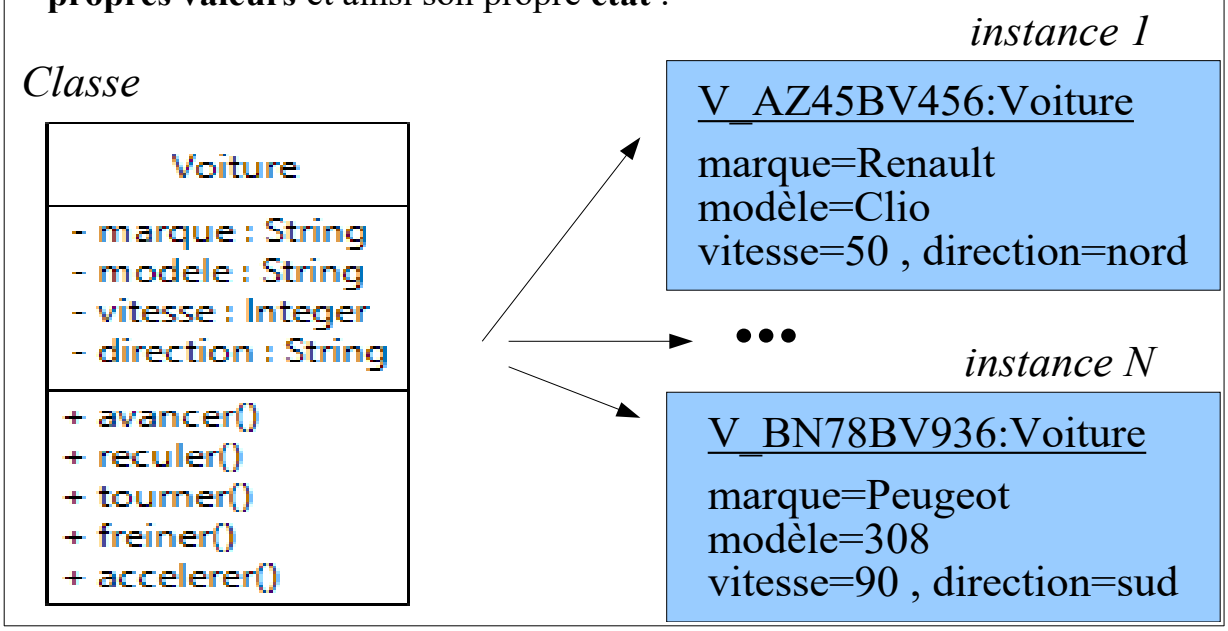


Toutes les instances d'une même classe ont une même structure commune et ont un même comportement.

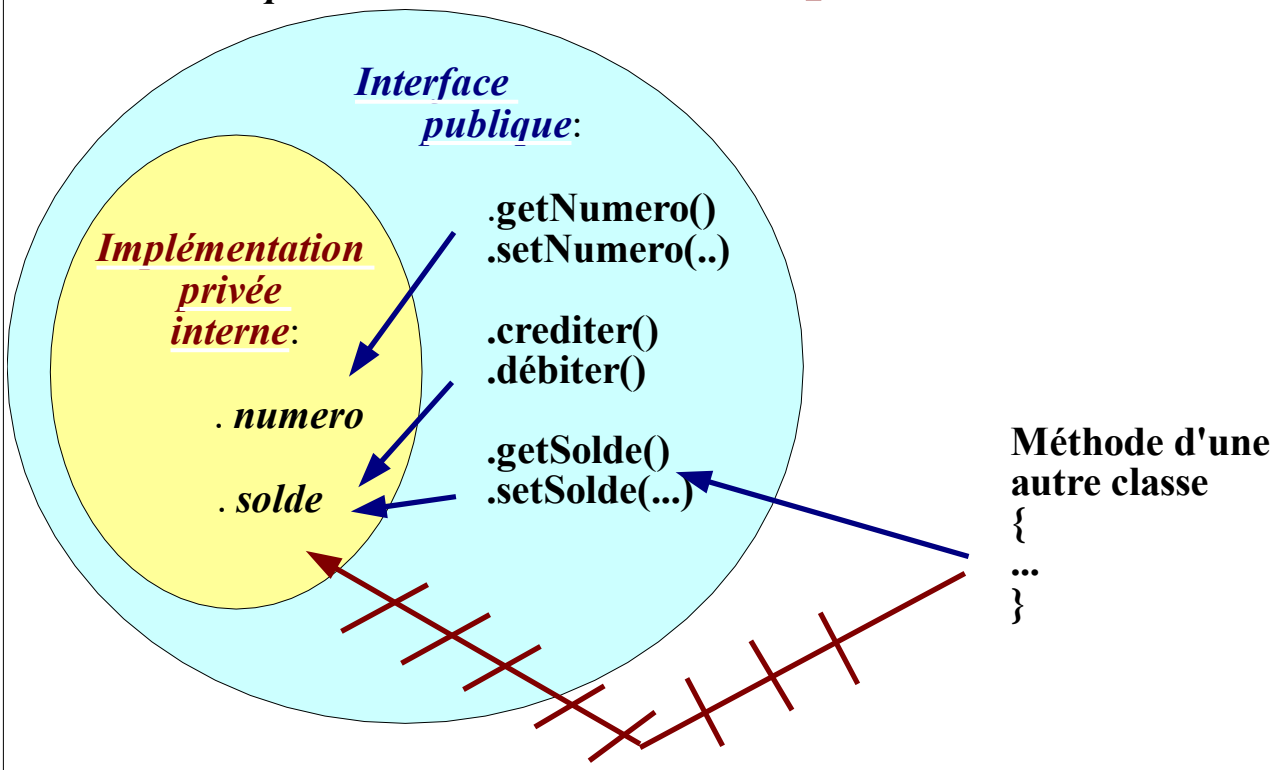
Instances

Rappel: un **exemplaire** (objet) issu à partir d'une certaine classe est appelé une **instance** .

Les différentes instances d'une même classe se distinguent par les **valeurs** *(assez souvent différentes)* **de leurs attributs** ==> **Chaque instance a ses propres valeurs** et ainsi son propre **état** .

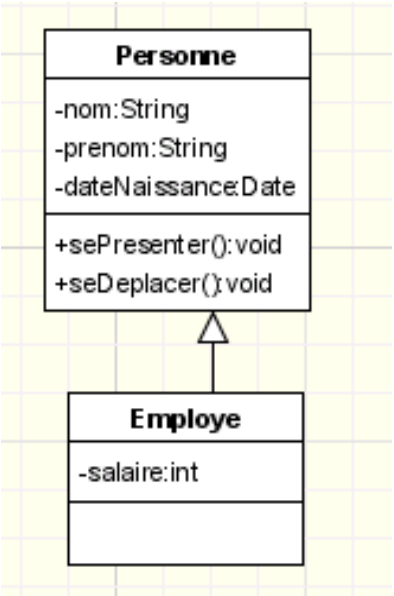


Encapsulation



L'accès direct aux attributs privés est interdit

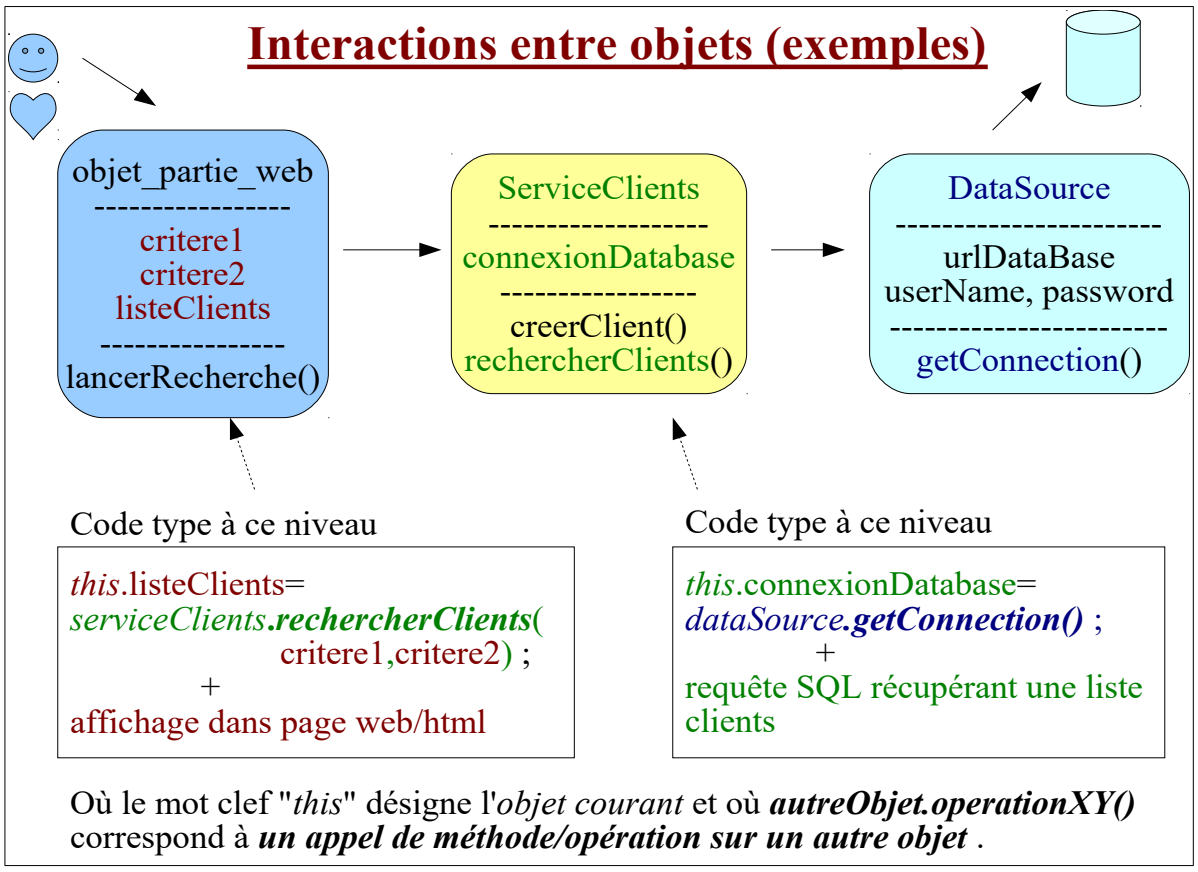
Héritage / Généralisation / Spécialisation



*** La notion d'héritage consiste à définir une nouvelle classe à partir d'une classe existante en spécialisant certaines choses.**

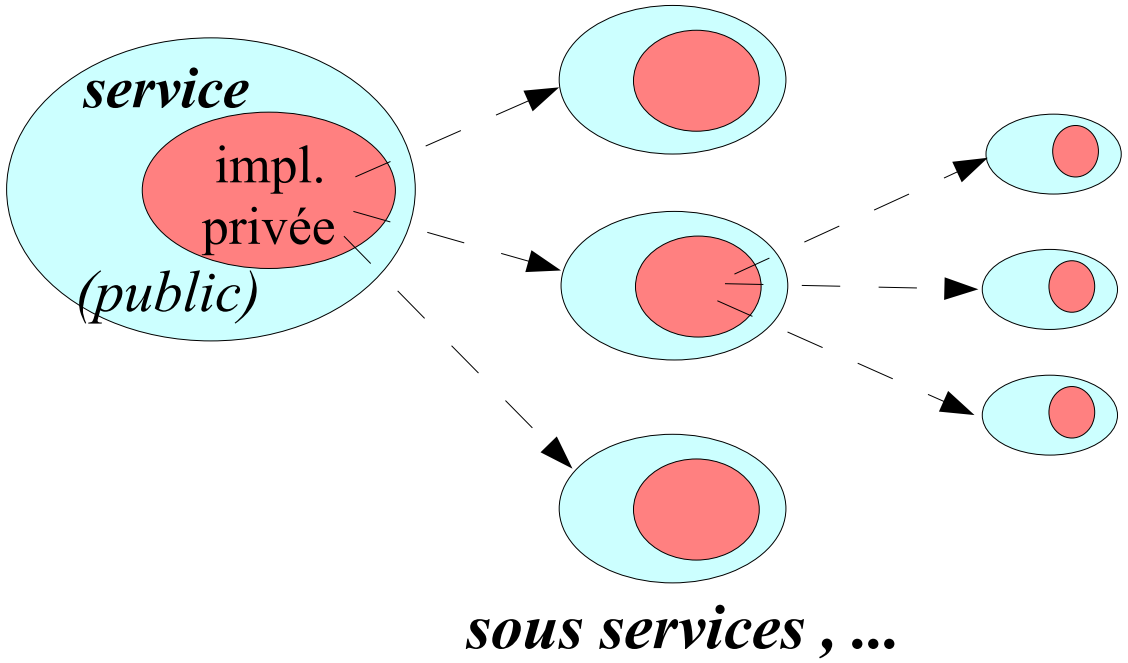
*** La classe dérivée (ou sous classe) reprendra tous les attributs et toutes les opérations de la surclasse .**

*** La structure de données est conservée au niveau des sous classes; elle peut néanmoins être enrichie via l'ajout de nouveaux attributs.**

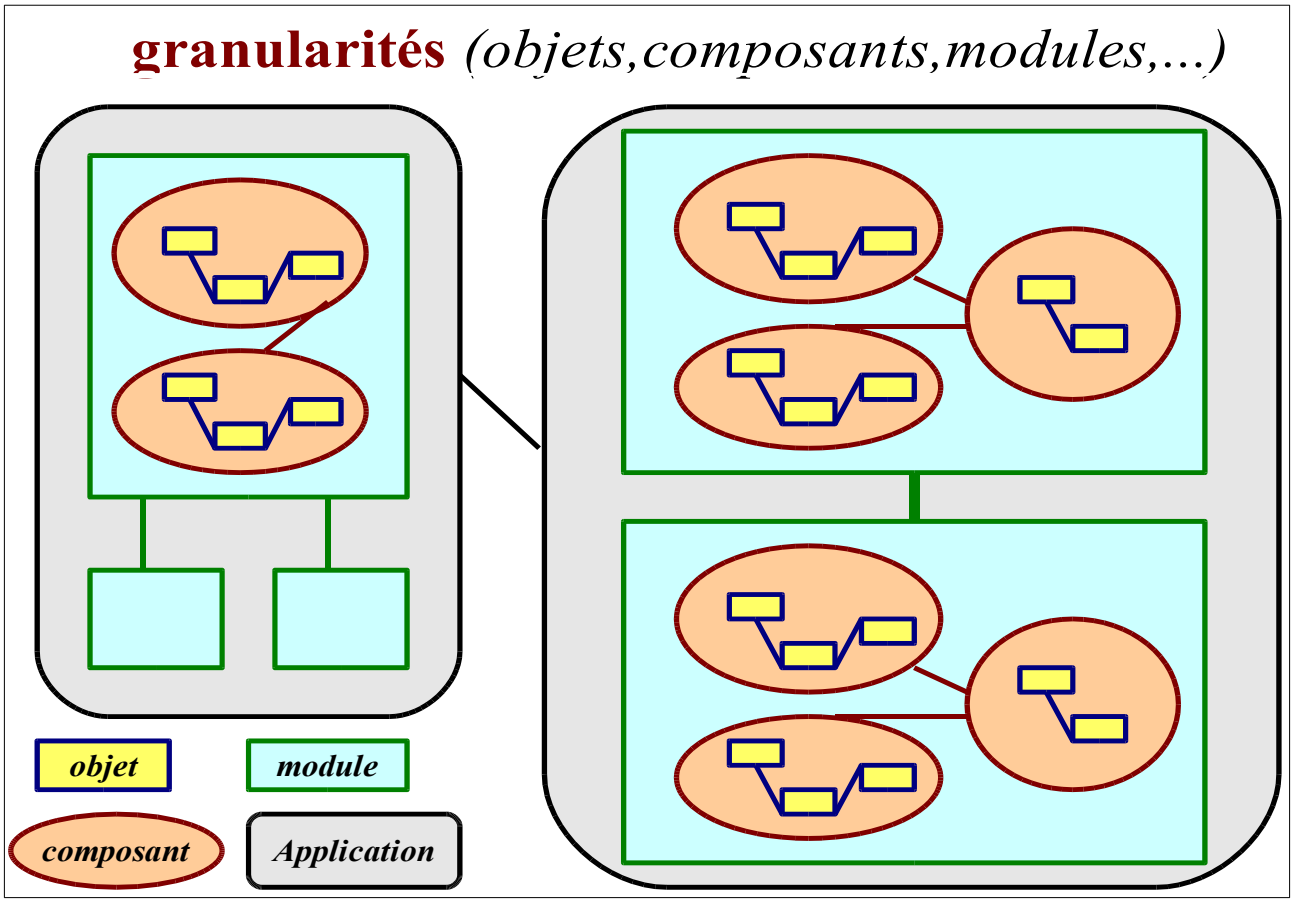

Interactions entre objets (exemples)
objet_partie_web
critere1
critere2
listeClients
lancerRecherche()
ServiceClients
connexionDatabase
creerClient()
rechercherClients()
DataSource
urlDataBase
userName, password
getConnection()
Code type à ce niveau
this.listeClients=
serviceClients.rechercherClients(
critere1,critere2) ;
+
affichage dans page web/html
Code type à ce niveau
this.connexionDatabase=
dataSource.getConnection() ;
+
requête SQL récupérant une liste clients
Où le mot clef "this" désigne l'objet courant et où autreObjet.operationXY()
correspond à un appel de méthode/opération sur un autre objet .

3. Granularité

Encapsulation & granularités



Modules , composants , objets



4. Modularité

4.1. Forte cohésion et couplage faible

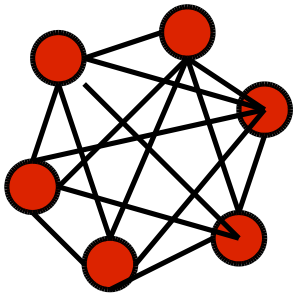
La **cohésion** mesure le degré de connectivité existant entre les éléments d'une classe unique. Une classe modélisant un aéroport aura une bien meilleure cohésion qu'une classe modélisant le roi d'une nation, le roi du jeu d'échec et le roi d'un jeu de carte . ***Il faut privilégier la cohésion fonctionnelle et éviter les cohésions faites de coïncidences***.

La question "Quels sont les points **invariants** au niveau de cette classe" est un bon critère permettant d'obtenir une bonne cohésion.

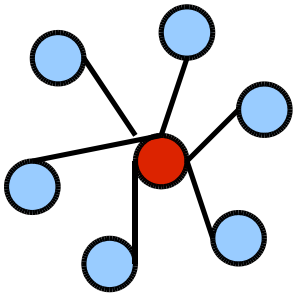
Couplage entre objets (idéalement faible)

Un ***objet élémentaire (tout seul)*** rend souvent des ***services assez limités***.

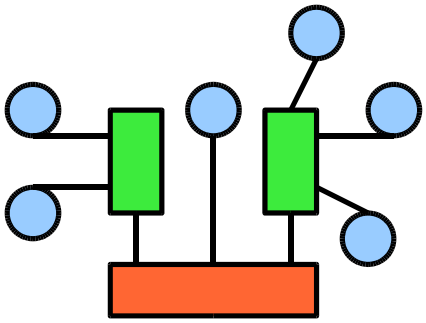
Un ***assemblage d'objets complémentaires*** rend globalement des ***services plus sophistiqués***. Cependant la complexité des liens entre les éléments peut éventuellement mener à un édifice précaire:



couplage trop fort
==> *complexe* ,
trop d'inter-relations
et de dépendances
==> *inextricable*.

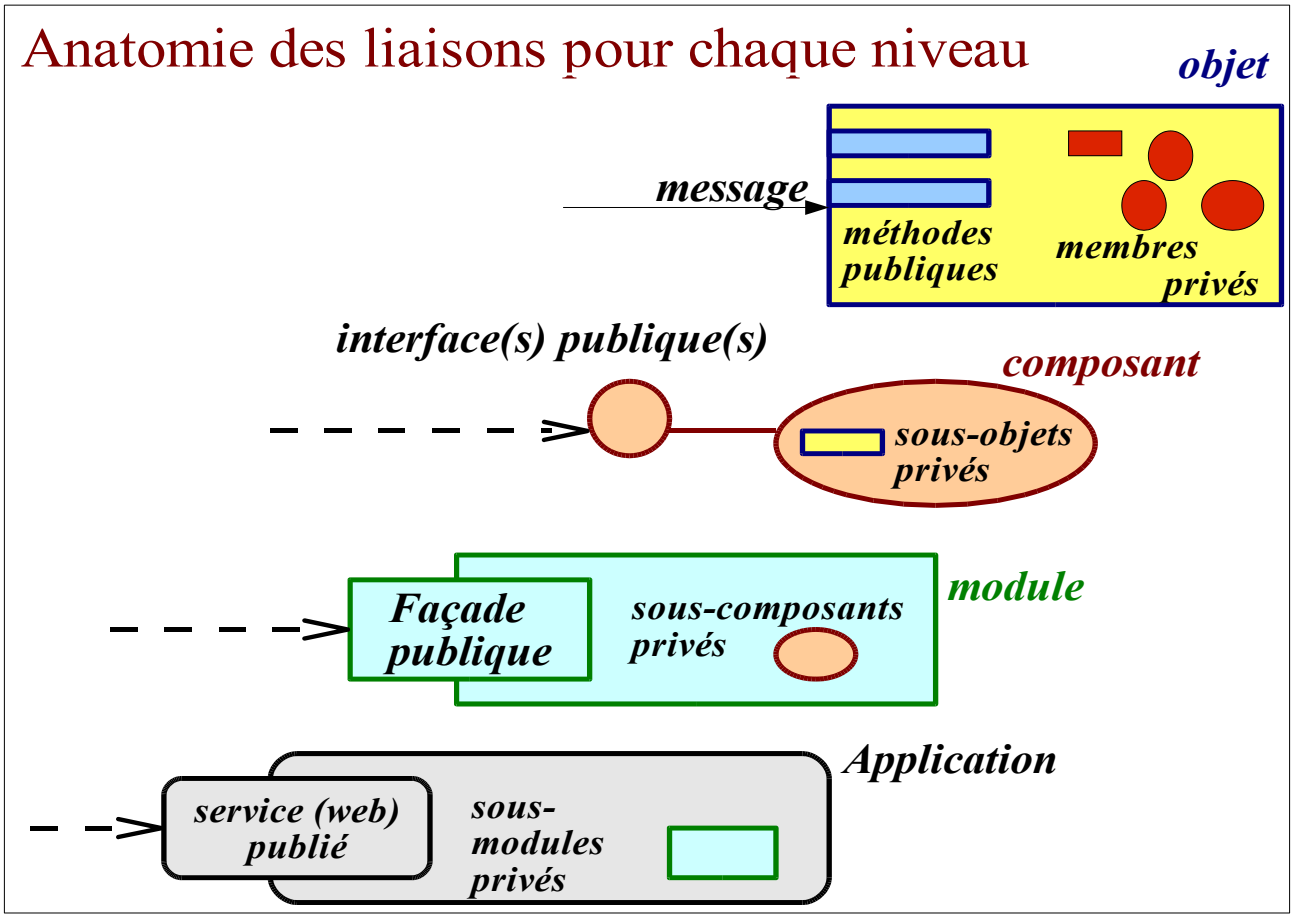


couplage faible
mais ***centralisé***
==> peu flexible
et point central
névralgique



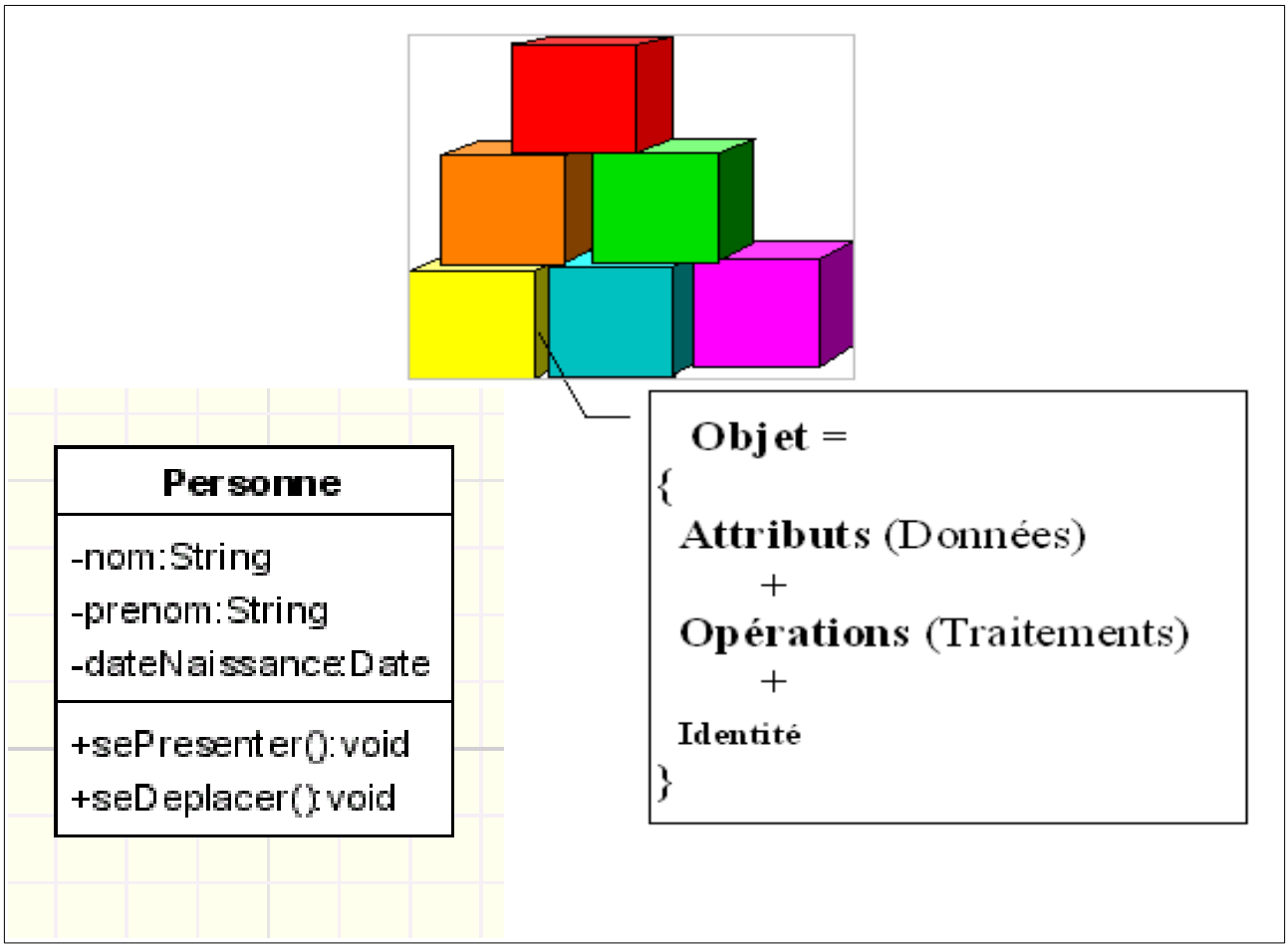
couplage faible
et ***décentralisé***
==> simple ,
relativement flexible
et plus robuste

==> pour approfondir --> "Patterns GRASP / répartition des responsabilités"
et "Principes de conception orientée objet"


Anatomie des liaisons pour chaque niveau
objet
message
méthodes publiques
membres privés
interface(s) publique(s)
composant
sous-objets privés
Façade publique
sous-composants privés
module
Application
service (web) publié
sous-modules privés

II - Essentiel de la programmation orientée objet

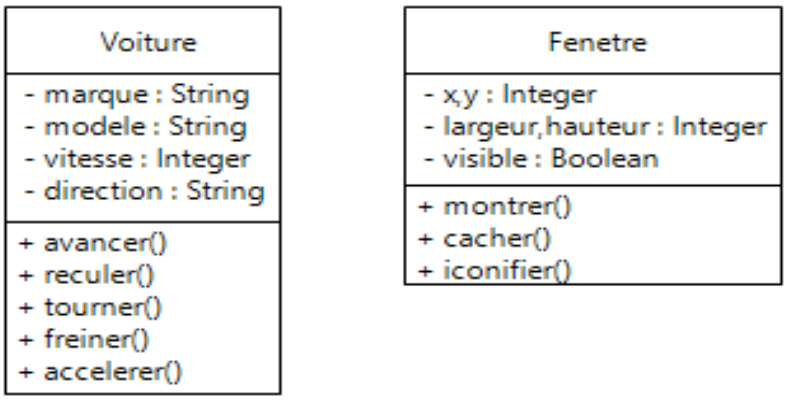
1. Objets, classes et instances



Notion de classe

Pour **regrouper les objets aux caractéristiques et comportements identiques**, on fait appel à la notion de **classe**.
Une **classe** peut être vue comme un **type** *(abstrait ou concret)* **d'objets**.

Une classe est une sorte de moule à partir duquel seront générés les objets que l'on appelle **instances** de la classe. *[Un objet est créé à l'image de sa classe]*

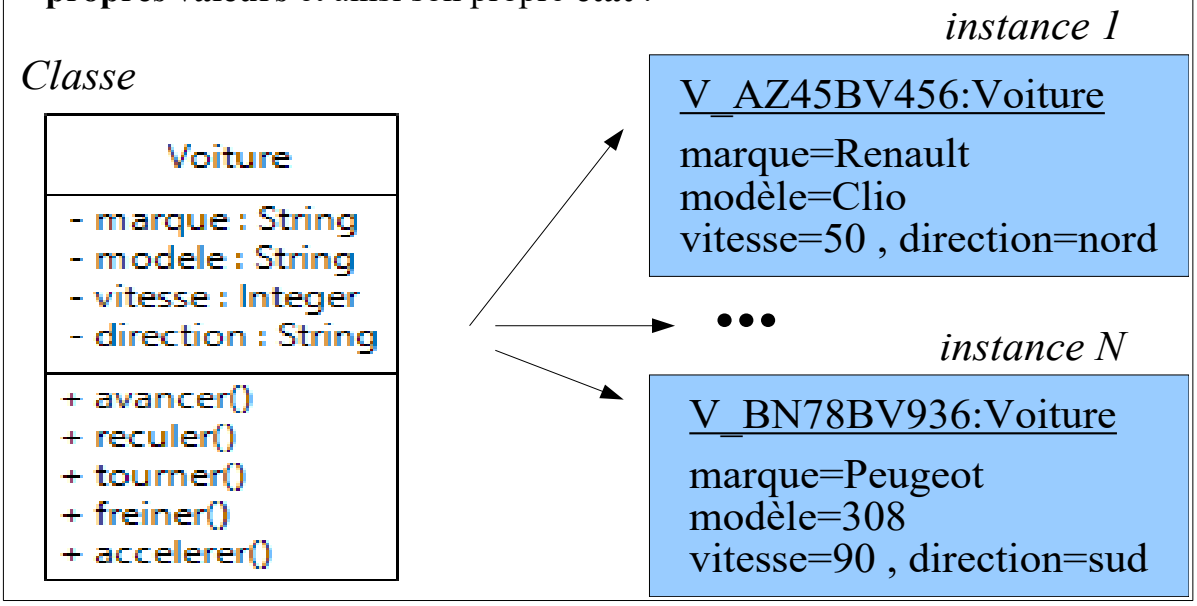


Toutes les instances d'une même classe ont une même structure commune et ont un même comportement.

Instances

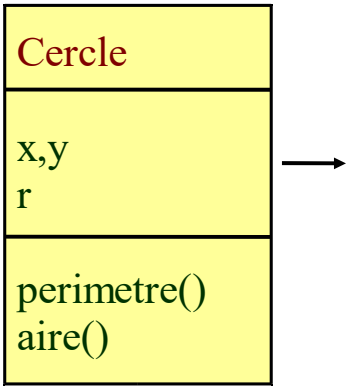
Rappel: un **exemplaire** (objet) issu à partir d'une certaine classe est appelé une **instance** .

Les différentes instances d'une même classe se distinguent par les **valeurs** *(assez souvent différentes)* **de leurs attributs** ==> **Chaque instance a ses propres valeurs** et ainsi son propre **état** .



Classes & Instances

Modélisation UML



Classe java (Cercle.java ➔ Cercle.class)

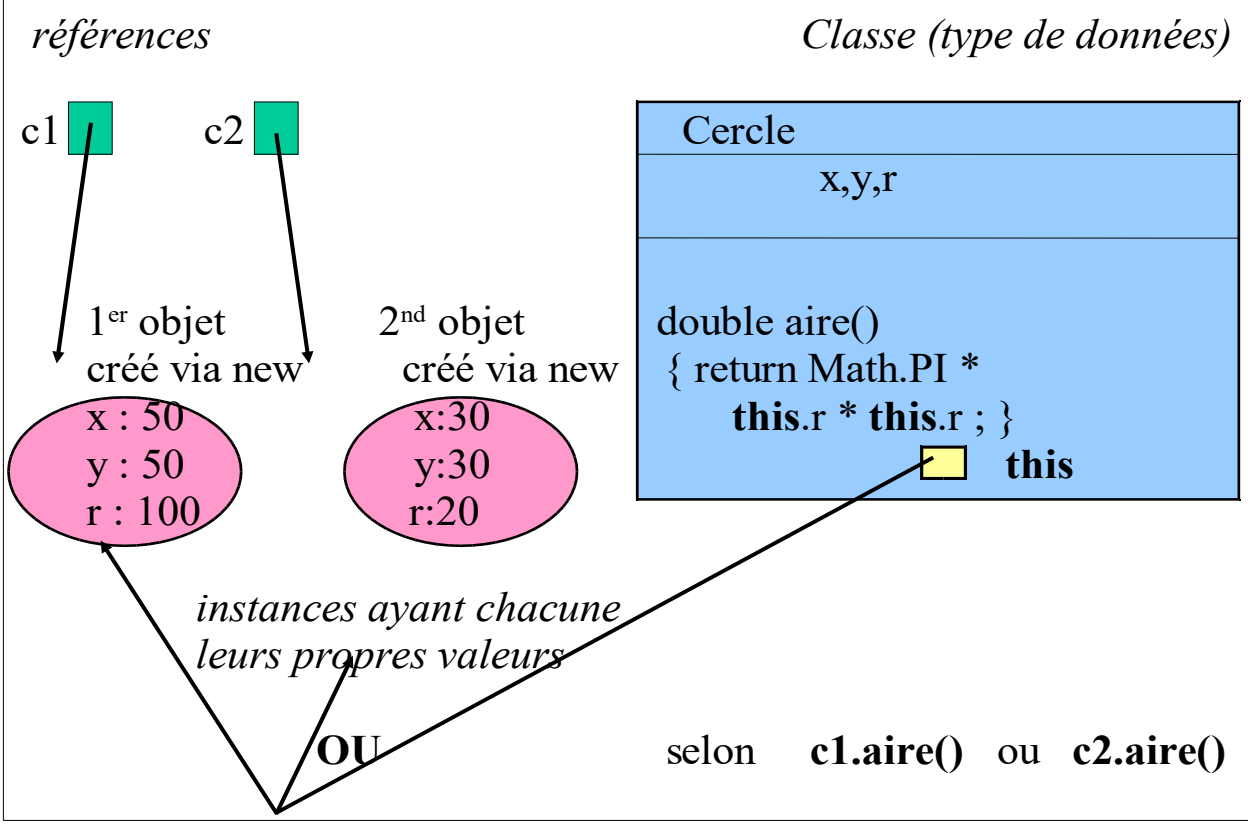
```
public class Cercle {
// Attributs (données):
    private double x,y; // coordonnées du centre
    private double r; // rayon
// Méthodes:
    public double perimetre() { return 2 * Math.PI * r; }
    public double aire() { return Math.PI * r *r ; }
}
```

Pour utiliser cette classe, il nous faut maintenant :

- déclarer une variable de type "référence sur un objet de classe Cercle"
- créer un objet ou une instance (exemplaire) de la classe via l'opérateur new.

Cercle c1; *//c1 est içi une référence sur un objet de type Cercle qui n'existe pas encore.*
c1 = new Cercle(); *//c1 référence maintenant un nouvel exemplaire de la classe Cercle.*

Classes & Instances



Exercices

Schématiser la structure des classes suivantes (en précisant les principaux attributs et les principales opérations):

- ***Personne*** (d'un point de vue *administratif*)
- ***Personne ou Patient*** (d'un point de vue *médical*)
- ***Adresse***
- ***Date***
- ***Télévision***

Exemple de code orienté objet en langage typescript

Version 1 (simplifiée) ne fonctionnant qu'avec ***"strictPropertyInitialization": false***
ou bien ***avec le suffixe! pas très conseillé sur tous les attributs/propriétés de la classe.***

```
class Compte{
        numero : number = 0;        //ou bien numero! : number;
        label : string = " ?";      //ou bien label! : string ;
        solde : number = 0.0;       //ou bien solde! : number;

        debiter(montant : number) : void {
                this.solde -= montant; // this.solde = this.solde - montant;
        }

        crediter(montant : number) : void {
                this.solde += montant; // this.solde = this.solde + montant;
        }
}
```

```
var c1 = new Compte(); //instance (exemplaire) 1
console.log("numero et label de c1: " + c1.numero + " " + c1.label);
console.log("solde de c1: " + c1.solde);

var c2 = new Compte(); //instance (exemplaire) 2

c2.solde = 100.0;
c2.crediter(50.0);

console.log("solde de c2: " + c2.solde);  //150.0
```

2. Constructeur

Un constructeur est une méthode qui sert à initialiser les valeurs internes d'une instance dès sa construction (dès l'appel à new) .

En langage typescript le constructeur se programme comme la méthode spéciale "**constructor**" (mot clef du langage) :

```
class Compte{
        numero : number;
        label: string;
        solde : number;

        constructor(numero:number, libelle:string, soldeInitial:number){
                this.numero = numero;
                this.label = libelle;
                this.solde = soldeInitial;
        }
}
```

```
var c1 = new Compte(1,"compte 1",100.0);
c1.crediter(50.0);
console.log("solde de c1: " + c1.solde);
```

NB: il n'est pas possible d'écrire plusieurs versions du constructeur :

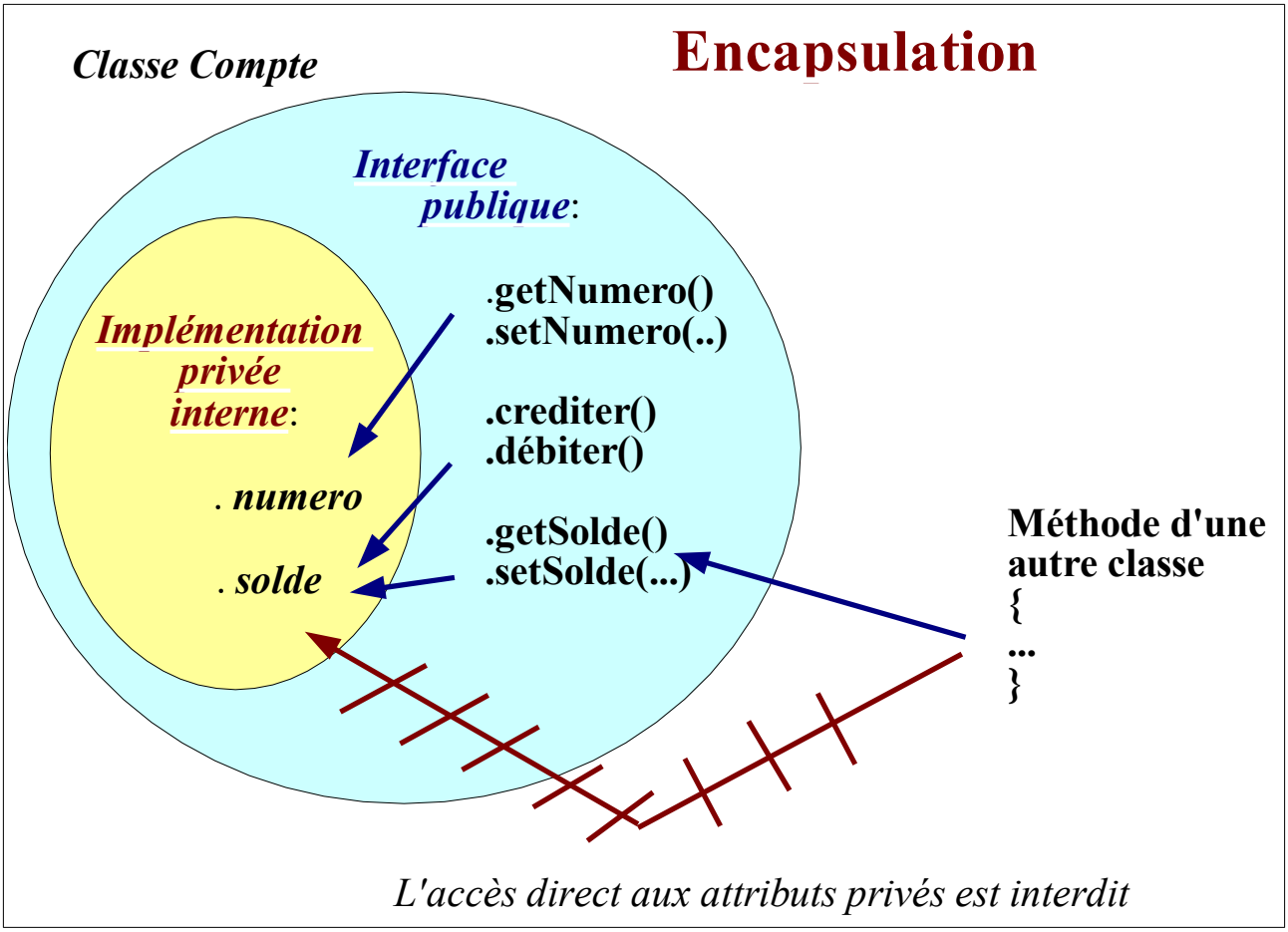
```
constructor(numero:number, libelle:string, soldeInitial:number){
        this.numero = numero;
        this.label = libelle;
        this.solde = soldeInitial;
    }

constructor(){
        this.=0;
        this.label="?";
        this.=0.0;
    }
```

Il faut donc quasi systématiquement utiliser la syntaxe **= *valeur_par_defaut*** sur les arguments d'un constructeur pour pouvoir créer une nouvelle instance en précisant plus ou moins d'informations lors de la construction :

```
class Compte{
    numero : number;
    label: string;
    solde : number;

    constructor(numero:number=0, libelle:string="?", soldeInitial:number=0.0){
        this.numero = numero;
        this.label = libelle;
        this.solde = soldeInitial;
    }//...
}
```

```
var c1 = new Compte(1,"compte 1",100.0);
var c2 = new Compte(2,"compte 2");
var c3 = new Compte(3);
var c4 = new Compte();
```

3. Encapsulation



Apports (intérêts) de l'encapsulation

* Les données internes étant propres à l'objet, on dit qu'elles sont privées (masquées): elles ne sont manipulables que par l'objet lui même.

* Un objet n'est sollicitable que par l'interface qu'il offre, c'est à dire via les opérations accessibles qui lui sont applicables.

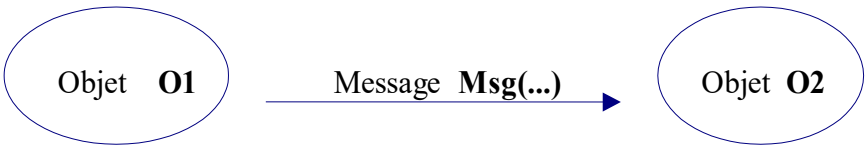
* La réalisation de ces opérations et la représentation physique de l'état de l'objet restent cachées et inaccessibles au monde extérieur.
  Du code externe à la classe ne peut pas accidentellement changer la valeur d'un attribut et mettre à mal la cohérence des valeurs internes d'un objet.
  Ceci assure une très bonne **protection des données**.

* L'encapsulation favorise la modularité et l'**évolutivité** des logiciels:

* La vue externe d'un objet ne faisant intervenir que l'interface publique, on a la possibilité de faire évoluer l'implémentation interne de celui-ci sans remettre en cause l'utilisation qui en est faite.

4. Communication par messages

Communication par messages



La partie dynamique des objets est assurée par la notion d'envoi de message.
Envoyer un message à un objet c'est lui dire ce qu'il doit faire.

Lorsqu'un objet reçoit un message, il active la méthode de même nom définit au niveau de sa classe.

Lorsque O1 envoi le message Msg à O2, il ne fait que demander à O2 d'effectuer l'opération qui est en relation avec le message .
message = requête.

Tout message comporte trois éléments:
* le destinataire (objet receveur)
* le sélecteur (Nom du message ou de la méthode)
* une liste d'arguments éventuellement vide

Quelques catégories classiques de méthodes:

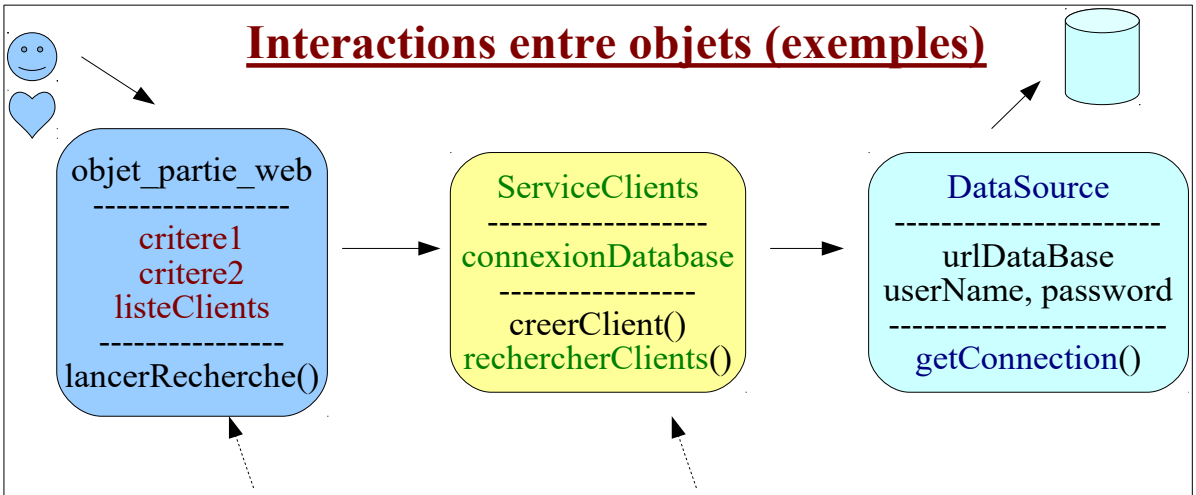
Accesseur : méthode publique de type ***getXxx***() (ex: *getNom*() , *getAge*() , ...) qui **retourne la valeur** (en copie) d'**une certaine donnée d'un objet**.

Modificateur / mutateur : méthode publique de type ***setXxx***() (ex: *setNom("dupond"), setAge(35),* ...) qui permet de **demander à l'objet de modifier (mettre à jour) une de ses valeurs**.

Constructeur : **méthode servant à initialiser certaines valeurs internes d'un objet dès le moment de sa création**.

Destructeur/Finaliseur: méthode déclenchée automatiquement juste avant qu'un objet soit détruit par les mécanismes internes d'un langage (ex: C++, Java, ...)

Interactions entre objets (exemples)



Code type à ce niveau

```
this.listeClients=
serviceClients.rechercherClients(
                    critere1,critere2) ;
            +
affichage dans page web/html
```

Code type à ce niveau

```
this.connexionDatabase=
dataSource.getConnection() ;
            +
requête SQL récupérant une liste
clients
```

Où le mot clef "*this*" désigne l'*objet courant* et où ***autreObjet.operationXY()*** correspond à ***un appel de méthode/opération sur un autre objet*** .

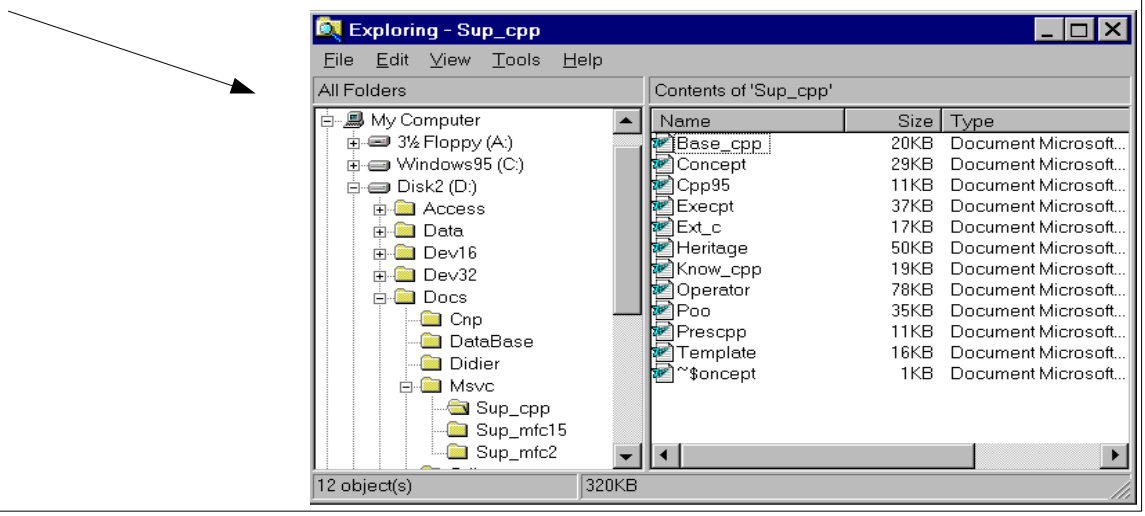
Exercices

* Quel sont les problèmes dans les écritures suivantes ?

objCercle.rayon = objCercle.rayon – 50;

objPersonne.adresse.rue = "avenue Y";

* Quels composants et messages échangés ?



5. private et get/set

Les exemples ci-dessous sont en langage "typescript" :

5.1. Propriété "private"

```
class Animal {
   private _size : number;
   name:string;
   constructor(theName: string = "default animal name") {
            this.name = theName;
            this._size = 100; //by default
          }
   move(meters: number = 0) {
      console.log(this.name + " moved " + meters + "m." + " size=" + this._size);
   }
}
```

var a1 = new Animal("favorite animal");

~~a1._size=120;~~ *//erreur détectée '_size' est privée et seulement accessible depuis classe 'Animal'.*

a1.move();

5.2. Accesseurs automatiques get xxx() /set xxx()

```
class Animal {
   private _size : number;
  public get size() : number{ return this._size;
         }
  public set size(newSize : number){
            if(newSize >=0)   this._size = newSize;
                  else console.log("negative size is invalid");
   }
  …}  //NB : le mot clef public est facultatif devant  "get" et "set" (public par défaut)
```

var a1 = new Animal("favorite animal");

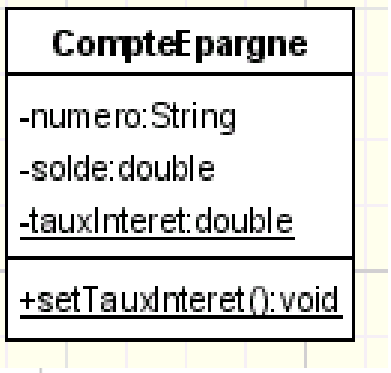
a1.size = -5; *// **calling set size()** → negative size is invalid (at runtime) , _size still at 100*

a1.size = 120; *// **calling set size()***

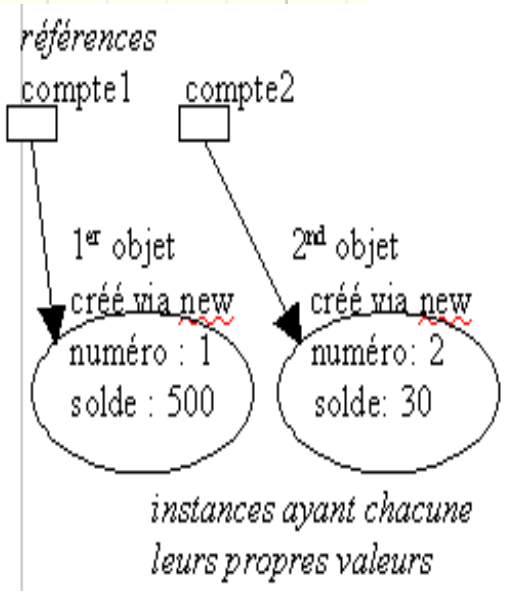
console.log("size=" + **a1.size**) ; *// **calling get size()** → **affiche size=120***

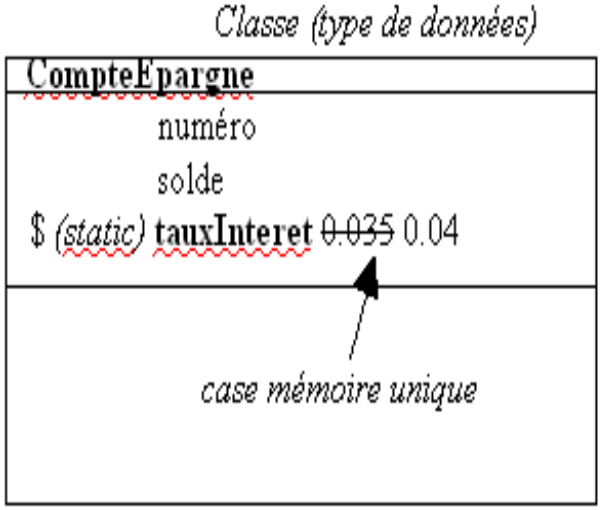
6. Variables et méthodes de classe (static)

Variables de classe (*static*)




```
public class CompteEpargne {
private String numero;
private double solde;
private static double tauxInteret = 0.035;
```





Variables et méthodes de classe (static)

compte1.setNuméro(1);
~~compte1.setTauxInteret(0.04);~~ *// possible mais très déconseillé (ambigu).*
CompteEpargne.setTauInteret(0.04); *// c'est l'idéal*

Le mot clé **static** permet également de définir une **méthode de classe** .
Une telle **méthode statique** a la particularité de **pouvoir être invoquée depuis un nom de classe** et non pas seulement depuis une instance particulière de la classe.

*// Appel de la méthode static **sqrt**() de classe "java.lang.**Math**"*
double distance = **Math.sqrt**(x*x+y*y);

Exercices

* Si getInstance() est une méthode de classe (static), quelle est alors la syntaxe d'invocation la plus appropriée ?

```
Cx objX = new Cx();  i = objX.getInstance();


i = Cx.getInstance();
```

* Soit Math une classe prédéfinie du langage Java.
  Que suggèrent alors les expressions suivantes:

```
périmètre  = 2* Math.PI * rayon  ;


hypoténuse = Math.sqrt(x*x + y*y);
```

Surcharge:

Certains langages objets (C++, JAVA,…) permettent de définir **plusieurs fonctions de mêmes noms** qui **se distinguent par le nombre et/ou le type des arguments**.
On parle alors de ***surcharge de fonction/méthodes.***
NB: Ceci n'est pas un concept mais un simple détail technique

exemple:

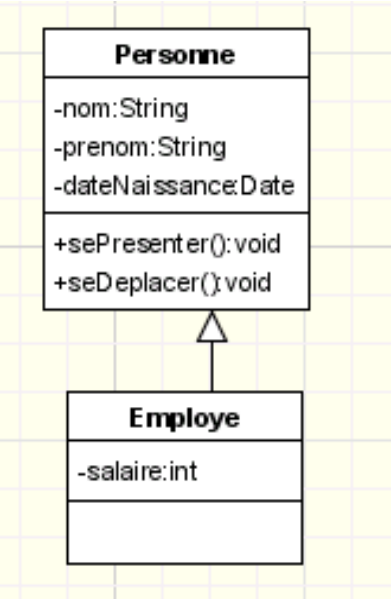
```
Rechercher(int position)
Rechercher(string nom)
…  // NB: Ces 2 versions peuvent éventuellement être codées
      au niveau d'une même classe.
```

On appelle **signature** (ou ***prototype***) l'ensemble des caractéristiques (nom de l'opération, types et nombre des paramètres) qui permettent d'identifier une **opération** (logique) ou bien une **fonction** (entité physique).

Tous les langages objets (même JAVA et le C++) considèrent ***qu'une opération générique ne peut avoir qu'une seule signature***.

7. Notion d'héritage

Héritage / Généralisation / Spécialisation

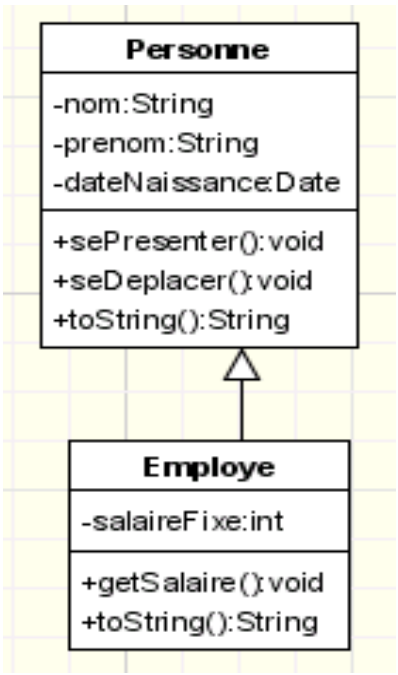


*** La notion d'héritage consiste à définir une nouvelle classe à partir d'une classe existante en spécialisant certaines choses.**

*** La classe dérivée (ou sous classe) reprendra tous les attributs et toutes les opérations de la surclasse .**

*** La structure de données est conservée au niveau des sous classes; elle peut néanmoins être enrichie via l'ajout de nouveaux attributs.**

Héritage – code Java



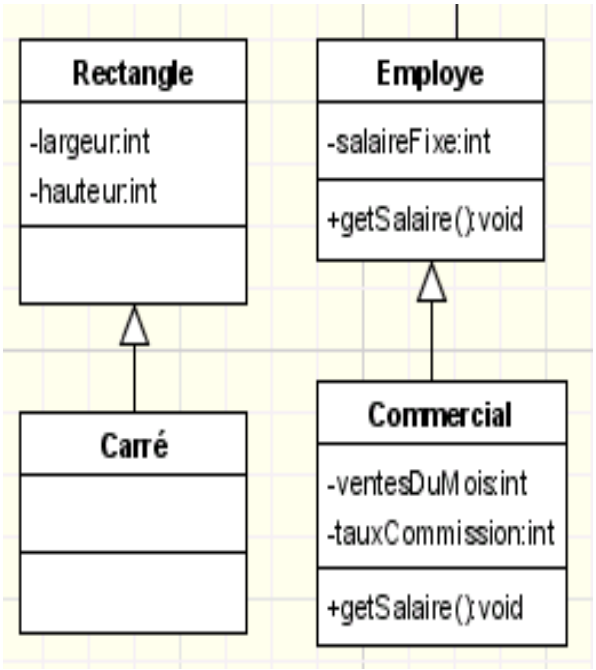

```
public class Personne {
    private String nom;
    private String prenom;
    private java.util.Date dateNaissance;

    public void sePresenter() {...}
    public void seDeplacer() {...}
    public String toString() { ... }
}
```

```
public class Employe extends Personne {
    private int salaireFixe;

    public void getSalaire() {...}
    public String toString() {...}
}
```

pers1.sePresenter() ; emp1.sePresenter(); emp1.getSalaire();

8. Redéfinitions

Eventuelle **rédéfinition** d'une méthode au sein d'une sous classe

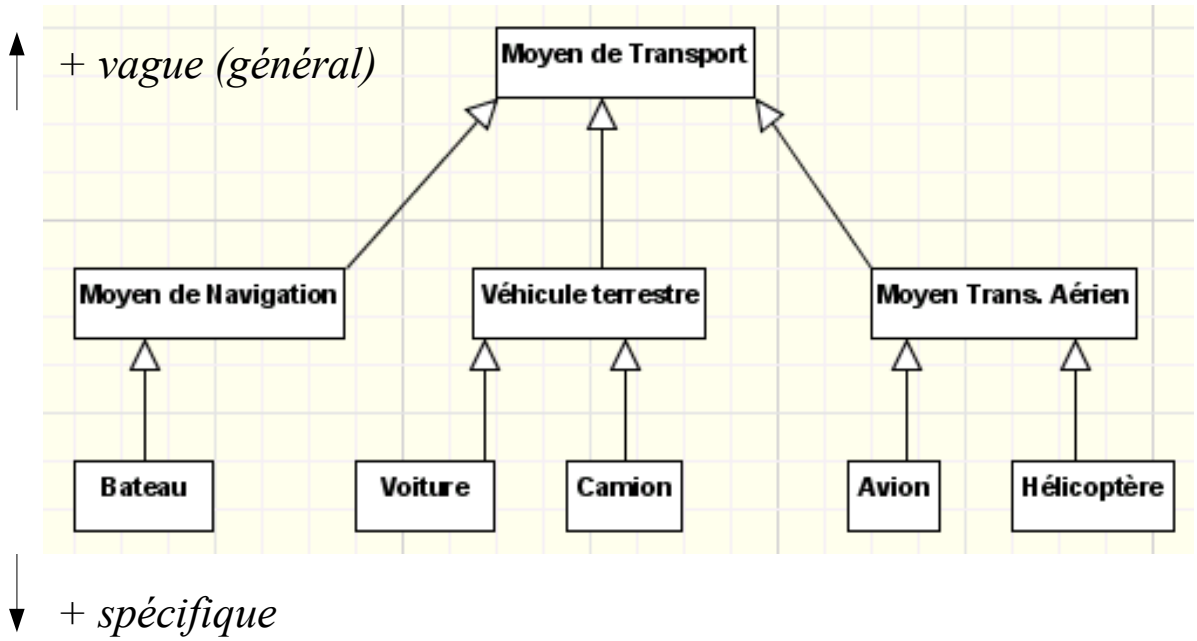


La classe dérivée peut très bien **redéfinir** le comportement de certaines opérations héritées .

Par exemple si la classe Carré hérite de Rectangle , elle peut redéfinir certaines opérations pour faire en sorte que largeur = longueur = coté .

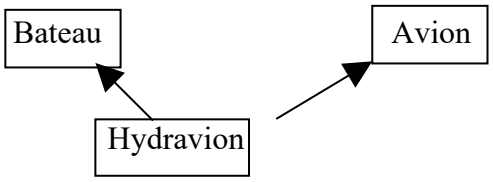
9. Arbre d'héritage

Arbre (ou graphe) d'héritage



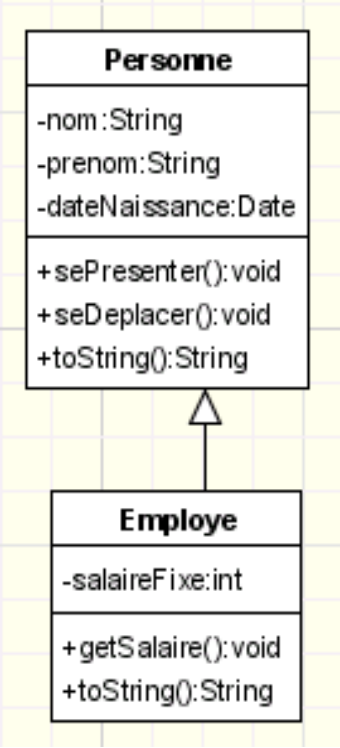
Une relation d'héritage est représentée par une grosse flèche triangulaire allant de la sous-classe vers la sur-classe et signifiant
"**is kind of** / **est une sorte de**"

Héritage multiple = cas où une classe hérite de plusieurs classes (ex: Hydravion héritant de Bateau et de Avion) .



NB: l'héritage multiple n'est supporté que par certains langages de programmation (ex: en C++ mais pas en Java).

Héritage et lien entre types de données



En héritant de *Personne* la classe *Employé* est alors implicitement considérée comme un *type de donné* spécifique mains néanmoins *compatible avec le type* générique *Personne*.

Un employé est un cas particulier de personne.

Personne p = *null;*
Employe emp = *new Employe("dupond" , ...);*

p = emp;

p.sePresenter(); ...

10. Polymorphisme

Polymorphisme fonctionnel (exemple)

porte_a_1_battant.ouvrir()

porteCoulissante1.ouvrir()

*littéralement "**plusieurs formes**" (*)*

Porte (*abstraite*)
---
ouvrir()
fermer()

PorteCoulissante
---
glissière

-glisser()
+ouvrir()
+fermer()

Porte_a_1_battant
---
battant

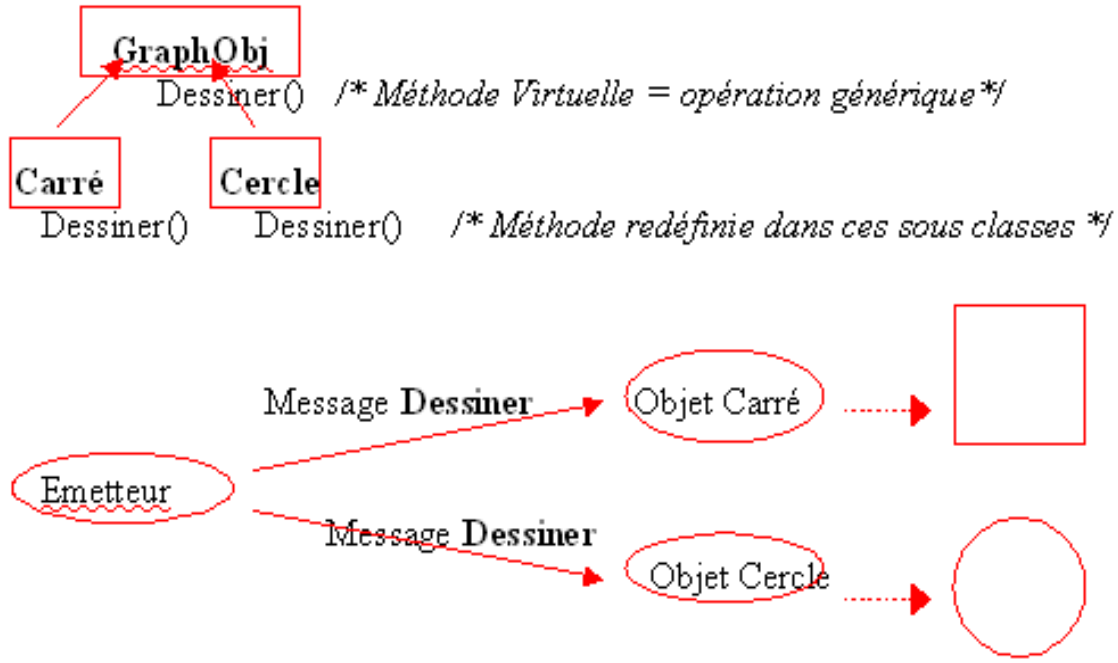
-pivoter
+ouvrir()
+fermer()

- *Même fonctionnalité abstraite "ouvrir".*
- *Même déclenchement.*
- *Même message envoyé "porte, ouvre toi !"*

(*) Les façons exactes d'ouvrir la porte diffèrent au niveau des types de portes qui reçoivent et traitent le message "demande d'ouverture".

Poly*morphisme* signifie littéralement "**plusieurs *formes***" .
Il s'agit ici des différentes formes que peut prendre l'action entreprise par un objet lorsqu'il reçoit un message. L'action (ou méthode) déclenchée dépendra du type (ou classe) précis(e) de l'objet qui recevra le message générique.

Le **polymorphisme** signifie qu'une même **opération** peut avoir des comportements différents suivant les classes.



Pratiquement:

L'utilisation du polymorphisme revient souvent à :

- ***Eliminer une batterie de test*** (if...else / switch...case).
- ***Déléguer l'action en demandant à l'objet de s'en charger***.
- ***Augmenter la modularité de l'ensemble***.
- ***Favoriser l'évolution*** (Ajout nouvelle classe au lieu de modifier tous les tests).

Code classique (via tableau ou collection de références):

```
TableauRefObjetsGraphiques[0] = carré1;
...
TableauRefObjetsGraphiques[3] = cercleB;
…
for(i=0;i<4;i++)
    TableauRefObjetsGraphiques[i].Dessiner();
```

Exemple de code d'héritage (avec polymorphisme) en typescipt :

```
class AnimalDomestique {
    name:string;
    constructor(name: string ="defaultAnimalName") { this.name= name;}
    decrire(){ console.log("AnimalDomestique nom=" + this.name );}
    parler(){  console.log("…") ; }
}
```

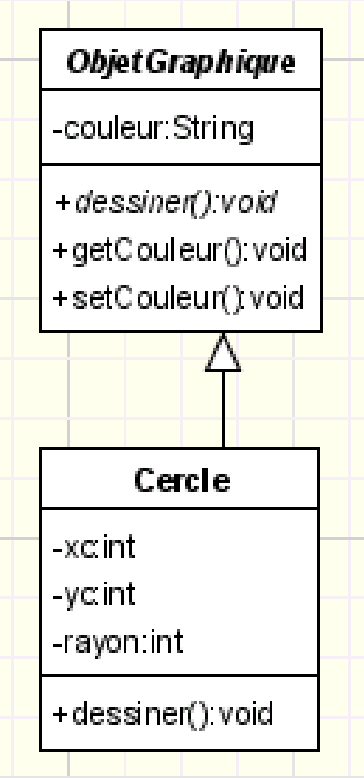
```
class Chat extends AnimalDomestique {
    nbHeuresSommeil /*:number */ = 14 ;
    constructor(name: string = "defaultCatName") { super(name); }
    decrire() {
        console.log("Je suis un chat qui dort " +this.nbHeuresSommeil + " h");
        super.decrire();
    }
    parler() { console.log("miaou miaou") ; }
    ronronner() { console.log("ronron...") ; }
}
```

```
class Chien extends AnimalDomestique {
    fonction : string | undefined ;
    constructor(name: string = "defaultDogName") { super(name); }
    decrire() {
        console.log("Je suis un chien , fonction=" + this.fonction );
        super.decrire();
    }
    parler() { console.log("whaouf whaouf") ; }
    monterLaGarde() { console.log("je monte la garde ...") ; }
}
```

```
var a = new AnimalDomestique(); //var a = new AnimalDomestique("animal");
a.decrire() ; //AnimalDomestique nom=defaultAnimalName

var chat1 = new Chat("malo");  //var chat1 = new Chat();
chat1.ronronner() ; //ronron
chat1.decrire() ; // Je suis un chat qui dort 14h AnimalDomestique nom=malo
chat1.parler(); // miaou miaou

var a2: AnimalDomestique = new Chien("Rantanplan");
a2.monterLaGarde() ;
if(a2 instanceof Chien)
     ( <Chien> a2 ).monterLaGarde();
a2.decrire(); //Je suis un chien fonction=undefined AnimalDomestique nom=Rantanplan
a2.parler() ; // whaouf whaouf  //polymorphisme  (Chien = sorte de Animal)
```

11. Classes abstraites

Classes abstraites



Une **classe abstraite** *(en italique en UML)* est une **classe intermédiaire dans l'arbre d'héritage qui ne sera pas directement instanciée**. Une telle classe sert simplement à **factoriser** certains attributs et certaines méthodes.

D'autre part, même s'il n'est pas possible d'instancier une classe abstraite***, on peut définir des pointeurs ou références génériques sur une classe abstraite. Ceci permet de référencer des objets de types différents*** pour ensuite invoquer des méthodes polymorphes.

La classe ObjetGraphique vue précédemment est un exemple de classe abstraite qui est concrétisée par ses sous-classes.

refObjGraph : ObjetGraphique | null =null;

refObjGraph = **new Cercle();** *// OK , + polymorphisme avec Rectangle , ...*

refObjGraph = **new ObjetGraphique();** *// interdit*

NB: Si un développeur définie une classe comme étant abstraite ,
un autre développeur est alors obliger de définir au moins un nouveau cas spécialisé (par héritage) afin de s'en servir utilement.

III - Présentation du formalisme UML

1. Méthodologie, formalisme , ...

Méthodologie = Formalisme + Processus

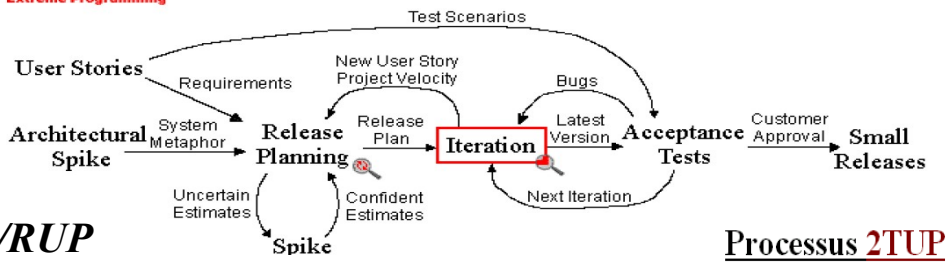
- **UML est un** formalisme
  (Notations standardisées [diagrammes] avec sémantiques précises)

- Un **processus** *(démarche méthodologique)* doit être utilisé conjointement (ex: UP , XP , ...).

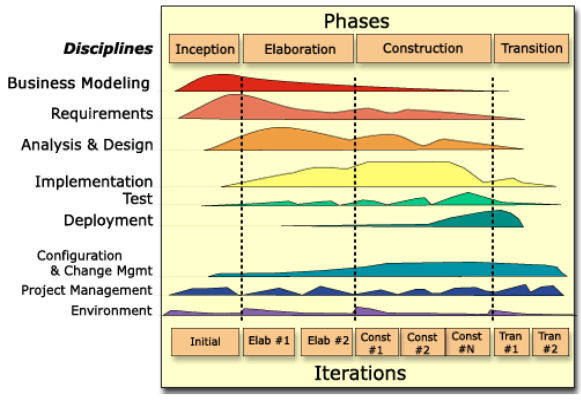
\+ *en pratique*: *les* ***Procédés*** (selon *outils / MDA/ ...*)

Processus méthodologiques (exemples):

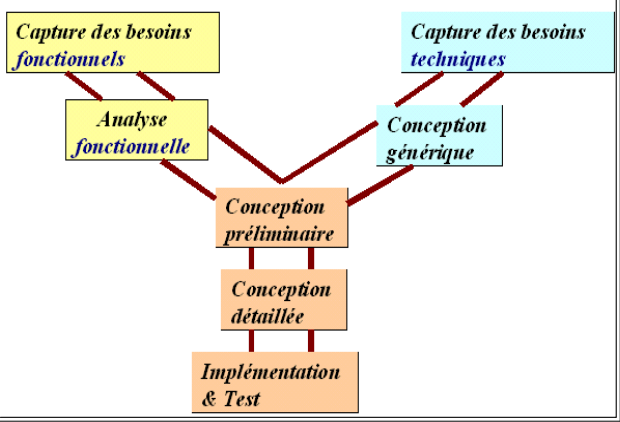
Extreme Programming Project



UP/RUP

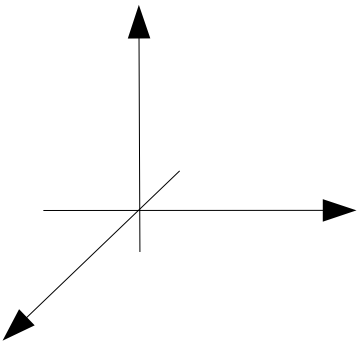


Processus 2TUP



Plein de variantes dans les façons d'utiliser UML

Formalisme
(notations, diagrammes)



Procédés
(outils UML, MDA,...)

Méthode/démarche
(activités de modélisation, spécifications,...)

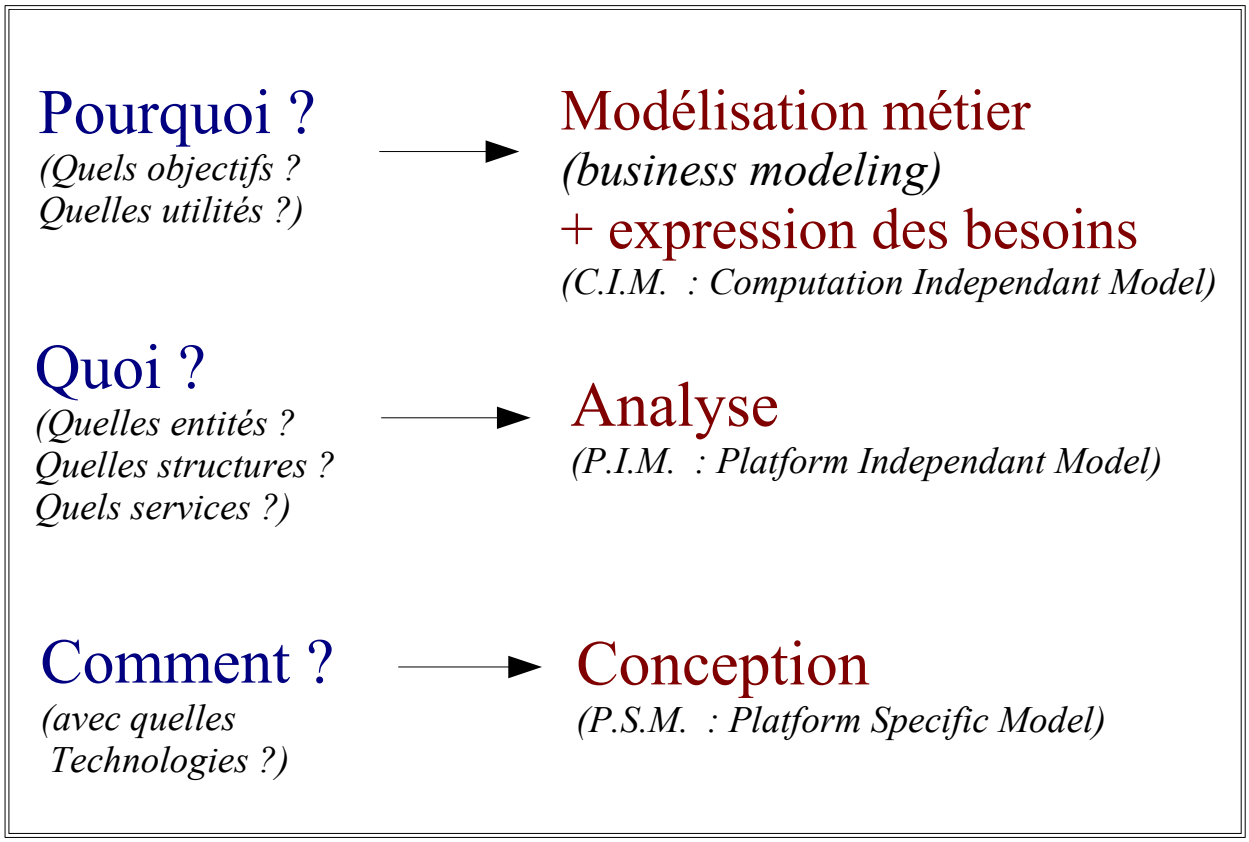
* UML pour
simple ébauche
ou bien
modèle précis ?

* *avec ou sans génération de code (MDA,...) ?*

* *Quelles spécifications ?*
Dans quel ordre ?
Avec quels diagrammes ?

Principales façons d'utiliser UML :

mode "esquisse" (minimum pour méthode agile)	• diagrammes informels et incomplets • Support de communication pour concevoir les parties critiques
mode "plan"	• Diagrammes formels relativement détaillés (avec annotations/commentaires en langage naturel) • Génération éventuelle d'un squelette de code à partir des diagrammes • Nécessité de compléter beaucoup le code pour obtenir un exécutable
mode "programmation/MDA"	• Spécification complète et formelle en UML • Génération automatique d'une grande partie du code d'une application à partir des diagrammes stéréotypés et de "templates de code à générer" • Nécessite des outils très spécifiques/pointus et beaucoup de paramétrages complexes(--> approche rarement rentable/envisageable)

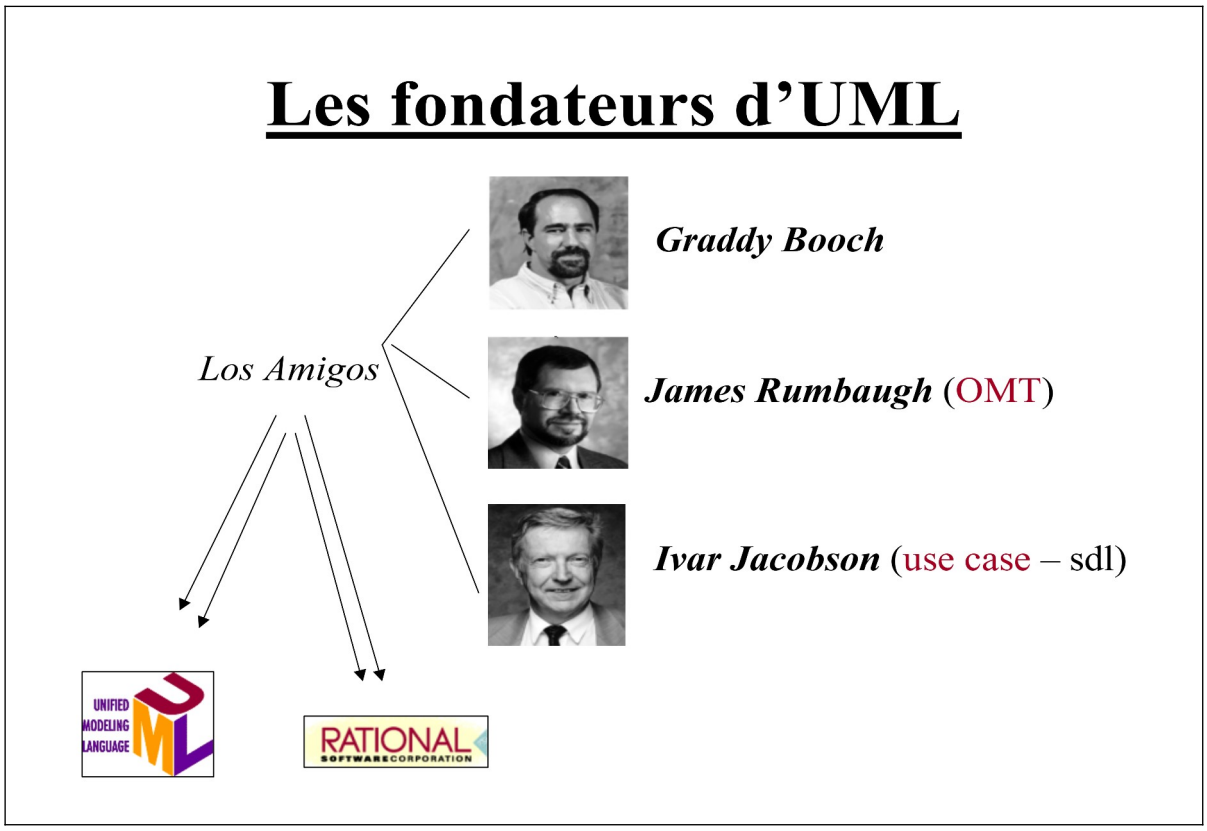


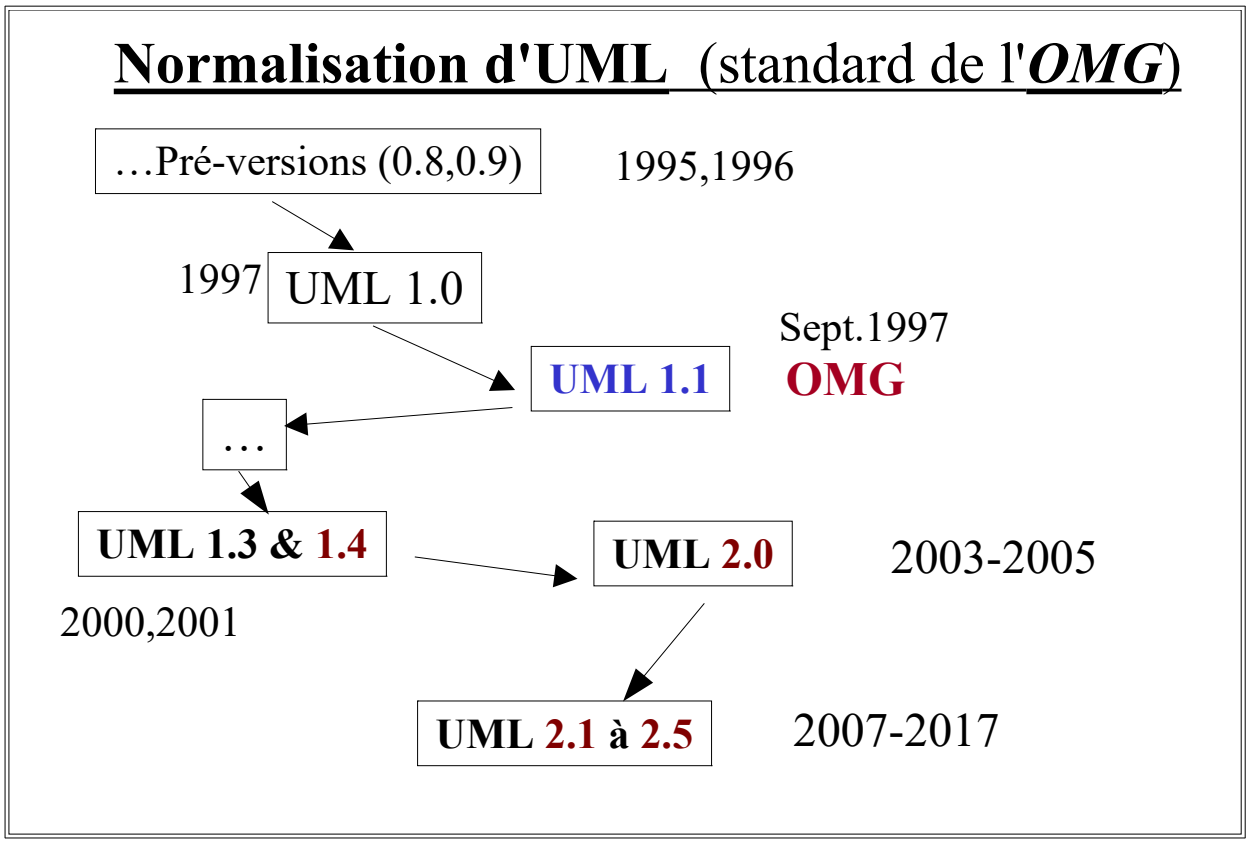
NB : UML peut aussi bien être utilisé pour définir des modèles

- **descriptifs** (décrire un existant : modèle métier ou architecture technique)
- **prescriptifs** (décrire le futur système à réaliser)

Un **modèle UML (orienté objet)** correspond toujours à un **certain point de vue** .
Plusieurs variantes sont généralement envisageables (rarement une seule solution).

2. Historique rapide d'UML





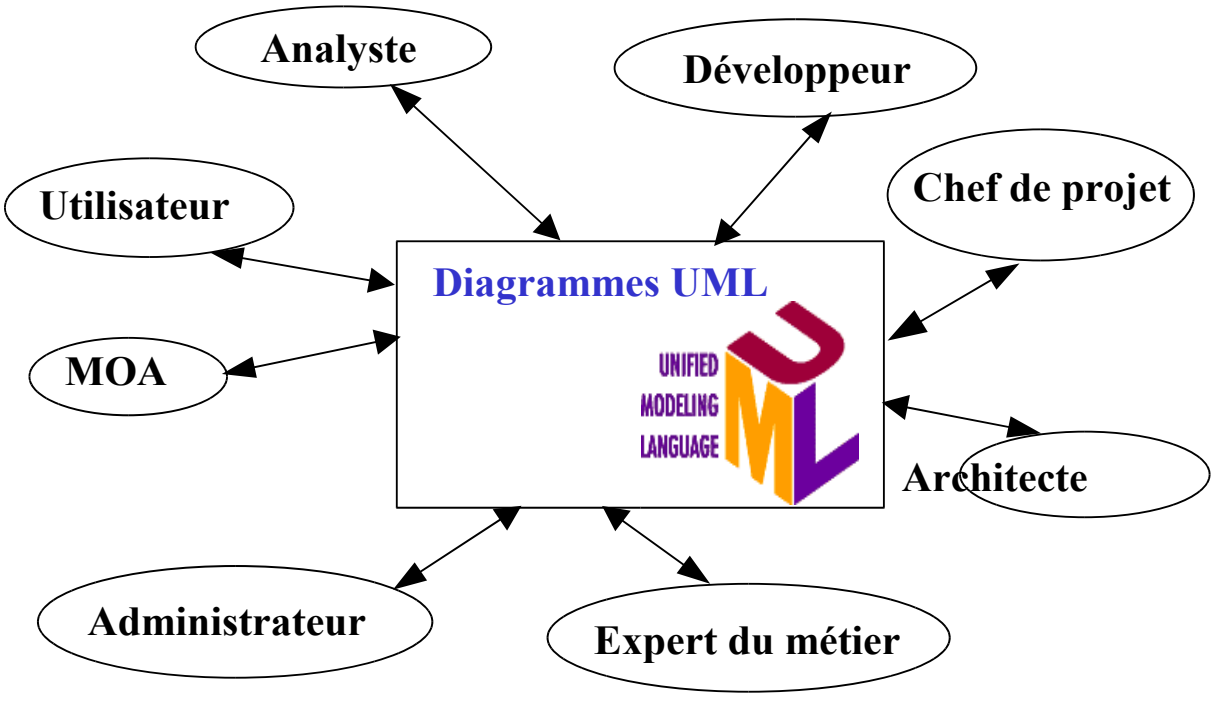
3. UML pour illustrer les spécifications

Principales **utilités d'UML**

- Diagrammes UML = partie importante des **Spécifications** fonctionnelles et techniques.
- **Cogiter** sur le "pourquoi/quoi/comment" en se basant se l'**essentiel** (qui ressort de la **modélisation** abstraite UML).
- Eventuel point d'entrée d'une **génération partielle de code** (via MDA ou ...).

4. Formalisme UML = langage commun

Différents points de vue / langage commun



5. UML universel mais avant tout orienté objet

UML ***universel*** mais ***avant tout orienté objet***

Méthode Merise

Modèle relationnel (MCD, ...)

Structure d'une base de données + ...

(époque mainframe et client/serveur)

Formalisme UML

Modèle objet *(et compléments) (classes, ...)*

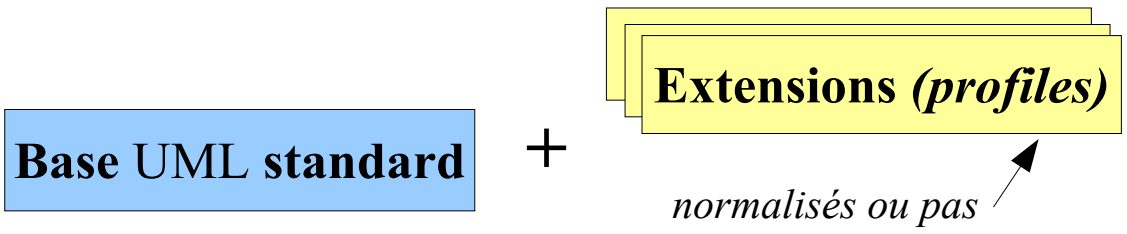
Structure d'une application orientée objet (c++,java, ...) *+indirectement structure d'une base de données (ou fichiers)*

(époque n-tiers et SOA)

6. Standard UML et extensions (profiles)

Quelques grands traits d'UML

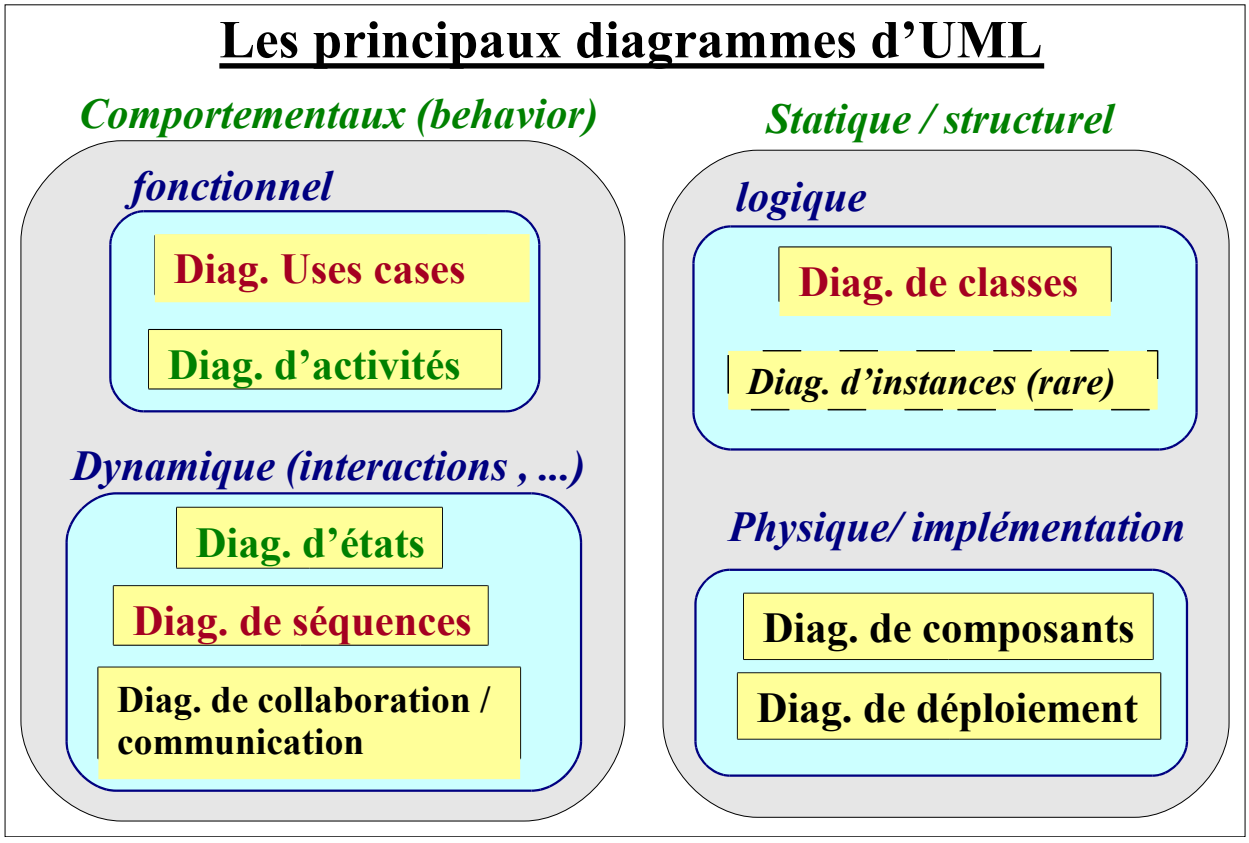
- ***Semi formel*** (plusieurs variantes possibles dans les diagrammes)
- Essentiellement basé sur les ***concepts objets*** *(pas du tout lié au modèle relationnel).*
- Se voulant être assez ***universel*** (java, c++, c# , ...) et utilisable aussi bien en informatique industrielle qu'en informatique de gestion.
- Pour communiquer , s'exprimer , cogiter , ...



7. Importance de la modélisation sur un projet complexe

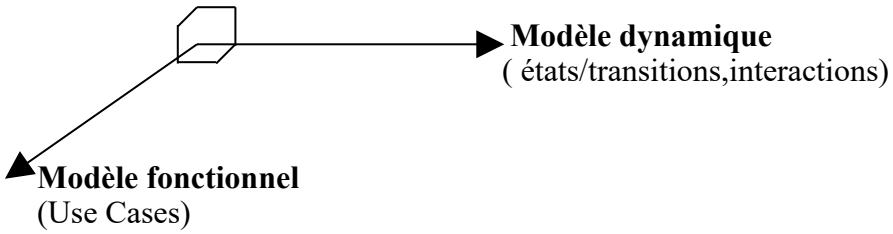
Plus un projet est grand et complexe , plus on a besoin :
- de point de repère pour s'y retrouver dans le code et la structure d'une application
- d'une bonne structuration "modulaire" pour arriver à une bonne décomposition (en bon travail d'équipe , bonne manière de gérer les tests de composants , ...)

8. Présentation des diagrammes UML



Modèles (diagrammes) complémentaires:

Modèle objet (classes, associations)



Les diagrammes secondaires d'UML 2

• "***Package diagram"*** = variante volontairement simplifiée du diagramme de classes où l'on ne montre que les "**packages**" et leurs inter-dépendances [*BONNE PRATIQUE*] .

• "***Composite Structure Diagram***" : diagramme complémentaire (annexe) vis à vis du diagramme de classes permettant de montrer la structure interne d'une classe (sous parties) --> **parts , connectors , ports , ...**

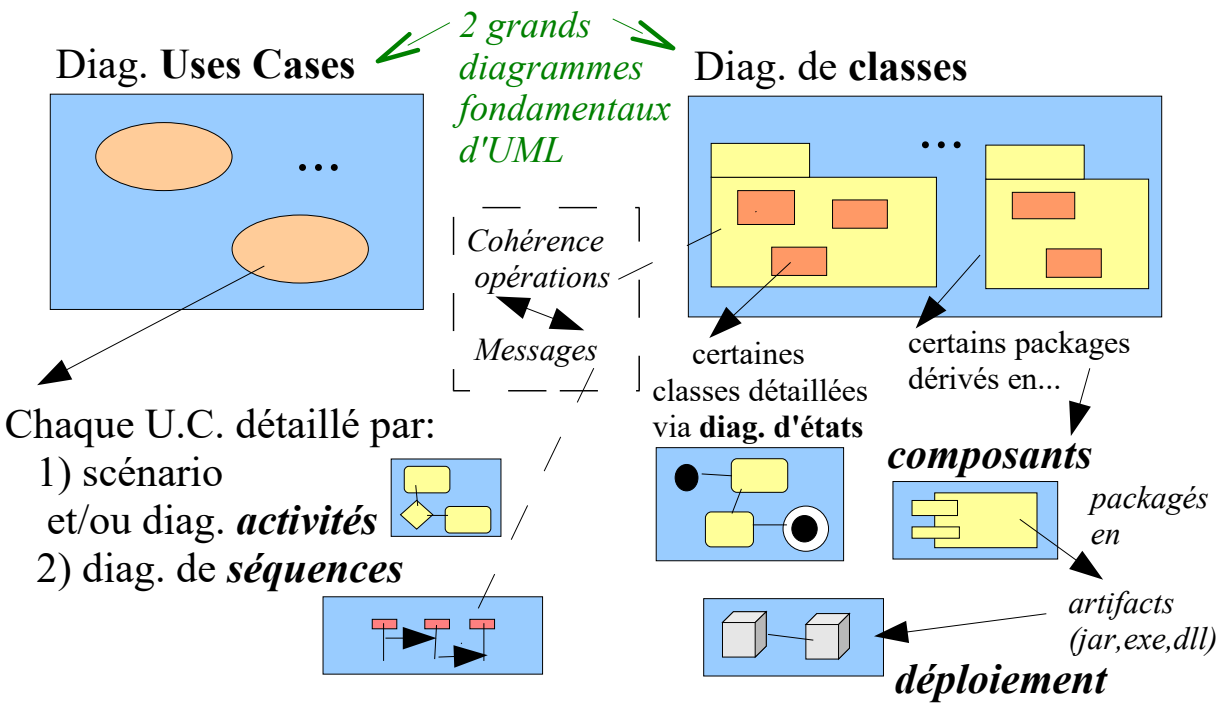
• "***interaction overview diagram***" : variante du diagramme d'activités où certaines activités sont modélisées par des cadres qui référencent (ou imbriquent) d'autre diagrammes d'interactions (ex : de séquences)

Diagrammes très secondaires d'UML 2

• "***Profiles diagram"*** = diagramme spécifique à UML 2 (***pour modéliser des extensions***="profile xyz pour UML") : on y spécifie de nouveaux "***stéréotypes***" (*ex : "<<real-time>>" , "<<stateless>>" , ...*) qui seront quelquefois analysés par des générateurs de code spécifiques .

• "***timing diagram***" = diagramme très technique (de type interactions) montrant des changements internes (avec échelle de valeurs sur axe vertical) suite à des stimulis/événements et conditionnés ou inscrits dans un cadre temporel (sur axe horizontal) => contraintes de temps, évolutions dans le temps.

Principaux liens entre les diagrammes UML



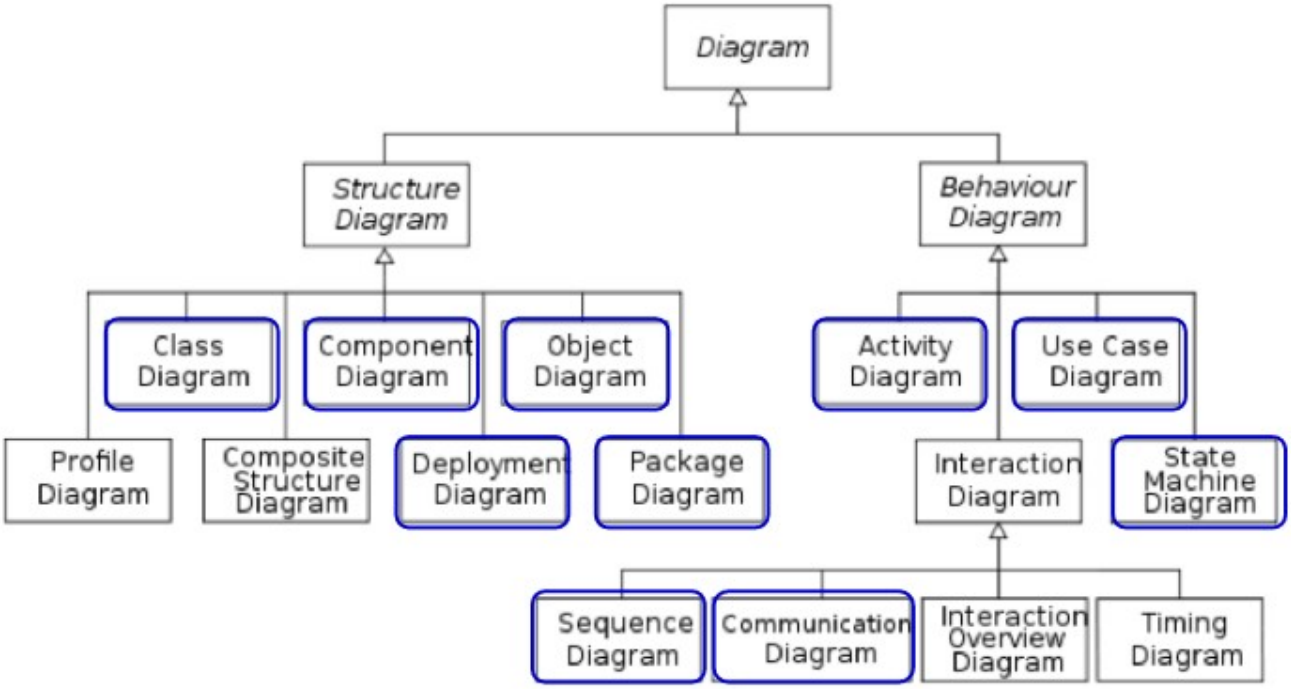
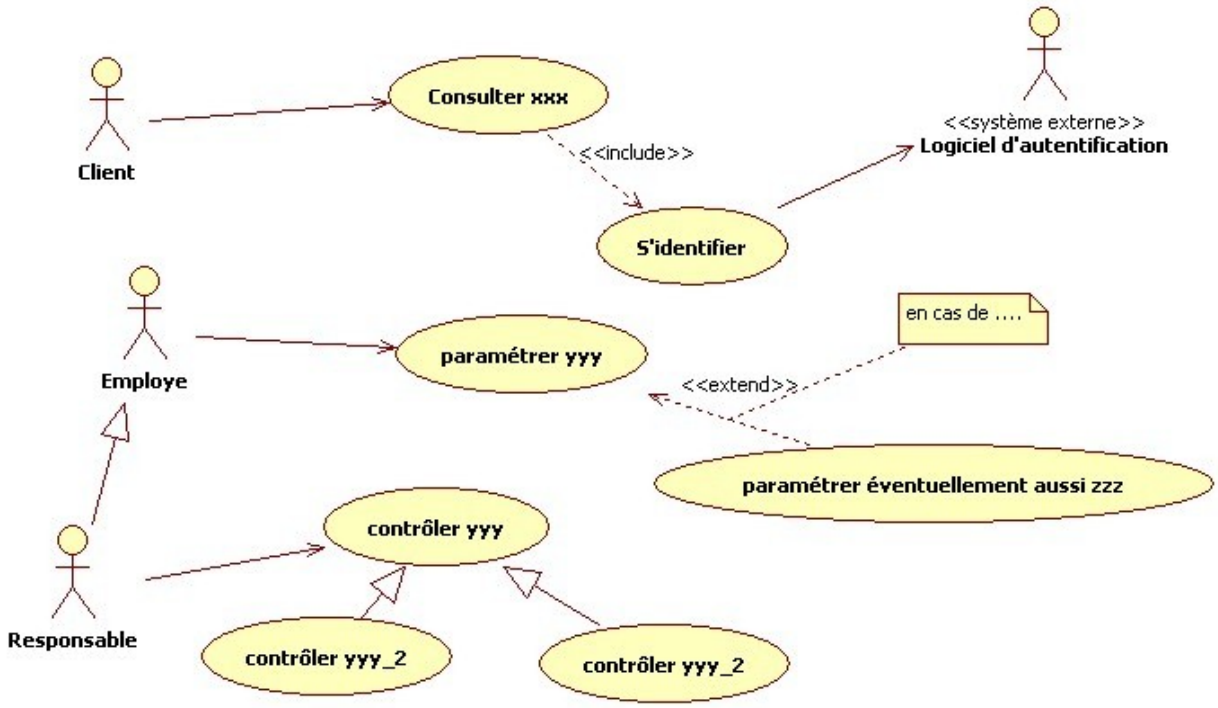


Diagram
Structure Diagram
Behaviour Diagram
Class Diagram
Component Diagram
Object Diagram
Activity Diagram
Use Case Diagram
Profile Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Interaction Diagram
State Machine Diagram
Sequence Diagram
Communication Diagram
Interaction Overview Diagram
Timing Diagram

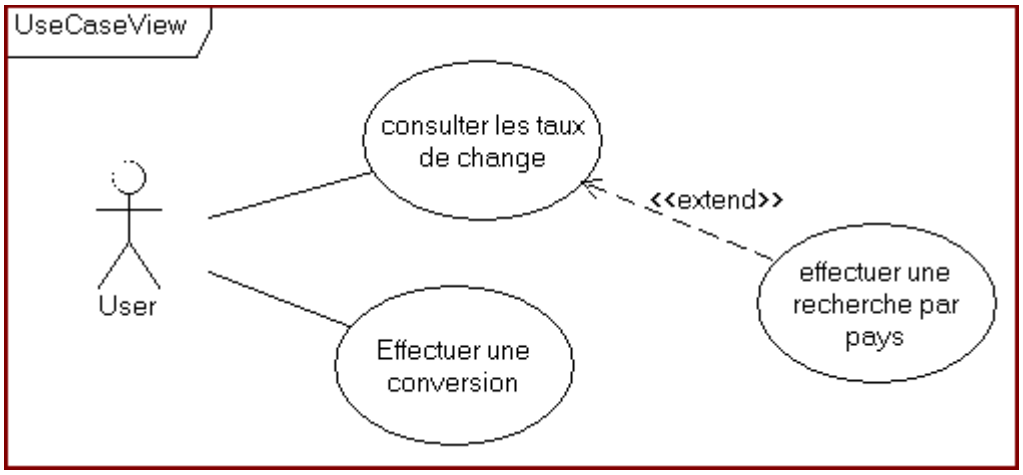
9. Descriptions sommaires des diagrammes UML

9.1. Diagramme des cas d'utilisations (Uses Cases)

exemple:



sur micro étude cas "*conversion de devises*":



9.2. Diagramme de classes

exemples:

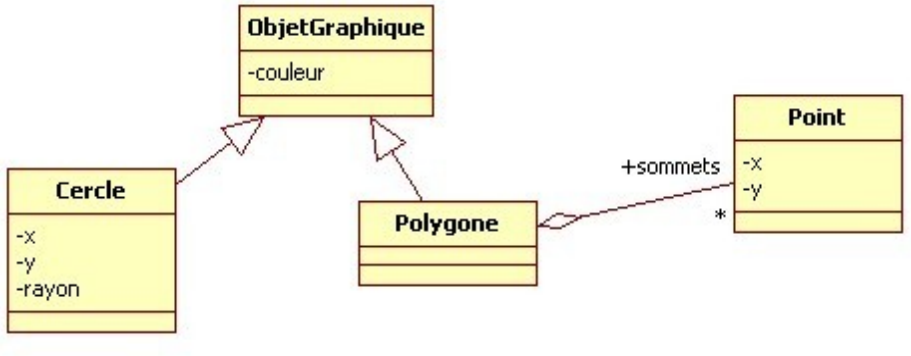
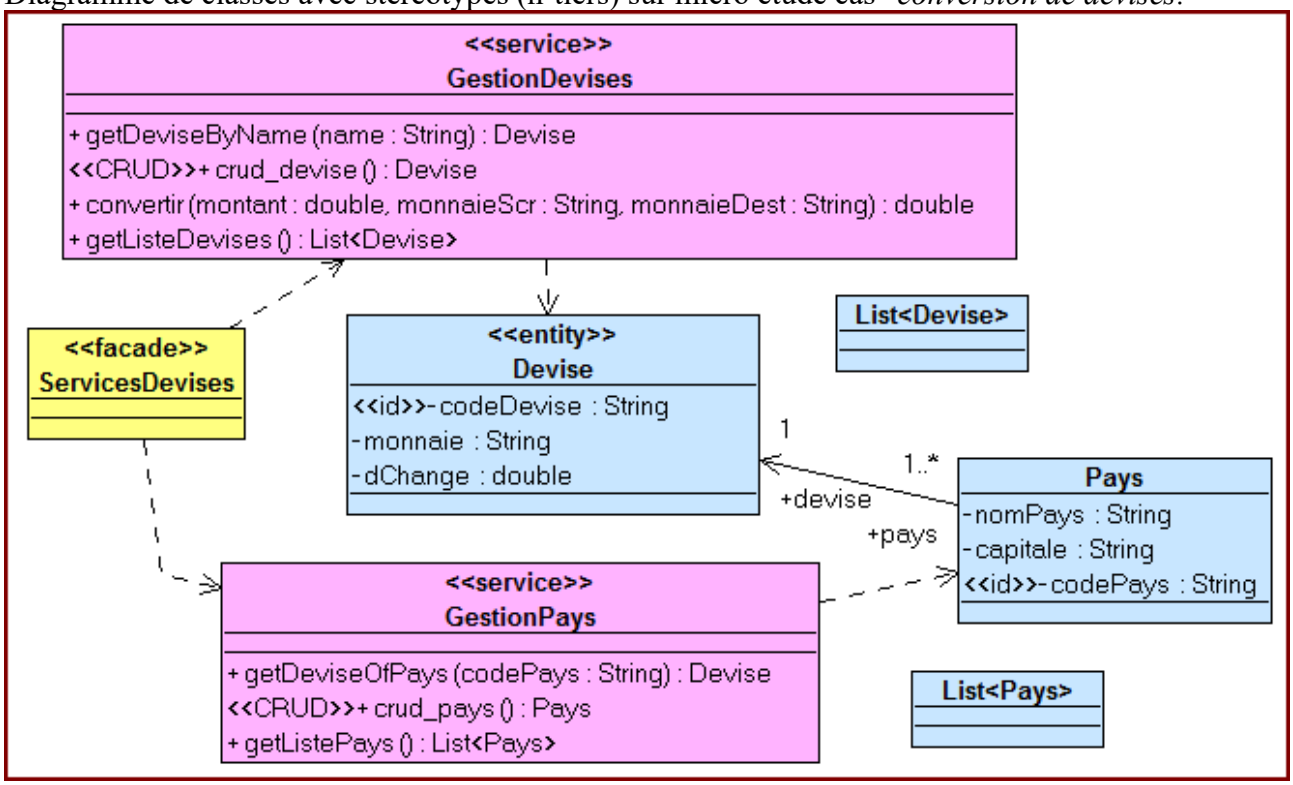
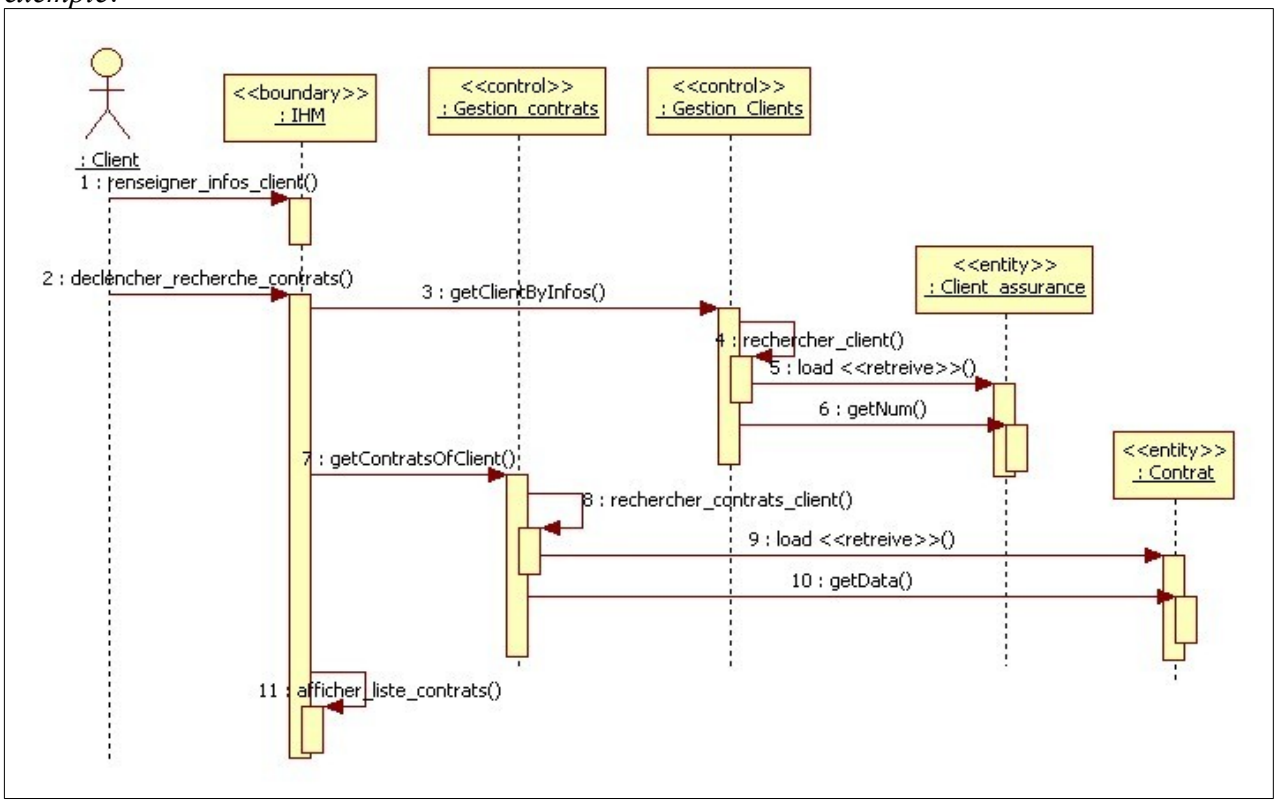


Diagramme de classes avec stéréotypes (n-tiers) sur micro étude cas "*conversion de devises:*



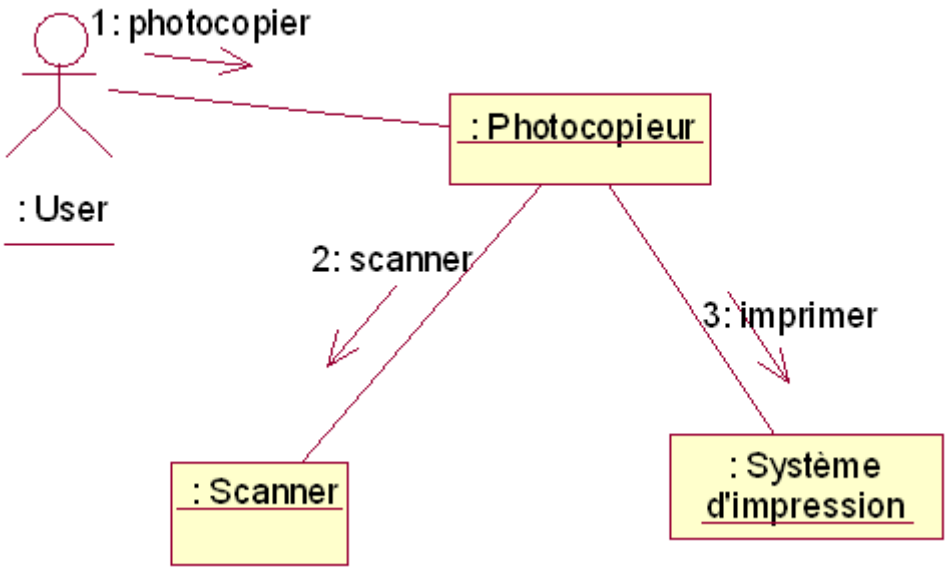
9.3. Diagramme de séquence

exemple:



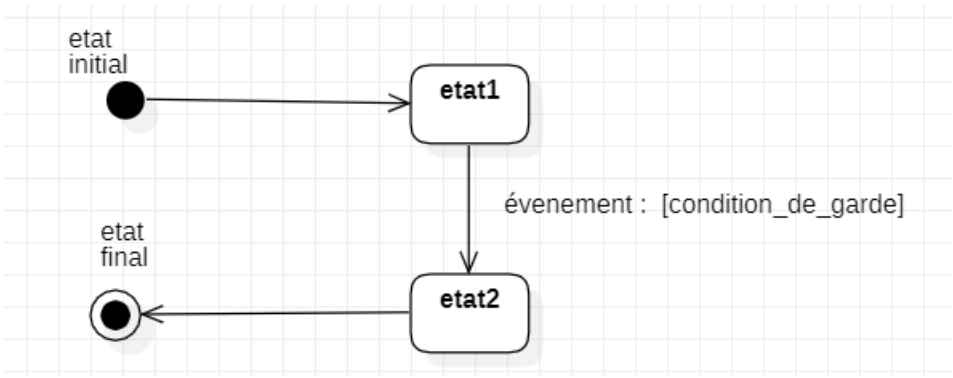
9.4. Diagramme de ~~collaboration (UML1)~~ / communication (UML2)

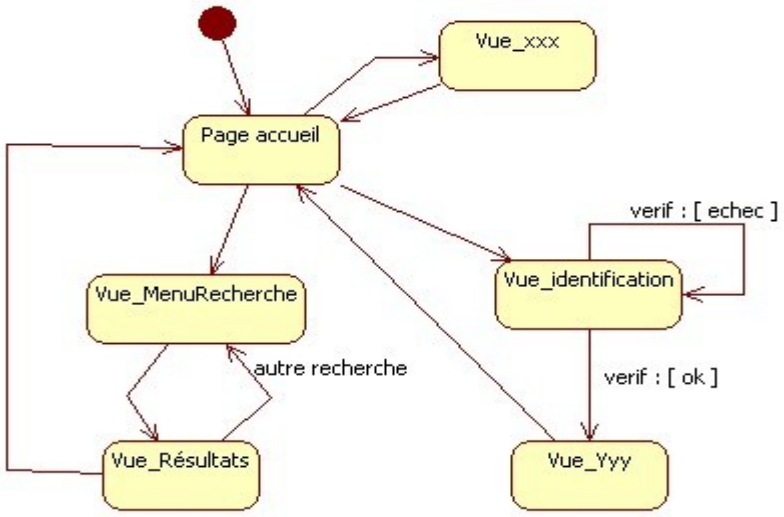
exemple:



9.5. Diagramme d'états (StateChart)

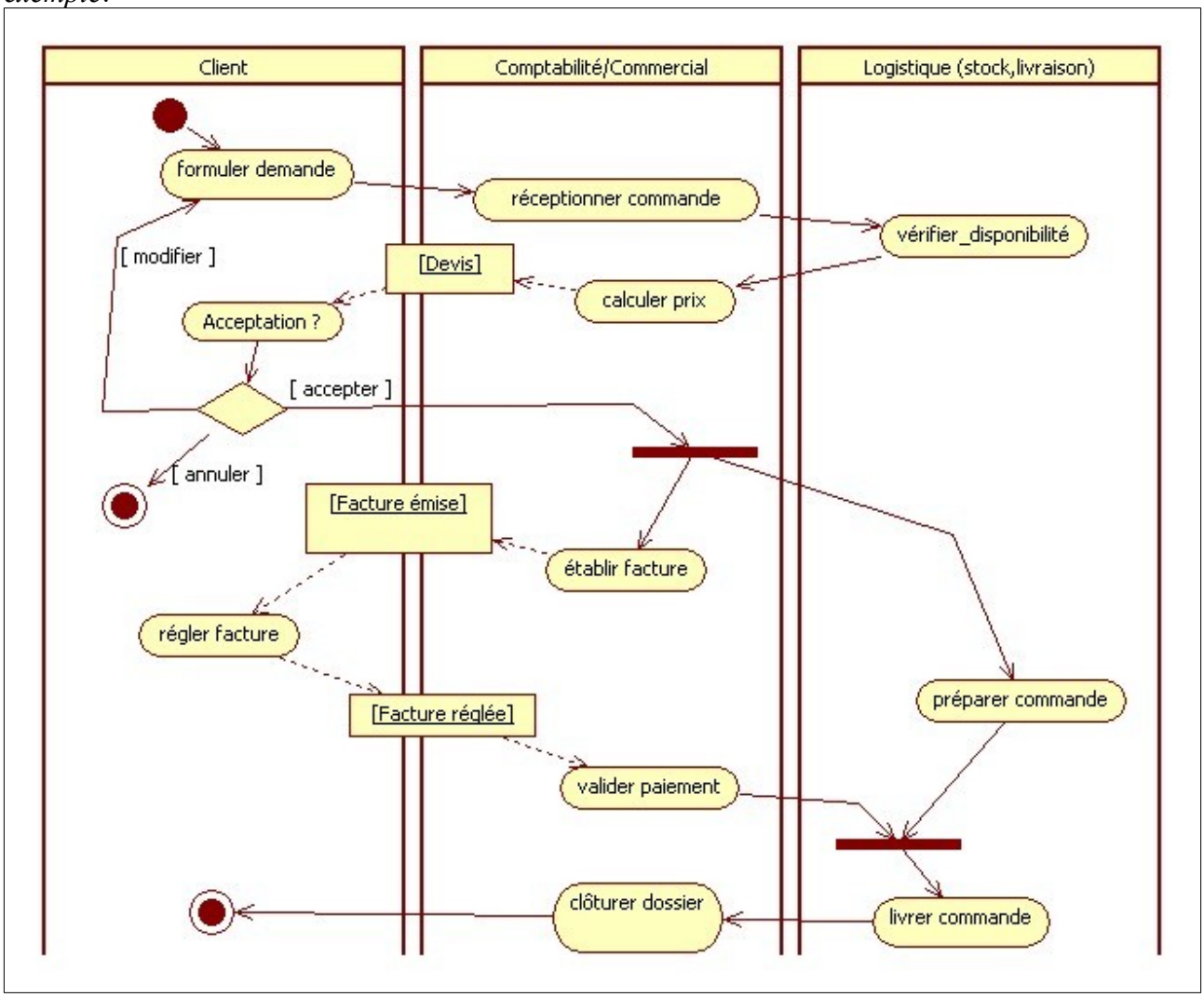
exemples:





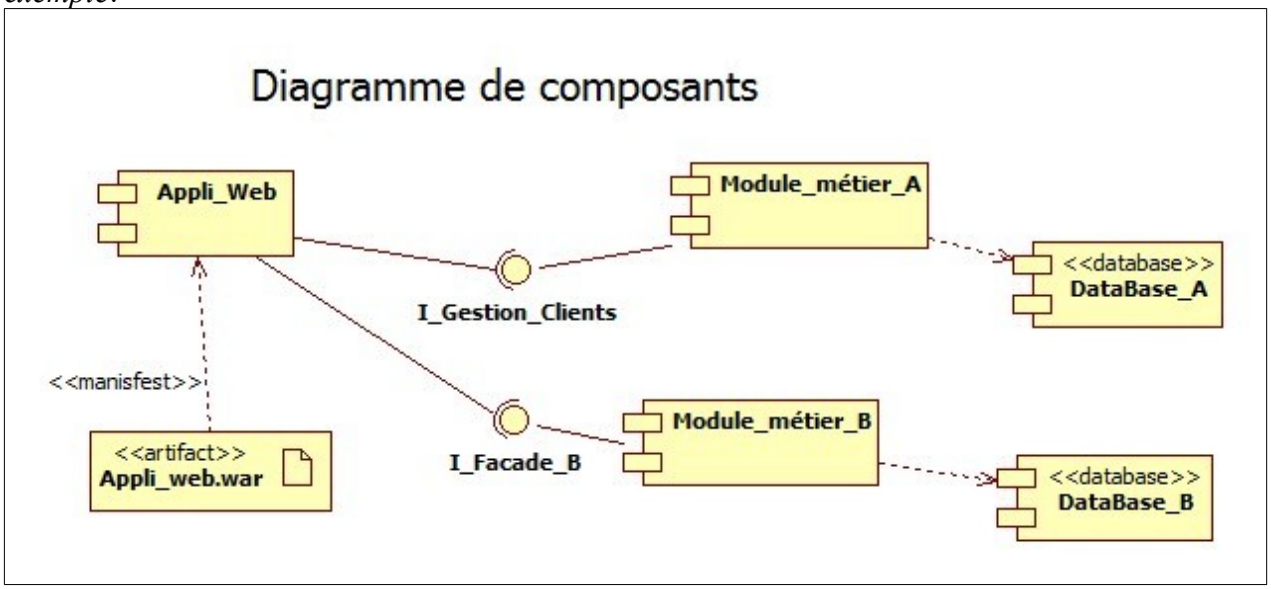
9.6. Diagramme d'activités

exemple:



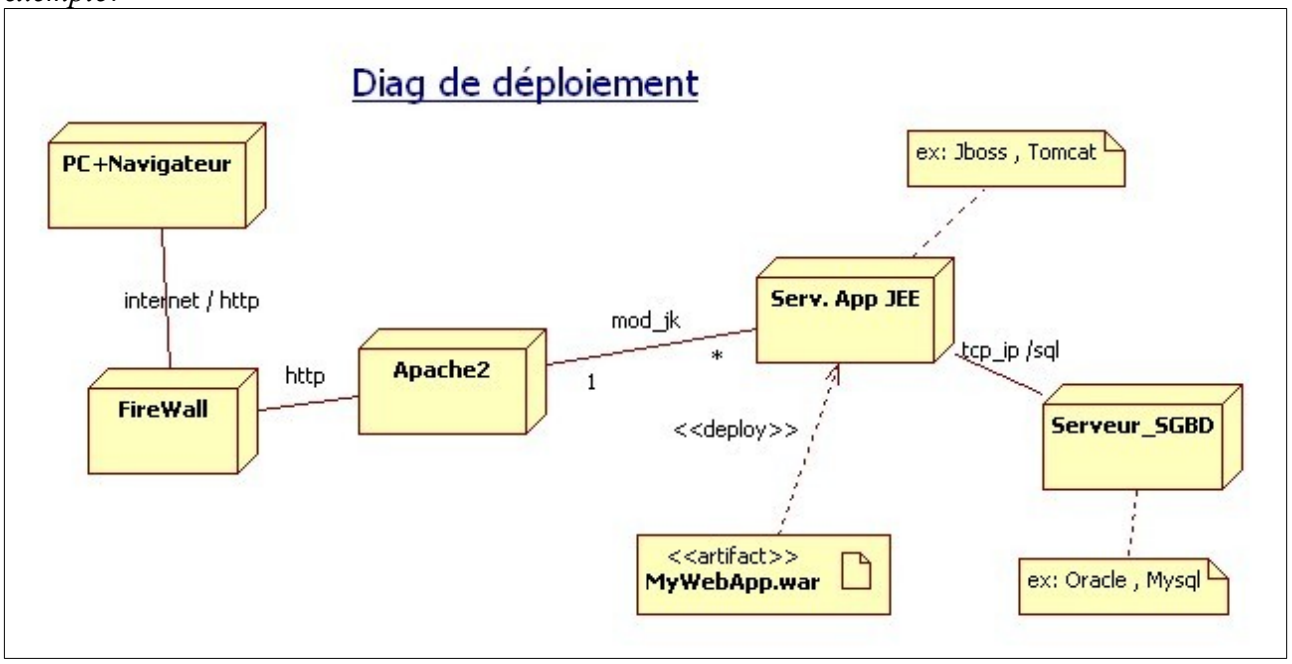
9.7. Diagramme de composants

exemple:



9.8. Diagramme de déploiement

exemple:



9.9. Diagrammes secondaires (variantes , ...)

*Diagramme de **robustesse** = diagramme de classe avec stéréotype <<boundary>> , <<control>> , <<entity>>*
*Diagramme **d'instances** = un peu comme diagramme de collaboration mais pour montrer l'état (valeurs des attributs) de quelques instances .*
*Diagramme de **packages** = diagramme de classes simplifié (avec que les packages et dépendances)*
....

10. Utilisations courantes des diagrammes UML

Diagrammes UML	*Utilisations courantes*
Diag. de packages	Vue d'ensemble sur **secteurs métiers/fonctionnels** (avec inter-dépendances) en analyse puis vue d'ensemble sur packages techniques de l'architecture logicielle en conception **Structure logique à grande échelle (pour vue d'ensemble)** --> diagramme principal ("main") , schéma d'urbanisation, ..
Diag. de classes	Montrer la **structure logique** des objets de l'application en **analyse** Montrer la **structure précise des composants** en **conception** ==> peut servir à générer le squelette du code **orienté objet**
Diag de structure composite	Montrer des **sous-constituants** (avec **ports**, **connecteurs** , ...) *--> quelquefois modélisé comme diag. de (sous-)composants*
Diag. de Uses Cases	Montrer les **principales fonctionnalités de l'application** et les liens avec les **acteurs extérieurs** (rôles utilisateurs , logiciels externes) ==> ***cartographie fonctionnelle*** Au moins un **scénario** attaché à chaque Use Case ==> guide pour l'analyse (traitements nécessaires) ==> guide pour les incrémentations du développement (*ordre de planification selon priorité des UC)* ==> guide pour les *jeux de tests*
Diag. de séquences	Montrer comment divers objets de l'application communiquent entre eux sous la forme d'une **séquence d'envois de messages *(interactions)***
Diag. de ~~collaboration~~ / communication	Montrer un ensemble d'objets qui collaborent entre eux et qui interagissent en s'envoyant des messages.
Diag. d'états	Montrer les différents **états** d'un objet ou d'un système complet (avec transitions=changements d'états déclenchés via événements).
Diag d'activités	Montrer une **suite logique d'activités** visant un objectif précis . ==> très utile pour modéliser des ***processus*** (avec début et fin) ==> bien adapté à la modélisation des ***workflows*** .
Diag de composants	Montrer la **structure des composants** (avec interfaces/connecteurs et ***dépendances***) --> essentiellement utile en conception
Diag de déploiement	Montrer la ***topologie (machine , réseau , serveurs , ...) de l'environnement cible*** (recette , production) afin de spécifier les détails du déploiement.

11. Quelques outils UML (Editeurs , AGL)

Outils/AGL UML	*Editeur*	*Open Source ?*	*Caractéristiques*
Rational Rose --> Rational XDE ---> RSA / RSM	Rational ---> **IBM**	non	Ancien leader du marché (dans les années 1996/2003) .Bon produit (très complet) mais assez cher .**(Rational Software Modeler)**
Together	Borland, Microfocus	non	Outil très ancien. Évolution récente ?
Star UML 1.x		oui	Produit gratuit assez complet (très inspiré de Rational Rose) .L'ancienne version 1.x n'a pas évolué depuis 2005 et n'évoluera plus (développé ancien langage "Delphi").
Star UML 2,3,4,5	MKLabs	partiellement	Nouvelle version de star uml entièrement redéveloppée en nodeJs (méta model json). licence d'environ 100 euros , version d'évaluation gratuite . Produit simple et intuitif .
Enterprise Architect	Sparx	Non mais pas cher (250 euros)	Basé sur environnement Microsoft .NET Outil assez complet , bonne ergonomie
MagicDraw UML	NoMagic, Dassault syst.	non	Bon produit , intègre très bien les normes récentes mais prix caché .
Visual Paradigm	VP	Non mais pas cher (90 euros)	Bon outil UML (complet et stable)
PowerAMC Designor	SDP	non	Outils pour MCD/Merise avec maintenant une partie UML
Visio (avec partie UML)	Microsoft	Non (environ 400 euros)	Outil graphique généraliste avec partie UML
Objecteering UML	Softeam (fr)	non	ergonomie très moyenne (ancien produit)
Modelio	Softeam (fr)	Cœur open source , extensions payantes	Bonne ergonomie , fichiers très propriétaires avec néanmoins import/export XMI.
Papyrus UML (plugin eclipse)	Topcased.org → Polarsys. ***Projet eclipse***	oui	Bon Plugin UML pour eclipse (bien/ très complet mais un peut sembler compliqué au départ)
GenMyModel	Startup près de Lille	non	Outil UML en ligne (nécessitant un simple navigateur) . Beaucoup de limitations en version gratuite .
...			

IV - Expression de besoin et analyse avec UML

1. Diagramme de contexte

On parle souvent en terme de "***diagramme de contexte***" pour désigner un petit diagramme de type "***vue d'ensemble***" permettant de **situer le système à développer dans son futur contexte d'utilisation**.

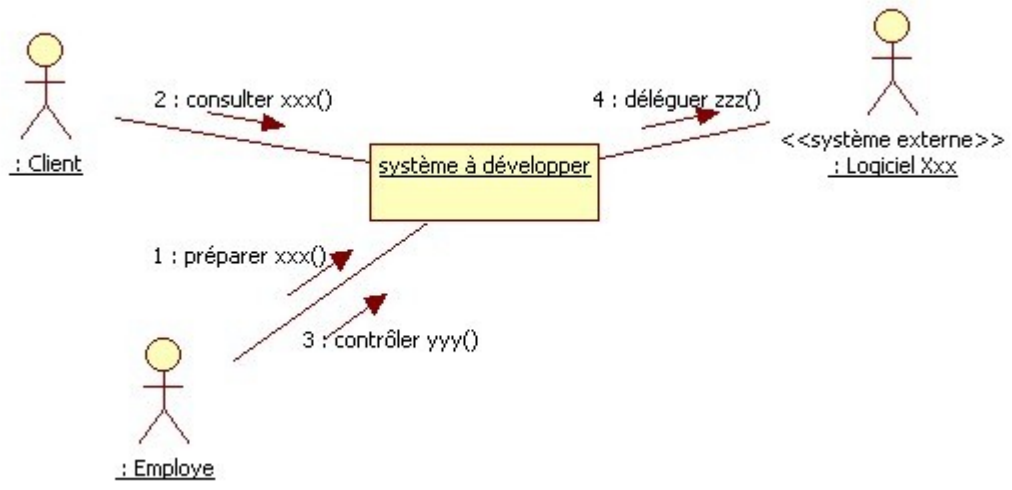
Bien que très utile, le diagramme de contexte n'est pas un diagramme officiel d'UML .
Selon les possibilités de l'outil UML on pourra concevoir le diagramme de contexte :

- comme un cas particulier de diagramme de collaboration/communication
- comme un diagramme simplifié de "Uses Cases" ou de "classes" (avec d'éventuels grands commentaires)
- ...

D'éventuels systèmes externes (qui seront sollicités pour déléguer/déclencher des services extérieurs) sont représentés comme des "acteurs UML" avec un stéréotype du genre <<système externe>> ou <<logiciel externe>> ou <<...>>

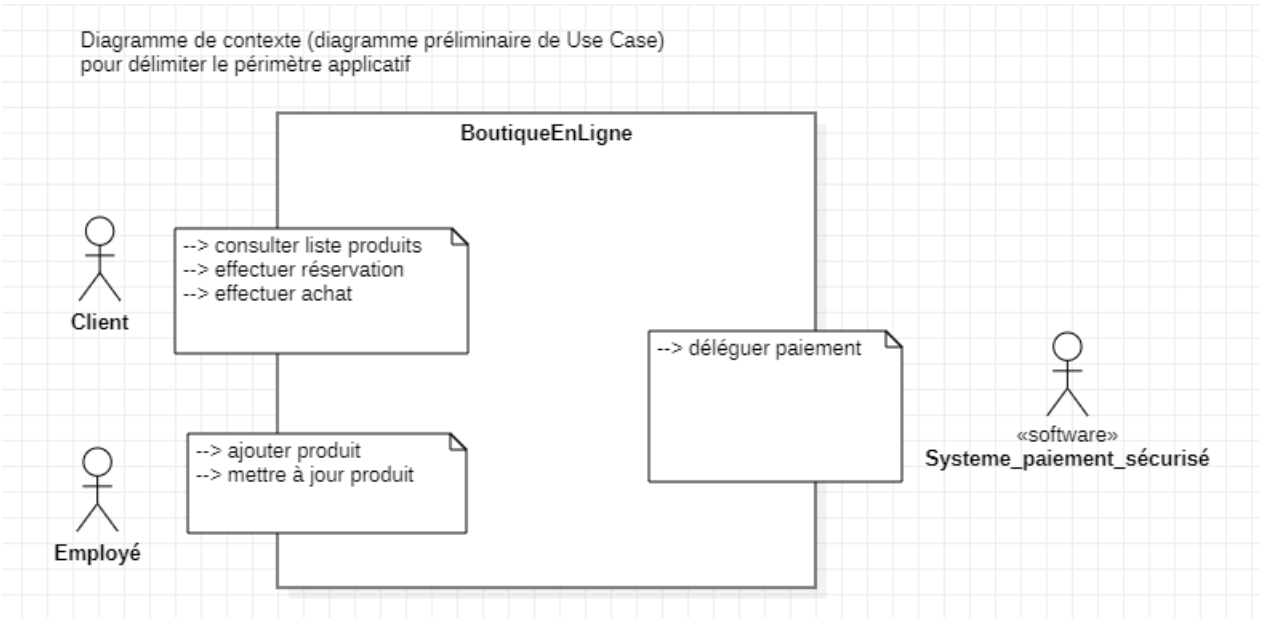
Les "messages" échangés à ce niveau sont souvent assimilés à des "***interactions" (systèmes ou utilisateurs)*** .

Exemple de diagramme de contexte élaboré sur la base d'un diagramme de collaboration/communication (avec ici l'ancien logiciel "StarUML 1") :



L'objet "système à développer" est placé en plein milieu du digramme.
Les **acteurs** (des futurs "Uses Cases") sont créés dans l'arborescence du modèle UML puis glissés/posés sur ce diagramme.
Finalement des **flèches d'interactions** sont posées sur des **liens** préalablement définis entre acteurs et objet "système à développer".

Exemple de **diagramme de contexte** bâti sur un diagramme de "Uses Cases" simplifié avec des **commentaires** *(réalisé ici avec star-uml 5)*:

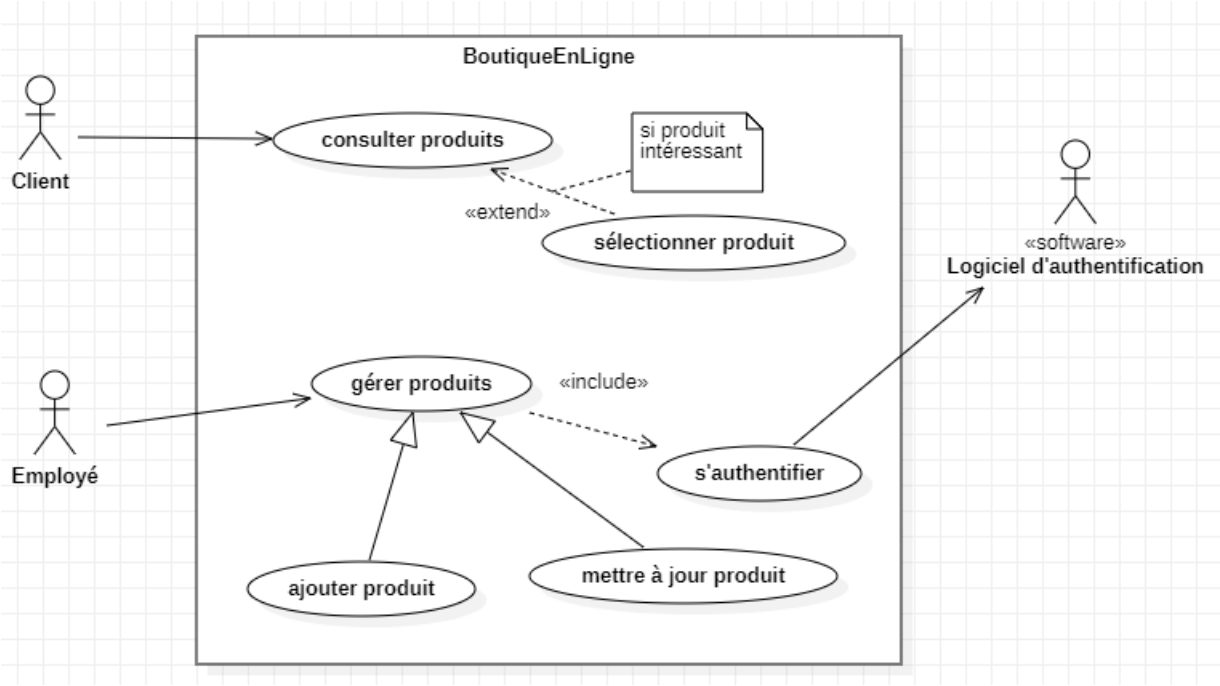


Bien que "non standard" / "non officiel" , un tel diagramme de contexte est **très rapide à élaborer** et donne une bonne vue d'ensemble sur les différents "futurs acteurs UML" et les interactions (entrantes ou sortantes) associées.

2. Etude des principales fonctionnalités (U.C.)

La notion de "**cas d'utilisation**" ou "**use case**" correspond à la fois à:

- **une (grande) fonctionnalité du système à développer**
- **un objectif (ou sous objectif) utilisateur**
- **un cas d'utilisation du système se manifestant par au moins une interaction avec un acteur extérieur .**



NB:

- Au sein d'un diagramme UML de "uses cases" comme le précédent , il est conseillé de **nommer les cas d'utilisation** par des termes du genre "***verbe_complément_d'objet_direct***". Les ***compléments d'objets*** permettront d'associer ultérieurement les cas d'utilisations aux ***entités métiers*** et aux ***services métiers.***
- NB: **<<*include*>>** pour sous tâche indispensables/obligatoires , **<<*extends*>>** pour tâches d'extensions facultatives.

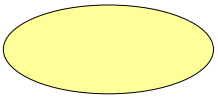
2.1. Méthodologie de base (au niveau des cas d'utilisation)

1) réaliser le (ou les) diagrammes de "Uses Cases" (vue d'ensemble / cartographie fonctionnelle)
2) Ajouter un descriptif (textuel ou autre) à chaque cas d'utilisation.
   La partie essentielle de ce document descriptif est le ***scénario nominal*** (séquence d'étapes ou ***"pas" élémentaires*** [sous forme d'*interaction acteur/système*] permettant d'assurer la fonctionnalité attendue).

--> Le tout de façon itérative (avec revue/relecture des diagrammes et des scénarios)

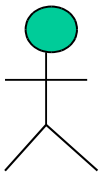
3. Diagramme des cas d'utilisations

Présentation des « Use Case »



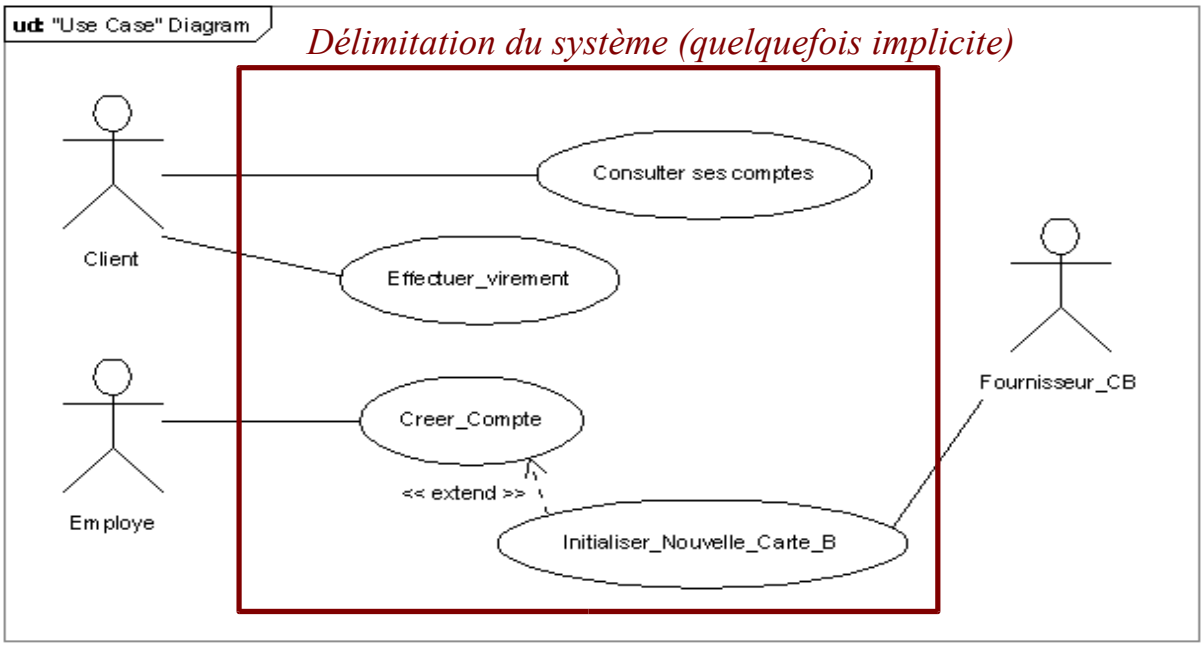
Faisant partie intégrante de UML, les "**USE CASE**" offrent un **formalisme** permettant de:

- **Délimiter (implicitement) le système** à concevoir.
- **Préciser** les **acteurs extérieurs** au système.
- **Identifier** et **clarifier** les **fonctionnalités du** futur **système.**



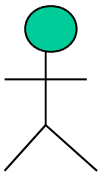
Nb: il s'agit d'une **vue externe** (***point de vue de l'utilisateur***).

Diagramme U.C. (Vue d'ensemble)



Acteurs

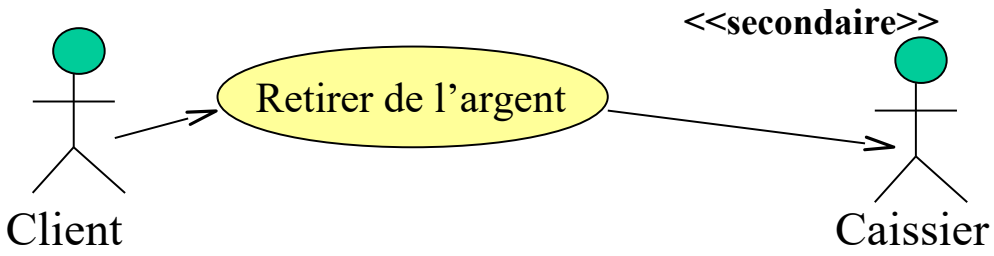
- Un **acteur** est une **entité extérieure au système** qui **interagit d'une certaine façon avec le système** en **jouant un certain rôle**.



- Un acteur ne correspond pas forcément à une catégorie de personnes physiques .
  Un automatisme quelconque (Serveur, tâche de fond , ...) peut être considéré comme un acteur s'il est extérieur au système.
  ==> Deux grands types d'acteurs *(éventuels stéréotypes)* : "***Role_Utilisateur***" , "***Système_Externe***"

Acteurs primaire et secondaire

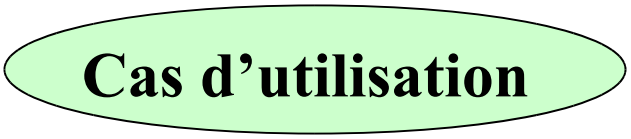
- Un cas d'utilisation peut être associé à 2 sortes d'acteurs:
  - **L'unique acteur primaire (principal)** qui déclenche le cas d'utilisation. **C'est à lui que le service est rendu**.
  - Les éventuels **acteurs secondaires** qui **participent** au cas d'utilisation en apportant une aide ***quelconque (ceux-ci sont*** sollicités par le système).



Cas d'utilisation

- Définition: ***Un cas d'utilisation (use case) est une fonctionnalité remplie par le système et qui se manifeste par un ensemble de messages échangés entre le système et un ou plusieurs acteur(s).***

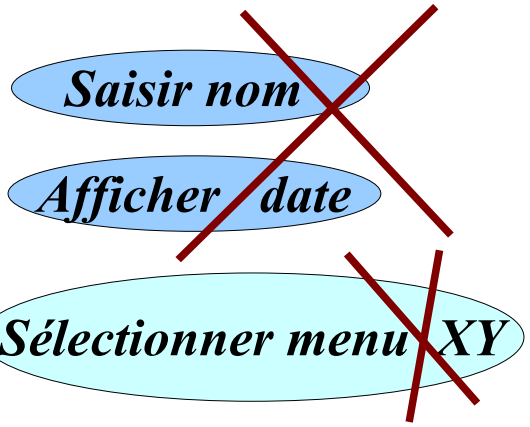
Notation:



A ne pas faire:

Trop détaillé(s)
(sera détail d'un futur scénario)

IHM mais ***pas fonctionnel*** *!*



A prendre en compte:

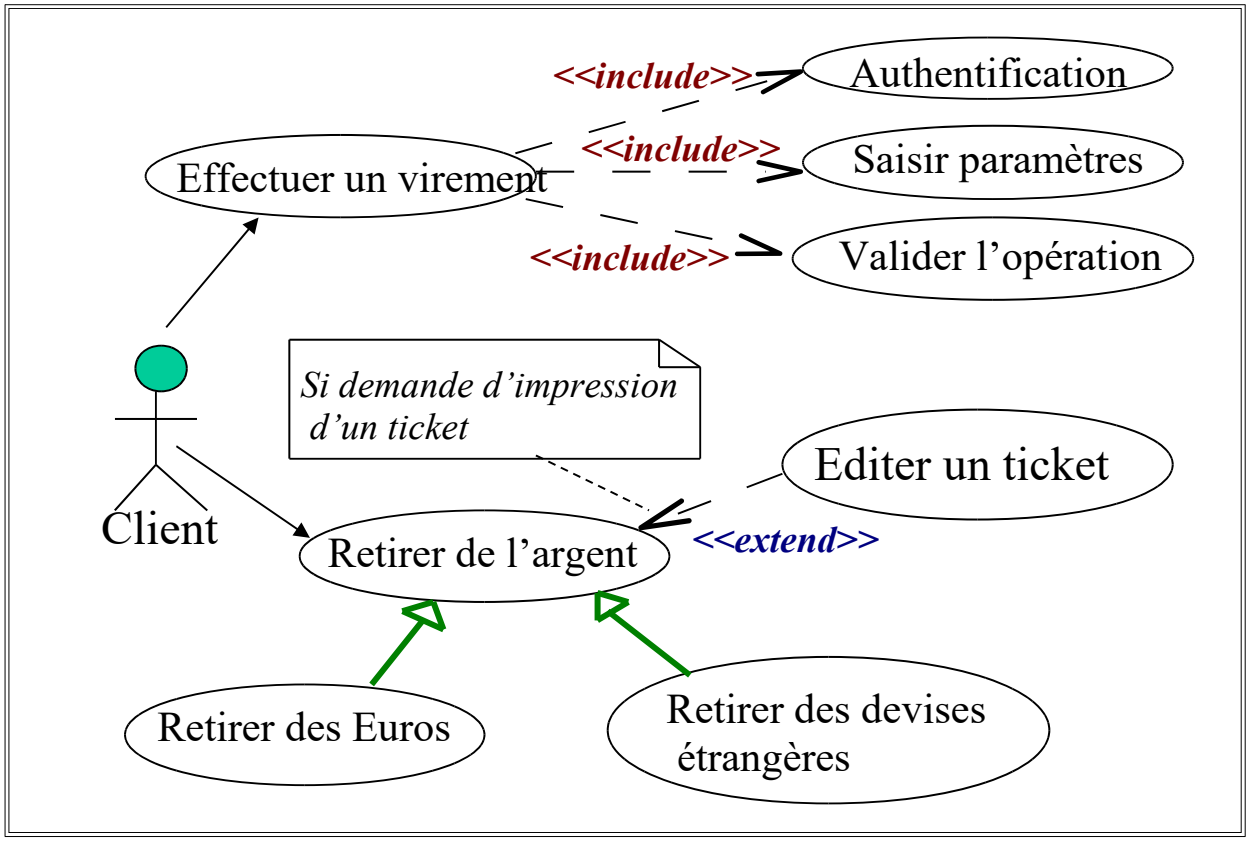
* cas d'utilisation <--> session utilisateur
  --> On peut donc relier entre eux les U.C. Effectués à peu près au même moment mais on doit séparer ce qui s'effectue à des instants éloignés (différentes sessions)
* cas d'utilisation --> avec interaction avec l'extérieur .

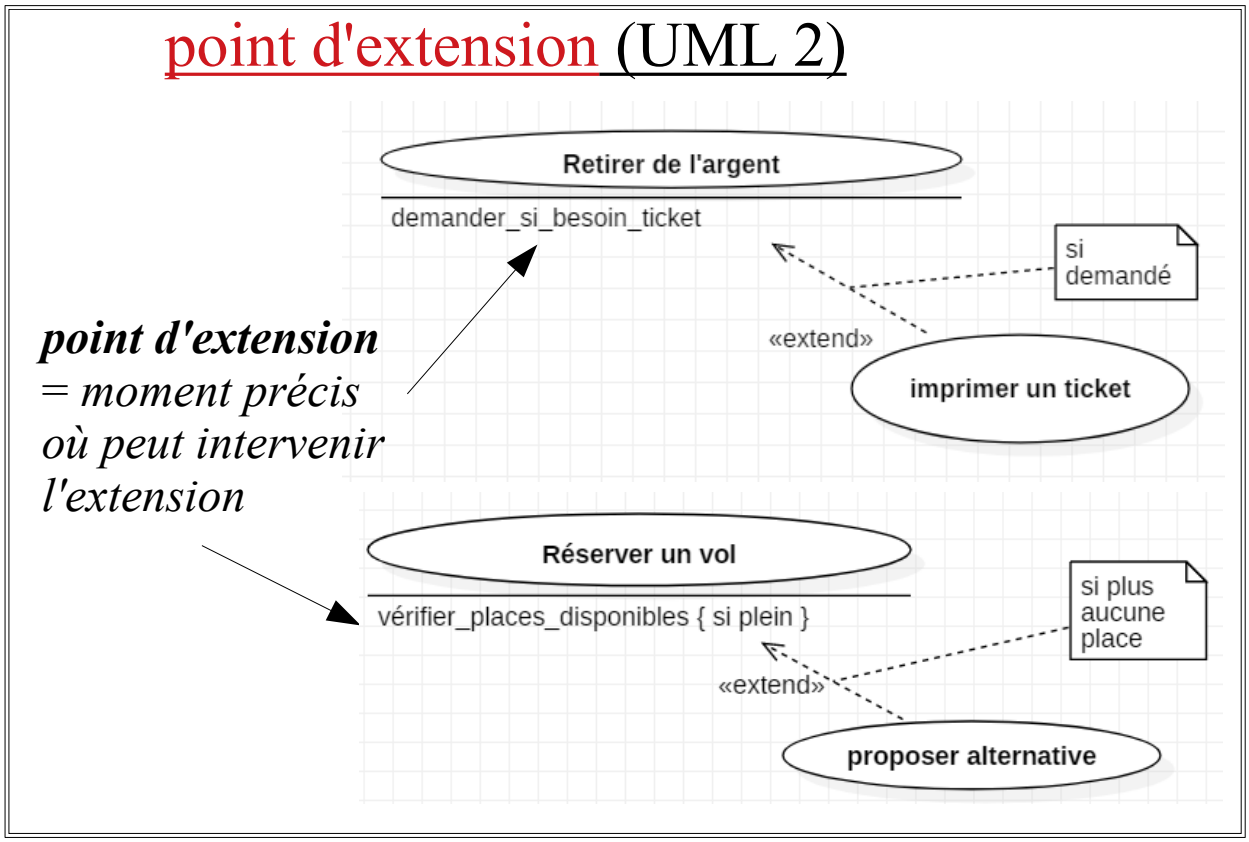
Relations entre UC

- Le cas d'utilisation A ***<<include>>*** le (sous-) cas d'utilisation B si ***B est une sous partie constante (systématique) de A***.

- Le cas d'utilisation *(supplémentaire)* C ***<<extend>>*** le cas d'utilisation A si **C est une partie additionnelle qui s'ajoute à A le cas échéant (facultativement)**.
  Une **note (commentaire) spéciale** appelée ***« cas d'extension »*** permet de préciser le cas ou C étend A.

- La **relation d'héritage** (ou généralisation) classique:
  D → **E** permet d'exprimer que le cas d'utilisation D **est une sorte** ***(variante, déclinaison)*** du cas d'utilisation E.

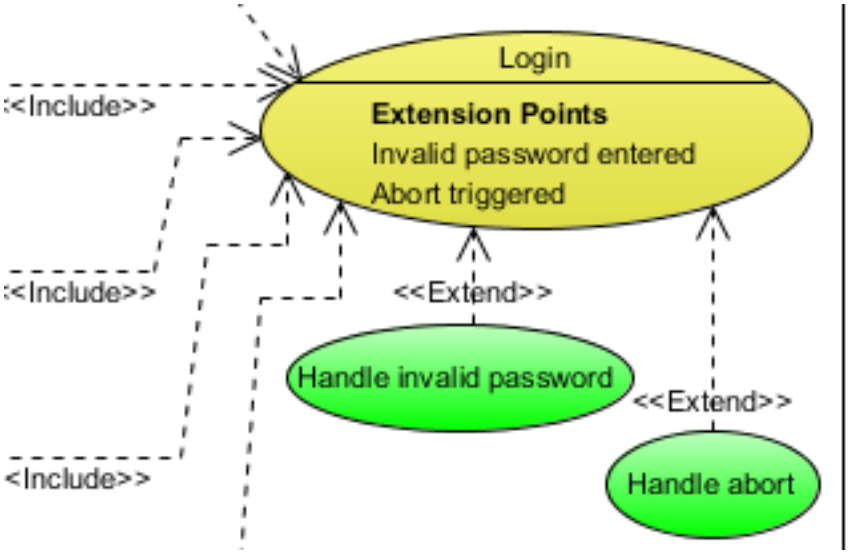




NB :

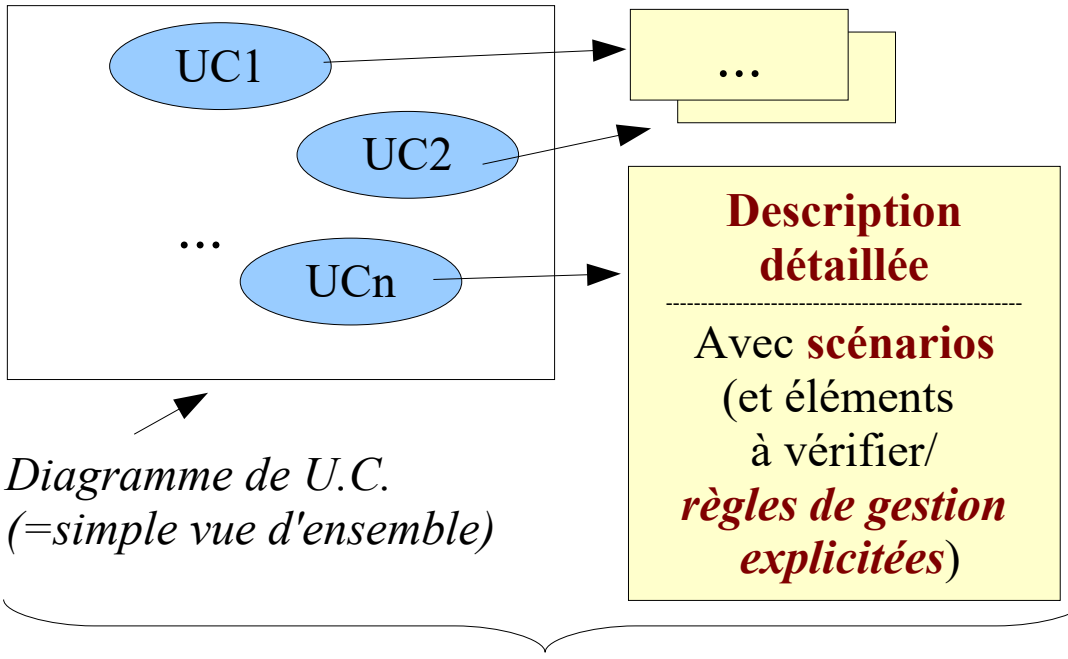
- Au sein d'un modèle UML (où tout est plus ou moins facultatif), un point d'extension bien formulé (ex : demander_si_besoin_ticket) permet de déduire le cas d'extension évident (ex : imprimer un ticket) qui peut alors ne pas être représenté (car considéré comme implicite) .

- selon l'éditeur UML utilisé la syntaxe des "Uses Cases" et des points d'extensions peut éventuellement varier un peu :



4. Scénarios et descriptions détaillés (U.C.)

Pour chaque cas d'utilisation



Spécifications fonctionnelles / reformulation du cahier des charges

Description textuelle (U.C.)

Bien que la norme UML n'impose rien à ce sujet, l'**usage** consiste à documenter chaque cas d'utilisation par un texte (word , html, ...) comportant les rubriques suivantes:

- **Titre** (et éventuelle numérotation)
- **Résumé** (description sommaire)
- Les **acteurs** (primaire, secondaire(s) , rôles décrits précisément)
- **Pré-condition(s)**
- **Description détaillée (scénario nominal)**
- **Exceptions** (scénario pour cas d'erreur)
- **Post-condition(s)**

Scénarios (U.C.)

Scénario nominal:

1) l'usager place la carte dans le lecteur
2) l'usager renseigne son code secret
3) le système authentifie et identifie l'usager
4) l'usager sélectionne le montant à retirer
5) le système déclenche la transaction (débiter le compte de l'usager)
6) le système rend la carte à l'usager
7) le système distribue les billets et un éventuel ticket

Scénario Nominal

Tout se passe bien

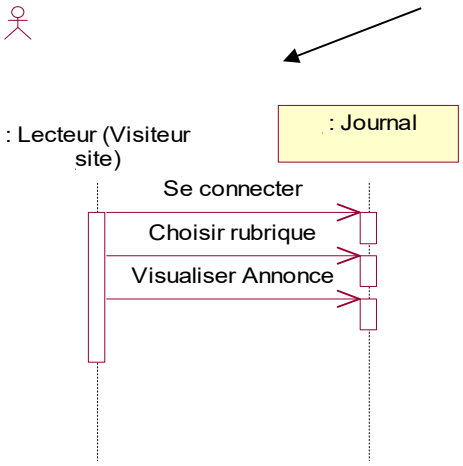
Cas/Déroulement le plus fréquent

Scénario d'exception "E1":

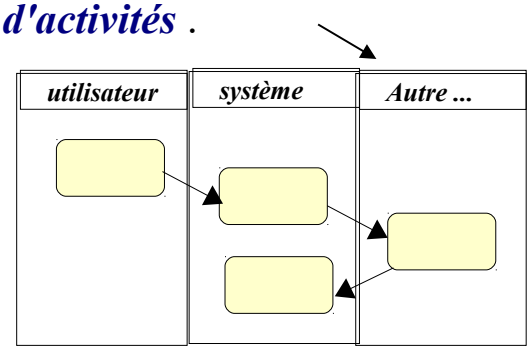
si étape l'étape (3) échoue trois fois de suite alors
- avaler carte
- ne pas effectuer les étapes (4) à (7)
- afficher un message explicatif

Illustration éventuelle (U.C.)

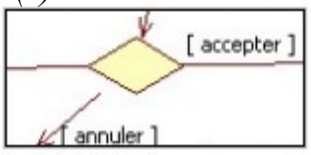
Un scénario peut éventuellement être graphiquement exprimé/illustré par un ***diagramme de séquence UML***.



Un ou plusieurs scénario(s) peuvent également être exprimés à travers un ***diagramme d'activités*** .



*Avec **décision**(s) :*



NB : **Un diagramme d'activité** (avec des décisions et différentes alternatives) **permet de synthétiser visuellement certains cas de figures** (qui seraient modélisés en mode texte via plein de scénarios longs/fastidieux à rédiger) .

NB :

Un scénario peut :

- soit être rédigé dans un ***fichier annexe*** (via un traitement de texte : Word ou libreOffice ou ...)
- soit être rédigé directement au sein de l'éditeur UML (par exemple dans la partie "***Documentation***")

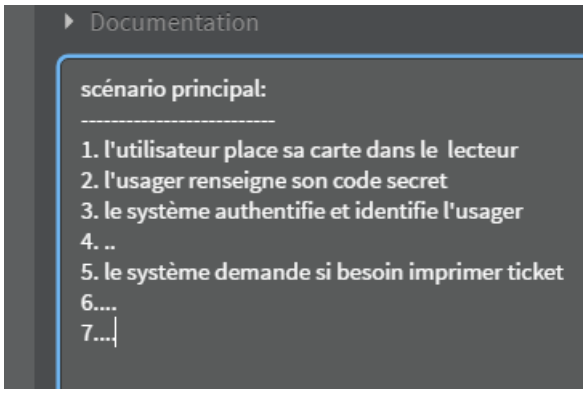


Tableau récapitulatif des U.C. (base pour la planification)

Use Case	*Estimation charge (j/h)*	*Priorité*	*Autres caractéristiques*
UC_1	6	++	techniquement difficile
UC_2	8	--	facile
UC_3	5	+	...
UC_n	6	-	....

NB: estimation charge = (modélisation + implémentation + tests + intégration)
Priorité en partie selon *<<include>> (+,++)* et *<<extend>> (-,--)*

5. Diagramme d'activités

Diagramme d'activités

Un **diagramme d'activités** montre les **activités effectuées séquentiellement ou de façon concurrente** par un ou plusieurs éléments (acteur, personne, objet).

Principales utilisations:

- **Workflow** et **processus métier**.
- **Organigramme** (pour algorithme complexe).

5.1. lien entre un diagramme d'activité et un use case

Un "**use case**" (cas d'utilisation) représente généralement un "**objectif utilisateur**" (ou sous objectif).

Pour atteindre cet objectif via un système informatique , l'utilisateur doit mettre en œuvre un **processus métier (succession de tâches/activités conditionnées)**.

Le diagramme d'activité UML est un des formalismes permettant de modéliser un processus métier.

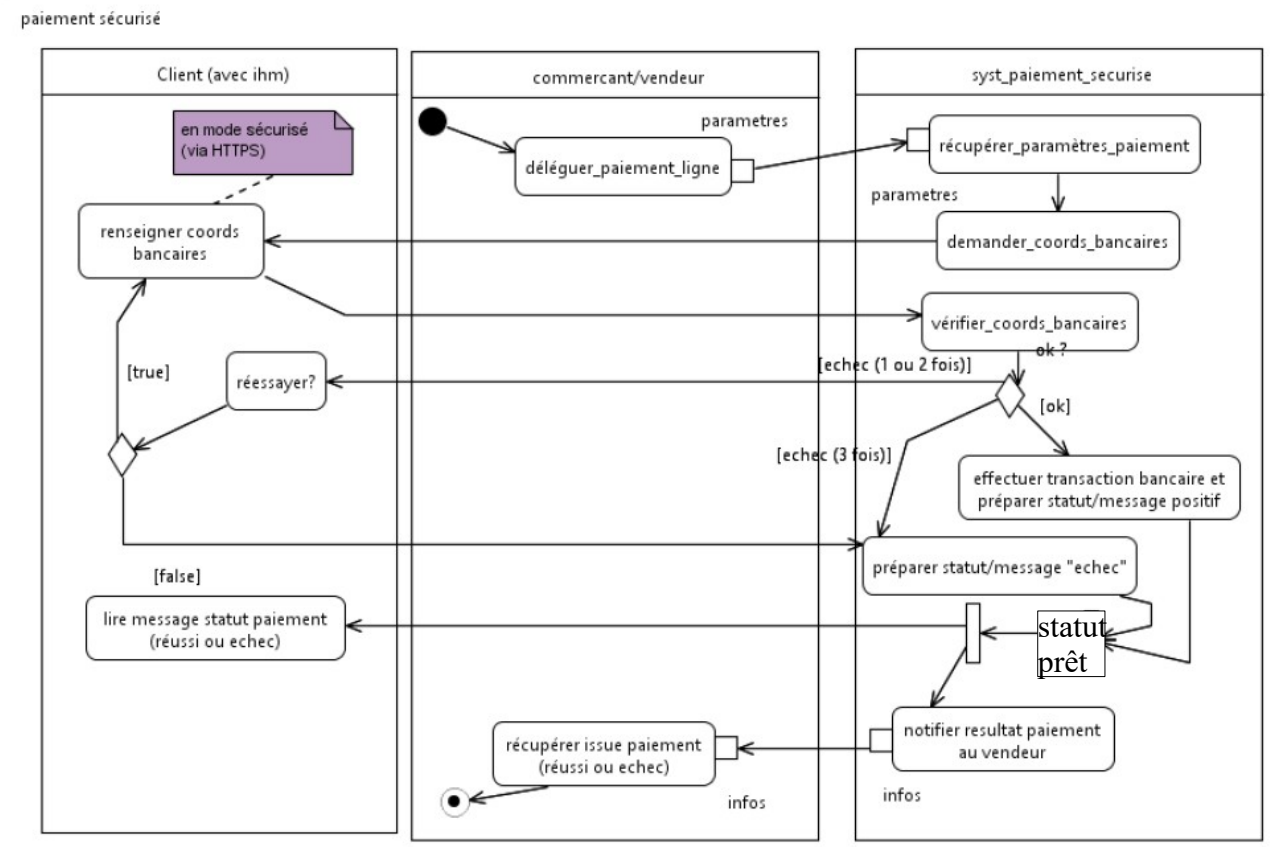
Dans l'arborscence d'un modèle UML, un diagramme d'activité est assez souvent placé comme un détails d'un cas d'utilisation (use case).

Les principaux atouts d'un diagramme d'activités sont les suivants :

- via les losanges de "décision" avec conditions sur les flèches sortantes, un digramme d'activités peut synthétiser plusieurs scénarios ou variantes.
- très lisible (si on se limite aux syntaxes compréhensibles par tout le monde)
- très parlant/significatif pour un non informaticien et donc pratique pour discuter des fonctionnalités fonctionnelles/métiers attendues par les utilisateurs

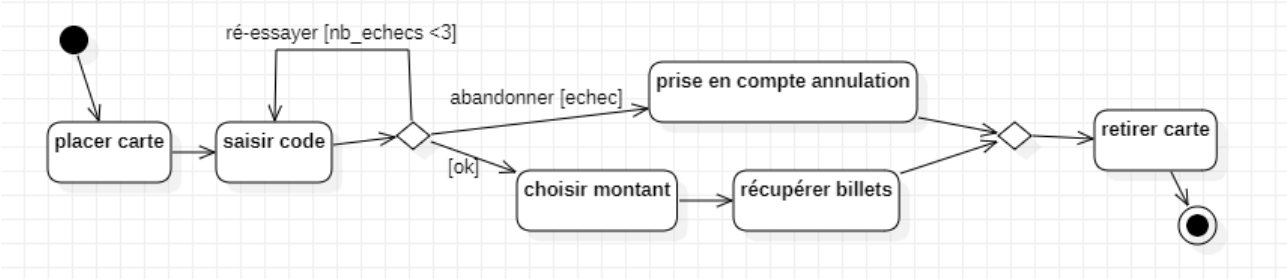
5.2. couloirs d'activités (facultatifs)

Ces **couloirs** *(sous forme de colonnes)* sont quelquefois appelé "**partitions**" ou encore " *swimlane / lignes d'eau*" et permettent d'indiquer "**qui** fait quoi" :



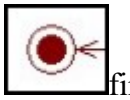
Au sein de l'exemple ci dessus les petits rectangles blancs sur les bords des actions correspondent à des "output pin" et "input pin" associés à des "object flows".

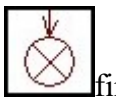
Exemple de diagramme simple (sans couloir) :



5.3. Noeuds spéciaux et de contrôles

 Initial/début *(normalement unique)*

 fin complète(de toutes les branches du processus)
En général : une fin ordinaire (atteinte de l'objectif) et d'éventuelles "fin" de type "annulation" .

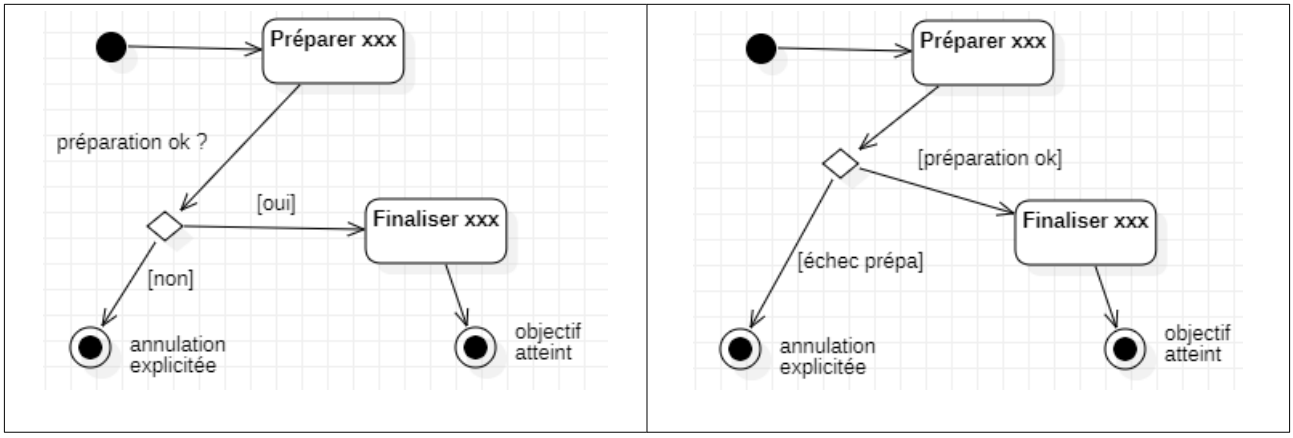
 fin de flot/branche *(ex : fin d'exécution d'un thread ou d'une tâche parallèle)*

Barre de synchronisation avec une entrée et plusieurs sorties :
 **bifurcation (traitement en //)**

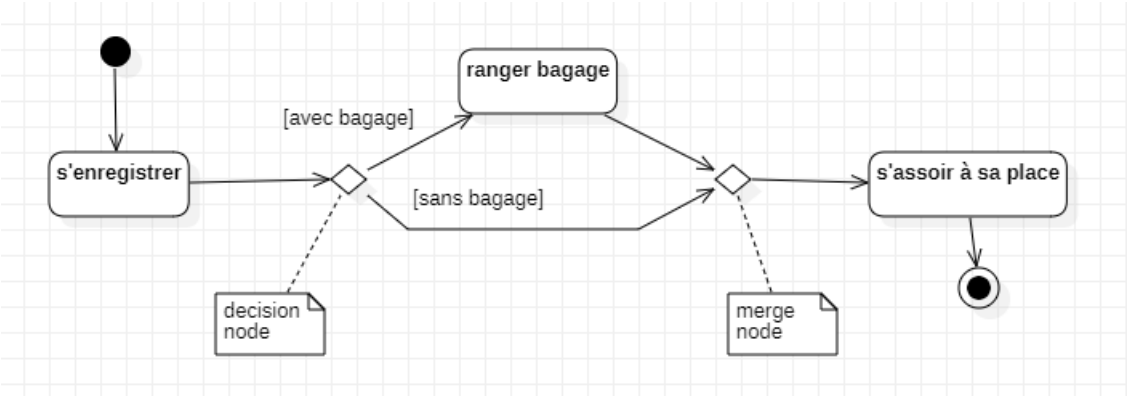
 union/**synchronisation de type "et logique"**

Pour modéliser un "ou" --> plusieurs transitions entrantes vers une même activité (sans barre de synchronisation).

Décision <> avec ***condition de garde*** entre [ ] :



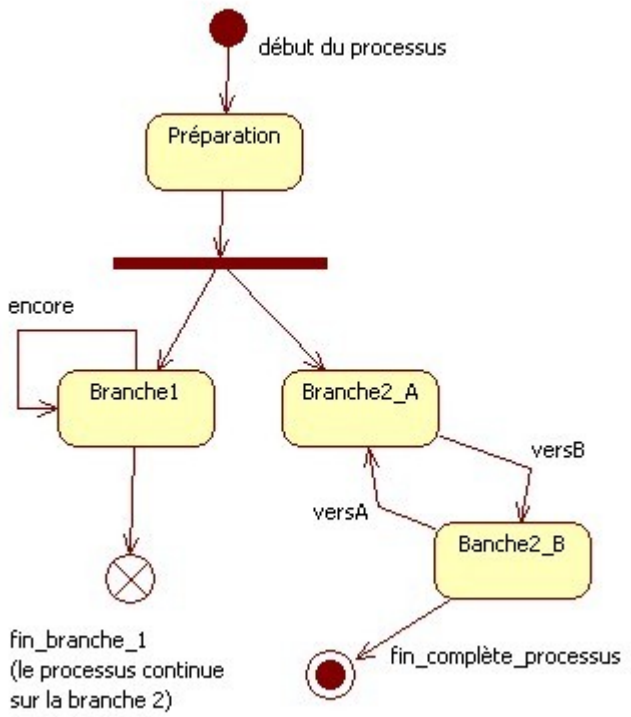
Selon l'outil UML , le texte de la décision peut être placé soit sur le losange de décision, soit sur la flèche entrante , ou encore être implicite (non affiché).
Sur les flèches sortantes du losange, les conditions à respecter ("guard") sont normalement automatiquement encadrées par des crochets.



Un "merge node" peut être vu comme fin de variante initiée par un "decision node" .

5.4. Final flow

Final Flow = Arrêt du flux sur une branche (une autre branche peut éventuellement continuer)
Exemple :



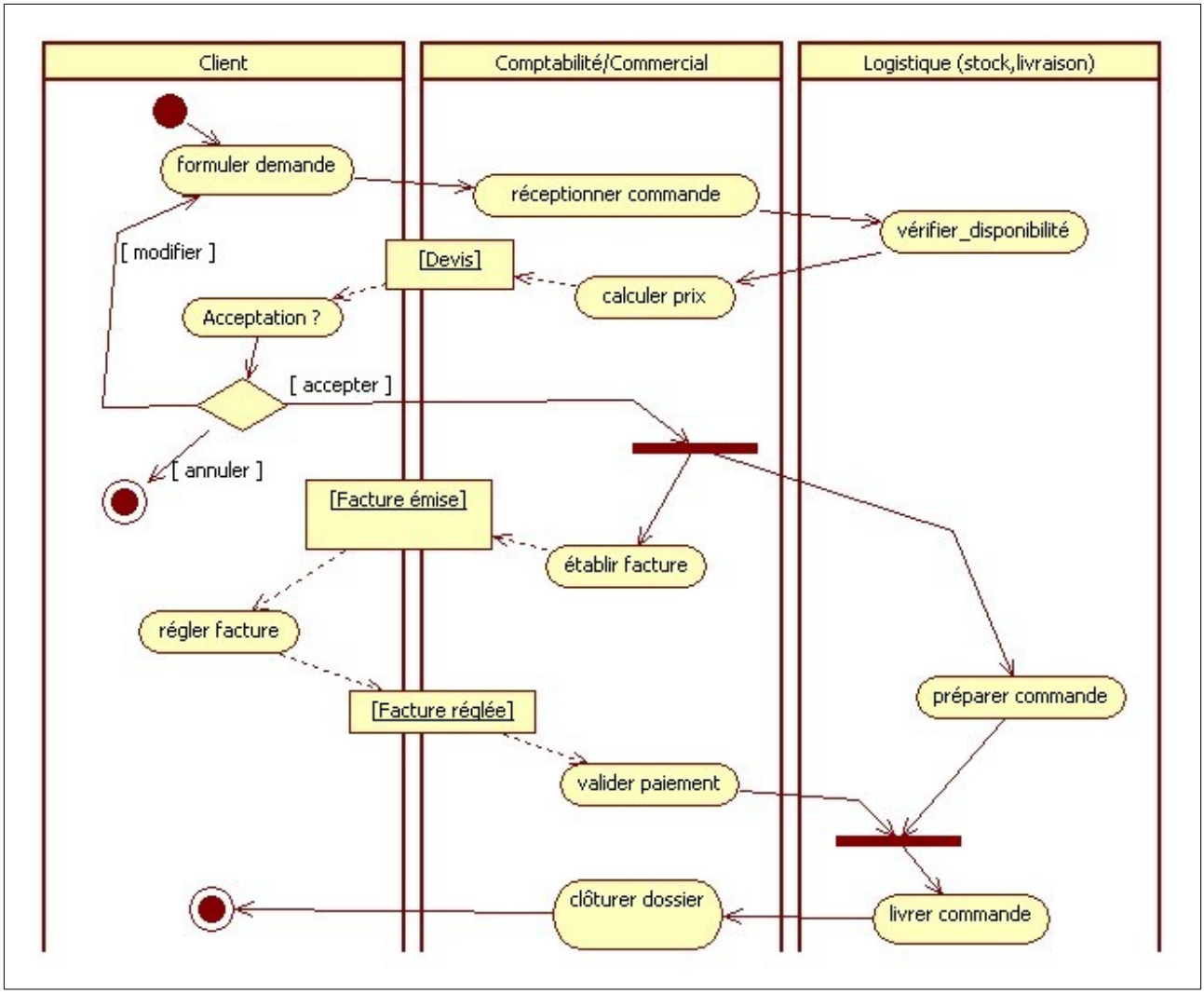
5.5. object flow (nœuds "objet") : UML 1 et 2

Un noeud "objet" correspond à un message (ou flot de données) construit par une activité préalable et qui sera souvent acheminé en entrée d'une autre activité (exemples: lettre , devis , facture , mail , ...).

Syntaxe "UML 1" encore supportée par quelques outils UML2 :

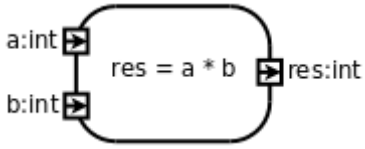
Un **rectangle** comportant **[le nom de la donnée et son état]** entre crochet.

Exemple (UML 1 et 2):

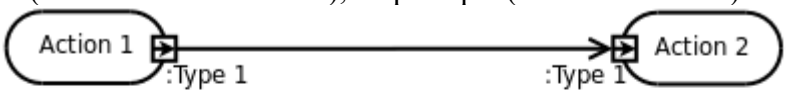


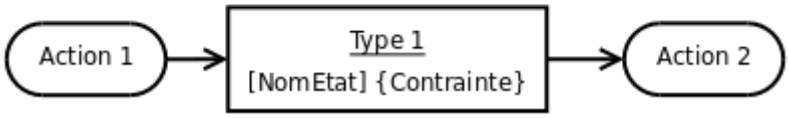
5.6. Pins et buffers (UML 2)

Depuis la version 2 d'UML , il faut placer des points ("pin") d'entrée ou de sortie sur une activité pour préciser un lien (éventuellement typé) avec les "object flow" entrant(s) ou sortant(s).
[avec une sémantique de passage de valeur(s) par copie(s)] .

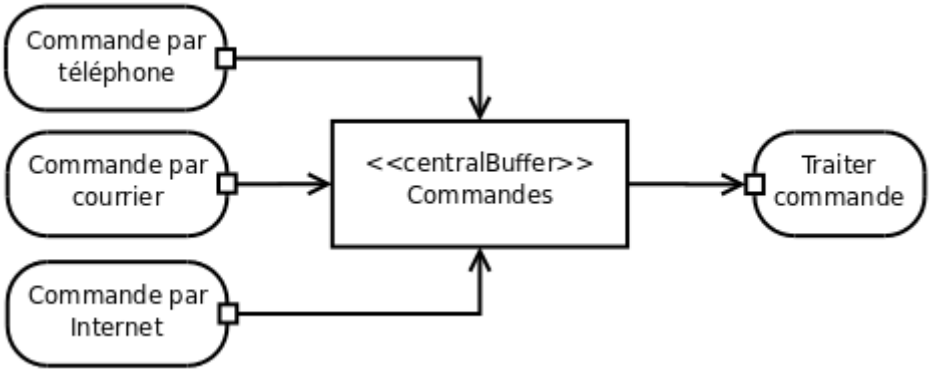


2 notations possibles (en théorie avec UML2), en pratique (selon outil UML):





En UML2, un éventuel nœud intermédiaire de type <<centralBuffer>> peut être placé pour bufferiser des messages à acheminer ensuite ailleurs.



NB: Lorsque le buffer/tampon intermédiaire gère en outre la persistance des données (dans une base de données ou ...) , on utilise alors <<dataStore>> plutôt que <<centralBuffer>> .

6. Analyse du domaine (glossaire , entités)

L'analyse du domaine est la toute première partie de l'analyse et son résultat fait partie intégrante des spécifications fonctionnelles générales. L'analyse du domaine consiste essentiellement à **définir et extraire l'ensemble des entités pertinentes** qui seront utiles au développement de l'application.

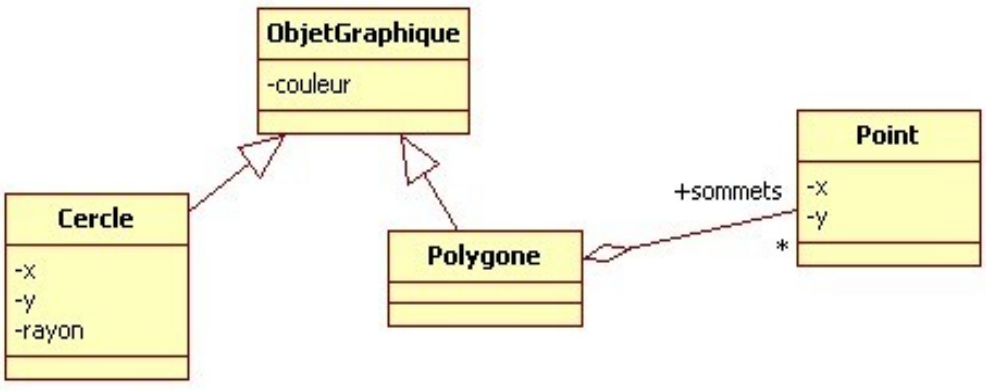
Partant d'un niveau très conceptuel (arbre sémantique, glossaire) elle permet d'aboutir à un **modèle logique de type "entités/relations"** résolument restreint à ce qui touche au domaine du système à développer (sans éléments inutiles) . Idéalement basées sur des formulations de type "chose concrète xxx est une sorte de chose abstraite yyy qui ....", les définitions d'un glossaire peuvent quelquefois suggérer des relations d'héritage (généralisation/spécialisation).

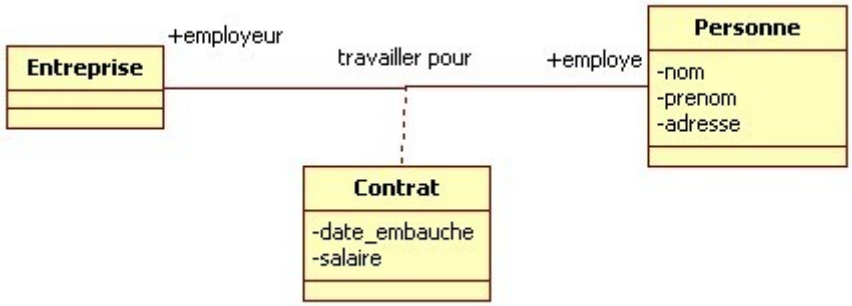
L'étape "**analyse du domaine**" est généralement effectuée de la façon suivante:

- Mise au point d'un ***glossaire décrivant les entités du domaines*** ==> tableau avec nom_entité , définition , caractéristiques_retenues .

Terme/entité *(classes)*	***Définition***	***Caractéristiques retenues*** *(attributs pertinents et relations)*
Cercle	Figure géométrique formée par l'ensemble des points situés à une distance R du centre	xc yc rayon
...	...	...

- Retranscrire ce glossaire au sein d'un **diagramme de classes** sommaire ne mentionnant que les ***classes d'entités*** avec les ***principaux attributs et les principales relations (associations, agrégation, héritages , ...)***. Ce diagramme ne doit normalement comporter quasiment aucune méthode ni classe orientée "traitements" .





Quelques remarques:

- Ne pas introduire trop tôt (dès de début de l'analyse) des détails techniques s'ils ne sont pas indispensables (ex: les types précis des données et les flèches de navigabilité sont des détails qui peuvent souvent n'être spécifiés qu'à la fin de l'analyse).
- Pour établir le glossaire , on se base sur tout ce qui existe (cahier des charges , rédaction des scénarios attachés aux cas d'utilisations, éventuelle modélisation métier, ...) et l'on cherche les noms communs (substantifs, ...) [Si lié au domaine de l'application et si données importantes à gérer/mémoriser ==> entité potentiellement utile]
- Les données utiles (liées au domaine de l'application) touchent aux aspects suivants:
  - états
  - échanges
  - descriptions
  - ....

7. Eléments structurants d'UML

Regroupements UML	*Sémantiques / caractéristiques*
Modèle	Ensemble très large regroupant tous les éléments de la modélisation d'une application (packages , diagrammes , classes , ...). Dans certains outils : éventuels sous modèles selon niveau de la modélisation (UseCaseModel , AnalysisModel , DesignModel)
Package	Regroupement significatif permettant de ranger ensemble les éléments qui appartiennent à un même domaine (fonctionnel et/ou technique).
Diagramme	Simple vue graphique représentant quelques éléments d'un modèle et leurs relations (le découpage en différents digrammes est purement pragmatique : selon la place)

7.1. Modèle

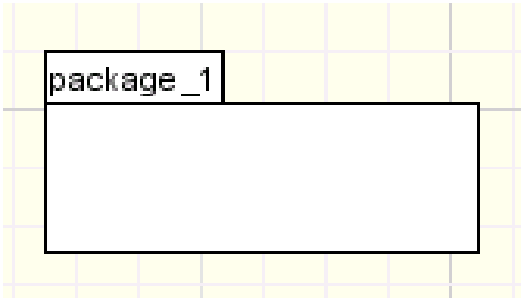
Correspondant généralement à une application entière (ou bien à un sous système) , un **modèle UML** est essentiellement constitué par une **arborescence d'éléments** .

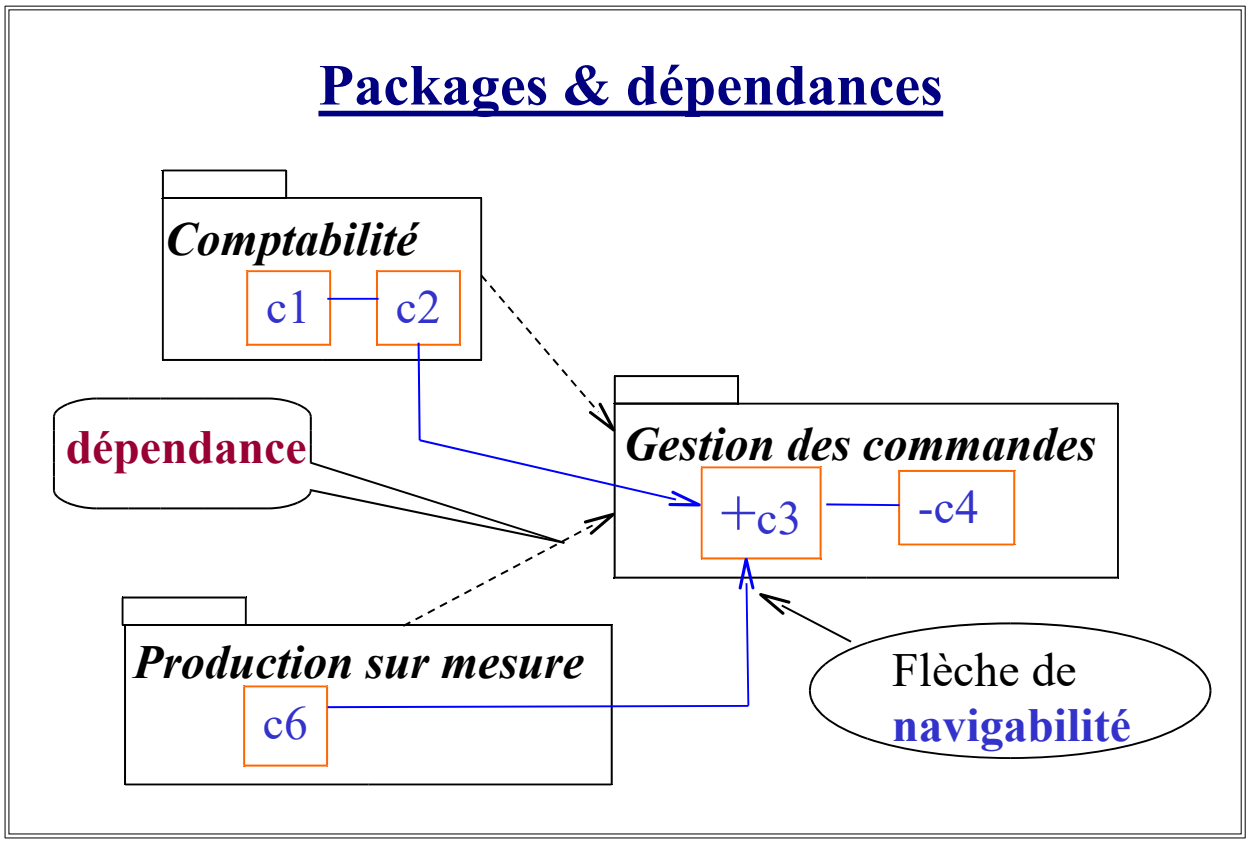
C'est à partir de son contenu (*xy.uml* ou *xy.xmi*) que l'on pourra éventuellement générer du code via MDA.

7.2. Packages

- Un **package** est un **regroupement d'éléments du modèle**. (un package peut contenir des classes, des relations , des sous-packages, …)
- **Cohérence fonctionnelle**.
- Correspondance avec la notion de dossier, de package Java et de namespace en C++.

Notation:





- La **navigabilité** entre C2 et C3 indique que la classe **C2** du package «comptabilité» utilise la classe **C3** du package «Gestion des commandes» (et pas dans l'autre sens).
- Lorsqu'au moins une classe d'un package (P1) utilise une classe d'un autre package **(P2), on dit que le package P1 est dépendant du package P2**.
  Conséquence: Une mise à jour importante du package P2 nécessitera souvent des modifications au niveau du package dépendant P1.
- Pour que l'ensemble soit ***facile à maintenir***, **il faut essayer de minimiser les dépendances entre les packages**.

NB: lorsqu'un élément UML est représenté dans un diagramme UML associé à un autre package , son package est alors signalé via ***{ from packageXY }*** .

7.3. Diagrammes

Un diagramme est une vue graphique (avec une syntaxe normalisée en UML) permettant de représenter quelques éléments d'un modèle UML.

NB:
- Un même élément (ex: classe) peut apparaître sur plusieurs digrammes.
- La plupart des outils permettent d'ajouter dans un diagramme un élément déjà existant via un simple glisser/poser partant le l'arborescence du modèle.
- Lorsque l'on souhaite supprimer un élément que sur le diagramme courant --> Suppr/Delete
- Lorsque l'on souhaite supprimer un élément dans tous les diagrammes et dans le modèle ---> ***Edit / Delete From Model*** .

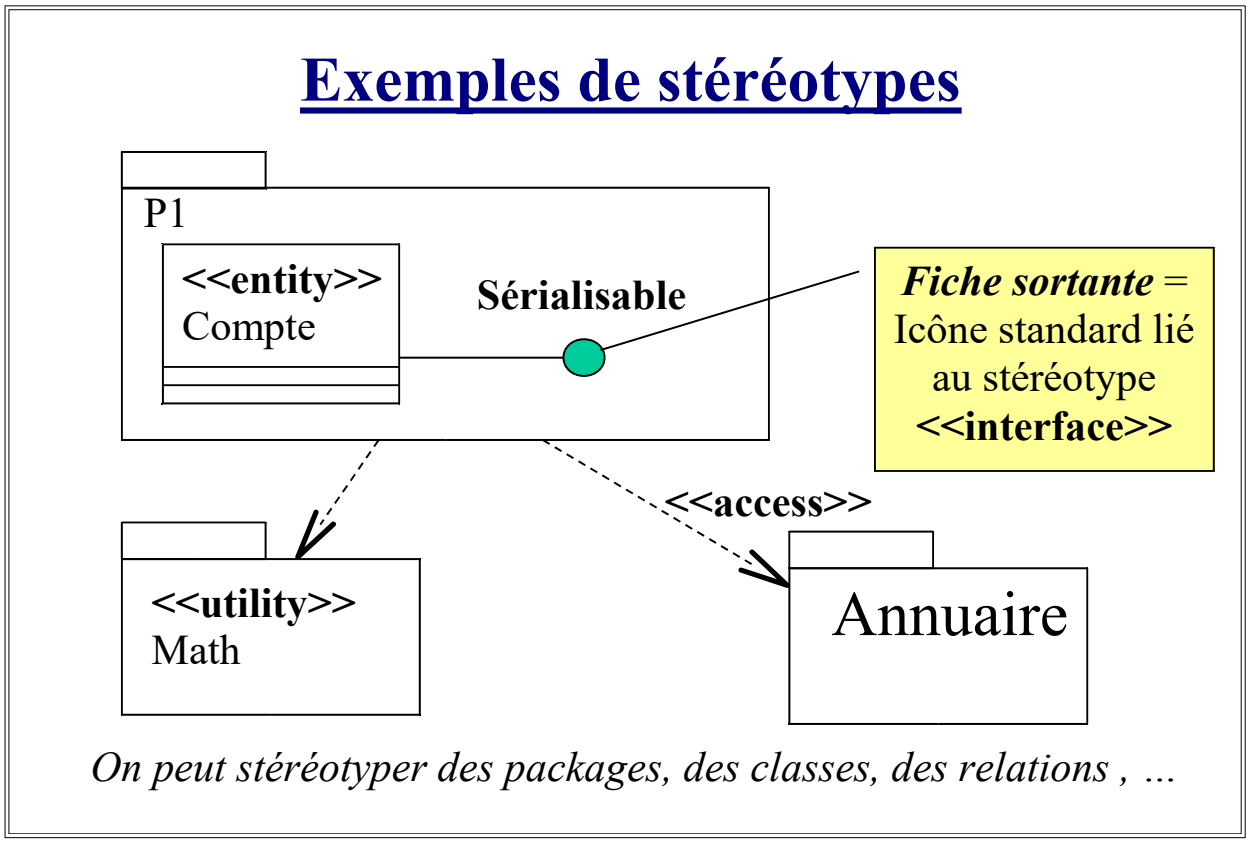
7.4. Stéréotypes

- Un **stéréotype** est une **nouvelle information** (supplémentaire à la classe ou une relation ou ...) permettant de **préciser la nature** d'une famille **d'éléments de la modélisation** UML.

- **Un stéréotype est une extension vis à vis des bases d'UML**. Ceci permet d'introduire de nouveaux concepts. **UML est ainsi ouvert** vers de nouvelles notions.

Notation:
On peut **soit encadrer le nom du stéréotype par << et >>**, soit inventer un **nouvel icône personnalisé** lié à un stéréotype.

NB: Les versions récentes d'UML autorise des stéréotypes multiples -->
<< stéréotype1, stéréotype2 , ... >>



Quelques exemples de stéréotypes:

<<entity>> , <<service>> , <<id>> , <<utility>> , <<enumeration>> , <<interface>> , ...

Selon l'outil UML utilisé, un stéréotype peut être:
- soit créé à la volée pour être utilisé immédiatement
- soit préalablement créé (parmi d'autres) au sein d'un profile UML pour être ensuite sélectionné.

7.5. Valeurs étiquetées (Tag Values)

Une valeur étiquetée (tag value) est une information supplémentaire de la forme
{ **nomPropriété = valeur** } que l'on peut ajouter sur n'importe quel élément d'un modèle UML.

Les "tag values" ne sont généralement pas visibles dans les diagrammes UML.
Ces valeurs cachées ne sont donc utiles que si elles sont ultérieurement analysées par un programme quelconque (générateur de documentation, générateur de code MDA, ...).

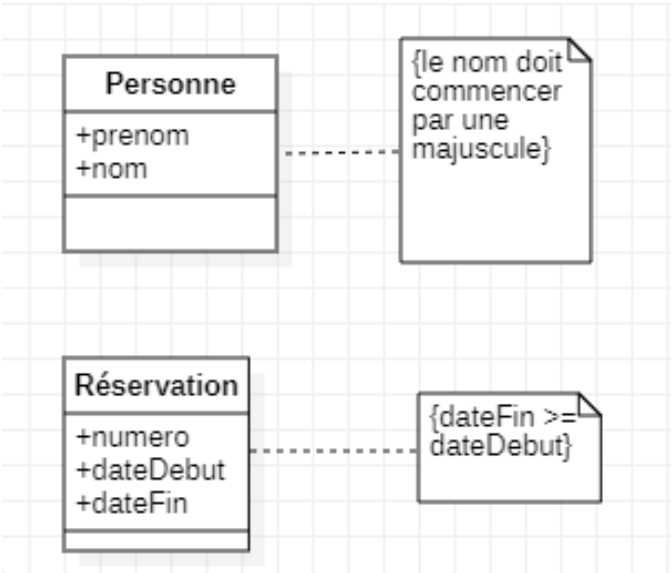
Exemples de valeurs étiquetées: auteur=didier , withDTO=true, version=V1

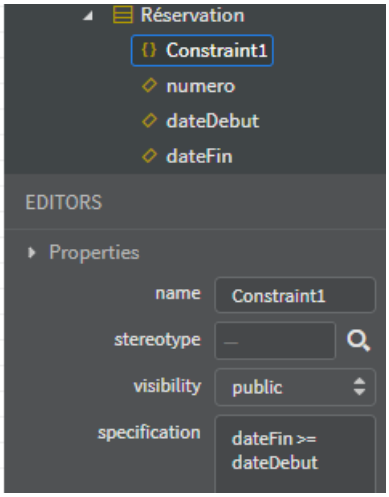
7.6. Contraintes

- Les **contraintes** servent à exprimer **des situations ou conditions qui doivent absolument être vérifiées** et que l'on ne peut pas exprimer simplement avec le reste des notations UML.
- Elles peuvent être exprimées en **langage naturel** ou bien via le langage spécifique **OCL** (Object Constraint Language).
- Elles peuvent s'appliquer à tous les éléments de la modélisation.
- ***Il existe quelques contraintes prédéfinies*** : **{ xor }** , **{ readonly }** , **{ frozen }** , { overlapping }, …

Syntaxe: **{ texte de la contrainte }** , *exemple*: **{ age > = 0 }**

Exemples de contraintes (ici placées dans des commentaires/notes) :





NB:
- Beaucoup d'outils UML sont capables de paramétrer des contraintes mais ils ne les affichent pas. Elles restent souvent invisibles dans les diagrammes.
- Le langage **OCL** est assez complexe (et n'est pas pris en charge par tous les outils UML). Il a néanmoins le mérite d'être assez formel (utile pour de la génération de code).

8. Diagramme de classes (notations , ...)

Diagramme de classes *(modèle structurel)*

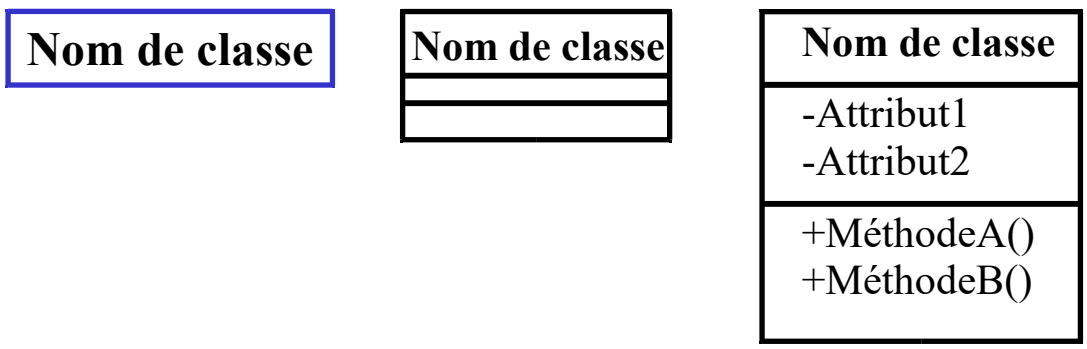
• Le **modèle statique/structurel** (basé sur les diagrammes de classes et d'instances) permet de décrire la **structure interne du système** (**entités existantes**, **relations entre les différentes parties**, …).

• Tout ce qui est décrit dans le diagramme de classes **doit être vrai tout le temps**, il faut raisonner en terme d'**invariant**.

• Le **diagramme de classes** est le plus important, il représente **l'essentiel de la modélisation objet** (c'est à partir de ce diagramme que l'essentiel du code sera plus tard généré).

Les classes (représentations UML)

Une **classe** représente un **ensemble d'objets** qui ont :
- une **même structure** (*attributs*)
- un **même comportement** (*opérations*)
- les **mêmes collaborations avec d'autres objets** (*relations*)



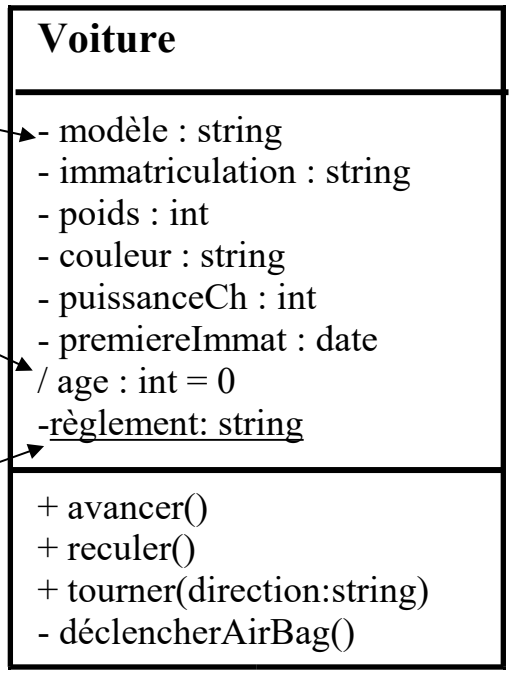
Attribut alias ***Propriété/Property*** , *Méthode* alias ***Opération*** .

Détails sur les éléments d'une classe

Visibilité
+ : ***public***
- : ***private***
: ***protected***

Le symbole / désigne un **attribut calculé** dit **dérivé** (implémentation: fonction getXXX() et/ou valeur en mémoire cache)

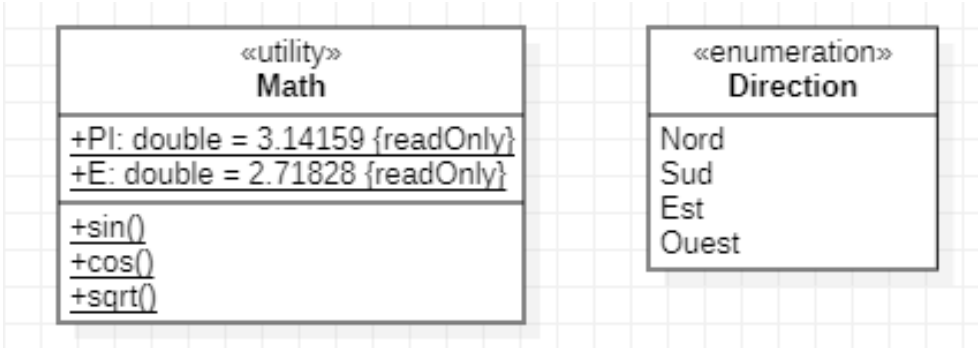
Un attribut souligné (ou préfixé par un **$**) est une **variable de classe** (***static*** en java et c++)



Classes spéciales (utilitaires, énumération, …)

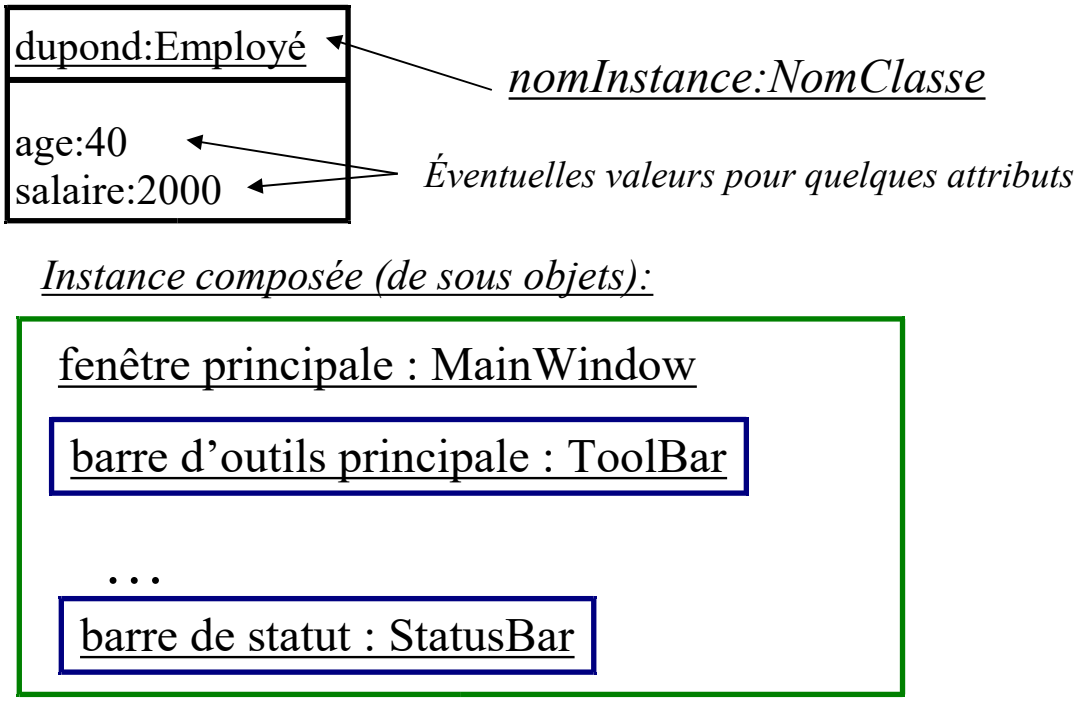
• Dans un monde matériel et rationnel ou tout est objet, il n'y a plus de place pour des fonctions globales (anarchiques).
Celles-ci doivent être rangées dans des classes utilitaires.

• Les constantes doivent elles aussi être placées dans des classes (ou interfaces) d'énumération.



y = **Math.sqrt**(x); obj.setDirection(**Direction.Nord**);

Eventuels (et rares) diagrammes d'instances



Associations (relations)

Dans le cas le plus simple une **association** est **binaire** et est indiquée par un *trait reliant deux entités (classes)*. Cette association comporte généralement un nom (souvent un verbe à l'infinitif) qui doit clairement indiquer la signification de la relation. ***Une association est par défaut bidirectionnelle***.

Dans le cas où le sens de lecture peut être ambigu, on peut l'indiquer via un triangle ou bien par le symbole >

Extrémités d'association

• Une **association** n'appartient pas à une classe mais à un package (celui qui englobe le diagramme de classe).

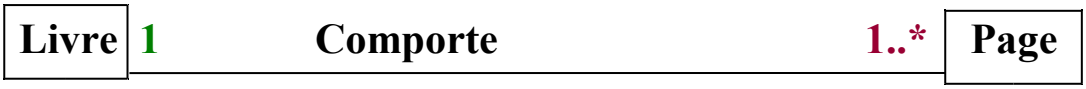
• On peut préciser des caractéristiques d'une association qui sont liées à une de ses **extrémités (association end):**
- ***rôle*** joué par un objet dans l'association
- ***multiplicité***
- ***navigabilité***
- …

Multiplicité UML (~~cardinalité~~)

1	exactement un
0..* ou *	**plusieurs** (éventuellement zéro)
0..1	zéro ou un
n (ex: 2)	**exactement n (ex: 2)**
1..*	**un à plusieurs** (au moins 1)

Les **multiplicités** permettent d'indiquer (pour chacune des classes) les nombres minimum et maximum d'instances mises en jeu dans une association.

Interprétation des multiplicités:

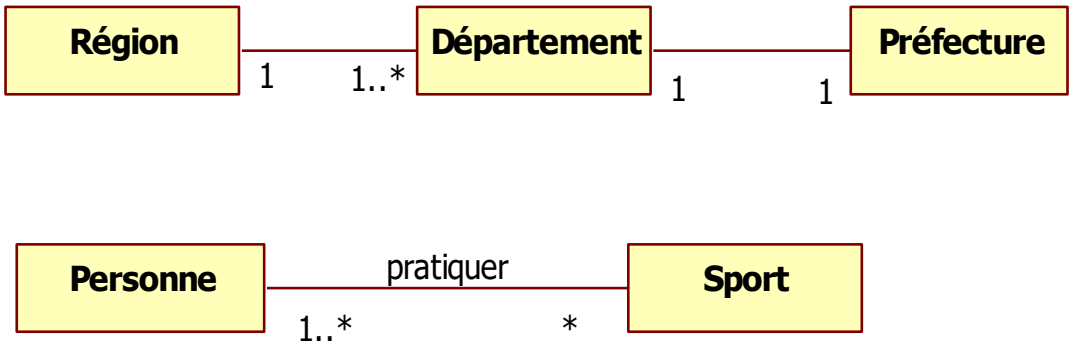


1 livre comporte ***au moins une*** *page*

et

une page de livre se trouve dans ***un et un seul*** *livre.*

Multiplicités (exemples)



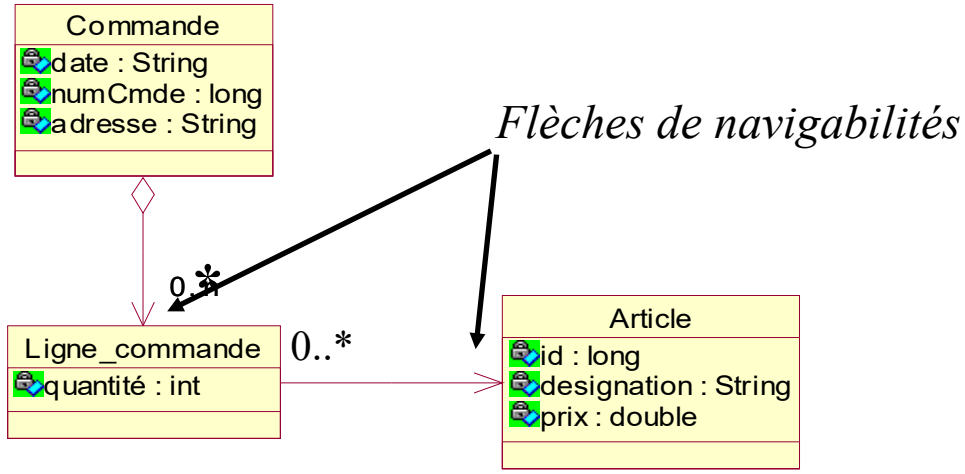
NB:

• Les multiplicités d'UML utilisent des notations inversées vis à des cardinalités de Merise.

• ***Les multiplicités dépendent souvent du contexte*** (système à modéliser).

Navigabilité

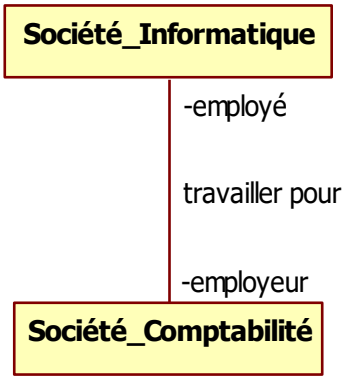
Une **flèche de navigabilité** permet de restreindre un accès par défaut bidirectionnel en un **accès unidirectionnel** ***plus simple à mettre en œuvre et engendrant moins de dépendance***.



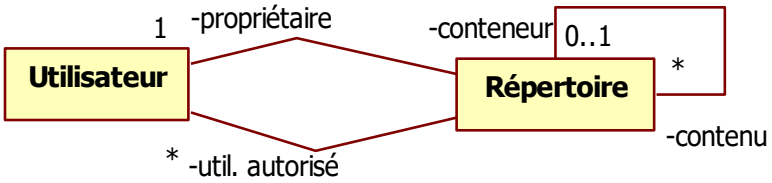
*Un article **n'a pas directement accès** à une ligne de commande*

Rôles

Les **rôles** (facultatifs) permettent d'indiquer le rôle joué par chaque entité dans le cadre d'une association.



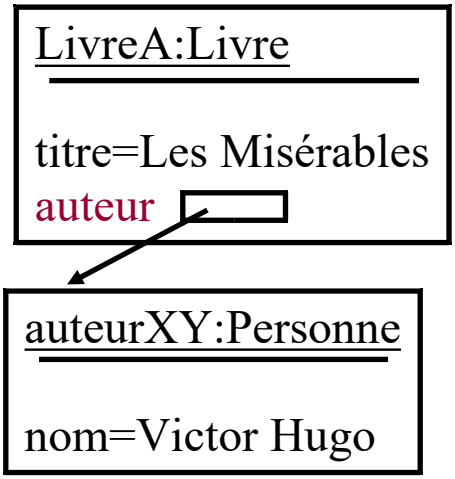
Ils peuvent servir à lever certaines ambiguïtés:



Rôles & implémentation

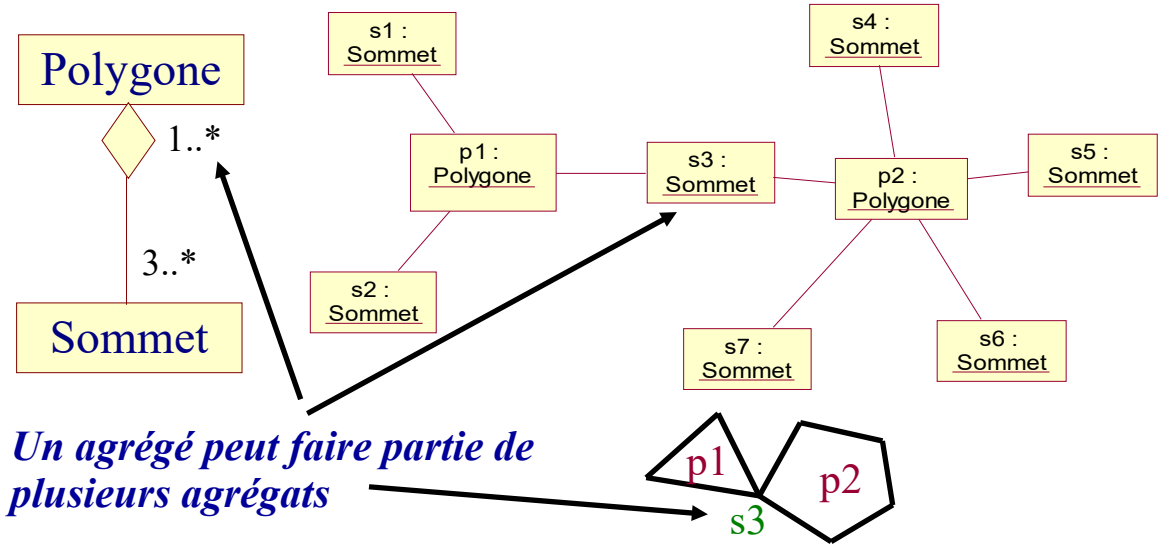

```
class Livre
{
private String titre;
private Personne auteur;
…
}
```

Les noms des rôles sont souvent utilisés pour nommer les références.



Agrégation

Un agrégat est composé de plusieurs sous objets (les agrégés). Cette relation particulière et très classique est symbolisée par un **losange** placé du coté de l'agrégat.



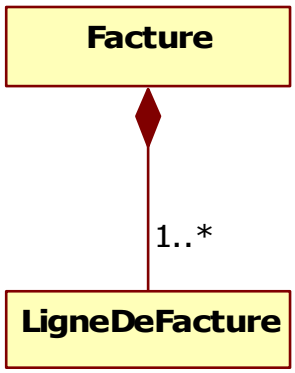
Agrégation (caractéristiques)

• Une **agrégation** est une association de type "**est une partie de**" qui vu dans le sens inverse peut être traduit par "**est composé de**".

• UML considère qu'une **agrégation est une association bidirectionnelle ordinaire** (le losange ne fait qu'ajouter une sémantique secondaire).

• Une agrégation (faible) ordinaire implique bien souvent que les sous objets soient **référencés** par leur(s) agrégat(s).

Composition (agrégation forte)



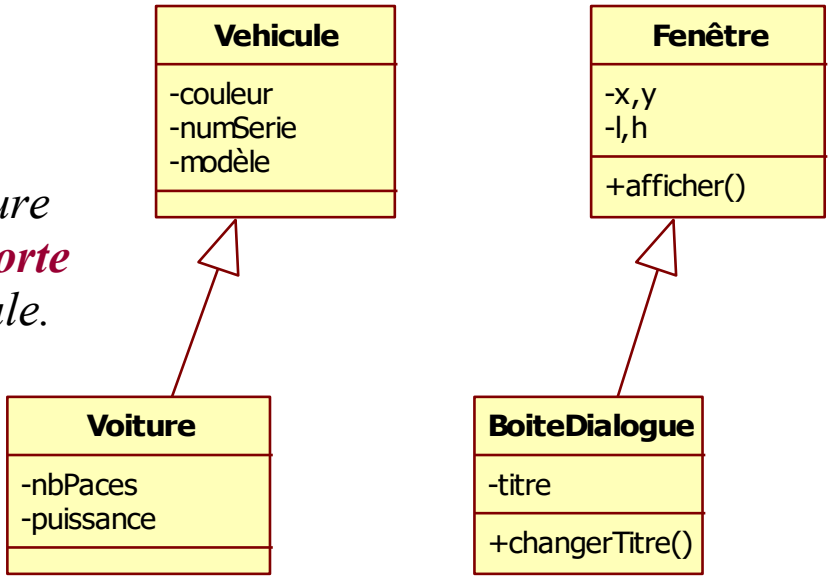
En général, le sous-objet n'existe que si le conteneur (l'agrégat) existe: **lorsque l'agrégat est détruit, les sous objets doivent également disparaître**.
("cascade-delete" en base de données).

Dans une agrégation forte, un sous objet ne peut appartenir qu'à un seul conteneur.

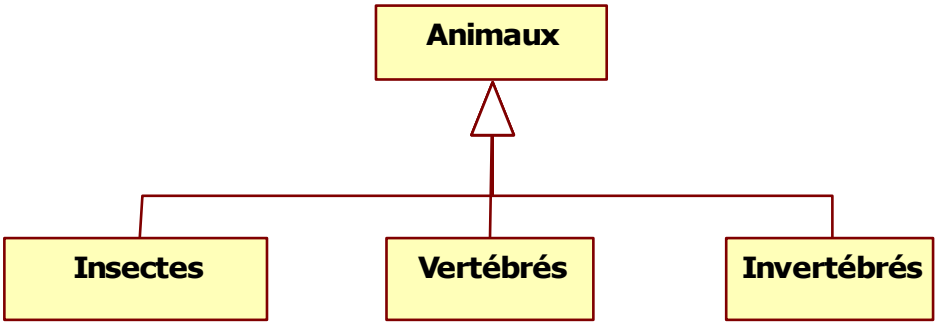
Remarque: Le **losange est quelquefois rempli de noir** pour montrer **que le sous objet est physiquement compris dans le conteneur** (l'agrégat). On parle alors d'agrégation forte (véritable **composition**).

Généralisation (héritage)

*Une voiture **est une sorte de** véhicule.*



Classification (généralisation)

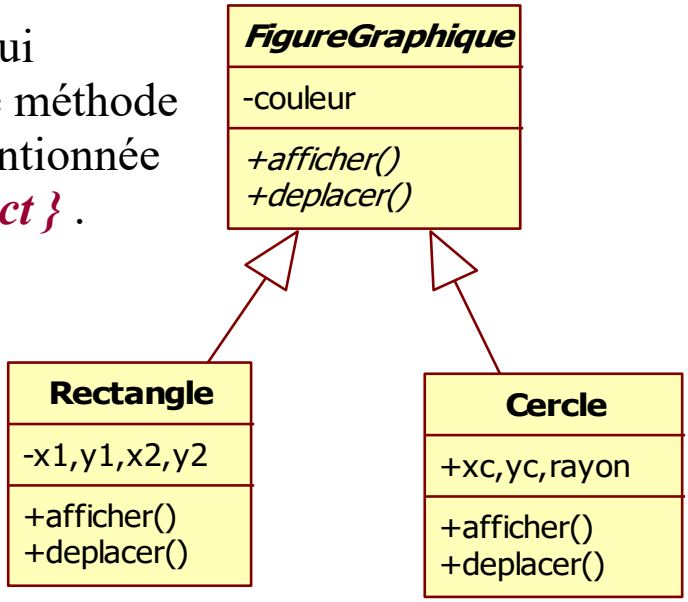


Les animaux peuvent être classées dans divers **groupes** (et sous groupes).

Classification selon caractéristiques discriminantes.

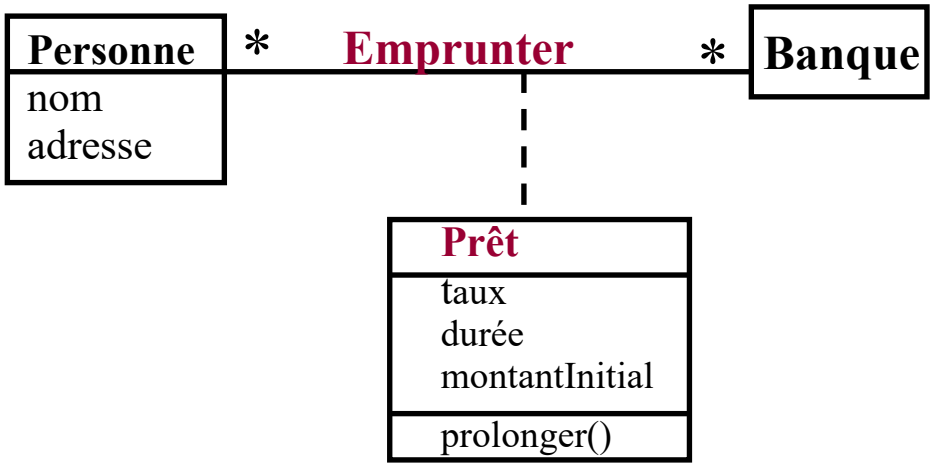
Classes abstraites et concrètes

Une ***classe abstraite*** (qui comporte au moins une méthode sans code) doit être mentionnée en ***italique ou { abstract }*** .



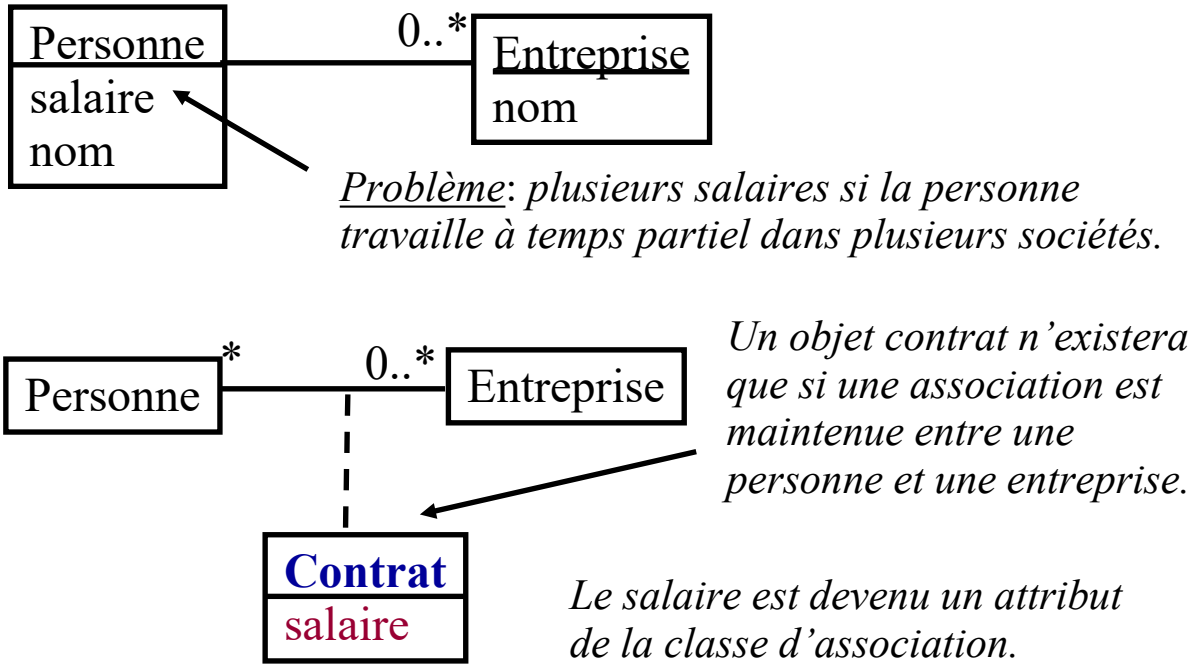
Classes d'association

Lorsqu'une **association comporte des données ou des opérations qui lui sont propre** (non liés à seulement une des entités mises en relation), on a souvent recours à des **classes d'association** .



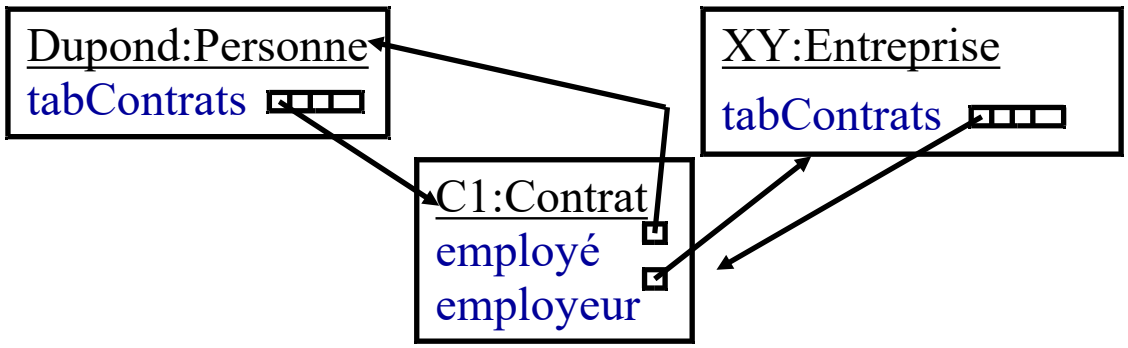
Remarque : Les classes d'association se transposent très bien dans le modèle relationnel comme des tables d'associations (avec au moins deux clefs étrangères)

Classes d'association (exemples)



Au sein d'une vision "programmation objet" (par exemple java) , une classe d'association UML sera assez souvent transposée en une classe de liaison (n-n => n-1 + 1-n) :

Implémentation des classes d'associations

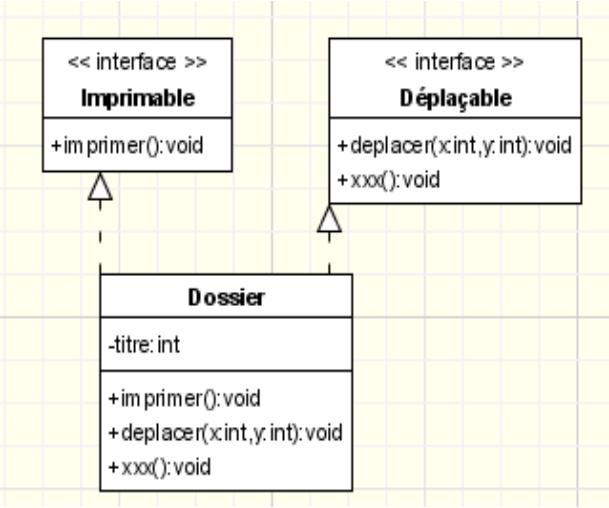


Un objet contrat devient un intermédiaire permettant d'accéder à chacune des entités de l'association:
Contrat1.getEmploye(); Contrat1.getEmployeur();

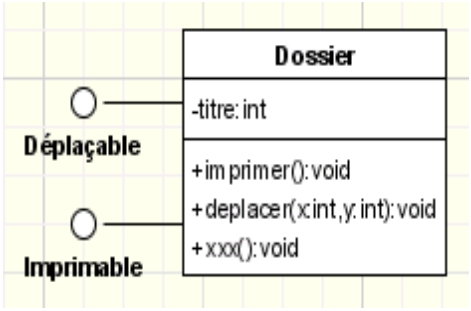
Interface

Une **interface** est une **classe abstraite** qui ***ne contient que des opérations génériques sans code***. Une interface est une simple collection de prototypes (signatures) de méthodes.

Pour être utile, une interface doit évidemment être entièrement codée au niveau d'une classe concrète.



NB: *Une **interface** est vue comme un **type** de données **abstrait***



V - Analyse applicative et conception avec UML

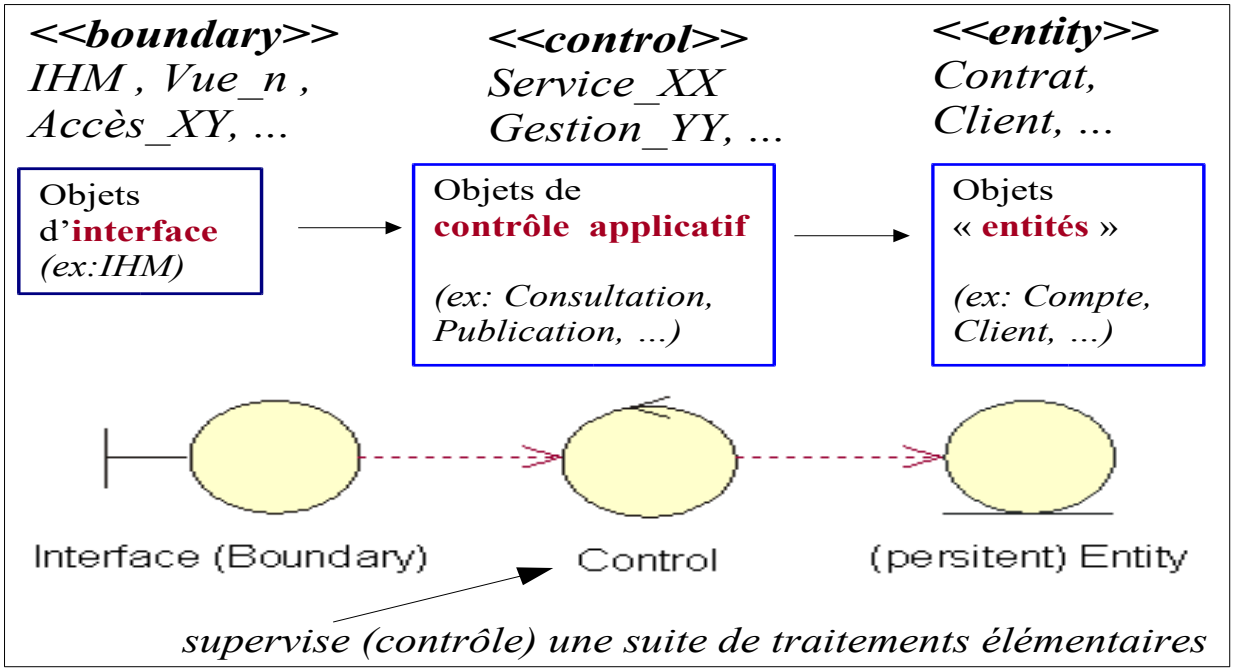
1. Analyse applicative (objectif et mise en oeuvre)

L'analyse applicative est la seconde grande phase de l'analyse (après celle du domaine).
De façon à enfin aboutir à un modèle réellement orienté objet (avec données et traitements assemblés), la phase d'analyse applicative vise essentiellement à compléter l'analyse du domaine (plutôt orientée "données") en introduisant des éléments fonctionnels (traitements/services "métiers" et "ihm"). Etant donné que cette phase est assez délicate (gros travail à mener méthodiquement), il est généralement recommandé de procéder de la façon suivante:

- **Identifier toutes les classes nécessaires** (avec les stéréotypes d'analyse <<entity>> , <<control>> ou <<service>> et <<boundary>> ou <<ihm>>).
- établir un (ou plusieurs) **diagramme(s) de classes montrant les classes participantes**

- Pour chaque "***Use Case***" identifié :
  - ***retranscrire le scénario nominal*** sur un nouveau ***diagramme de séquence uml montrant les envois dynamiques de messages entre les objets identifiés de l'analyse applicative***.
  - montrer sur certains diagrammes de classes la liste des ***opérations (méthodes) ajoutées*** sur les ***classes qui réalisent (par collaboration) les fonctionnalités d'un cas d'utilisation***.

Stéréotypes d'analyse de Jacobson :



==> rien d'interdit d'utiliser des stéréotypes plus significatifs ou plus dans l'air du temps tels que par exemple:

<<service>> à la place de <<control>>
<<ihm>> ou **<<proxy>>** à la place de <<boundary>>

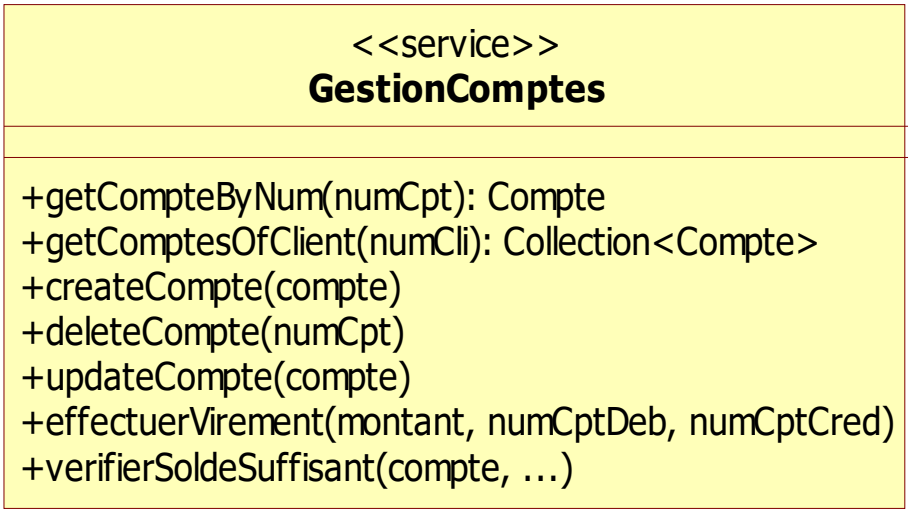
2. Responsabilités (n-tiers) et services métiers

Eléments d'une application	*Responsabilités*
Vues IHM	Afficher , Saisir , Choisir/Sélectionner , Déclencher , Confirmer , ...
Services métiers	Objets de traitements ré-entrants (partagés entre les différents utilisateurs) et apportant les services nécessaires au fonctionnement de l'application . Méthodes souvent transactionnelles (rollback en cas d'erreur , commit si tout se passe bien)
Entités (souvent persistantes)	Mémoriser (en mémoire et en base de données) toutes les informations importantes du domaine de l'application

Principales méthodes d'un service métier:

- **Opérations "C.R.U.D."** (Create , Retreive, Update, Delete)
- Méthodes de **vérifications** (liées à des règles de gestion)
- **Autres méthodes métiers** (ex: effectuerVirement() , ...)

Exemple (service métier "GestionComptes"):



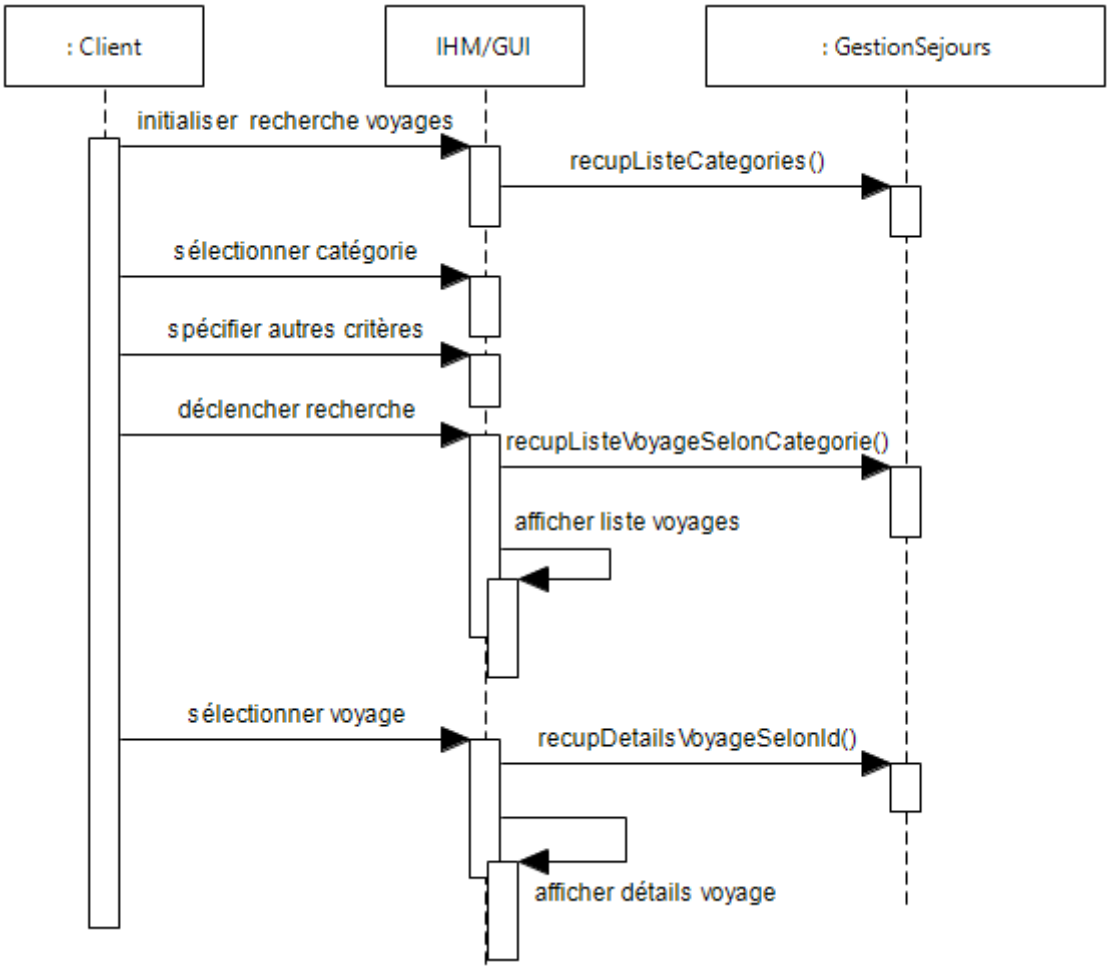
Pour identifier les objets de contrôles applicatifs / services métiers on peut se baser sur les compléments d'objets directs des U.C. ou bien sur les noms des packages .

On peut également se baser sur les entités les plus importantes ==> "***ServiceXxx***" ou "***GestionXxx***" avec stéréotype ***<<control>> ou <<service>>***

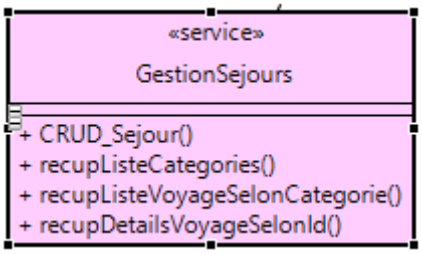
3. Réalisation des cas d'utilisations

Procédure à suivre:

- **Retranscrire les scénarios attachés aux U.C. en des diagrammes UML d'interactions** (séquence, collaboration/communication, ...) .
- **Enrichir les diagrammes de classes** (nouvelles méthodes = messages reçus)



cohérent avec

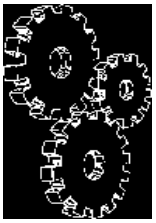


Conseil : ne pas faire apparaître les classes de type <<entity>> dans les diagrammes de séquence de niveau analyse car la sous séquence exacte/réalisable dépend des choix technologiques (conception) .

4. Modèle dynamique – diagrammes d' interactions

Modèle dynamique (UML)

Le modèle **dynamique** vise à représenter les **comportements** des objets du système et leurs **collaborations**. Ce modèle est ***complémentaire*** vis à vis du modèle statique (il est généralement élaboré en parallèle).
Le modèle dynamique est basé sur **deux grandes sortes de diagrammes**:



- Des **diagrammes d'interaction:**
  - * **séquences**
  - * **collaboration/communication**

- Des **automates**:
  - * **diagrammes d'états**
  - * **diagrammes d'activités**

5. Diagramme UML de Collaboration / Communication

Diagramme de ~~collaboration~~ *communication*

Un diagramme de communication organise certaines instances dans l'espace (avec des liaisons) et montre certaines interactions (messages numérotés).

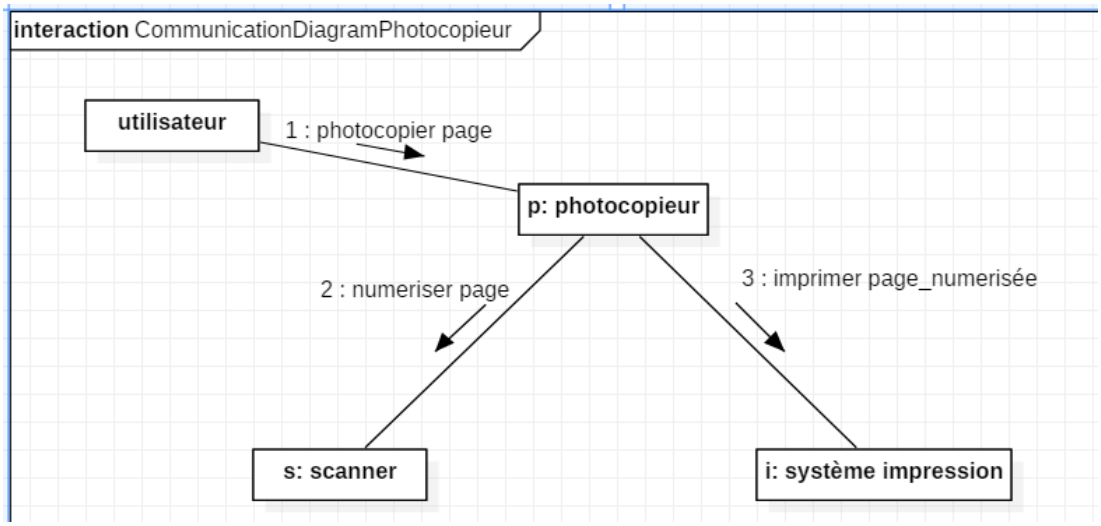
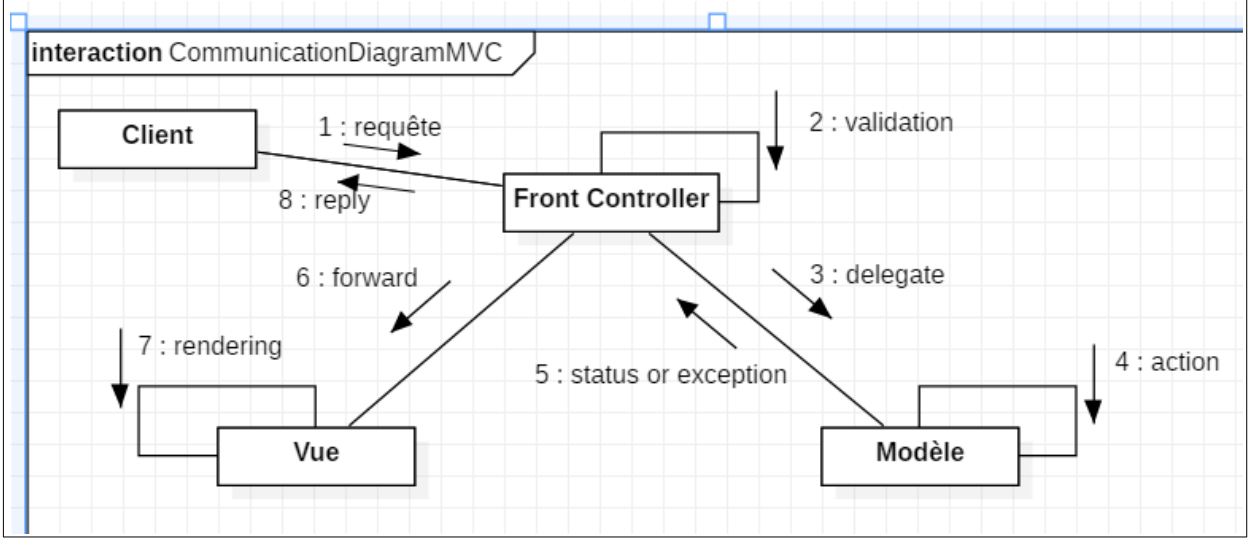
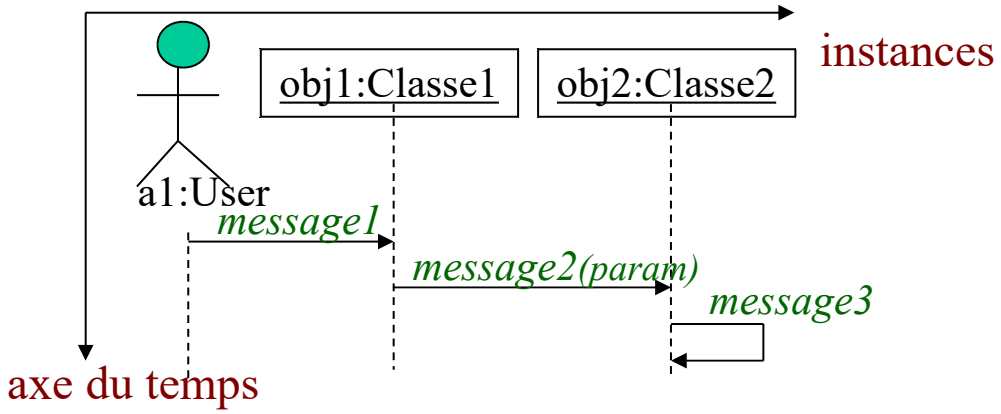


Diagramme de communication UML = ***pratique pour illustrer certains principes de fonctionnement***
(ici : le design pattern *MVC* : *Model-Vue-Controller*)



6. Diagramme de séquences (UML)

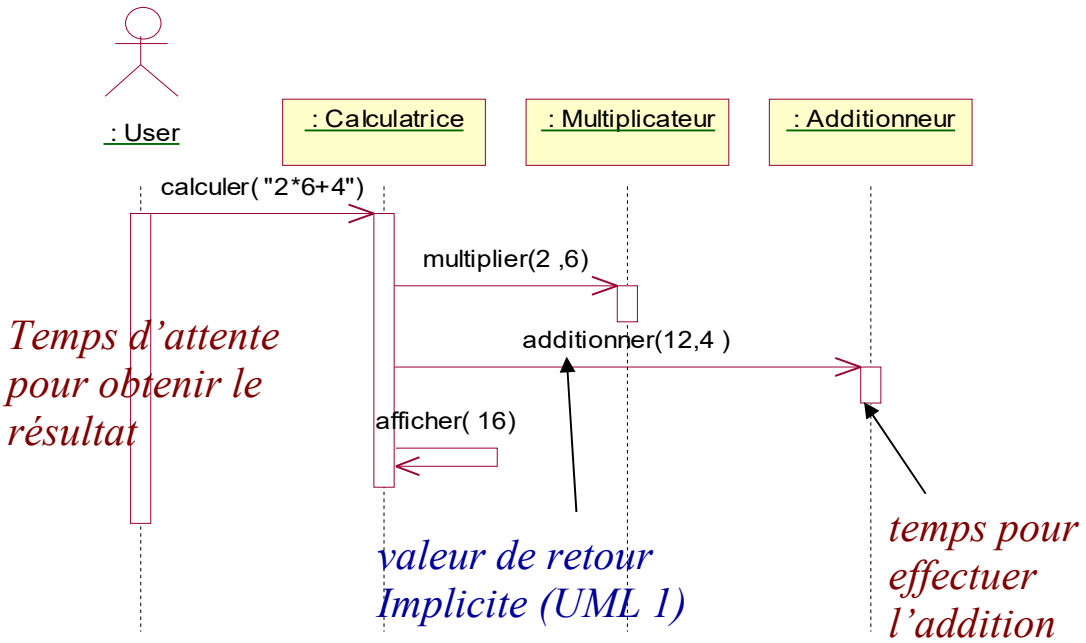
Diagramme de séquences



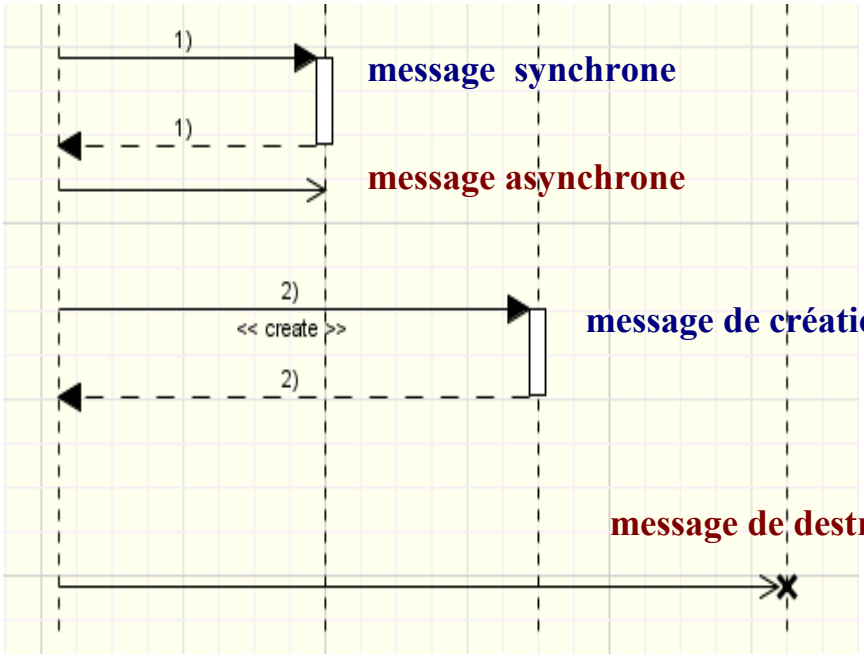
• Un **diagramme de séquence** montre un ensemble de messages échangés entres des objets pour remplir une fonctionnalité donnée ou pour obtenir un résultat.

• **Les interactions sont organisées dans le temps**.

Contrôle du focus (UML 1)

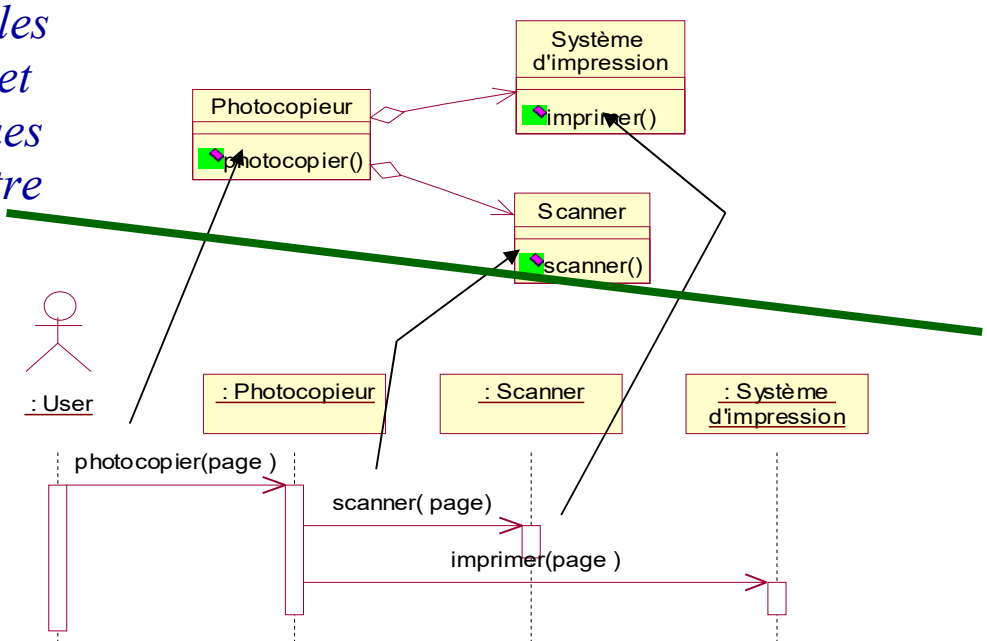


Différents types de messages (UML2)



Cohérence entre les digrammes

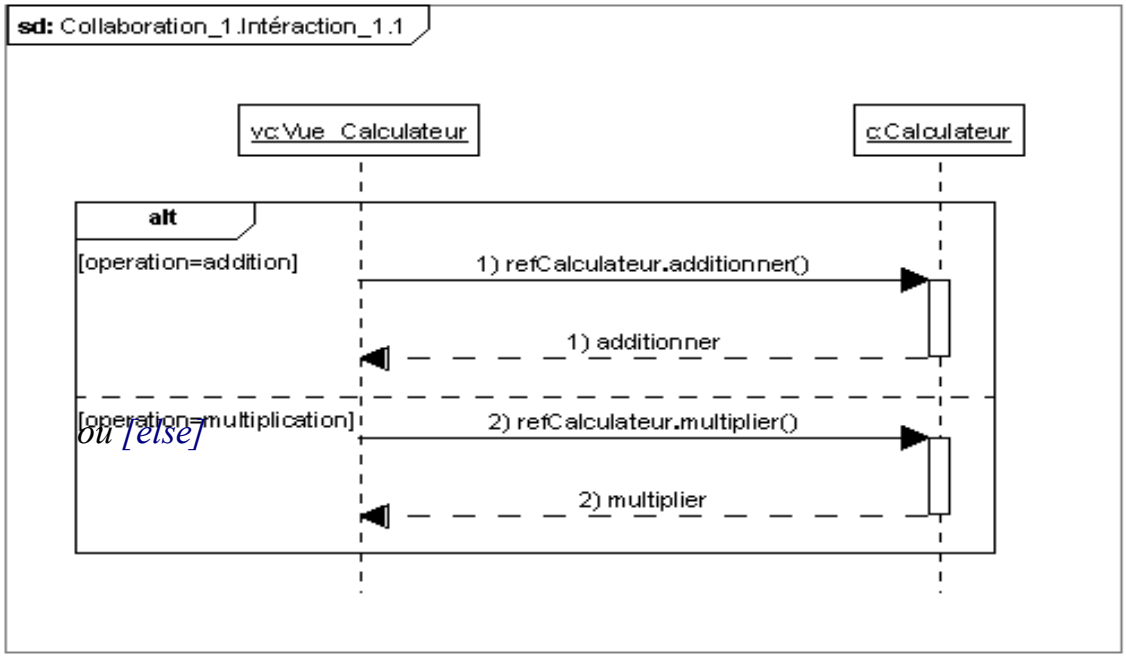
Les modèles statiques et dynamiques doivent être cohérents



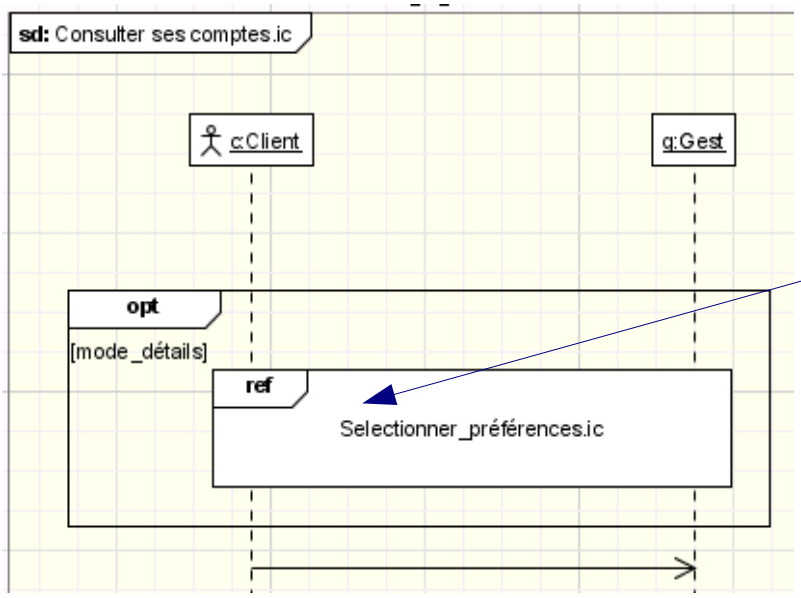
Message reçu = méthode de même nom dans la classe

7. Diagramme de séquences (notations avancées)

Alternative (**alt**) de UML2 --> *Si [...] Alors ... Sinon ...*
ou Si [...] alors ....Sinon (Si [...] alors ...) Sinon ... / Choice

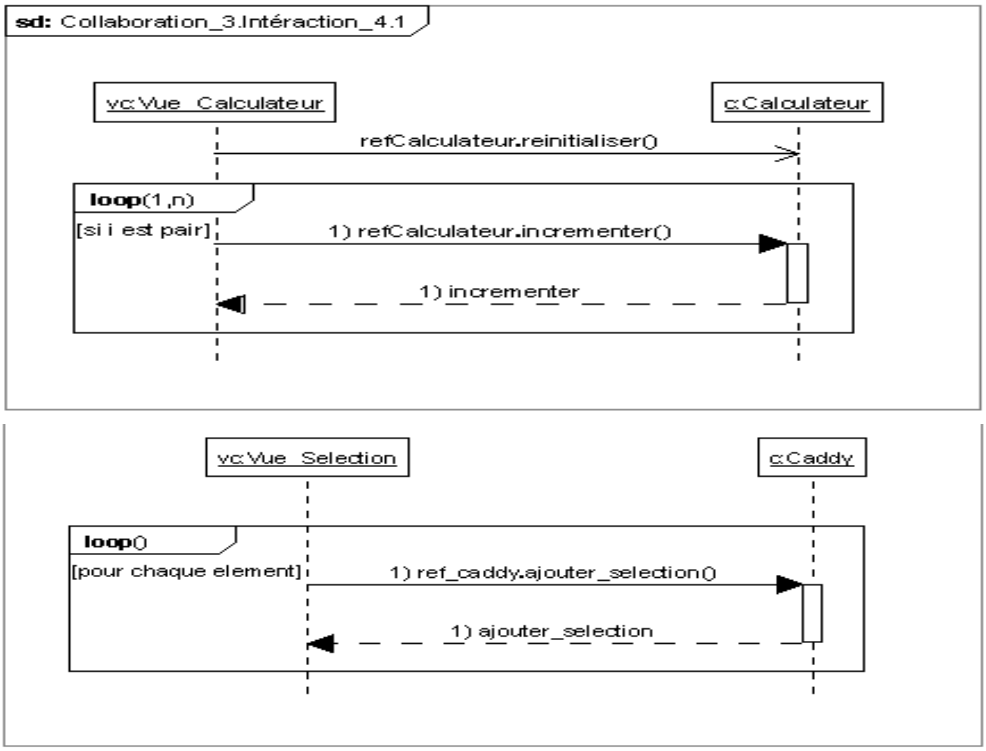


option conditionnée UML2 (**opt**)
---> ***Si*** *[...]* ***Alors*** *(sans else)*

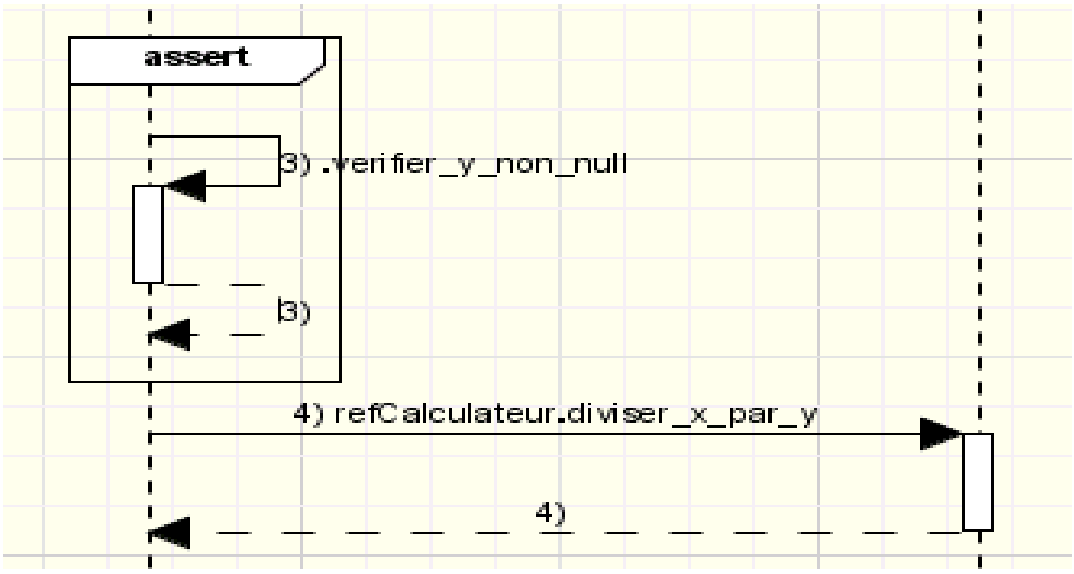


référence
vers (autre/sous)
interaction

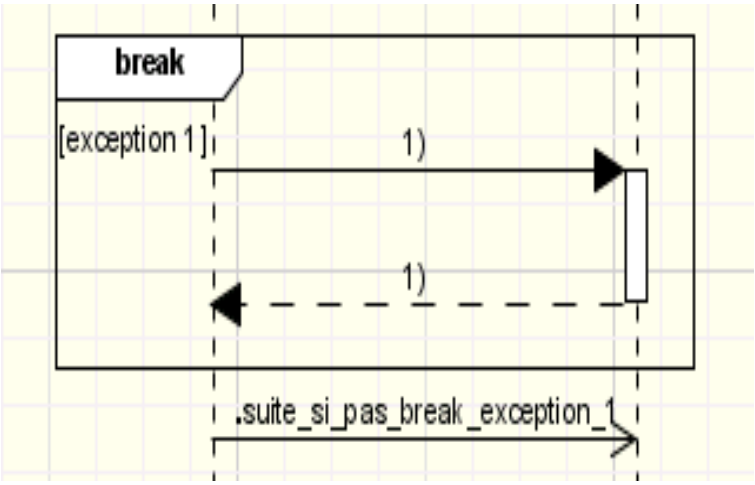
Boucle / Loop (UML2)



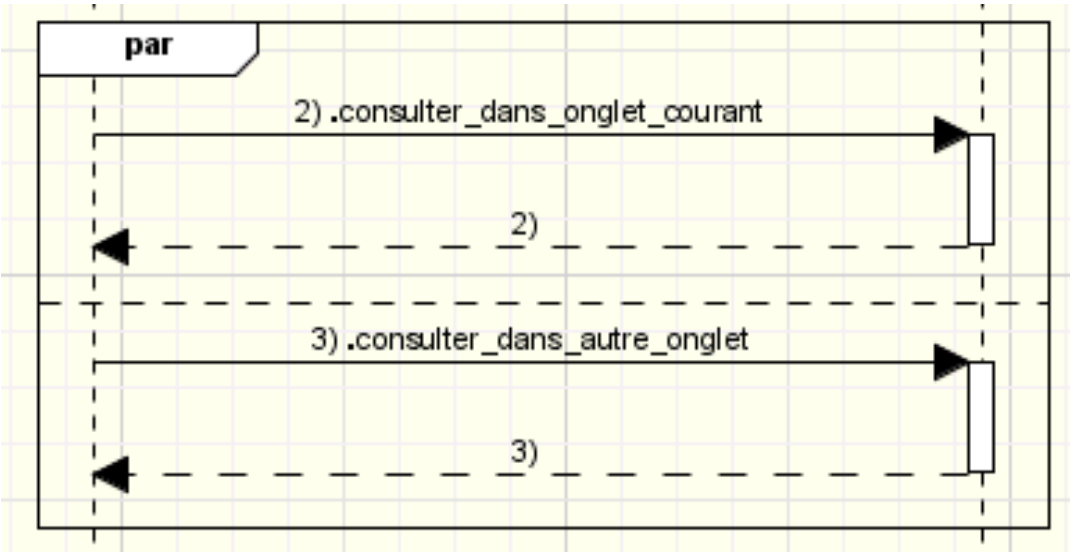
assertion UML2 (**assert**) --> séquence de tests à absolument vérifier pour pouvoir continuer



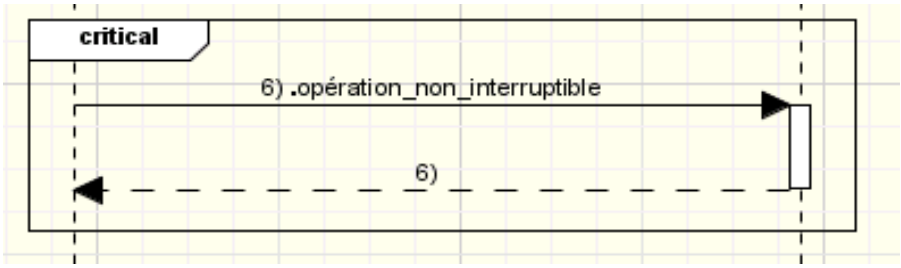
break (UML2) --> traitement en cas d'exception
(***sortie de la séquence normale***, la suite n'est pas exécutée)



par (en parallèle) UML2



crit (section critique) UML2 --> interaction que l'on ne peut pas interrompre



neg (négation) UML2 --> ***séquence invalide***
(pour montrer ce qu'il ne faut surtout pas faire)!

séquence lâche (**seq**) ou stricte (**strict**) UML2
---> ordre imposé ou pas sur certaines sous tâches

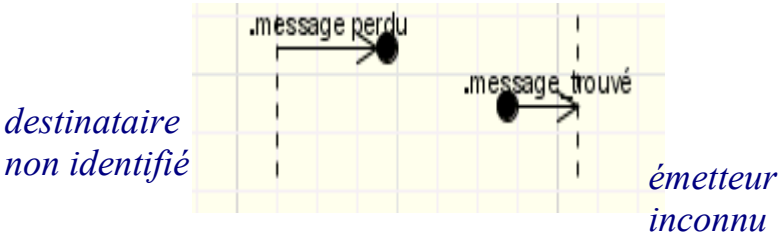
consider { message_important1, m2}

ignore { message_insignifiant_1 , ...}

Autres détails des diagrammes de séquence

On peut indiquer (en texte clair ou commentaire selon le produit):

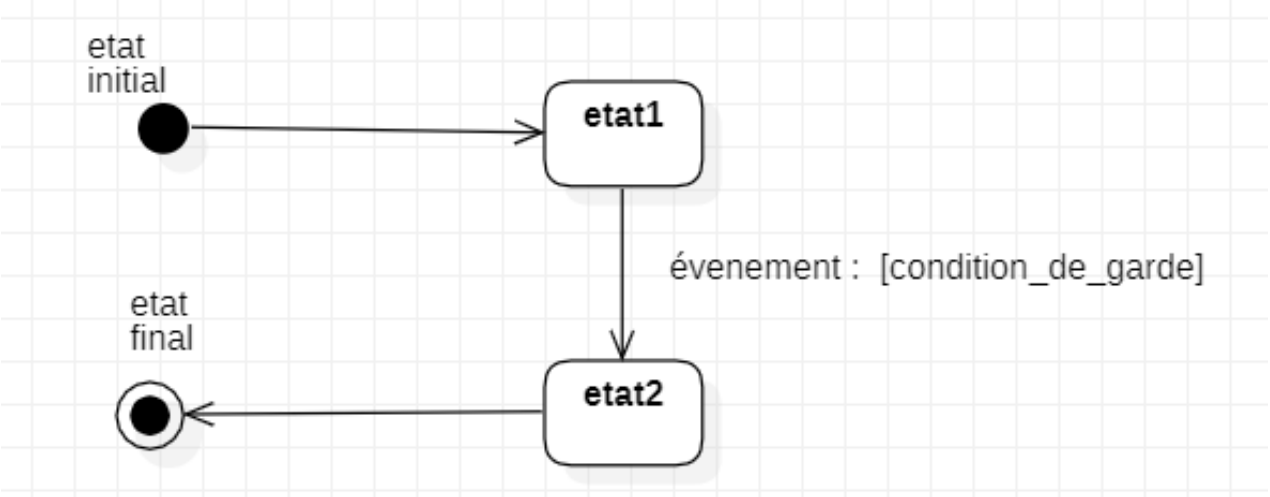
- des *conditions à vérifier* pour envoyer un message: [x > 0]
- des *contraintes*: **{ ... }**
- des indications sur la durée de vie des objets :
  new (instanciation) ,
  delete ou **X** (destruction).



…

8. Diagramme d'états et de transitions (StateChart)

Principales notations (graphe d'états)



NB:
• Chaque **état** est représenté par un **rectangle aux coins arrondis**.
• Un ***nom d'état*** est potentiellement un ***adjectif qualificatif*** (ex : *éteint* , *allumé*)
• Un état complexe peut éventuellement être décomposé en sous états (généralement renseignés dans un autre diagramme).
• On appelle ***transition*** un ***changement d'état*** . Celui ci est souvent déclenché par un ***événement extérieur*** et est parfois ***conditionné*** par un ***[gardien]*** .

NB:

- **Un état, c'est l'état de quelquechose** (un objet , un sous système, …) et donc un diagramme d'état sera souvent placé comme un détails d'une classe dans l'arborescence d'un modèle UML.
- Une transition d'un état vers lui même (self transition) implique généralement une sortie et une nouvelle entrée dans celui-ci .

Utilisations classiques des diagrammes d'états :

- **cycle de vie d'une donnée importante** (création en base , nombreuses mises à jour conditionnées , suppression)
- ~~états d'âme d'Eric~~
- **diagramme d'états d'une interface graphique** (***navigations***, …)
- états d'objets industriels (ex : vérin contracté ou deployé)
- ...

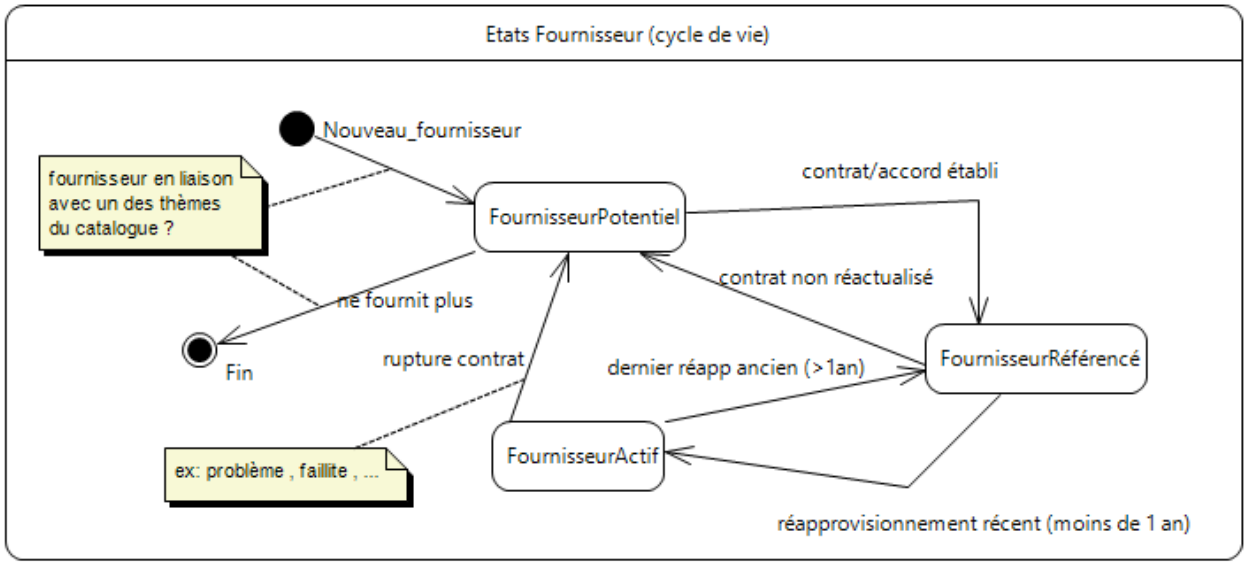
Quelques noms/types (classiques) d'événements :

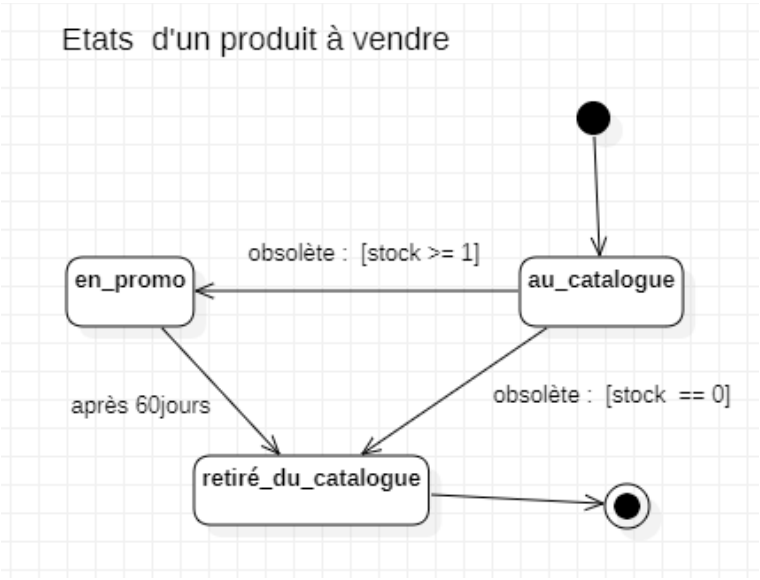
Types d'événements	*exemples*	*sémantiques*
Change Event	**when**(exp_booleenne)	L'expression booléenne devient vraie (après changement)
Signal Event	*feu vert , ...*	Signal reçu (sans réponse à renvoyer)
Call Event	*demande_xy_reçue , ...*	Appel reçu
Time Event	**after**(temporisation)	Période de temps écoulée

Exemple(s):

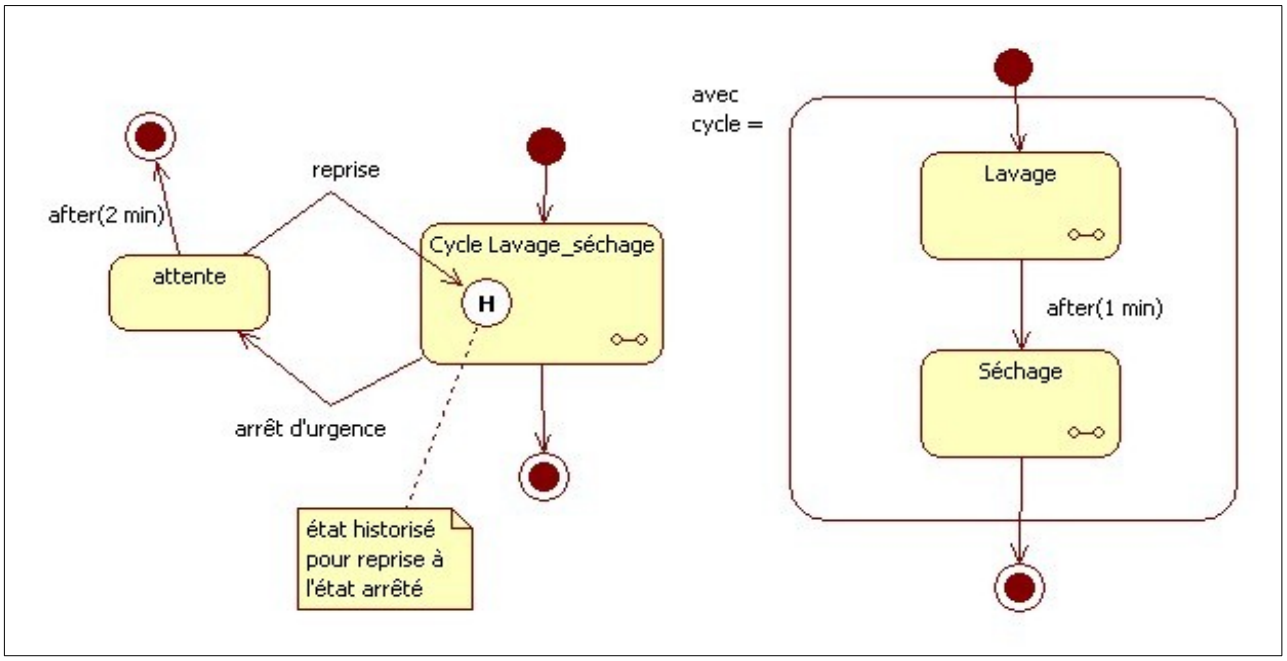
Lumière (avec minuterie) : *allumée* ------ ***after(30s)***----> *éteinte*

8.1. Exemples simples

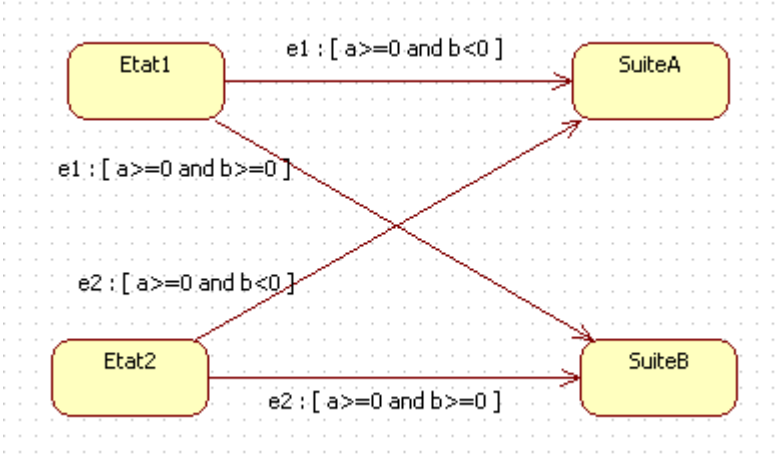


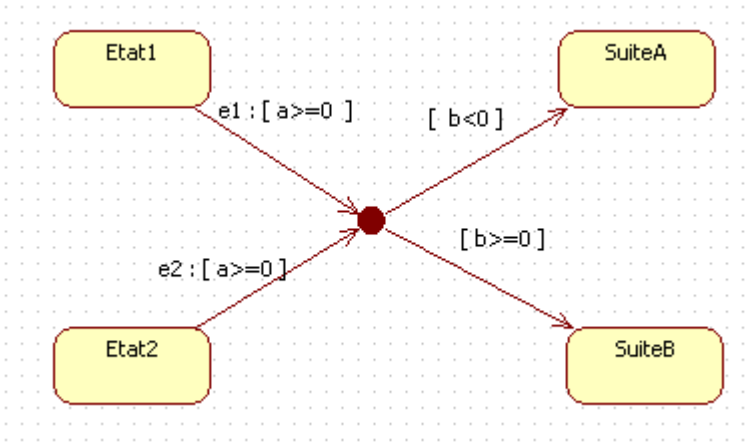


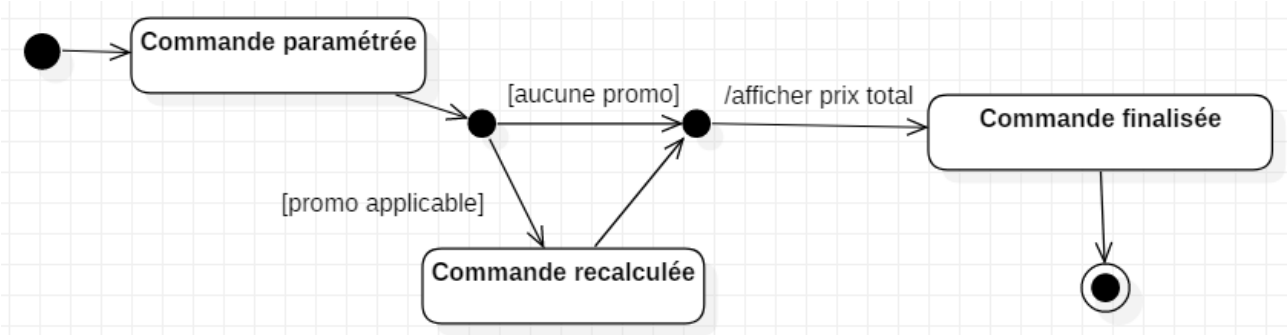
8.2. Etats historisés (rares , pour informatique industrielle)



8.3. Point de jonction (depuis UML2)

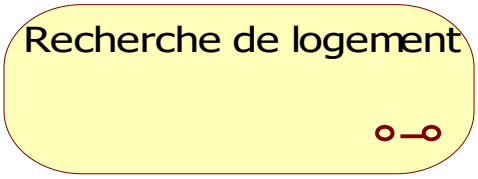






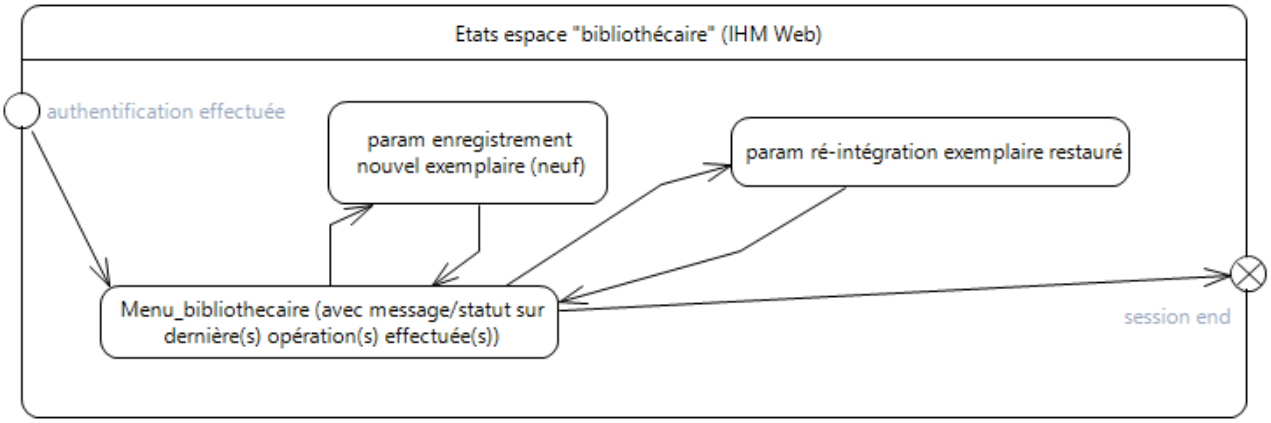
8.4. Etat composite (super état)

Vision abrégée d'un état composite : "***submachine state***" (à adapter selon outil UML):

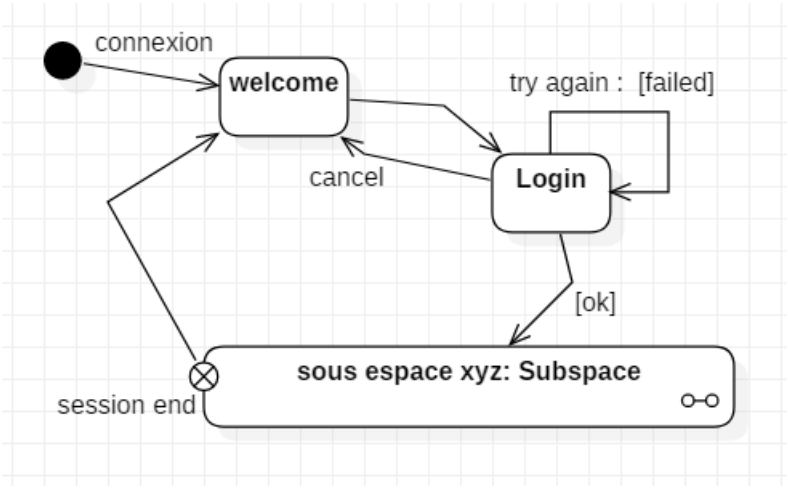


NB : *Quelquefois considéré comme état ordinaire*

Point de connexions (en **entrée** et en **sortie**) **sur la frontière d'un état composite (dans sous diagramme)** :



Au sein d'un autre diagramme (parent ou autre) , l'**état composite** "***Espace xxx***" sera noté comme un état presque ordinaire (simple rectangle) , on l'on pourra tout de même éventuellement placer des "***référence à des connections***" pour affiner les branchements des transitions (changements d'états) :



8.5. Liens avec diagramme de classe.

Un diagramme d'états (en tant qu'états de quelquechose) est souvent rattaché à une classe .

Un état peut **souvent être codé** par **un ou plusieurs attributs complémentaires** pouvant prendre **plein de types/formes différent(e)s** :

- "booléen"
- "énumération"
- "lien_vers_xy" existant ou pas ,
- "dateActionXy" nulle ou pas ,
- ...

9. Rôles de la conception

La **conception** à pour **rôle** de définir "**comment**" les choses doivent être **mise en oeuvre**:

- quelle **architecture** (client-serveur , n-tiers, SOA , ...)
- quelles **technologies** (langages , frameworks , ...)
- quelle **infrastructure** (serveurs à mettre en place , ...)

avec tous les détails nécessaires .

Étymologiquement:

conception ==> **inventer** (concevoir) une solution

pragmatiquement:

conception ==> très souvent **réutiliser/choisir une solution/technologie** *(framework)*
[***ne pas ré-inventer***]

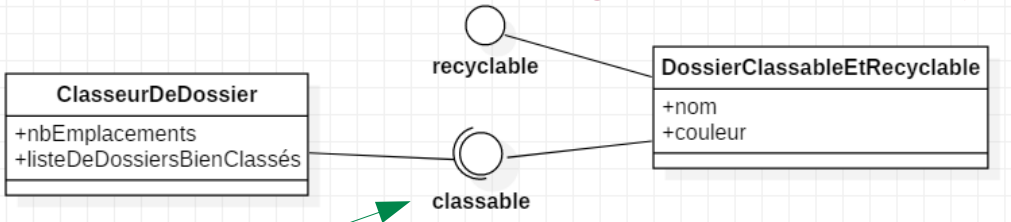
NB :

- Au début des années 1990, il n'y avait pas encore beaucoup de framework parfaitement au point et il fallait à l'époque beaucoup inventer .
- Au début des années 2000, quelques api/frameworks bien structurés (ex : .net/C# , Java/JEE) dominaient le marché et il faillait alors comprendre et intégrer (ne pas ré-inventer)
- A l'aube des années 2020, l'architecture micro-services offre plein de variantes dans les possibilités de mise en œuvre et il faut donc bien argumenter/spécifier les choix d'architecture et avoir quelquefois recours à une double modélisation :
  - cohérence à grande échelle (urbanisation, communications entre applications, ...)
  - modèle orienté objet détaillé à petite échelle (ex : api-rest) .

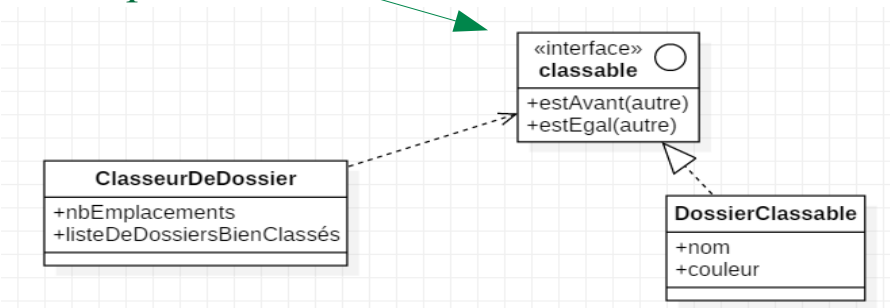
10. Interfaces (diag. de classes)

Interface

Une interface correspond à une collection d'opérations sans code (classe spéciale sans aucun attribut , seulement des signatures de méthodes).

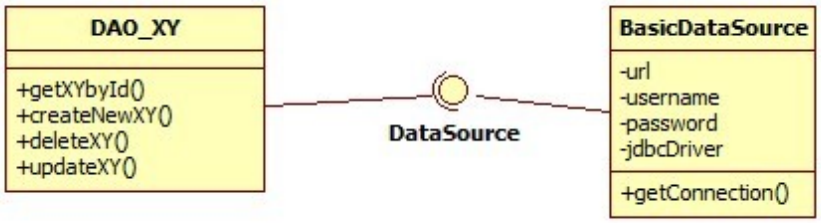


2 notations possibles

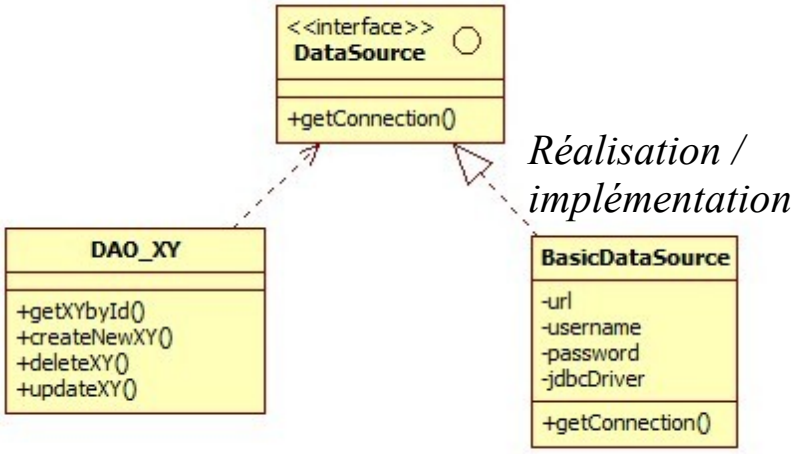


Tout comme une classe abstraite, une **interface** correspond à un **type de données** (permettant de déclarer des références).

Notation Compacte (dépendance)



Notation développée (avec détails)

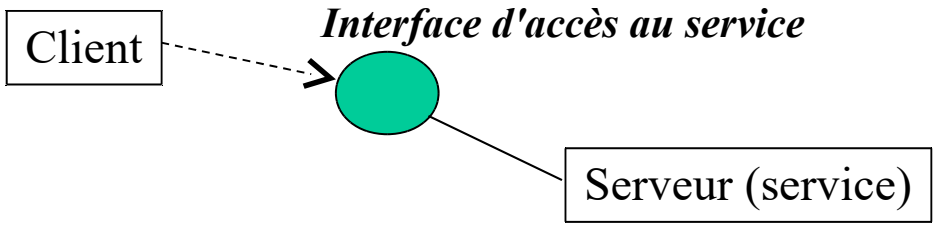


Réalisation / implémentation

Interface (contrat)

• Une **interface** peut être considérée comme un **contrat** car chaque classe qui choisira d'implémenter (réaliser) l'interface sera obligée de programmer à son niveau toutes les opérations décrites dans l'interface.

• Chacune de ces opérations devra être convenablement codée de façon à rendre le **service effectif** qu'un client est en droit d'attendre lorsqu'il appelle une des méthodes de l'interface.

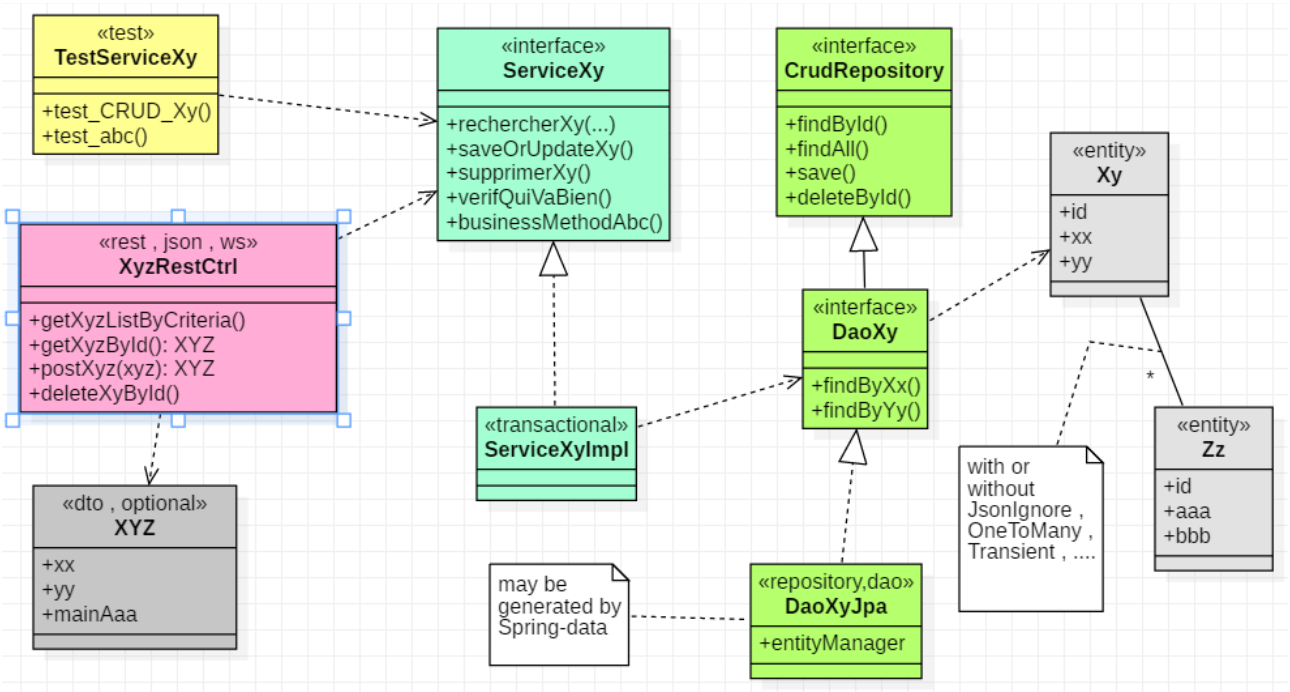


NB : Désignant un paquet de fonctionnalités invocables, certaines interfaces ont des noms finissant par "able" :

- Imprimable, Printable
- Serializable
- Affichable
- Configurable
- ...

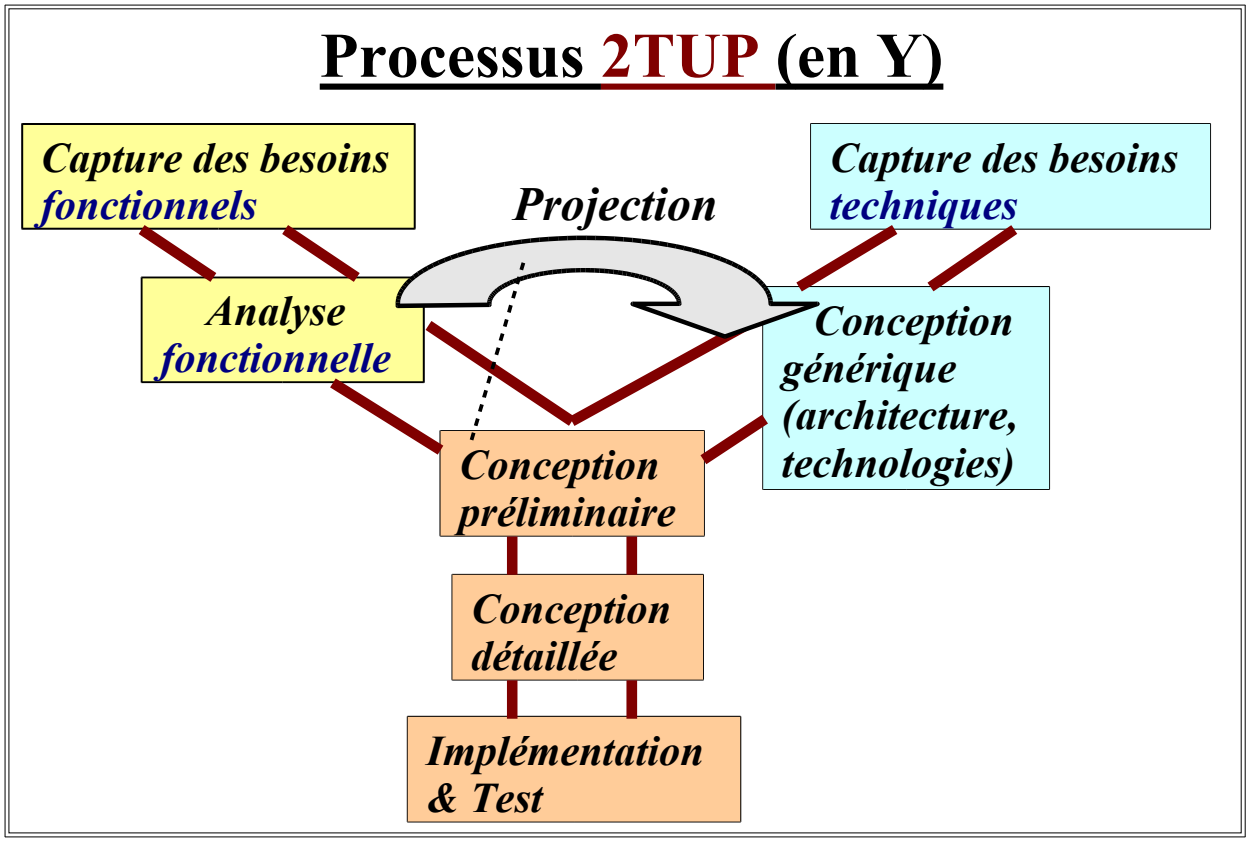
11. Eventuelle conception générique

Exemple: architecture technique (d'une api REST) basée sur une des technologies des années 2018_2019 (Spring-boot / Spring-mvc / Spring-Data ....) et devant prendre en charge les exigences techniques d'une application de gestion (transactions, ...) .



12. Projection du fonctionnel dans technologies

Il s'agit ici de **projeter** le résultat de l'analyse au sein d'une architecture logique et technique définie durant l'étape "architecture" (ou "conception générique").



Principal résultat de la conception préliminaire (projection):

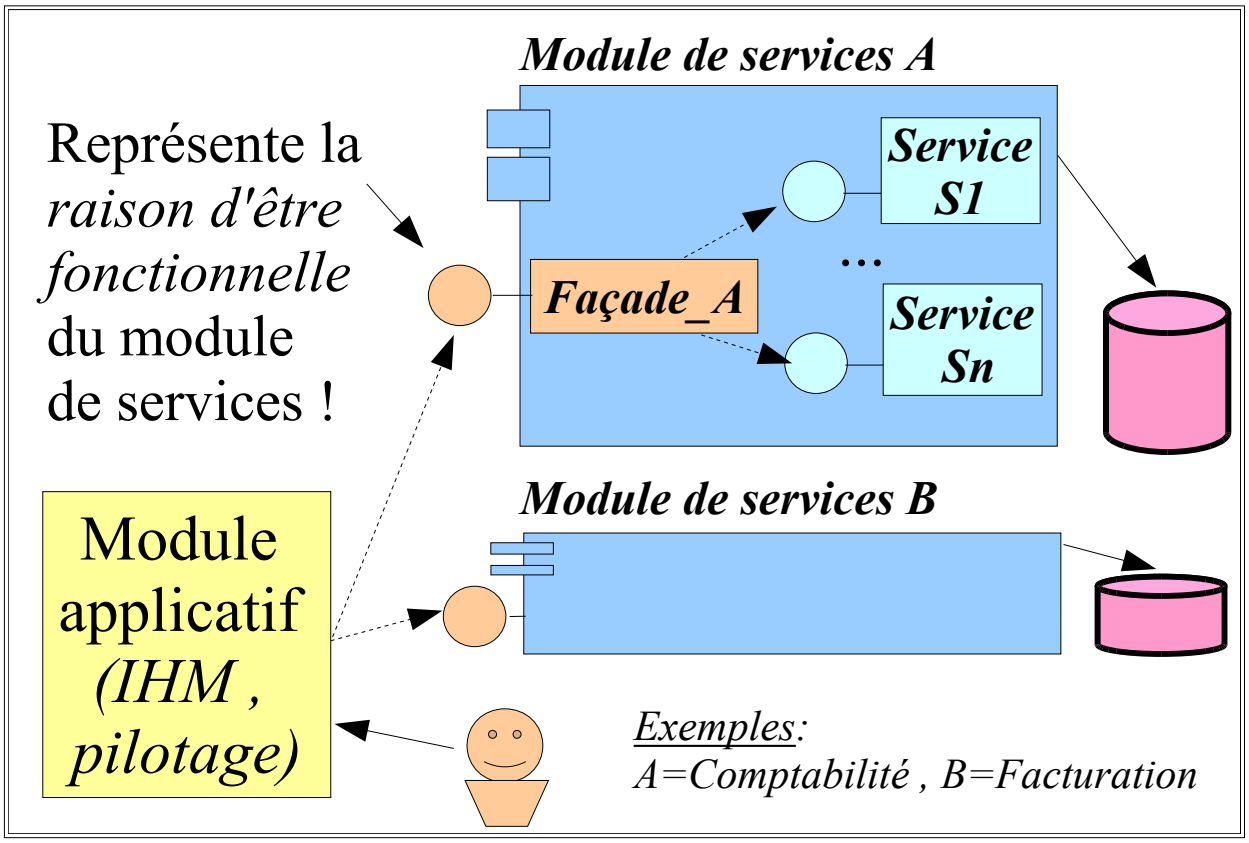
n "packages fonctionnels" * m "couches/niveaux techniques"

===> n*m packages dans le code à réaliser/produire:

- fr.xxx.yyy.AppliA.partieIHM.**web**
- fr.xxx.yyy.AppliA.*partieFonctionnelleA*.**service**
- fr.xxx.yyy.AppliA.*partieFonctionnelleA*.**entity**
- fr.xxx.yyy.AppliA.*partieFonctionnelleB*.**service**
- fr.xxx.yyy.AppliA.*partieFonctionnelleB*.**entity**
- ...

==> Il vaut mieux appliquer systématiquement certaines règles de projection (via MDA ou ....) pour être efficace/rapide .

13. Eventuels modules (avec interfaces,façades)



D'un point de vue fonctionnel , une façade donne du sens à un module de service. Son nom doit donc être bien choisi (pour être évocateur).

D'un point de vue technique, une façade n'est qu'un nouvel élément intermédiaire de type "Façade d'accueil" qui servira à orienter les clients vers les différents services existants.

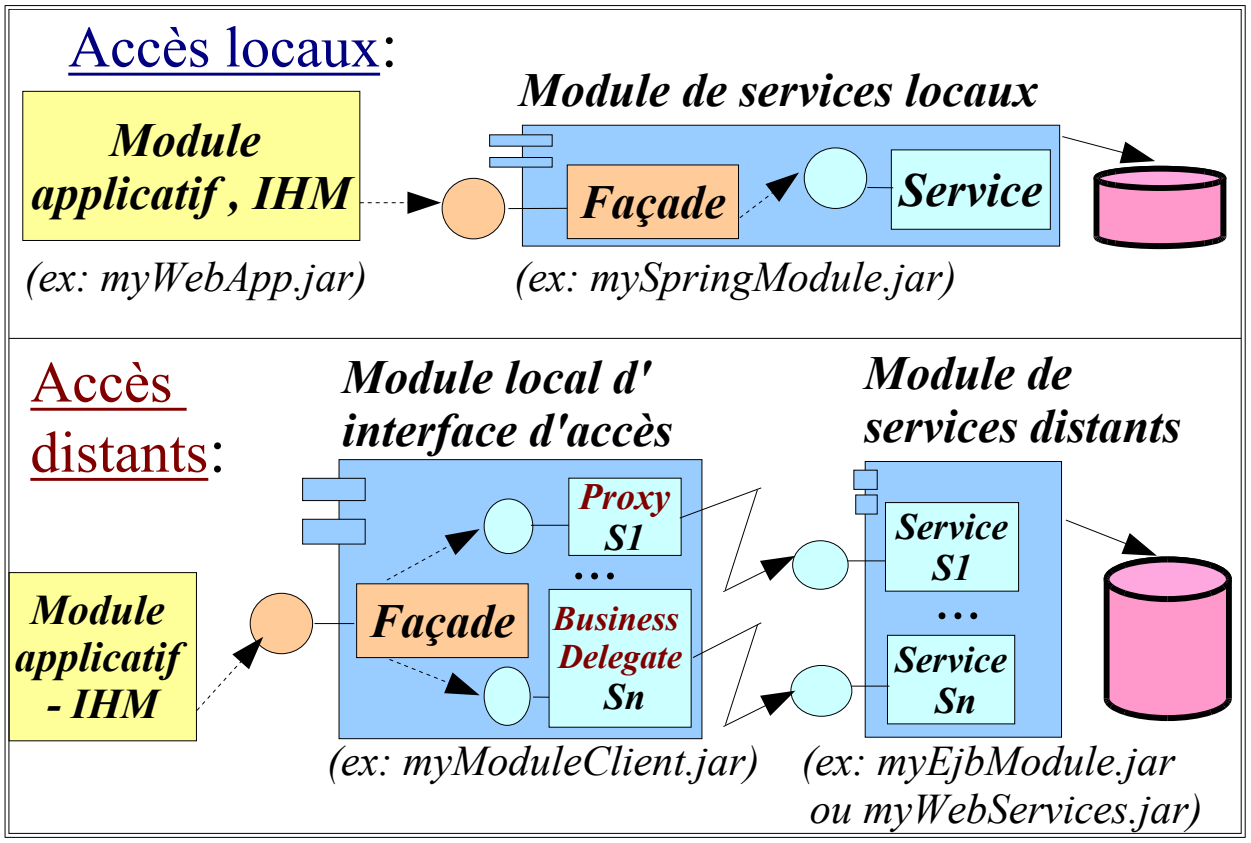
Exemple:

Façade_Comptabilité
. getServicePostesComptables()
.getServiceBilan()
.getServiceJournal()
.getServiceGrandLivre()
...
.getServiceN()

utilisation:

facadeCompta.getServiceBilan().xxx()

facadeCompta.getServiceGrandLivre().yyy();



Derrière une façade (très souvent locale) on peut éventuellement trouver des services distants.

Un "**proxy**" est un représentant local d'un service distant .

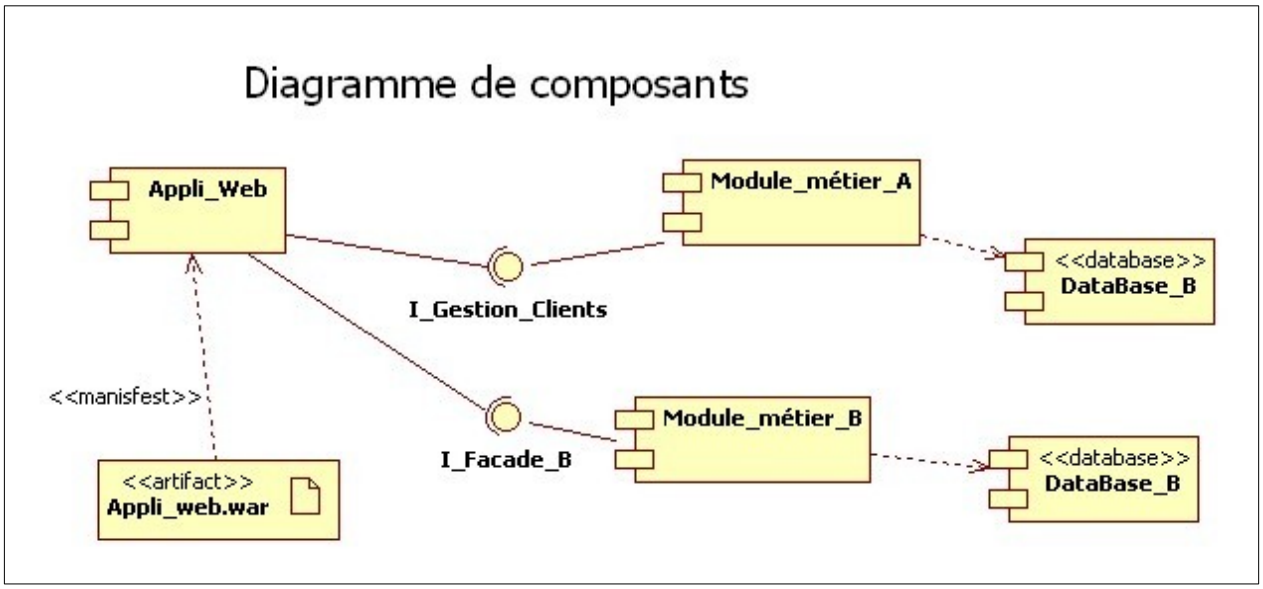
Ce terme (plutôt technique) désigne assez souvent du code généré automatiquement (à partir d'un fichier WSDL par exemple).

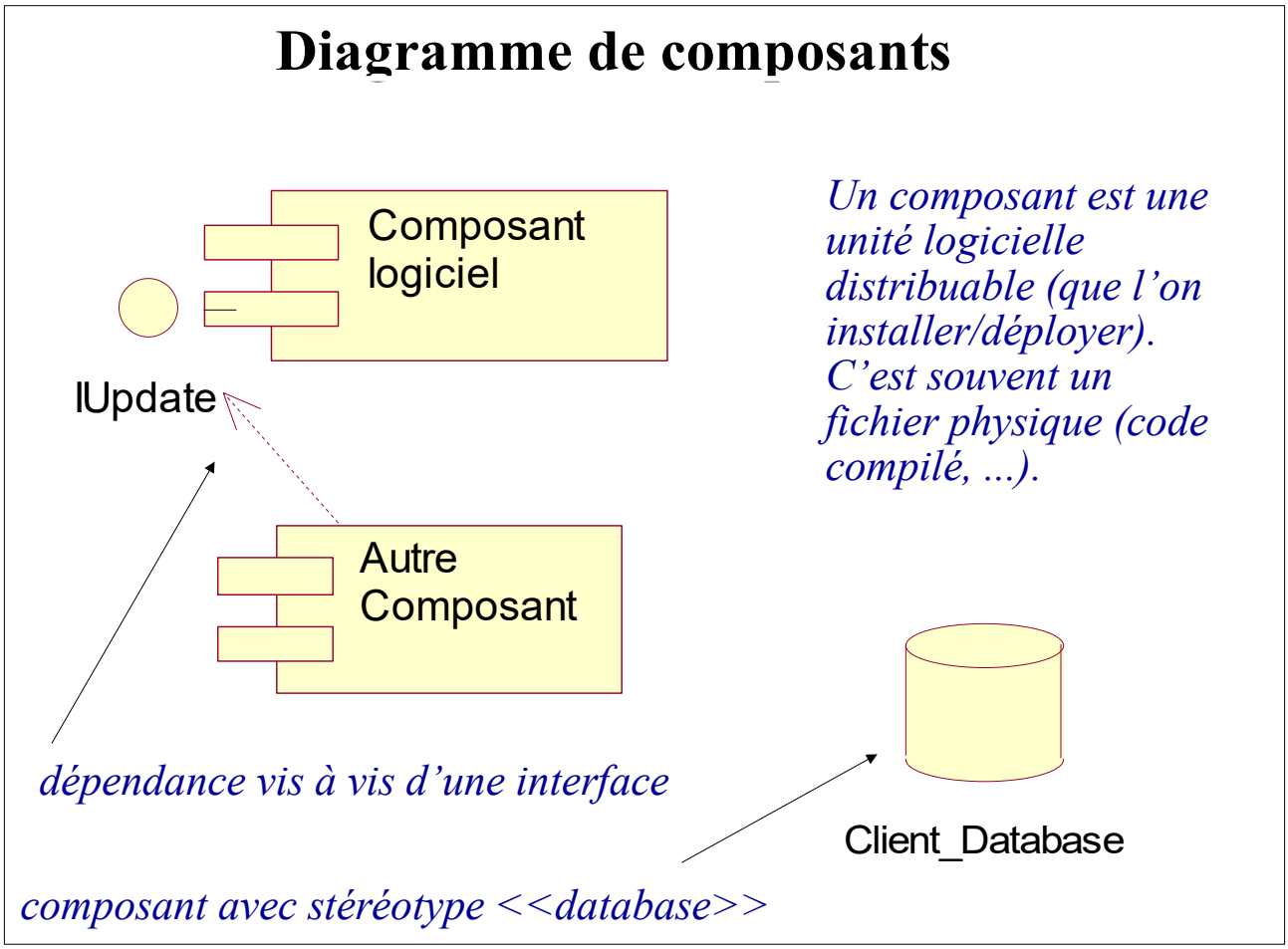
Pour bien contrôler l'interface locale d'un service distant (de façon à n'introduire aucune dépendance vis à vis d'une technologie particulière), on met parfois en oeuvre des objets de type "**business delegate**" qui cache dans le code privé d'implémentation tous les aspects techniques liés aux communications réseaux:

- localisation du service , connexion (lookup(EJB) ou ...)
- préparations/interprétations des messages/paramètres .
- …

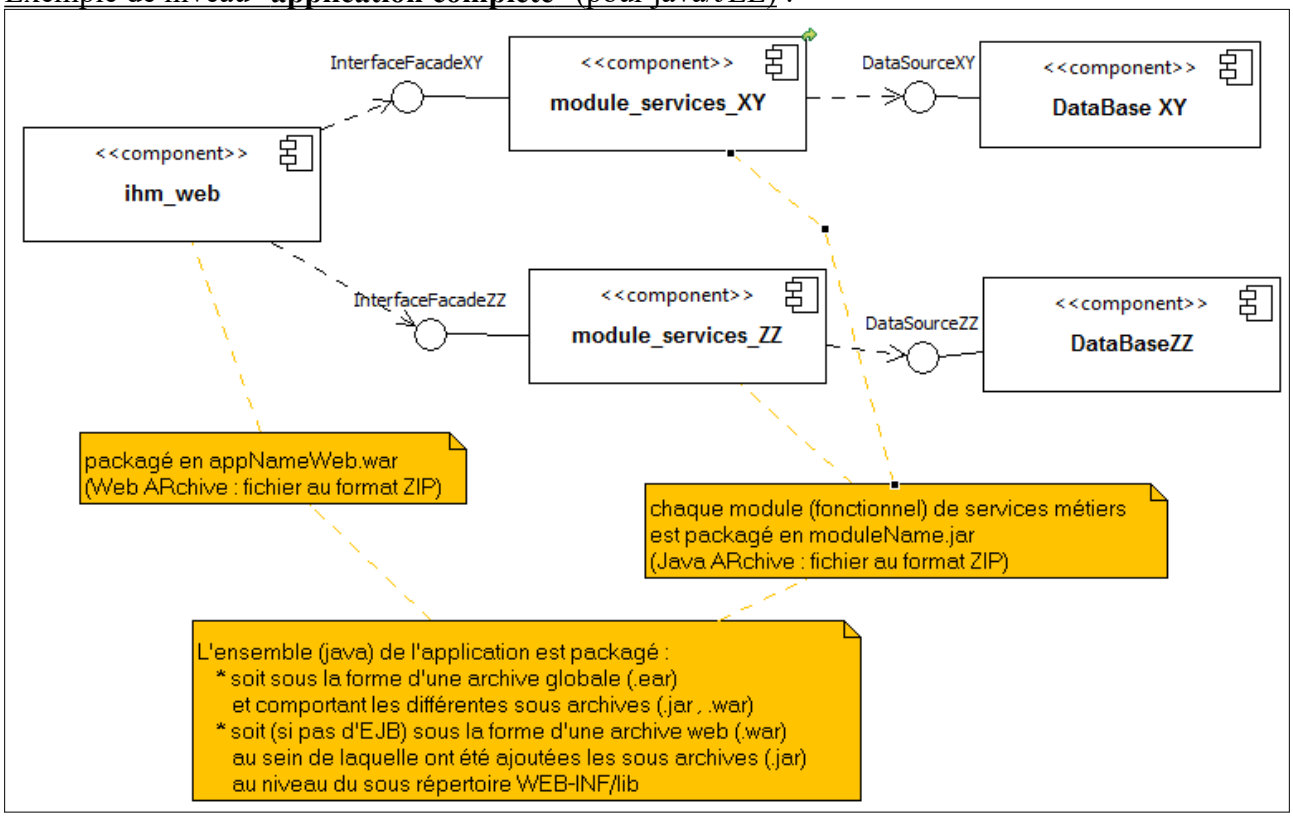
14. Diagramme de composants UML

14.1. Composants

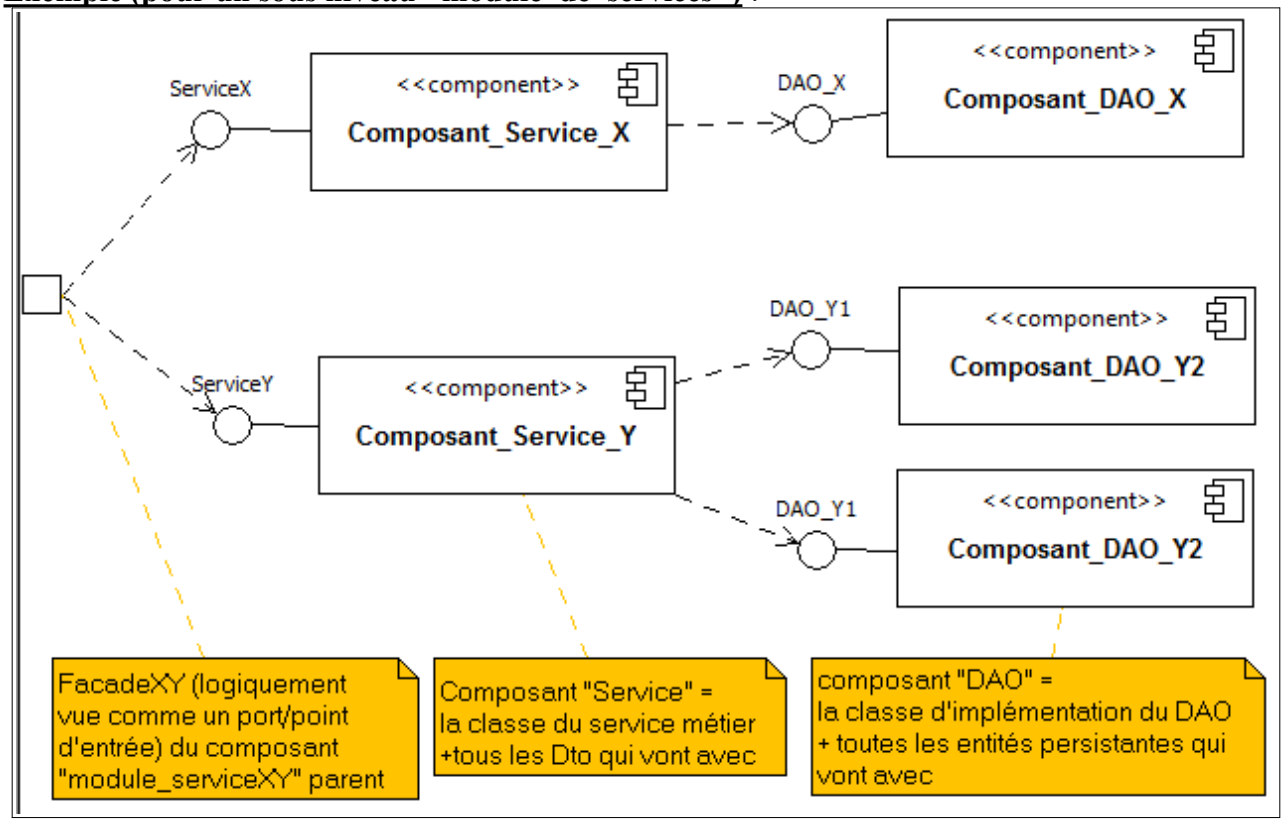




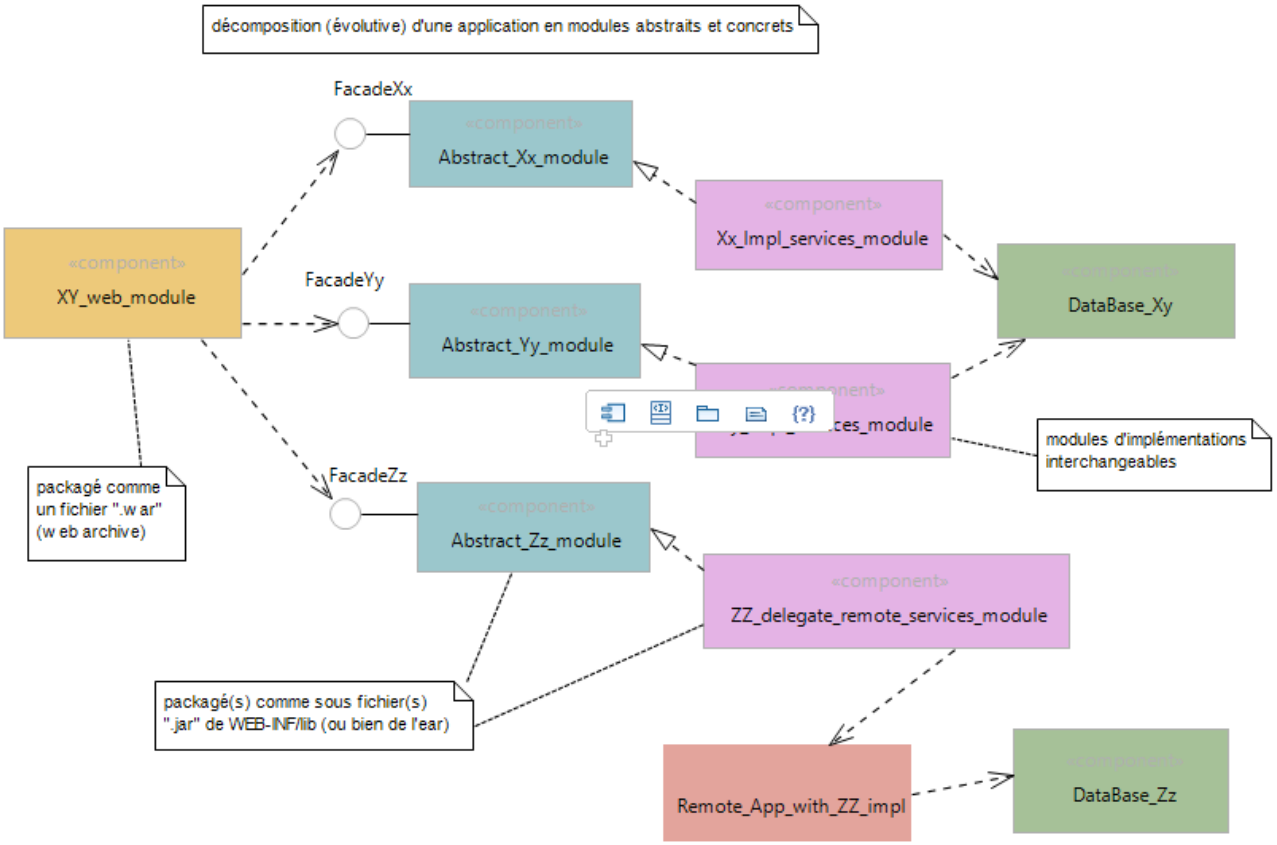
Exemple de niveau "**application complète**" (pour java/JEE) :



Exemple (pour un sous niveau "module_de_services") :



14.2. Diagramme de composants avec dépendances affinées



Un module abstrait comporte essentiellement :

- des **interfaces**
- des types de données en entrées/sorties des appels de méthodes (ex : DTO)
- une éventuelle "façade"

Un module d'implémentation comportera :

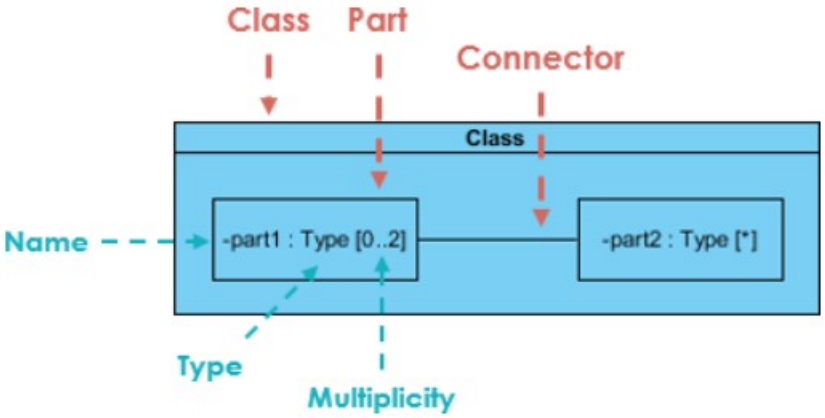
- le code effectif d'implémentation (en local)

ou bien

- un code qui délègue les traitements vers des services extérieurs

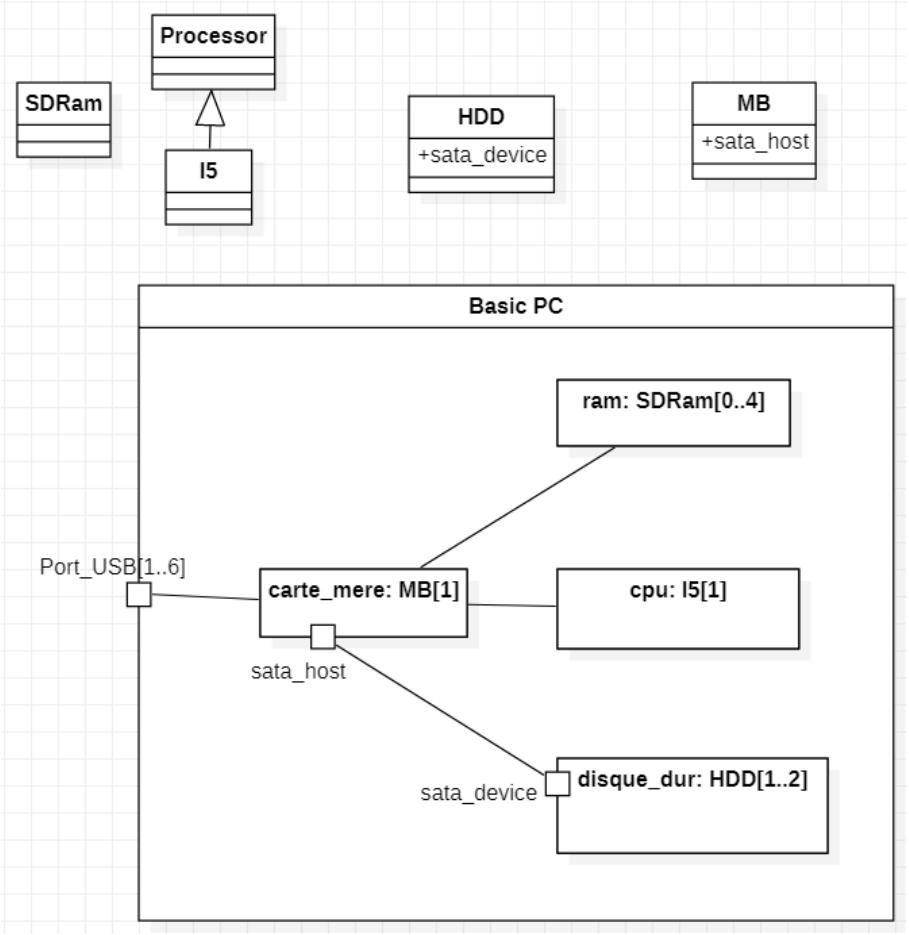
15. Eventuel diagramme de structure composite

Il s'agit essentiellement d'une extension au diagramme de classes permettant de **montrer des sous parties (avec multiplicités précises) avec des connecteurs et d'éventuels ports d'accès** .



(illustration tirée de la documentation "visual-paradigm").

Avec Star-UML 3 , on peut obtenir le résultat suivant (avec "suppress attributes" coché et "suppress reception" décoché dans le menu contextuel de la classe composite) :

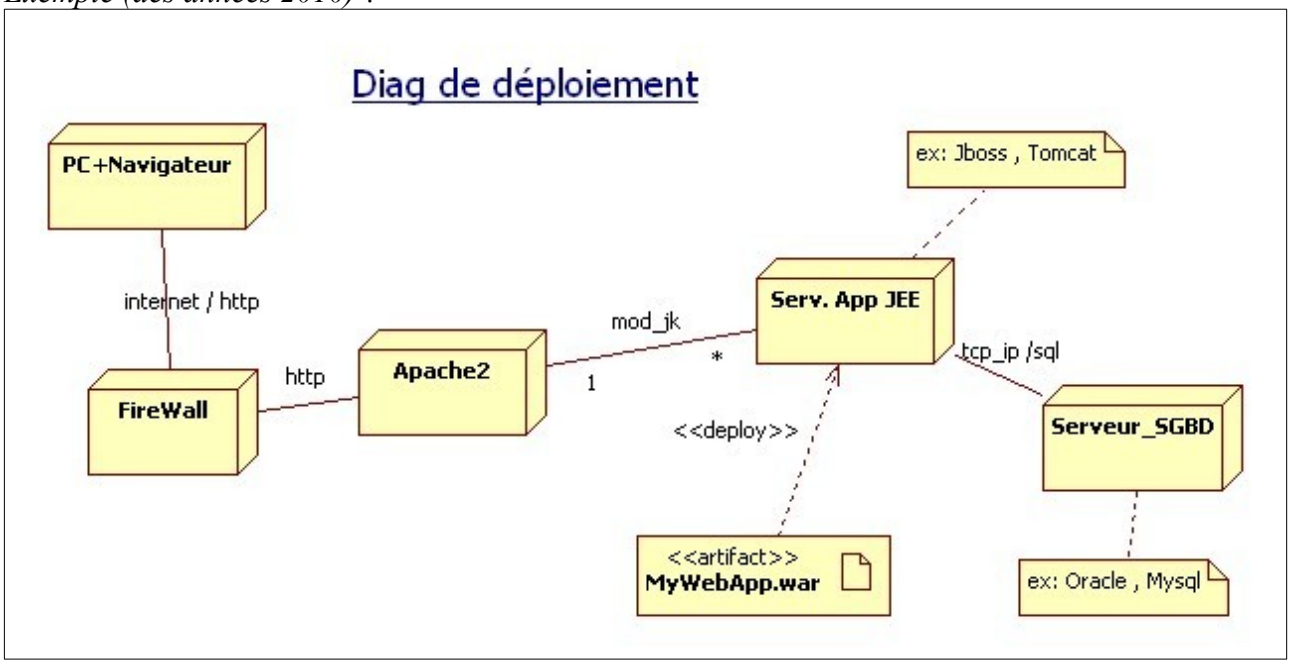


16. Diagramme de déploiement

Sachant qu'il ne faut pas sous estimer la spécification de l'environnement cible (intégration / pré-production , ...) , un diagramme de déploiement UML permet d'indiquer :

- les grandes lignes de la topologie d'une partie du S.I. (serveurs , liaisons réseaux, ...)
- les éléments à installer/déployer ou configurer (ex: base de données , applications , …)

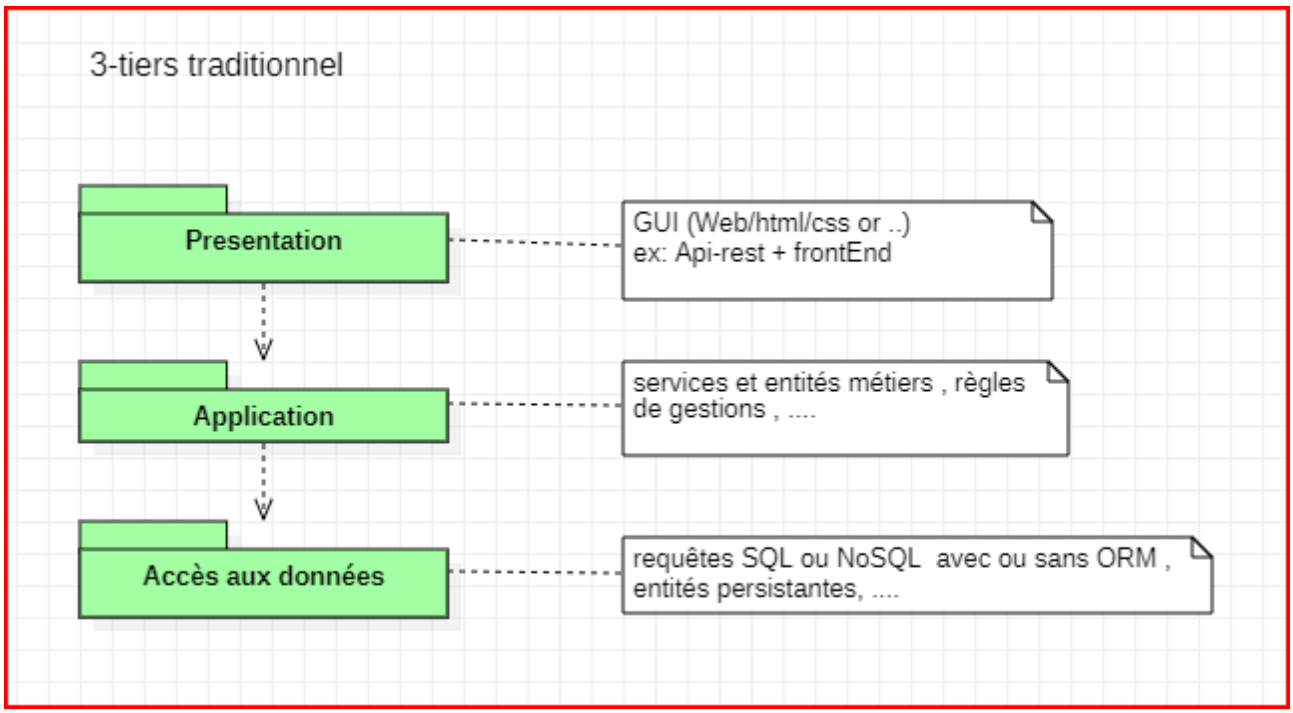
Exemple (des années 2010) :



Exemple (des années 2020) :
→ ***à faire en TP***

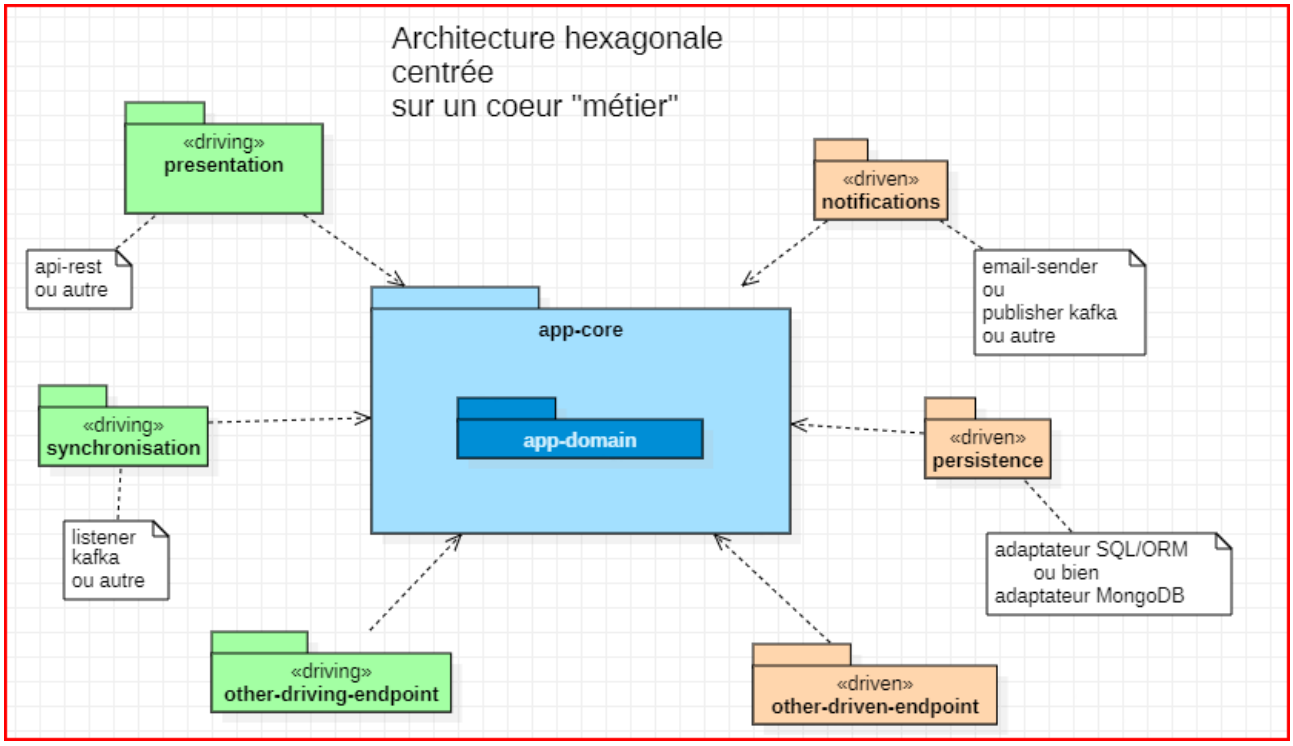
17. Quelques architectures classiques

17.1. Architecture n-tiers

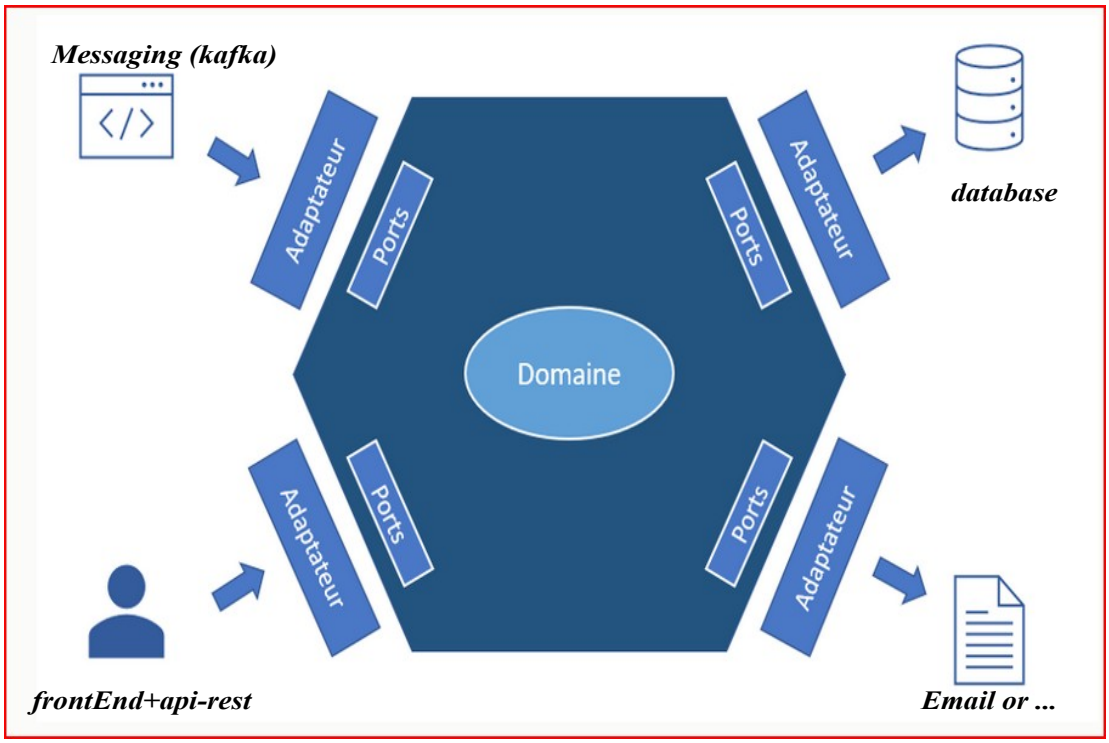


Dans ce cadre , c'est la couche "application" qui s'adapte à la couche "accès aux données" et la couche "présentation" qui s'adpate à la couche "application" .

17.2. Architecture hexagonale



Dans ce cadre , c'est la couche "persistance/accès aux données" qui s'adapte en rendant les services techniques nécessaires au coeur central "core / domain" .



NB : les architectures "en oignon" et "clean code" ne sont que des variantes de l'architecture hexagonale (de *Alistair Cockburn*).

VI - Essentiel sur les "Design patterns"

1. Présentation du concept de Design Pattern

Notion de "Design Pattern"

Un **Design Pattern** [DP] est un **modèle de conception réutilisable** (dont on peut s'inspirer de multiples fois sans pour autant toujours coder les choses de la même façon).

C'est concrètement un document d'une dizaine de pages comprenant:

Description d'un problème récurrent (contexte)

Modélisation (ex: ***UML***) d'une solution de conception générique

Description des avantages/inconvénients

Exemples de code , Variantes

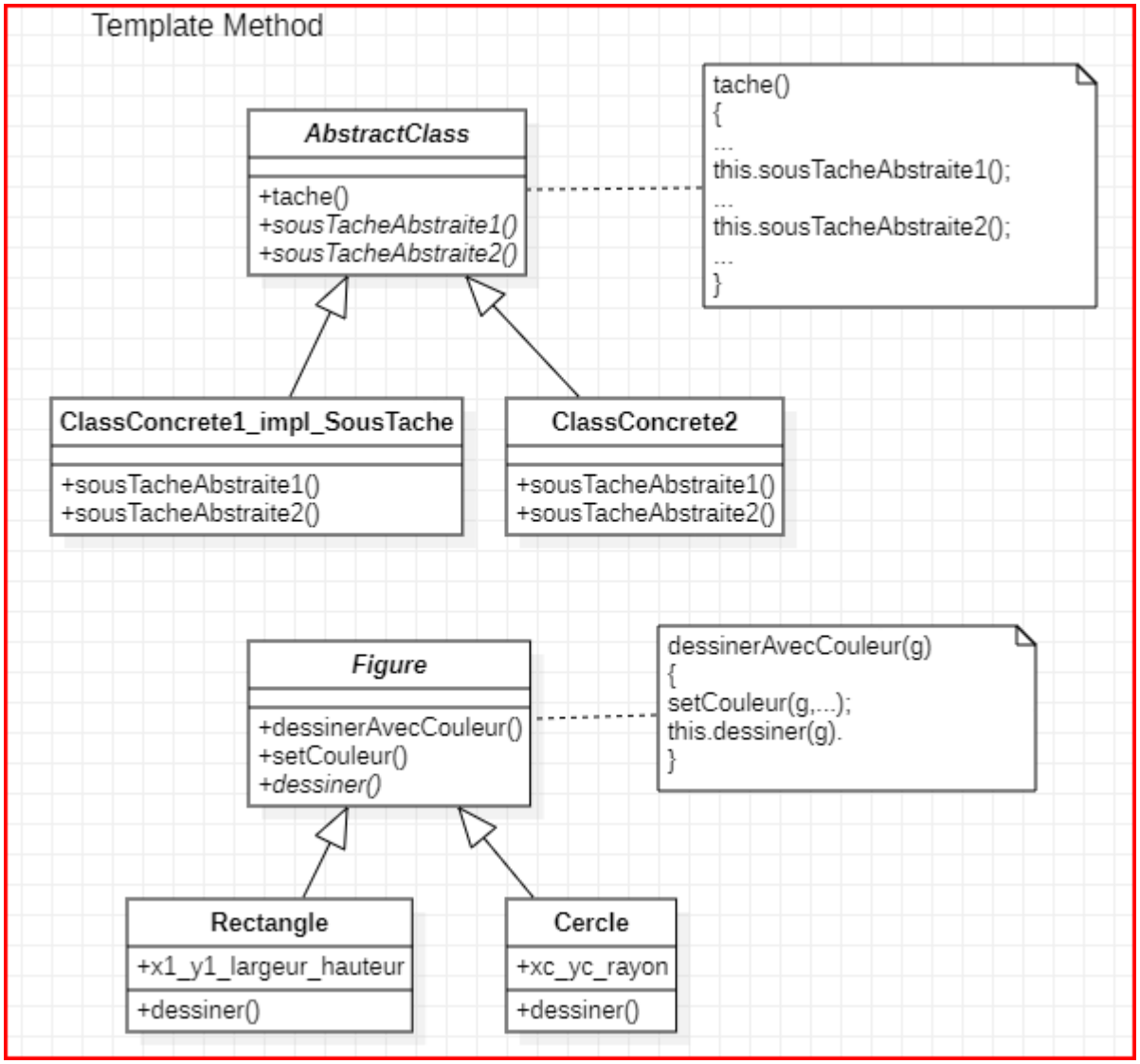
NB:

L'inventeur du concept de "Design Pattern" (**Christopher Alexander**) ne travaillait pas dans le domaine de l'informatique.
Selon lui :

<< ***Chaque modèle décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le coeur de la solution de ce problème, d'une façon telle que l'on peut réutiliser cette solution des millions de fois, sans jamais le faire deux fois de la même manière*** >> .

1.1. Exemple: Patron/modèle de méthode (Template method)

Factoriser ce qui est commun , différentier le spécifique .



2. Importance du contexte et anti-patterns

2.1. contexte d'un design pattern et objectif visé

Attention: un Design Pattern n'est généralement intéressant qu'au sein d'un contexte particulier:

- Certaines conditions doivent généralement être vérifiées (volume , ...) .
- But à atteindre ?
- ....

L'utilisation inadéquate (non réfléchie ou systématique) d'un design pattern est quelquefois contre-productive ==> ceci constitue un des "anti-pattern" (choses à ne pas faire , modélisation d'un dis-fonctionnement ou problème potentiel, ...).

2.2. patterns et anti-patterns

	Design Pattern	**Anti Pattern**
Pb Récurrent / Exemple	Objectif visé revenant souvent	Problème (défaut) apparaissant souvent
Modélisation / Généralisation	Modèle d'une chose à reproduire	Modèle d'une chose à éviter
ex de code / variantes	ex de code dont on peut s'inspirer + variantes	ex de mauvais code (ou mauvaise technique) à prohiber + alternatives conseillées

Quelques exemples d'anti-patterns:

- **trop de "copier/coller" dans le code**
  ==> reproduction/prolifération de choses à faire évoluer en parallèle ou d'éventuels défauts
  ==> maintenance difficile
  ==> une factorisation est souvent possible (via une petite restructuration du code)
- **trop de "if" ou "switch/case" sans polymorphisme**
- **code "mort" (jamais utilisé)**

...

3. Principaux "design patterns"

3.1. Les grandes séries de "Design Patterns" orientés "objet"

Les grandes séries de "design patterns"

• ***GRASP*** *[* ***G****eneral* ***R****esponsability* ***A****ssignment* ***S****oftware* ***P****atterns]*
==> bonnes patiques (peu formalisées mais grandes lignes directrices) , simples à comprendre et intuitifs ==> patterns adaptés dès la fin de l'analyse (non techniques)

• ***<<Design Principles>>*** *(Robert MARTIN , Bertrand MEYER)*
==> grands principes objets assez formalisés (règles à respecter) . Essentiellement liée à la structure générale des modules , cette série de patterns est tout à fait adaptée à la conception préliminaire.

• ***GOF*** *(Gang Of Four)*
==> série assez technique de "design patterns" (proches du code)
==> adapté pour la conception (préliminaire et détaillée).

• ...

3.2. "Design Patterns" du GOF

Principaux "Design Pattern" *(G.O.F.)*

	Rôle		
	Créateur	**Structurel**	**Comportemental**
Classe / Statique	**Fabrication**	**Adaptateur (stat.)**	Interprète
			Patron de Méthode
Objet / Dynamique	**Fabrique Abstraite**	**Adaptateur (dyn.)**	Chaîne de responsabilité
	Monteur	**Composite**	Commande
	Prototype	**Décorateur**	Itérateur
	Singleton	Façade	Médiateur
		Poids Mouche	Mémento
		Pont	Observateur
		Procuration	État
			Stratégie
			Visiteur

G.O.F. ==> **Gang Of For** (le gang des 4 personnes qui ont lancé les "Design Patterns" dans le monde de la conception orientée objet) ==> Thèse & Livre :

Livre (de référence) fortement conseillé :

DESIGN PATTERNS
Catalogue de modèles de conception réutilisables [traduction francaise de Jean-Marie Lasvergères.] / **Vuibert / Addison Wesley**

Auteurs (les 4):
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

3.3. Autres ensembles de designs patterns

- **GRASP**
- **Object Principles (Meyer & Martin)**

4. Différences entre "Design Pattern" et "Framework"

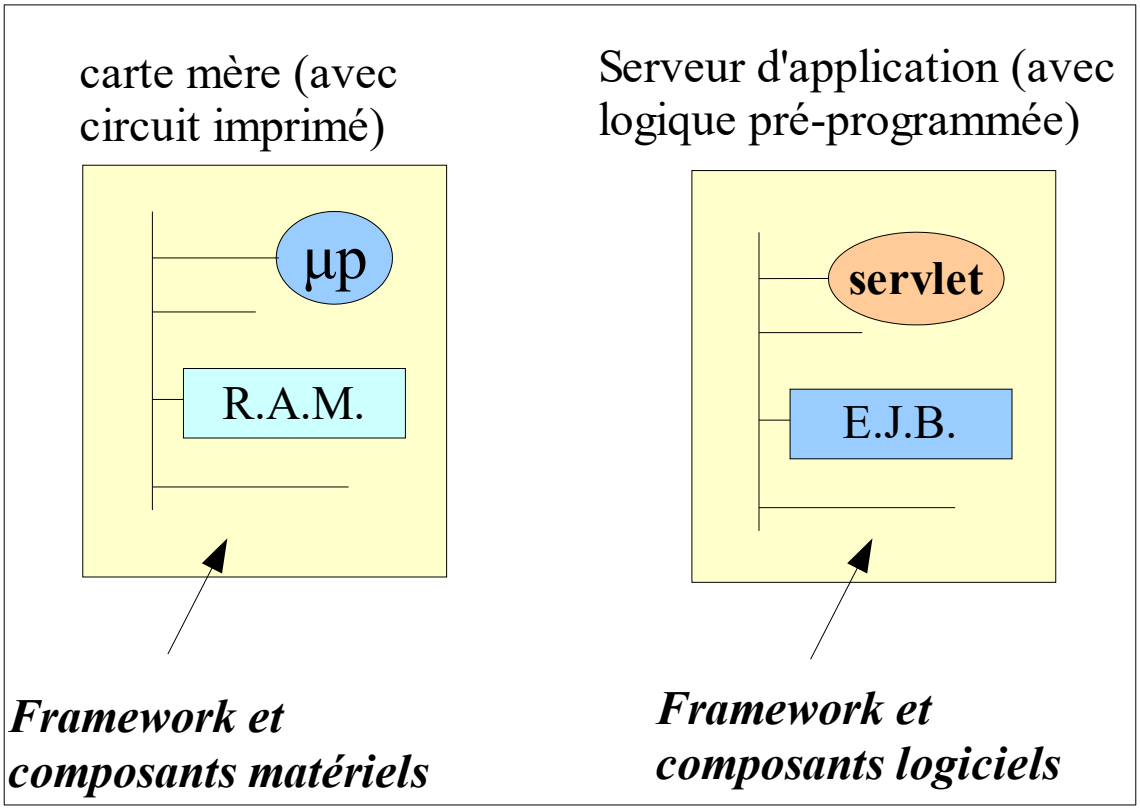
Notion de "Framework"

Un "Design Pattern" est un modèle qui reste à programmer (en fonction du contexte).

Un "**Framework**" est *en grande partie déjà programmé*: il s'agit d'un **schéma applicatif "pré-cablé" prêt à recevoir des composants logiciels**.

Analogie classique:

Carte mère avec circuit pré-cablé pour recevoir des composants matériels (CPU, Ram, carte PCI).
- Idée d'industrialisation.



…

5. Liste des principaux "design patterns" du GOF

Design Pattern	Description
Abstract factory (fabrique abstraite)	Instanciation indirecte de familles de produits. (ex: Look & Feel "Windows" ou "X11/Motif" Java / SWING). En changeant de fabrique on génère des objets différents (look1,look2, ...) qui ont néanmoins les mêmes fonctionnalités (même interfaces).
Adapter (adaptateur)	Intermédiaire permettant de convertir l'interface (figée) d'une classe existante avec celle attendue par un client.
Bridge (pont)	Correspondance (pont) entre 2 hiérarchies de classes (ex: XXX - XXXImpl de AWT). Séparation de l'abstraction et de la représentation/implémentation.
Builder (monteur)	Fabriquer des parties via des objets monteurs dirigés par un objet directeur. Séparer la construction de la représentation.
Chain of responsability (Chaîne de responsabilité)	Requête avec: 1. 1 émetteur. 2. des récepteurs chaînés entre eux (hop, hop ,hop jusqu'à traitement effectif [+ éventuelle valeur ajoutée sur les intermédiaires aux responsabilités bien définies]). ==> exemple ACL = Access Control List.
Command (Commande)	Encapsulation d'une requête dans une méthode (ex: execute()) d'un objet commande. ==> Liste de commandes ==> permet undo/redo et le déclenchement d'une même commande depuis plusieurs parties de l'IHM (menu, toolbar, bouton poussoir).
Composite	Composition récursive à niveau de profondeur quelconque (héritage + composition combinés)
Decorator (Décorateur / Enveloppe transparente)	Enveloppe transparente rajoutant de nouvelles fonctionnalités.
Facade	Interface unifiée pour la totalité d'un sous système / accueil plein de variantes (ex : façade agnostique , ...)
Factory method (méthode de fabrication)	Création indirecte d'instance déléguée au niveau d'une méthode de type "create()" [beaucoup de variantes]
Flyweight (poids mouche)	Comment gérer plein de petits objets ? ==> petit objet partagé (avec une partie interne intrinsèque) et avec des méthodes comportant une référence sur un contexte (partie externe à la charge du client).
Interpreter (interpréteur)	Grammaire (phrases / expressions à interpréter , exemple: 2x+y+5*z/3). ==> construction d'un arbre syntaxique dont les nœuds sont des objets. Fonction interpréter() récursive et polymorphe.
Iterator (itérateur)	Traverser (balayer/parcourir) une collection (liste/tableau) sans avoir à connaître la structure interne et de façon à accéder à chacun des éléments.
Mediator (médiateur)	Intermédiaire commun à un paquet d'objet (coordonnant les interactions)==> pour réduire le couplage.
Mémento	Petit objet secondaire (avec interfaces large et fine) permettant de mémoriser l'état d'un objet principal de façon à restaurer les valeurs de celui-ci plus tard (Respect de l'encapsulation, permet un undo).
Observer (observateur)	Définit une dépendance de un à plusieurs 1 Mise à jour / des Notifications (callback) 1 Diffusion / des Souscripteurs
Prototype	Créer de nouveaux objets à partir d'un exemplaire déjà instancié

	(clonage). (ex: Palette d'objets graphiques à cloner).
Proxy (proximité)	Fournir un substitut pour accéder indirectement à un objet (souvent distant).
Singleton	Une seule instance pour une certaine classe (Méthode statique pour créer (ou obtenir) cette instance.
State (état)	Différents sous-objets ayant une interface commune codent différents comportements d'un objet englobant ayant plusieurs états (comme s'il changeait de classe).
Strategy (stratégie)	Classe abstraite d'algorithmes interchangeables pour un certain contexte. [externaliser une responsabilité avec variantes]
Template method (Patron de méthode)	Algorithme abstrait basé sur un même squelette (méthode d'une sur-classe) dont différentes sous tâches (parties) sont codées comme des méthodes polymorphes dans diverses sous classes.
Visitor (visiteur)	Sur un objet que l'on peut parcourir (ex : hiérarchie de noeuds d'un arbre ou liste) on va déclencher (via des **visiteurs actifs**) des opérations génériques durant une traversée. ==> objets visitables et visiteurs doivent être pensés et modélisés pour être compatibles.
...	

6. Design Patterns fondamentaux du GOF

6.1. Singleton

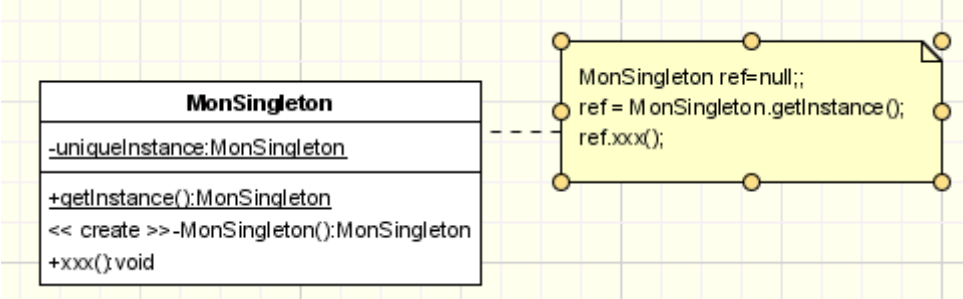
Objectif : Une et une seule instance pour une classe donnée.

Solution classique:

- variable de classe (static) **uniqueInstance** =null
- méthode de classe (static) **getInstance()**

```
{ if (uniqueInstance == null) {
     uniqueInstance = new Xxx();
    }
  return uniqueInstance; // instance nouvellement ou anciennement créée .
}
```

- **constructeur privé** pour interdire des "***new***" directs sans passer par *getInstance*()



NB : il faudra penser à ajouter **"synchronized"** (ou autre) sur ***getInstance*()** dans un **contexte "multi-thread" .**

Avantages du singleton:

Etre sûr qu'une seule instance sera utilisée à un niveau donné permet:

- **d'optimiser la mémoire**
- **de gérer un contexte (avec données partagées) à un endroit central bien précis (initialisation , lecture/écriture , ...)**

D'autre part, **l'appel d'une méthode statique est très pratique pour récupérer l'instance unique depuis un endroit quelconque du programme**.

Le Singleton est assez souvent utilisé sur les fabriques et les façades (une seule instance suffit souvent)

Eventuels inconvénients:

Effet "Kitchen Sink" (siphon) ==> Le singleton ramène à lui tous les appels "static" , ce qui peut poser quelques problèmes si les choses doivent évoluées (changement ? changement partiel ? , ...).

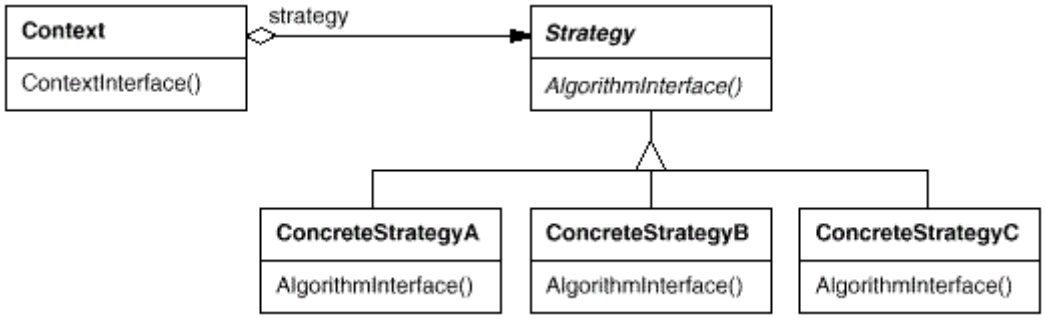
--> On peut dans certains cas , mettre en oeuvre des "pseudo singleton" liés à un contexte bien précis (ex : contexte Spring ou module angular ou ...)

Exemple de code (Singleton) :

```
public class ProduitDaoFactory {
private static ProduitDaoFactory uniqueInstance = null;

public synchronized static ProduitDaoFactory getInstance() {
                if(uniqueInstance==null){
                        uniqueInstance=new ProduitDaoFactory();
                }
                return uniqueInstance;
        }

private ProduitDaoFactory()          {        super();   }

private String commonData; //shared inside singleton (with get/set)
public String getCommonData() { return commonData;     }
public void setCommonData(String commonData) { this.commonData = commonData; }
//...
}
```

```
private void subTestSingleton(){
        ProduitDaoFactory produitDaoFactory = ProduitDaoFactory.getInstance();
        String data = produitDaoFactory.getCommonData();
        Assert.assertTrue(data.equals("my shared data inside singleton"));
}

@Test
public void testSingleton(){
        //ProduitDaoFactory prodDaoFactory = new ProduitDaoFactory();
                                                //impossible si constructeur privé
        ProduitDaoFactory prodDaoFactory = ProduitDaoFactory.getInstance();
        prodDaoFactory.setCommonData("my shared data inside singleton");
        subTestSingleton();
}
```

6.2. Stratégie

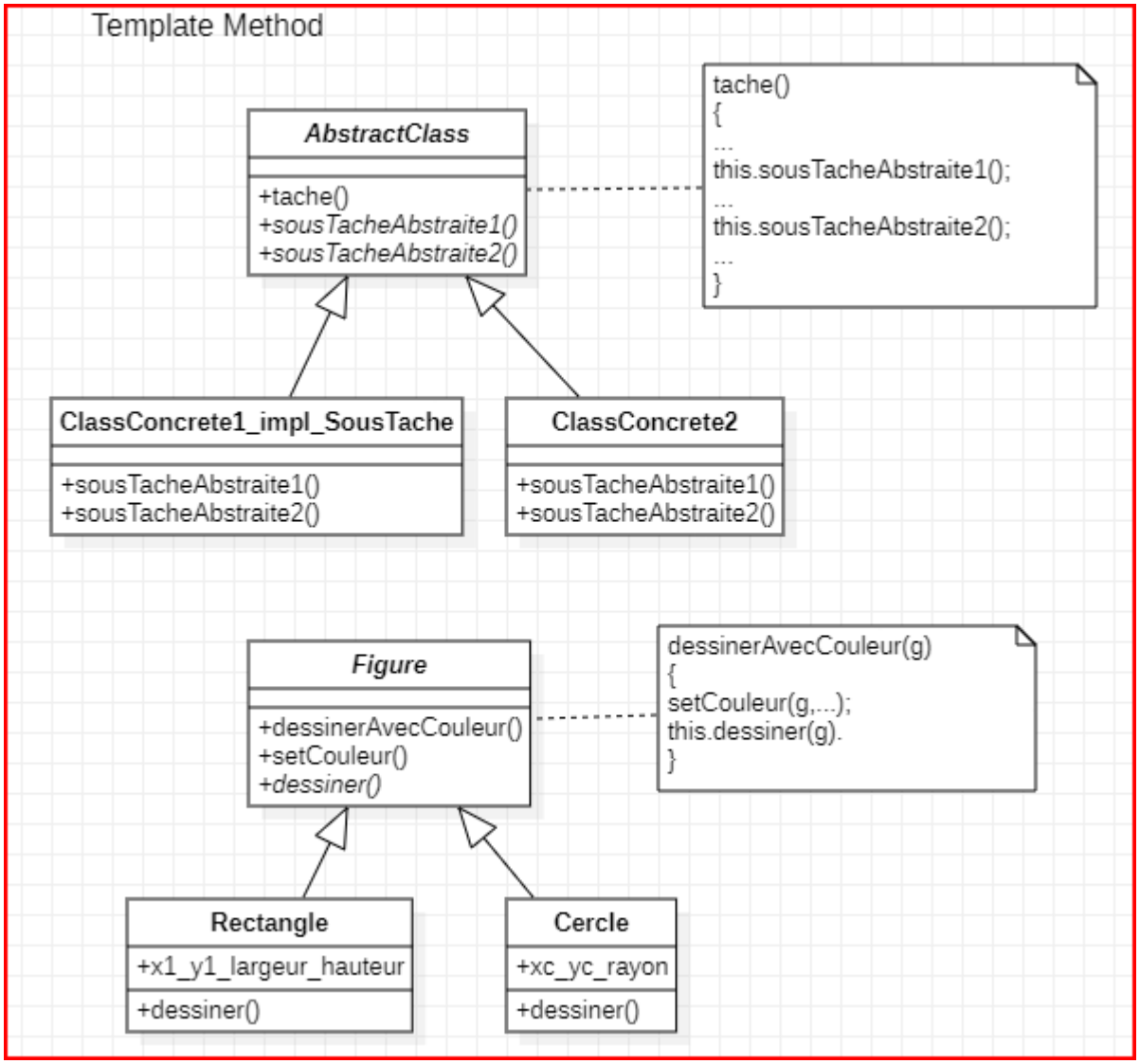


NB: Le design pattern "***stratégie"*** met non seulement l'accent sur le ***polymorphisme*** mais également sur le fait d'***externaliser une certaine responsabilité annexe vis à vis du contexte de départ***.
C'est généralement une très bonne idée favorisant nettement la **modularité** de l'ensemble.

NB: Le design pattern "***D.A.O.***" (Data Access Object) peut être vu comme un cas particulier de stratégie (une stratégie d'accès aux données)
"DataSource" est une stratégie pour établir une connexion à une base de données .

6.3. Patron/modèle de méthode (Template method)

Factoriser ce qui est commun , différentier le spécifique .



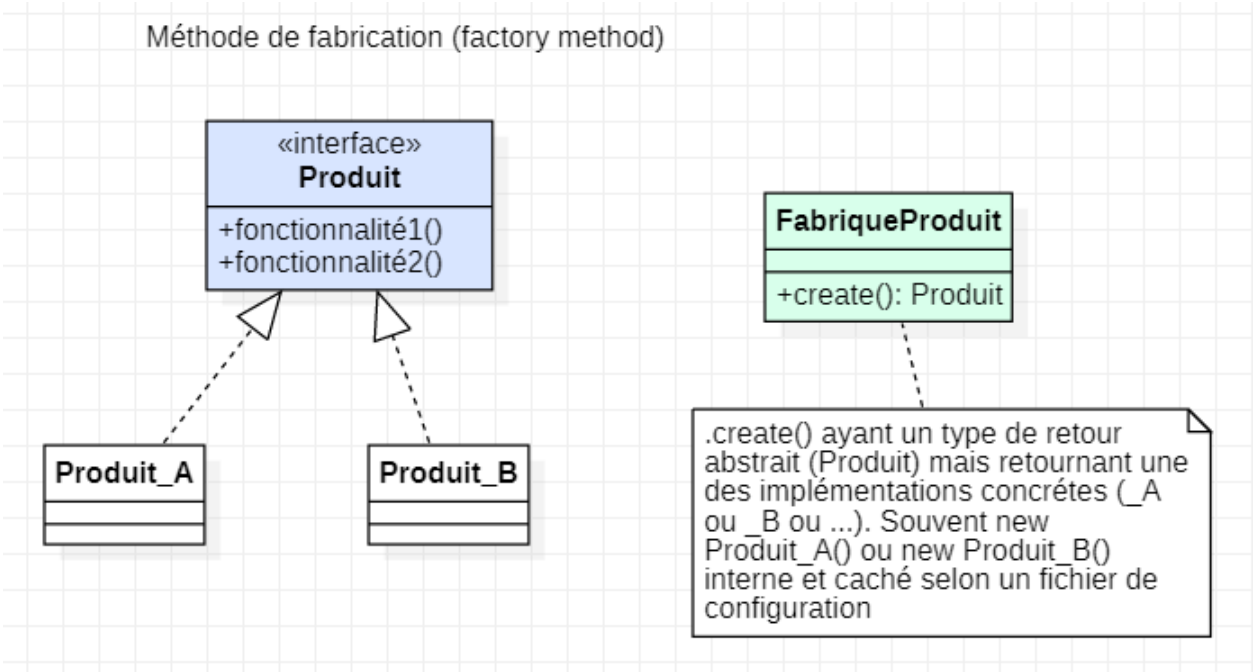
6.4. Méthode de fabrication (factory method)

Problème courant: comment masquer le type exact d'un objet à créer ainsi que les détails de son initialisation ?

Solution courante: Utiliser une méthode de fabrication sur un objet intermédiaire (souvent appelé "fabrique"). On évite ainsi une instanciation directe du genre xxx = new CXxx("v1","v2");

Ce code est alors caché au sein d'une méthode " .create() " éventuellement "static" .

Le code interne de ".create()" peut alors utiliser des mécanismes quelconques (ex: fichier de configuration) pour créer et initialiser un composant CXxxV1 ou CXxxV2 et le retourner ensuite de façon abstraite (type de retour de .create() = interface IXxx)



Variantes: le code interne de la méthode de fabrique (ex: create()) peut :

- instancier une nouvelle instance via new
- retourner un objet pré-construit et rangé dans un pool
- déclencher une instanciation générique via Class.forName("package.NomClasse").newIntance()
- ....

Exemple :

produitDao.properties

```
#produitDao=tp.dao.ProduitDaoSimu
produitDao=tp.dao.ProduitDaoJdbc
#produitDao=tp.dao.ProduitDaoJpa
```

```
public class ProduitDaoFactory {
//...
public ProduitDao createProduitDao(){
    ProduitDao dao =null;
    String daoImplClassName = MyPropertiesUtil.propertyValueFromEntryOfPropertyFile(
                                        "produitDao.properties","produitDao");
try {
        logger.info("daoImplClassName="+daoImplClassName);
        /*
        if(daoImplClassName.equals("tp.dao.ProduitDaoSimu") )
                dao = new tp.dao.ProduitDaoSimu();
        else if(daoImplClassName.equals("tp.dao.ProduitDaoJdbc") )
                dao = new tp.dao.ProduitDaoJdbc();
        else if(daoImplClassName.equals("tp.dao.ProduitDaoJpa") )
                dao = new tp.dao.ProduitDaoJpa();
        */
        dao = (ProduitDao) Class.forName(daoImplClassName)
                                .getDeclaredConstructor().newInstance();
} catch (Exception e) {
        e.printStackTrace();
}
      return dao;
}
}
```

```
package tp.util;

import java.io.InputStream;
import java.util.Properties;

import org.slf4j.LoggerFactory;

public class MyPropertiesUtil {
	private static org.slf4j.Logger logger = LoggerFactory.getLogger(MyPropertiesUtil.class);

	public static Properties propertiesFromCPRelativePathFile(String relativePathFile){
		Properties props = new Properties();
		try {
		InputStream inStream =  Thread.currentThread().getContextClassLoader()
		                                .getResourceAsStream(relativePathFile);
		props.load(inStream);
		} catch (Exception e) {
		logger.error("cannot load properties file : " + relativePathFile , e.getMessage() );
		}
		return props;
	}

	public static String propertyValueFromEntryOfPropertyFile(
		String relativePathPropertyFile,    String propertyName)
	{
		String propValue=null;
		Properties props = propertiesFromCPRelativePathFile(relativePathPropertyFile);
		propValue=props.getProperty(propertyName);
		return propValue;
	}
}
```

NB : **Le design pattern "injection de dépendances"** (alias "ioc") présenté dans le chapitre "design patterns JEE et n-tiers" **est une évolution du design pattern "factory" (sorte de "méga fabrique" qui fabrique et assemble plein de composants)** .

6.5. Facade

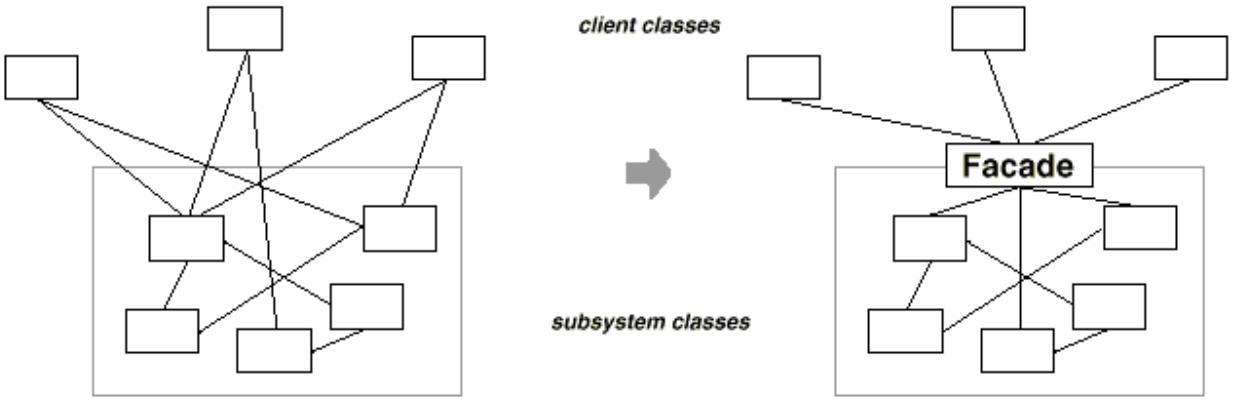
Le design pattern "façade" comporte plein de variantes.

Au sens le plus général, il s'agit de mettre en place une façade (simple à utiliser) de façon à indirectement bénéficier des services rendus par différents éléments d'un module (quelquefois complexe).

- Une façade de type "accueil/redirection" est une facade d'entrée unique pour un module complet ==> on est alors dépendant que d'un seul élément central (qui souvent redirige).
- Une façade "technologiquement agnostique" est un objet (que l'on instancie via un simple new) et qui cache toutes les technologies utilisées en arrière plan (ex: Spring, ...) .

Problème à résoudre (GOF):

Un module sans façade comporte de multiple points d'entrée que l'on est obligé de connaître pour initialiser les appels. D'autre part, ce n'est pas facile de s'y retrouver en cas d'évolution (quelles répercutions suite à tel changement ?).



Solution générale:

Introduire un nouvel élément intermédiaire qui jouera le rôle de "façade / accueil / point d'entrée unique" pour ce module.

Variantes de la solution:

Variantes	*Caractéristiques*	*Astuces (?)*
Nouvelle classe reprenant toutes les méthodes de toutes les classes internes (délégation en interne)	cette classe peut devenir énorme si le module est grand.	solution monolithique pas très évoluée
Simple intermédiaire de type "accueil" redirigeant vers les classes internes du module	ceci permet de ramener à 1 le nombre de point d'entrée à connaître mais le code des appels est, après redirection de l'accueil, de nouveau directement dépendant des éléments internes du module.	solution moyenne : accueil sans abstraction.
Intermédiaire de type "accueil/façade/fabrique" redirigeant vers des éléments concrets internes qui sont retournés comme des éléments abstraits .	Un seul point d'entrée à connaître + vision externe abstraite (pouvant être indépendante de la structure interne du module) ==> évolution interne facilitée.	**solution astucieuse (très bien)** mais un peu longue à programmer (interfaces , façade , ...)

NB : On appelle "**façade agnostique**" , une façade qui masque volontairement la technologie utilisée en interne dans le module (exemple : "spring" ou "cdi") .

NB : "agnostique" (signifiant littéralement "sceptique en matière de métaphysique ou de religion") est ici à prendre au sens "je ne crois pas à la prédominance de la technologie "spring" ou "cdi" pour encore les 20 ans à venir" et je préfère basé mon code sur les fondamentaux du langage de programmation ("static" , "singleton" , "constructeur" , ...)

Exemple :

```
package tp.service;
public interface MyFacade {
        //GestionProduits , GestionConversion et GestionTva sont ici 3 interfaces de services
        public GestionProduits getGestionProduits();
        public GestionConversion getGestionConversion();
        public GestionTva getGestionTva();

        public void cleanUpResources(); //si nécessaire (librérer ressources internes)
}
```

```
public class MyFacadeImpl implements MyFacade {

   private GestionConversion gestionConversion=null;
   private GestionTva gestionTva=null;
   private GestionProduits gestionProduits=null;
  //... + singleton + ...

private MyFacadeImpl(){
 //En version "Agnostique", une façade cache entièrement la technologie "Spring" ,
 //  "CDI" ou "IOC maison" qui prend en charge les composants derrière la façade

 MyBeanFactory myIocFactory = MyBeanFactory.getInstance();
 myIocFactory.initIocConfig();
 this.gestionConversion =  myIocFactory.getBean(GestionConversion.class);
 this.gestionTva =         myIocFactory.getBean(GestionTva.class);
 this.gestionProduits =    myIocFactory.getBean(GestionProduits.class);
 }

public GestionProduits getGestionProduits() {return gestionProduits;}
public GestionConversion getGestionConversion() {return gestionConversion;}
public GestionTva getGestionTva() {return gestionTva;}

public void cleanUpResources() {
                gestionProduits.cleanUpResources(); //à adapter avec Spring ou autre sur vrai projet
  }
}
```

Exemple d'utilisation de la façade :

```
MyFacade myFacade = MyFacadeImpl.getInstance();
double sommeFrancs = myFacade.getGestionConversion().euroToFrancs(15);
System.out.println("15 euros : " + sommeFrancs + " francs");
double sommeTva = myFacade.getGestionTva().getTva(20.0 , 200);
System.out.println("TVA (20.0) pour 200 Euros ==>" + sommeTva);
myFacade.cleanUpResources();
```

Relativité de l'importance des façades :

La mise en oeuvre du design pattern "façade" est généralement intéressante sur un gros projet (application de grande taille) .

Associer façade et "business delegate" (voir chapitre "design patterns pour jee et n-tiers") peut éventuellement être intéressant pour bien structurer le code d'une application cliente (ou middleware) déléguant des appels vers de nombreux serveurs .

L'aspect "agnostique" n'est généralement intéressant que s'il est appliqué à fond.

Mieux vaut ne pas appliquer de "semi indépendance technologique" si 80 % de l'application est à fond basé sur un framework bien spécifique du type "CDI" ou "Spring" .

6.6. Décorateur (enveloppe transparente)

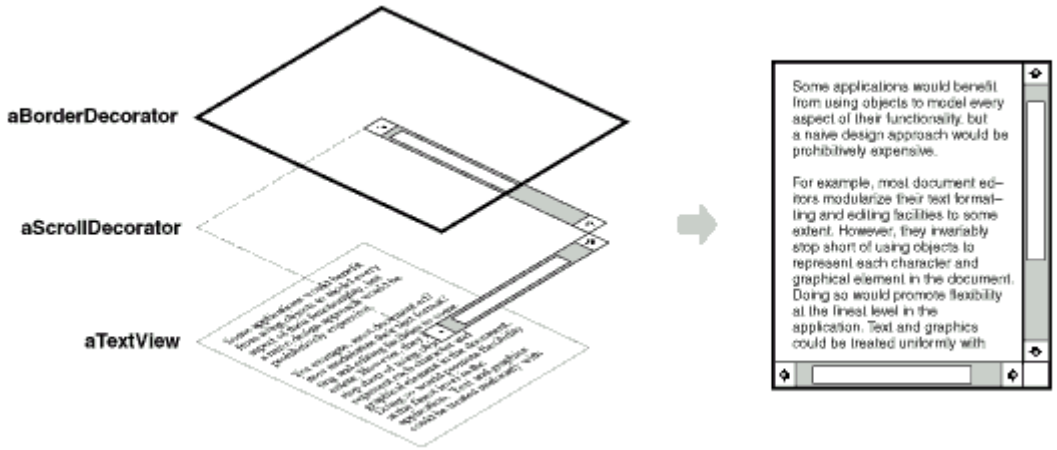
Provenance du nom "décorateur":

Soit "Composant Visuel" une classe correspondant à un composant graphique (sorte de fenêtre) de base.
Cette classe comporte les méthodes classiques "Afficher() , …"

On souhaite maintenant **manipuler de façon uniforme** différents types de composants graphiques avec de nouvelles fonctionnalités:

- Vue (éditeur) de texte (toute simple).
- Vue de texte avec ascenseurs et gestion automatique du scrolling.
- Vue de texte avec bordures de redimensionnement automatique et titre.
- Vue de texte combinant bordures et ascenseurs.

NB: les éventuelles décorations (ascenseurs, bordures) sont toujours gérées automatiquement.

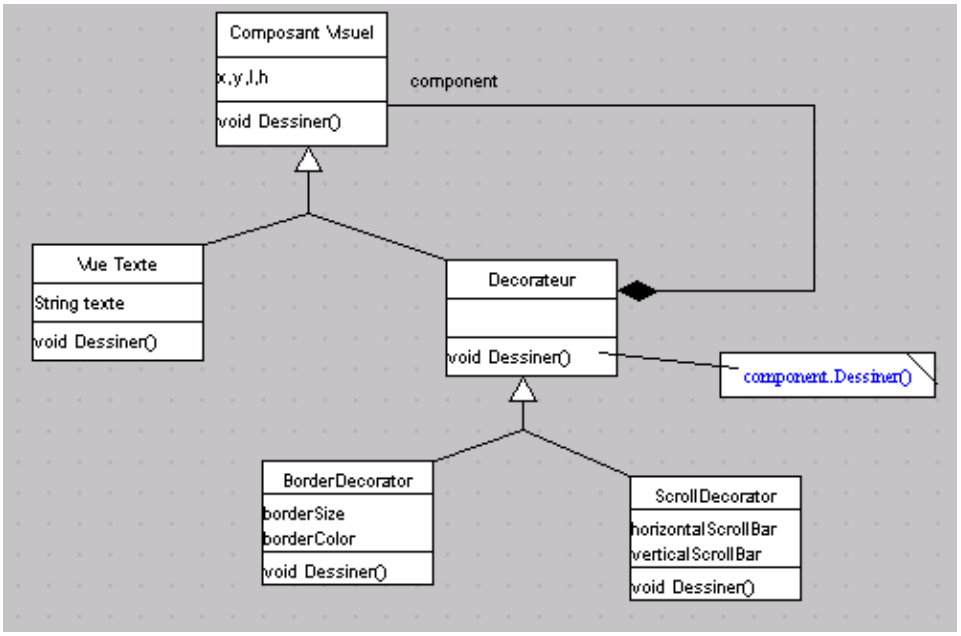


Solution intuitive mais moyenne:

N'utiliser que l'héritage et créer plein de classes telles que "Composant graphique avec bordures" , "Composant graphique avec Ascenseurs" , …

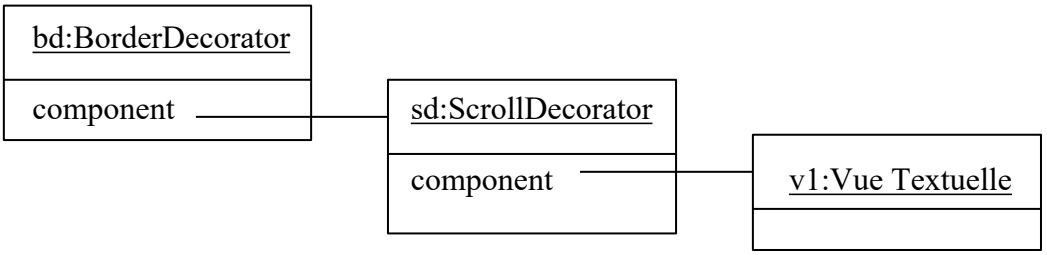
Problème: cette solution est un peu trop statique (figée). Si l'on souhaite une classe combinant bordures et ascenseurs , il faut effectuer un nouvel héritage (éventuellement multiple).

*Solution plus flexible proposée (Design Patter "**Decorator**"):*



Il suffit d'effectuer une série d'imbrications pour obtenir une vue avec bordures et ascenseurs.

D'autre part, la classe Décorateur hérite de "Composant Visuel" et est donc vue comme un composant visuel ordinaire (bien qu'il ait une composition en interne).



Sémantique "enveloppe transparente":

Le design pattern ci-dessus correspond à la notion de "**enveloppe transparente**".

Le code d'utilisation ne voit l'objet manipulé que comme un composant de base ordinaire.

L'**objet réellement manipulé** peut en fait être une **enveloppe intermédiaire** qui:

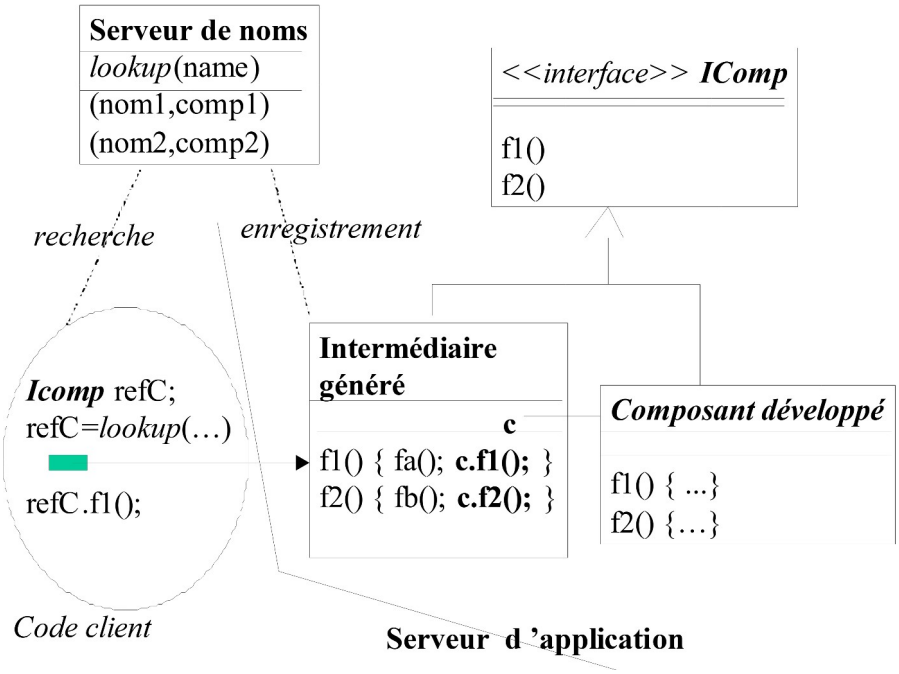
- **ajoute de nouvelles fonctionnalités**
- **délègue les traitements de base au niveau de l'objet imbriqué**.

Quelques exemples concrets du Design pattern "Decorateur":

- classe **JScrollPane** de Java/Swing
- classe ***BufferedReader*** et ***InputStreamReader*** de ***java.io*** ,

Interposition/intercepteur dans certains serveurs d'applications :

La plupart des serveurs d'applications (Jboss-wildfy-ou-eap , WebLogic, WebSphere, ….) utilisent des astuces dérivées de ce design pattern pour ajouter de nouvelles fonctionnalités aux composants que l'on déploie dedans.



Nouvelles fonctionnalités couramment apportées: gestion des transactions, de la sécurité et de la montée en charge.

Limitations et extension:

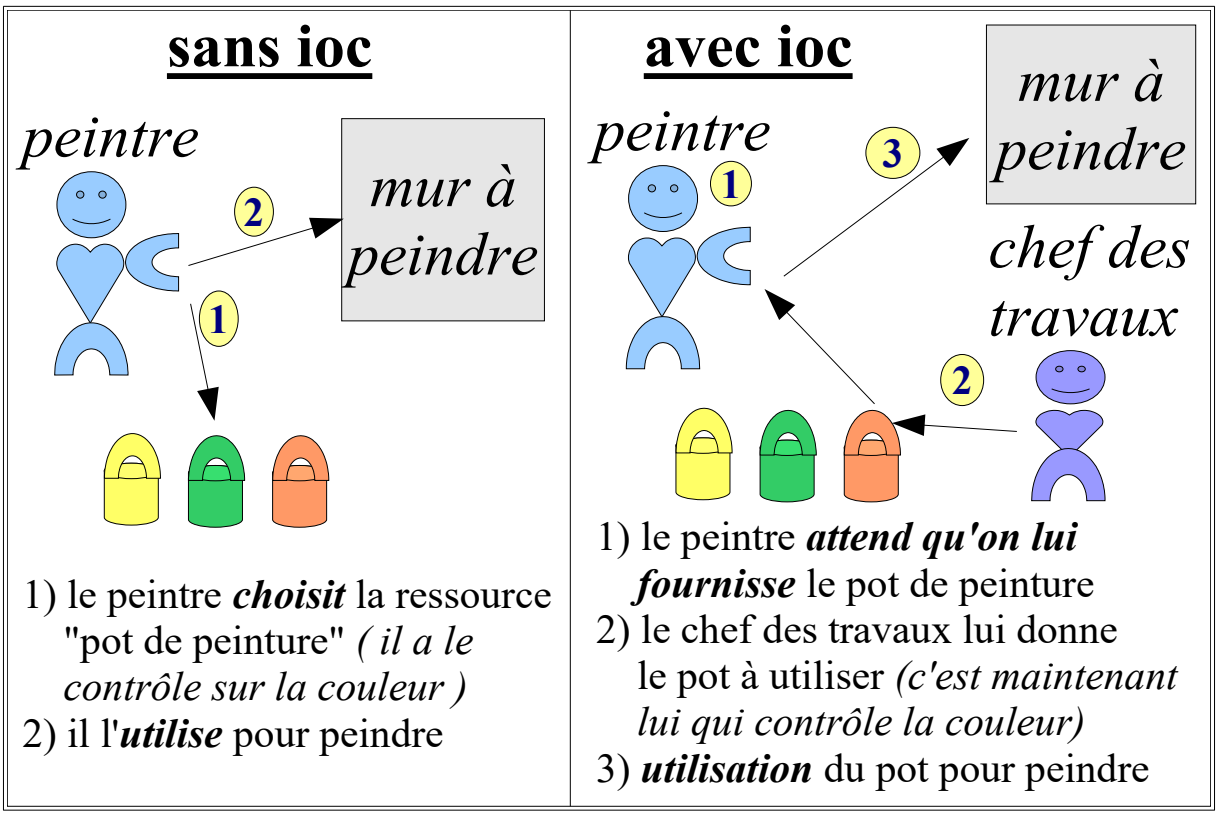
Pour être applicable , le design pattern décorateur doit partir d'un élément de base bien défini qu'il faudra envelopper.
Cet élément de base doit comporter un certain nombre de méthodes à conserver dans les décorateurs. **Ce paquet de méthodes doit être en nombre fini et les prototypes de ces méthodes doivent être connus et stables**.

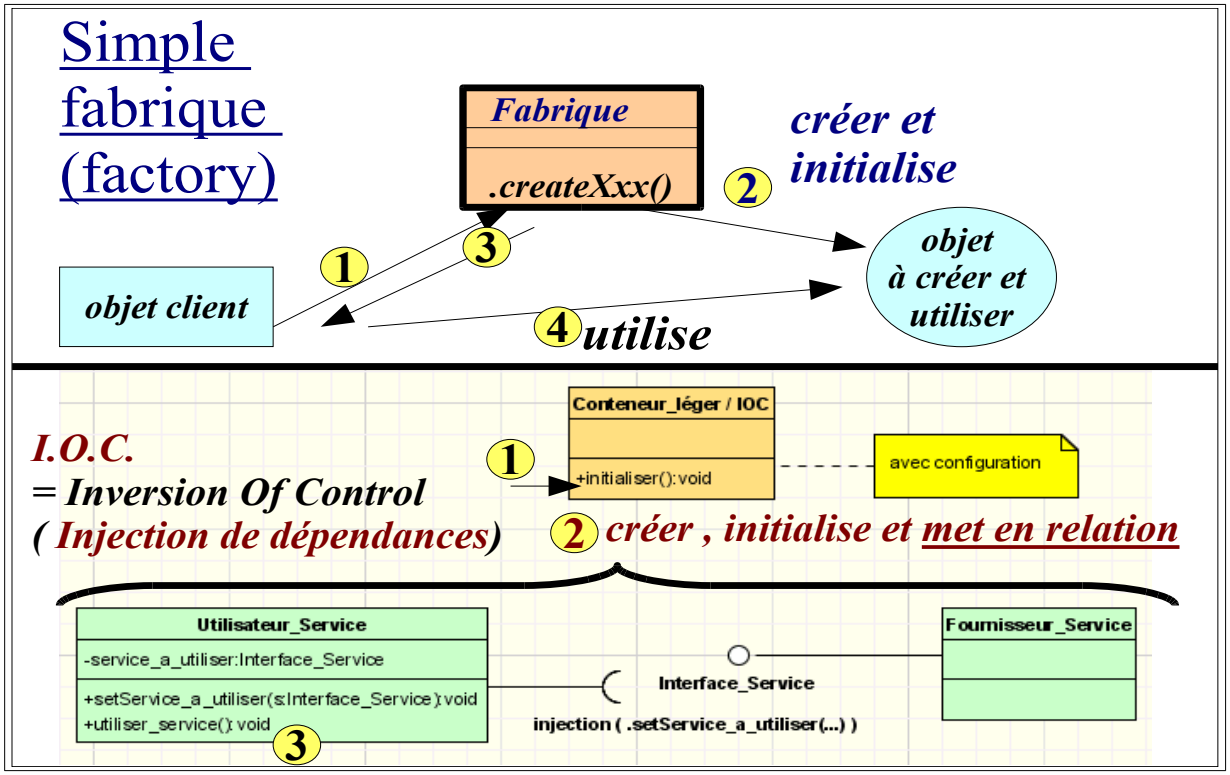
Lorsque les prototypes des méthodes ne sont pas connus et/ou lorsque le nombre de méthode de l'élément de base peut varier on doit alors utiliser la ***programmation par aspects (A.O.P.)***.

…

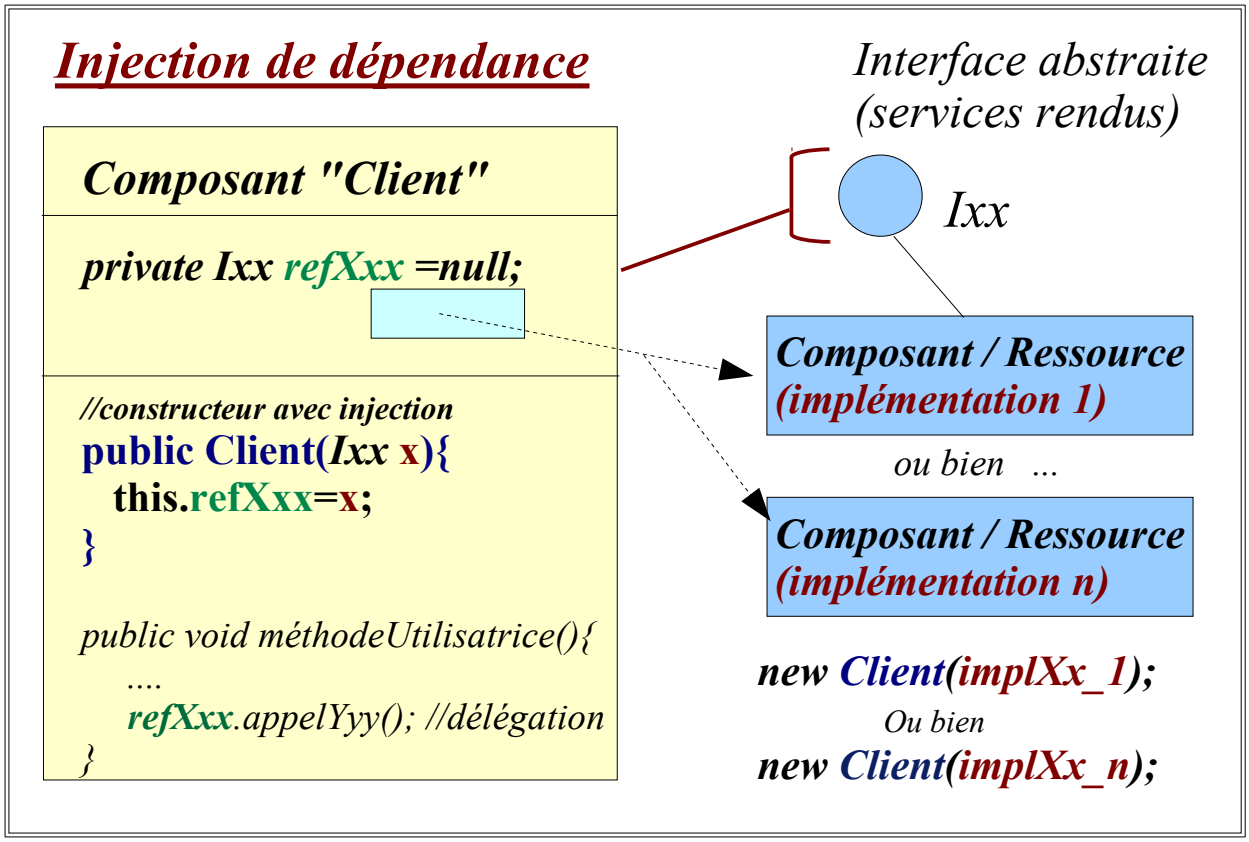
7. Design Pattern "I.O.C." / injection de dépendances

7.1. IOC = inversion of control





7.2. injection de dépendance



Le ***design pattern "IOC"*** (*Inversion of control*) correspond à la notion d'**injection de dépendances abstraites**.

Concrètement au lieu qu'un composant "client" trouve (ou choisisse) lui même une ressource compatible avec l'interface Ixx avant de l'utiliser , cet **objet client**
exposera une méthode de type:

```
public void setRefXxx(Ixx res)
```

ou bien un constructeur de type:

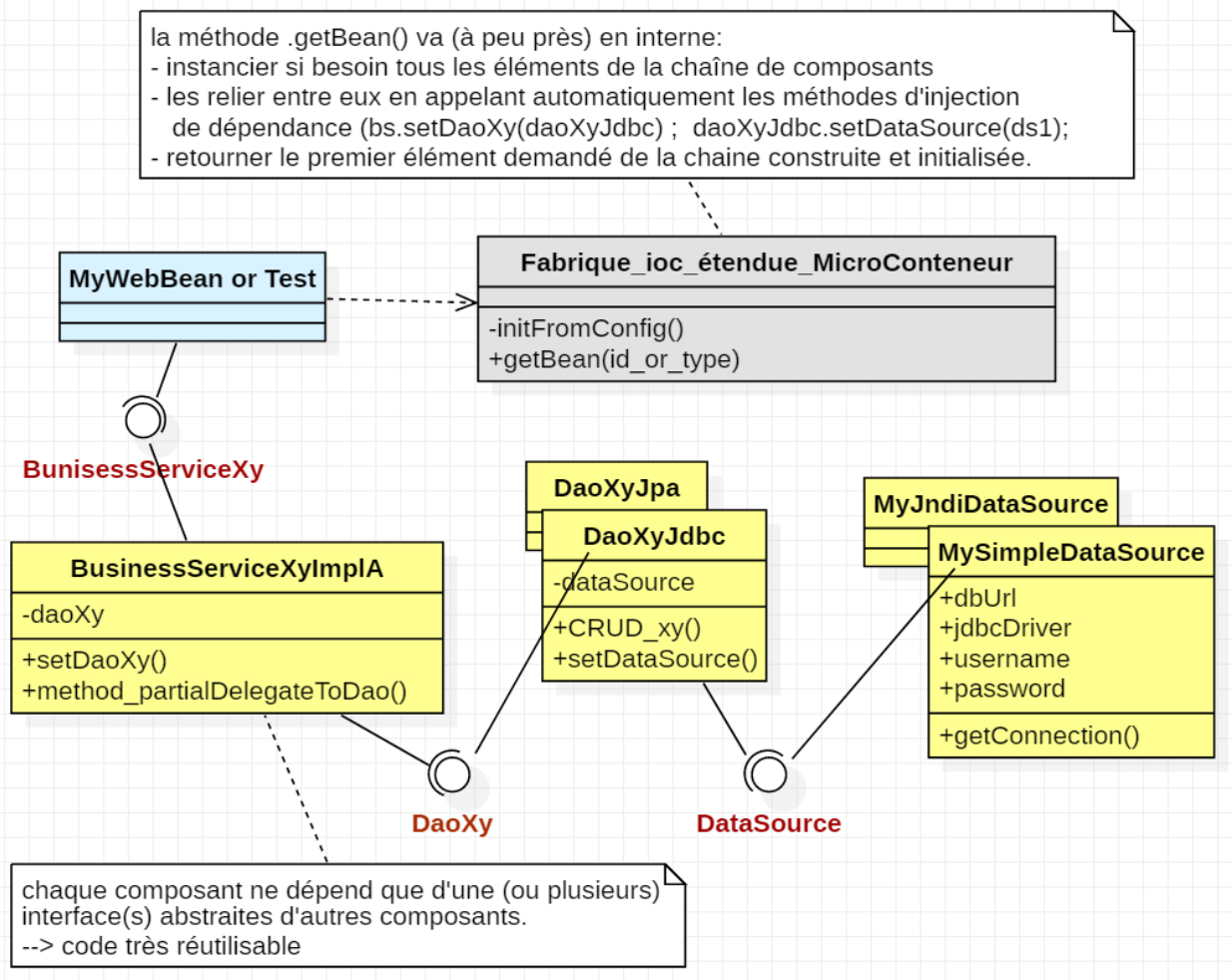
```
public Client(Ixx res)
```

ou bien une référence annotée de type :

```
@Autowired ou bien @Inject
private Ixx refXxx;
```

permettant qu'on lui fournisse la ressource à ultérieurement utiliser. Un tel composant sera ainsi très réutilisable .

7.3. avec conteneur I.O.C. (super fabrique globale)



7.4. Micro-kernel / conteneur léger

Pour être facilement exploitable , le design pattern "injection de dépendances" nécessite un **petit framework** généralement appelé "**micro-kernel**" ou "**conteneur léger**" prenant à sa charge les fonctionnalités suivantes:

- **Enregistrement des "ressources"** (composants concrets basés sur interfaces abstraites) avec des **identifiants** *(noms logiques)* associés.

- **Instanciation et/ou initialisation des composants en tenant compte des dépendances à injecter (==> liaisons automatiques avec composants "ressources" nécessaires)**

Ceci nécessite **quelques paramétrages** (***fichier de configuration XML*** ou bien ***annotations*** au sein du code ou bien via une ***configuration spécifique (java, implicite ou explicite, ...)***).
…

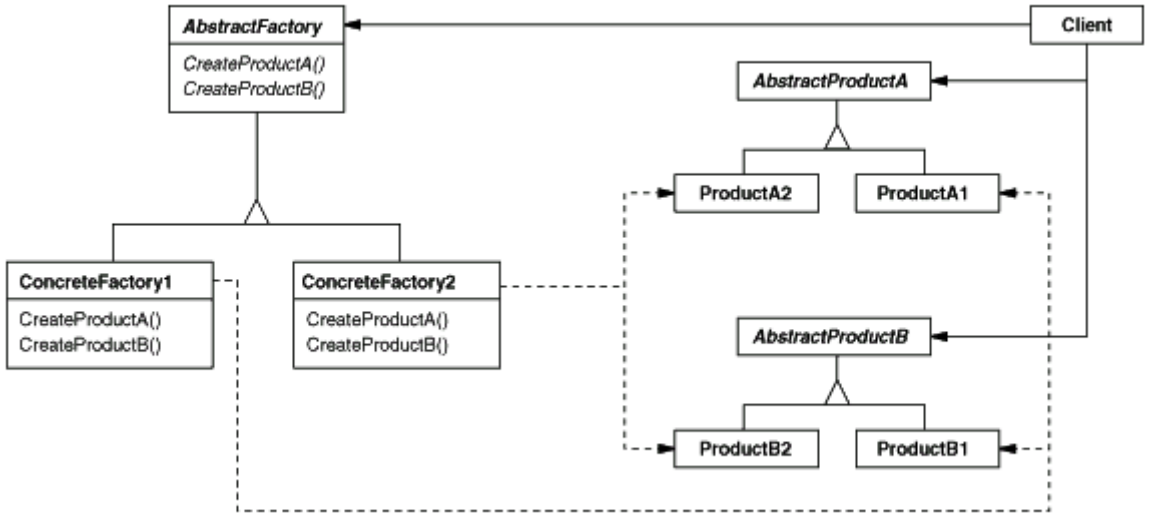
8. Quelques Design Pattern plus spécifiques en détails

8.1. Fabrique abstraite (abstract factory)

Frabrique = objet en fabriquant d'autres . Une classe de fabrique peut elle même implémenter une interface abstraite .

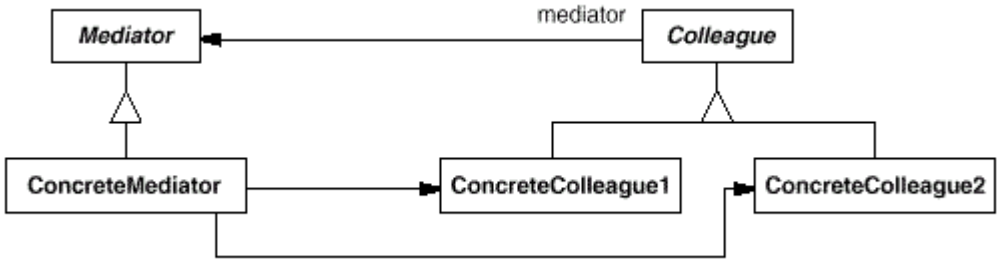
On peut alors envisager une hiérarchie de fabriques (abstraite et concrètes) associée à une hiérarchie de produits (abstrait et concrets).

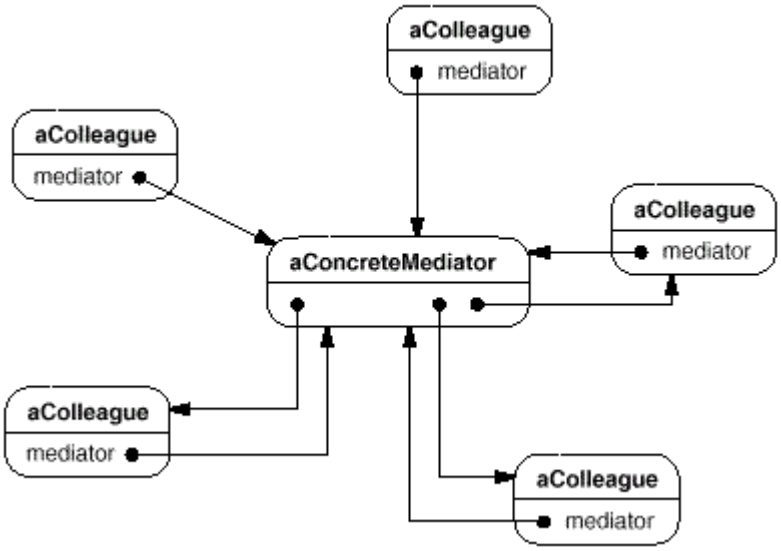
==> en basculant de type de fabrique et en re-déclenchant le processus de fabrication , on recréer alors automatiquement toute une famille de produits dans une autre version (ex: autre "look & feel" , autre implémentation , ...).



...

8.2. Médiateur (à peaufiner via le principe d'inversion de dépendances)





==> Evite une multitude de dépendances directes entre collègues.

==> Le médiateur (concret) devrait idéalement être injecté de façon abstraite au sein des collègues [principe d'inversion des dépendances / logique événementielle / ...]

NB :

On peut mettre en oeuvre des médiateurs internes au sein d'une application de grande taille.

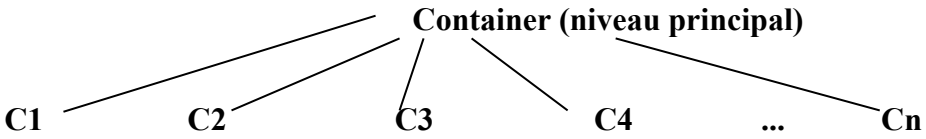
On peut également mettre en oeuvre des médiateurs entre petites ou moyenne applications communiquant entre elles : c'est le principe des serveurs intermédiaires de type "EAI" ou "ESB" (architecture SOA et éventuellement micro-services) .

8.3. Introduction au D.P. "Observateur" (vues cohérentes sur mêmes données, ...) et logique événementielle

Problème à résoudre: comment rendre facilement cohérentes n vues sur un même document (paquet commun de données) ? ==> une mise à jour partielle d'une vue et du document associé doit être répercutée sur toutes les autres vues.

Autrement dit,considérons que"composant" soit içi un synonyme de "vue" --->

De quelle manière doit procéder un composant C1 pour déclencher des mises à jour cohérentes dans des composants frères C2, C3, ..., Cn ?



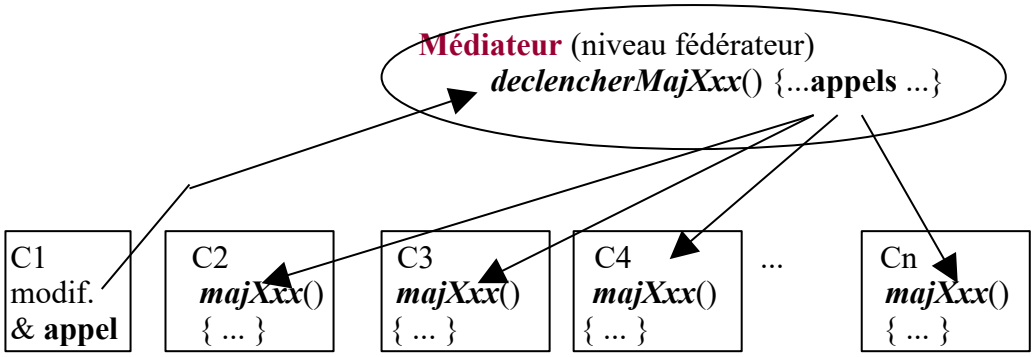
Solution 1 (pas bien !!!)

Suite à un modification interne , le composant C1 appel directement une fonction du genre ***majXxx*** sur chacun des autres composants C2, C3, ... Cn .

Cette solution est assez mauvaise car le composant C1 devient ainsi dépendant des composants C2, C3 et Cn. Si chaque composant fonctionne de cette façon , on se retrouve avec une multitude de dépendances dans tous les sens => le logiciel sera très difficilement maintenable (évolutions laborieuses) . Cette mauvaise architecture est souvent appelée plat de nouilles ou usine à gaz.

Solution 2 (déjà mieux)

Suite à un modification interne , le composant C1 appel une fonction du genre *declencherMajXxx*() sur son conteneur parent (ex: fenêtre mère). Cet élément principal (quelquefois appelé ***médiateur***) va alors réagir en appelant une fonction du genre *majXxx*() sur chacun des autres composants C2, C3, ... Cn .



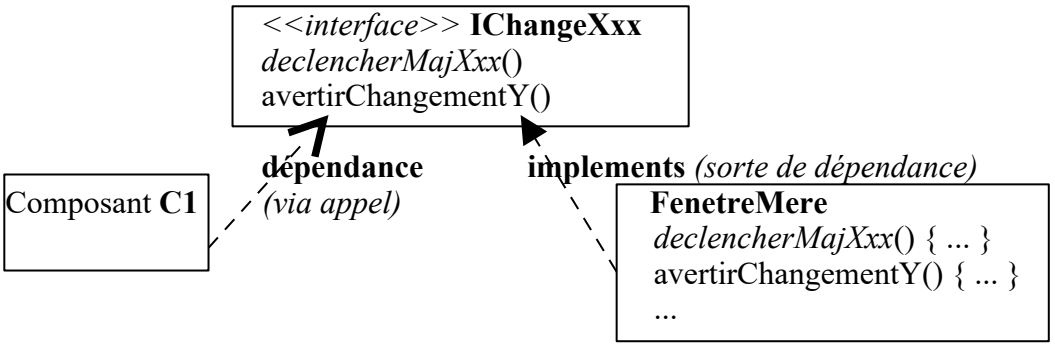
Cette solution est déjà beaucoup mieux structurée car les sous composants C1, C2, ... Cn ne se voient pas directement et sont donc complètement indépendants.

Reste tout de même un problème non négligeable: le composant C1 devient fortement dépendant du niveau principal (conteneur parent) et peut donc difficilement être réutilisable dans un autre contexte.

Solution 3 [composants avec logique événementielle] (bien !!!)

La meilleur des solutions au problème consiste à ***peaufiner la solution 2*** en y inversant la dépendance entre C1 et son conteneur parent (niveau fédérateur) . **L'inversion d'une dépendance s'effectue en introduisant une interface**:

Au lieu d'appeler explicitement la fonction ***declencherMajXxx()*** sur un type précis de conteneur parent , **le composant C1 se contente d'appeler cette fonction sur une chose vague dont le type n'est qu'une simple interface événementielle**.



Pour que ce mécanisme fonctionne, il faut que l'objet précis qui implémente l'interface *(et donc la fonction declencherMajXxx())* ***passe à C1 une référence sur lui même de façon à ce que C1 puisse plus tard y accéder***.
Par exemple une méthode de type ***add/setChangeXxxListener(IChangeXxx obj)*** permet d'enregistrer au sein du composant le (ou les) objet(s) sur le(s)quel(s) il faut déclencher des mises à jour.
On se retrouve donc dans le schéma événementiel classique du langage java (listener, ...).
....

==========

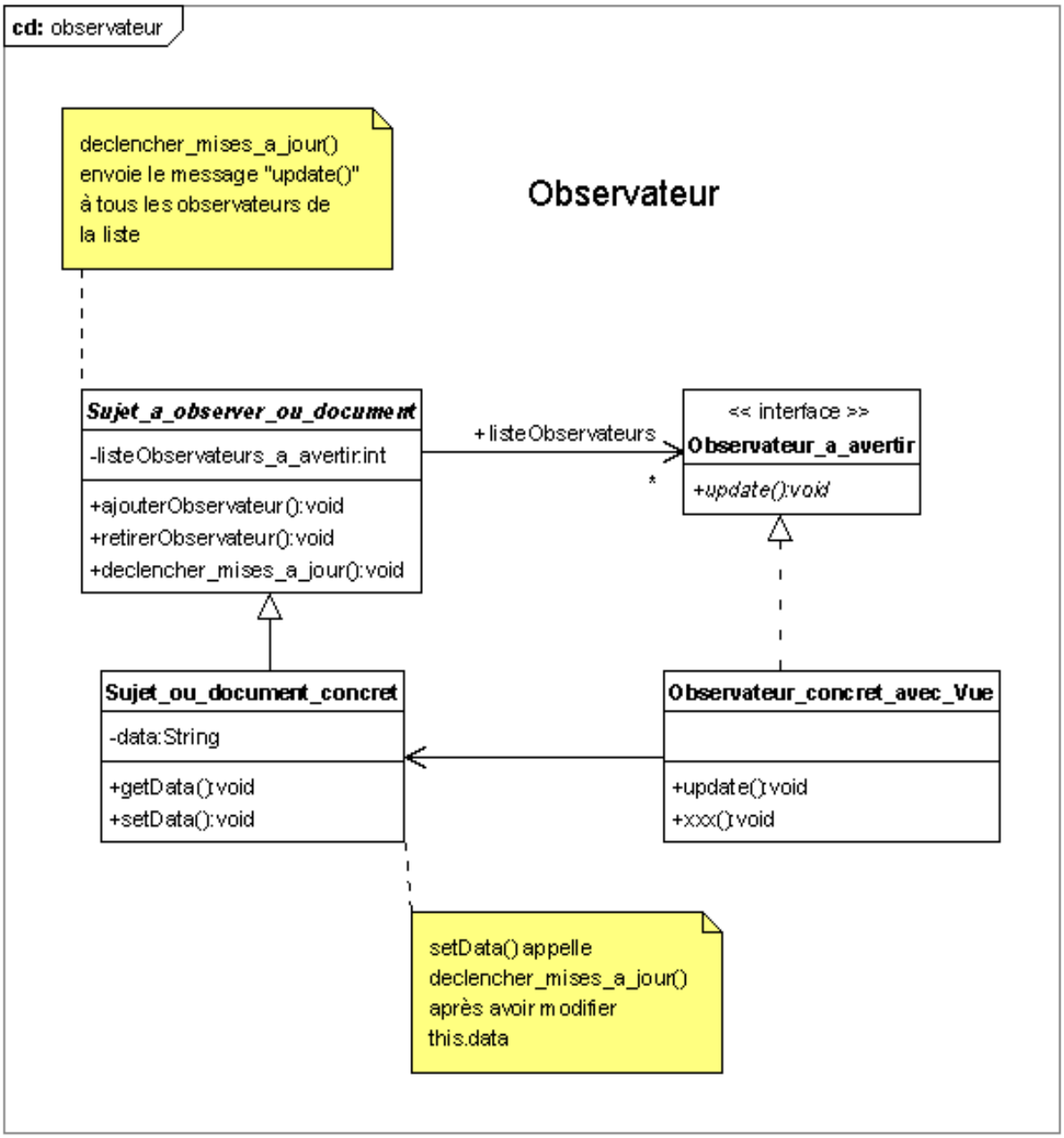
Dans la variante officielle du D.P. "observateur" (G.O.F.) , on ne parle pas en terme de composants devant indirectement déclencher des mises à jours sur d'autres composants mais on parle en terme de :

- **Observateurs** (ou *Vues*) modifiant quelquefois une partie d'un sujet (document) commun .
- **Sujet** (*document*) **observable** avertissant spontanément tous les observateurs qui se sont préalablement enregistrés comme tels qu'une mise à jour est à effectuer (du coté de leurs "vues").

.../...

8.4. design pattern "Observateur":

Variante classique sur les termes : ***Sujet Observable /Observateurs*** ou ***Document/Vues***



Quelques implémentations concrètes :

- le langage **java** comporte la classe java.util.**Observable** et l'interface java.util.**Observer**
- le langage **C#** (.net ≥ v4) comporte des **IObservable<T>** et **IObserver<T>** complexes
- Framework **RX-JS** (utilisé dans Angular2)
- …..

Exemple de code (partiel) **java** :

```
...
import java.util.Observable;

public class SubjectWithCommonData extends Observable {
        private String commonData;

        public String getCommonData() {    return commonData;          }

        public void setCommonData(String commonData) {
                this.commonData = commonData;
                this.setChanged();
                this.notifyObservers(); //avertir tous les observateurs qu'un changement
                 // a été effectué et INDIRECTEMENT DECLENCHER DES mises à jour
        }
}
```

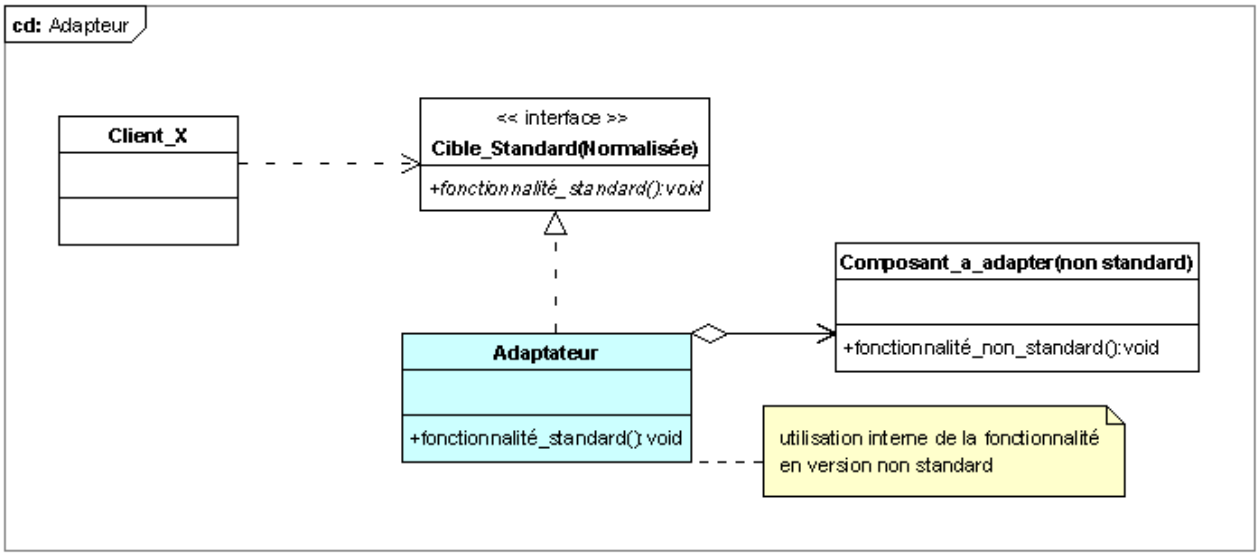
```
import java.util.Observer;  //version prédéfinie "java"
public class MyObserver1 implements Observer{
//...
private SubjectWithCommonData subjectWithCommonData;
        @Override
        public void update(Observable o, Object arg) {
         //arg est une éventuelle indication et o peut être "casté" en "subjectWithCommonData"
                String couleur = subjectWithCommonData.getCommonData();
                //mise à jour locales (au cas par cas)
        }
        public void setSubjectWithCommonData(SubjectWithCommonData subject) {
                this.subjectWithCommonData = subject;
                subject.addObserver(this);
                //POINT CLEF (enregistrement d'un observateur à ultérieurement avertir)
        }
}
```

8.5. Adaptateur

Problème courant: un composant "Ca" doit utiliser "Cx" mais l'interface de "Cx" ne convient pas . D'autre part, les composants "Ca" et "Cx" ont des interfaces figées que l'on ne peut pas modifier.

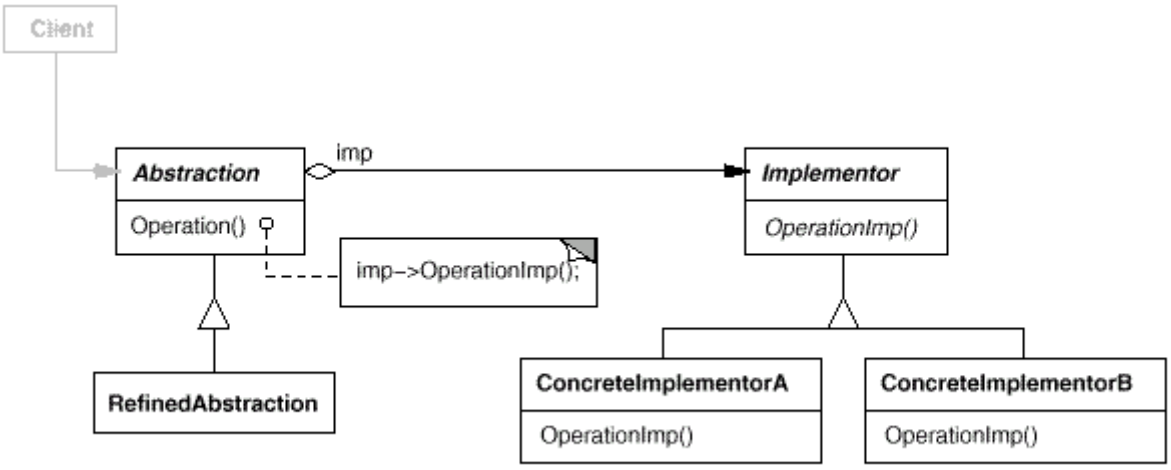
Solution ==> introduire un composant intermédiaire Ci qui va :

- implémenter l'interface attendue par l'appelant "Ca"
- re-déléguer en interne à Cx la plupart des appels/fonctionnalités



NB : L'adaptateur peut également être implémenté (lorsque c'est possible) via un héritage vis à vis du composant à adapter

8.6. Pont (bridge) (souvent pour multi-plateformes)



Le design pattern "bridge" ("pont") a été concrètement utilisé au sein de l'api "**AWT**" (Abstract Window Toolkit) des premières versions de java : derrière les représentations abstraites "Window" , "Button" , ... se cachaient des délégations des implémentations internes spécifiques à une plateforme **("linux/x-windows" ou "Microsoft Windows" ou "Mac")** .

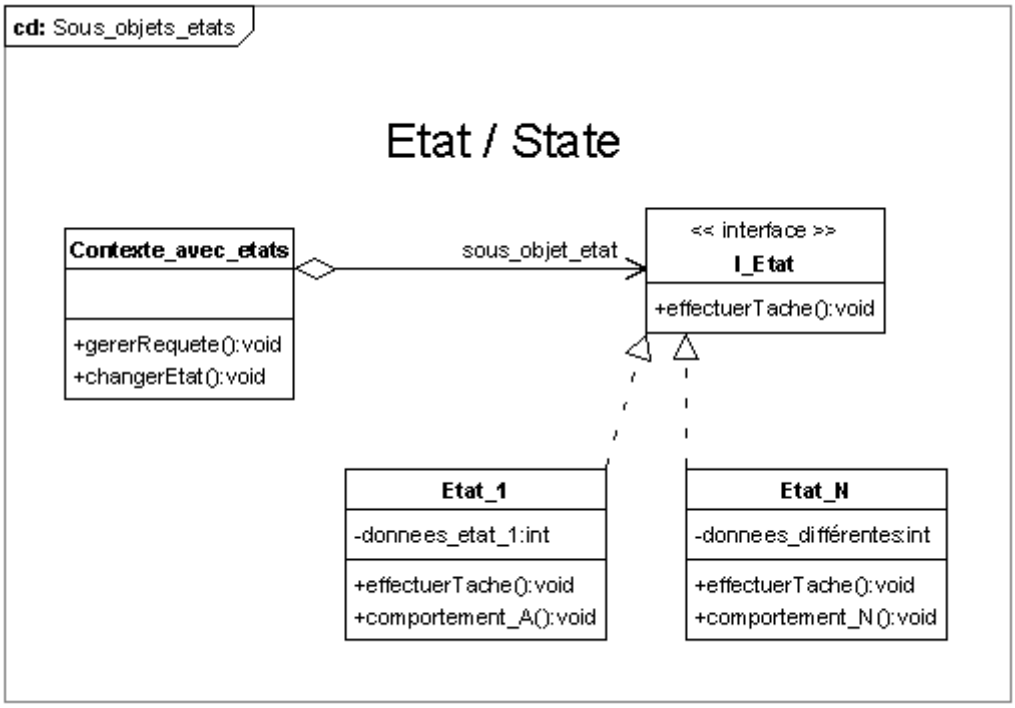
Certains frameworks beaucoup plus modernes **(ex : "react-native" , "flutter" , ...)** s'appuient partiellement sur cette idée pour qu'un code unique puisse (après un éventuel "build spécifique") **s'adapter à différentes plateformes mobiles ("Android" , "iPhone" , ...)**

Autrement dit, le design pattern "bridge/pont" est très souvent mis en oeuvre par des éditeurs de frameworks sophistiqués et beaucoup plus rarement utilisé dans l'implémentation d'une application fonctionnelle ordinaire.

Attention : Bien que pouvant apporter "certains bénéfices" au niveau de la portabilité (**multi-plateformes"**) , **la mise en oeuvre du design pattern "bridge" / "pont" nécessite un très gros travail** (beaucoup de lignes à coder et à maintenir) .

8.7. (sous objet) Etat (State)

Problème courant: un objet doit gérer en interne un certain nombre d'états (avec pour chaque état des comportements attendus différents)



Solution classique: **l'objet en question peut matérialiser chaque état comme un sous objet interne spécifique (un changement d'état se résume alors à changer de sous objet interne /switch).**

Exemple classique : objet IHM avec différents états (à chaque état peut correspondre un sous objet de type "écran" ou "fenêtre" ou "page HTML" ou ...).

ex1: CardLayout (java.awt)

ex2: diagramme d'états UML pour modéliser les différents états d'une IHM complexe (chaque état : écran ou action[traitement_ou_présentation] , transitions : navigation / conditions à respecter)

ex3 : <router-outlet> et switch entre sous-composants "angular" (routing/navigation en mode "Single Page Application") .

8.8. Prototype (à cloner)

Au lieu de créer de nouvelles instances via un code inélégant du genre :

```
switch(type_xy_to_create){
case TYPE_CERCLE :
   obj = new Cercle() ; break ;
case  TYPE_LIGNE :
   obj = new Ligne() ; break ;
case  TYPE_RECTANGLE :
   obj = new Rectangle() ; break ;
....
}
```

On peut créer un ***pool d'instances prototypes*** dès l'initialisation de l'application.

On peut ensuite ranger ces instances dans une ***table d'association*** (pour que chaque type de prototype soit associé à un icône adéquat dans une "**palette**").

L'instanciation d'une nouvelle instance (en fonction du choix courant dans la palette) peut alors se faire via un code simplifié ressemblant à :

```
current_icone = evenement.getSource() ;
obj = mapPrototypes.get(current_icone).clone() ;
```

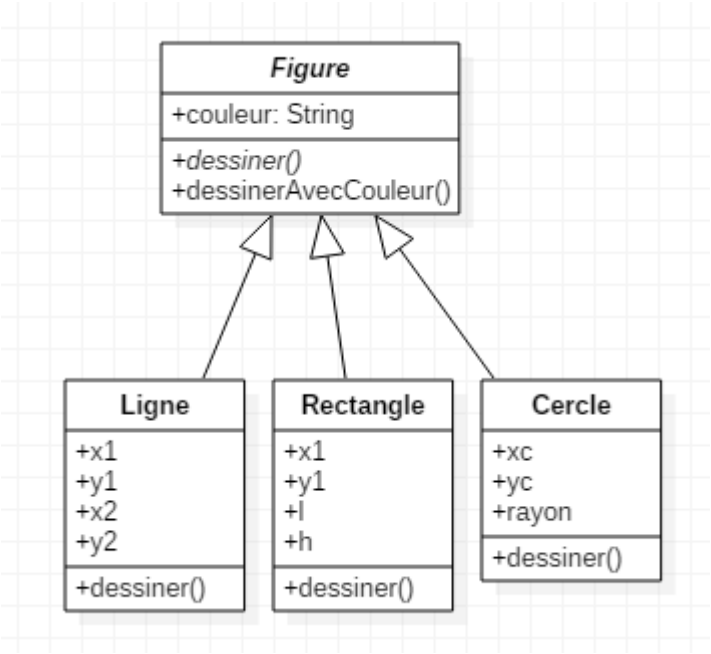
NB : La méthode .**clone**() est prédéfinie sur la classe racine "**Object**" du langage java.

Cependant, le clonage automatique n'aura pas toujours le niveau de profondeur souhaité (si l'on ne reprogramme rien). On sera quelquefois amené à redéfinir la méthode ".clone()" sur certaines classes.

8.9. Visiteur (actif)

Le "visiteur" est un design pattern assez complexe qui est cependant quelquefois très intéressant si l'on souhaite décomposer un objet (ou une hiérarchie d'objets) en au moins deux parties complémentaires bien distinctes de façon à bien séparer certaines responsabilités (découplage) .

Prenons l'exemple d'un programme de dessin vectoriel. La hiérarchie de classe intuitive qui vient au premier abord à l'esprit est souvent du type suivant :



Bien qu'opérationnelle , cette hiérarchie unique à un potentiel inconvénient :

Elle n'est réutilisable que dans le cadre d'une technologie d'affichage bien déterminée (ex : affichage AWT/SWING en java) mais ne pourra pas être facilement réutilisée dans un contexte d'affichage différent (ex : SVG , JavaFx, ….).

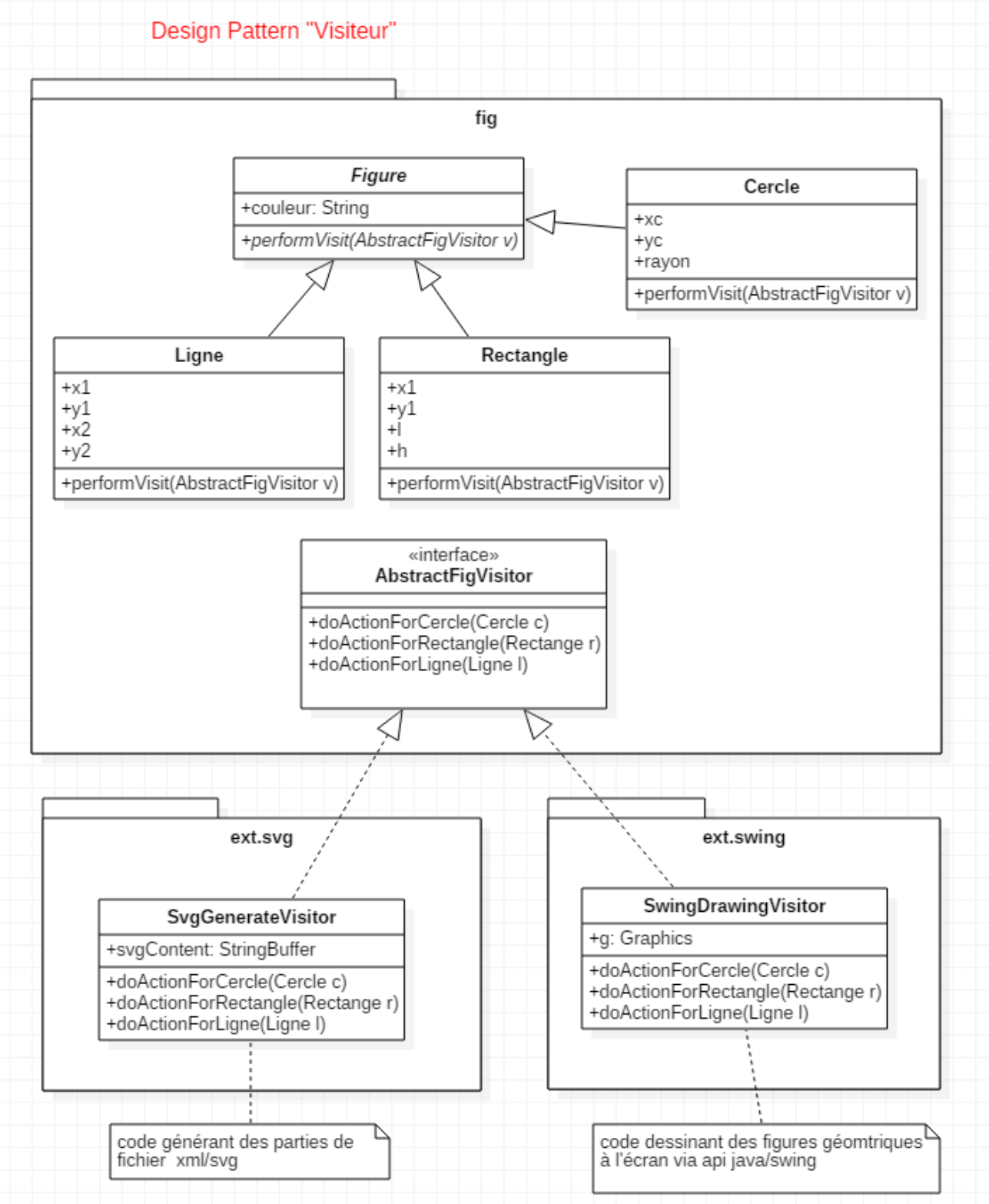
Pour obtenir une meilleur ré-utilisabilité, il est fortement conseillé de découpler les deux responsabilités suivantes :

- gestion des coordonnées (hiérarchie principale)
- gestion des affichages/rendus (avec éventuellement différentes variantes).

Pour atteindre cet objectif, le design pattern "Visiteur" consiste à :

1) Définir une classe ou interface de "***Visiteur abstrait***" permettant d'***effectuer de futurs actions (quelconques)*** **sur tel ou tel type d'élément concret** de la structure (hiérarchie) principale.
[ex : interface "***AbstractVisitor***" avec méthodes "***doActionForConcreteElementA***(a)" , "***doActionForCercle***(c)" , "***doActionForRectangle***(r)" , …]
2) Ajouter une méthode polymorphe de type "***performVisit***(visitor)" ou "***accept***(visitor)" que l'on redéfinira sur chaque classe d'élément concret avec un code type ressemblant à :
performVisit(AbstractVisitor visitor){
visisitor.doActionForCercle /*ou Rectangle ou ….*/ (this) ;
}
3) Définir par exemple une classe "***Renderer2D***" héritant de "***AbstractVisitor***" et implémentant chacune des méthodes abstraites "***doActionForXxxx***(x)" avec le code d'affichage adéquat *(ex : g.drawLine(ligne.getX1() , ligne.getY1() , ligne.getX2(), ligne.getY2()) ;)*
4) Parcourir en boucle tous les éléments *(ligne, rectangle ou cercle avec coordonnées)* de la structure *(ex : dessin)* en invoquant sur chaque élément la méthode ***performVisit***() à laquelle on passera en argument une instance du visiteur concret "Renderer2D". Ceci devrait normalement déclencher tous les affichages nécessaires.

Point technique clef du design pattern "Visiteur" : re-**propager automatiquement le comportement "polymorphe" intrinsèques de certains objets sur des éléments externes (les visiteurs)** . Le **visiteur (en tant qu'extension externe ou "plugin")** pourra ainsi effectuer une action appropriée (vis à vis du type exact d'objet visité) sans avoir besoin d'effectuer le moindre "if(... instanceof ….) ".



8.10. Mémento (mémorisation d'état)

Un objet "**Mémento**" est un **objet secondaire qui sert à mémoriser l'état d'un objet** de façon à le ***restaurer*** (si besoin) par la suite ou bien de façon à pourvoir effectuer des ***comparaisons*** entre l'état courant et un ancien état .

8.11. Commande (déclenchement uniforme d'actions)

Un objet "Command_XY" est un petit objet intermédiaire qui encapsule de façon uniforme le déclenchement d'une certaine action spécifique .

Une classe ou interface "Command" abstraite pourra par exemple comporter une méthode qui s'appellera toujours ".execute()" ou bien un couple de méthodes ".do() et .undo()" .

Chaque sous classe concrète (ex : CommandInsert , CommandDelete , ….) redéfinira la méthode ".execute()" en déclenchant une action bien précise .

On pourra éventuellement envisager une méthode ".undo()" symétrique de ".do() ou .execute()" permettant de déclencher une action contraire pour défaire ce qui a été fait lorsque c'est encore possible.

---> en stockant dans une pile les "commandes/actions" lancées par l'utilisateur on pourra alors effectuer une série de "do" ou "undo" (de types exacts éventuellement différents).

On pourra souvent relier (de façon événementielle) une même instance d'un objet "Command" aux différents éléments d'une interface graphique qui sont prévus pour déclencher une même action (ex : "bouton poussoir" , "menuItem" , "icône" d'une "toolbar" , …).

Comme pour tout design pattern, on peut envisager différentes façons de coder une "Commande".

Le critère important (à idéalement prendre en compte) est un "contexte applicatif" que l'on peut souvent associer à une commande (ex1 : référence sur objet courant à insérer ou à supprimer , ex2 : coordonnées d'un élément à créer/dessiner , …).

De ceci, peut éventuellement découler deux types de "Commandes" :
* des commandes brutes (ex : New , Save , Close , Quit , Delete , Insert , ….) directement associées aux éléments de l'IHM (sans contexte) *[une instance de chaque type suffit]*.

- des commandes élaborées (de second niveau) "avec contexte" (ex : DeleteObject avec refObj en tant que propriété interne) *[plusieurs instances sont souvent nécessaires pour encapsuler les différents états contextuels]*.
-

8.12. Chaîne de responsabilités (responsable(s) auxiliaire(s) en "backup")

Client appelant "opXy()" sur A1 avec A1 chaîné avec A2 lui même chaîné avec An .

A1() peut déléguer l'opération "opXy()" à A2 et ainsi de suite sans que :

* le Client se rendre compte de "quoi que ce soit"

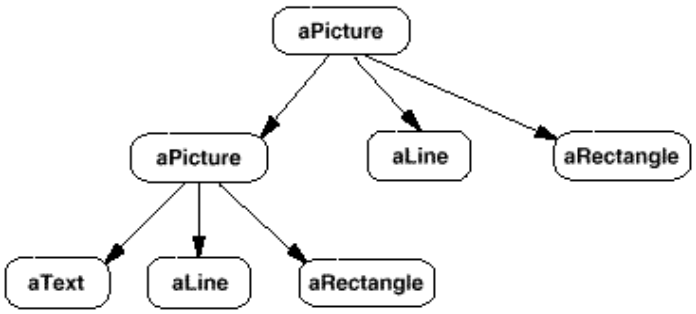
* le Client connaisse la longueur de la chaîne (aucun parcours en boucle n'est nécessaire)

Exemple concret (présenté par le "gof") :
Une information d' aide contextuelle ….

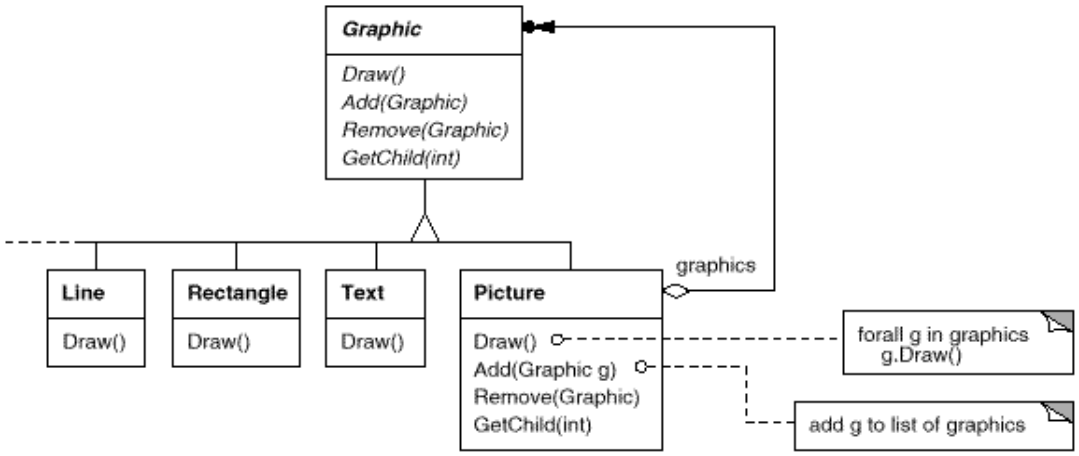
Autre exemple concret et sémantiquement proche: **ACL** (Access Control List)

8.13. Composite

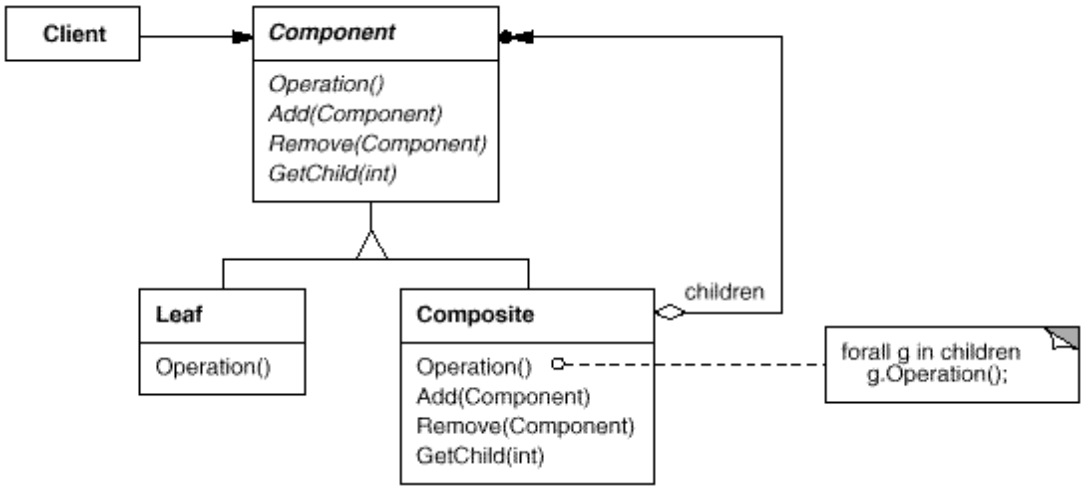
Problématique: Comment géré de façon élégante un niveau quelconque d'imbrication ?



Solution:



structure générale de la solution:



…

9. Présentation des "*design principles*"

Grands principes de conception orientée objet

Une autre série de "design patterns" appelés ***<<design principles>>*** et élaborés par *Bertrand MEYER* et *Robert MARTIN* permettent (si besoin) d'approfondir certains points et de formaliser un peu plus quelques principes objets fondamendaux.
On parle alors en terme de *principes* ou de *règles* là où jusqu'içi *GRASP* s'exprimait essentiellement en terme de *bonnes pratiques*.

Essentiellement liée à la structure générale des modules , cette série de patterns est tout à fait adaptée à la conception préliminaire.

Préambule (sur le vocabulaire employé dans ce chapitre)

La série des « design principles » utilise intensément le terme « package » à interpréter ici au sens « module » ou « artifact » plutôt que « namespace » .

Autrement dit « package » est ici à comprendre comme « packaging » (ex : archive « .jar ») .

10. Gestion des évolutions et dépendances

Gestion des évolutions et dépendances (1)

OCP (Open-Close P.) / Principe d'ouverture-fermeture:

Un module doit être ouvert aux extensions
mais fermés aux modifications.

LSP (Liskov Substitution Principle) / Principe de substitution de Liskov:

Les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans même le savoir (*substitution parfaite par une sous classe*).

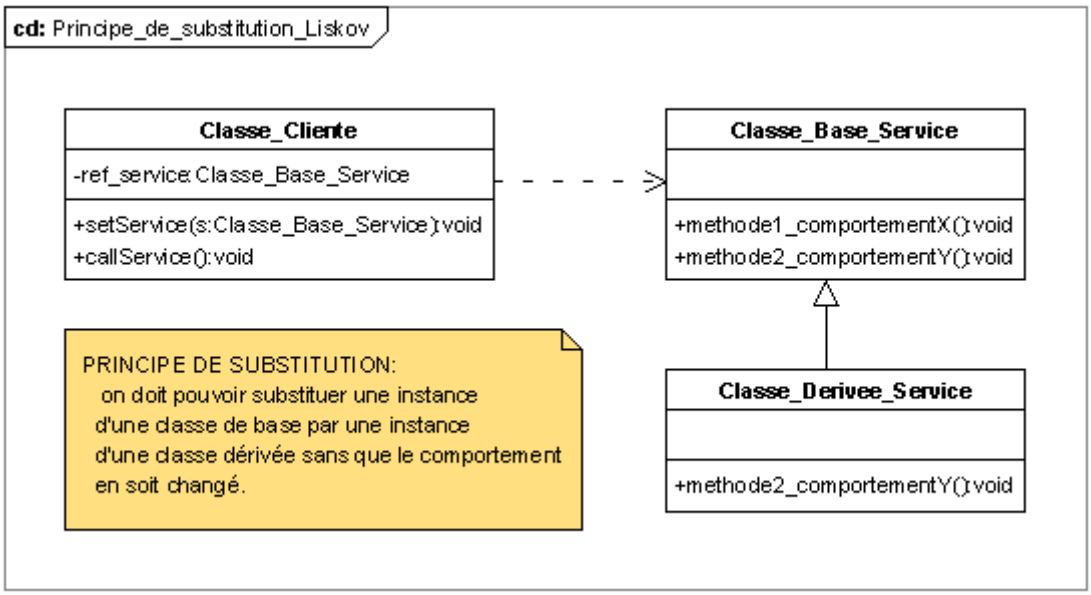
.../...

10.1. OCP et Principe de substitution (de Liskov)

Exemple (ouverture/fermeture):

* Un bon schéma d'héritage (avec niveau abstrait , sous classes concrètes et polymorphisme approprié) est naturellement ouvert à des ***extensions*** potentielles qui prendront la forme de ***nouvelles sous classes***.
* A l'inverse un code basé sur de multiples *if (xxx instanceof Cxx) ...else if(...instanceof Cyy)* est assez fermé car il *doit malheureusement être modifié pour évoluer*.
* D'un point de vue technique le LSP favorise l'OCP.
* D'un point de vue sémantique, *un LSP trop artificiel peut conduire un des implémentations non applicables (avec éventuelles exceptions à gérer)* , ce qui n'est pas mieux qu'un unique test "instanceof" (idéalement lié à toute une sous branche de l'arbre d'héritage).

Un objet d'une classe CX doit normalement pouvoir être substitué par n'importe quel objet d'une sous classe CY sans que le comportement en soit changé.



Autrement dit : **La sémantique (ou le comportement abstrait) ne doit pas être changé entre l'implémentation d'une opération dans une classe et dans une sous classe .**

10.2. Principe d'inversion des dépendances

Gestion des évolutions et dépendances (2)

DIP (Dependency Inversion P.) / Principe d'inversion des dépendances:

A. Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. Tous deux doivent dépendre d'abstractions

B. Les abstractions ne doivent pas dépendre de détails. Les détails doivent dépendre d'abstractions.

ISP (Interface Segregation P.) / Principe de séparation des interfaces:

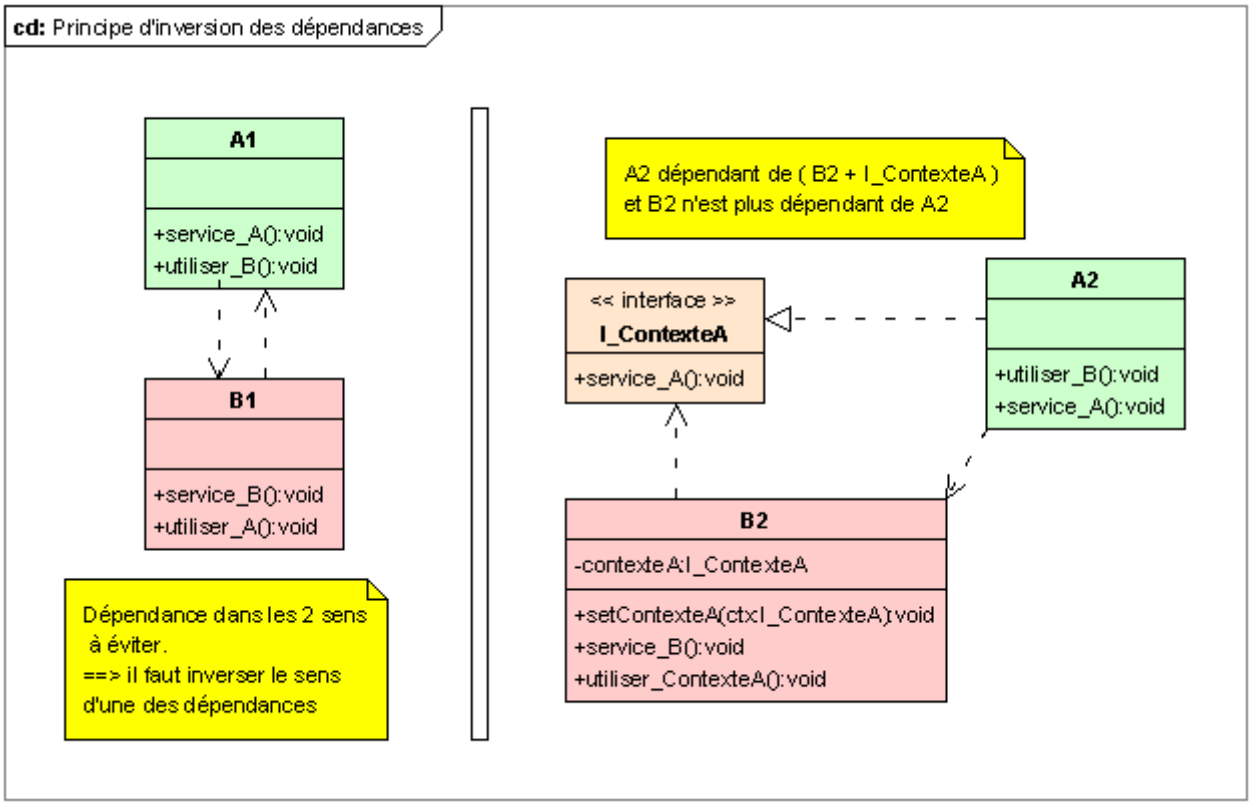
Les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas.

Autrement dit:

DIP_A : Un module de haut niveau doit toujours voir un service de plus bas niveau de façon abstraite (interface).

Pour qu'un module de bas niveau soit réutilisable par plusieurs modules de haut niveau, il doit interagir avec ceux-ci via des "callback" déclarées sur des contextes abstraits *(ex: méthodes événementielles et "listener")*

ISP : Ne pas hésiter à faire en sorte qu'une classe implémente plusieurs interfaces complémentaires. Les futurs clients ne seront alors dépendants que d'un seul point d'entrée abstrait.



Il vaut mieux éviter des dépendances dans les 2 sens entre deux classes ou packages (ou modules) :

A-->B et B-->A

On introduit alors une interface (I_ContexteA) de type "interface sortante coté B" devant être ultérieurement implémentée par A et on met en place une méthode d'enregistrement de contexte I_ContexteA coté B: ".setContexteA(IA ...) ou constructeur ..." .

Les nouvelles dépendances sont alors les suivantes:

A-->I_ContexteA (en l'implémentant)

B-->I_ContexteA (en l'utilisant)

A-->B

NB:

- L'interface I_ContexeA est censée être packagée coté B (dans le même package)
- Ce principe est très utilisé pour le traitement des événements.

==========

Exemple concret :

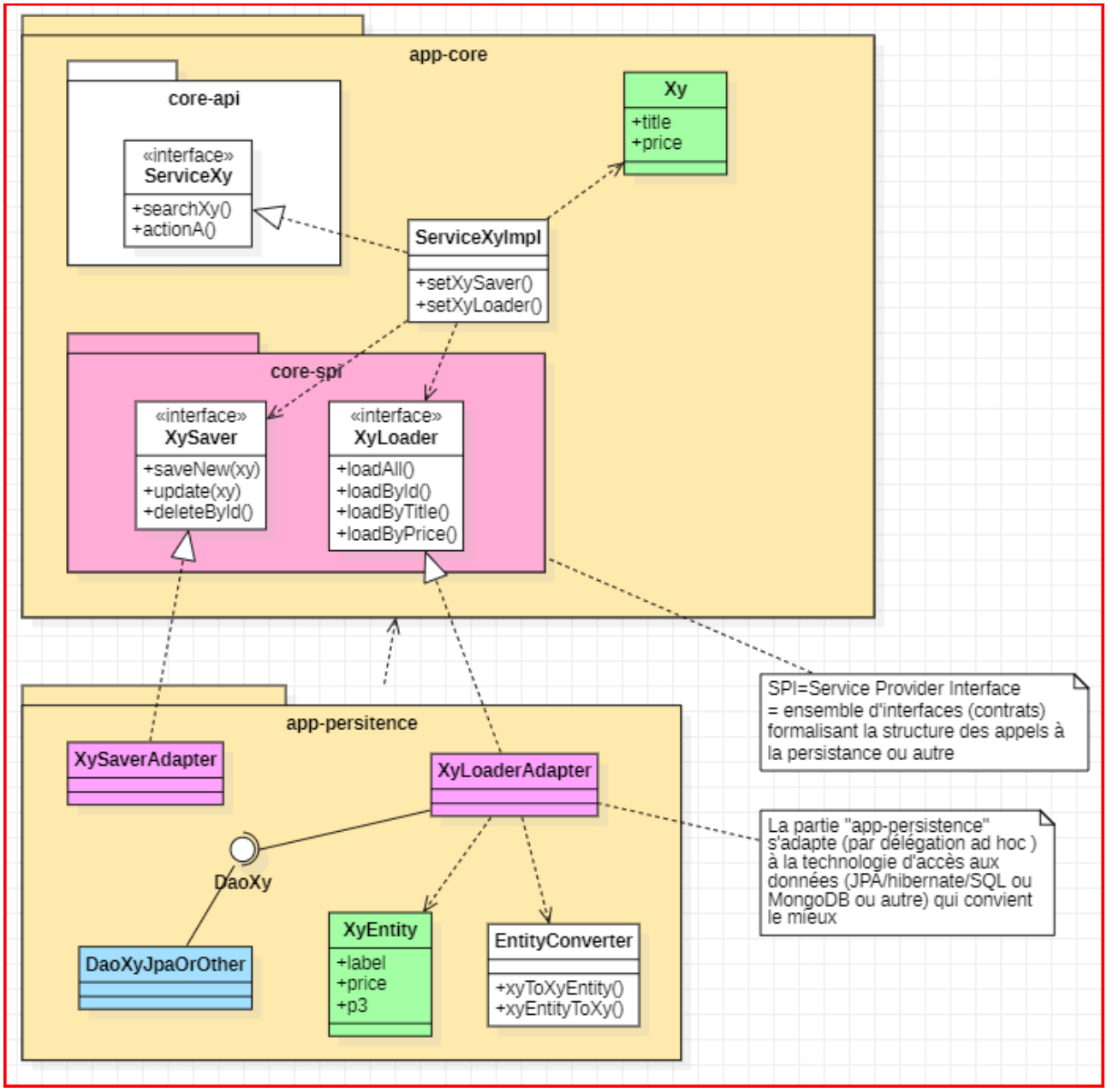
B2 = class Button (ou JButton) devant être réutilisable (dans package awt/swing)

I_ContexteA = interface ActionListener

A = Fenetre quelqonce utilisant un bouton

coté "bouton" , l'enregistrement du contexte se fait via addActionListener(...)

10.3. SPI = Exemple concret d'application du principe d'inversion des dépendances



C'est le point clef pour comprendre le fonctionnement de l'architecture hexagonale.

11. Organisation d'une application en modules

Organisation de l'application en modules

REP (Reuse/Release Equivalence P.) / Principe d'équivalence Réutilisation/Livraison:
La granularité en termes de réutilisation est le package.
Seuls des packages livrés sont susceptibles d'être réutilisés.

CRP (Common Reuse P.) / Principe de réutilisation commune:
Réutiliser une classe d'un package, c'est réutiliser le package entier.

CCP (Common Closure P.) / Principe de fermeture commune:
Les classes impactées par les mêmes changements doivent être placées dans un même package.

Autrement dit:

Ne pas hésiter à modifier un élément interne (et caché/privé) d'un package car ceci n'a pas d'impact négatif sur la réutilisation du package.

Décomposer s'il le faut un gros package en un ensemble de petits packages pour affiner les dépendances *(pour ne redéployer que ce qui a changé et pour favoriser de futures évolutions)*.

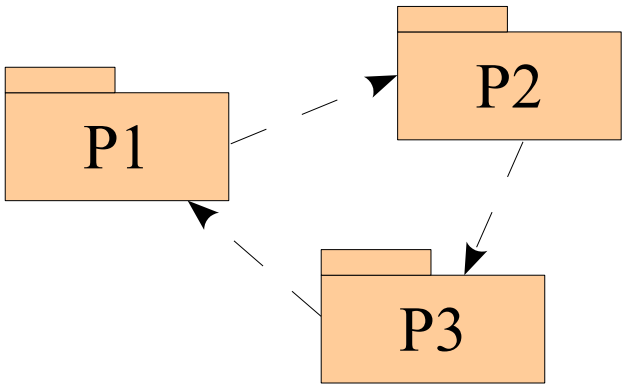
12. Gestion de la stabilité de l'application

Gestion de la stabilité de l'application

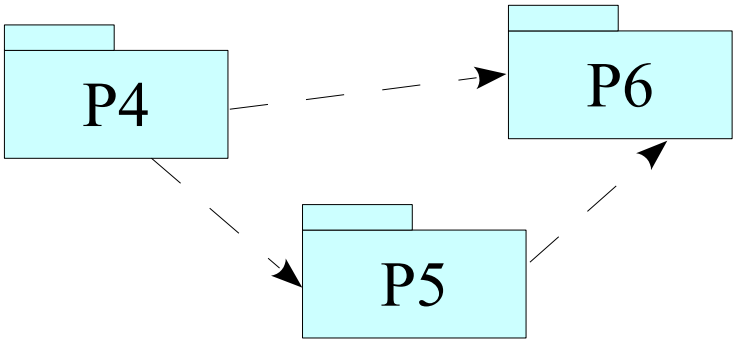
ADP (Acyclic-Depedencies P.) / Principe des dépendances acycliques: Les dépendances entre packages doivent former un graphe acyclique (sans dépendance(s) circulaire(s)).

SDP (Stable Depedencies P.) / Principe de relation dépendance/stabilité: Un package doit dépendre uniquement de packages plus stables que lui.

SAP (Stable Abstractions P.) / Principe de stabilité des abstractions: Les packages les plus stables doivent être les plus abstraits. Les packages instables doivent être concrets. Le degré d'abstraction d'un package doit correspondre à son degré de stabilité.

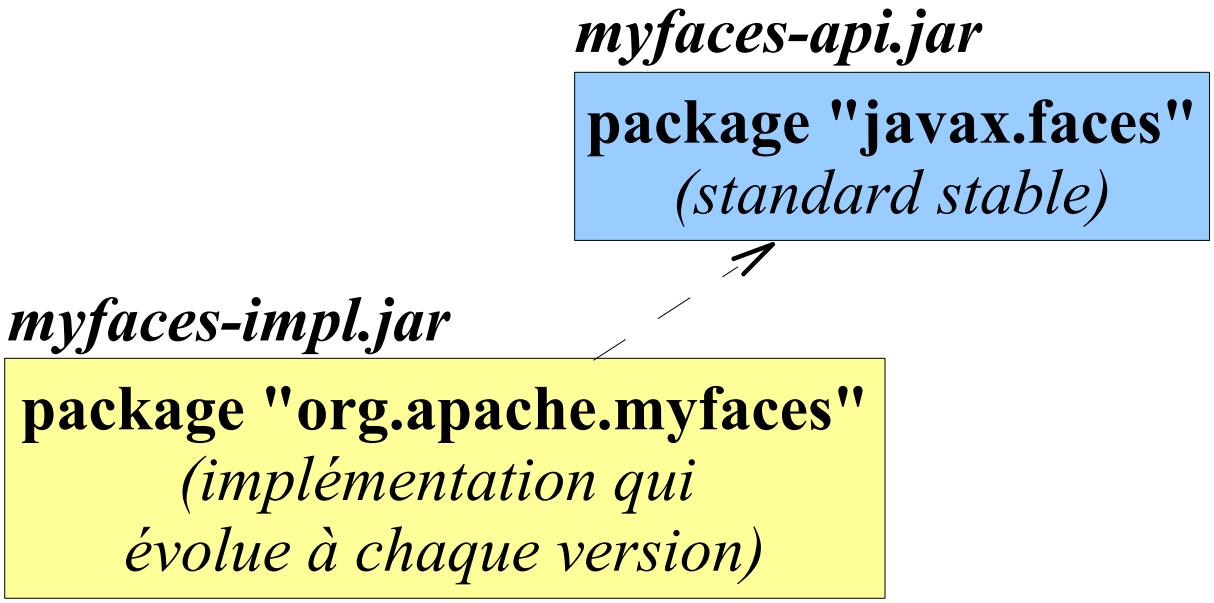


pas bien !



bien !

exemple: implémentation "***myfaces / Apache***" du framework WEB "***JSF***"



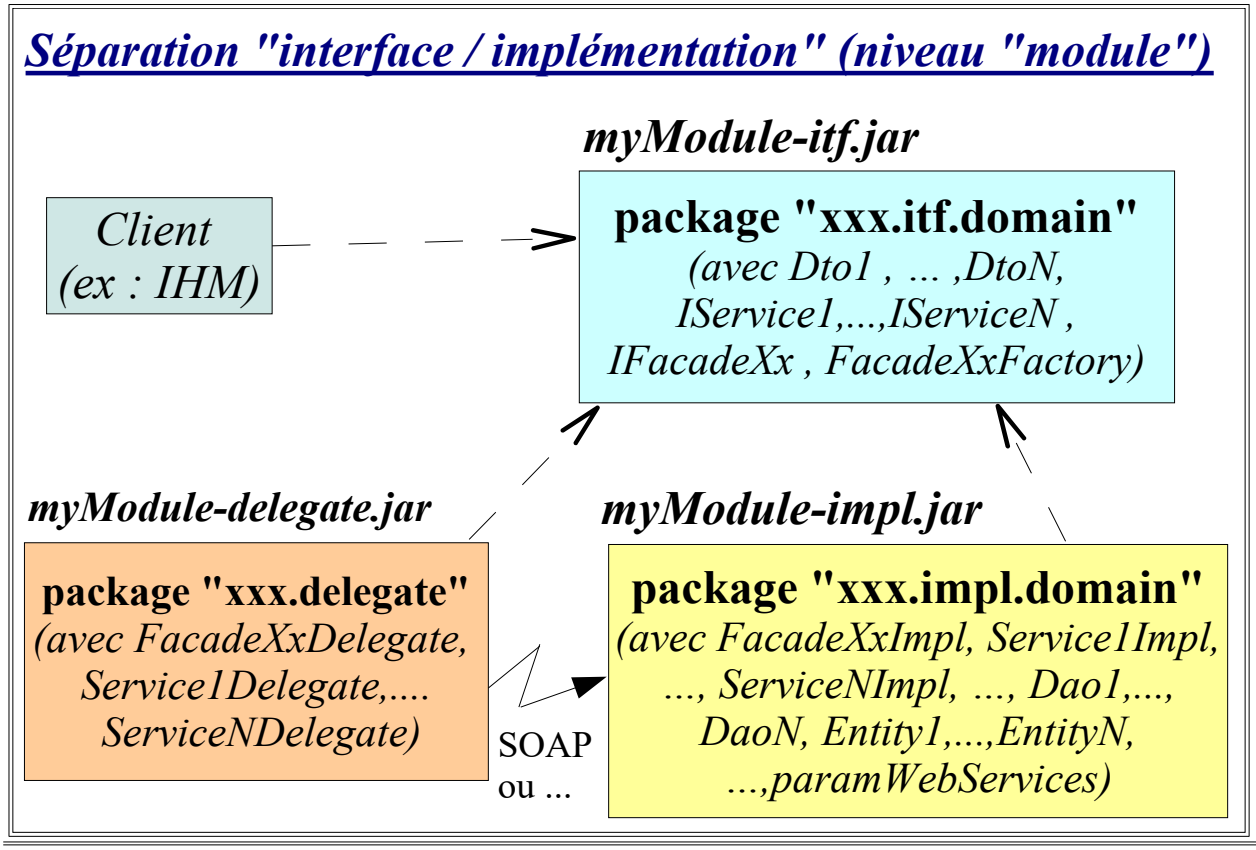
Application courante :

client_module -----> **moduleXY_itf**

(implémenté via)

mod_XY_local_impl *ou bien* **mod_XY_delegate_impl** ----> services_distant

(selon ".jar" présent dans le classpath) .



NB : Il existe des "métriques de packages" qui permettent de mesurer le respect des principales règles de "bonne conception orientée objet" .

ANNEXES

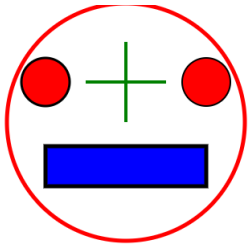
VII - Annexe – Exercices et TP

1. Concepts objets et code associé

Expérimenter les concepts objets via du code concret (dans un langage orienté objet tel que java ou typescript) sur le thème d'une application de dessin permettant de dessiner (directement ou pas) des lignes , des rectangles et des ellipses ou cercles.

NB : On pourra éventuellement programmer des classes "Line" , "Rectangle" , "Cercle" (avec coordonnées, couleurs) et une méthode .toSvgString() permettant de générer des fichiers .svg (cas particulier de fichier .xml) que l'on pourra afficher via un navigateur internet

exemple.svg



```
<svg xmlns='http://www.w3.org/2000/svg' height='400' width='500'>
 <circle cx='100' cy='100' r='30' stroke='black' stroke-width='3' fill='red' />
 <line x1='200' y1='50' x2='200' y2='150' stroke='green' stroke-width='4' fill='none' />
 <line x1='150' y1='100' x2='250' y2='100' stroke='green' stroke-width='4' fill='none' />
 <circle cx='300' cy='100' r='30' stroke='black' stroke-width='2' fill='red' />
 <rect x='100' y='180' width='200' height='50'
       fill='blue' stroke='black' stroke-width='5' />
 <circle cx='200' cy='150' r='148' stroke='red' stroke-width='5' fill='none' />
</svg>
```

2. Modélisation UML

Petite étude ce cas :

Application de dessin (en ligne sur internet) avec les fonctionnalités basiques suivantes :
- dessin dynamique dans une zone canvas de HTML5
- export sous forme de fichiers ".svg" téléchargeables

Les clients "premium" qui auront payer un petit abonnement pourront en plus :
- sauvegarder leurs dessins sur le site de l'application
- rendre facultativement "public" leurs dessins pour que ceux-ci soient consultables par les autres clients

3. Application de certains design patterns

Améliorer la structure du code de l'application de dessin en appliquant quelques designs patterns appropriés (tels que "command" ou "visiteur" ou autres).