

---

# Expression des besoins et analyse avec UML

## Table des matières

I - Présentation du cours (objectifs).....	5
II - Cadre UML.....	6
1. Méthodologie, formalisme , .....	6
2. Historique rapide d'UML.....	8
3. UML pour illustrer les spécifications.....	9
4. Formalisme UML = langage commun.....	9
5. UML universel mais avant tout orienté objet.....	10
6. Standard UML et extensions (profiles).....	10
7. Présentation des diagrammes UML.....	11
8. Descriptions sommaires des diagrammes UML.....	12
9. Utilisations courantes des diagrammes UML.....	18
10. Quelques outils UML (Editeurs , AGL).....	19

<b>III - Démarche , études de cas , .....</b>	<b>20</b>
1. Activités de modélisation et spécifications.....	20
2. Cycle "itératif & incrémental" .....	22
3. Enchaînement des activités de modélisation.....	23
4. Bibliothèque / médiathèque (étude de cas).....	23
5. Agence de voyage (étude de cas alternative).....	24
6. Autres étude de cas.....	25
<b>IV - Diagramme de contexte / périmètre applicatif.....</b>	<b>26</b>
1. Début de la modélisation UML ?.....	26
2. Diagramme de contexte.....	27
<b>V - Expr. besoins fonctionnels / Uses Cases.....</b>	<b>29</b>
1. Etude des principales fonctionnalités (U.C.).....	29
2. Diagramme des cas d'utilisations.....	30
3. Scénarios et descriptions détaillés (U.C.).....	34
4. Méthodologie fonctionnelle détaillée.....	36
5. Gestion de projet et planification basées sur les cas d'utilisation.....	37
6. Eventuelle planification des livrables selon XP.....	37
<b>VI - Diagramme d'activités.....</b>	<b>39</b>
1. Diagramme d'activités.....	39
2. Distinction "micro activité / macro activité" selon granularité.....	44
<b>VII - Règles de gestions.....</b>	<b>45</b>
1. Formulations des exigences et traçabilité.....	45
2. Exemples d'expression des règles de gestion.....	45
<b>VIII - Génération documentations / spécifications.....</b>	<b>46</b>
1. Documentation / Livrables / Spécifications.....	46
<b>IX - Modélisation métier (business modeling).....</b>	<b>48</b>
1. Rôle et contenu de la modélisation métier.....	48
2. Modélisation du contexte d'utilisation.....	50
3. Diagramme des cas d'utilisations métiers.....	51
<b>X - Expr. Besoins IHM (maquettes, navigations).....</b>	<b>52</b>
1. Expression des besoins "IHM" (interface graphique).....	52

<b>XI - Fondamentaux orientés "objet".....</b>	<b>56</b>
1. Concepts objets fondamentaux.....	56
2. Granularité.....	63
3. Modularité.....	64
<b>XII - Analyse du domaine (entités) / diag. Classes.....</b>	<b>66</b>
1. Analyse du domaine (glossaire , entités).....	66
2. Diagramme de classes (notations , ... ).....	68
3. Eléments structurants d'UML.....	79
<b>XIII - Diagrammes d'états (cycle de vie , ... ).....</b>	<b>83</b>
1. Diagramme d'états et de transitions (StateChart).....	83
2. Utilisation d'un diagramme d'état pour illustrer un cycle de vie.....	87
<b>XIV - Réalisation de Uses Cases / diag. Séquences.....</b>	<b>89</b>
1. Analyse applicative (objectif et mise en oeuvre).....	89
2. Responsabilités (n-tiers) et services métiers.....	90
3. Repère méthodologique (rappel).....	91
4. Réalisation des cas d'utilisations.....	92
5. Modèle dynamique – diagrammes d' interactions.....	93
6. Diagramme UML de Collaboration / Communication.....	94
7. Diagramme de séquences (UML).....	95
8. Diagramme de séquences (notations avancées).....	97
<b>XV - Besoins techniques / env. / diag déploiement.....</b>	<b>101</b>
1. Spécification de l'environnement cible.....	101
2. Exemples d'exigences techniques classiques.....	102
<b>XVI - Aperçu général sur la conception.....</b>	<b>103</b>
1. Rôles de la conception.....	103
2. Activités de la conception.....	103
3. Conception générique.....	105
4. Objectifs de la conception préliminaire.....	106
<b>XVII - Implémentation , retours tests , itérations.....</b>	<b>107</b>
<b>XVIII - Différences entre UML et Merise.....</b>	<b>109</b>
1. différences cardinalités/multiplicités.....	109

---

<b>XIX - Annexes - Patterns "GRASP" .....</b>	<b>110</b>
1. Affectation des responsabilités (GRASP).....	110
2. Les 4 patterns GRASP fondamentaux.....	112
3. Les 5 paterns GRASP spécifiques.....	116
<b>XX - Annexes - Méthodologies (aperçu rapide).....</b>	<b>119</b>
1. Bonnes pratiques UP .....	119
2. Méthodes agiles.....	120
3. XP (Extreme Programming).....	122
4. Méthode agile SCRUM.....	126
<b>XXI - OCL (Object Constraint Language).....</b>	<b>132</b>
1. OCL (Object Constraint Language) - Présentation.....	132
<b>XXII - Essentiel outil "Papyrus UML" .....</b>	<b>133</b>
1. Utilisation de Papyrus_UML (éditeur UML2 eclipse).....	133
2. Génération de documentation (gendoc2).....	145
<b>XXIII - Annexe – Bibliographie, Liens WEB , outils.....</b>	<b>151</b>
1. Bibliographie et liens vers sites "internet" .....	151

# I - Présentation du cours (objectifs)

Le cours "*Expression des besoins et analyse avec UML*" présente l'utilisation du *formalisme UML* sur les *phases fonctionnelles (orientées "métier")* de la *modélisation*.

Cette formation se focalise sur la *sémantique* et la *syntaxe* des *diagrammes normalisés d'UML* et sur la *réalisation* de *spécifications concrètes (SFG, SFD)*.

Cette formation est aussi bien utile à :

- la maîtrise d'œuvre (**moe**) qui doit **réaliser/écrire les spécifications**.
- la maîtrise d'ouvrage (**moa**) qui doit bien savoir **lire et comprendre un modèle pour le valider**.

Une présentation très rapide (en début de formation) de l'essentiel des concepts objets permettra de bien comprendre les spécificités orientées objets d'UML.

Les aspects suivants seront développés dans les détails:

- diagramme de contexte et notion de modélisation métier
- cas d'utilisation (avec scénarios et illustration sous forme de diagramme d'activités)
- expression des règles de gestion
- expression des besoins ihm (Interface Homme-Machine) & technique.
- glossaire/dictionnaire des entités du domaine
- diagramme de packages (secteurs métiers/fonctionnels)
- diagramme de classes (pour entités du domaine) avec associations, rôles et multiplicités
- classes correspondant aux "services métiers" et dépendances
- modélisation d'un cycle de vie d'une entité avec un diagramme d'états
- diagrammes de séquences (pour la réalisation des cas d'utilisation) et cohérence.
- démonstration de génération de documentation (avec gendoc2 ou ...)

Déroulement:

Après une présentation générale du cadre UML (formalisme, méthodologie, infrastructure, ...) de 1h30 environ, la formation sera construite autour de l'accompagnement théorique et pratique nécessaire à la mise en œuvre d'une petite étude de cas.

Avec à chaque stade :

- présentation d'une phase de la démarche méthodologique consensuelle (synthèse des bonnes pratiques usuelles).
- présentation de quelques éléments du formalisme UML (syntaxes, diagrammes, sémantiques, ...)
- présentation rapide de quelques façons d'utiliser l'outil UML (en TP)
- TP semi-directif consistant à réaliser une partie de la modélisation liée à l'étude de cas.
- présentation d'une solution du TP (avec argumentation des choix effectués) et petit discours sur ce qu'il est important de retenir.

Ainsi, au terme de la formation,

- tous les diagrammes UML auront été abordés (en passant plus de temps sur ceux qui sont vraiment importants et utiles).
- une première utilisation pratique d'UML (guidée par une certaine démarche ré-applicable) aura été expérimentée.

## II - Cadre UML

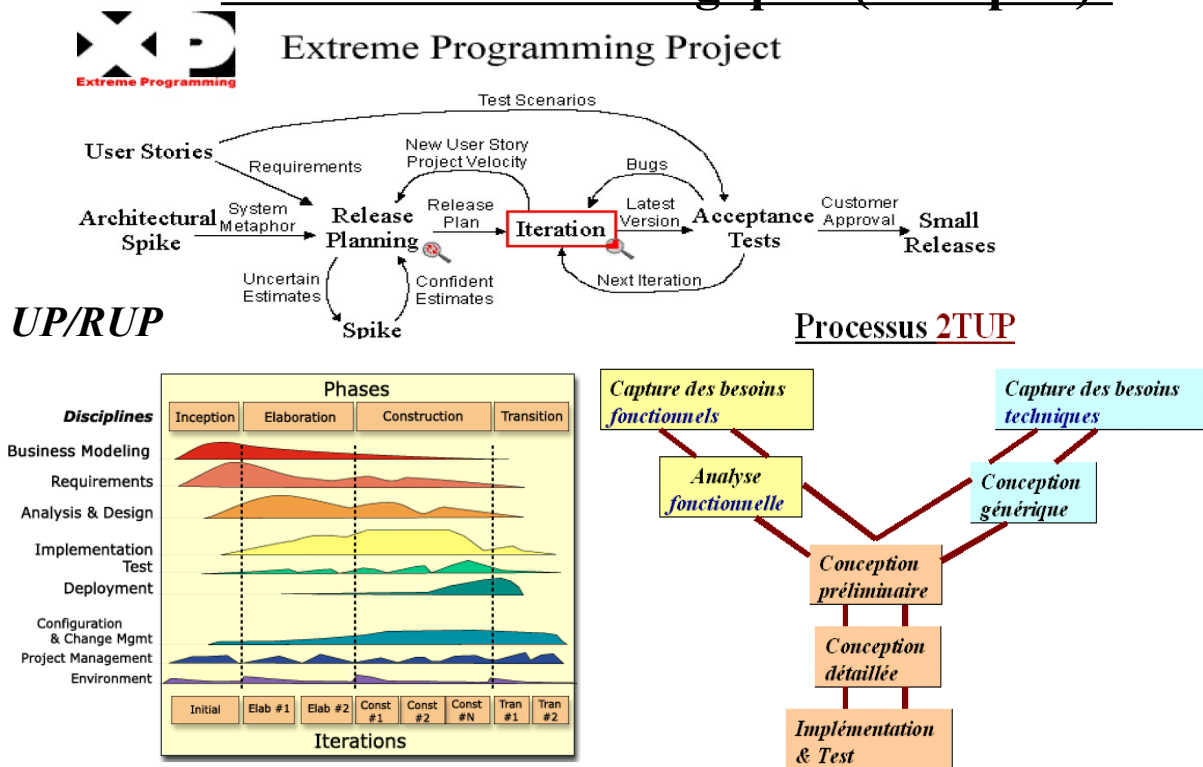
### 1. Méthodologie, formalisme , ....

#### 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/ ...)



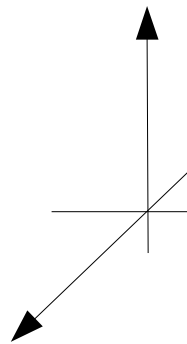
#### Processus méthodologiques (exemples):



## 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 ?

### Pourquoi ?

(Quels objectifs ?  
Quelles utilités ?)



### Modélisation métier

(business modeling)

+ **expression des besoins**

(C.I.M. : Computation Independant Model)

### Quoi ?

(Quelles entités ?  
Quelles structures ?  
Quels services ?)

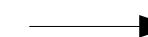


### Analyse

(P.I.M. : Platform Independant Model)

### Comment ?

(avec quelles  
Technologies ?)

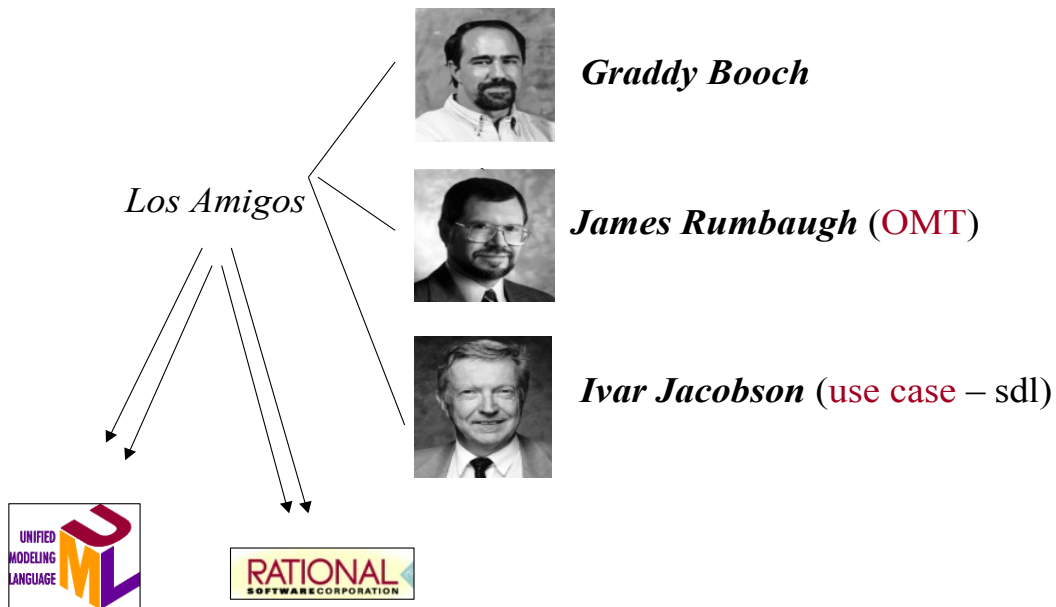


### Conception

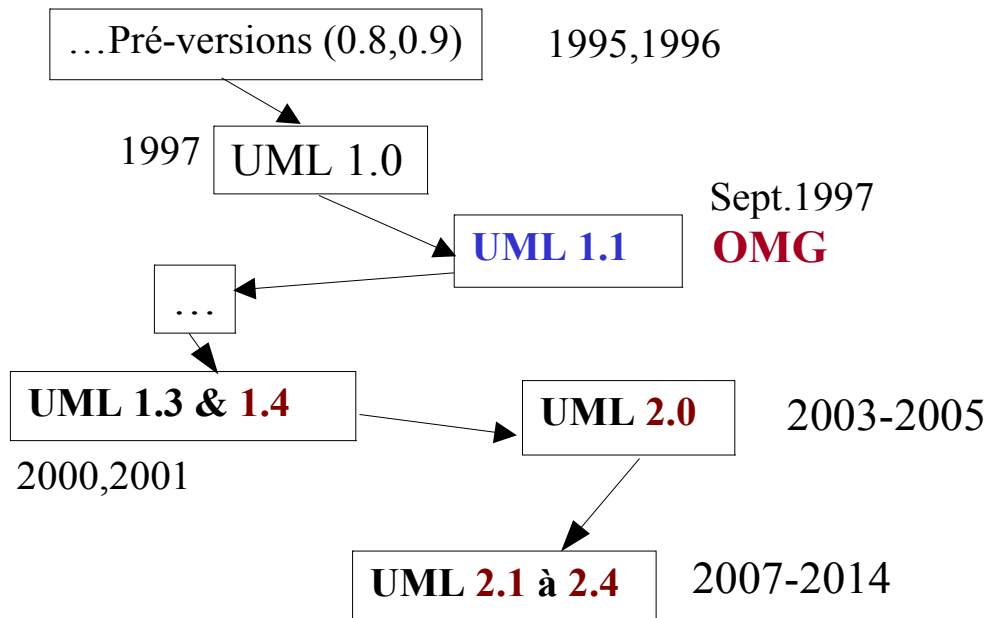
(P.S.M. : Platform Specific Model)

## 2. Historique rapide d'UML

### Les fondateurs d'UML



### Normalisation d'UML (standard de l'OMG)





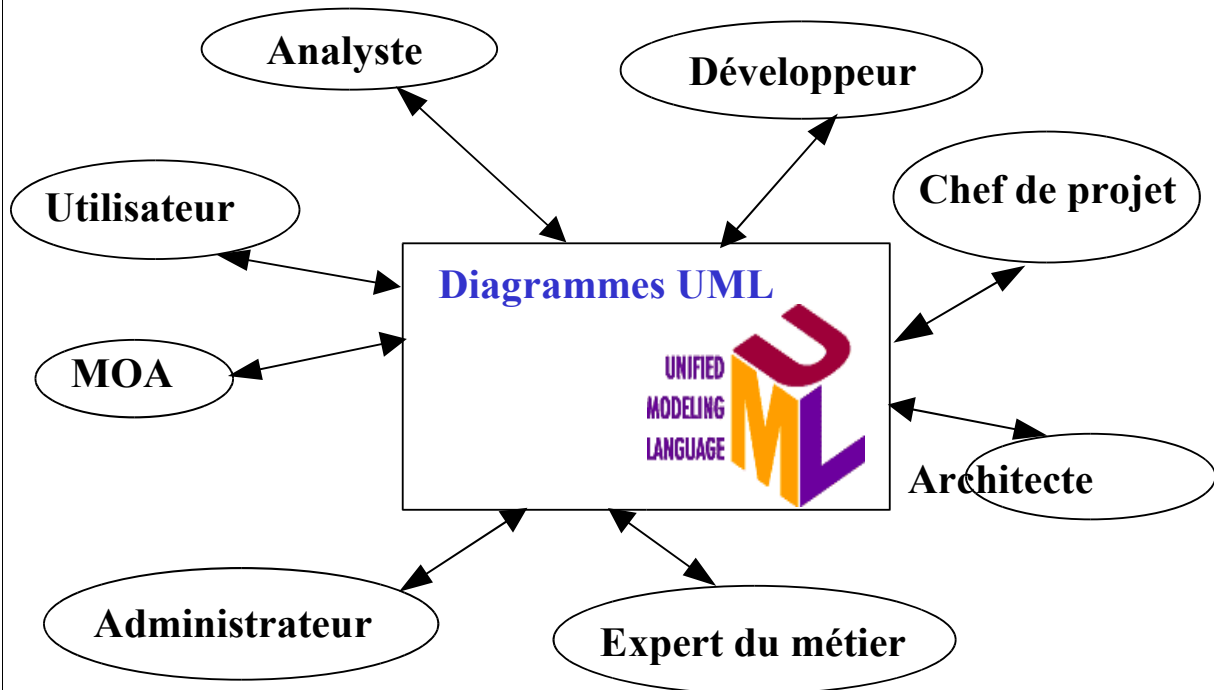
### 3. UML pour illustrer les spécifications

#### Principales utilités d'UML

- ♦ Diagrammes UML = partie importante des **Spécifications** fonctionnelles et techniques.
- ♦ **Cogiter** sur le "pourquoi/quoi/comment" en se basant sur 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 , ...

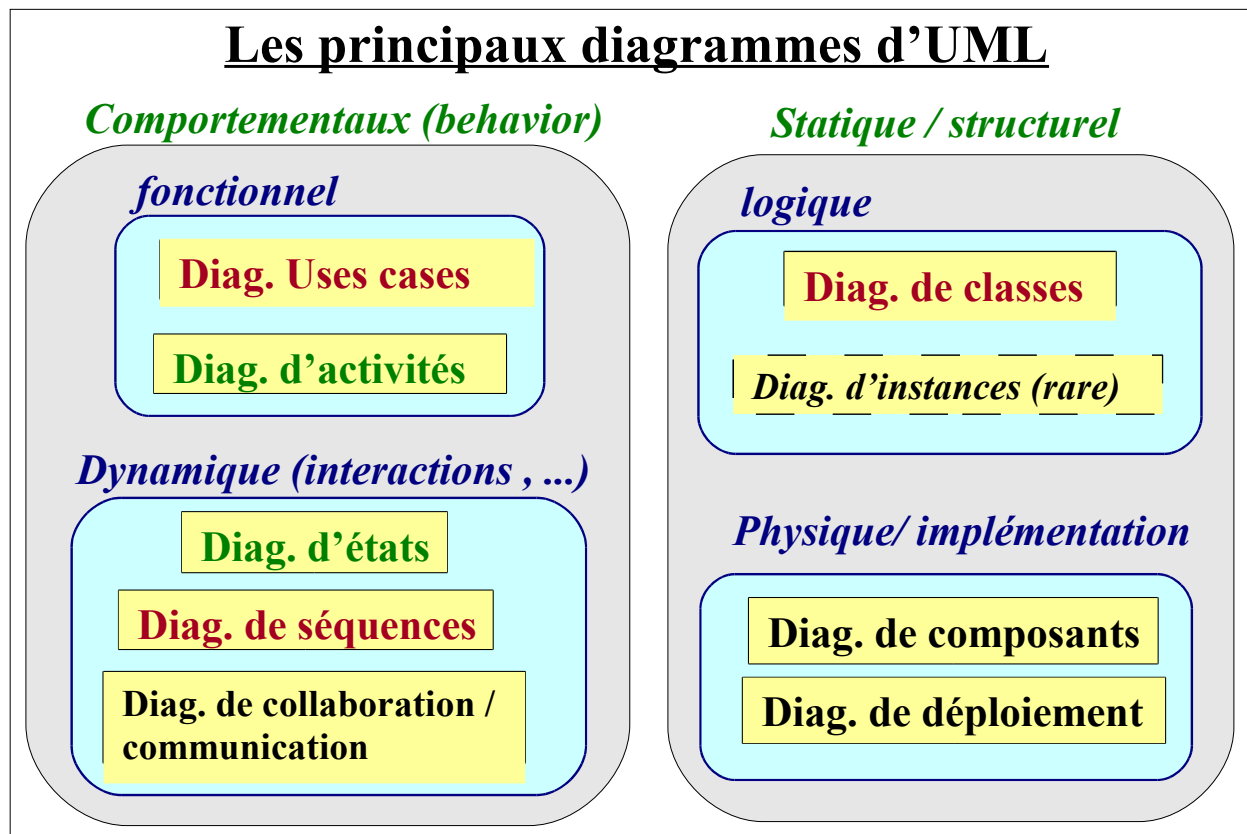
**Base UML standard**

+

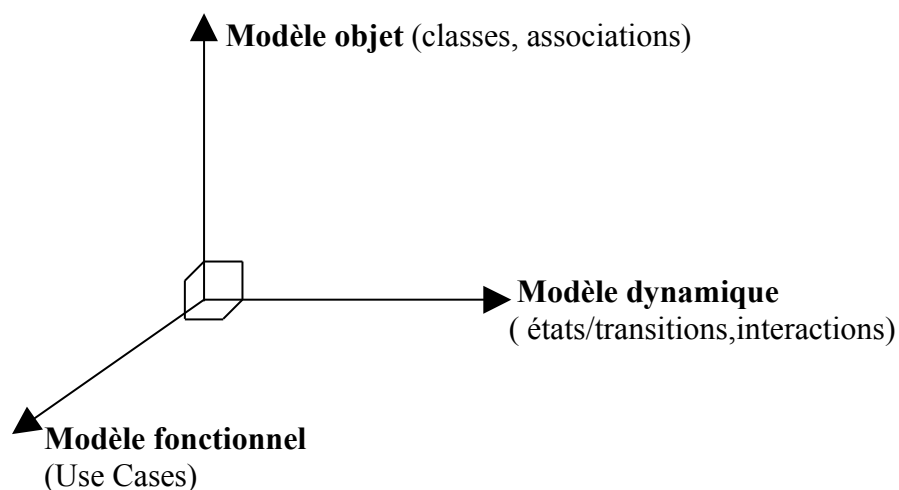
**Extensions (profiles)**

*normalisés ou pas* ↗

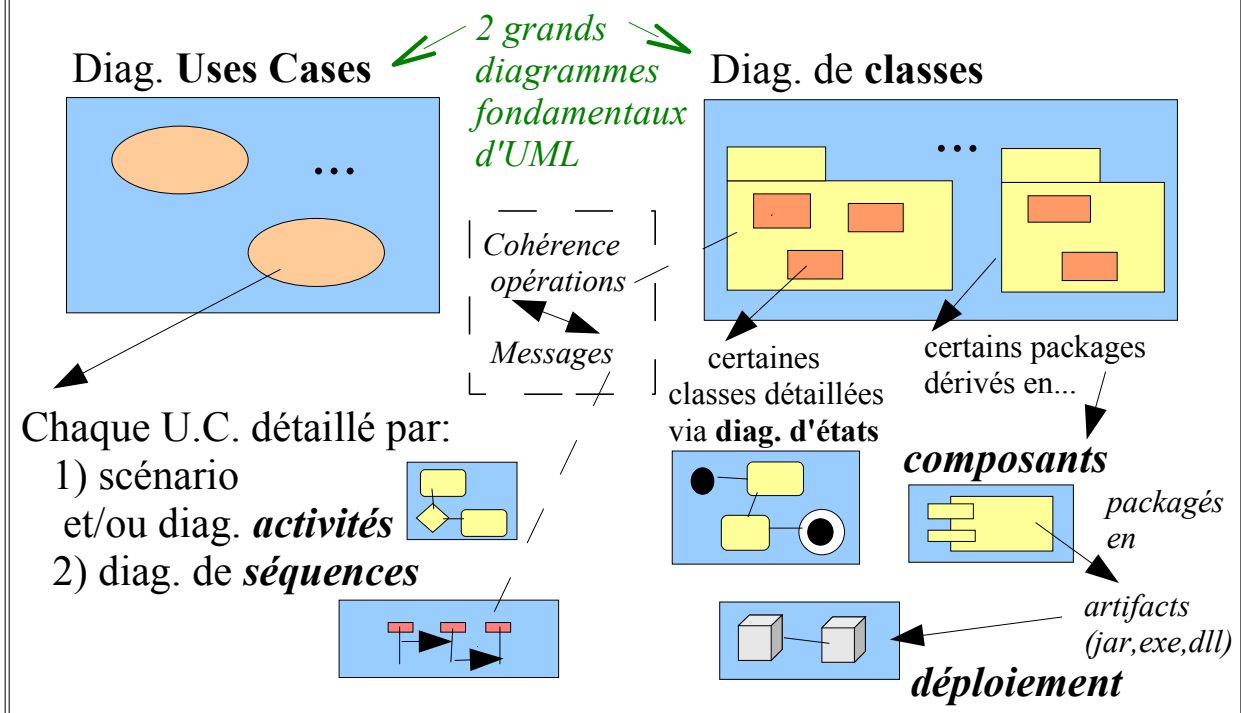
## 7. Présentation des diagrammes UML



Modèles (diagrammes) complémentaires:



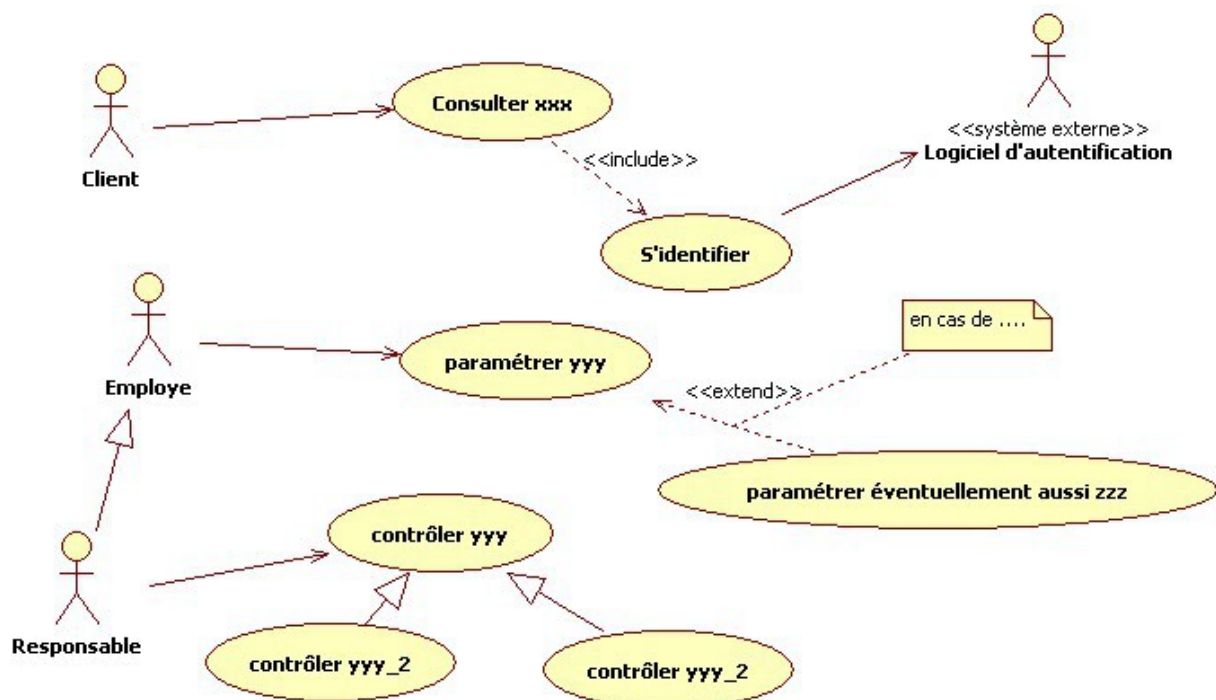
## Principaux liens entre les diagrammes UML



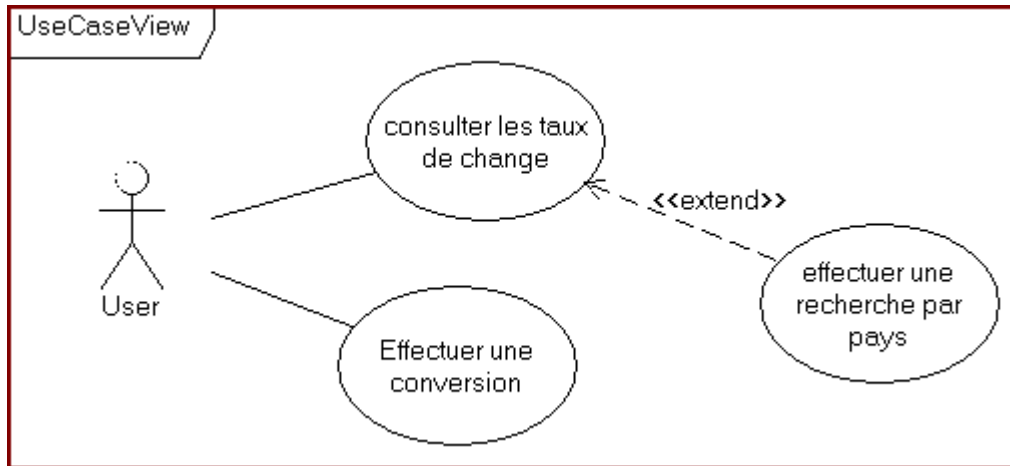
## 8. Descriptions sommaires des diagrammes UML

### 8.1. Diagramme des cas d'utilisations (Uses Cases)

exemple:



sur micro étude cas "conversion de devises":



## 8.2. Diagramme de classes

exemples:

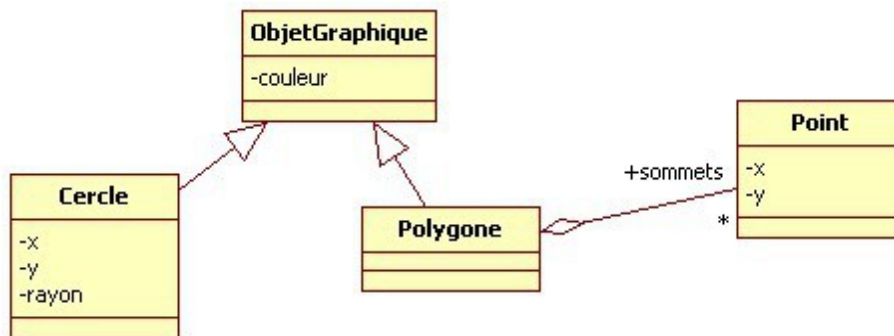
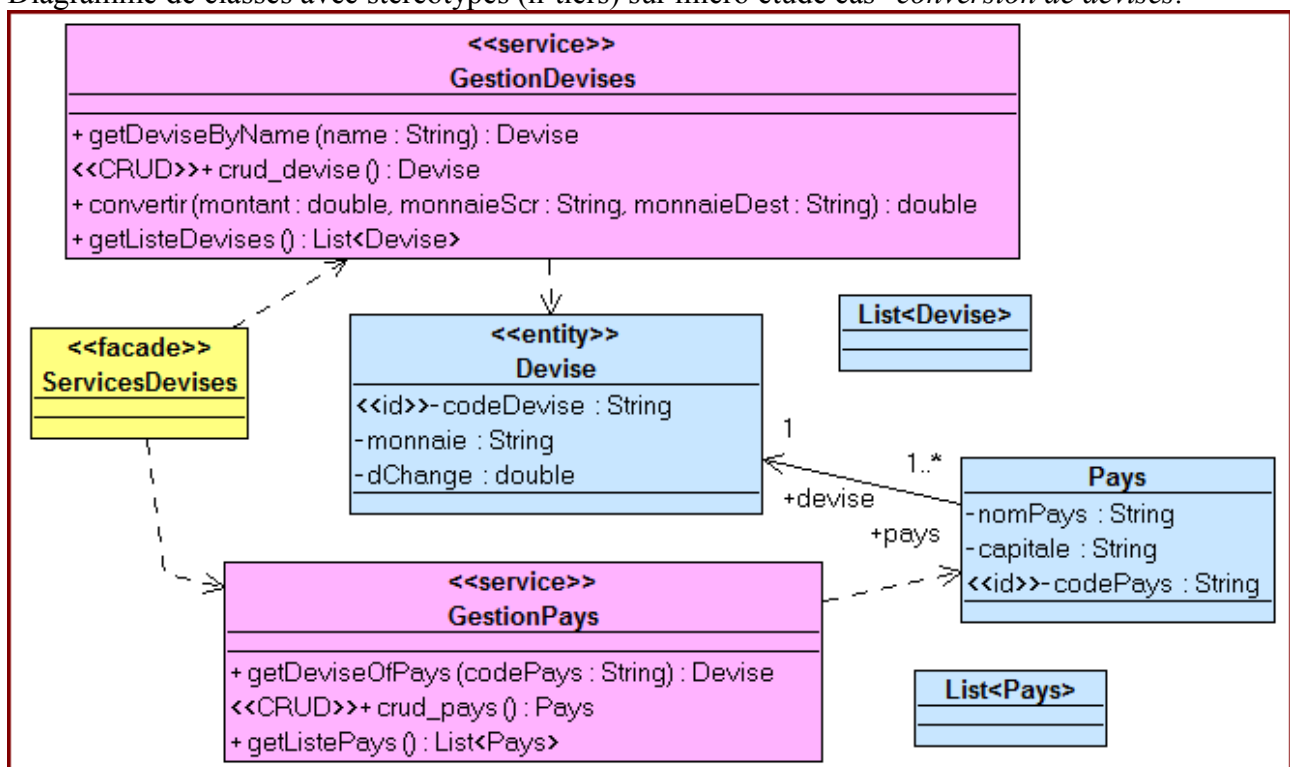
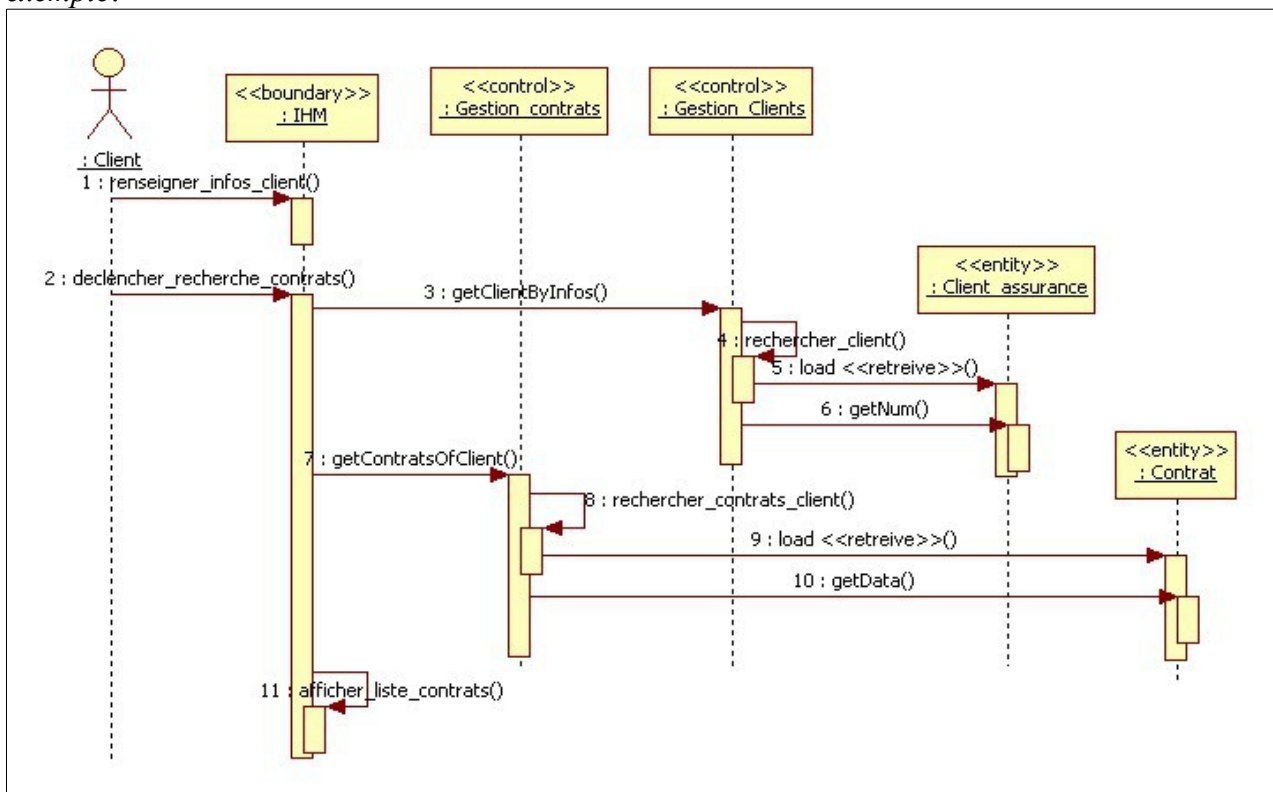


Diagramme de classes avec stéréotypes (n-tiers) sur micro étude cas "conversion de devises":



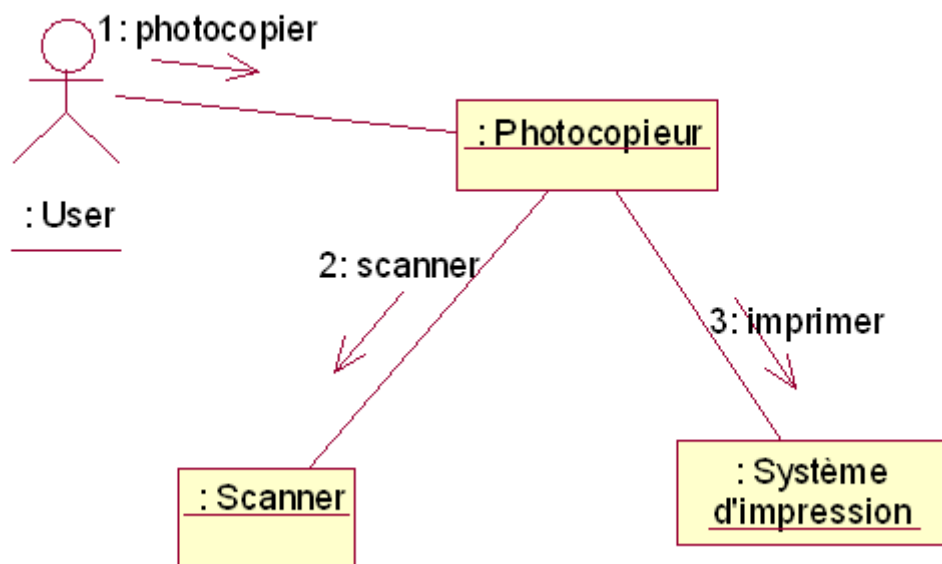
### 8.3. Diagramme de séquence

exemple:



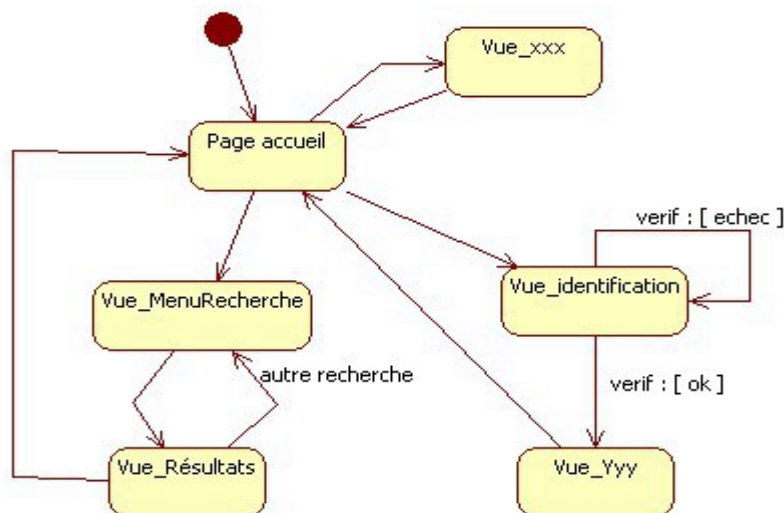
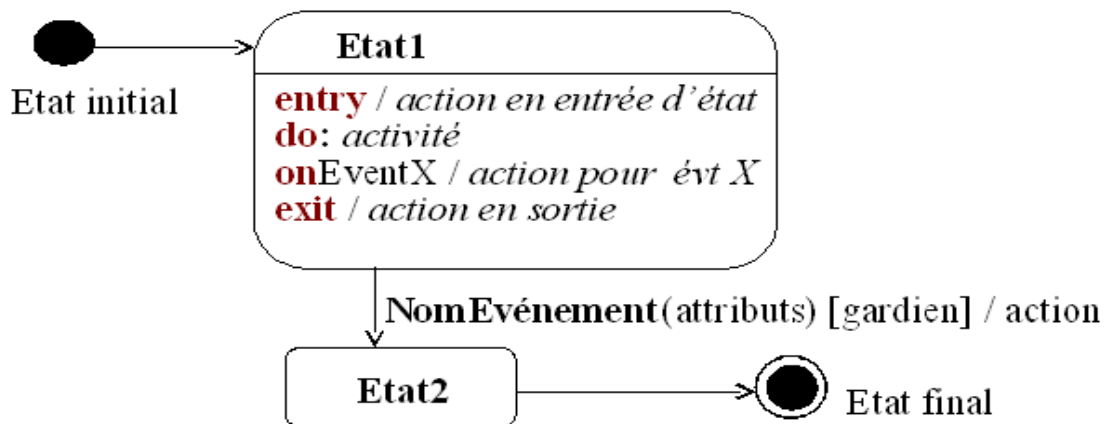
### 8.4. Diagramme de collaboration (UML1) / communication (UML2)

exemple:



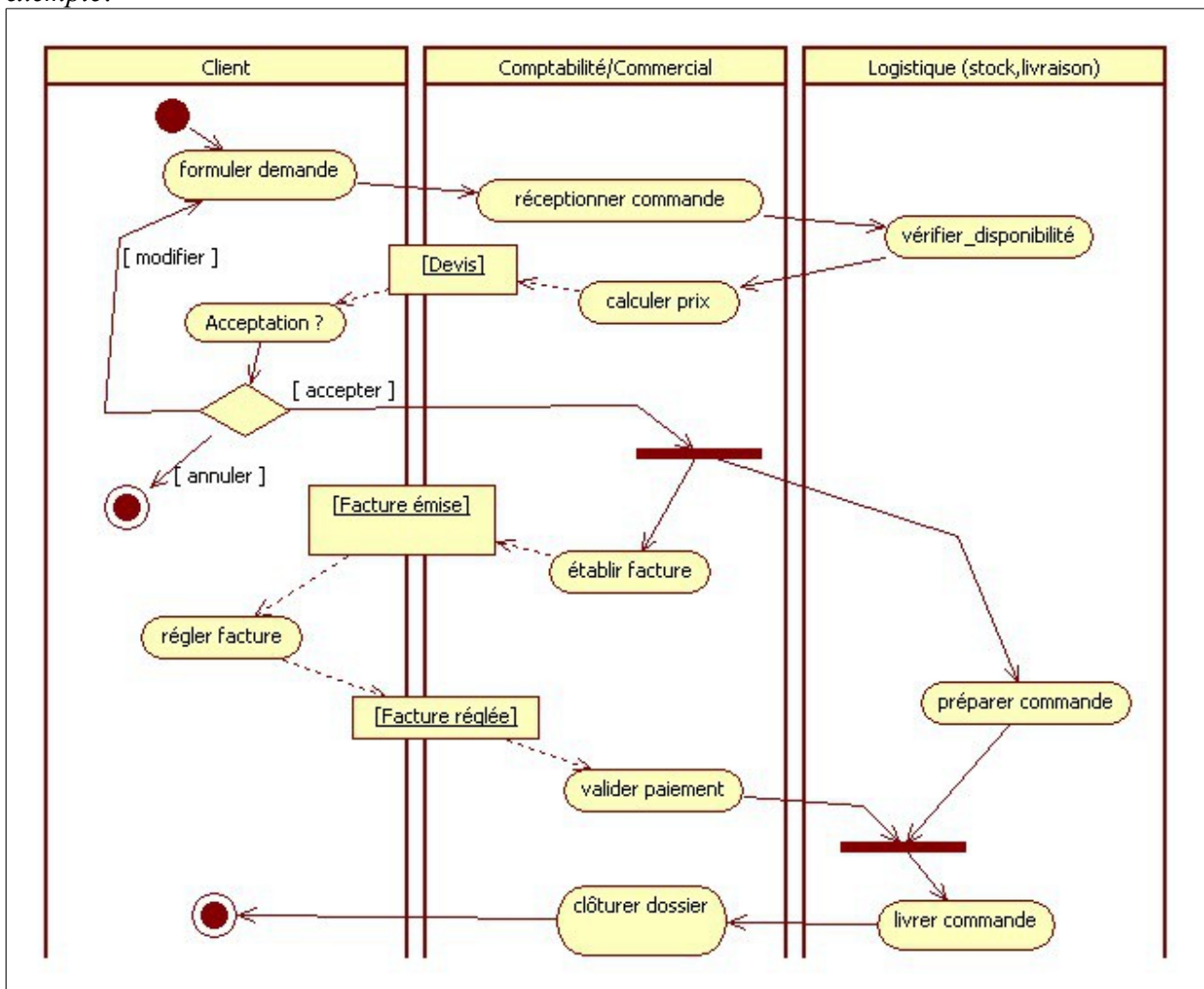
## 8.5. Diagramme d'états (StateChart)

exemples:



## 8.6. Diagramme d'activités

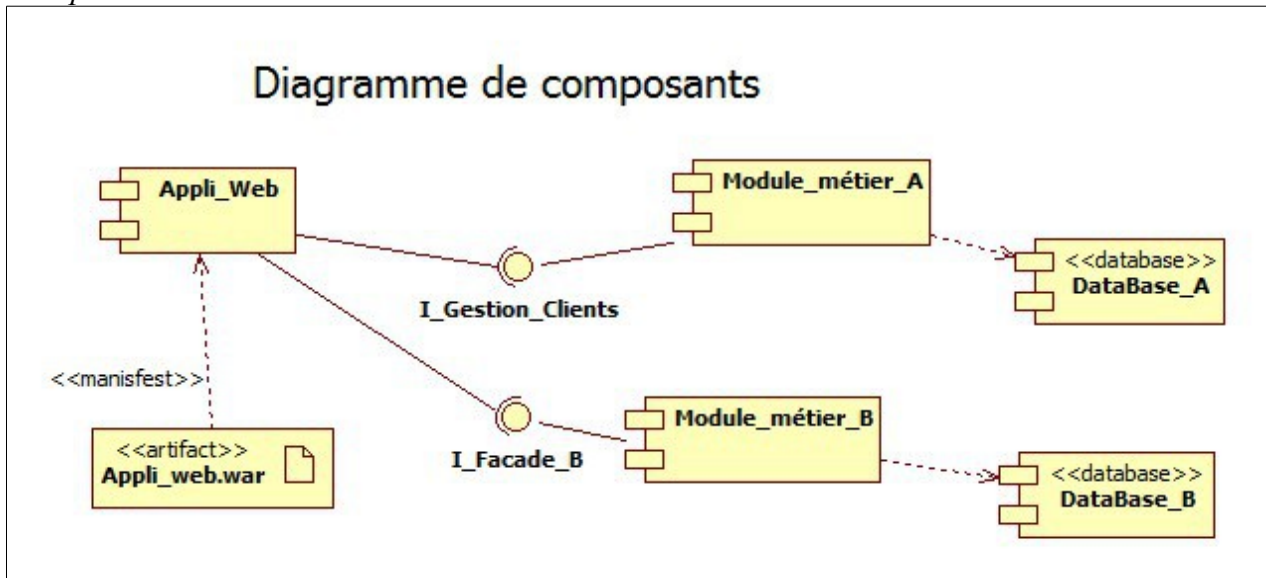
exemple:





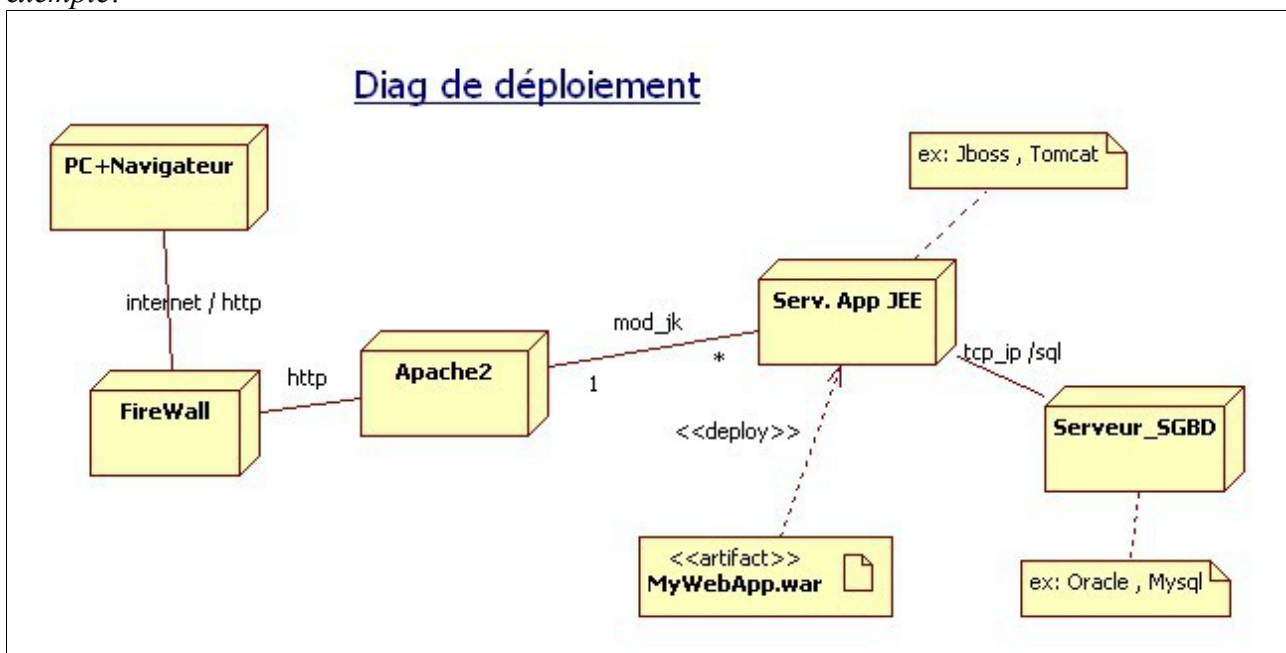
## 8.7. Diagramme de composants

exemple:



## 8.8. Diagramme de déploiement

exemple:



## 8.9. 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)

....

## 9. Utilisations courantes des diagrammes UML

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

## 10. Quelques outils UML (Editeurs , AGL)

Outils/AGL UML	Editeur	Open Source ?	Caractéristiques
<b>Rational Rose</b> --> <b>Rational XDE</b> ---> <b>RSA / RSM</b>	Rational ---> IBM	non	Ancien leader du marché (dans les années 1996/2003) .Bon produit (très complet) mais assez cher .( <b>Rational Software Modeler</b> )
<b>Together</b>	--> Borland	non	Outil très ancien. Évolution récente ?
<b>Poseidon UML</b> for Java .	Gentleware	non	version de base presque gratuite (anciennement gratuite) basée sur Argo UML . Bonnes fonctionnalités .  Ergonomie moyenne / correcte.
<b>Star UML</b>  (Une nouvelle version est en train de ré-apparaître . À tester)		oui	Produit gratuit assez complet (très inspiré de Rational Rose) .  <u>Avantage</u> : très bien dans l'état (intuitif , facile à utiliser)  <u>Défaut</u> : n'a pas évolué depuis 2005 évoluera plus (développé ancien langage "Delphi").
<b>Enterprise Architect</b>	Sparx	Non mais pas cher (250 euros)	Basé sur environnement Microsoft .NET Outil assez complet , bonne ergonomie
<b>MagicDraw UML</b>	NoMagic	non	Bon produit , intègre très bien les normes récentes mais prix caché .
<b>Visual Paradigm</b>	VP	Non mais pas cher (90 euros)	Bon outil UML (complet et stable) mais basé sur technologie assez ancienne.
PowerAMC Designor	SDP	non	Outils pour MCD/Merise avec maintenant une partie UML
<b>Visio</b> (avec partie UML)	Microsoft	Non ( environ 400 euros)	Outil graphique généraliste avec partie UML
Objectteering UML	Softeam (fr)	non	ergonomie très moyenne (ancien produit)
<b>Modelio</b>	Softeam (fr)	Cœur open source , extensions payantes	Bonne ergonomie , fichiers très propriétaires avec néanmoins import/export XML.
<b>Eclipse UML</b>	Omondo	oui	plugin UML pour eclipse (pour round trip)
<b>Topcased UML</b> (plugin <b>Papyrus</b> )	Topcased.org → Polarsys. <b>Projet eclipse</b>	oui	Bon Plugin UML pour eclipse (bien/ très complet mais un peut sembler compliqué au départ)
<b>UML-Designer</b> et <b>sirius</b> (uml + ...)	OBEO	oui	UML-Designer est assez proche de Papyrus mais est néanmoins concurrent sérieux.
<b>GenMyModel</b>	Startup près de Lille	non	Outil UML en ligne (nécessitant un simple navigateur) . Quelques limitations (en v1)
...			

## III - Démarche , études de cas , ...

### 1. Activités de modélisation et spécifications

#### Activités de Modélisation - principales disciplines:

- **Modélisation métier (business modeling)**  
(de niveau entreprise , sous systèmes, organisationnel  
+ objectifs / utilités ?)
- **Modélisation du contexte d'utilisation**  
( contexte ? , environnement technique ? , cadrage ?)
- **Expression des besoins liés au système à développer**  
(fonctionnalités ? , scénarios ? , IHM ? )
- **Analyse** (services et entités internes? , collaboration  
dynamique entre les constituants? , ...)
- **Conception** (architecture technique ? , ... , modules ? ,  
composant ? contrats entre modules ? , détails ?)
- **Implémentation & tests** ( fonctionne bien ? , ...)

1) Pourquoi?

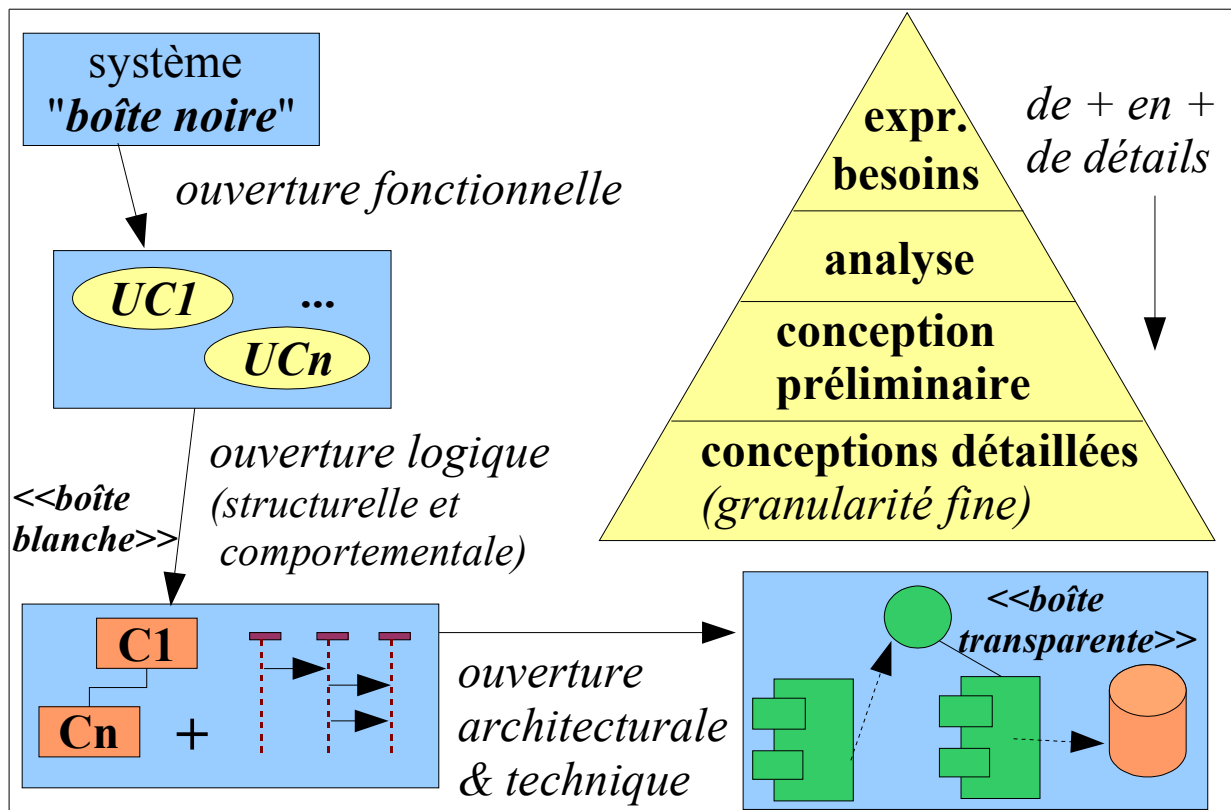
2) Quoi?

3) Comment?

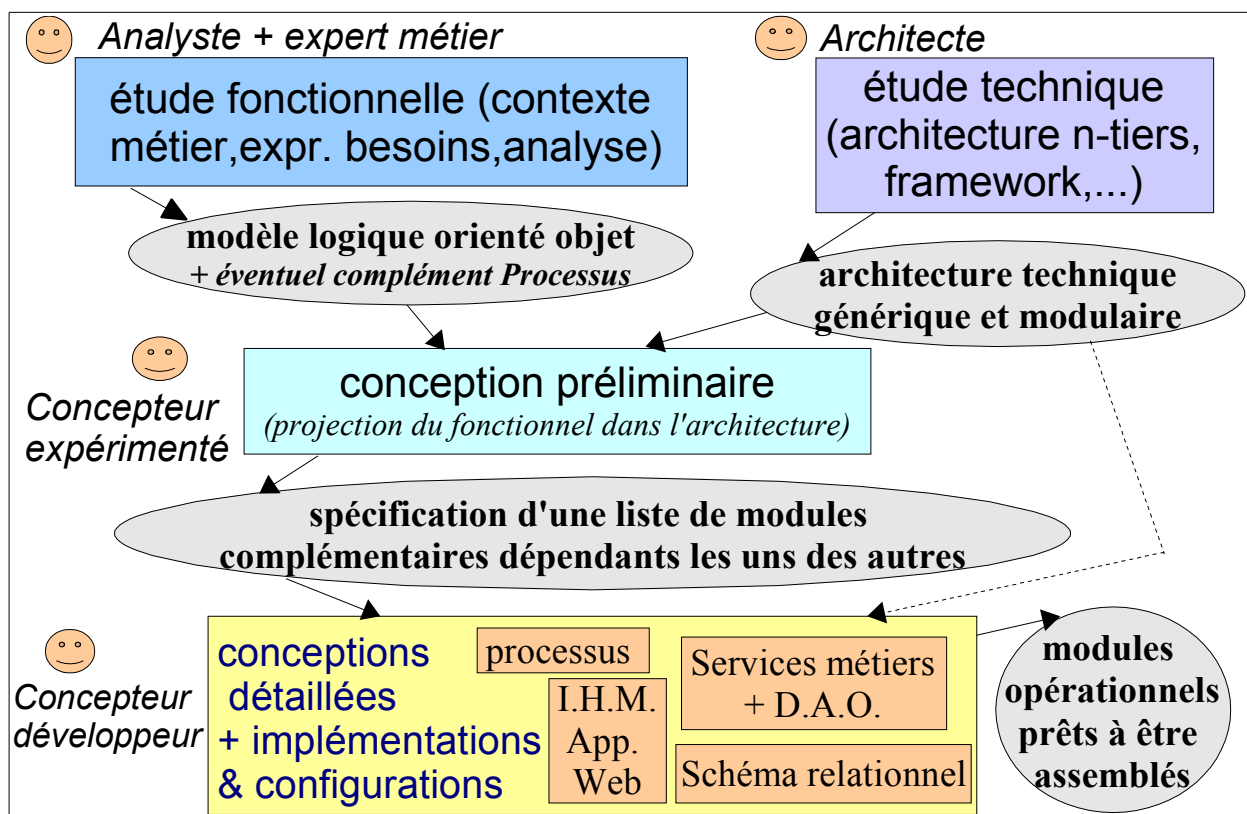
#### Fruits de la modélisation - principales spécifications:

- **Spécifications "métier" et contextuelles**  
(périmètre applicatif , contexte métier et organisationnel)
- **Spécifications fonctionnelles générales**  
(entités du domaine+ fonctionnalités (U.C.) + IHM  
+ services métiers)
- **Spécifications fonctionnelles détaillées**  
(+ réalisations des U.C. , + architecture logique)  
====> **modèle logique complet et précis**
- **Spécifications purement techniques**  
(plans types pour technologies , frameworks, ....)
- **Spécifications applicatives et techniques**  
====> **modèle physique** (dans les grandes lignes ,  
composants/modules , contrats/interfaces, ...)

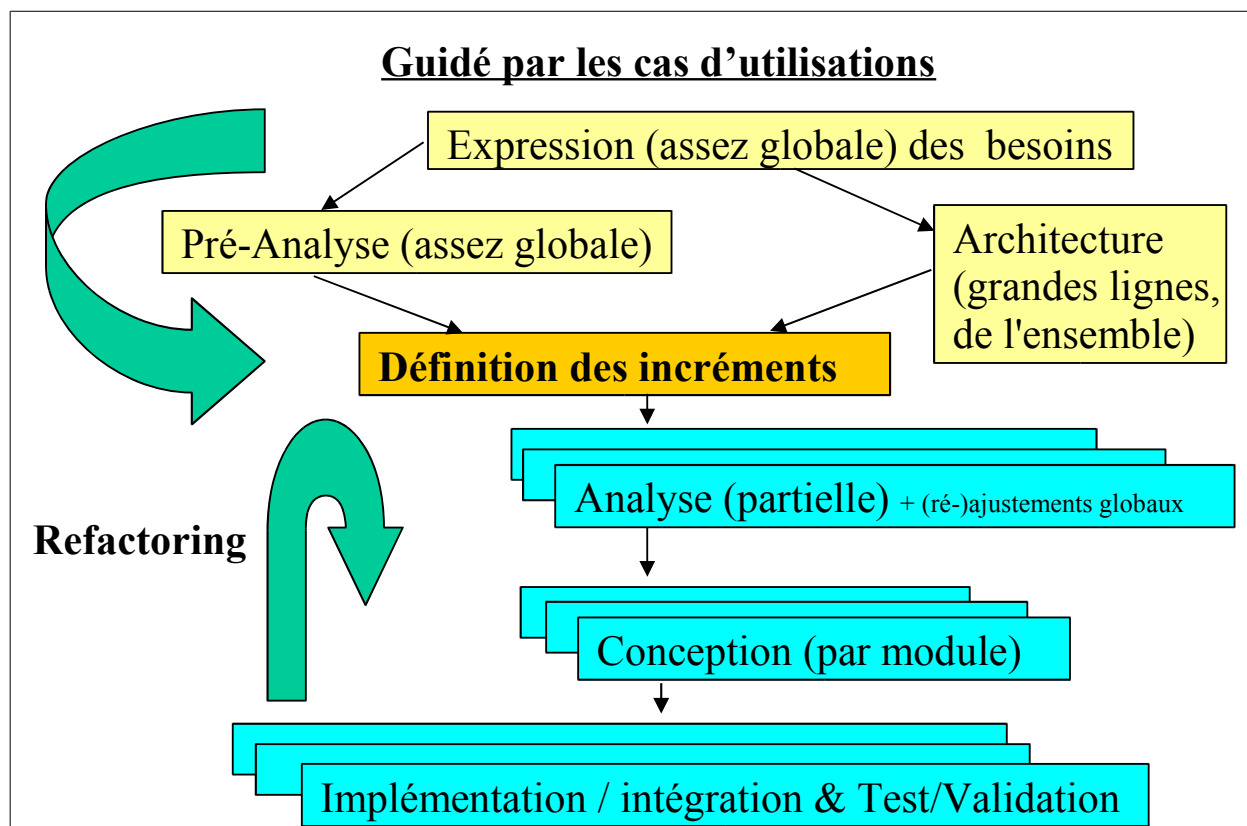
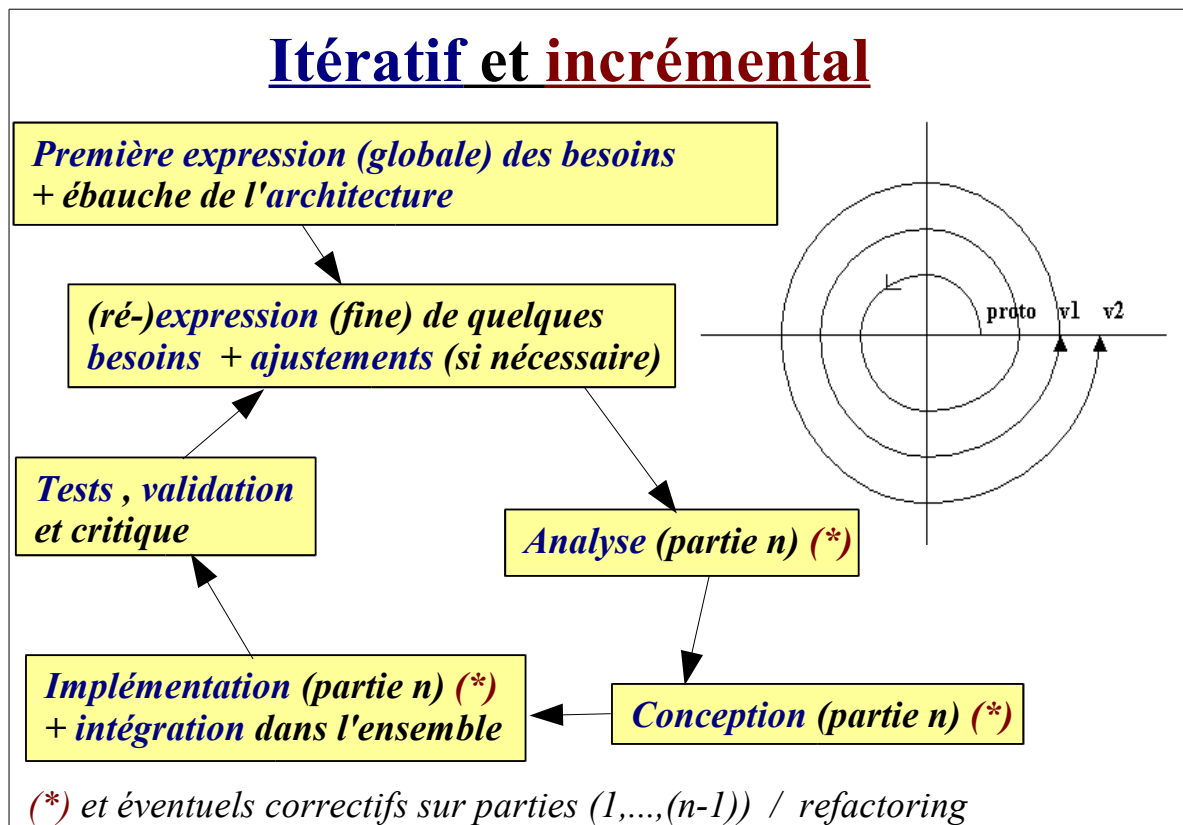
## 1.1. Vue imagée (modélisation, démarche progressive)



## 1.2. Chacun sa spécialité (analyse, architecte, ...)

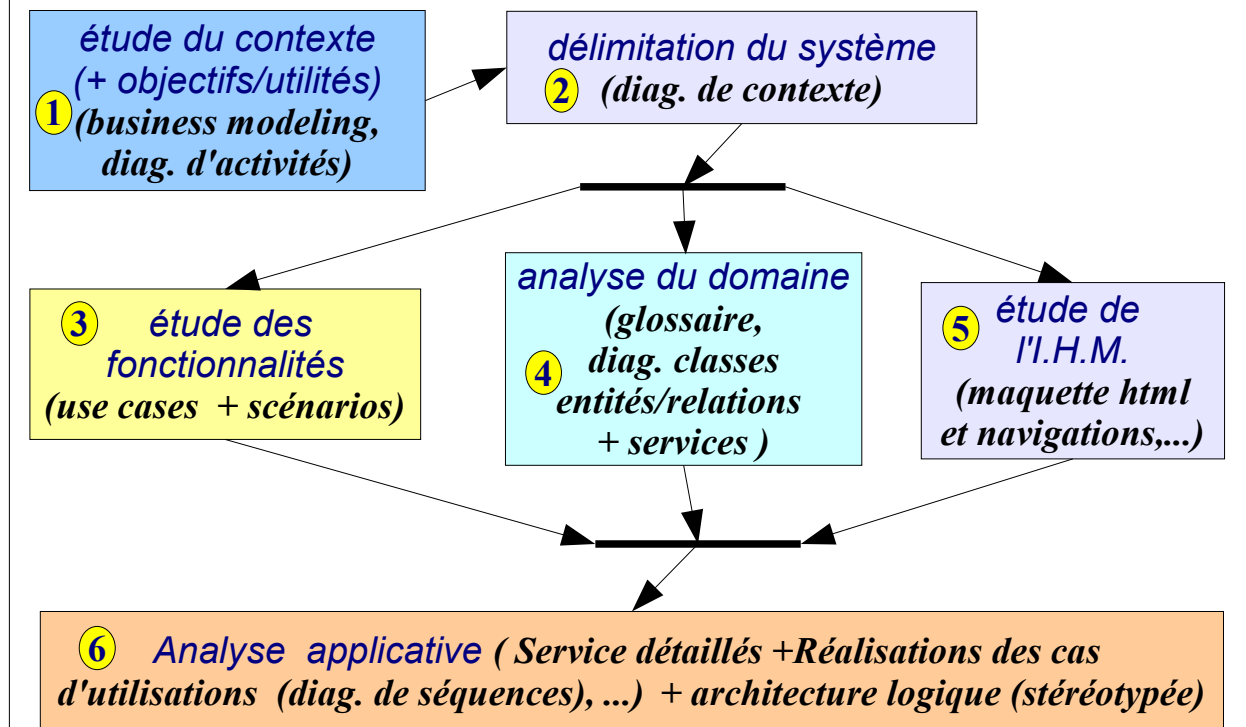


## 2. Cycle "itératif & incrémental"



### 3. Enchaînement des activités de modélisation

#### *Enchaînement classique d'activités sur la partie fonctionnelle*



### 4. Bibliothèque / médiathèque (étude de cas)

#### 4.1. Etude de cas "bibliothèque"

(version simple/réalisable pour TP sur 2 à 3 jours):

Dans une bibliothèque, des abonnés peuvent emprunter des exemplaires d'oeuvres littéraires auprès du bibliothécaire qui enregistre la date de début et la durée de l'emprunt.

Un responsable abonnement prend en compte les inscriptions des nouveaux abonnés et gère les contentieux (exemplaires non rendus dans le délai imparti, abîmés ...).

Un abonné est identifié par un numéro et décrit par ses nom, prénom, adresse et âge.  
Pour chaque abonné on connaît le nombre d'exemplaires empruntés et non encore rendus.

Une oeuvre littéraire est caractérisée par son titre.  
Chacune peut comporter plusieurs exemplaires identiques.  
On connaît le nombre d'exemplaires disponibles.  
Chaque exemplaire est identifié par un numéro.  
On souhaite en connaître l'état.

Un exemplaire neuf doit être enregistré par le bibliothécaire pour être disponible.  
[Un enregistrement comptable sera éventuellement généré/déclenché en déléguant cette fonctionnalité à un logiciel externe "système\_comptabilité" ]



Le bibliothécaire vérifie l'état de chaque exemplaire restitué.  
Tout exemplaire abîmé est donné à l'atelier de reliure pour restauration.  
Cet exemplaire n'est plus disponible.  
Une fois restauré, l'exemplaire est réintégré par le bibliothécaire.

#### **4.2. Eventuelle petite variante sur l'étude de cas :**

*(pour obtenir un ex d'héritage)*

- Bibliothèque ==> Médiathèque
- Exemplaire d'oeuvres littéraires ==> Exemplaire de Livre/BD ou CD ou DVD ou ...

#### **4.3. Eventuelle petite extension sur l'étude de cas :**

*(Pour éventuelle version/itération 2)*

- + borne (ou pc ou site\_web) de consultation du fond de la bibliothèque  
recherche par catégorie et/ou par auteur , ....
- + éventuelle réservation (si aucun exemplaire dispo)

### **5. Agence de voyage (étude de cas alternative)**

#### **5.1. Etude de cas "Agence de voyage"**

FH (Fun Holiday) est une agence de voyage fictive qui a:

- 1 siège social basé à Paris.
- une cinquantaine d'agences en province.

Cette société propose à ses clients des solutions complètes :  
transport + séjour (hôtel et repas) + activités.

FH souhaite se doter d'un système informatique permettant de:

- gérer les réservations (depuis agence + depuis internet ).
- consulter le catalogue des séjours ( + gérer offres promos).
- gérer l'aspect logistique (avions , trains , guide , ...)

#### **5.2. Grands choix techniques et contraintes:**

- Développement itératif et incrémental.
- UML , architecture n-tiers, internet.
- SGBDR (Oracle , MySQL , ....)
- Technologies "Java/JEE"
- SOA , ESB

#### **5.3. Spécifications du cahier des charges:**

**Gestion du catalogue:**

V1 --> Consultation (lecture seulement) du catalogue des séjours .

V2 --> Mise à jour possible du catalogue par un agent habilité



V3 --> gérer des offres promotionnelles  
(basées sur une offre du catalogue , mais dates fixes et prix réduits).

**Gestion des réservation:**

Gestion du nombre de places  
Depuis agence (traitements éventuels de formulaires "papier" envoyés par courrier ).  
Depuis internet (le paiement sécurisé sera délégué à la banque BqY ).

**Gestion de la logistique:**

Annulation si trop peu d'inscrits 10 jours avant le départ.  
Mission "guide de FH" ou sous-traitance des activités  
Transport (Etapas,...)  
...

## 6. Autres étude de cas

### 6.1. Frontal Web / consultation de comptes & transferts

Etudier/Modéliser un nouveau sous système informatique "*Frontal\_Web\_Banque*" qui communiquera avec le SI existant de la Banque et qui permettra de :

- consulter les comptes sur internet
- effectuer des virements internes
- lister les dernières opérations effectuées sur un compte (sélectionné)

NB: une authentification du client est indispensable .

### 6.2. Autre(s)

====> à proposer / réaliser (ex: location/réservation de voitures , ....).

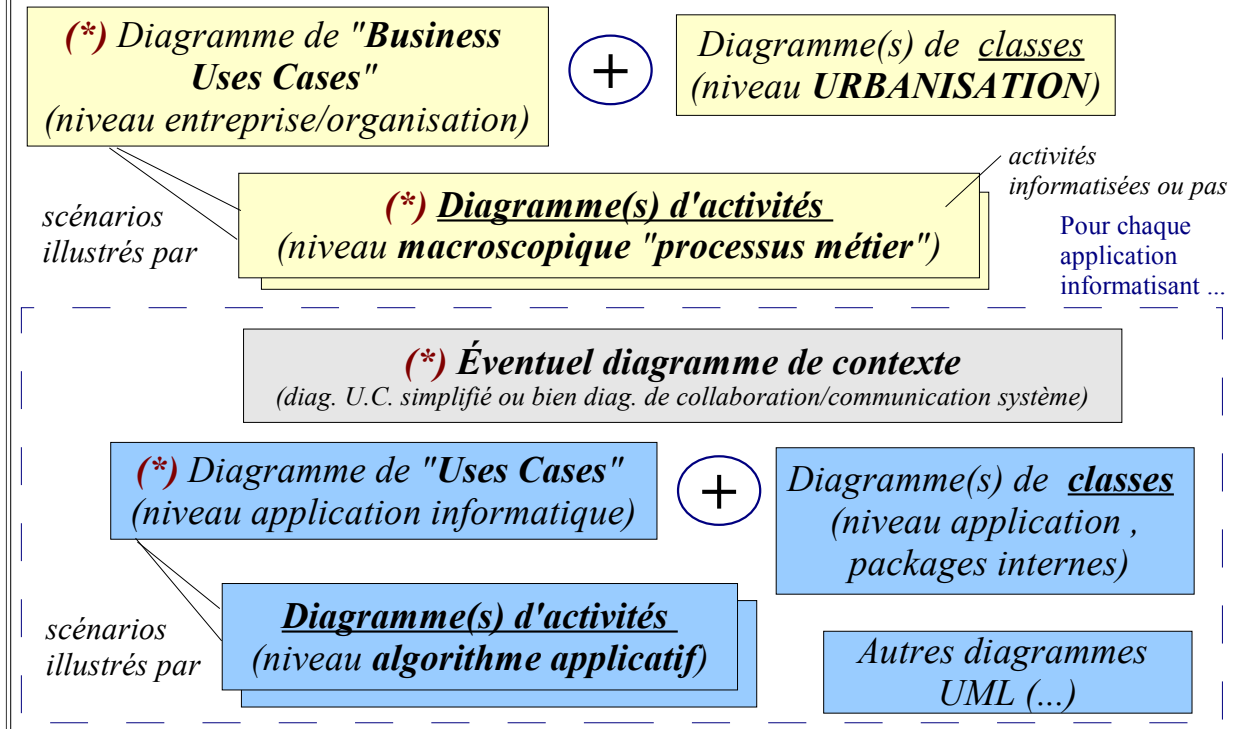
## IV - Diagramme de contexte / périmètre applicatif

### 1. Début de la modélisation UML ?

Qui est le premier ?

L'œuf ou la poule ? L'essence ou l'existence ? ....

#### **(\*) début de la modélisation UML ? (U.C. ou diag. Activités) ?**



Si l'on modélise une application informatique au périmètre clairement identifié, on peut commencer la modélisation UML par un diagramme de contexte (soit sous forme de diagramme de uses cases simplifié, soit sous forme de diagramme de collaboration/communication).

Si l'on tient en plus à :

- illustrer le contexte d'utilisation du logiciel et/ou
- cogiter sur les parties à informatiser ou pas

on peut alors commencer la modélisation UML par des diagrammes d'activités macroscopiques (de niveau "modélisation métier", dépassant la frontière d'une application informatique bien précise).

Si l'on tient en plus à :

- justifier l'utilité des processus métiers (un par objectif ou sous objectif métier) et/ou
- organiser les processus (un par "business use case")

on peut alors commencer la modélisation UML par un diagramme de "uses cases métiers".

Attention, si la modélisation UML porte partiellement sur une portée de type "modélisation métier" il est très conseillé de ranger dans des endroits différents les modèles/diagrammes de niveau "métier" et les modèles/diagrammes de niveau "application informatique précise" :

- \* soit dans des modèles ou packages bien séparés d'un même fichier ".uml"
- \* soit dans des fichiers ".uml" bien séparés

## 2. 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.

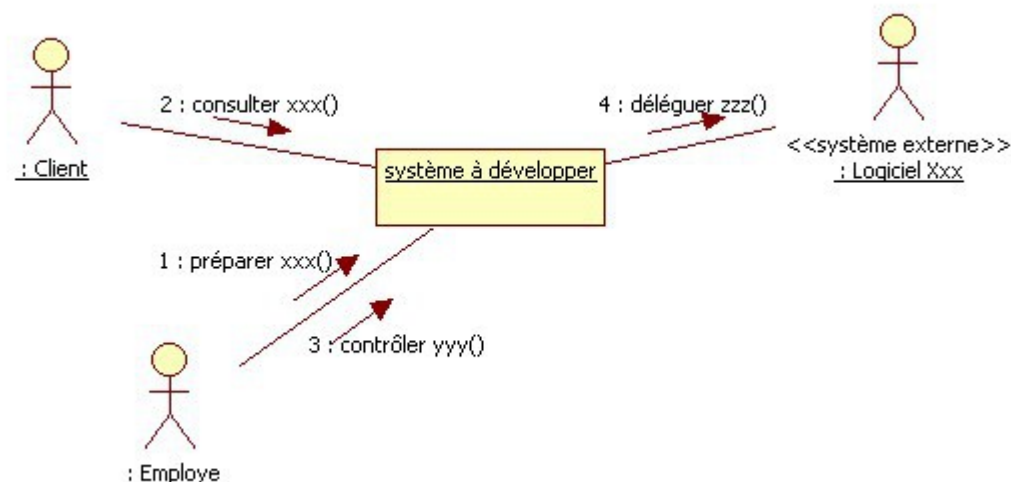
Selon les possibilités de l'outil UML on pourra concevoir le diagramme de contexte :

- 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 le logiciel "StarUML") :

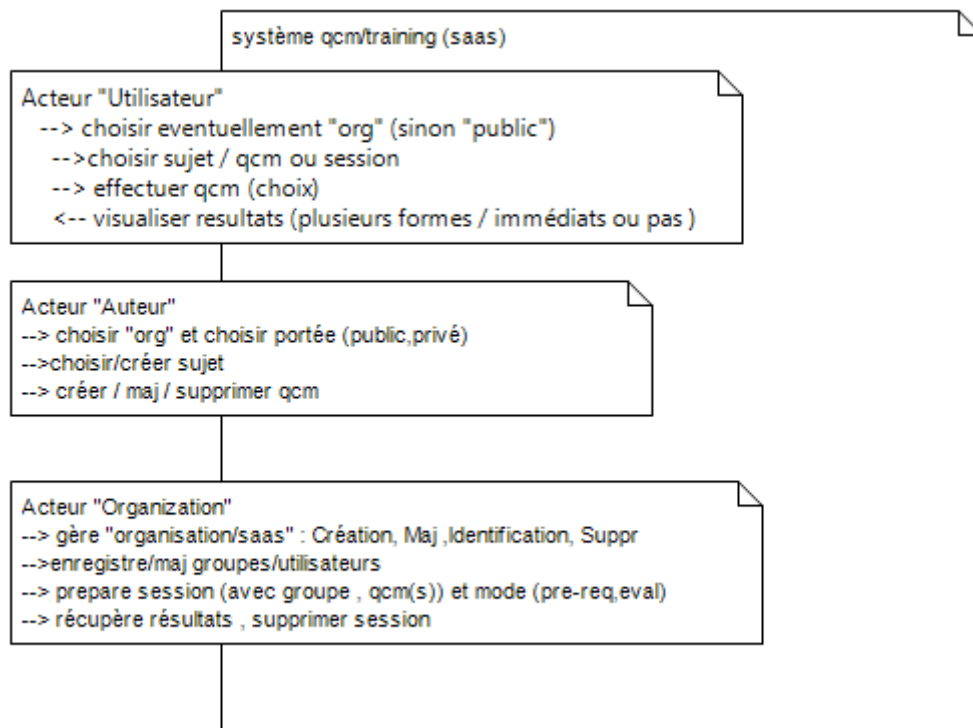


L'objet "système à développer" est placé en plein milieu du diagramme.

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" ou de "Classes" avec des **commentaires** :



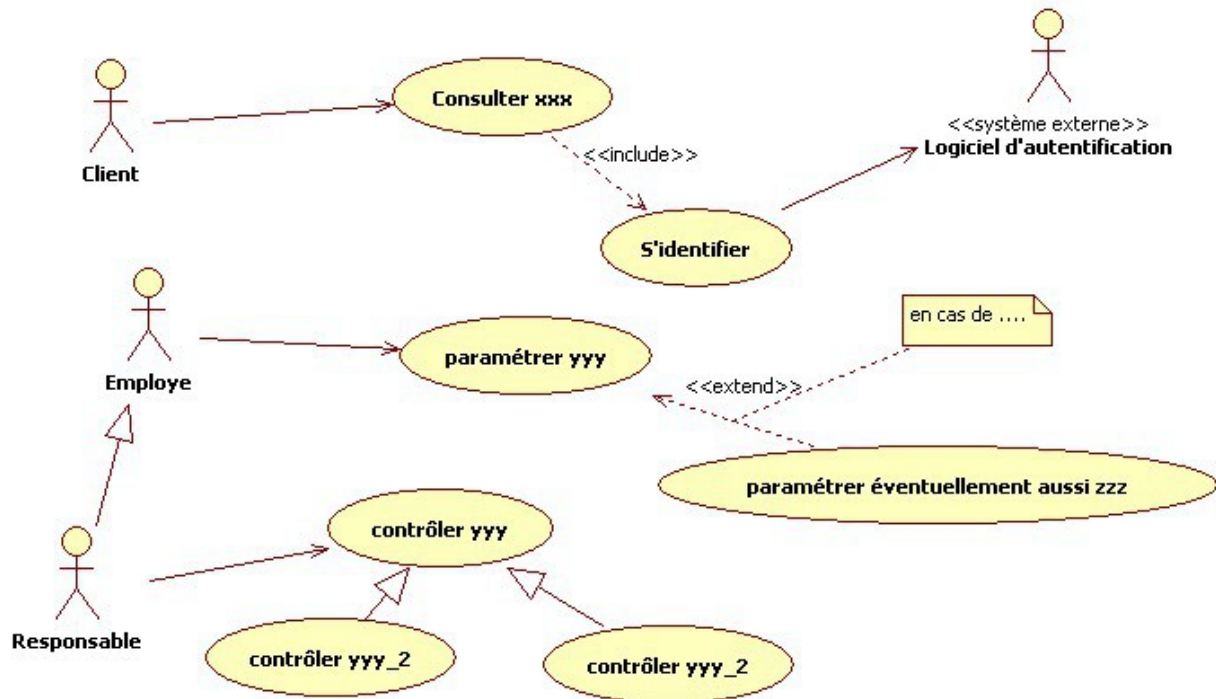
Bien que "non standard" / "non officiel", un tel diagramme de contexte est **très rapide à élaborer** et donne une bonne vue d'ensemble sur les différents "futurs acteurs UML" et les interactions (entrantes ou sortantes) associées.

# V - Expr. besoins fonctionnels / Uses Cases

## 1. Etude des principales fonctionnalités (U.C.)

La notion de "cas d'utilisation" 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.

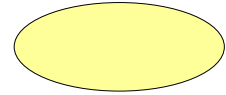
### 1.1. Méthodologie de base (au niveau des cas d'utilisation)

- 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)

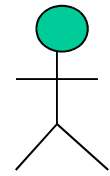
## 2. Diagramme des cas d'utilisations

### Présentation des « Use Case »



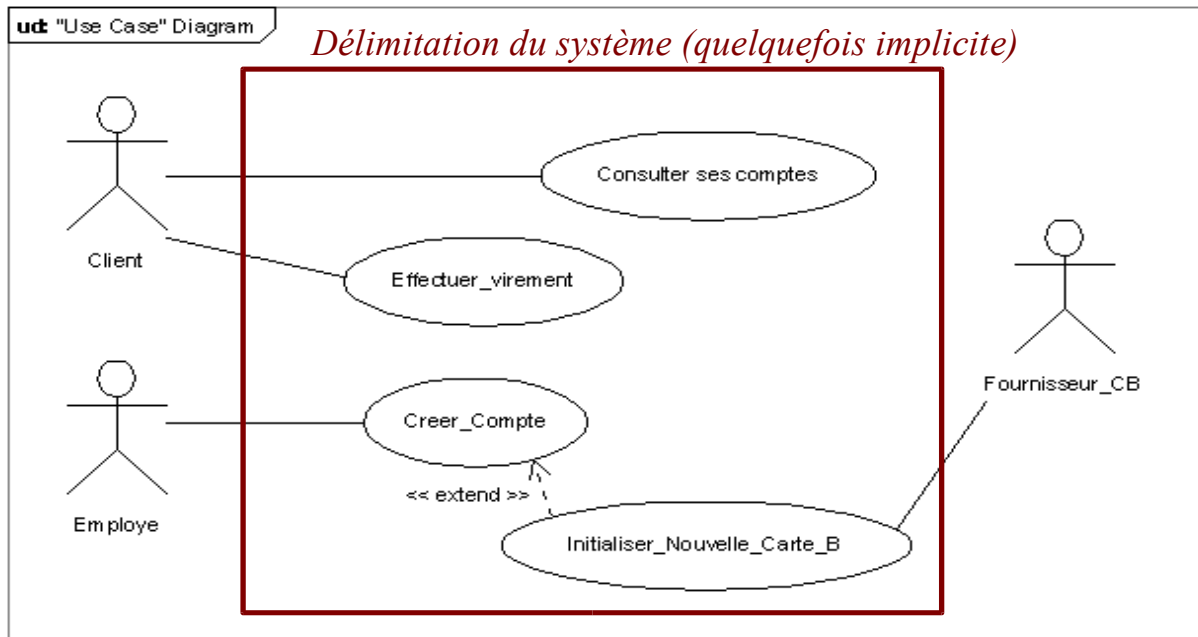
Faisant partie intégrante de UML, les "USE CASE" offrent un **formalisme** permettant de:

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



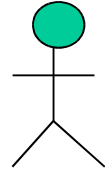
Nb: il s'agit d'une **vue externe** (point de vue de l'utilisateur).

### 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 automate 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).*

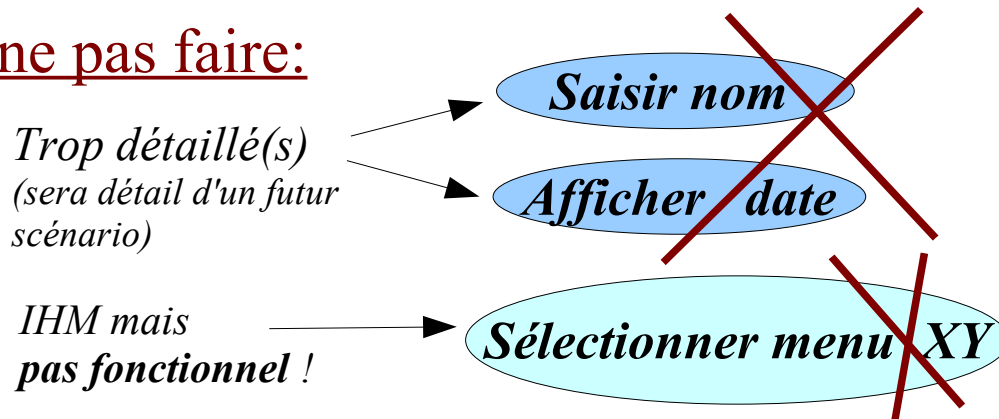
Notation:



### A ne pas faire:

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

*IHM mais  
pas fonctionnel !*



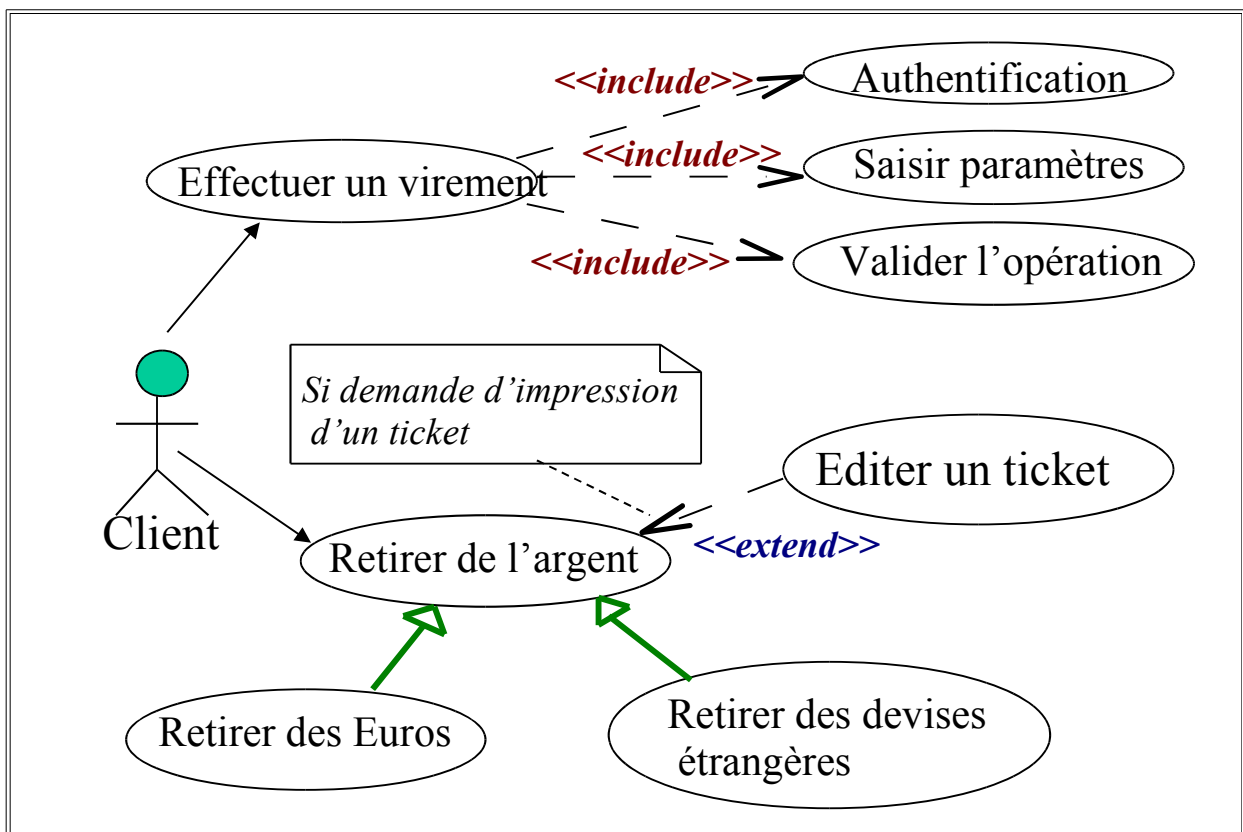
### A prendre en compte:

- \* 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 .

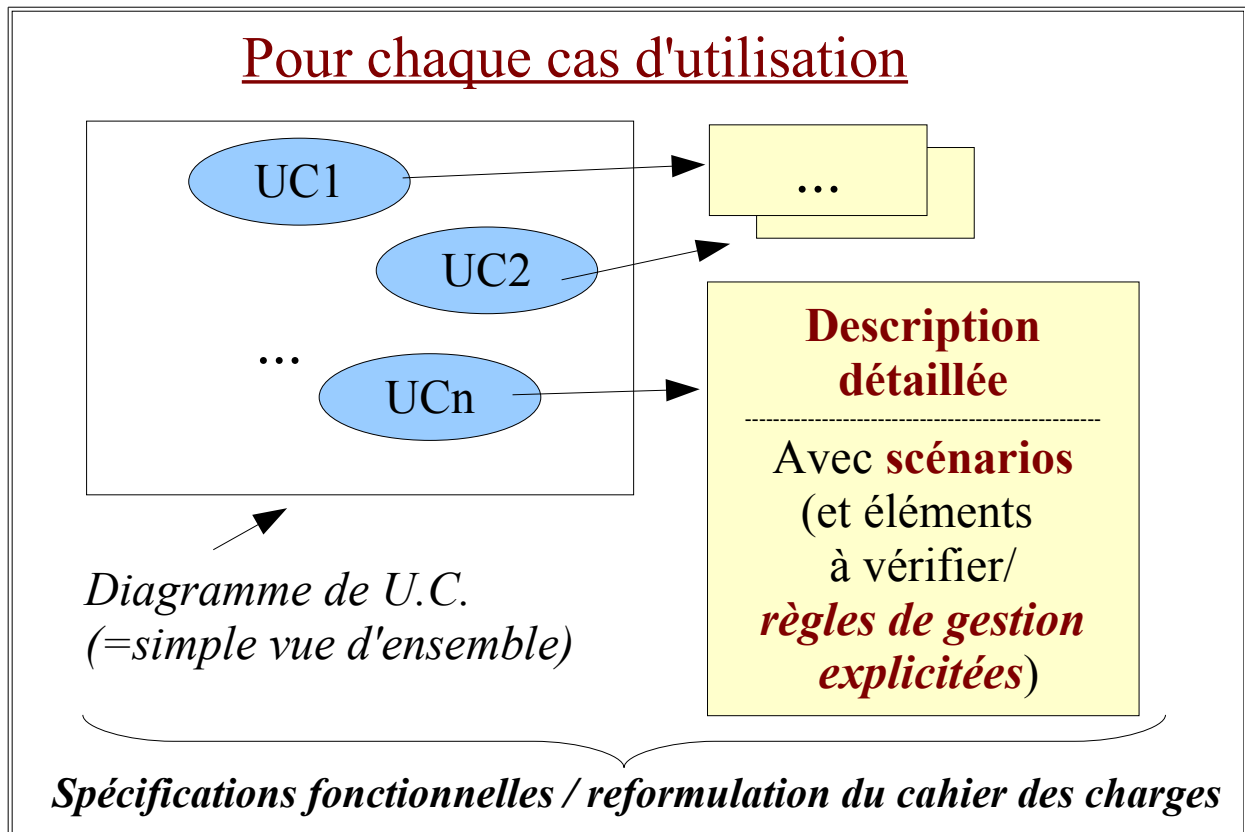


## Relations entre UC

- 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 supplémentaire qui s'ajoute à A le cas échéant (facultativement).**  
Une **note (commentaire) spéciale** appelée « **cas d'extension** » permet de préciser le cas où C étend A.
- La **relation d'héritage** (ou généralisation) classique: **D  $\rightarrow$  E** permet d'exprimer que le cas d'utilisation D est une **sorte (variante, déclinaison)** du cas d'utilisation E.



### 3. Scénarios et descriptions détaillées (U.C.)



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

### *Scénario Nominal*

----  
Tout se passe bien

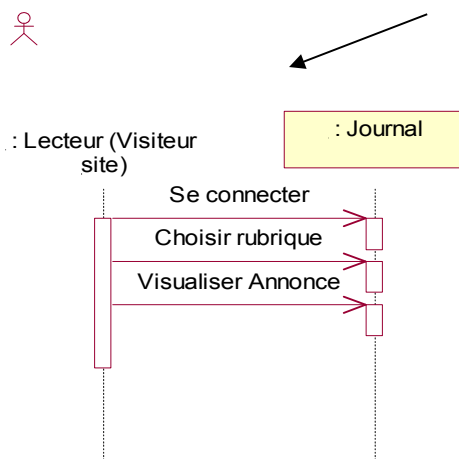
----  
Cas/Déroulement  
le plus fréquent

### Scénario d'exception "E1":

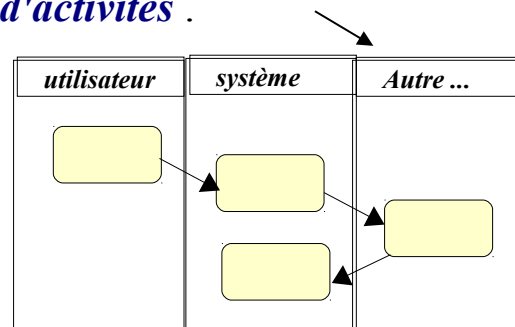
- si l'étape (3) échoue trois fois de suite alors
- avaler carte
  - ne pas effectuer les étapes (4) à (7)
  - afficher un message explicatif

## Illustration éventuelle (U.C.)

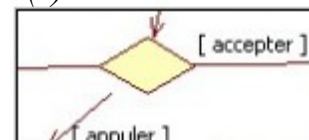
Un scénario peut éventuellement être graphiquement exprimé/illustré par un *diagramme de séquence UML*.



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



Avec *décision(s)* :



**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>> (-,--)

## 4. Méthodologie fonctionnelle détaillée

### Vocabulaire :

Cas d'utilisation principaux ==> de niveau "tâche"

Sous cas d'utilisation (sous fonctionnalités) ==> de niveau "fonction"

### 4.1. première étape (niveau "tâche")

Pour toutes les interactions entre les acteurs primaires et le système (cf. diagramme de contexte de la phase d'étude préliminaire) définissez un UC correspondant:

**Nom :** celui de l'interaction.

**Niveau :** *Tâche*

**Objectif :** cf. définition de l'interaction

**Scénario nominal :** à écrire !

**Scénario d'erreurs :** à ne pas écrire tout de suite !

### 4.2. deuxième étape (niveau "fonction")

Pour chacun des pas du scénario nominal d'un UCs niveau Tâche, essayez de définir un sous UC correspondant :

**Nom :** celui du pas

**Niveau :** *Fonction*

**Objectif :** celui du pas

**Scénario nominal :** à écrire !

**Scénario d'erreurs :** à ne pas écrire tout de suite !

### 4.3. Troisième étape (sur itération ultérieure)

==> scénarios d'erreur .

(seulement une fois la base bien stabilisée et validée par un prototype) !

### 4.4. Organisation conseillée pour les diagrammes

- généralement un diagramme général représentant tous les UC de niveau Tâche.
- un diagramme par UC niveau *tâche* : l'UC niveau *tâche* et les UCs de niveau *fonction* associés.

## 5. Gestion de projet et planification basées sur les cas d'utilisation

### Critères pour définir les priorités à attacher aux cas d'utilisation:

- Priorité fonctionnelle a priori (celle du client)
- Fréquence d'utilisation.
- Inclus > incluant > extension > spécialisation\*
- Nombre d'UC l'incluant
- Risques estimés (difficulté technique , ... )
- Ordre d'utilisation (1<sup>er</sup> > 2<sup>e</sup>...)

(\*) priorité selon indications du client et selon ( <<include>> + , <<extends>> - )

(\*\*) estimation de charge = temps total pour :

- modélisation (partie analyse + conception)
- développement
- tests unitaires

## 6. Eventuelle planification des livrables selon XP

### planification prévisionnelle initiale:

Livrables prévus sous les 20 premiers jours	UC1 , UC6 , UC4
Livrables prévus sous les 20 jours suivants (globalement 40 jours après le début)	UC3 , UC7 , UC5
Livrables prévus sous les 20 jours suivants (globalement 60 jours après le début)	UC2 , UC8

**planification revue au bout de 20 jours ouvrés:**

**[ retard sur UC4 (2 jours) + UC9 = nouveau besoin prioritaire du client (5j/h) ]**

Livrables réalisés et livrés sous les 20 premiers jours	UC1 , UC6
Livrables prévus sous les 20 jours suivants (globalement 40 jours après le début)	<b>fin_UC4 , UC9 , UC3 , UC7</b>
Livrables prévus sous les 20 jours suivants (globalement 60 jours après le début)	<b>UC5 , UC2 , UC8</b>

**planification revue au bout de 40 jours ouvrés:**

**[ retard sur UC7 (2 jours) + UC10 = nouveau besoin prioritaire du client (6j/h) ]**

Livrables réalisés et livrés sous les 20 premiers jours	UC1 , UC6
Livrables réalisés et livrés sous les 20 jours suivants (globalement 40 jours après le début)	<b>UC4 , UC9 , UC3</b>
Livrables réalisés et livrés sous les 20 jours suivants (globalement 60 jours après le début)	<b>UC10, fin_UC7 , UC5 , UC2</b>
<b><i>Eléments jamais livrés ou bien livrés plus tard si avenant et budget .</i></b>	<b>UC8 (le moins prioritaire !!!)</b>

## VI - Diagramme d'activités

### 1. 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 ).

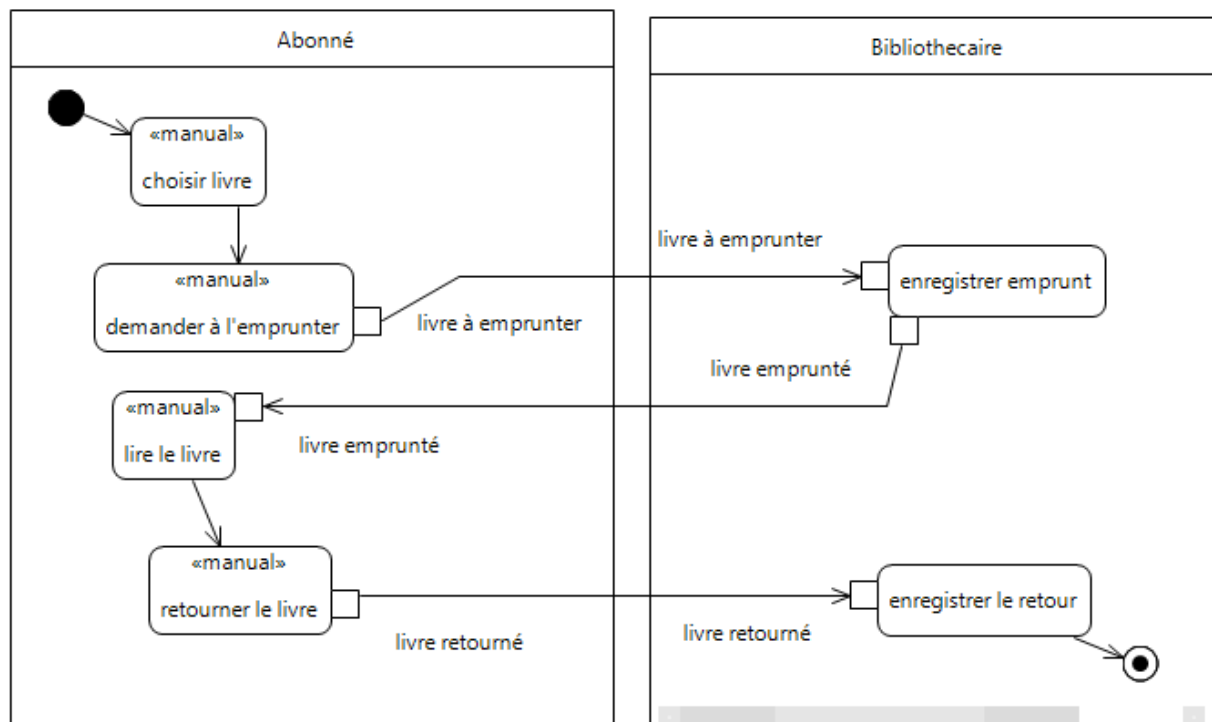
Principales utilisations:

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

#### 1.1. 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**" :

emprunter des livres pour les lire



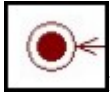
Au sein de l'exemple ci dessus le stéréotype <<manual>> (non normalisé) peut aider à préciser si une tâche/action est plutôt manuelle (ou informatisée sinon) et les petits rectangles blancs sur les

bords des actions correspondent à des "output pin" et "input pin" associés à des "object flows".

## 1.2. 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 :



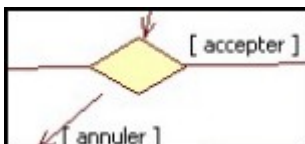
bifurcation (traitement en //)



union/synchronisation de type "et logique"

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

**Décision** et *condition de garde* entre [ ] :





### 1.3. nœuds d'activités et variantes (actions, ...)

Un diagramme d'activités UML (activity group) est un graphe dont la plupart des nœuds sont des nœuds d'activités/actions (correspondants à des traitements ou des tâches).

UML2 distingue plusieurs types de nœuds d'activités (ou d'action) selon la granularité et la nature des traitements à modéliser.

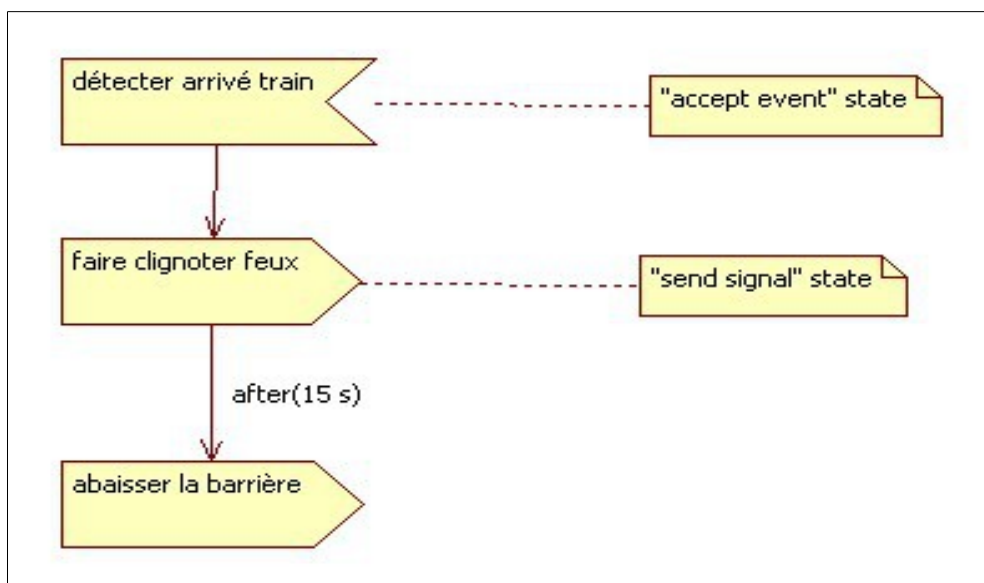
#### Types d'actions/activités souvent utiles en **expressions des besoins** :

<b>Opaque Action</b>	Action/activité quelconque (dont la nature pourra être ultérieurement précisée/affinée en conception)
<b>Call behaviour</b>	Appel global d'une autre sous activité (sans mentionner une opération précise). Souvent associé à un lien hypertexte vers autre diagramme.

#### Différents types très (trop) précis d'actions (en général pour la conception):

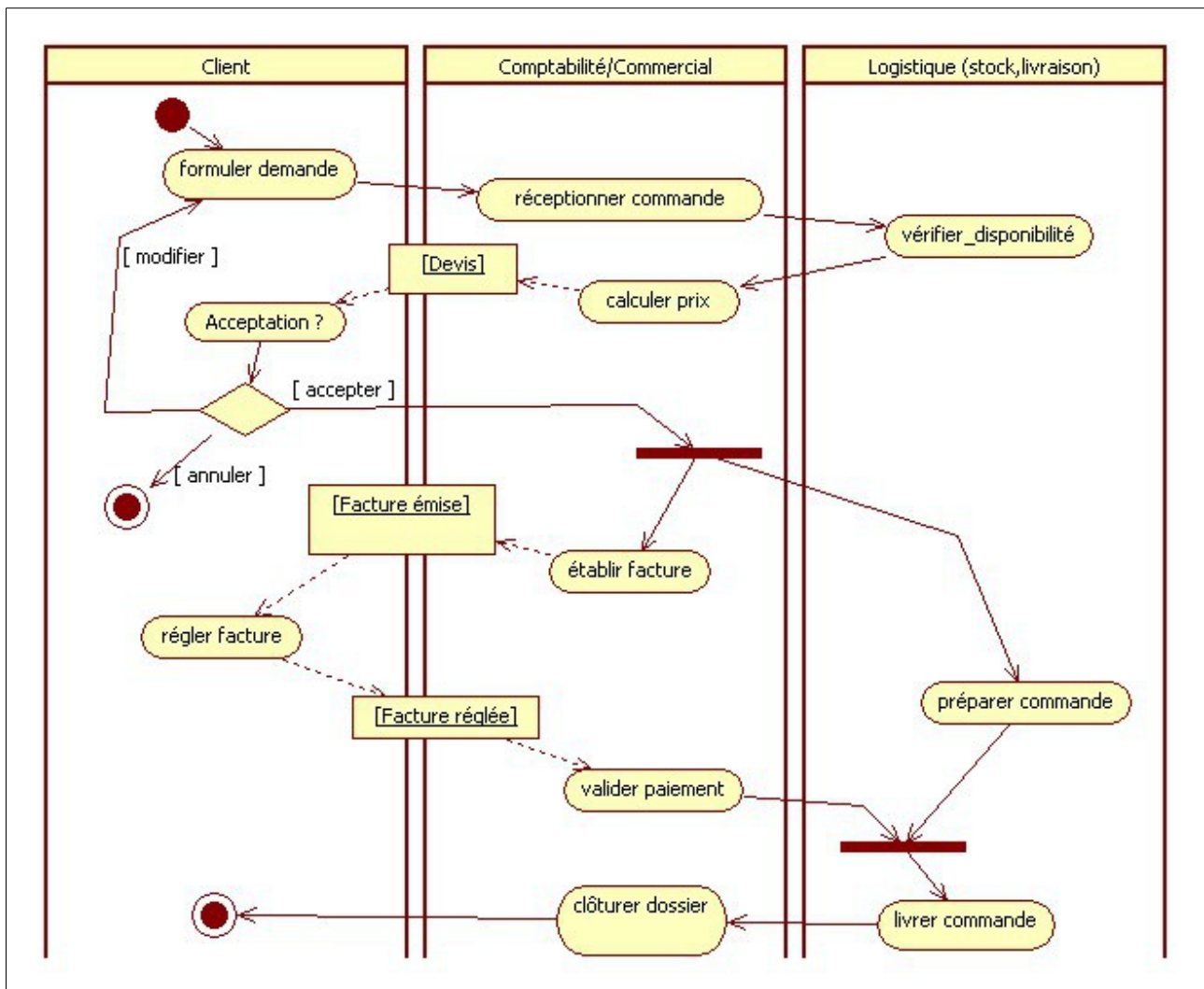
<b>Call operation</b>	Appel (en mode souvent synchrone ou rarement asynchrone) d'une méthode/opération avec passage possible de paramètre et récupération potentielle d'une valeur de retour
<b>Send</b>	Envoi d'un message ou d'un signal
<b>Accept event</b>	Attente (bloquante) d'un événement (souvent lié à une notification asynchrone)
<b>Accept call</b>	Variante de "accept_event" pour les appels entrants (avec réponse à donner ultérieurement via reply)
<b>Reply</b>	Répondre (lié à un accept_call)
<b>créer/instancier</b>	Créer un nouvel objet
<b>destroy</b>	Détruire un objet (ex: delete en c++)
<b>Raise exception</b>	Soulever (remonter) une exception

### 1.4. Notations pour actions particulières (UML2)



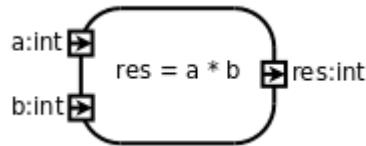
## 1.5. object flow (nœuds "objet") : UML 1 et 2

Un nœud "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 , ....).

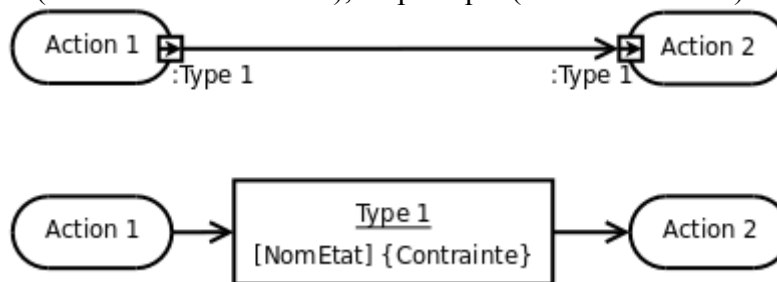


## 1.6. Pins et buffers (UML 2)

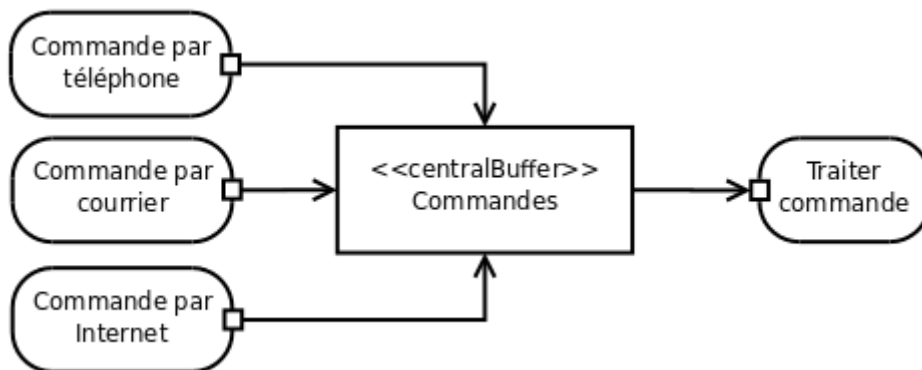
Depuis la version 2 d'UML, il faut placer des points ("pin") d'entrée ou de sortie sur une activité pour préciser un lien (éventuellement typé) avec les "object flow" entrant(s) ou sortant(s).  
[ avec une sémantique de passage de valeur(s) par copie(s) ] .



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.

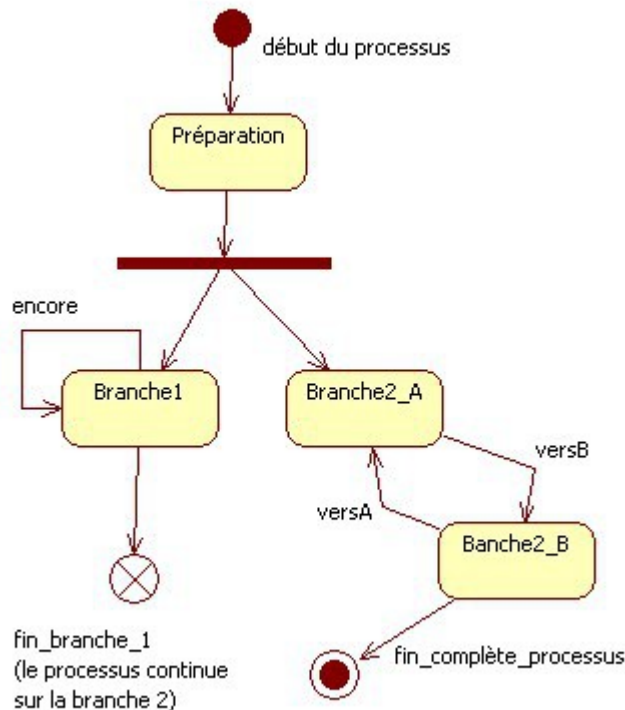


NB: Lorsque le buffer/tampon intermédiaire gère en outre la persistance des données (dans une base de données ou ...) , on utilise alors <<dataStore>> plutôt que <<centralBuffer>> .

## 1.7. Final flow

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

Exemple :



## 2. Distinction "micro activité / macro activité" selon granularité

Un **diagramme d'activités UML** peut servir à modéliser des choses à des échelles (ou niveaux de granularité) assez variables:

- **macros activités :**
  - *durées* potentiellement *longues* (plusieurs jours)
  - avec souvent pleins d'intervenants (ex: client , logistique , fournisseurs , ....)
  - *liées à des "buts/objectifs métiers" représentés via des "business uses cases"*
  - ...
- **micros activités (actions)**
  - *durées limitées* (session utilisateur , quelques minutes )
  - centrées sur un intervenant principal (acteur primaire)
  - *associées à des cas d'utilisations (uses cases)*
  - ...

## VII - Règles de gestions

### 1. Formulations des exigences et traçabilité

Au sein des spécifications fonctionnelles bâties autour d'UML, les exigences sont généralement formulées aux endroits suivants :

- Règles de gestions génériques systématiques (exprimées mode texte) à appliquer partout.
- Règles courantes nommées (exprimées mode texte) à appliquer si référencées par commentaire (ou autre) dans un diagramme UML.
- Règle métier de type "vérifier solde suffisant ( > seuil )" clairement formulée en tant qu'élément d'un scénario d'un cas d'utilisation .
- Contrainte d'intégrité (formulée comme contrainte UML de type { age > 0 } ) au sein d'un diagramme de classes .

### 2. Exemples d'expression des règles de gestion

#### 2.1. Règles de gestion génériques

NB : Les différentes règles de gestion récurrentes qui suivent devront être systématiquement appliquées sur l'ensemble des formulaires (de saisies) de l'application :

<i>Références des règles de gestion</i>	<i>Définitions de ces règles</i>
rMajNomPropre	Tout nom propre devra commencer par une majuscule (ex : Nom de Personne , nom de Pays , ....)
rDateDebutFin	Toute période (avec date de début et date de fin) devra respecter la règle élémentaire suivante <code>date_fin &gt;= date_debut</code> .
rValeurDecimale	Toute valeur décimale (avec virgule) devra soit être saisie en deux parties une partie "entière" et une partie "décimales_après_virgule" (ex : centimes) ou bien saisie comme une chaîne de caractères retraitées/réanalysée où le séparateur pourra aussi bien être "." que "," .
...	

D'autre part , certaines des règles génériques suivantes seront à respecter aux endroits de la spécification où l'on y fera référence :

<i>Références des règles de gestion</i>	<i>Définitions de ces règles</i>
rMaj	A saisir entièrement en majuscule
rVerifAdresseFr	Vérification d'une adresse française (ville/village existant , codePostal valide, ...) souvent via une technologie de type Ajax (à la volée)
rDateNonPassee	Renseigner une date (par saisie ou par sélection) non passée (valeurs possibles : aujourd'hui ou avenir seulement) , ...
...	

# VIII - Génération documentations / spécifications

## 1. Documentation / Livrables / Spécifications

### Spécifications des exigences:

- notion de jeux de tests associés à telle ou telle exigence,
- traçabilité à gérer à tous les niveaux (modélisation , implémentation , tests )

### 1.1. Part d'UML dans la spécification des besoins

Les différents diagrammes UML utiles à l'expression des besoins (Uses Cases , contexte , ....) doivent normalement être considérés comme:

- des illustrations/reformulations visuelles des principaux aspects du cahier des charges
- des éléments fondamentaux de la modélisation (acteurs , cas d'utilisations , ....) auxquels il faudra se référer pour organiser le projet (règles de traçabilité avec répercussions sur l'implémentation et les jeux de tests)

==> Les éléments clefs du modèle UML (acteurs , cas d'utilisations , ....) doivent donc faire l'objet d'une identification sérieuse (par numéro ou par nom , ....) de façon à pouvoir y faire référence dans les phases ultérieures (analyse , conception , implémentation, tests , ....) et pour s'y retrouver dans les itérations ultérieures (versions 2,3, n ).

### 1.2. Parties de livrables classiques (avec pleins de variantes envisageables)

#### Spécifications du contexte d'utilisation et du périmètre applicatif:

- Diagrammes d'activités illustrant certains processus métiers (contexte d'utilisation)
- Diagramme de contexte (avec acteurs extérieurs et interactions) pour délimiter le périmètre applicatif
- Définition précise de chaque acteur (rôle utilisateur , logiciel externe, ...)

#### Spécifications des fonctionnalités attendues:

- Diagramme(s) des cas d'utilisations (avec relations <<extend>> , <<include>> , ...)
- Description détaillée de chaque cas d'utilisation avec scénarios associés et règles de gestion.
- Illustration de certains cas d'utilisation par des diagrammes d'activités
- éventuels diagrammes de séquences (avec acteurs , ihm et services métiers)

#### Spécifications des entités et services du domaine de l'application:

- Glossaire (dictionnaire des données)
- Diagramme de packages (avec secteurs/domaines métiers)
- Au moins un diagramme de classes par package avec :
  - entités , associations (avec rôles , navigabilités et multiplicités)
  - services métiers et dépendances
- éventuels diagrammes d'états (pour cycle de vie d'une entité métier importante)

#### Spécifications de l'IHM:

- Diagramme d'états montrant les principaux états de l'IHM avec transitions = navigations entre écrans/pages/vues ==> vue d'ensemble sur l'organisation de l'IHM , sur les navigations.

- Maquette HTML (look et contenu des écrans + navigations hypertextes)
- Eventuelle structure composite et générique des écrans (ex: sous page d'entête avec menu , ...) - digramme de classes

### **Spécifications techniques:**

- contraintes techniques (volume , temps de réponse , charge , ....)
- niveau de sécurité souhaité (authentification , ....)
- contraintes transactionnelles , ...
- éléments imposés de l'architecture (type de serveur , réseau , existant , ....)

## **1.3. Format conseillé pour les modèles et la documentation**

### **Dans l'idéal :**

- Documentation en grande partie (re-)générée automatiquement et dans plusieurs formats (html , pdf , ...) à partir des éléments du modèle UML de référence (lui même stocké dans un référentiel avec une gestion de version) et de quelques éléments annexes (descriptions textuelles , glossaires , tableaux récapitulatifs).
- Documentation au format HTML ou pdf publiée sur un site web d'un intranet et ainsi mise à disposition de tous les participants du projet.

### **En pratique:**

- Aucune norme sur le sujet (==> beaucoup de variantes selon outils)
- Documentation générée avec les moyens du bord (selon outils et scripts à disposition)
- Question clef : dépendre ou pas d'outils spécialisés (avec pérennités souhaitées)

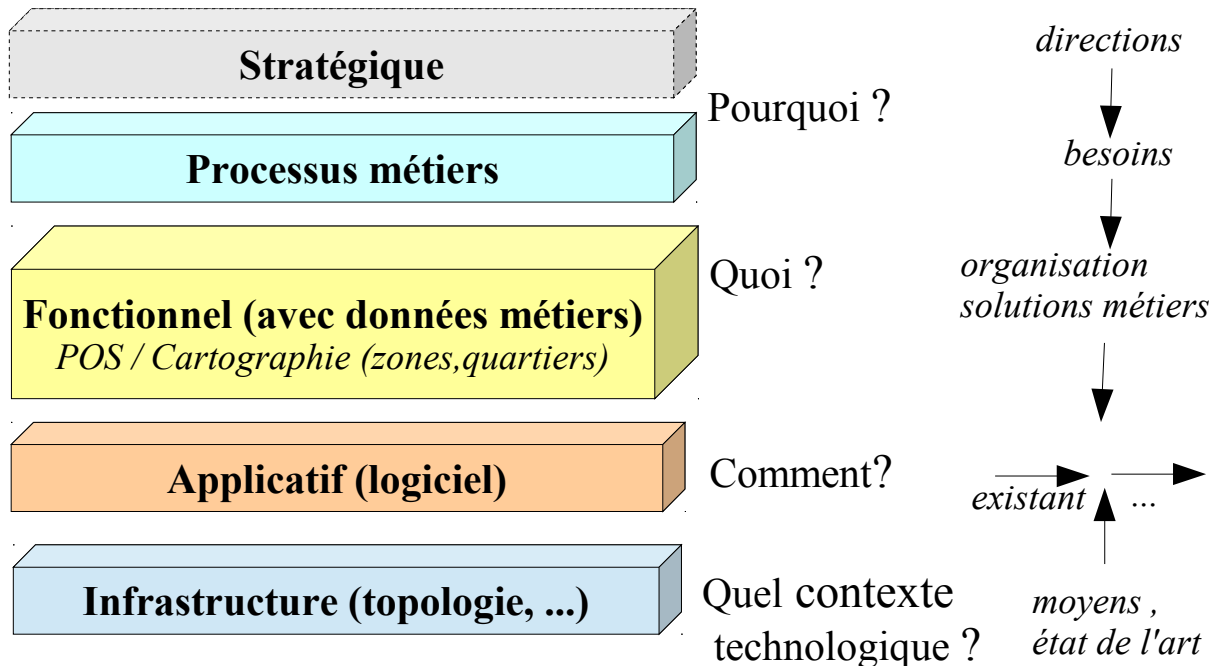
### **écueils classiques (choses à éviter):**

- Documentation générée lentement et manuellement (par copier/coller de diagrammes exportés sous forme d'images ".png" , ".gif"ou ".svg" ) et qui devient rapidement décalée par rapport aux nouvelles versions (doc qui n'est plus à jour) .
- Documentation uniquement sous forme de "diagrammes imprimés" et sans référentiel (risque d'aboutir à une "documentation de placard")

## IX - Modélisation métier (business modeling)

### 1. Rôle et contenu de la modélisation métier

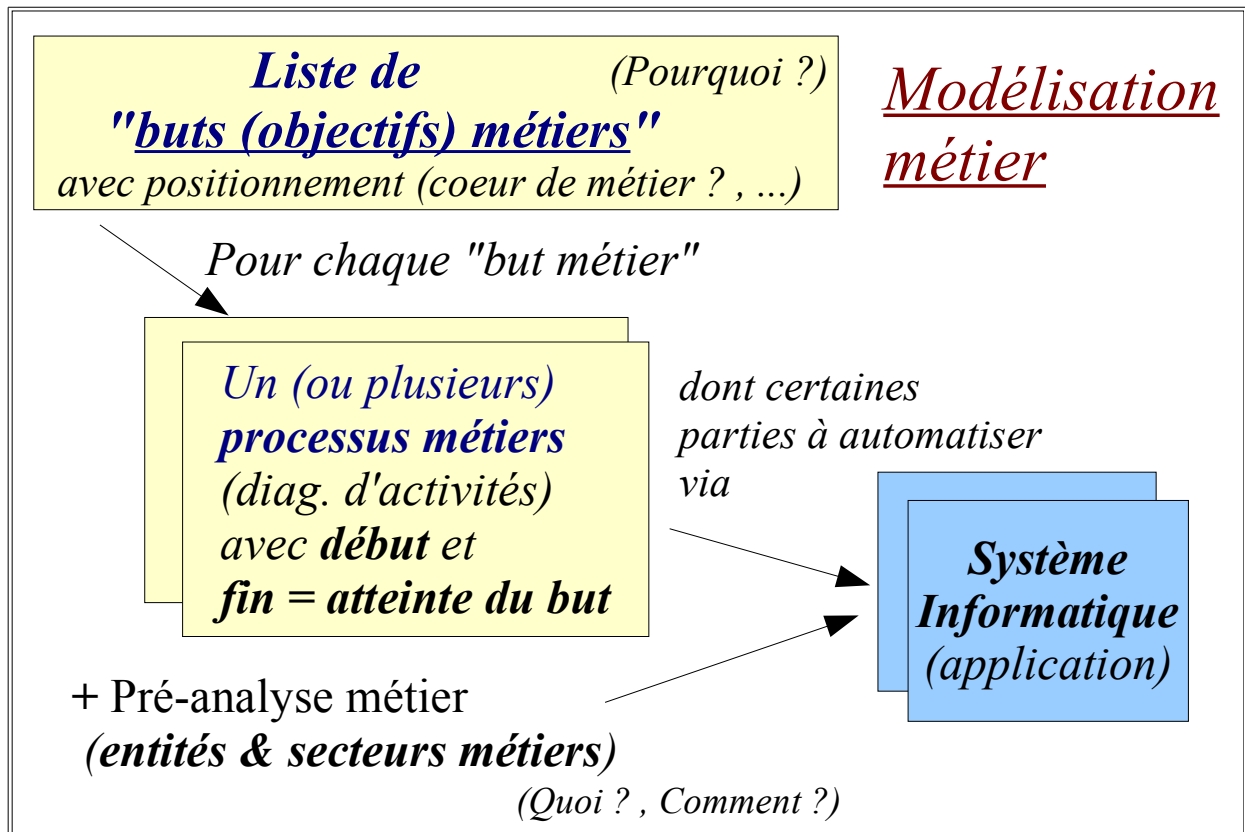
#### Cadre général (à 5 niveaux) pour l'urbanisation



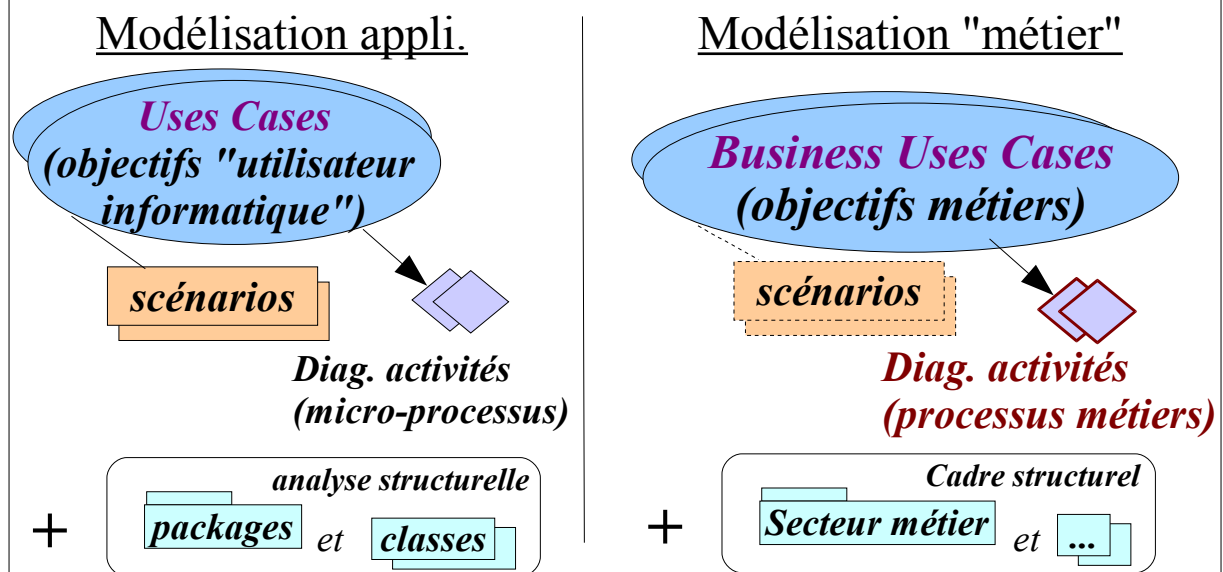
### Modélisation métier (business modeling)

- Expression du "**pourquoi ?**" (*quelle(s) utilité(s) ? , quels **objectifs** ? , ...*)
- **Contexte très large** (entreprise + partenaires , ...) *dépassant les frontières d'un seul système informatique*
- **Segmentation (découpage/regroupement)**  
--> **secteurs métiers** , packages métiers , ...
- **Processus métiers** (*diagrammes d'activités , ....*) *avec activités informatisées ou non .*





**Modélisation métier avec UML** à voir comme une transposition "métier" d'une *modélisation d'application informatique (cas classique en UML)*



## 2. Modélisation du contexte d'utilisation

Avant même de s'intéresser aux fonctionnalités d'un système à développer, il est très souvent nécessaire d'**identifier précisément ce système et de le situer dans son futur contexte d'utilisation**. Le terme anglais souvent utilisé pour désigner cette phase de modélisation du contexte d'utilisation est "*business modeling*" (*modélisation métier*).

Ceci permet de bien:

- comprendre l'*utilité* du système à modéliser/développer
- ***montrer ce qu'il y a autour*** (autre(s) système(s), catégorie(s) d'utilisateurs, ...)
- ***cadre le système*** (quelles grandes responsabilités/fonctionnalités, périmètre applicatif)

Les documents (diagrammes, ...) résultants de cette phase de modélisation du contexte permettront:

- de réfléchir (au niveau de la maîtrise d'ouvrage) sur les affectations/répartitions de certaines fonctionnalités du S.I. vis à vis d'un ensemble de systèmes coopératifs.
- de ***donner un éclairage sur les objectifs du système*** (utilité métier, raison d'être du logiciel ...) aux informaticiens de la maîtrise d'oeuvre.

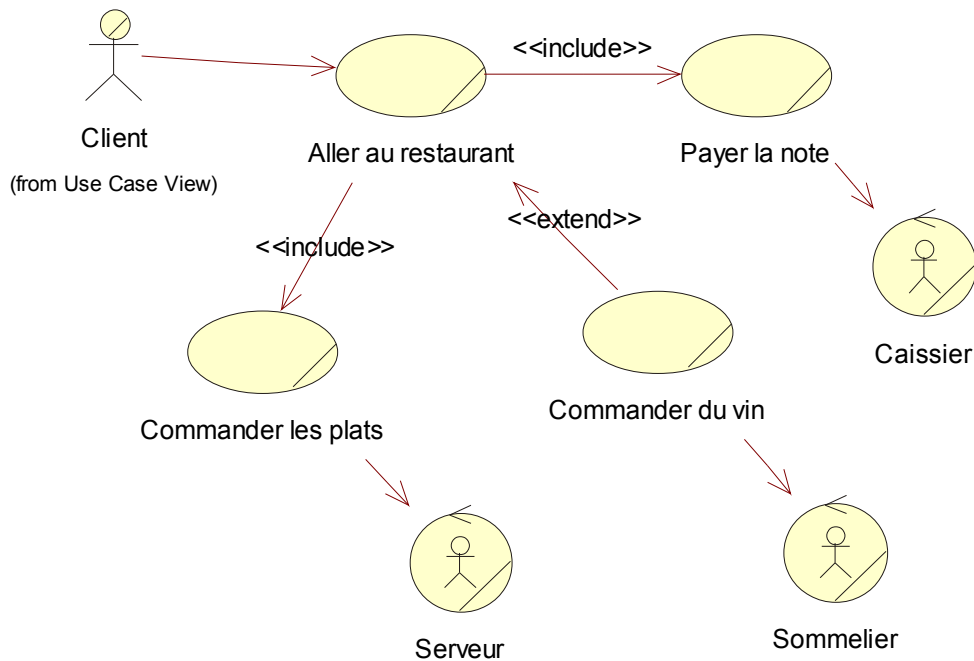
Remarque importante:

Les activités qui apparaissent sur un diagramme d'activité ne seront pas toutes systématiquement informatisées.

Un diagramme d'activité (modélisant dans les grandes lignes un processus métier) est donc souvent plutôt à considérer comme une "vue d'ensemble" sur un contexte d'utilisation plutôt que comme un modèle précis permettant de décrire un système informatique précis.

C'est finalement pour ***bien montrer à quoi le système informatique va servir*** que ***le diagramme d'activité "processus métier"*** est utilisé dans la ***toute première phase de la modélisation UML***.

### 3. Diagramme des cas d'utilisations métiers



Un diagramme de "*business uses cases*" ou "*cas d'utilisations métiers*" est une extension pour UML (provenant de RUP) et qui vise à **montrer les fonctionnalités d'un service ou département d'une l'entreprise** plutôt que les fonctionnalités d'un système informatique précis.

En plus de la notion d'acteur UML (implicitement externe), le stéréotype **<<worker>>** (ici associé aux caissier, sommelier et serveur) désigne **une personne interne (ex: employé)**.

# X - Expr. Besoins IHM (maquettes, navigations)

## 1. Expression des besoins "IHM" (interface graphique)

**IHM** = Interface Homme Machine

**GUI** = Graphical User Interface

### 1.1. Maquette

Bien que non UML , une **maquette HTML** (*look des écrans + navigations hypertextes*) est très utile pour:

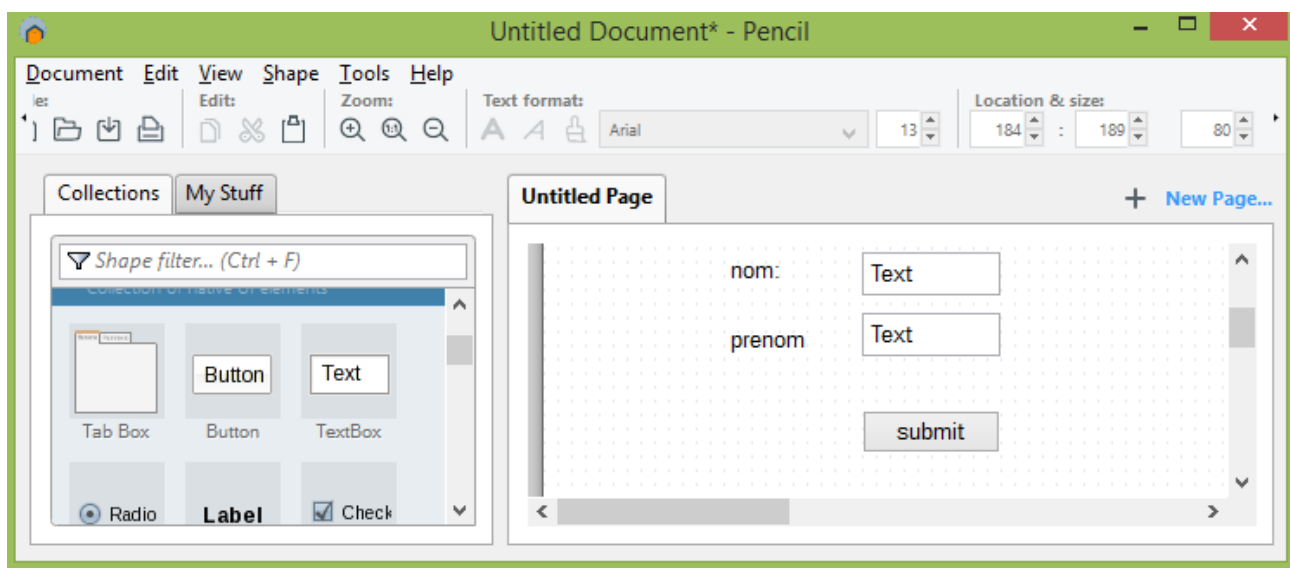
- dialoguer avec le commanditaire du système (maîtrise d'ouvrage) et les utilisateurs
- valider les fonctionnalités attendues et leurs enchaînements.
- valider les grandes lignes de la charte graphique
- penser à des détails que l'on aurait oubliés

Cette maquette indispensable peut éventuellement être complétée par des diagrammes UML (d'activités ou d'états/transitions) .

Parmi quelques programmes utilitaires permettant de dessiner rapidement une maquette , on peut citer :

- [Pencil Project](#)
- [Lumzy](#)
- [MockFlow](#)
- [Cacoo](#)
- [Balsamiq](#)
- [Mockingbird](#)
- [iPlotz](#)

... (liste loin d'être exhaustive)



## 1.2. Modélisation des enchaînements d'écrans (navigations)

Une liste de dessins d'écrans (parties de la maquette) c'est déjà très bien mais il peut manquer une vue d'ensemble sur :

- ce qui existe
- les enchaînements prévus (autorisés ou pas , ...)
- ....
- 

Un diagramme annexe illustrant des navigations entre pages "web" ou écrans est alors très utile. On peut utiliser plusieurs formalismes pour schématiser des navigations entre parties de l'IHM. Rien n'est normalisé sur ce point.

Au sein des diagrammes UML existants , c'est le diagramme d'états (StateChart) qui est le plus approprié pour modéliser des navigations .

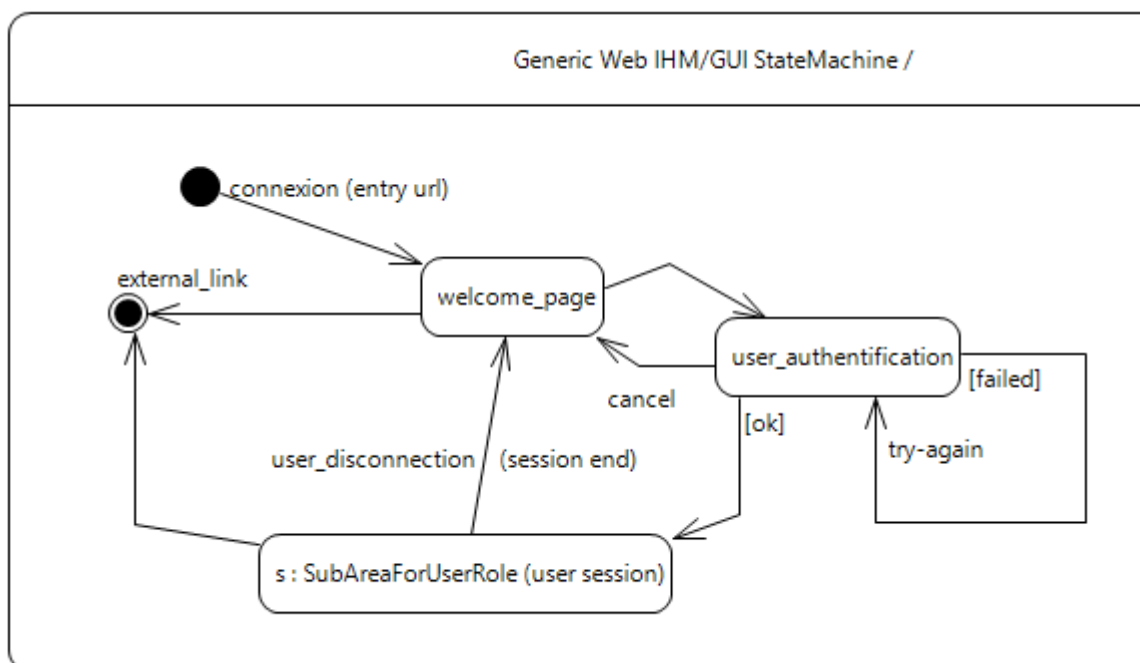
L'IHM passe d'un état à un autre lors d'un changement de page ou d'écran.

Une transition (changement d'état) est associée à un événement (ex : clic sur lien hypertexte) et est quelquefois conditionné.

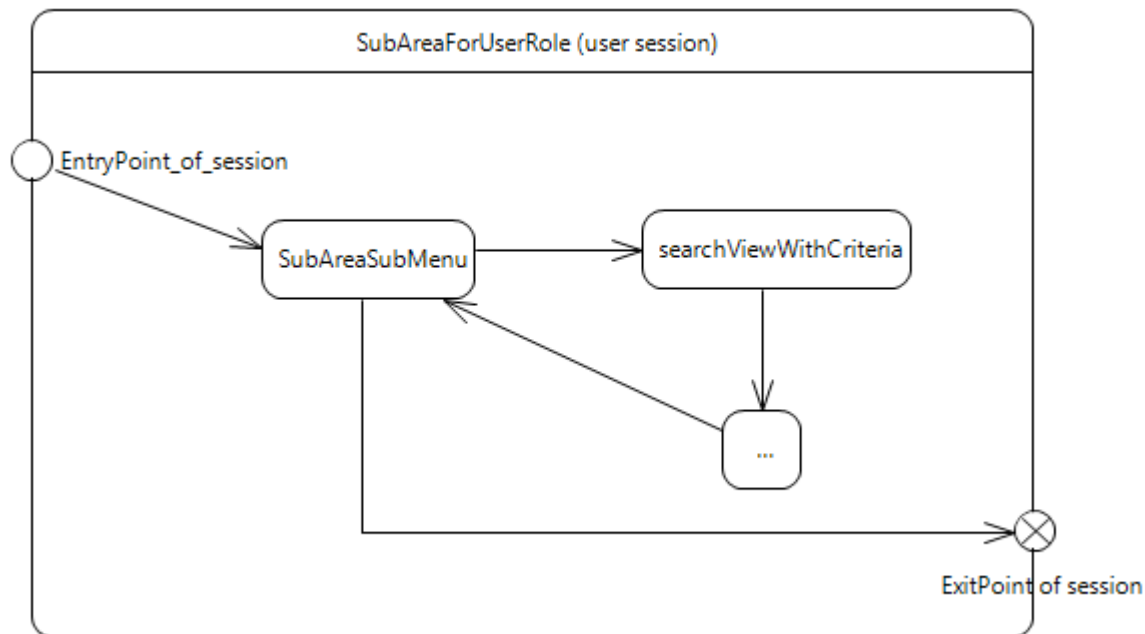
Remarque : Les diagrammes d'états seront étudiés d'une façon approfondie dans un chapitre ultérieur (analyse des cycles de vie). Des Tps pourront également être effectués au niveau de cette phase.

Effectuer un Tp sur une modélisation UML de l'IHM n'est pas un objectif prioritaire. Il vaut mieux affecter aux chapitres ultérieurs le temps précieux accordé aux Tps .

Exemple générique (à spécialiser au cas par cas) :



Le sous état "s : SubAreaForUserRole" se décompose à son tour en un autre diagramme :



### Pertinence (à relativiser en fonction du contexte) :

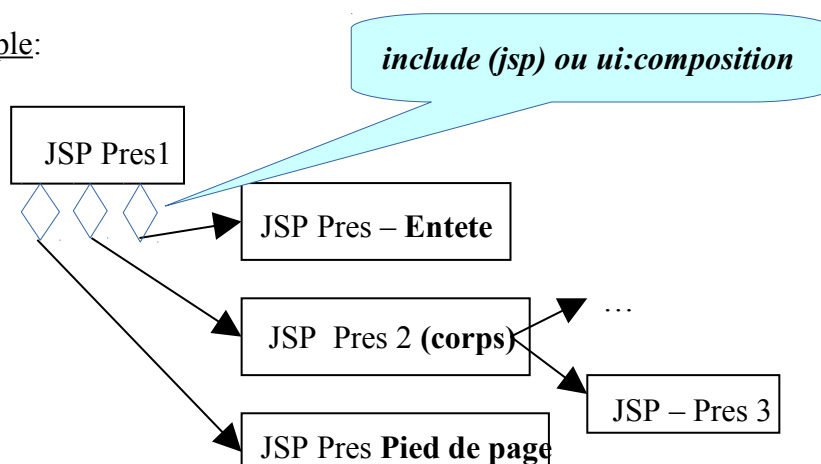
La modélisation des enchaînements d'écran est très importante dans le cas d'une IHM Web (sous forme de pages html générées dynamiquement par code java ou autre) .

Certains frameworks WEB comportent un fichier de configuration qui centralise toutes les navigations possibles (ex: *struts-config.xml* ou *faces-config.xml* dans le monde java).

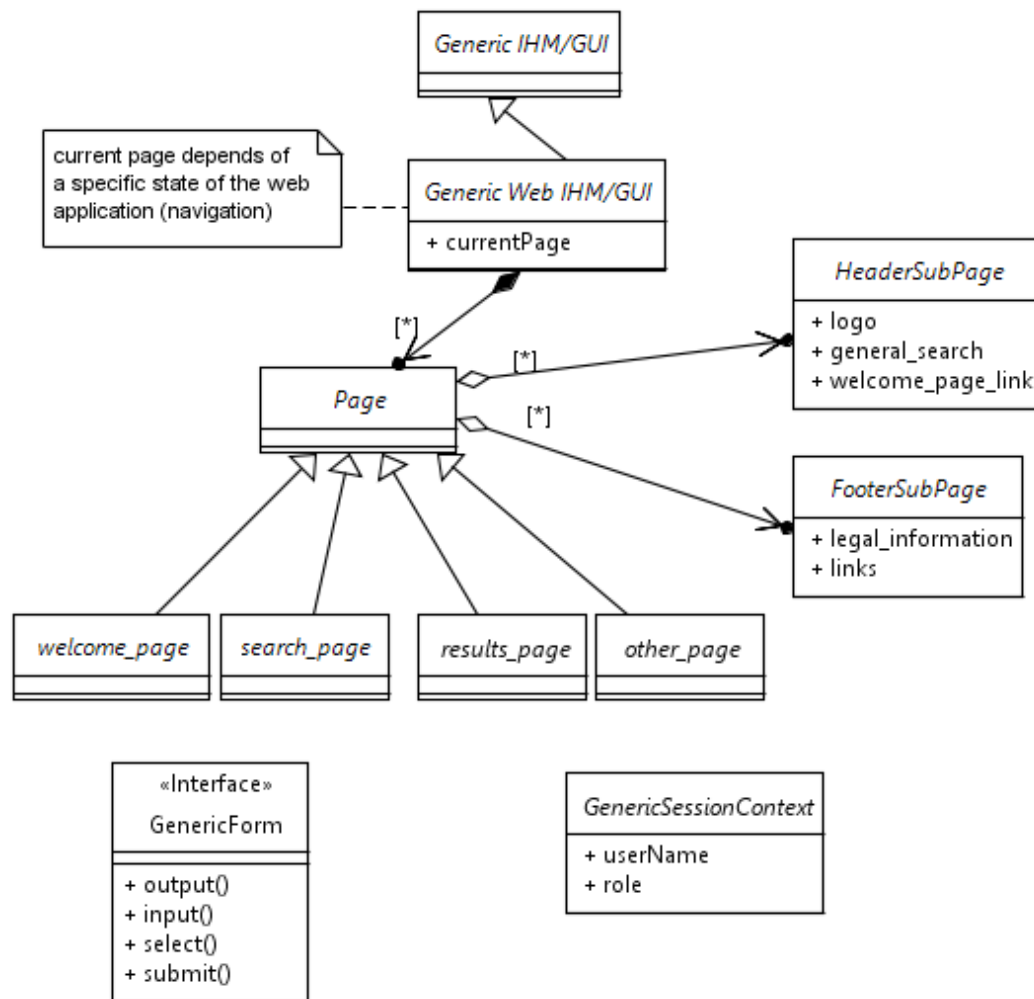
## 1.3. Modélisation composite et générique des écrans (modèles)

La plupart des interfaces graphiques mises en place aujourd'hui sont généralement assez sophistiquées et elles sont généralement structurées sur un modèle composite (incorporation de sous pages "entête" , "...." ).

exemple:



Un petit schéma (UML ou non) peut quelquefois être assez utile pour décrire la structuration générique attendue au niveau des écrans.



### Conséquences :

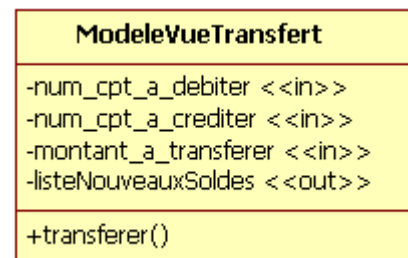
- Modélisation souhaitable du contenu des entêtes , menus et pieds de pages.
- La partie visible des enchaînements d'écrans est assez souvent concentrée sur la partie "corps" (les "menu" , "entête" et "pied de page" changent rarement).

## 1.4. Modélisation abstraite des écrans (contenu / fonctions)

On peut *éventuellement* définir des *modélisations abstraites des principales Vues de l'IHM*.

==> **Ceci n'est utile que dans le cadre d'une génération automatique de code.**

Exemple (à exprimer dans un diagramme de classes UML) :



Les stéréotypes `<<in>>` et `<<out>>` permettent d'exprimer ce qui sera saisi/sélectionné ou bien affiché.

# XI - Fondamentaux orientés "objet"

## 1. Concepts objets fondamentaux

<b>Classe</b>	Type d'objet . Tous les objets d'une même classe ont la même structure (attributs / données internes) et le même comportement (méthodes / fonctions membres)
<b>Instance</b>	Objet (exemplaire) créé à partir d'une certaine classe (via new en java)
<b>Héritage</b>	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]
<b>Polymorphisme</b>	Plusieurs variantes possibles pour une même opération générique , sélection automatique en fonction du type exact de l'objet mis en jeu.
<b>Interface</b>	Classe complètement abstraite (sans code mais avec prototypes de fonctions) permettant de préciser un contrat fonctionnel .

--> approfondir si besoin en cours avec le formateur.

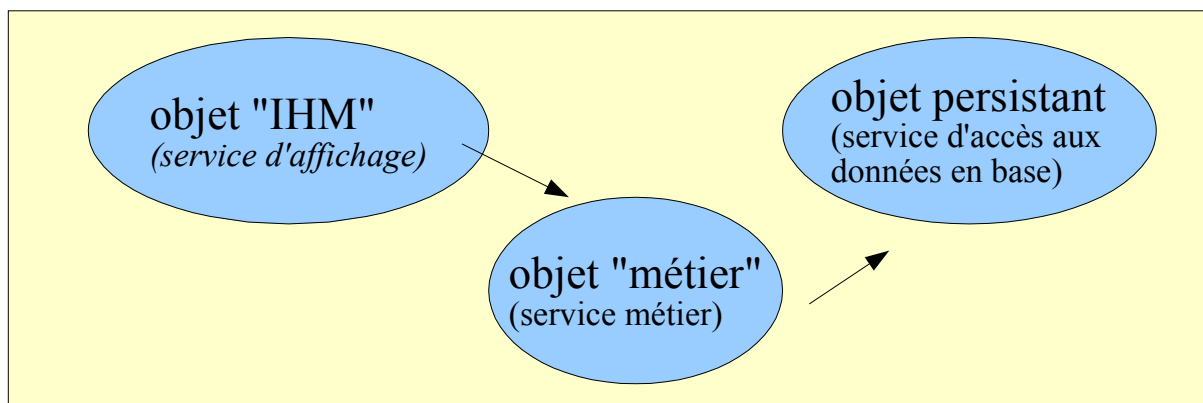
--> approfondir si besoin ensuite via Wikipédia ou "comment ça marche" ou autre.

*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.**



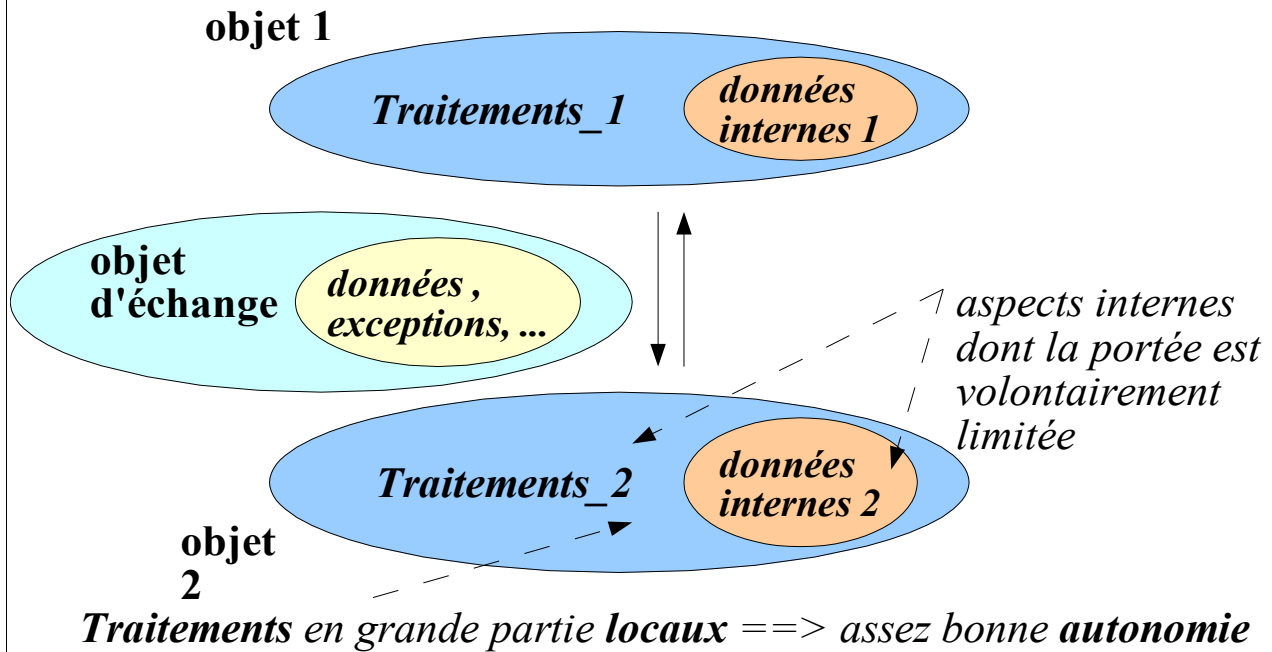


## Grand traits de l'approche objet:

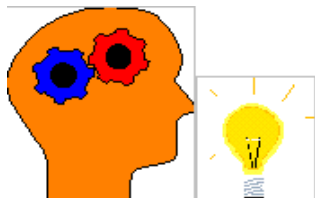
Un programme est un ensemble de petites **entités** . Chacune ayant :

- 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).



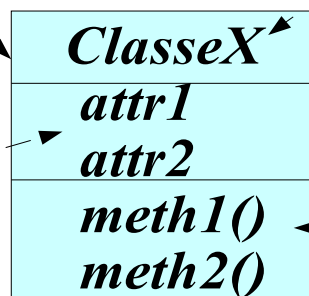
## Abstraction des données (réalité ==> types abstraits)



*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 .*

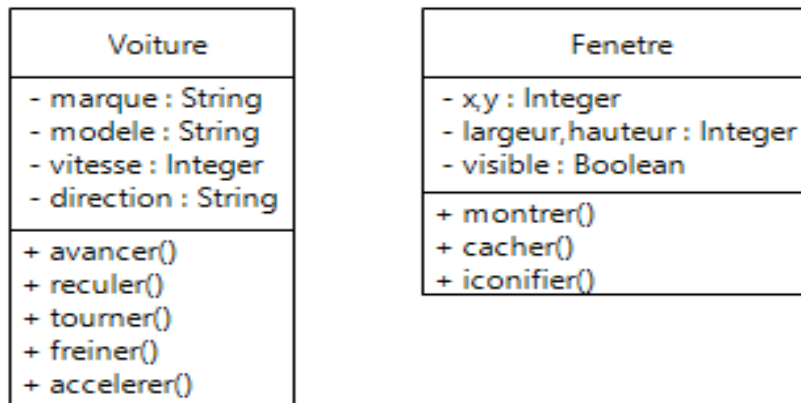
instances

## 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*]



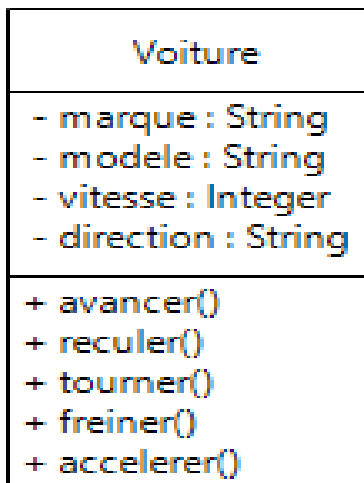
**Toutes les instances d'une même classe ont une même structure commune et ont un même comportement.**

## 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.

*Classe*



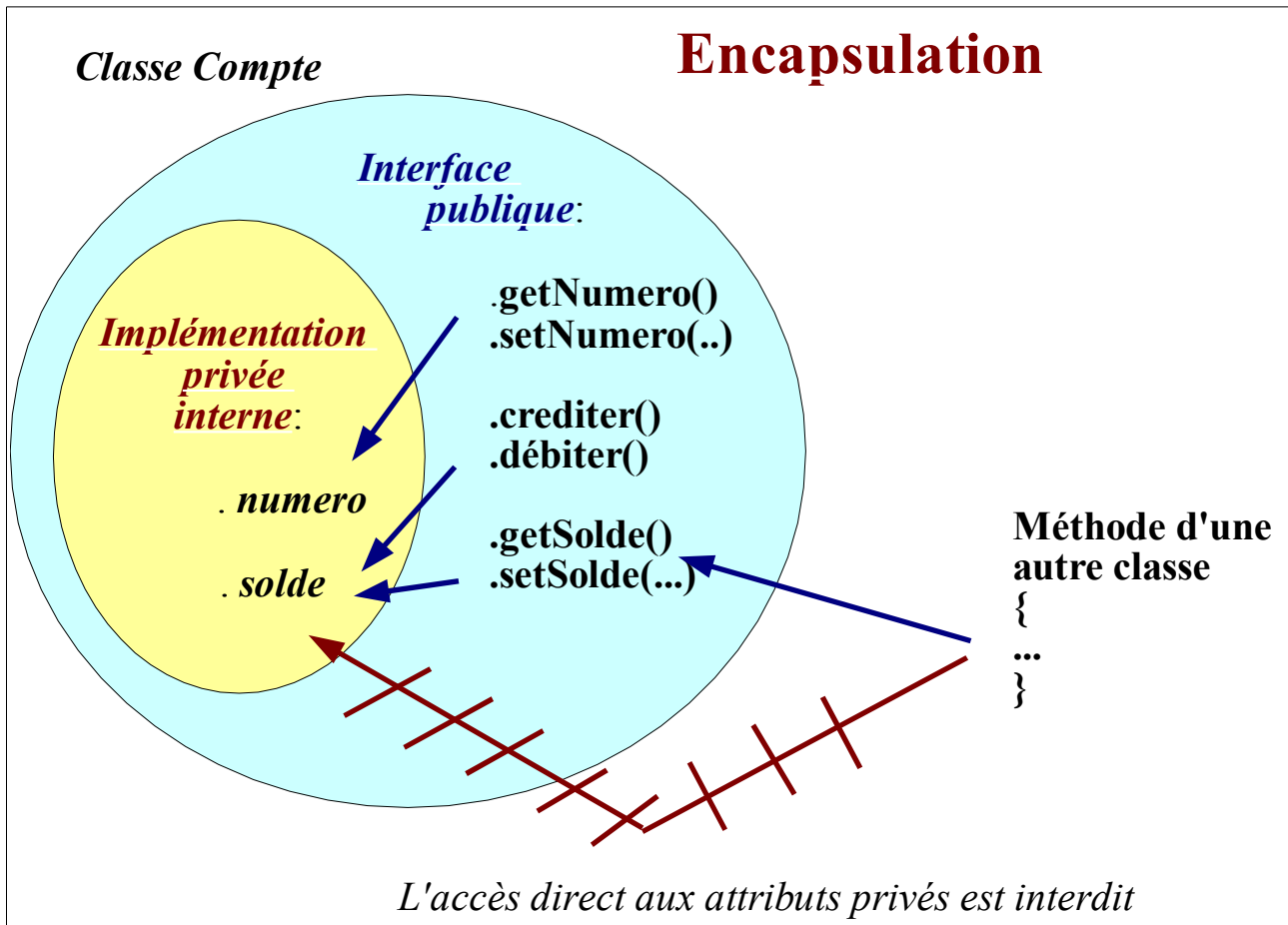
*instance 1*

V\_AZ45BV456:Voiture  
marque=Renault  
modèle=Clio  
vitesse=50 , direction=nord

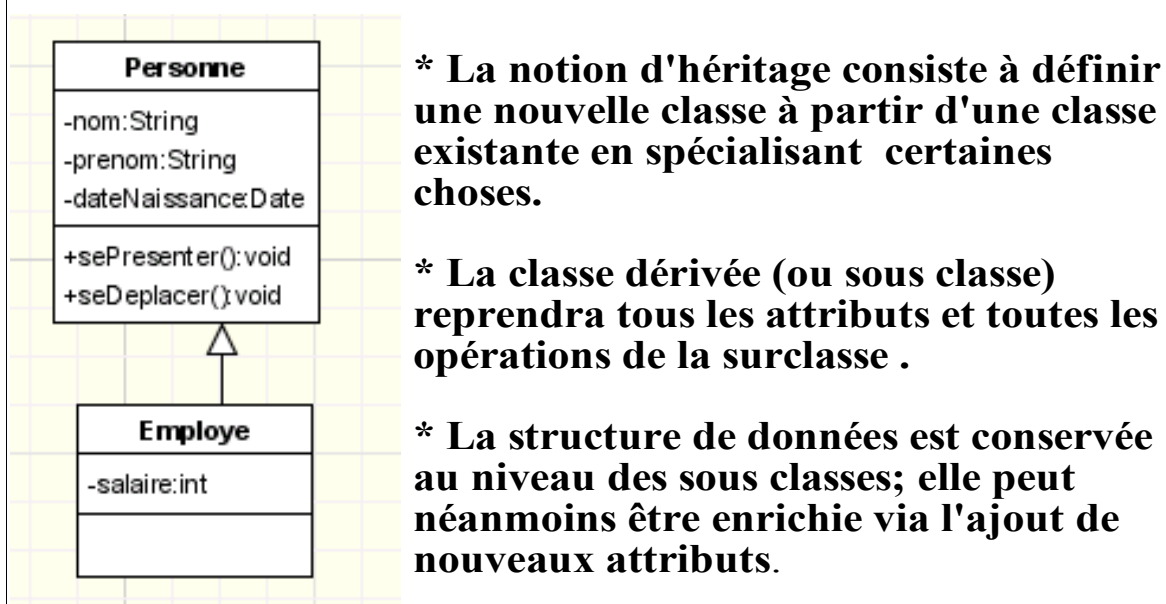
...

*instance N*

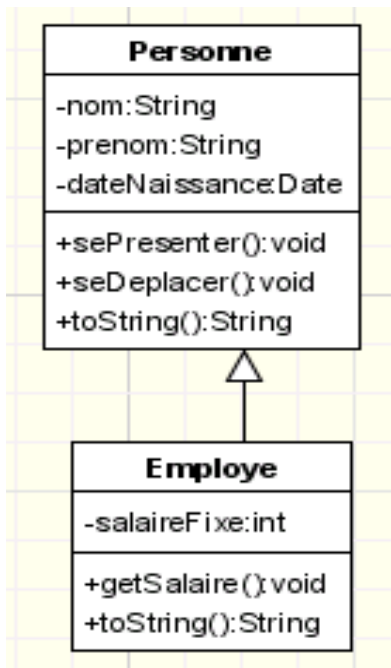
V\_BN78BV936:Voiture  
marque=Peugeot  
modèle=308  
vitesse=90 , direction=sud



## Héritage / Généralisation / Spécialisation



## 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() { ... }
}
  
```

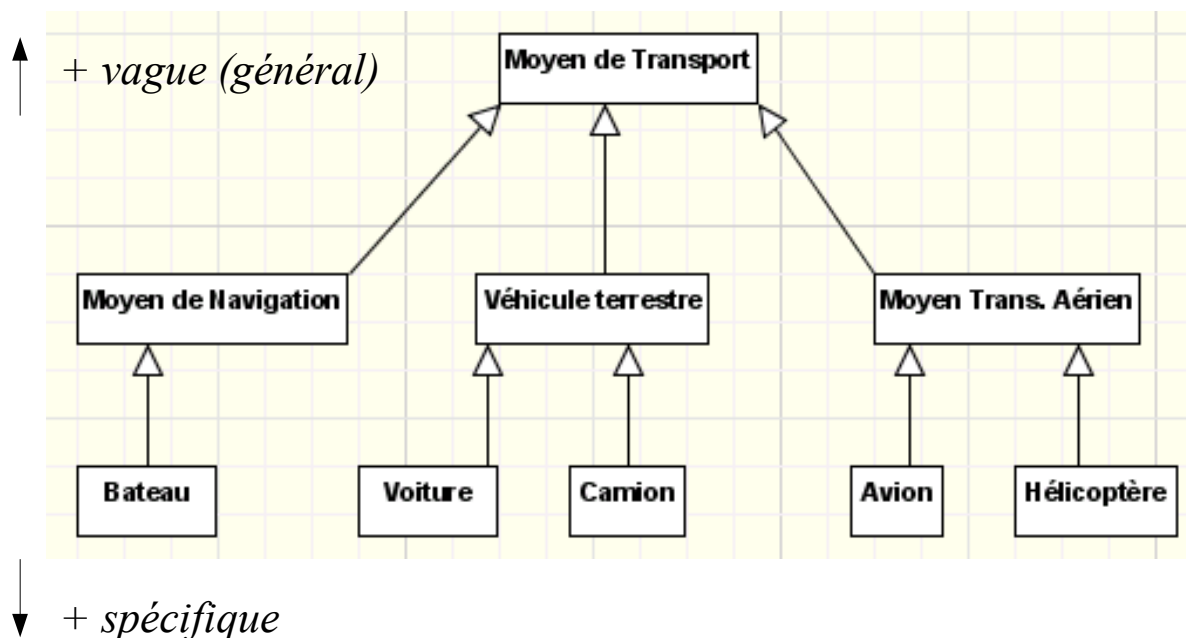
```

public class Employe extends Personne {
    private int salaireFixe;

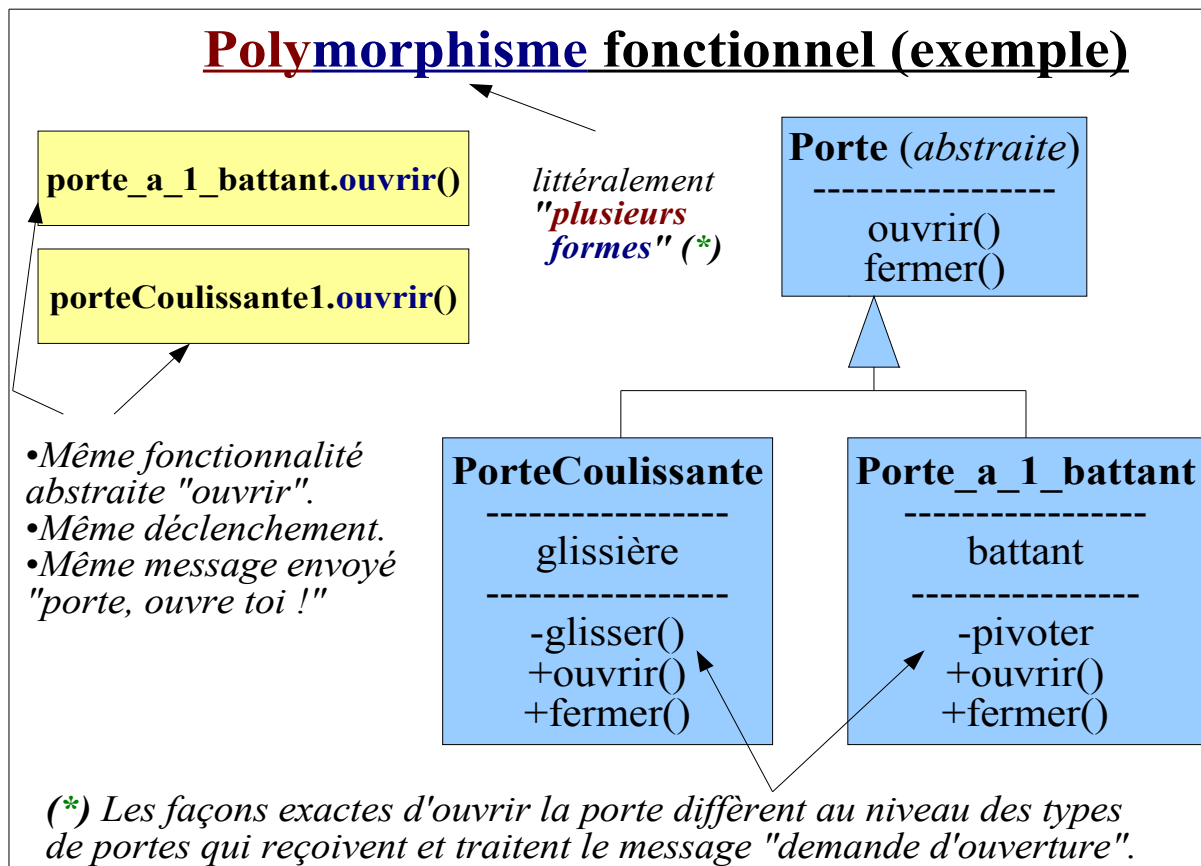
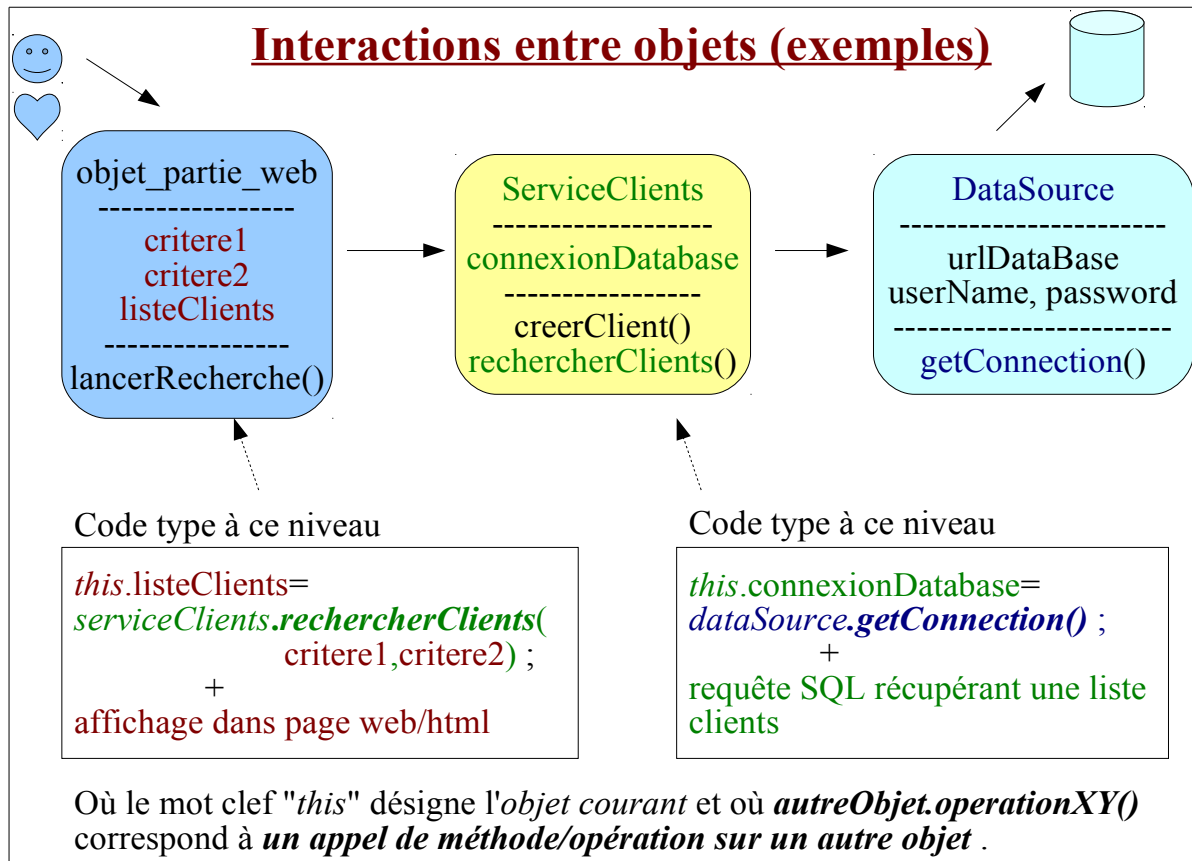
    public void getSalaire() {...}
    public String toString() {...}
}
  
```

*pers1.sePresenter() ; empl.sePresenter(); empl.getSalaire();*

## Arbre (ou graphe) d'héritage



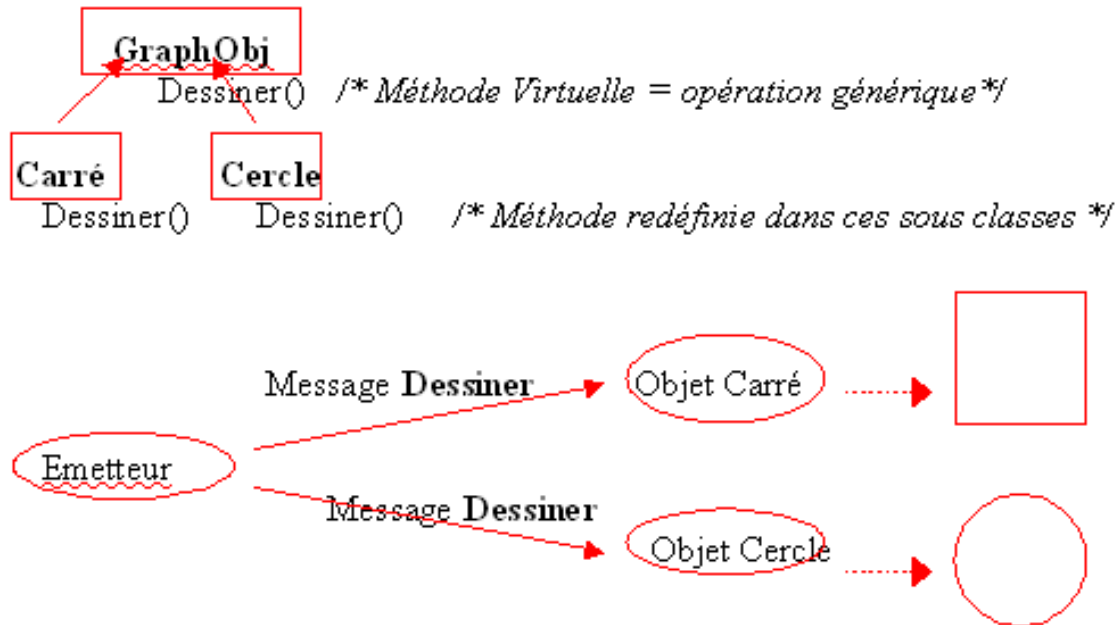
Une relation d'héritage est représentée par une grosse flèche triangulaire allant de la sous-classe vers la sur-classe et signifiant  
**"is kind of / est une sorte de"**



**Polymorphisme** 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.

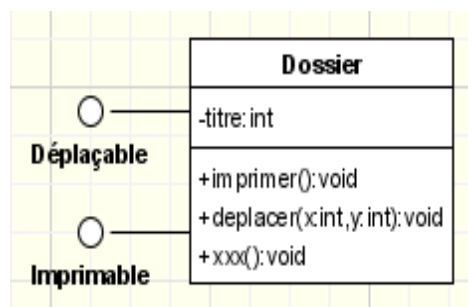
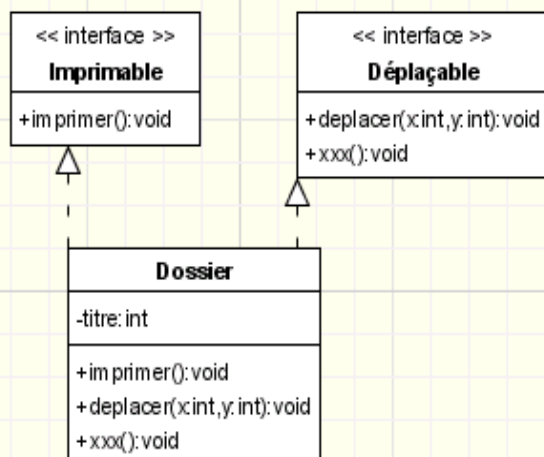


## Interface

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.

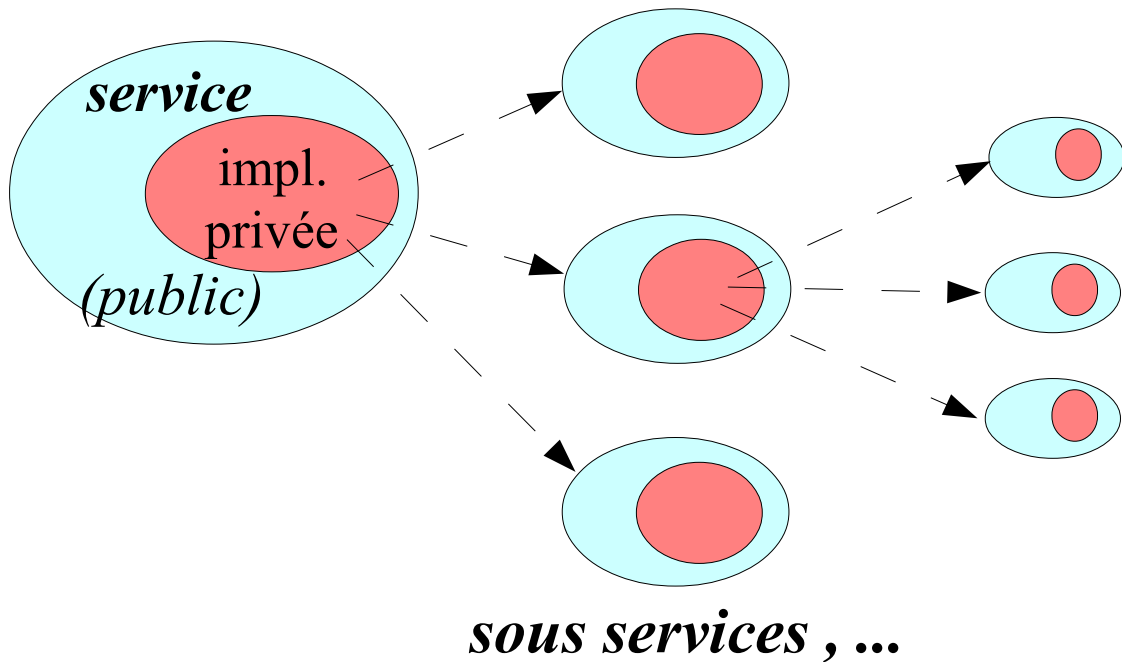
Pour être utile, une interface doit évidemment être entièrement codée au niveau d'une classe concrète.

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



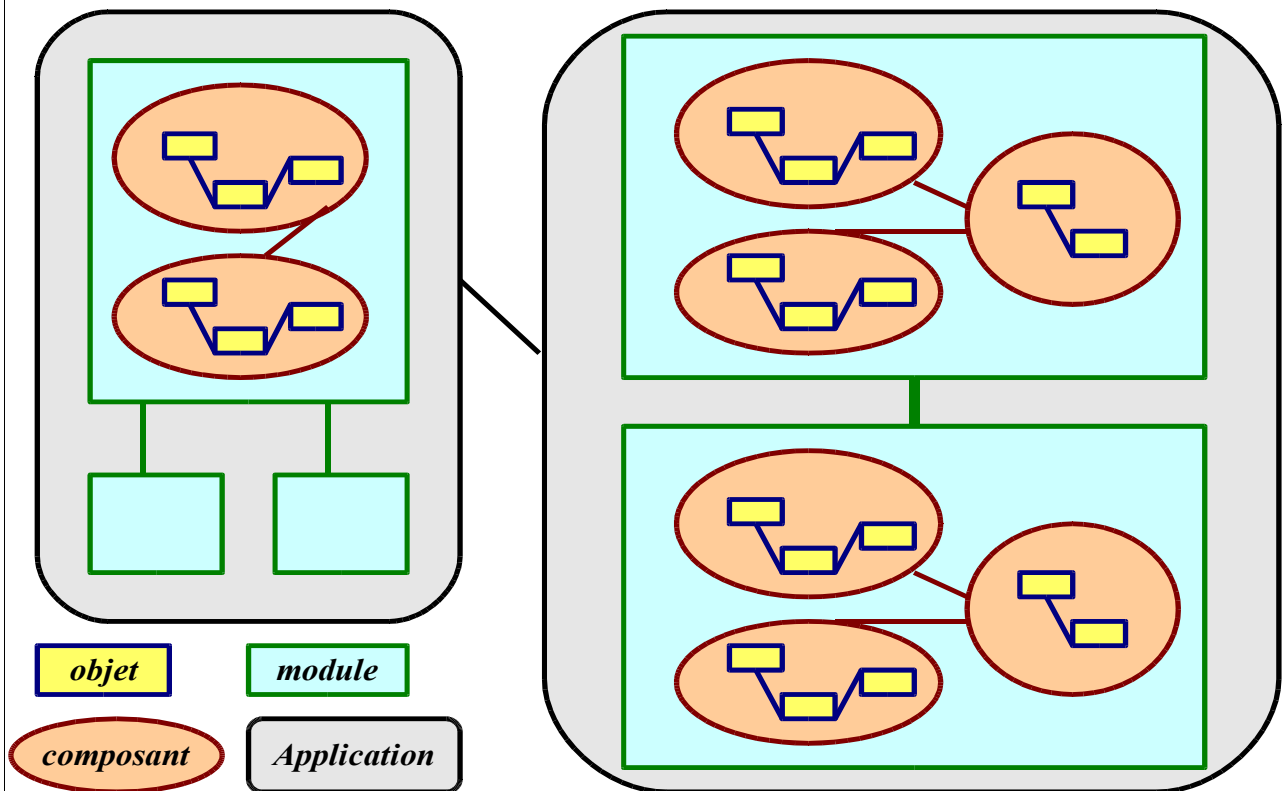
## 2. Granularité

### Encapsulation & granularités



Modules , composants , objets

**granularités** (*objets, composants, modules, ...*)



### 3. Modularité

#### 3.1. Forte cohésion et couplage faible

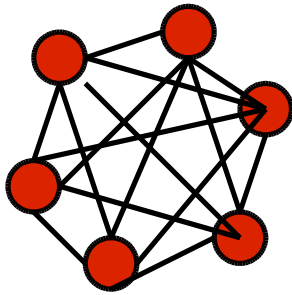
La **cohésion** mesure le degré de connectivité existant entre les éléments d'une classe unique. Une classe modélisant un aéroport aura une bien meilleure cohésion qu'une classe modélisant le roi d'une nation, le roi du jeu d'échec et le roi d'un jeu de carte . ***Il faut privilégier la cohésion fonctionnelle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

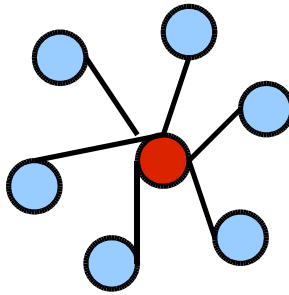
#### Couplage entre objets (idéalement faible)

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

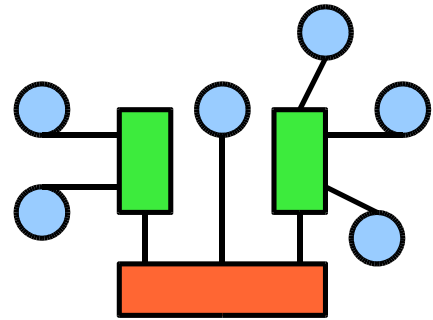
Un *assemblage d'objets complémentaires* rend globalement des *services plus sophistiqués*. Cependant la complexité des liens entre les éléments peut éventuellement mener à un édifice précaire:



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



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

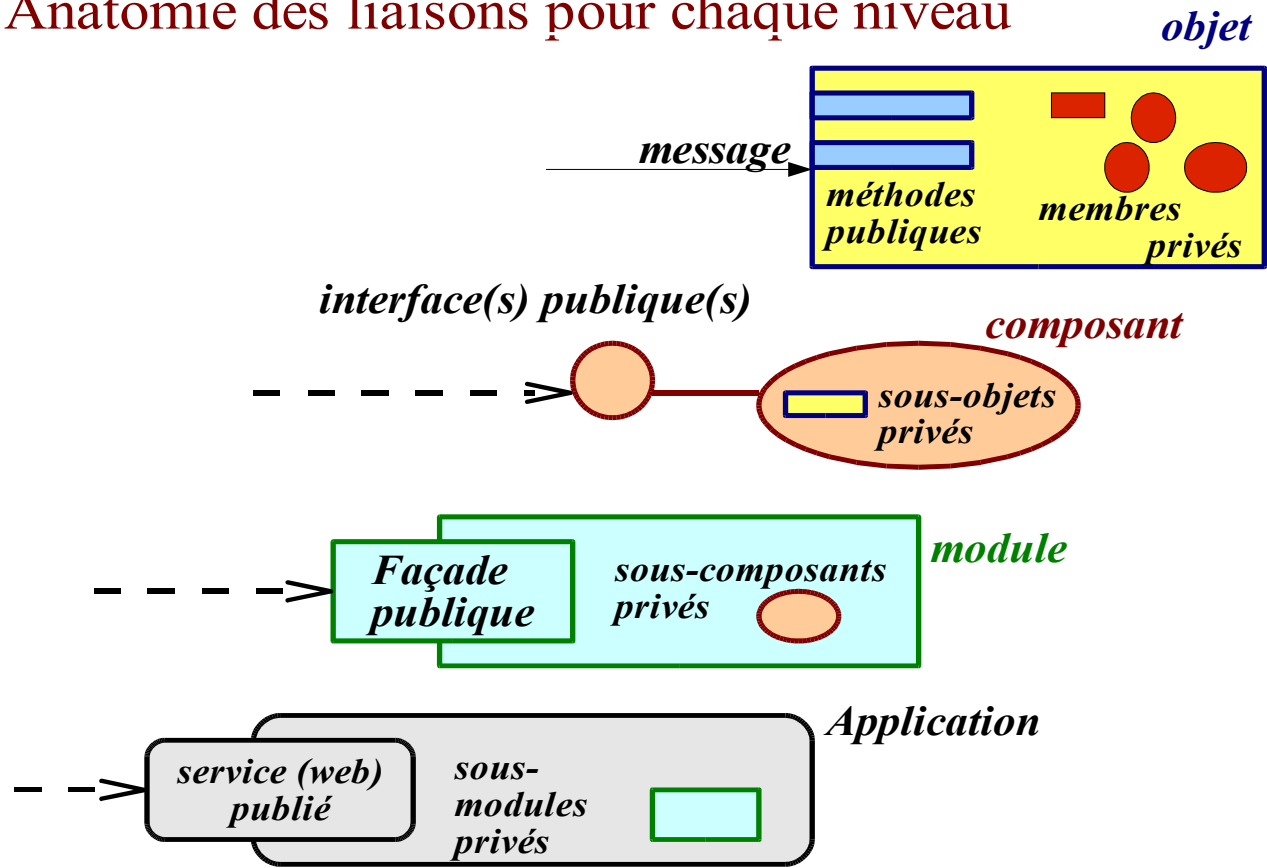


***couplage faible***  
 et ***décentralisé***  
 ==> simple ,  
 relativement flexible  
 et plus robuste

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



## Anatomie des liaisons pour chaque niveau



## XII - Analyse du domaine (entités) / diag. Classes

### 1. 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.

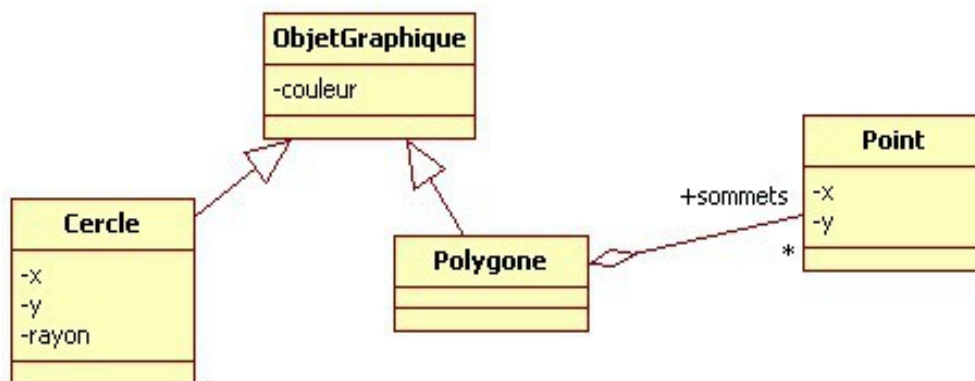
Partant d'un niveau très conceptuel (arbre sémantique, glossaire) elle permet d'aboutir à un **modèle logique de type "entités/rerelations"** résolument restreint à ce qui touche au domaine du système à développer (sans éléments inutiles) . Idéalement basées sur des formulations de type "chose concrète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

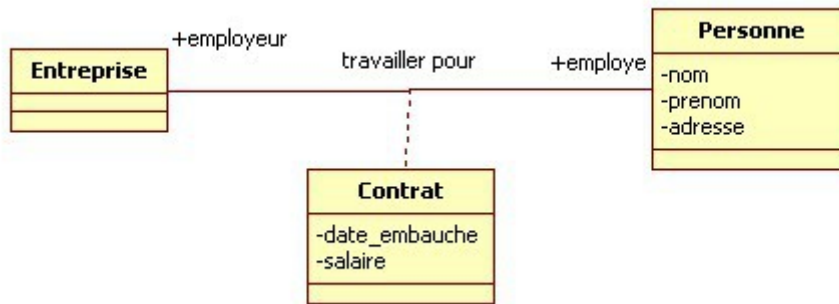
L'étape "**analyse du domaine**" est généralement effectuée de la façon suivante:

- Mise au point d'un **glossaire décrivant les entités du domaines** ==> tableau avec nom\_entité , définition , caractéristiques\_retenues .

<i>Terme/entité (classes)</i>	<i>Définition</i>	<i>Caractéristiques retenues (attributs pertinents et relations)</i>
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 le 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
  - ....

## 2. Diagramme de classes (notations , ...)

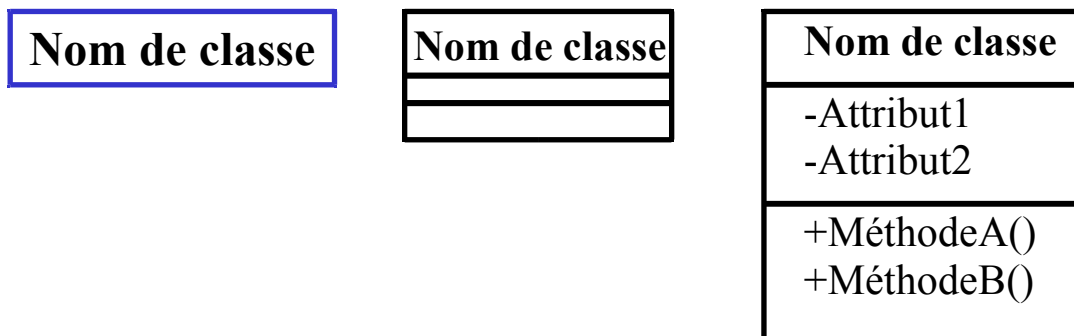
### Diagramme de classes (*modèle structurel*)

- 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*)

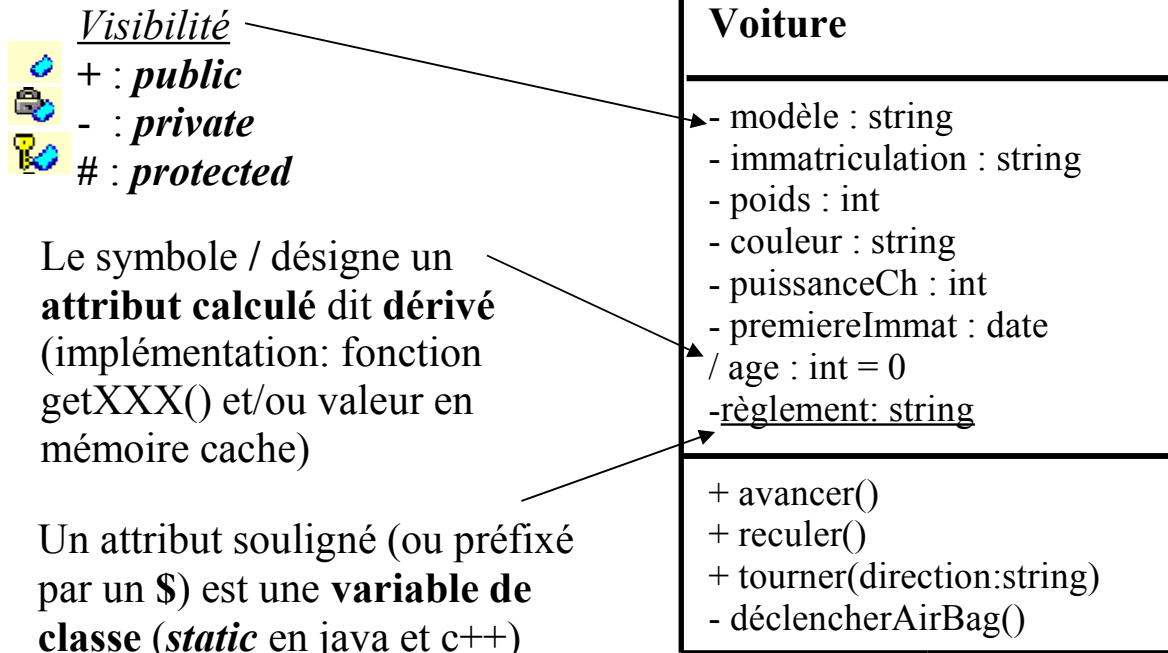


*Attribut* alias **Propriété/Property**

,

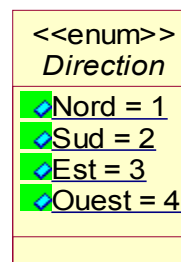
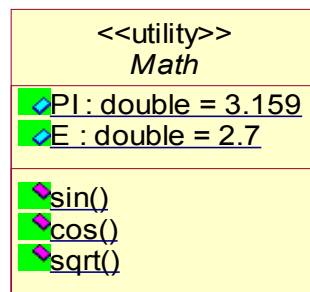
*Méthode* alias **Opération** .

## Détails sur les éléments d'une classe



## Classes spéciales (utilitaires, énumération, ...)

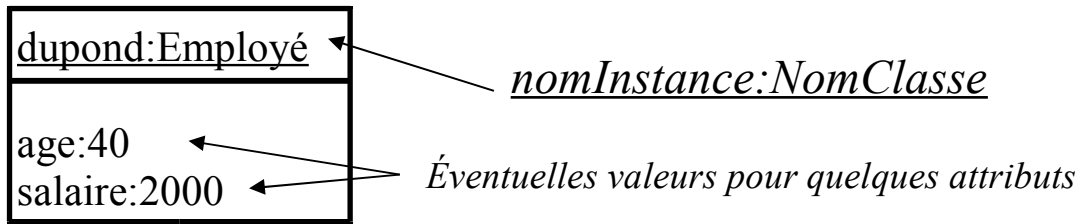
- Dans un monde matériel et rationnel ou tout est objet, il n'y a plus de place pour des fonctions globales (anarchiques).  
Celles-ci doivent être rangées dans des classes utilitaires.
- Les constantes doivent elles aussi être placées dans des classes (ou interfaces) d'énumération.



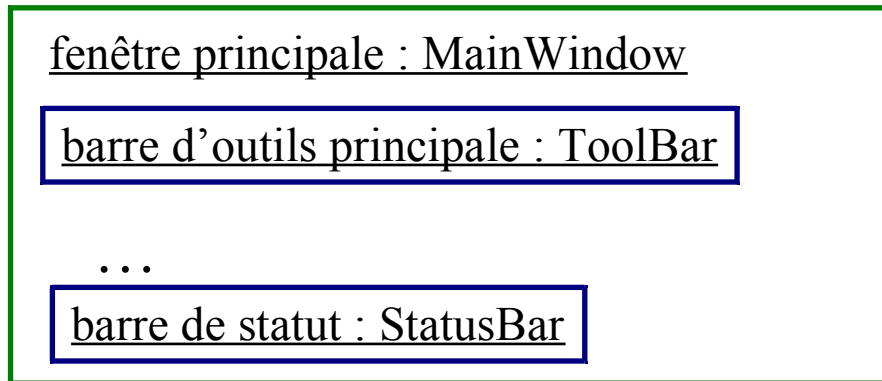
y = **Math.sqrt**(x);

obj.setDirection(**Direction.Nord**);

## Eventuels (et rares) diagrammes d'instances



*Instance composée (de sous objets):*



## 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)

<b>1</b>	exactement un
<b>0..* ou *</b>	<b>plusieurs</b> (éventuellement zéro)
<b>0..1</b>	zéro ou un
<b>n (ex: 2)</b>	exactement n (ex: 2)
<b>1..*</b>	<b>un à plusieurs</b> (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.

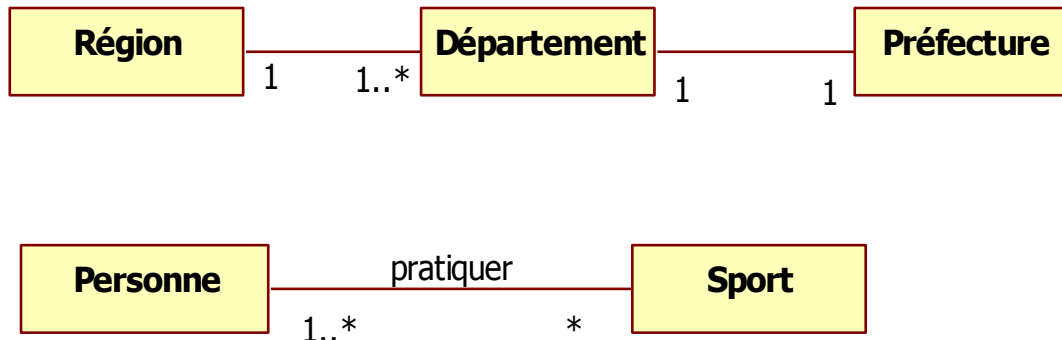
### Interprétation des multiplicités:

Livre	<b>1</b>	Comporte	<b>1..*</b>	Page
-------	----------	----------	-------------	------

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

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

## Multiplicités (exemples)

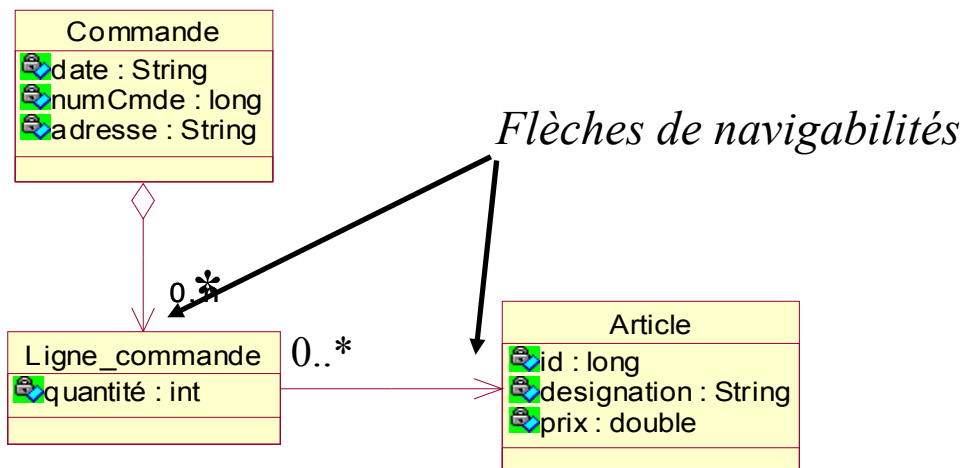


NB:

- 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**.

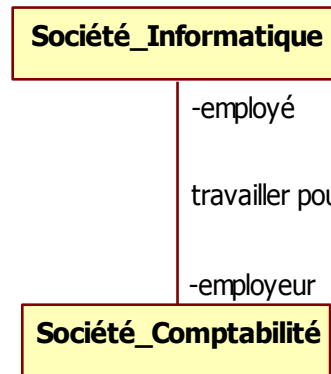


*Un article n'a pas directement accès à une ligne de commande*

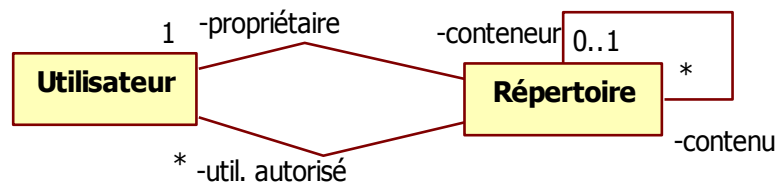


## Rôles

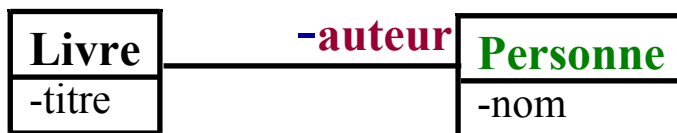
Les rôles  
(facultatifs)  
permettent  
d'indiquer le rôle  
joué par chaque  
entité dans le cadre  
d'une association.



*Ils peuvent servir  
à lever certaines  
ambiguïtés:*



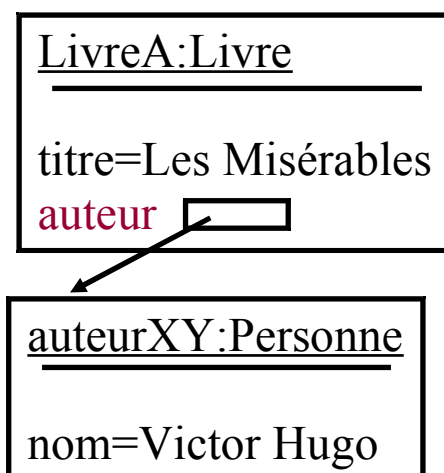
## Rôles & implémentation



```

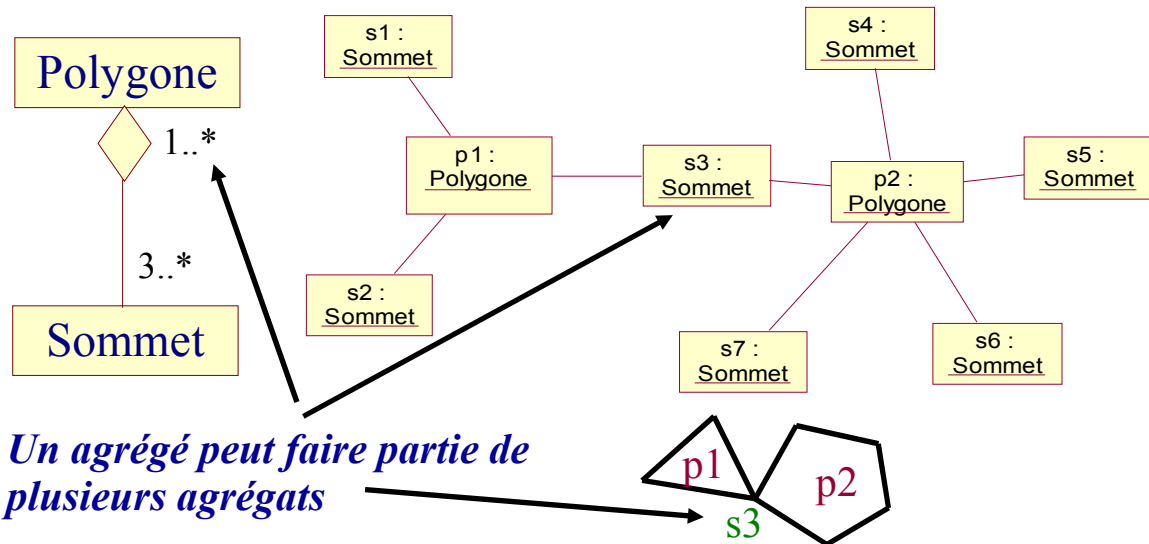
class Livre
{
    private String titre;
    private Personne auteur;
    ...
}
  
```

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



## Agrégation

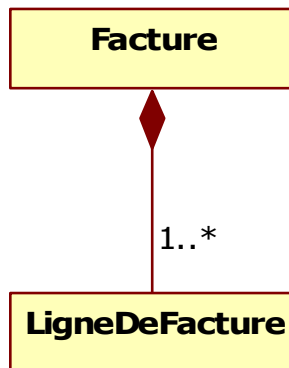
Un **agrégat** est composé de plusieurs sous objets (les agrégés). Cette relation particulière et très classique est symbolisée par un **losange** placé du coté de l'agrégat.



## 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 **référéncés** par leur(s) agrégat(s).

## Composition (agrégation forte)

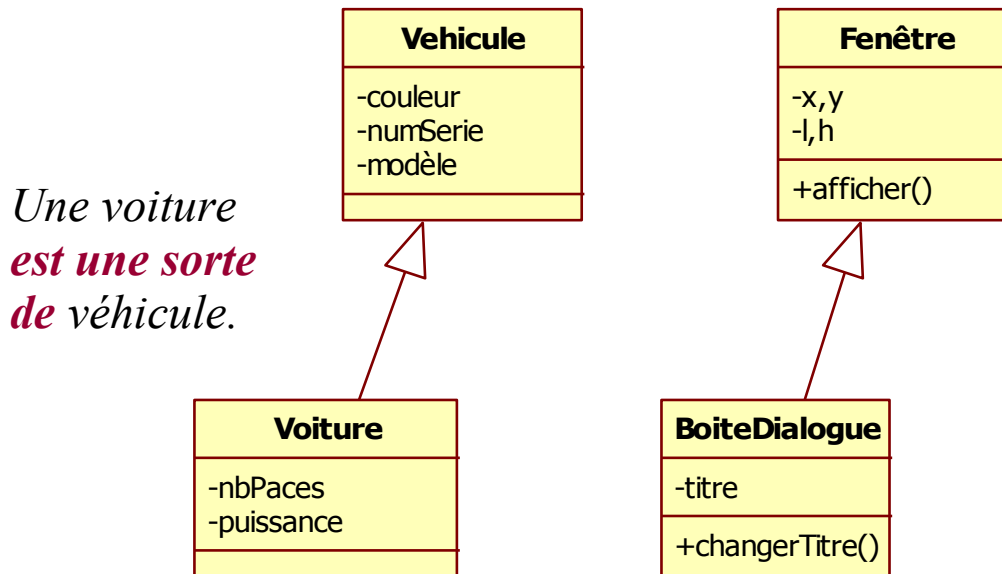


En général, le sous-objet n'existe que si le conteneur (l'agrégat) existe: **lorsque l'agrégat est détruit, les sous objets doivent également disparaître.** (*"cascade-delete" en base de données*).

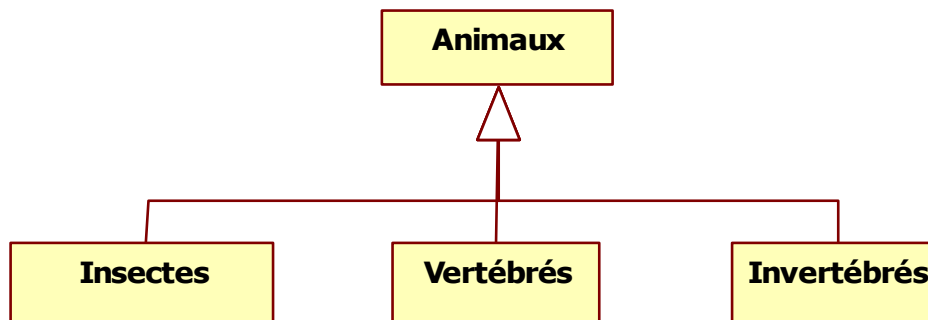
*Dans une agrégation forte, un sous objet ne peut appartenir qu'à un seul conteneur.*

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)



## Classification (généralisation)

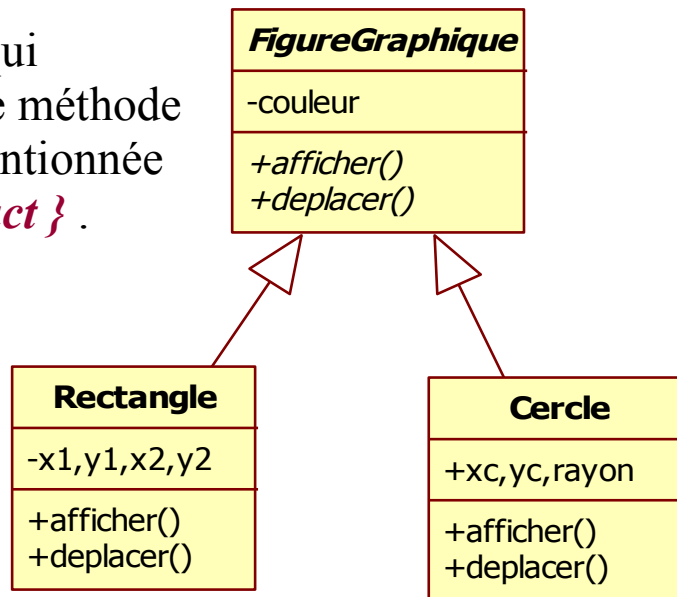


Les animaux peuvent être classées dans divers **groupes** (et sous groupes).

*Classification selon caractéristiques discriminantes.*

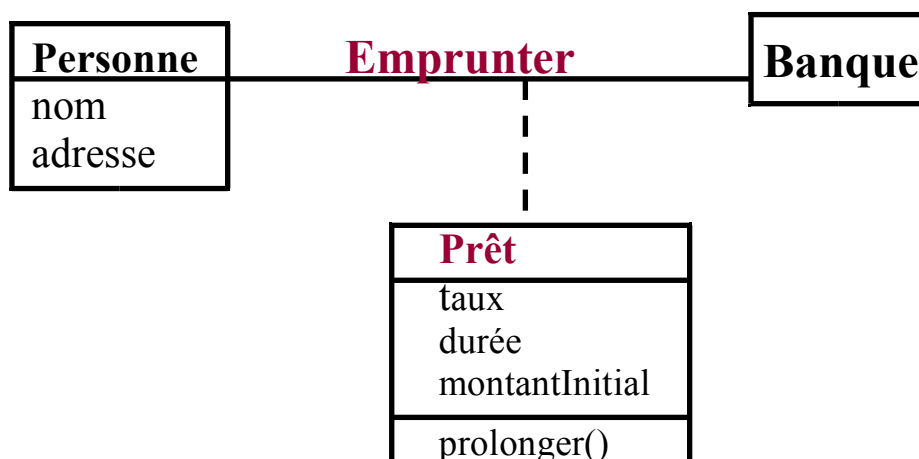
## Classes abstraites et concrètes

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

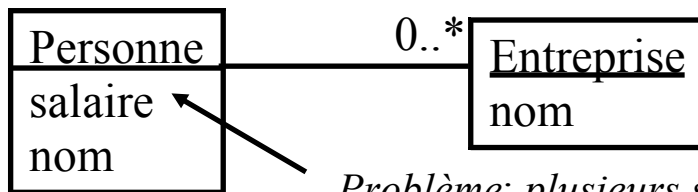


## Classes d'association

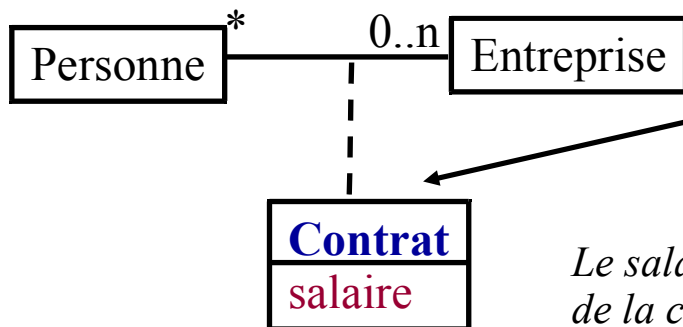
Lorsqu'une **association** comporte des données ou des **opérations** qui lui sont propre (non liés à seulement une des entités mises en relation), on a souvent recours à des **classes d'association**.



## Classes d'association (exemples)



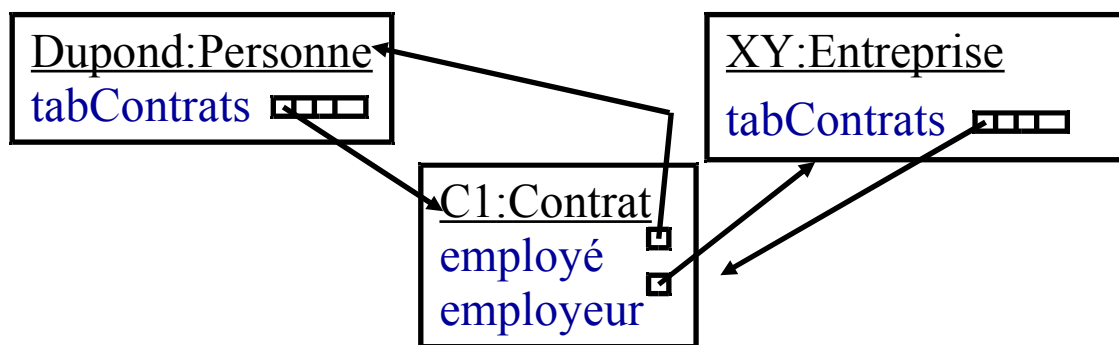
*Problème: plusieurs salaires si la personne travaille à temps partiel dans plusieurs sociétés.*



*Un objet contrat n'existera que si une association est maintenue entre une personne et une entreprise.*

*Le salaire est devenu un attribut de la classe d'association.*

## 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()**;*

Package(s) et Namespaces

NB: Si A,B et C sont 3 packages imbriqués alors "A::B::C" est le namespace associé.

### 3. Éléments structurants d'UML

<i>Regroupements UML</i>	<i>Sémantiques / caractéristiques</i>
<b>Modèle</b>	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)
<b>Package</b>	Regroupement significatif permettant de ranger ensemble les éléments qui appartiennent à un même domaine (fonctionnel et/ou technique).
<b>Diagramme</b>	Simple vue graphique représentant quelques éléments d'un modèle et leurs relations (le découpage en différents digrammes est purement pragmatique : selon la place )

#### 3.1. Modèle

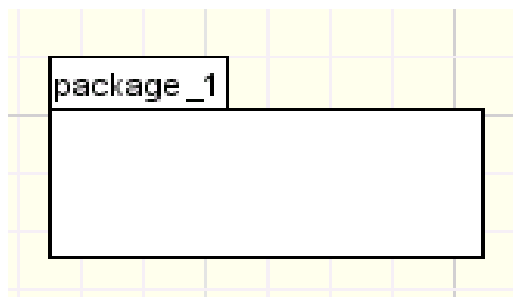
Correspondant généralement à une application entière (ou bien à un sous système) , un **modèle UML** est essentiellement constitué par une **arborescence d'éléments** .

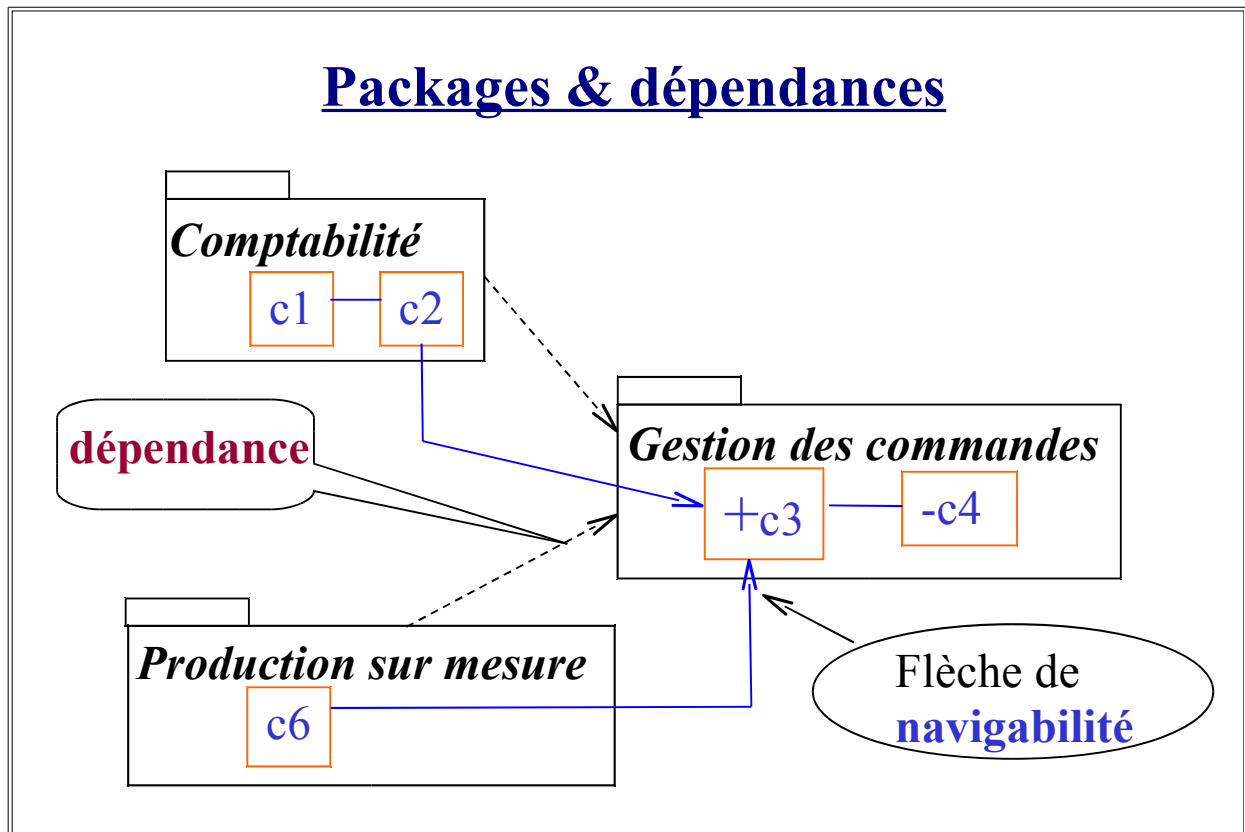
C'est à partir de son contenu (*xy.uml* ou *xy.xmi*) que l'on pourra éventuellement générer du code via MDA.

#### 3.2. 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}**.

### 3.3. 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 diagrammes.
- La plupart des outils permettent d'ajouter dans un diagramme un élément déjà existant via un simple glisser/poser partant de 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**.



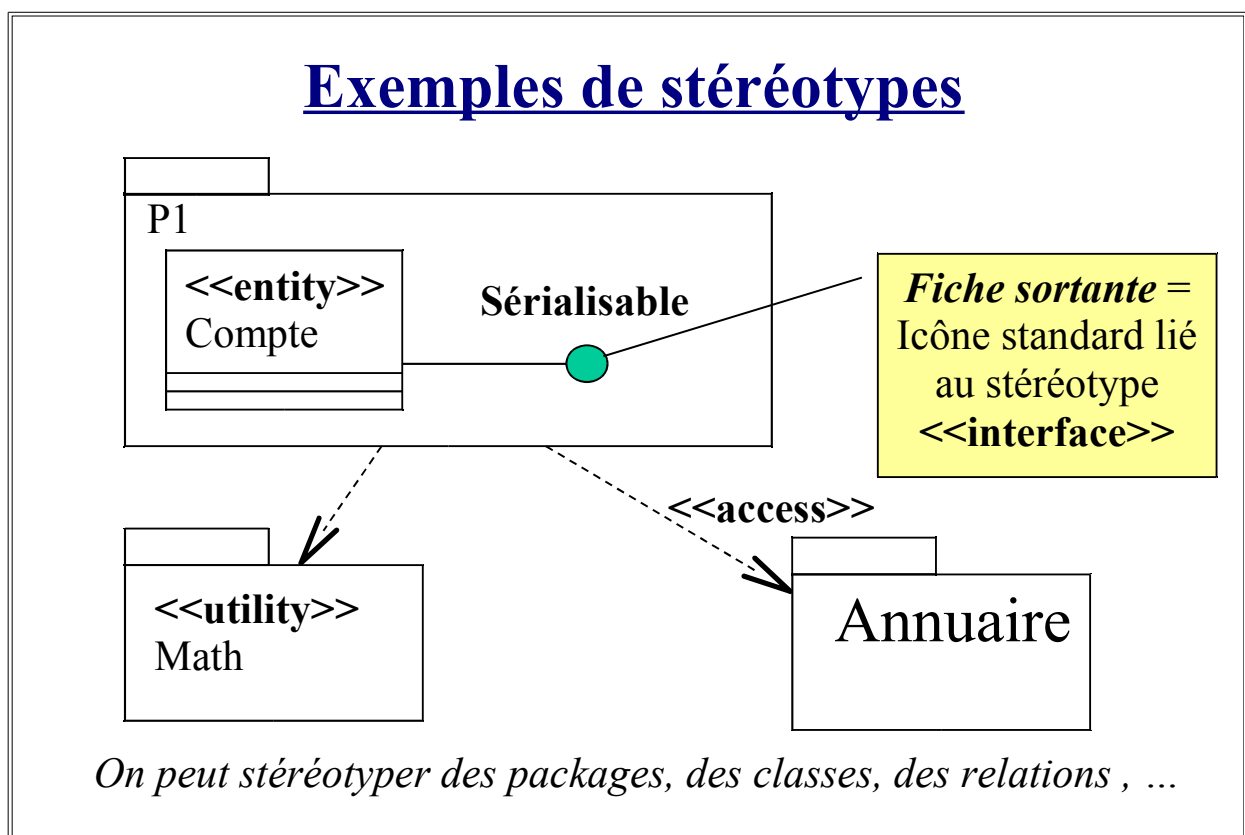
### 3.4. Stéréotypes

- 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.

### 3.5. 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.

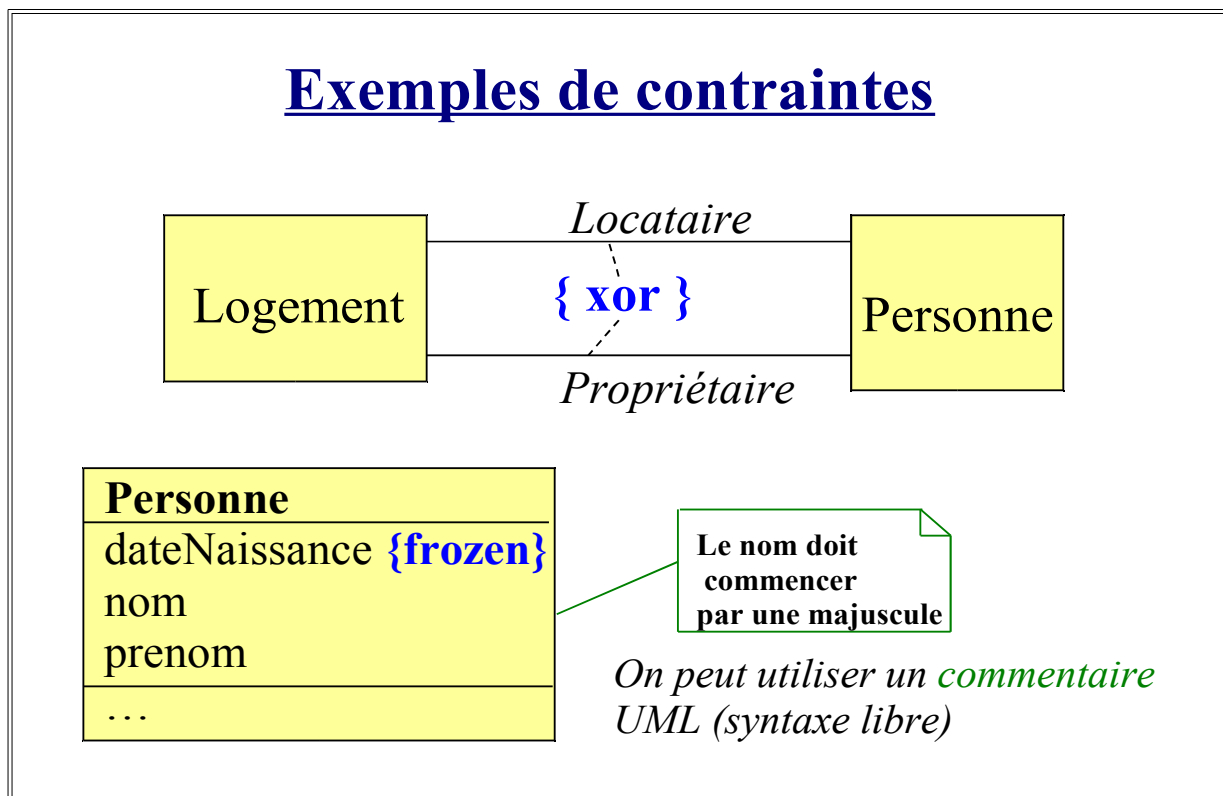
Ces valeurs cachées ne sont donc utiles que si elles sont ultérieurement analysées par un programme quelconque (générateur de documentation, générateur de code MDA, ...).

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

### 3.6. 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 }



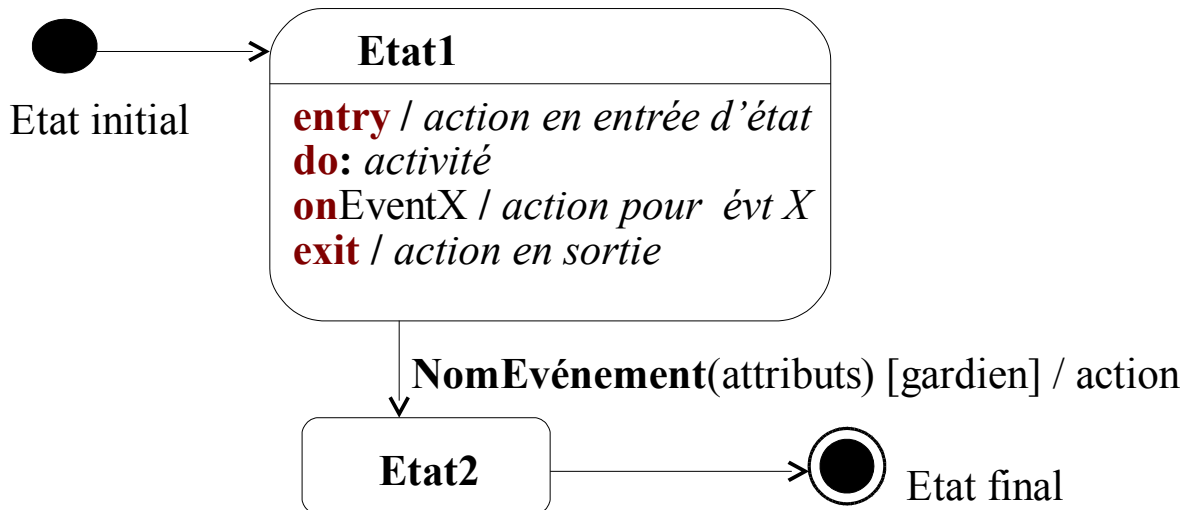
NB:

- 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).

## XIII - Diagrammes d'états (cycle de vie , ...)

### 1. Diagramme d'états et de transitions (StateChart)

#### Principales notations (graphe d'états)



NB :

- Chaque **état** est représenté par un **rectangle aux coins arrondis**.
- Un état peut comporter certains détails (activités, actions,...) et peut éventuellement être décomposé en sous états (généralement renseignés dans un autre diagramme).
- Une **activité** dure un certain temps et peut éventuellement être interrompue.
- Une **action** est quant à elle immédiate (opération instantanée, jamais interrompue).

NB : Une transition d'un état vers lui même (self transition) implique une sortie et une nouvelle entrée dans celui-ci .

NB2 : Le diagramme précédent (de l'époque UML1) montre "entry" et "exit" en tant que détails d'un état. Avec UML2, il est préconisé de placer ces détails dans un sous diagramme (avec "entryPoint" et "exitPoint") comme on le verra au sein de quelques pages.

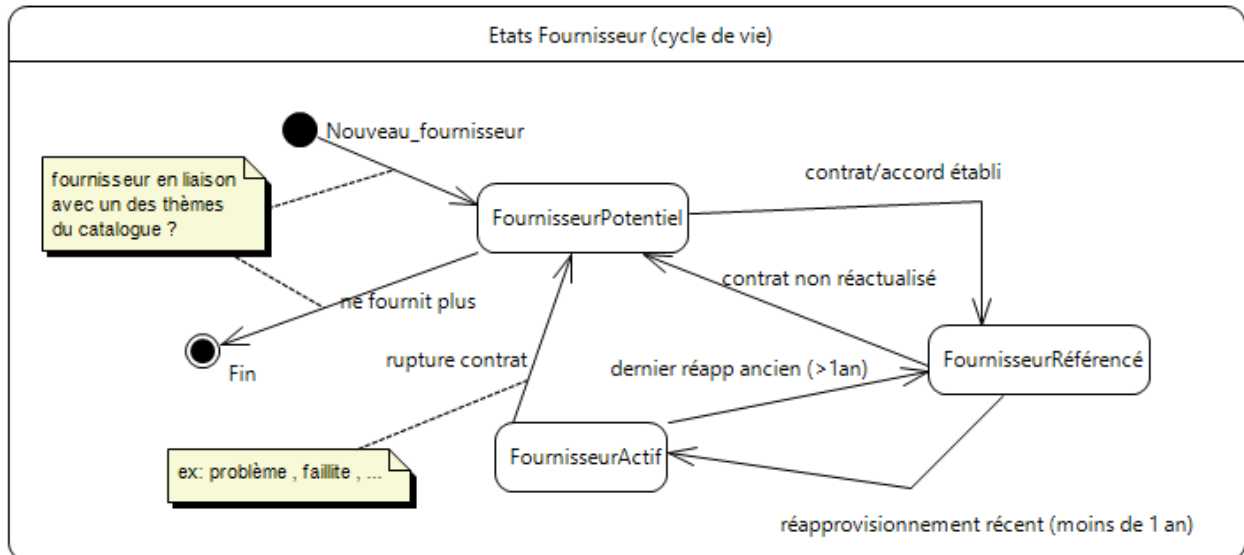
Quelques types (classiques) d'événements :

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

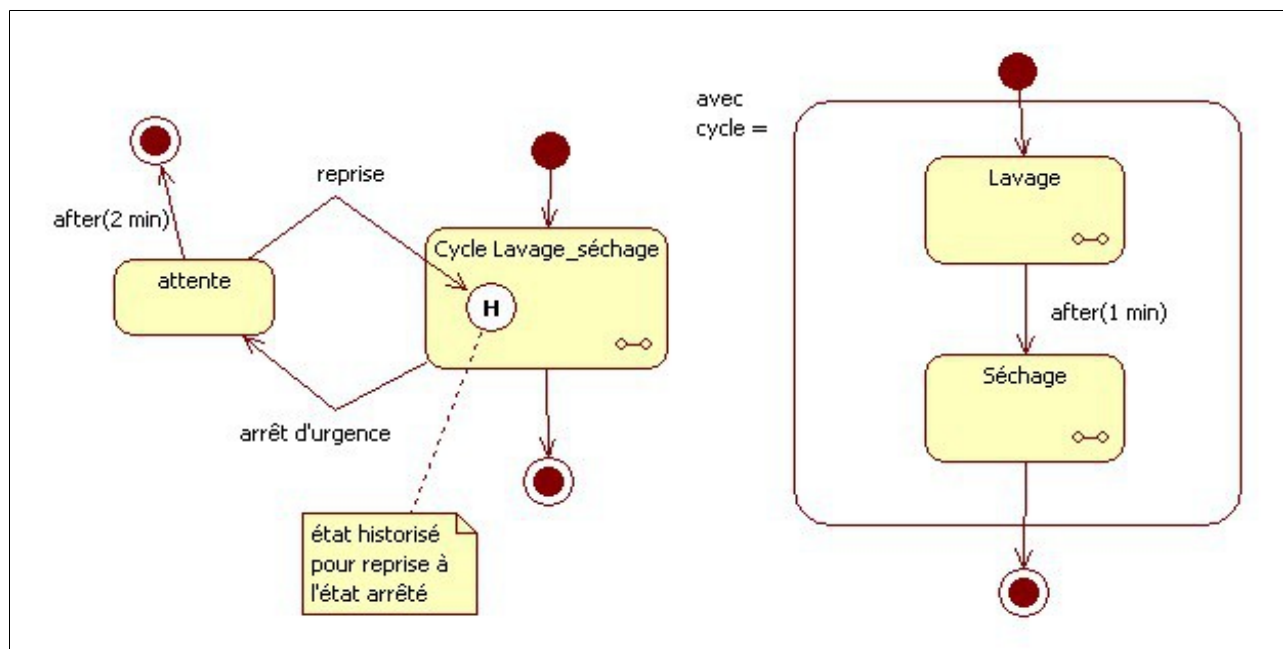
Exemple(s):

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

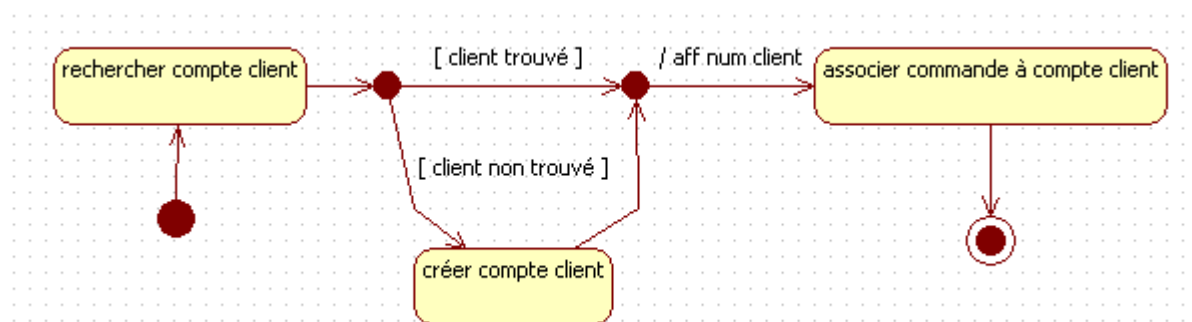
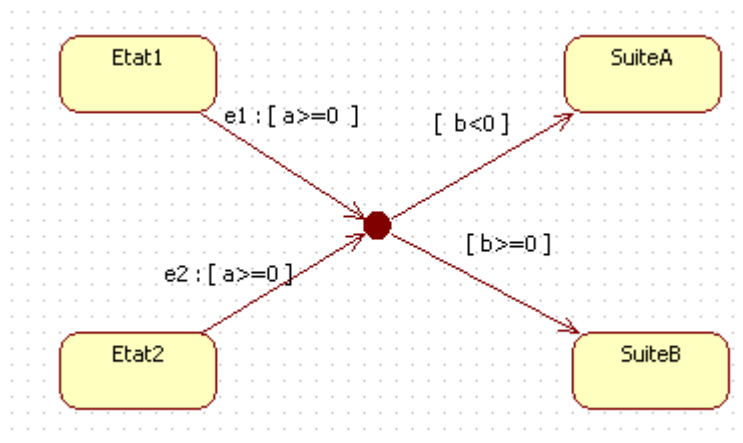
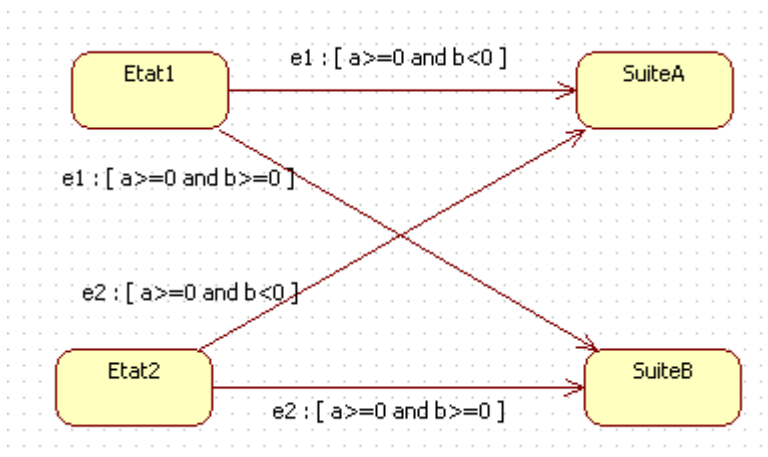
## 1.1. Exemple simple



## 1.2. Etats historisés (rares , pour informatique industrielle)



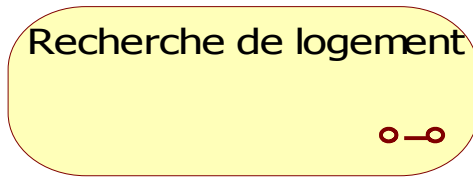
### 1.3. Point de jonction (depuis UML2)



avec des noms d'états à comprendre ici comme en *"train de ...."* sachant que les noms des états sont idéalement des **qualificatifs**.

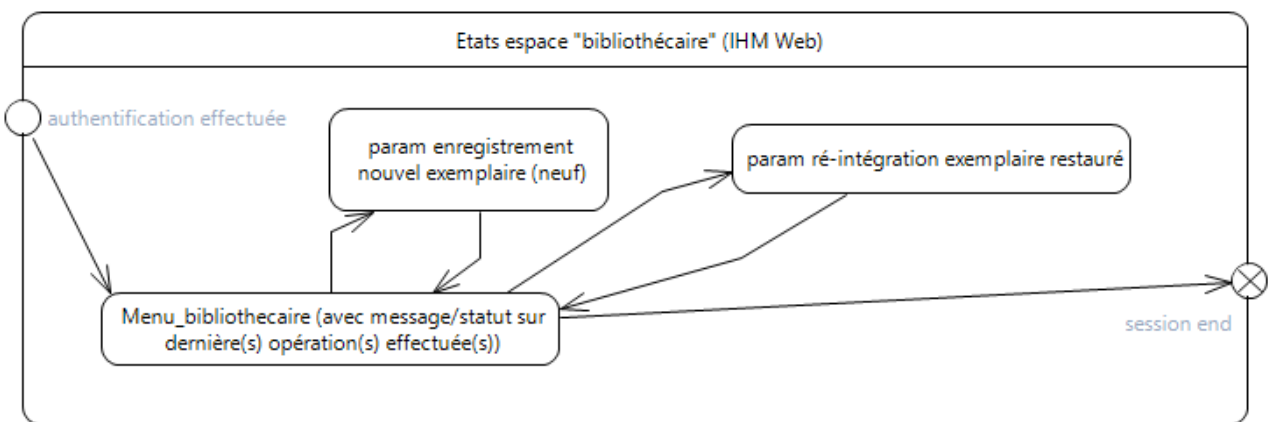
## 1.4. Etat composite (super état)

Notation abrégée d'un état composite (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).

## 1.5. Liens avec diagramme de classe.

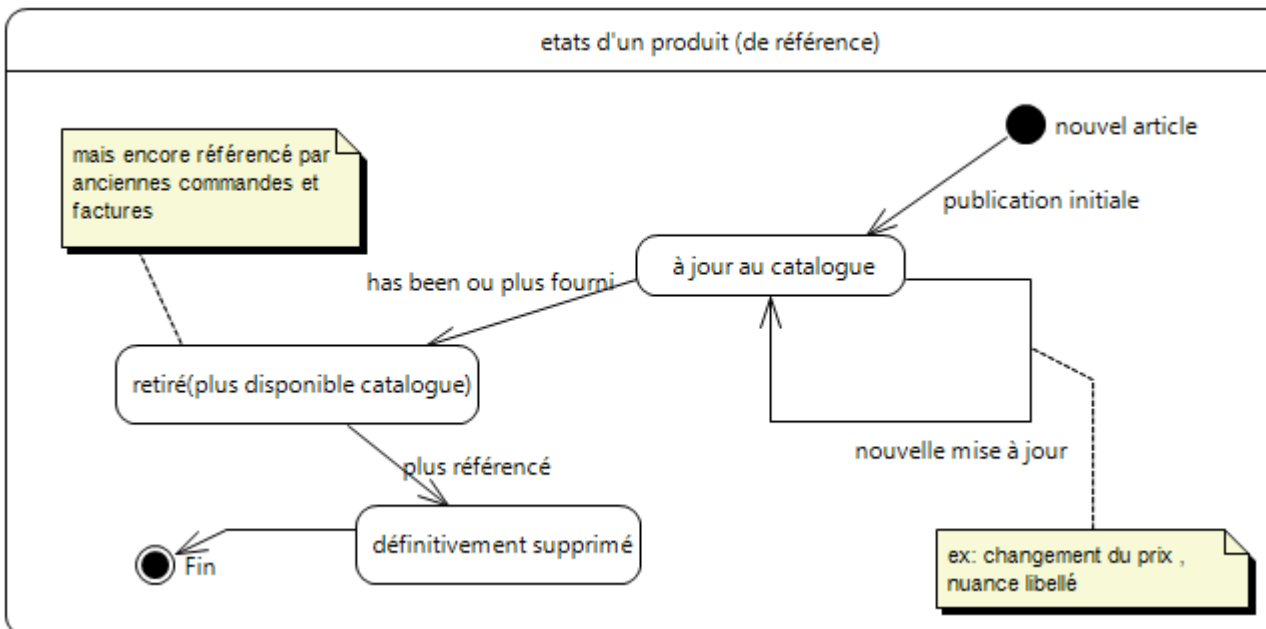
Un diagramme d'état (en tant qu'états de quelque chose) est souvent rattaché à une classe .

**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 ,
- ...

## 2. Utilisation d'un diagramme d'état pour illustrer un cycle de vie

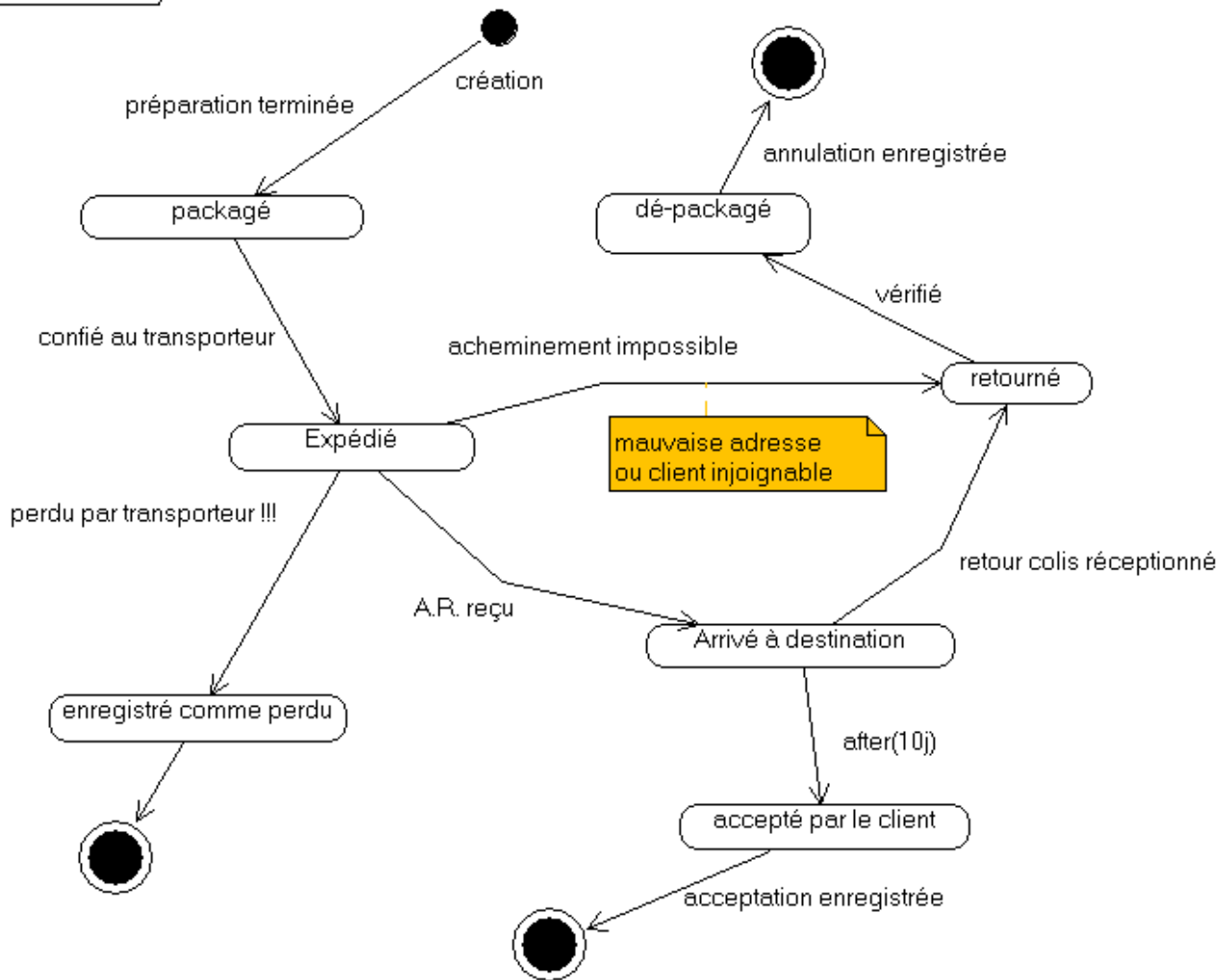
### 2.1. Exemple 1 (cycle de vie d'un produit à vendre)



### 2.2. Exemple 2 (cycle de vie d'un colis)

Colis
+ numColis
+ poids
+ adresse_livraison
+ volume

StateMachine





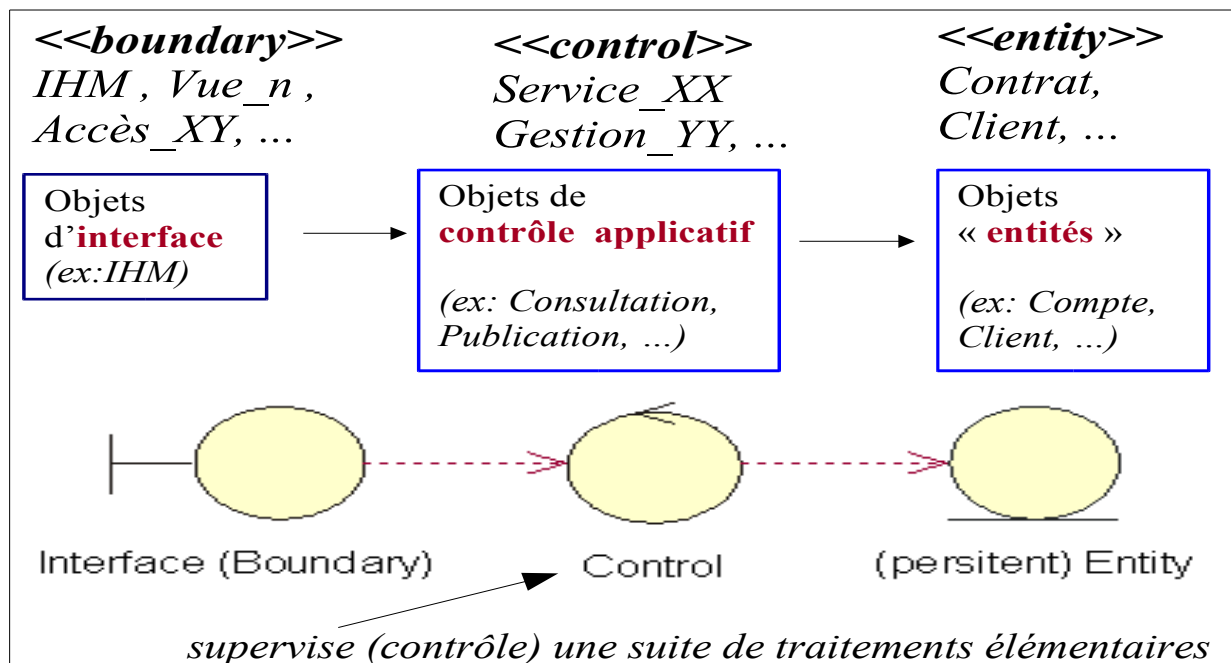
# XIV - Réalisation de Uses Cases / diag. Séquences

## 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>>

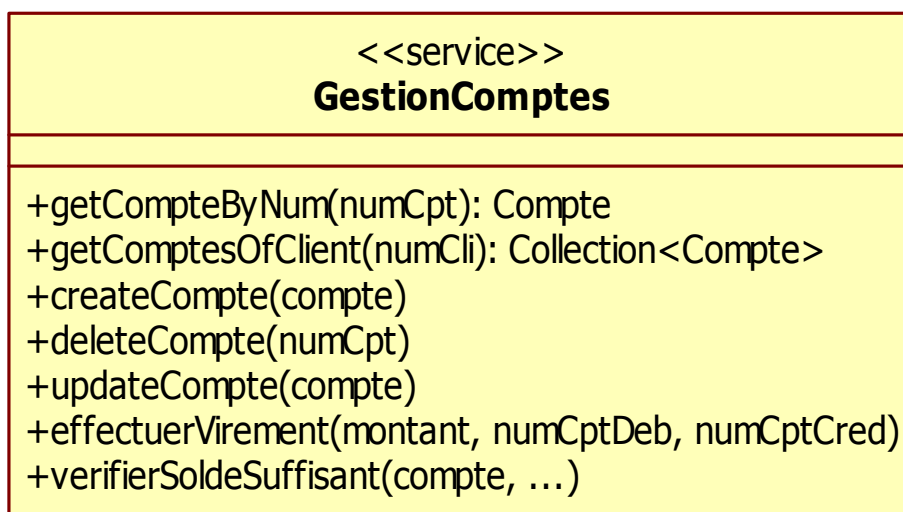
## 2. Responsabilités (n-tiers) et services métiers

<i>Eléments d'une application</i>	<i>Responsabilités</i>
<b>Vues IHM</b>	Afficher , Saisir , Choisir/Sélectionner , Déclencher , Confirmer , ....
<b>Services métiers</b>	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)
<b>Entités (souvent persistantes)</b>	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"):

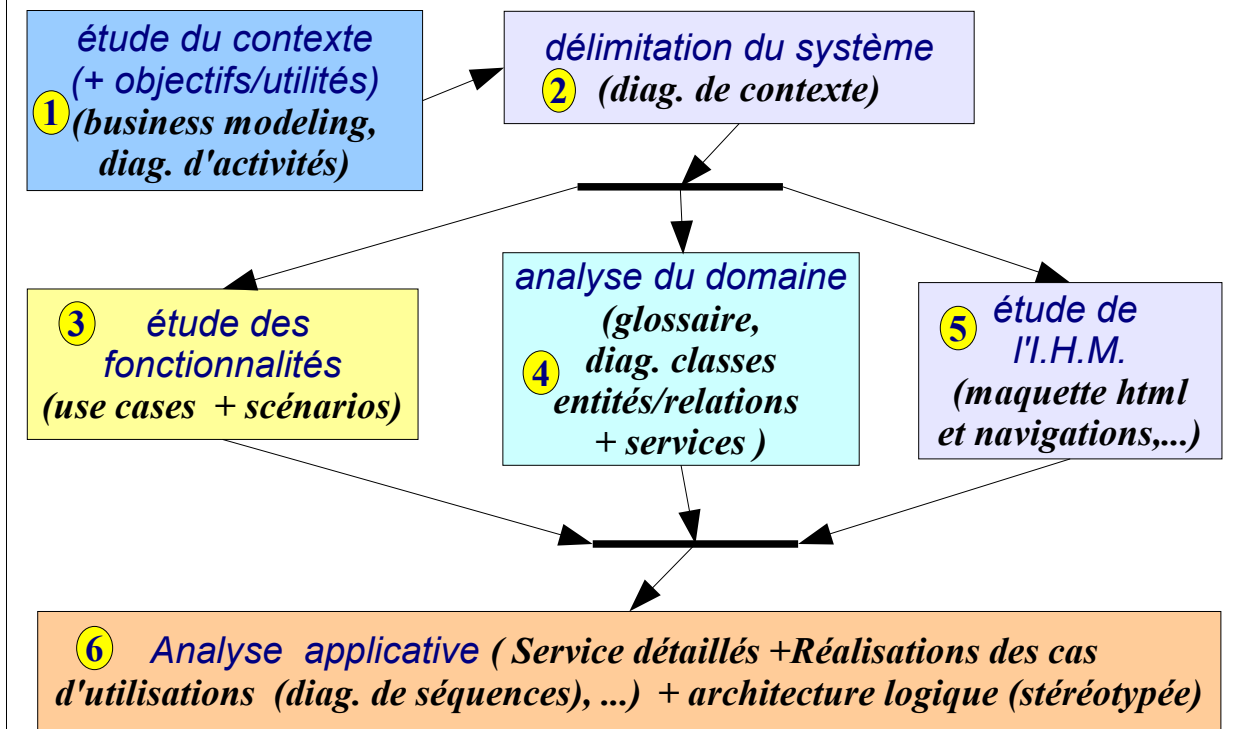


**Pour identifier les objets de contrôles applicatifs / services métiers** on peut se baser sur les compléments d'objets directs des U.C. ou bien sur les noms des packages .

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

### 3. Repère méthodologique (rappel)

#### *Enchaînement classique d'activités sur la partie fonctionnelle*



**L'analyse applicative** a pour principal but de consolider tous les éléments (jusqu'ici séparés/ décorrélés de la modélisation) en montrant clairement leurs complémentarités et leurs **collaborations**.

==> Enfin le moment de réunir "fonctionnalités + entités de données + IHM" en un tout "orienté objet" et cohérent.

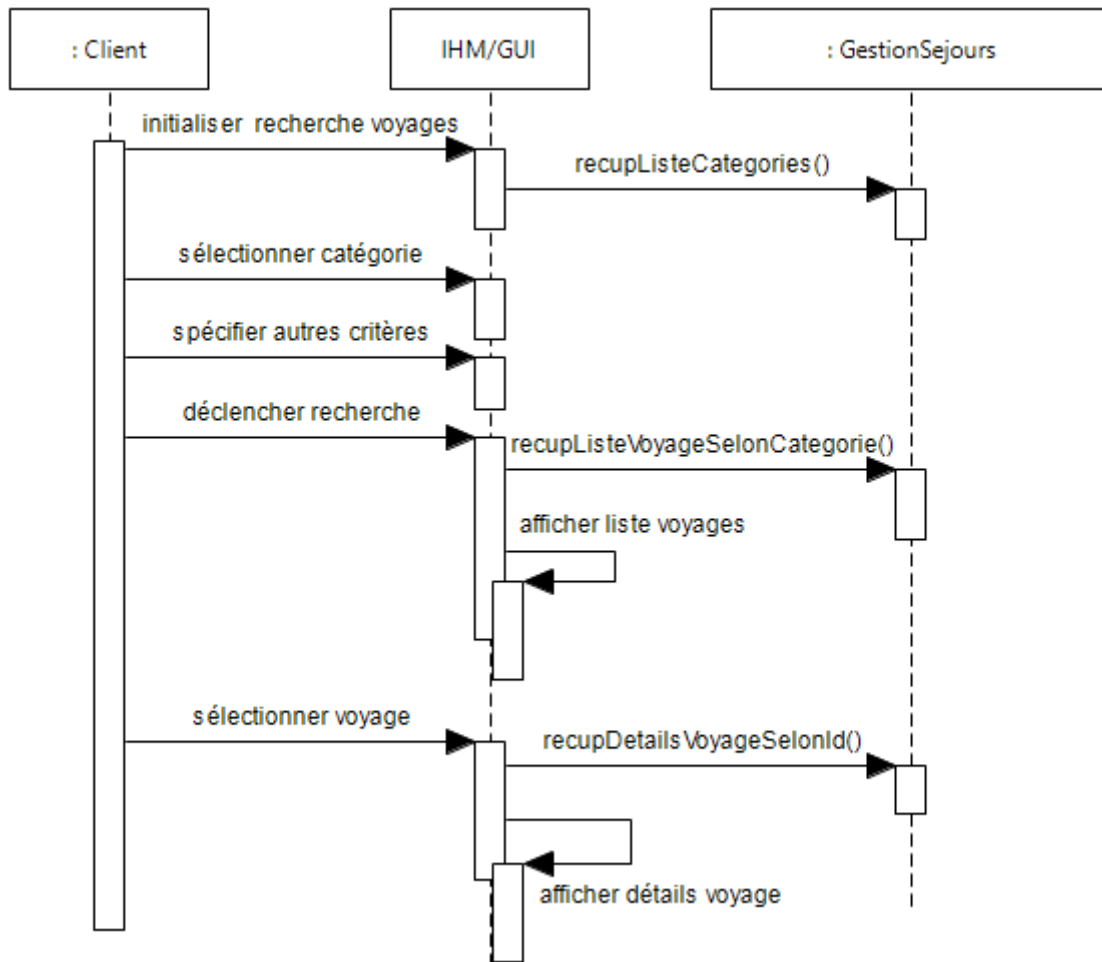
Le principal fil conducteur réside dans la "réalisation des cas d'utilisations" autrement dit dans la retranscription des scénarios des cas d'utilisations en digrammes de séquences.

NB: Ne pas oublier de peaufiner l'analyse en peaufinant bien le découpage en packages fonctionnels (selon catégories métiers).

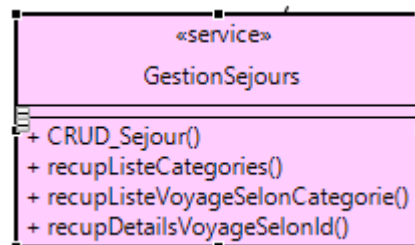
## 4. Réalisation des cas d'utilisations

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) .

## 5. 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**:

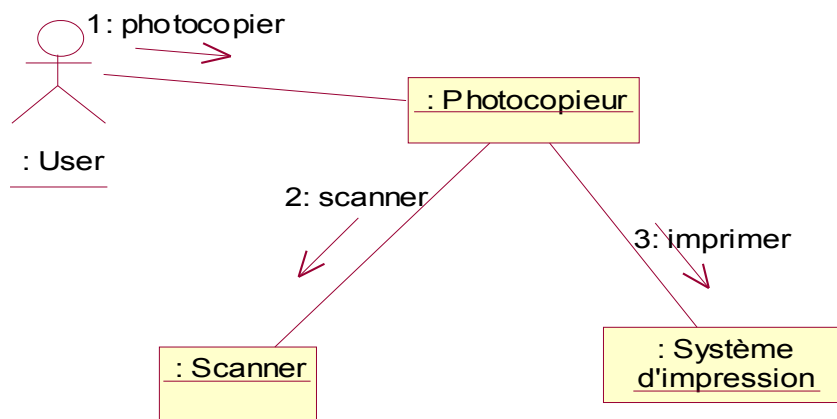
- Des **diagrammes d'interaction**:
  - \* **séquences**
  - \* **collaboration/communication**
- Des **automates**:
  - \* **diagrammes d'états**
  - \* **diagrammes d'activités**



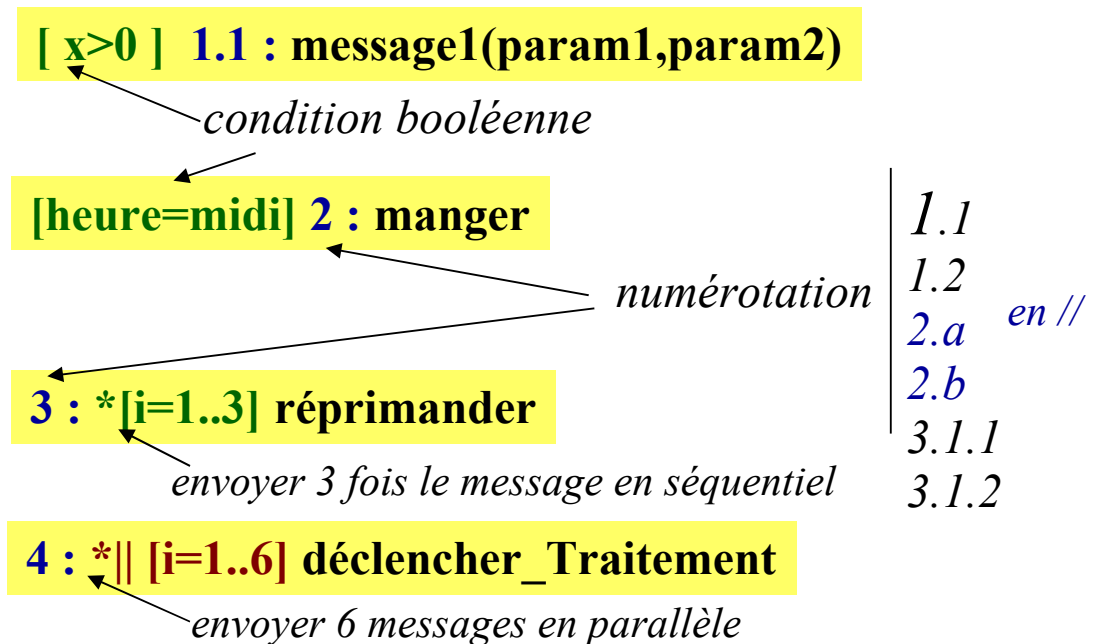
## 6. Diagramme UML de Collaboration / Communication

### Diagramme de **collaboration** (ou de **communication** - UML2)

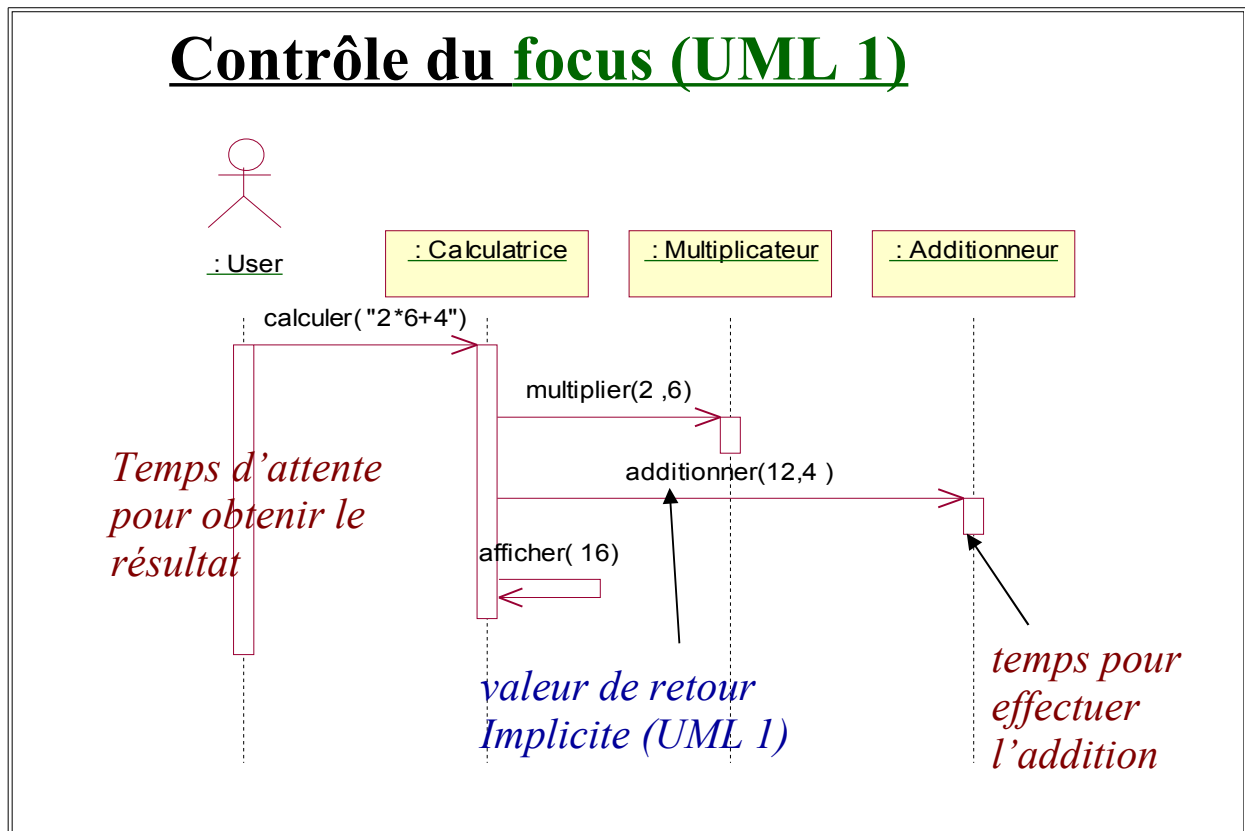
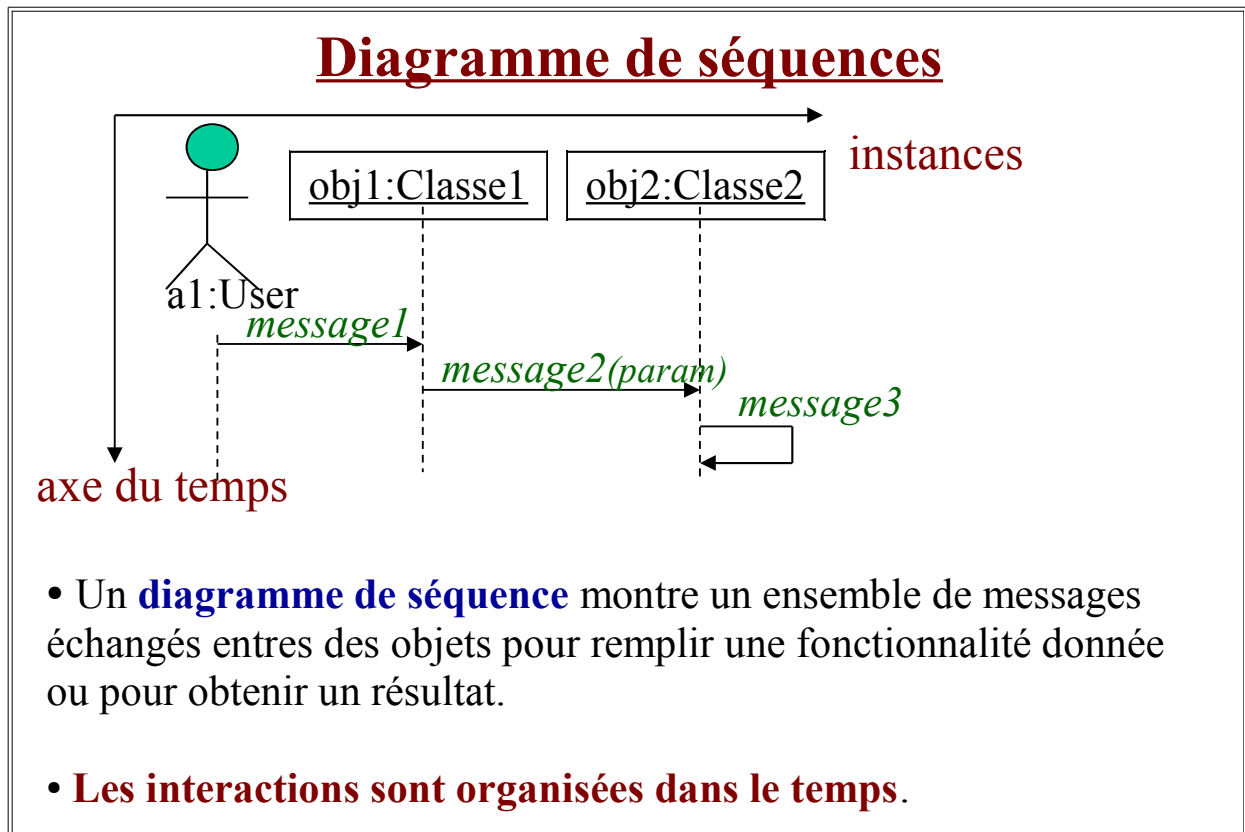
Un **diagramme de collaboration** organise certaines instances dans l'espace (avec des liaisons) et montre certaines **interactions** (messages numérotés).



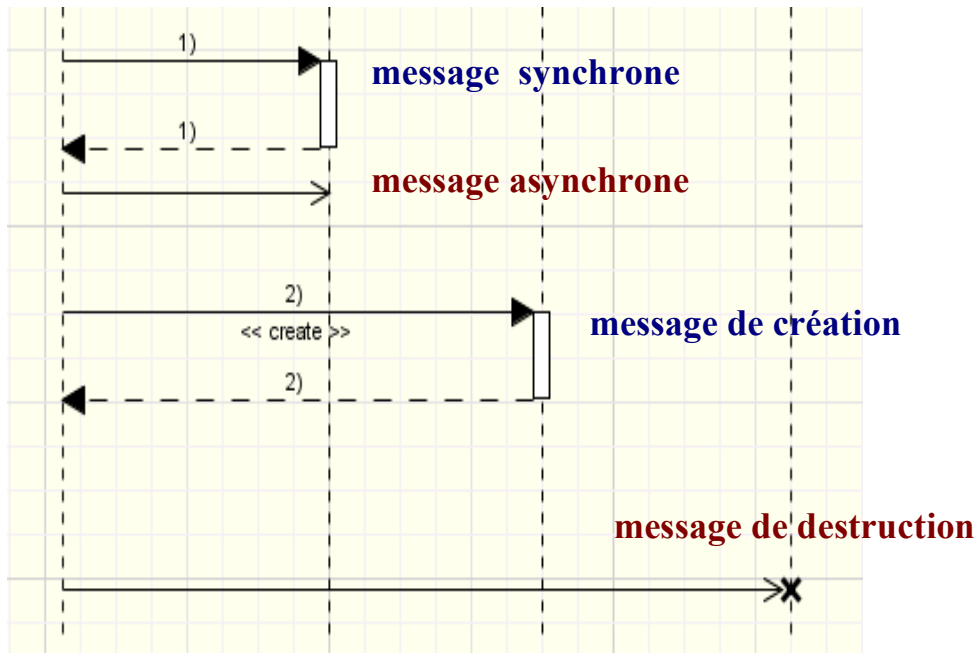
### Détails de synchronisation sur les messages



## 7. Diagramme de séquences (UML)

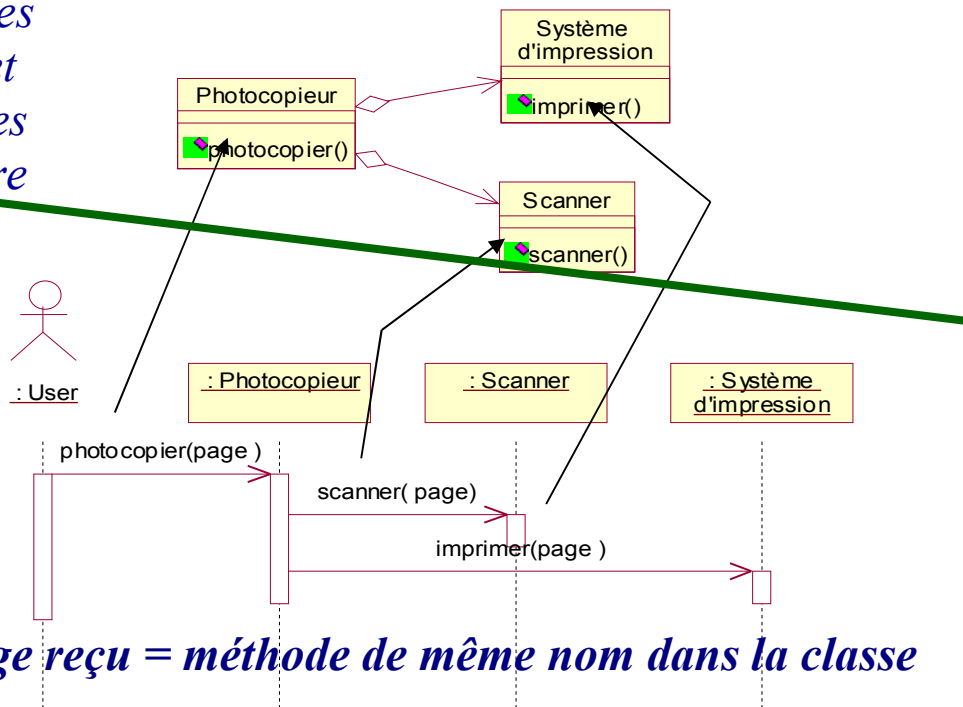


## Différents **types** de messages (UML2)



## Cohérence entre les digrammes

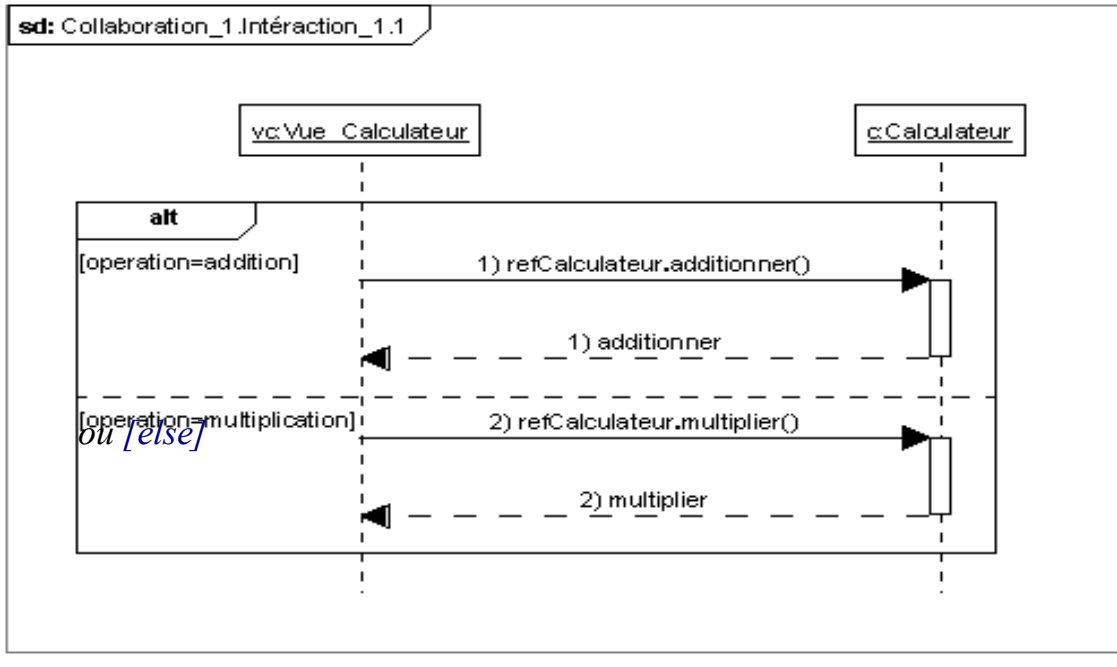
*Les modèles statiques et dynamiques doivent être cohérents*





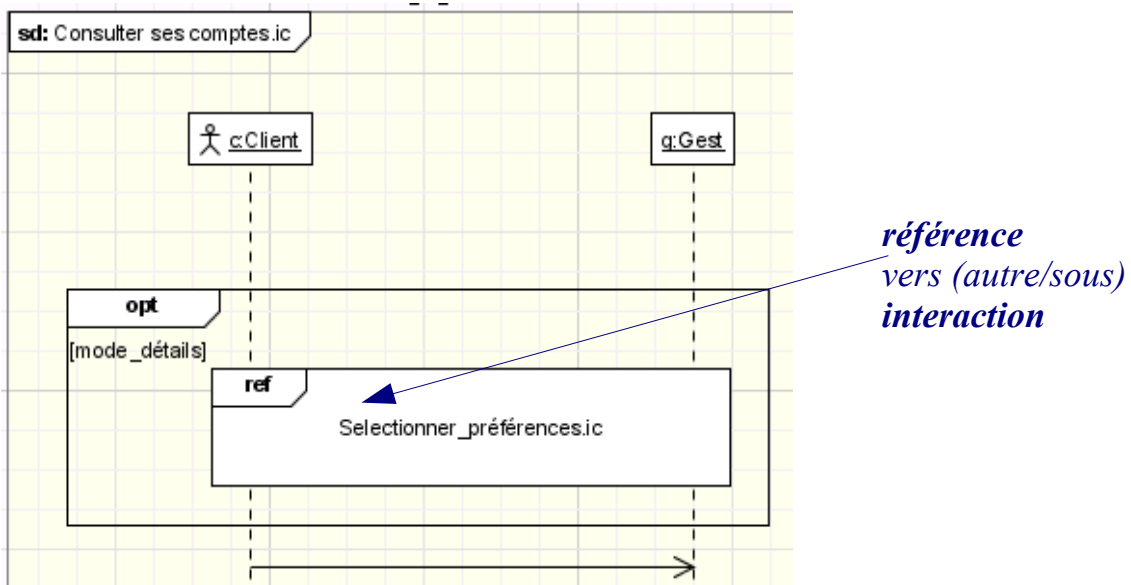
## 8. Diagramme de séquences (notations avancées)

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

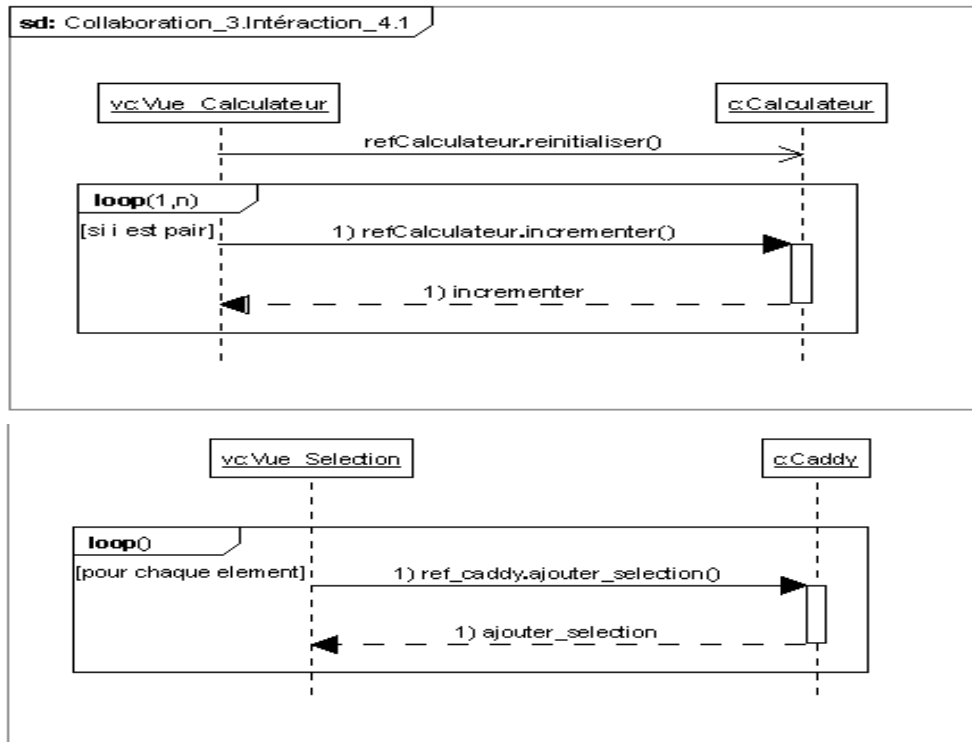


option conditionnée UML2 (**opt**)

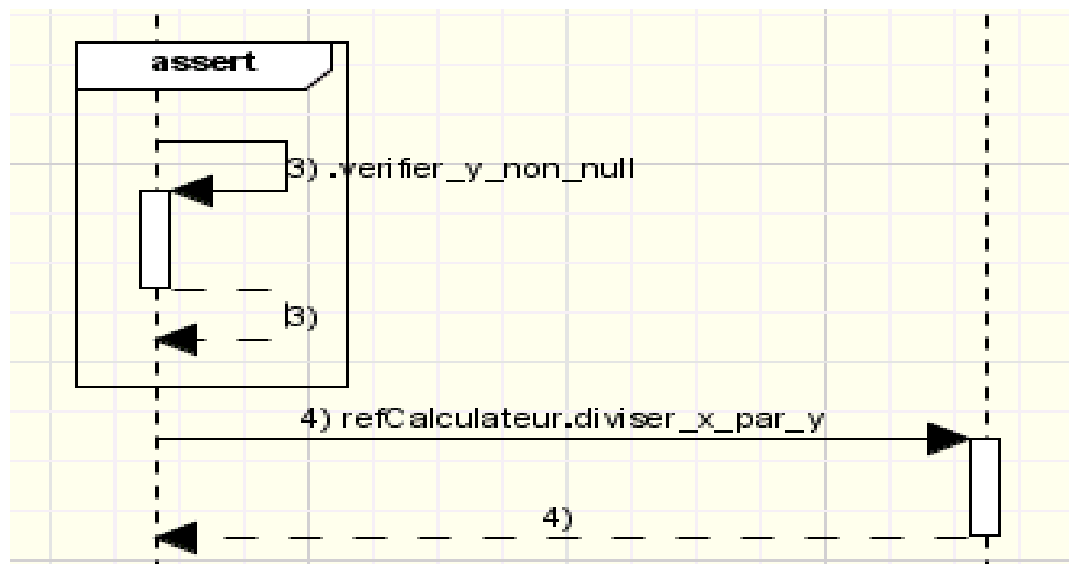
---> *Si [...] Alors (sans else)*



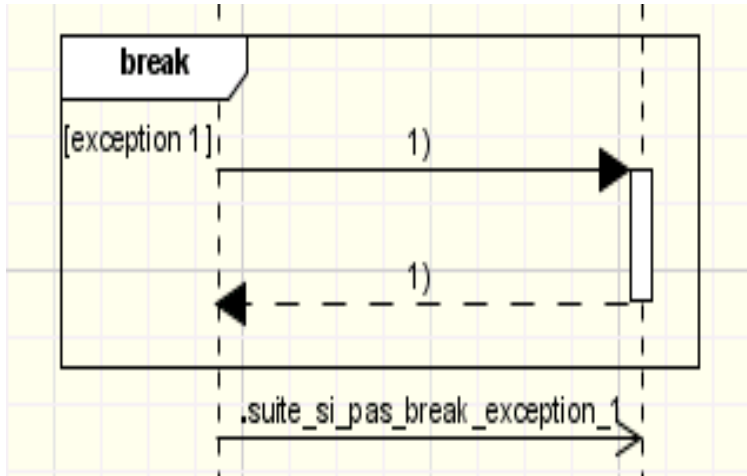
## Boucle / Loop (UML2)



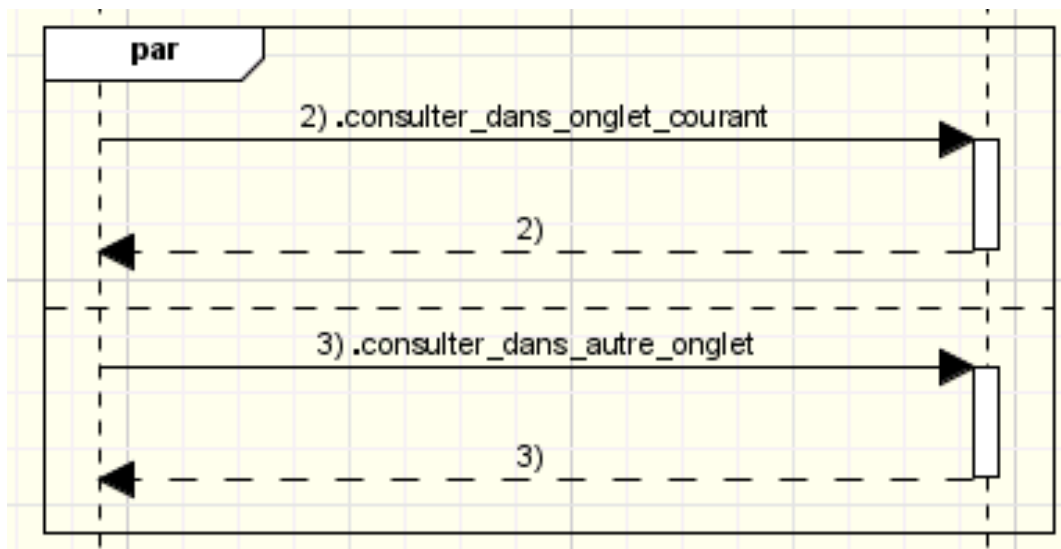
**assertion UML2 (assert)** --> séquence de tests à absolument vérifier pour pouvoir continuer



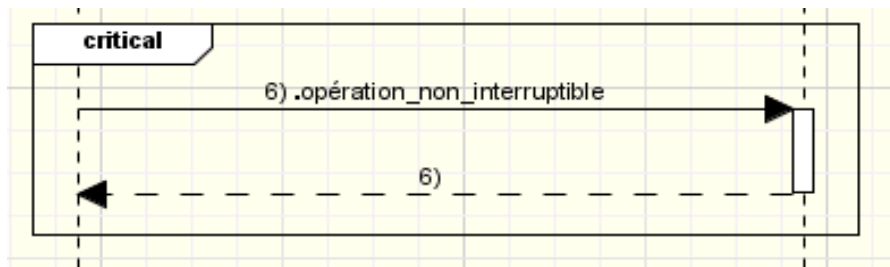
**break** (UML2) --> traitement en cas d'exception  
(*sortie de la séquence normale*, la suite n'est pas exécutée)



**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

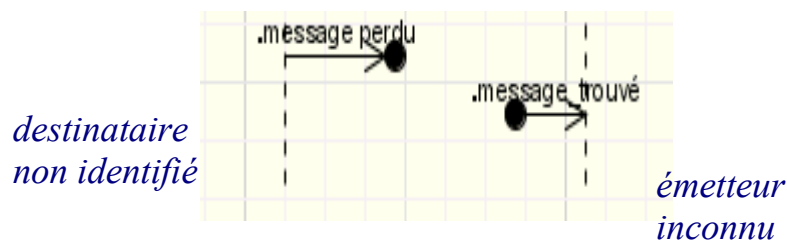
**consider** { message\_important1, m2 }

**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).



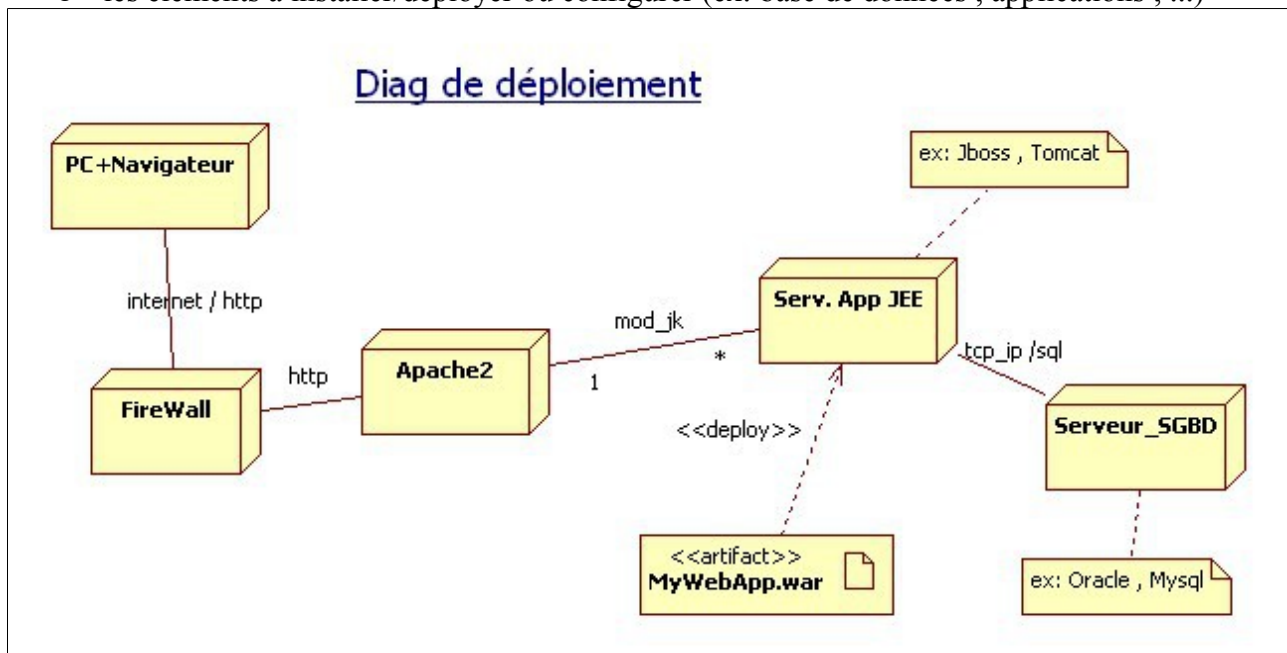
# XV - Besoins techniques / env. / diag déploiement

## 1. Spécification de l'environnement cible

### 1.1. Eventuelle formulation du contexte d'intégration technologique via un diagramme de déploiement UML.

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 , ...)



### 1.2. Éléments de la conception qui découleront des besoins techniques exprimés

- architecture / frameworks techniques / conception générique
- type de base de données (selon volume et transactions XA nécessaires)
- mise en place de "cluster" (si beaucoup d'utilisateurs potentiels / montée en charge prévue ou si haute disponibilité souhaitée)
- politique générale de sécurité (authentification --> à quel niveau ? , ...)
- ....

### 1.3. Eventuelle spécification d'un socle technique préconisé

Pour des raisons diverses (homogénéité souhaitée, compatibilité à assurer , ....) , on peut éventuellement être amené à considérer que la préconisation/spécification d'un socle technique fasse un peu partie de l'expression des besoins techniques.

Si ceci est le cas, il faudra idéalement:

- avoir sérieusement validé le socle technique préconisé (tests & qualifications sérieuses effectués par la direction technique + retour d'expérience sur anciens projets)
- modéliser ce socle technique par différents diagrammes UML génériques montrant essentiellement le rôle de chacun des composants logiciels à mettre en place dans l'architecture

logicielle choisie(et souvent basée sur un ou plusieurs frameworks [ex: JEE + JSF/Spring/Hibernate]).

## 2. Exemples d'exigences techniques classiques

Dimensionnement prévu (avec besoin de bonnes performances):

Volume de données	Environ 50000 exemplaires , 3000 abonnés et 3 emprunts par semaine
Utilisateurs simultanés	Environ 10 utilisateurs (bibliothécaires + responsables)
Temps de réponse	<= 1s en moyenne (<=5s maxi)
...	

Fonctionnalités techniques attendues :

Transactions	Nécessaires sur tous les traitements générant des modifications dans la base de données
Authentification	* pour "bibliothécaire" et "responsable abonnement" * mode classique (username,password) , pas besoin de ldap ni de sso
Confidentialité	Version 1 (intranet) : RAS Future version2 (internet) : cryptage SSL/HTTPS
...	

Grandes lignes sur l'environnement technologique :

L'application bibliothèque devra être compatible avec l'environnement cible suivant :

- base de données MySQL
- serveur Tomcat7 et jdk 1.7 sous linux 64bits

Aspects organisationnels et documentaires :

- Besoin d'un guide utilisateur au format pdf
- Besoin d'un guide d'architecture technique (pdf) pour la future maintenance et évolution de l'application
- Besoin de réceptionner l'application dans le format suivant :
  - archive web prêt à être déployé sur un serveur tomcat (.war)
  - script sql de structuration de la base de données (tables vides)
  - script sql annexe avec jeu de données (pour tests rapides)
  - guide de configuration/installation
  - code source sous forme de projet maven .

# XVI - Aperçu général sur la conception

## 1. Rôles de la conception

La **conception** a 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:

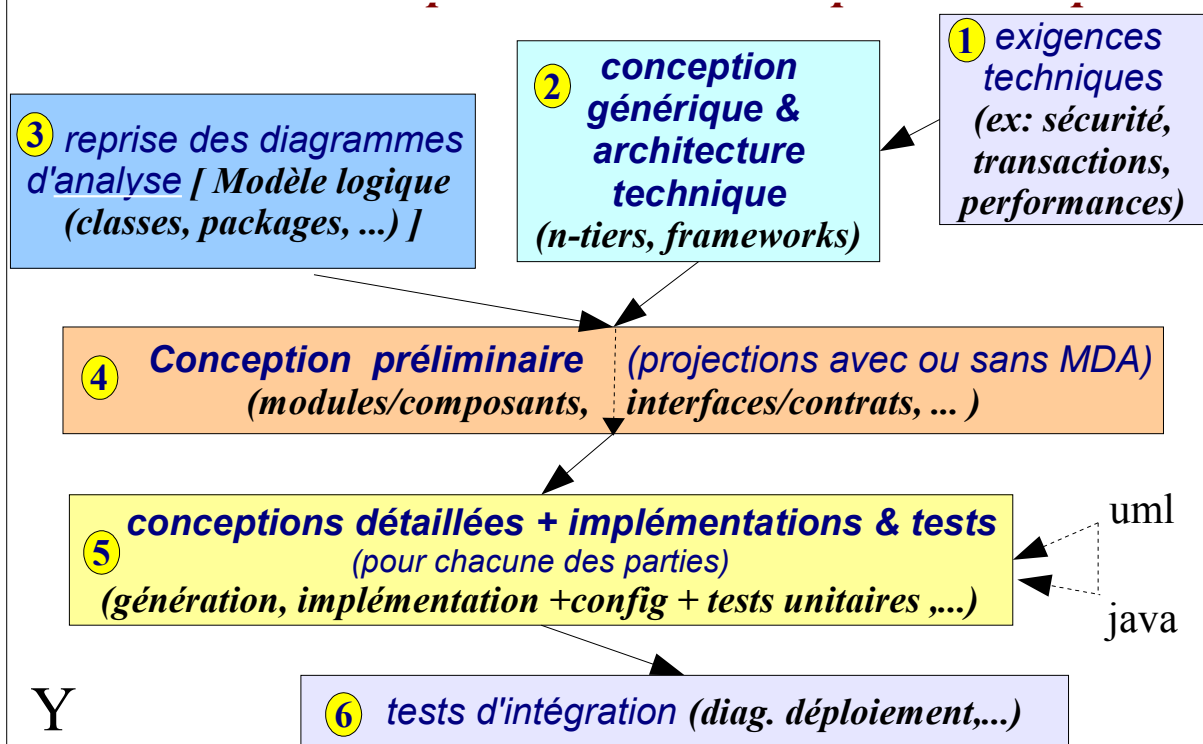
conception ==> **inventer** (concevoir) une solution

pragmatiquement:

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

## 2. Activités de la conception

### Enchaînement classique d'activités sur la partie conception



## 2.1. conception générique et architecture technique.

- Modéliser une bonne fois pour toutes les éléments récurrents.
- Structurer les grandes lignes de l'architecture technique en s'appuyant sur des **frameworks** (prédéfinis ou "maisons")

## 2.2. Conception préliminaire

- **Projeter** le résultat de l'analyse (fonctionnel / métier) dans l'architecture technique (logique ou physique) choisie.
- **Identifier les différents modules** qui seront ultérieurement modélisés et développés séparément .
- **Bien spécifier les interfaces (contrats) entre les différents modules**

## 2.3. conceptions détaillées

- conceptions détaillées (séparées , en // ) de chacun des modules identifiés par la conception préliminaire.
- **Modélisation UML ==> codage/implémentation ou bien codage direct (avec modèle générique dans la tête) ==> reverse engineering (doc UML)**

## 2.4. implémentation & tests

- Tests unitaires via JUnit ou ... (validation de chaque module)

## 2.5. tests d'intégration

- Test d'intégration global (collaboration efficace entre les différents modules ?)

---

### Important:

Entre petit et gros (projet/équipe) , la principale différence tient en un besoin plus ou moins important d'homogénéité:

Style libre pas gênant si développement tout seul et s'il n'y pas trop de répétition.

Style libre (exotique) très gênant si développement en équipe ou si diversification de style dans les différentes parties devant être ultérieurement assemblées.



### 3. Conception générique

#### 3.1. Infrastructure technique générique (framework , ...)

De façon à :

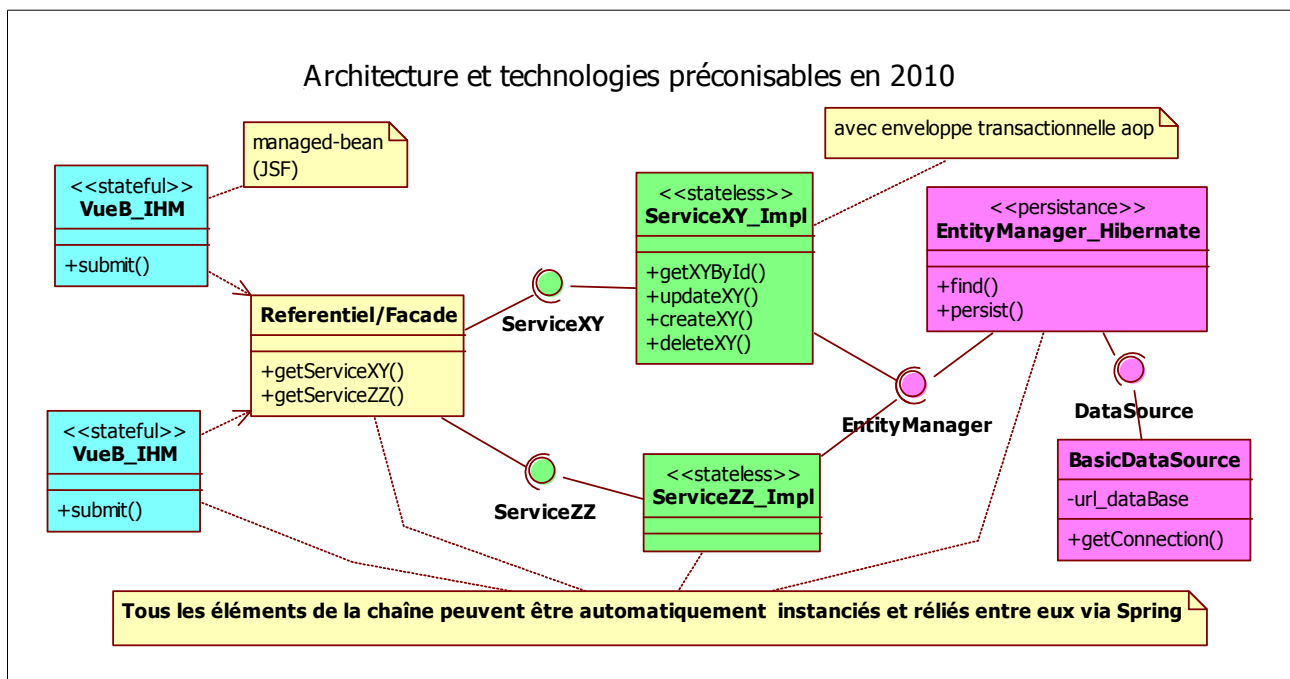
- répondre aux principaux besoins techniques préalablement identifiés .
- bien dissocier l'interface d'une couche de son implémentation ,

on a généralement besoin de s'appuyer sur différents frameworks:

- frameworks techniques prédéfinis (J2EE, MVC2/Struts , ...)
- frameworks spécifiques (à bâtir de toutes pièces en s'inspirant de divers "Design Pattern")

Des diagrammes UML de classes ou de collaboration peuvent être à ce niveau utiles pour décrire la structure et les principes de fonctionnement d'un framework (que ce dernier soit prédéfini ou pas).

Exemple: architecture technique basée sur l'état de l'art du moment (framework JEE , Spring , ....) et devant prendre en charge les exigences techniques de l'application.



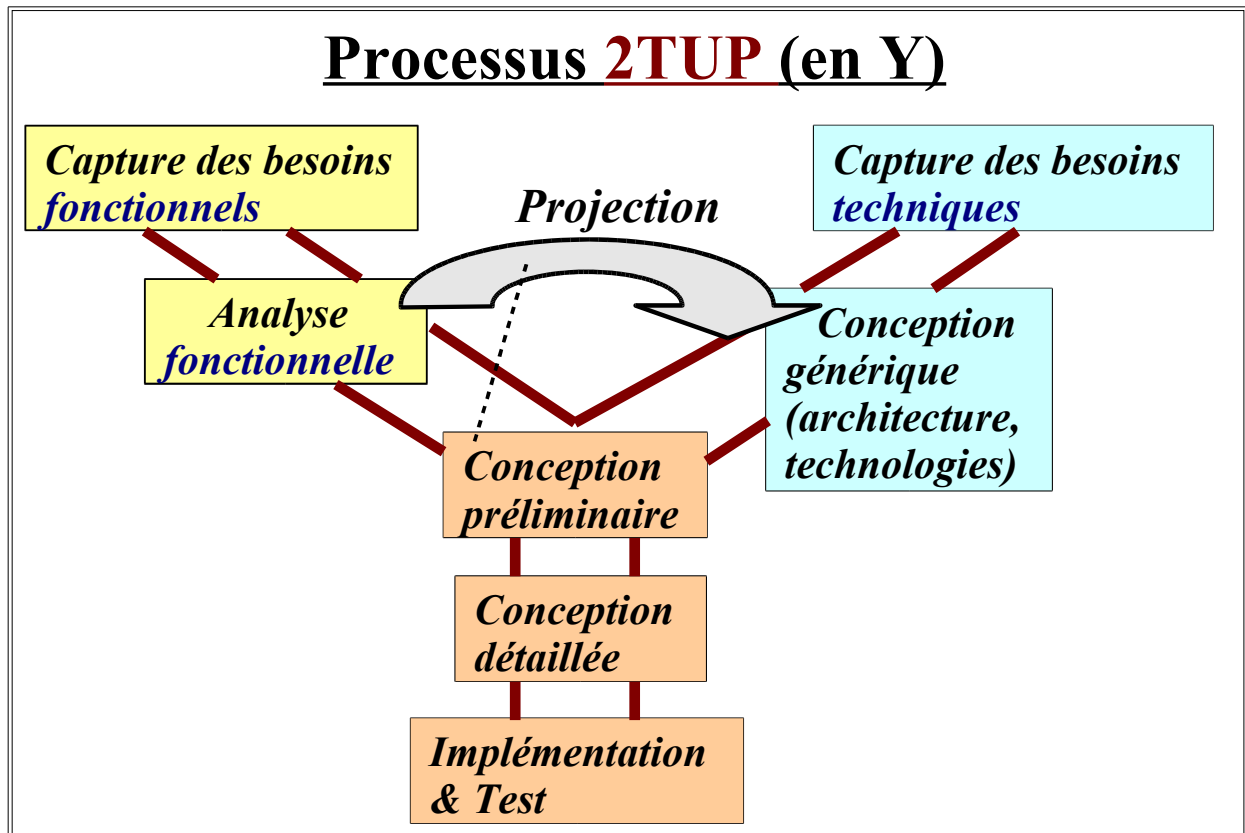
#### 3.2. Intérêts de la conception générique

Les principaux intérêts de la conception générique sont les suivants:

- ne modéliser qu'une seule fois les éléments récurrents
- dégager les invariants et bâtir ou réutiliser des solutions génériques permettant de gagner beaucoup de temps sur le développement.
- modéliser un template pour une éventuelle génération de code automatique (ex: accéleo / MDA).

## 4. Objectifs de la conception préliminaire

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.partielHM.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 .

## XVII - Implémentation , retours tests , itérations

De façon à progresser , il est **indispensable** d'*effectuer un suivi* de ce qui sera développé en aval de la modélisation effectuée .

*Modélisation initiale (premières idées) ==> implémentation & tests*

*==> bonnes et mauvaises critiques ==> nouvelle itération dans le cycle  
(modélisation ré-ajustée  
si nécessaire , ....)*

*Bonne pratique !!!*

# ANNEXES

## XVIII - Différences entre UML et Merise

### 1. différences cardinalités/multiplicités

L'une des principales difficultés du mapping objet-relationnel réside dans le besoin impératif de ne pas confondre "cardinalité" (Merise, relationnel) et multiplicité (UML).

Termes idéals:

Avec UML (objets) ==> **Classes/Types , associations et multiplicités.**

Avec Modèle relationnel (MCD , MLD) ==> **Entités , relations et cardinalités.**

Cardinalité:

La **cardinalité** (*exemple: (1,N)*) correspond au **nombre** minimum et maximum **de fois où une entité précise peut participer à une relation** .

Autrement dit , la **cardinalité** indique les **nombre** minimum et maximum **de fois où une entité précise peut être reliée** à une autre (assez souvent d'un autre type) **dans le cadre d'une certaine relation**. [NB: l'entité précise considérée correspond physiquement à un enregistrement d'une table relationnelle et est assimilable à une instance (objet précis)]

Exemple: dans la relation "1 Personne possède zéro , une ou plusieurs Voitures" , une entité de type "Personne" peut être reliée 0 à N fois à une entité de type "Voiture" et on indique donc une cardinalité de (0,N) du côté "Personne" dans les diagrammes relationnels (MCD,MLD).

Multiplicité (UML):

La **multiplicité** (*exemple: "0..1" ou "0..\*"*) correspond au **nombre** minimum et maximum **d'élément(s) d'un certain type (d'une certaine "Classe") qui peuvent être conceptuellement associé(s)** à un autre élément (souvent d'un autre type) **dans le cadre d'une certaine association**.

Exemple: dans l'association "1 Personne possède zéro , une ou plusieurs Voitures" , une seule entité **de type "Personne"** est généralement associée à une certaine Voiture (cas particulier d'une voiture pas encore achetée ==> associée à zéro propriétaire). On indique donc généralement une multiplicité de "1" ou "0..1" du côté "**Classe Personne**" dans les diagrammes de classes UML.

NB: **Classe** = ensemble des éléments ayant la même structure (attributs) et les mêmes comportements (méthodes) , **multiplicité** = expression du **nombre d'éléments dans le sous ensemble des éléments associés/associables à l'autre extrémité de l'association**.

Les notions de cardinalité et de multiplicité sont donc très différentes (quasiment inversées) .

## XIX - Annexes - Patterns "GRASP"

### 1. Affectation des responsabilités (GRASP)

#### Affectation/répartition des responsabilités (patterns **GRASP** de *Craig Larman*)

**GRASP** = *General Responsibility Assignment Software Patterns*

Les patterns "GRASP" vise l'ultime objectif suivant:

- Comment répartir au mieux les **responsabilités** (*services rendus aux travers d'un sous-ensemble cohérent de méthodes publiques*) au niveau d'un ensemble de classes plus ou moins inter-connectées ?
- Quelles sont les *affectations* qui garantissent le mieux la **modularité** et l'**extensibilité** de l'ensemble ?

#### Les 4 patterns "GRASP" fondamentaux

**Expert:** affecter la responsabilité à la classe qui détient l'information.

**Faible couplage:** la répartition des responsabilités doit conduire à un faible couplage (relative indépendance)

**Forte cohésion :** la répartition des responsabilités doit conduire à une forte cohésion (pas de dispersion , ...)

**Création:** La responsabilité de créer une instance incombe à la classe qui agrège, contient, enregistre, utilise étroitement ou dispose des données d'initialisation de la chose à créer.

## 5 patterns "GRASP" plus spécifiques

**Contrôleur:** classe supervisant des interactions élémentaires , stéréotype <<*control*>> , en liaison avec scénario de U.C.

**Polymorphisme:** pour petites variations au niveau des sous classes tout en gardant une homogénéité et une bonne extensibilité.

**Fabrication pure:** affecter un ensemble de responsabilités fortement cohésif à une *classe artificielle* ou de commodité qui ne représente pas un concept du modèle du domaine .

**Indirection:** ajouter un *intermédiaire* entre 2 éléments pour éviter de les coupler de façon trop rigide.

**Protection des variations:** *anticiper* de futures *variations* et les placer derrières des *interfaces stables*.

## 2. Les 4 patterns GRASP fondamentaux

### 2.1. Expert

#### Expert (GRASP)

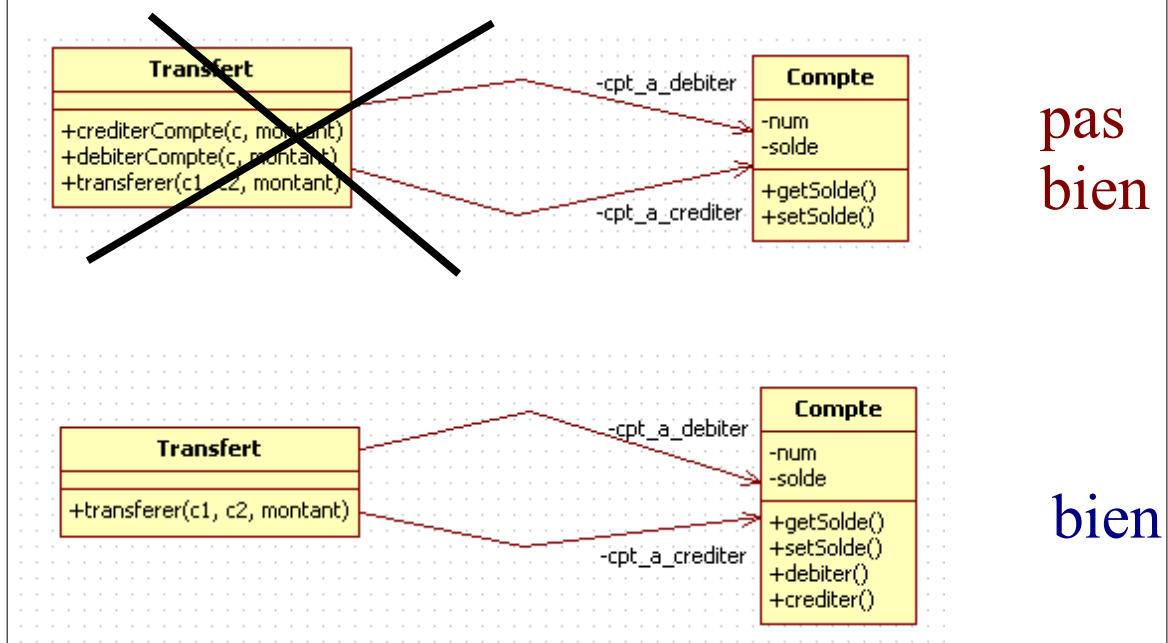
L'*expert*, c'est celui qui sait (qui détient quelques informations clefs) et qui peut donc prétendre pouvoir assumer une certaine responsabilité.

--> Placer de préférence les services/méthodes (traitements internes) au plus près des données utiles en passant par le moins d'intermédiaires possibles (*l'autonomie est à rechercher au niveau des objets*).

--> Déléguer le plus possible  
(sous services / sous responsabilités / subsidiarité , ...)

--> Ne prendre du recul vis à vis des informations que s'il faut agir à un niveau relativement global.

#### Expert (exemple)





## 2.2. Faible couplage

### Faible couplage (GRASP)

Le couplage désigne la densité des liens/relations existants entre les différents objets d'un système.

--> Trop de couplage (beaucoup d'inter-dépendances,...) amène généralement à une grande complexité, une certaine fragilité de l'édifice et à l'impossibilité de réutiliser un seul élément sans avoir à comprendre et réutiliser aussi tout ce qu'il y a autour.

--> Inversement un couplage trop faible est quelquefois la marque d'une chose monolithique (pas assez décomposée) ou bien une entité assez isolée rendant peu de services utiles.

--> Le bon niveau de couplage est une affaire de compromis et de jugement (*idéalement faible pour minimiser les dépendances*) (*avec quelques liaisons tout de même pour ne pas conduire à trop de parties totalement isolées/déconnectées*).

### Faible couplage (exemple)

*faible couplage  
non respecté !!!*



S'il faut choisir entre 2 cablages, choisir celui qui utilise le moins de fils

## 2.3. Forte cohésion

### Forte cohésion (GRASP)

La forte cohésion d'une classe ou d'un système désigne la cohérence fonctionnelle de l'ensemble (la non dispersion des responsabilités)

--> Une faible cohésion est très souvent la marque d'une mauvaise conception (pas assez de réflexion , d'organisation) et menant à un édifice difficile à comprendre.

--> Une entité fortement cohésive doit normalement faire peu de choses mais le faire bien (à fond ou presque).

--> "Forte cohésion" va souvent dans le même sens que "spécialisation fonctionnelle" dans l'élaboration d'un édifice modulaire.

### Forte cohésion (contre exemple)

#### Refrigerateur\_TV\_Alarme

```
+refrigerer()
+regler_température()
+...()
+selectionner_chaineTV()
+régler_volume_sonore()
+...()
+déclencher_alarme()
+programmer_alarme()
+...()
```

*tout et  
n'importe-quoi !*

## 2.4. Création

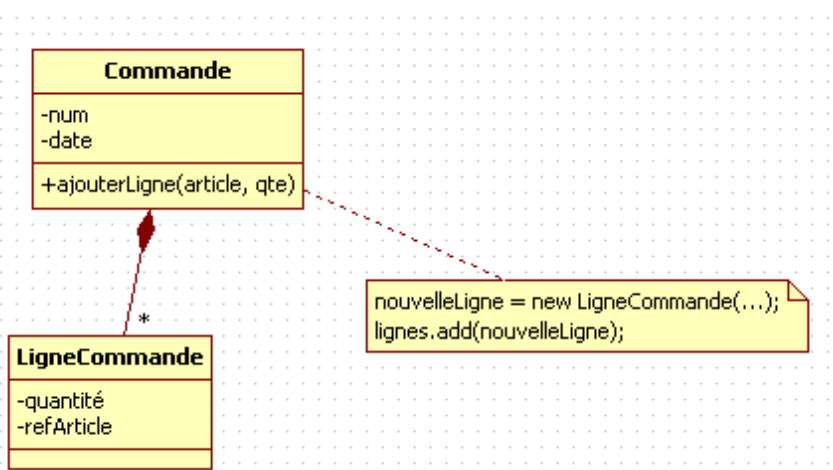
### Créateur (GRASP)

Le design pattern *Créateur* indique que la *responsabilité de créer une nouvelle instance* est bien souvent à affecter à l'élément qui contient, enregistre, initialise ou utilise le nouvel objet qui sera créé.

--> Ceci est assez logique et intuitif car dans l'écriture "*refNewObj = new Cxx(a,b);*" les valeurs *a* et *b* servent à initialiser la nouvelle instance et sont connues par celui qui enregistre ou initialise (ou ...) et la référence retournée permet d'accéder à la nouvelle instance à manipuler ou contenir (ou ..)

--> NB: Appliquer le pattern GRASP "*Créateur*" en fin d'analyse ne dispense pas d'appliquer ultérieurement les patterns "*Factory*" ou "*IOC / injection de dépendances*" en phase de conception de façon à anticiper des variations technologiques ou des extensions.

### Créateur (exemple)



*c'est à la commande que revient la responsabilité de créer une nouvelle ligne de commande.*

Exemple2: C'est normalement la boîte de dialogue qui doit créer ses propres sous éléments (boutons "ok" , "cancel").

### 3. Les 5 patrons GRASP spécifiques

#### 3.1. Contrôleur

##### Contrôleur (GRASP)

Un élément "*contrôleur*" est censé superviser ou coordonner un ensemble cohérent d'opérations/interactions sur des objets quelquefois éparpillés.

--> On peut citer 3 types très classiques de contrôleurs:

- le *contrôleur d'IHM* dans le modèle *MVC* (gestion des événements , déclenchement des actions & affichages)
- le *contrôleur de façade* comme point d'entrée unique d'un module métier complet (rôle d'aiguillage)
- le *contrôleur de séquence applicative ou métier* (très souvent associé à la réalisation d'un *scénario* attaché à un *cas d'utilisation*): classe "*GestionnaireXxx*" , "*CoordonnateurXxx*" ou "*SessionXxx*" et ayant le stéréotype d'analyse <<*control*>> (*Jacobson*).

#### 3.2. Polymorphisme

##### Polymorphisme (GRASP)

Une même opération abstraite peut être implémentée plusieurs fois avec des variantes liées aux types exacts des objets.

==> un même fond (service rendu) mais plusieurs formes possibles (au cas par cas) devant quelquefois cohabitées.

==> automatisme des langages objets dans le choix de la variante (sans if/else ...à coder).

### 3.3. Fabrication pure

#### Fabrication pure (GRASP)

\* Une ***fabrication pure*** est une ***réalisation/construction totalement artificielle*** qui ne correspond pas directement à un des éléments du domaine (*de la réalité*) mais qui est nécessaire pour obtenir une bonne conception (faible couplage, ...).

\* Une fabrication pure est généralement une *entité assez abstraite* (ex: fabrique , D.A.O. , intermédiaireXY, ...) qui est obtenue par *décomposition comportementale ou représentationnelle/structurelle*. Autrement dit certaines responsabilités sont extraites d'un élément du domaine pour être déplacées dans une *entité annexe artificielle*.

### 3.4. Indirection

#### Indirection (GRASP)

En ajoutant (de façon avisée) un ***nouvel élément intermédiaire*** (et par conséquence une ***nouvelle indirection*** dans une séquence d'interactions), on répartit généralement mieux les responsabilités et l'on aboutit à un semble plus modulaire.

Exemples:

\* "Source de données" pour indirectement établir une connection à une base de données.

\* "Décorateur (gof)" pour indirectement ajouter quelques fonctionnalités à un élément de base.

### 3.5. Protection des variations

#### Protection des variations (GRASP)

Certaines évolutions/modifications du système sont quelquefois prévisibles et l'on peut anticiper leurs mises en oeuvre en plaçant l'instable derrière une interface stable.

==> La notion d'encapsulation va dans ce sens (public / privé).

==> Les interfaces abstraites constituent le moyen le plus classique et le plus efficace de se protéger contre les variations. Etant néanmoins vue comme un contrat idéalement inaltérable, l'interface doit absolument être bien pensée pour demeurer stable et utilisable (non jetable) *[sinon c'est quelquefois pire que mieux]*.

## XX - Annexes - Méthodologies (aperçu rapide)

### 1. Bonnes pratiques UP

#### Points communs des U.P.

- \* **macro-processus incrémental** (ex: *proto1, proto2, alpha, bêta, v1, v2*)
- \* **piloté par les risques** (*technique et architecture appropriée, satisfaire les besoins des utilisateurs, anticiper les Pb, prototypes, tests, validation, ...*).
- \* **construit autour de la création et de la maintenance d'un modèle** (ex: *Diagrammes UML, ...*)
- \* **itératif** (*itérer sur une succession d'étapes = micro-processus n° i effectué, ré-effectué, peaufiné, ...*).
- \* **orienté composant** (*réutilisabilité, déploiement, standardisation, ...*).
- \* **centré sur l'architecture du système**

#### 1.1. RUP (Rational UP – origine d'UP)

RUP (Rational U.P.) est le processus U.P. proposé par la **société Rational** (maintenant rachetée par **IBM**). RUP est un précurseur, il est à l'origine même de UP.

Les principales caractéristiques de RUP sont les suivantes:

- Processus très détaillé (beaucoup de choses doivent être **explicitées** sur des **documents à produire**).  
Ceci en fait un processus assez "**bureaucratique**" qui convient bien sur des (très) **gros projets**.
- Processus nécessitant des outils sophistiqués et des équipes de travail assez importantes.

## 2. Méthodes agiles

Méthodes adaptatives plutôt que prédictives .

### 2.1. Origine des méthodes agiles : une réponse à un malaise

#### 2.1.a. *Quelques éléments du malaise du début des années 2000*

- Contexte économique difficile – **pression forte**
- **Rapports quelquefois tendus entre moa et moe**
- **Confusion entre les besoins estimés et réels** (mauvaise communication)
- **Les négociations contractuelles (positions retranchées) font oublier l'objectif initial: chacun s'abrite derrière les modalités d'un forfait qui fige dans le marbre des spécifications très souvent incomplètes.**
- **beaucoup de dérapages (budget , retards , ....)**
- **certains projets sont abandonnés (ou sont fortement restreints).**
- **Les Méthodes (Merise=has been , UML/RUP : trop lourd , ....) sont souvent mal utilisées et les AGL ne tiennent pas leurs promesses.**

#### 2.1.b. *Facteurs clefs des échecs*

- Le **manque de communication** à tout niveau (moa/moe, ...)
- Une **mauvaise compréhension des besoins**
- L'**insuffisance des tests**
- L'absence d'une démarche prudente "gros projet – commencer petit"
- Les **effets "Tunnel"** (cycle non itératif mais linéaire)
- L'insuffisance de l'architecture
- L'absence de maturité des outils utilisés
- La mauvaise formation des personnes
- Le cadre contractuel inadapté

#### 2.1.c. *Facteurs à prendre en compte pour avoir une chance de réussir*

Il faut prendre en compte les risques !!!

4 grands facteurs:

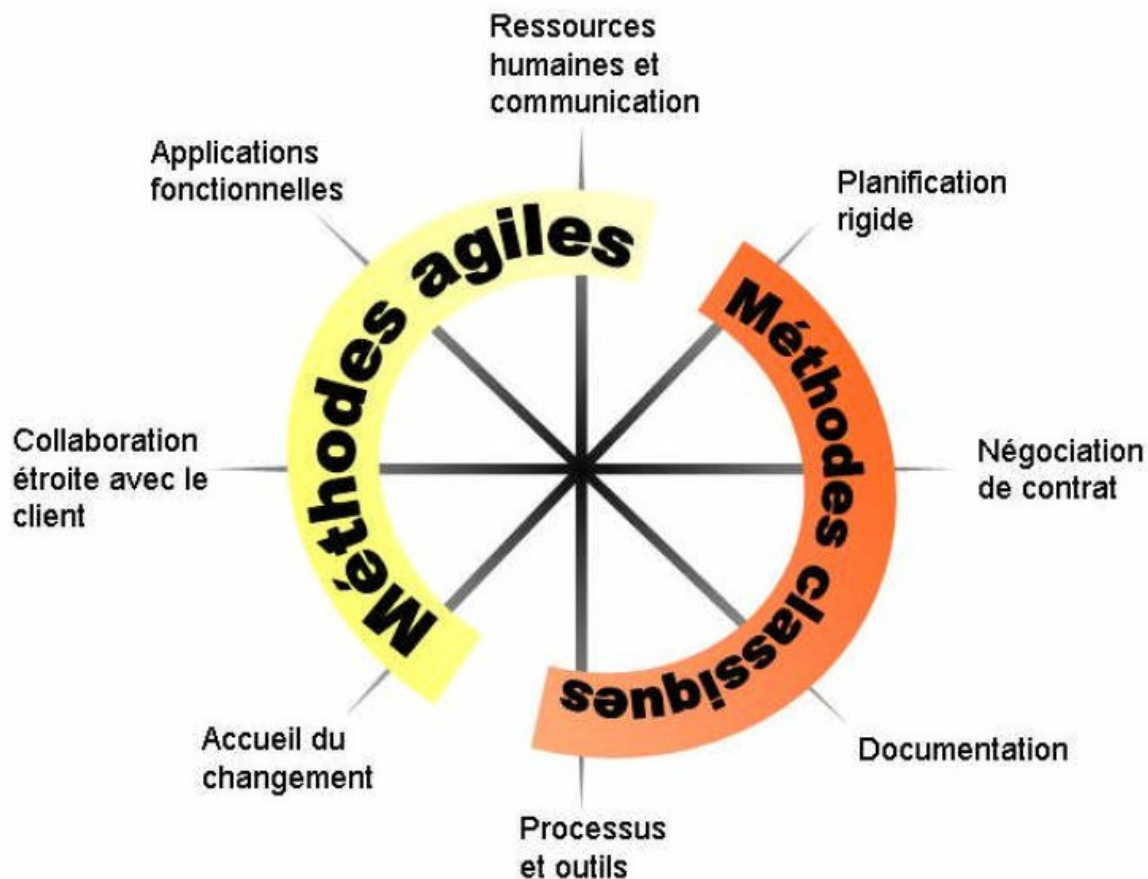
- **Coût**
- **Qualité**
- **Durée**
- **Périmètre fonctionnel**

### 2.2. Principales caractéristiques des méthodes agiles

#### 2.2.a. *Manifeste des priorités (fondamentaux des méthodes agiles)*



- **Priorité des personnes et des échanges/communications/interactions sur les procédures et les outils**
- **Priorité de la collaboration continue avec le client sur la négociation de contrat**
- **Priorité d'applications opérationnelles (avec commentaires) sur une documentation exhaustive (pléthorique) .**
- **Priorité de l'acceptation des changements sur la planification**



### 2.2.b. Grands Principes des méthodes agiles

- **Délivrer rapidement et très fréquemment des versions opérationnelles, pour favoriser un feed-back client permanent**
- **Assurer une coopération forte entre client et développeurs**
- **Garder un haut niveau de motivation(rôle du chef)**

- **Le fonctionnement de l'application est le premier indicateur du projet**
- **Garder un rythme soutenable (pas plus de 40 heures par semaine)**
- **Viser l'excellence technique et la simplicité**
- **Accueillir favorablement le changement (sachant qu'il faut du courage pour accepter de "jeter" certaines parties devenues inutiles).**
- **Se remettre en cause régulièrement (refactoring , tests , ....)**

## 2.3. Panorama des principales méthodes agiles

Méthode agile	Nombre optimal de personnes dans l'équipe	Principales caractéristiques
Crystal Clear	<=6	Pratiques très peu contraignantes. Méthode très peu formalisée
XP (eXtreme Programming)	<=12	Importance de la communication informelle et de l'autonomie de l'équipe Intégration journalière
SCRUM		Réunion quotidienne de motivation, de synchronisation, de déblocage et de partage de connaissances.===== Phase initiale , sprint , cloture
RAD (Rapid. App. Dev. )		Ancêtre des méthodes agiles insiste sur ce cycle incrémental et itératif .
FDD (Feature Driven Development)	<=20	Feature et Features set : fonctionnalité et groupe de fonctionnalités). Priorité donnée aux fonctionnalités porteuses de valeur . ==== Itérations très courtes – livrables / features
DSDM (Dynamic Software Development Method)		Très sensible aux rôles et responsabilités bien définis: Sponsor exécutif, ambassadeur utilisateur visionnaire, utilisateur conseiller, ... en plus du facilitateur et des rapporteurs
ASD (Adaptive Software Development)		Extensible mais très générale (à adapter au cas pas pas)
[partiellement UP ?]		Pour gros projet – formalisation importante
...		

## 3. XP (Extreme Programming)



La méthode agile "XP (eXtreme Programming)" est née officiellement en octobre [1999](#) avec le livre *Extreme Programming Explained* de [Kent Beck](#).

### 3.1. Principales caractéristiques de XP

(selon wikipédia)

Dans le livre *Extreme Programming Explained*, la méthode est définie comme :

- une tentative de réconcilier l'humain avec la productivité ;
- un style de développement ;
- une discipline de développement d'applications informatiques.

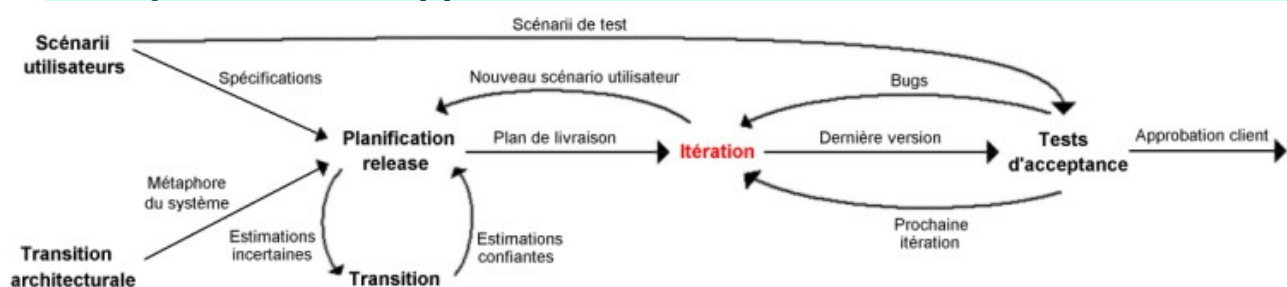
**Son but principal est de réduire les coûts du changement.** Dans les méthodes traditionnelles, les besoins sont définis et souvent fixés au départ du projet informatique ce qui accroît les coûts ultérieurs de modifications. **XP s'attache à rendre le projet plus flexible et ouvert au changement en introduisant des valeurs de base, des principes et des pratiques.**

Les principes de cette méthode ne sont pas nouveaux : ils existent dans l'industrie du logiciel depuis des dizaines d'années et dans les méthodes de management depuis encore plus longtemps.

L'originalité de la méthode est de les pousser à l'extrême :

- puisque la revue de code est une bonne pratique, elle sera faite en permanence (par un binôme) ;
- puisque les tests sont utiles, ils seront faits systématiquement avant chaque implantation ;
- puisque la conception est importante, elle sera faite tout au long du projet (*refactoring*) ;
- puisque la simplicité permet d'avancer plus vite, nous choisirons toujours la solution la plus simple ;
- puisque la compréhension est importante, nous définirons et ferons évoluer ensemble des métaphores ;
- puisque l'intégration des modifications est cruciale, nous l'effectuerons plusieurs fois par jour ;
- puisque les besoins évoluent vite, nous ferons des cycles de développement très rapides pour nous adapter au changement.

### 3.2. Cycle de développement XP



L'Extreme Programming repose sur des cycles rapides de développement (des itérations de quelques semaines) dont les étapes sont les suivantes :

- une phase d'exploration détermine les scénarios clients ("user stories" / mini "use case") qui seront à prendre en charge pendant cette itération ;
- l'équipe transforme les scénarios en tâches à réaliser et en tests fonctionnels ;
- chaque développeur s'attribue des tâches et les réalise avec un binôme ;
- lorsque tous les tests fonctionnels passent, le produit est livré.

Le cycle se répète tant que le client peut fournir des scénarios à implémenter. Généralement le cycle de la première livraison se caractérise par sa durée et le volume important de fonctionnalités embarquées. Après la première mise en production, les itérations peuvent devenir plus courtes (une semaine par exemple).

### 3.3. Valeurs de XP

L'eXtreme Programming repose sur cinq valeurs fondamentales :

- **La communication** : c'est le moyen fondamental pour éviter les problèmes.
- **La simplicité** : la façon la plus simple d'arriver au résultat est la meilleure. Anticiper les extensions futures est une perte de temps.
- **Le feed-back** : le retour d'information est primordial pour le programmeur et le client. Les tests unitaires indiquent si le code fonctionne. Les tests fonctionnels donnent l'avancement du projet. Les livraisons fréquentes permettent de tester les fonctionnalités rapidement.
- **Le courage** : certains changements demandent beaucoup de courage. Il faut parfois changer l'architecture d'un projet, jeter du code pour en produire un meilleur ou essayer une nouvelle technique.
- **Le respect**

### 3.4. Pratiques (extrêmes?) de XP

- **Client sur site** : un représentant du client doit, si possible, être présent pendant toute la durée du projet. Il doit avoir les connaissances de l'utilisateur final et avoir une vision globale du résultat à obtenir. Il réalise son travail habituel tout en étant disponible pour répondre aux questions de l'équipe.
- **Jeu du Planning** ou **Planning poker** : le client crée des scénarios pour les fonctionnalités qu'il souhaite obtenir. L'équipe évalue le temps nécessaire pour les implémenter. Le client sélectionne ensuite les scénarios en fonction des priorités et du temps disponible. On joue avec les plannings (réaffectation glissante des aspects secondaires et introduction surprise d'un nouvel élément fonctionnel dans le cahier des charges).
- **Intégration continue** : lorsqu'une tâche est terminée, les modifications sont immédiatement intégrées dans le produit complet. On évite ainsi la surcharge de travail liée à l'intégration de tous les éléments avant la livraison. Les tests facilitent grandement cette intégration : quand tous les tests passent, l'intégration est terminée.
- **Petites livraisons** : les livraisons doivent être les plus fréquentes possible. L'intégration continue et les tests réduisent considérablement le coût de livraison.
- **Rythme soutenable** : l'équipe ne fait pas d'heures supplémentaires. Si le cas se présente, il faut revoir le planning. Un développeur fatigué travaille mal.
- **Tests de recette (ou tests fonctionnels)** : À partir des scénarios définis par le client, l'équipe crée des procédures de test qui permettent de vérifier l'avancement du développement. Lorsque tous les tests fonctionnels passent, l'itération est terminée. Ces tests sont souvent automatisés mais ce n'est pas toujours possible.
- **Tests unitaires** : avant d'implémenter une fonctionnalité, le développeur écrit un test qui vérifiera que son programme se comporte comme prévu. Ce test sera conservé jusqu'à la fin du projet, tant que la fonctionnalité est requise. À chaque modification du code, on lance tous les tests écrits par tous les développeurs, et on sait immédiatement si quelque chose ne fonctionne plus.
- **Conception simple** : plus l'application est simple, plus il sera facile de la faire évoluer lors des prochaines itérations.
- **Utilisation de métaphores** : on utilise des métaphores et des analogies pour décrire le système et son fonctionnement. Le fonctionnel et le technique se comprennent beaucoup mieux lorsqu'ils sont d'accord sur les termes qu'ils emploient.
- **Refactoring (ou remaniement du code)** : amélioration régulière de la qualité du code sans en modifier le comportement. On retravaille le code pour repartir sur de meilleures bases tout en gardant les mêmes fonctionnalités.
- **Appropriation collective du code** : l'équipe est collectivement responsable de l'application. Chaque développeur peut faire des modifications dans toutes les portions du code, même celles qu'il n'a pas écrites. Les tests diront si quelque chose ne fonctionne plus.

- **Convention de nommage** : puisque tous les développeurs interviennent sur tout le code, il est indispensable d'établir et de respecter des normes de nommage pour les variables, méthodes, objets, classes, fichiers, etc.
- **Programmation en binôme** : la programmation se fait par deux. Le premier appelé *pilote* tient le clavier. C'est lui qui va travailler sur la portion de code à écrire. Le second appelé *partner* (ou *co-pilote*) est là pour l'aider en gardant un œil critique et en suggérant de nouvelles possibilités ou en décelant d'éventuels problèmes (correction des erreurs, avis différent, aide, ...). Les développeurs changent fréquemment de partenaire ce qui permet d'améliorer la connaissance collective de l'application et d'améliorer la communication au sein de l'équipe.

### 3.5. Autres pratiques extrêmes et variantes

Sur des petits projets, on travaille généralement en équipe réduite. Il faut alors savoir un peu tout faire. On se forge ainsi une bonne expérience où les aspects pragmatiques l'emportent sur les grands discours théoriques. L'apprenti dépasse le maître et devient virtuose. Commencent alors les pratiques extrêmes :

- on jongle avec les générateurs de code, le copier-coller et l'inspiration du moment.
- ...

Bien qu'assez extrêmes et pas toujours applicables, ces pratiques permettent quelquefois :

- de pouvoir aller très vite.
- d'être plus réactif
- ...

*citation (de S.G. ou ...): le tact dans l'audace c'est de savoir jusqu'où on peut aller trop loin.*

#### **Critiques et variantes/adaptations :**

Quand on connaît le coût (assez élevé) d'une journée de développement (salaire du développeur + charges sociales, ...), on peut se demander si le principe du travail en binôme est réellement applicable.

Le principe du travail en binôme est généralement judicieux que si l'est utilisé au bon moment (et pas systématiquement / tout le temps).

Lorsqu'il y a du "turn over" dans une équipe, le fait de travailler à deux permet de bien intégrer un nouveau développeur au sein de l'équipe existante :

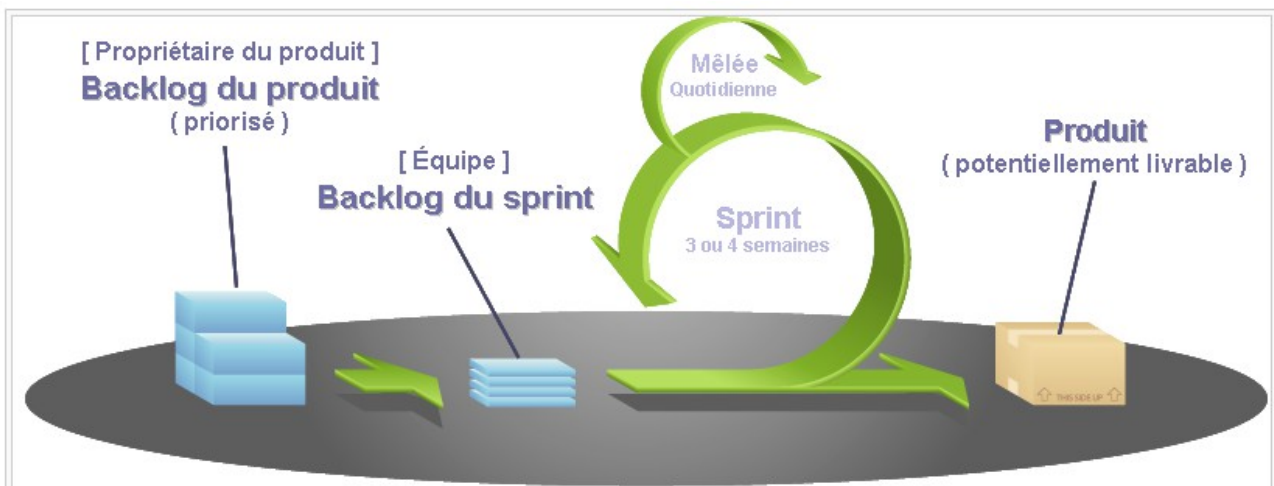
- les premiers jours, le nouvel arrivant observe (en tant que co-pilote) les manières de développer au niveau du projet (environnement de dev, convention de nommage, ...)
- et ensuite, c'est "tiens, prends le volant". Le nouvel arrivant code de son mieux et le développeur rodé au projet vérifie si c'est bien fait et préconise des ajustements aussitôt.

On peut éventuellement s'autoriser des variantes par rapport à ce qui a été rédigé/formalisé au sein de la méthode XP. Le principe "0" serait : ne pas appliquer systématiquement XP tel quel mais seulement ce qui semble utile dans la méthode XP.

## 4. Méthode agile SCRUM

"SCRUM" est une méthode agile pour la gestion de projet de développement de logiciels .

La métaphore de **Scrum** (mêlée du rugby) apparaît pour la première fois dans une publication de Takeuchi et Nonaka intitulée *The New New Product Development Game* qui s'appliquait à l'époque au monde industriel.



### 4.1. Principales caractéristiques de la méthode SCRUM

(selon wikipédia)

Le terme **Scrum** est emprunté au rugby à XV et signifie **mêlée**. Ce processus s'articule en effet autour d'une **équipe soudée, qui cherche à atteindre un but**, comme c'est le cas en rugby pour avancer avec le ballon pendant une mêlée.

**Le principe de base de Scrum** est de **focaliser l'équipe sur une partie limitée et maîtrisable des fonctionnalités à réaliser**. Ces **incréments** se réalisent successivement lors de périodes de durée fixe de une à quatre semaines, appelées **sprints**.

Chaque sprint possède, préalablement à son exécution, un **but** à atteindre, défini par le *directeur de produit*, à partir duquel sont choisies les fonctionnalités à implémenter dans cet incrément. Un sprint aboutit toujours à la livraison d'un produit partiel fonctionnel. Pendant ce temps, le *ScrumMaster* a la charge de minimiser les perturbations extérieures et de résoudre les problèmes non techniques de l'équipe.

Un principe fort en Scrum est la **participation active du client pour définir les priorités dans les fonctionnalités du logiciel et pour choisir celles qui seront réalisées dans chaque sprint**. Il peut à tout moment compléter ou modifier la liste des fonctionnalités à produire, mais jamais celles qui sont en cours de réalisation pendant un sprint. ...

### 4.2. Rôles des intervenants/participants dans la méthode SCRUM

Directeur de	Product	Représentant des clients et utilisateurs. C'est lui qui définit l'ordre
--------------	---------	---



<b>produit</b>	<i>Owner</i>	<p>dans lequel les fonctionnalités seront développées et qui prend les décisions importantes concernant l'orientation du projet. Le terme <i>directeur</i> n'est d'ailleurs pas à prendre au sens hiérarchique du terme, mais dans le sens de l'<i>orientation</i>.</p> <p>Dans l'idéal, le directeur de produit travaille dans la même pièce que l'équipe. Il est important qu'il reste très disponible pour répondre aux questions de l'équipe et pour lui donner son avis sur divers aspects du logiciel (interface utilisateur par exemple)</p>
<b>Equipe (de développement, auto gérée)</b>	<i>Team</i>	<p>Il n'y a pas de notion de hiérarchie interne : toutes les décisions sont prises ensemble et personne ne donne d'ordre à l'équipe sur sa façon de procéder. Contrairement à ce que l'on pourrait croire, les équipes auto-gérées sont celles qui sont les plus efficaces et qui produisent le meilleur niveau de qualité de façon spontanée.</p> <p>L'équipe s'adresse directement au directeur de produit. Il est conseillé qu'elle lui montre le plus souvent possible le logiciel développé pour qu'il puisse ajuster les détails.</p>
<b>Facilitateur / animateur</b>	<i>Scrum-Master</i>	<p>Chargé de protéger l'équipe de tous les éléments perturbateurs extérieurs et de résoudre ses problèmes non techniques (administratifs par exemple). Il doit aussi veiller à ce que les valeurs de Scrum soient appliquées, il est le garant de la méthode. En revanche, il n'est pas un chef de projet ni un intermédiaire de communication avec les clients.</p> <p>On parle parfois d'<b>équipe étendue</b>, qui intègre en plus le <i>ScrumMaster</i> et le directeur de produit. Ce concept renforce l'idée que client et fournisseur travaillent d'un commun effort vers le succès du projet.</p>
<b>Intervenants (externes)</b>	<i>Stakeholders</i>	<p>Personnes qui souhaitent avoir une vue sur le projet sans réellement s'investir dedans. Il peut s'agir par exemple d'experts techniques ou d'agents de direction qui souhaitent avoir une vue très éloignée de l'avancement du projet.</p>

### 4.3. Planification "SCRUM"

#### Releases et sprints

Scrum est un processus *itératif* : les itérations sont appelées des **sprints** et durent en théorie 30 jours calendaires. En pratique, les itérations durent généralement entre 2 et 4 semaines.

Chaque sprint possède un **but** et on lui associe une liste d'*items de backlog de produit* (fonctionnalités) à réaliser. Ces items sont décomposés par l'équipe en tâches élémentaires de quelques heures, les *items de backlog de sprint*. ...

Pour améliorer la lisibilité du projet, on regroupe généralement des itérations en **releases**.

Bien que ce concept ne fasse pas explicitement partie de Scrum, il est utilisé pour mieux identifier les versions. En effet, comme chaque sprint doit aboutir à la livraison d'un produit partiel, une release permet de marquer la livraison d'une version aboutie, susceptible d'être mise en exploitation.

**Réunion quotidienne :**

Au quotidien, une réunion, le **ScrumMeeting**, permet à l'équipe et au Scrum Master de *faire un point d'avancement sur les tâches et sur les difficultés rencontrées*.

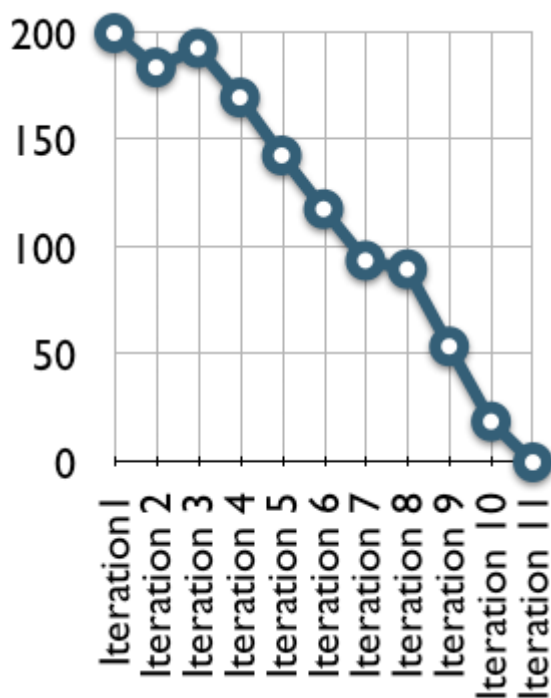
**4.4. Gestion des besoins/fonctionnalités****Backlog de produit :**

L'objectif est d'établir une liste de fonctionnalités à réaliser, que l'on appelle **backlog de produit** (NDT : Le terme *backlog* peut être traduit par *cahier*, *liste* ou *carnet de commandes*).

À chaque item de backlog sont associés deux attributs :

- une estimation en **points arbitraires**
- une valeur *client*, qui est définie par le directeur de produit (retour sur investissement par exemple).

La somme des points des items du backlog de produit constitue le *reste à faire* total du projet. Cela permet de produire un **release burndown chart**, qui montre les points restant à réaliser au fur et à mesure des sprints.

**Backlog de sprint :**

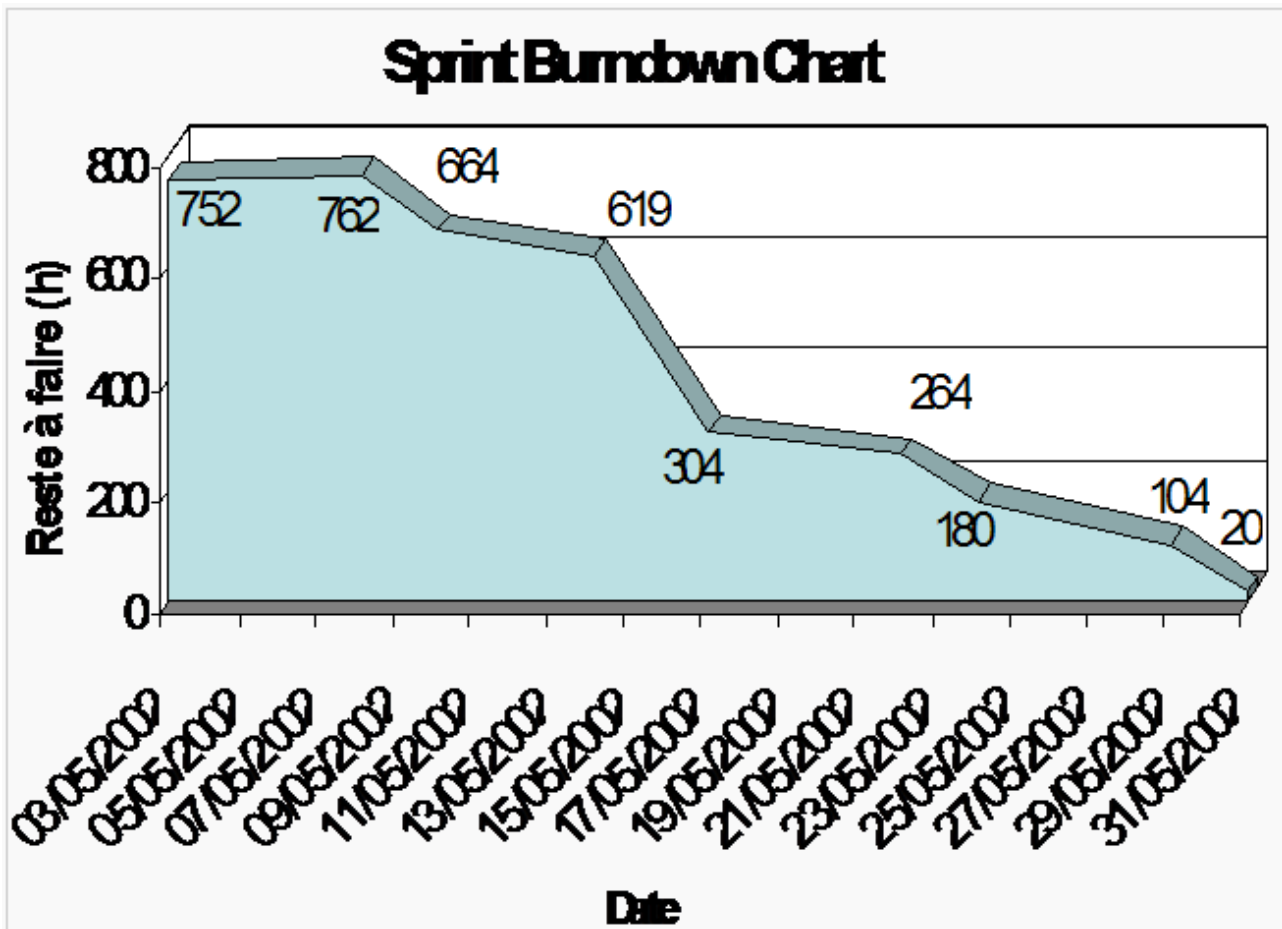
Lorsqu'on démarre un sprint, on choisit quels items du backlog de produit seront réalisés dans ce sprint. L'équipe décompose ensuite chaque item en liste de tâches élémentaires (techniques ou non), chaque tâche étant estimée en heures et ne devant pas durer plus de 2 jours. On constitue ainsi le **backlog de sprint**.

Pendant le déroulement du sprint, chaque équipier s'affecte des tâches du backlog de sprint et les réalise. Il met à jour régulièrement dans le backlog du sprint le reste à faire de chaque tâche. Les



tâches ne sont pas réparties initialement entre tous les équipiers, elles sont prises au fur et à mesure que les précédentes sont terminées.

La somme des heures des items du backlog de sprint constitue le *reste à faire* total du sprint. Cela permet de produire un **sprint burndown chart** qui montre les heures restantes à réaliser au fur et à mesure du sprint.



#### 4.5. Éléments pour estimations

Scrum ne définit pas spécialement d'unités pour les items des backlogs. Néanmoins, certaines techniques se sont imposées de fait.

##### **Items de backlog de produit = user stories = use case UML**

Les items de backlog de produit sont souvent des *User Stories* empruntées à [Extreme Programming](#). Ces User Stories sont estimées en *points relatifs*, sans unité. L'équipe prend un item représentatif et lui affecte un nombre de points arbitraire. Cela devient un référentiel pour estimer les autres items. Par exemple, un item qui vaut 2 points représente deux fois plus de travail qu'un item qui en vaut 1. Pour les valeurs, on utilise souvent les premières valeurs de la [suite de Fibonacci](#) (1,2,3,5,8,13), qui évitent les difficultés entre valeurs proches (8 et 9 par exemple).

L'intérêt de cette démarche est d'avoir une idée du travail requis pour réaliser chaque fonctionnalité sans pour autant lui donner une valeur en jours que le directeur de produit serait tenté de considérer comme définitivement acquise. En revanche, on utilise la *vélocité* pour planifier le projet à l'échelle macroscopique de façon fiable et précise.

##### **Calcul de vélocité :**

Une fois que tous les items de backlog de produit ont été estimés, on attribue un certain nombre d'items à réaliser aux sprints successifs. Ainsi, une fois un sprint terminé, on sait combien de points

ont été réalisés et on définit alors la **vélocité** de l'équipe, c'est-à-dire le nombre de points qu'elle peut réaliser en un sprint.

En partant de cette vélocité et du total de points à réaliser, on peut déterminer le nombre de sprints qui seront nécessaires pour terminer le projet (ou la release en cours). L'intérêt, c'est qu'on a une vision de plus en plus fiable (retours d'expérience de sprint en sprint) de la date d'aboutissement du projet, tout en permettant d'aménager les items de backlog du produit en cours de route.

#### **Items de backlog de sprint=tâche (quelques heures) :**

Les items de backlog de sprint sont généralement exprimés en heures et ne doivent pas dépasser 2 journées de travail, sinon il convient de les décomposer en plusieurs items. Par abus de langage, on emploie le terme de *tâches*, les concepts étant très proches.

## 4.6. Déroulement d'un sprint

### **Réunion de planification :**

Tout le monde est présent à cette réunion, qui ne doit pas durer plus de 4 heures. La **réunion de planification** (*Sprint Planning*) consiste à définir d'abord un but pour le sprint, puis à choisir les items de backlog de produit qui seront réalisés dans ce sprint. Cette première partie du *sprint planning* représente l'engagement de l'équipe. Compte tenu des conditions de succès énoncées par le directeur de produit et de ses connaissances techniques, l'équipe s'engage à réaliser un ensemble d'items du backlog de produit.

Dans un second temps, l'équipe décompose chaque item du backlog de produit en liste de tâches (items du backlog du sprint), puis estime chaque tâche en heures. Il est important que le directeur de produit soit présent dans cette étape, il est possible qu'il y ait des tâches le concernant (comme la rédaction des règles métier que le logiciel devra respecter et la définition des tests fonctionnels).

### **Au quotidien :**

Chaque journée de travail commence par une réunion de 15 minutes maximum appelée **mêlée quotidienne** (*Daily Scrum*). Seuls l'équipe, le directeur de produit et le ScrumMaster peuvent parler, tous les autres peuvent écouter mais pas intervenir (leur présence n'est pas obligatoire). A tour de rôle, chaque membre répond à 3 questions :

- *Qu'est-ce que j'ai fait hier ?*
- *Qu'est-ce que je compte faire aujourd'hui ?*
- *Quelles sont les difficultés que je rencontre ?*

L'équipe se met ensuite au travail. Elle travaille dans une même pièce, dont le ScrumMaster a la responsabilité de maintenir la qualité d'environnement. Les activités se déroulent éventuellement en parallèle : analyse, conception, codage, intégration, tests, etc. Scrum **ne définit volontairement pas** de démarche technique pour le développement du logiciel : l'équipe s'auto-gère et décide en toute autonomie de la façon dont elle va travailler.

Remarque : Il est assez fréquent que les équipes utilisent la démarche de *développement guidé par les tests* (Test Driven Development en anglais). Cela consiste à coder en premier lieu les modules de test vérifiant les contraintes métier, puis à coder ensuite le logiciel à proprement parler, en exécutant les tests régulièrement. Cela permet de s'assurer entre autres de la non-régression du logiciel au fil des sprints.

### **Revue de sprint :**

À la fin du sprint, tout le monde se réunit pour effectuer la **revue de sprint**, qui dure au maximum 4 heures. L'objectif de la revue de sprint est de valider le logiciel qui a été produit pendant le sprint. L'équipe commence par énoncer les items du backlog de produit qu'elle a réalisés. Elle effectue

ensuite une démonstration du logiciel produit. C'est sur la base de cette démonstration que le directeur de produit valide chaque fonctionnalité planifiée pour ce sprint.

Une fois le bilan du sprint réalisé, l'équipe et le directeur de produit proposent des aménagements sur le backlog du produit et sur la planification provisoire de la release. Il est probable qu'à ce moment des items soient ajoutés, modifiés ou réestimés, en conséquence de ce qui a été découvert.

#### **Rétrospective du sprint :**

La **rétrospective du sprint** est faite en interne à l'équipe (incluant le ScrumMaster). L'objectif est de comprendre ce qui n'a pas bien marché dans le sprint, les erreurs commises et de prendre des décisions pour s'améliorer. Il est tout à fait possible d'apporter des aménagements à la méthode Scrum dans le but de s'améliorer.

## **4.7. Compléments/approfondissements (pour SCRUM)**

#### **Lancement du projet :**

Scrum présuppose que le backlog de produit est déjà défini au début du projet. Une approche possible pour constituer ce backlog est de réaliser une **phase de lancement**. Cette phase de lancement s'articule autour de deux axes de réflexion :

- l'étude d'opportunité
- l'expression initiale des besoins.

#### **Documentation de projet :**

Produire de la documentation est souvent utile mais aussi souvent inutile. En plus, il faut la maintenir à jour, quelque chose qui est rarement fait sur place. Pour savoir s'il faut rédiger un document, on peut se poser une question très simple : **Est-ce que ce document va m'être vraiment utile et tout de suite ?**

Voici quelques exemples de documents utiles et dans quels cas :

- diagrammes métiers (processus, objets, etc.), associé au backlog de produit : uniquement si la logique métier du client qui concerne l'application est vraiment complexe. Dans ce cas, l'équipe devrait produire ce document avec lui ;
- diagramme de séquence, associé à un item du backlog du produit : uniquement si la fonctionnalité aura une utilisation complexe, tant au niveau métier qu'applicatif ;
- diagrammes d'architecture du logiciel (classes, modules, composants, etc.), pour le projet : indispensable pour avoir toujours sous les yeux une vue de l'architecture et s'assurer ainsi qu'elle est de qualité ;
- les manuels utilisateur, à chaque sprint : les manuels sont produits à chaque sprint et pas en fin de projet. Utiliser des vidéos de démonstrations commentées est une solution efficace

Bref, un document ne doit être produit que si **son utilité est réelle et immédiate**.

# XXI - OCL (Object Constraint Language)

## 1. OCL (Object Constraint Language) - Présentation

Le mini langage "OCL" sert à **exprimer des contraintes (orientées objets)** dans le cadre d'une modélisation UML.

OCL existe depuis très longtemps (dès les versions 1.x d'UML).

Cependant la plupart des anciens outils UML ne géraient pas bien OCL (pas d'affichage ou ...).

Lorsque l'on utilise UML pour simplement produire des spécifications qui seront imprimées et ré-interprétées visuellement, OCL n'apporte pas grand chose de plus par rapport à une formulation libre des contraintes.

Par contre, lorsqu'un modèle UML est ré-interprété par un outil spécialisé (ex: générateur de code MDA), OCL permet de mieux exprimer les contraintes (en apportant plus de rigueur et de précision).

Aujourd'hui OCL est essentiellement utilisé pour:

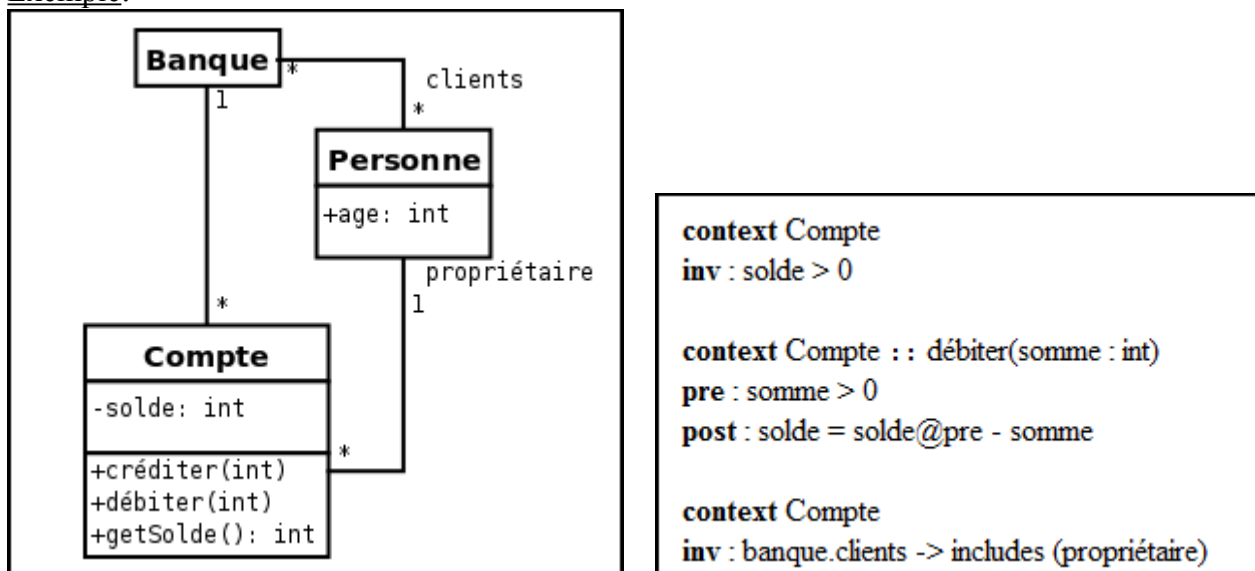
- exprimer des contraintes dans un modèle UML
- exprimer des conditions (gardiens) au niveau des transitions des diagrammes d'états
- exprimer des requêtes portant sur une partie du modèle .  
ces requêtes sont (entre autre) utilisable au sein des templates accéléo (générateur MDA)

Point clef pour comprendre la syntaxe OCL:

Les contraintes **OCL** sont très souvent exprimées en **relatif** par rapport à l'objet courant.

L'**objet courant** est référencé par le mot clef "**self**" (équivalent au mot clef "**this**" des langages "C++" et "Java").

Exemple:



Quelques URL pour approfondir:

<http://laurent-audibert.developpez.com/Cours-UML/html/Cours-UML021.html>

<http://www.omg.org/spec/OCL/2.2/PDF/>

# XXII - Essentiel outil "Papyrus UML"

## 1. Utilisation de Papyrus\_UML (éditeur UML2 eclipse)

(MDT) **Papyrus (UML)** est un **éditeur UML open source** qui fonctionne sous forme de *plugin pour l'environnement de développement **ECLIPSE*** (très utilisé pour le développement d'applications en JAVA). Le tout étant basé sur JAVA, **Papyrus\_UML** est "**multi-plateforme**" et fonctionne entre autres sur **Windows** , **Unix/Linux** , Mac .

Historiquement, les premières versions de Papyrus ont été conçues par le **CEA**.

Un projet similaire "**Topcased UML**" avait été pris en charge par "**Airbus + écoles et universités proches de Toulouse**". En 2011/2012/2013/2014, ces deux produits sont actuellement en train de fusionner (en même temps que s'effectue une reprise officielle de la partie "editeur UML papyrus" par la communauté "eclipse").

En tant que "(sous) projet eclipse officiel" , l'éditeur "**Papyrus UML**" peut s'intégrer (par simple ajout) dans le tout dernier **eclipse** (ex : 4.3 / Kepler en 2013/2014).

En tant que "sous partie" des versions récentes de "**Topcased UML RCP**" , "Papyrus UML" peut être rapidement utilisé dans **un environnement tout intégré** (avec générateur de code java et de documentation).

*L'un des principaux intérêts des outils "Topcased UML" et "Papyrus UML" tient dans le fait qu'en tant que "plugins bien intégrés à eclipse" , on peut les utiliser conjointement avec d'autres plugins importants (ex: accéléo\_M2T , gendoc2, ...).*

Ainsi à partir d'un même outil ECLIPSE, on peut:

- **créer/paramétrer des modèles UML** (diagrammes avec stéréotypes)
- **(re-)générer automatiquement de la documentation au format ".doc" ou ".odt"** (avec le plugin intégré gendoc2) pour produire des spécifications
- **(re-)générer automatiquement une bonne partie du code de l'application** (avec le plugin "accéléo\_M2T" / MDA).

Ceci permet de travailler efficacement avec de bon atouts pour obtenir des éléments produits (spécifications , code , tests) bien **cohérents** entre eux.

**NB:** Pour bien utiliser cette plateforme de développement (et ses différents plugins) , il faut savoir effectuer les **bons paramétrages** à chaque niveau:

- bien paramétrer les modèles UML (avec des stéréotypes adéquats)
- bien paramétrer la génération de documentation (avec des "templates" de "docs")
- bien paramétrer la génération de code (avec des "templates" de code)

Tout ceci correspond au final à **un investissement en "temps de mise au point"** qui peut se rentabiliser sur des projets importants.

## 1.1. Intégration/installation de Papyrus UML

Préalable : une machine virtuelle JAVA (idéalement JDK 1.6 ou 1,7) doit être installée sur le poste de développement.

Il y a au final deux grands modes d'intégration de "MDT Papyrus UML":

- intégration pré-établie (prête à être téléchargée)  
---> **Topcased RCP** (*Rich Client Platform*)
- intégration spécifique (à construire soit même en partant d'ECLIPSE et en y ajoutant un à un tous les plugins jugés utiles : Papyrus\_UML , accéléo\_M2T , gendoc2 , ... )

NB:

En 2013, pour la version "Eclipse 4.3 / Kepler" , le mode opératoire pour installer le plugin "MDT Papyrus" est le suivant :

**Menu Help / Software Update , ...**

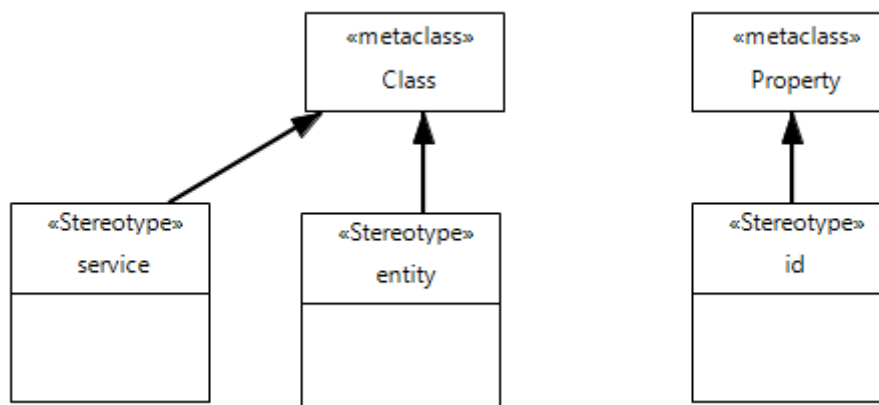
releases kepler / **Modeling** / (modeling components) / **papyrus**

## 1.2. Création d'un profil UML (avec stéréotypes)

Pour placer des stéréotypes (ex: <<entity>> , <<id>>) dans un modèle UML , il faut d'abord les créer dans un fichier de type "**UML profile**".

Mode opératoire:

- **New / Other ... / Papyrus / Papyrus model**
- **Select "Profile" et "UML profile diagram" (+include template "primitives types").**
- Créer de nouveaux **stéréotypes** (depuis la palette et en renseignant leurs **noms**).
- Placer et paramétrer des "**metaclass**" (ex: "Class" , "Property" , ... ) .
- Relier les "**stéréotypes**" aux "**metaclass**" via des flèches d'**extension** pour indiquer "**sur quoi les stéréotypes seront applicables**".
- Bien sauvegarder le fichier généré (et choisir une **version**).



NB: un stéréotype peut éventuellement comporter des propriétés.

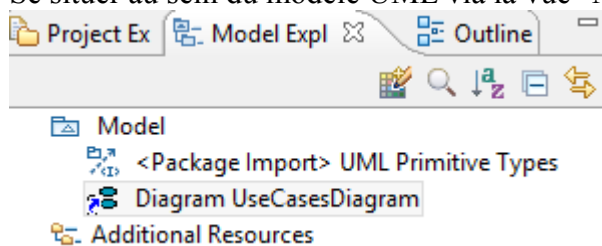
Quelques idées de stéréotypes:

<<entity>>	Entité persistante
<<id>>	Identifiant (proche de "clef primaire")
<<service>>	Service métier
<<requestScope>> , <<sessionScope>> , <<applicationScope>>	Scope / portée des objets de la partie "IHM_WEB"
<<stateful>> , <<stateless>>	Avec ou sans état (traitements ré-entrants et partagés ?)
<<facade>> , <<dao>> , <<dto>> , ...	Design pattern / pour la conception
<<in>> , <<out>> , <<inout>> , <<select>>	Pour paramétrer les fonctionnalités souhaitées coté IHM (entrée/saisie , sortie/affichage , sélection , ...)
<<transactional>> , <<CRUD>>	Fonctionnalités diverses attendues (transactionnel , ...)
<<module_web>> , <<module_services>> , <<database>>	Types de composants
...	
...	

### 1.3. Création et initialisation d'un modèle UML (pour application)

Mode opératoire:

- Créer un nouveau modèle via le menu "New / Other.../ Papyrus/ Papyrus Model"
- Sélectionner "UML" et (include template "primitives types")
- Sélectionner éventuellement le nom et le type de diagramme initial (ex : UsesCases ou "Class") (NB : d'autres diagrammes pourront être ajoutés ultérieurement au modèle).
- Se situer au sein du modèle UML via la vue "Model explorer" :



- Sélectionner la racine du modèle UML (Model) et ajuster la propriété "profiles" en "cliquant" sur "+" et en sélectionnant le fichier ".....profile.uml" .
- ...
- Bien (re-)sauvegarder le fichier du modèle UML.

Remarque :

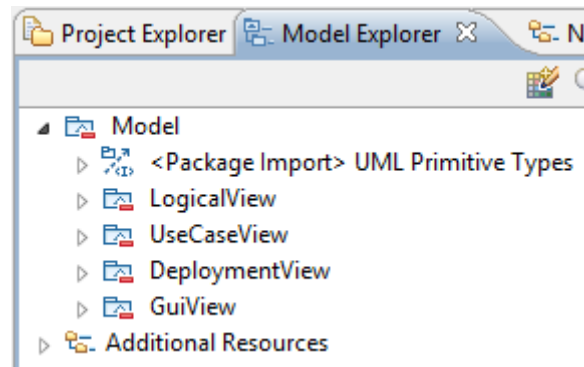
L'assistant de création de modèle "uml papyrus" ne crée pour l'instant qu'un point de départ très rudimentaire (rien ou un diagramme au choix à la racine du modèle).

Si l'on souhaite mieux organiser la structure interne d'un modèle papyrus UML , on peut éventuellement se placer à la racine "Model" et créer (via le menu contextuel "add Child / new Model") des sous modèles de type "UseCaseView" , "LogicalView" de façon à mieux ranger les



parties du modèle applicatif.

Exemple :

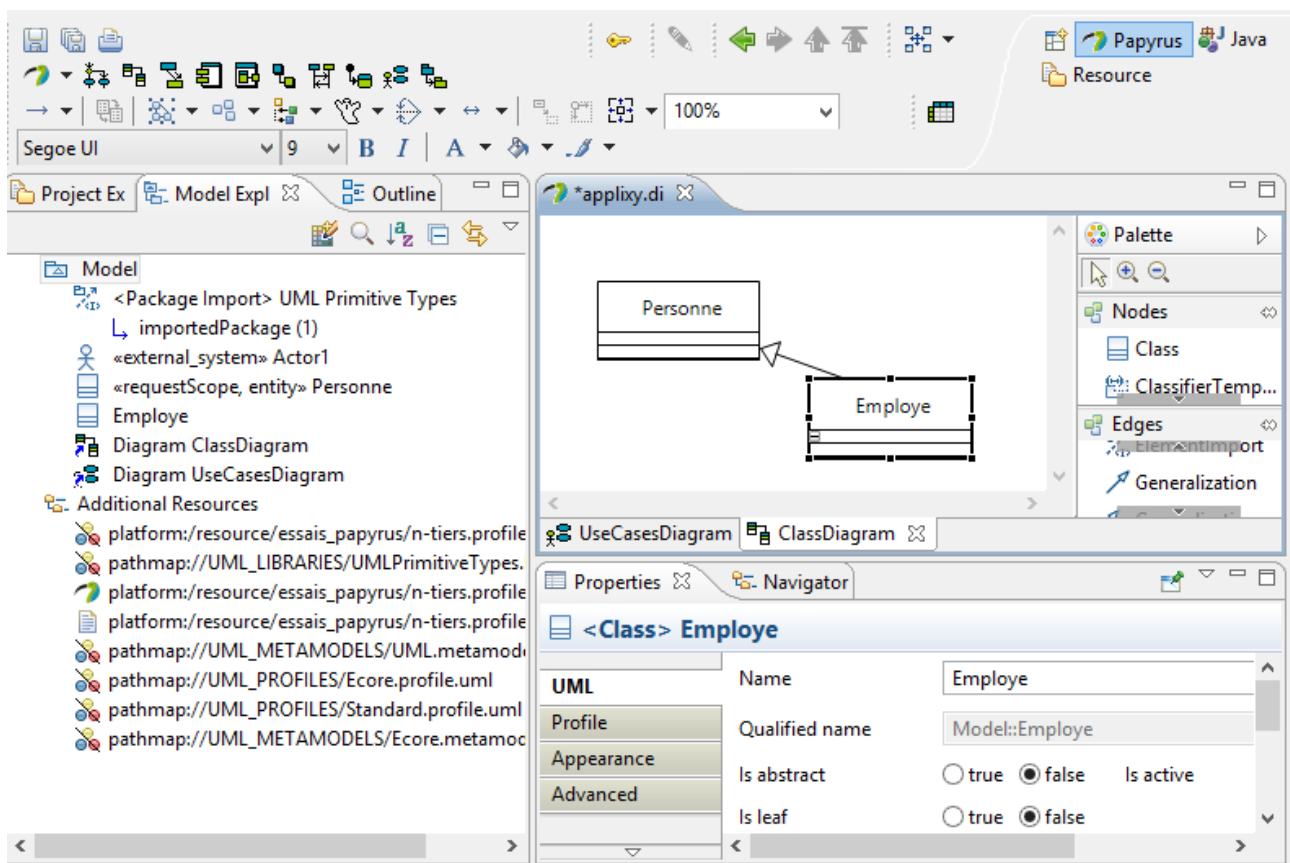


## 1.4. Utilisation générale de l'outil Papyrus UML

**Chaque modèle UML est sauvegardé dans trois fichiers complémentaires:**

- le fichier **".uml"** comporte tous les éléments significatifs du modèle UML (packages, classes, ....). C'est à partir de ce fichier que l'on peut extraire les informations essentielles du modèle UML pour générer du code (via un outil MDA tel qu'accéléo\_m2t par exemple)
- les fichiers **".notation"** et **".di"** comportent les coordonnées (x,y,...) des éléments internes des diagrammes UML.

==> pour éditer graphiquement un modèle UML, il faut ouvrir (via un double-clic) le fichier **".di"** (ou l'ensemble). Le fichier **".uml"** de même nom sera alors automatiquement pris en charge et mis à jour.





De façon intuitive, la vue "Model Explorer" permet de naviguer dans l'arborescence interne du modèle UML et la vue "Properties" permet de fixer les propriétés de l'élément sélectionné. Une palette permet de choisir le type d'élément à ajouter au modèle.

- ~~pas de menu contextuel "add attribute/property" ni "add operation/method"~~ mais des éléments "**Property**" et "**Operation**" à récupérer dans la palette et à ajouter aux classes du diagramme.
- Souvent besoin de cacher (via **Filter/hide compartment** ou **Filter / ....**) certains éléments secondaires.

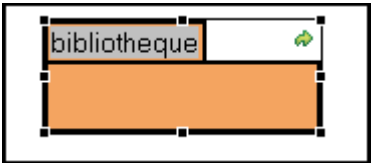
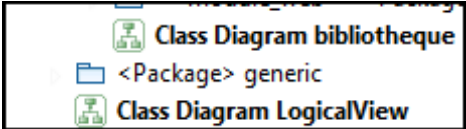
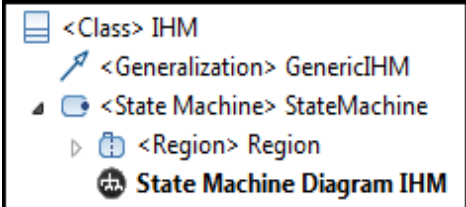
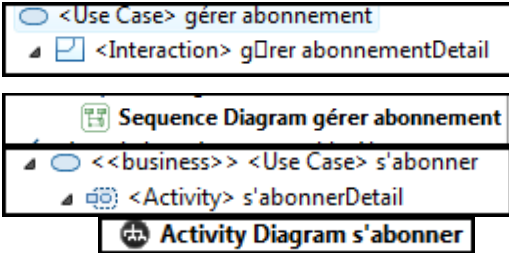
## 1.5. Organisation conseillée des packages et des diagrammes



L'un des principaux "points forts" des outils "Topcased UML" et "Papyrus UML" c'est de bien gérer la **cohérence** entre les **packages** et les **diagrammes** de façon à pouvoir naviguer de façon efficace dans l'arborescence d'un modèle UML.

Mode opératoire conseillé:

- Créer un élément UML (ex: package, classe, ...) dans un diagramme et le paramétrer.
- **Sélectionner un package existant (modélisé auparavant)** dans l'arborescence "**model explorer**", puis activer le menu "**add class diagram**" en donnant un nom explicite à ce diagramme (ex : packageXY\_ClassDiagram), etc .... / etc ....
- Par la suite (une fois la création/association effectuée), un **double-clic** ultérieur sur le package au sein du diagramme permettra de **naviguer d'un niveau vers un sous niveau** (de détails).

Sous diagrammes classiques pour indiquer les détails:

Eléments du modèle UML	diagramme(s) classique(s) associé(s) pour les détails	Exemple(s)
<b>Package</b>	Diagramme de classes (ou ...)	 
<b>Classe</b>	Diagramme d'états (state machine) ou diag. de structure composite	
<b>Use Case</b>	diagramme d'activités et/ou diagrammes de séquences	

<b>Composant</b>	Diagramme de (sous)composants ou de structure composite , ...	 <Component> module_services_XY  Component Diagram module_services_XY
------------------	---	--








Remarques importantes:

- Pour bien organiser un modèle UML, il faut réfléchir le plus tôt possible à la décomposition en différents packages.
- Il est toujours possible de renommer ou déplacer à la souris un package (après coup / par la suite) via le "model\_explorer".
- Cette "bonne organisation" des éléments du modèle UML est surtout utile pour que des scripts des templates (.docx/.odt) pour gendoc2 puissent efficacement retrouver les diagrammes de façon à générer automatiquement une bonne documentation.

**1.6. Edition d'un diagramme de classes (spécificités)**

Après avoir sélectionner une association , on peut activer le menu contextuel "Filters/All-No-Managed connectors labels" ce qui fait apparaître la boîte de dialogue suivante :

Select the labels to display.

Label Role	Displayed Text
<input checked="" type="checkbox"/>  <Association> class1_class2_1	
<input type="checkbox"/>  Name	class1_class2_1
<input type="checkbox"/>  0..1 SourceMultiplicity	
<input type="checkbox"/>  SourceRole	+ class1
<input type="checkbox"/>  Stereotype	[No Text To Display]
<input type="checkbox"/>  0..1 TargetMultiplicity	
<input type="checkbox"/>  TargetRole	+ class2

Ceci est très pratique pour afficher ou cacher certains détails associés aux associations (rôles , ....)

Dans la fenêtre des propriétés:

**\*Paramétrer les détails d'une d'association (navigabilité, agrégation, composition, ....)**

Member End

Name:

Owner:

Navigable: ☐ true ☒ false

Aggregation:

Multiplicity:

Member End

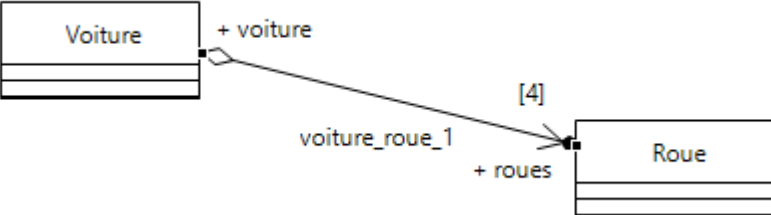
Name:

Owner:

Navigable: ☒ true ☐ false

Aggregation:

Multiplicity:

**\* Paramétrer une opération (nom, type de retour, paramètres , ....)**

Name	rechercherExemplaireParNumero		
Is abstract	<input type="radio"/> true <input checked="" type="radio"/> false	Is leaf	<input type="radio"/> true <input checked="" type="radio"/> false
Is query	<input type="radio"/> true <input checked="" type="radio"/> false	Is static	<input type="radio"/> true <input checked="" type="radio"/> false
Concurrency	sequential	Visibility	public
Method	<div> </div>		
		Owned parameter	<div> </div>
			<> <Parameter> num : Integer

Create a new Parameter

Name	num		
Is exception	<input type="radio"/> true <input checked="" type="radio"/> false	Is ordered	
Is stream	<input type="radio"/> true <input checked="" type="radio"/> false	Is unique	
Direction	in	Effect	
Visibility	public		
Default value	<Undefined>	Multiplicity	
Type	<Prim...teger>		

UML Profile

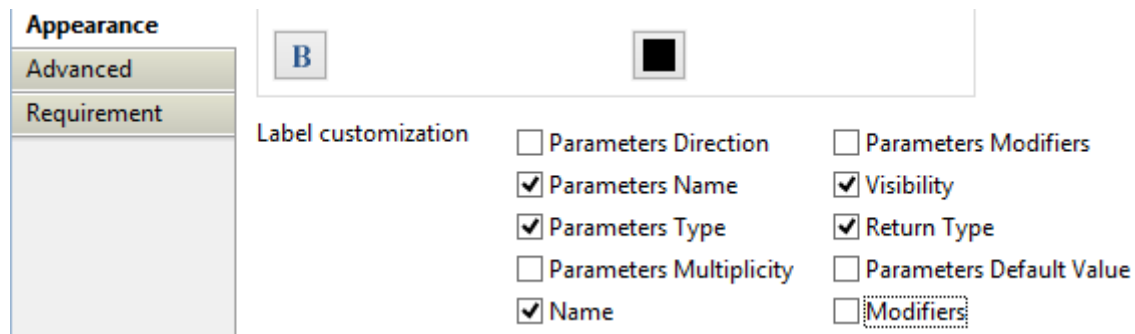
pour un paramètre d'entrée , laisser direction="in" , **pour la valeur de retour , direction = return**

Create a new Parameter

Name	return		
Is exception	<input type="radio"/> true <input checked="" type="radio"/> false		
Is stream	<input type="radio"/> true <input checked="" type="radio"/> false		
Direction	return		
Visibility	public		
Default value	<Undefined>		
Type	<<entity>> <Class> Exemple		

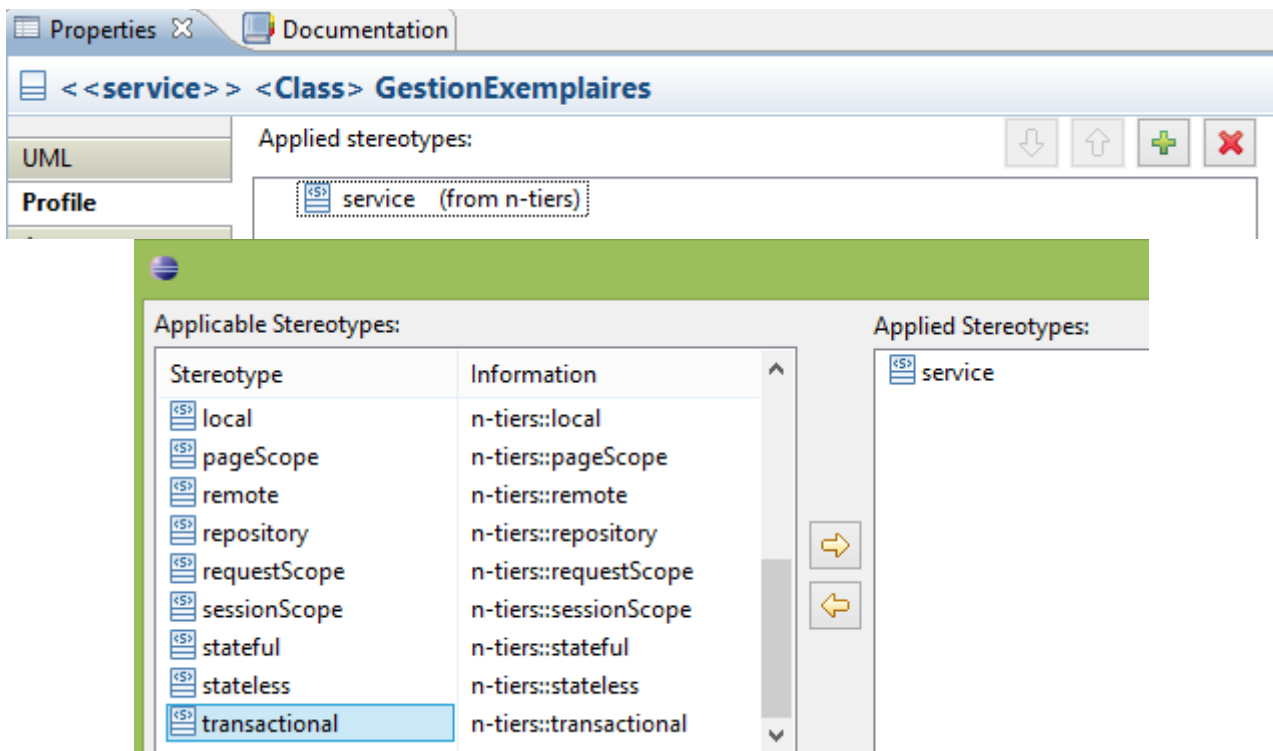
résultat → `rechercherExemplaireParNumero(num: Integer): Exemple`

et (dans le sous onglet "Appearance"), décocher "param direction", "visibility" et "modifiers":



**\* Choix d'un (ou plusieurs) stéréotype(s) à appliquer:**

Sélectionner un des éléments du modèle (classe ou propriété ou package ou ....)  
et sélectionner un stéréotype via le sous onglet "profile" :

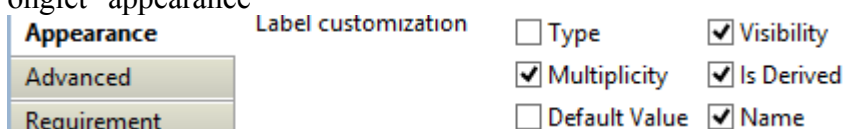


**NB :** Pour que certains stéréotypes applicables soient proposés, il faut qu'au préalable au moins un profile UML (autre fichier ".uml" comportant un paquet de stéréotypes) ait été associé à la racine du modèle via le sous onglet "profile" des propriétés.

**\* sélectionner un type de données pour une propriété ou le cacher :**

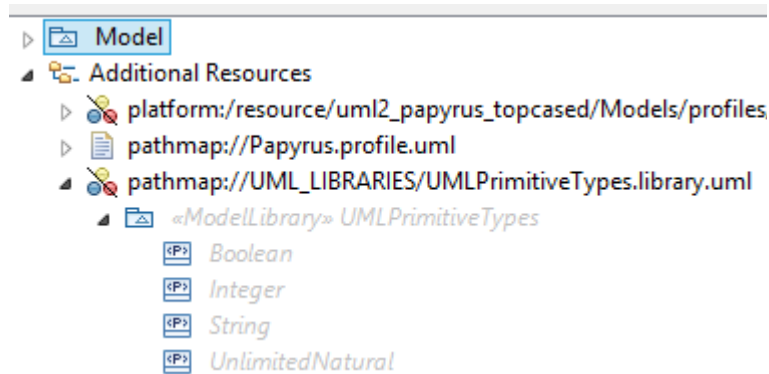
Sélectionner une propriété de la classe,

Si l'on souhaite (en analyse) cacher le type encore indéfini, il faut alors décocher "type" dans le sous onglet "appearance"

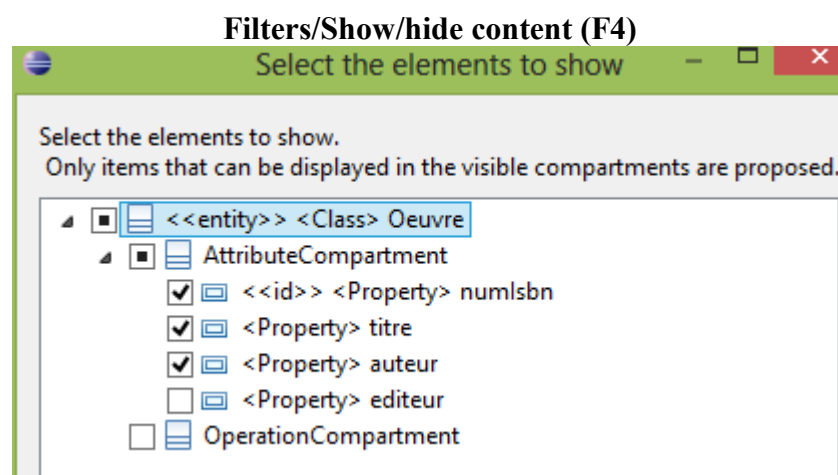


Si l'on souhaite (en conception détaillée), préciser un type de données parmi les types primitifs

prédéfinis d'UML, il faut depuis le sous onglet "UML", le choisir via "..." en face "type :"



\* Montrer ou cacher **Graphiquement** les différents éléments (propriétés/opérations) d'une classe:



\* spécifier une **agrégation** ou une **composition** d'un coté d'une **association** :

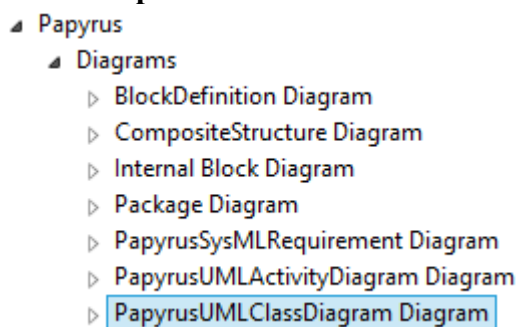
Aggregation  → pour obtenir un losange blanc sur l'extrémité inverse .  
 Aggregation  → pour obtenir un losange noir sur l'extrémité inverse .

\* spécifier une **classe d'association**

→ d'abord placer une association ordinaire et une classe ordinaire , relier ensuite par une liaison de type "AssociationClass" dans la sous palette "Edge" en partant de l'association et en pointant vers la classe . [Bug mi-2013 : les pointillés disparaissent lorsque l'on ferme et ré-ouvre le diagramme ]

## 1.7. Préférences/options sur l'éditeur Papyrus UML

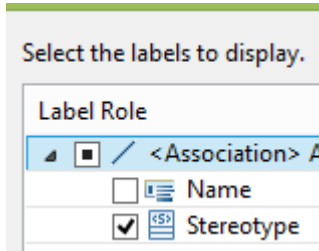
Windows/preferences/



## 1.8. Edition d'un diagramme de Use Cases (spécificités)

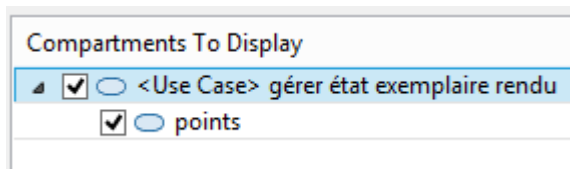
\* **Montrer ou cacher les différents éléments d'une association sélectionnée:**

*Filters / No Connector Label ou / Managed Connector Label*



\* **Montrer ou cacher un point d'extension (lié à un <<extend>>) :**

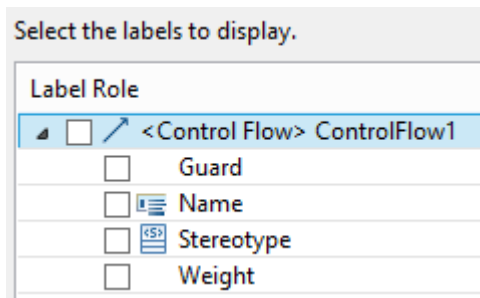
*Filters/ Show/hide compartments*



## 1.9. Edition d'un diagramme d'activités (spécificités)

\* **Montrer ou cacher les détails d'une liaison (controlFlow) :**

*Filters / No Connector Label ou / Managed Connector Label*



\* **Paramétrage des pins (output et input) autour d'un flot d'objet(s) :**

Placer un "**object flow**" entre deux actions via la palette.

Au moment de l'établissement de la liaison, la boîte de dialogue suivante apparaît alors pour paramétrer le nom et/ou le type des objets (données/document) qui seront véhiculés d'une activité à l'autre. A partir de ce paramétrage, l'éditeur "papyrus" va automatiquement construire des "pins" de même nature de chaque côté de la liaison.

Please fill information for pins creation

Pins initialization ?

Name: data

Type: [ ] [ ]

\* **Précautions à prendre pour bien paramétrer les liaisons d'entrées et de sorties** autour d'un "losange de décision" ou d'un "fork" :

→ De façon à contourner certains bugs temporaires de papyrus, il vaut mieux bien définir la (ou les) entrée(s) avant de définir la (ou les) sortie(s).

\* **Paramétrage d'un "call behavior action" pour naviguer d'un diagramme d'activité à un sous autre :**

Create a new Call Behavior Action

Create a new Behavior ?

☐ Create behavior

Behavior type: Activity -> Behavior

Name: Activity1

Element owner: <Activity> calcul , confirmation et paiement de l'acompte [ ]

Or assign an existing one

☒ Select behavior

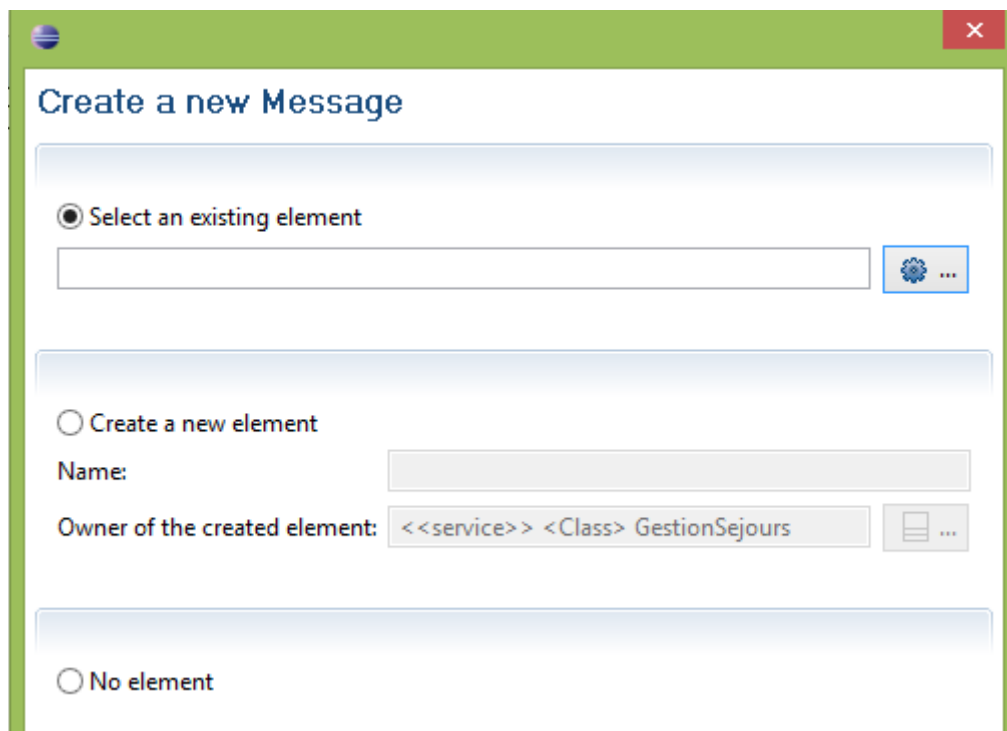
Behavior: [ ] [ ]

NB : de façon à bien contrôler la position du sous diagramme dans la hiérarchie du "model explorer" , on peut soit pré-crée le sous diagramme pour le sélectionner ensuite , soit bien paramétrer l'élément propriétaire ("owner") lors d'une création à la volée.

## 1.10. Edition d'un diagramme de séquences (spécificités)

Mode opératoire:

- créer un diagramme de séquence (idéalement en tant que détail d'un "use case")
- placer des **lignes de vie ("LifeLine")** et préciser les **types représentés** (acteurs, classes, ...) en effectuant des "glisser/poser" d'un élément (acteur ou classe) du "model explorer" vers l'entête du "LifeLine" et confirmer l'opération via un click sur "set represent ...".
- placer des **blocs d'exécution** sur les lignes de vie
- placer des **messages** entre un bloc d'exécution et un autre
- **paramétrer** les messages (*saisir un nom ou bien sélectionner une opération disponible au niveau de type d'objet qui reçoit le message*)



Remarque importante :











Lorsque (via l'option "create new element" de cette boîte de dialogue) l'on crée de nouvelles méthodes/opérations dans la classe de l'objet qui reçoit le message, celles-ci sont présentes dans le "model\_explorer" mais n'apparaissent pas automatiquement dans les diagrammes de classes. Pour faire graphiquement apparaître les nouvelles opérations dans les classes d'un diagramme de classes, il faut activer le menu contextuel "filters/ show/hide contents" et sélectionner les nouvelles méthodes (supplémentaires) à afficher.

NB: On peut également placer des fragments combinés (avec mot clef "alt", "opt", "loop", ...) d'UML2.



## 1.11. Edition d'un diagramme d'états (spécificités)

\* Paramétrage interne d'un état (do , exit , entry ) :

State invariant	<Undefined>	  	Entry	<Undefined>
Do activity	 <Opaque Behavior> choix période, transport	  	Exit	<Undefined>
Submachine	<Undefined>	  		

## 1.12. Edition d'un diagramme de composants (spécificités)

RAS

## 1.13. Edition d'un diagramme de déploiement (spécificités)

RAS

## 1.14. Génération de code java (via le générateur par défaut de Topcased)

- Ouvrir (si besoin) la vue "Navigator"
- Sélectionner le fichier appXY.uml et activer le menu contextuel "Code Generator / generate ....Java".

→ le code généré apparaît alors dans le projet courant .

## 2. Génération de documentation (gendoc2)

**Gendoc2** est un **plugin eclipse** permettant de *générer de la documentation* (au format **".docx"** de word ou bien **".odt"** de OpenOffice ) *à partir des informations extraites dans un modèle UML* (".uml" et ".notation" , ".di" ).

NB: à partir des formats ".odt" ou ".docx" , il est assez facile de **générer une version ".pdf"** .

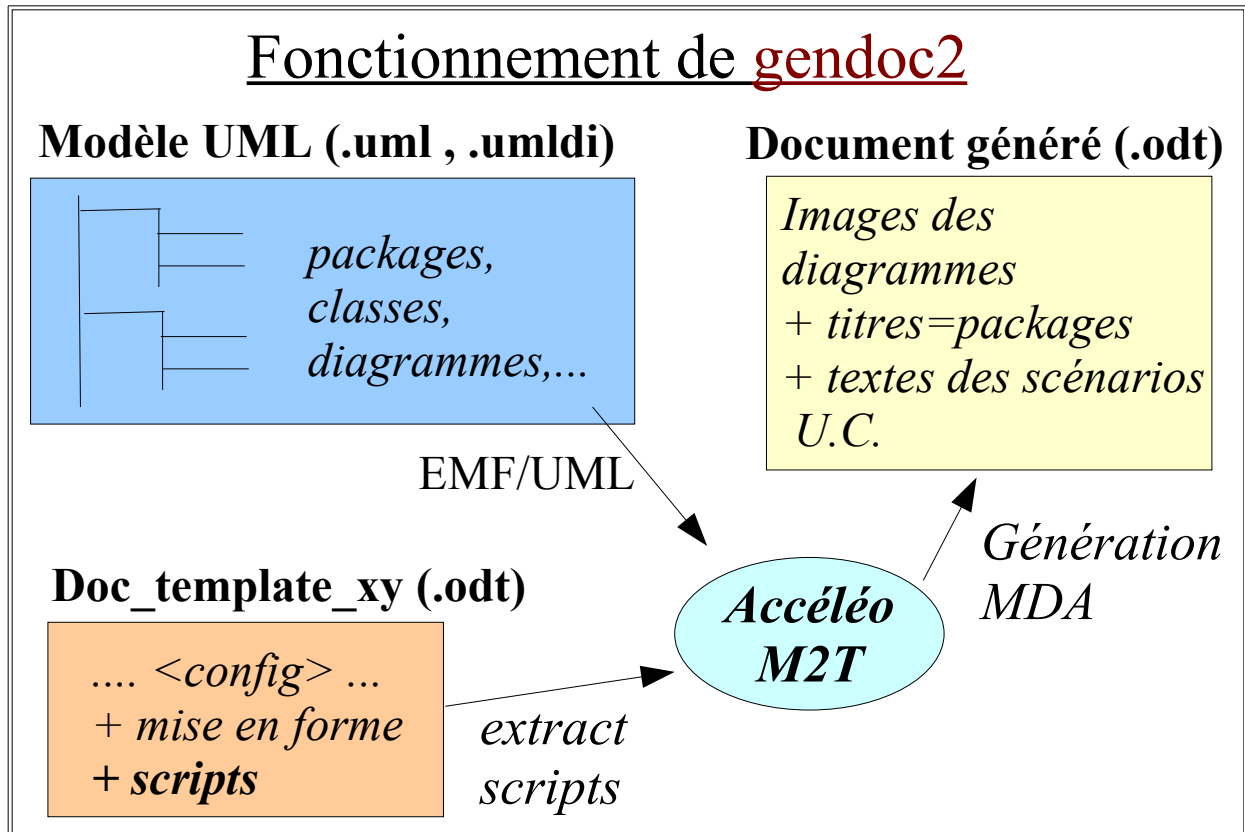
**Gendoc2** est déjà **intégré à Topcased\_RCP >=5** et prêt à l'emploi.

L'ancienne version "gendoc" utilisait en interne une ancienne version du générateur MDA "accéléo"  
La nouvelle version "gendoc2" utilise en interne la nouvelle version 3 d'accéléo (accéléo\_M2T).

NB: En combinant des *fichiers générés par gendoc2* avec des *fichiers statiques* , on peut assez

rapidement produire des **spécifications** assez complètes de bonnes qualités et toujours cohérentes avec les dernières versions des modèles .

## 2.1. Principe de fonctionnement de gendoc2



**NB:** Le déclenchement du processus de génération de documentation s'effectue simplement en:

- se plaçant sur un fichier modèle "doc\_template.odt" (ou bien .docx)
- activant le menu contextuel "Generate Documentation".

**NB2 :** Les exemples de configurations qui suivent sont adaptés à l'éditeur "papyrus".

## 2.2. Paramétrages généraux (configuration, contexte(s))

Un fichier modèle de documentation à générer (doc\_template) doit comporter (généralement dès le début) un **bloc de configuration XML** qui sera pris en compte par gendoc2 et qui ne sera pas affiché au sein de la documentation produite.

Ce bloc de configuration sert essentiellement à préciser les **chemins d'accès** nécessaires pour localiser le modèle UML , le "template" initial et la documentation à générer.

Syntaxe et exemple:

```

<config>
<param key='workspace' value='c:\tp\tp-uml\my-topcased-uml-wksp' />
<param key='project' value='${workspace}\uml2_papyrus_topcased' />
<param key='appName' value='bibliotheque' />
<param key='model' value='${project}\Models\applications\${appName}\${appName}.uml' />
<output path='${project}/Documentation/${appName}/Generated/exprBesoins.odt' />
</config>
<context model='${model}' importedBundles='gmf;papyrus' searchMetamodels='true' />
  
```

Ensuite , dans le reste du fichier modèle à générer, on pourra trouver un ou plusieurs blocs (éventuellement complémentaires) de type `<context ....>` pour préciser des **chemins internes au modèle UML** qui seront considérés comme des **bases** (ou points de départs) de l'**extraction d'informations UML via des scripts**.

```
...
<context element='Model/UseCaseView' /><gendoc>
.... script basé sur Model/UseCaseView
</gendoc>

<context element='Model/LogicalView' /><gendoc>
.... script basé sur Model/LogicalView
</gendoc>
...
```

## 2.3. Généralités sur les scripts de gendoc2

Un script pour gendoc2 est encadré par la balise XML `<gendoc>....</gendoc>`

Il comporte des instructions entre [ ] qui seront interprétées par accéléro M2T .

Ces instructions entre [ ] servent essentiellement à :

- boucler sur les éléments internes du modèle UML
- filtrer les éléments recherchés selon divers critères (types, ...)
- effectuer des opérations de mise en forme (concaténation, ...)
- ...

Des sous (sous) boucles de type ([for] ... [/for] imbriqués) sont possibles et assez fréquentes.

*Syntaxe fondamentale:* [ for (nomVar :TypeElementUML | surQuoiOnBoucle ) ] [nomVar/] [/for]

Attention à ne pas placer trop d'élément de type "espace" ou "saut de ligne" car ceux-ci seront répétés en boucle lors de la génération de documentation

Cette contrainte explique pourquoi les fermetures des instructions ne sont pas souvent placées de façon symétrique par rapport aux ouvertures (décalages fréquents dans l'indentation).

Exemple:

```
<context element='Model/UseCaseView' />
Expression des besoins fonctionnels (Uses Cases) ici en texte caché (open office ou word)
```

```
<gendoc>
```


```
[for (uc:UseCase|self.ownedElement->filter(UseCase))]
```

```
U.C. "[uc.name/]"
```

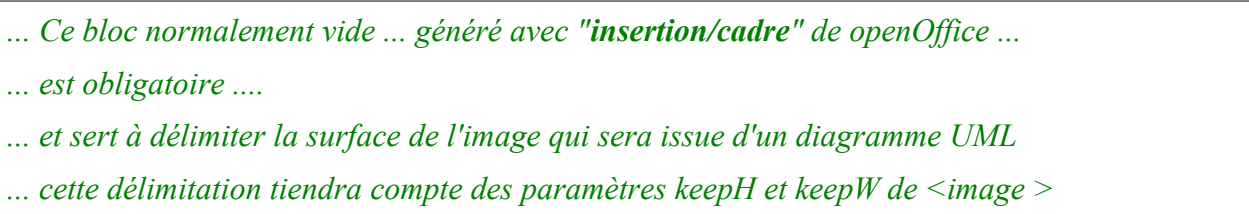

```
[for(ligne:String|uc.getDocumentation().splitNewLine())][ligne /]
```

```
[/for]
```

```
[for (a:Activity|self.ownedElement->filter(Activity) )]
```

```
[for(subActivitydiag : Diagram | a.getPapyrusDiagrams())]
<image object='[subActivitydiag.getDiagram() /]' keepW='true'>

</image>[/for][for]
[/for]</gendoc>
```

## 2.4. Scripts avec images/diagrammes

```
<gendoc>
[for(diag : Diagram | self.getPapyrusDiagrams())]
<image object='[diag.getDiagram() /]' keepW='true'>

</image>[/for]
[for (p:Package|self.ownedElement->filter(Package))]
[p.name/]
[for(ligne:String|p.getDocumentation().splitNewLine())][ligne /][for]
[for(subdiag : Diagram | p.getPapyrusDiagrams())]
<image object='[subdiag.getDiagram() /]' keepW='true'>

</image>[/for][for]
[/for]</gendoc>
```

*... Ce bloc normalement vide ... généré avec "insertion/cadre" de openOffice ...  
 ... est obligatoire ....  
 ... et sert à délimiter la surface de l'image qui sera issue d'un diagramme UML  
 ... cette délimitation tiendra compte des paramètres keepH et keepW de <image >*

## 2.5. Document maître pour fédérer plusieurs fichiers générés

On peut éventuellement utiliser un fichier "*spécifications\_fonctionnelles.odm*" au format "document maître de OpenOffice" pour fédérer:

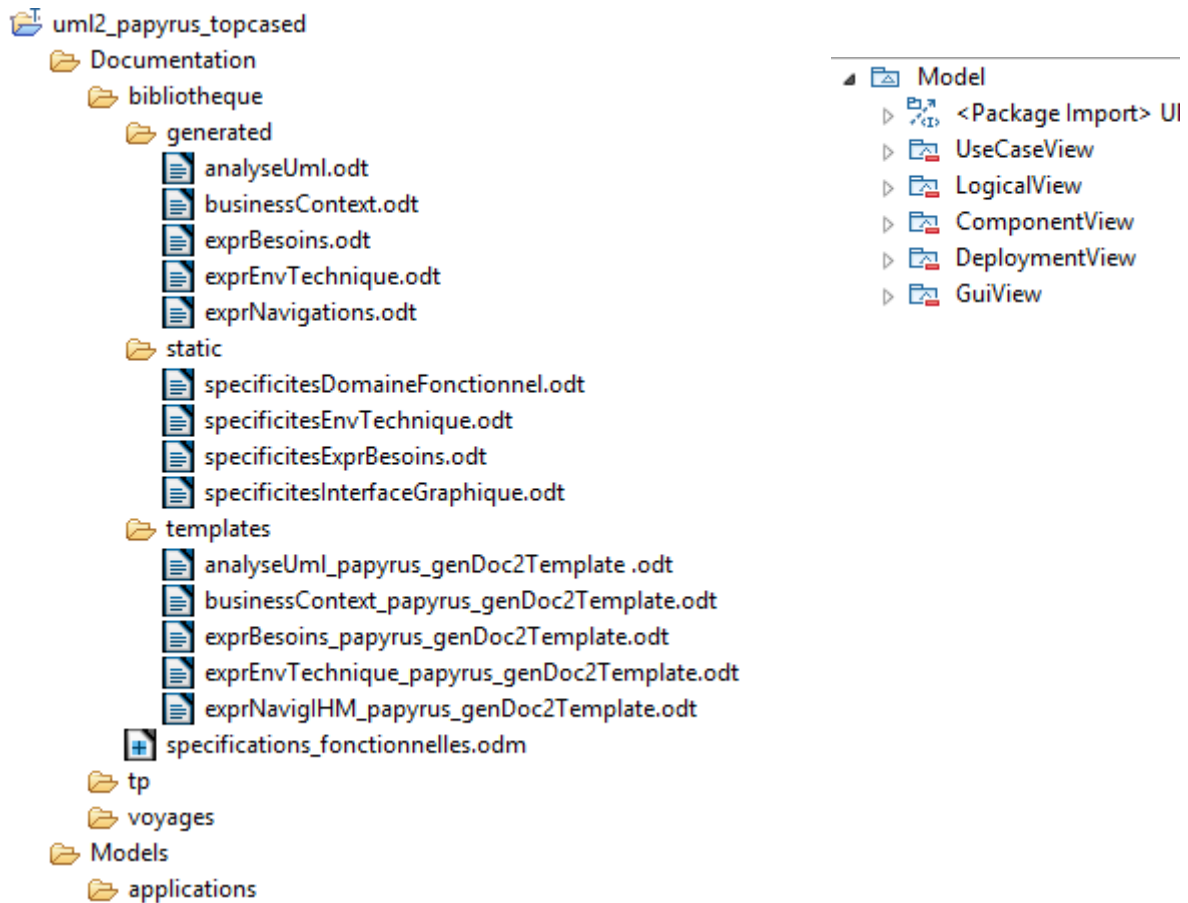
- des fichiers issus d'une génération automatique "UML/gendoc2"
- des fichiers éditer manuellement avec des contenus très spécifiques
- ...

Ceci permet en outre :

- d'obtenir une bonne numérotation des chapitres
- de construire facilement une table des matières globale
- de facilement exporter le tout au format pdf.

Exemple d'organisation de la documentation :

\* analyseUml\_....Template génère la doc structurelle (diag classes , packages , diagrammes d'états et



diag. séquences) ---- à partir de LogicalView et à partir des séquences attachées aux UseCase(View)

- \* exprBesoins.....Template génère l'expression des besoins (uses cases + scénarios + diag d'activités) à partir de UseCasesView
- \* exprEnvTechnique..Template génère le diagramme de déploiement à partir de DeploymentView
- \* businessContext...Template génère businessContext à partir du modèle annexe "context\_XXX.uml"

\* exprNavigIHM....Template génère des diagrammes sur l'IHM (structure + navigations) depuis la partie GuiView

## XXIII - Annexe – Bibliographie, Liens WEB , outils

### 1. Bibliographie et liens vers sites "internet"

Site de référence (OMG)	<a href="http://www.uml.org/">http://www.uml.org/</a>
UML en français (site web didactique) – UML 1	<a href="http://uml.free.fr/">http://uml.free.fr/</a>
Cours UML en ligne (syntaxe UML2)	<a href="http://laurent-audibert.developpez.com/Cours-UML/">http://laurent-audibert.developpez.com/Cours-UML/</a>
<a href="http://www.eyrolles.com/Informatique/">http://www.eyrolles.com/Informatique/</a>	Pour voir les livres qui existent sur UML