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.....9
3. UML pour illustrer les spécifications.....10
4. Formalisme UML = langage commun.....10
5. UML universel mais avant tout orienté objet.....11
6. Standard UML et extensions (profiles).....11
7. Présentation des diagrammes UML.....12
8. Descriptions sommaires des diagrammes UML.....15
9. Utilisations courantes des diagrammes UML.....21

10. Quelques outils UML (Editeurs , AGL).....22

III - Démarche , études de cas , ...23
1. Activités de modélisation et spécifications.....23
2. Cycle "itératif & incrémental".....25
3. Enchaînement des activités de modélisation.....26
4. Bibliothèque / médiathèque (étude de cas).....27
5. Agence de voyage (étude de cas aternative).....28
6. Autres étude de cas.....29

IV - Diagramme de contexte / périmètre applicatif.....30
1. Début de la modélisation UML ?.....30
2. Diagramme de contexte.....31

V - Expr. besoins fonctionnels / Uses Cases.....33
1. Etude des principales fonctionnalités (U.C.).....33
2. Diagramme des cas d'utilisations.....34
3. Scénarios et descriptions détaillés (U.C.).....39
4. Gestion de projet et planification basées sur les cas d'utilisation.....42
5. Eventuelle planification des livrables selon XP.....42

VI - Diagramme d'activités.....44
1. Diagramme d'activités.....44
2. Notations avancées pour (rares) diagrammes d'activités très détaillés ou très techniques.....50

VII - Règles de gestions.....52
1. Formulations des exigences et traçabilité.....52
2. Exemples d'expression des règles de gestion.....52

VIII - Génération documentations / spécifications.....53
1. Documentation / Livrables / Spécifications.....53

IX - Modélisation métier (business modeling).....55
1. Rôle et contenu de la modélisation métier.....55
2. Modélisation du contexte d'utilisation.....57
3. Diagramme des cas d'utilisations métiers.....58

X - Expr. Besoins IHM (maquettes, navigations).....59
1. Expression des besoins "IHM" (interface graphique).....59

XI - Fondamentaux orientés "objet.....63
1. Concepts objets fondamentaux.....63
2. Granularité.....70
3. Modularité.....71

XII - Analyse du domaine (entités) / diag. Classes.....73
1. Analyse du domaine (glossaire , entités).....73
2. Diagramme de classes (notations , ...).....75
3. Eléments structurants d'UML.....86

XIII - Diagrammes d'états (cycle de vie , ...).....90
1. Diagramme d'états et de transitions (StateChart).....90
2. Utilisation d'un diagramme d'état pour illustrer un cycle de vie.....95

XIV - Réalisation de Uses Cases / diag. Séquences.....97
1. Analyse applicative (objectif et mise en oeuvre).....97
2. Responsabilités (n-tiers) et services métiers.....98
3. Repère méthodologique (rappel).....99
4. Réalisation des cas d'utilisations.....100
5. Modèle dynamique – diagrammes d' interactions.....101
6. Diagramme UML de Collaboration / Communication.....102
7. Diagramme de séquences (UML).....103
8. Diagramme de séquences (notations avancées).....105
9. Diagramme "*interaction overview" (UML 2)*.....109

XV - Besoins techniques / env. / diag déploiement.....110
1. Spécification de l'environnement cible.....110
2. Exemples d'exigences techniques classiques.....111

XVI - Aperçu général sur la conception.....112
1. Rôles de la conception.....112
2. Activités de la conception.....113
3. Conception générique.....115
4. Projection du fonctionnel dans technologies.....116
5. Objectifs de la conception préliminaire.....117

XVII - Implémentation , retours tests , itérations.....119

XVIII - Différences entre UML et Merise.....121

1. différences cardinalités/multiplicités.....121

XIX - Annexe – UML et mapping objet-relationnel.....122

1. Cas de figures (down-top , top-down , ...).....122
2. Correspondances essentielles "objet-relationnel".....122
3. Correspondances "objet-relationnel" avancées.....125
4. Stéréotypes UML pour O.R.M.....126

XX - Annexes - Patterns "GRASP".....128

1. Affectation des responsabilités (GRASP).....128
2. Les 4 patterns GRASP fondamentaux.....130
3. Les 5 paterns GRASP spécifiques.....134

XXI - Annexes - Méthodologies (aperçu rapide).....137

1. Bonnes pratiques UP.....137
2. Méthodes agiles.....138
3. XP (Extreme Programming).....140
4. Méthode agile SCRUM.....144

XXII - OCL (Object Constraint Language).....150

1. OCL (Object Constraint Language) - Présentation.....150

XXIII - Essentiel outil "StarUML".....151

1. Utilisation Star-UML 3 (outils UML2).....151

XXIV - Annexe – Bibliographie, Liens WEB , outils.....166

1. Bibliographie et liens vers sites "internet".....166

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 effectué) 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

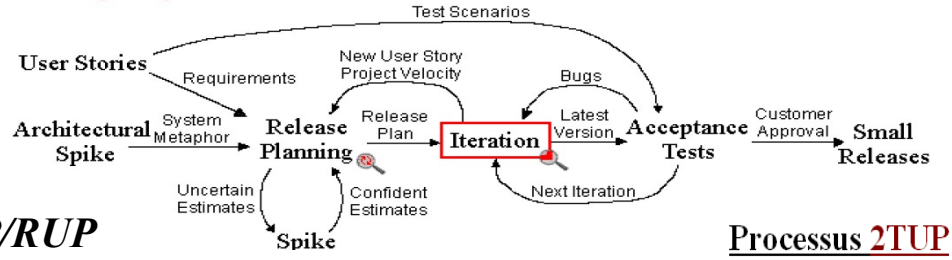
- **UML est un** formalisme
  (Notations standardisées [diagrammes] avec sémantiques précises)

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

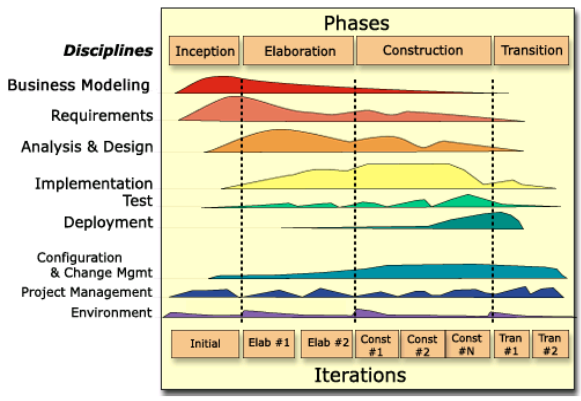
\+ *en pratique*: *les* ***Procédés*** (selon *outils / MDA/ ...*)

Processus méthodologiques (exemples):

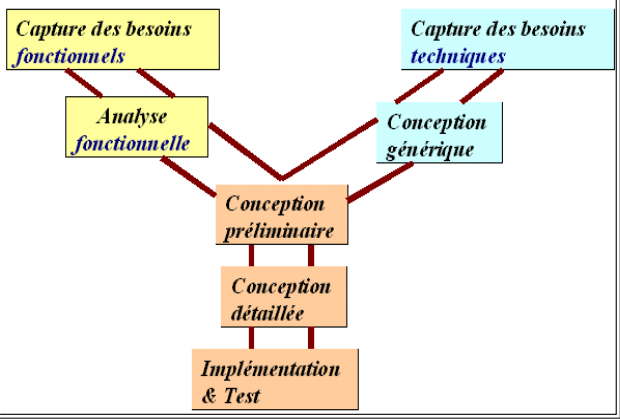
Extreme Programming Project



UP/RUP

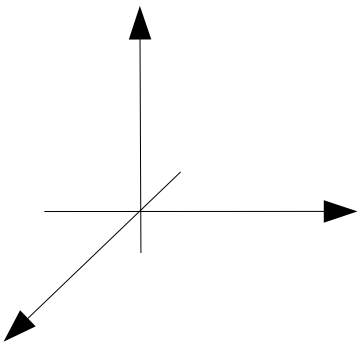


Processus 2TUP



Plein de variantes dans les façons d'utiliser UML

Formalisme
(notations, diagrammes)



Procédés
(outils UML, MDA,...)

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

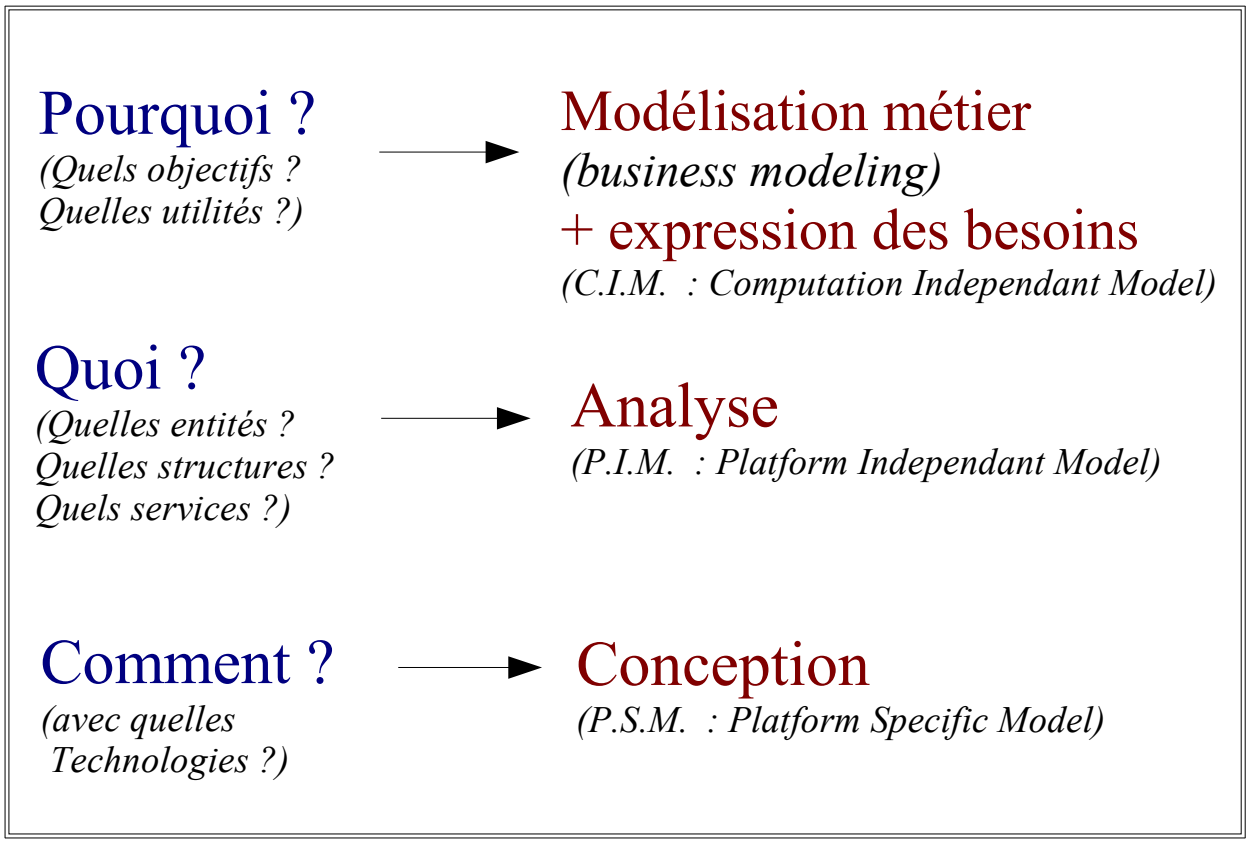
* UML pour
simple ébauche
ou bien
modèle précis ?

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

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

Principales façons d'utiliser UML :

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

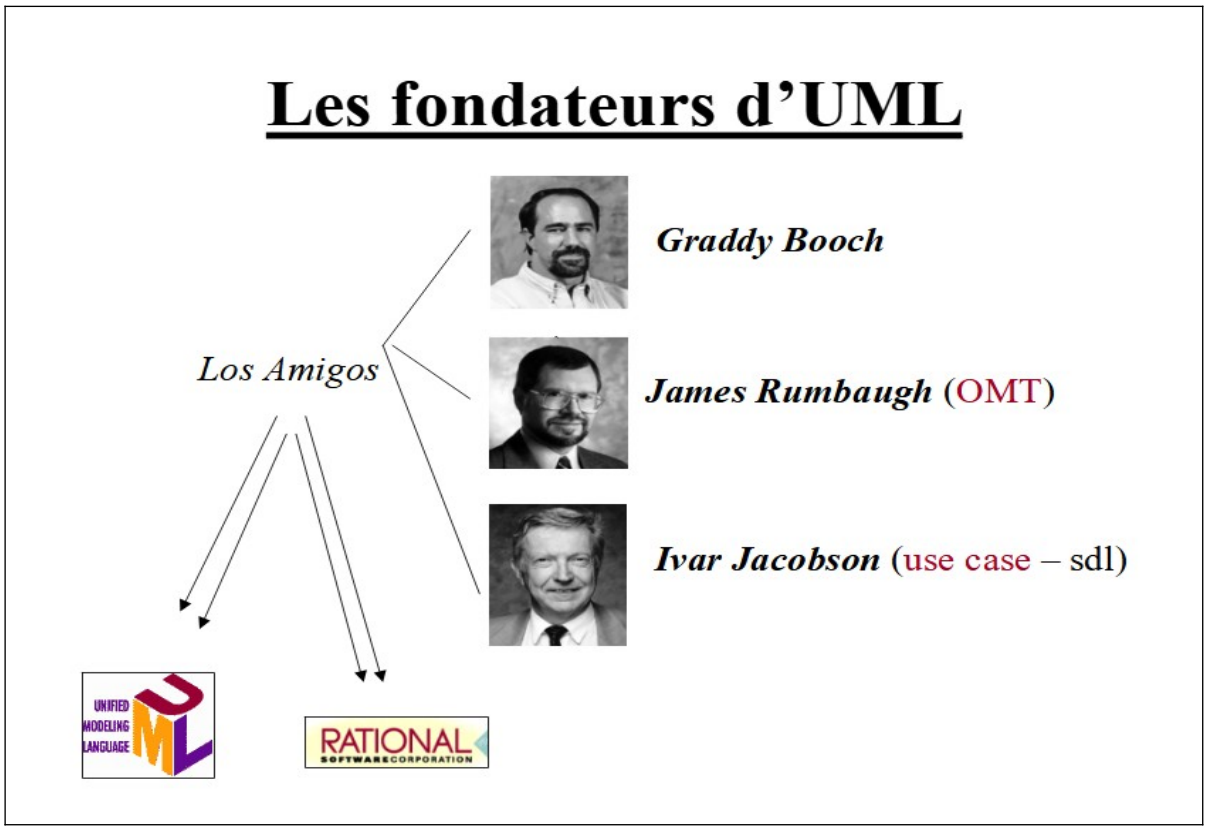


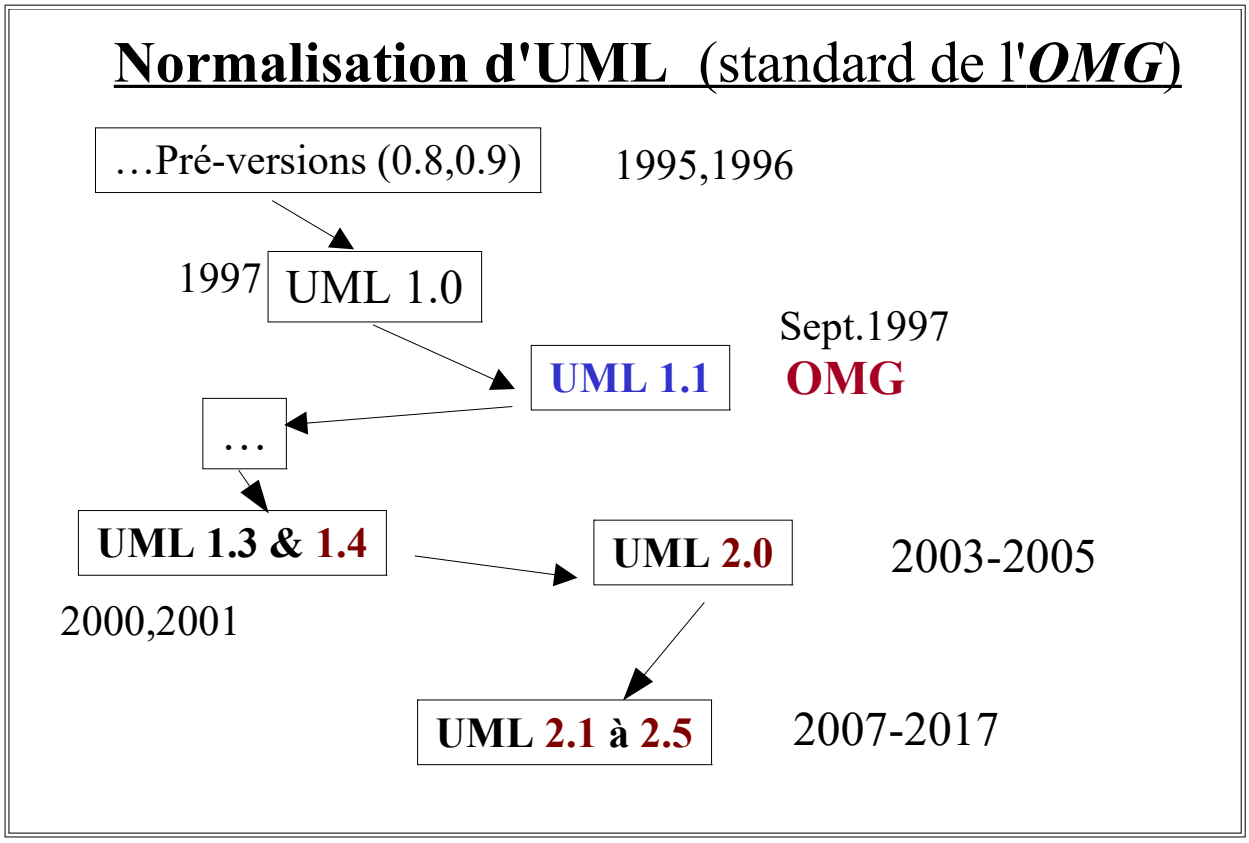
NB : UML peut aussi bien être utilisé pour définir des modèles

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

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

2. Historique rapide d'UML





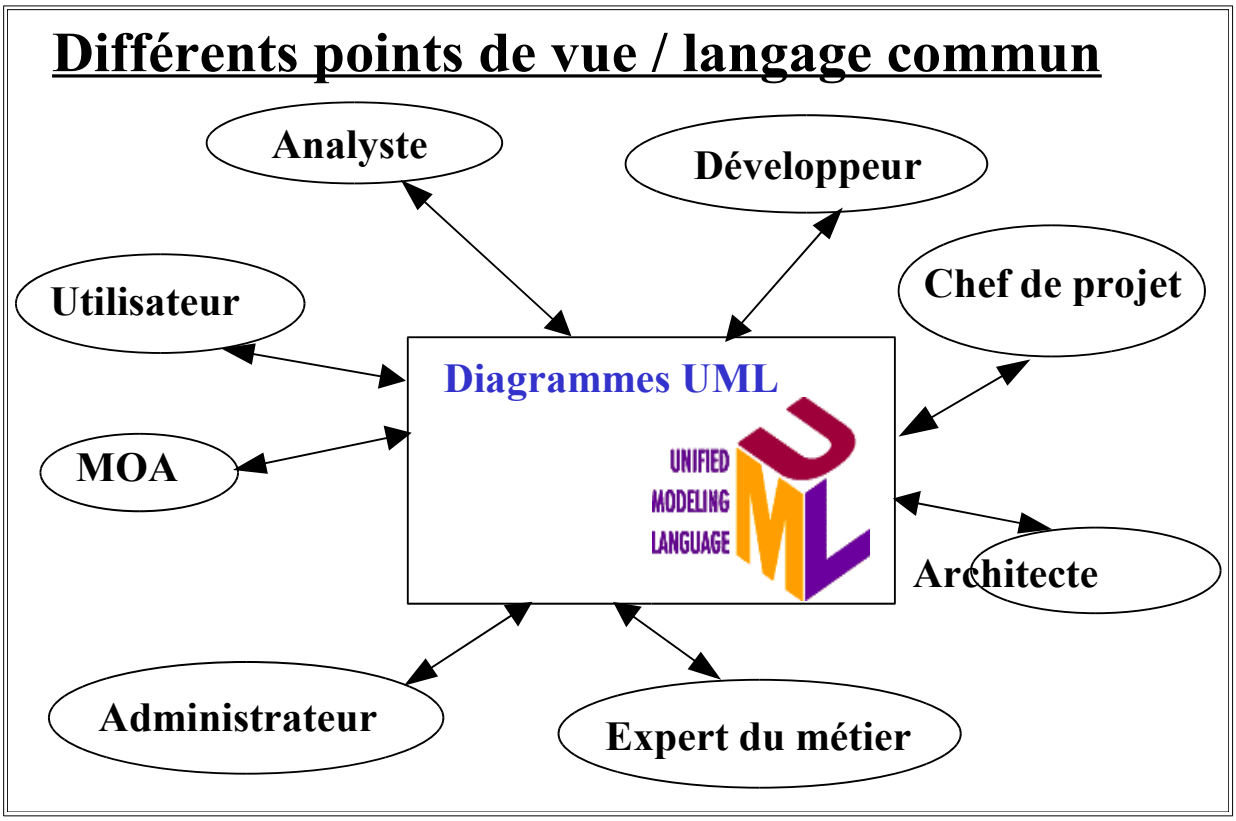
3. UML pour illustrer les spécifications

Principales **utilités d'UML**

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

4. Formalisme UML = langage commun

Différents points de vue / langage commun



5. UML universel mais avant tout orienté objet

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

Méthode Merise

Modèle relationnel (MCD, ...)

Structure d'une base de données + ...

(époque mainframe et client/serveur)

Formalisme UML

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

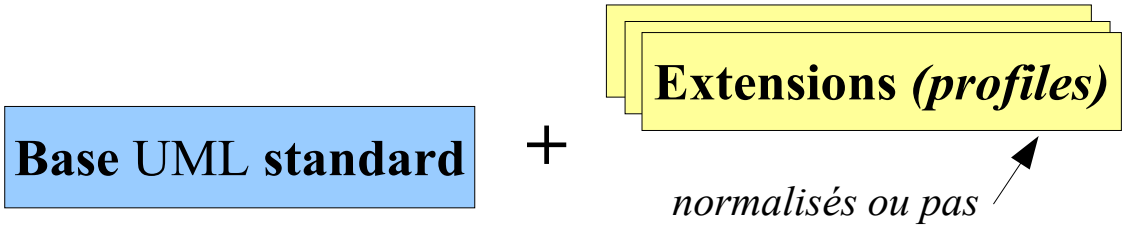
Structure d'une application orientée objet (c++,java, ...) *+indirectement structure d'une base de données (ou fichiers)*

(époque n-tiers et SOA)

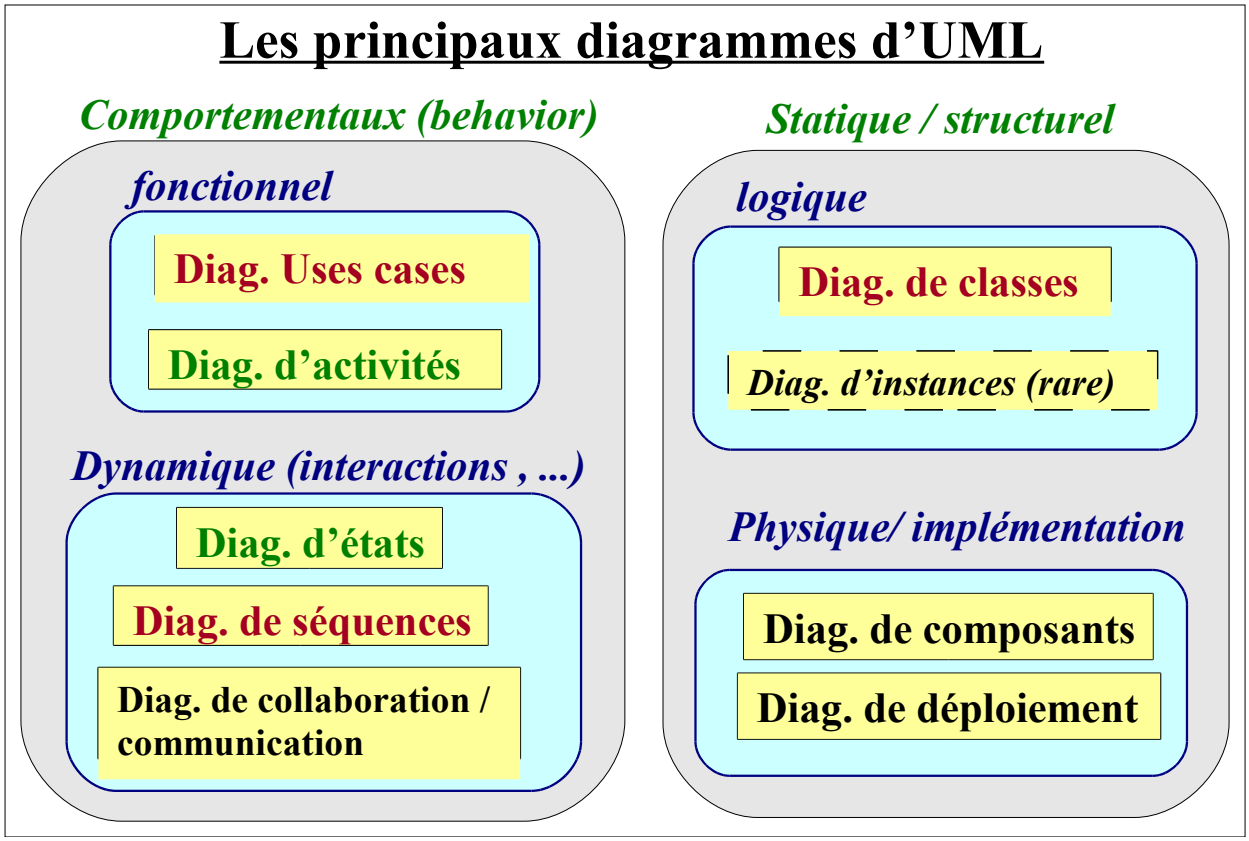
6. Standard UML et extensions (profiles)

Quelques grands traits d'UML

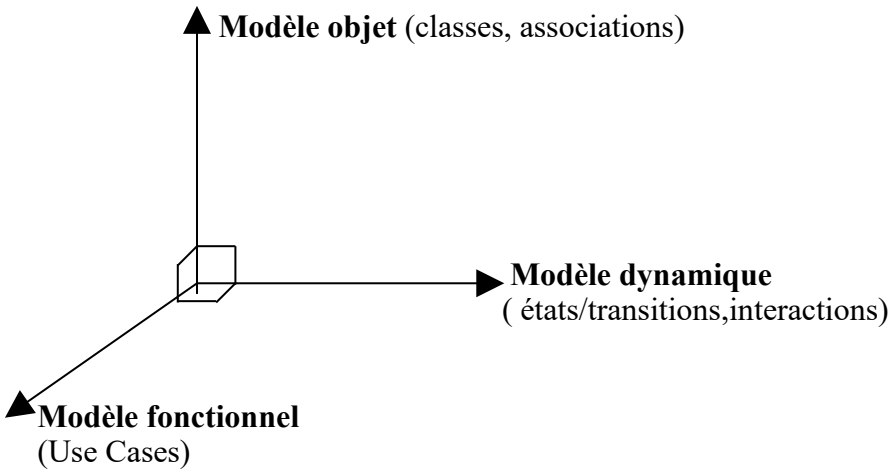
- ***Semi formel*** (plusieurs variantes possibles dans les diagrammes)
- Essentiellement basé sur les ***concepts objets*** *(pas du tout lié au modèle relationnel).*
- Se voulant être assez ***universel*** (java, c++, c# , ...) et utilisable aussi bien en informatique industrielle qu'en informatique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



7. Présentation des diagrammes UML



Modèles (diagrammes) complémentaires:



Les diagrammes secondaires d'UML 2

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

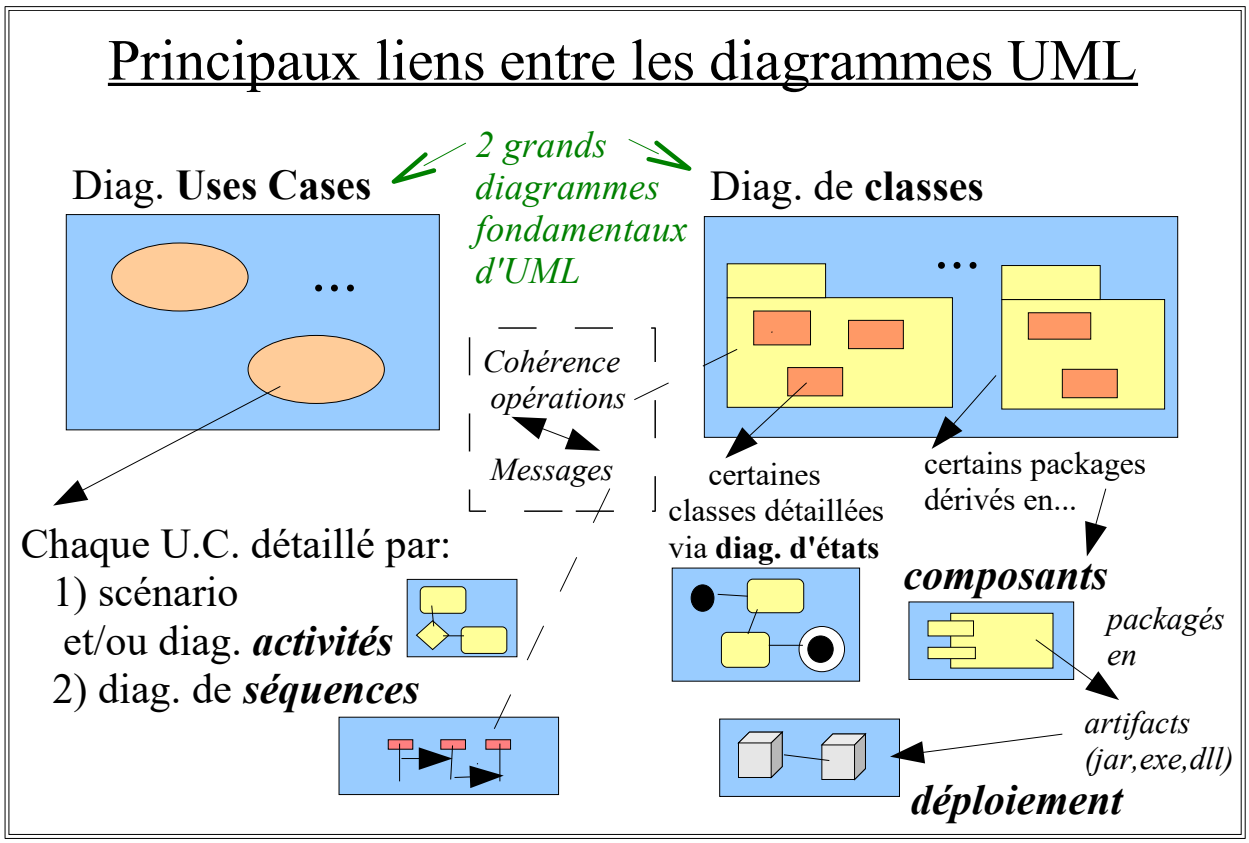
• "***Composite Structure Diagram***" : diagramme complémentaire (annexe) vis à vis du diagramme de classes permettant de montrer la structure interne d'une classe (sous parties) --> **parts , connectors , ports , ...**

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

Diagrammes très secondaires d'UML 2

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

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


Principaux liens entre les diagrammes UML
Diag. Uses Cases
2 grands diagrammes fondamentaux d'UML
Diag. de classes
Cohérence opérations
Messages
certaines classes détaillées via diag. d'états
certains packages dérivés en...
composants
packagés en
artifacts (jar,exe,dll)
déploiement
Chaque U.C. détaillé par:
1) scénario
et/ou diag. activités
2) diag. de séquences

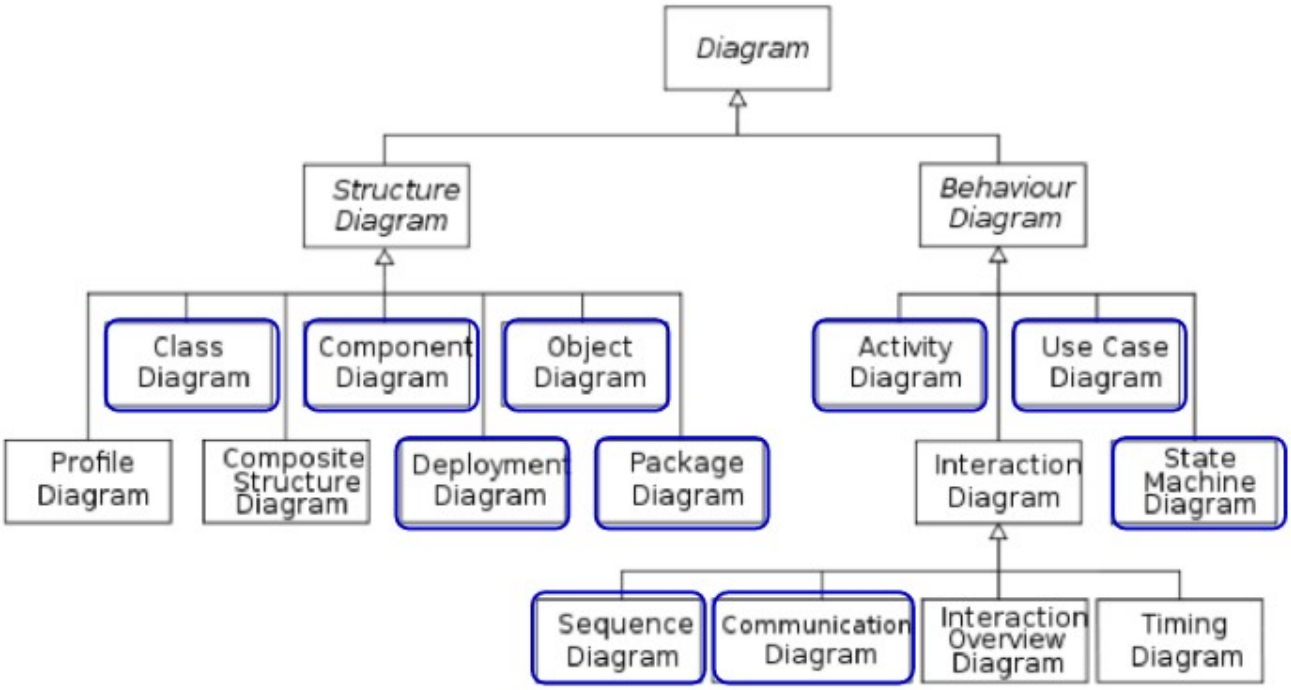
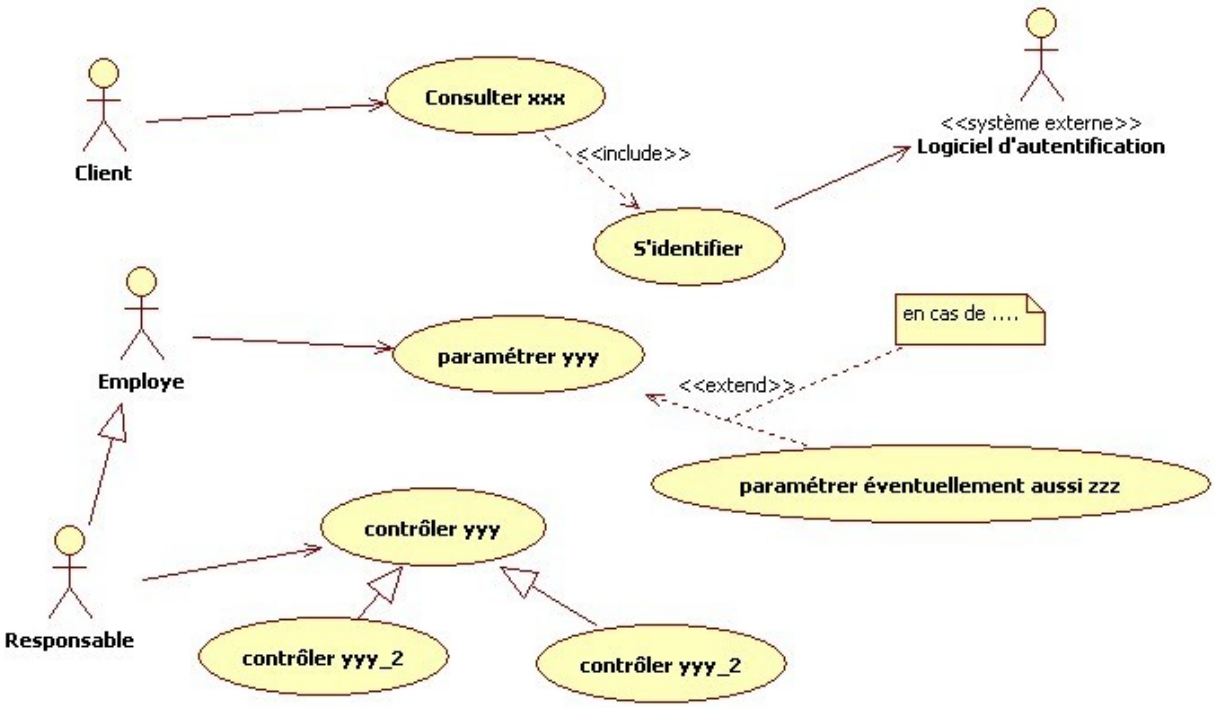


Diagram
Structure Diagram
Behaviour Diagram
Class Diagram
Component Diagram
Object Diagram
Activity Diagram
Use Case Diagram
Profile Diagram
Composite Structure Diagram
Deployment Diagram
Package Diagram
Interaction Diagram
State Machine Diagram
Sequence Diagram
Communication Diagram
Interaction Overview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

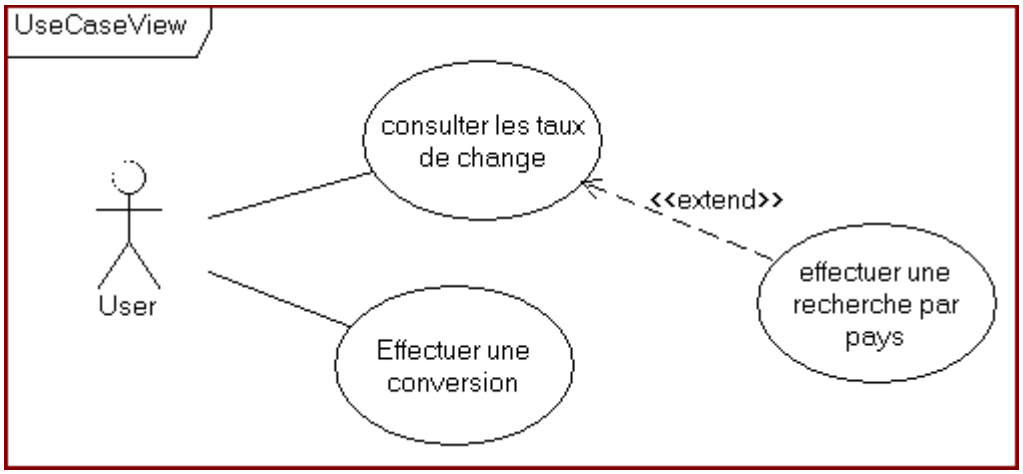
8. Descriptions sommaires des diagrammes UML

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

exemple:



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



8.2. Diagramme de classes

exemples:

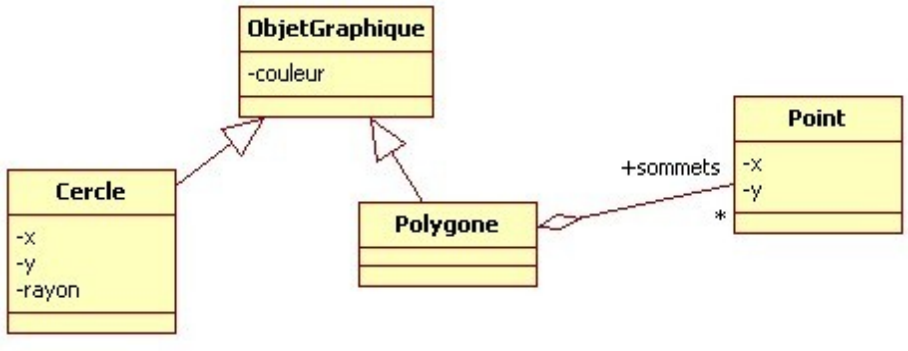
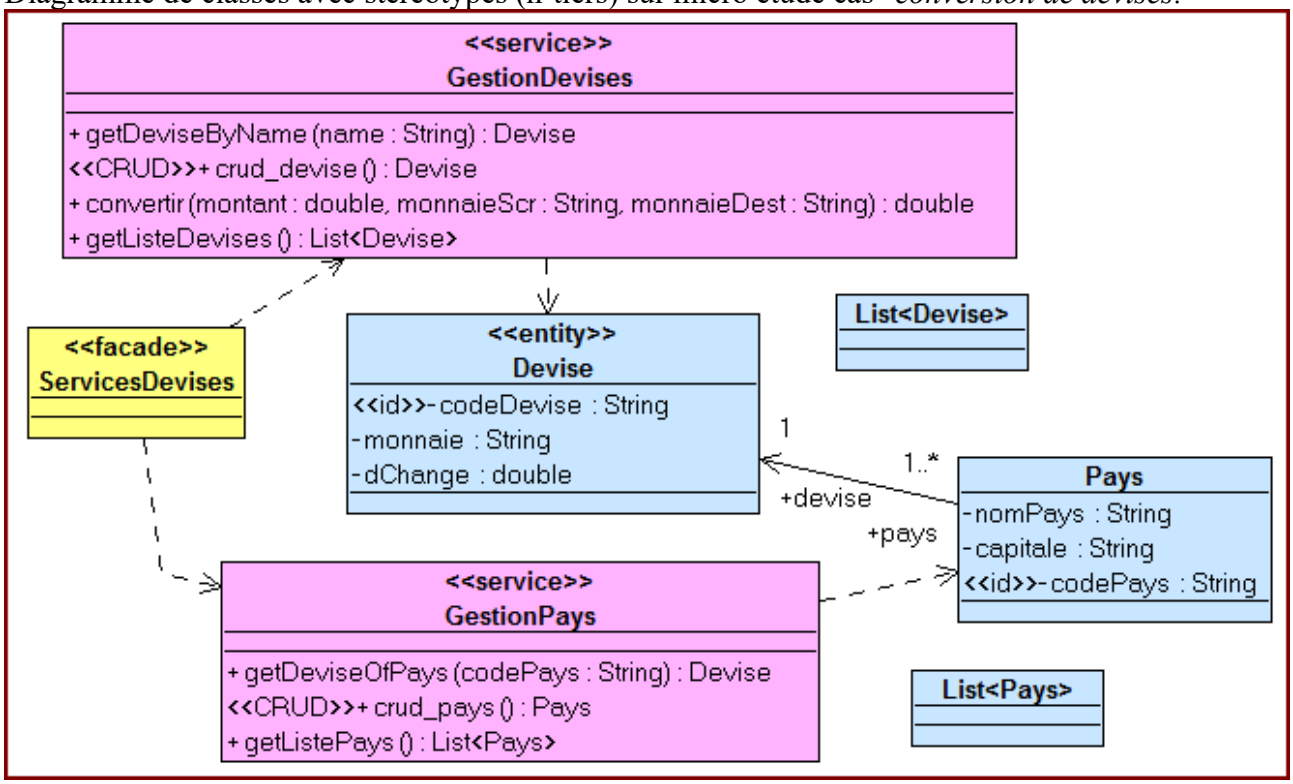
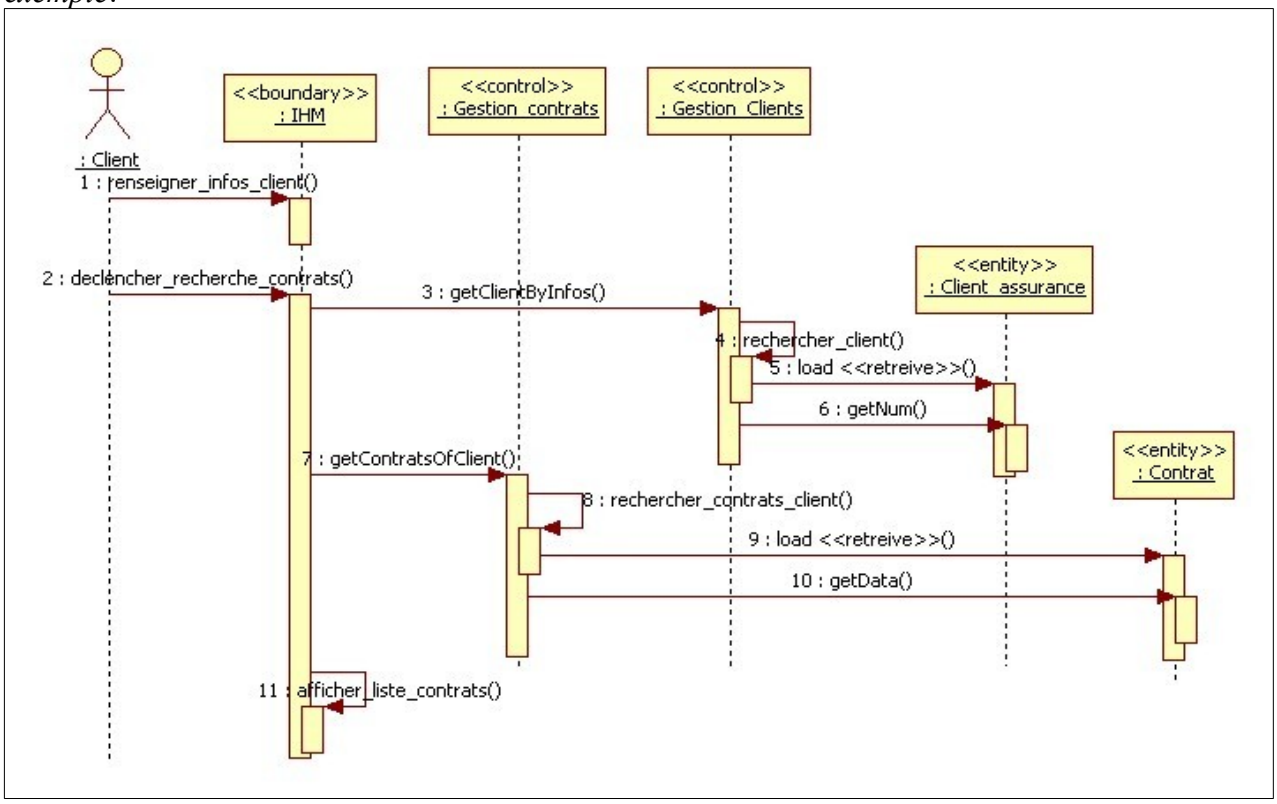


Diagramme de classes avec stéréotypes (n-tiers) sur micro étude cas "*conversion de devises:*



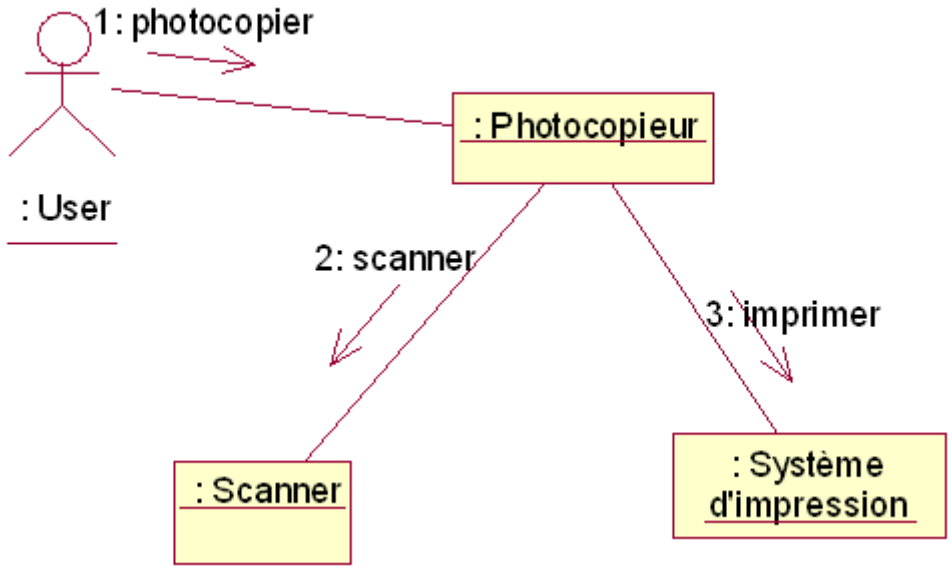
8.3. Diagramme de séquence

exemple:



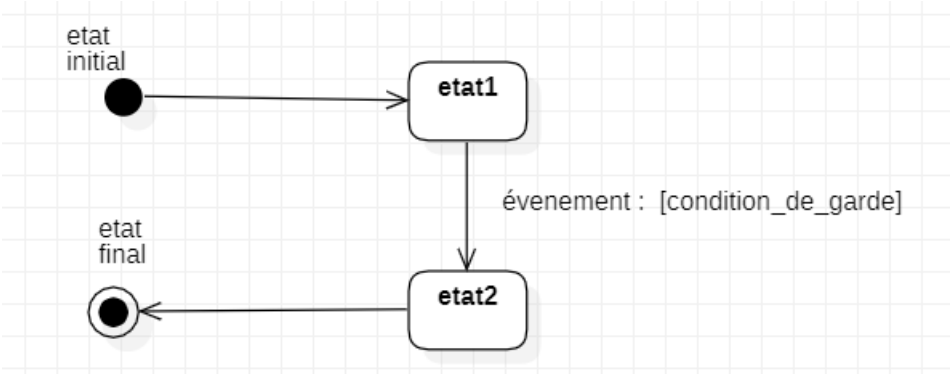
8.4. Diagramme de ~~collaboration (UML1)~~ / communication (UML2)

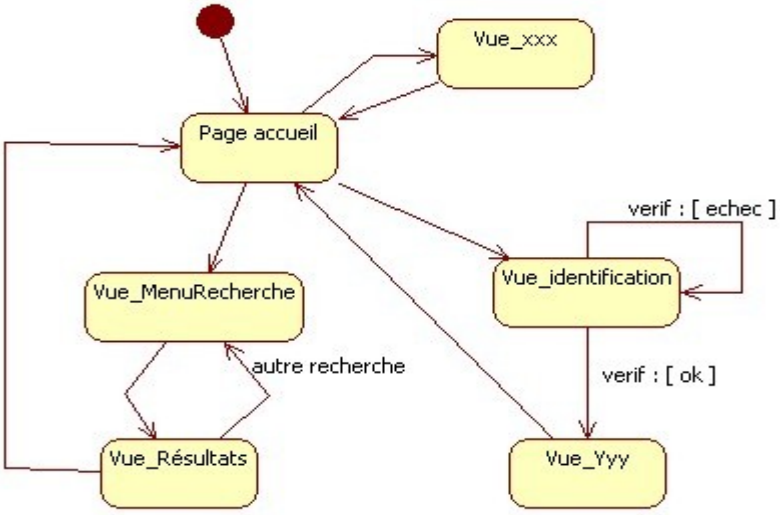
exemple:



8.5. Diagramme d'états (StateChart)

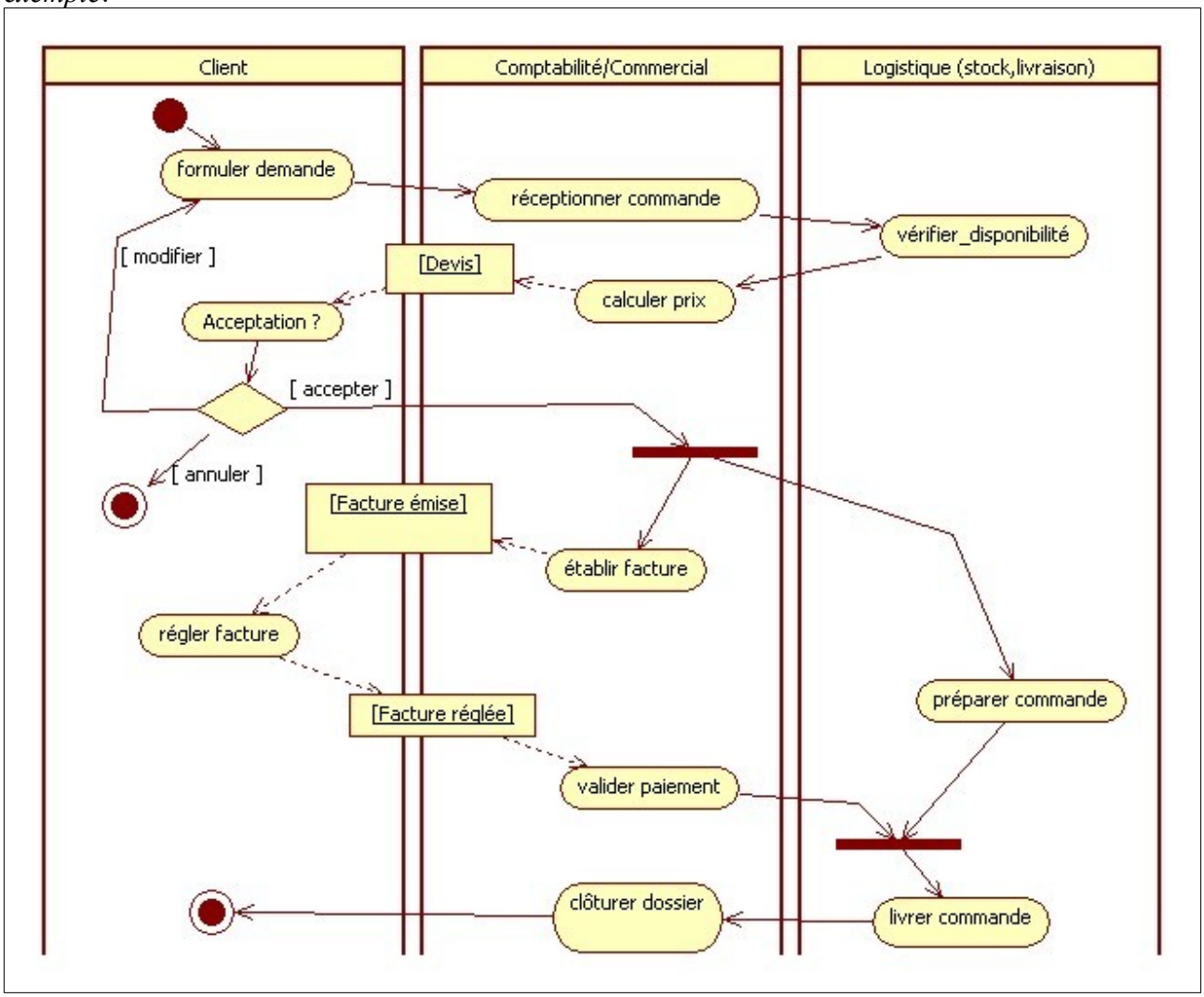
exemples:





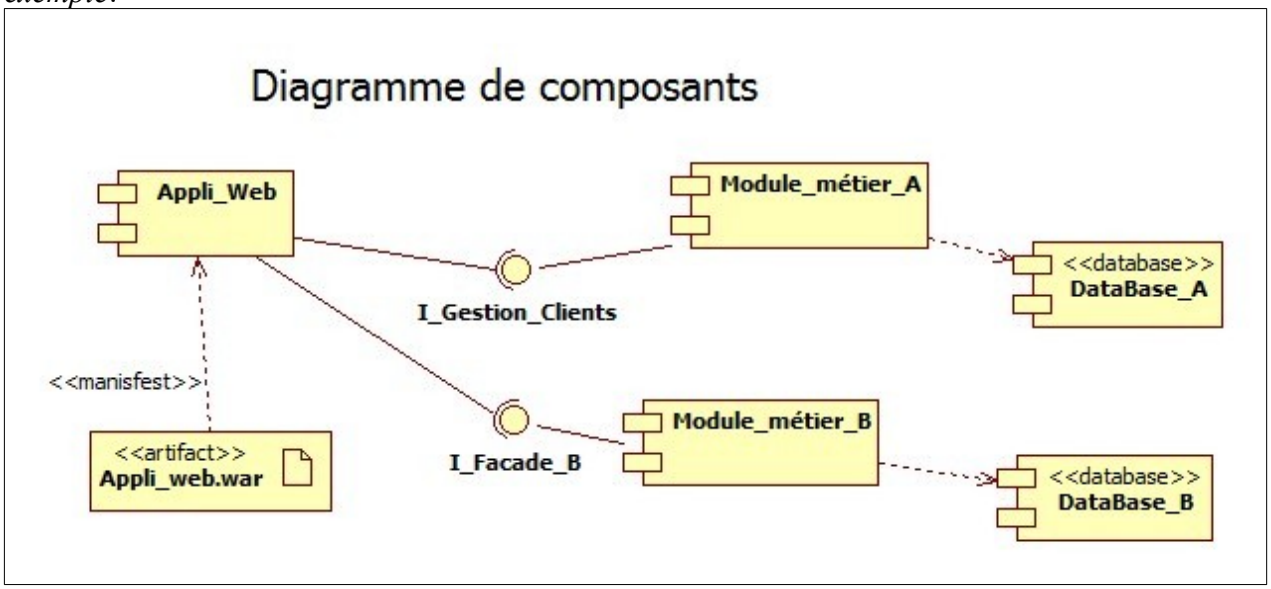
8.6. Diagramme d'activités

exemple:



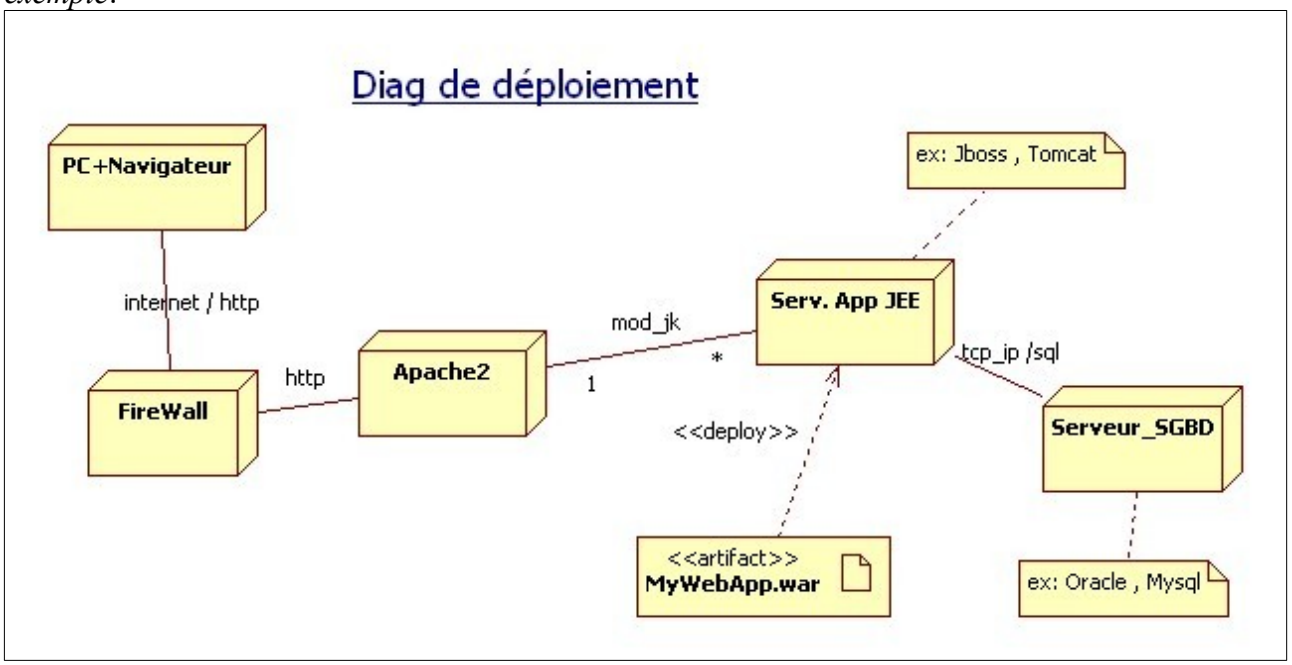
8.7. Diagramme de composants

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

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

10. Quelques outils UML (Editeurs , AGL)

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

III - Démarche , études de cas , ...

1. Activités de modélisation et spécifications

Activités de Modélisation - principales disciplines:

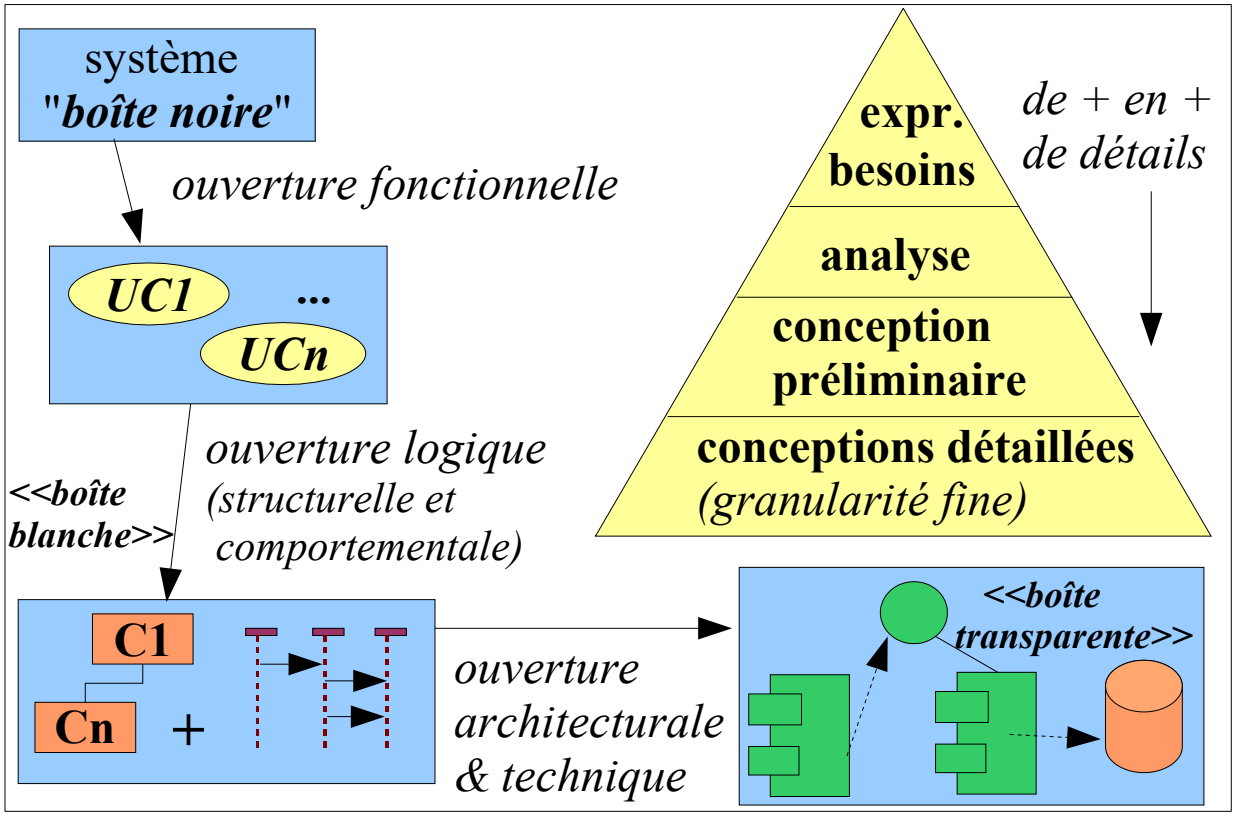
- ***Modélisation métier (business modeling)***
  (de niveau entreprise , sous systèmes, organisationnel + objectifs / utilités ?)
- ***Modélisation du contexte d'utilisation***
  (contexte ? , environnement technique ? ,cadrage ?)
- ***Expression des besoins liés au système à développer***
  (fonctionnalités ? , 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?

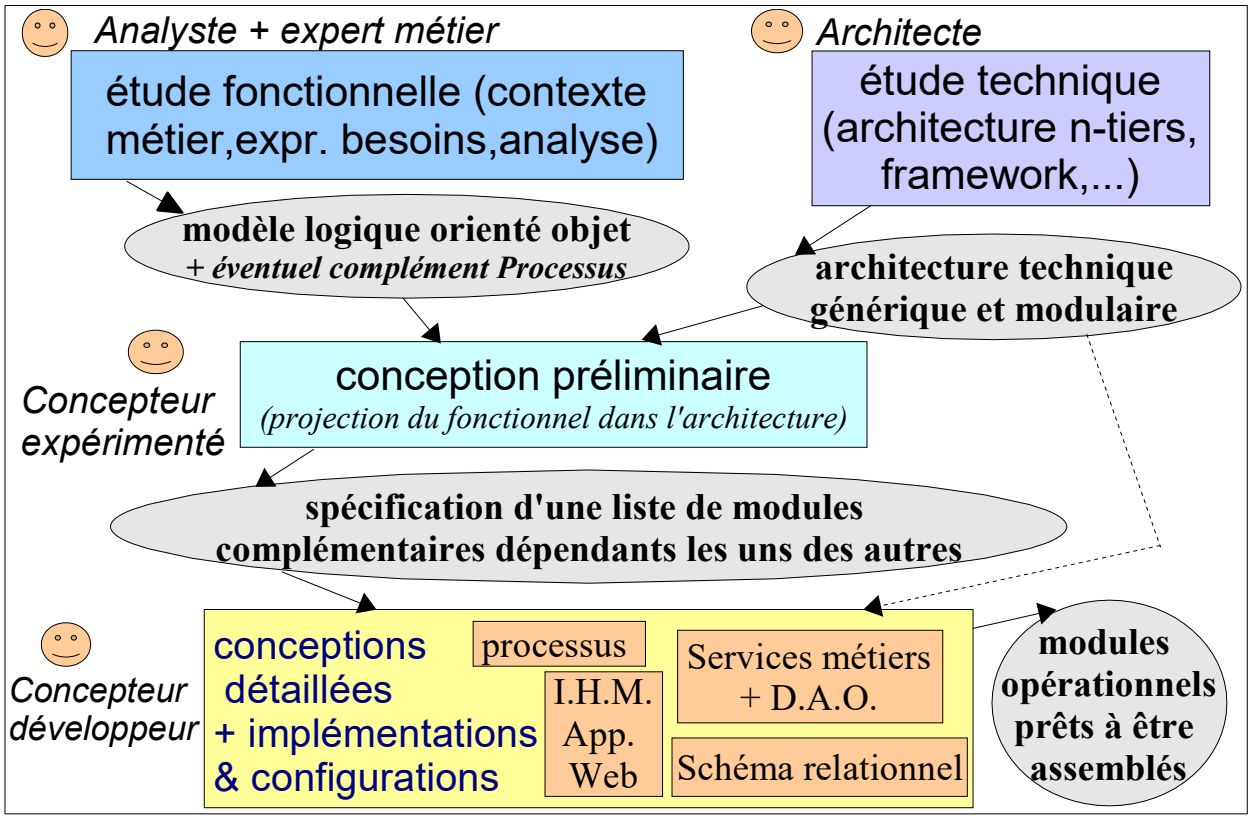
Fruits de la modélisation - principales spécifications:

- ***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***
- ***Dossier d'architecture technique***
  (plans types pour technologies , frameworks, ...)
- ***Spécifications applicatives et techniques***
  ===> ***modèle physique*** (dans les grandes lignes , composants/modules , contrats/interfaces, ...)

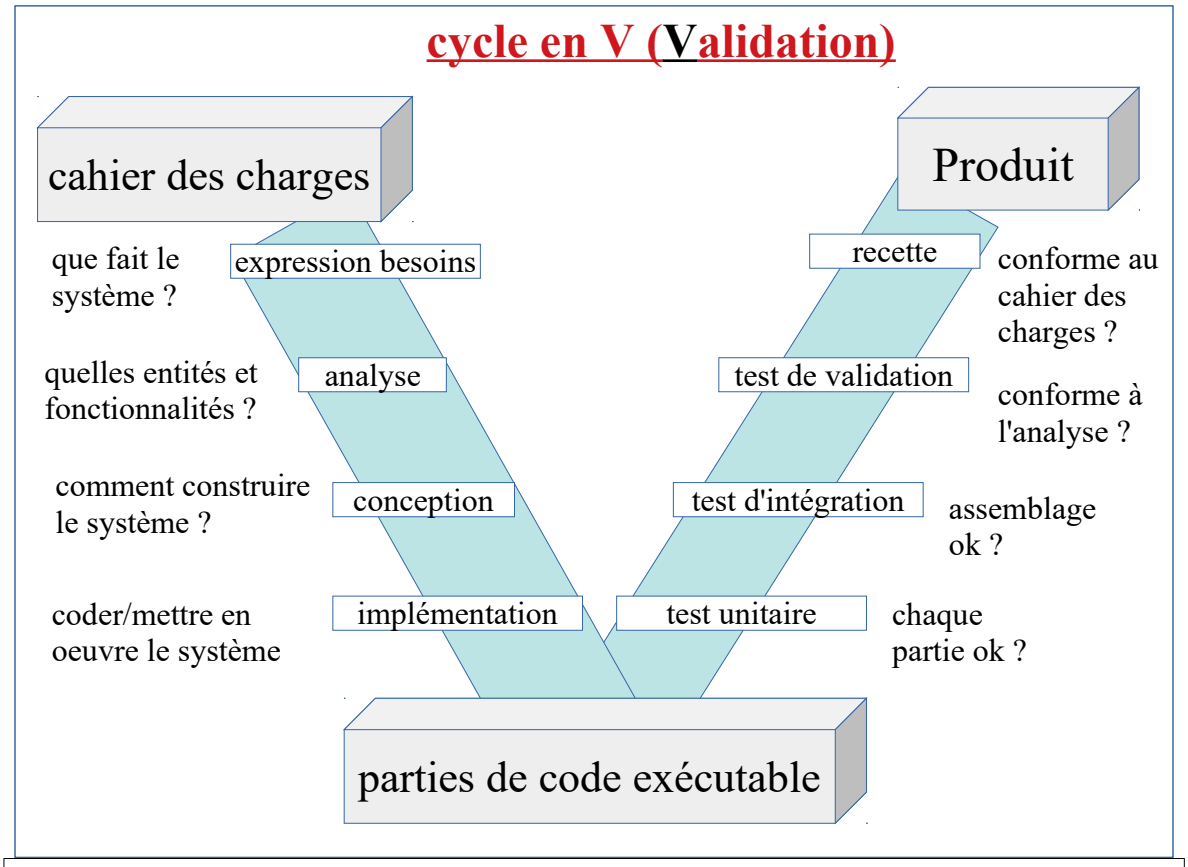
1.1. Vue imagée (modélisation, démarche progressive)

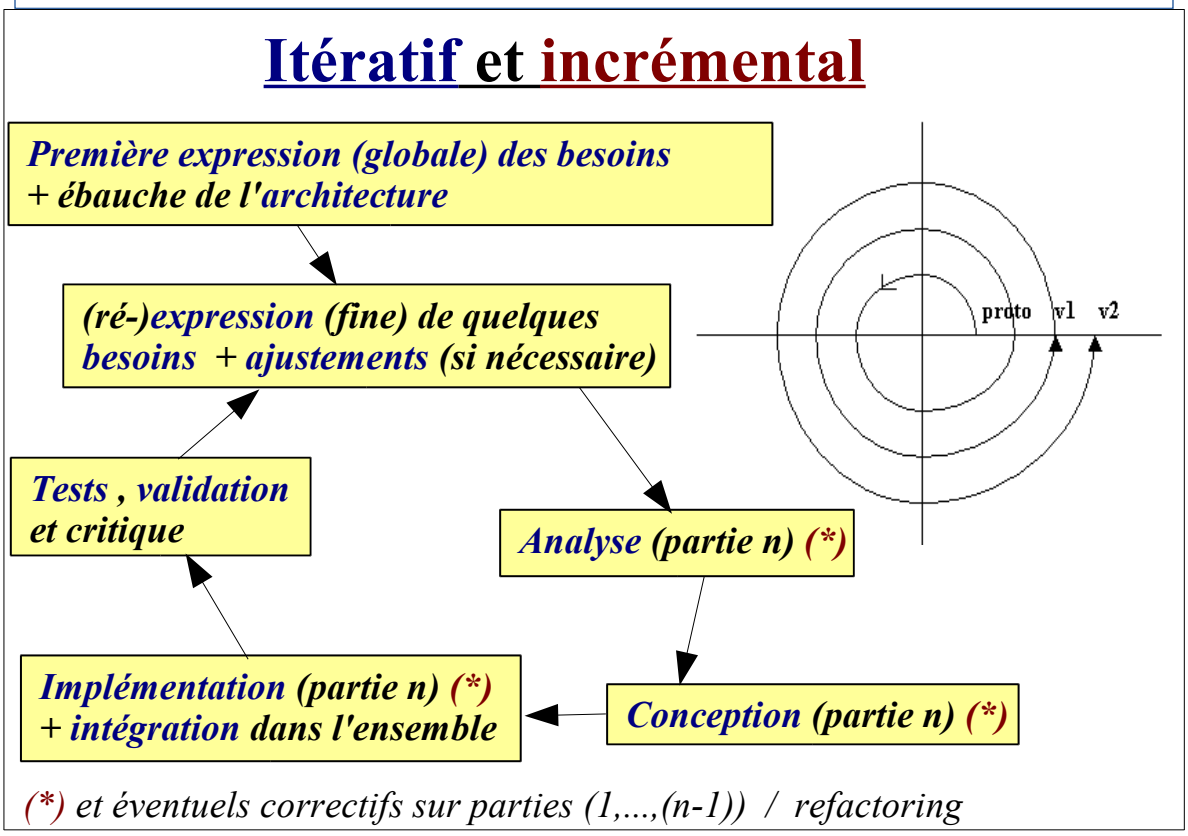


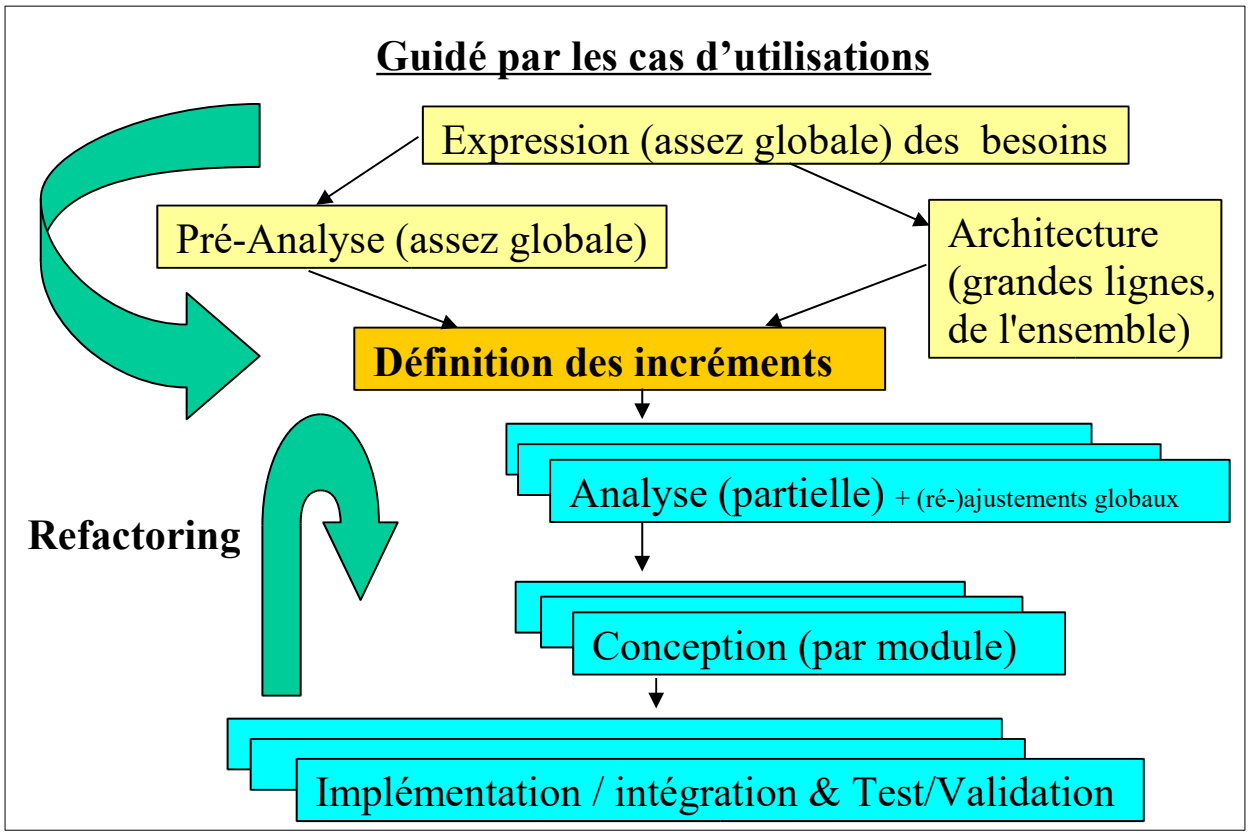
1.2. Chacun sa spécialité (analyse, architecte, ...)



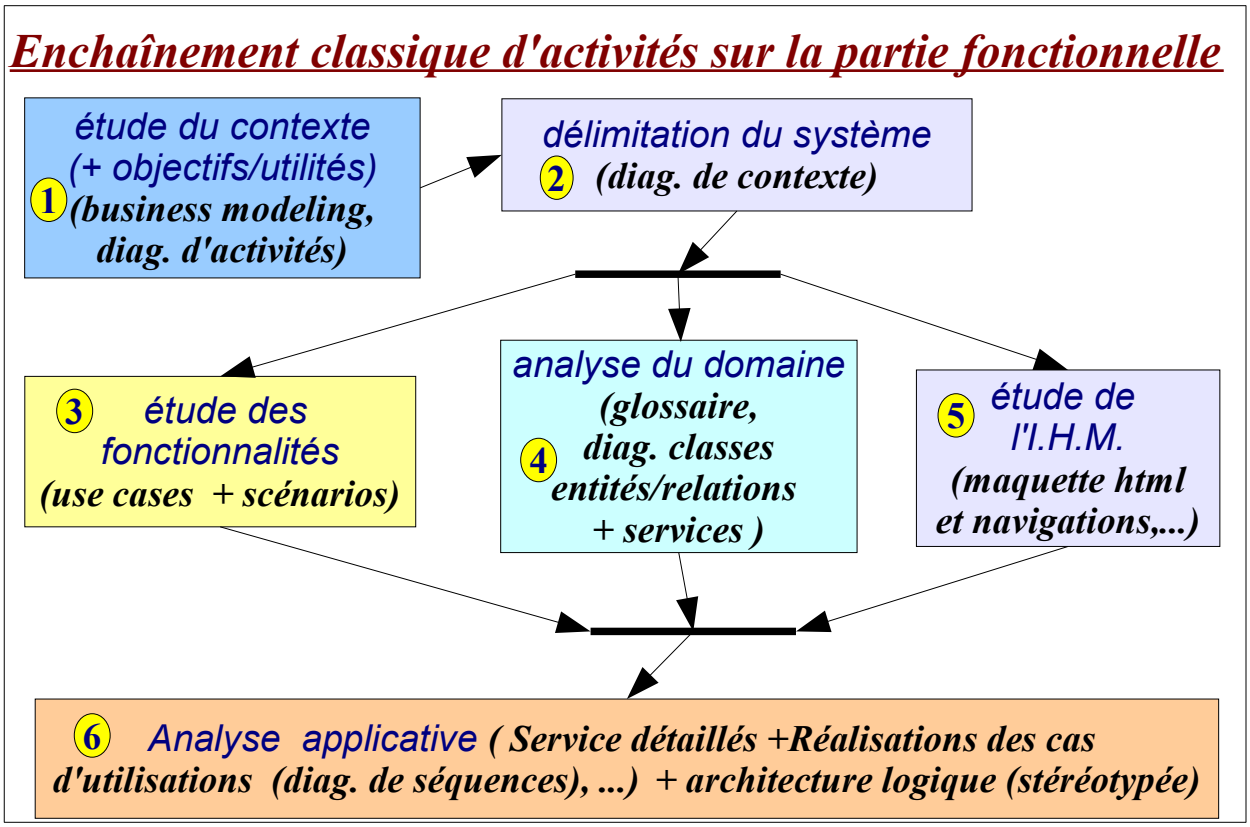
2. Cycle "itératif & incrémental"







3. Enchaînement des activités de modélisation



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, son auteur, son éditeur (avec identifiant ISBN).
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 événement "achat exemplaire neuf effectué" sera envoyé à un logiciel externe "système_comptabilité" de façon à générer un enregistrement comptable .

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 temporairement 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 aternative)

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 (MySQL , ...)
- IHM Web "HTML/css/js" + Api REST (ex : nodeJs, Php, java/spring-boot)
- Cloud/micro-services

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 BqeY).

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 (Etapes,...)
...

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)

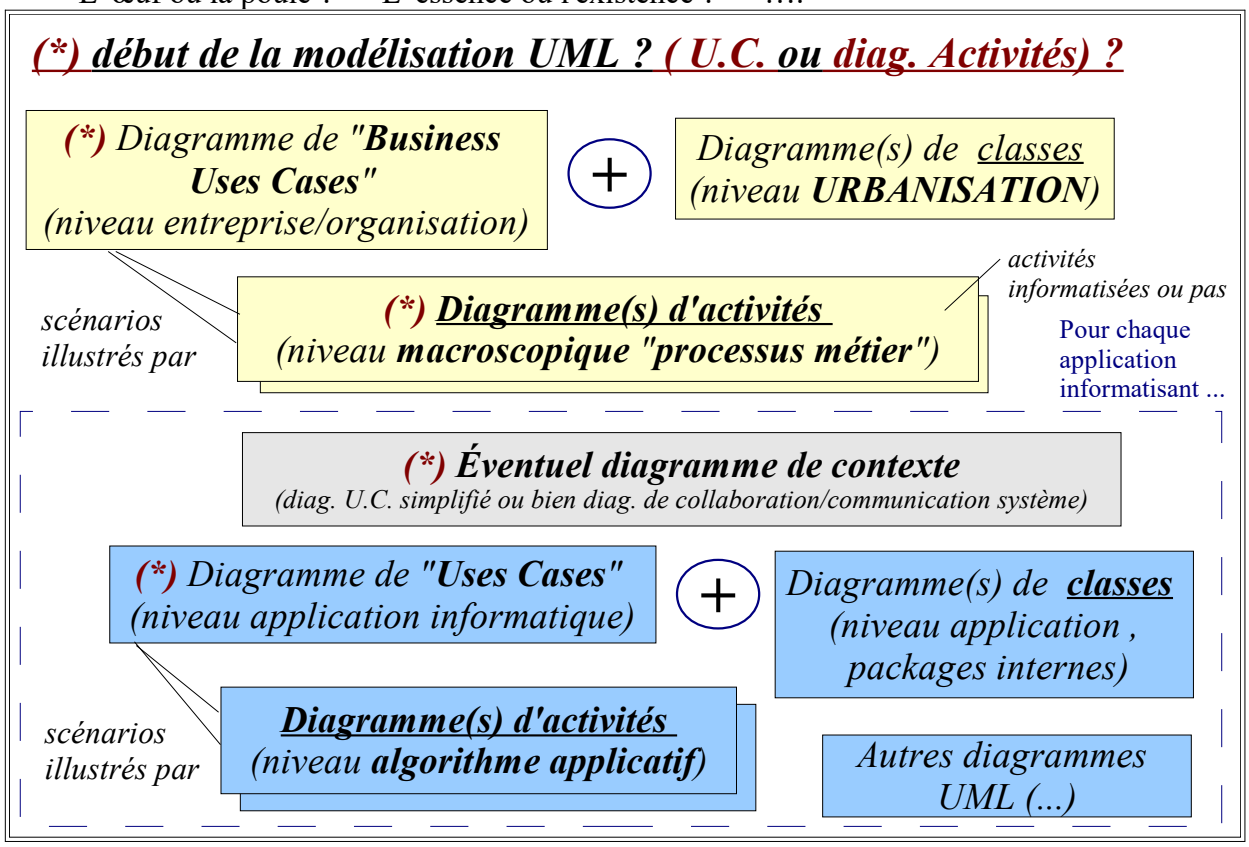
===> à proposer / réaliser (ex: location/réservation de voitures , ...).

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 ? ….



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**.

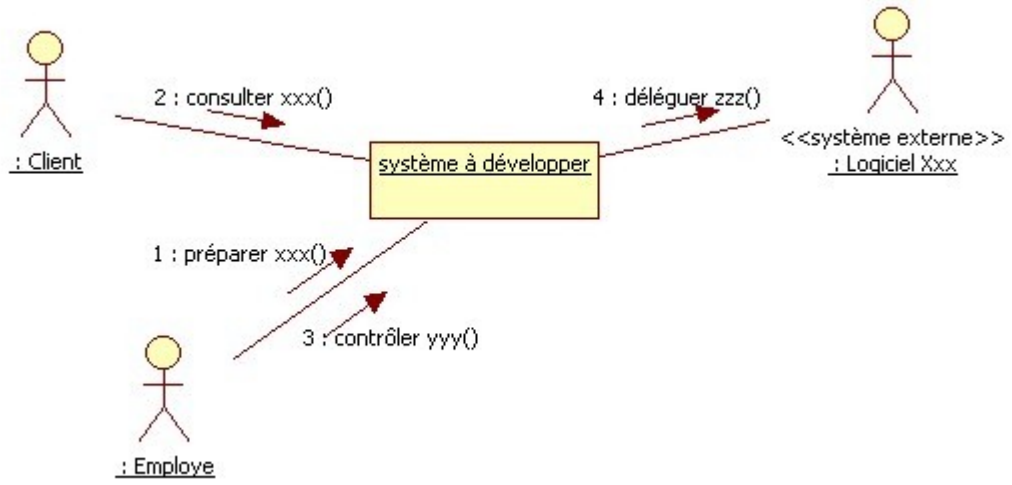
Bien que très utile, le diagramme de contexte n'est pas un diagramme officiel d'UML .
Selon les possibilités de l'outil UML on pourra concevoir le diagramme de contexte :

- 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 <<...>>

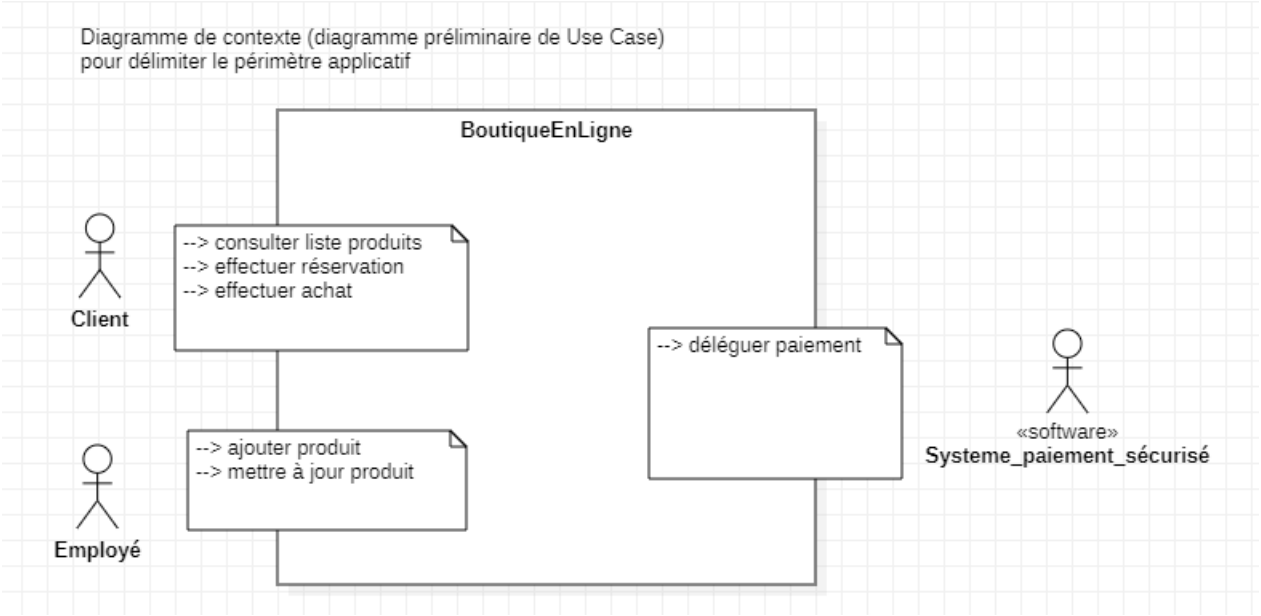
Les "messages" échangés à ce niveau sont souvent assimilés à des "***interactions" (systèmes ou utilisateurs)*** .

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



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

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



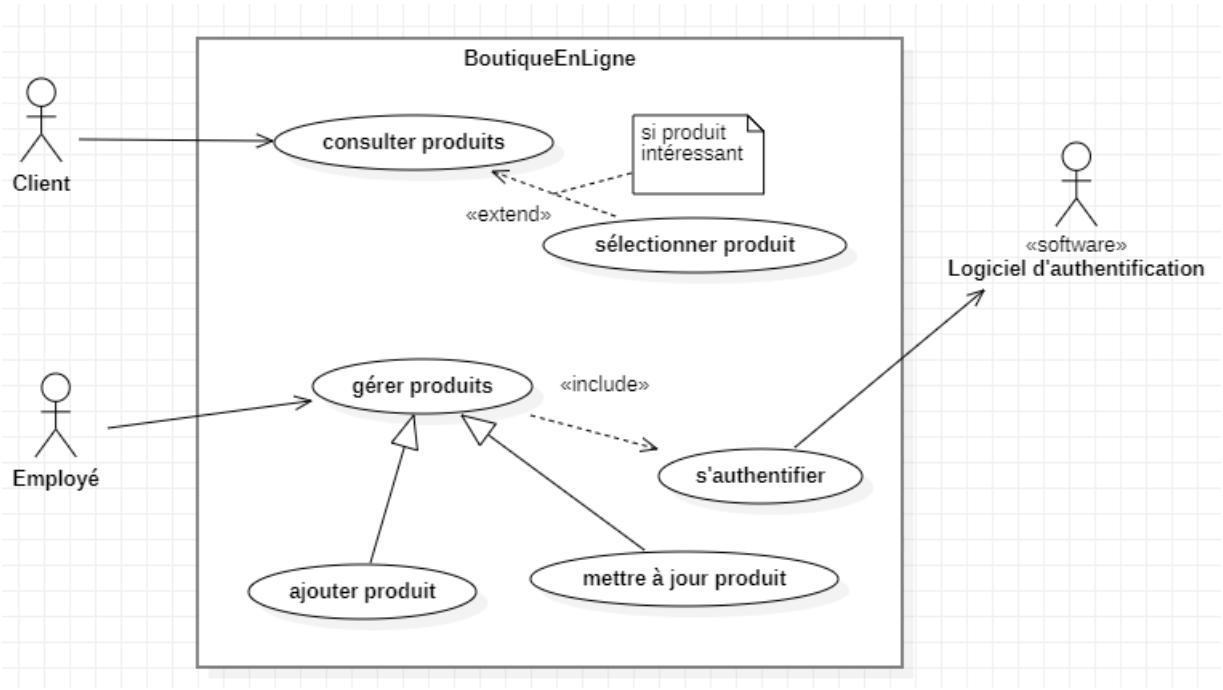
Bien que "non standard" / "non officiel" , un tel diagramme de contexte est **très rapide à élaborer** et donne une bonne vue d'ensemble sur les différents "futurs acteurs UML" et les interactions (entrantes ou sortantes) associées.

V - Expr. besoins fonctionnels / Uses Cases

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

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

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



NB:

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

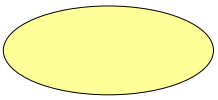
1.1. Méthodologie de base (au niveau des cas d'utilisation)

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)

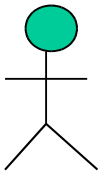
2. Diagramme des cas d'utilisations

Présentation des « Use Case »



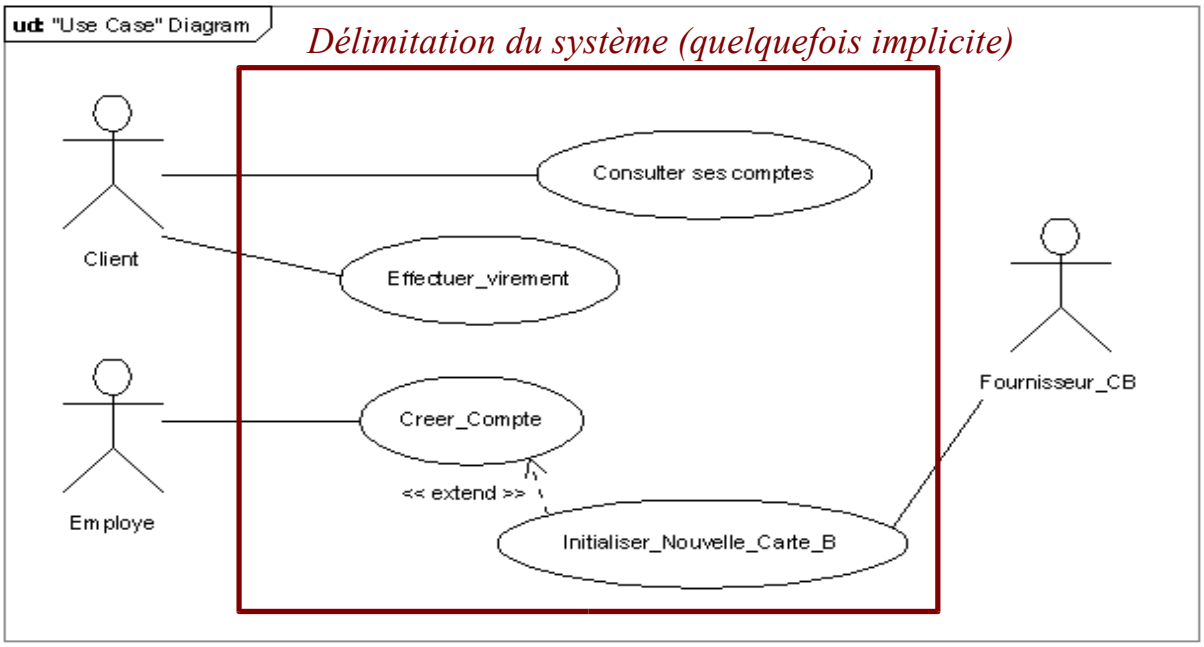
Faisant partie intégrante de UML, les "**USE CASE**" offrent un **formalisme** permettant de:

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



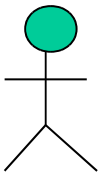
Nb: il s'agit d'une **vue externe** (***point de vue de l'utilisateur***).

Diagramme U.C. (Vue d'ensemble)



Acteurs

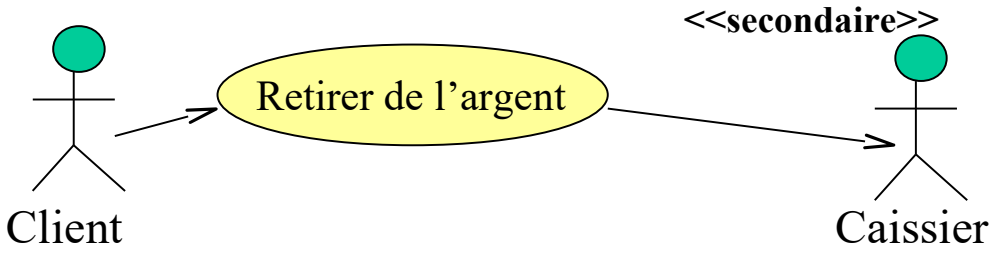
- Un **acteur** est une **entité extérieure au système** qui **interagit d'une certaine façon avec le système** en **jouant un certain rôle**.



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

Acteurs primaire et secondaire

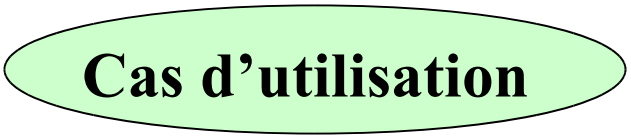
- Un cas d'utilisation peut être associé à 2 sortes d'acteurs:
  - **L'unique acteur primaire (principal)** qui déclenche le cas d'utilisation. **C'est à lui que le service est rendu**.
  - Les éventuels **acteurs secondaires** qui **participent** au cas d'utilisation en apportant une aide ***quelconque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



Cas d'utilisation

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

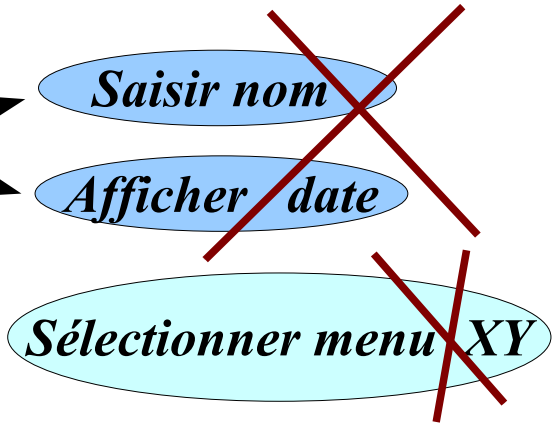
Notation:



A ne pas faire:

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

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



A prendre en compte:

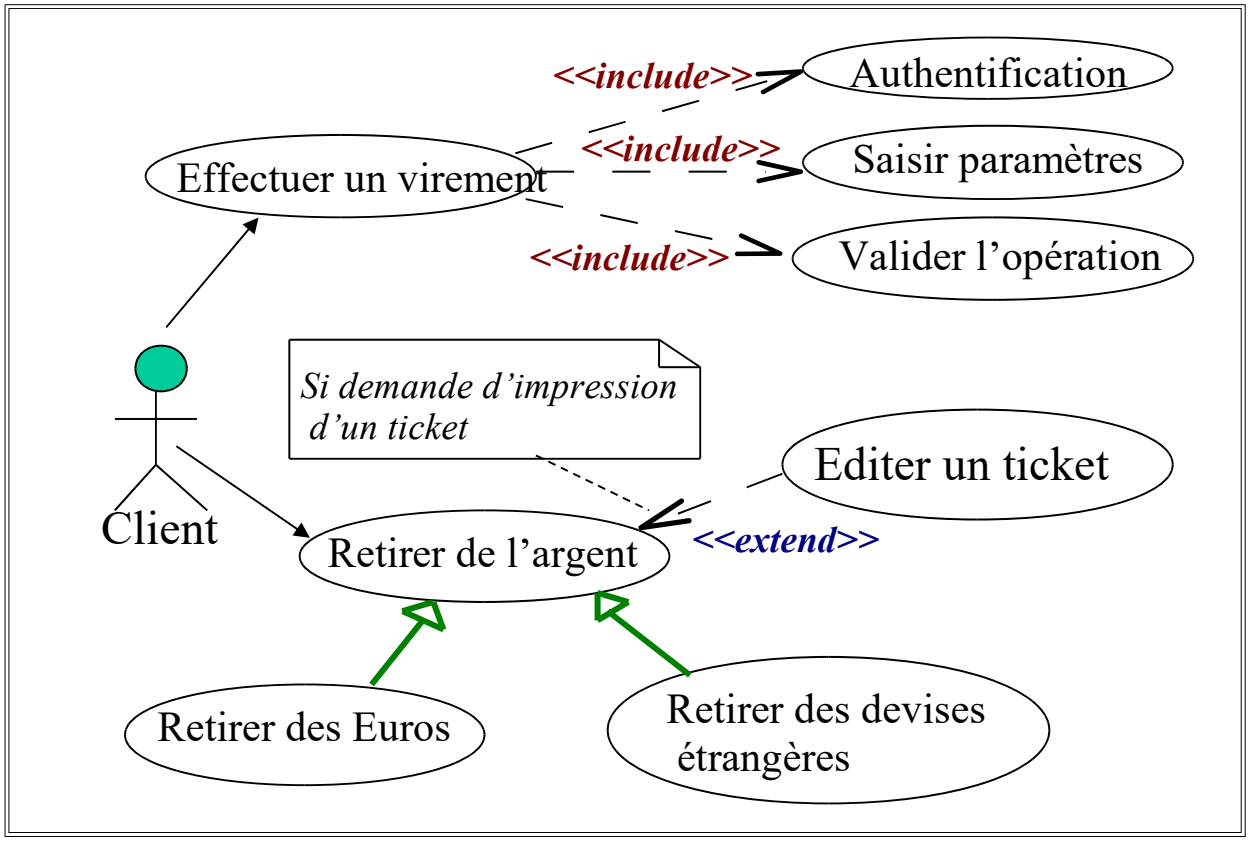
* cas d'utilisation <--> session utilisateur
  --> On peut donc relier entre eux les U.C. Effectués à peu près au même moment mais on doit séparer ce qui s'effectue à des instants éloignés (différentes sessions)
* cas d'utilisation --> avec interaction avec l'extérieur .

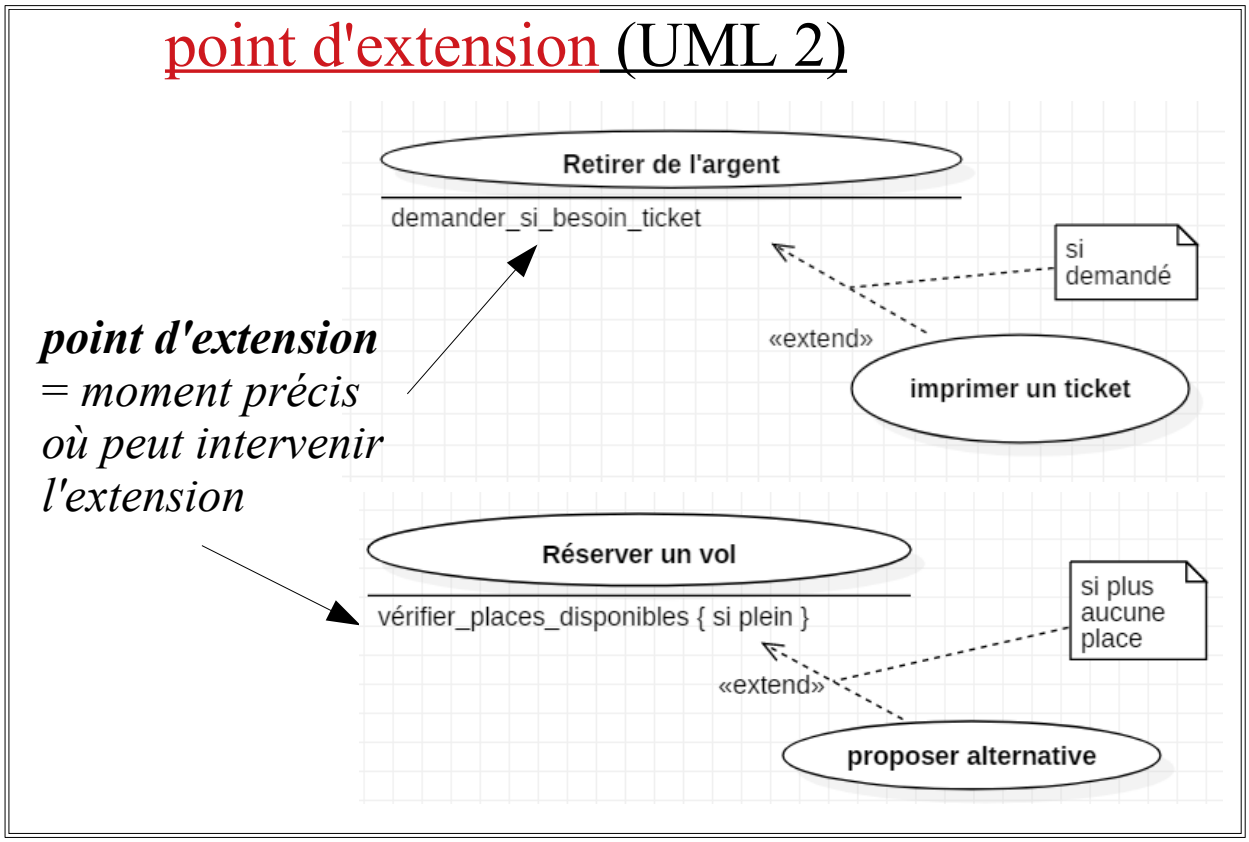
Relations entre UC

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

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

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

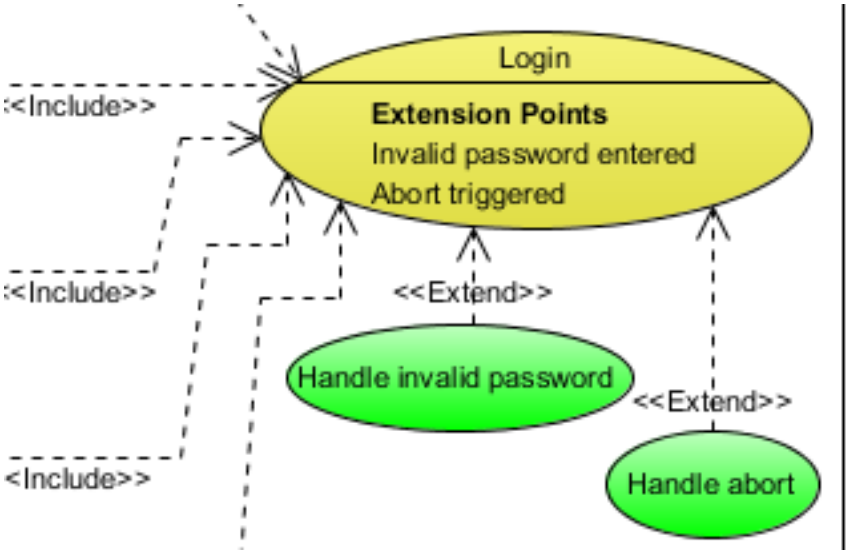




NB :

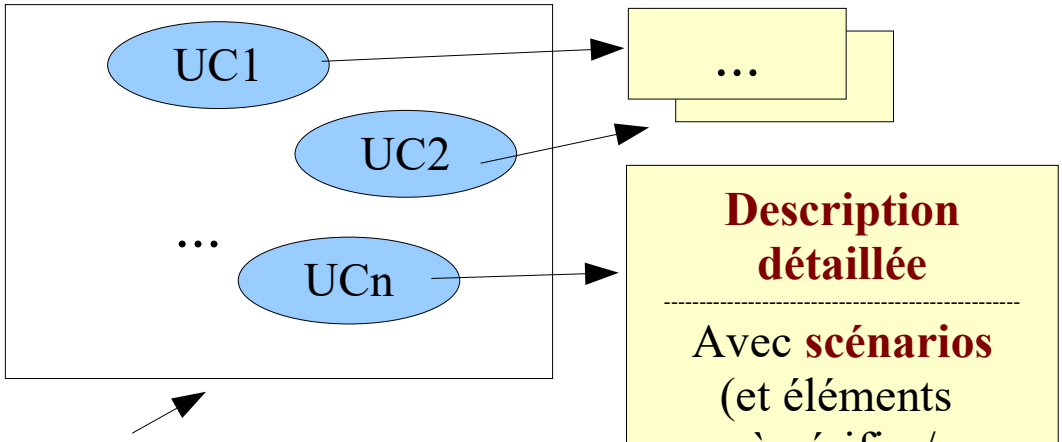
- Au sein d'un modèle UML (où tout est plus ou moins facultatif), un point d'extension bien formulé (ex : demander_si_besoin_ticket) permet de déduire le cas d'extension évident (ex : imprimer un ticket) qui peut alors ne pas être représenté (car considéré comme implicite) .

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



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

Pour chaque cas d'utilisation



Spécifications fonctionnelles / reformulation du cahier des charges

Description textuelle (U.C.)

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

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

Scénarios (U.C.)

Scénario nominal:

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

Scénario Nominal

Tout se passe bien

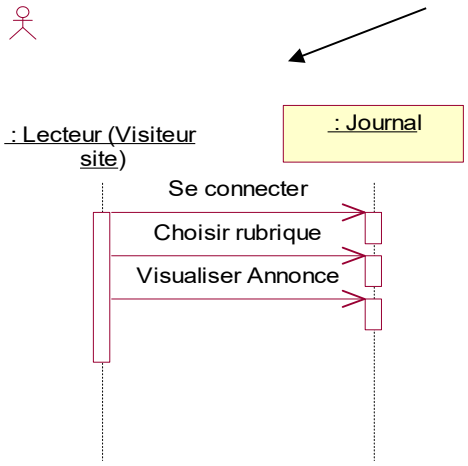
Cas/Déroulement le plus fréquent

Scénario d'exception "E1":

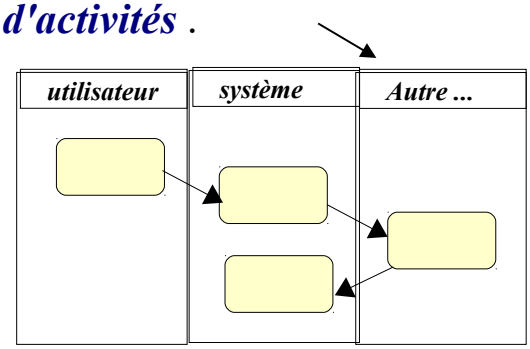
si étape l'étape (3) échoue trois fois de suite alors
- avaler carte
- ne pas effectuer les étapes (4) à (7)
- afficher un message explicatif

Illustration éventuelle (U.C.)

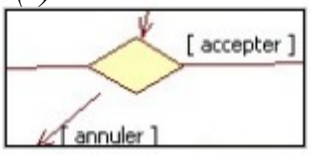
Un scénario peut éventuellement être graphiquement exprimé/illustré par un ***diagramme de séquence UML***.



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



*Avec **décision**(s) :*



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

NB :

Un scénario peut :

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

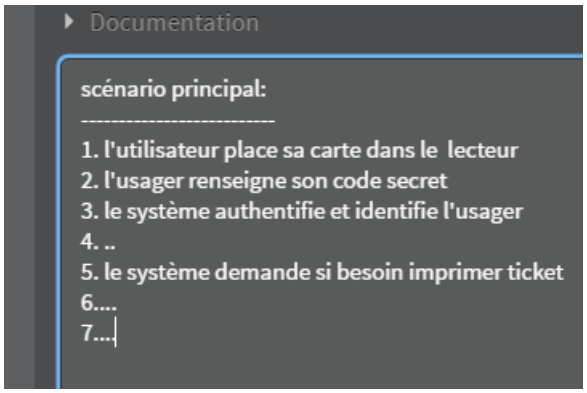


Tableau récapitulatif des U.C. (base pour la planification)

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

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

4. 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 ($1^{er} > 2^{e}$…)

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

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

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

5. 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)	**fin_UC4** , **UC9** ,UC3 , UC7
Livrables prévus sous les 20 jours suivants (globalement 60 jours après le début)	**UC5** , UC2 , UC8

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)	**UC4** , **UC9** ,UC3
Livrables réalisés et livrés sous les 20 jours suivants (globalement 60 jours après le début)	**UC10, fin_UC7 , UC5** , UC2
Eléments jamais livrés ou bien livrés plus tard si avenant et budget .	UC8 ***(le moins prioritaire !!!)***

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

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

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

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

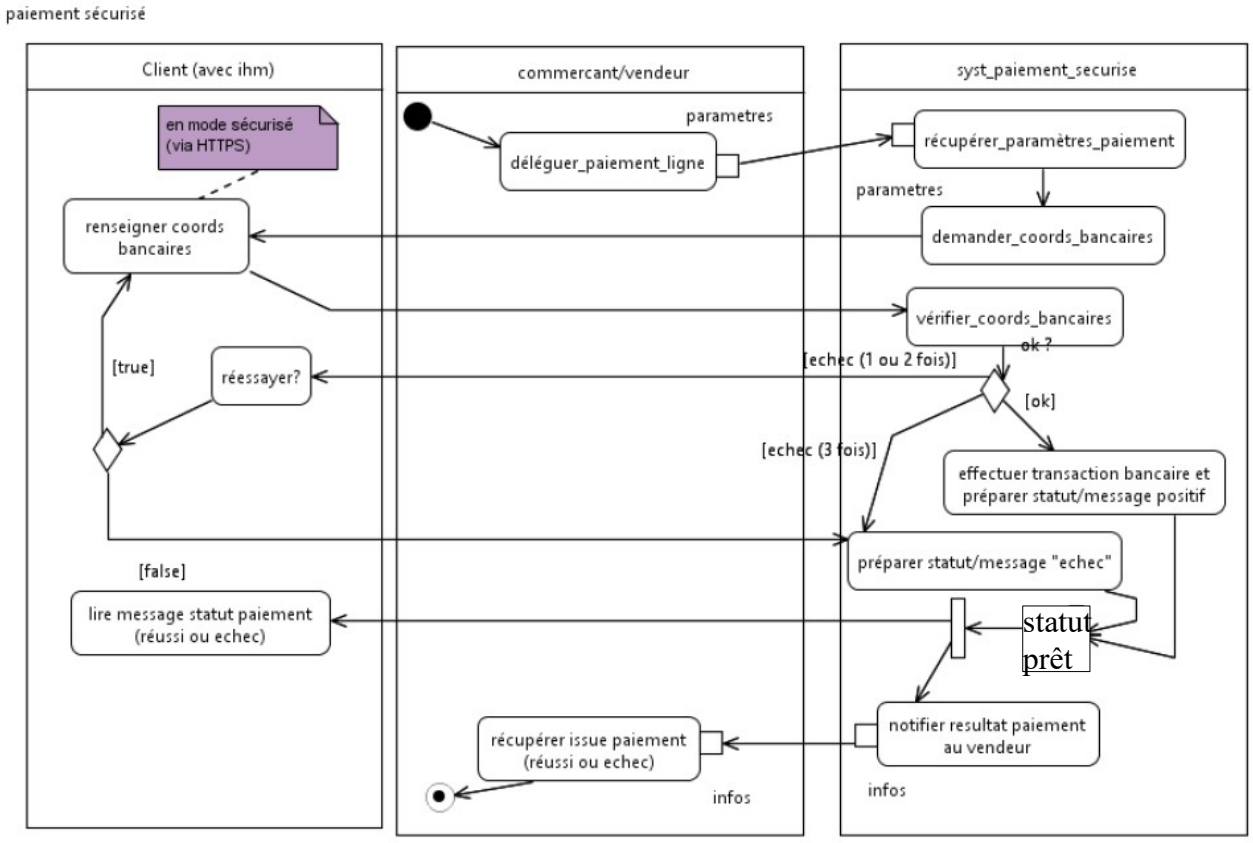
Dans l'arborscence d'un modèle UML, un diagramme d'activité est assez souvent placé comme un détails d'un cas d'utilisation (use case).

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

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

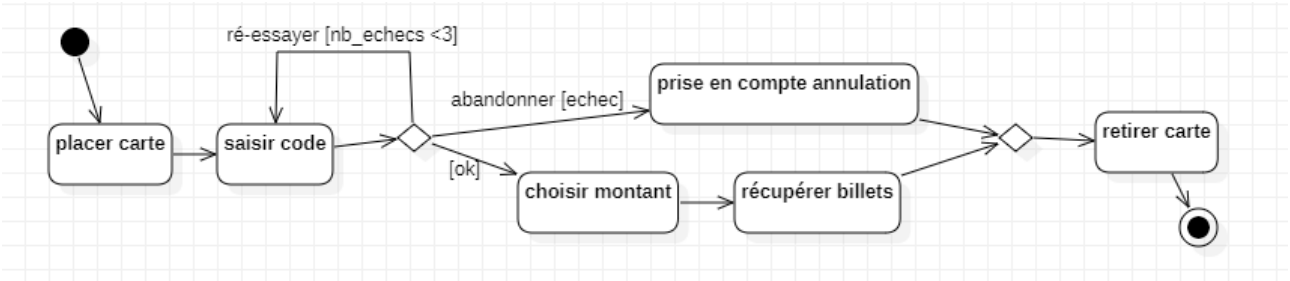
1.2. couloirs d'activités (facultatifs)

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



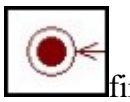
Au sein de l'exemple ci dessus les petits rectangles blancs sur les bords des actions correspondent à des "output pin" et "input pin" associés à des "object flows".

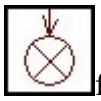
Exemple de diagramme simple (sans couloir) :



1.3. Noeuds spéciaux et de contrôles

 Initial/début *(normalement unique)*

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

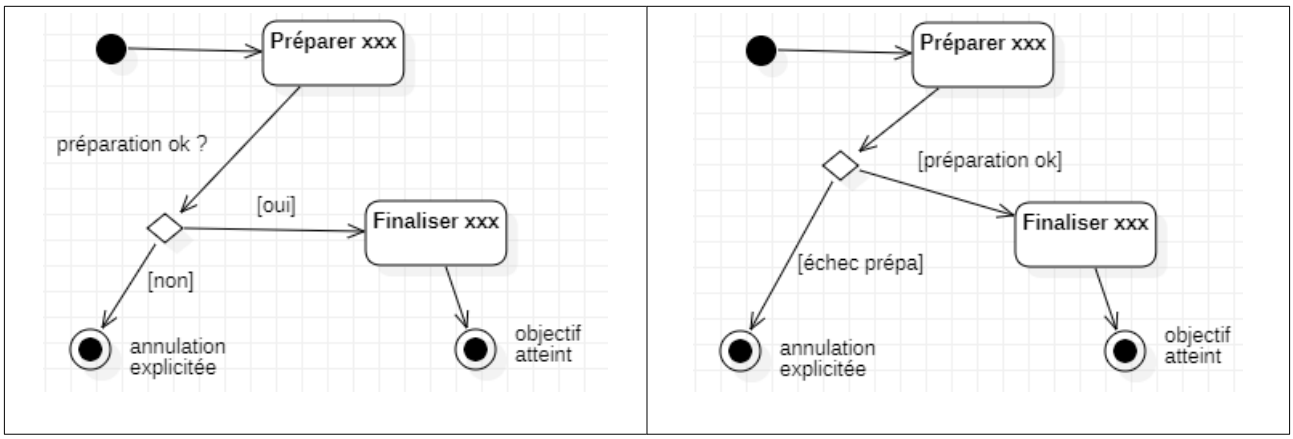
 fin de flot/branche *(ex : fin d'exécution d'un thread ou d'une tâche parallèle)*

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

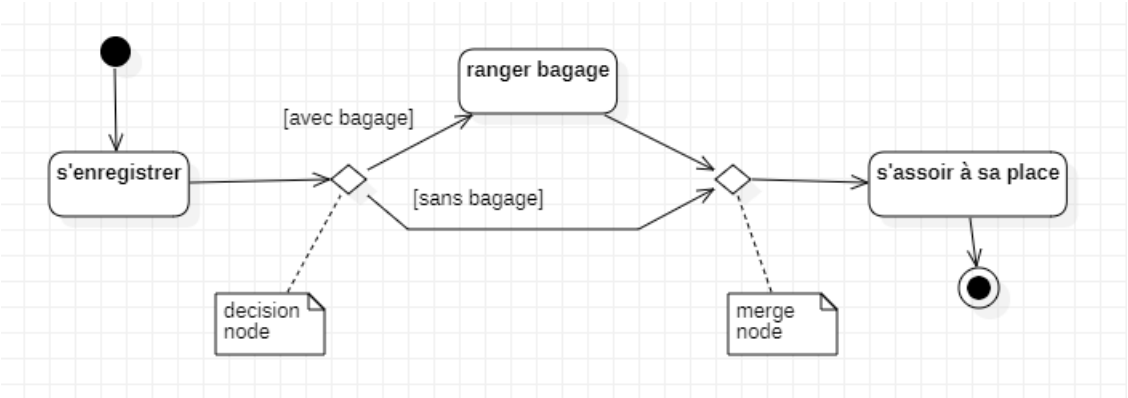
 union/**synchronisation de type "et logique"**

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

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



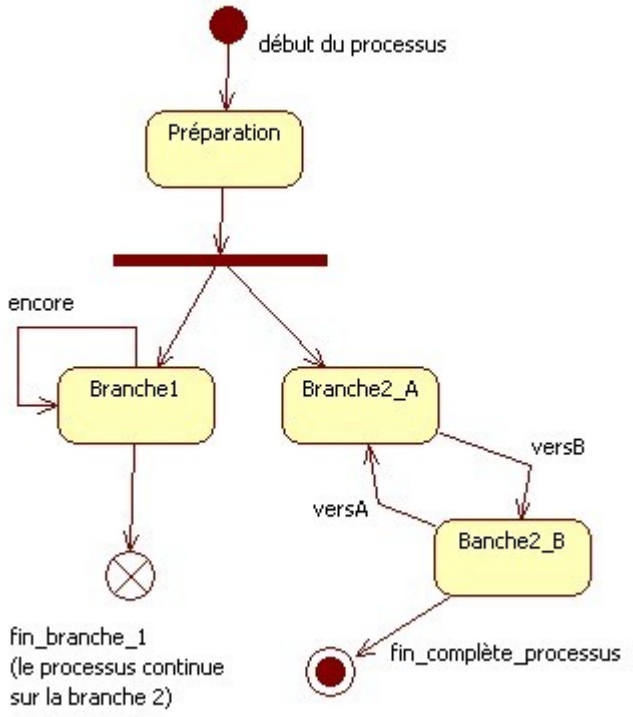
Selon l'outil UML , le texte de la décision peut être placé soit sur le losange de décision, soit sur la flèche entrante , ou encore être implicite (non affiché).
Sur les flèches sortantes du losange, les conditions à respecter ("guard") sont normalement automatiquement encadrées par des crochets.



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

1.4. Final flow

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



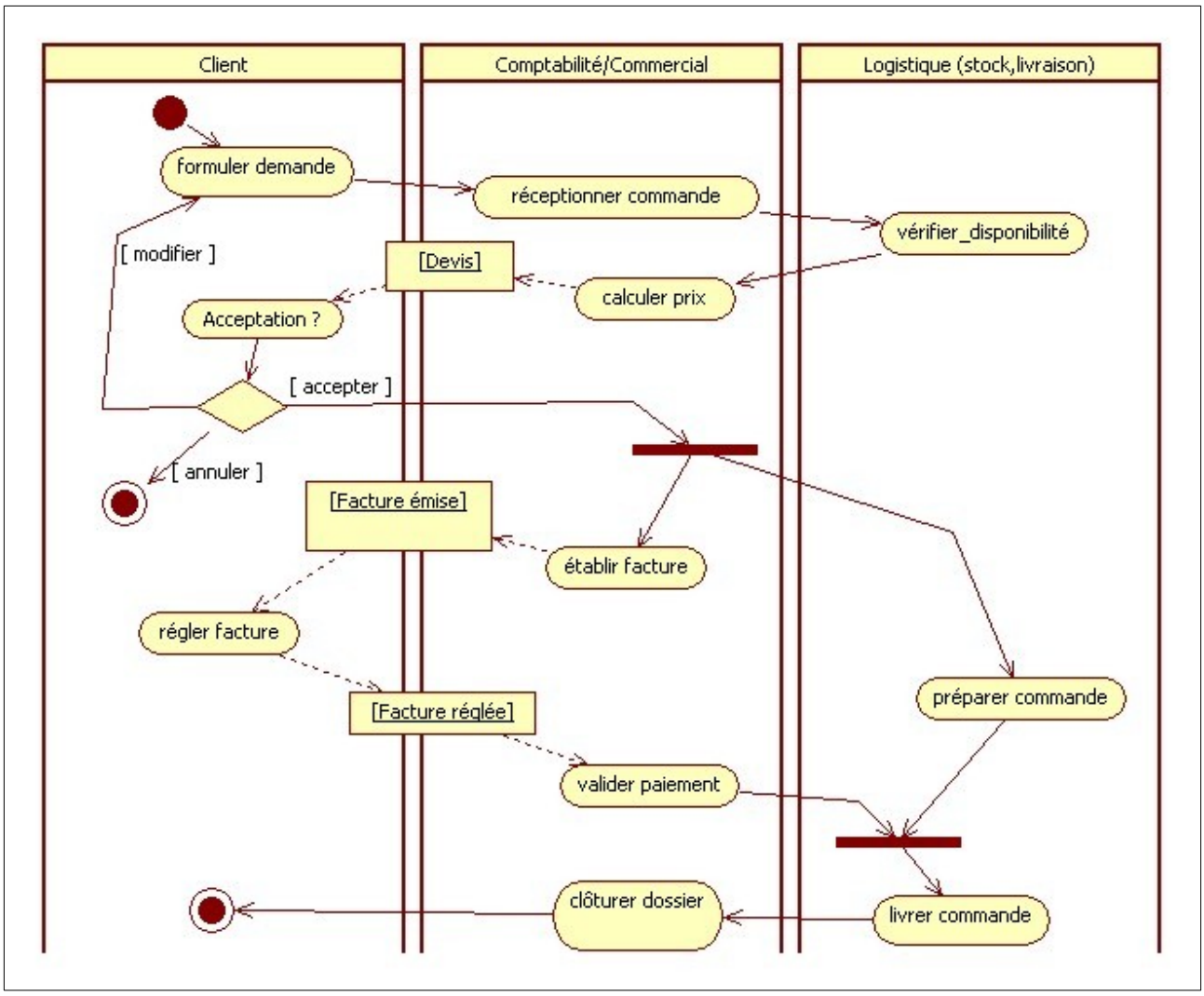
1.5. object flow (nœuds "objet") : UML 1 et 2

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

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

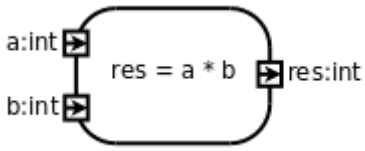
Un **rectangle** comportant **[le nom de la donnée et son état]** entre crochet.

Exemple (UML 1 et 2):

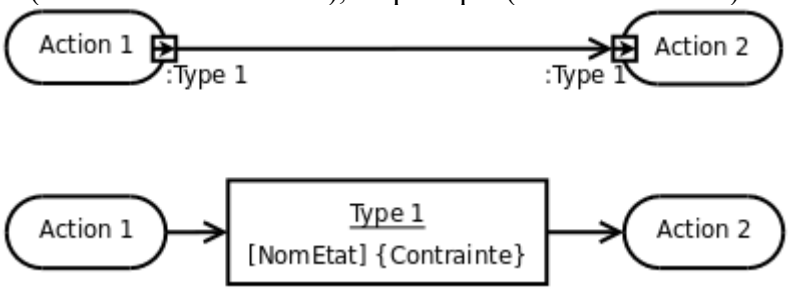


1.6. Pins et buffers (UML 2)

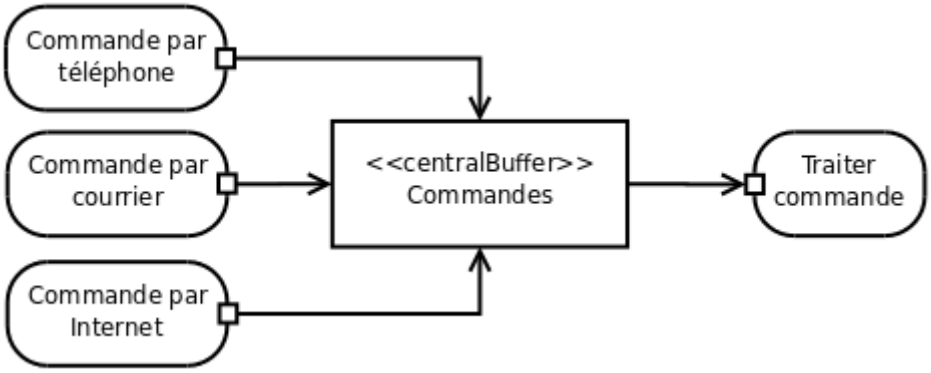
Depuis la version 2 d'UML , il faut placer des points ("pin") d'entrée ou de sortie sur une activité pour préciser un lien (éventuellement typé) avec les "object flow" entrant(s) ou sortant(s).
[avec une sémantique de passage de valeur(s) par copie(s)] .



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>> .

2. Notations avancées pour (rares) diagrammes d'activités très détaillés ou très techniques

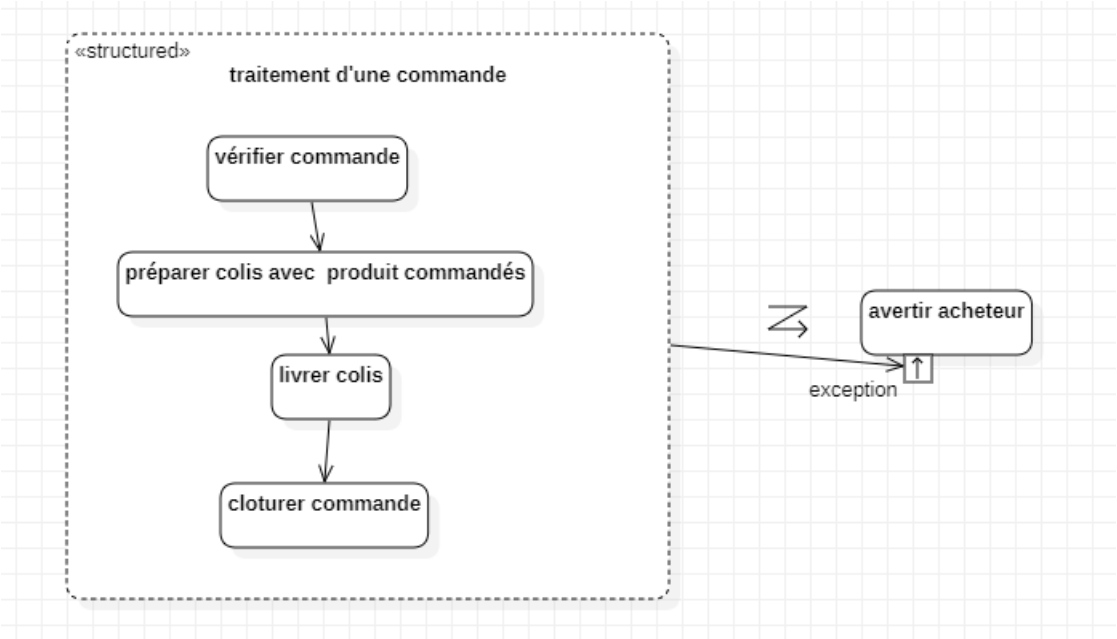
2.1. Distinction "micro activité / macro activité" selon granularité

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

- **macros activités** (**activity**):
  - ***durées*** potentiellement ***longues*** (plusieurs jours)
  - avec quelquefois 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)***
  - **…**

2.2. Diagramme d'activité structuré et avec exception



2.3. 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 :

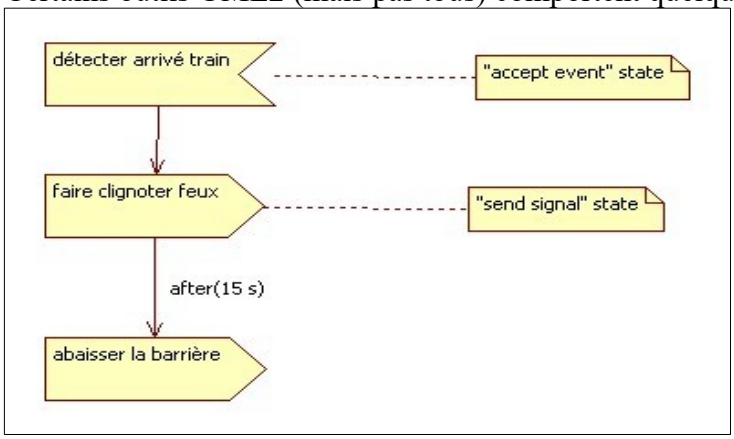
Opaque Action	Action/activité quelconque (dont la nature pourra être ultérieurement précisée/affinée en conception)
Call behaviour	Appel global d'une autre sous activité (sans mentionner une opération précise). Souvent associé à un lien hypertexte vers autre diagramme.

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

Call operation	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
Send	Envoi d'un message ou d'un signal
Accept event	Attente (bloquante) d'un événement (souvent lié à une notification asynchrone)
Accept call	Variante de "accept_event" pour les appels entrants (avec réponse à donner ultérieurement via reply)
Reply	Répondre (lié à un accept_call)
créer/instancier	Créer un nouvel objet
destroy	Détruire un objet (ex: delete en c++)
Raise exception	Soulever (remonter) une exception

2.4. Notations pour actions particulières (UML2)

Certains outils UML2 (mais pas tous) comportent quelques notations particulières



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 :

Références des règles de gestion	***Définitions de ces règles***
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 date_fin >= date_debut .
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 :

Références des règles de gestion	***Définitions de ces règles***
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.

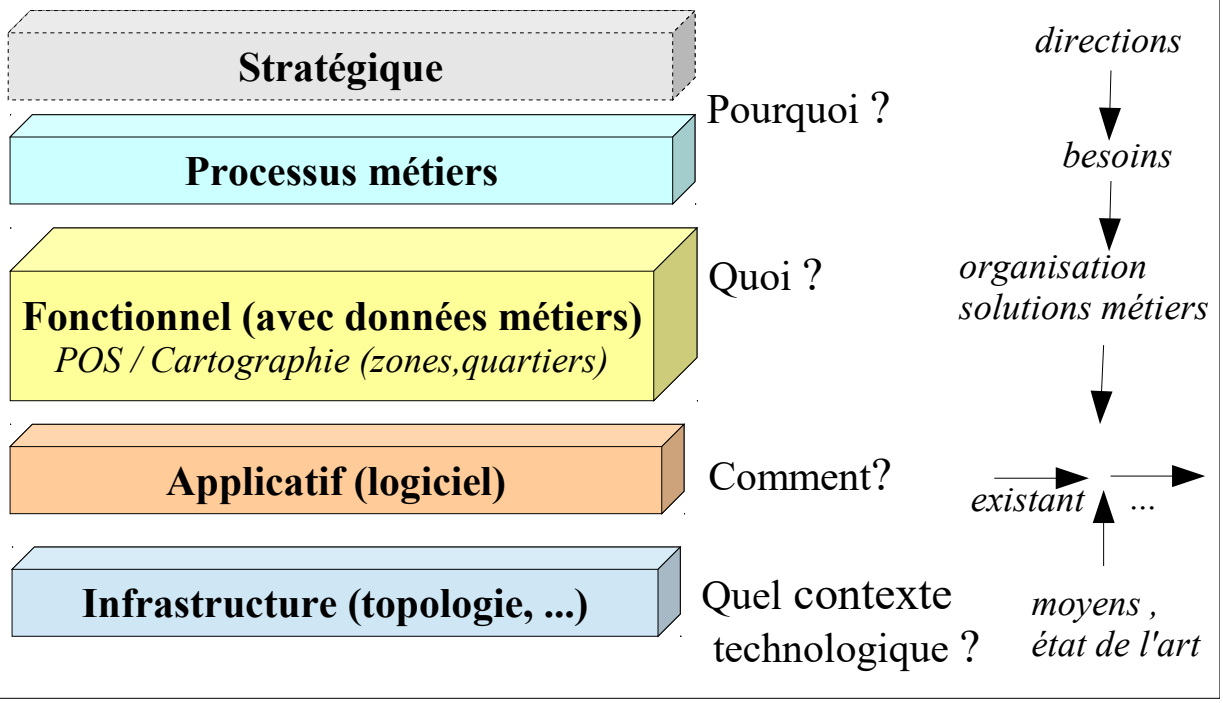
En pratique:
- Aucune norme sur le sujet (==> beaucoup de variantes selon outils)
- Documentation générée avec les moyens du bord (selon outils et scripts à disposition)
- Question clef : dépendre ou pas d'outils spécialisés (avec pérennités souhaitées)

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)

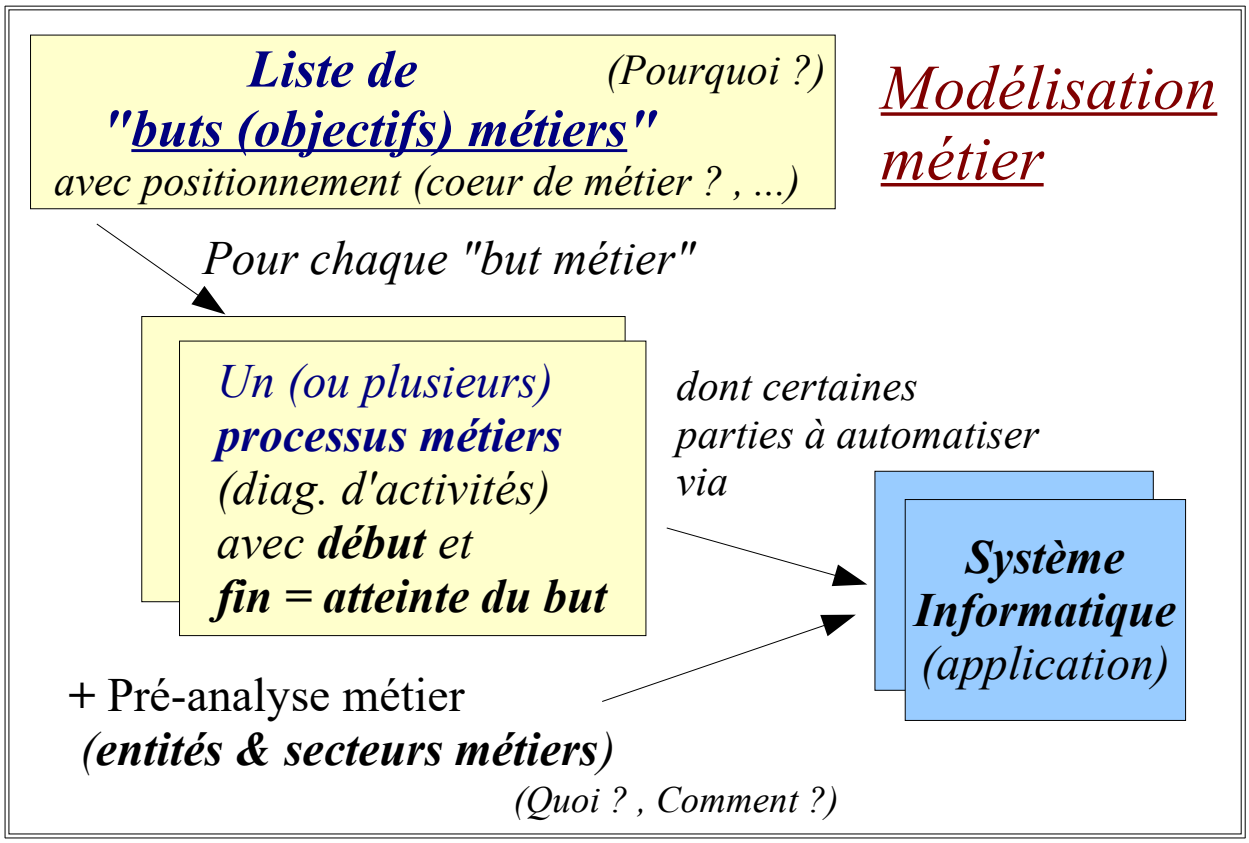
1. Rôle et contenu de la modélisation métier

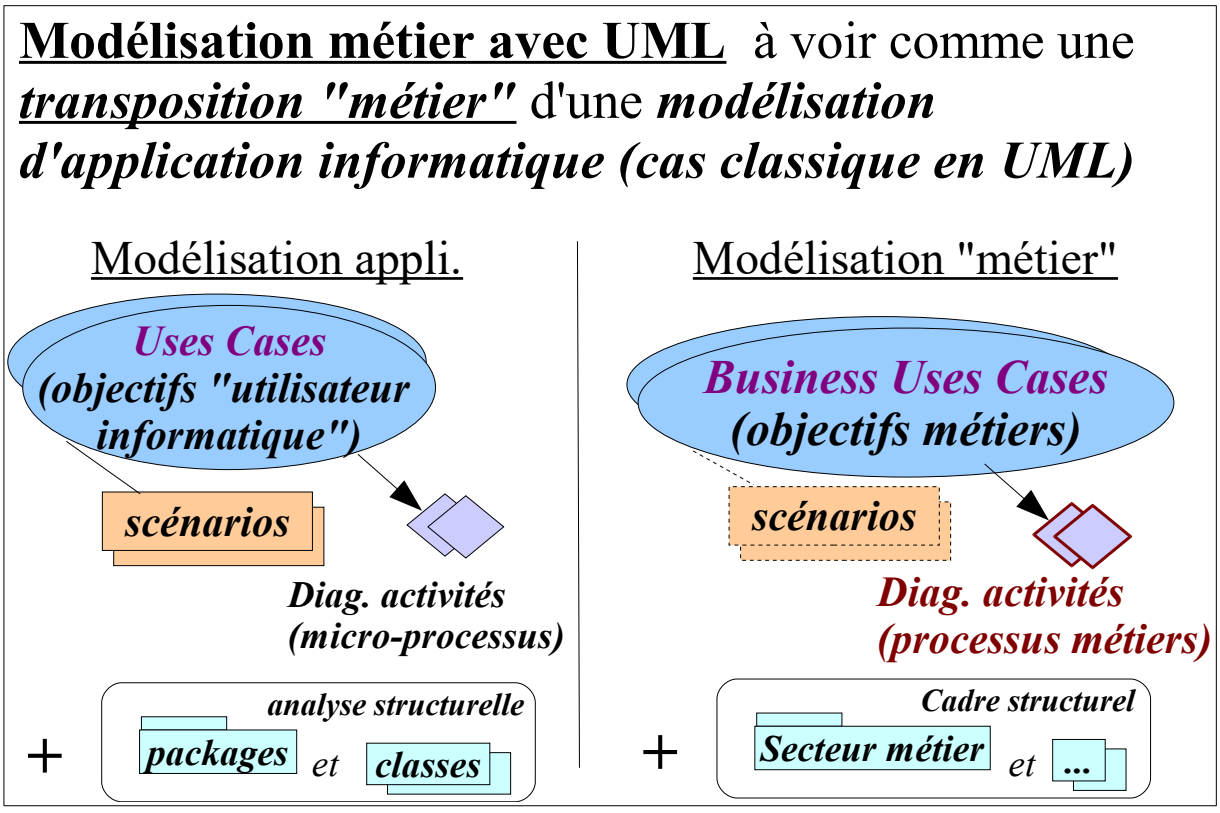
Cadre général (à 5 niveaux) pour l'urbanisation



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* .


Liste de "buts (objectifs) métiers"
(Pourquoi ?)
avec positionnement (coeur de métier ? , ...)
Modélisation métier
Pour chaque "but métier"
Un (ou plusieurs) processus métiers (diag. d'activités) avec début et fin = atteinte du but
dont certaines parties à automatiser via
Système Informatique (application)
+ Pré-analyse métier
(entités & secteurs métiers)
(Quoi ? , Comment ?)


Modélisation métier avec UML à voir comme une transposition "métier" d'une modélisation d'application informatique (cas classique en UML)
Modélisation appli.
Uses Cases (objectifs "utilisateur informatique")
scénarios
Diag. activités (micro-processus)
+
analyse structurelle
packages
et
classes
Modélisation "métier"
Business Uses Cases (objectifs métiers)
scénarios
Diag. activités (processus métiers)
+
Cadre structurel
Secteur métier
et
...

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:

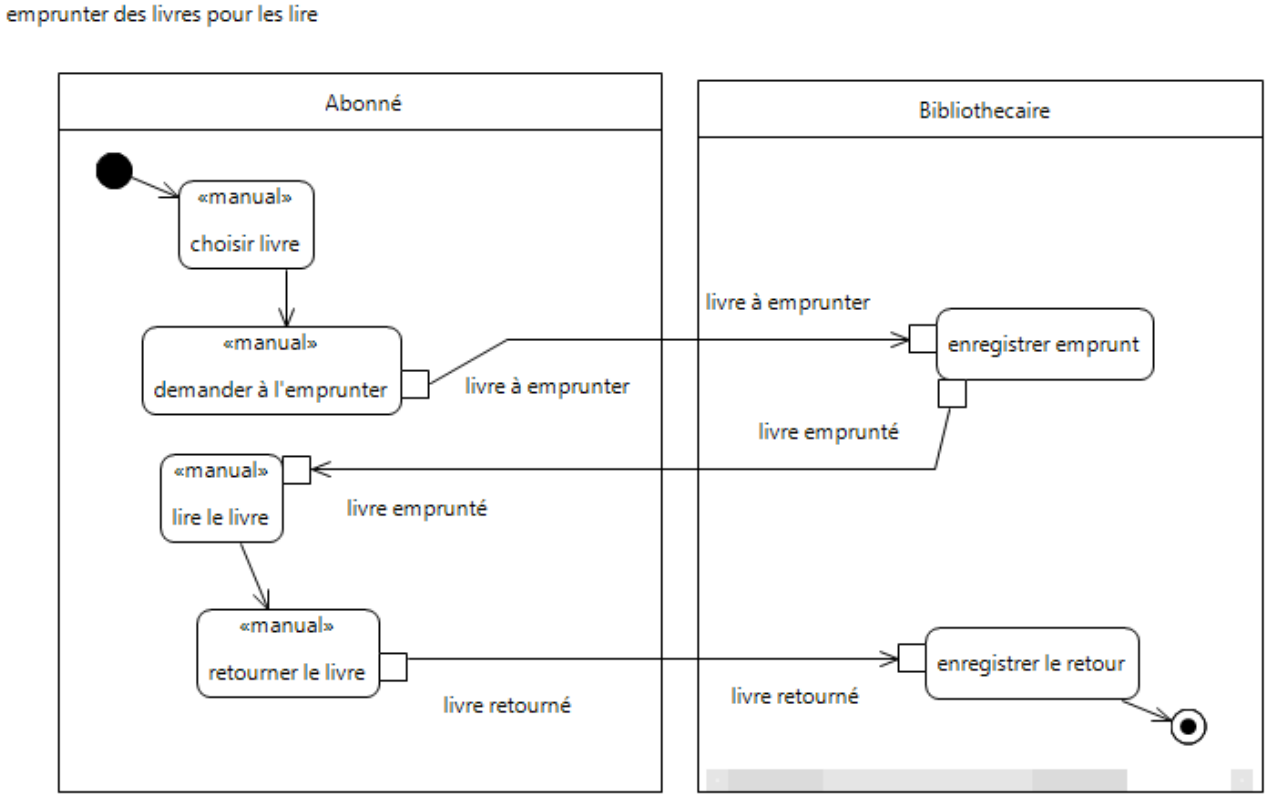
- comprendre l'***utilité*** du système à modéliser/développer
- ***montrer ce qu'il y a autour*** (autre(s) système(s) , catégorie(s) d'utilisateurs , ...)
- ***cadrer le système*** (quelles grandes responsabilités/fonctionnalités, périmètre applicatif)

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.

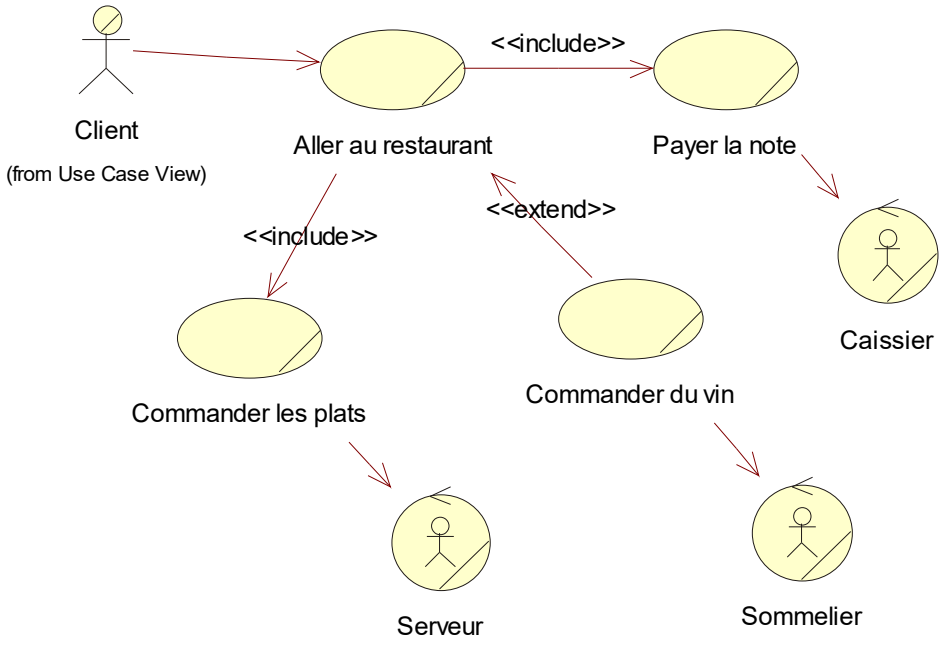
Remarque importante:

- 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*** .



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".

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 = **I**nterface **H**omme **M**achine
GUI = **G**raphical **U**ser **I**nterface

1.1. Maquette

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

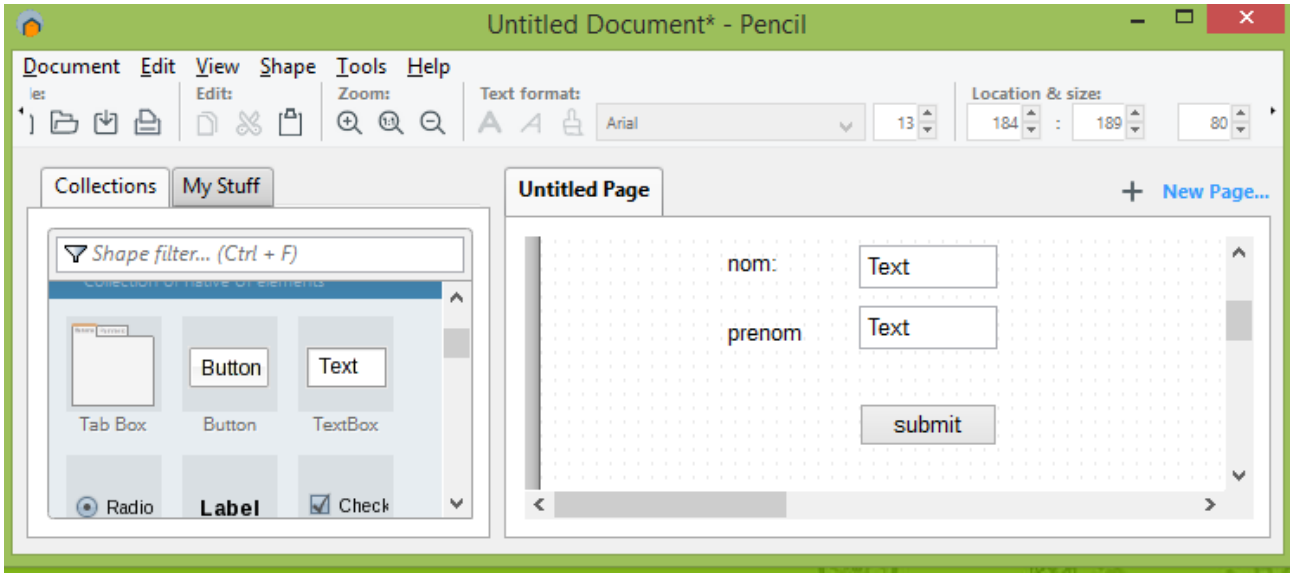
- dialoguer avec le commanditaire du système (maîtrise d'ouvrage) et les utilisateurs
- valider les fonctionnalités attendues et leurs enchaînements.
- valider les grandes lignes de la charte graphique
- penser à des détails que l'on aurait oubliés

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 :

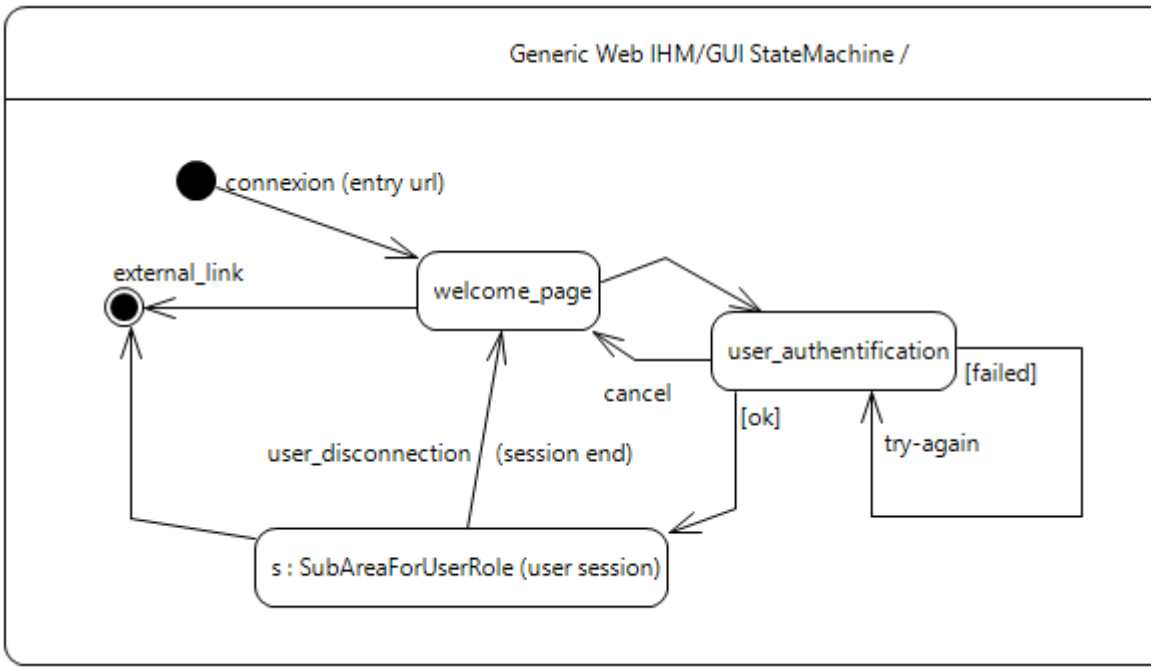
- ce qui existe
- les enchaînements prévus (autorisés ou pas , ...)
- ….
-

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.

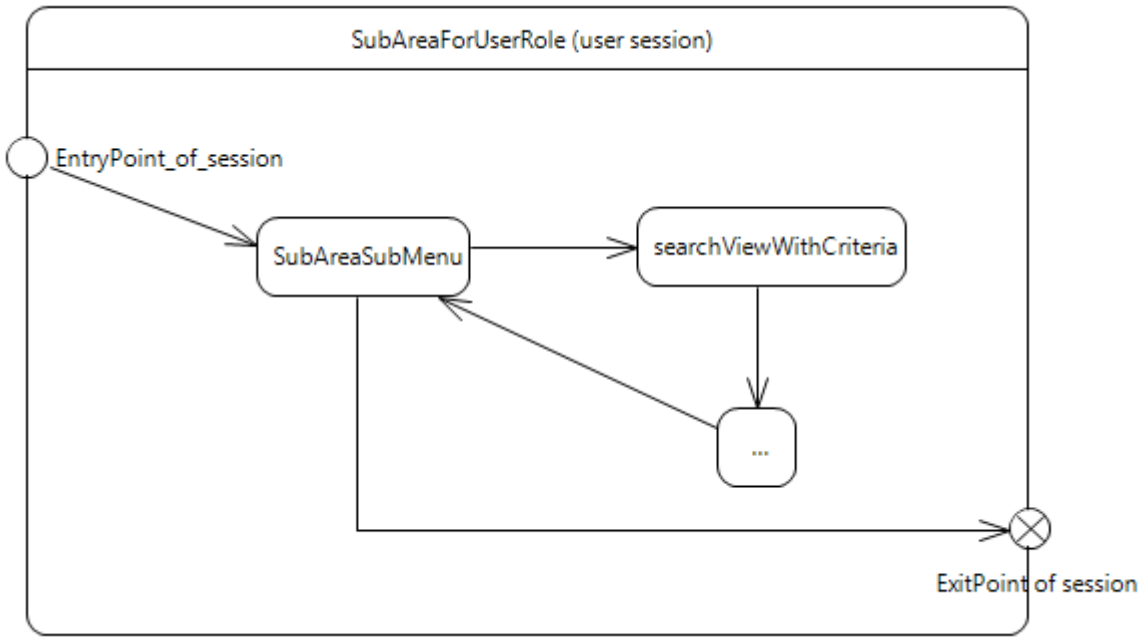
Au sein des diagrammes UML existants , c'est le diagramme d'états (StateChart) qui est le plus approprié pour modéliser des navigations .
L'IHM passe d'un état à un autre lors d'un changement de page ou d'écran.
Une transition (changement d'état) est associée à un événement (ex : clic sur lien hypertexte) 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 :



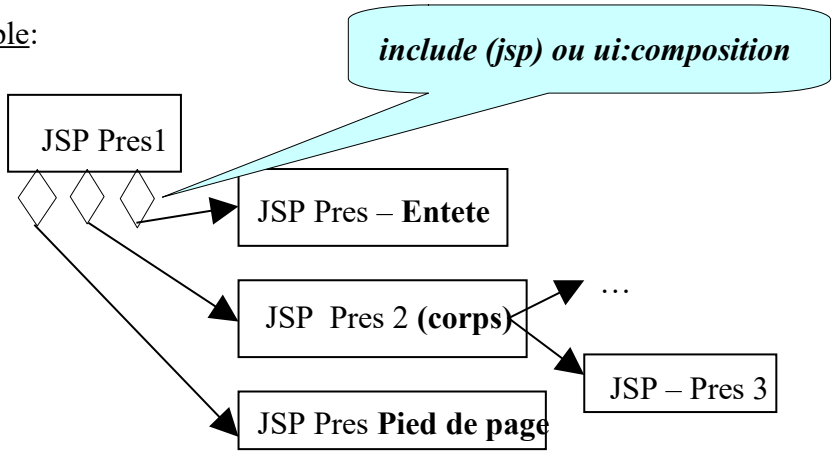
Pertinence (à relativiser en fonction du contexte) :

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-congif.xml* ou *faces-config.xml* dans le monde java).

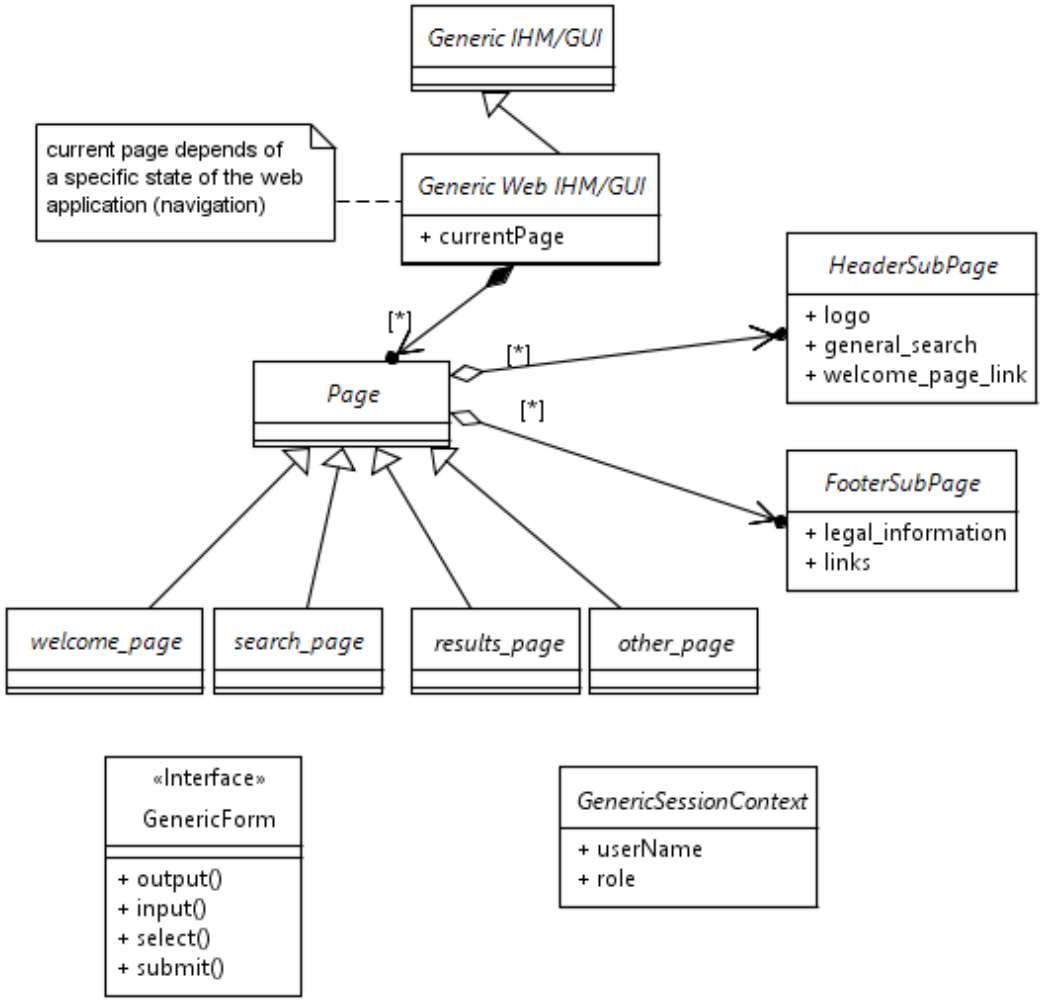
1.3. Modélisation composite et générique des écrans (modèles)

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.



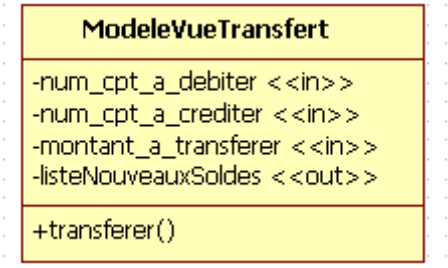
Conséquences :

- 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

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

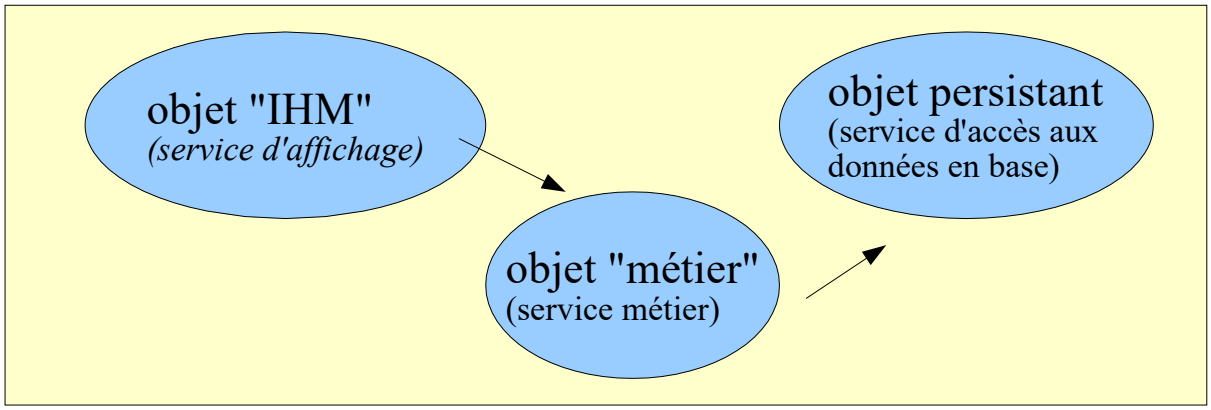
--> approfondir si besoin en cours avec le formateur.
--> approfondir si besoin ensuite via Wikipédia ou "comment ça marche" ou autre.

Avant tout,

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

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

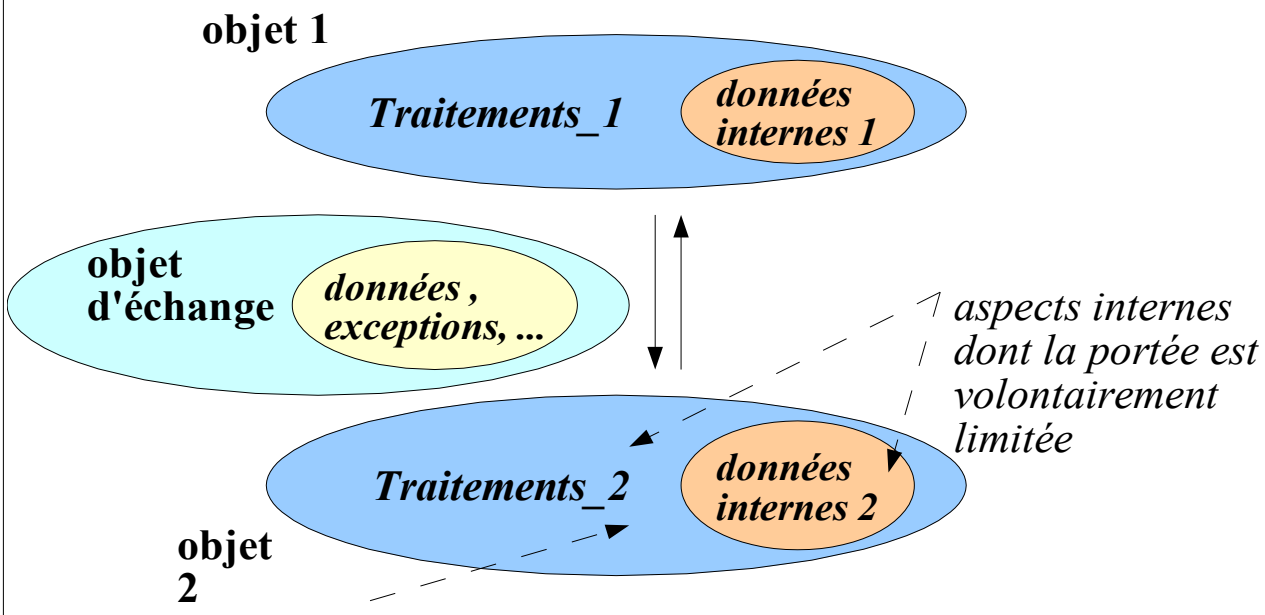
Un objet interagit avec un autre en lui envoyant des messages de façon à solliciter certains services.



Grand traits de l'approche objet:

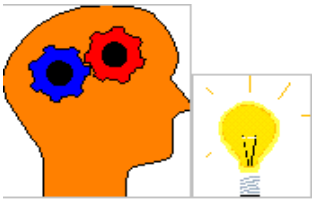
Un programme est un ensemble de petites **entités** . Chacune ayant :
- son propre **état** *(lié au **données internes** et aux inter-relations)*
- son propre **comportement** *(**traitements internes** pour fonctionner)*

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



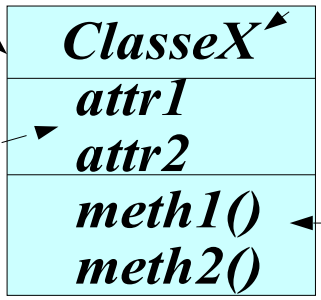
Traitements** en grande partie **locaux** ==> assez bonne **autonomie

Abstraction des données (réalité => types abstraits)



abstraction

De quoi parle-t-on ?



Eléments pertinents ?

Quelle utilité ?
Quels services ?

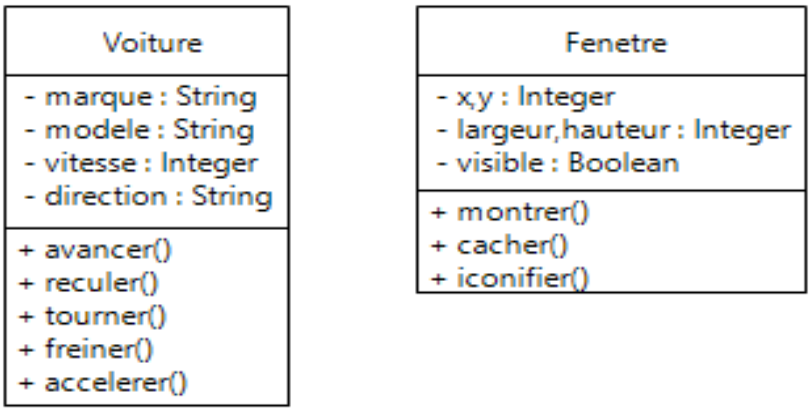
*Le **mécanisme d'abstraction** consiste à **créer ses propres types, appelés types abstraits** (ou classes) de façon à les intégrer dans une modélisation (vue simplifiée) du monde réel .*

instances

Notion de classe

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

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

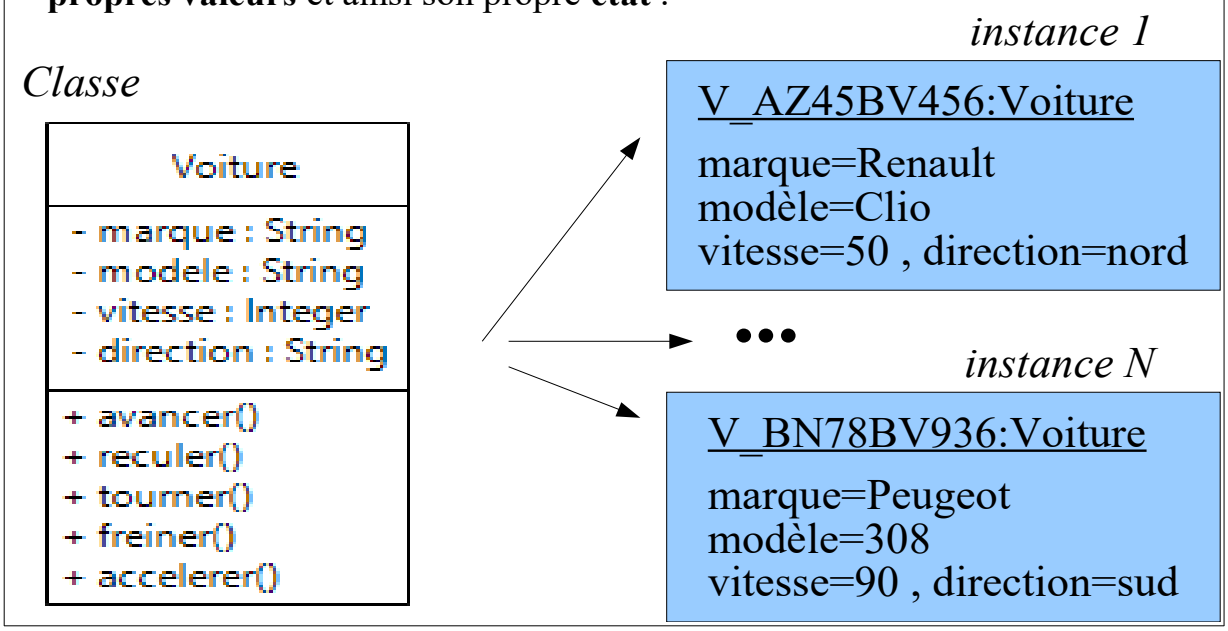


Toutes les instances d'une même classe ont une même structure commune et ont un même comportement.

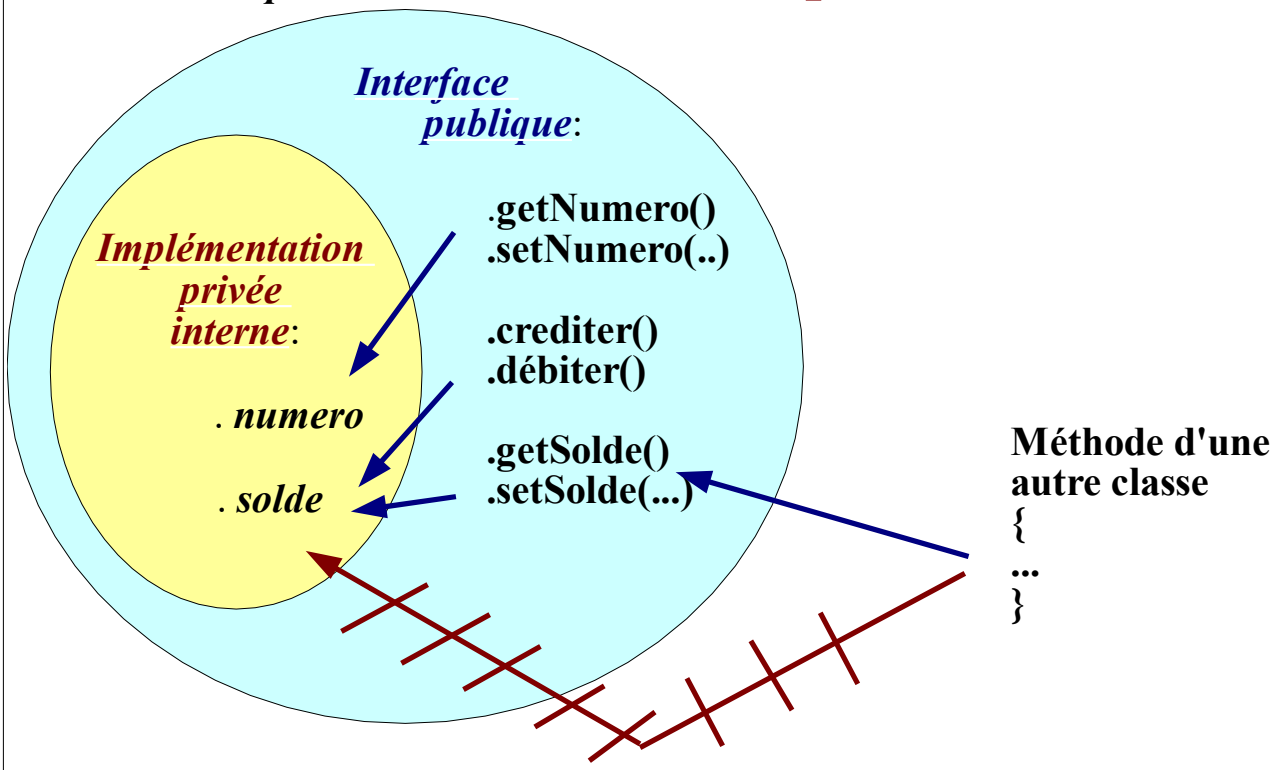
Instances

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

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

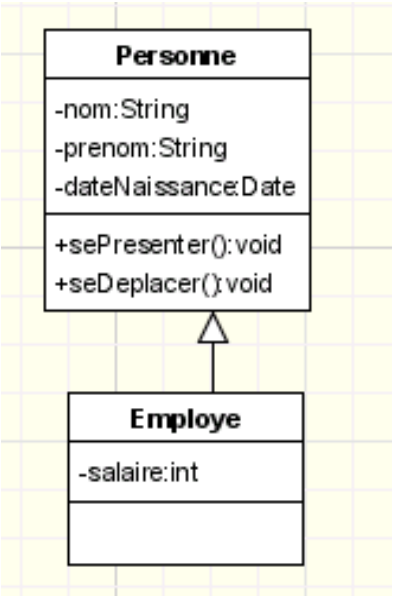


Encapsulation



L'accès direct aux attributs privés est interdit

Héritage / Généralisation / Spécialisation

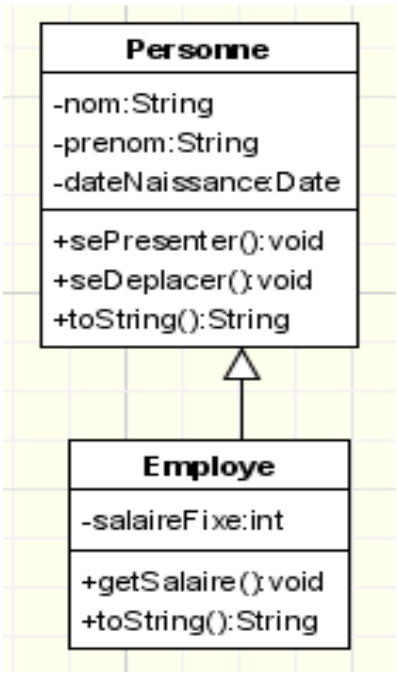


*** La notion d'héritage consiste à définir une nouvelle classe à partir d'une classe existante en spécialisant certaines choses.**

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

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

Héritage – code Java

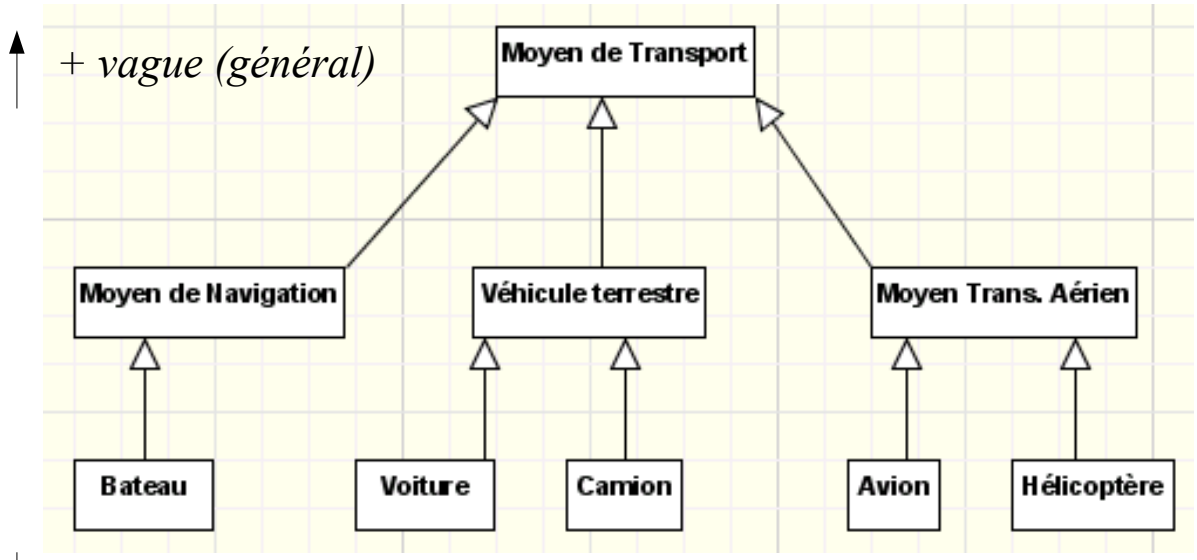



```
public class Personne {
    private String nom;
    private String prenom;
    private java.util.Date dateNaissance;

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

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

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

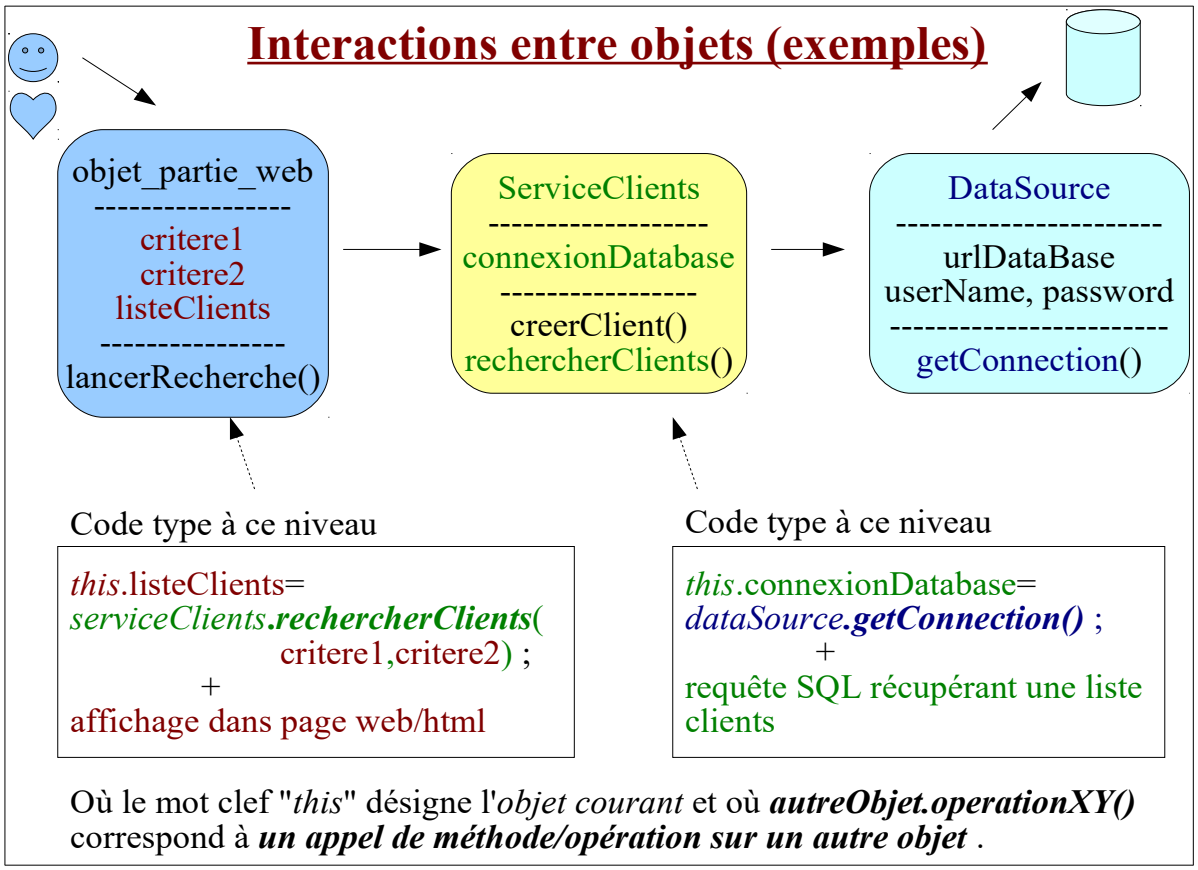
pers1.sePresenter() ; emp1.sePresenter(); emp1.getSalaire();

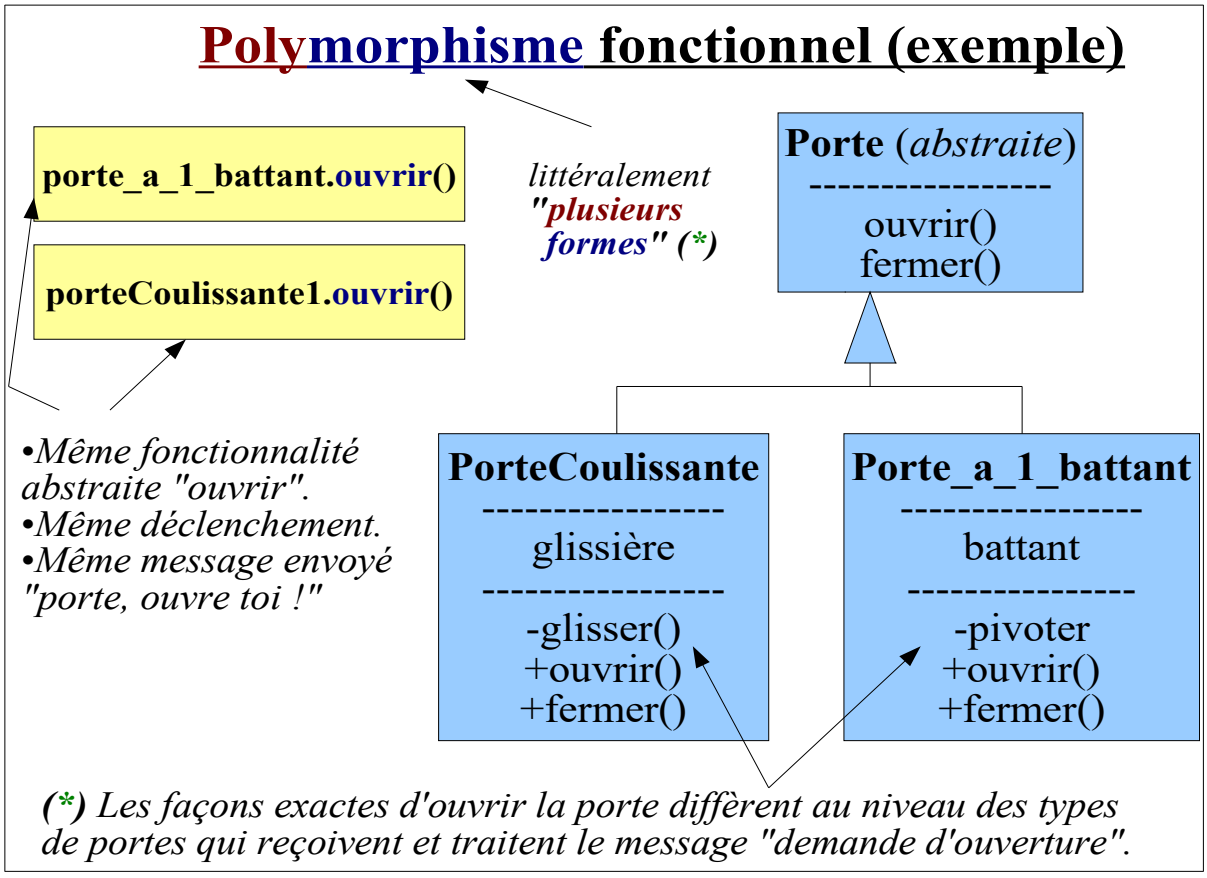
Arbre (ou graphe) d'héritage



Une relation d'héritage est représentée par une grosse flèche triangulaire allant de la sous-classe vers la sur-classe et signifiant

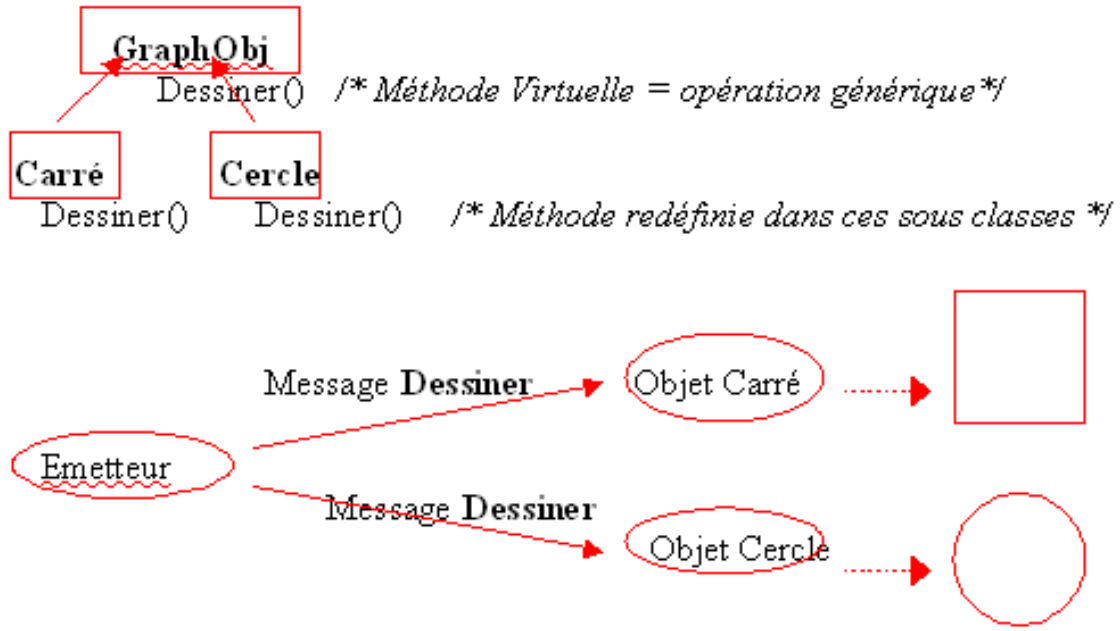
"is kind of / est une sorte de"


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


Polymorphisme fonctionnel (exemple)
porte_a_1_battant.ouvrir()
porteCoulissante1.ouvrir()
littéralement
"plusieurs formes" (*)
Porte (abstraite)
ouvrir()
fermer()
•Même fonctionnalité abstraite "ouvrir".
•Même déclenchement.
•Même message envoyé "porte, ouvre toi !"
PorteCoulissante
glissière
-glisser()
+ouvrir()
+fermer()
Porte_a_1_battant
battant
-pivoter
+ouvrir()
+fermer()
(*) Les façons exactes d'ouvrir la porte diffèrent au niveau des types de portes qui reçoivent et traitent le message "demande d'ouverture".

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.

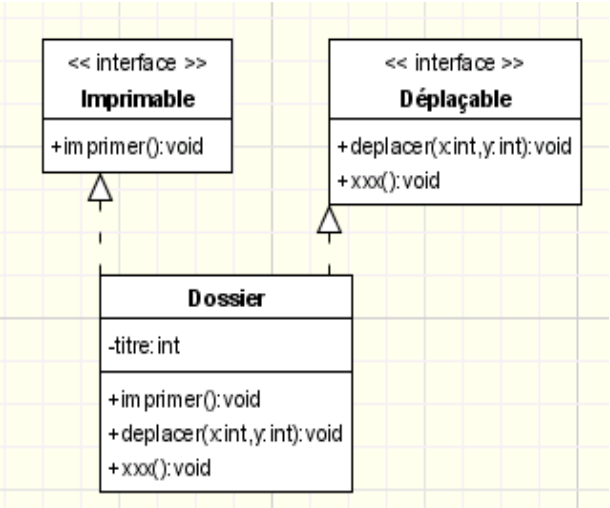
Le **polymorphisme** signifie qu'une même **opération** peut avoir des comportements différents suivant les classes.



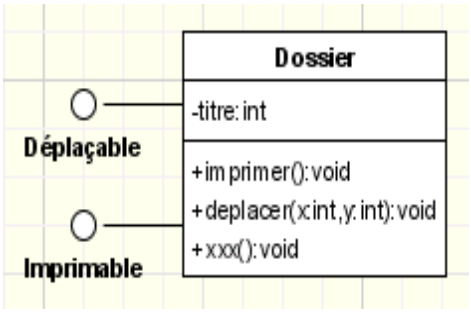
Interface

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.

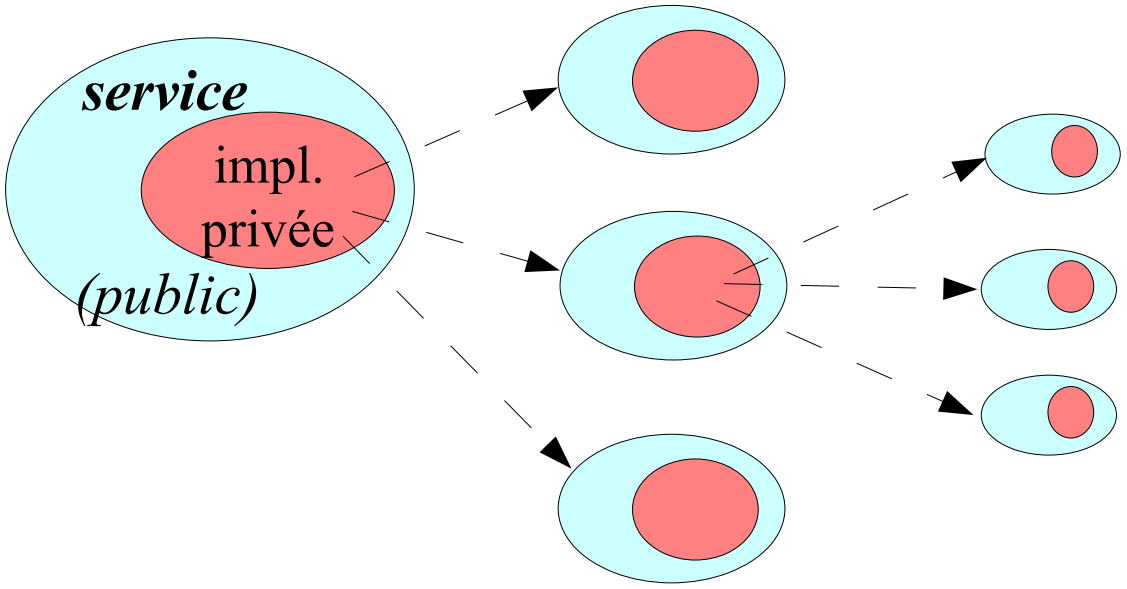


NB: *Une **interface** est vue comme un **type** de données **abstrait***



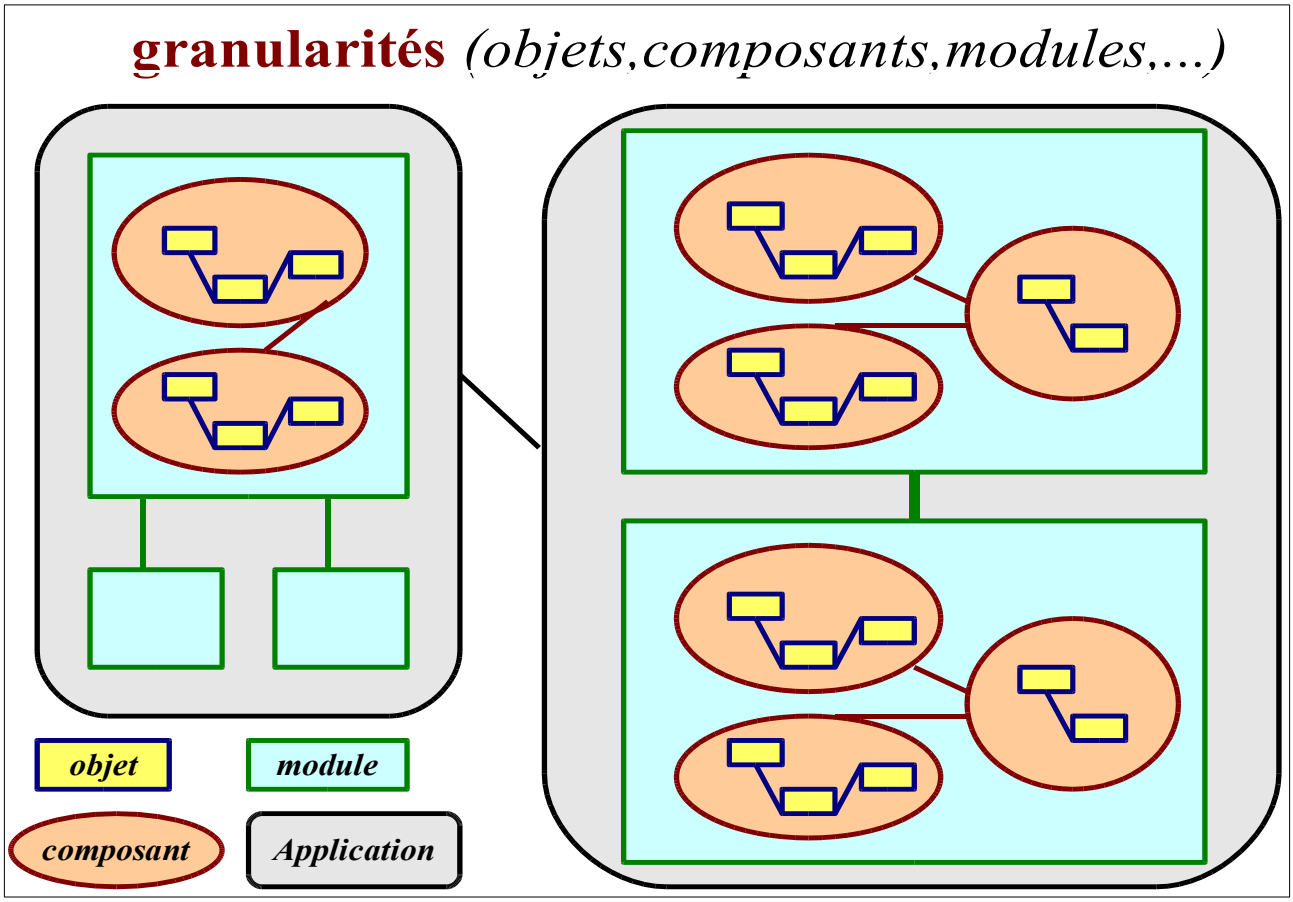
2. Granularité

Encapsulation & granularités



sous services , ...

Modules , composants , objets



3. Modularité

3.1. Forte cohésion et couplage faible

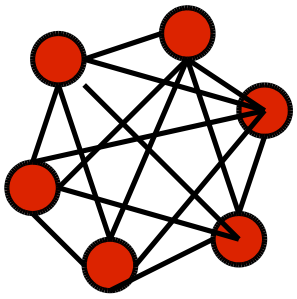
La **cohésion** mesure le degré de connectivité existant entre les éléments d'une classe unique. Une classe modélisant un aéroport aura une bien meilleure cohésion qu'une classe modélisant le roi d'une nation, le roi du jeu d'échec et le roi d'un jeu de carte . ***Il faut privilégier la cohésion fonctionnelle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

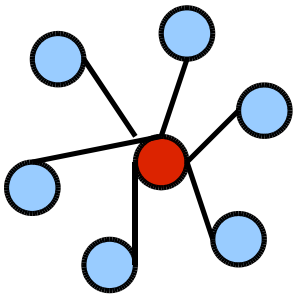
Couplage entre objets (idéalement faible)

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

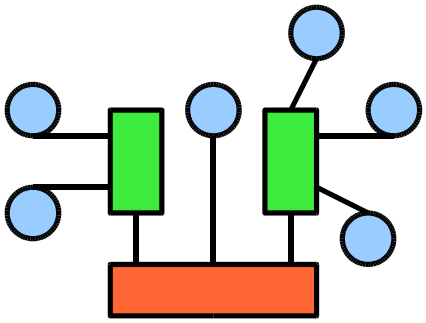
Un ***assemblage d'objets complémentaires*** rend globalement des ***services plus sophistiqués***. Cependant la complexité des liens entre les éléments peut éventuellement mener à un édifice précaire:



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

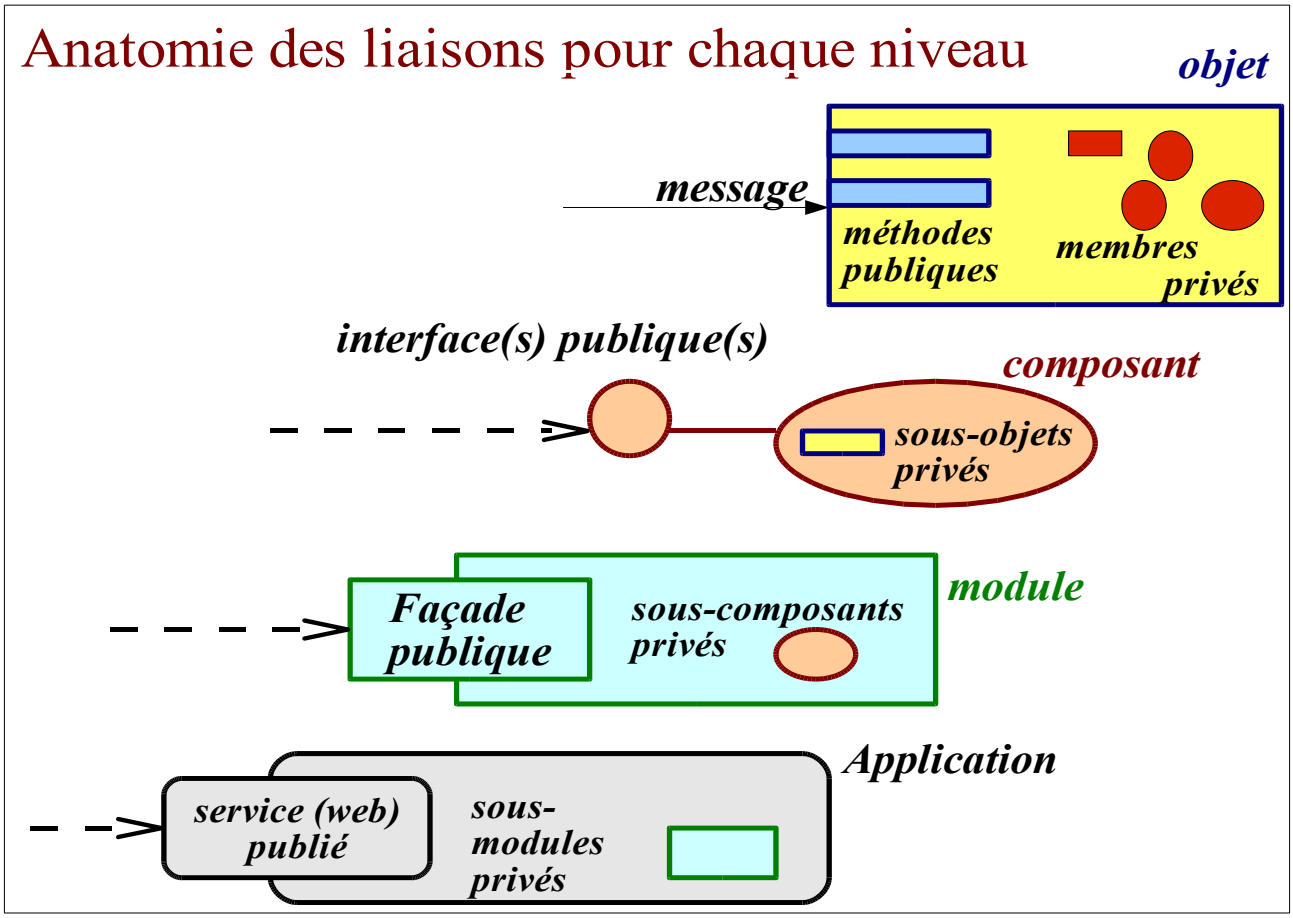


couplage faible
mais ***centralisé***
==> peu flexible
et point central
névralgique



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
objet
message
méthodes publiques
membres privés
interface(s) publique(s)
composant
sous-objets privés
Façade publique
sous-composants privés
module
service (web) publié
sous-modules privés
Application

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 d**éfinir et extraire l'ensemble des entités pertinentes** qui seront utiles au développement de l'application.

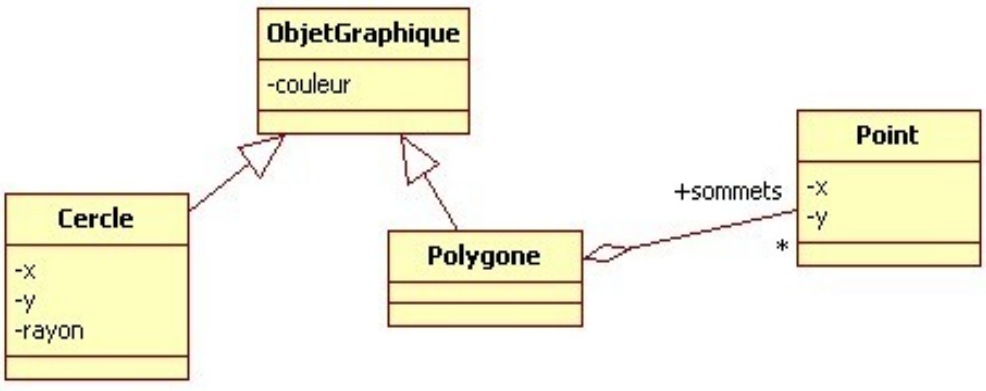
Partant d'un niveau très conceptuel (arbre sémantique, glossaire) elle permet d'aboutir à un **modèle logique de type "entités/relations"** résolument restreint à ce qui touche au domaine du système à développer (sans éléments inutiles) . Idéalement basées sur des formulations de type "chose concrète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

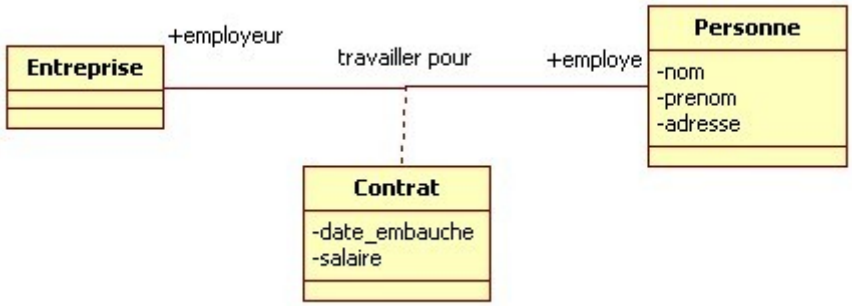
L'étape "**analyse du domaine**" est généralement effectuée de la façon suivante:

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

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

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





Quelques remarques:

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

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

Diagramme de classes *(modèle structurel)*

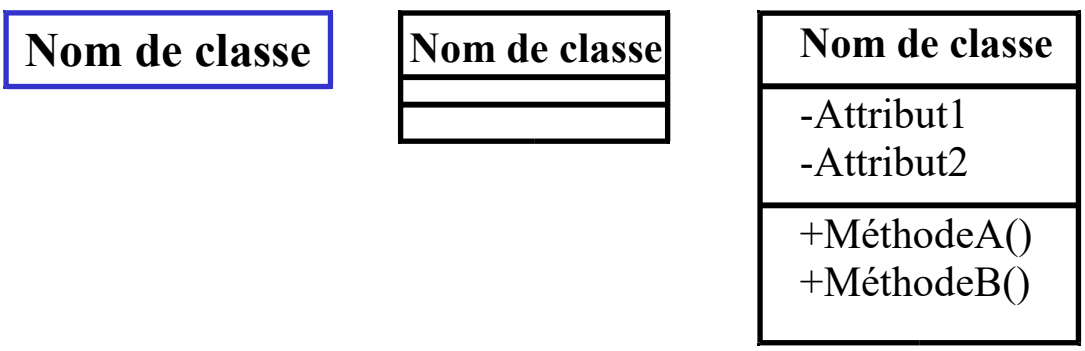
• Le **modèle statique/structurel** (basé sur les diagrammes de classes et d'instances) permet de décrire la **structure interne du système** (**entités existantes**, **relations entre les différentes parties**, …).

• 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*)



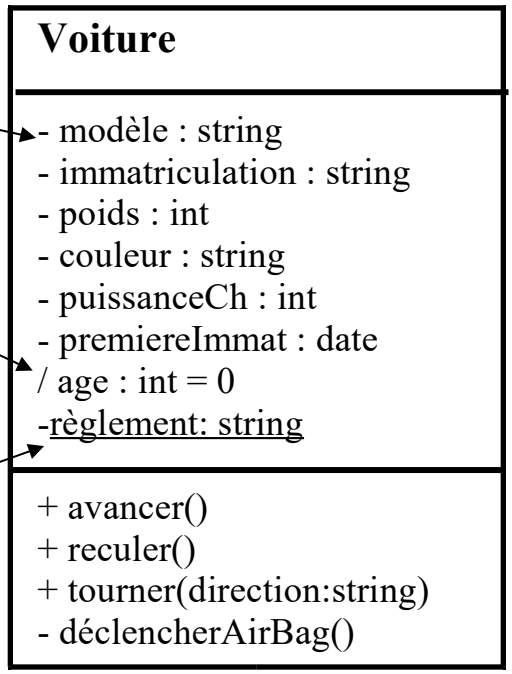
Attribut alias ***Propriété/Property*** , *Méthode* alias ***Opération*** .

Détails sur les éléments d'une classe

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

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

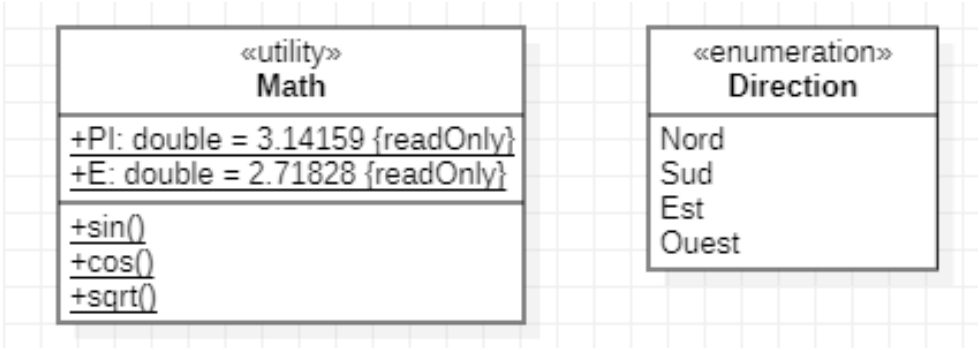
Un attribut souligné (ou préfixé par un **$**) est une **variable de classe** (***static*** en java et c++)



Classes spéciales (utilitaires, énumération, …)

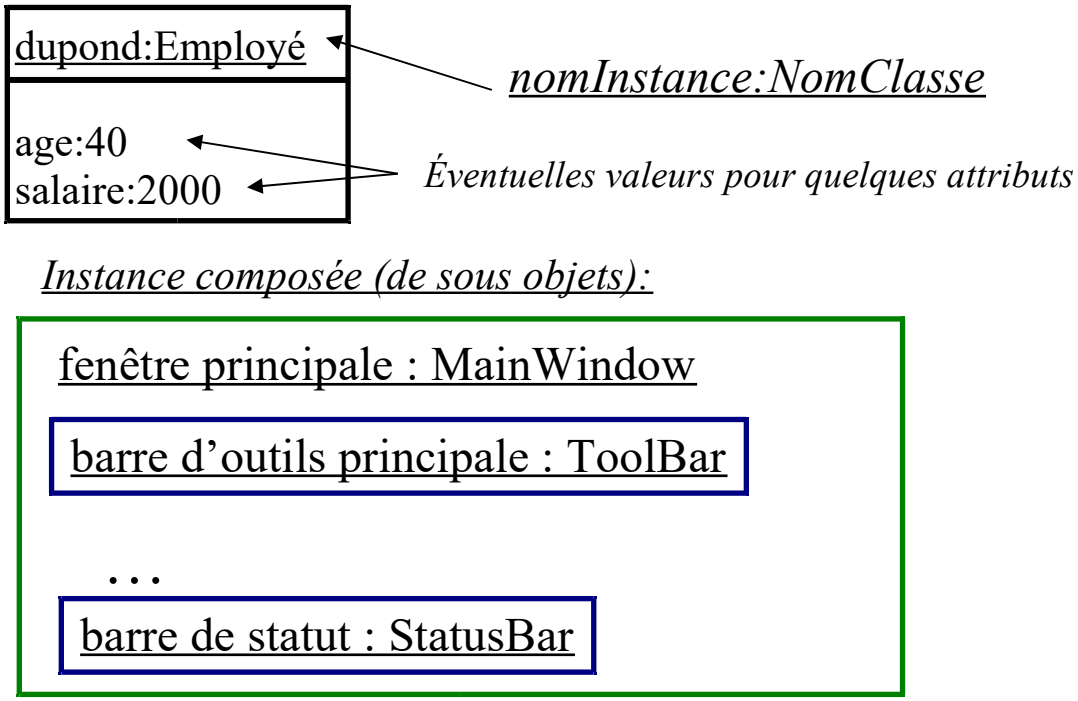
• Dans un monde matériel et rationnel ou tout est objet, il n'y a plus de place pour des fonctions globales (anarchiques).
Celles-ci doivent être rangées dans des classes utilitaires.

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



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

Eventuels (et rares) diagrammes d'instances



Associations (relations)

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

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

Extrémités d'association

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

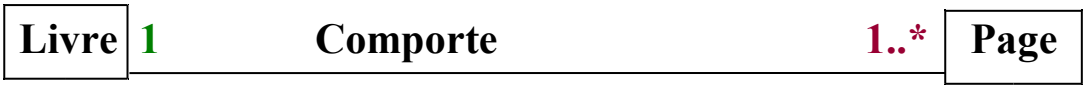
• On peut préciser des caractéristiques d'une association qui sont liées à une de ses **extrémités (association end):**
- ***rôle*** joué par un objet dans l'association
- ***multiplicité***
- ***navigabilité***
- …

Multiplicité UML (~~cardinalité~~)

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

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

Interprétation des multiplicités:

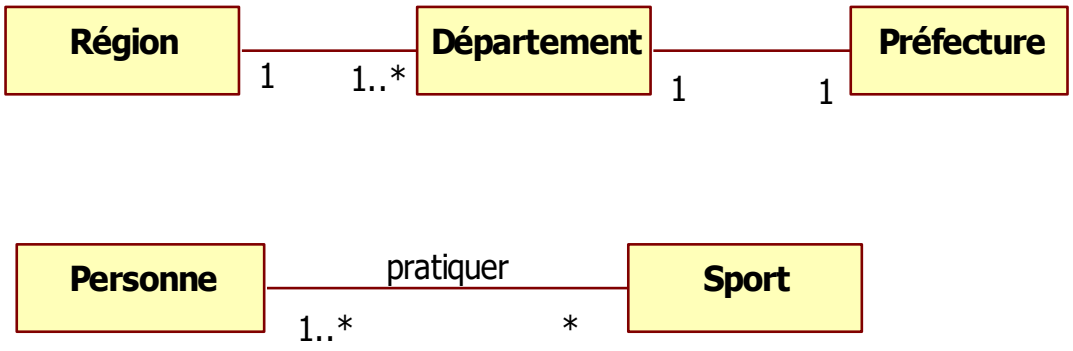


1 livre comporte ***au moins une*** *page*

et

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

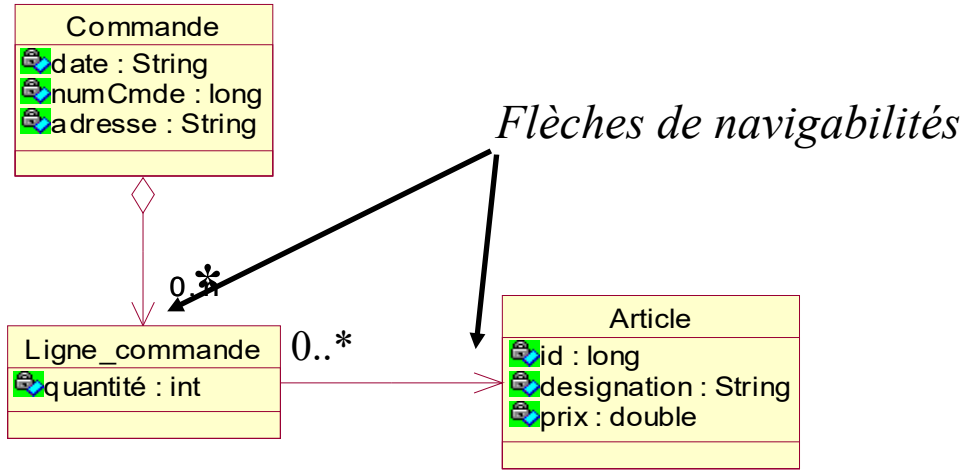
Multiplicités (exemples)



NB:
• Les multiplicités d'UML utilisent des notations inversées vis à des cardinalités de Merise.
• ***Les multiplicités dépendent souvent du contexte*** (système à modéliser).

Navigabilité

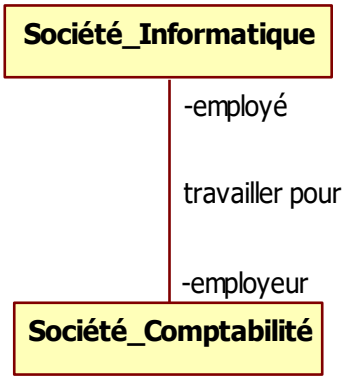
Une **flèche de navigabilité** permet de restreindre un accès par défaut bidirectionnel en un **accès unidirectionnel** ***plus simple à mettre en œuvre et engendrant moins de dépendance***.



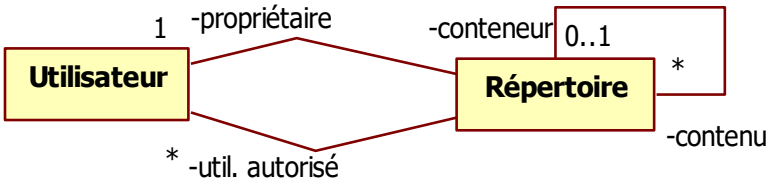
*Un article **n'a pas directement accès** à une ligne de commande*

Rôles

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



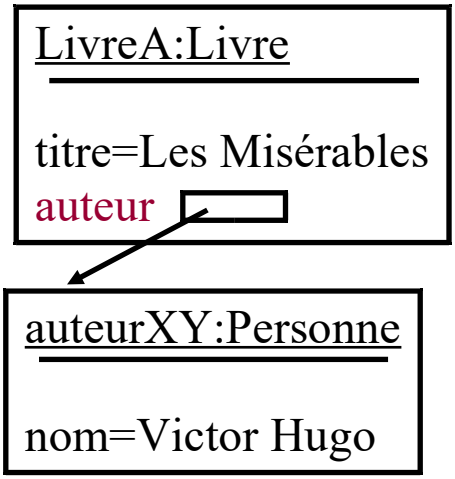
Ils peuvent servir à lever certaines ambiguïtés:



Rôles & implémentation

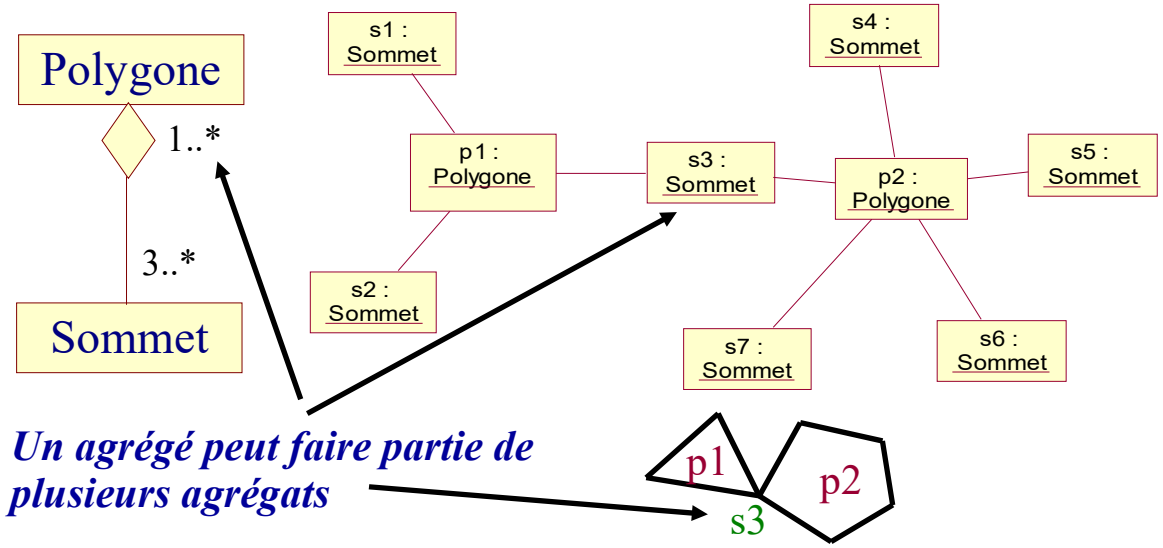

```
class Livre
{
private String titre;
private Personne auteur;
…
}
```

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



Agrégation

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



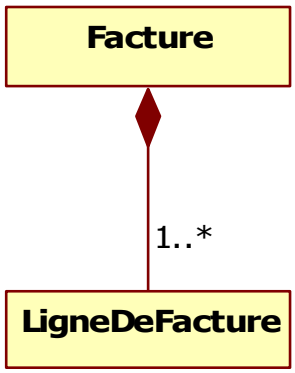
Agrégation (caractéristiques)

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

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

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

Composition (agrégation forte)



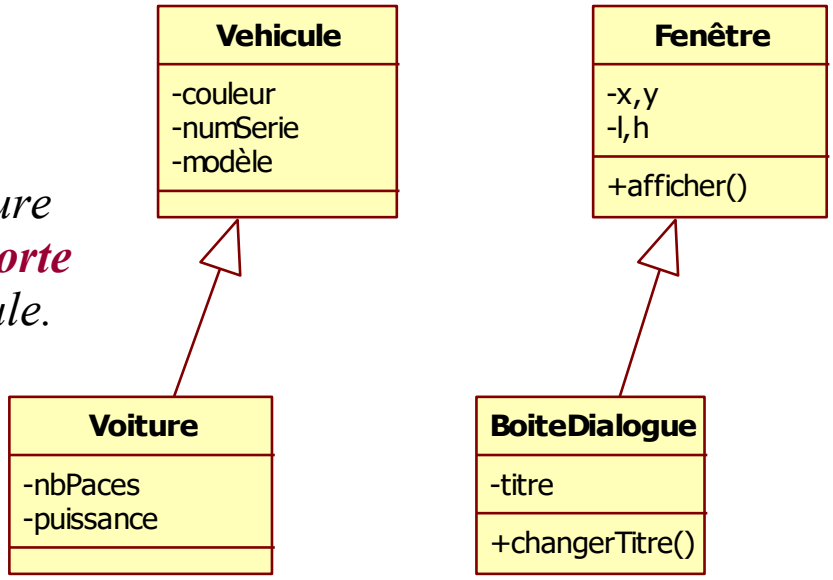
En général, le sous-objet n'existe que si le conteneur (l'agrégat) existe: **lorsque l'agrégat est détruit, les sous objets doivent également disparaître**.
("cascade-delete" en base de données).

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

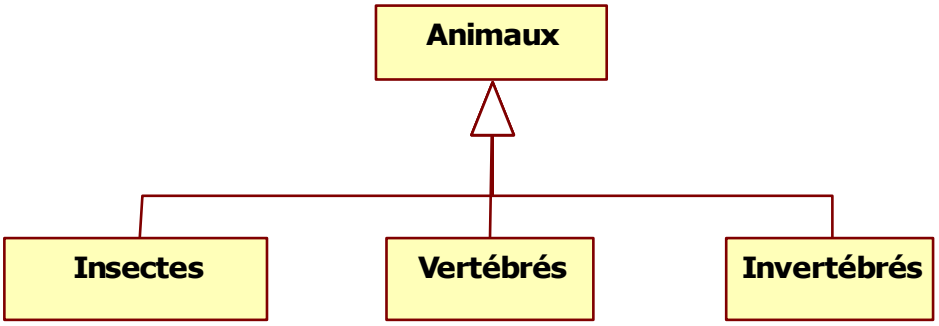
Remarque: Le **losange est quelquefois rempli de noir** pour montrer **que le sous objet est physiquement compris dans le conteneur** (l'agrégat). On parle alors d'agrégation forte (véritable **composition**).

Généralisation (héritage)

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



Classification (généralisation)

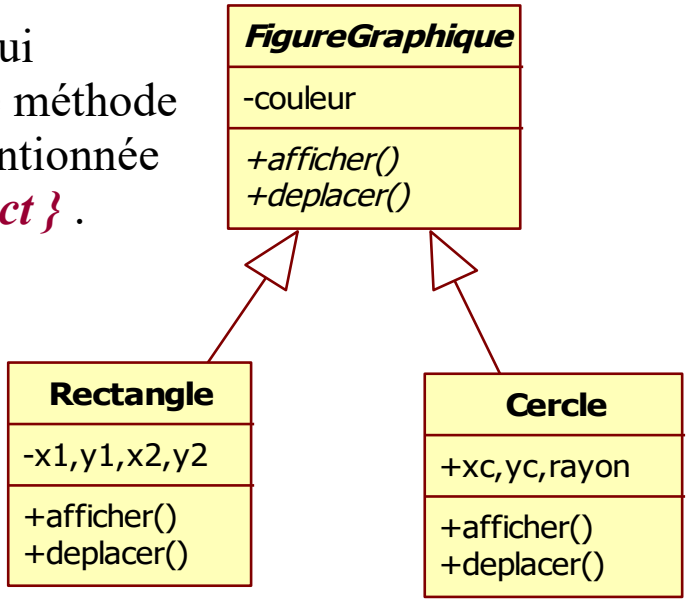


Les animaux peuvent être classées dans divers **groupes** (et sous groupes).

Classification selon caractéristiques discriminantes.

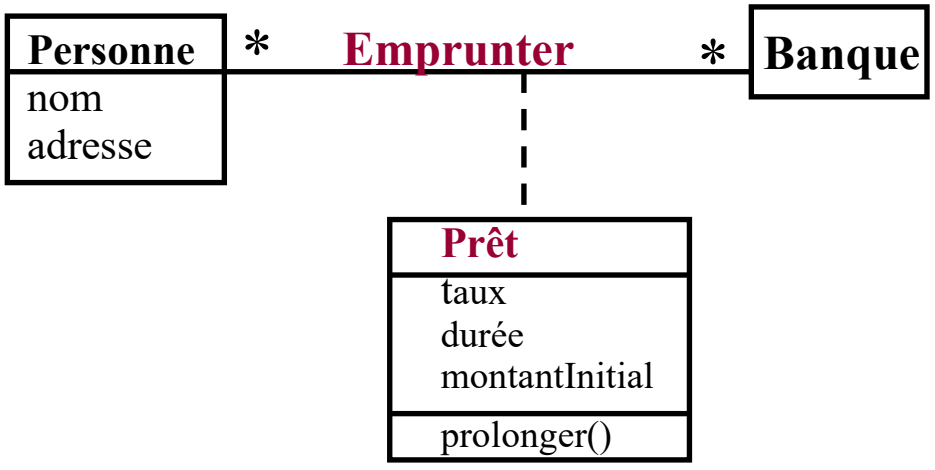
Classes abstraites et concrètes

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



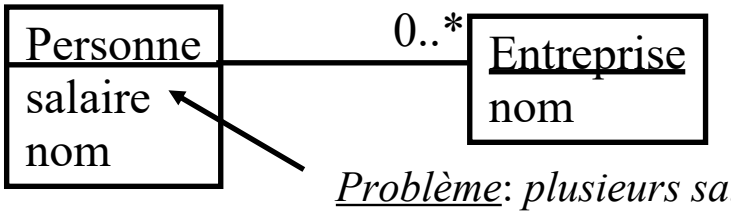
Classes d'association

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

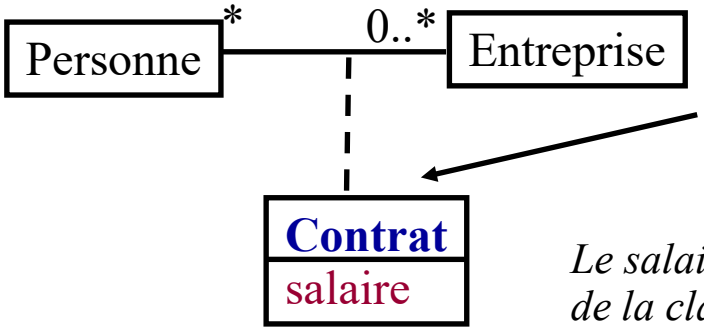


Remarque : Les classes d'association se transposent très bien dans le modèle relationnel comme des tables d'associations (avec au moins deux clefs étrangères)

Classes d'association (exemples)



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

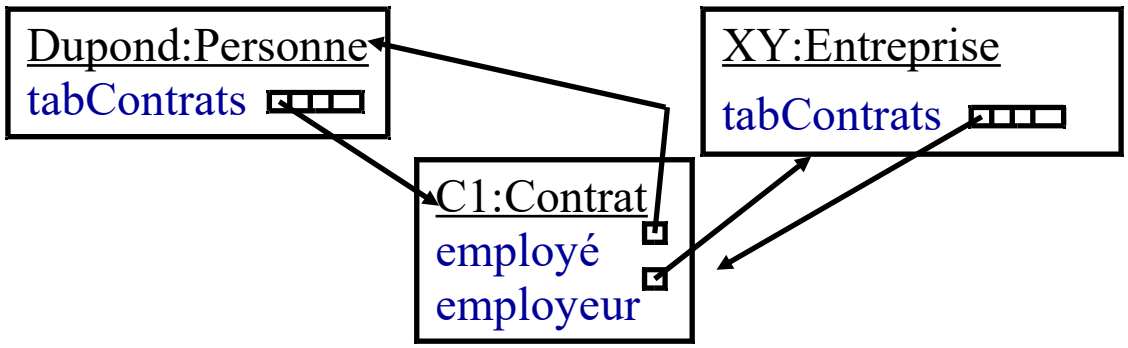


Un objet contrat n'existera que si une association est maintenue entre une personne et une entreprise.

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

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

Implémentation des classes d'associations



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

3. Eléments structurants d'UML

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

3.1. Modèle

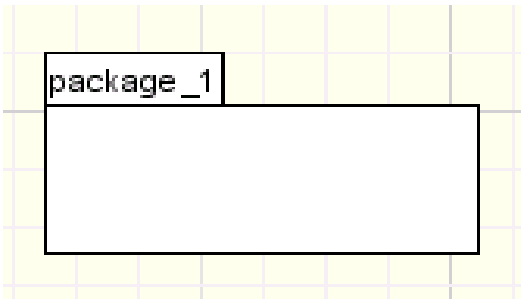
Correspondant généralement à une application entière (ou bien à un sous système) , un **modèle UML** est essentiellement constitué par une **arborescence d'éléments** .

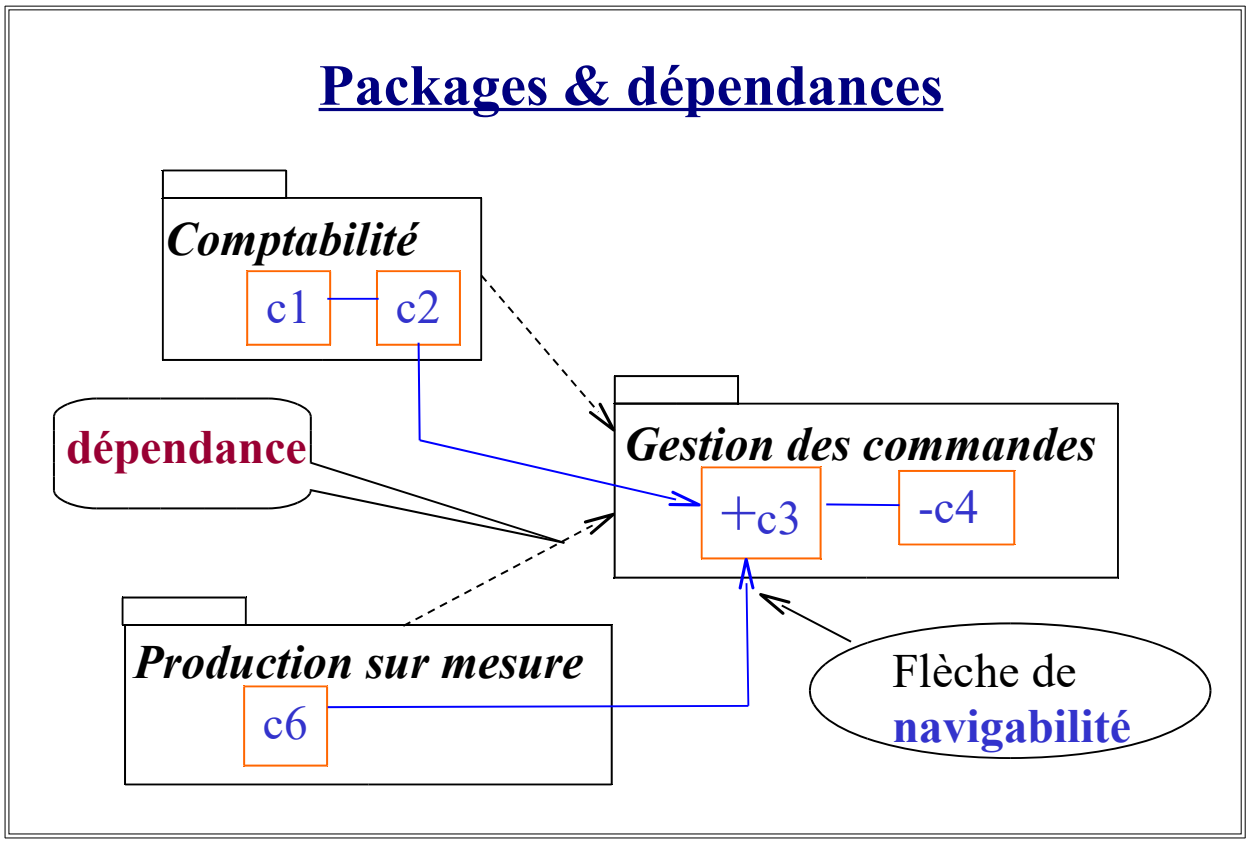
C'est à partir de son contenu (*xy.uml* ou *xy.xmi*) que l'on pourra éventuellement générer du code via MDA.

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

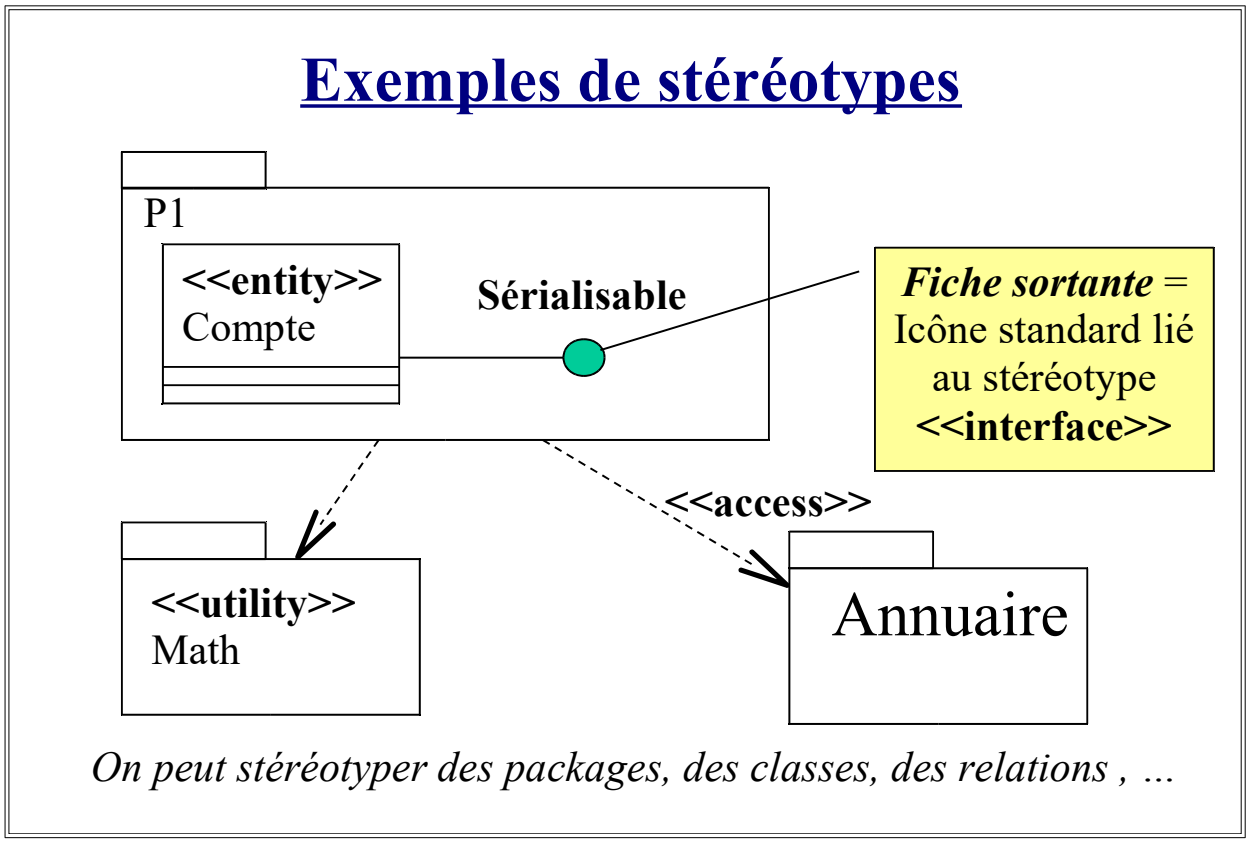
3.4. Stéréotypes

- 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, ...).

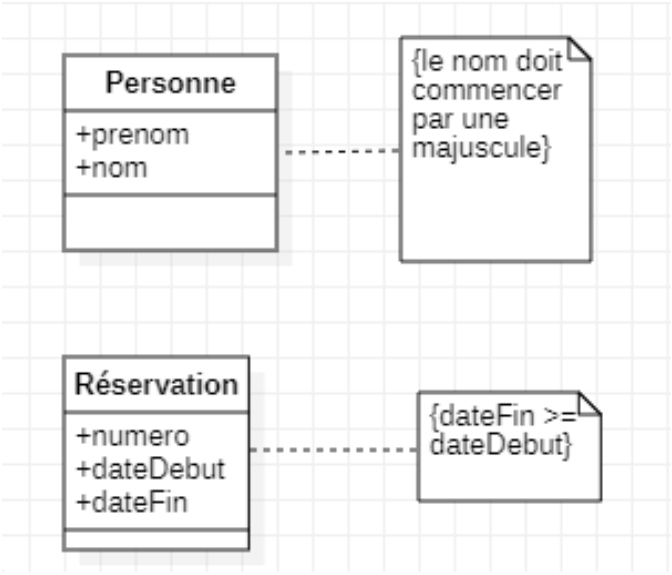
Exemples de valeurs étiquetées: auteur=didier , withDTO=true, version=V1

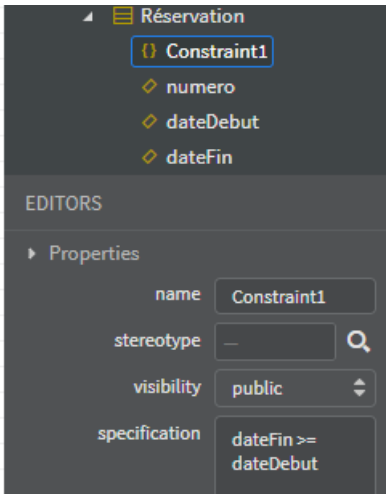
3.6. Contraintes

- 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 }**

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





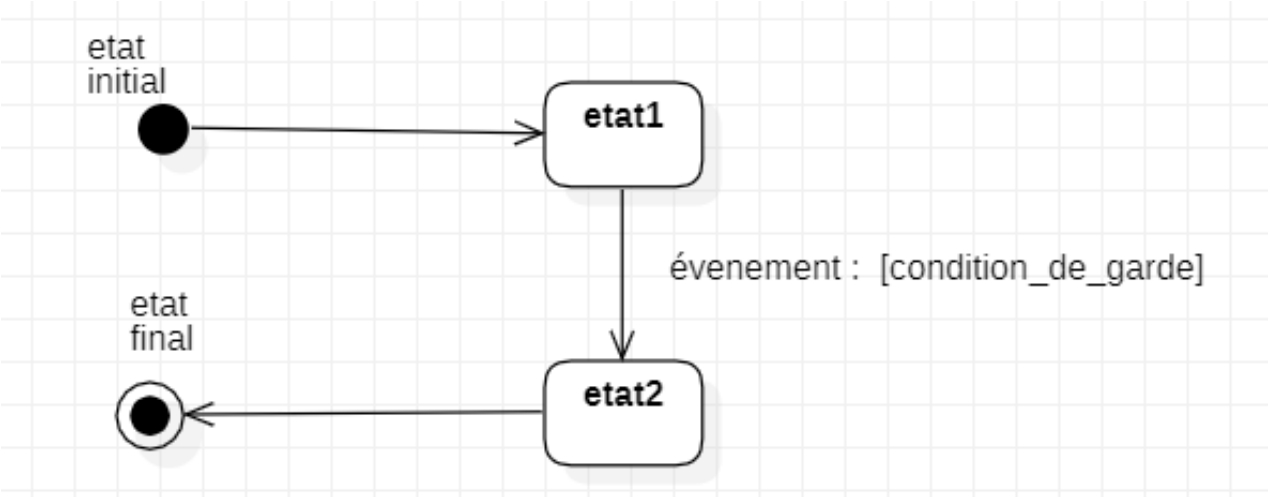
NB:

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

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

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

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

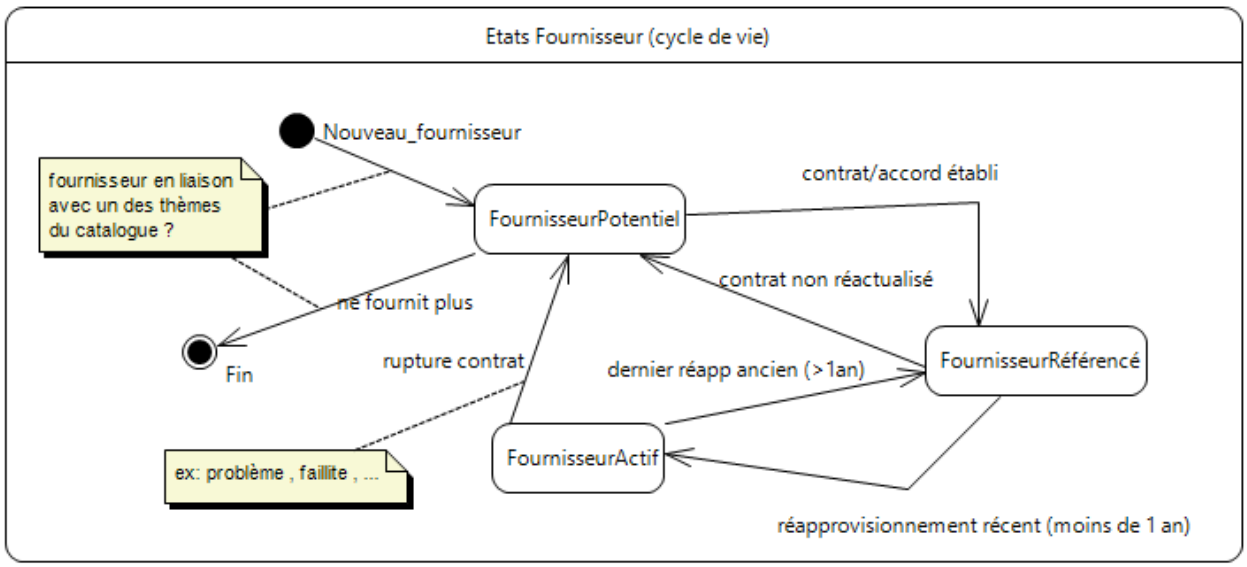
Quelques noms/types (classiques) d'événements :

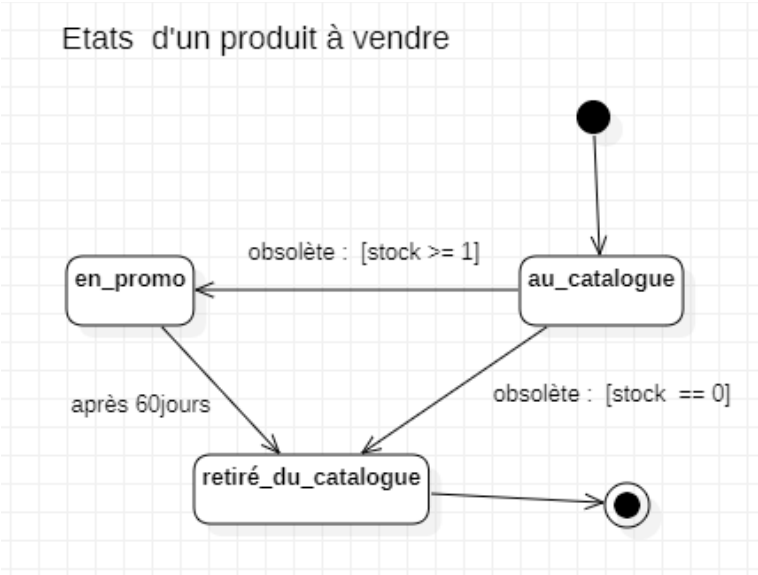
Types d'événements	***exemples***	***sémantiques***
Change Event	**when**(exp_booleenne)	L'expression booléenne devient vraie (après changement)
Signal Event	*feu vert , ...*	Signal reçu (sans réponse à renvoyer)
Call Event	*demande_xy_reçue , ...*	Appel reçu
Time Event	**after**(temporisation)	Période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

Exemple(s):

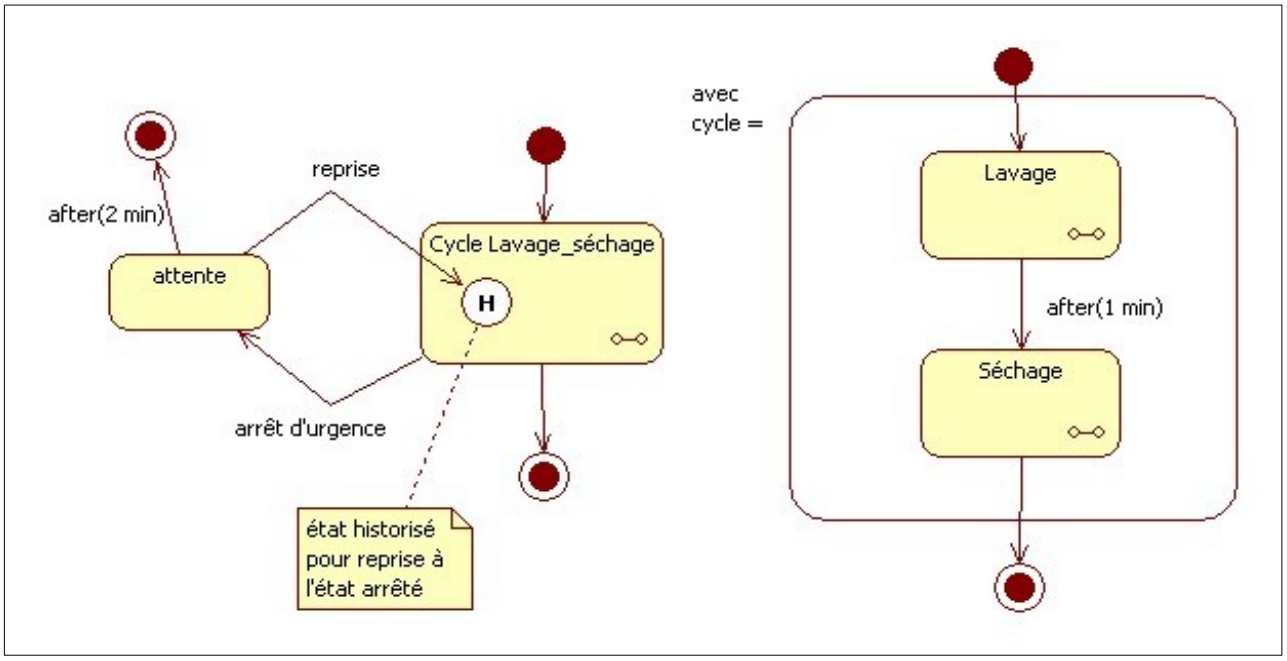
Lumière (avec minuterie) : *allumée* ------ ***after(30s)***----> *éteinte*

1.1. Exemples simples

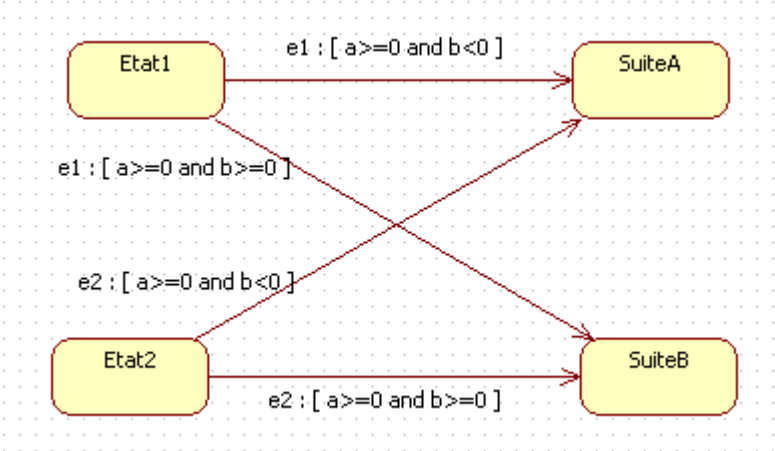


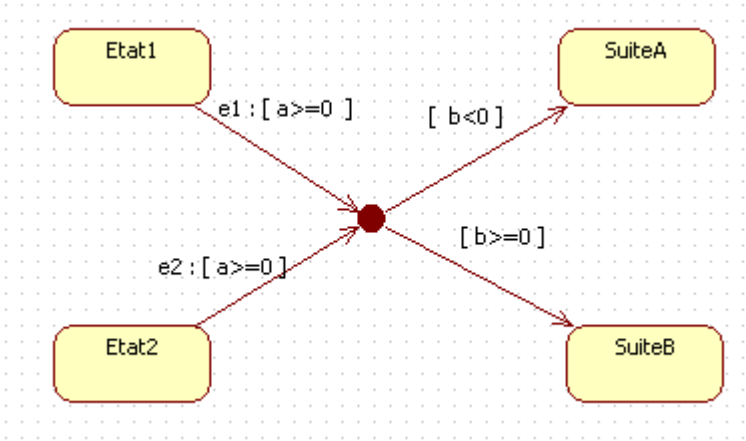


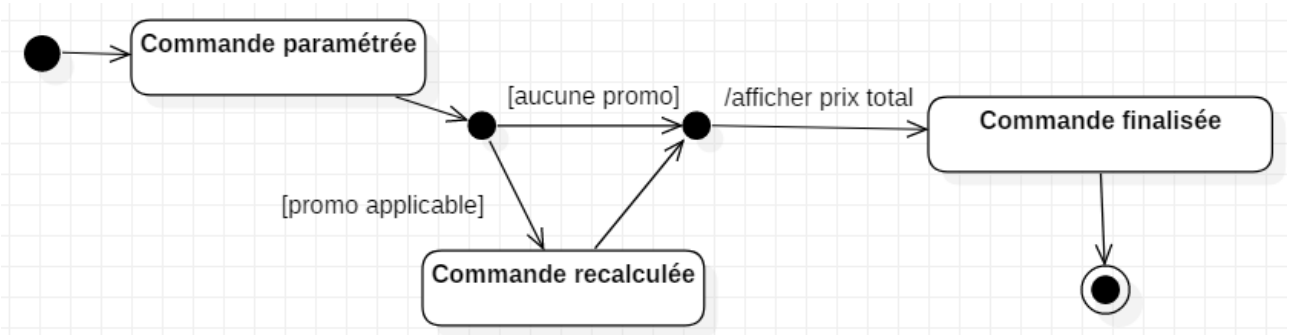
1.2. Etats historisés (rares , pour informatique industrielle)



1.3. Point de jonction (depuis UML2)

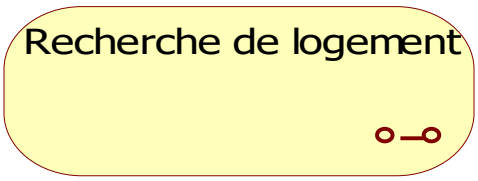






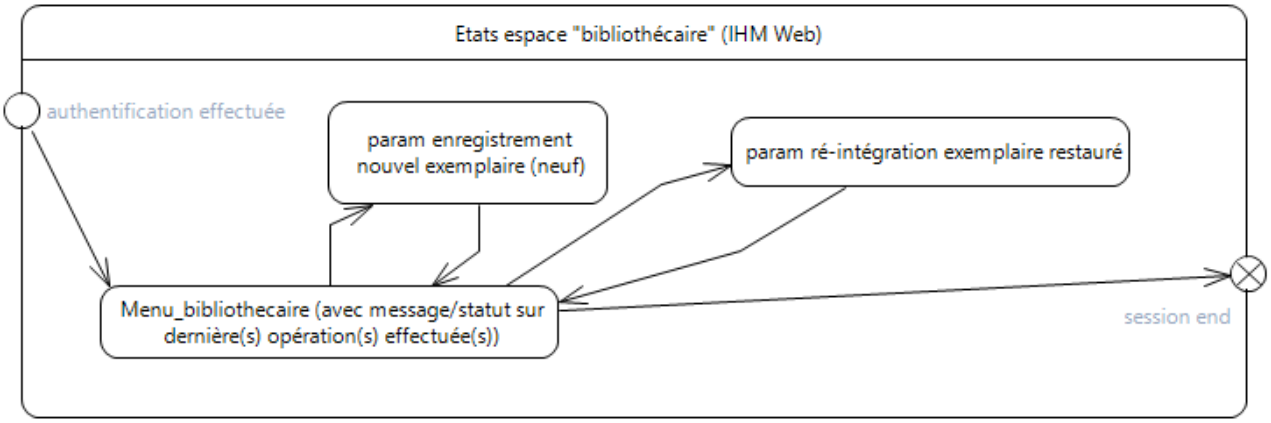
1.4. Etat composite (super état)

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

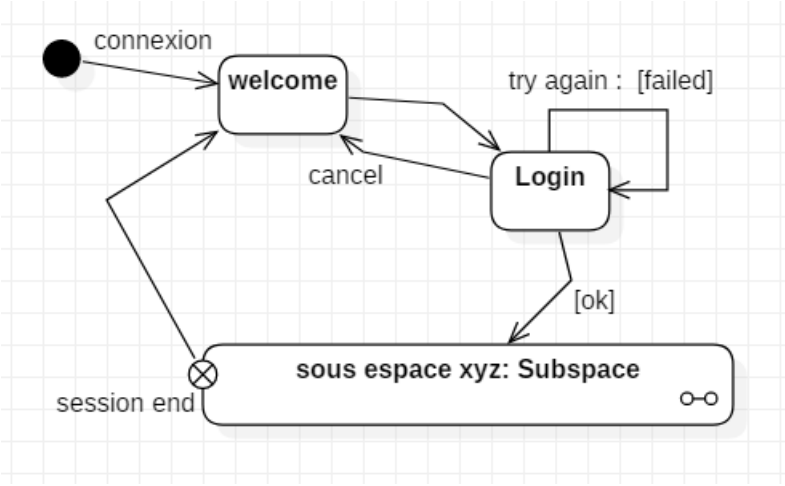


NB : *Quelquefois considéré comme état ordinaire*

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



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



1.5. Liens avec diagramme de classe.

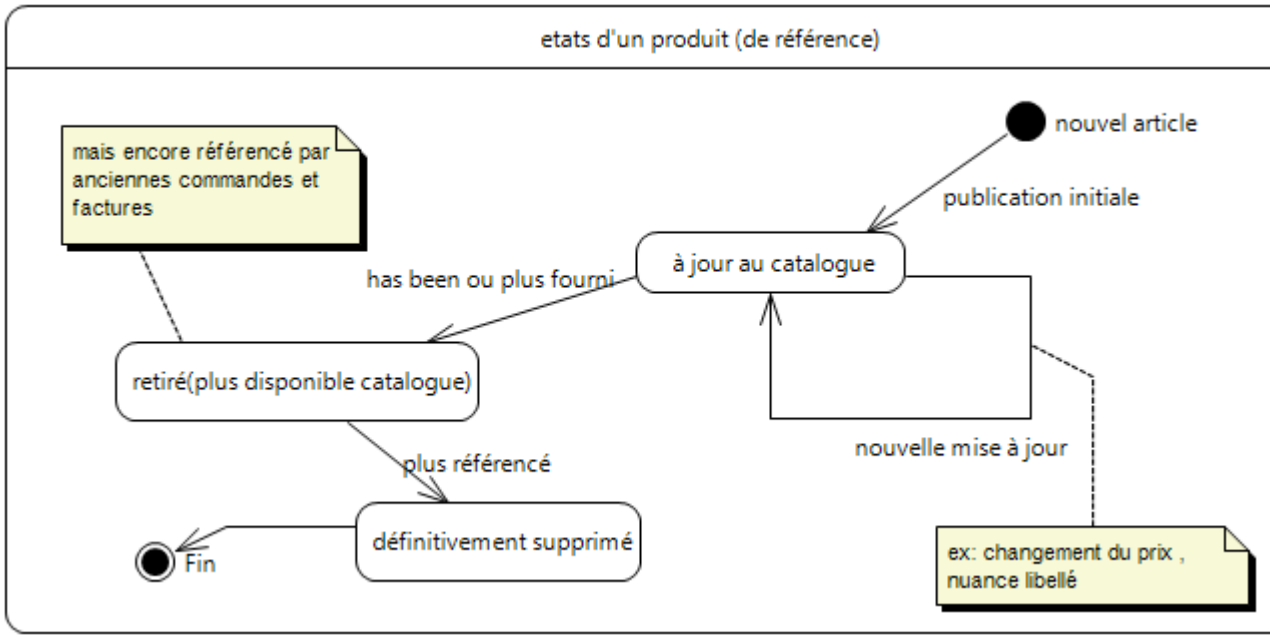
Un diagramme d'états (en tant qu'états de quelquechose) est souvent rattaché à une classe .

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

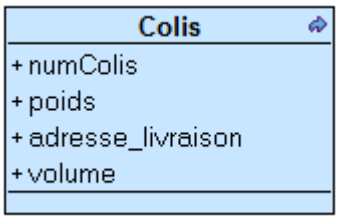
- "booléen"
- "énumération"
- "lien_vers_xy" existant ou pas ,
- "dateActionXy" nulle ou pas ,
- ...

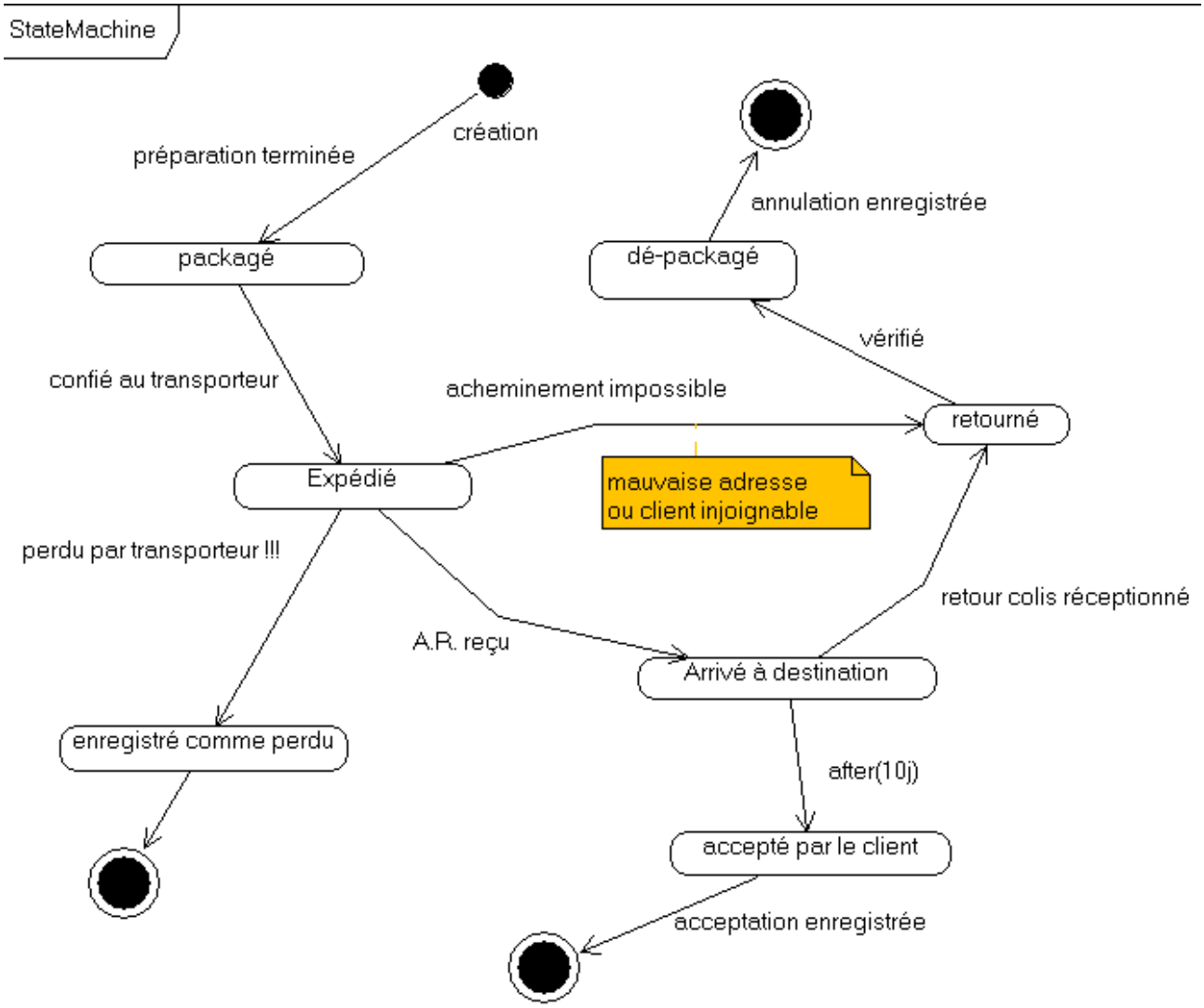
2. Utilisation d'un diagramme d'état pour illustrer un cycle de vie

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



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





XIV - Réalisation de Uses Cases / diag. Séquences

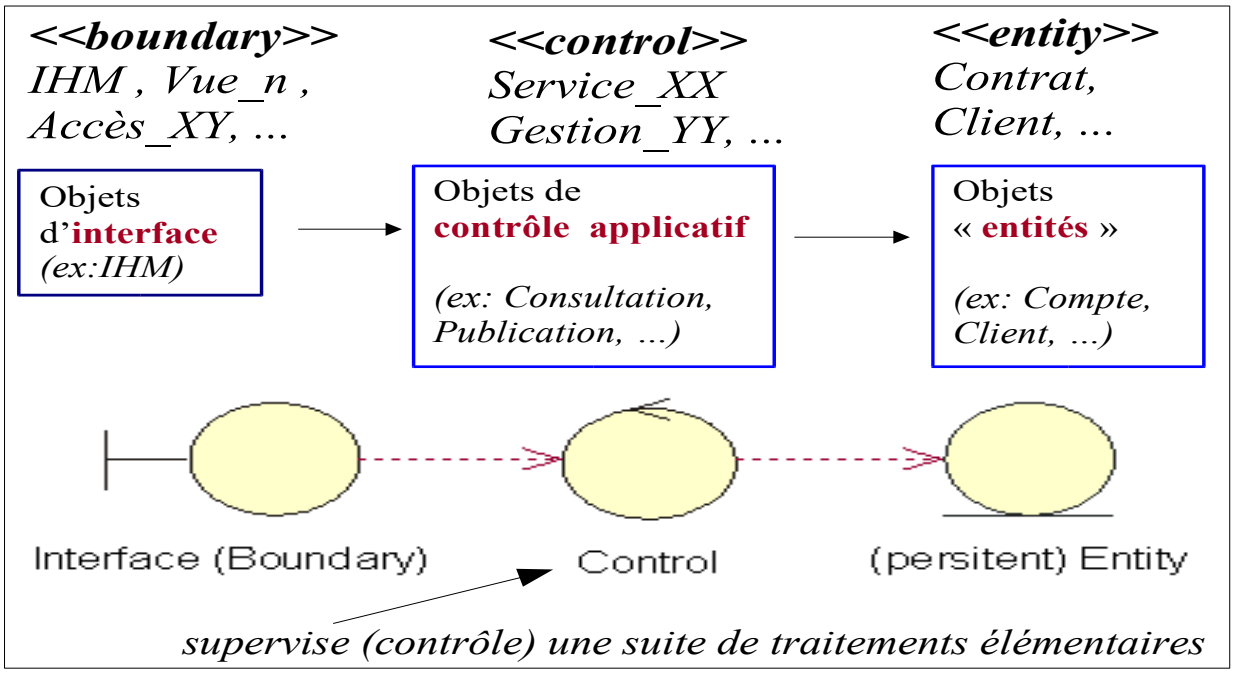
1. Analyse applicative (objectif et mise en oeuvre)

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>>

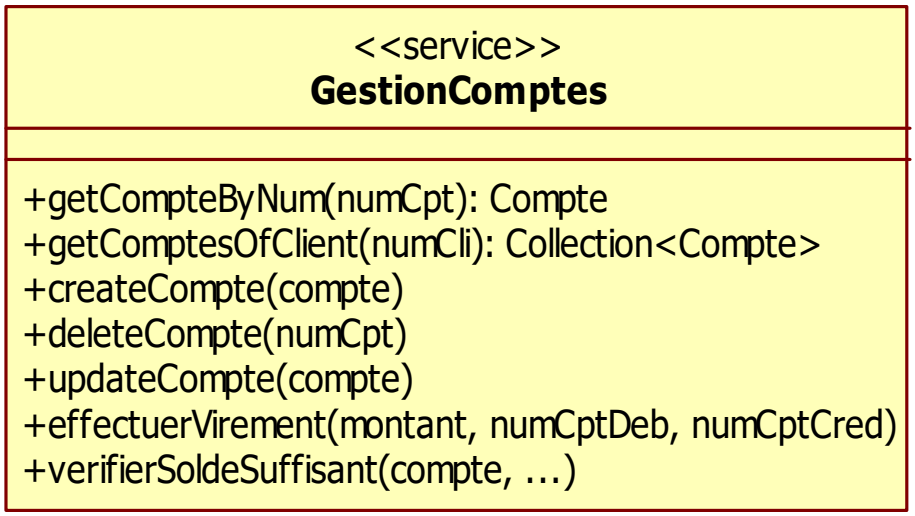
2. Responsabilités (n-tiers) et services métiers

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

Principales méthodes d'un service métier:

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

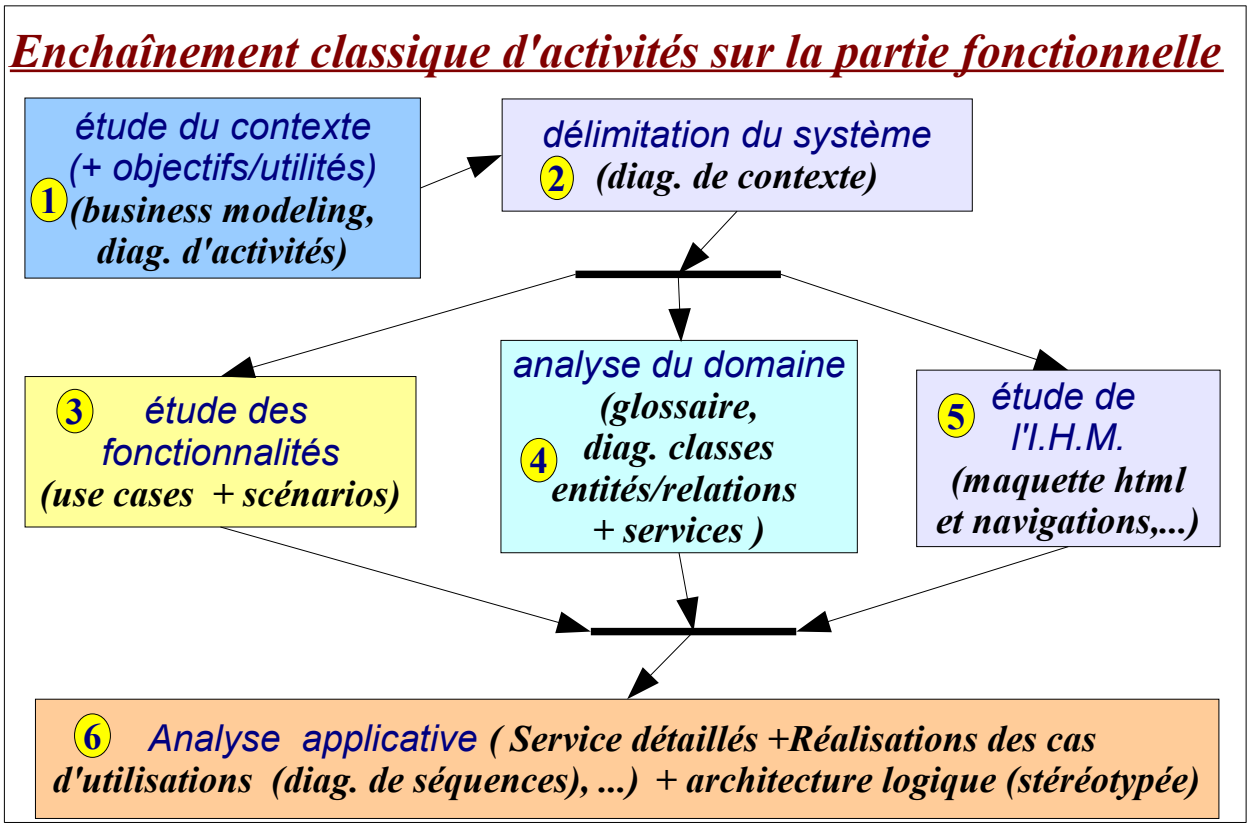
Exemple (service métier "GestionComptes"):



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

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

3. Repère méthodologique (rappel)



L'analyse applicative à 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.

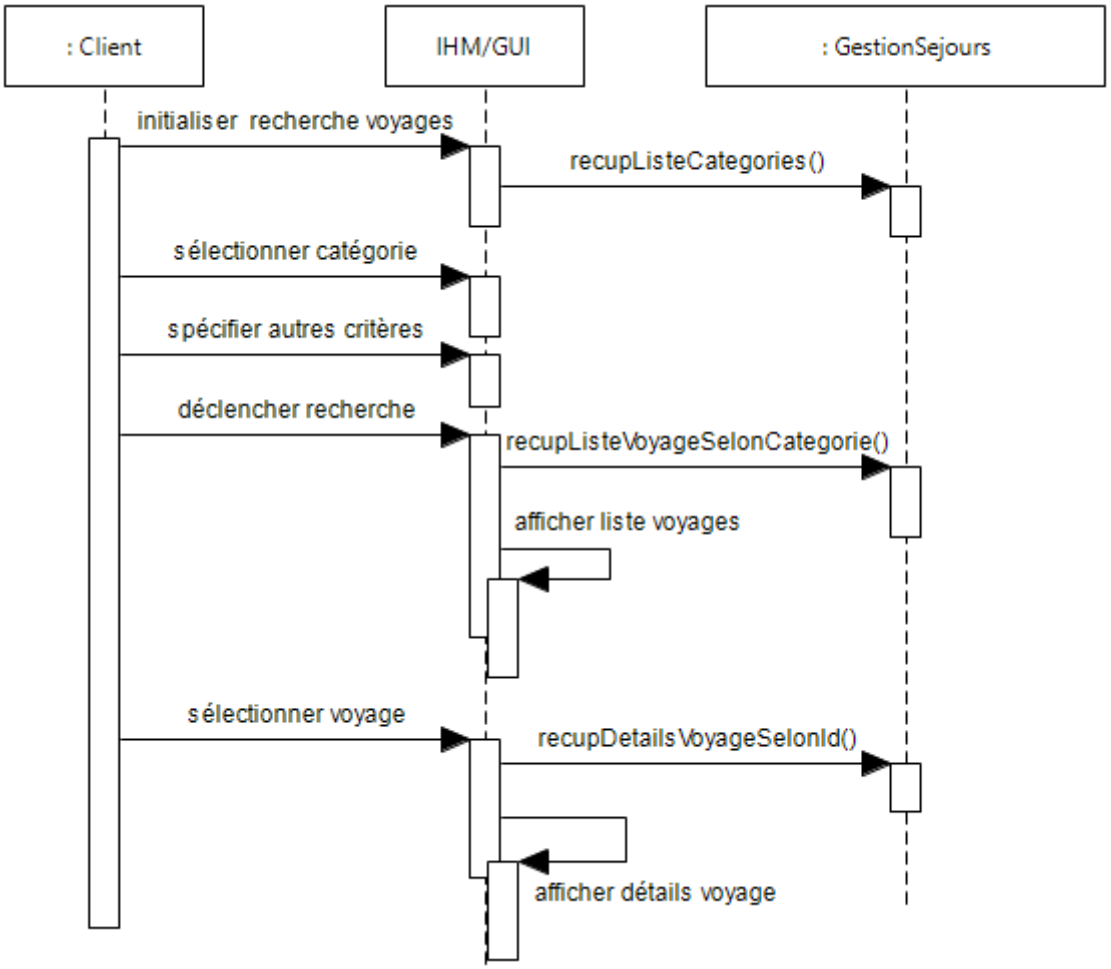
Le principal fil conducteur réside dans la "réalisation des cas d'utilisations" autrement dit dans la retranscription des scénarios des cas d'utilisations en digrammes de séquences.

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

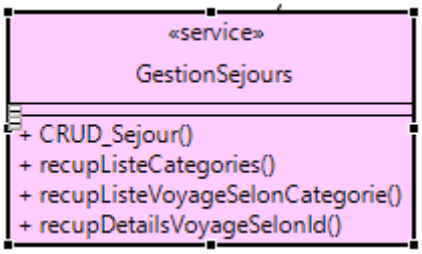
4. Réalisation des cas d'utilisations

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

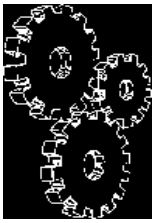


Conseil : ne pas faire apparaître les classes de type <<entity>> dans les diagrammes de séquence de niveau analyse car la sous séquence exacte/réalisable dépend des choix technologiques (conception) .

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~~ *communication*

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

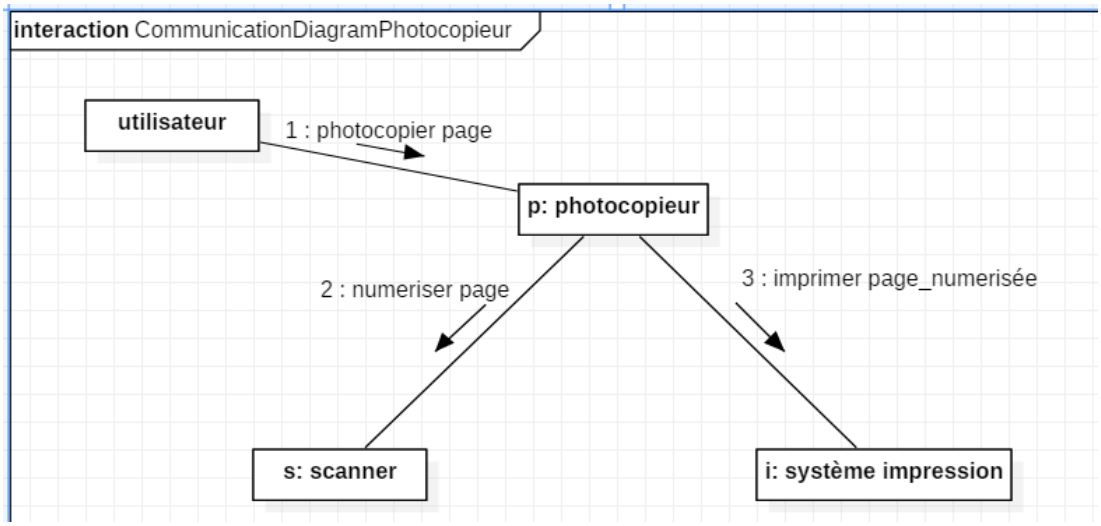
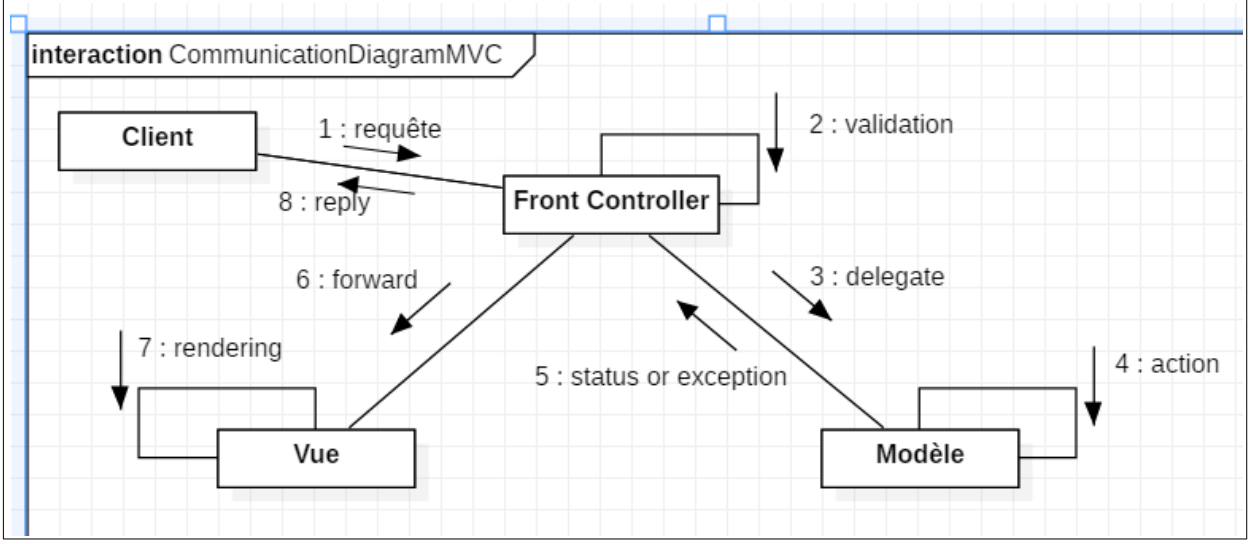
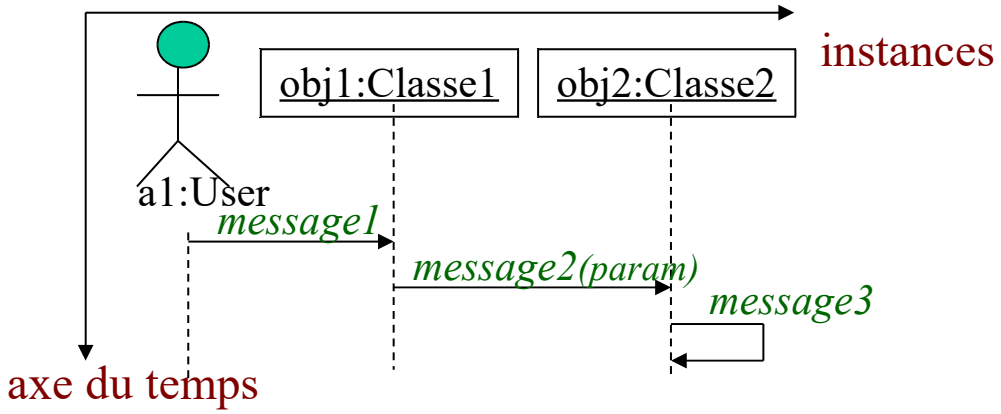


Diagramme de communication UML = ***pratique pour illustrer certains principes de fonctionnement***
(ici : le design pattern *MVC* : *Model-Vue-Controller*)



7. Diagramme de séquences (UML)

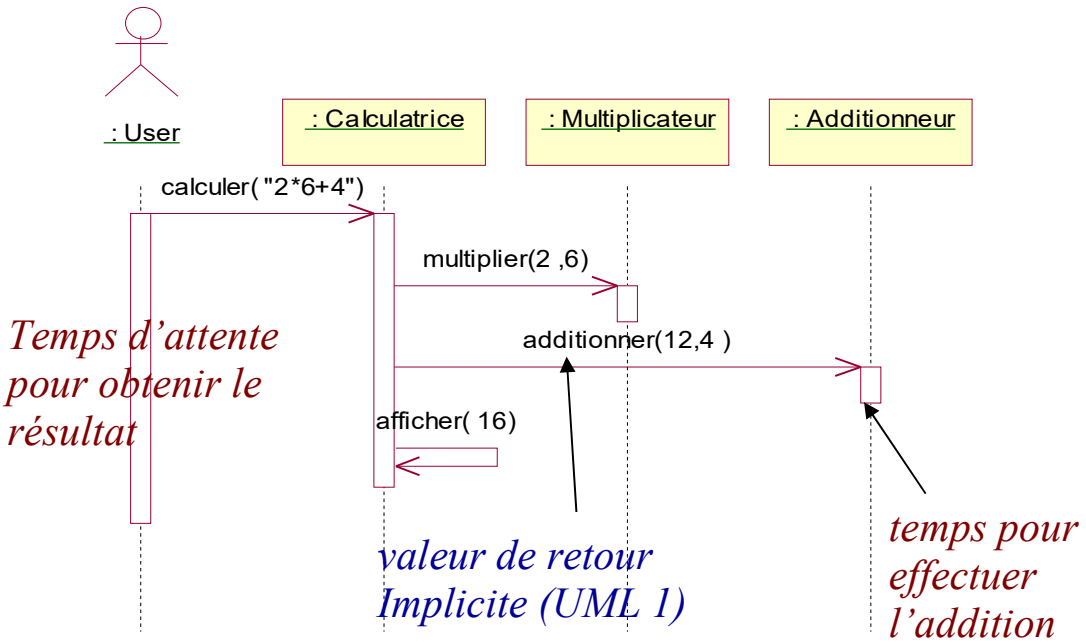
Diagramme de séquences



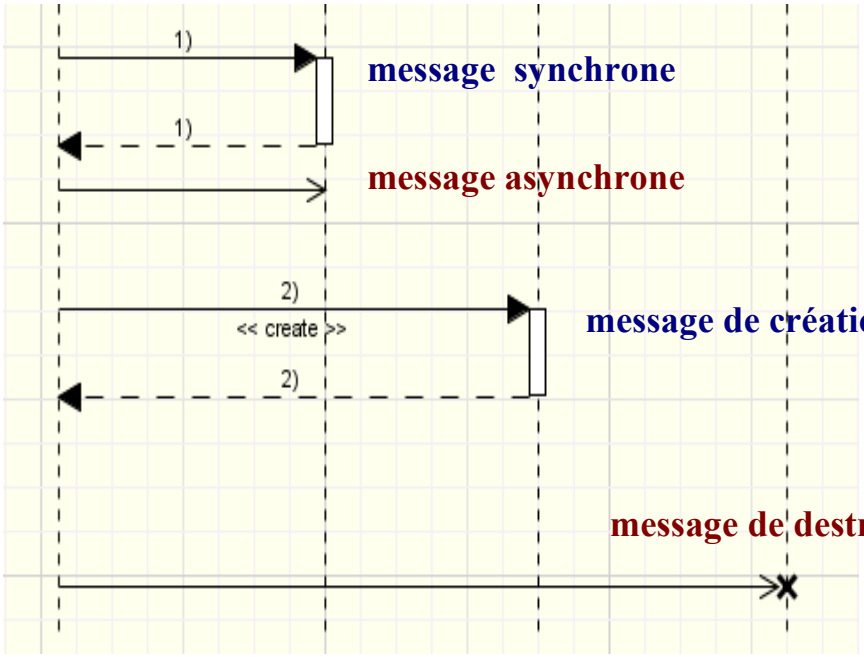
• Un **diagramme de séquence** montre un ensemble de messages échangés entres des objets pour remplir une fonctionnalité donnée ou pour obtenir un résultat.

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

Contrôle du focus (UML 1)

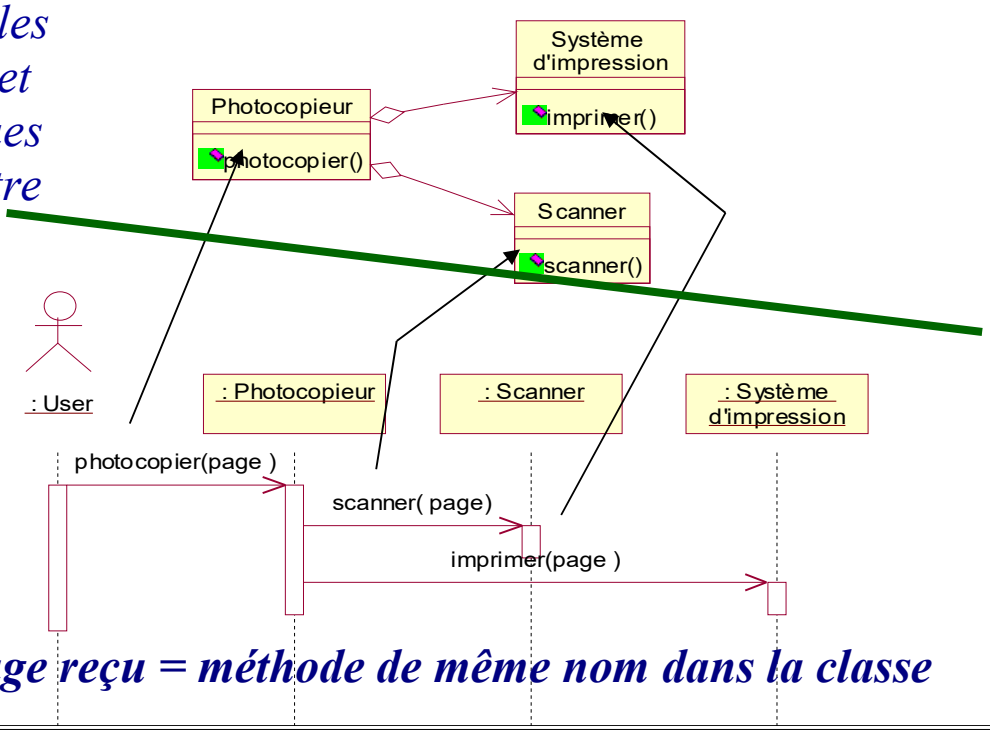


Différents types de messages (UML2)



Cohérence entre les digrammes

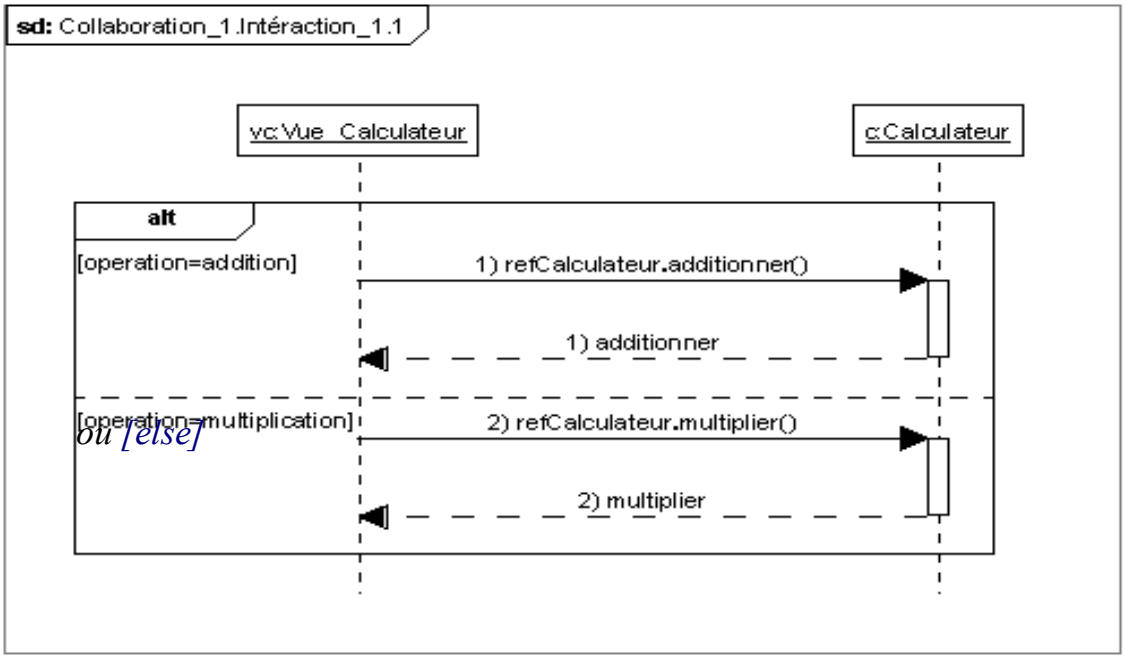
Les modèles statiques et dynamiques doivent être cohérents



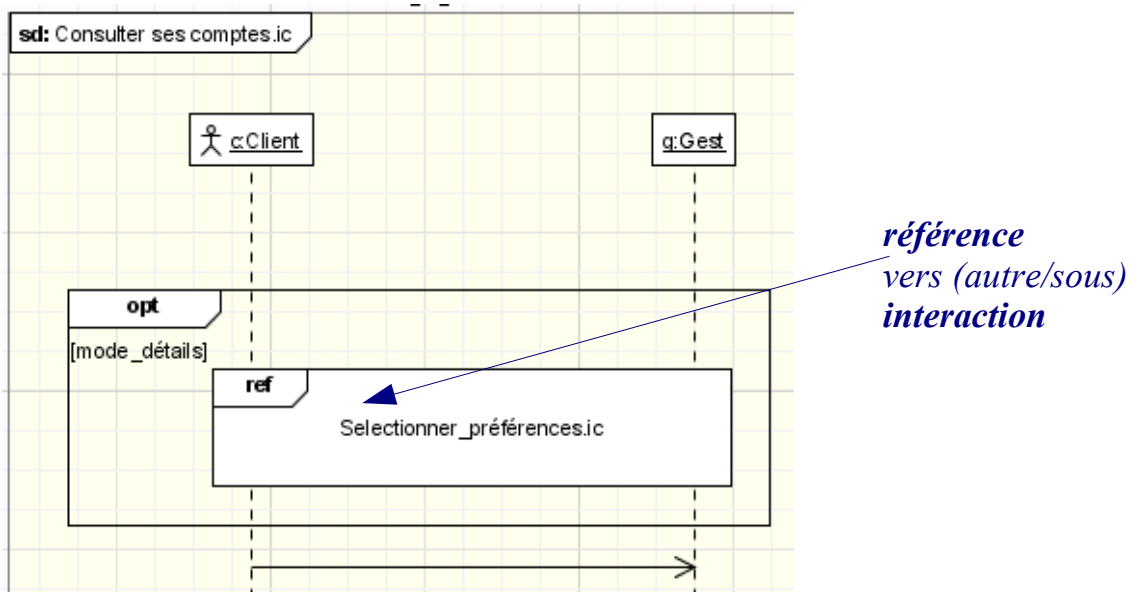
Message reçu = méthode de même nom dans la classe

8. Diagramme de séquences (notations avancées)

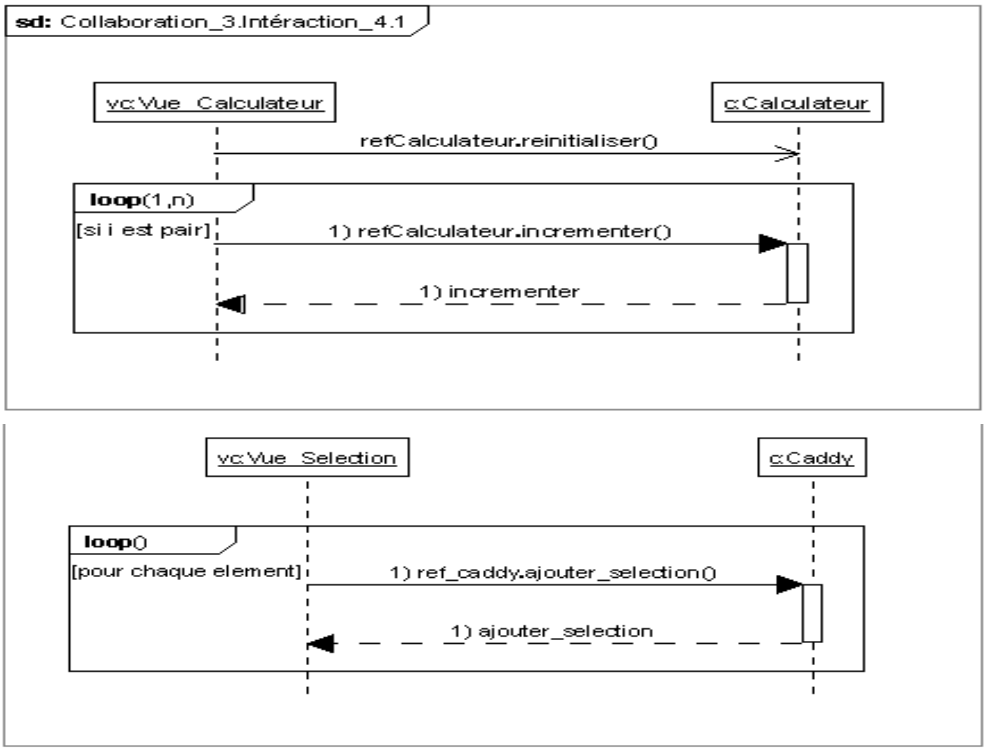
Alternative (**alt**) de UML2 --> *Si [...] Alors ... Sinon ...*
ou Si [...] alors ....Sinon (Si [...] alors ...) Sinon ... / Choice



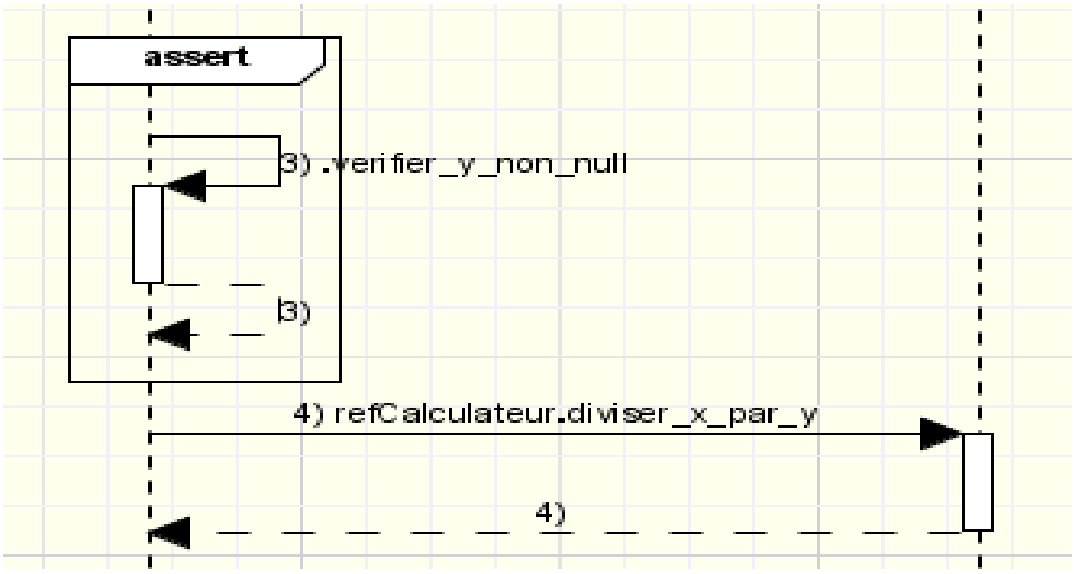
option conditionnée UML2 (**opt**)
---> ***Si*** *[...]* ***Alors*** *(sans else)*



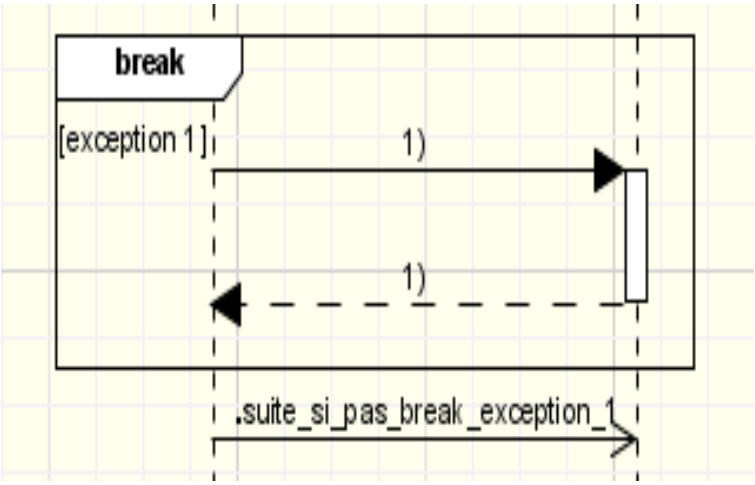
Boucle / Loop (UML2)



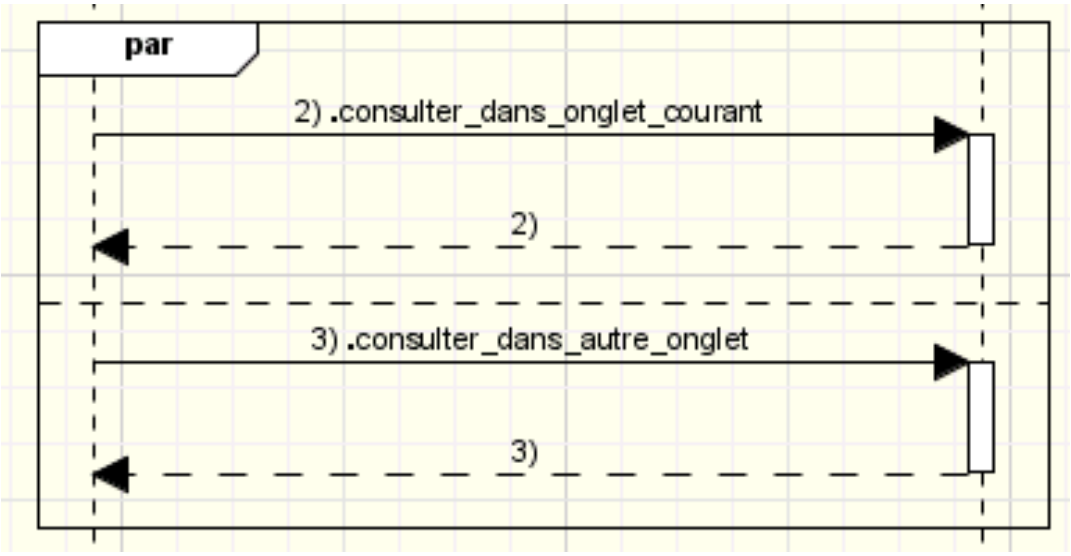
assertion UML2 (**assert**) --> séquence de tests à absolument vérifier pour pouvoir continuer



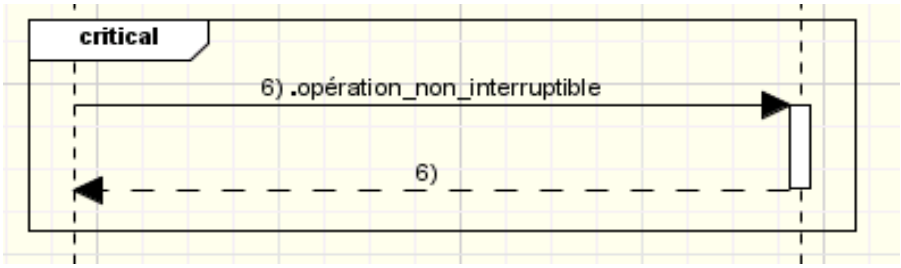
break (UML2) --> traitement en cas d'exception
(***sortie de la séquence normale***, la suite n'est pas exécutée)



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

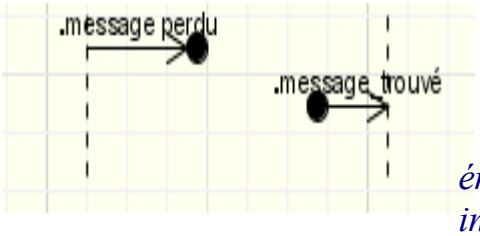
consider { message_important1, m2}

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).



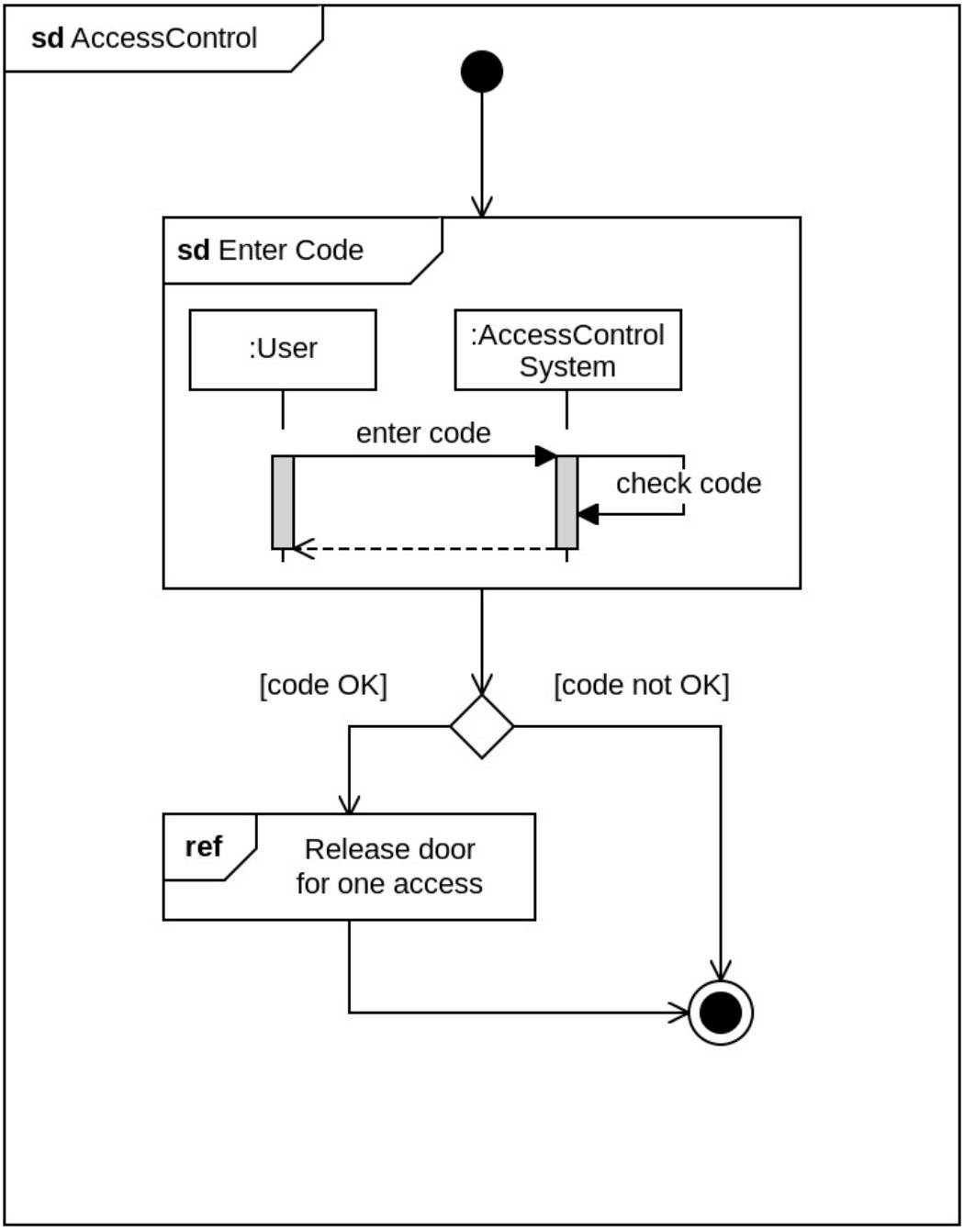
destinataire non identifié

émetteur inconnu

9. Diagramme "*interaction overview" (UML 2)*

Il s'agit essentiellement d'une **variante du diagramme d'activités** où **certaines activités sont modélisées par des cadres qui référencent (ou imbriquent) d'autre diagrammes d'interactions (ex : de séquences)**.

Exemple (tiré de wikipédia) :



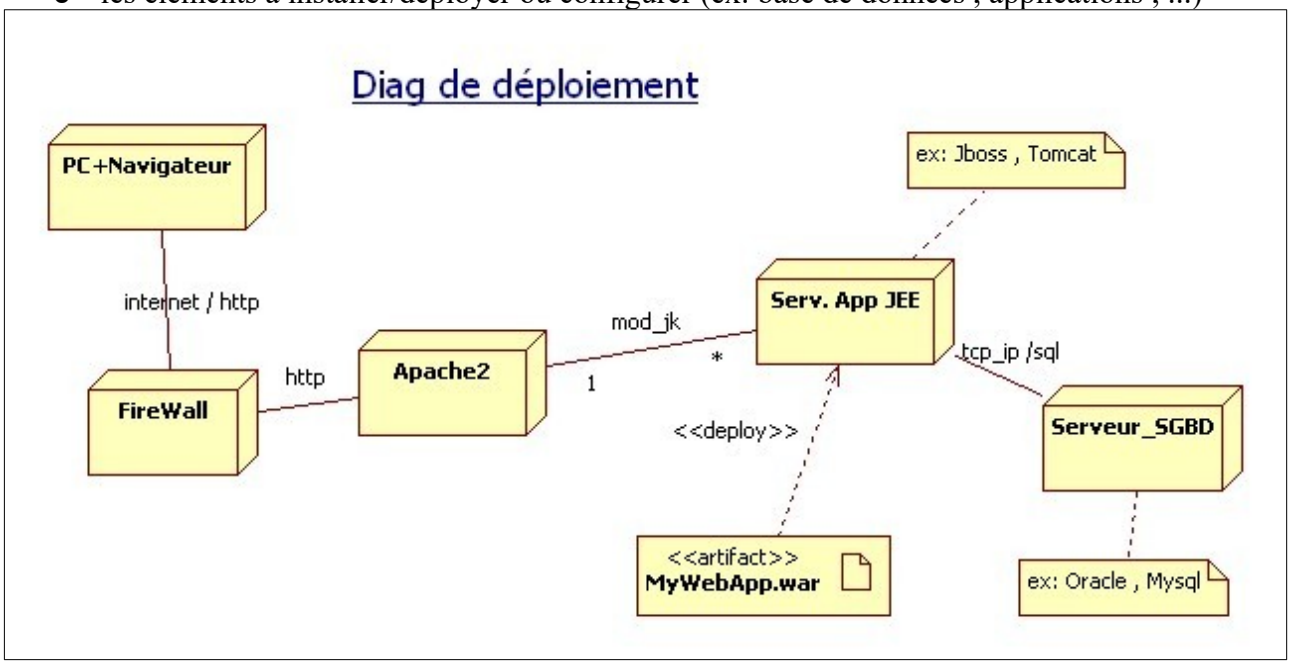
XV - Besoins techniques / env. / diag déploiement

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. Eléments 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 disponiblité 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** à pour **rôle** de définir "**comment**" les choses doivent être **mise en oeuvre**:

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

avec tous les détails nécessaires .

Étymologiquement:

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

pragmatiquement:

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

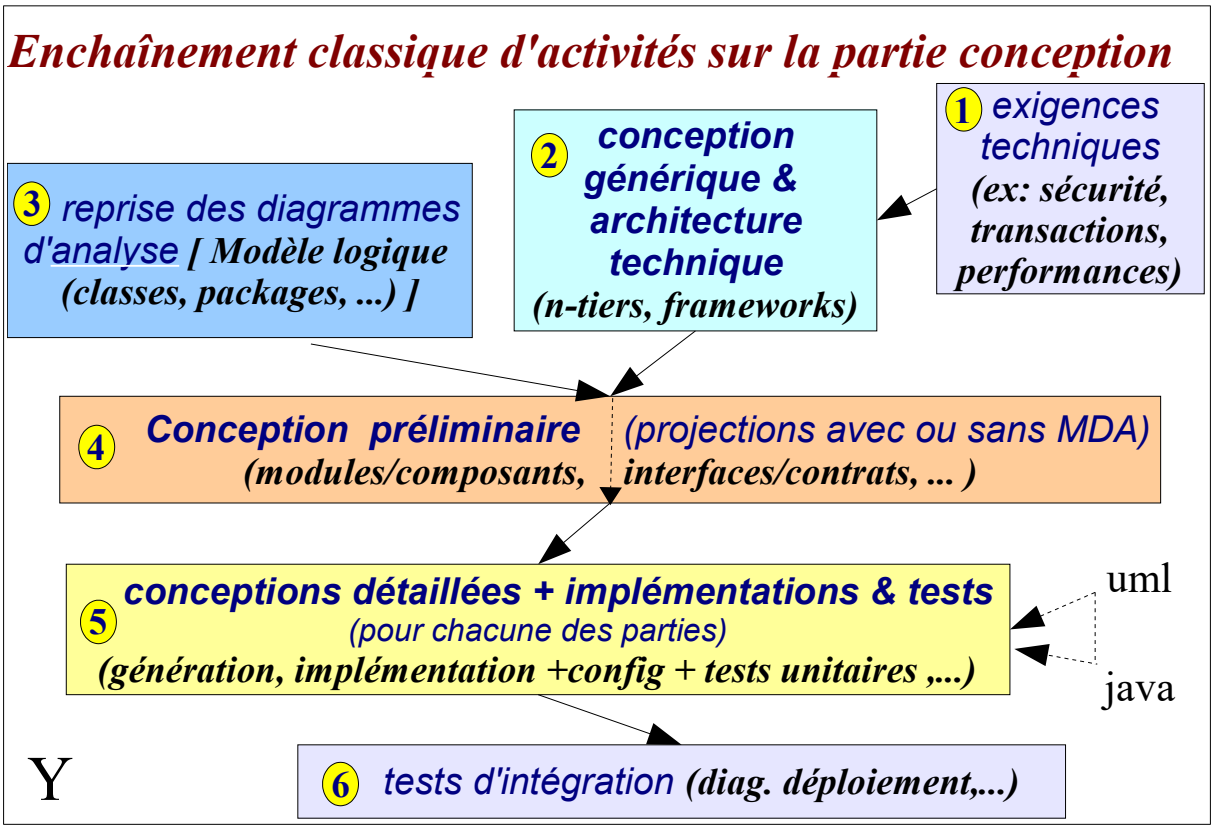
NB :

- Au début des années 1990, il n'y avait pas encore beaucoup de framework parfaitement au point et il fallait à l'époque beaucoup inventer .

- Au début des années 2000, quelques api/frameworks bien structurés (ex : .net/C# , Java/JEE) dominaient le marché et il faillait alors comprendre et intégrer (ne pas ré-inventer)

- A l'aube des années 2020, l'architecture micro-services offre plein de variantes dans les possibilités de mise en œuvre et il faut donc bien argumenter/spécifier les choix d'architecture et avoir quelquefois recours à une double modélisation :
  - cohérence à grande échelle (urbanisation, communications entre applications, ...)
  - modèle orienté objet détaillé à petite échelle (ex : api-rest) .

2. Activités de la conception



2.1. conception générique et architecture technique

indispensable et utile !!! ***(travail d'un "architecte technique")***

- Modéliser une bonne fois pour toute les éléments récurrents.
- Structurer les grandes lignes de l'architecture technique en s'appuyant sur des **frameworks** (prédéfinis ou "maisons") et en tenant compte de l'**état de l'art** : technologies et infrastructures du moment (exemple : api-rest et conteneurs "docker").

2.2. Conception préliminaire

souvent utile !!! *(au moins à "dégrossir" par un* ***concepteur expérimenté****)*

- **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. éventuelles conceptions détaillées

à faire que si certains détails techniques doivent être formalisées (ex : sur gros projet)
---> attention : très gros travail (chronophage , pas toujours rentable, dépendances vis à vis
d'outils ou de technologies manquant de pérennité --> perte de temps potentielle)

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

2.4. implémentation & tests

indispensable et utile !!! ***(travail du "développeur")***

- 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 :

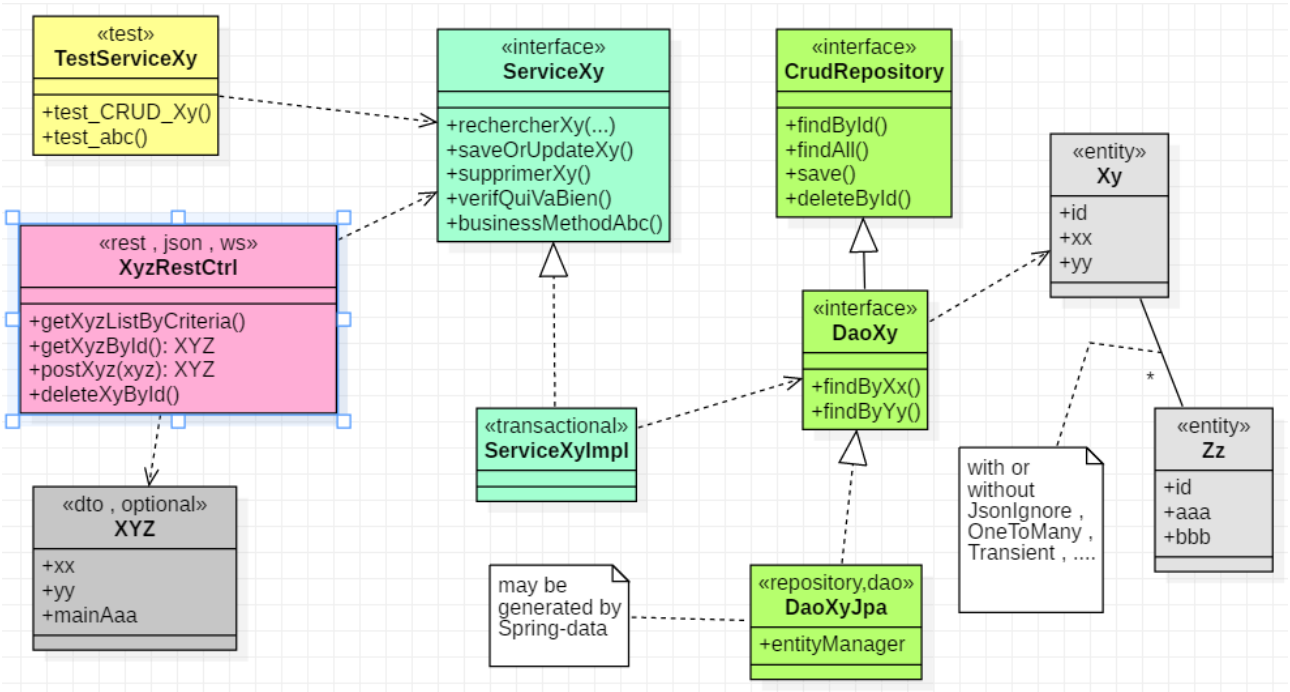
- **répondre aux principaux besoins techniques** préalablement identifiés .
- **bien dissocier l'interface d'une couche de son implémentation** ,

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

- **frameworks techniques prédéfinis** (JEE, MVC2/Struts/JSF2 , AngularJs, ...)
- **frameworks spécifiques** (à batir 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 (d'une api REST) basée sur une des technologies des années 2018_2019 (Spring-boot / Spring-mvc / Spring-Data ....) et devant prendre en charge les exigences techniques d'une application de gestion (transactions, ...) .



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: accéléo / MDA).

4. Projection du fonctionnel dans technologies

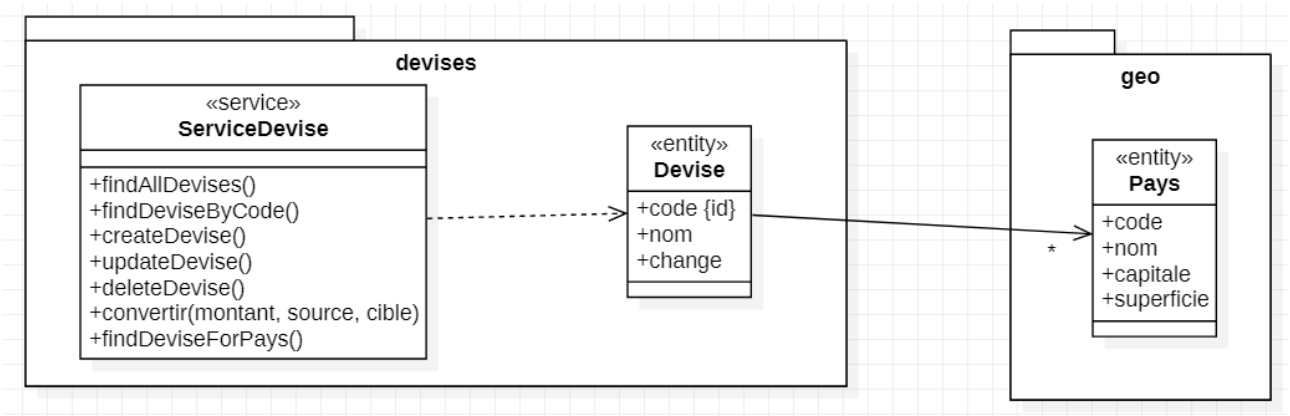
Les activités d'expression des besoins fonctionnels et d'analyse fonctionnelle mènent généralement à un jeu de diagrammes UML qui précisent assez bien :

- les structures de données nécessaires à l'application (en base , pour les échanges/communications)
- les rôles des personnes qui vont utliser le logiciel (acteurs des "cas d'utilisation")
- les principaux "services métiers" de l'application (avec méthodes de traitements essentielles)

Mais tout ceci reste très éloigné de la structure orientée objets exacte d'une application qui dépend quant à elle de :

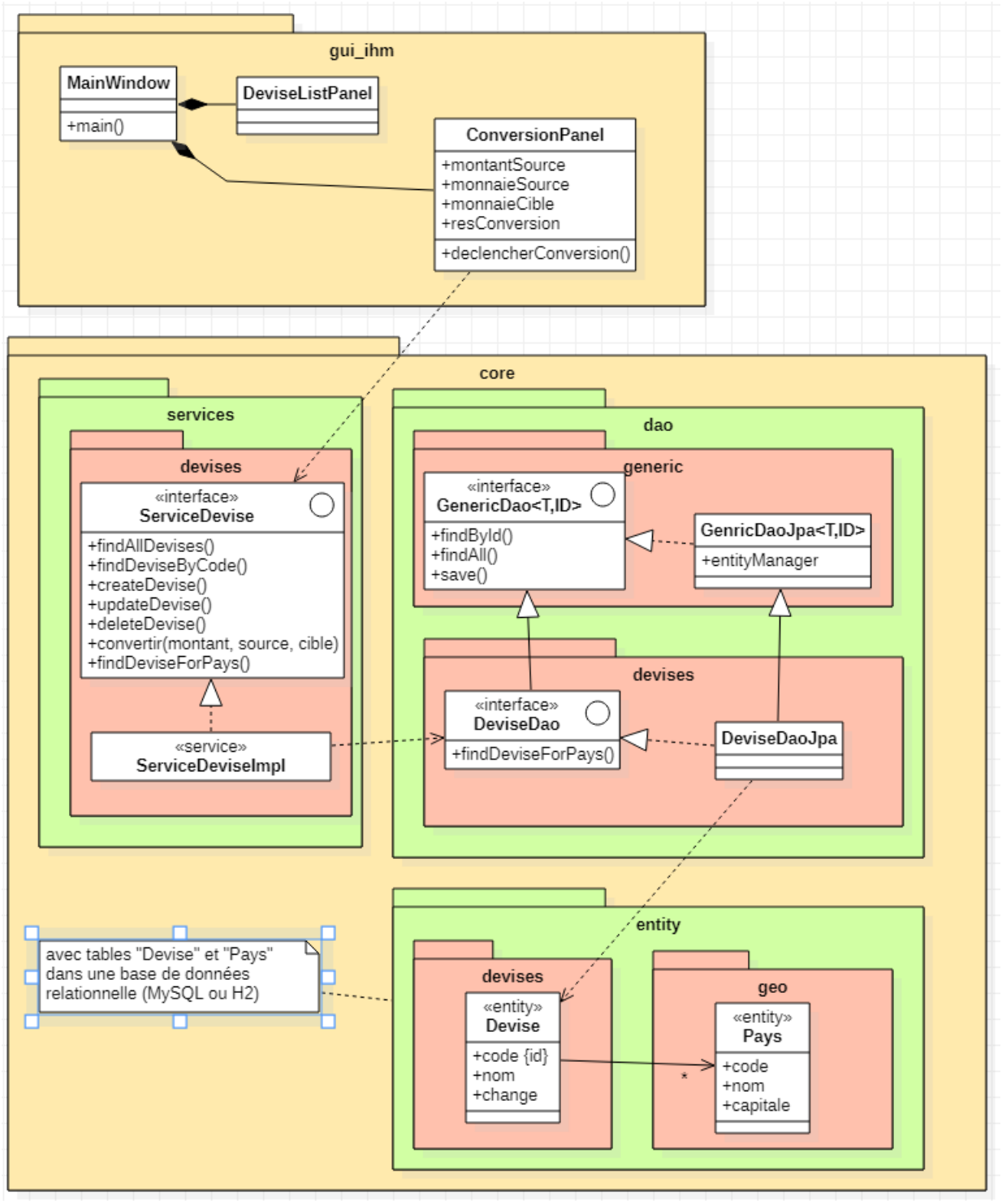
- l'architecture globale (assemblage de technologies , éventuellement distribuées)
- du choix du ou des langage(s) de programmation (avec types de données spécifiques : int ou Integer en java , number en javascript/typescript , ...)
- du choix des frameworks (ex : JPA/Hibernate , Spring , ...)
- du choix des "design patterns" mis en oeuvre (code simple ou évolué)
- ....

Pour bien comprendre l'étendue des alternatives , le mini exemple ci-dessus va montrer une ou deux projection(s) d'une même analyse fonctionnelle dans une (ou 2) des 36000 combinaisons de technologies possibles :



.../...

Projection possible en java embarqué et ihm multi-fenêtrées (non web):

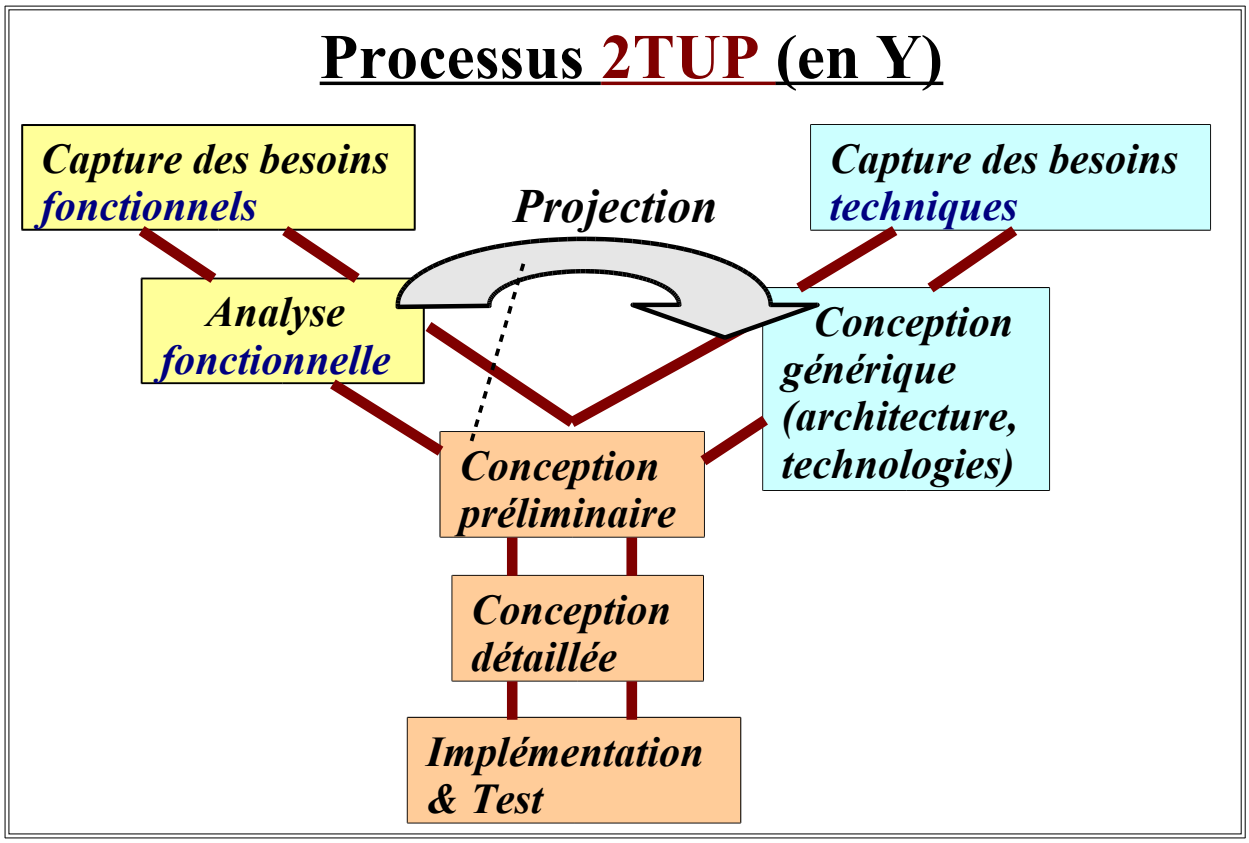


.../...

Future deuxième projection possible en mode "font-end web" + "back-end" ...

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

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

XVII - Implémentation , retours tests , itérations

De façon à progresser , il est **<u>indispensable</u>** 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 ==> nouveau 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 **nombres** minimum et maximum **de fois où une entité précise peut participer à une relation** .

Autrement dit , la **cardinalité** indique les **nombres** 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 en 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 coté "Personne" dans les diagrammes relationnels (MCD,MLD).

Multiplicité (UML):

La **multiplicité** (*exemple*: "0..1" ou "0..*") correspond au **nombres** 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 la 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 coté "**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 - Annexe – UML et mapping objet-relationnel

1. Cas de figures (down-top , top-down , ...)

1.1. Top-down (modèle logique --> base de données)

Dans le cas d'une application toute neuve (sans existant à reprendre) , la base de donnée n'existe pas et elle peut alors être vue comme un des "dérivés physiques" du modèle logique UML .

Certaines règles de conception permettent ainsi de produire un schéma relationnel (avec tables , clefs primaires et étrangères) à partir d'un modèle logique UML (stéréotype <<id>> , associations , rôles).

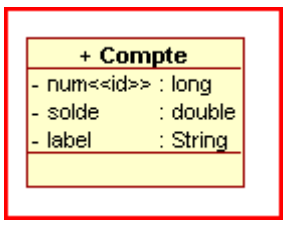
1.2. Down-top (base de données existante --> modèle logique)

Dans un certain nombre de cas, la base de données existe (parce qu'elle est partagée avec un projet existant ou alors parce qu'il s'agit d'une migration d'une application existante vers Java) avant que la conception de l'application ne soit réalisée .
Le passage d'un modèle relationnel à un modèle logique UML s'effectue essentiellement en appliquant (dans le sens inverse) les règles de conception de l'approche "top-->down" .

2. Correspondances essentielles "objet-relationnel"

2.1. Entité (UML) / Table simple

Eléments UML	*Eléments relationnels correspondants*
Classe (avec stéréotype <<entity>>)	Table relationnelle
Attribut(s) avec stéréotype <<id>> (identifiant)	Clef primaire (1 colonne ou +)
Attribut simple (de type String , int , double)	Colonne ordinaire de la table (VARCHAR , INTEGER , ...)



----> **Create Table** ***Compte***(
num LONG ***PRIMARY KEY*** ,
solde DOUBLE,
LABEL VARCHAR(64));

Quelques conseils généraux:

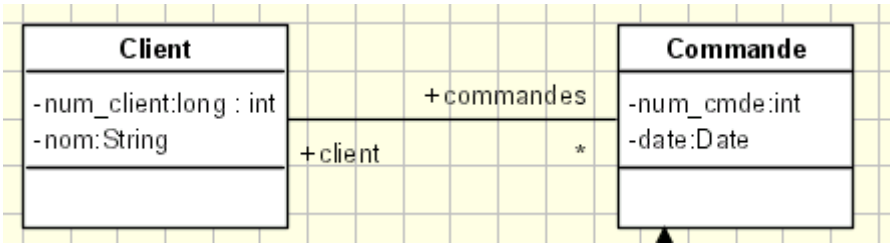
- Eviter des clefs primaires formées par des IDs significatifs (ex: (nom,prenom)) en base , mais préférer des IDs de type compteurs qui s'incrémentent lorsque c'est possible.
- ...

2.2. Association UML / Jointure entre tables

Eléments UML	*Eléments relationnels correspondants*
Association UML (1-n) avec rôles	Clef étrangère (proche du nom du rôle UML affecté à l'élément de multiplicité 1 ou 0..1)
...	

1-1
clef étrangère avec **unique=true**

1-n

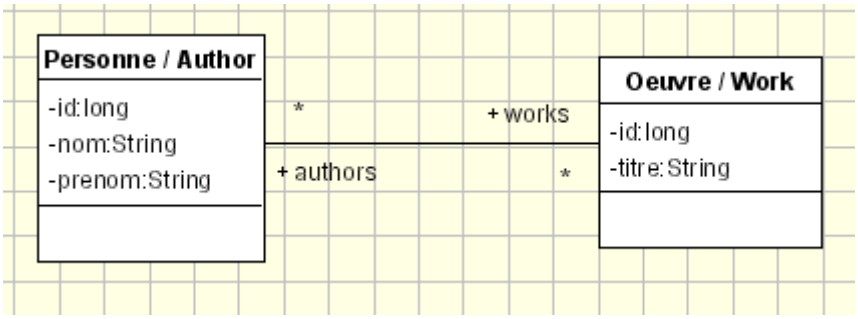


T_Client
num_client (pk) , nom , ...

many-to-one

T_Commande
num_cmde (pk) , **client (fk)**, date, ...

n-n



T_Author
auth_id (pk) , nom ,prenom , ...

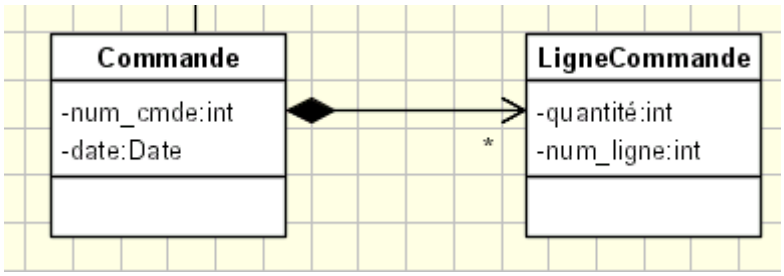
Author_Work
author_id (fk) , work_id(fk)

many-to-many

T_Work
w_id (pk) , titre , ...

2.3. Composition UML / table "détails" avec clef primaire composée

Eléments UML	*Eléments relationnels correspondants*
Composition UML (agrégation forte / losange noir)	Clef primaire composée du coté de la table "détails" . Ou bien éventuellement jointure simple avec opérations en cascade ("cascade delete", ...)
...	



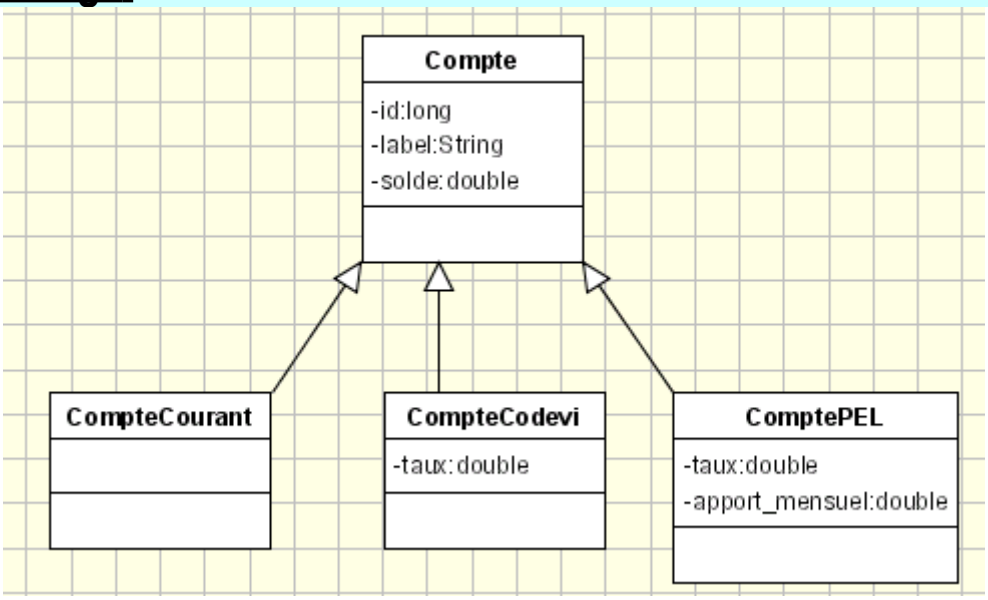
T_Commande
num_cmde (pk) , date, ...

list / composite-element

T_LigneCommande
commande_id (pk) **,** ***num_ligne (pk)*** , quantité

3. Correspondances "objet-relationnel" avancées

3.1. Héritage



Solution/Stratégie 1 : "Une table par hiérarchie de classe" – propriété discriminante

Une seule grande table permet d'héberger les instances de toute une hiérarchie de classe.

Pour distinguer les instances des différentes sous classes , on utilise une propriété discriminante (à telle valeur correspond telle sous classe).

Cette stratégie (relativement simple) est assez pratique et adaptée dans le cas où il y a peu de différences structurelles ente les sous classes .

Solution/Stratégie 2 : "Une table par classe " (joined-subclass)

1 table avec points commun (liée à la classe de base et avec id/pk)

+

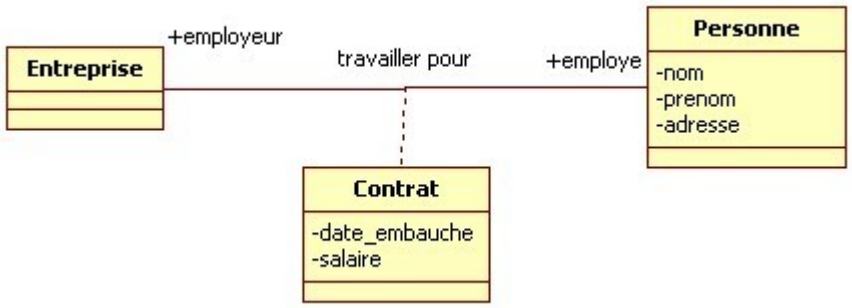
1 table pour chaque classe fille avec "colonnes suppléments" et clef étrangère fk référençant l'id de la table des points communs .

Solution/Stratégie 2 : "Une table par classe concrète"

Cette stratégie n'est pas idéale car elle casse un peu le lien entre sous classe et super classe .

==> problème sur polymorphisme .

3.2. Classe d'association & Table de jointure avec attributs



Contrat est une classe d'association (en UML) . Cette classe comporte des attributs (date d'embauche, salaire,...) qui détaillent l'association "travailler pour" (avec multiplicités non indiquées dans l'exemple ci dessus).

NB: Une modélisation UML à base de classe d'association est de niveau assez conceptuel (un peu comme le MCD de Merise).

==> D'un point de vue plus technique:

- un objet de type "Contrat" sera un objet intermédiaire (en mémoire) entre un objet "Entreprise" et un objet "Personne" .
- Une table "Contrat" (dans la base de données) sera très souvent une table de jointure avec:
  * une clef primaire composée : (idEntreprise, idPersonne)
  * des colonnes supplémentaires : date_embauche , salaire , ...

3.3. Profil UML pour Données relationnelles

Une tentative de normalisation des aspects "données relationnelles" sous forme de "UML Profile" (extension normalisée pour UML) a été initiée par Rose/IBM au début des années 2000 mais n'a pas été réellement suivie.

On peut donc considérer, que les stéréotypes à utiliser pour paramétrer les aspects relationnels sont libres (<<id>> ou <<pk>> par exemple pour indiquer la clef primaire).

Idée à débattre:

Modèle UML stéréotypé (avec <<id>> , ...)

==> génération de code MDA (ex: Accéléo)

==> DDL/SQL (Create table .....)

==> reverse engineering (via Open ModelSphere ou ...)

===> MLD (Merise / relationnel)

4. Stéréotypes UML pour O.R.M.

Bien qu'il n'existe pas de standard véritablement suivi/utilisé sur le sujet, on peut utiliser des stéréotypes UML proches de ceux-ci :

Stéréotypes UML	sémantique	Traduction JPA
<<entity>>	Entité métier persistante avec clef primaire	**@Entity**
<<id>>	Clef primaire (ou partie de clef primaire)	**@Id**
....		

Les associations @OneToOne , @OneToMany , @ManyToMany , ...
peuvent être déduites à partir des multiplicités UML.

Pour paramétrer certains détails (coté secondaire des relations , générateur de clef primaire , ...)
ou pourra éventuellement utiliser quelques **valeurs étiquetées UML (tag Values) considérées quelquefois comme des propriétés des stéréotypes**:

- **generator = auto** (pour auto_increment) ou ...
- **joinColumn** = nom_colonne_clef_étrangère
- **table** = nom_table_si_différent_classeUML
- **inverse**=true (ou secondary = true) pour le coté secondaire d'une relation bidirectionnelle (coté où il y a mappedBy="..." en JPA)
- **notNull=true**
- **length=32**
- ....

XX - Annexes - Patterns "GRASP"

1. Affectation des responsabilités (GRASP)

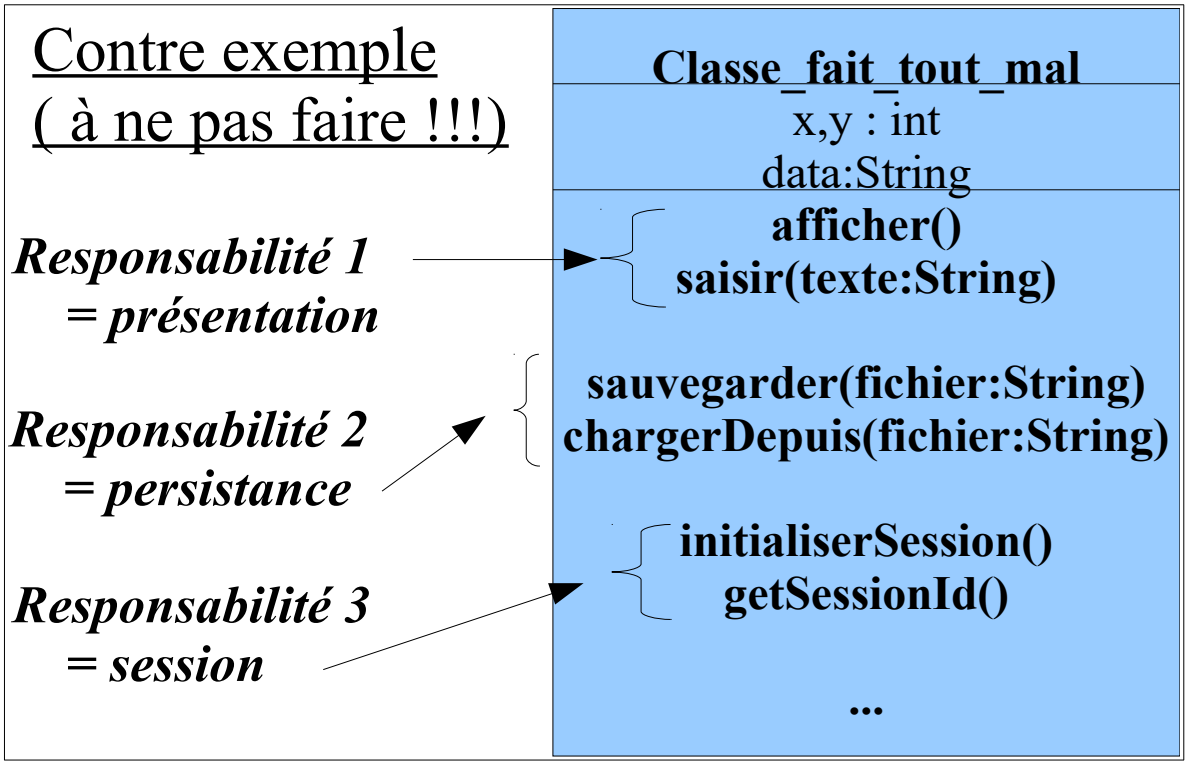
Affectation/répartition des responsabilités (patterns ***GRASP*** de *Craig Larman*)

GRASP = ***G****eneral* ***R****esponsability* ***A****ssignment* ***S****oftware* ***P****atterns*

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 répartion des responsabilités doit conduire à un faible couplage (relative indépendance)

Forte cohésion : la répartion 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 homogénéïté 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)

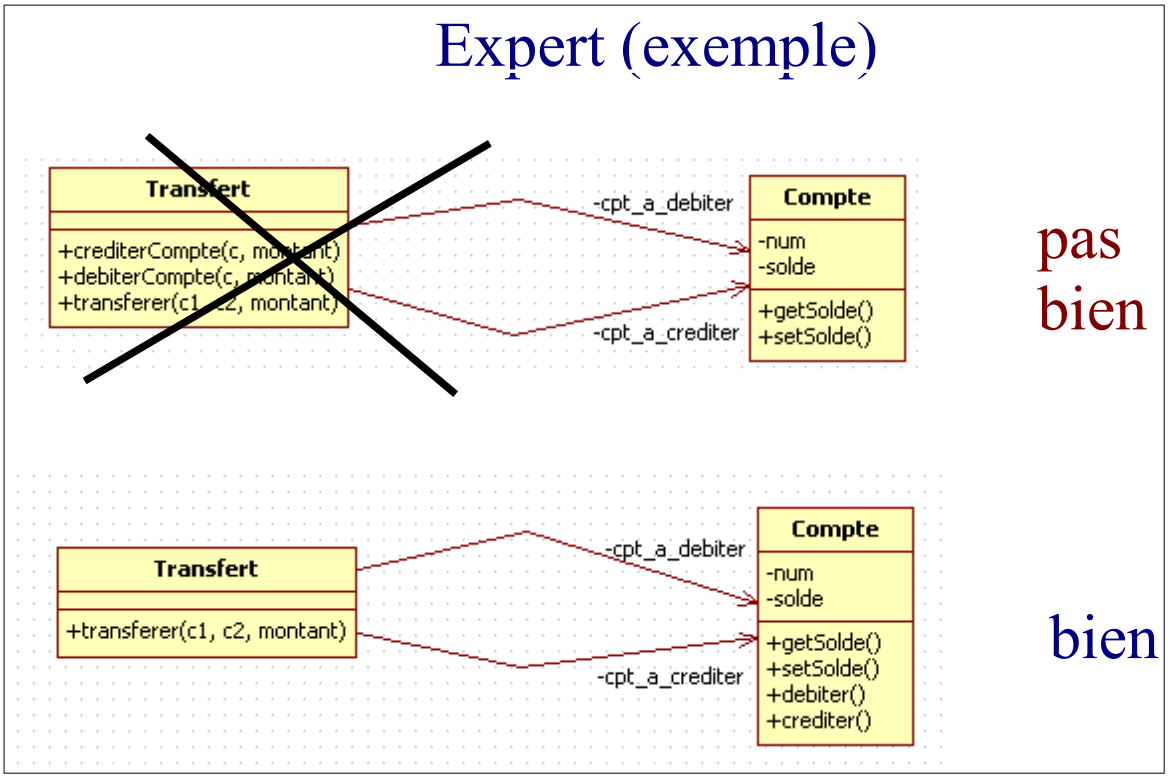
L'*expert*, c'est celui qui sait (qui détient quelques informations clefs) et qui peut donc prétendre pouvoir assumer une certaine responsabilité.

--> 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 monolythique (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)

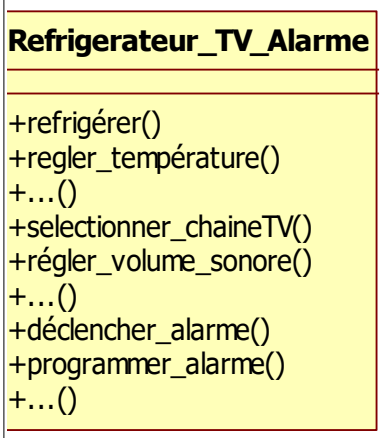
La forte cohésion d'une classe ou d'un système désigne la cohérence fonctionnelle de l'ensemble (la non dispersion des responsabilités)

--> 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)



tout et n'importe-quoi !

2.4. Création

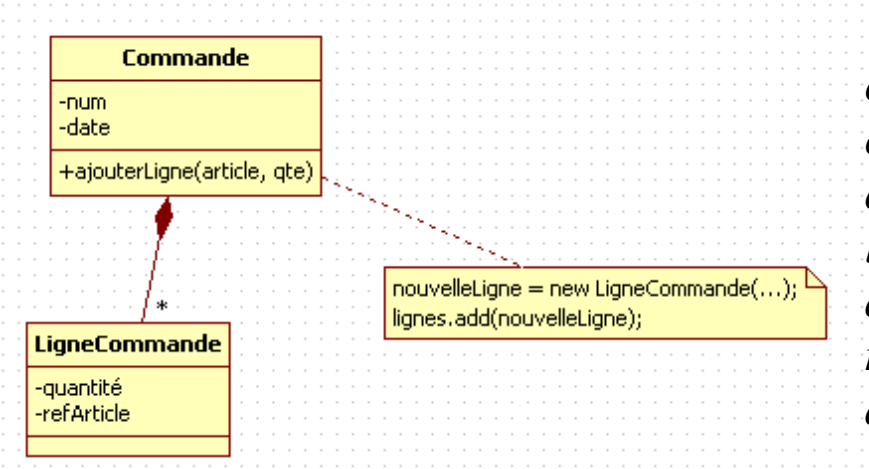
Créateur (GRASP)

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 paterns 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 ensemble 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]*.

XXI - 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 ente 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 !!!

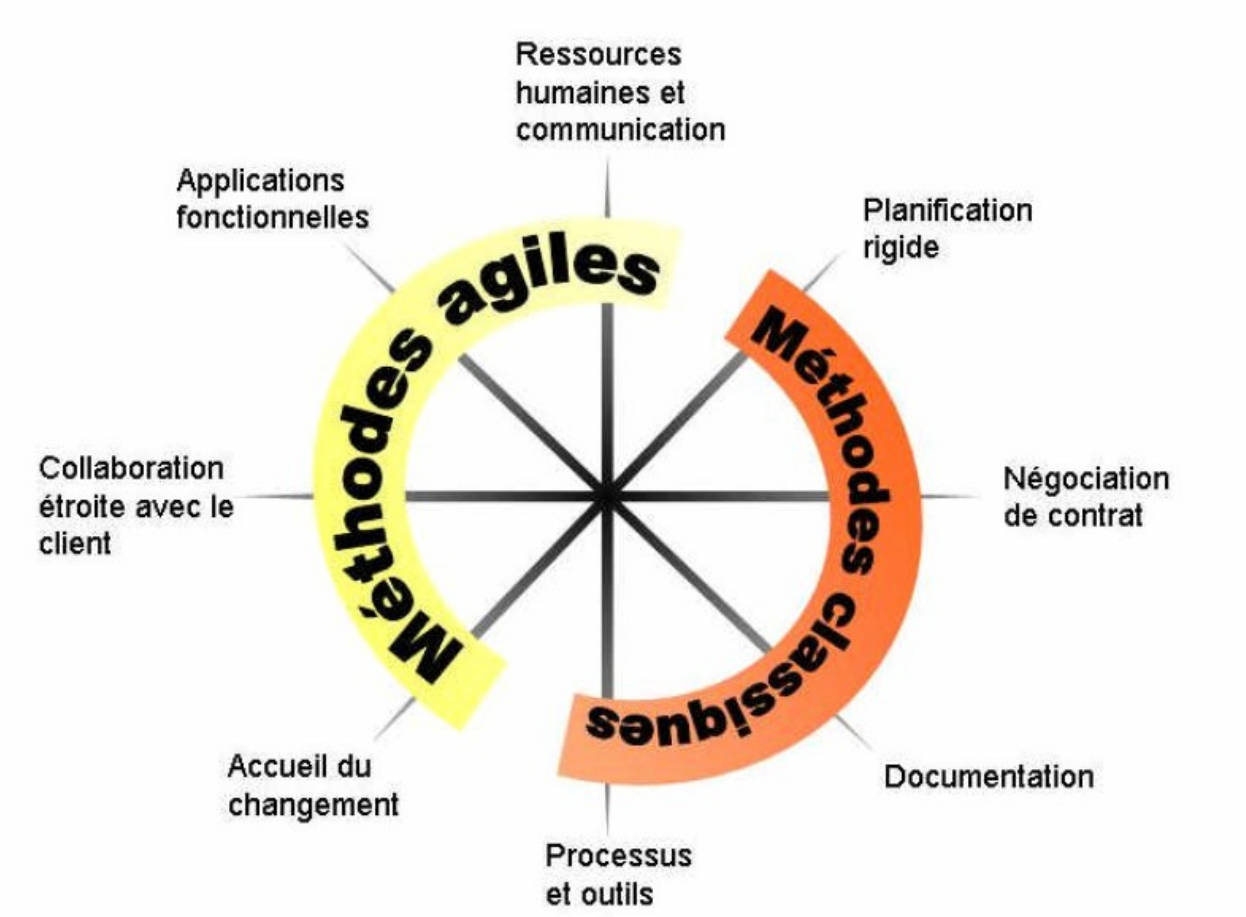
4 grands facteurs:

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

2.2. Principales caractéristiques des méthodes agiles

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

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



2.2.b. Grands Principes des méthodes agiles

- **<u>Délivrer rapidement et très fréquemment des versions opérationnelles</u>, pour favoriser un <u>feed-back</u> client permanent**
- **Assurer une <u>coopération forte</u> entre client et développeurs**
- **Garder un haut niveau de motivation(rôle du chef)**
- **<u>Le fonctionnement de l'application est le premier indicateur du projet</u>**

- **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.)		Ancètre 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 :

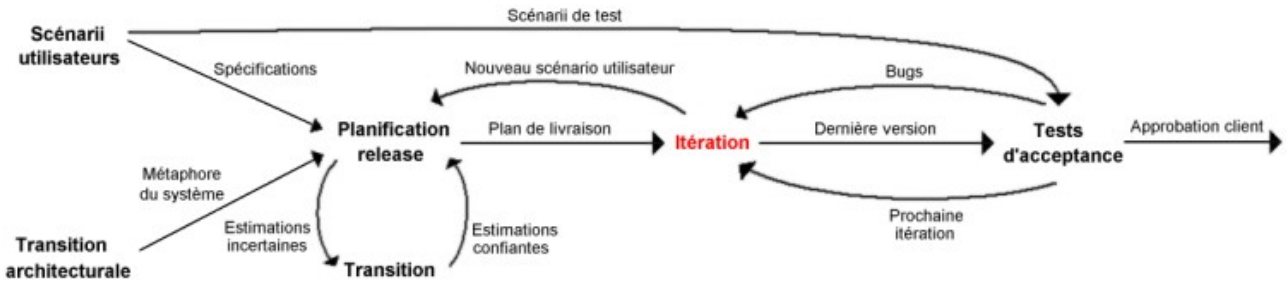
- une tentative de réconcilier l´humain avec la productivité ;
- un style de développement ;
- une discipline de développement d´applications informatiques.

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 extrèmes 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 :

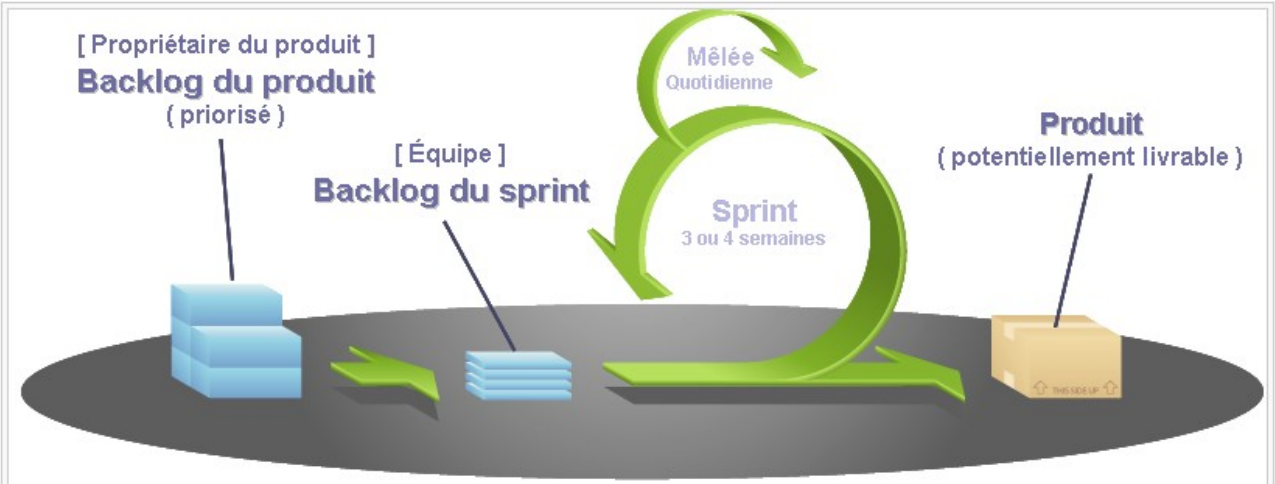
- les premiers jours , le nouvel arrivant observe (en tant que co-pilote) les manières de développer au niveau du projet (environnement de dev , convention de nommage , …)
- et ensuite , c'est "tiens , prends le volant ". Le nouvel arrivant code de son mieux et le développeur rodé au projet vérifie si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

produit	*Owner*	dans lequel les fonctionnalités seront développées et qui prend les décisions importantes concernant l'orientation du projet. Le terme *directeur* n'est d'ailleurs pas à prendre au sens hiérarchique du terme, mais dans le sens de l'*orientation*. 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 utilsateur par exemple)
Equipe *(de développenent , auto gérée)*	*Team*	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. 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.
Facilitateur / animateur	*Scrum-Master*	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. On parle parfois d'**équipe étendue**, qui intègre en plus le *ScrumMaster* et le directeur de produit. Ce concept renforce l'idée que client et fournisseur travaillent d'un commun effort vers le succès du projet.
Intervenants (externes)	*Stakeholders*	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.

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

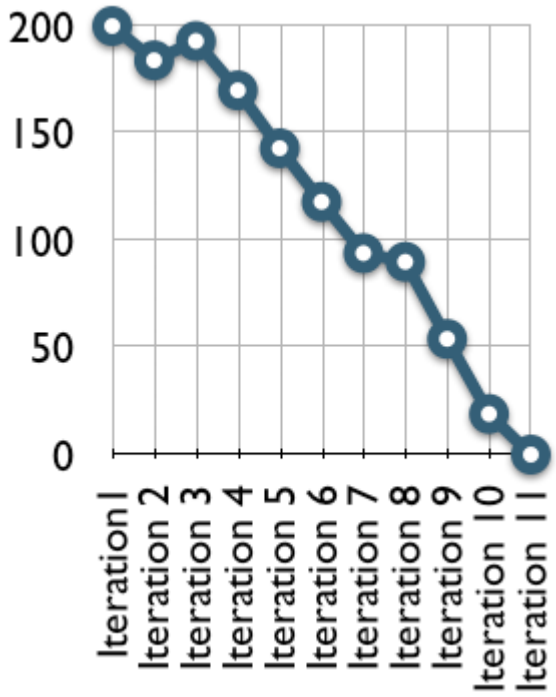
Backlog de produit :

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.

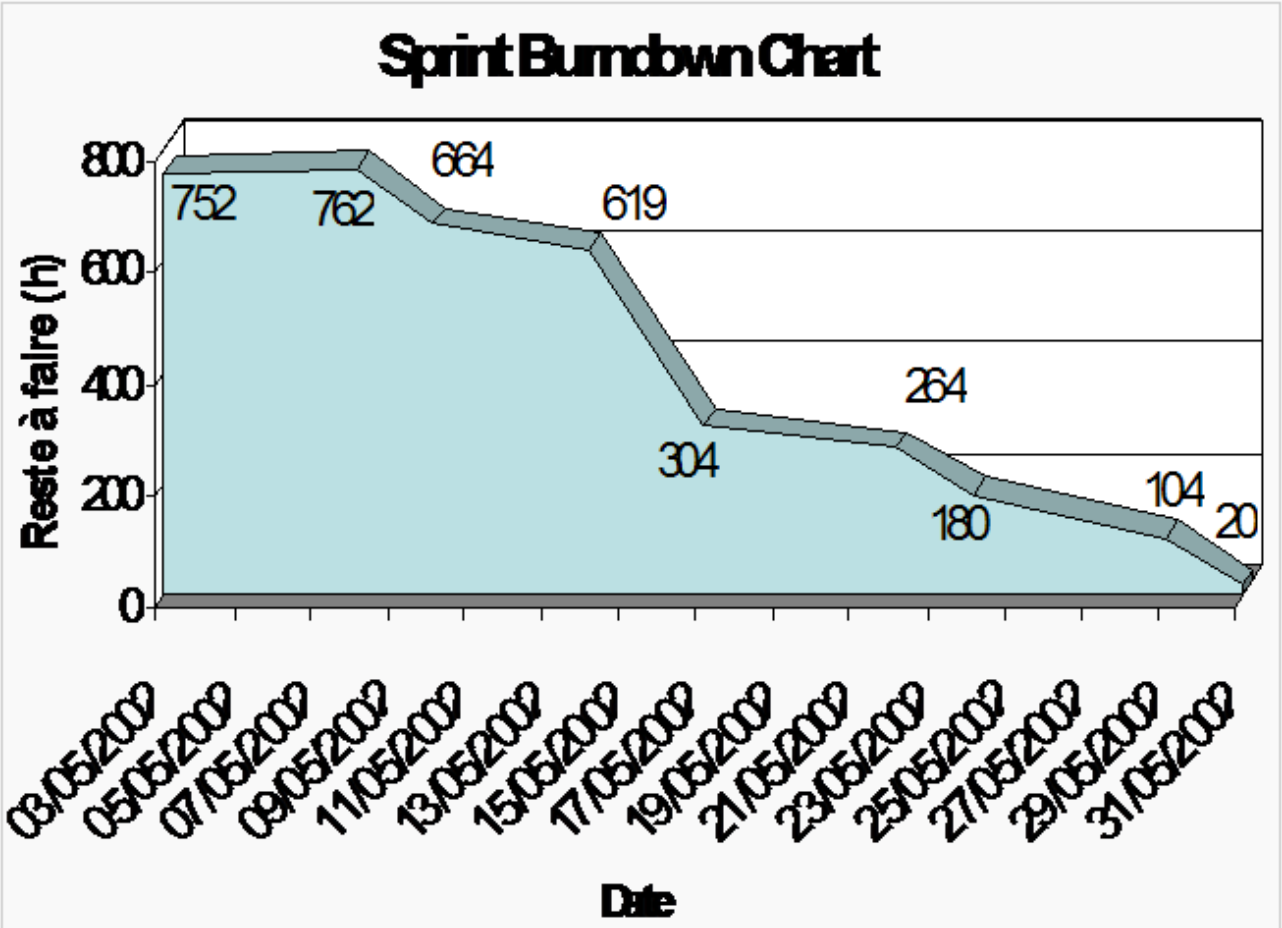


Backlog de sprint :

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. Eléments 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**.

XXII - 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).

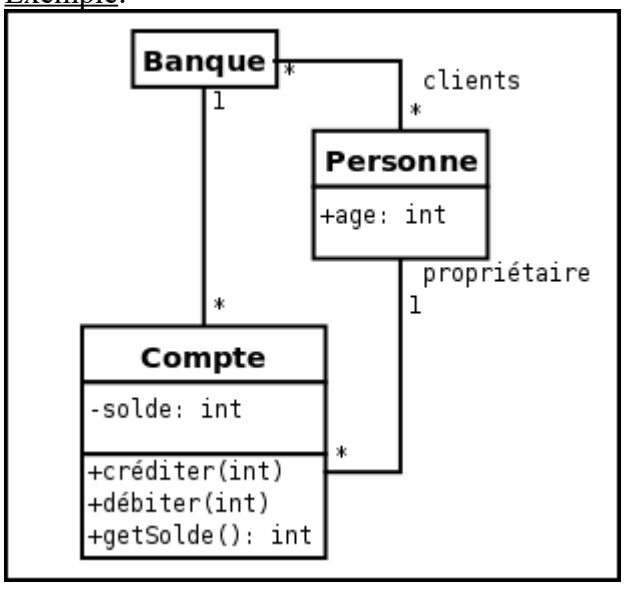
Aujourd'hui OCL est essentiellement utilisé pour:

- 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:



context Compte
inv : solde > 0

context Compte : : débiter(somme : int)
pre : somme > 0
post : solde = solde@pre - somme

context Compte
inv : banque.clients -> includes (propriétaire)

Quelques URL pour approfondir:
http://laurent-audibert.developpez.com/Cours-UML/html/Cours-UML021.html
http://www.omg.org/spec/OCL/2.2/PDF/

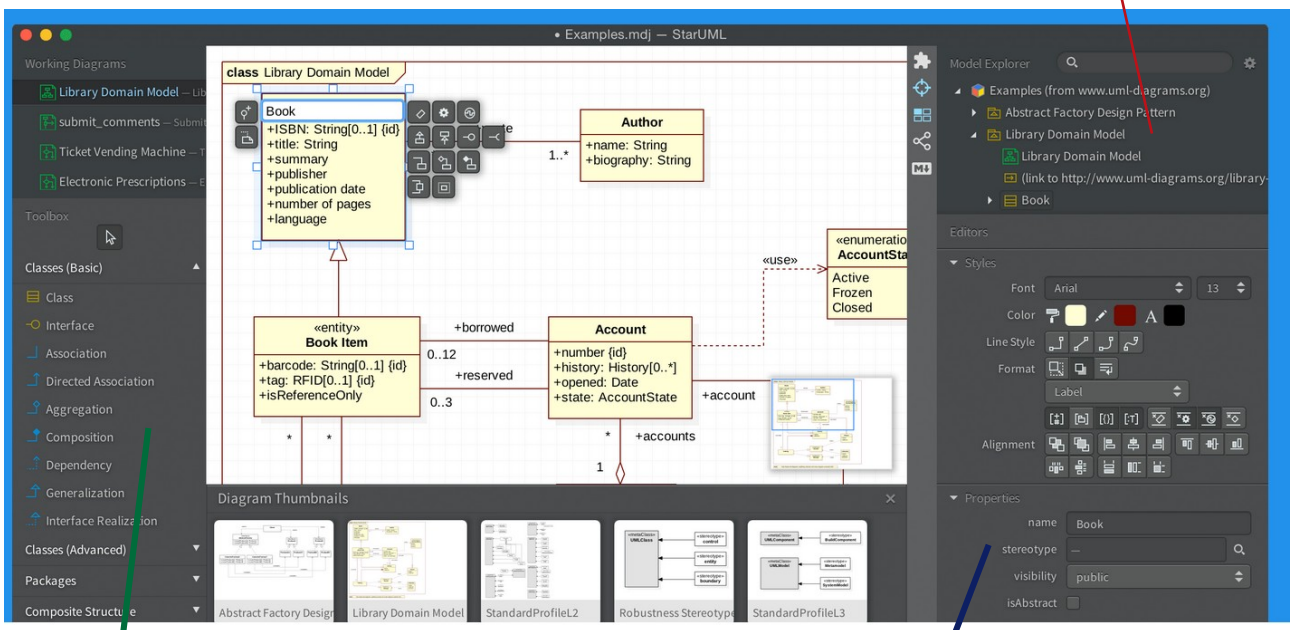
XXIII - Essentiel outil "StarUML"

1. Utilisation Star-UML 3 (outils UML2)

Star Uml (v2,v3) est un outils UML relativement simple et intuitif .

URL du site officiel de "Star UML" : **http://staruml.io/**

navigation dans le(s) modèle(s)



palettes

fenêtre de propriétés (paramétrages élément sélectionné)

documentation officielle ---> **https://docs.staruml.io/**

Quelques généralités :

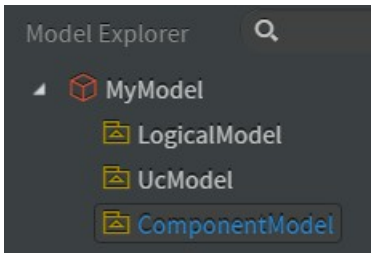
- **Générer une image à partir du diagramme courant :**

  --> **file/ export diagram as ... / PNG ou SVG ou JPEG**

- **Changer couleur de fond et apparences :**
  --> click droit / Format / font , ... color , alignment , ...

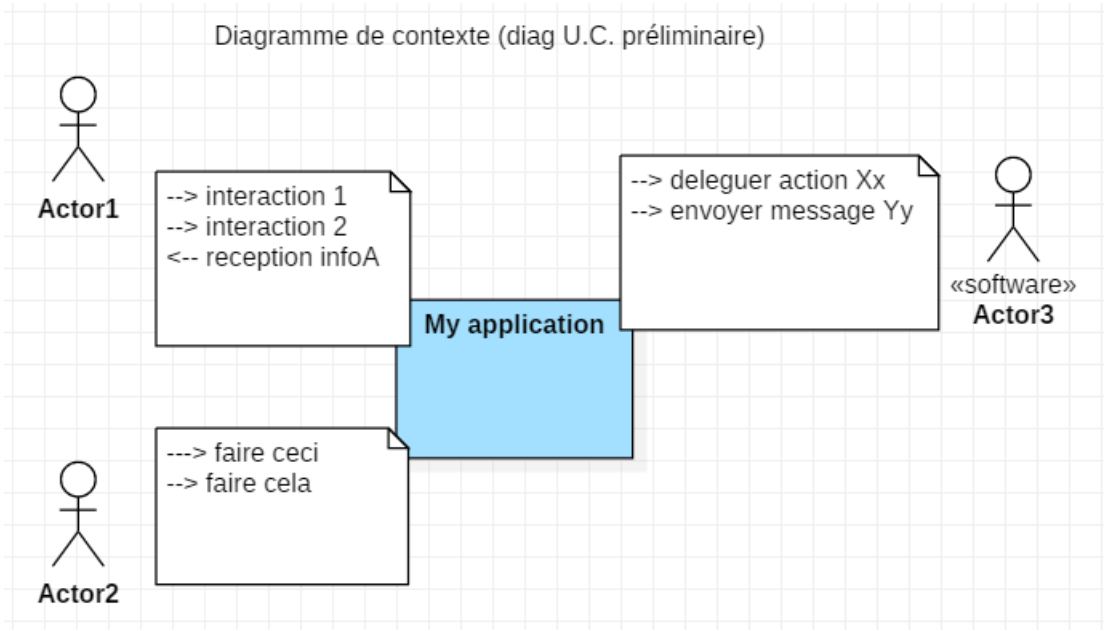
1.1. Organisation des modèles (facultatif)

Via la palette de propriétés , renommer la racine de l'arbre des modèles.
Créer d'éventuels "sous-modèles" .



1.2. Edition d'un diagramme de cas d'utilisation préliminaire (diag de contexte)

Dans "UcModel" (ou ailleurs) , générer un nouveau diagramme de cas "use case" préliminaire appelé "diagramme de contexte" (*ContextUcDiagram*):

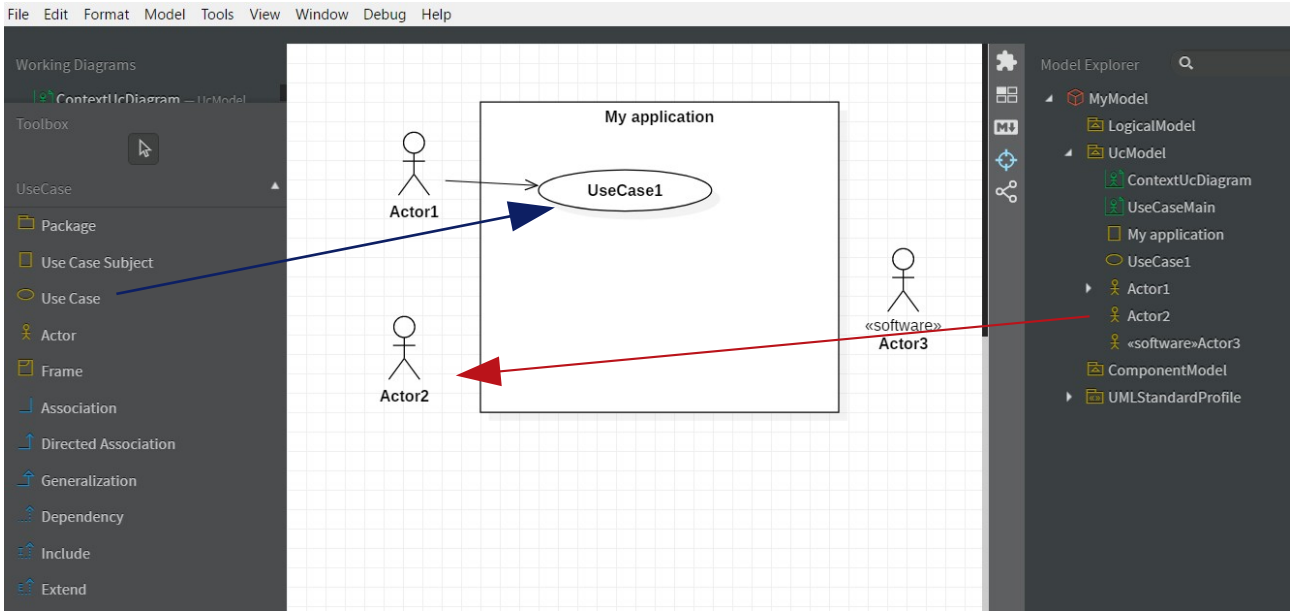


NB : L'appli pourra y être modélisée comme un "Use Case Subject" et les interactions utilisateurs pourront y être modélisées au sein de commentaires (Notes) placés à coté de chaque acteur (rôle utilisateur) .
Si un acteur représente une autre application, on pourra lui affecter le stéréotype <<software>> via la fenêtre des propriétés .

1.3. Edition d'un diagramme de cas d'utilisation

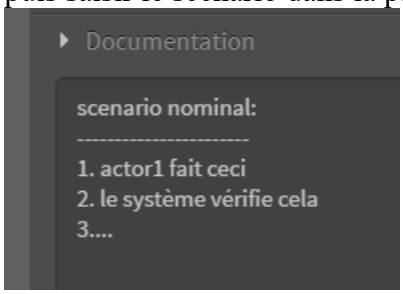
Créer un nouveau diagramme de cas d'utilisation (ex : UseCaseMain) et y placer les acteurs et l'apllication préalablement modélisés via des "drag & drop" .

Les autres manipulations sont "hyper intuitives" .



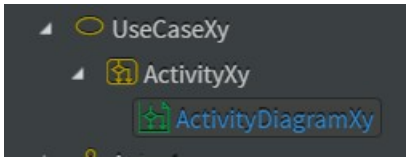
1.4. Edition rapide d'un scénario (pour U.C.)

Sélectionner un "Use Case"
puis saisir le scénario dans la partie "Documentation" :



1.5. Ajout d'un diagramme d'activité (en tant que détails d'un U.C.)

Sélectionner un "Use Case" dans l'arborescence du modèle (éventuellement indirectement via "select in explorer").
Déclencher le menu "add diagram / activity diagram" .
Renommer éventuellement les noms des activités et digrammes :

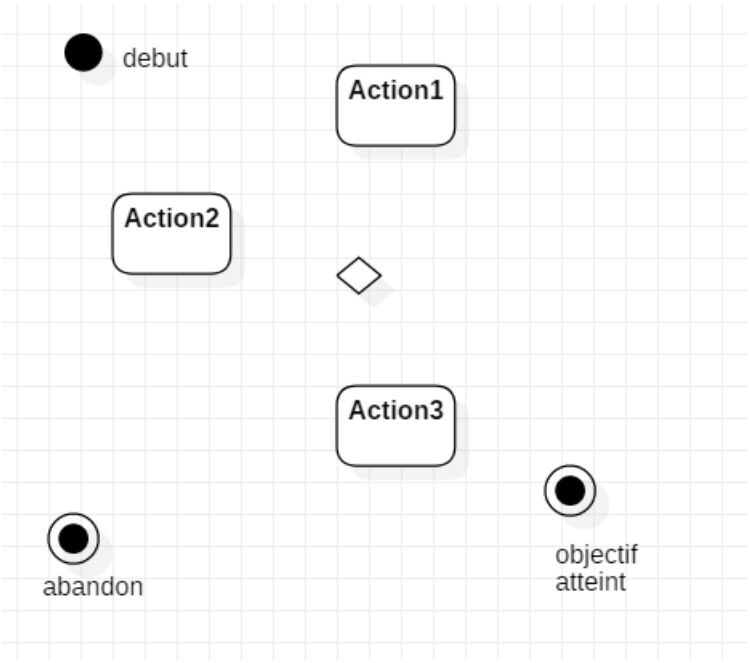


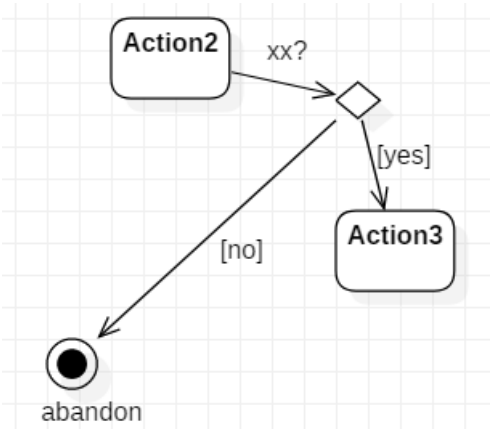
1.6. Edition d'un diagramme d'activité

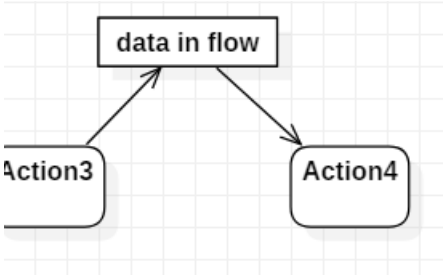
1. placer (si besoin seulement) les "couloirs/partitions/swim lanes"
2. placer assez tôt un "debut" et une (ou plusieurs) fin(s)

placer si besoin du "texte" à coté des "initial" et "final" .

3. placer certaines actions , décisions, ...
   placer si besoin du "texte" à coté des "décisions"
4. placer seulement ensuite les "control flow"
5. les conditions (alias gardiens) doivent être précisées via la propriété "**gard** :" et devraient normalement apparaître entre des crochets [ ]

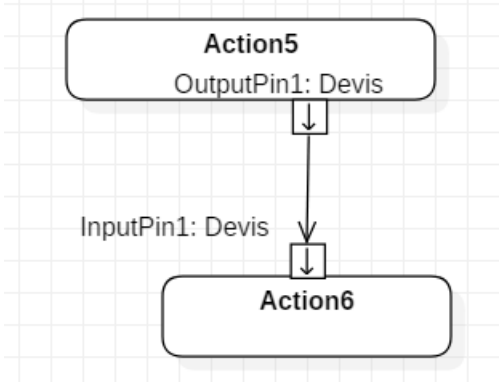


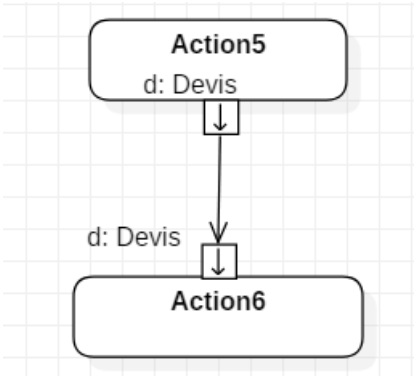




(via **Object Node** & Control Flow)

ou bien

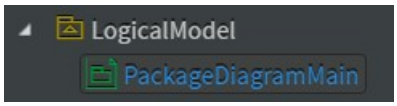


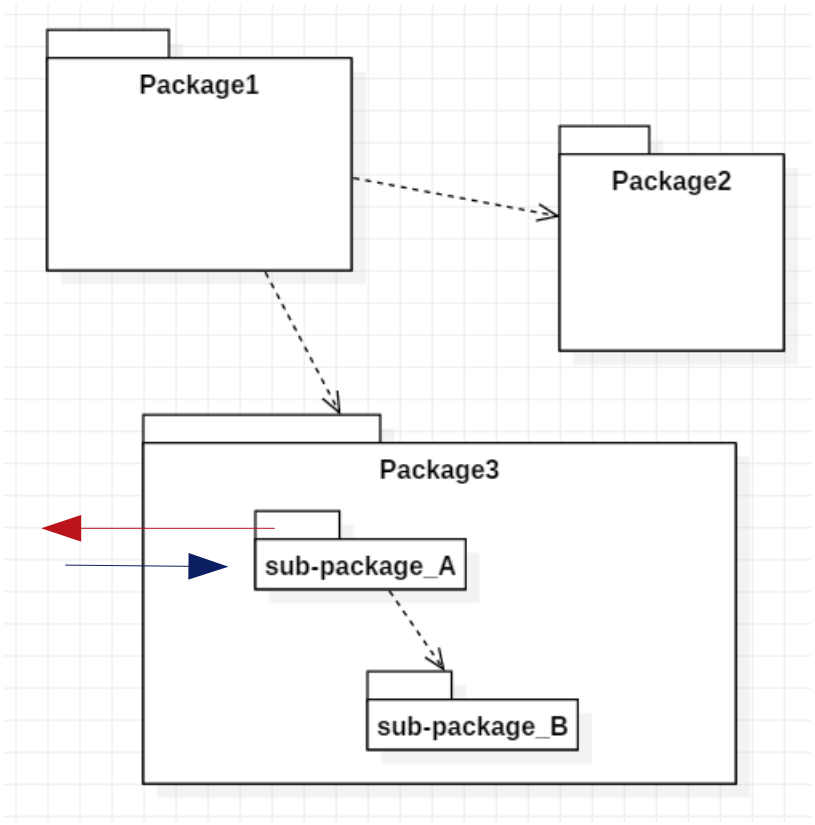


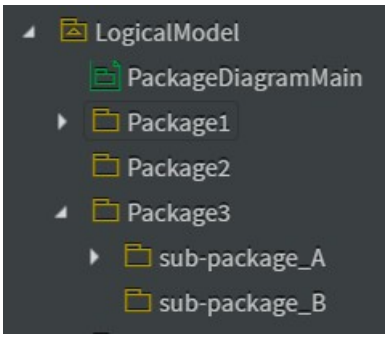
en typant (Devis) et en renommant (d) les "**outputPin**" et "**inputPin**"

1.7. Organisation des diagrammes de packages & classes

Un diagramme de package (dans "LogicalModel" ou ailleurs) est un bon point d'entrée pour la partie "modélisation structurelle / logique" :

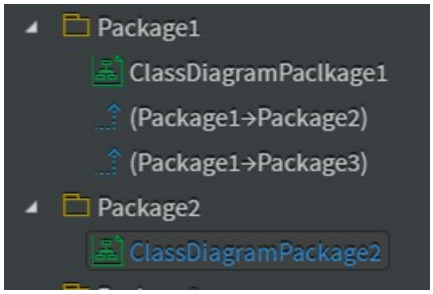




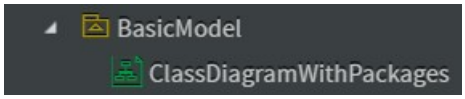


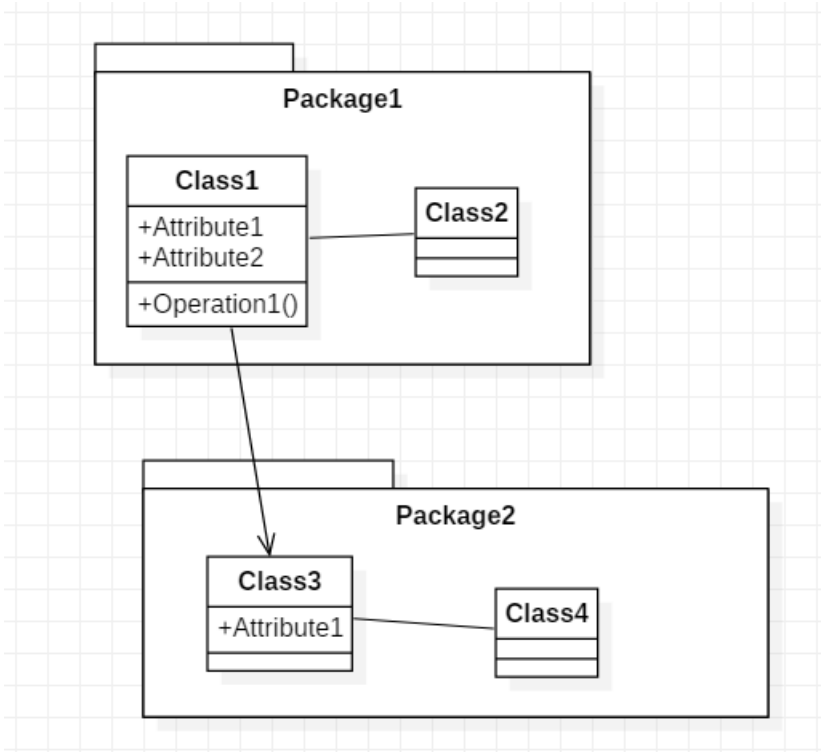
Astuce : il faut sortir puis ré-insérer un sous package dans son package parent pour que l'imbrication soit bien comprise (prise en compte) par "star uml" .

On aura souvent besoin de créer au moins un diagramme de classes par package :



NB : Dans un cas ultra simple (ou bien pour apprendre les bases d'UML sans trop de complexités inutiles) on pourra éventuellement tout placer dans un seul gros diagramme de classes :



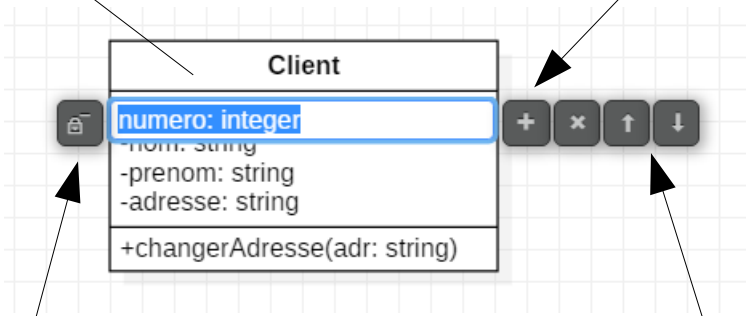


mais en pratique , sur de véritables projets informatiques , ceci n'est pas réalisable (car manque de place !!!!) .

1.8. Edition d'un diagramme de classes

click droit, add attribute , add operation, ...

créer plusieurs attributs les uns à la suite des autres

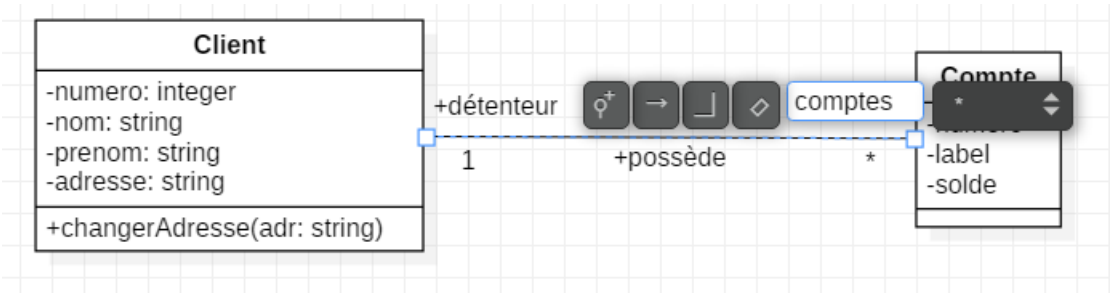


visibility (- private , + public)

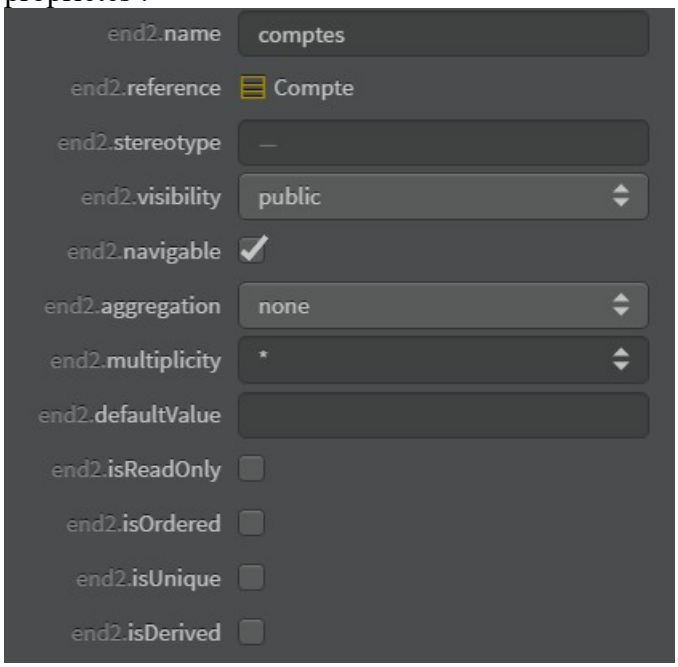
changer l'ordre des attributs

Pour préciser le type d'un attribut , on peut directement saisir "numero **: integer**" au clavier .
idem pour les types des paramètres des opérations (et les types de retour).

Dans le cadre d'une association entre 2 classes, les **multiplicités** s'ajustent via un **menu contextuel** apparaissant lorsque l'on se place sur une des extrémités de l'association sélectionnée :

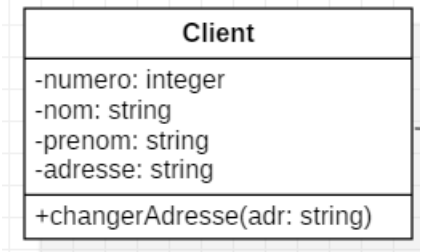


NB : toutes les propriétés d'une association sont finement paramétrables via la fenêtre de propriétés :



Par exemple, en décochant "***navigable***" , on rend une association unidirectionnelle (avec une flèche de navigabilité sur l'extrémité inverse).
Et en basculant "***aggregation***" de "none" à "***shared***" ou "***composite***" , on ajoute un losange blanc ou noir sur l'extrémité paramétrée .

Pour montrer (ou cacher) des détails sur les éléments d'une classe , on pourra activer le menu contextuel **Format ... / Show .... (ou Suppress ...)**

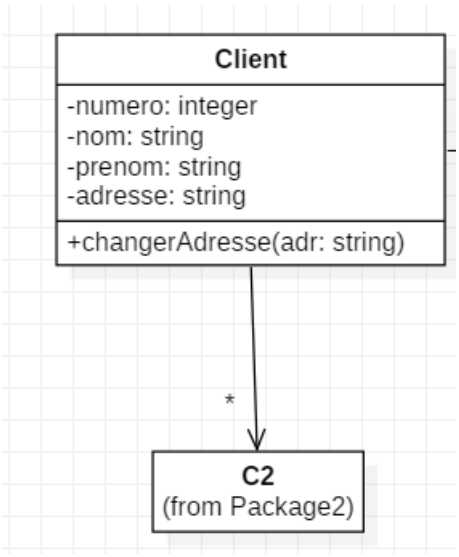


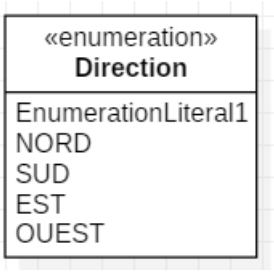
(suppress attributes & operations)

Besoin fréquent : une même classe peut être montrée avec plein de détails sur un premier diagramme et montrée sans détails sur d'autres diagrammes (via drag & drop et format .../ suppress attributes & operations) .

NB : si la classe "C2" appartient au "package2" , alors dans un diagramme de classes associé au "package1" contenant la classe "Client" on pourra faire apparaître "C2" via un drag & drop (idéalement suivi de format .../ suppress attributes

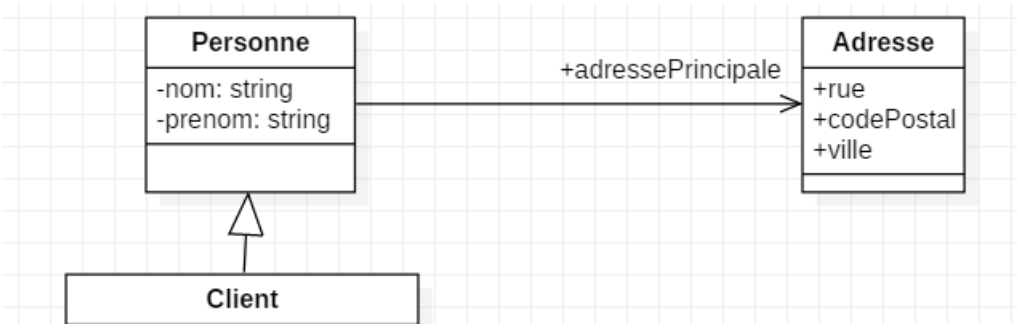
& operations) de façon à montrer une association entre la classe "Client" du package courant et la classe "C2" d'un autre package :



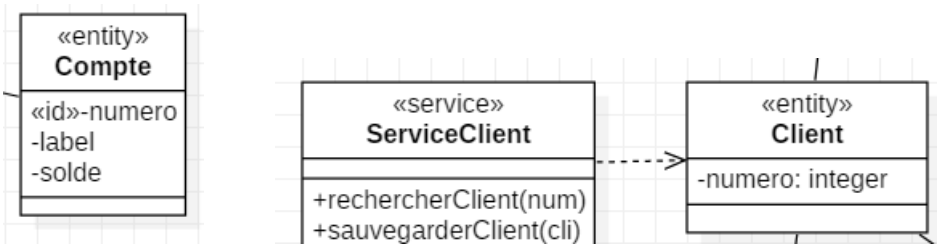


Enumeration avec "enumerationLitterals" :

NB : Le rôle "adressePrincipale" de "Adresse" associé à "Personne" sera ultérieurement vu comme une variable d'instance de "Personne" (**pas besoin** d'un attribut adressePrincipale : Adresse dans la classe Personne , sinon doublon).

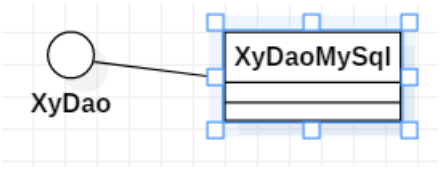


Ajout de **stéréotype** (par saisie directe dans la fenêtre de propriétés)



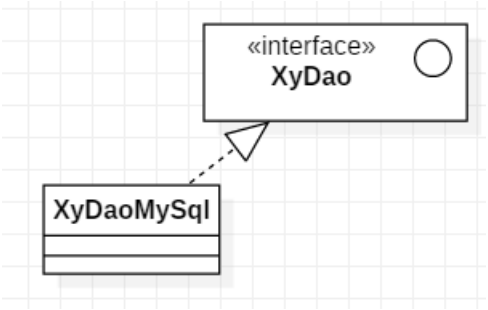
Interfaces

par défaut une interface (en UML) est représentée par un cercle (vision "point d'entrée de certains traitements d'une classe reliée à l'interface via une interface Réalization") :

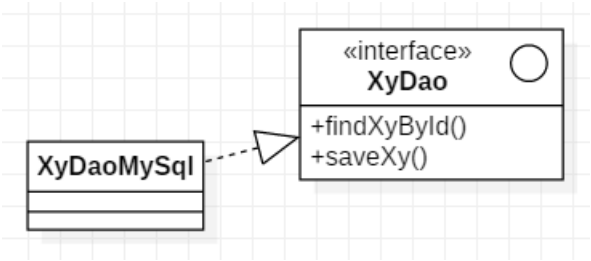


Si l'on souhaite montrer certains détails (ex : liste des opérations de l'interface = contrat), on aura souvent besoin de basculer de notation via un changement d'affichage du stéréotype.

Pour cela , il faut sélectionner l'interface (le cercle) et activer le menu contextuel "**format .../ stéréotype display .../ label ou label with decoration**"

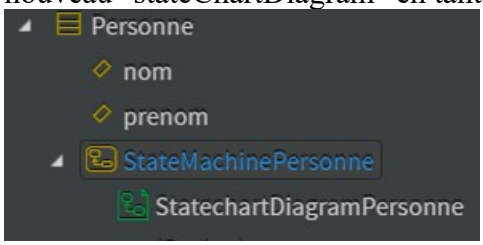


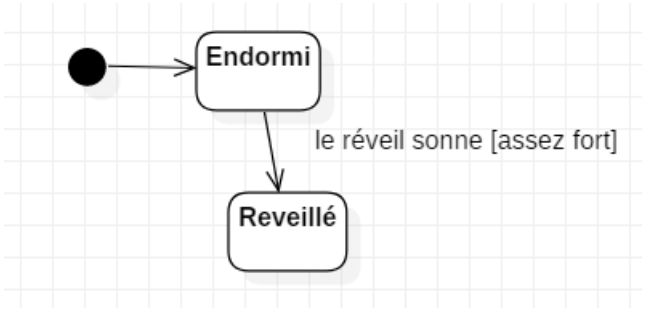
On peut ensuite faire apparaître la zone des opérations de l'interface via le menu contextuel "***Format .../ Suppress attributes (à décocher)***" .



1.9. Edition d'un diagramme d'états

Un diagramme d'états représente souvent les états d'un objet (ou d'un sous système).
On peut souvent sélectionner une des classes de l'arborescence du modèle pour ensuite créer un nouveau "stateChartDiagram" en tant que détails .

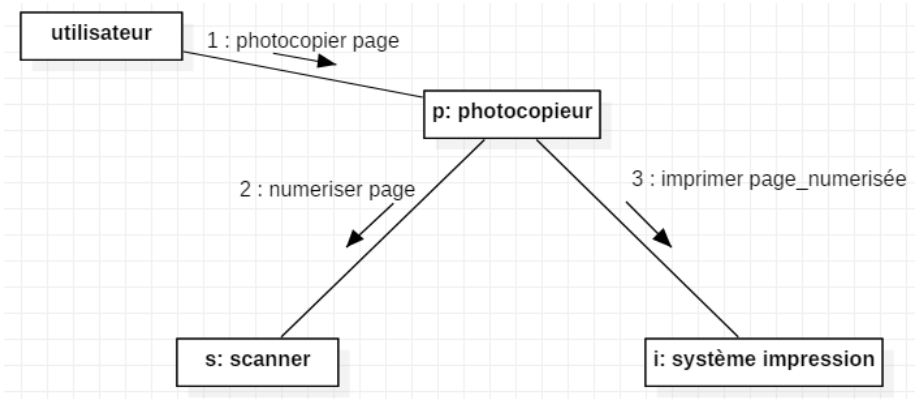




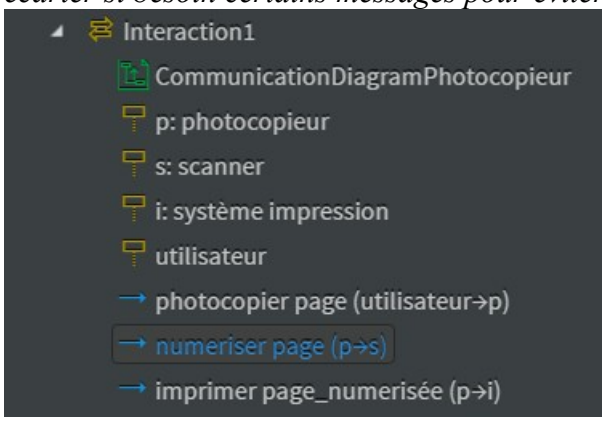
condition "assez fort" en tant que **gard**(ien) .

1.10. Edition d'un diagramme de communication

Quelquefois en tant que détails d'un "use case" .



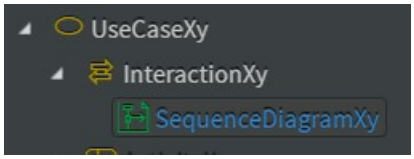
écarter si besoin certains messages pour éviter des superpositions illisibles .



"***Move Down"*** , "***Move Up***" dans l'arborescence du modèle pour **changer l'ordre** *(la numérotation)*

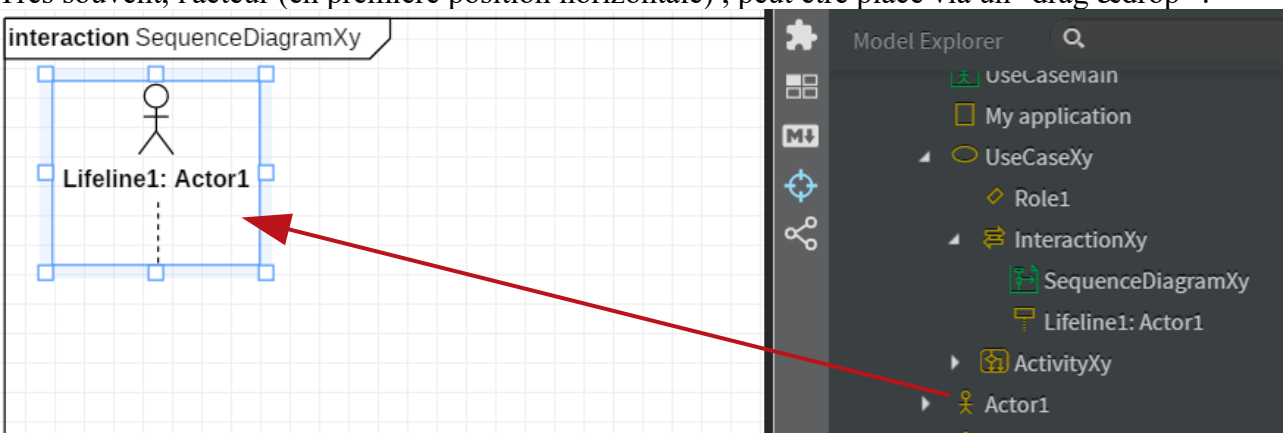
1.11. Edition d'un diagramme de séquence

Un diagramme de séquence sert souvent à illustrer (ou reformuler) un scénario lié à un cas d'utilisation.
Il est donc assez souvent pertinent de sélectionner un "use case" avant de créer un nouveau diagramme de séquence (en tant que détails) :

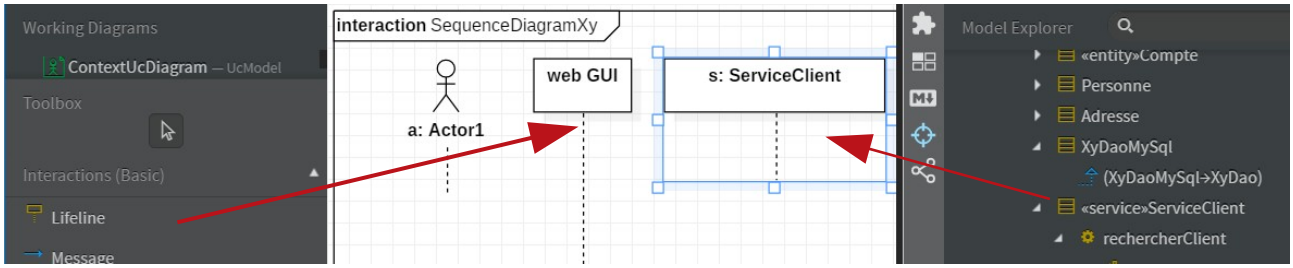


renommer "interaction" et "sequenceDiagram" peut être un "plus" .

Très souvent, l'acteur (en première position horizontale) , peut être placé via un "drag &drop" :



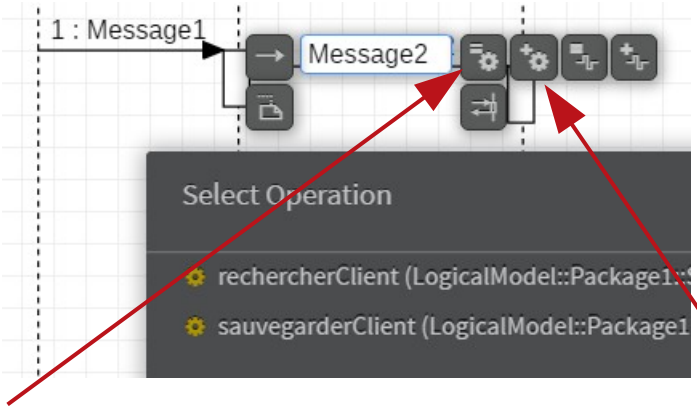
Conseil : renommer les "Lifeline..." via des alias très court (ex : "~~lifeline1~~ : Actor1" => "**a** : Actor1") pour gagner de la place .



Lorsqu'un élément du diagramme de séquence ne correspond à aucune des classes déjà modélisées , on peut utiliser l'icône "**Lifeline**" de la palette (exemple pour "GUI ou IHM")

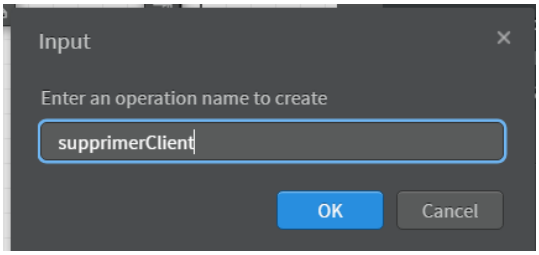
Lorsqu'un élément du diagramme de séquence correspond à une instance d'une des classes déjà modélisées , on peut utiliser **drag &drop** depuis l'arborescence du modèle (exemple pour "ServiceClient")

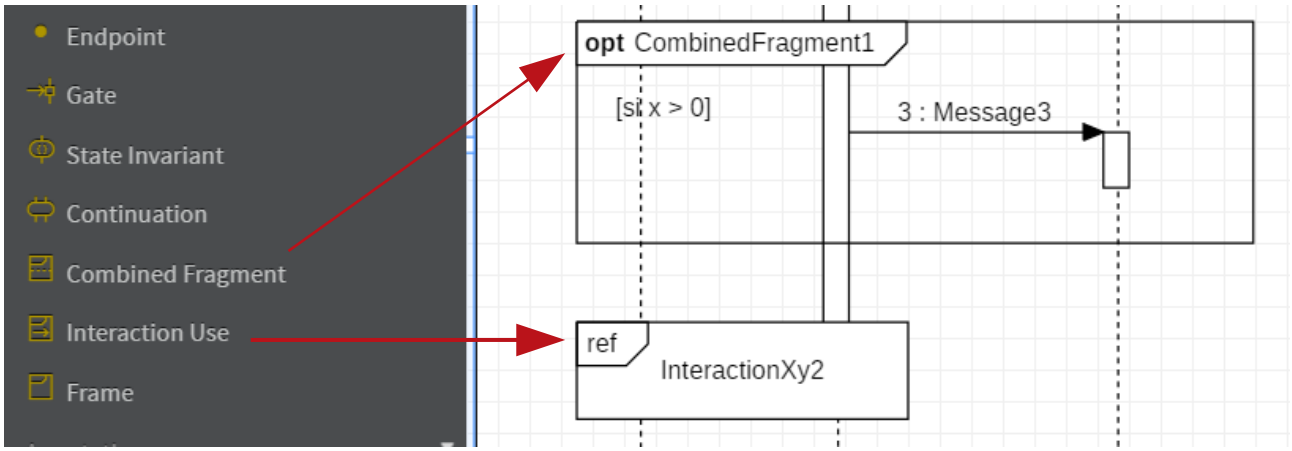
Une fois placés (en haut) les différents éléments devant communiquer entre eux, on aura souvent besoin d'**étirer (en hauteur) les lignes de vie (en pointillés)** .



select operation si *nom de message = opération existante de l'objet recevant le message*

add operation pour créer en même temps une nouvelle operation sur la classe de l'objet recevant le message .





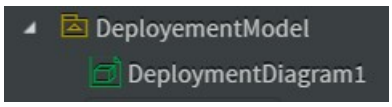
ref (pour faire référence à une (sous-) interaction , par exemple associée à un (sous-)U.C. en <<include>>)

opt (comme "*optional*") avec condition de ***guard***e .

NB : utiliser les fragments combinés avec parcimonie car 2 grands défauts :

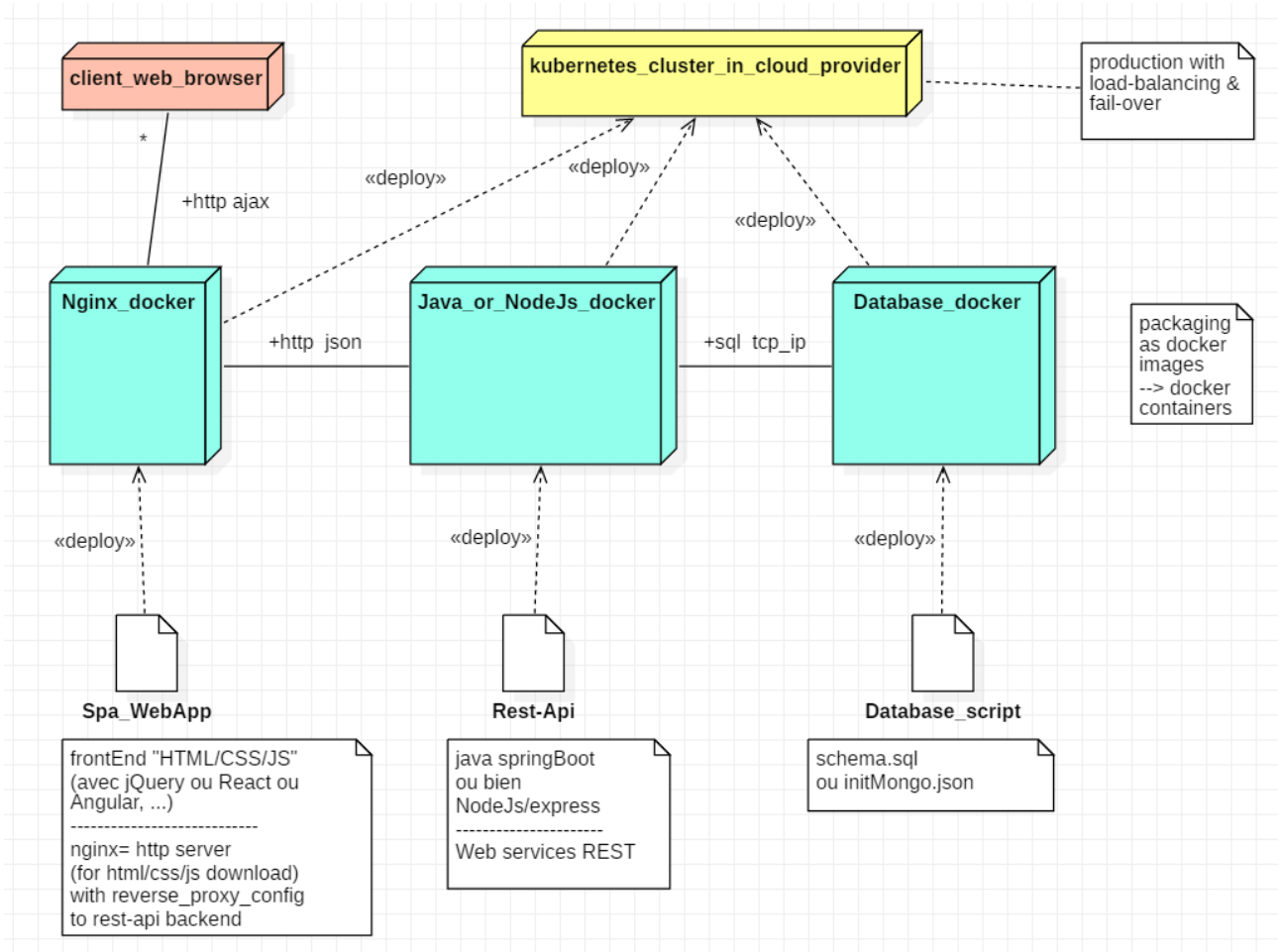
- ça surcharge énormément le diagramme (qui devient plus difficile à lire et interpréter)
- très compliqué à modifier par la suite avec l'outil UML (éléments superposés difficiles à sélectionner pour effectuer des changements) .

1.12. Edition d'un diagramme de déploiement



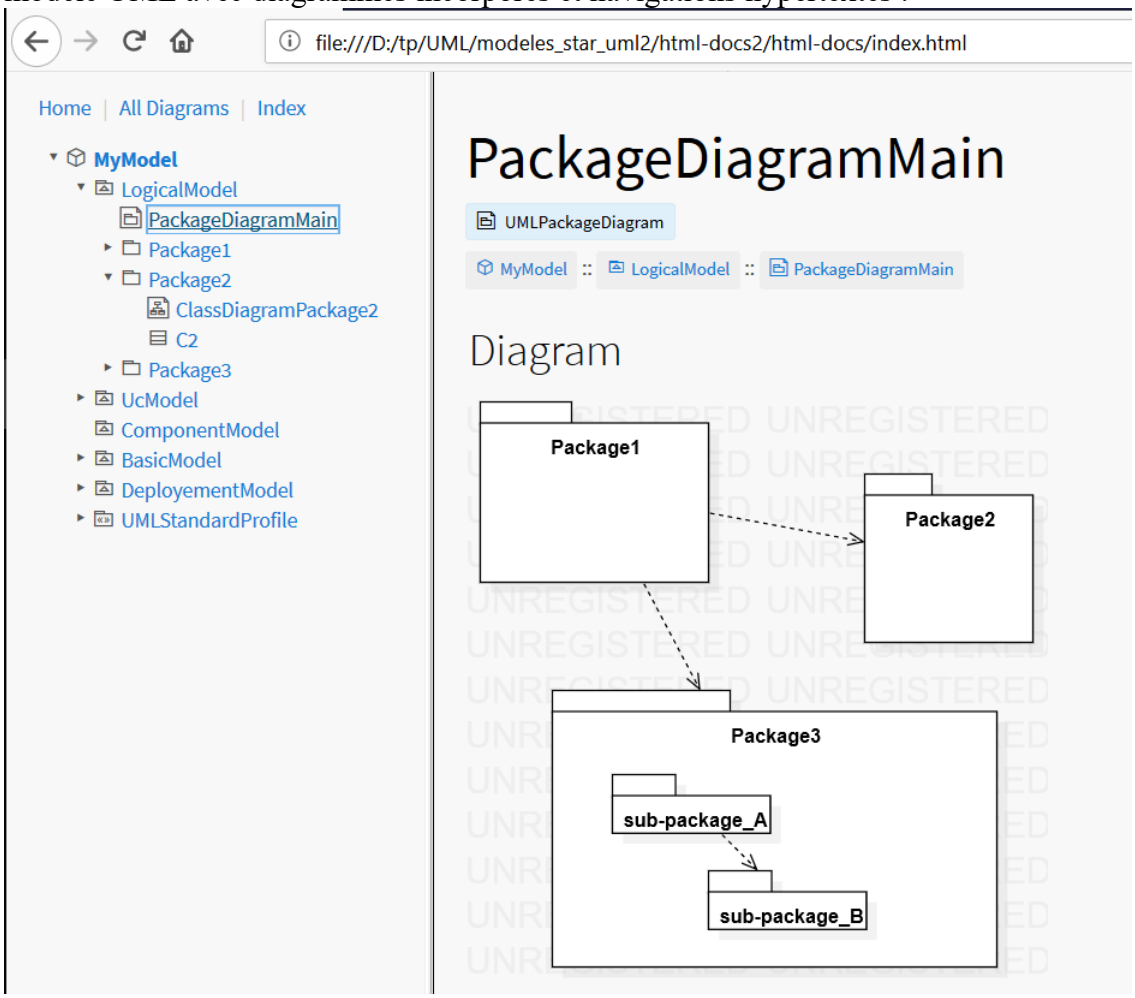
Contenu du diagramme : au cas par cas (selon architecture et technologies)

Utiliser essentiellement : ***"Node" , "Artifact" , "Communication path" , ...***

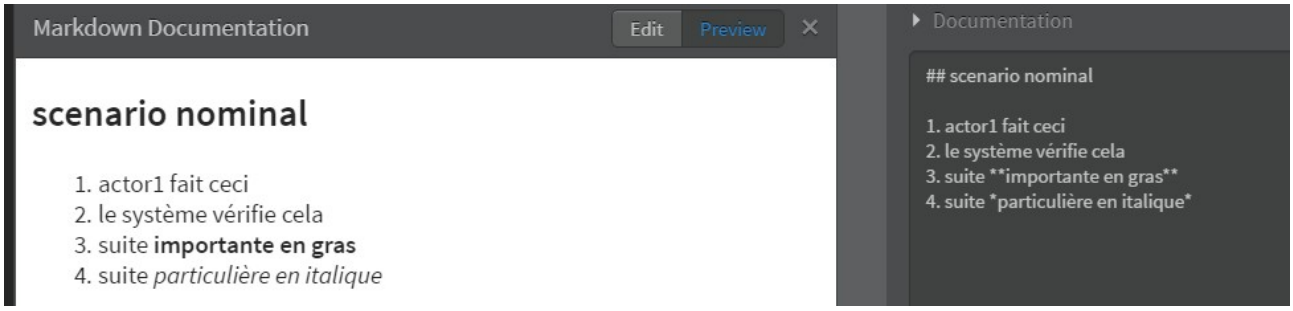


1.13. Exportation d'un modèle UML entier sous forme de site HTML facilement consultable à distance

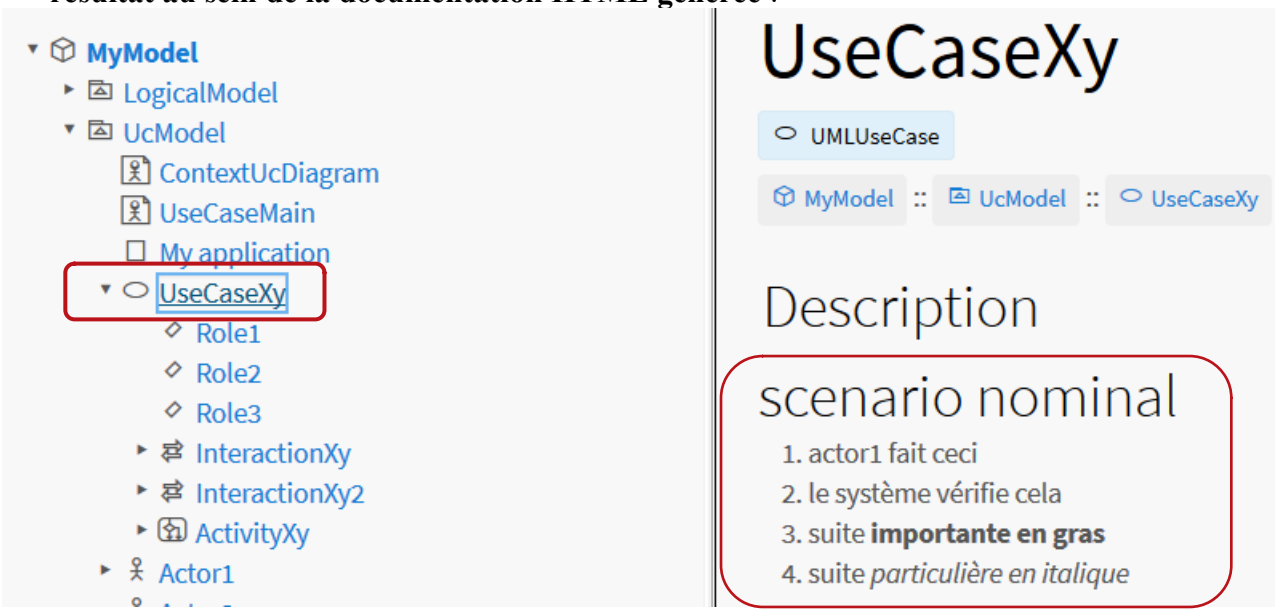
le menu **file / export ... / HTML Doc** permet de générer automatiquement un paquet de pages HTML *(à publier sur un serveur HTTP)* correspondant à une bonne description arborescente du modèle UML avec diagrammes incorporés et navigations hypertextes .



Point important : la documentation associée à un cas d'utilisation pourra (au sein de l'outil "star uml 3") est encodé en texte spécial au format "**markdown**" *(partiellement supporté)*
(ex : préfixe ## pour titre de niveau 2 , encadrement par * * pour italique , encadrement par ** ** pour gras, ..., double saut de ligne pour nouveau paragraphe , ...)



--> **résultat au sein de la documentation HTML générée** :



XXIV - Annexe – Bibliographie, Liens WEB , outils

1. Bibliographie et liens vers sites "internet"

Site de référence (OMG)	http://www.uml.org/
autre site de référence sur diagrammes UML	https://www.uml-diagrams.org/
guide sur l'outils "Visual paradigm" avec plein de bons exemples sur les syntaxes UML	https://www.visual-paradigm.com/guide/
UML en français (site web didactique) – UML 1	http://uml.free.fr/
Cours UML en ligne (syntaxe UML2)	http://laurent-audibert.developpez.com/Cours-UML/
http://www.eyrolles.com/Informatique/	Pour voir les livres qui existent sur UML