

Modélisation métier avec UML

Table des matières

I - Présentation du cours (objectifs).....	8
II - Cadre UML.....	9
1. Méthodologie, formalisme ,	9
2. Historique rapide d'UML.....	11
3. UML pour illustrer les spécifications.....	12
4. Formalisme UML = langage commun.....	12
5. UML universel mais avant tout orienté objet.....	13
6. Standard UML et extensions (profiles).....	13
7. Présentation des diagrammes UML.....	14
8. Descriptions sommaires des diagrammes UML.....	15
8.1. Diagramme des cas d'utilisations (Uses Cases).....	15
8.2. Diagramme de classes.....	16
8.3. Diagramme de séquence.....	17
8.4. Diagramme de collaboration (UML1) / communication (UML2).....	17

8.5. Diagramme d'états (StateChart).....	18
8.6. Diagramme d'activités.....	19
8.7. Diagramme de composants.....	20
8.8. Diagramme de déploiement.....	20
8.9. Diagrammes secondaires (variantes , ...).....	20
9. Utilisations courantes des diagrammes UML.....	21

III - Démarche , études de cas ,22

1. Activités de modélisation et spécifications.....	22
1.1. Vue imagée (modélisation, démarche progressive).....	23
2. Enchaînement des activités de modélisation.....	23
3. Cycle "itératif & incrémental".....	24
4. Etudes de cas (pour business modeling).....	25
4.1. Etude de cas "simulation emprunt (appli. informatique)"	25
4.2. Seconde étude de cas orientée "business modeling" – Site de ventes en lignes	25
4.3. Etude de cas "Agence de voyage" (étude de cas alternative).....	26

IV - Cas d'utilisation et scénarios (1er aperçu).....27

1. Etude des principales fonctionnalités (U.C.).....	27
1.1. Méthodologie de base (au niveau des cas d'utilisation).....	27
2. Diagramme des cas d'utilisations.....	28
3. Scénarios et descriptions détaillés (U.C.).....	32

V - Concepts objets et modularité.....35

1. Concepts objets fondamentaux.....	35
2. Granularité.....	41
3. Modularité.....	42
3.1. Forte cohésion et couplage faible.....	42

VI - Aperçu sur architectures n-tiers.....44

1. Structure globale d'une application.....	44
---	----

VII - Responsabilité , CRC , patterns GRASP.....46

1. Notion de responsabilité (contexte orienté objet).....	46
2. Idée des cartes "CRC"	46
3. Affectation des responsabilités (GRASP).....	47
4. Les 4 patterns GRASP fondamentaux.....	49
4.1. Expert.....	49
4.2. Faible couplage.....	50

4.3. Forte cohésion.....	51
4.4. Création.....	52

VIII - Analyse du domaine (entités) / diag. classes.....53

1. Analyse du domaine (glossaire , entités).....	53
2. Diagramme de classes (notations , ...).....	55
3. Eléments structurants d'UML.....	66
3.1. Modèle.....	66
3.2. Packages.....	66
3.3. Diagrammes.....	67
3.4. Stéréotypes.....	68
3.5. Valeurs étiquetées (Tag Values).....	69
3.6. Contraintes.....	69

IX - Modélisation des services métiers.....70

1. Analyse applicative (objectif et mise en oeuvre).....	70
2. Responsabilités (n-tiers) et services métiers.....	71
3. Repère méthodologique (rappel).....	72
4. Réalisation des cas d'utilisations.....	73
5. Diagramme de séquences (UML).....	74

X - Business Modeling (cadre, spécificités).....76

1. Rôle et contenu de la modélisation métier.....	76
2. Modélisation du contexte d'utilisation.....	78
3. Intérêts de la modélisation métier.....	79
3.1. Réfléchir sur le coeur de métier et l'organisationnel.....	79
3.2. Guider les restructurations/évolutions	79
3.3. Constituer un référentiel métier et organisationnel	79
3.4. Éclairage sur le contexte métier d'un système informatique.....	79
4. Petite méthodologie pour "modélisation métier".....	79

XI - Contextes ,périmètres et portée.....80

1. Diagramme de collaboration système(s).....	81
2. Diagramme de contexte.....	81

XII - Business Uses Cases.....82

1. Diagramme des cas d'utilisations métiers.....	82
2. Intérêts des "Business Uses Cases" au sein de la modélisation métier.....	82

XIII - Processus métier (diagramme d'activités).....83

1. Diagramme d'activités.....	83
1.1. couloirs d'activités.....	83
1.2. Noeuds spéciaux et de contrôles.....	84
1.3. nœuds d'activités et variantes (actions, ...).....	85
1.4. Notations pour actions particulières (UML2).....	85
1.5. object flow (nœuds "objet") : UML 1 et 2.....	86
1.6. Pins et buffers (UML 2).....	87
1.7. Final flow.....	88
2. Distinction "micro activité / macro activité" selon granularité.....	88
3. Quelques conseils sur les diagrammes d'activités.....	89

XIV - Graphe sémantique et secteurs métiers.....90

1. Modularité recherchée (modules métiers).....	90
2. Proxy métier/fonctionnel.....	91
3. Diagrammes de classes (de niveau métier).....	92

XV - Diagrammes d'états.....93

1. Diagramme d'états et de transitions (StateChart).....	93
1.1. Point de jonction.....	94
1.2. Etat composite (super état).....	95
1.3. Etats historisés.....	95
2. Dualité "diagramme d'états / diagramme d'activités"	96

XVI - Expr. Besoins IHM (diag d'états , ...).....98

1. Expression des besoins "IHM"	98
1.1. Maquette.....	98
1.2. Modélisation des enchaînements d'écrans (navigations).....	98
1.3. Modélisation composite et générique des écrans (modèles).....	99
1.4. Modélisation abstraite des écrans (contenu / fonctions).....	99

XVII - Besoins techniques / diag. déploiement.....100

1.1. Eventuelle formulation du contexte d'intégration technologique via un diagramme de déploiement UML.....	100
1.2. Besoins techniques à exprimer.....	100
1.3. Eléments de la conception qui découleront des besoins techniques exprimés.....	100

XVIII - Aperçu général sur la conception.....101

1. Rôles de la conception.....	101
2. Chacun sa spécialité (analyse, architecte, ...).....	101
3. Activités liées à la conception.....	102
3.1. conception générique et architecture technique.....	102

3.2. Conception préliminaire.....	102
3.3. conceptions détaillées.....	102
3.4. implémentation & tests.....	102
3.5. tests d'intégration.....	103
4. Conception avec MDA:.....	103

XIX - Implémentation , retours tests , itérations.....104

1.1. Tests , retours , critiques , itérations.....	104
1.2. Annexe intéressante (pour approfondir).....	104

XX - Infrastructure UML (MDA , outils , ...).....106

1. Infrastructure pour UML, MDA,	106
1.1. Challenge.....	106
1.2. Principales normes de l'OMG.....	106
2. XMI.....	107
2.1. Combinaisons éventuelles d'outils complémentaires.....	107
2.2. Limitations de xmi et des imports/exports.....	107
2.3. Socle technique apporté par eclipse (ou équivalent).....	108
3. MDA.....	109
3.1. Modèles , Méta-Modèles et Méta-Méta-Modèles.....	110
3.2. Quelques produits "MDA" concrets:.....	111

XXI - OCL (Object Constraint Language).....112

1. OCL (Object Constraint Language) - Présentation.....	112
---	-----

XXII - Notations UML diverses et avancées.....113

1. Diagramme de séquences (notations avancées).....	113
2. Diag. de classes et interfaces	117
3. Autres notations avancées (UML2).....	118
3.1. Diagramme de structures composites.....	118
3.2. Notion de "Port" (alias "point d'accès").....	118

XXIII - Différences entre UML et Merise + O.R.M.....119

1. Préliminaire: différences cardinalités/multiplicités.....	119
2. Cas de figures (down-top , top-down , ...).....	120
2.1. Top-down (modèle logique --> base de données).....	120
2.2. Down-top (base de données existante --> modèle logique).....	120
3. Correspondances essentielles "objet-relationnel"	120
3.1. Entité (UML) / Table simple.....	120
3.2. Association UML / Jointure entre tables.....	121
3.3. Composition UML / table "détails" avec clef primaire composée.....	122

4. Correspondances "objet-relationnel" avancées.....	123
4.1. Héritage	123
4.2. Classe d'association & Table de jointure avec attributs.....	124
4.3. Profil UML pour Données relationnelles.....	124
XXIV - BPMN (aperçu).....	125
1. BPMN (présentation).....	125
XXV - Orientation SOA,problématique,architecture.....	127
1. Modélisation SOA.....	127
2. Notions et concepts étroitement liés à SOA.....	127
2.1. Format de données pivots et gouvernance des données.....	127
2.2. Transactions longues et compensations.....	130
2.3. Stabilités des éléments référencés.....	130
3. Méthode Praxème.....	131
XXVI - Méthodologies (aperçu rapide).....	133
1. Processus unifiés (UP).....	133
1.1. Meilleures pratiques communes.....	133
2. Principaux Processus.....	134
2.1. RUP (Rational UP – origine d'UP).....	134
2.2. 2TUP (2 tracks UP) – Pocessus en Y.....	134
2.3. XP (eXPerience & eXtreme Programing).....	135
3. Présentation de RUP.....	136
4. Approche itérative.....	136
5. Vocabulaire.....	137
6. 4 grandes phases.....	138
6.1. Phase de conceptualisation / commencement (inception).....	139
6.2. Phase d'élaboration.....	140
6.3. Phase de construction.....	140
6.4. Phase de transition.....	140
7. XP (Extreme Programming).....	141
8. Eventuelle planification des livrables selon XP.....	142
XXVII - Norme UML et MétaModèle.....	143
1. Quelques éléments du méta-modèle UML2.....	143
XXVIII - Essentiel outil "Topcased UML".....	146
1. Utilisation de TopCased_UML.....	146
1.1. Intégration/installation de Topcased UML.....	147

1.2. Création d'un profil UML (avec stéréotypes).....	147
1.3. Création et initialisation d'un modèle UML (pour java).....	149
1.4. Utilisation générale de l'outil TopCased UML.....	150
1.5. Organisation conseillée des packages et des diagrammes.....	151
1.6. Edition d'un diagramme de classes (spécificités).....	152
1.7. Préférences/options sur l'éditeur topcased UML.....	153
1.8. Edition d'un diagramme de Use Cases (spécificités).....	154
1.9. Edition d'un diagramme d'activités (spécificités).....	154
1.10. Edition d'un diagramme de séquences (spécificités).....	156
1.11. Edition d'un diagramme d'états (spécificités).....	157
1.12. Edition d'un diagramme de composants (spécificités).....	158
1.13. Edition d'un diagramme de déploiement (spécificités).....	158
1.14. Glisser/poser d'un élément externe pour y faire référence.....	159
1.15. Décomposition d'un modèle UML en plusieurs sous fichiers.....	159
2. Génération de documentation (gendoc2).....	160
2.1. Principe de fonctionnement de gendoc2.....	160
2.2. Paramétrages généraux (configuration, contexte(s)).....	161
2.3. Généralités sur les scripts de gendoc2.....	161
2.4. Scripts avec images/diagrammes.....	163
2.5. Document maître pour fédérer plusieurs fichiers générés.....	163
XXIX - Annexe – Bibliographie, Liens WEB , outils.....	164
1. Bibliographie et liens vers sites "internet".....	164
2. Quelques outils UML (Editeurs , AGL).....	164

I - Présentation du cours (objectifs)

Le cours "Modélisation métier avec UML" (ou Business Modeling) est

plutôt orienté "maîtrise d'ouvrage" (moa)

et se focalise essentiellement sur les concepts objets et le sens apporté par la modélisation UML aux éléments fonctionnels du métier (*processus métiers , entités métiers, secteurs métiers , activités métiers*).

Une présentation des concepts objets permettra de bien comprendre les spécificités orientées objets d'UML.

Les aspects suivants seront développés dans les détails:

- aperçu rapide des architectures & technologies informatiques
- diagramme d'activités , graphe sémantique
- découpages en packages (secteurs métiers) et décomposition fonctionnelle.
- Uses Cases et "Business Uses Cases"
- diagramme de classes (domaine,)
- notion de responsabilité et patterns "GRASP"

L'analyse (entités du domaine , services métiers ,) sera développée tout en restant accessible / compréhensible .

Un *aperçu très rapide des activités de conception* permettra de se faire une idée de l'exploitation ultérieure des spécifications réalisées.

Déroulement:

Après une présentation générale du cadre UML (formalisme , méthodologie , infrastructure , ...), la formation sera construite autour de l'accompagnement théorique et pratique nécessaire à la mise en oeuvre d'une ou deux petite(s) étude(s) 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 utiles à la modélisation métier 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.

Les deux phases de la formation:

- Phase 1 : apprentissage des fondamentaux (Concepts objets , principaux diagrammes) via une étude de cas classique (application informatique) .
- Phase 2 : spécificités de la modélisation métier et approfondissement des thèmes associés (processus métiers, diagramme d'activités) sur une 2ème étude de cas.

II - Cadre UML

1. Méthodologie, formalisme ,

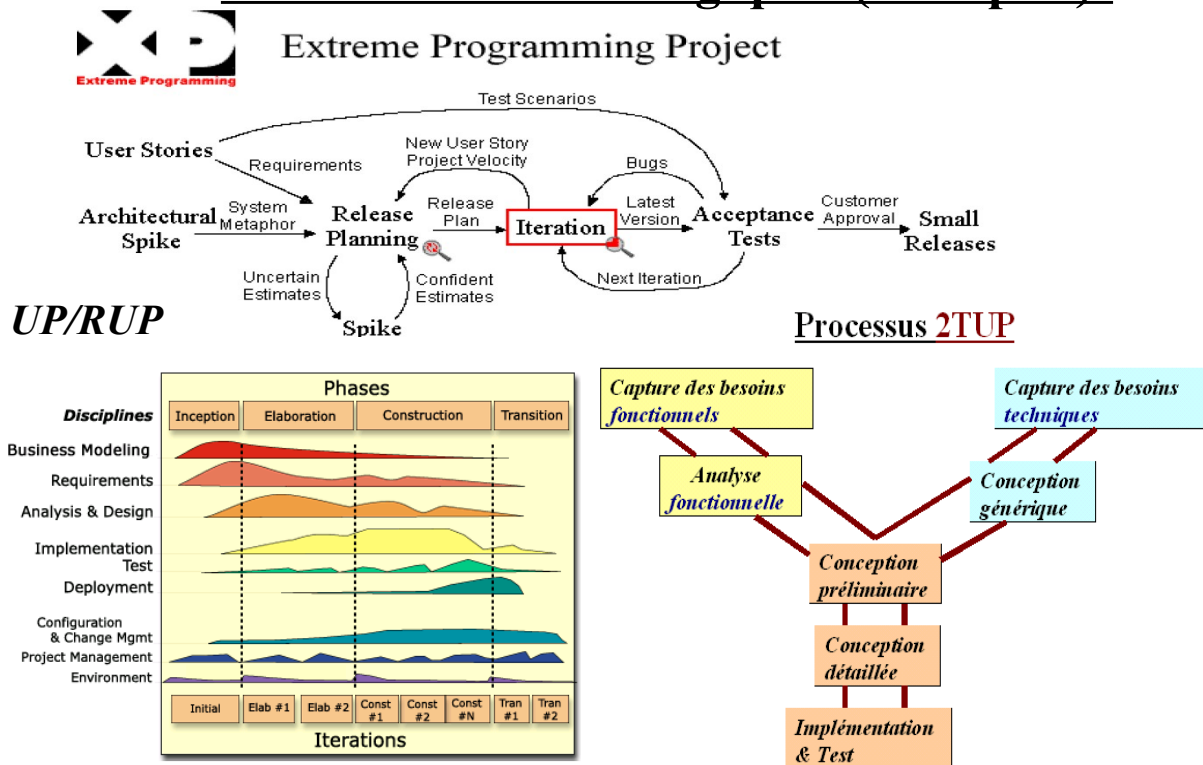
Méthodologie = Formalisme + Processus

- **UML** est un formalisme
(Notations standardisées
[diagrammes] avec sémantiques précises)
- Un **processus** (démarche méthodologique) doit être utilisé conjointement (ex: UP , XP , ...).



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

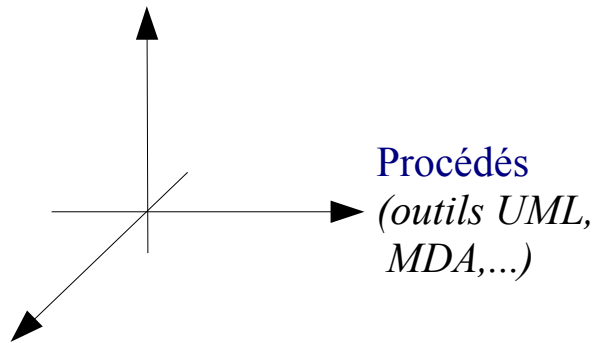
Processus méthodologiques (exemples):



Plein de variantes dans les façons d'utiliser UML

Formalisme

(notations, diagrammes)



Procédés

(outils UML, MDA,...)

Méthode/démarche

(activités de modélisation, spécifications,...)

* UML pour
simple ébauche
ou bien
modèle précis ?

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

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

Pourquoi ?

(Quels objectifs ?
Quelles utilités ?)



Modélisation métier

(business modeling)

+ **expression des besoins**

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

Quoi ?

(Quelles entités ?
Quelles structures ?
Quels services ?)



Analyse

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

Comment ?

(avec quelles
Technologies ?)

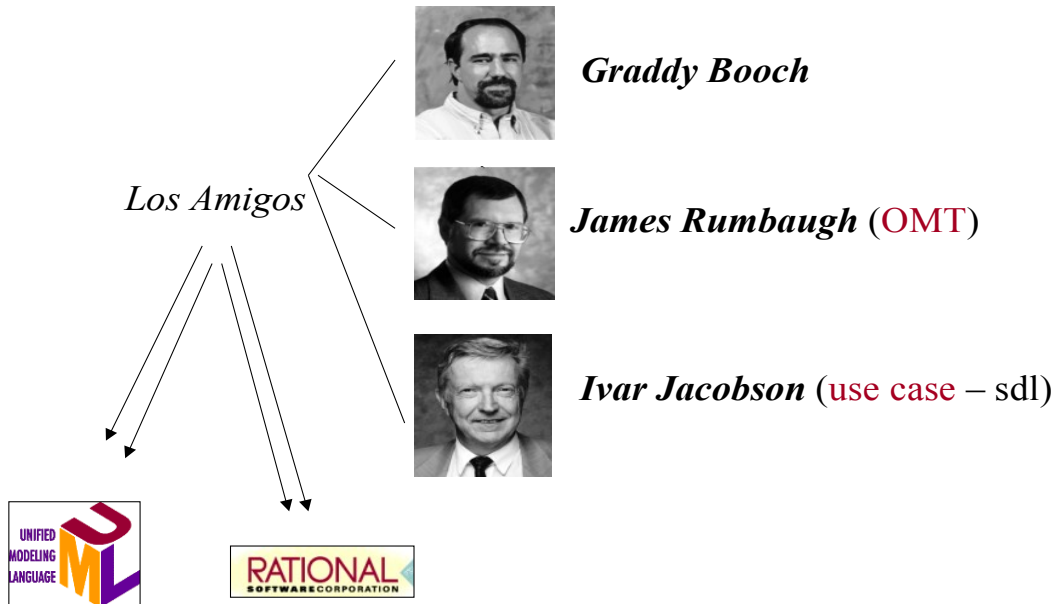


Conception

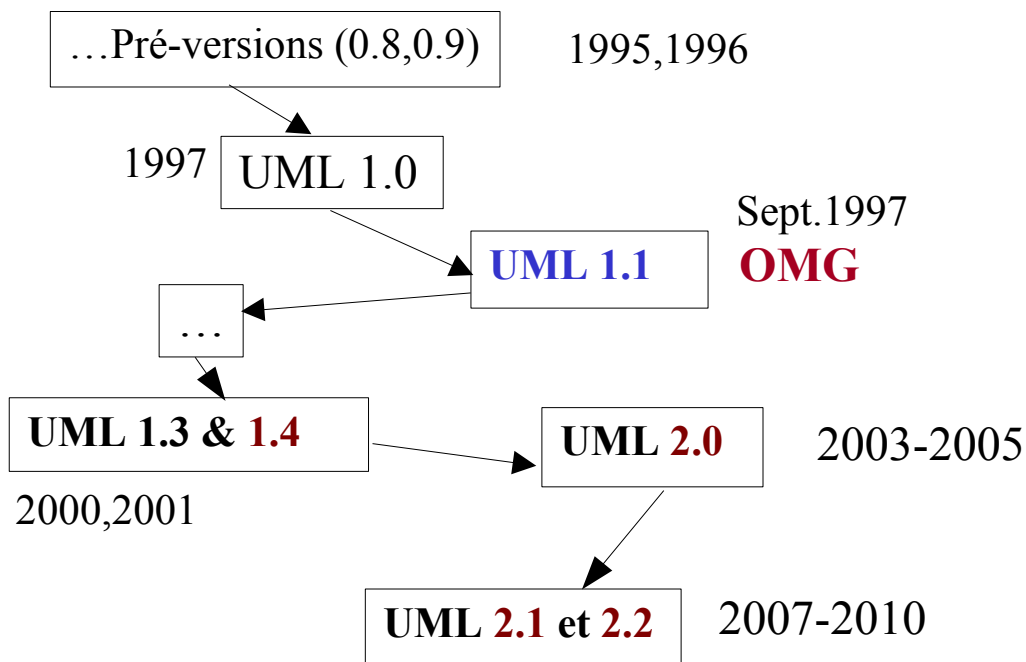
(P.S.M. : Platform Specific Model)

2. Historique rapide d'UML

Les fondateurs d'UML



Normalisation d'UML (standard de l'OMG)



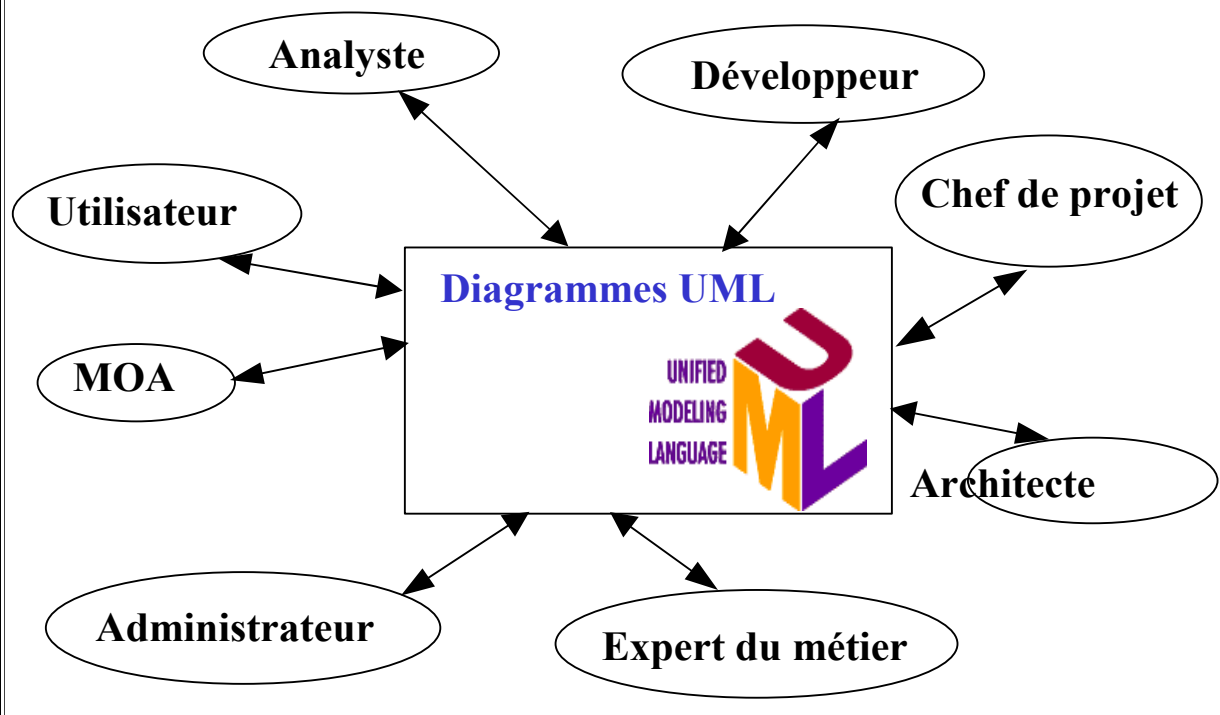
3. UML pour illustrer les spécifications

Principales utilités d'UML

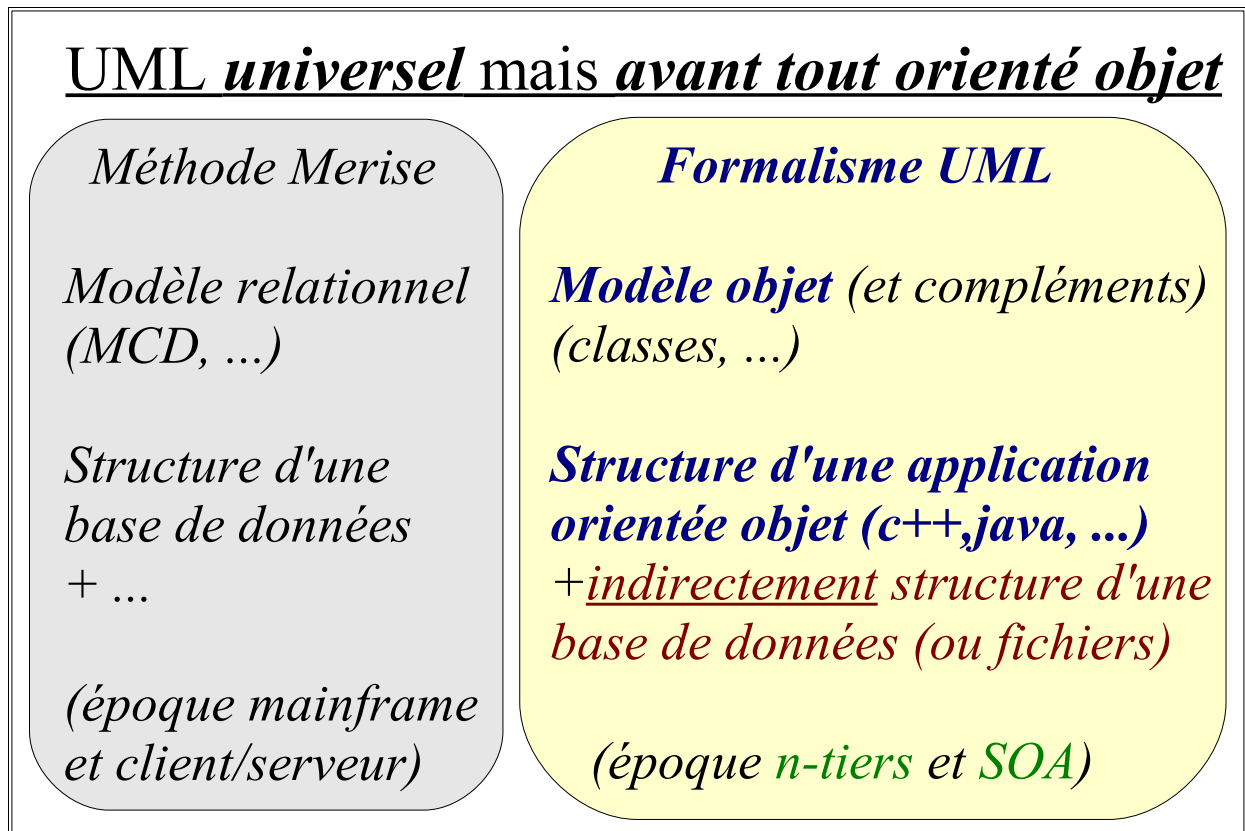
- ♦ Diagrammes UML = partie importante des **Spécifications** fonctionnelles et techniques.
- ♦ **Cogiter** sur le "pourquoi/quoi/comment" en se basant sur l'**essentiel** (qui ressort de la **modélisation** abstraite UML).
- ♦ Eventuel point d'entrée d'une **génération partielle de code** (via MDA ou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

4. Formalisme UML = langage commun

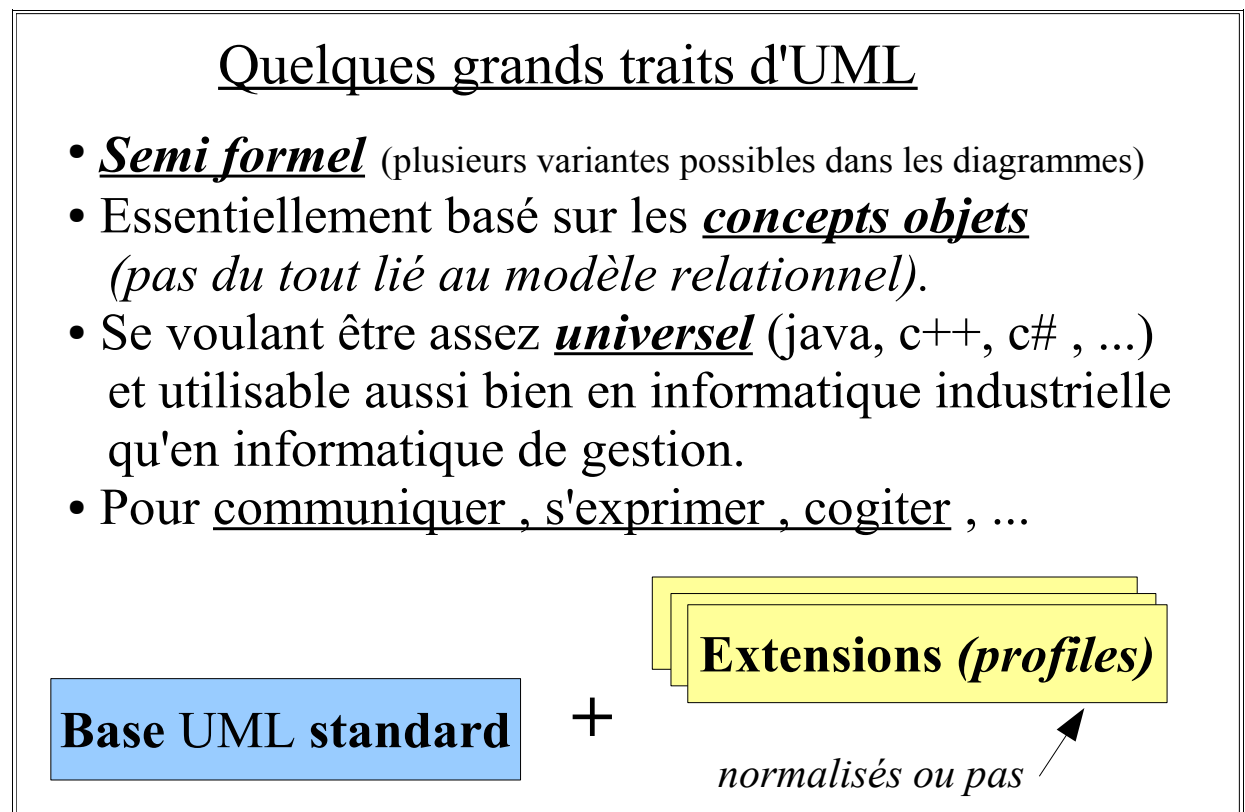
Différents points de vue / langage commun



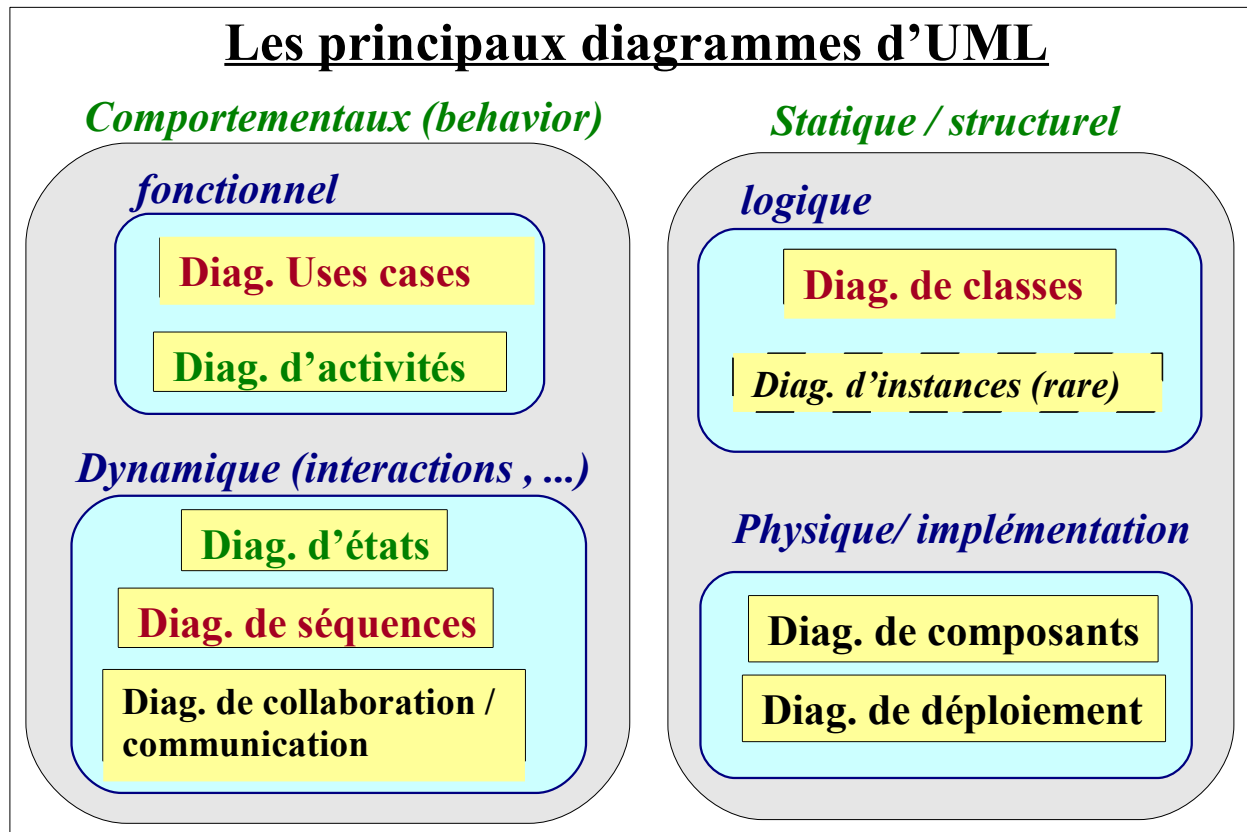
5. UML universel mais avant tout orienté objet



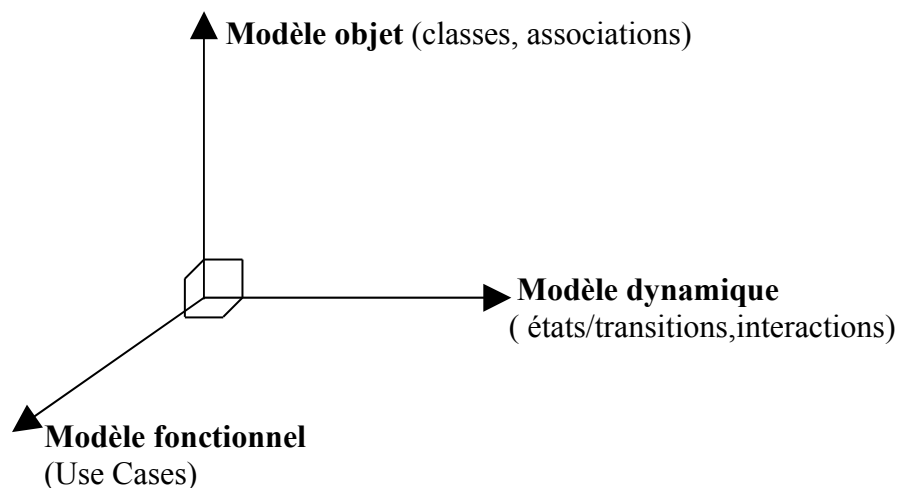
6. Standard UML et extensions (profiles)



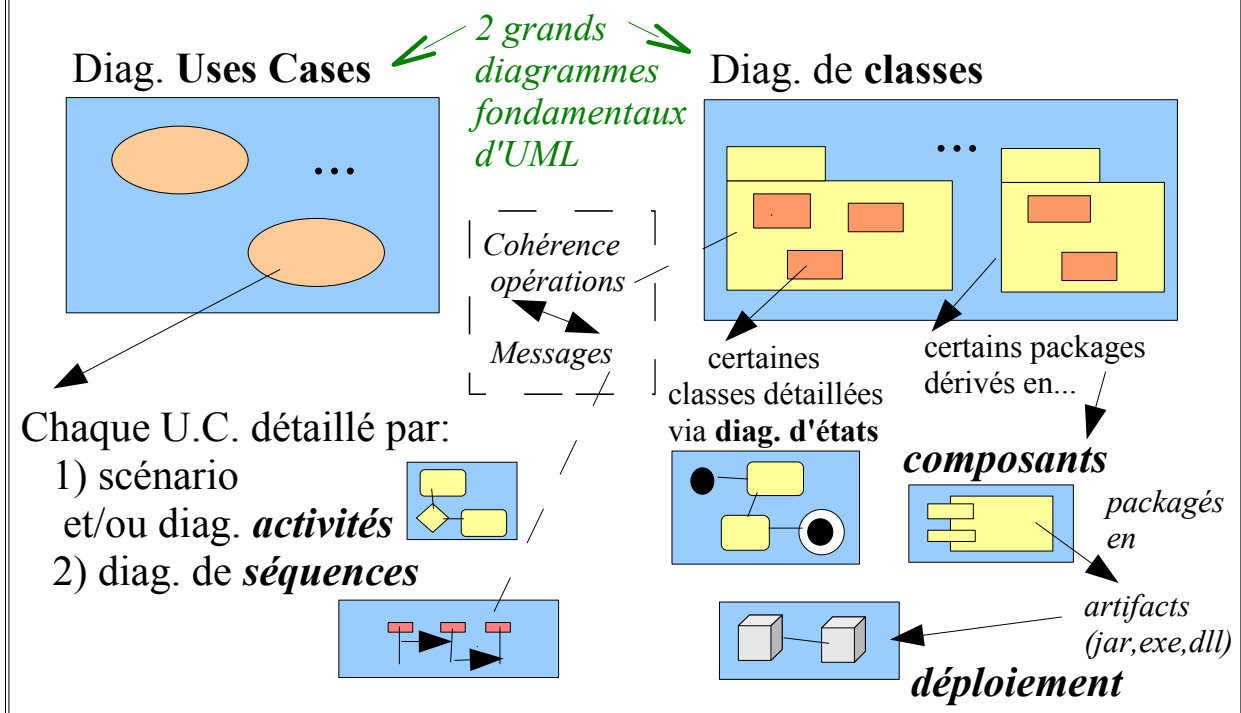
7. Présentation des diagrammes UML



Modèles (diagrammes) complémentaires:



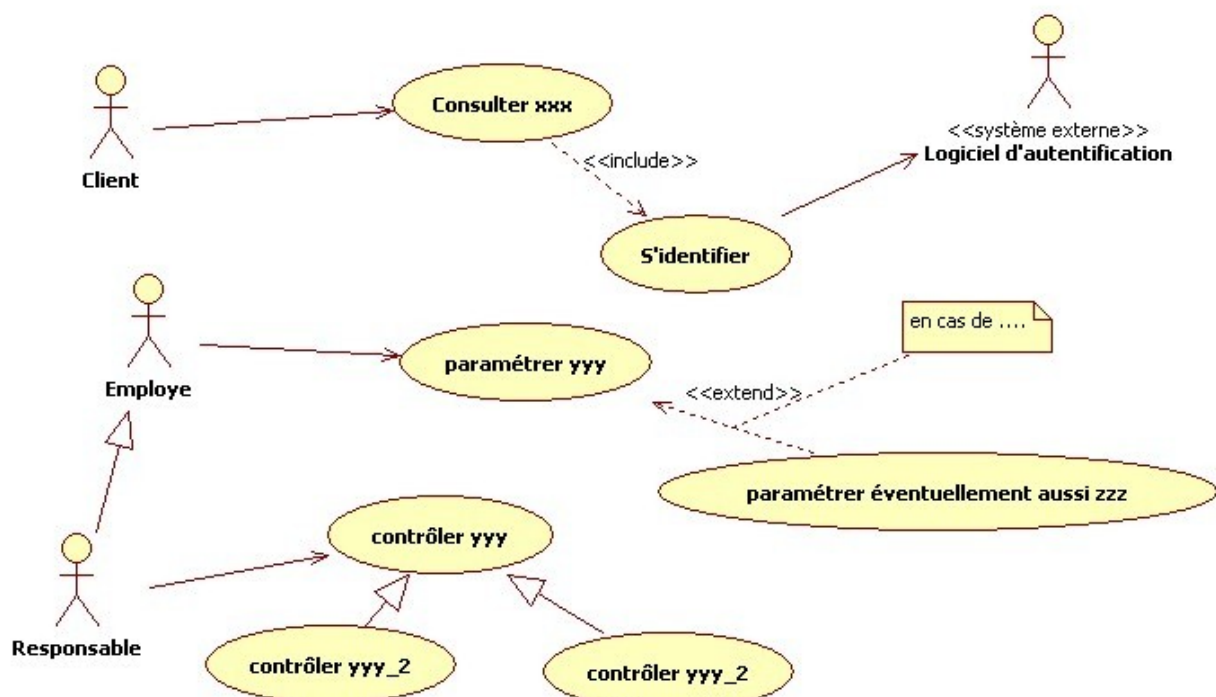
Principaux liens entre les diagrammes UML



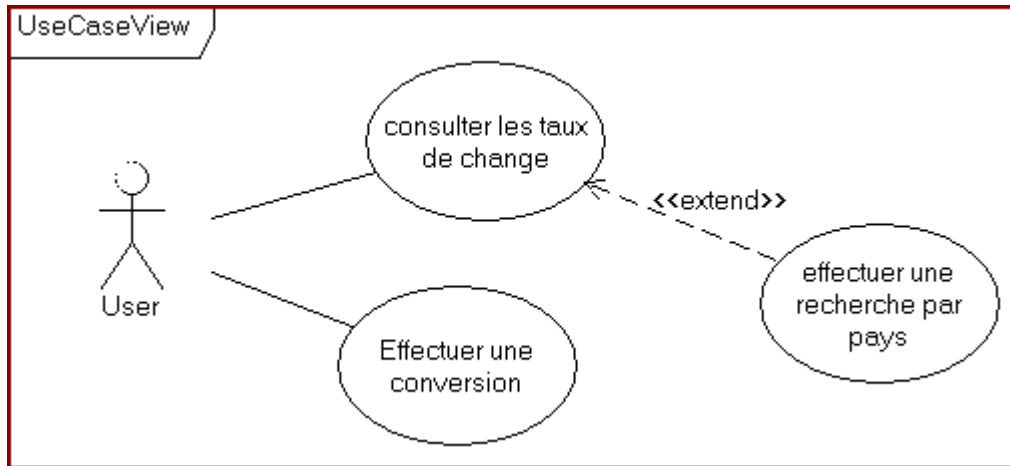
8. Descriptions sommaires des diagrammes UML

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

exemple:



sur micro étude cas "conversion de devises":



8.2. Diagramme de classes

exemples:

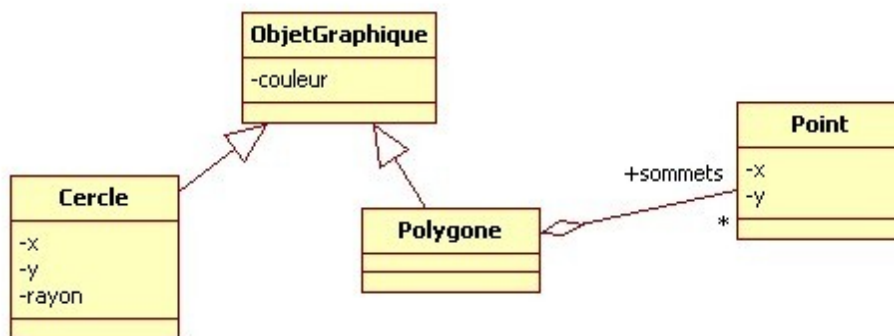
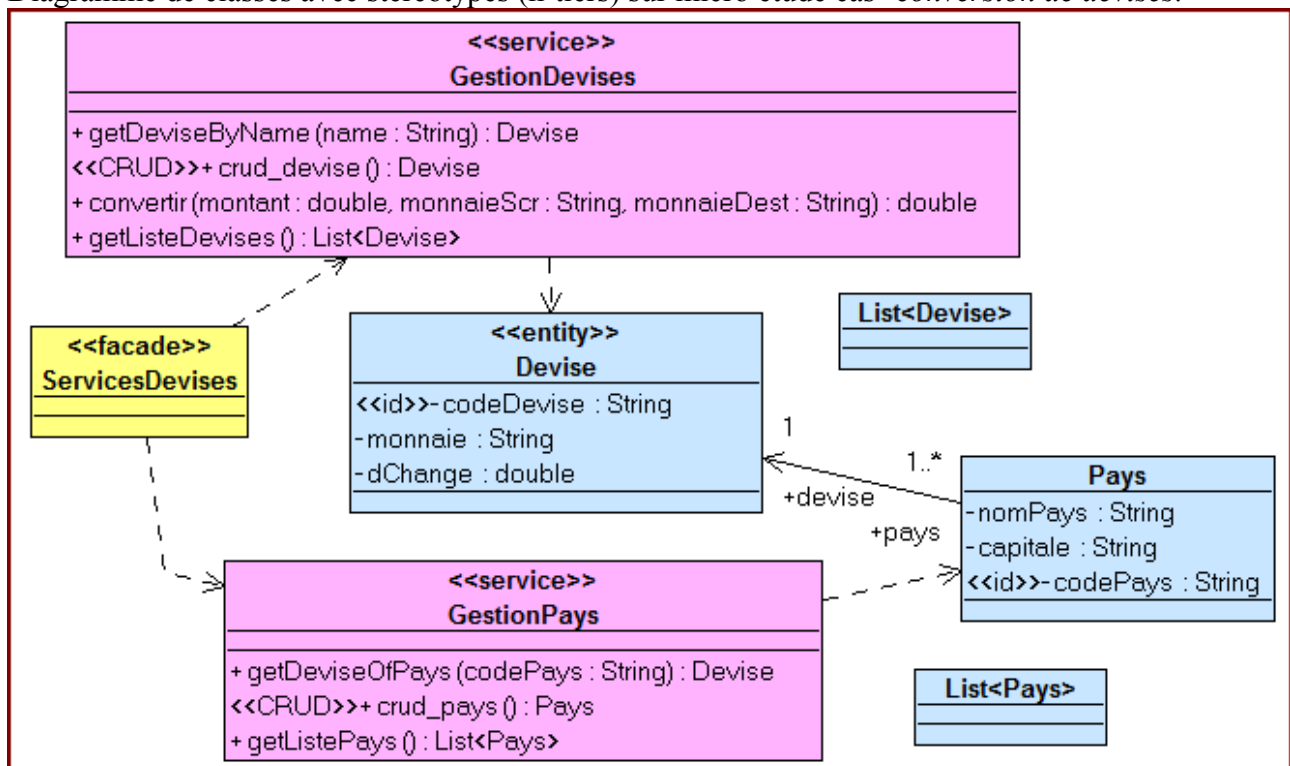
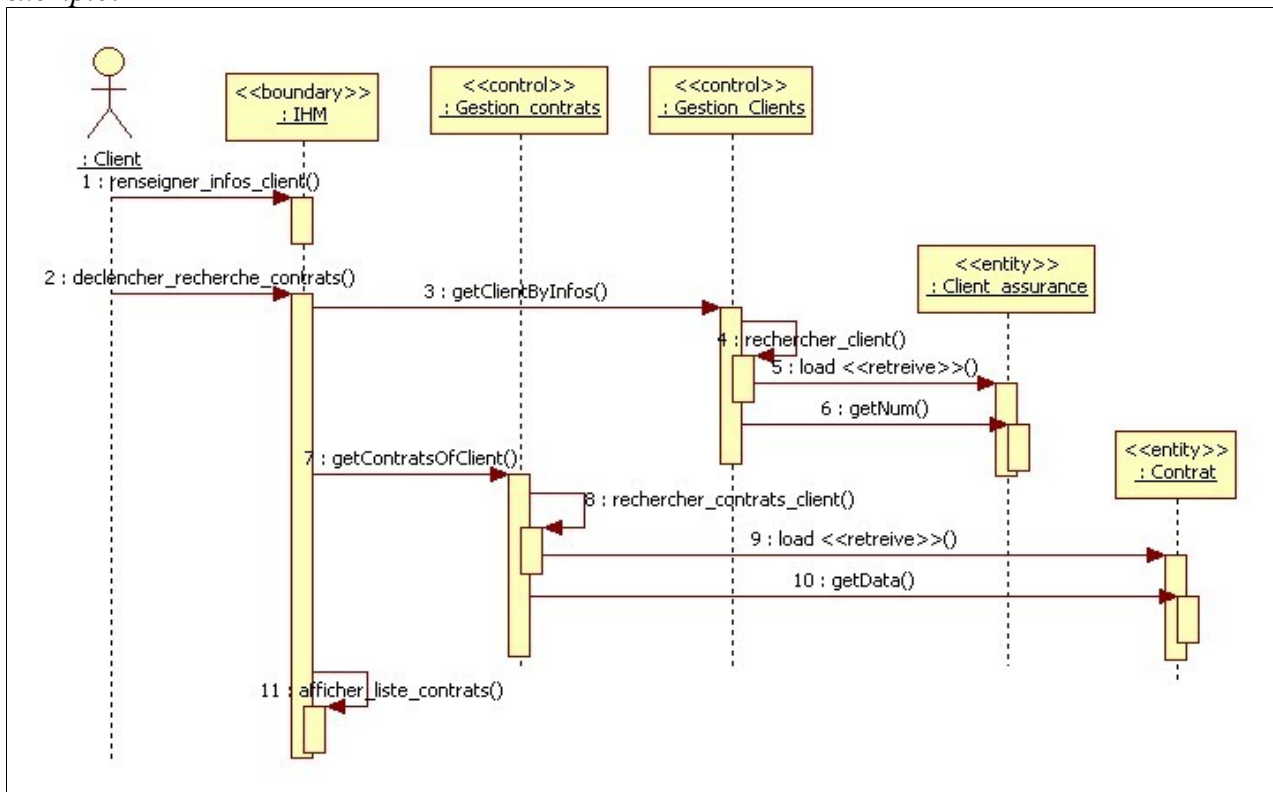


Diagramme de classes avec stéréotypes (n-tiers) sur micro étude cas "conversion de devises":



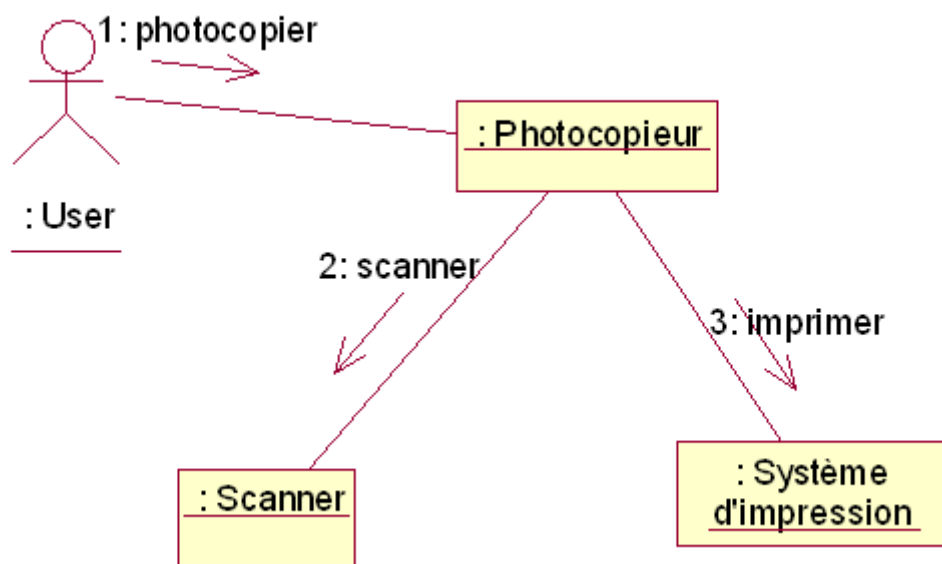
8.3. Diagramme de séquence

exemple:



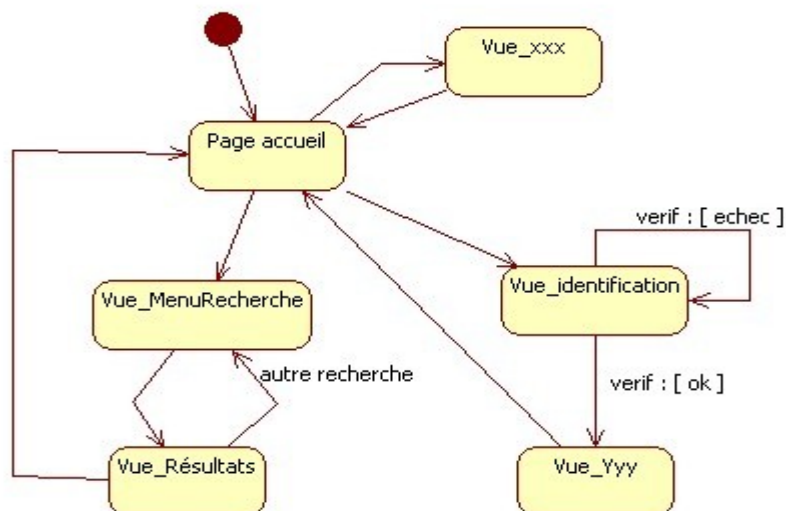
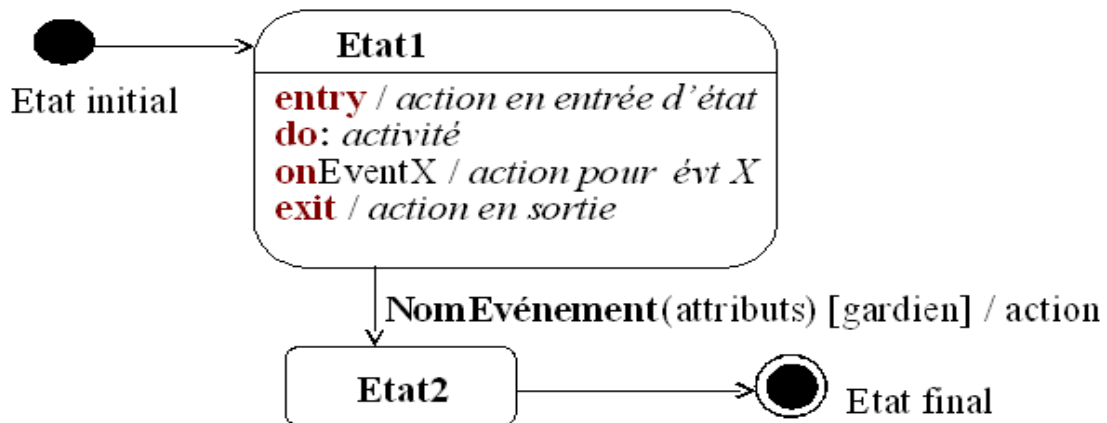
8.4. Diagramme de collaboration (UML1) / communication (UML2)

exemple:



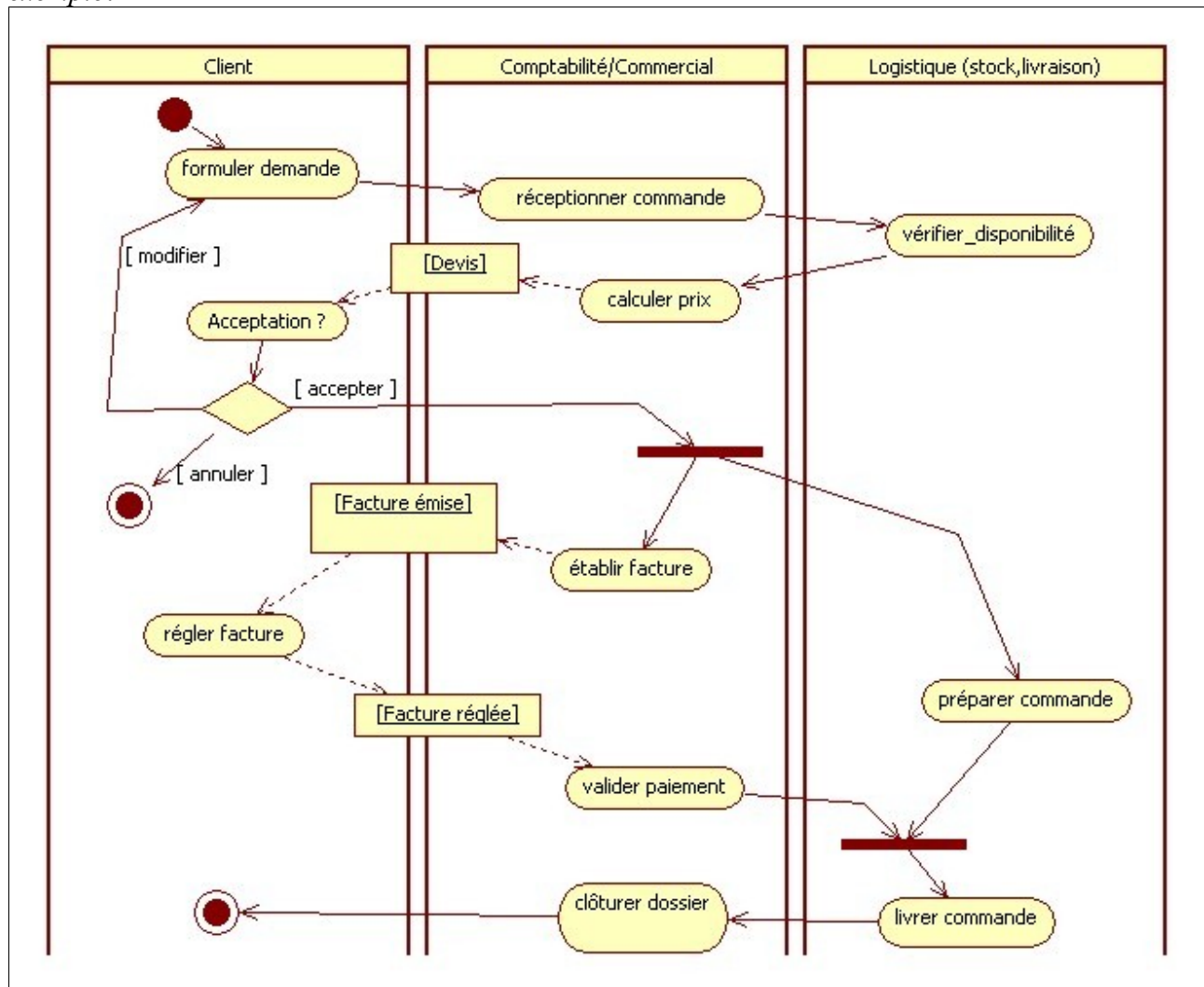
8.5. Diagramme d'états (StateChart)

exemples:



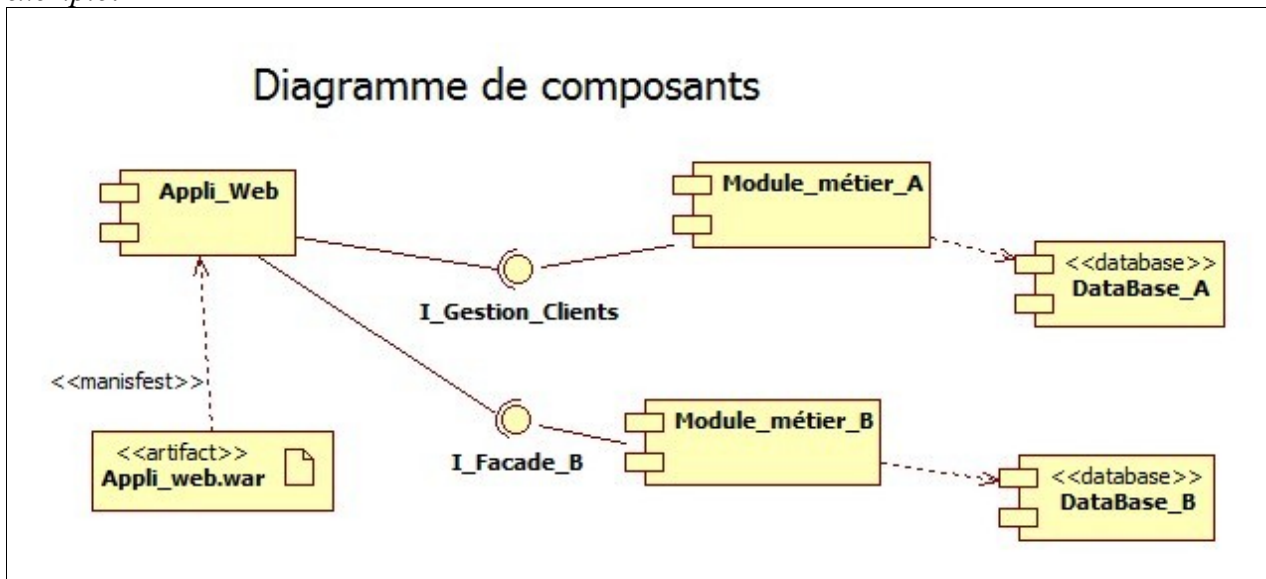
8.6. Diagramme d'activités

exemple:



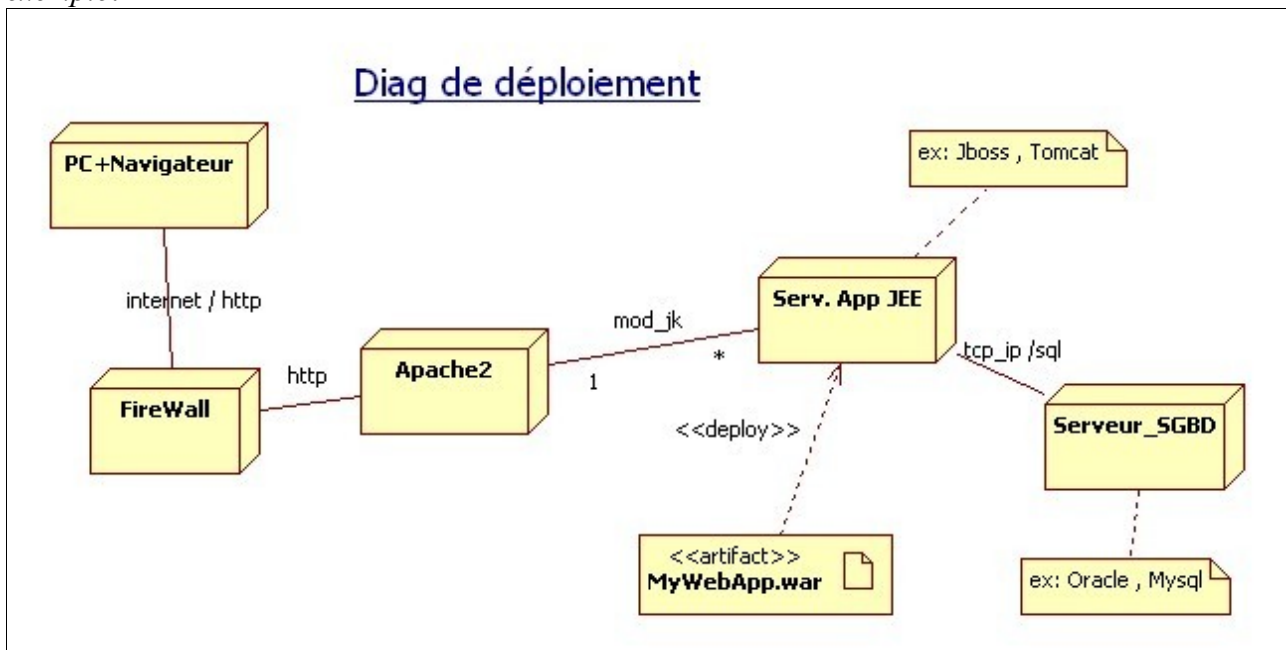
8.7. Diagramme de composants

exemple:



8.8. Diagramme de déploiement

exemple:



8.9. Diagrammes secondaires (variantes , ...)

Diagramme de robustesse = diagramme de classe avec stéréotype **<<boundary>>** , **<<control>>** , **<<entity>>**

Diagramme d'instance = un peu comme diagramme de collaboration mais pour montrer l'état (valeurs des attributs) de quelques instances .

....

9. Utilisations courantes des diagrammes UML

<i>Diagrammes UML</i>	<i>Utilisations courantes</i>
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 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 (<i>ordre de planification selon priorité des UC</i>) ==> guide pour les <i>jeux de tests</i>
Diag. de séquences	Montrer comment divers objets de l'application communiquent entre eux sous la forme d'une séquence d'envois de messages (<i>interactions</i>)
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 évènements).
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 (<i>machine , réseau , serveurs ,</i>) de l'environnement cible (recette , production) afin de spécifier les détails du déploiement.

III - Démarche , études de cas , ...

1. Activités de modélisation et spécifications

Activités de Modélisation - principales disciplines:

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

1) Pourquoi?

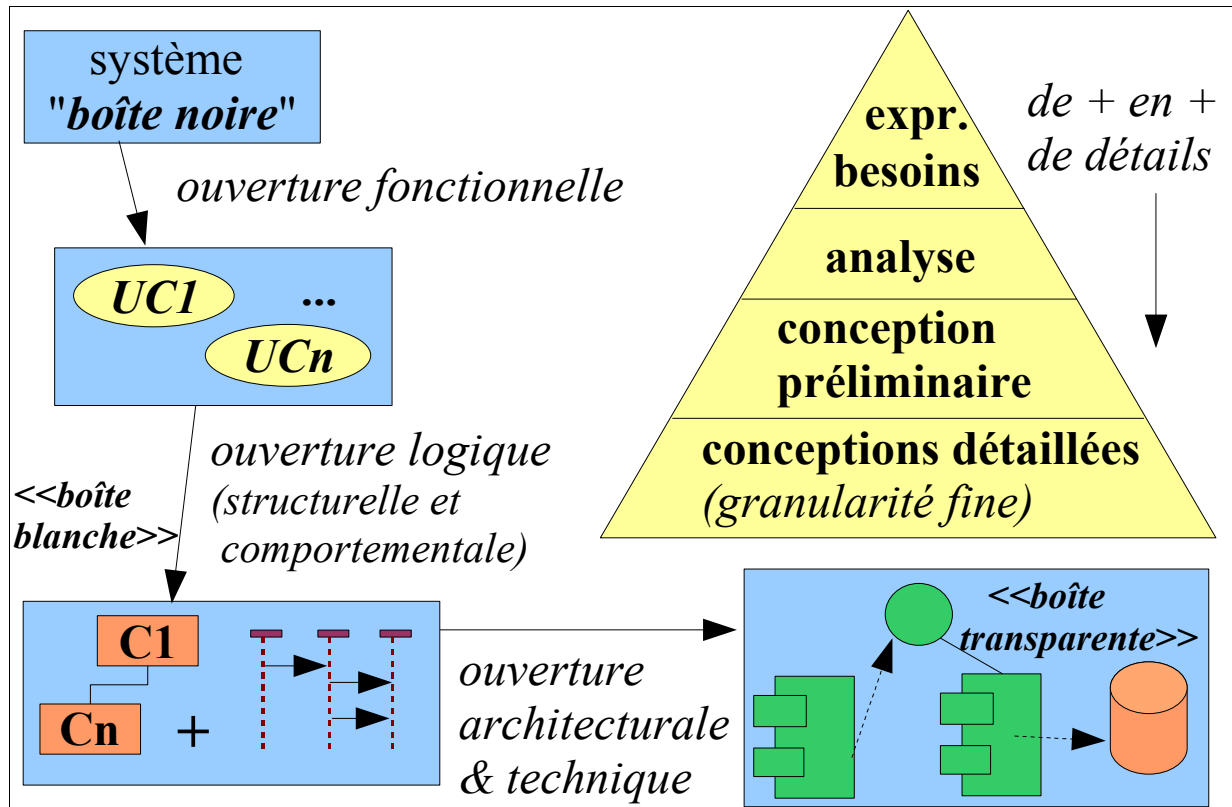
2) Quoi?

3) Comment?

Fruits de la modélisation - principales spécifications:

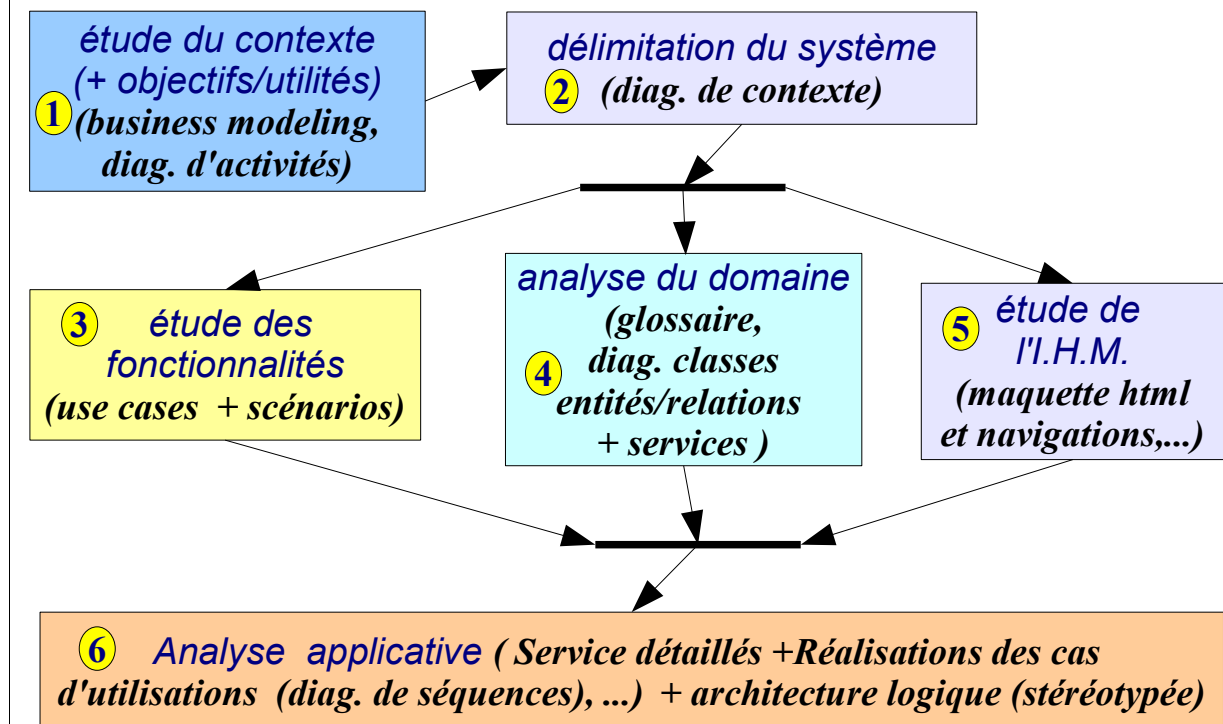
- **Spécifications "métier" et contextuelles**
(périmètre applicatif , contexte métier et organisationnel)
- **Spécifications fonctionnelles générales**
(entités du domaine+ fonctionnalités (U.C.) + IHM
+ services métiers)
- **Spécifications fonctionnelles détaillées**
(+ réalisations des U.C. , + architecture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

1.1. Vue imagée (modélisation, démarche progressive)

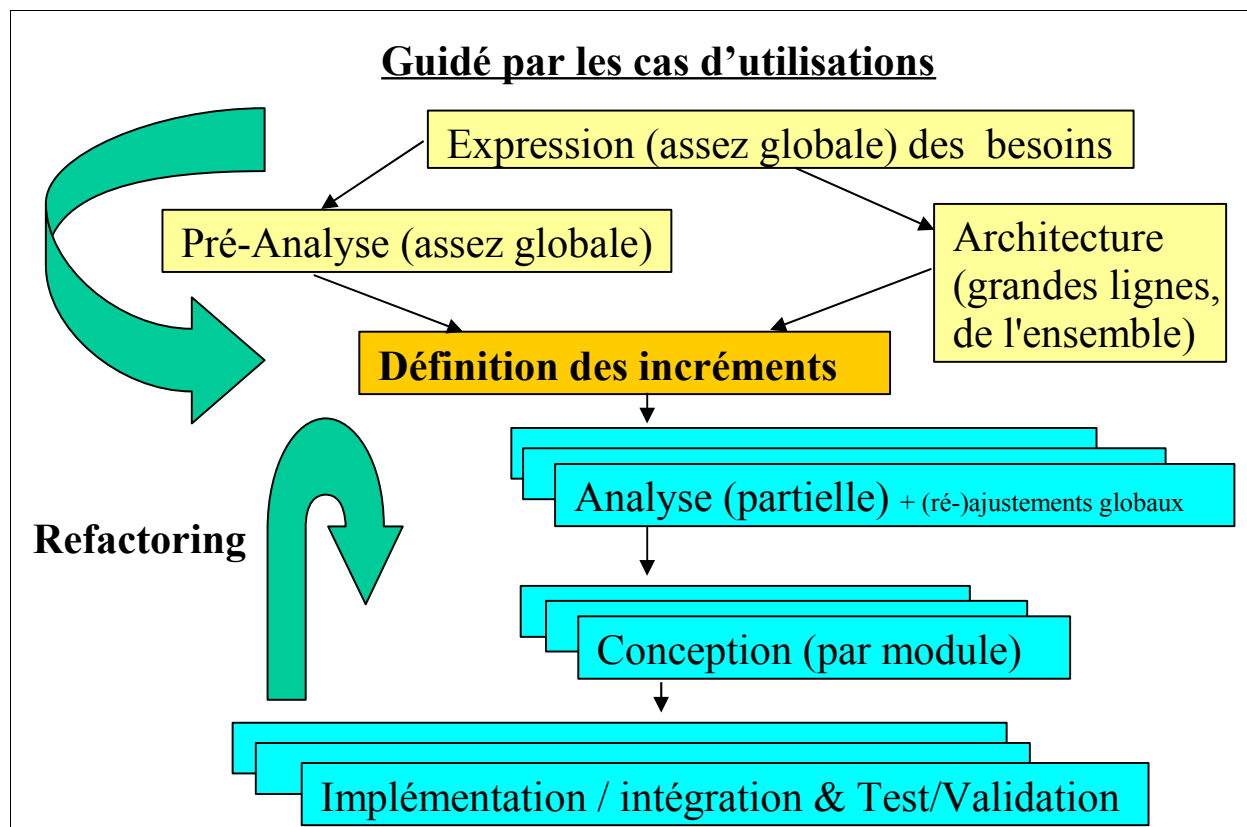
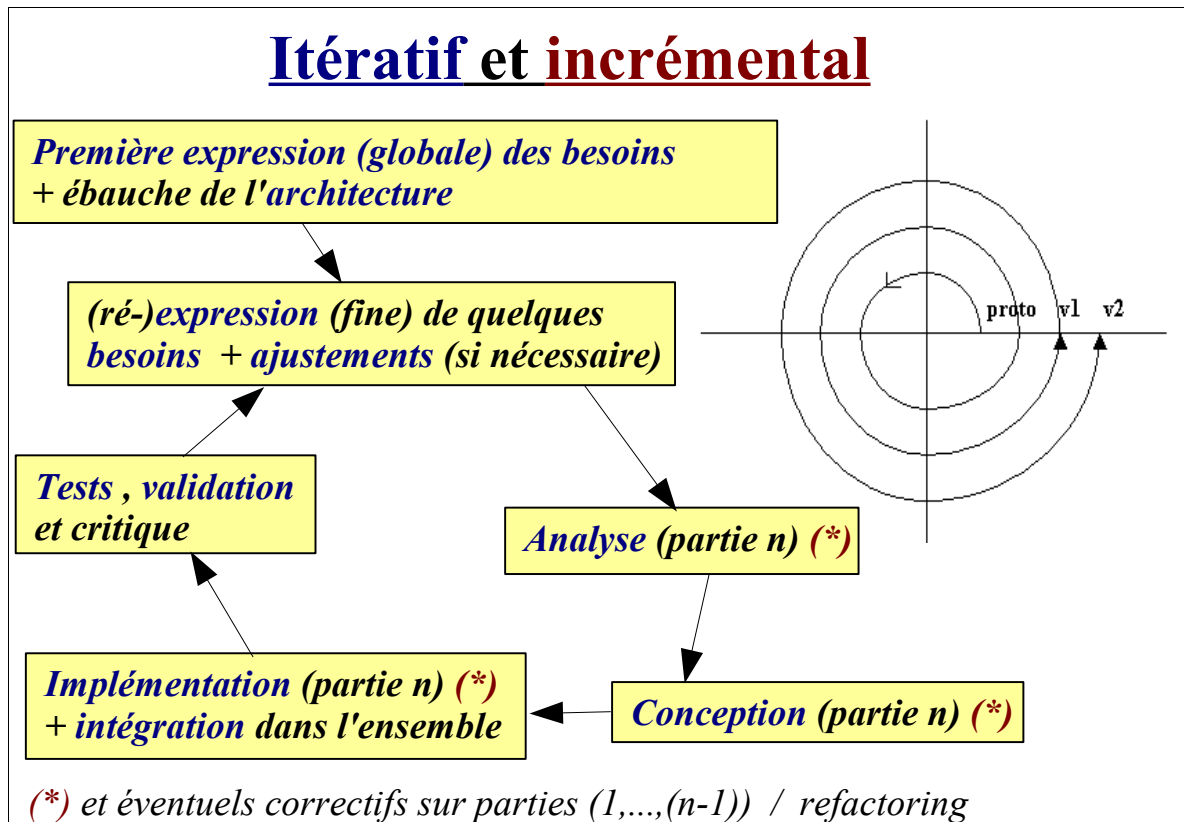


2. Enchaînement des activités de modélisation

Enchaînement classique d'activités sur la partie fonctionnelle



3. Cycle "itératif & incrémental"



4. Etudes de cas (pour business modeling)

4.1. Etude de cas "simulation emprunt (appli. informatique)"

Modéliser en UML une application informatique (au niveau d'une banque) qui permet d'effectuer des simulations d'emprunts depuis un navigateur internet.

==> éléments en entrée (à fournir par le client) :

- * montant (à emprunter)
- * durée (nb_mois pour le remboursement)

L'application sera décomposée en services:

- * Service "SimulateurEmprunt" (orchestrant les sous services suivants)
- * Service "GestionTauxFrais" avec :
 - * getFraisDossier(selon montant_emprunte) avec calcul par plages .
 - * getTauxCourant(selon_durée) avec appel du même type de service au niveau "BCE" (externe) + majoration de 20% .
- * Service "CalculsFinanciers" avec entre autre une méthode
"calculerMensualite(montant_emprunté, nb_mois, taux_pct) "

La réponse du "SimulateurEmprunt" au client sera une "PropositionPret" (avec "taux" , "frais_dossier" et "mensualité" / plus rappels des "montant" et "durée").

4.2. Seconde étude de cas orientée "business modeling" – Site de ventes en lignes

Modélisation métier libre .

Sujet : Système Informatique (à grande échelle) d'un organisme spécialisé dans la **vente en ligne** (style "CSiscout" / "WebDistrib" / "PixMania" / "Amazon" ou autres).

--> éléments à prendre (un peu) en compte:

- gestion des stocks (produit commandé disponible ?)
- gestion des clients (adresses, ...)
- paiements, facturation
- expédition (packaging des colis , livraison , ...)

(Eventuel graphe sémantique / ébauche)

Elément central (tout tourne autour : **commande** de produits à livrer)

--> au moins un diagramme d'états et quelques "processus métiers / diag. d'activités".

Eléments externes:

- > Fournisseurs de produits (pour ré-approvisionner les stocks)
- Prestataire de transport pour les colis (ex: "La-Poste")

4.3. Etude de cas "Agence de voyage" (étude de cas alternative)

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 , ...)

Grands choix techniques et conseils:

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

Spécifications du cahier des charges:

Gestion du catalogue:

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

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

V3 --> gérer des offres promotionnelles

(basées sur une offre du catalogue , mais dates fixes et prix réduits).

Gestion des réservation:

Gestion du nombre de places

Depuis agence (traitements éventuels de formulaires "papier" envoyés par courrier).

Depuis internet (le paiement sécurisé sera délégué à la banque BqY).

Gestion de la logistique:

Annulation si trop peu d'inscrits 10 jours avant le départ.

Mission "guide de FH" ou sous-traitance des activités

Transport (étapes,...)

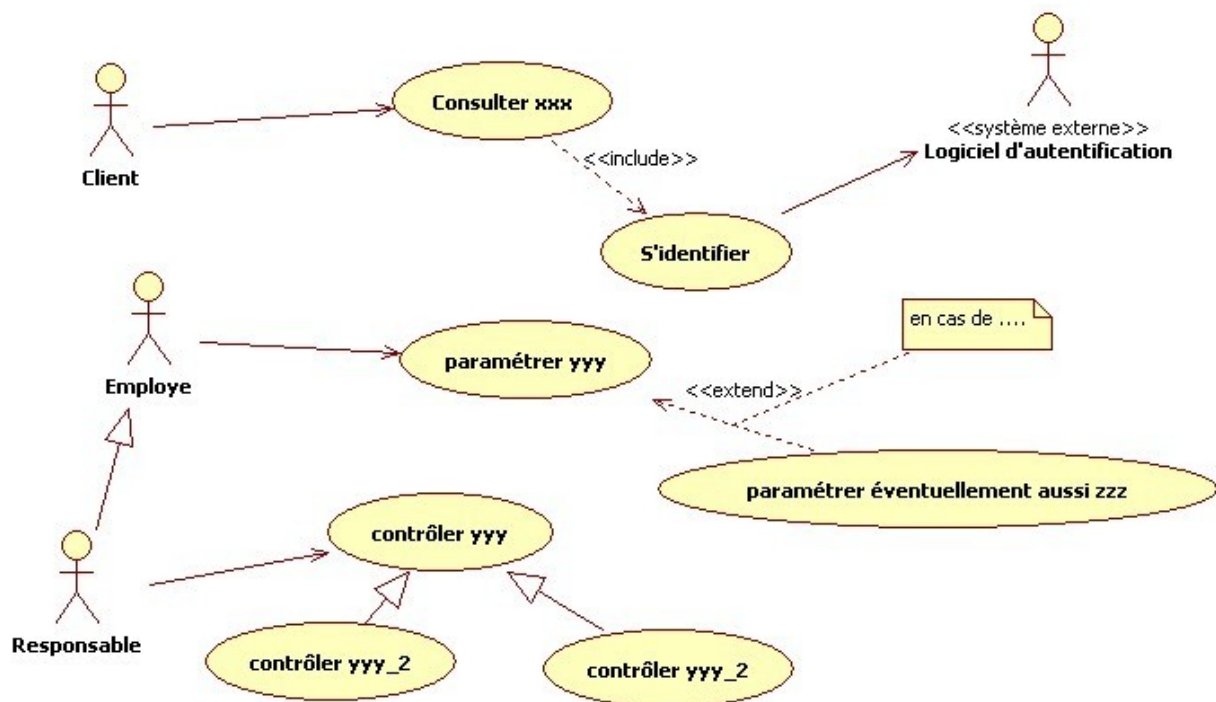
...

IV - Cas d'utilisation et scénarios (1^{er} aperçu)

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

La notion de "cas d'utilisation" correspond à la fois à:

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



NB:

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

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

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

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

2. Diagramme des cas d'utilisations

Présentation des « Use Case »

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

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

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

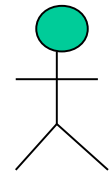
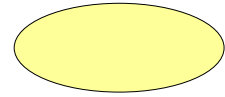
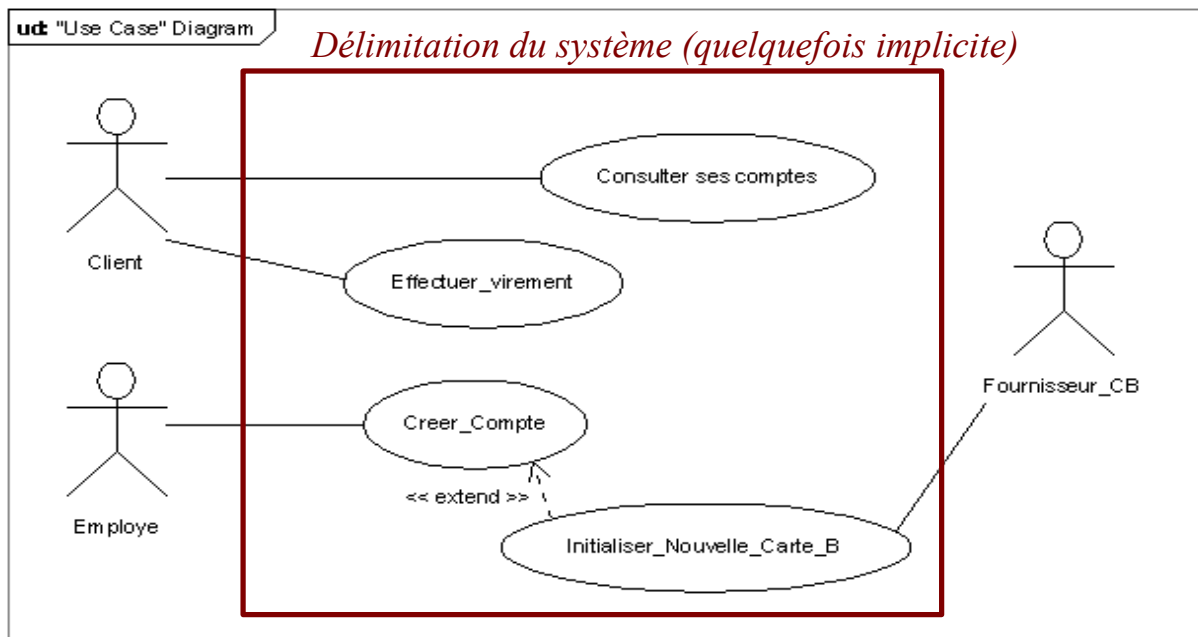
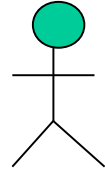


Diagramme U.C. (Vue d'ensemble)



Acteurs

- Un **acteur** est une **entité extérieure au système** qui **interagit d'une certaine façon avec le système** en **jouant un certain rôle**.
- Un acteur ne correspond pas forcément à une catégorie de personnes physiques .
Un 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).*

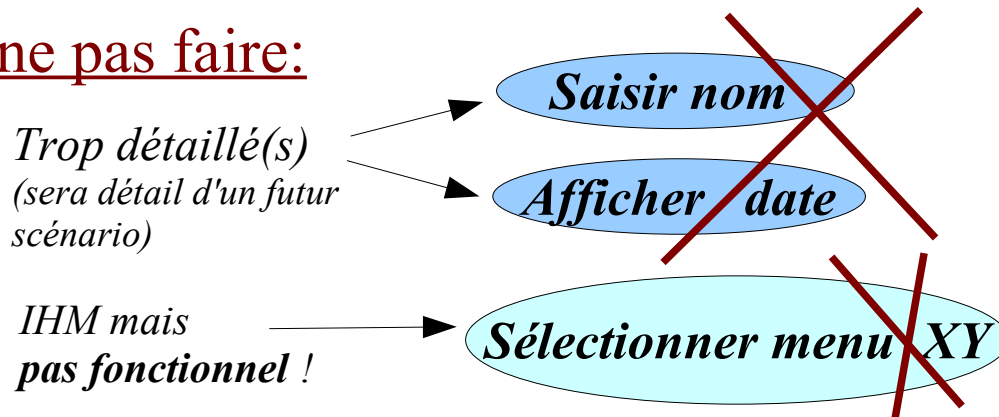
Notation:



A ne pas faire:

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

*IHM mais
pas fonctionnel !*

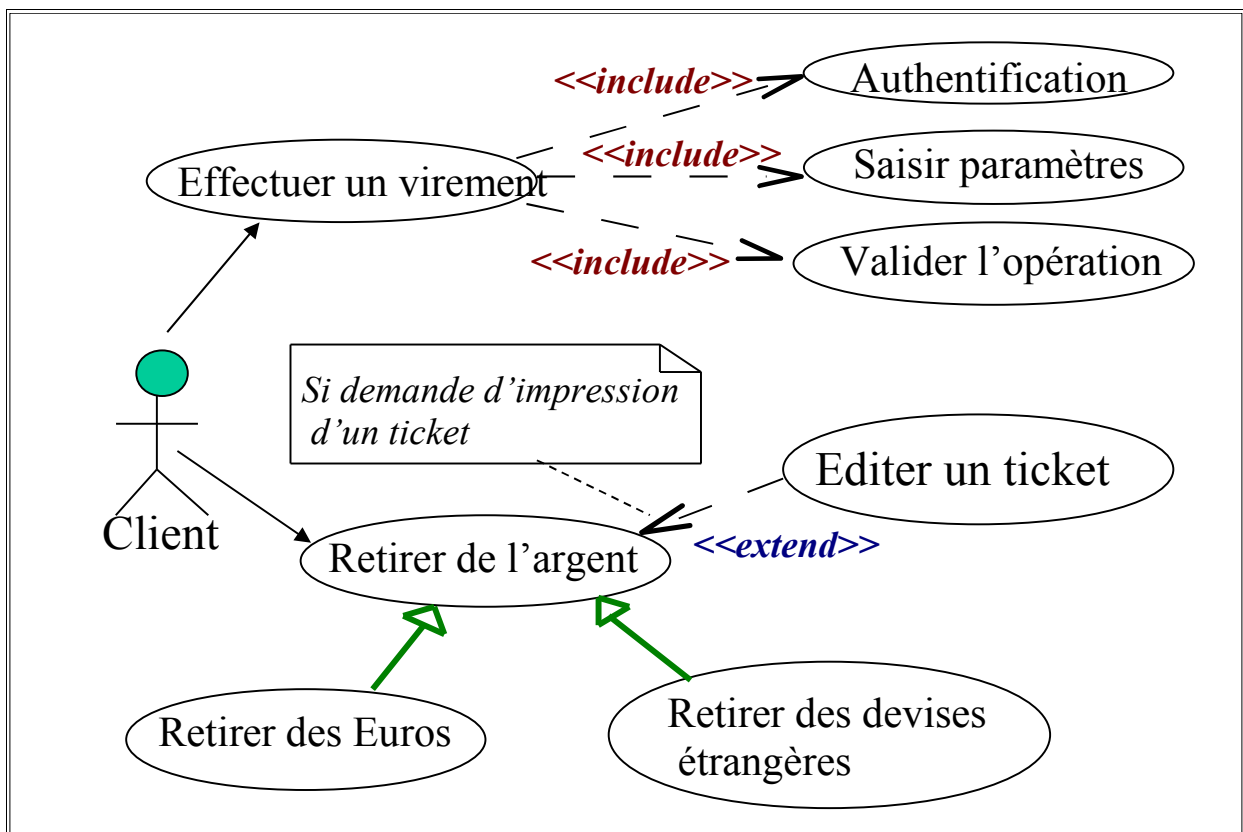


A prendre en compte:

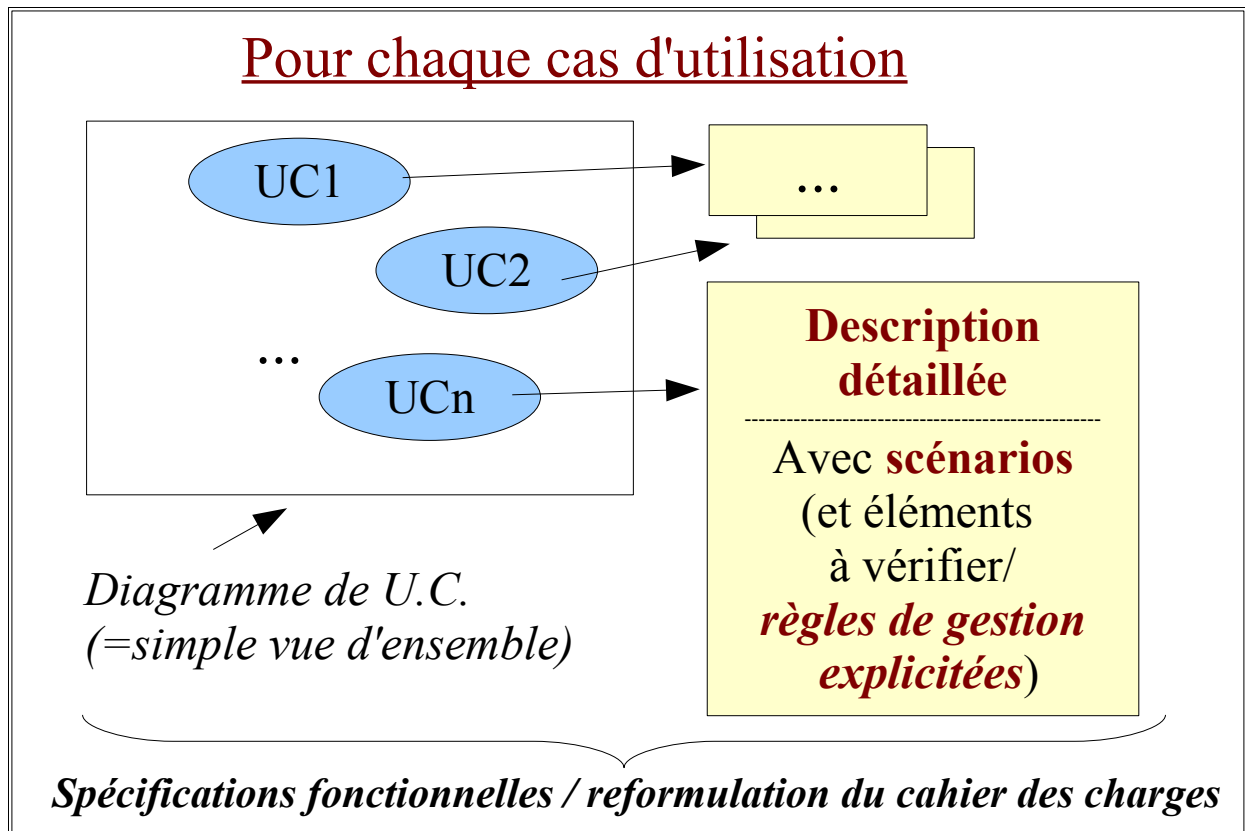
- * cas d'utilisation <--> session utilisateur
--> On peut donc relier entre eux les U.C. Effectués à peu près au même moment mais on doit séparer ce qui s'effectue à des instants éloignés (différentes sessions)
- * cas d'utilisation --> avec interaction avec l'extérieur .

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



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



Description textuelle (U.C.)

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

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

Scénarios (U.C.)

Scénario nominal:

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

Scénario Nominal

Tout se passe bien

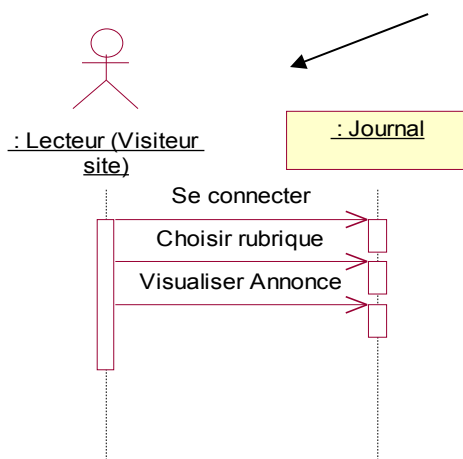
Cas/Déroulement
le plus fréquent

Scénario d'exception "E1":

- si l'étape (3) échoue trois fois de suite alors
- avaler la 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 de micro-activités**.

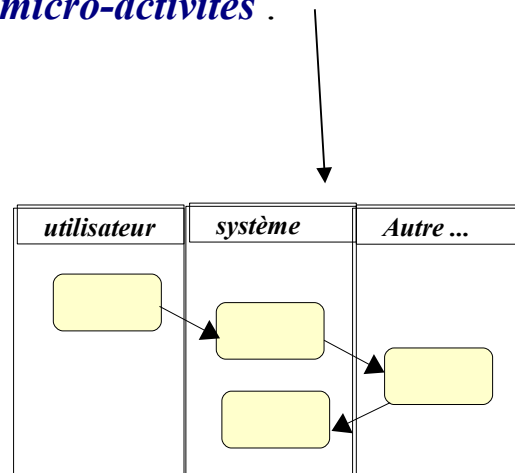


Tableau récapitulatif des U.C. (base pour la planification)

<i>Use Case</i>	<i>Estimation charge (j/h)</i>	<i>Priorité</i>	<i>Autres caractéristiques</i>
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>> (-,--)

V - Concepts objets et modularité

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: Employé 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.

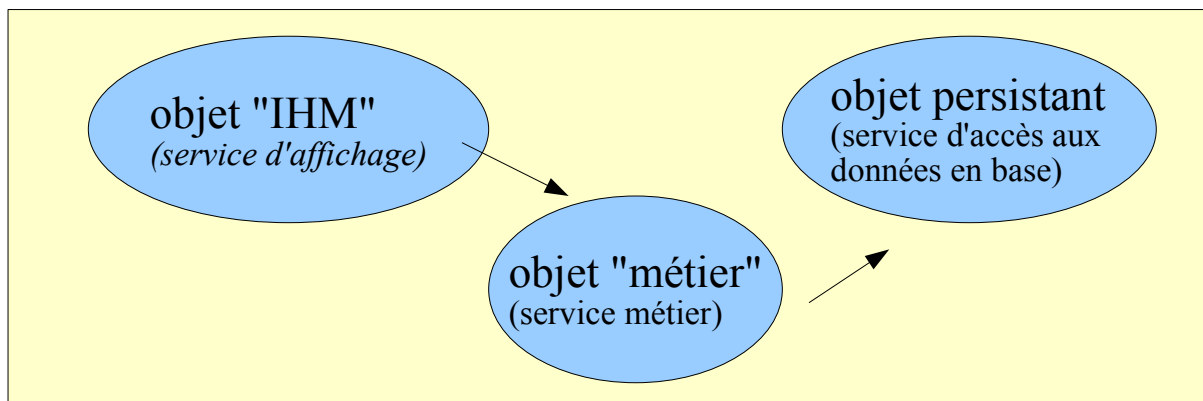
--> approfondir si besoin ensuite via Wikipédia ou "comment ça marche" ou autre.

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**.

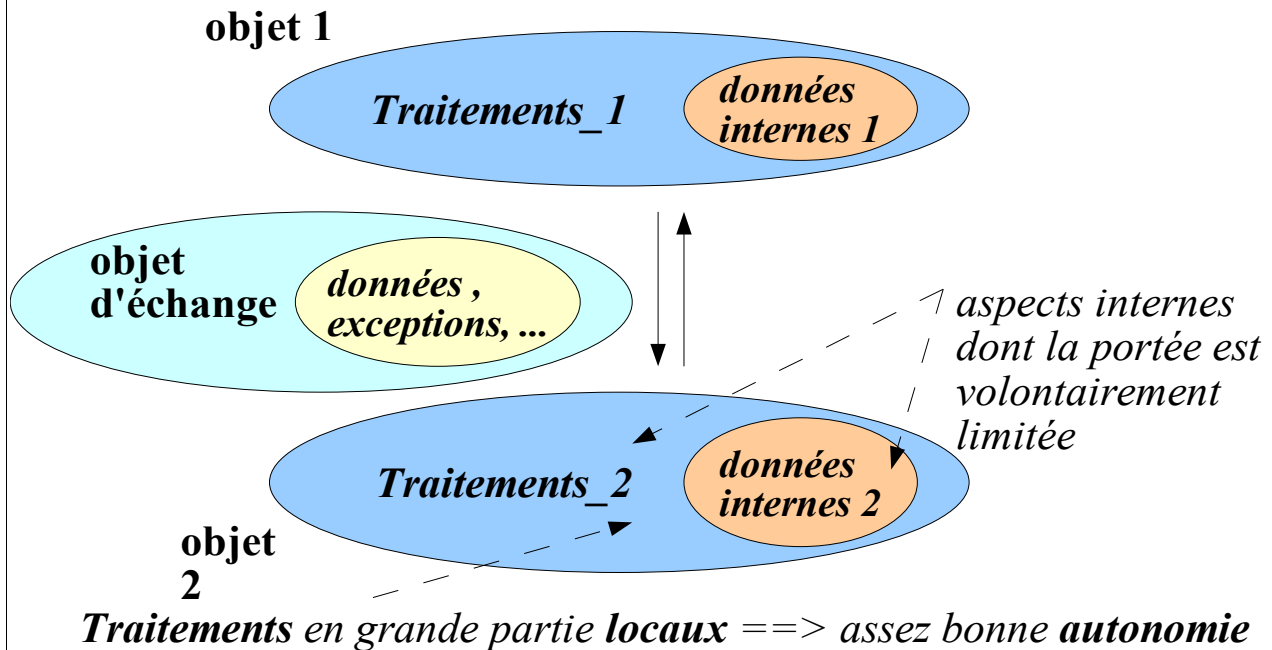


Grand traits de l'approche objet:

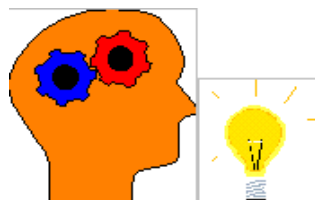
Un programme est un ensemble de petites **entités** . Chacune ayant :

- 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).



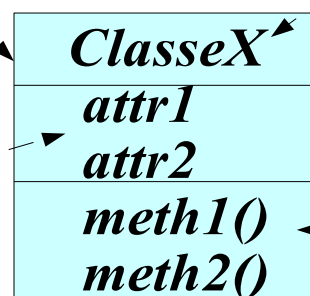
Abstraction des données (réalité => types abstraits)



abstraction

De quoi parle-t-on ?

Eléments pertinents ?



**Quelle utilité ?
Quels services ?**

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

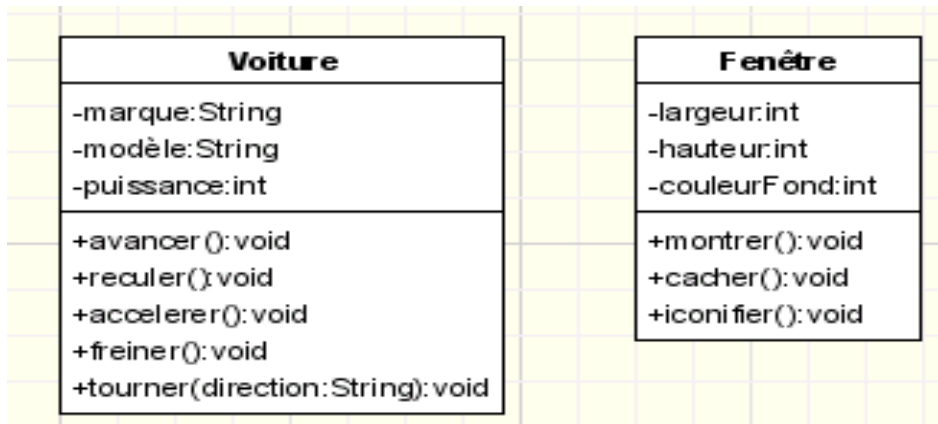
instances

Notion de classe

Pour **regrouper les objets aux caractéristiques et comportements identiques**, on fait appel à la notion de **classe**.

Une **classe** peut être vue comme un **type** (*abstrait ou concret*) d'objets.

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



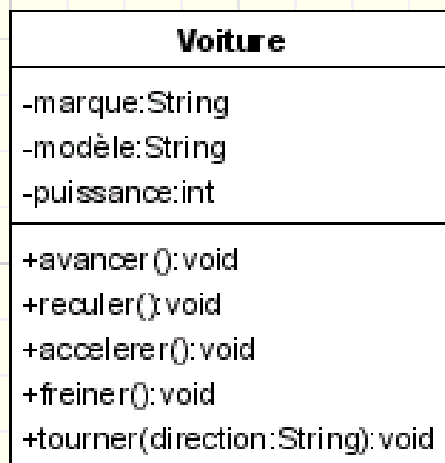
Toutes les instances d'une même classe ont une même structure commune et ont un même comportement.

Instances

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

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

Classe



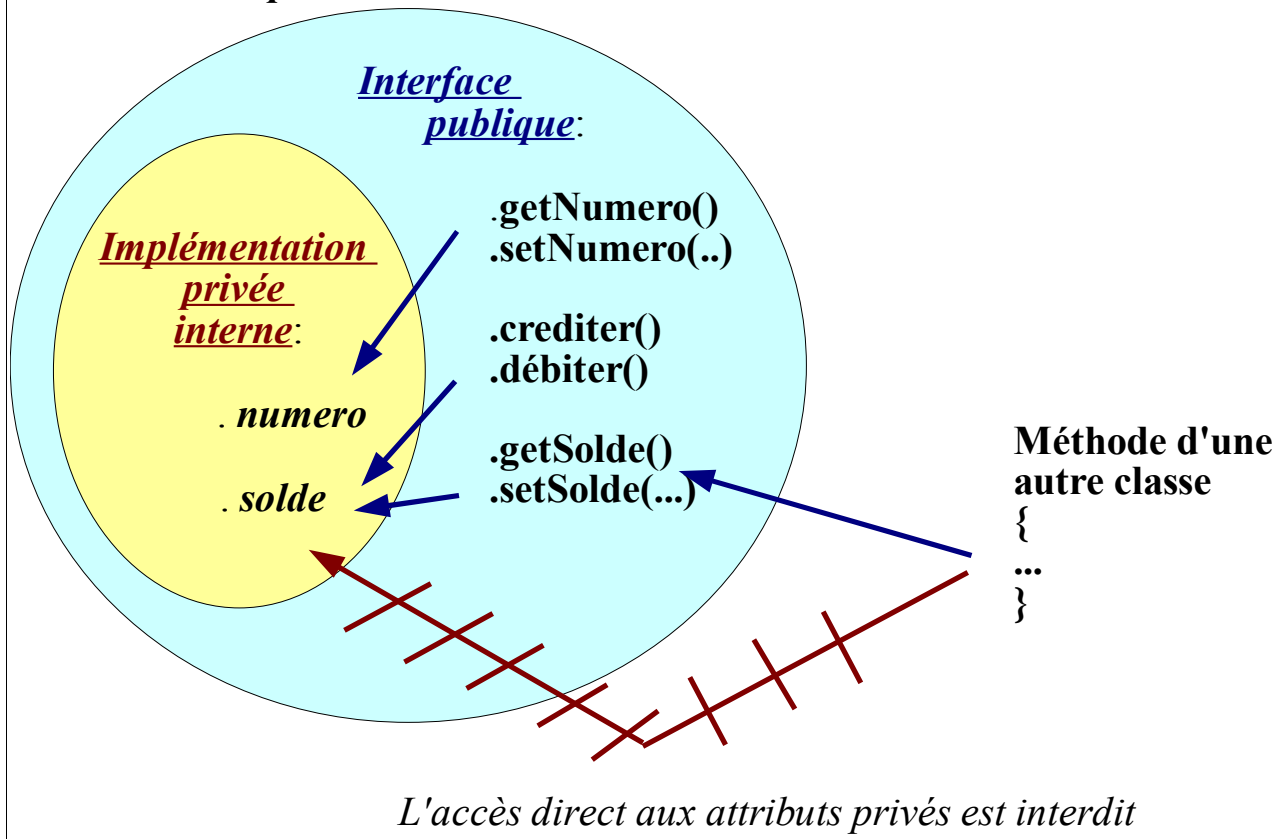
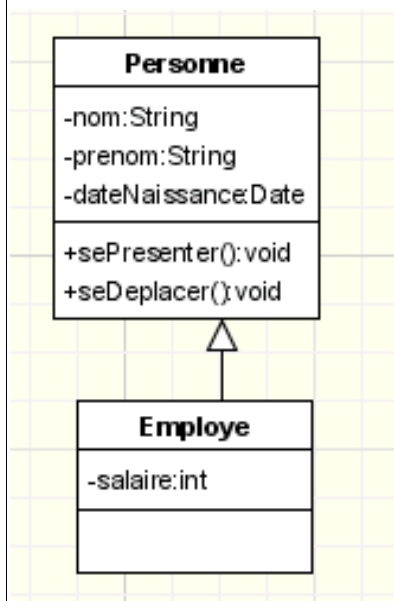
instance 1

V_AZ45BV456:Voiture
 marque=Renault
 modèle=Clio
 puissance=5

...

instance N

V_BN78BV936:Voiture
 marque=Peugeot
 modèle=307
 puissance=6

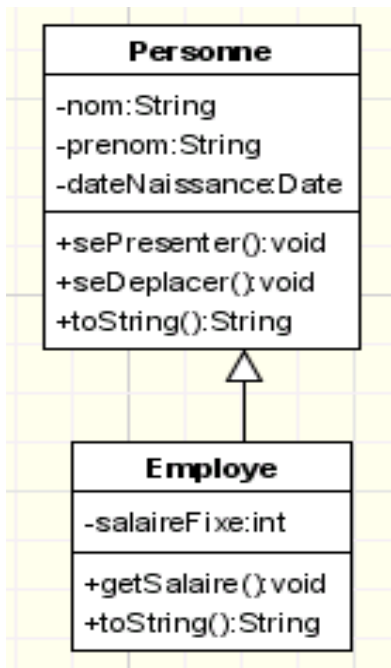
Classe Compte**Encapsulation****Héritage / Généralisation / Spécialisation**

* 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() { ... }
}
  
```

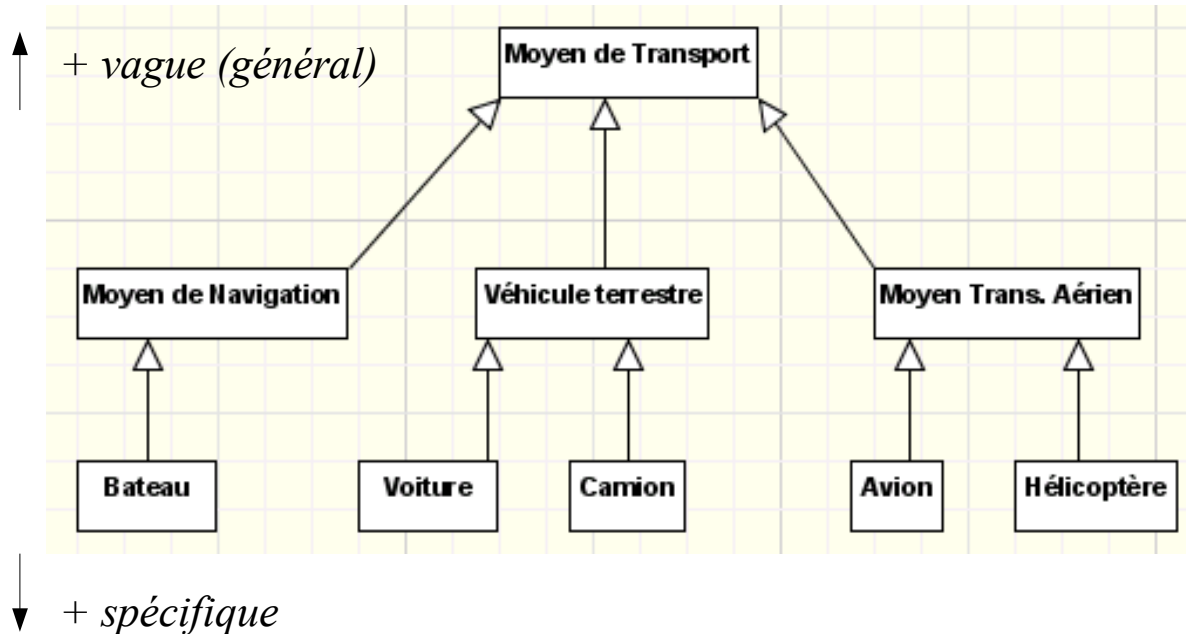
```

public class Employe extends Personne {
    private int salaireFixe;

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

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

Arbre (ou graphe) d'héritage



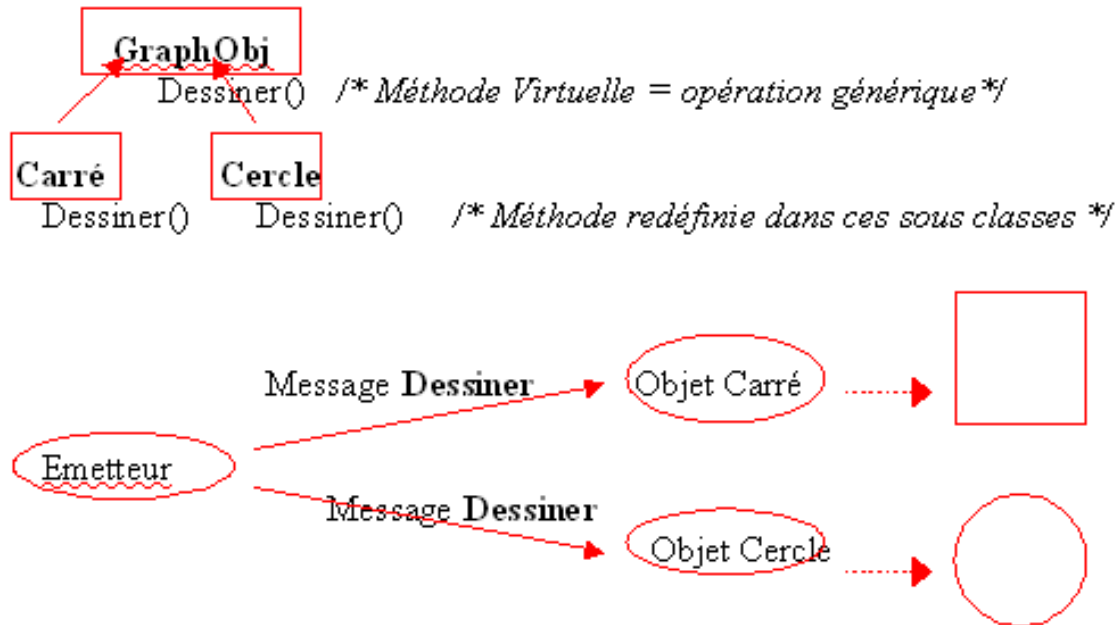
Une relation d'héritage est représentée par une grosse flèche triangulaire allant de la sous-classe vers la sur-classe et signifiant

"is kind of / est une sorte de"

Polymorphisme signifie littéralement "**plusieurs formes**".

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

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

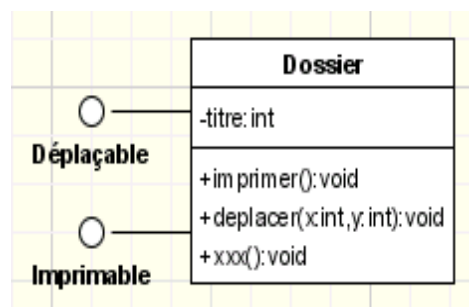
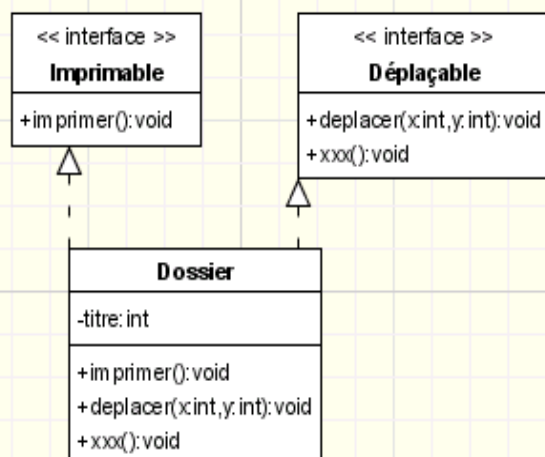


Interface

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

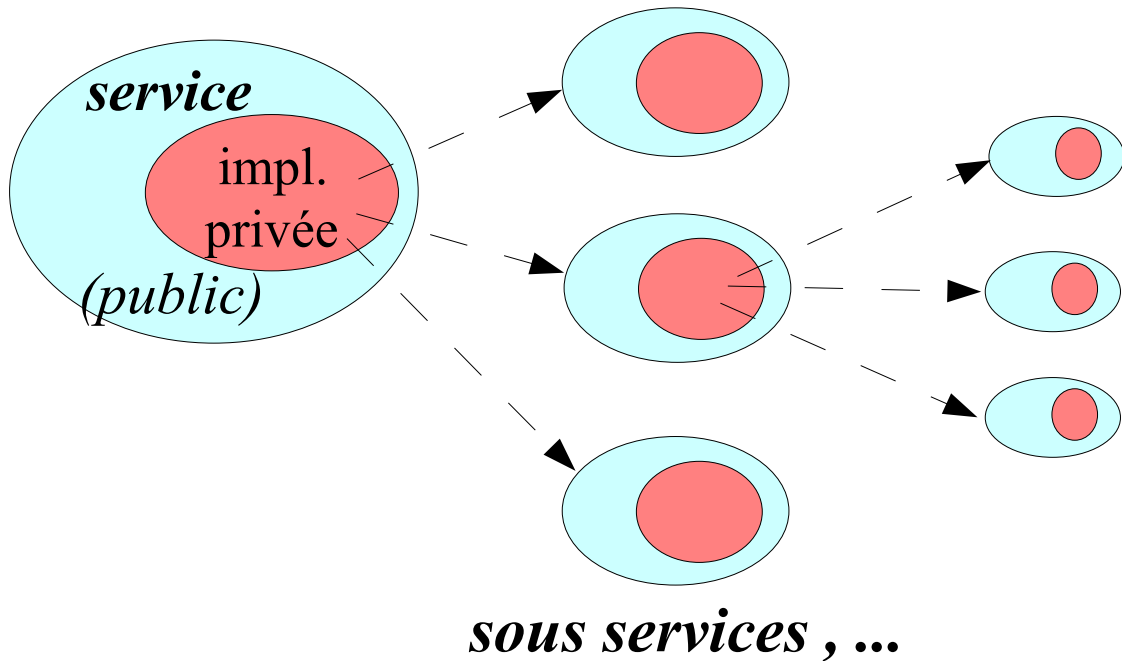
Pour être utile, une interface doit évidemment être entièrement codée au niveau d'une classe concrète.

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

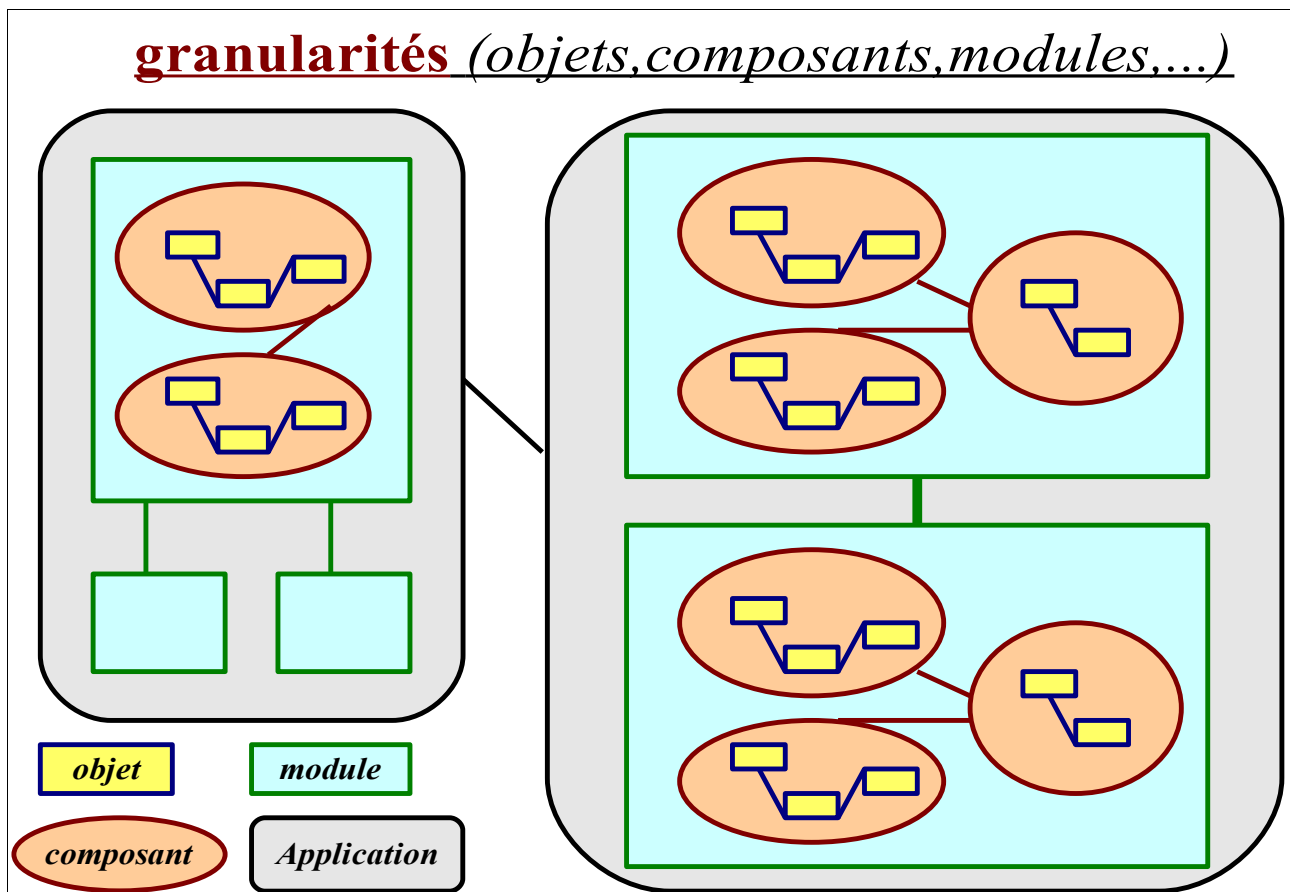


2. Granularité

Encapsulation & granularités



Modules , composants , objets



3. Modularité

3.1. Forte cohésion et couplage faible

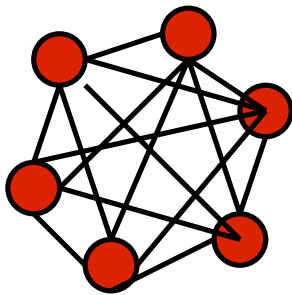
La **cohésion** mesure le degré de connectivité existant entre les éléments d'une classe unique. Une classe modélisant un aéroport aura une bien meilleure cohésion qu'une classe modélisant le roi d'une nation, le roi du jeu d'échec et le roi d'un jeu de carte . ***Il faut privilégier la cohésion fonctionnelle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

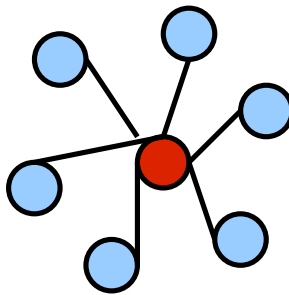
Couplage entre objets (idéalement faible)

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

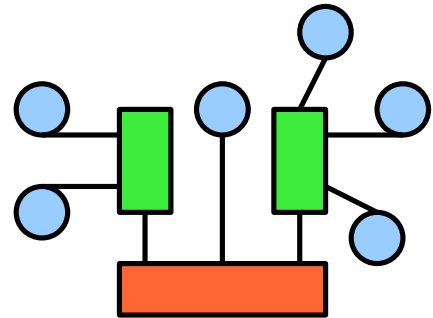
Un *assemblage d'objets complémentaires* rend globalement des *services plus sophistiqués*. Cependant la complexité des liens entre les éléments peut éventuellement mener à un édifice précaire:



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

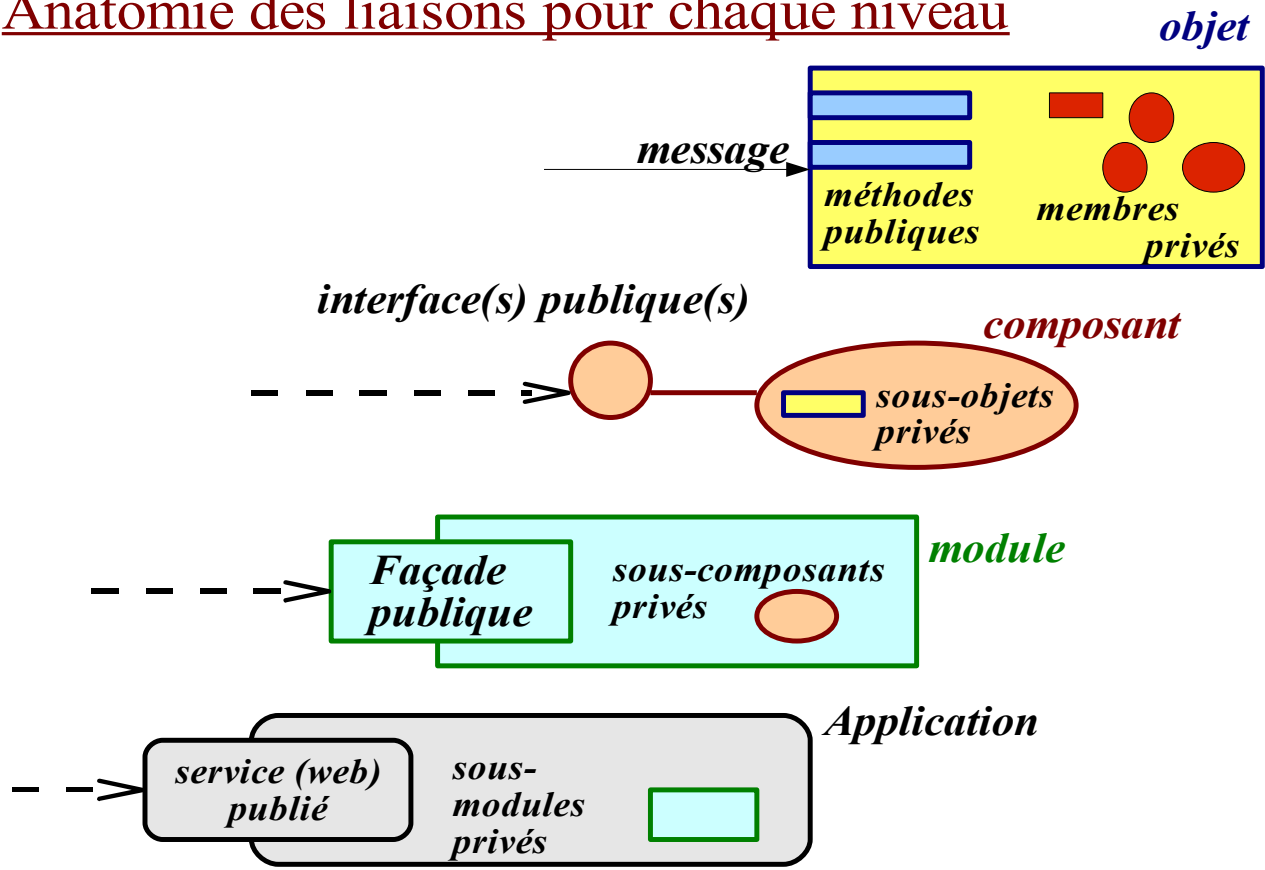


couplage faible
 mais *centralisé*
 ==> peu flexible
 et point central
 névralgique



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

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

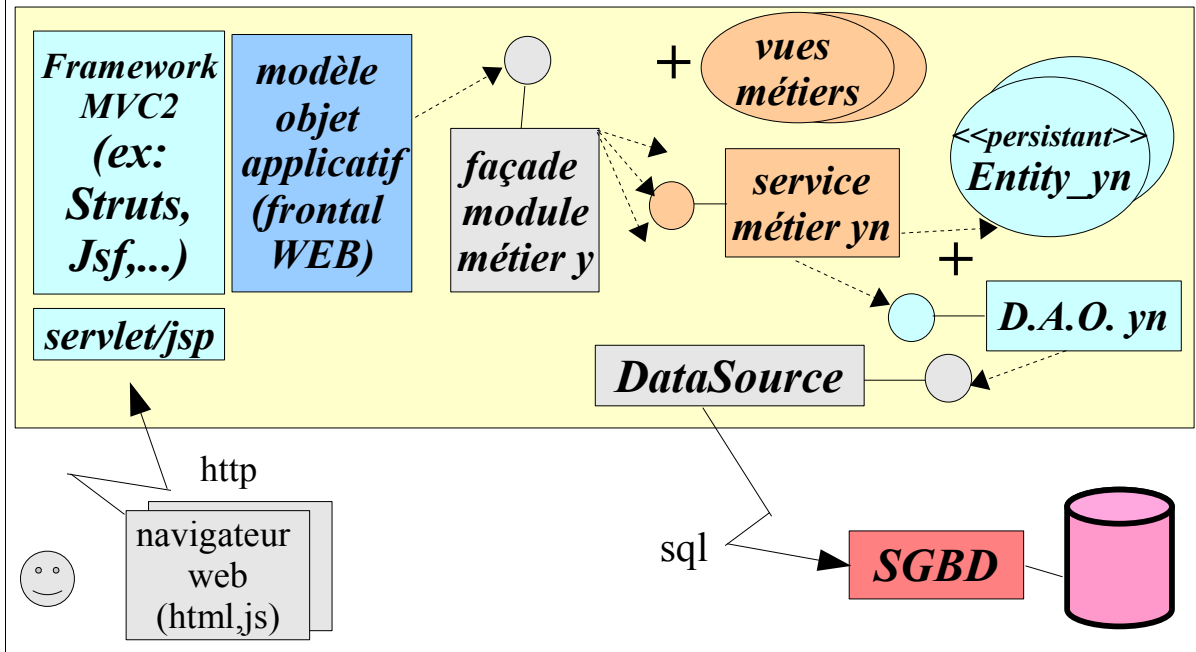
Anatomie des liaisons pour chaque niveau

VI - Aperçu sur architectures n-tiers

1. Structure globale d'une application

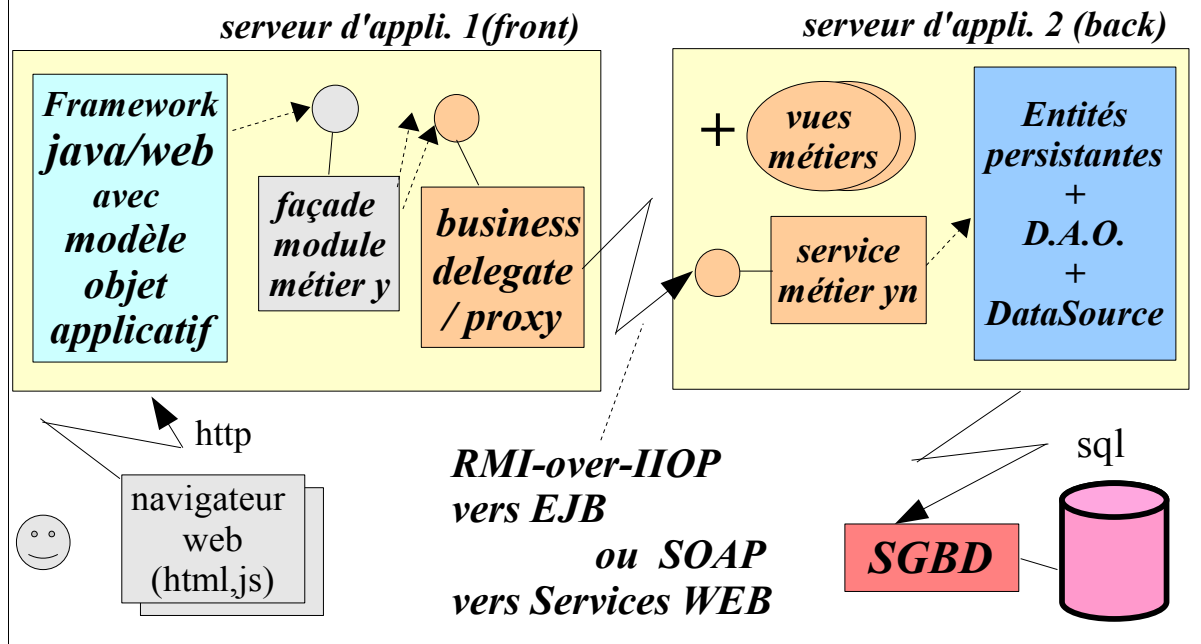
Tiers "présentation" et tiers "services métiers" déployés ensemble sur un même serveur JEE :

Architecture 3-tiers simplifiée: Application Web et services métiers sur un même serveur d'applications



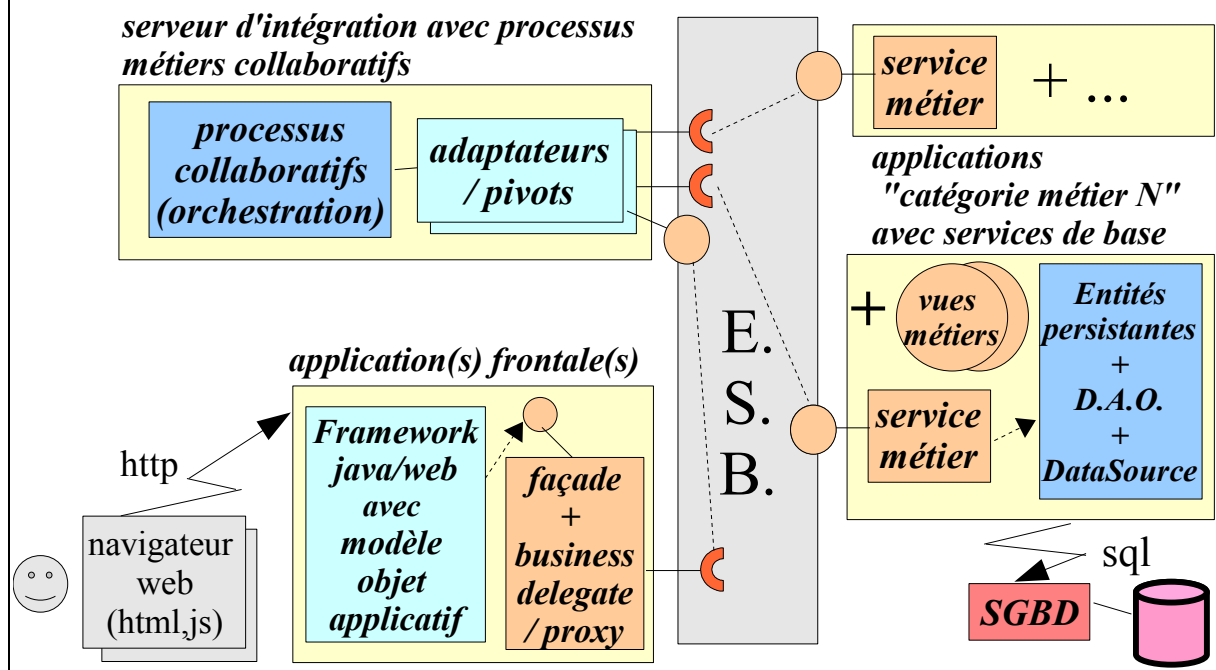
Accès distants aux "services métiers" (déployés sur un autre serveur JEE) :

Architecture 3-tiers distribuée: Application Web, services métiers sur 2 serveurs distincts.



Eventuelle utilisation d'une infrastructure SOA (ESB/JBI) pour prendre en charge des services de hauts niveaux qui invoquent des services de plus bas niveaux (ex: "commande" invoquant "facturation" et "expédition"):

Architecture n-tiers orientée services (S.O.A.): avec Bus "E.S.B." et processus collaboratifs.



VII - Responsabilité , CRC , patterns GRASP

1. Notion de responsabilité (contexte orienté objet)

On appelle "**Responsabilité**" un *ensemble de méthodes/opérations* qui forment en tout un *paquet bien cohérent et bien délimité de traitements informatiques*.

Exemples:

afficher() , setVisible() , saisir() ,

==> Responsabilité "*Affichage / IHM / Présentation*"

rechercher() , sauvegarder() , supprimer() , créer/insérer()

==> Responsabilité "*Persistence / accès aux données*"

Règle de bon sens:

Comme un être humain, une classe d'objet informatique devrait idéalement assumer un nombre assez limité de responsabilités (et le faire bien) [inverse de "tout superficiellement"].

2. Idée des cartes "CRC"

Dans les années "90" , une méthodologie objet avait eu l'idée d'utiliser des petites cartes pour modéliser rapidement la raison d'être d'une classe:

- * Nom de la **C**lasse
- * Ses **R**esponsabilités propres
- * Collaboration avec d'autres éléments (autres responsabilités déléguées)

CRC: Classe , Responsabilité(s) , Collaboration(s).

3. Affectation des responsabilités (GRASP)

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

GRASP = *General Responsibility Assignment Software Patterns*

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

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

Les 4 patterns "GRASP" fondamentaux

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

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

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

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

5 patterns "GRASP" plus spécifiques

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

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

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

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

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

4. Les 4 patterns GRASP fondamentaux

4.1. Expert

Expert (GRASP)

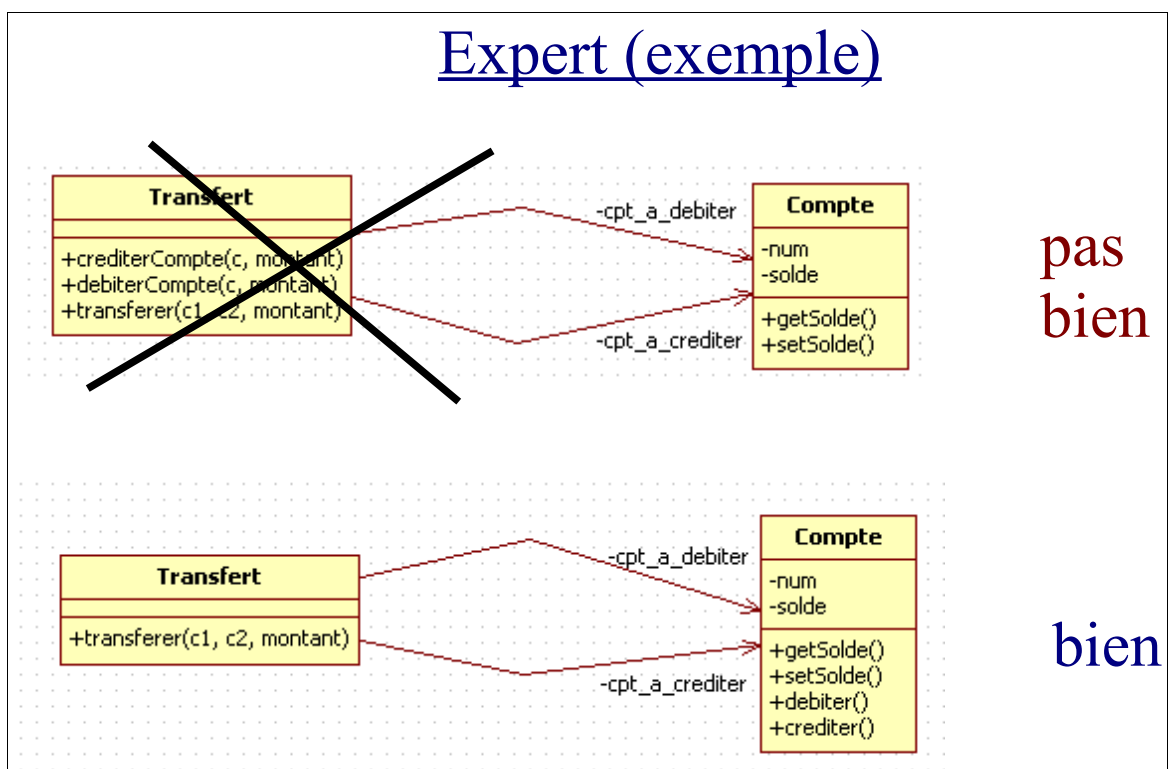
L'*expert*, c'est celui qui sait (qui détient quelques informations clefs) et qui peut donc prétendre pouvoir assumer une certaine responsabilité.

--> 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)



4.2. Faible couplage

Faible couplage (GRASP)

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

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

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

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

Faible couplage (exemple)

*faible couplage
non respecté !!!*



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

4.3. Forte cohésion

Forte cohésion (GRASP)

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

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

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

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

Forte cohésion (contre exemple)

Refrigerateur_TV_Alarme

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

*tout et
n'importe-quoi !*

4.4. Création

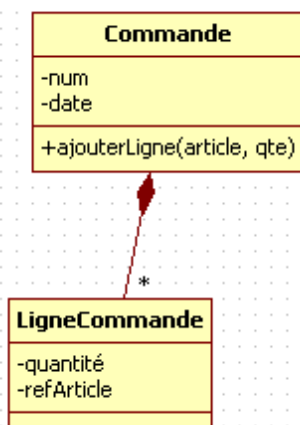
Créateur (GRASP)

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)



```
nouvelleLigne = new LigneCommande(...);
lignes.add(nouvelleLigne);
```

*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").

VIII - Analyse du domaine (entités) / diag. classes

1. Analyse du domaine (glossaire , entités)

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

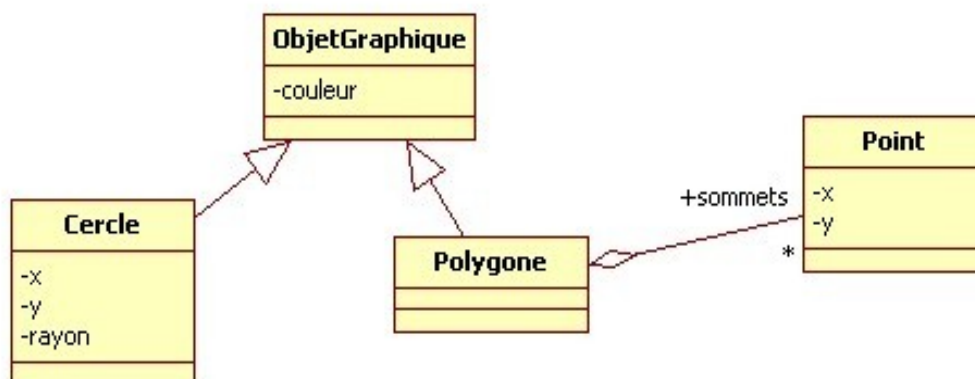
Partant d'un niveau très conceptuel (arbre sémantique, glossaire) elle permet d'aboutir à un **modèle logique de type "entités/rerelations"** résolument restreint à ce qui touche au domaine du système à développer (sans éléments inutiles) . Idéalement basées sur des formulations de type "chose concrète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

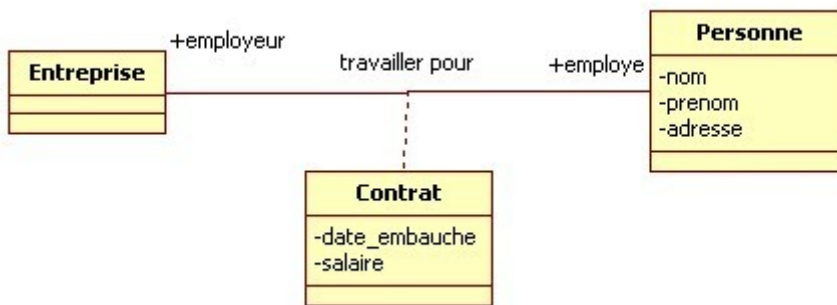
L'étape "**analyse du domaine**" est généralement effectuée de la façon suivante:

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

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

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





Quelques remarques:

- Ne pas introduire trop tôt (dès 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*)

Nom de classe

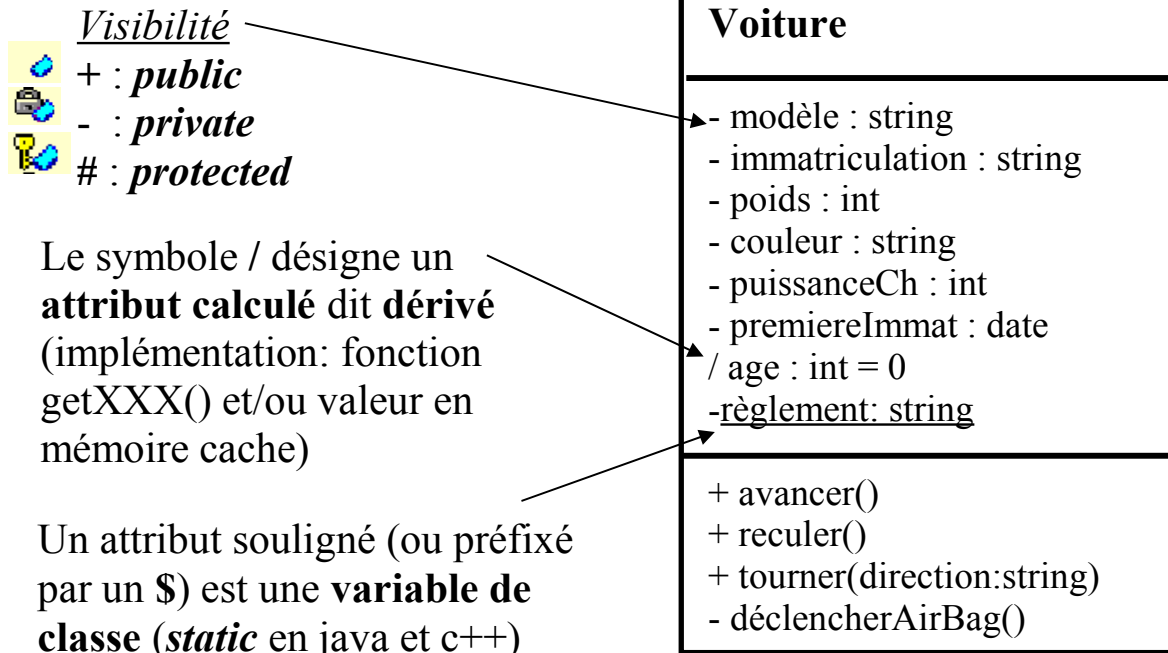
Nom de classe

Nom de classe

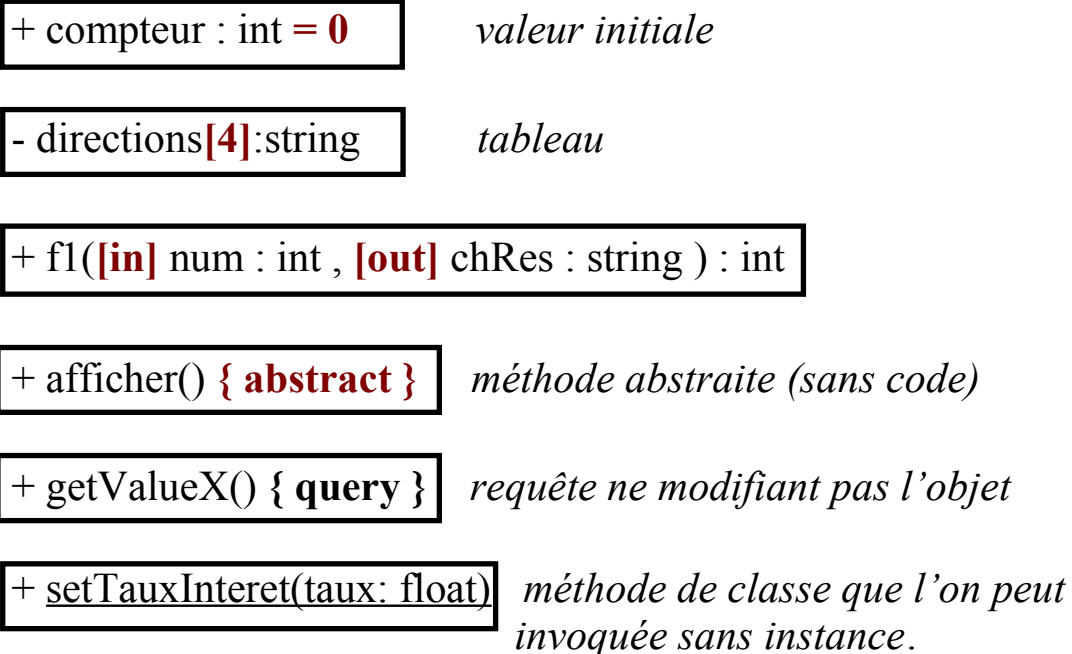
-Attribut1
-Attribut2

+MéthodeA()
+MéthodeB()

Détails sur les éléments d'une classe

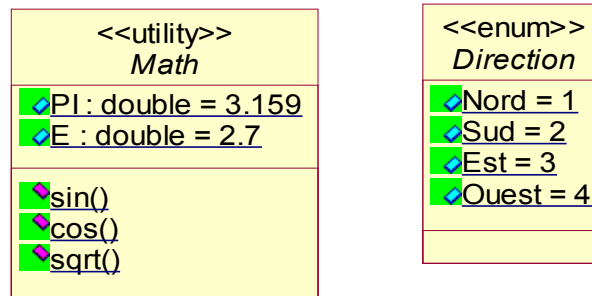


Autres détails sur les caractéristiques d'une classe



Classes spéciales (utilitaires, énumération, ...)

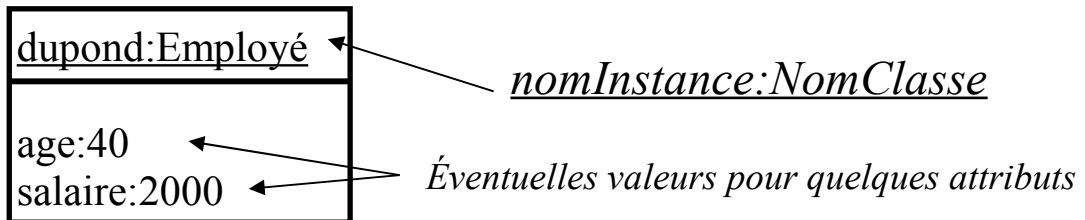
- Dans un monde matériel et rationnel ou tout est objet, il n'y a plus de place pour des fonctions globales (anarchiques).
Celles-ci doivent être rangées dans des classes utilitaires.
- Les constantes doivent elles aussi être placées dans des classes (ou interfaces) d'énumération.



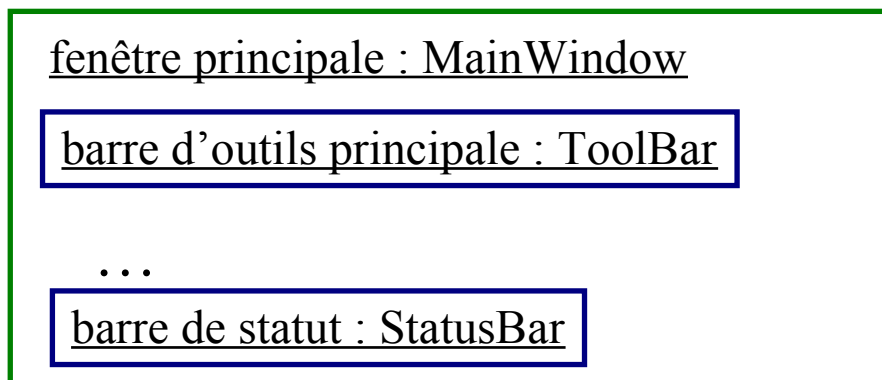
y = **Math.sqrt**(x);

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

Eventuels (et rares) diagrammes d'instances



Instance composée (de sous objets):



Associations (relations)

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



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



Extrémités d'association

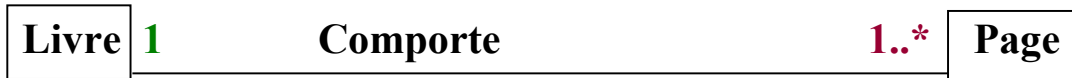
- Une **association** n'appartient pas à une classe mais à un package (celui qui englobe le diagramme de classe).
- On peut préciser des caractéristiques d'une association qui sont liées à une de ses **extrémités (association end)**:
 - *rôle* joué par un objet dans l'association
 - *multiplicité*
 - *navigabilité*
 - ...

Multiplicité UML (cardinalité)

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)

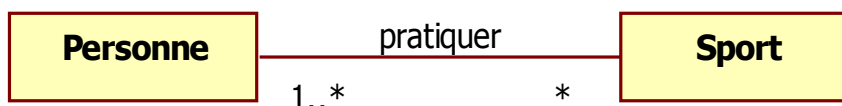
Les **multiplicités** permettent d'indiquer (pour chacune des classes) les nombres minimum et maximum d'instances mises en jeu dans une association.

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)

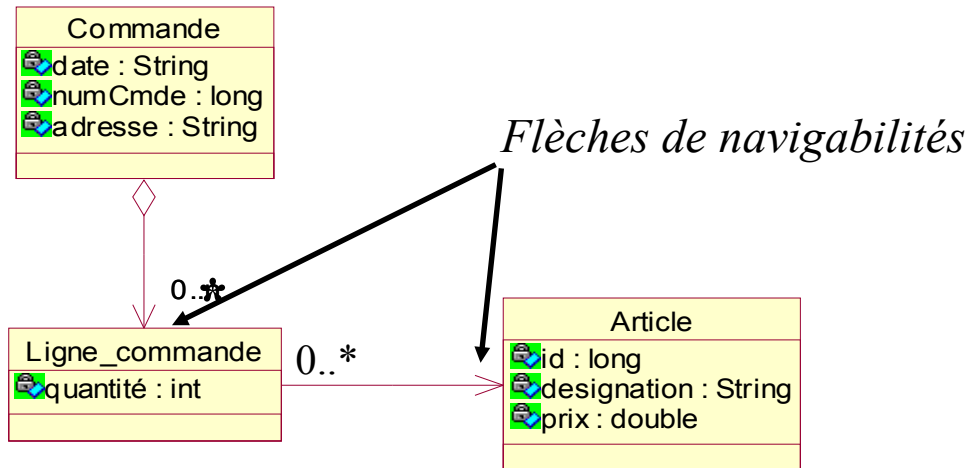


NB:

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

Navigabilité

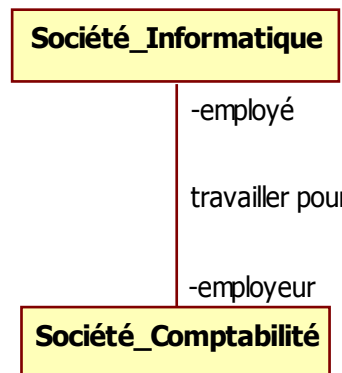
Une **flèche de navigabilité** permet de restreindre un accès par défaut bidirectionnel en un **accès unidirectionnel** *plus simple à mettre en œuvre et engendrant moins de dépendance*.



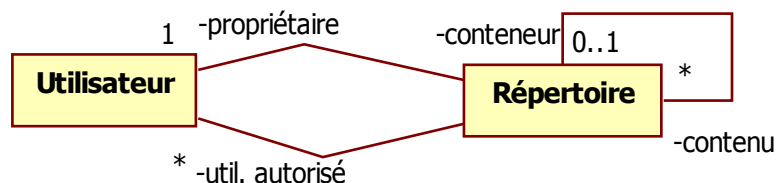
Un article n'a pas directement accès à une ligne de commande

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.

```

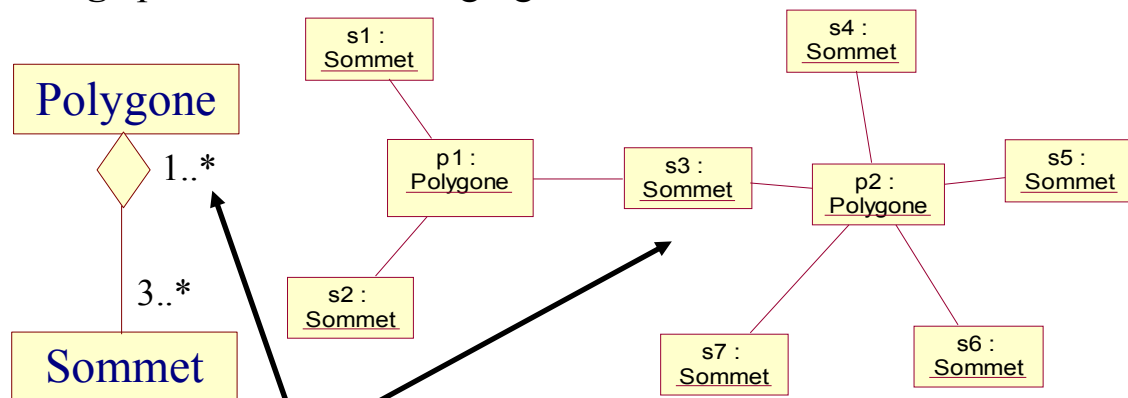
LivreA:Livre
-----
titre=Les Misérables
auteur
  
```

```

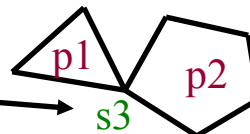
auteurXY:Personne
-----
nom=Victor Hugo
  
```

Agrégation

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



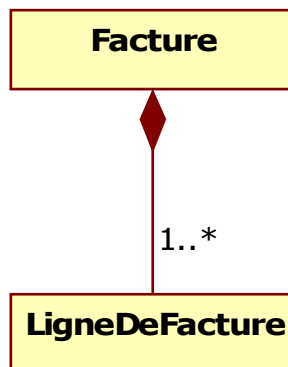
Un agrégé peut faire partie de plusieurs agrégats



Agrégation (caractéristiques)

- Une **agrégation** est une association de type "**est une partie de**" qui vu dans le sens inverse peut être traduit par "**est composé de**".
- UML considère qu'une **agrégation est une association bidirectionnelle ordinaire** (le losange ne fait qu'ajouter une sémantique secondaire).
- Une agrégation (faible) ordinaire implique bien souvent que les sous objets soient **référéncés** par leur(s) agrégat(s).

Composition (agrégation forte)



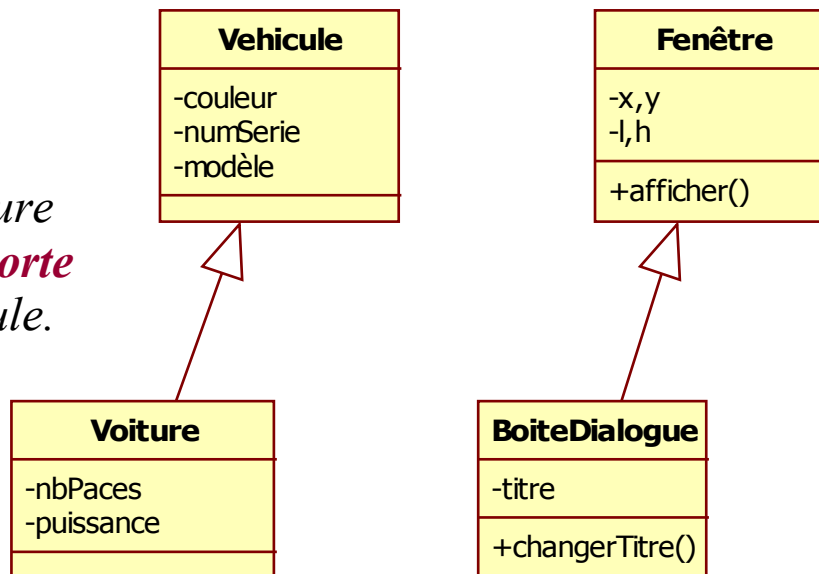
En général, le sous-objet n'existe que si le conteneur (l'agrégat) existe: **lorsque l'agrégat est détruit, les sous objets doivent également disparaître.** (*"cascade-delete" en base de données*).

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

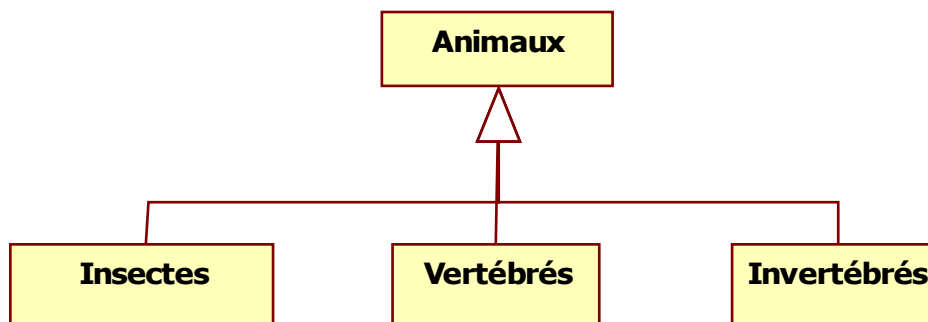
Remarque: Le losange est quelquefois rempli de noir pour montrer que le sous objet est physiquement compris dans le conteneur (l'agrégat). On parle alors d'**agrégation forte** (véritable **composition**).

Généralisation (héritage)

*Une voiture
est une sorte
de véhicule.*



Classification (généralisation)

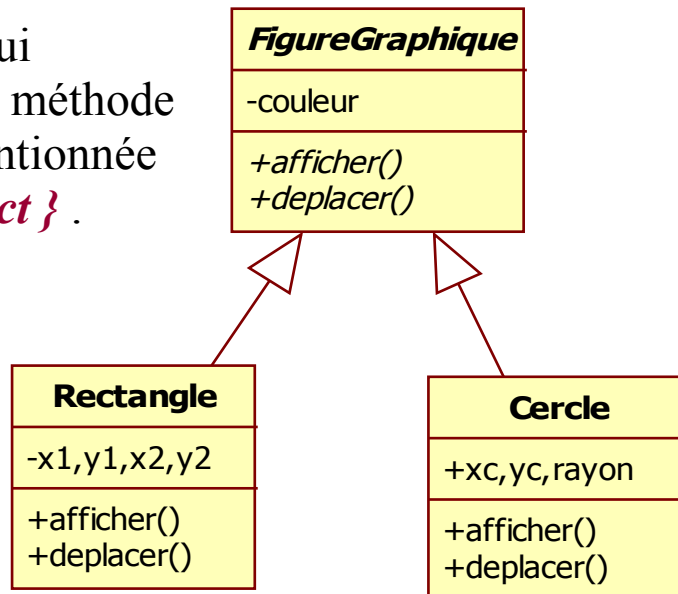


Les animaux peuvent être classées dans divers **groupes** (et sous groupes).

Classification selon caractéristiques discriminantes.

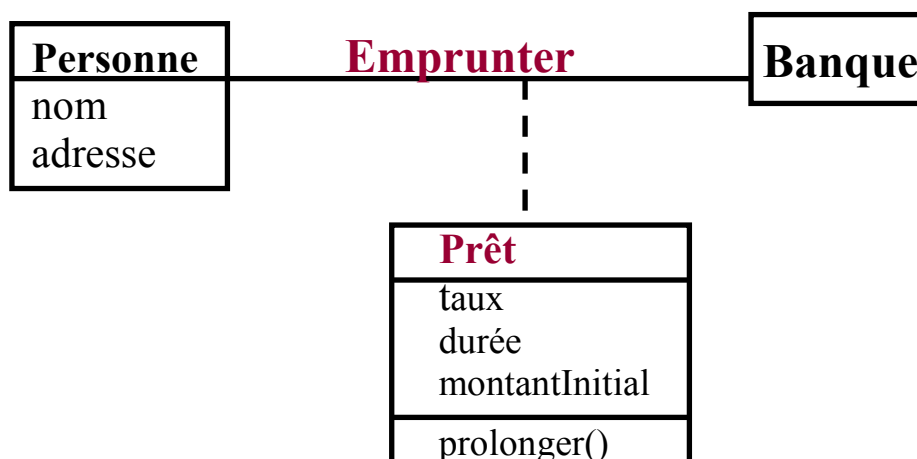
Classes abstraites et concrètes

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

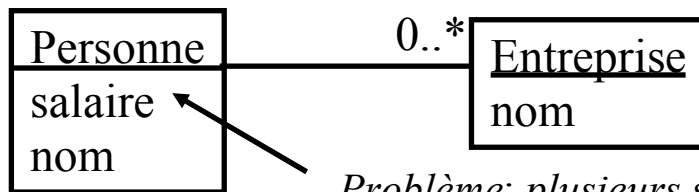


Classes d'association

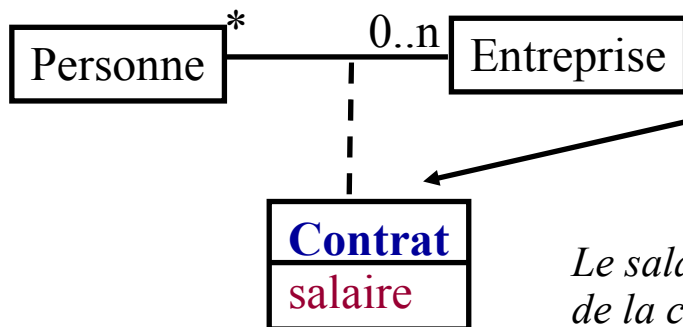
Lorsqu'une **association** comporte des **données** ou des **opérations** qui lui sont **propres** (non liés à seulement une des entités mises en relation), on a souvent recours à des **classes d'association**.



Classes d'association (exemples)



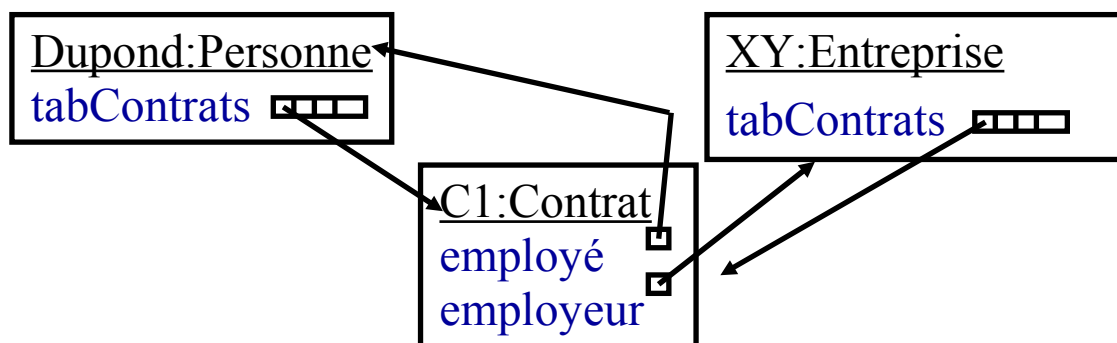
Problème: plusieurs salaires si la personne travaille à temps partiel dans plusieurs sociétés.



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

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

Implémentation des classes d'associations



Un objet contrat devient un intermédiaire permettant d'accéder à chacune des entités de l'association:

*Contrat1.**getEmploye()**; Contrat1.**getEmployeur()**;*

Package(s) et Namespaces

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

3. Éléments structurants d'UML

<i>Regroupements UML</i>	<i>Sémantiques / caractéristiques</i>
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 diagrammes est purement pragmatique : selon la place)

3.1. Modèle

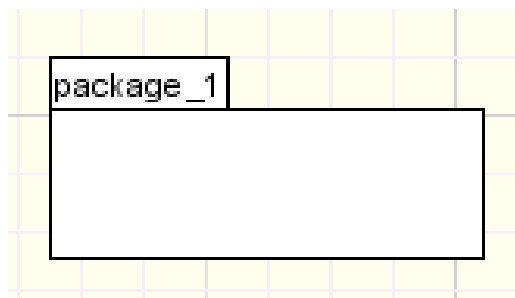
Correspondant généralement à une application entière (ou bien à un sous système) , un **modèle UML** est essentiellement constitué par une **arborescence d'éléments** .

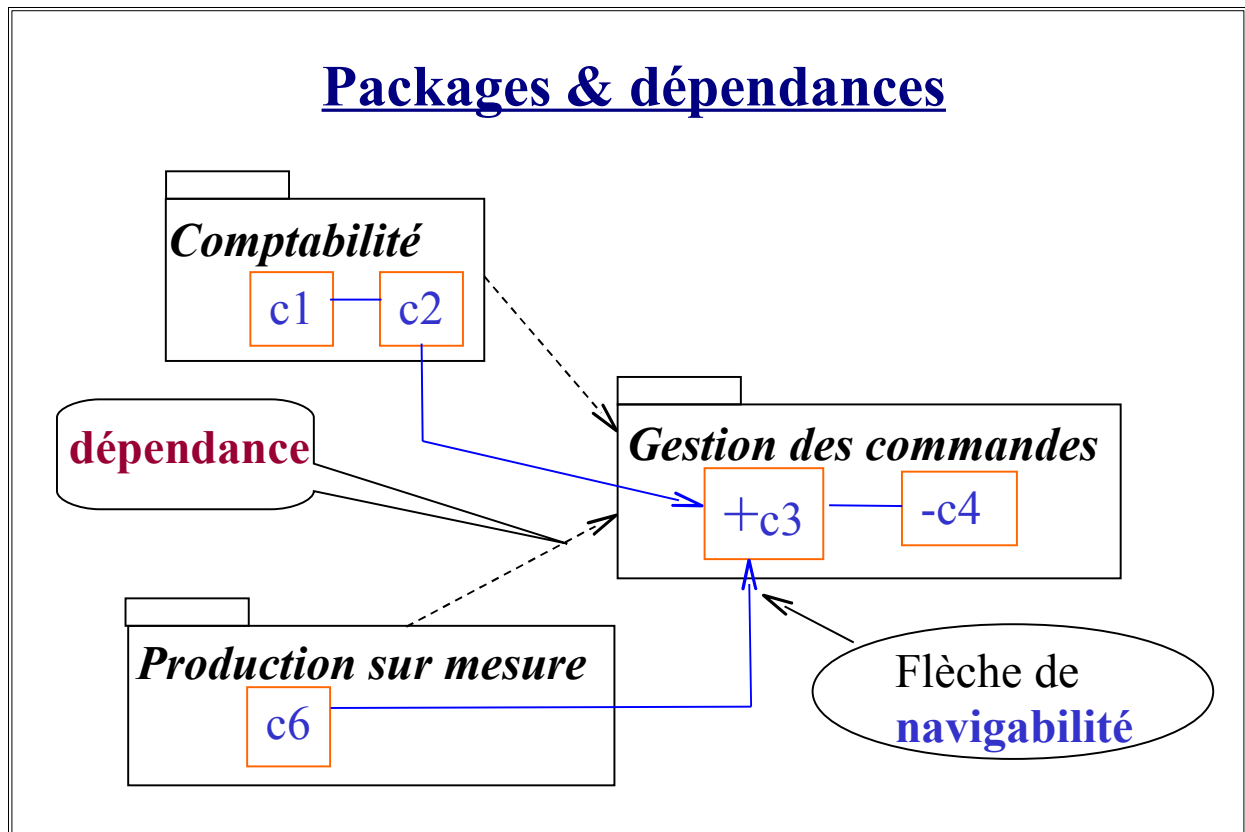
C'est à partir de son contenu (*xy.uml* ou *xy.xmi*) que l'on pourra éventuellement générer du code via MDA.

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 de l'arborescence du modèle.
- Lorsque l'on souhaite supprimer un élément que sur le diagramme courant --> Suppr/Delete
- Lorsque l'on souhaite supprimer un élément dans tous les diagrammes et dans le modèle --> **Edit / Delete From Model**.

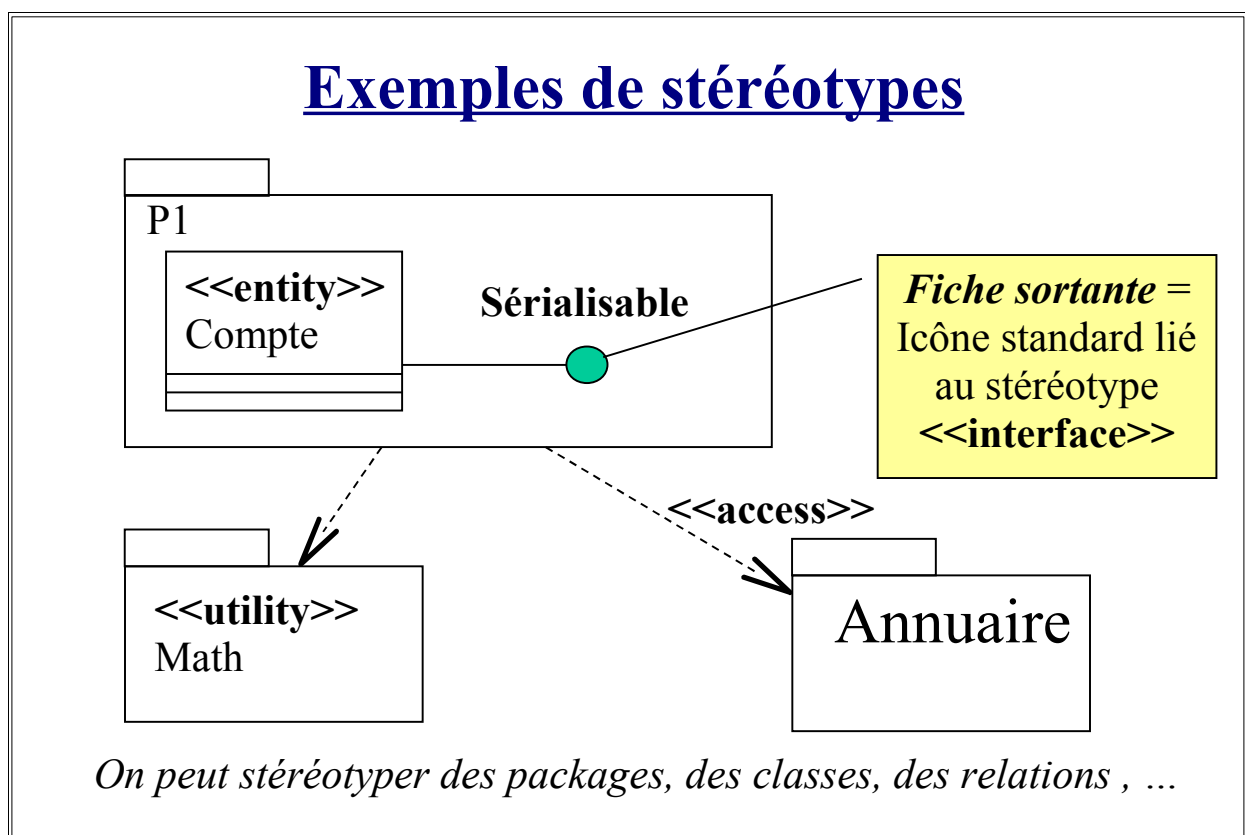
3.4. Stéréotypes

- Un **stéréotype** est une **nouvelle information** (supplémentaire à la classe ou une relation ou ...) permettant de **préciser la nature** d'une famille d'**éléments de la modélisation UML**.
- Un **stéréotype** est une **extension vis à vis des bases d'UML**. Ceci permet d'introduire de nouveaux concepts. **UML est ainsi ouvert** vers de nouvelles notions.

Notation:

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

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



Quelques exemples de stéréotypes:

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

Selon l'outil UML utilisé, un stéréotype peut être:

- soit créé à la volée pour être utilisé immédiatement
- soit préalablement créé (parmi d'autres) au sein d'un profile UML pour être ensuite sélectionné.

3.5. Valeurs étiquetées (Tag Values)

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

Les "tag values" ne sont généralement pas visibles dans les diagrammes UML.

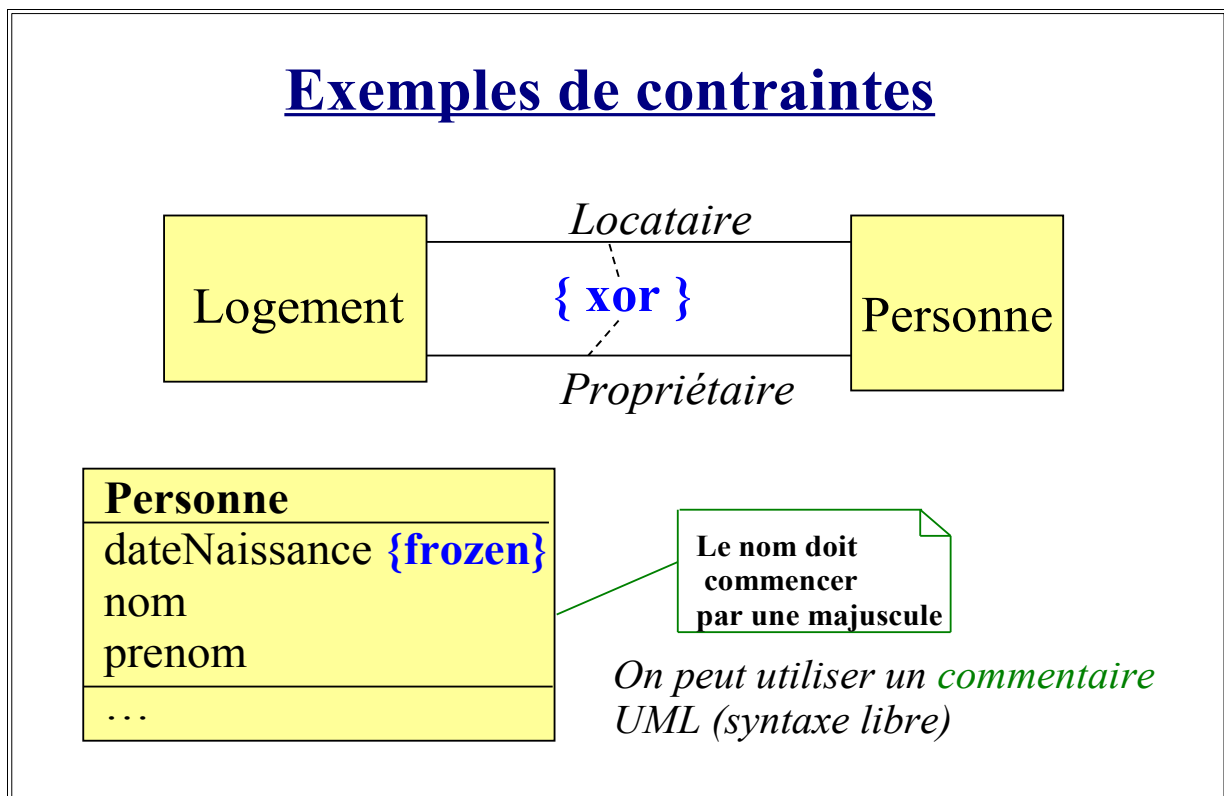
Ces valeurs cachées ne sont donc utiles que si elles sont ultérieurement analysées par un programme quelconque (générateur de documentation, générateur de code MDA, ...).

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

3.6. Contraintes

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

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



NB:

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

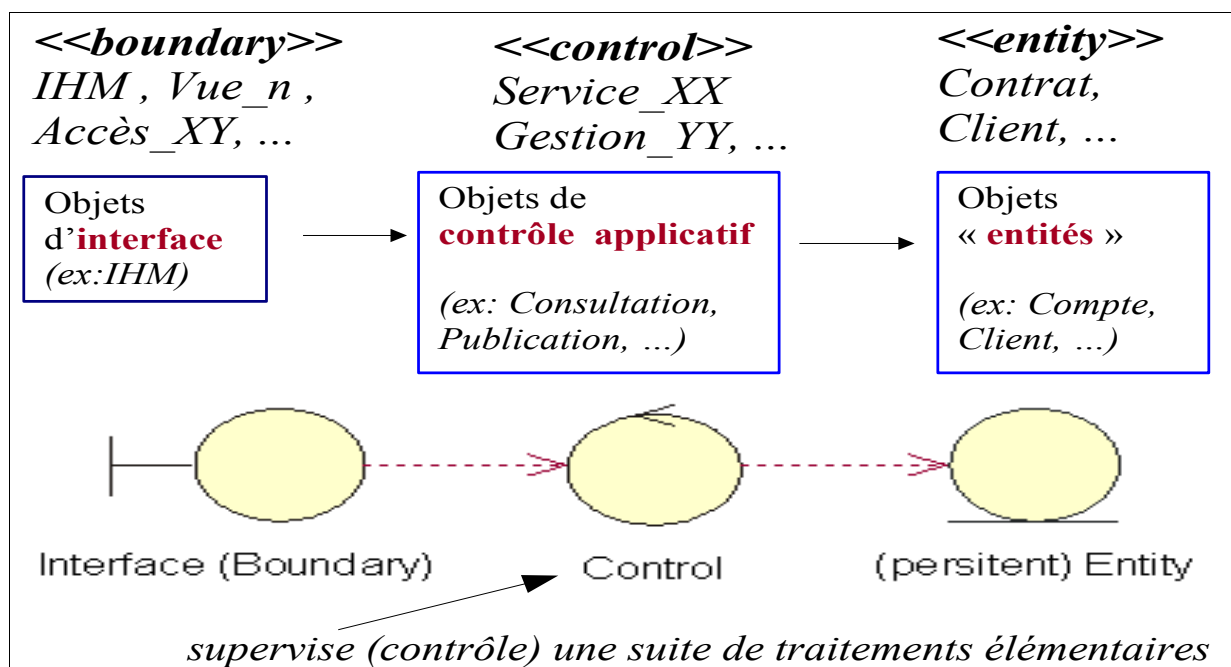
IX - Modélisation des services métiers

1. Analyse applicative (objectif et mise en oeuvre)

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

- **Identifier toutes les classes nécessaires** (avec les stéréotypes d'analyse <<entity>> , <<control>> ou <<service>> et <<boundary>> ou <<ihm>>).
- établir un (ou plusieurs) **diagramme(s) de classes montrant les classes participantes**
- Pour chaque "Use Case" identifié :
 - *retranscrire le scénario nominal* sur un nouveau *diagramme de séquence uml montrant les envois dynamiques de messages entre les objets identifiés de l'analyse applicative*.
 - montrer sur certains diagrammes de classes la liste des *opérations (méthodes) ajoutées* sur les *classes qui réalisent (par collaboration) les fonctionnalités d'un cas d'utilisation*.

Stéréotypes d'analyse de Jacobson :



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

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

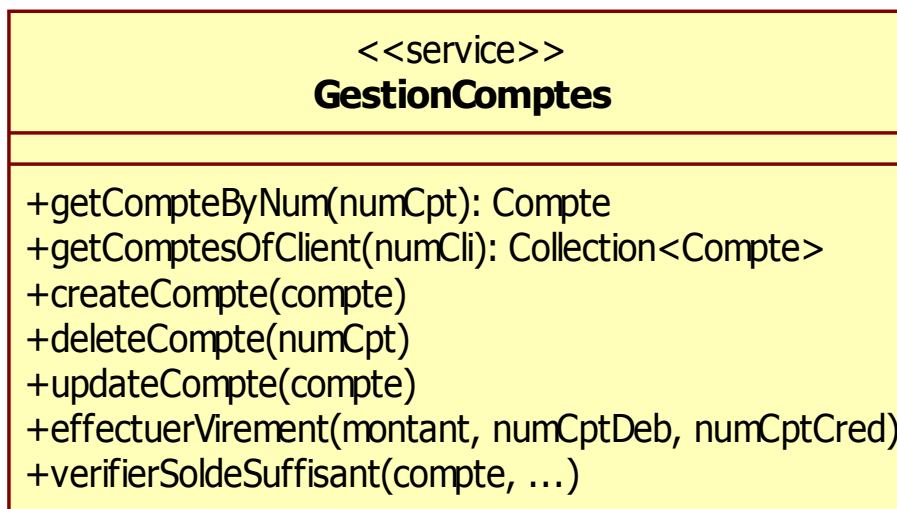
2. Responsabilités (n-tiers) et services métiers

<i>Eléments d'une application</i>	<i>Responsabilités</i>
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"):

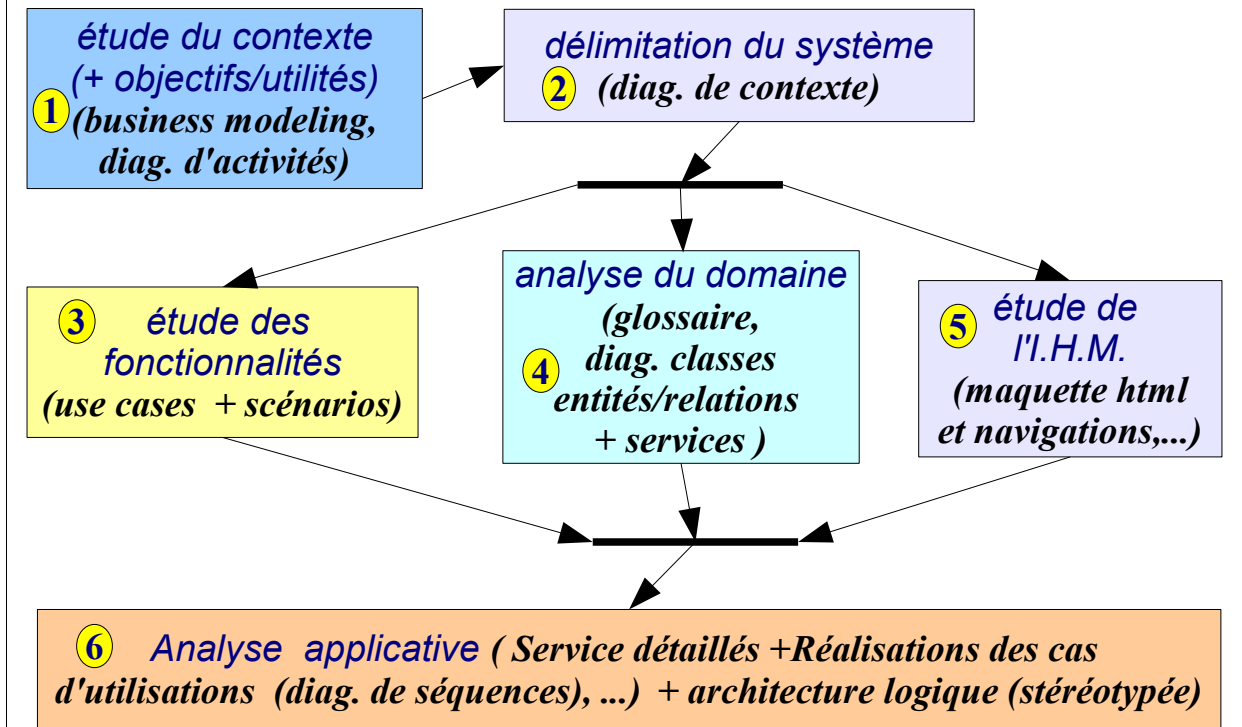


Pour identifier les objets de contrôles applicatifs / services métiers on peut se baser sur les compléments d'objets directs des U.C. ou bien sur les noms des packages .

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

3. Repère méthodologique (rappel)

Enchaînement classique d'activités sur la partie fonctionnelle



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

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

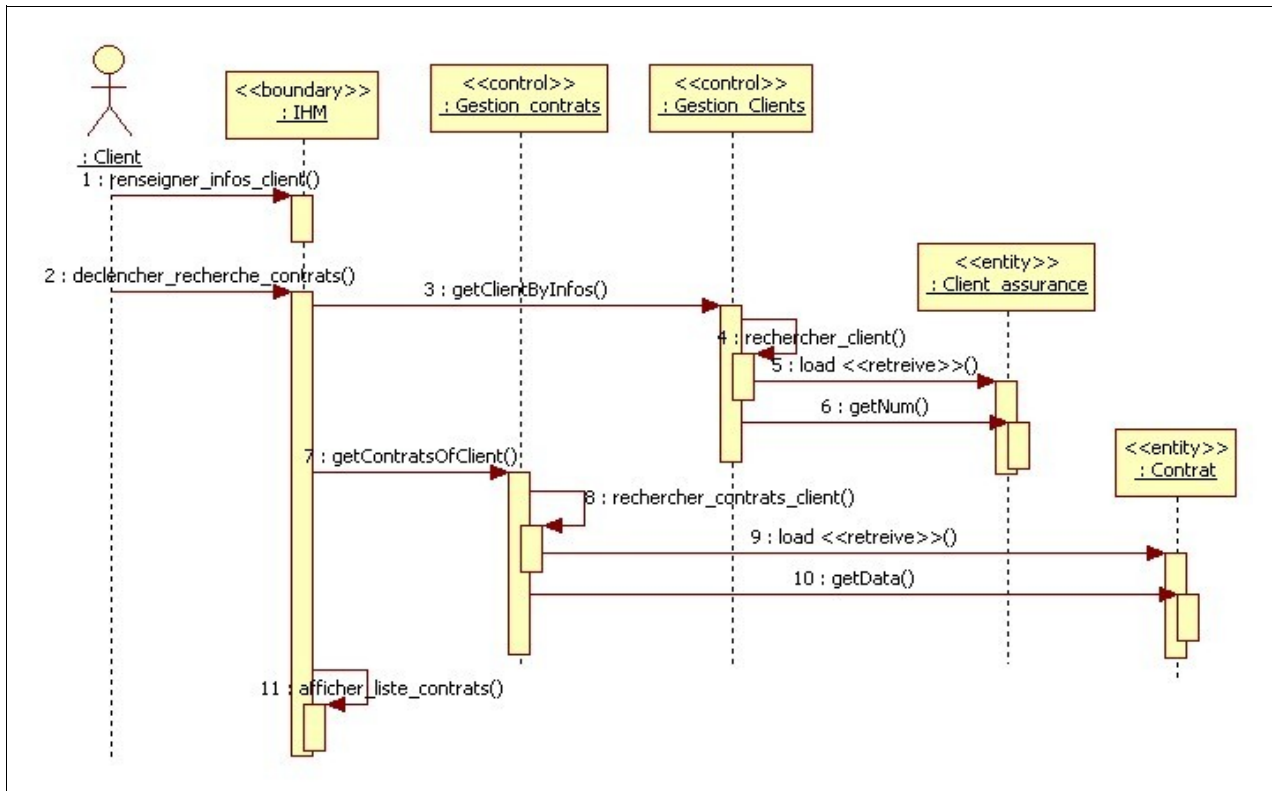
Le principal fil conducteur réside dans la "réalisation des cas d'utilisations" autrement dit dans la retranscription des scénarios des cas d'utilisations en digrammes de séquences.

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

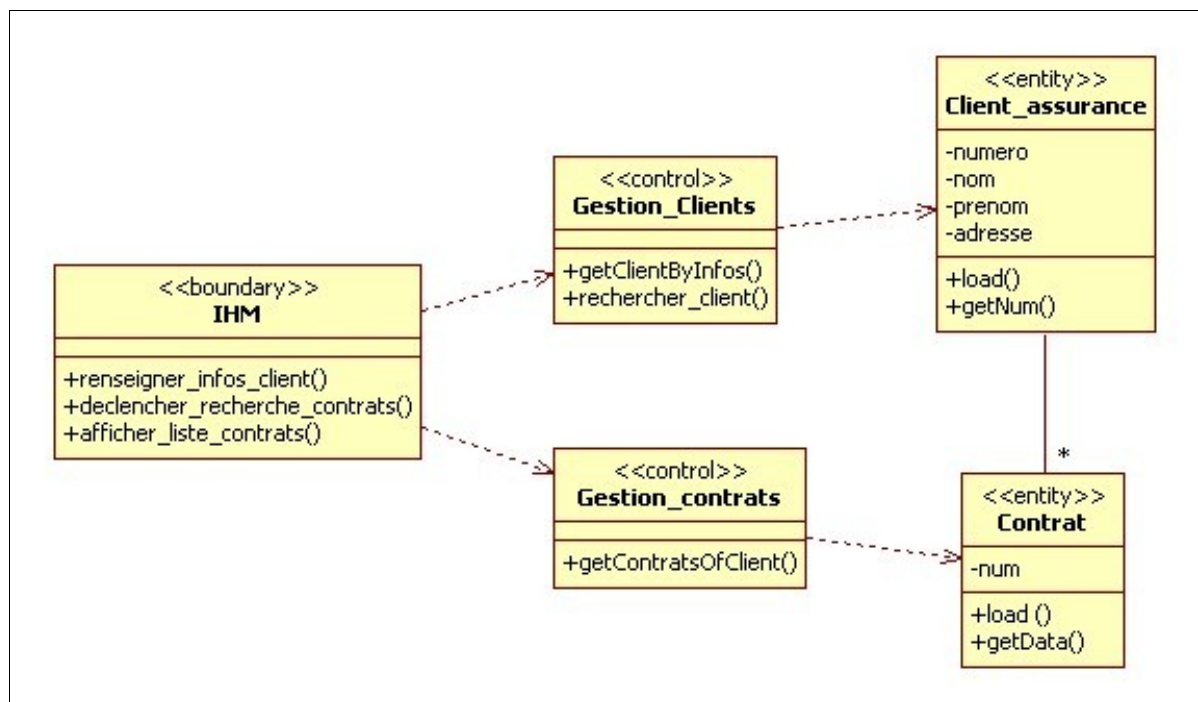
4. Réalisation des cas d'utilisations

Procédure à suivre:

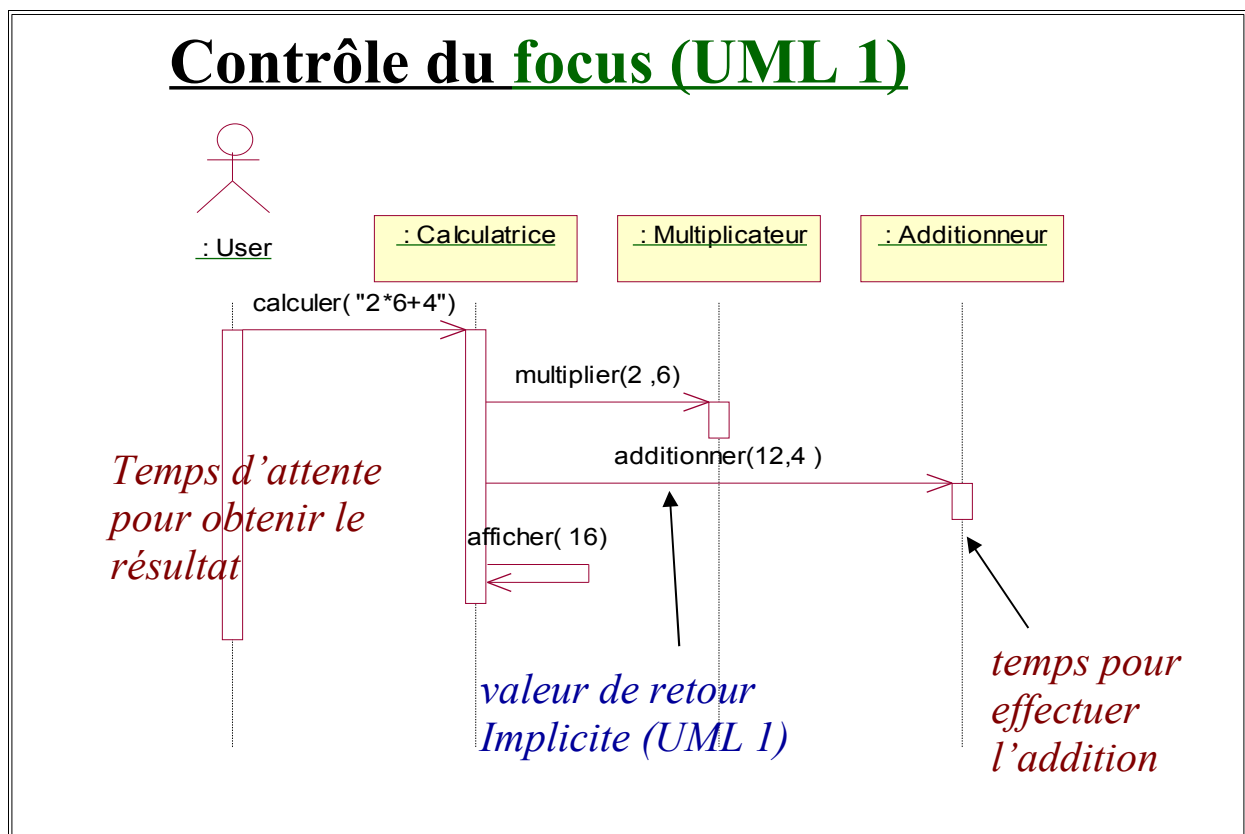
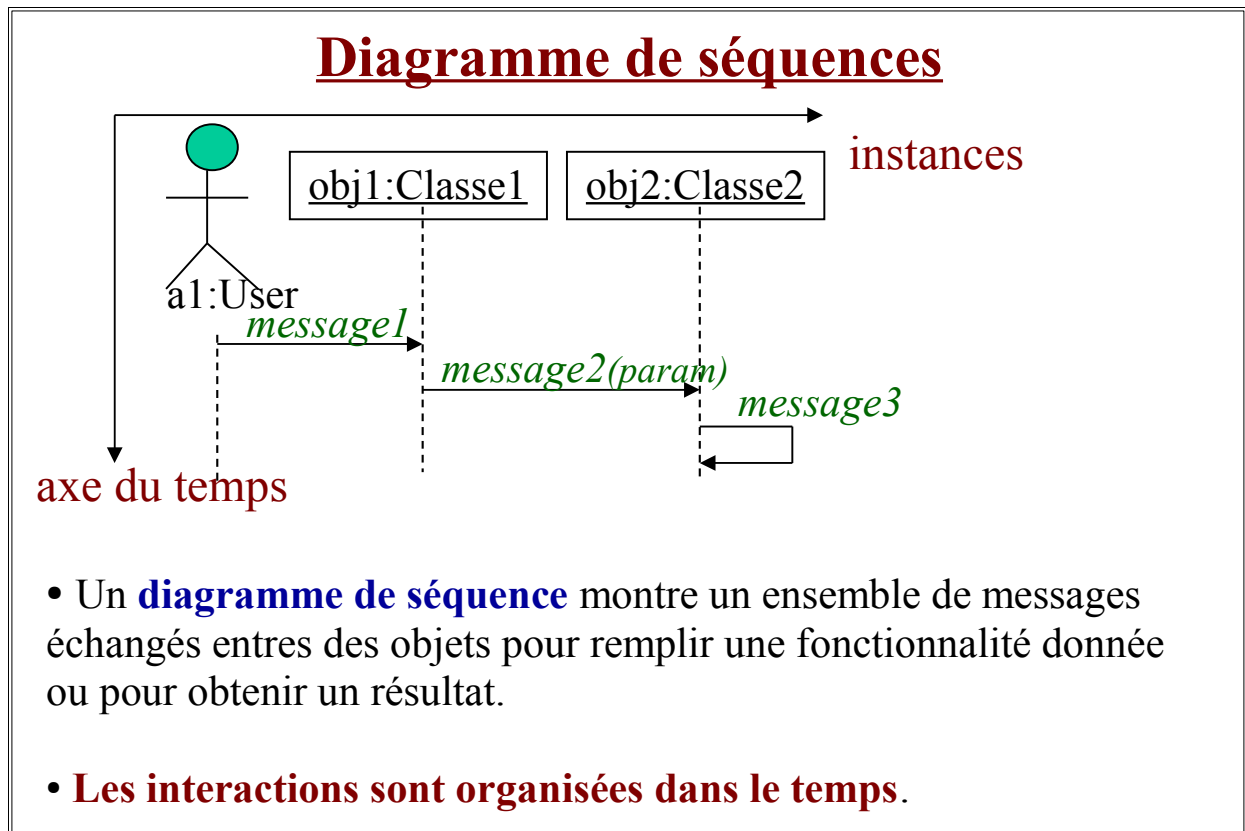
- Retranscrire les scénarios attachés aux U.C. en des diagrammes UML d'interactions (séquence, collaboration/communication, ...).
- Enrichir les diagrammes de classes (nouvelles méthodes = messages reçus)



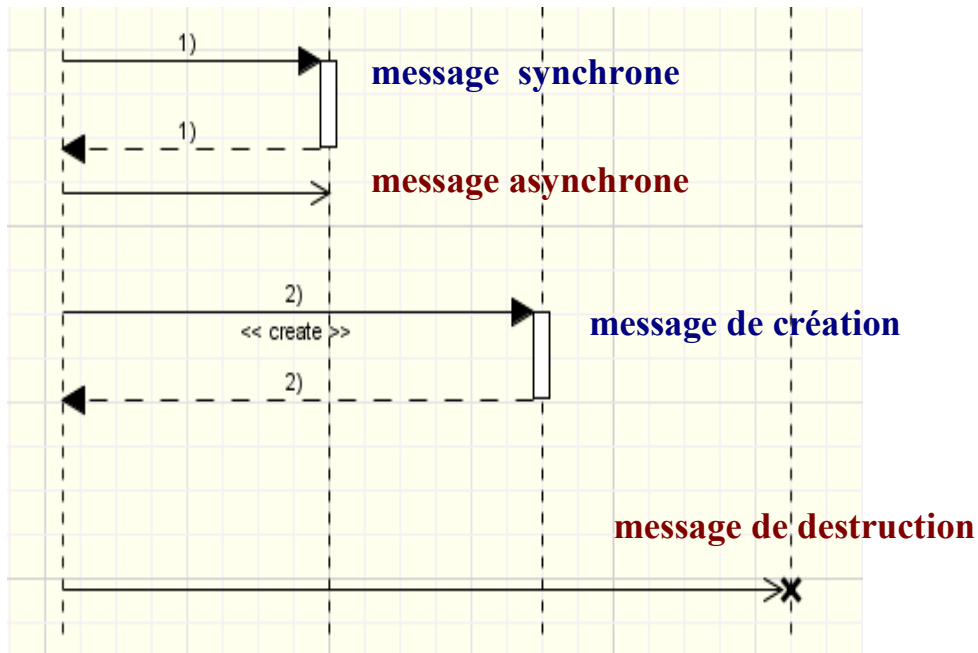
Nb: un service métier peut éventuellement en invoquer un autre.



5. Diagramme de séquences (UML)

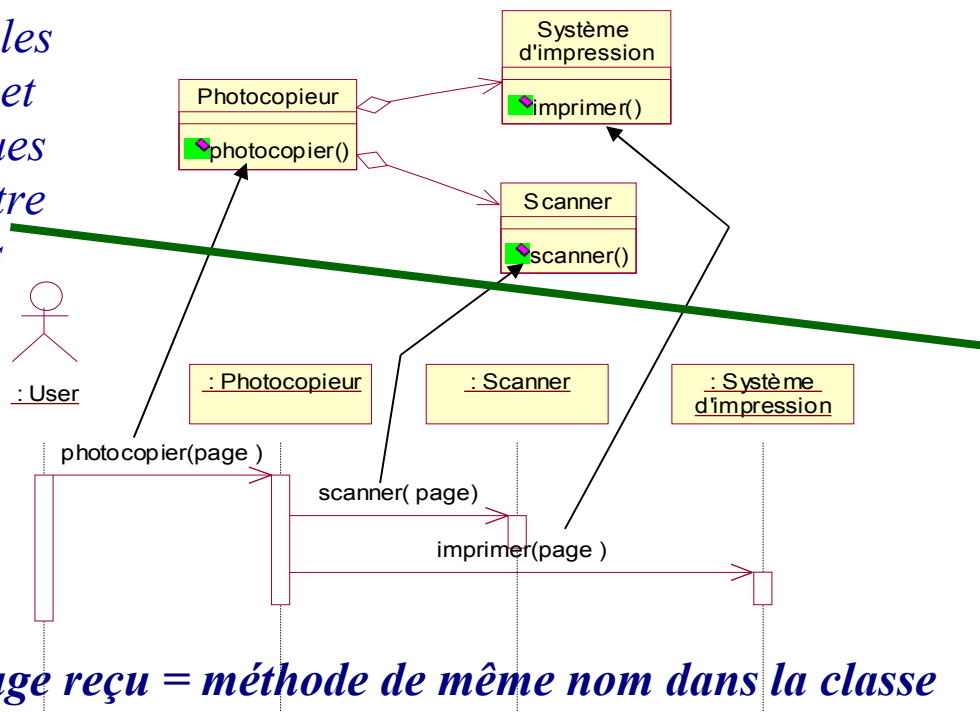


Différents **types** de messages (UML2)



Cohérence entre les digrammes

*Les modèles
statiques et
dynamiques
doivent être
cohérents*



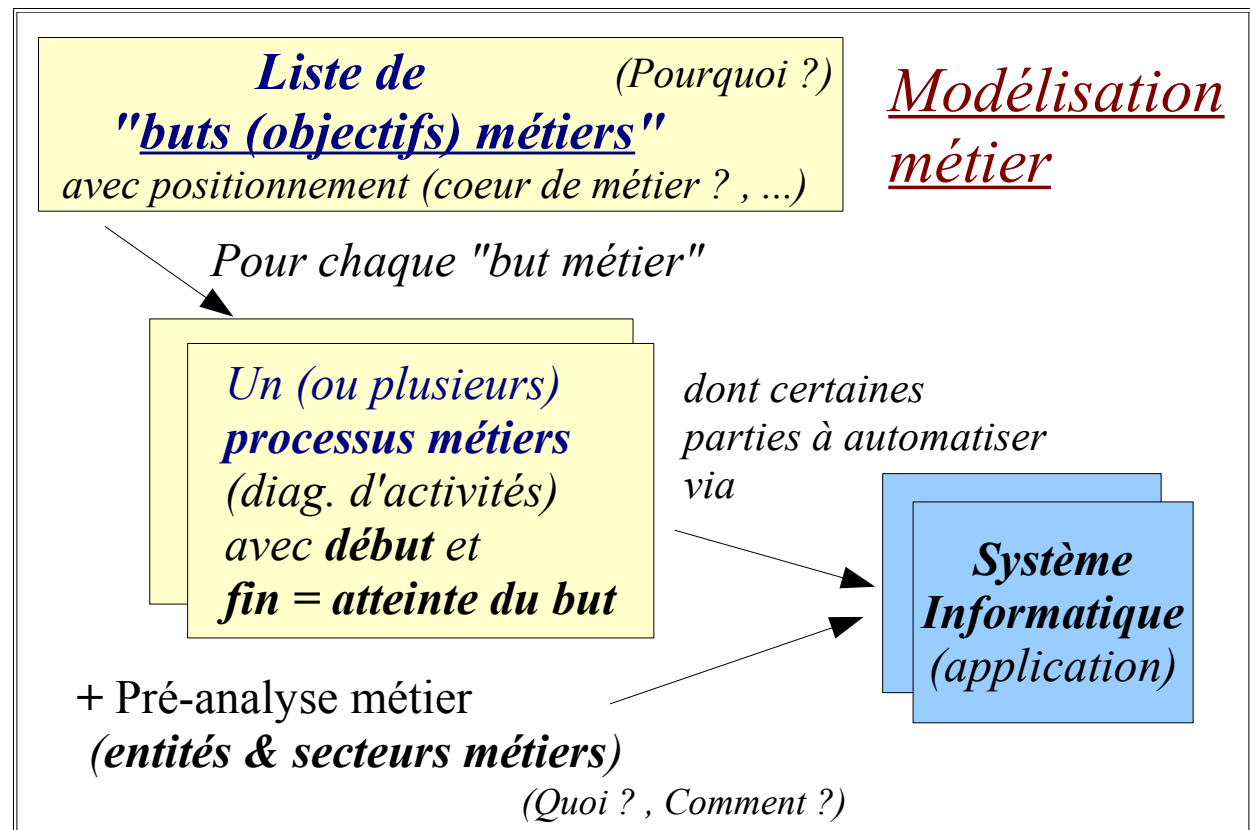
Message reçu = méthode de même nom dans la classe

X - Business Modeling (cadre, spécificités)

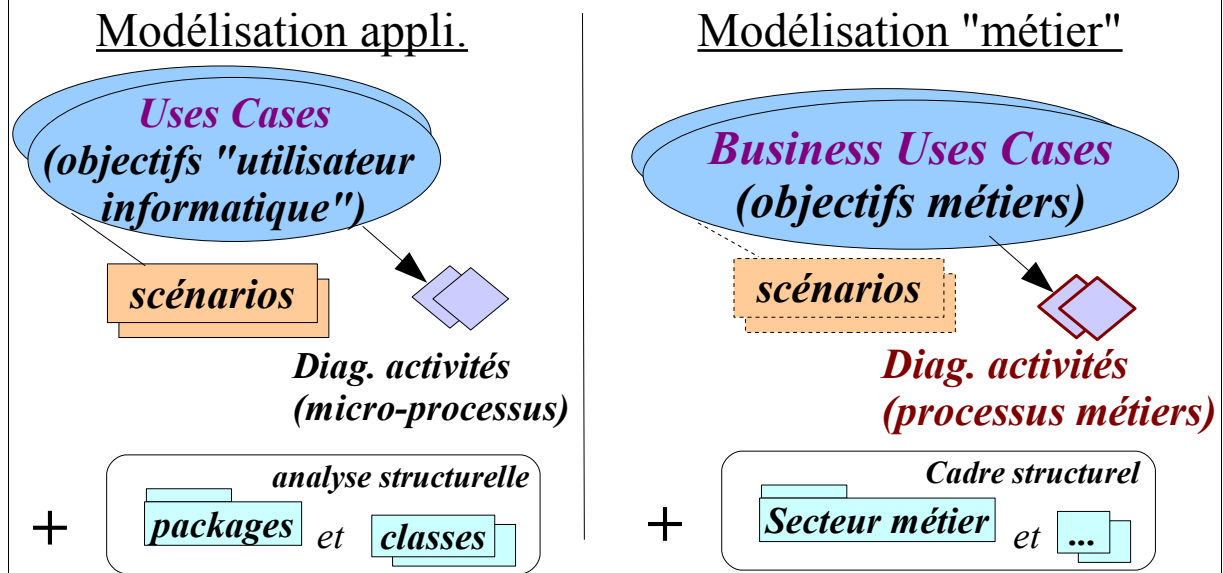
1. Rôle et contenu de la modélisation métier

Modélisation métier (*business modeling*)

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



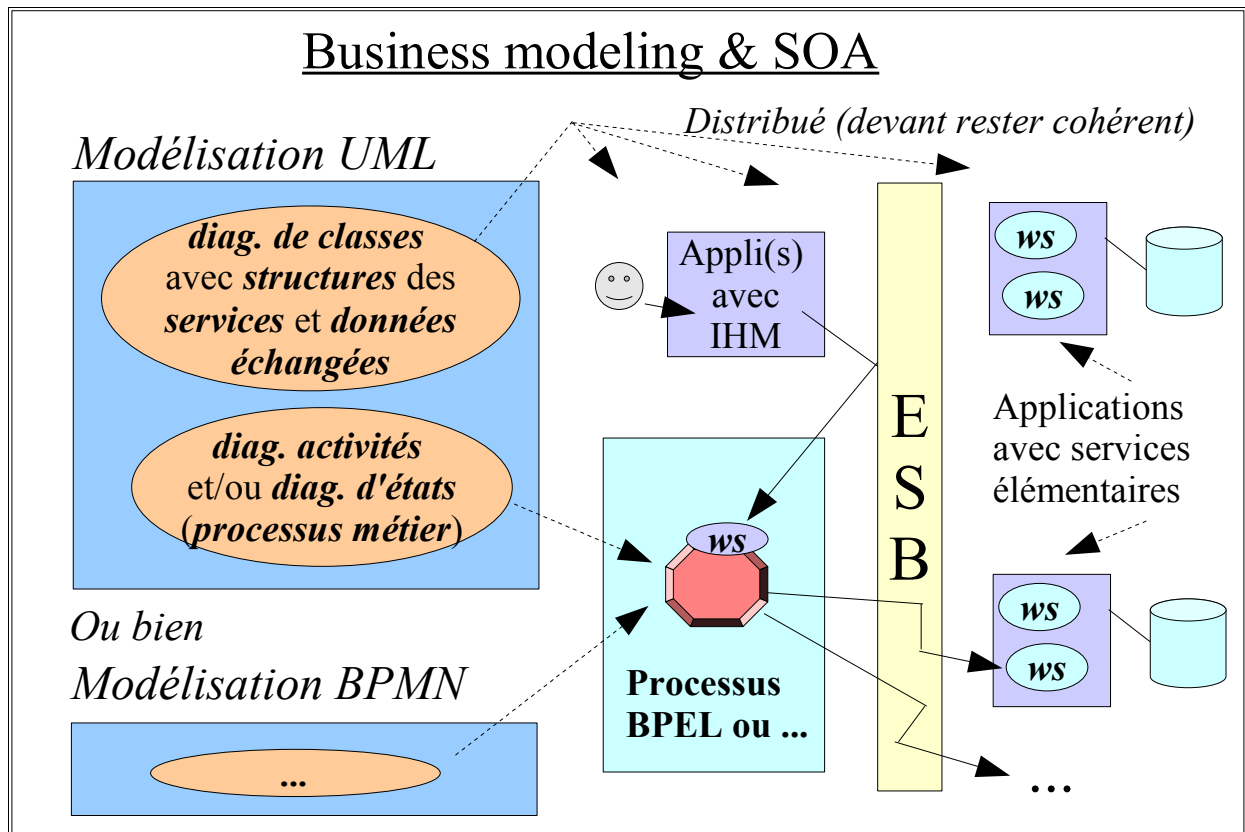
Modélisation métier avec UML à voir comme une ***transposition "métier"*** d'une ***modélisation d'application informatique (cas classique en UML)***



Apprentissage (étapes conseillées)

- 1) Etude rapide d'une appli. simple (ex: conversion de devises)
[en une journée et demie au maximum]
 - 1.a) Cas d'utilisations + scénarios + diag. d'activités
 - 1.b) Concepts objets (présentation , théorie, ...)
 - 1.c) Diagramme de classes (entités + services)
 - 1.d) Réalisation des cas d'utilisations (diag. de séquences)

- 2) Transposition vers modélisation métier (autre étude de cas)
[en une journée et demie environ]
 - 2.a) Business Uses Cases
 - 2.b) Processus métiers (diag. activités , diag. d'états)
 - 2.c) éventuel graphe sémantique (entités métiers)
 - 2.d) identification des services métiers + secteurs métiers
 - 2.e) éventuel diag. de séquences (appels inter-services)



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**.

3. Intérêts de la modélisation métier

3.1. Réfléchir sur le coeur de métier et l'organisationnel

- modéliser pour appréhender , comprendre , communiquer ,
- modéliser pour s'organiser , construire ,
-

3.2. Guider les restructurations/évolutions

- pour les éléments à grosses granularités (« services » , « organisation » , SI global ou ...)
- grands axes
- cartographies des services

3.3. Constituer un référentiel métier et organisationnel

- pour se repérer
- comme base de capitalisation/réutilisation

3.4. Éclairage sur le contexte métier d'un système informatique

- Utilité
- contexte , cadre d'utilisation
- ...

4. Petite méthodologie pour "modélisation métier"

- 1) avoir bien en tête le positionnement métier de l'entreprise (raison d'être , cœur de métier , missions , ...)
- 2) identifier clairement les (nouveaux) "**but**s métiers" (réponses au pourquoi ?) et les représenter dans un diagramme de "**cas d'utilisations** métiers" .
- 3) première ébauche des processus métiers (diagrammes d'activités UML intuitifs / macro) [**processus métier** avec **début** et **fin=atteinte d'un objectif (ou sous objectif) métier**].
- 4) première ébauche de l'analyse métier (graphe sémantique , concepts et entités métiers)
- 5) première ébauche du découpage fonctionnel (modules/packages/secteurs métiers)
- 6) + suite ...

...

...

XI - Contextes ,périmètres et portée

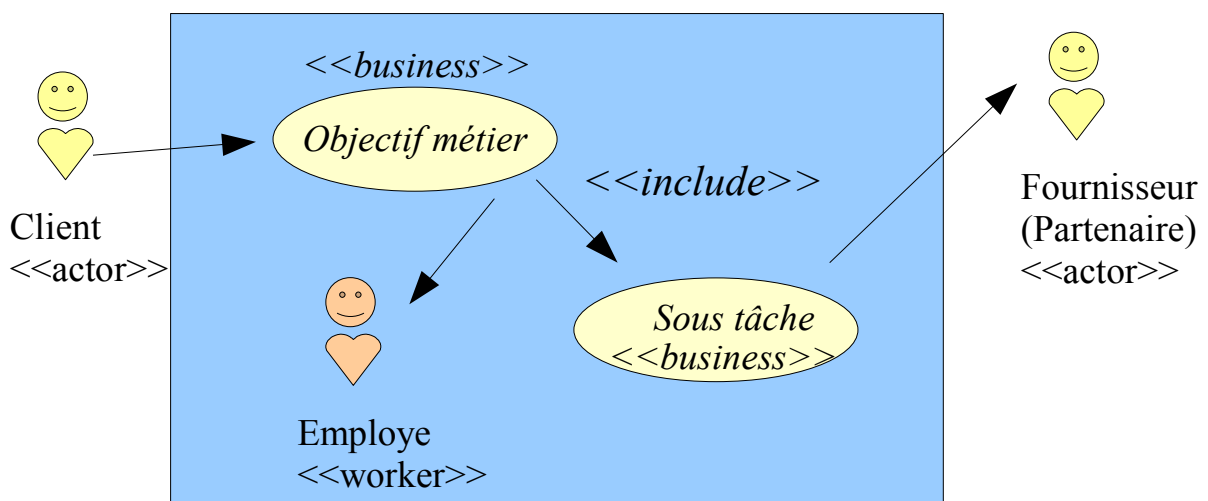
Modélisation métier (portée et systèmes)

IBM dit quelquefois que SOA c'est "programming in the large" .
La portée de la modélisation métier est également potentiellement très grande --> ex: partie d'une entreprise + partenaires externes.

Conséquences:

- * Les entités de la modélisation métier correspondent souvent à des données échangées (ex: commande,devis,facture).
- * Une certaine cohérence (au niveau des format des documents et services) devrait idéalement être respectée (à grande échelle).
- * Il est essentiel de définir au plus tôt le "système" de référence (celui que l'on modélise) et sa portée.
--> exemple: système considéré: "entrepriseXY" ,
partenaires vus comme "acteurs extérieurs"

Portée définie par le diagramme des "business use case"



1. Diagramme de collaboration système(s)

Une fois que l'on a identifié les parties qui sont ou seront informatisées , il est primordial de bien délimiter les différents systèmes (ou sous systèmes) informatiques qui devront collaborer et communiquer entre eux.

Un **diagramme de collaboration/communication système** est tout à fait approprié pour cela

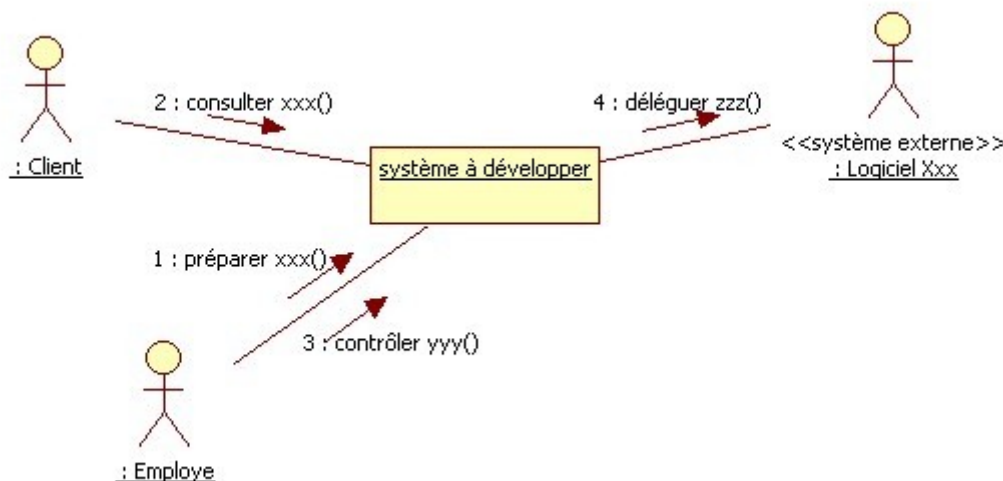
On montre à ce niveau les différents "sous-systèmes" à développer comme une (ou des) boîte(s) noires(s).

2. Diagramme de contexte

Lorsque l'on s'intéresse à un seul sous-système pour l'étudier et le modéliser en profondeur , on parle plutôt en terme de "**diagramme de contexte**" pour désigner le petit diagramme de collaboration/communication (ou de Uses Cases) UML permettant de situer le système à développer dans son futur contexte d'utilisation.

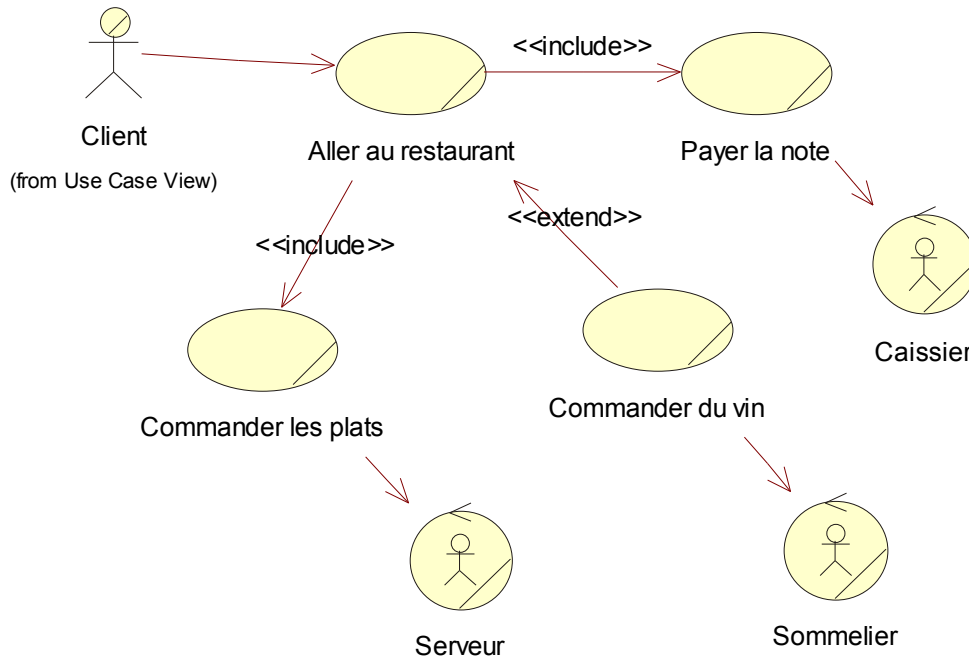
D'éventuels systèmes externes (qui seront sollicités pour déléguer/déclencher des services extérieurs) sont représentés comme des "acteurs UML" avec un stéréotype du genre <<système externe>> ou <<logiciel externe>> ou <<...>>

Les "messages" échangés à ce niveau sont souvent assimilés à des "**interactions systèmes**".



XII - Business Uses Cases

1. 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)**.

2. Intérêts des "Business Uses Cases" au sein de la modélisation métier

- Etant donné qu'un "**Business Use Case**" correspond à un **objectif métier**, ceci constitue le **point de départ** idéal de la modélisation métier. On exprime (dans les grandes lignes) le "**Pourquoi?**".
- Les futurs "processus métiers / diagrammes d'activités" seront ensuite rattachés au "business uses cases" [raison d'être du processus = atteindre l'objectif métier]. Cette considération permet d'organiser l'arborescence des diagrammes au sein du modèle.
- Les "Business Uses Cases" permettent également de préciser la **portée** de la modélisation (en clarifiant ce qui est "interne" ou "externe" par rapport au système considéré).

XIII - Processus métier (diagramme d'activités)

1. Diagramme d'activités

Diagramme d'activités

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

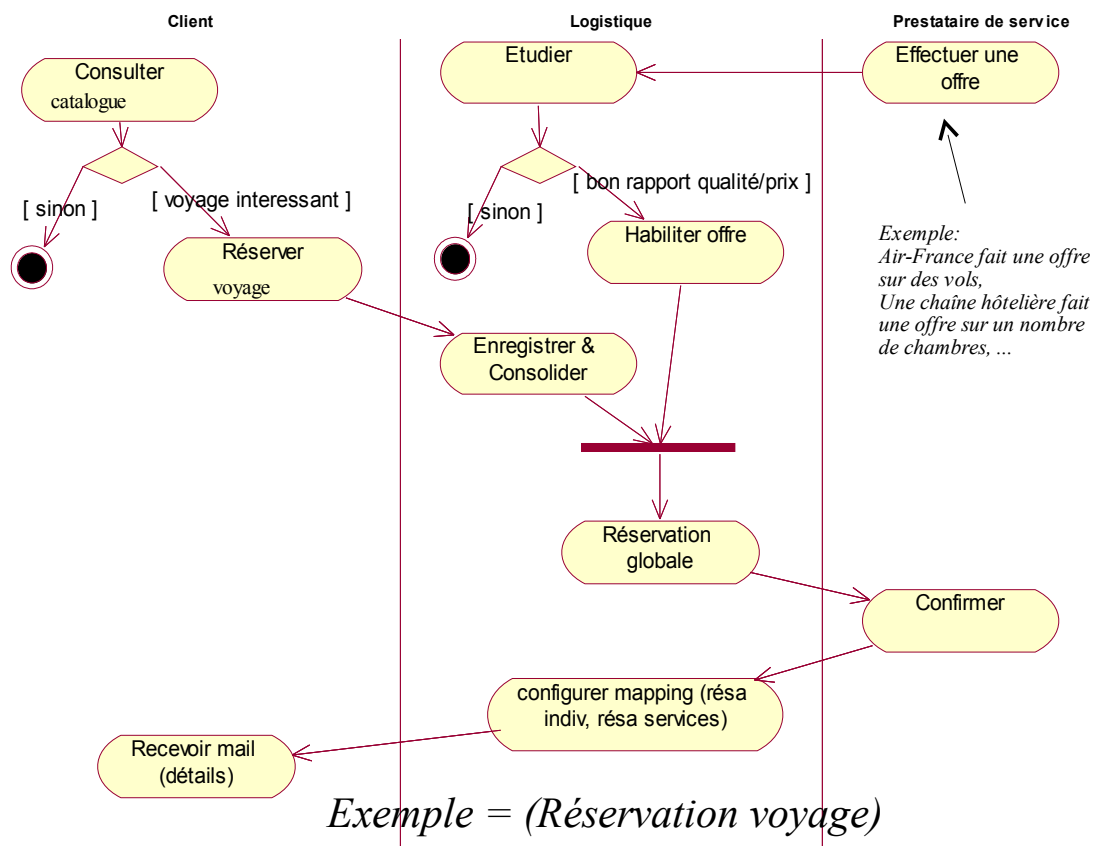
Principales utilisations:

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

1.1. couloirs d'activités

Ces **couloirs** sont quelquefois appelé "**partitions**" ou "swimlane / lignes d'eau" et permettent d'indiquer "**qui fait quoi**".

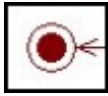
Processus métier avec couloirs d'activités



1.2. Noeuds spéciaux et de contrôles



Initial/début



fin complète(de toutes les branches du processus)



fin de flot/branche

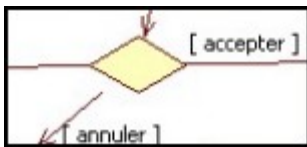


bifurcation (traitement en //)



union/synchronisation de type "et logique"

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



Décision et condition de garde entre []

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

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

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

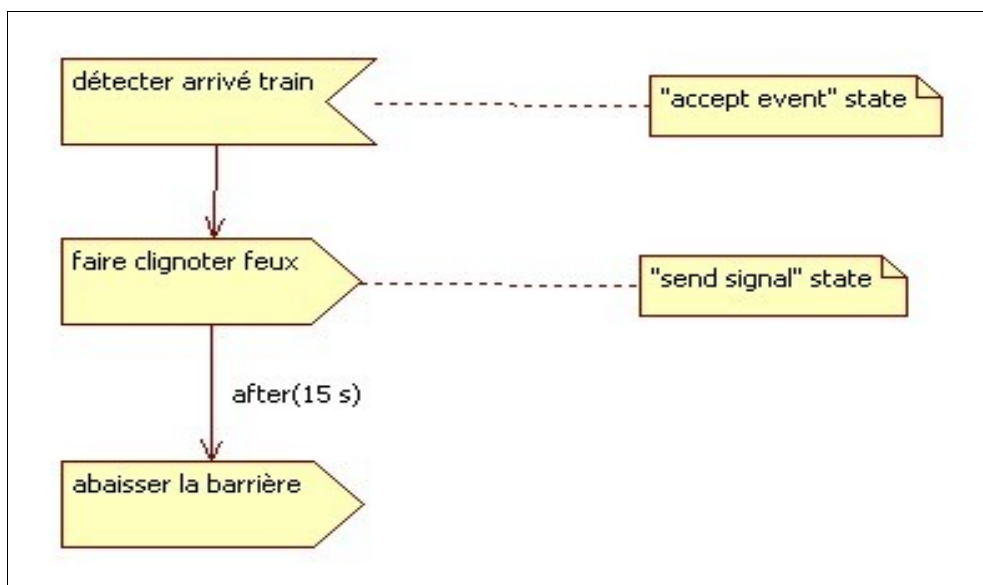
Action = traitement court et élémentaire (quasi atomique).

Activité = traitement potentiellement long qui peut généralement se décomposer en une séquence de sous traitements.

Différents types d'actions:

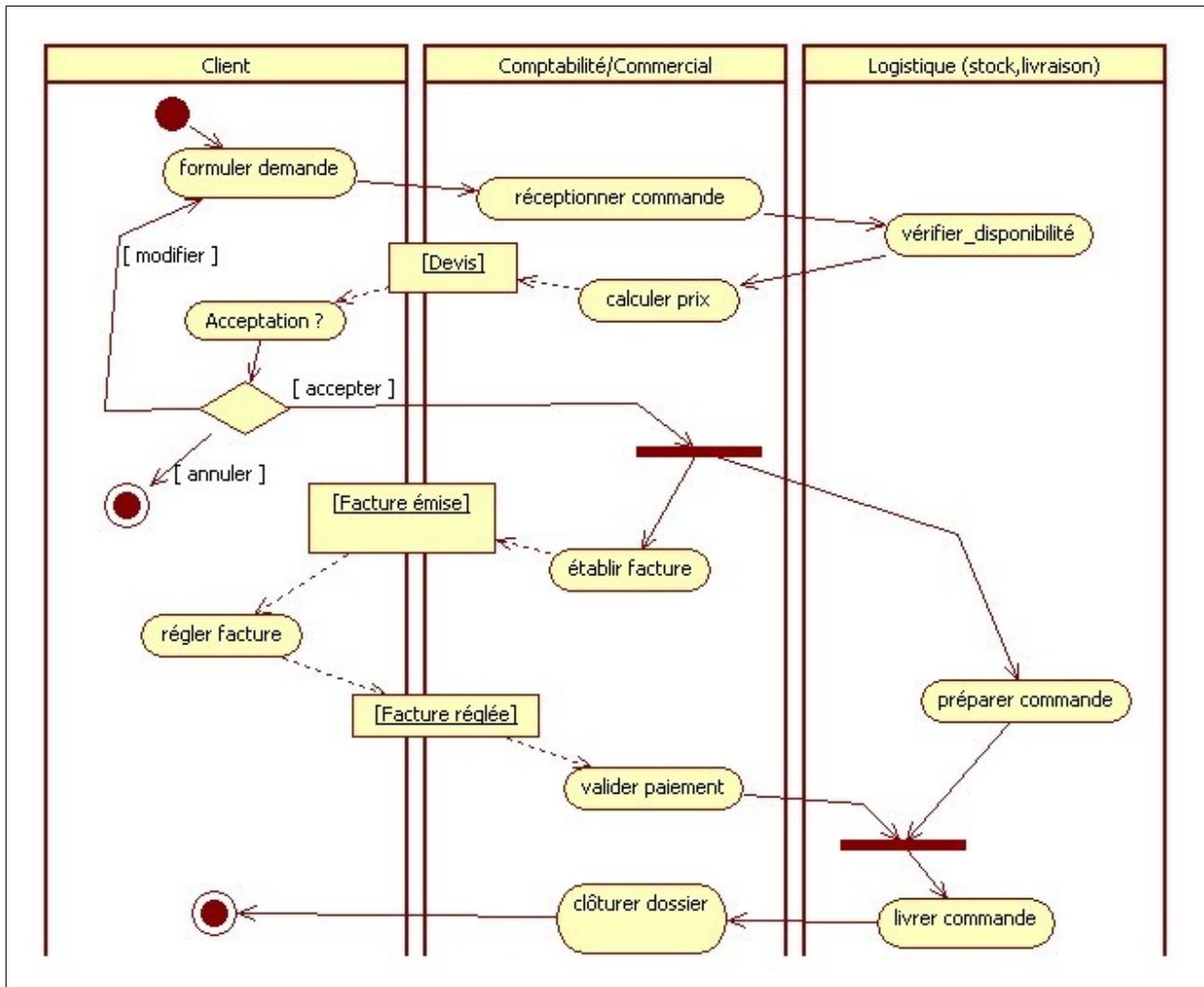
Call operation	Appel (en mode synchrone ou asynchrone) d'une méthode/opération avec passage possible de paramètre et récupération potentielle d'une valeur de retour
Call behaviour	Appel global d'une autre activité <i>[ou autre comportement court ?]</i> (sans mentionner une opération précise)
Send	Envoi 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 synchrones
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

1.4. Notations pour actions particulières (UML2)



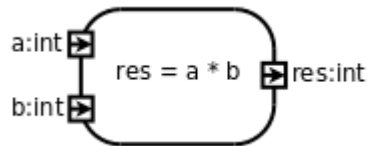
1.5. object flow (nœuds "objet") : UML 1 et 2

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

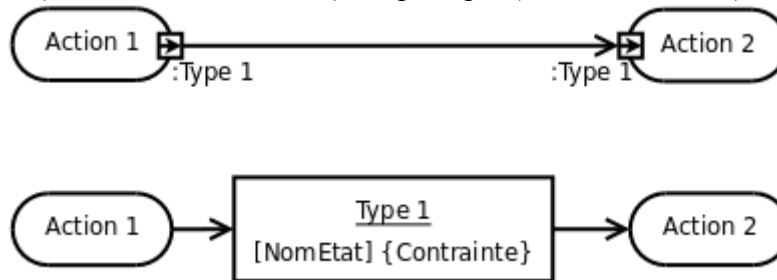


1.6. Pins et buffers (UML 2)

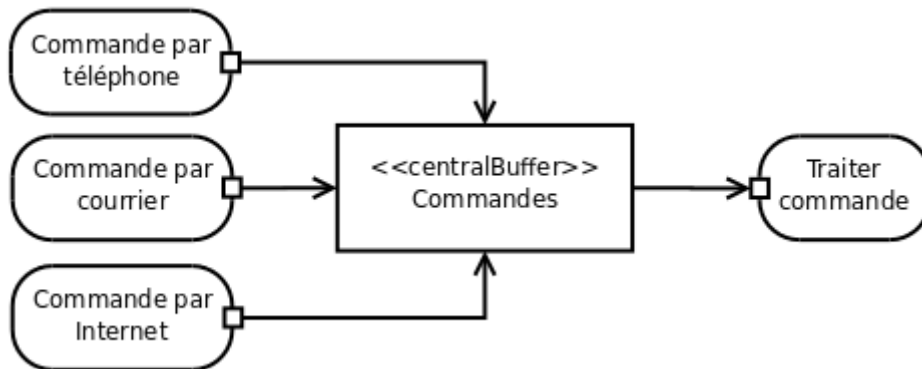
Depuis la version 2 d'UML, il faut placer des points ("pin") d'entrée ou de sortie sur une activité pour préciser un lien (éventuellement typé) avec les "object flow" entrant(s) ou sortant(s).
[avec une sémantique de passage de valeur(s) par copie(s)] .



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



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

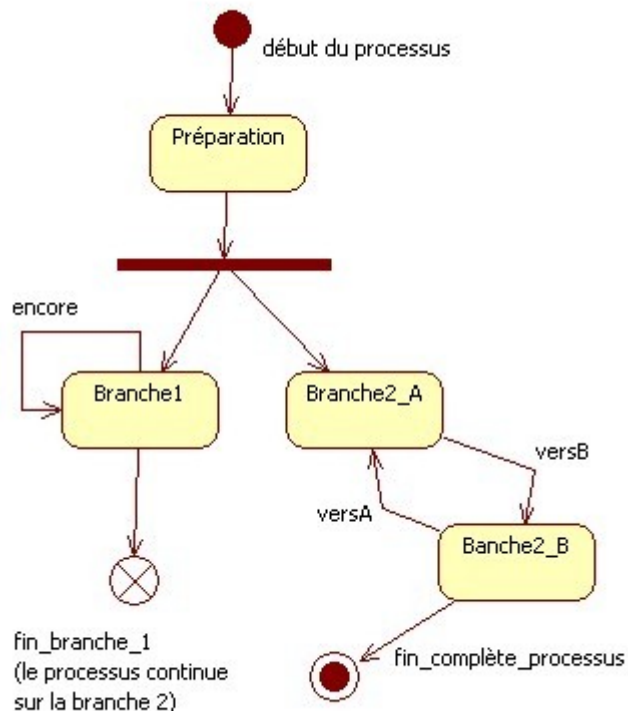


NB: Lorsque le buffer/tampon intermédiaire gère en outre la persistance des données (dans une base de données ou ...) , on utilise alors <<dataStore>> plutôt que <<centralBuffer>> .

1.7. Final flow

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

Exemple :



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

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

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

3. Quelques conseils sur les diagrammes d'activités

La bonne dose de "précisions":

La version 2 d'UML a ajouté tout un tas de nouvelles syntaxes qui:

- apportent quelquefois plein de précisions
- ont malheureusement tendance à surcharger un diagramme et le rendre moins clair (moins lisible).

Il est donc fortement conseillé d'adapter la précision des diagrammes UML à l'usage que l'on souhaite en faire:

- Beaucoup de détails sont nécessaires si l'on souhaite générer du code (ex: BPEL ,).
- Les grandes lignes suffisent amplement pour cogiter sur les processus et exprimer les décisions et différents branchements qui s'en suivent.

Bien structurer les processus (diag. d'activités):

S'efforcer d'avoir pour chaque processus:

- Un début unique (bien identifié)
- Une fin principale claire (atteinte du but , de l'objectif)
- D'éventuelles fins de type "annulation" (mais bien gérées : avec éventuelles opérations de compensations).
- Si possible tenir compte des différents cas (conditions vérifiées ou pas) et exprimer des "ré-essais" ou autres.

XIV - Graphe sémantique et secteurs métiers

1. Modularité recherchée (modules métiers)

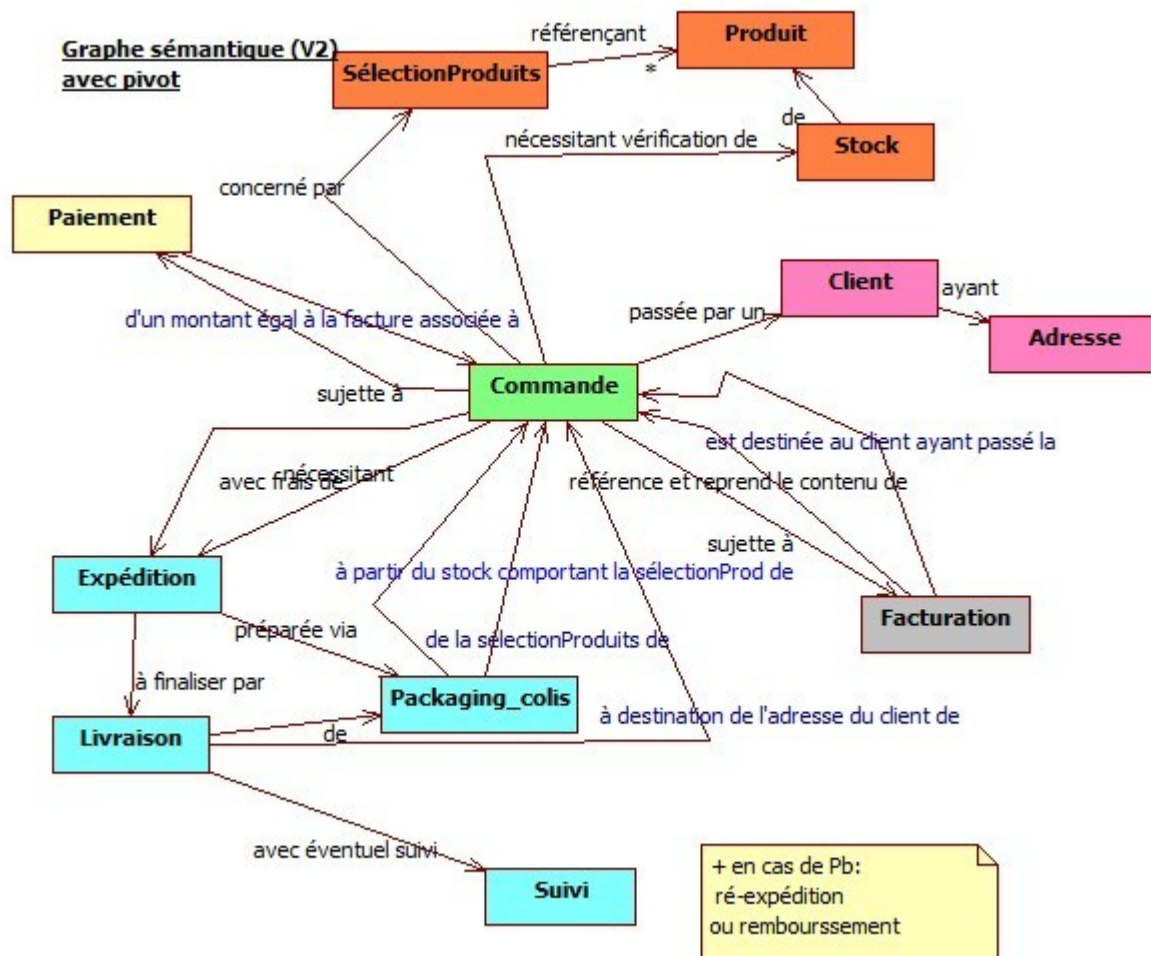
En plus de l'essentielle modélisation des processus métiers, une bonne modélisation métier (orientée objet avec UML) doit absolument se préoccuper d'un bon découpage en "modules métiers" .

Sachant qu'un **module métier** est aujourd'hui assez souvent réalisé comme "**un paquet homogène de services métiers centrés autour d'une entité métier fondamentale**", la réflexion à mener (au niveau du découpage) porte avant tout sur le "pourquoi ?/ quoi ?" que sur le "comment ?" .

Autrement dit , il ne faut s'arrêter sur un découpage superficiel (de niveau "traitements" ou uniquement organisationnel) mais ***il faut effectuer une analyse métier en profondeur pour faire émerger des blocs fonctionnels cohérents*** .

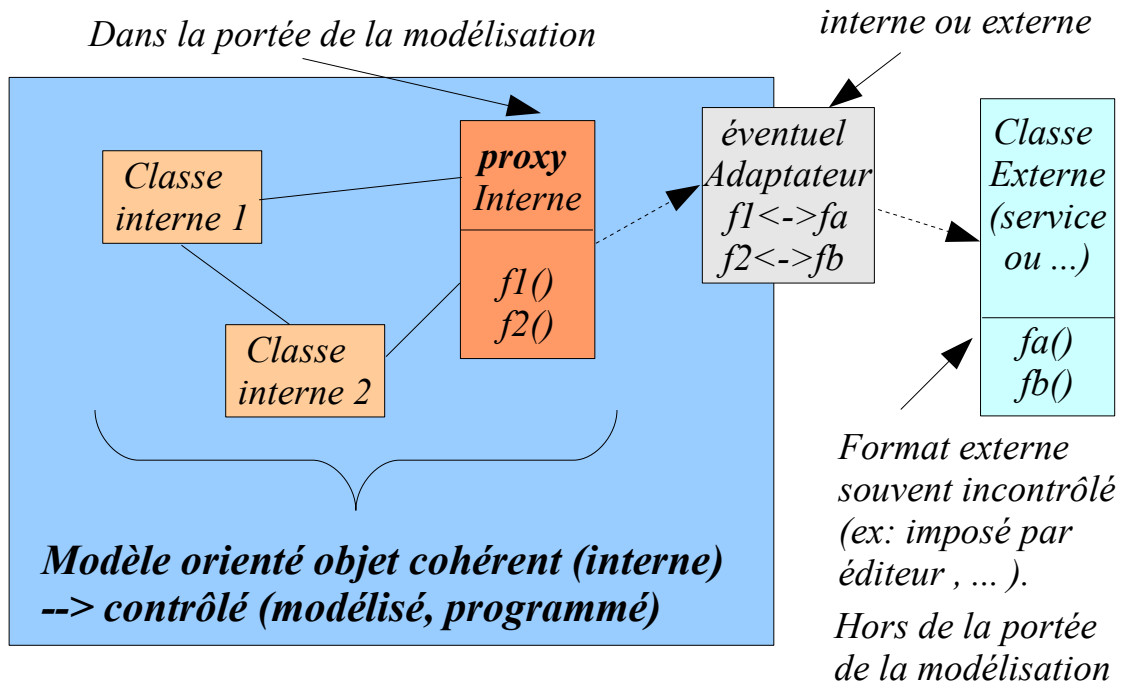
Un "diagramme de classes UML" simplifié (ne montrant pas les attributs ni les méthodes) mais simplement les noms des entités et leurs relations/associations peut s'avérer être très efficace pour réfléchir sur les concepts et les leurs inter-dépendances . On parle assez souvent en terme de **graphe sémantique** pour désigner ce type de diagramme.

Exemple:



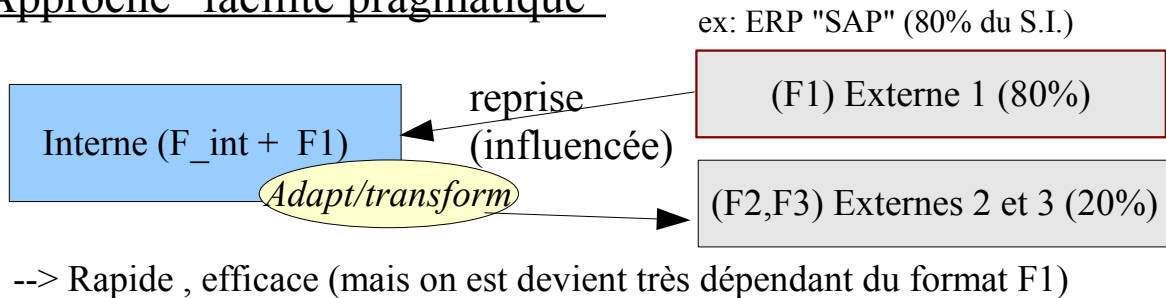
2. Proxy métier/fonctionnel

Notion fondamentale de "proxy métier"

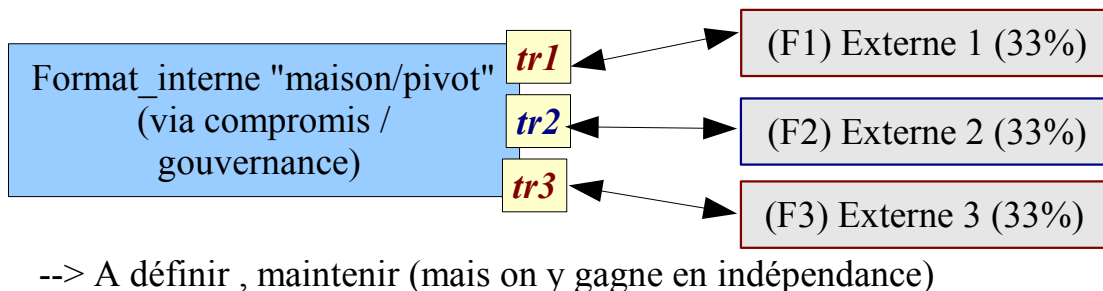


Choix du format interne (*) quelquefois appelé "format pivot" en SOA

Approche "facilité pragmatique"



Approche "gouvernance des données"



3. Diagrammes de classes (de niveau métier)

Même formalisme , logique et démarche que pour une modélisation d'application.

Seule la portée change (en général):

Les **packages** correspondent souvent à des "**secteurs métiers**".

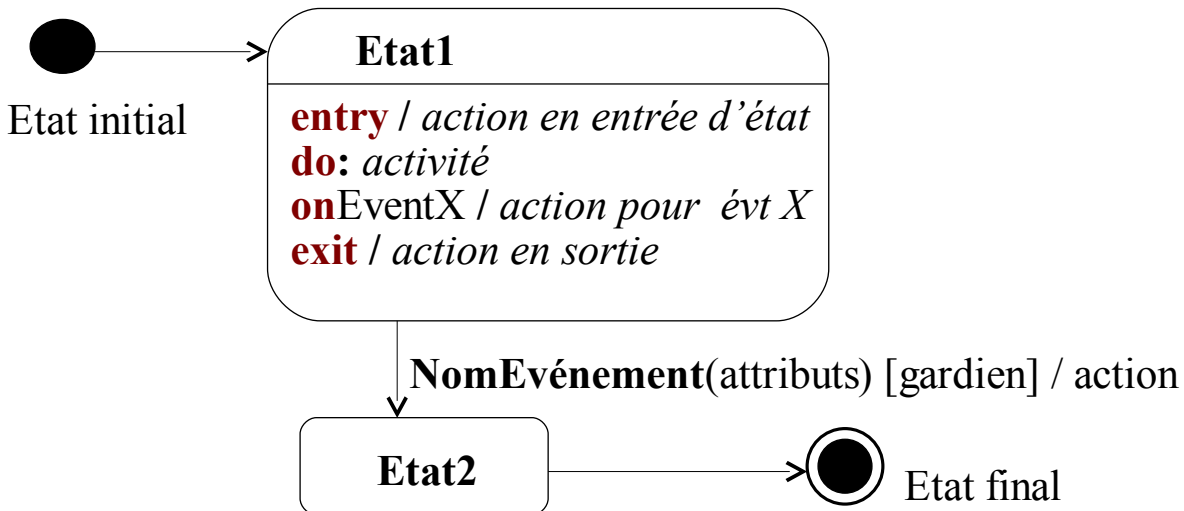
Les classes utiles (à modéliser) se déclinent en 3 catégories complémentaires:

- services métiers (interne : service rendu ou proxy : service externe appelé)
- données échangés (passées en tant qu'arguments des méthodes/opérations des services)
- entités persistante (ex: dossier de suivi , état d'avancement , ...)

XV - Diagrammes d'états

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

Principales notations (graphe d'états)



NB:

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

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

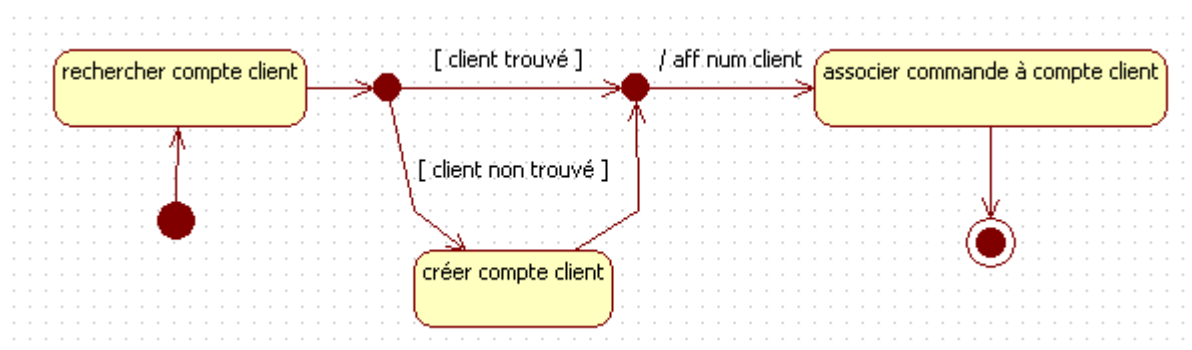
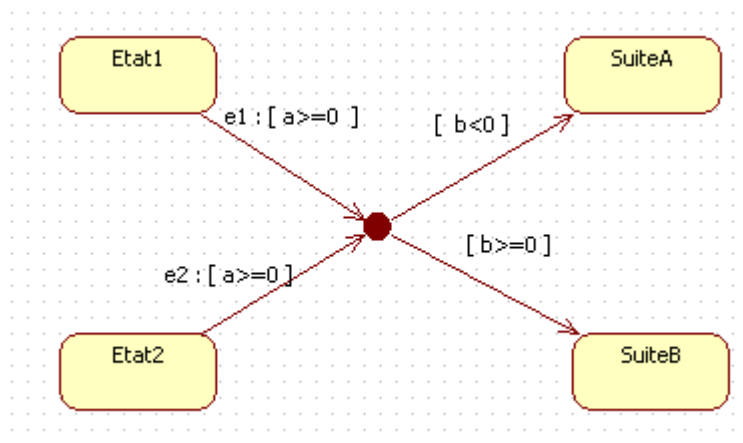
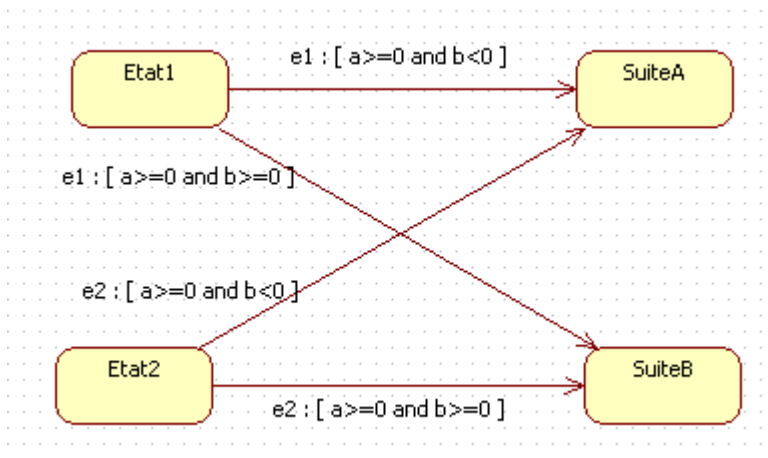
Types (classiques) d'événements

Types d'événements	exemples	sémantiques
Change Event	when (exp_booléenne)	L'expression booléenne devient vraie (après changement)
Signal Event		Signal reçu (sans réponse)
Call Event		Appel reçu
Time Event	after (temporisation)	Période de temps écoulée

Exemple(s):

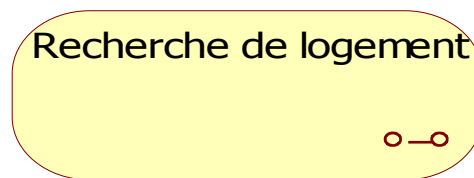
Lumière (avec minuterie) : *allumée* ----- **after(30s)**----> *éteinte*

1.1. Point de jonction

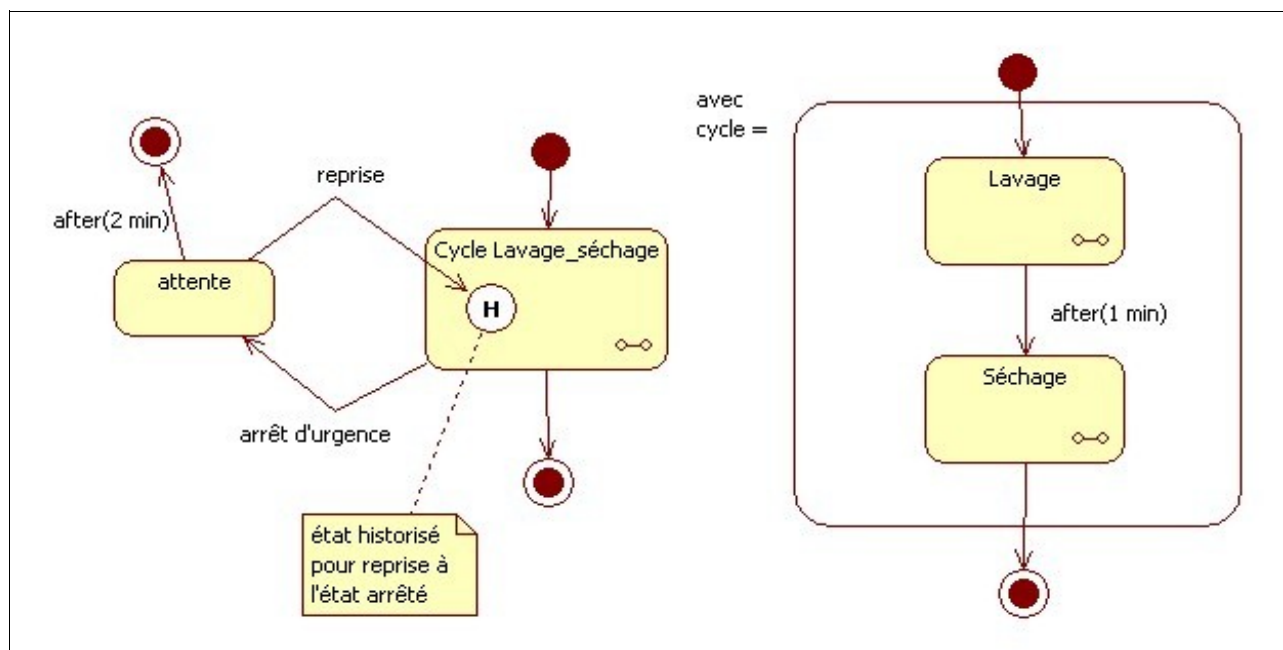


1.2. Etat composite (super état)

Notation abrégée d'un état composite:



1.3. Etats historisés



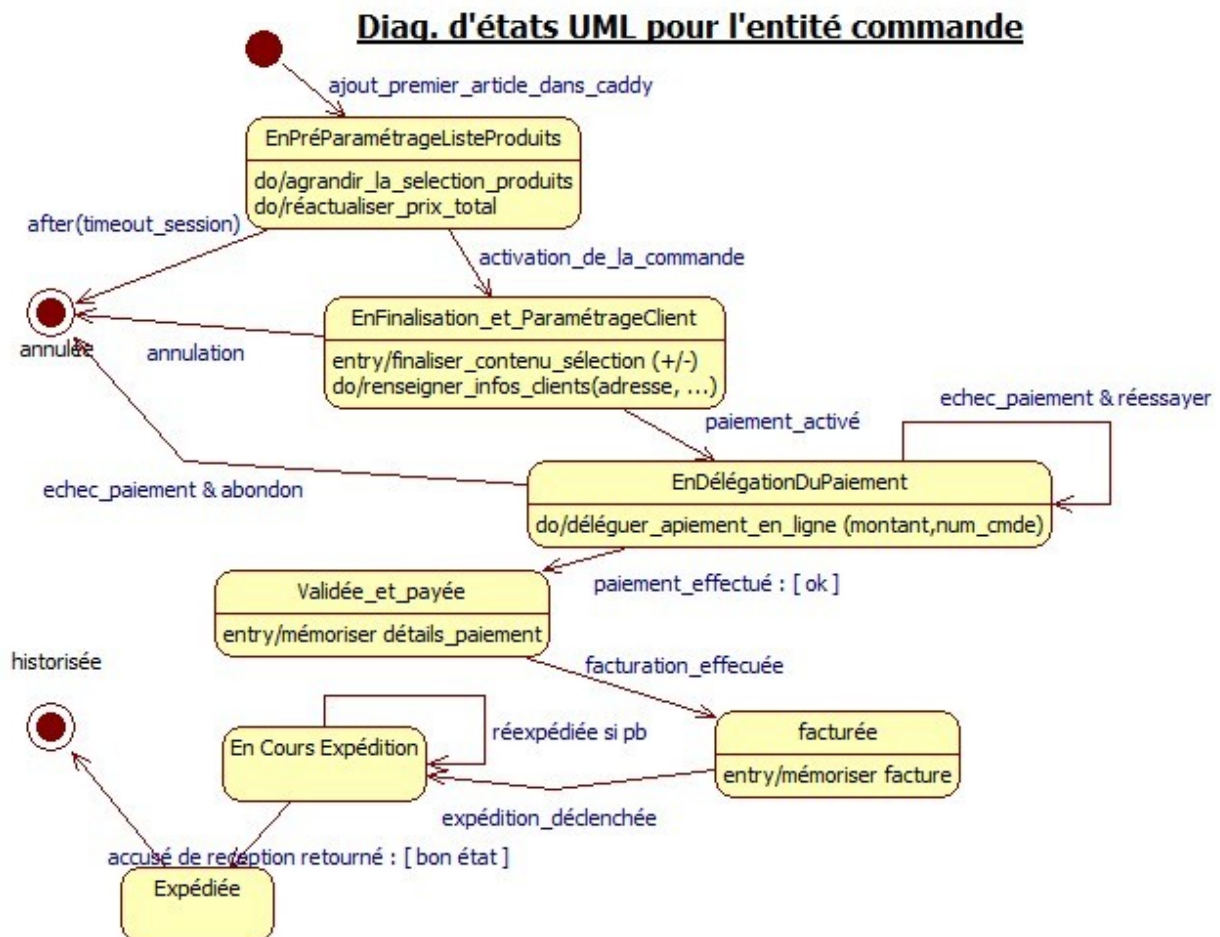
...

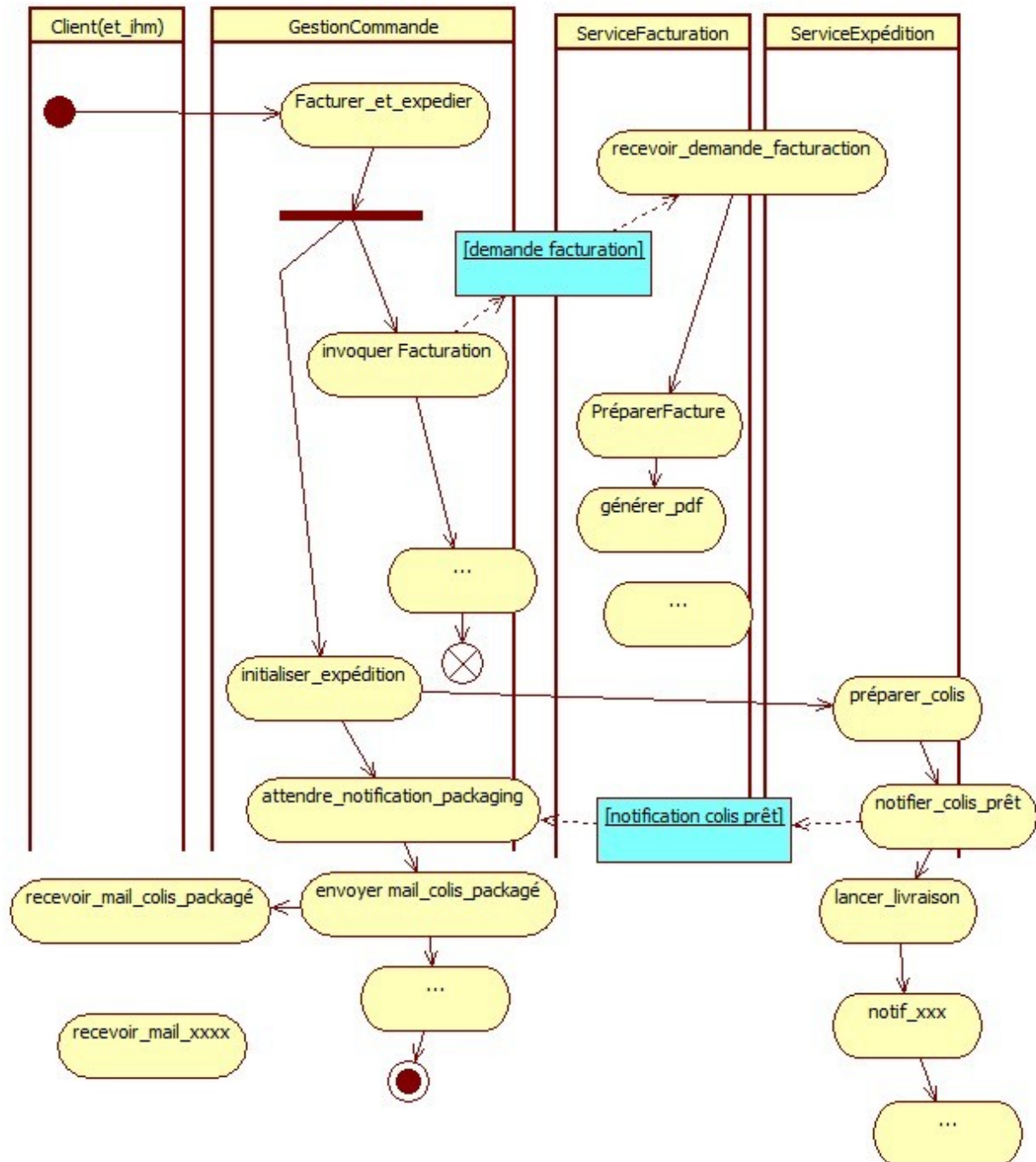
2. Dualité "diagramme d'états / diagramme d'activités"

Les diagrammes UML suivants peuvent éventuellement être appréhendés comme **deux vues inversées d'une même réalité**:

- le **diagramme d'activités UML** montre une **succession logique de traitements/activités** où les entités manipulées sont mentionnées de façon secondaire comme des "object flow".
- le **diagramme d'états UML** montre les **différents "grands états" successifs d'une entité métier importante** où les traitements/activités sont mentionnées de façon secondaire comme des événements extérieurs.

Exemples:





NB: Certaines méthodologies (ex: Praxème) préconisent d'utiliser **préférentiellement des diagrammes d'états UML** car les entités métiers sont souvent plus stables que les activités (sujettes à des ré-organisations).

Cependant, le diagramme d'états UML est moins intuitif que le diagramme d'activités .

XVI - Expr. Besoins IHM (diag d'états , ...)

1. Expression des besoins "IHM"

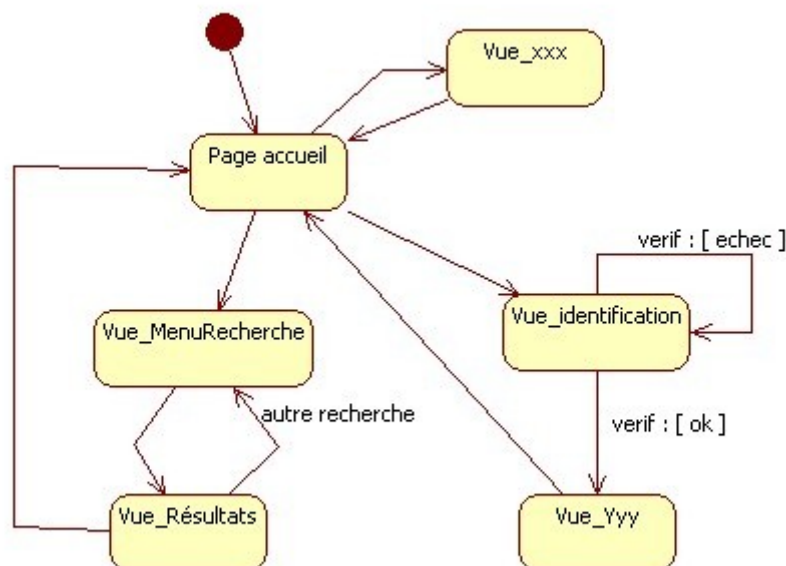
1.1. Maquette

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) .

1.2. Modélisation des enchaînements d'écrans (navigations)



Pertinence :

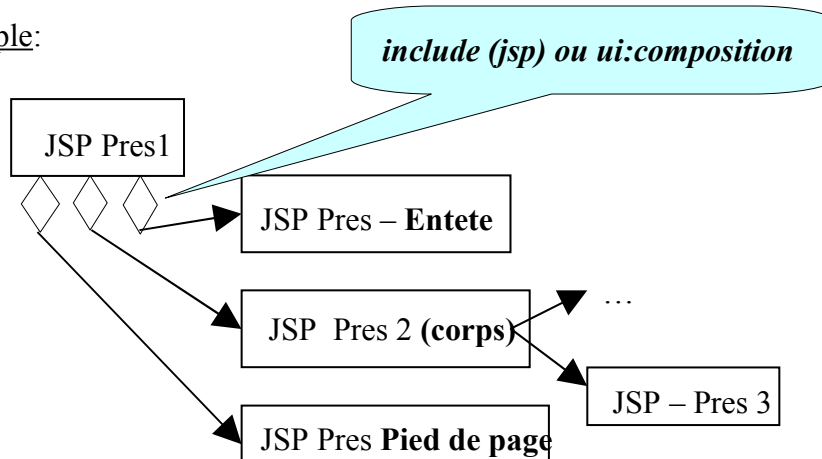
La modélisation des enchaînements d'écran est très importante dans le cas d'une IHM Web (sous forme de pages html générées dynamiquement par code java ou autre) .

Certains frameworks WEB comportent un fichier de configuration qui centralise toutes les navigations possibles (ex: *struts-congif.xml* ou *faces-congif.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 "entete" , "....").

exemple:



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

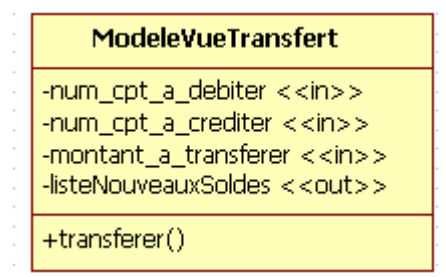
Conséquences :

- 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 est particulièrement intéressant dans le cadre d'une génération automatique de code.

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



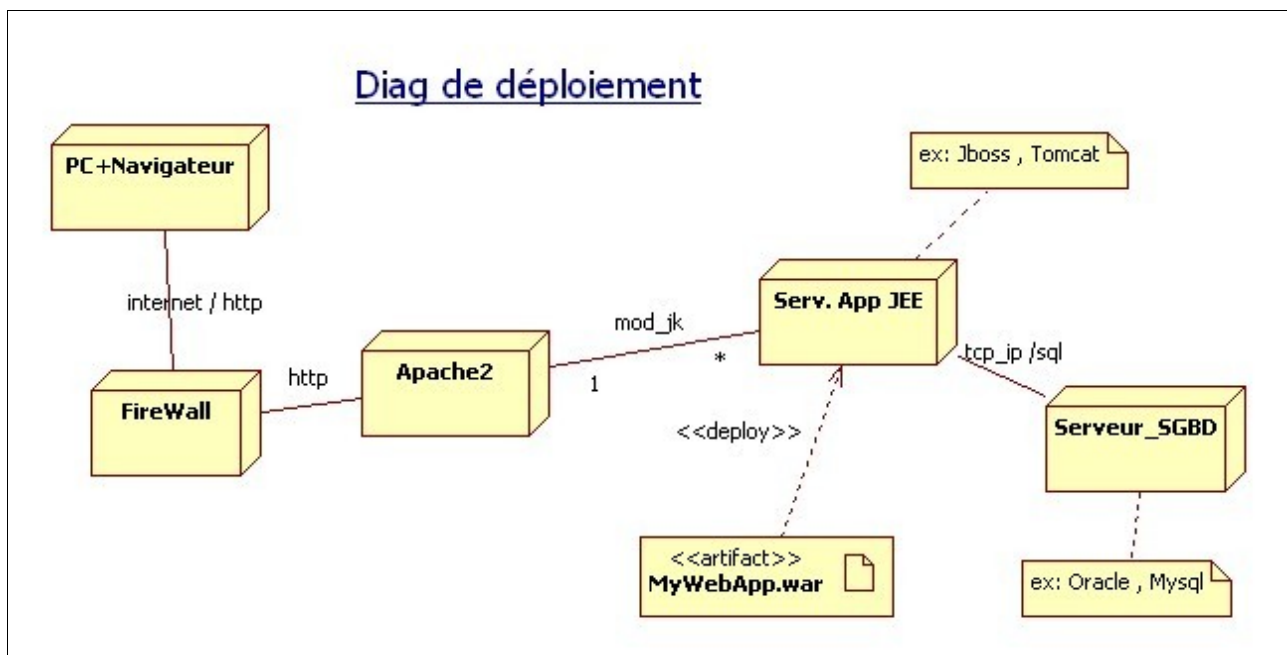
Les stéréotypes <<in>> et <<out>> permettent d'exprimer ce qui sera saisi/sélectionné ou bien affiché

XVII - Besoins techniques / diag. déploiement

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. Besoins techniques à exprimer

- Temps de réponse maximum (ex: 5 secondes)
- Volume de données , sécurité minimale , ...
 ==> c'est important (mais UML n'apporte rien sur ces aspects) ---> formulation libre (en mode texte par exemple)

1.3. Éléments de la conception qui découleront des besoins techniques exprimés

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

XVIII - Aperçu général sur la conception

1. Rôles de la conception

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

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

avec tous les détails nécessaires .

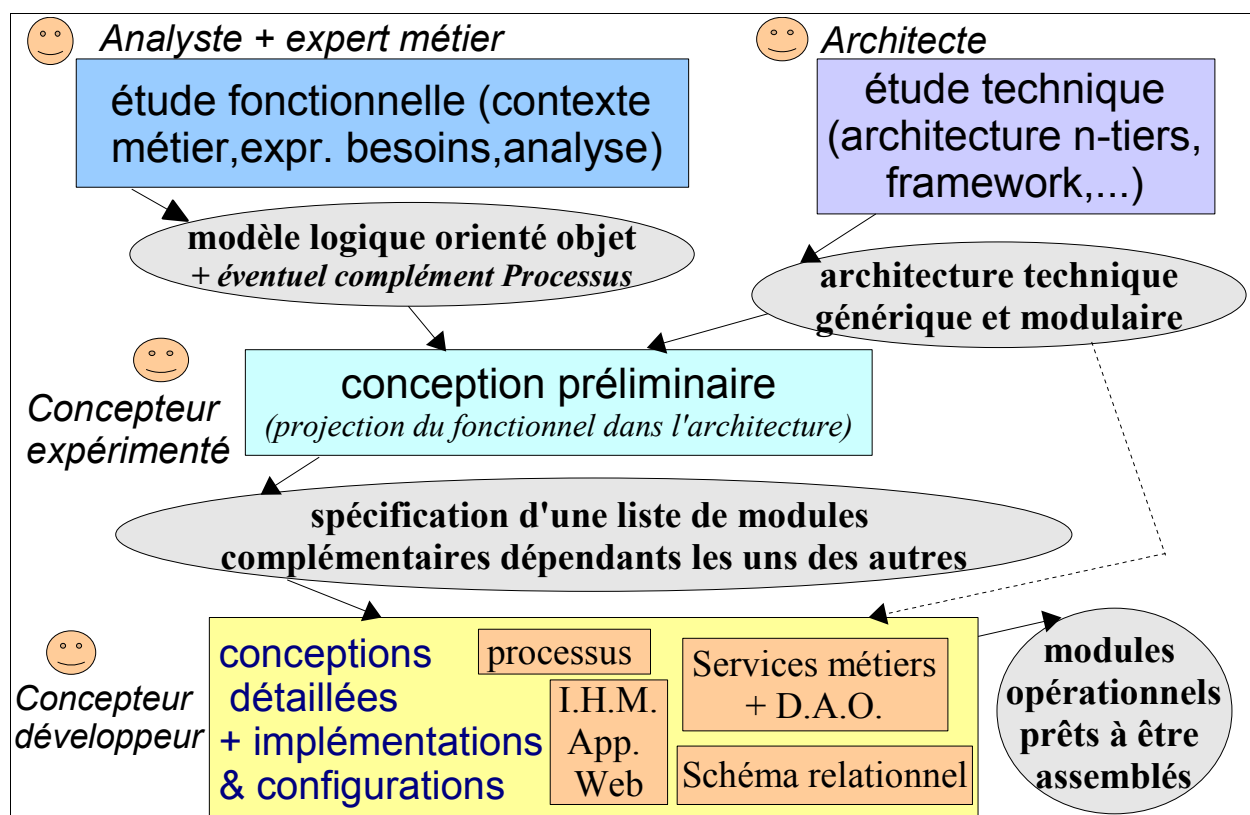
Étymologiquement:

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

pragmatiquement:

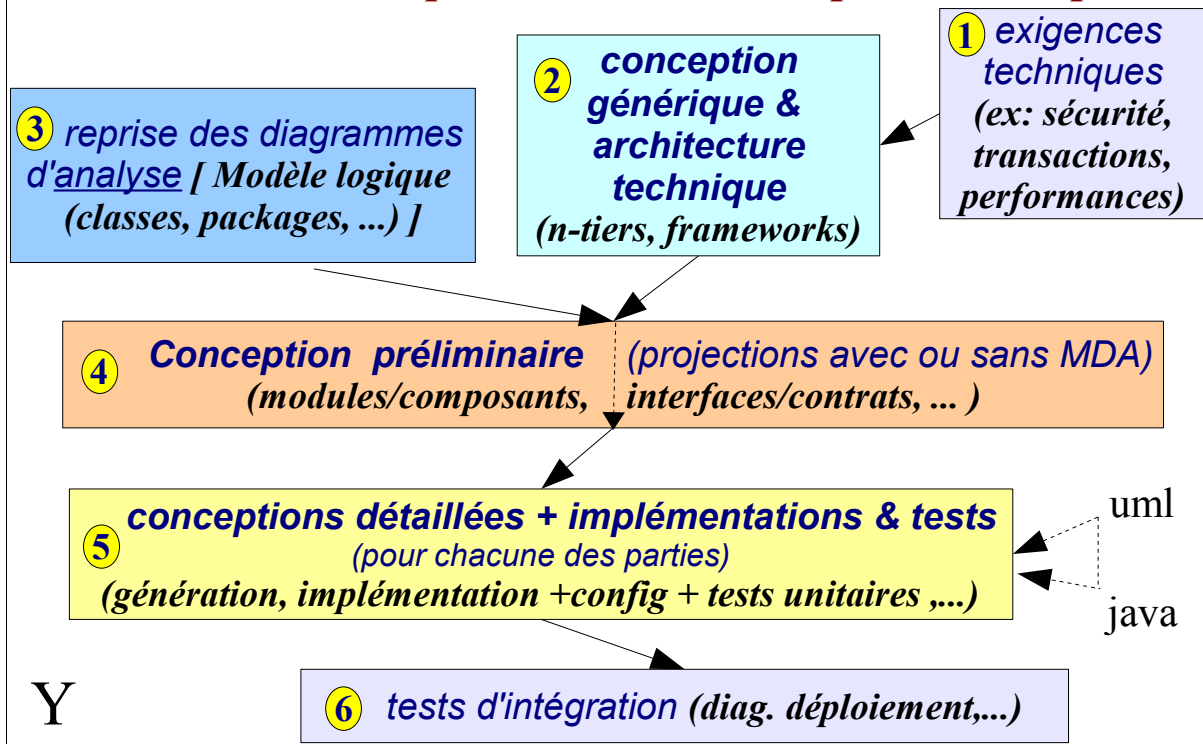
conception ==> très souvent **réutiliser/choisir une solution/technologie (framework)**
[ne pas ré-inventer]

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



3. Activités liées à la conception

Enchaînement classique d'activités sur la partie conception



3.1. conception générique et architecture technique.

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

3.2. Conception préliminaire

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

3.3. conceptions détaillées

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

3.4. implémentation & tests

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

3.5. tests d'intégration

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

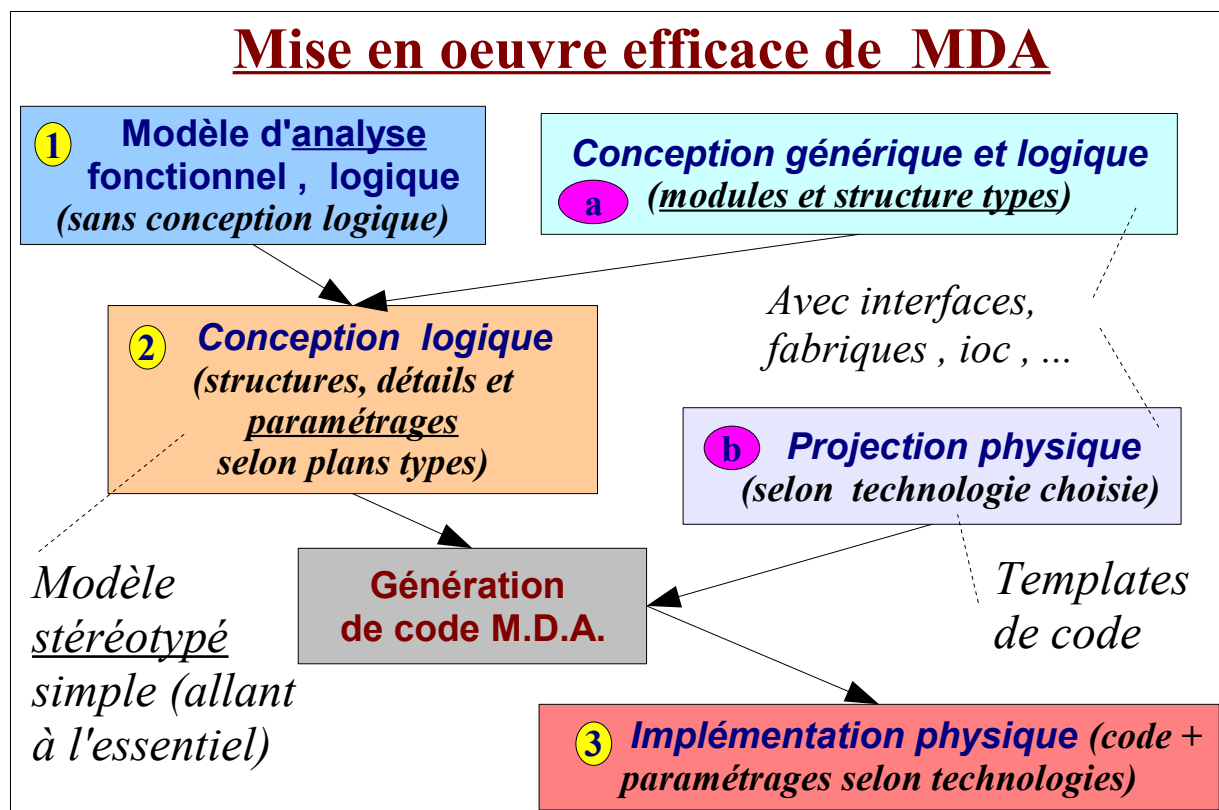
Important:

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.

4. Conception avec MDA:



XIX - Implémentation , retours tests , itérations

1.1. Tests , retours , critiques , itérations

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

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

*==> bonnes et mauvaises critiques ==> nouveau cycle
(modélisation ré-ajustée
si nécessaire ,)*

Bonne pratique !!!

1.2. Annexe intéressante (pour approfondir)

---> Aperçu rapide des méthodologies (UP , XP)

- processus RUP
- processus 2TUP (en Y)
- "XP" et planification qui évolue

ANNEXES

XX - Infrastructure UML (MDA , outils , ...)

1. Infrastructure pour UML, MDA, ...

1.1. Challenge

Comment modéliser et automatiser des processus de développement qui reposent sur des éléments mouvants (architectures et technologies qui évoluent sans cesse,) ?

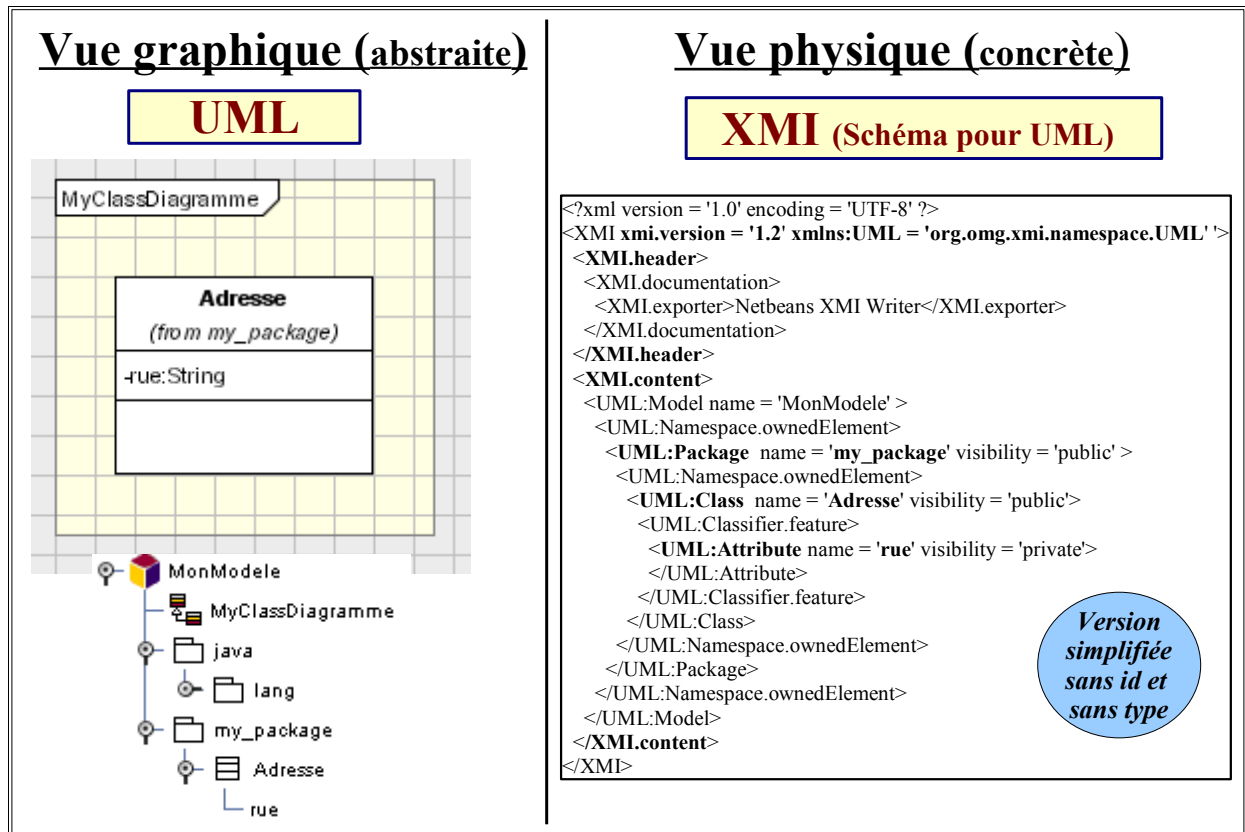
1.2. Principales normes de l'OMG

UML (Unified Modeling Language)	Formalisme (notations + sémantiques) permettant de bâtir divers diagrammes correspondants à différents vues d'une modélisation orientée objet d'un système informatique
M.D.A. (Model Driven Architecture)	Cadre formalisé dans lequel pourra s'inscrire un processus de développement piloté par l'élaboration de divers modèles : CIM : Computation Independant Model ==> <i>niveau conceptuel / expr. besoins</i> PIM : Platform Independant Model ==> <i>modèle d'analyse</i> PSM : Platform Specific Model ==> <i>modèle de conception</i> L'objectif principal de MDA est de pouvoir paramétrer certains passages automatisés d'un modèle à un autre (partiellement).
M.O.F. (Meta Object Facility)	Meta-meta-modèle basé sur l'IDL de CORBA
X.M.I. (XML MetaData Interchange)	Fichiers XML permettant d'échanger des modèles entre divers produits (générateurs de diagrammes, outils de développement java ,)
CORBA / OMA <i>ORB = Object Request Broker</i> <i>(négociateur de requêtes objets) = bus logiciel généralement construit sur TCP/IP et IIOP</i> <i>CORBA = Common ORB Architecture</i>	Architecture objet distribuée basée sur un bus logiciel (l'ORB) et sur différents services annexes (nommage: COSNaming,).
CWM (Common Warehouse Metamodel)	Meta-(méta-) modèle commun pour les entrepôts de données et référentiels (repository avec stockage et récupération d'éléments d'une modélisation UML)
SPEM (Software Process Engineering MetaModel)	Meta modèle pour processus de modélisation (U.P. ,)
xxx Profile for UML (avec xxx = XML , Real-Time ,)	Extension standardisée d'UML vis à vis d'un certain domaine (schéma XML , temps-réel , ...)

---> Combinaison de technologies pour offrir une infrastructure solide pour des **AGL** (Atelier de génie logiciel).

2. XMI

Le **XML Metadata Interchange (XMI)** est un standard de l'**OMG** pour échanger les metadonnées via [XML](#). Il peut être utilisé pour toutes metadonnées dont le metamodel peut-être exprimé en [Meta-Object Facility \(MOF\)](#). L'usage le plus commun de XMI est l'échange de modèles [UML](#).



2.1. Combinaisons éventuelles d'outils complémentaires

Outils UML généraliste (pour expression des besoins , analyse , ...)

====> export XMI

----- ==> import XMI

vers plugin UML pour IDE (eclipse, ...)
et/ou vers ou outils MDA (AndroMDA , accéleo, ...)

2.2. Limitations de xmi et des imports/exports

Les opérations d'import/export (au format XMI) entre différents outils UML sont en pratique rendues délicates pour les raisons suivantes:

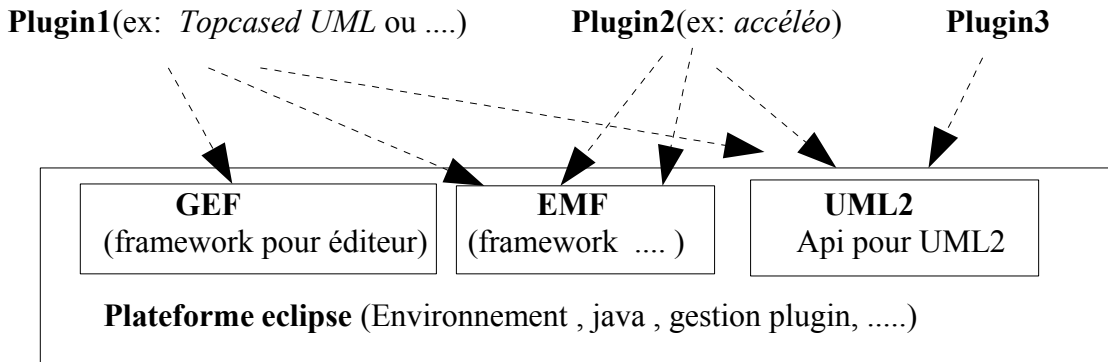
- différentes versions d'UML (UML 1.4 , UML2.0 , UML2.1)
- différentes versions d'XMI (xmi 1.1 , xmi 1.2 , xmi 2)
- différents moyens de stocker les coordonnées des éléments des diagrammes UML (dans fichier xmi , dans fichier annexe , ...)

====> Entre des outils compléments différents (de différents éditeurs ou de différentes époques) , les opérations d'import/export sont souvent partielles (on ne récupère qu'une partie des modèles) ou quelquefois carrément inopérantes (en cas d'incompatibilité).

Bonne nouvelle: les versions récentes des spécifications UML et XMI sont de plus en plus précises et les imports/exports sont ainsi assez bien gérés avec des outils très récents. Il y a tout de même des petites pertes d'informations.

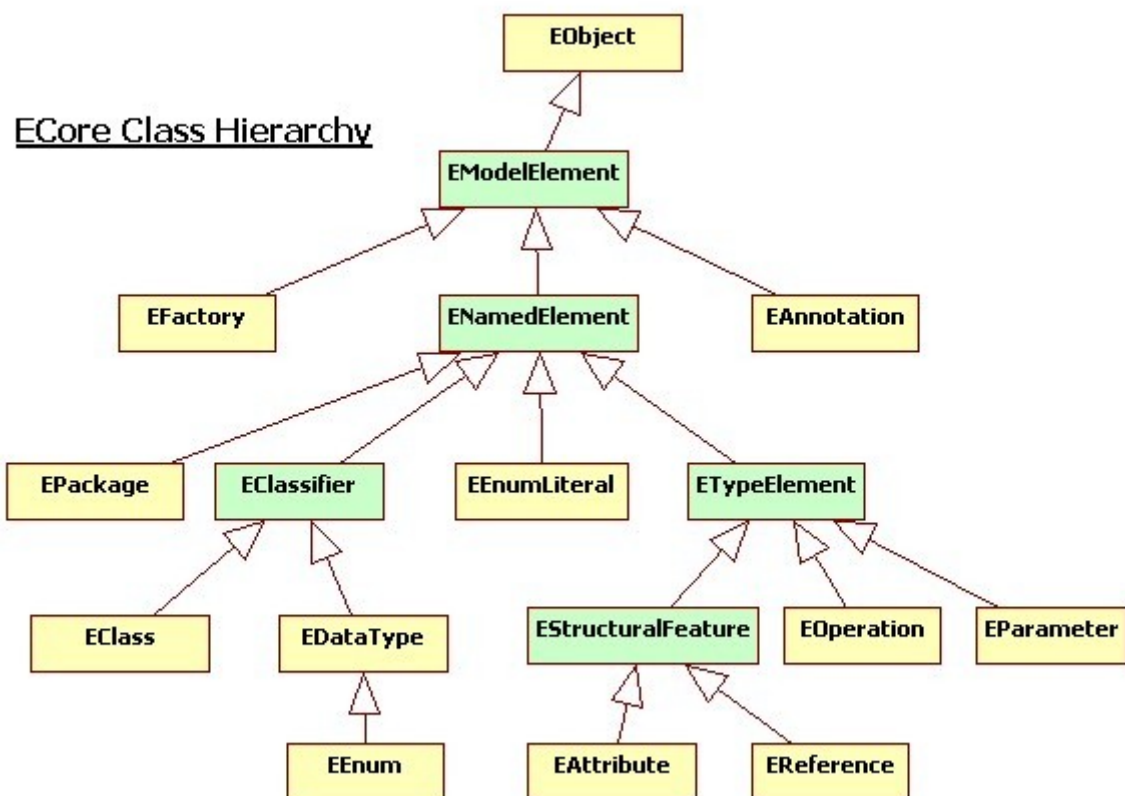
2.3. Socle technique apporté par eclipse (ou équivalent)

Les I.D.E. les plus populaires dans le monde du développement JAVA (eclipse , netbeans,) offrent un socle technique important qui est très souvent utilisé en interne par les outils UML.



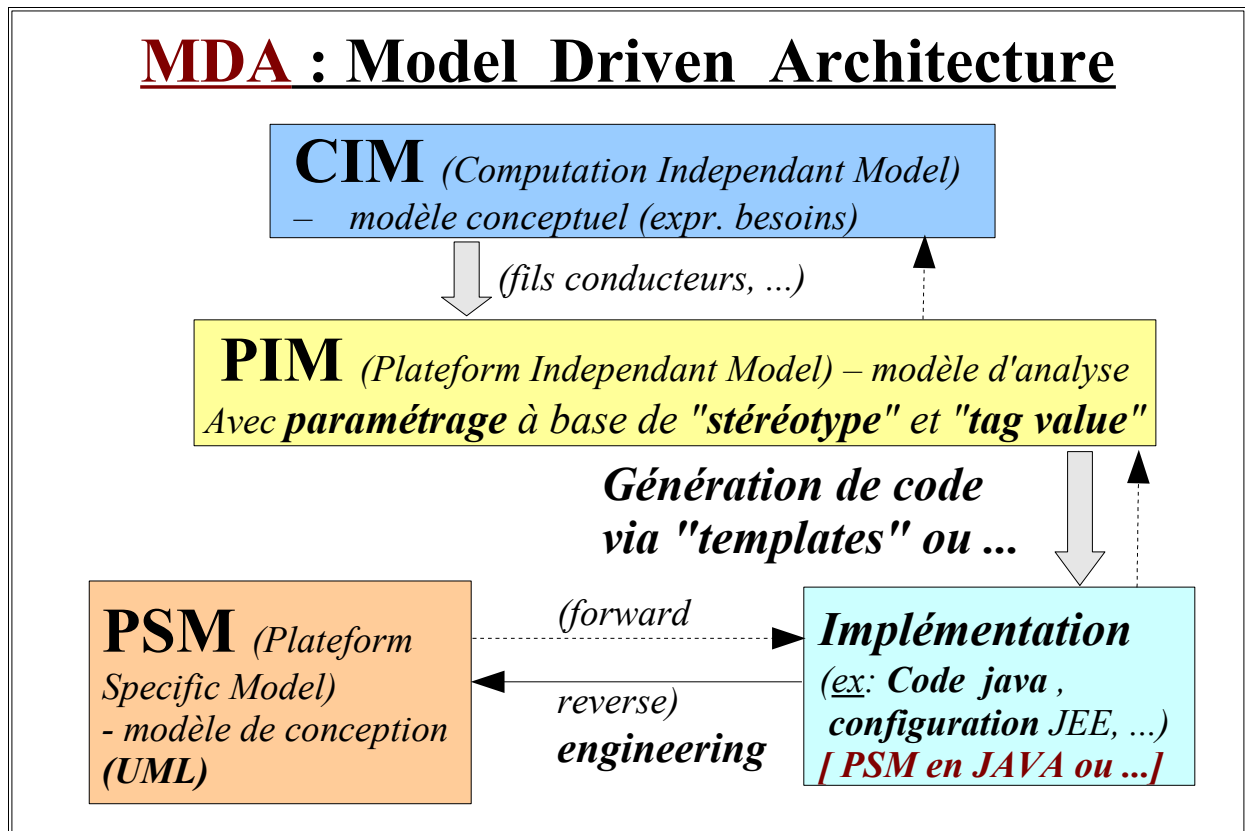
Ainsi , utilisant les mêmes formats internes (.xmi , .uml) en utilisant le même socle technique (GEF/EMF/UML2) , les différents plugins récents pour eclipse (Eclipse UML/Omondo , Topcased UML , accéléo ,) parviennent simplement à dialoguer/coopérer très efficacement.

Méta modèle "ECore" (pour UML ou ...) fourni par la plateforme "Eclipse":



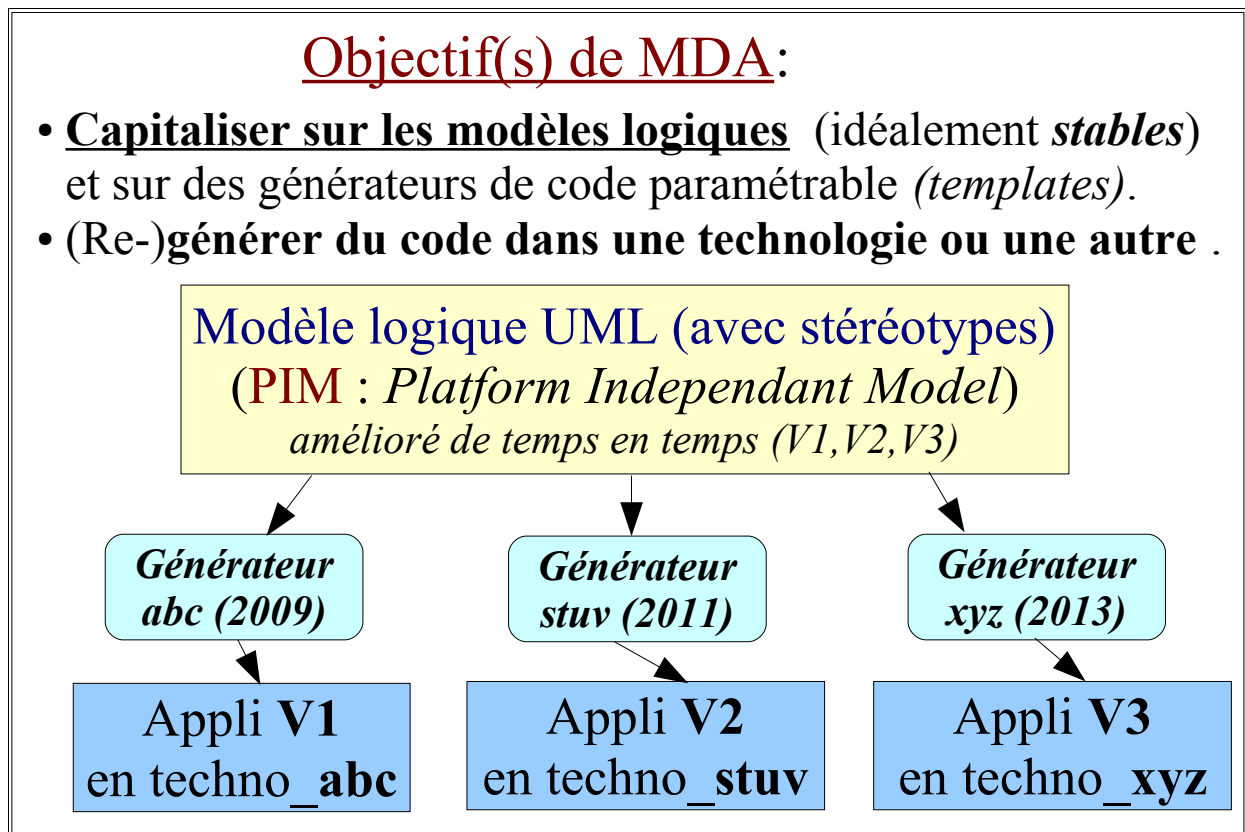
3. MDA

MDA : Model Driven Architecture

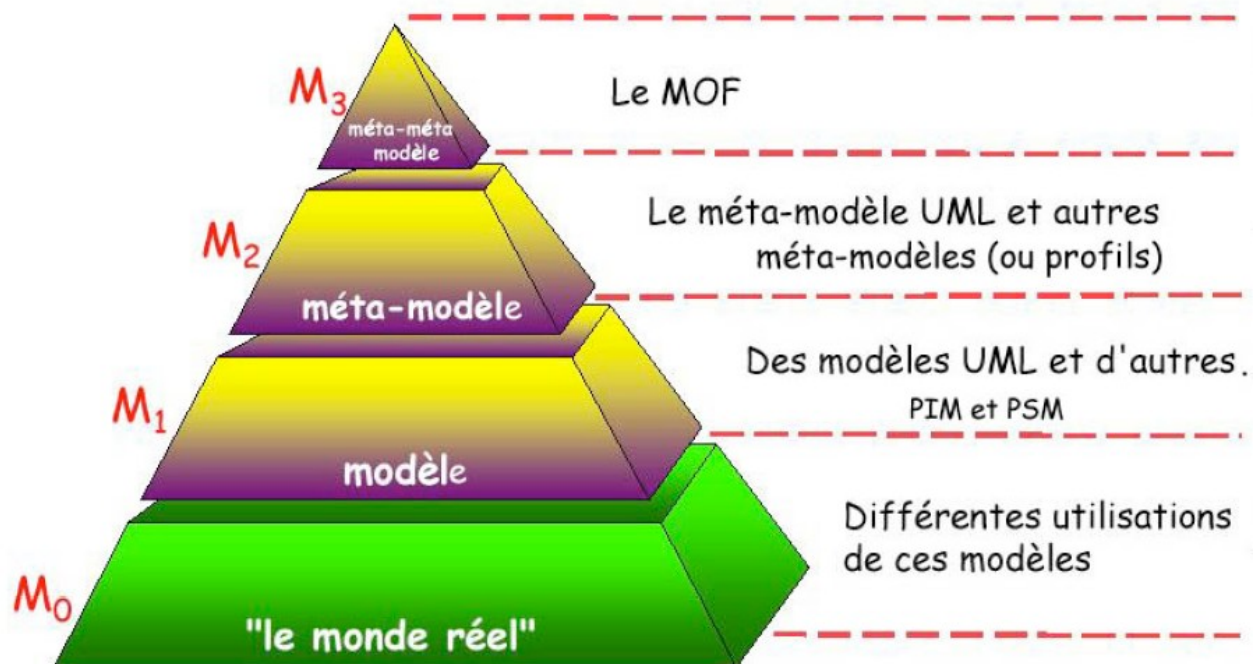


Objectif(s) de MDA:

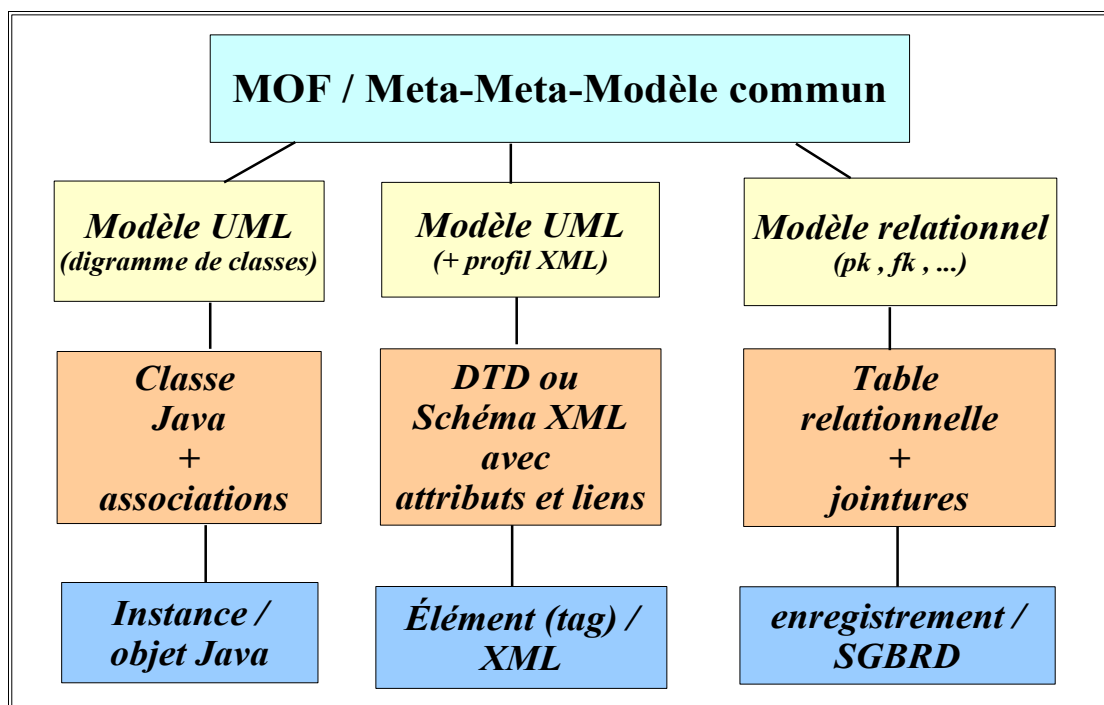
- **Capitaliser sur les modèles logiques** (idéalement *stables*) et sur des générateurs de code paramétrable (*templates*).
- (Re-)générer du code dans une technologie ou une autre .



3.1. Modèles , Méta-Modèles et Méta-Méta-Modèles



- Concrètement , les niveaux M_2 et M_3 sont des standards à utiliser (MOF , UML, ...).
Le niveau M_3 (MOF) parle en terme d'*entités* de *relations* et de *packages*. et les généralités abstraites de XMI (basé sur MOF et XML) sont à ce niveau .
Le niveau M_2 (UML , Profil UML, ...) parle en terme de *classes* , d'*attributs* et de *méthodes* .
Des **versions concrètes de XMI** sont à ce niveau .
- Le niveau M_1 correspond aux modèles et diagrammes à générer (ex: Classe "**Personne**")
- Le niveau M_0 est le monde "réel" des objets (ex: ["**Dupond**" , "rue xxx ..." , ...])



NB: En pratique , au sein d'une application , **plusieurs technologies ont besoin de coopérer** :

- **Modèle relationnel** et **base de données** associée pour la **persistance**
- **Modèle objet** pour la **gestion des processus** et la **représentation des entités**
- **Documents XML** pour les **échanges**

Grosso-modo **en conception**, à partir d'un modèle assez abstrait issu de l'analyse (PIM) , on a besoin de générer **plusieurs petits modèles spécifiques (PSM)** devant être **cohérent entre eux** :

- Le schéma relationnel d'une base de données (**xxx.sql**)
- La structure des échanges XML (Schéma XML <---> Profil XML pour UML)
- La structure des objets métiers et de représentation (Java)

3.2. Quelques produits "MDA" concrets:

<i>Outils/Générateur MDA</i>	<i>Editeur</i>	<i>Open Source ?</i>	<i>Caractéristiques</i>
Andro MDA	oui	Pionnier en MDA , différences selon versions, prévu pour être intégré/piloté par Maven.
Accéléo	Obéo (France)	oui	Très simple à paramétrer , très bonne intégration dans eclipse
...			

XXI - OCL (Object Constraint Language)

1. OCL (Object Constraint Language) - Présentation

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

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

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

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

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

Aujourd'hui OCL est essentiellement utilisé pour:

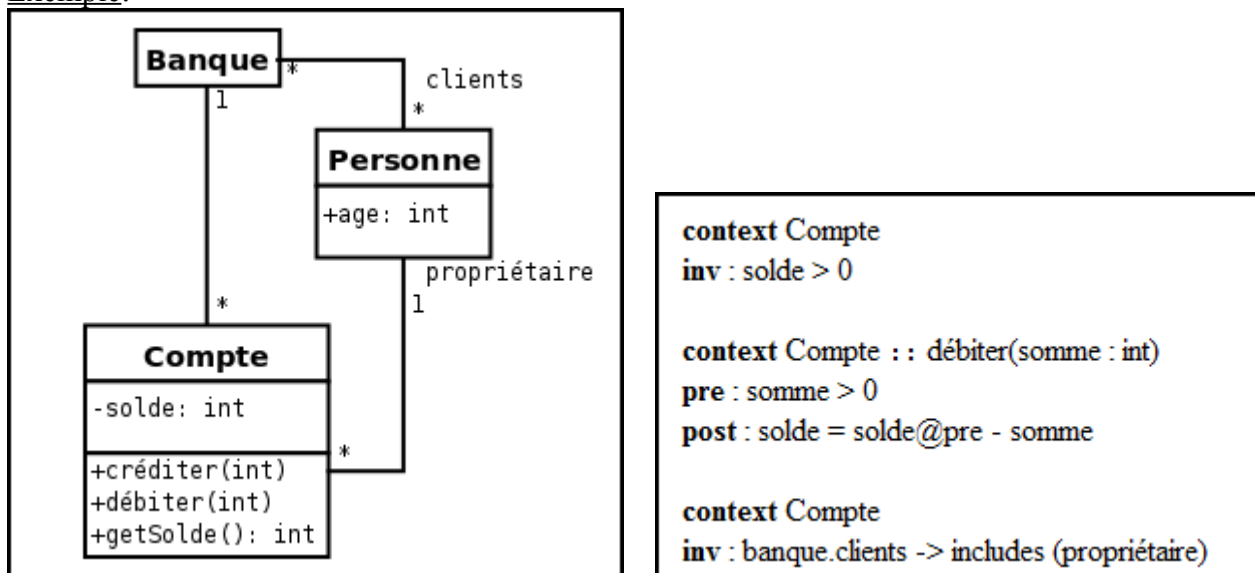
- exprimer des contraintes dans un modèle UML
- exprimer des conditions (gardiens) au niveau des transitions des diagrammes d'états
- exprimer des requêtes portant sur une partie du modèle .
ces requêtes sont (entre autre) utilisable au sein des templates accéléo (générateur MDA)

Point clef pour comprendre la syntaxe OCL:

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

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

Exemple:



Quelques URL pour approfondir:

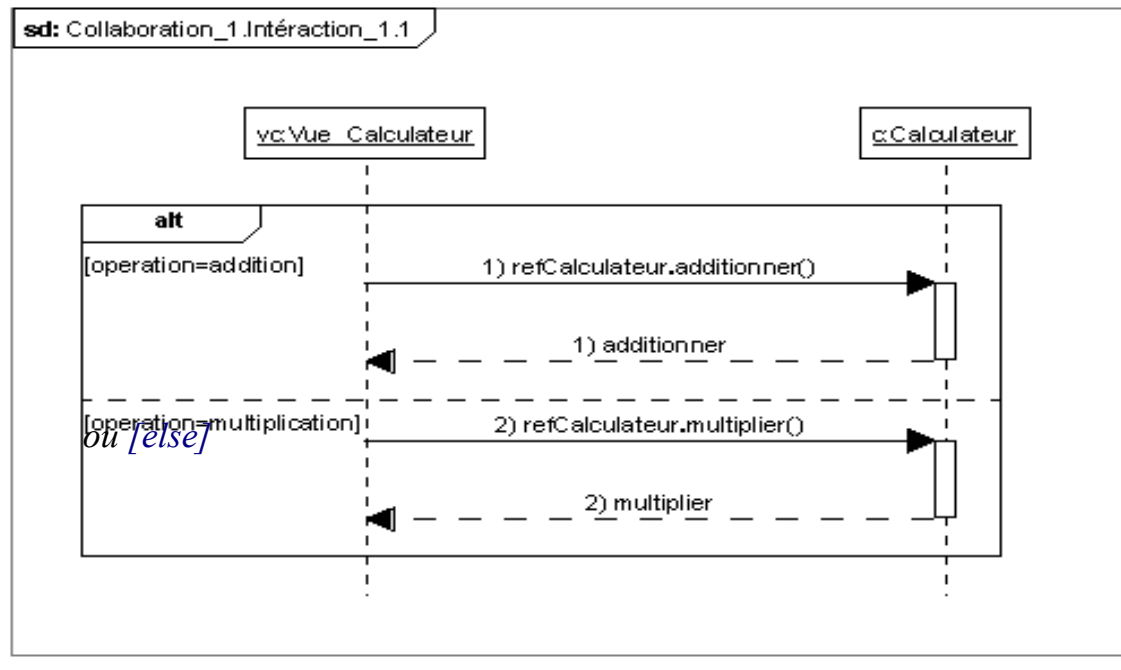
<http://laurent-audibert.developpez.com/Cours-UML/html/Cours-UML021.html>

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

XXII - Notations UML diverses et avancées

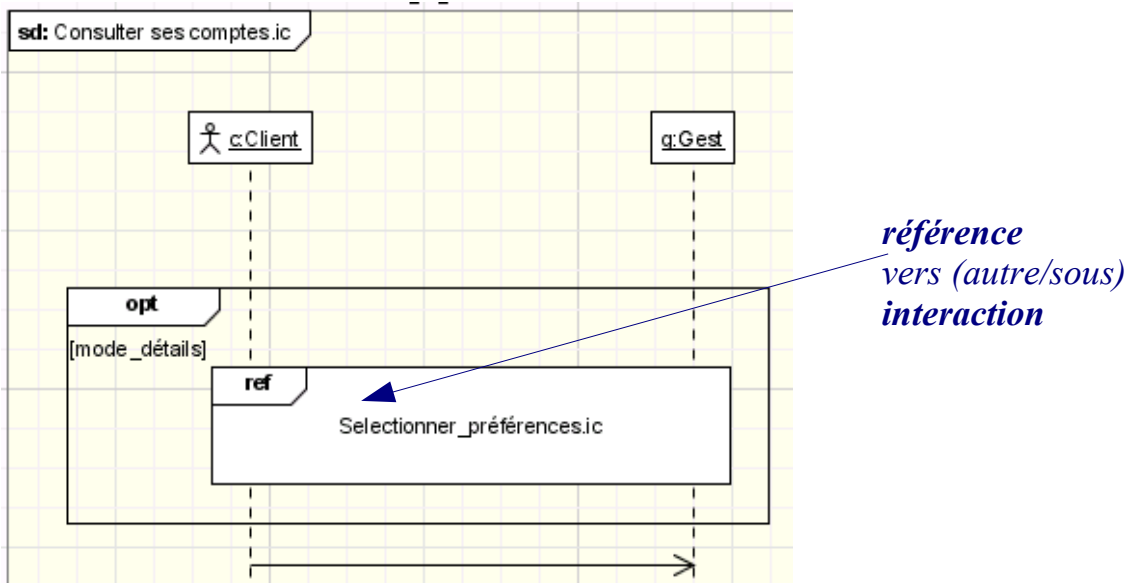
1. Diagramme de séquences (notations avancées)

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

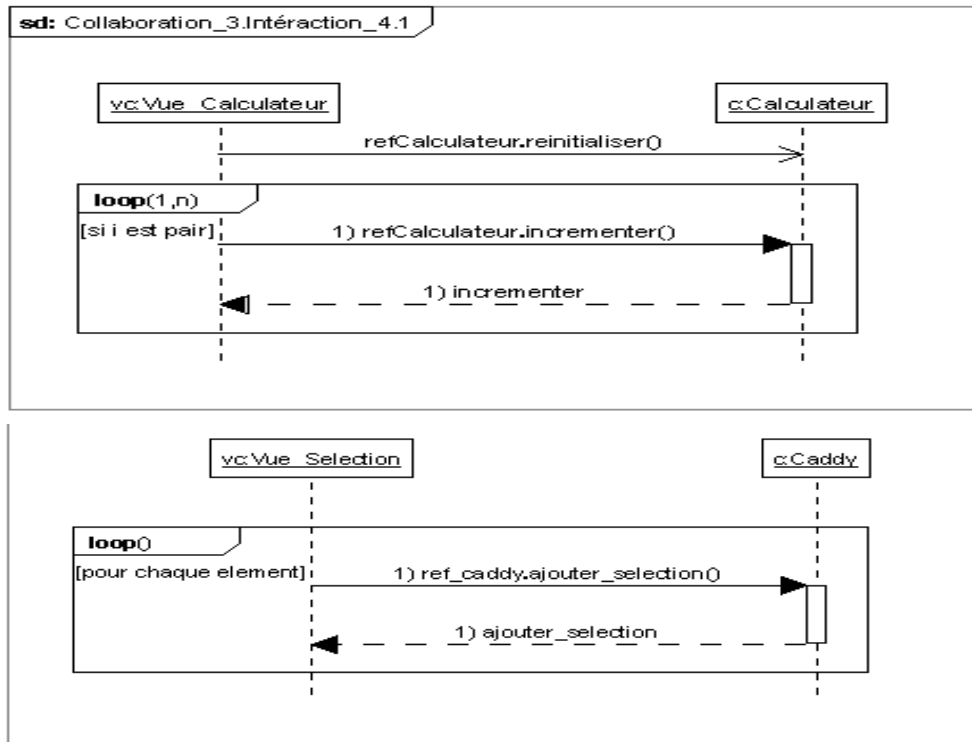


option conditionnée UML2 (opt)

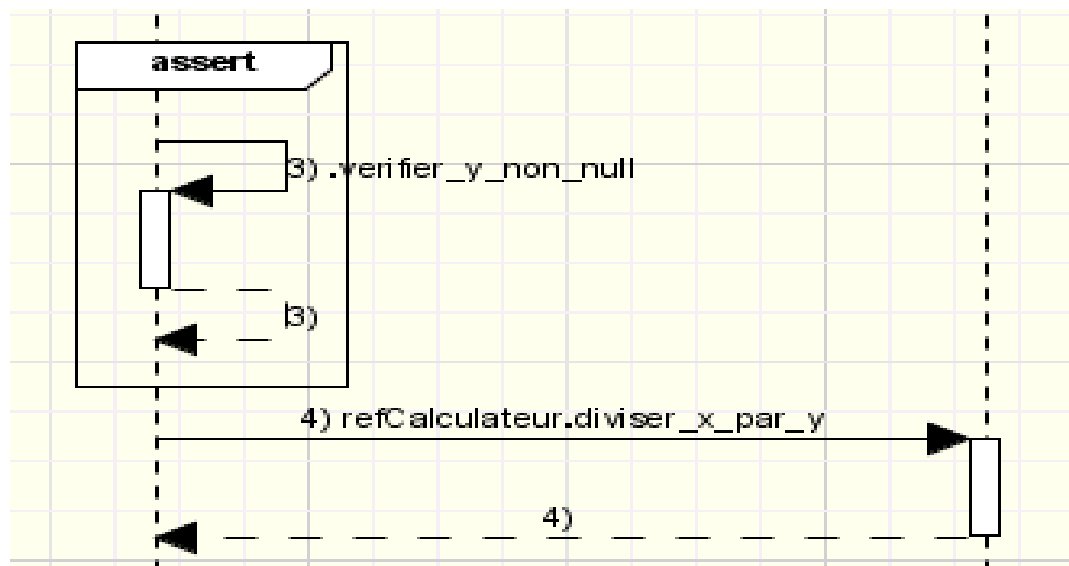
---> *Si [...] Alors (sans else)*



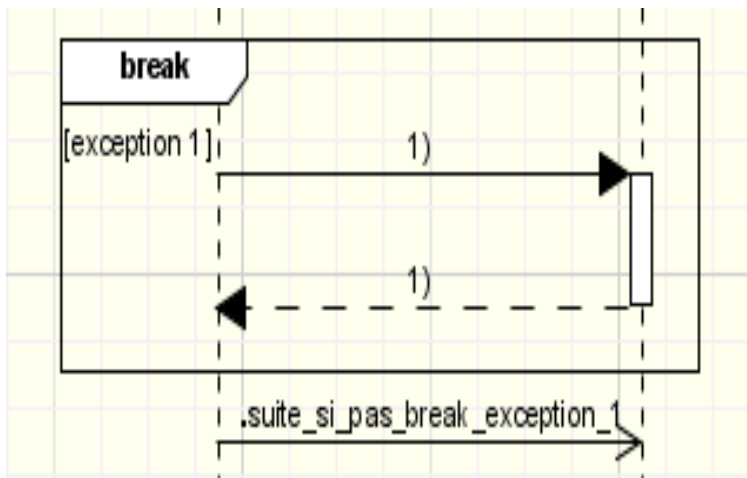
Boucle / **Loop** (UML2)



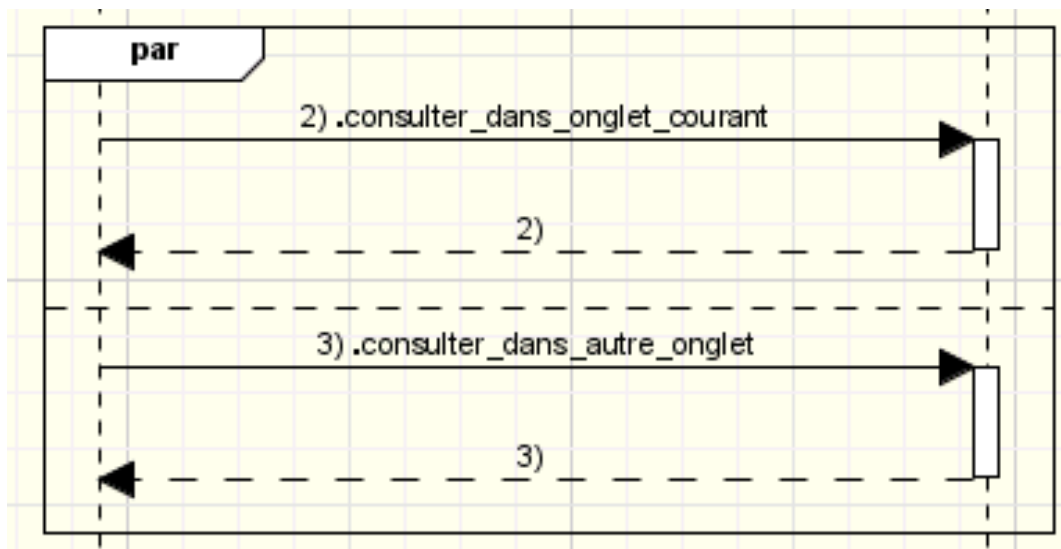
assertion UML2 (assert**)** --> s  quence de tests    absolument v  rifier pour pouvoir continuer



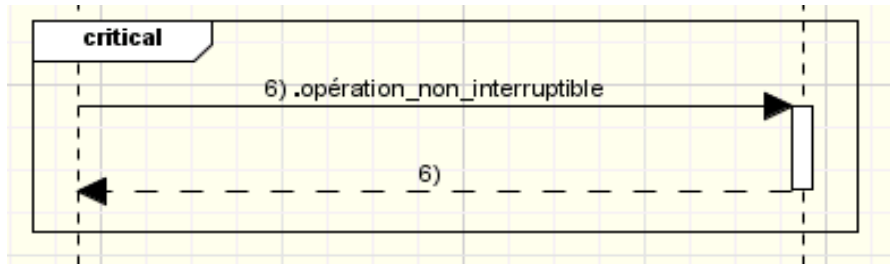
break (UML2) --> traitement en cas d'exception
 (sortie de la séquence normale, la suite n'est pas exécutée)



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

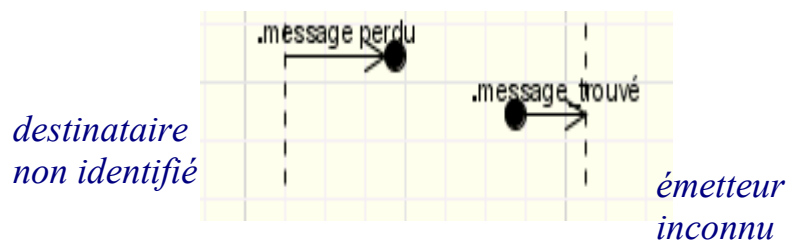
consider { message_important1, m2 }

ignore { message_insignifiant_1 , ... }

Autres détails des diagrammes de séquence

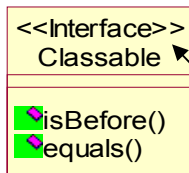
On peut indiquer (en texte clair ou commentaire selon le produit):

- 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).



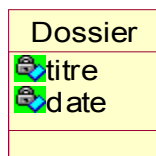
2. Diag. de classes et interfaces

Interface

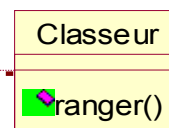


Une interface correspond à une collection d'opérations sans code (classe spéciale sans aucun attribut, seulement des prototypes de méthodes).

2 notations possibles

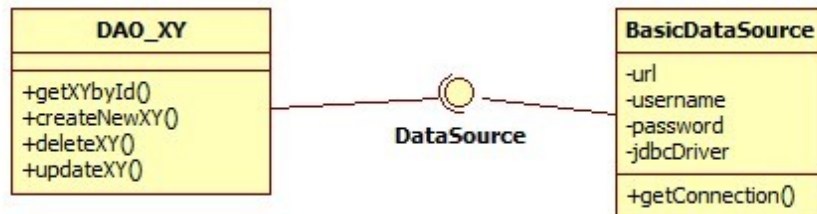


Classable

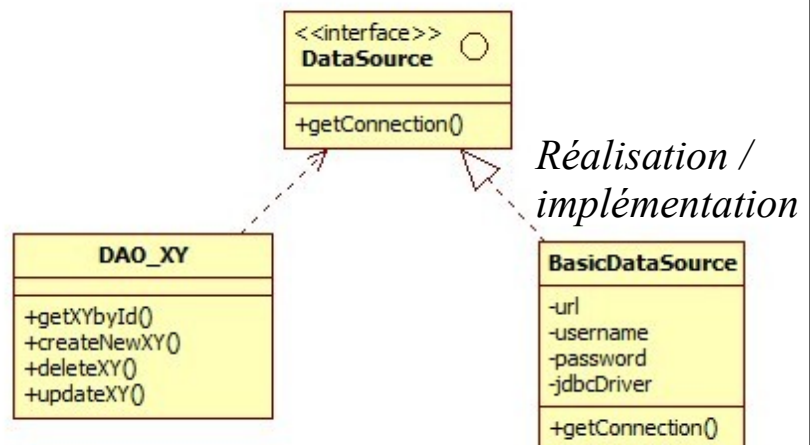


Tout comme une classe abstraite, une **interface** correspond à un **type de données** (permettant de déclarer des références).

*Notation
Compacte
(dépendance)*

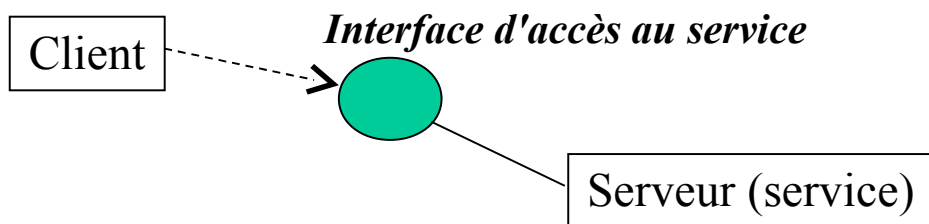


*Notation
développée
(avec détails)*



Interface (contrat)

- Une **interface** peut être considérée comme un **contrat** car **chaque** classe qui choisira d'implémenter (réaliser) l'interface sera obligée de programmer à son niveau toutes les opérations décrites dans l'interface.
- Chacune de ces opérations devra être convenablement codée de façon à rendre le **service effectif** qu'un client est en droit d'attendre lorsqu'il appelle une des méthodes de l'interface.



3. Autres notations avancées (UML2)

3.1. Diagramme de structures composites

--> diagramme UML2 (assez récent) permettant de montrer les constituants internes d'un objet .
 --> diagramme essentiellement utile en conception (ex: voiture composé d'un moteur , composé lui même de cylindres , pistons, ...)

3.2. Notion de "Port" (alias "point d'accès")

Un "port" (quelquefois appelé "slot") correspond à un point d'entrée (vu de l'extérieur) vers des fonctionnalités internes et cachés d'un module.

Exemple: Port_USB d'un ordinateur ,

Exemple2: "Endpoint_WebService" relié à une classe Java .

XXIII - Différences entre UML et Merise + O.R.M.

1. Préliminaire: 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 un enregistrement d'une table relationnelle et est assimilable à une instance (objet précis)]

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

Multiplicité (UML):

La **multiplicité** (exemple: "0..1" ou "0..*") correspond au **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 côté "**Classe Personne**" dans les diagrammes de classes UML.

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

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

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

2.1. 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).

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

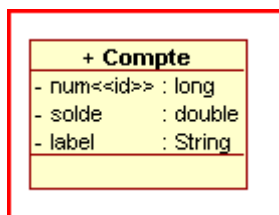
Dans un certain nombre de cas, la base de données existe (parce qu'elle est partagée avec un projet existant ou alors parce qu'il s'agit d'une migration d'une application existante vers Java) avant que la conception de l'application ne soit réalisée .

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" .

3. Correspondances essentielles "objet-relationnel"

3.1. Entité (UML) / Table simple

<i>Eléments UML</i>	<i>Eléments relationnels correspondants</i>
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.
- ...

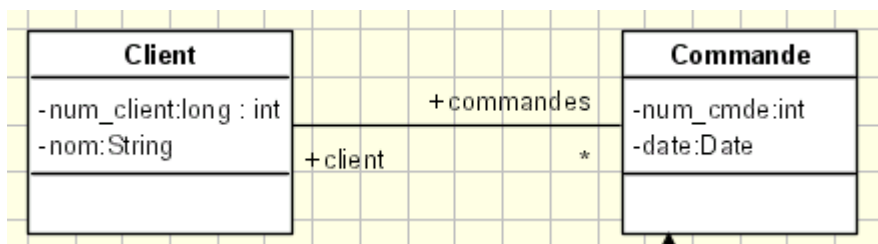
3.2. Association UML / Jointure entre tables

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

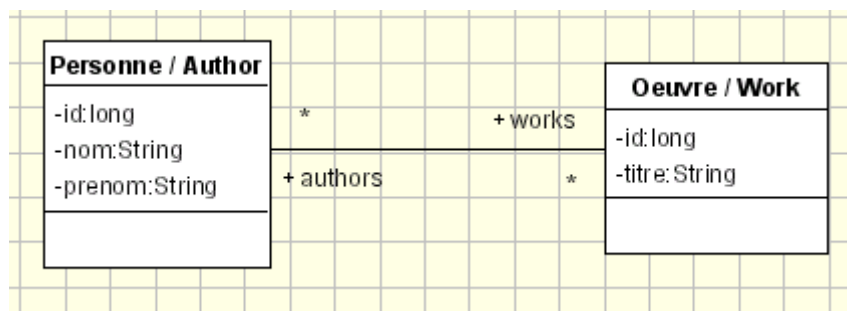
num_client (pk) , nom ,

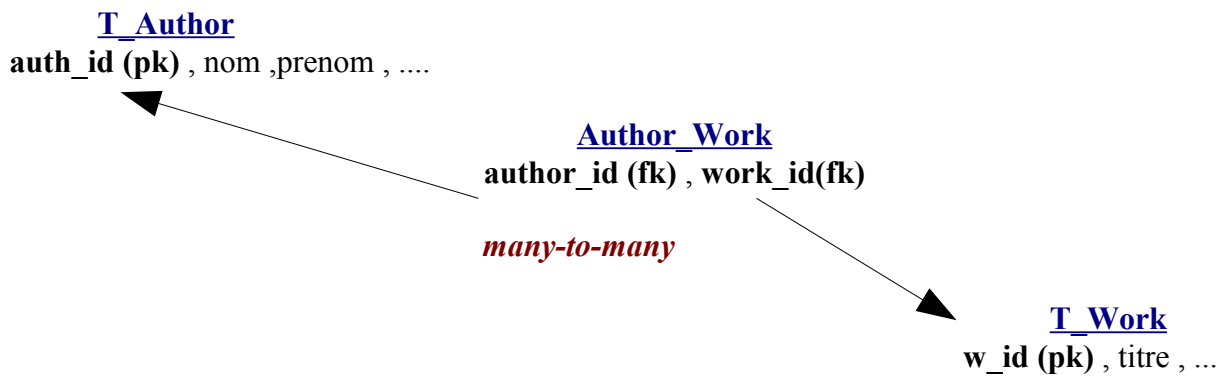
many-to-one

T_Commande

num_cmde (pk) , client (fk), date,

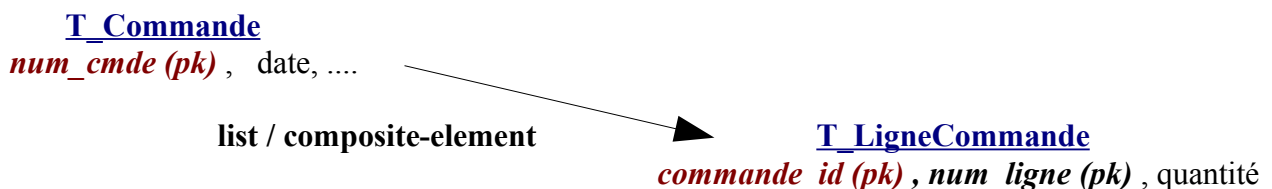
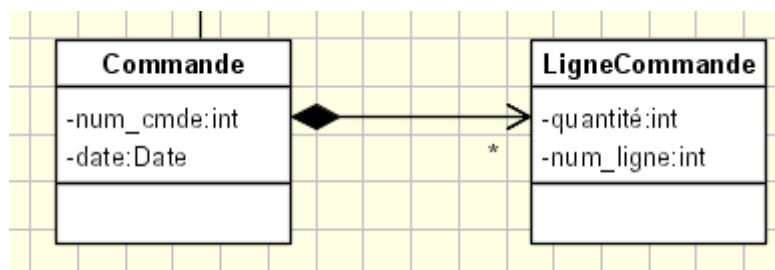
n-n





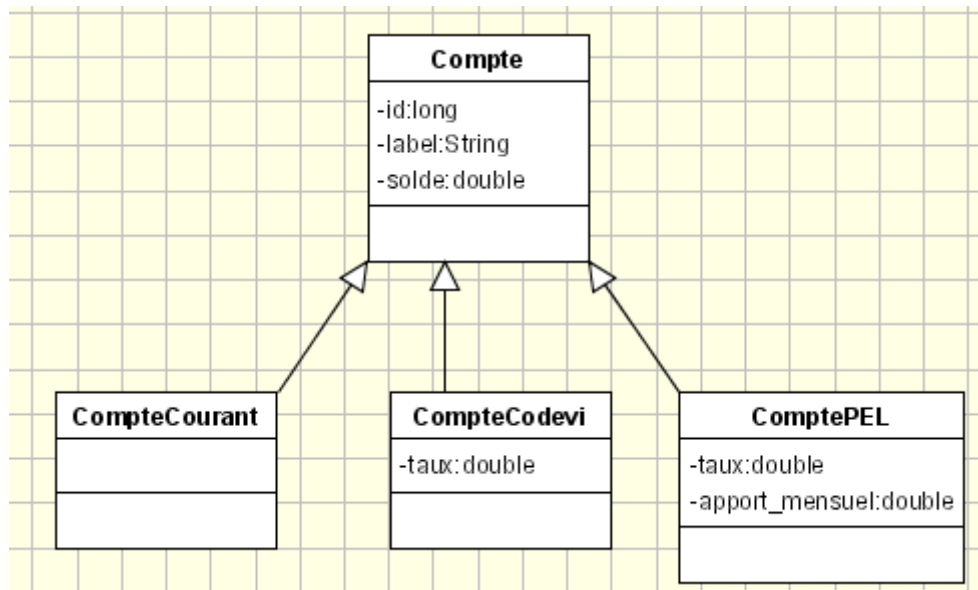
3.3. Composition UML / table "détails" avec clef primaire composée

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



4. Correspondances "objet-relationnel" avancées

4.1. 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)

+

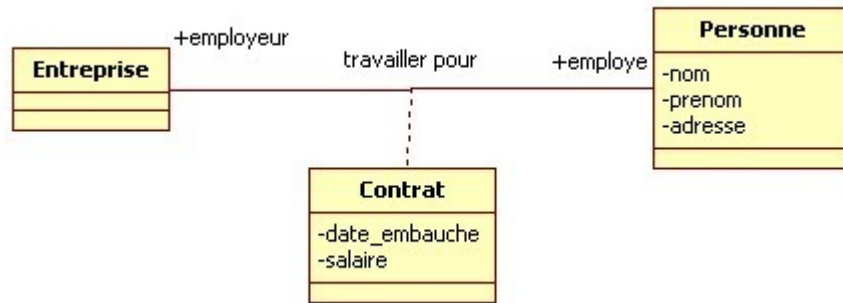
1 table pour chaque classe fille avec "colonnes suppléments" et clef étrangère fk référençant l'id de la table des points communs .

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 .

4.2. 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 ,

4.3. 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)

XXIV - BPMN (aperçu)

1. BPMN (présentation)

BPMN signifie *Business Process Management Notation*

Il s'agit d'un formalisme de modélisation spécifiquement adapté à BPEL et à la modélisation fine des processus métiers (sous l'angle des activités).

Un diagramme **BPMN** ressemble beaucoup à un diagramme d'activité UML . Les Choses exprimées sont à peu près les mêmes.

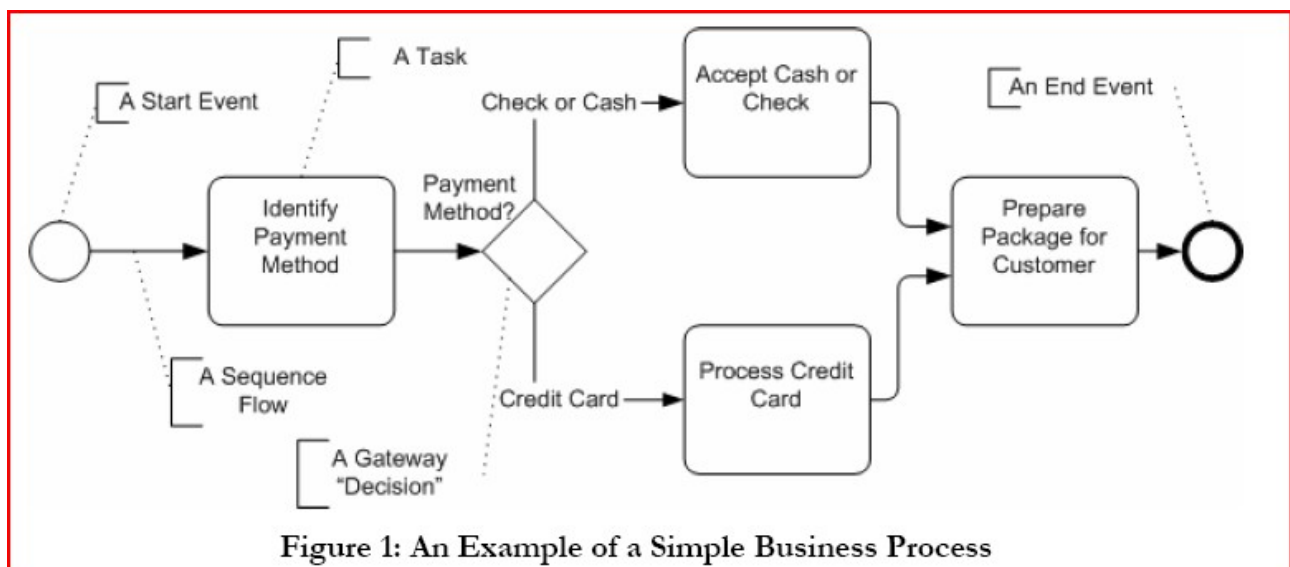
La Norme UML évoluera peut être de façon à intégrer les spécificités BPMN.

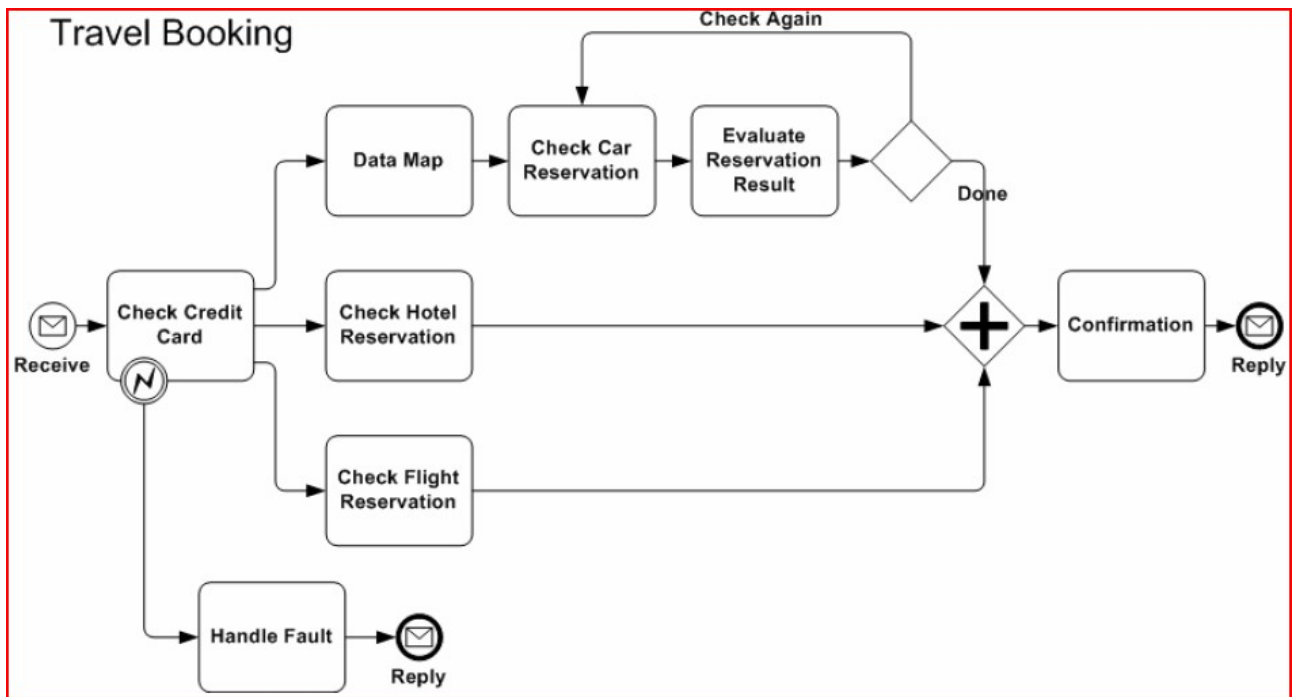
Les différences entre UML et BPMN sont les suivantes:

- la syntaxe des diagrammes d'activités UML est dérivée des diagrammes d'états UML et est plutôt orientés "technique de conception" que "processus métier"
- à l'inverse la syntaxe des diagrammes BPMN est plus homogène et plus parlante pour les personnes qui travaillent habituellement sur les processus métiers.
- UML étant très généraliste et avant tout associé à la programmation orientée objet, il n'y a pas beaucoup de générateurs de code BPEL qui utilisent un diagramme d'activité UML
- à l'inverse la plupart des éditeurs BPMN sont associés à un générateur de code BPEL. Les éléments fin d'une modélisation BPMN ont été pensées dans ce sens.

Outil concret pour la modélisation BPMN ---> *Intalio BPMN* .

Exemples de diagrammes BPMN:





====> notations à éventuellement approfondir si besoin .

XXV - Orientation SOA, problématique, architecture

1. Modélisation SOA

UML est tout à fait approprié pour modéliser les structures de données (xsd) qui seront échangées au niveau des services WEB et des autres éléments de l'infrastructure SOA.

Eléments importants d'une bonne modélisation SOA:

- Services (à avant tout voir de manière fonctionnelle , c'est à dire comme des services métiers)
- Structure de données échangées (entrées, sorties, exceptions/faults) .
Le vocabulaire "ValueObject" / "Data Transfert Object" est ici assez approprié.
- Catégorie/secteur métier englobant tel ou tel service (délimitation systémique ==> organisation en packages logiques / namespaces xml).
- Dépendances fonctionnelles entre les services (et avec structures de données produites/consommées indirectement concernées).
- Processus métiers avec logiques/règles métiers.
- Quelques éléments techniques fondamentaux (volume, perf , synchrone ou asynchrone,)
-

2. Notions et concepts étroitement liés à SOA

2.1. Format de données pivots et gouvernance des données

SOA mène idéalement à une architecture décentralisé/départementalisée où différents systèmes collaborent via un couplage faible (pas de liens très étroits, relative indépendance des systèmes).

Bien que certains éléments techniques de l'infrastructure SOA (transformations des formats de données via xslt ou ... , conversion de protocoles ,) puissent aider à rendre techniquement indépendants les différents systèmes devant communiquer entre eux, il reste toujours des dépendances assez fortes d'un point de vue fonctionnel.

Par exemple, on peut transformer un "client" en "customer", une chaîne d'adresse en éléments décomposés (rue, code_postal , ville) . On peut aussi récupérer un user_name manquant via un service intermédiaire d'extraction du user_name en fonction d'un user_id mais on ne peut pas transformer un numéro_de_client et numéro_de_produit.

Autre point à prendre en considération : qui dépend de qui (d'un point de vue système)? Quels sont les systèmes "cœur de métier" dont on maîtrise l'évolution. Quels sont les systèmes annexes (bien souvent achetés) et pourtant fondamentaux/indispensables dont on ne maîtrise pas tout à fait l'évolution (en fonction des choix et de la pérennité des éditeurs).

Ce vaste sujet (d'urbanisation) est très important au niveau de la modélisation SOA.

La bonne approche pragmatique souvent appelée "gouvernance des données" consiste à garantir une bonne agilité du SI en appliquant les dispositions suivantes:

- applications périphériques complètement indépendantes
- éléments collaboratifs principaux (cœur de métier) avec données "pivot" dont la structure est en

partie déterminée par gouvernance (prise en compte de multiples besoins).

Soit appli1

---> tx1(requête_format1)

----> requête_format_pivot

(format normalisé par organisation)

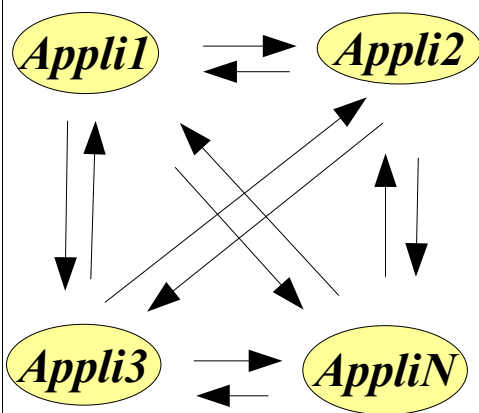
---> tx2(vers_format_appli2)

---> requête_format2 reçu par appli2

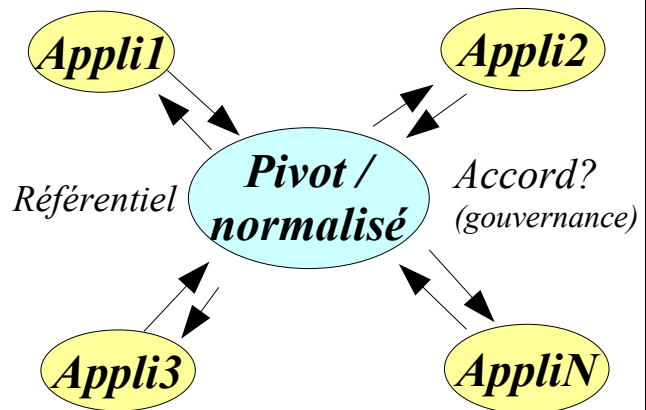
où tx1(...) et tx2(...) sont deux transformations (éventuellement) nécessaires.

-

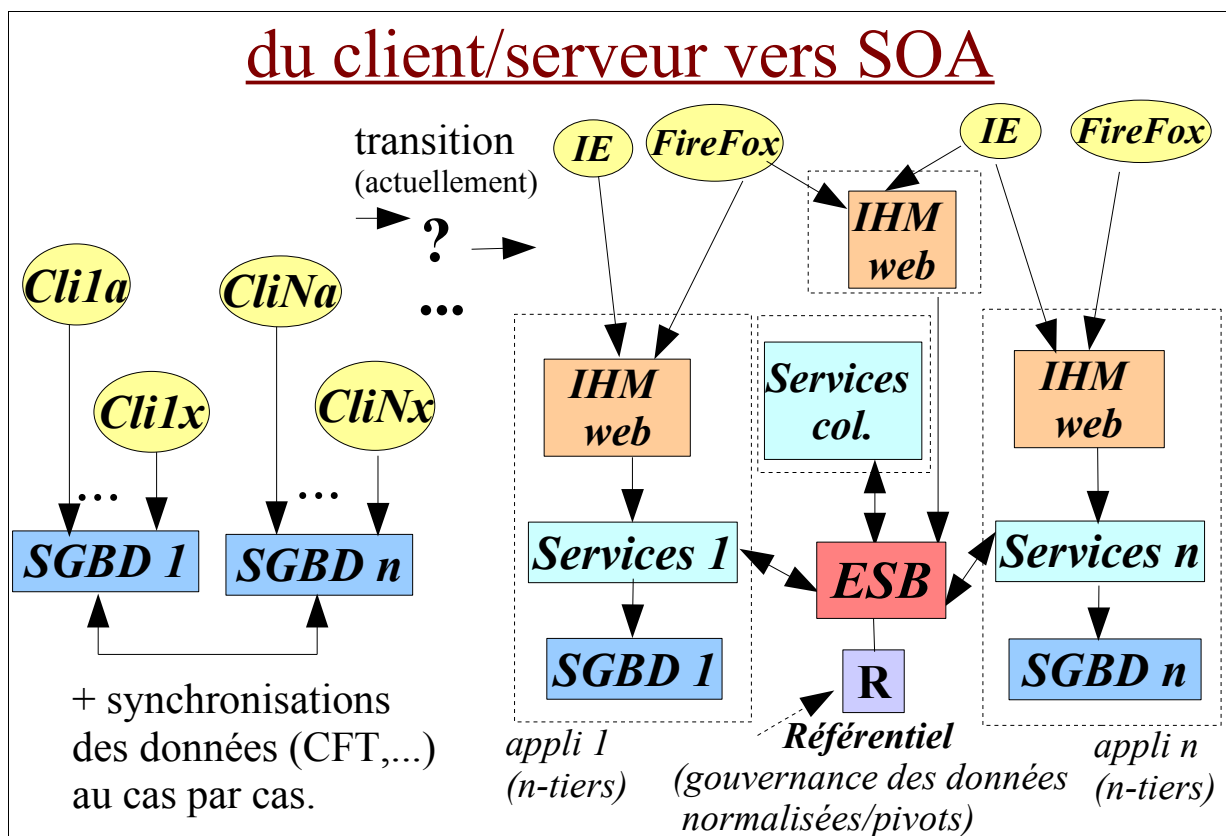
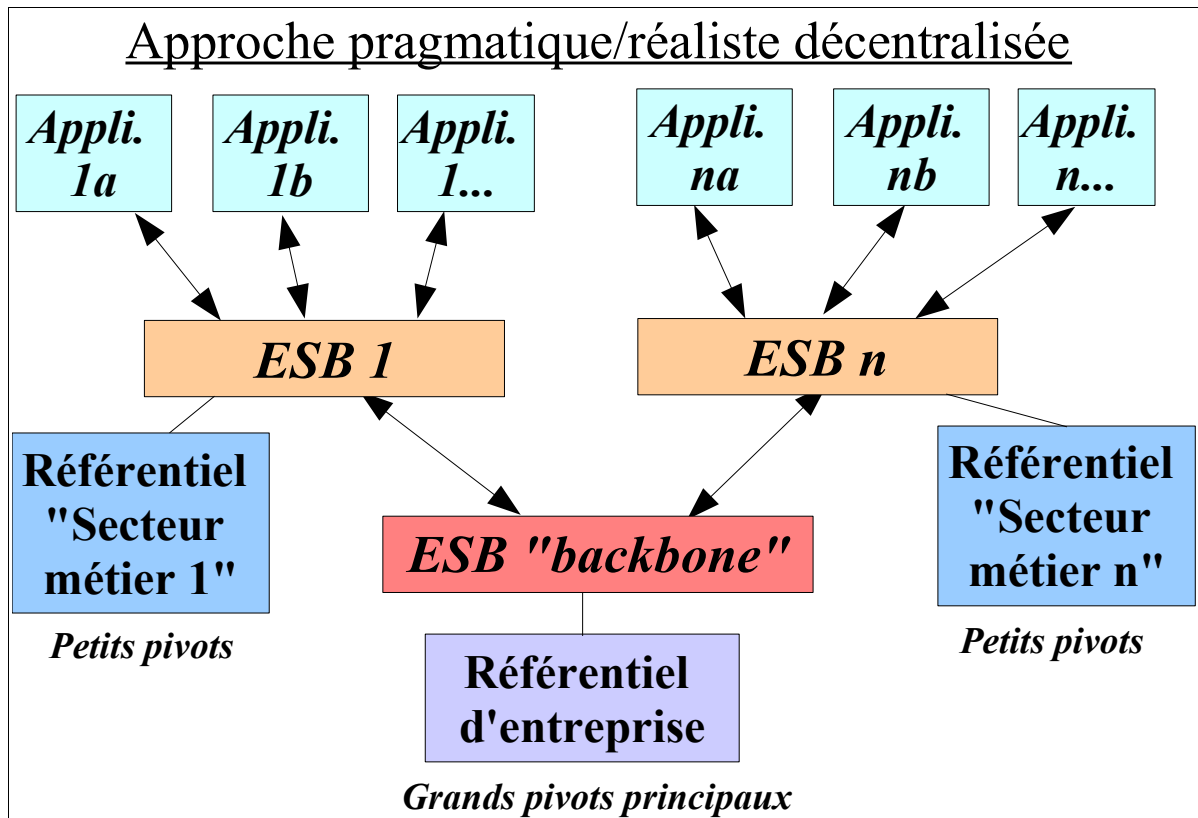
SOA : format normalisé / pivot



Si transformations
des formats de données
au cas par cas :
--> besoins en $O(n^2)$!



Si transformations avec format
de données intermédiaire
normalisé (pivot) :
--> besoins en $O(n)$!

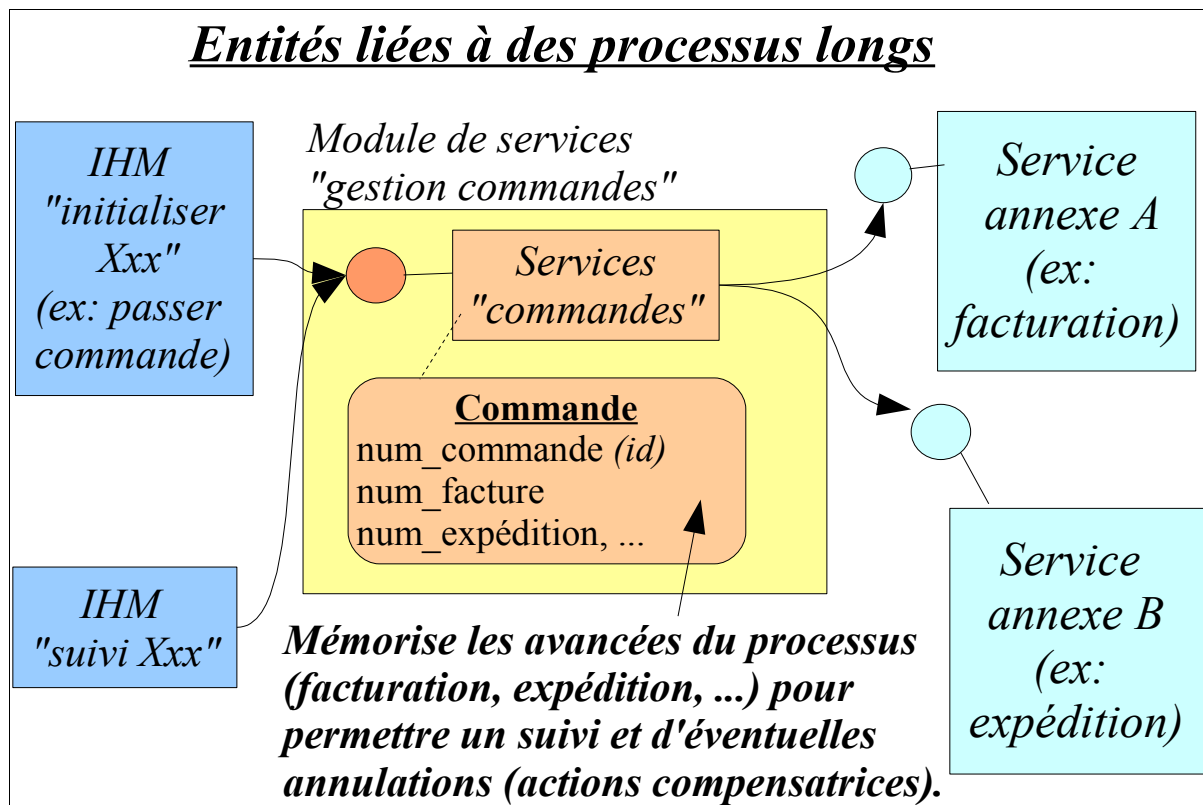


2.2. Transactions longues et compensations

De nombreux échanges SOA sont effectués en mode asynchrone lorsque certains traitements ou sous processus (ex: livraison) sont longs.

Les éventuelles transactions associées, longues elles aussi, ne peuvent pas se permettre d'exiger un verrouillage ou une isolation temporaire des données mises en jeu sur une longue période.

En cas d'échec d'une transaction longue, il faut prévoir des mécanismes de compensation (action inverse annulant l'action d'origine : exemple= remboursement si livraison non effectuée).



Autre exemple classique : entité "dossier" avec num_dossier,

2.3. Stabilités des éléments référencés

Dans un système à contraintes d'intégrités centralisés (telle qu'une base de données unique prise en charge par un SGBDR), il est souvent impossible de supprimer involontairement/accidentellement une entité qui est encore référencée par une autre.

A l'inverse dans un système plutôt décentralisé comme internet ou SOA, une entité peut être supprimée dans un système (application) sans que les autres systèmes soient au courant (ex: URL devenue invalide, service utilisant un "vieil" id pour référencer une entité qui existait et n'existe plus).

==> Les spécifications (cahier des charges, modélisations, ...) devraient idéalement spécifier clairement certains éléments tels que la durée de validité d'une référence (et/ou des régulières vérifications, ...).

3. Méthode Praxème

La méthodologie Praxème comporte quelques bonnes idées concernant l'approche SOA.

Traits saillants:

- distinguer les aspects logiques, pragmatiques, géographiques, physiques,
- modéliser d'abord les aspects structurels et logique/fonctionnel des processus et services métiers (ex: diagrammes de classes UML + diagrammes d'états UML)
- modéliser seulement ensuite les activités

Autrement dit :

- ne surtout pas modéliser une série d'activités en s'appuyant sur une organisation pragmatique existante qui n'est pas remise en cause ou que l'on ose pas restructurer .
- Raisonner "fonctionnalités abstraites/logiques devant être apportées" plutôt que "comment relier ça exactement".

Auteurs de Praxème: Philippe DESFRAY , Dominique VAUQUIER .

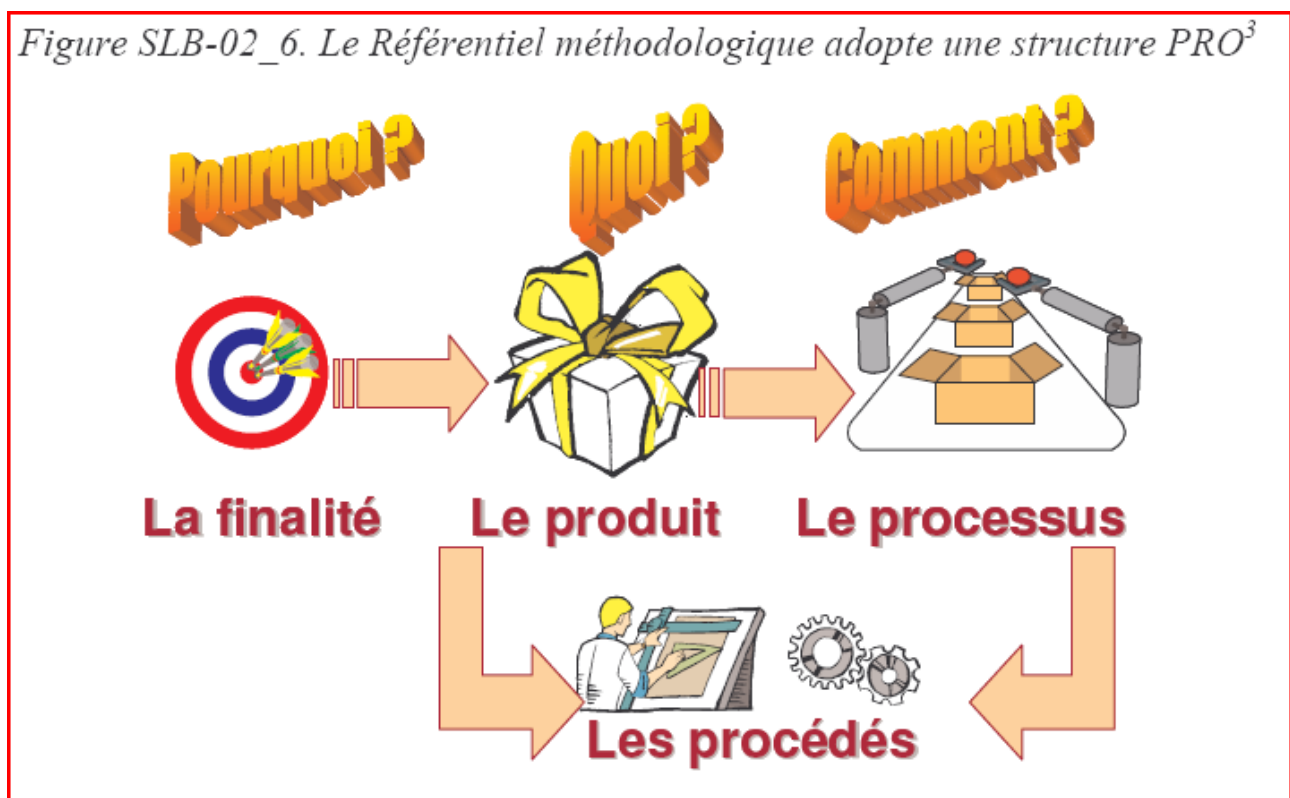


Figure PxM-02_3. Le schéma de la topologie du système

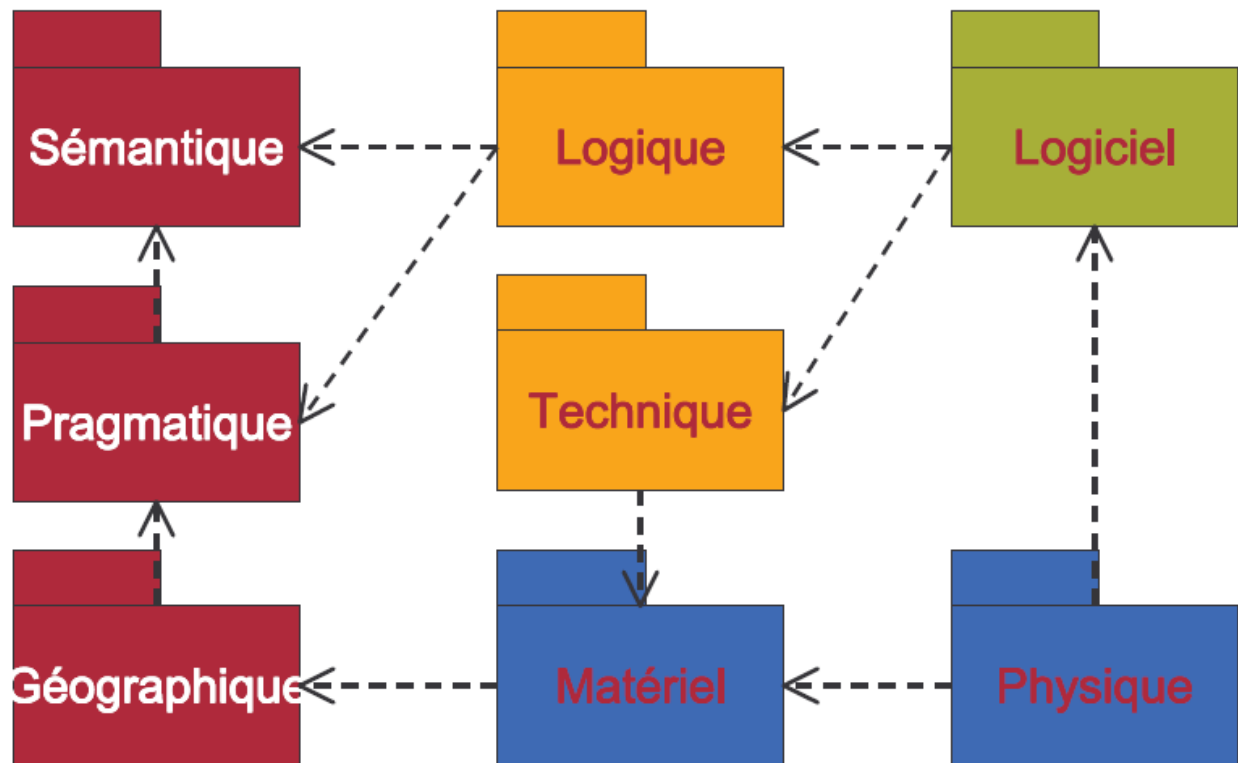
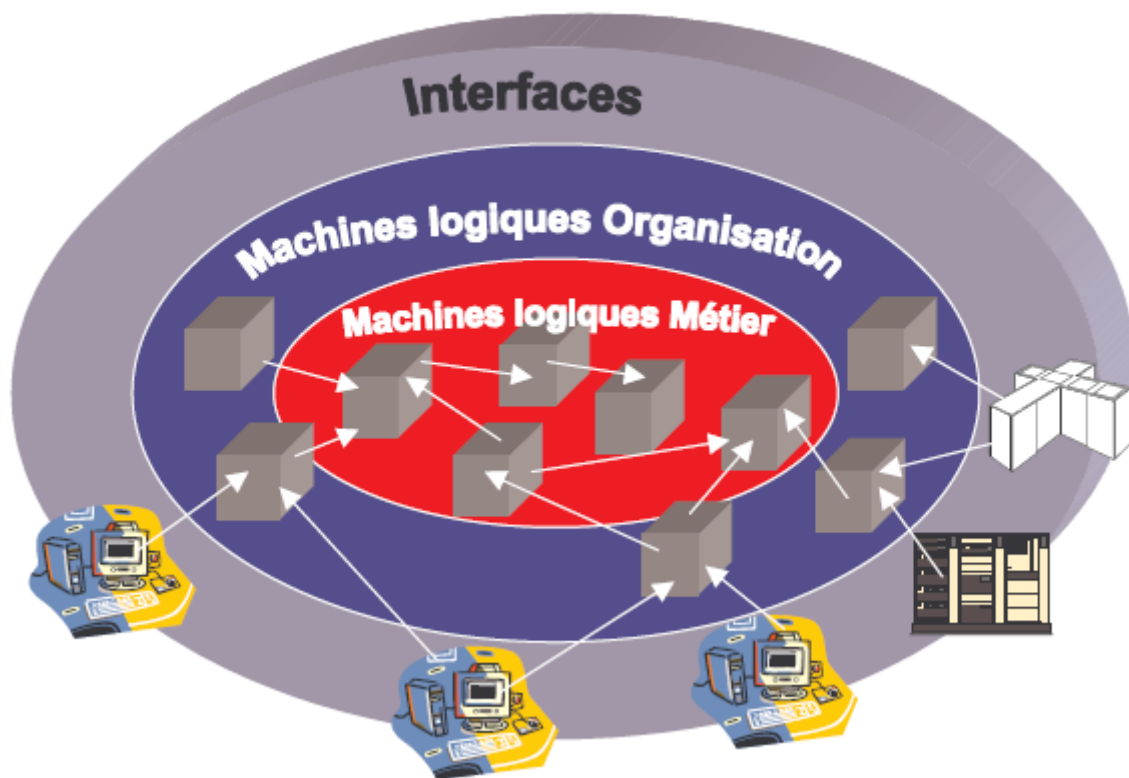


Figure PxM-02_11. La stratification du système d'information, déduite des règles de l'architecture logique



XXVI - Méthodologies (aperçu rapide)

1. Processus unifiés (UP)

1.1. Meilleures pratiques communes

Processus unifiés (UP)

Processus unifiés (UP) : Il s'agit en fait de **processus basés sur une trame commune** issue des **meilleures pratiques de développements**.

Quelques exemples de processus unifiés:

RUP (Rational U.P.) (origine de UP)

2TUP (2 Tracks U.P.) (Processus en Y)

U.P. basé sur **XP** (eXpérience / eXtreme Programing)

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**

2. Principaux Processus

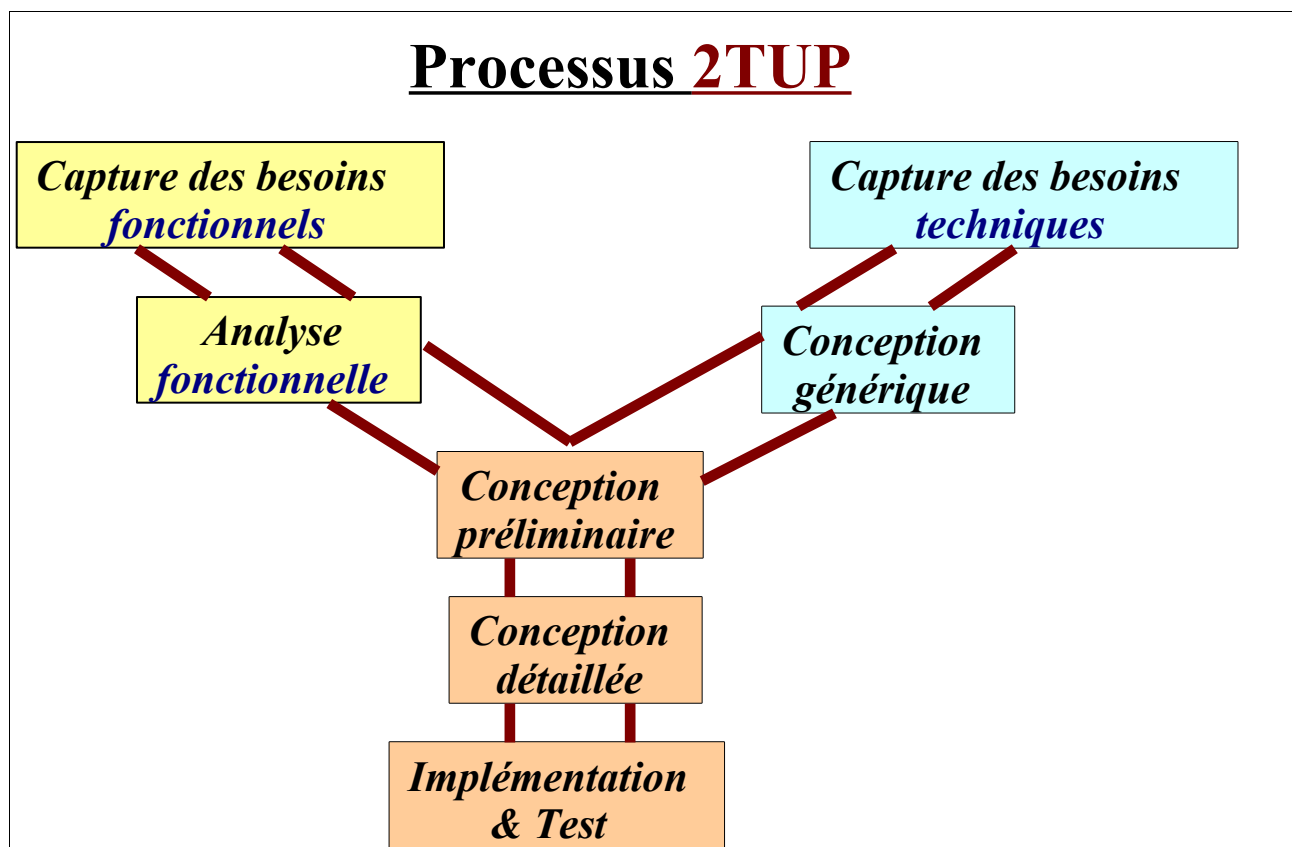
2.1. RUP (Rational UP – origine d'UP)

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.

2.2. 2TUP (2 tracks UP) – Processus en Y



L'originalité de ce **processus en Y** tient dans ses **2 pistes (tracks) initiales bien distinctes**:

- la **branche** de gauche dite **fonctionnelle** ne se focalise que sur les aspects "métiers"
- la **branche** de droite dite **technique** se focalise quant à elle sur les aspects technologiques génériques (ex: sécurité, stockage, transactions, ...).

Une partie délicate de la conception préliminaire consiste alors à **PROJETER** les aspects FONCTIONNELS dans la Solution TECHNOLOGIQUE choisie lors de la conception générique.

[NB: Le livre "UML en action" (entièrement basé sur *2TUP*) permet d'approfondir le sujet]

2.3. XP (eXPerience & eXtreme Programing)

eXPerience

- **Tenir compte de l'expérience (critiques, retours, ...).**
- Via un découpage très fin (parties de UC) on définit des **incréments et des livraisons par parties à des dates fixes**(ex: tous les 3 mois).
- **Planification fine et non figée, tenant compte des priorités et des risques**
 → **Renégociations partielles portant sur le contenu à livrer.**
 → **Ré-évaluer souvent la planification du projet en fonction des changements et des retours .**

Pragmatisme (eXtreme Programming)

- Ne pas anticiper sur des évolutions qui n'interviendront peut être jamais.
- **Tester très fréquemment.**
- Effectuer du **refactoring** (refonte & réadaptation du code).
- **Test de non régression.**
- (Typage faible & souplesse) primant sur rigidité.
- Eventuelle programmation par équipe de 2 (une personne qui code , une personne qui corrige immédiatement les erreurs , qui aide , qui donne son avis , ...).

3. Présentation de RUP

R.U.P. (Rational Unified Process) est un des membres de la famille des **processus unifiés (U.P.)**. A ce titre, il se conforme aux bonnes pratiques de développement suivantes:

- **Itératif et incrémental.**
- **Centré sur une architecture modulaire à base de composants.**
- **Focalisé sur les véritables besoins (contraintes techniques, fonctionnalités, ...).**
- **Basé sur des modèles que l'on crée visuellement et que l'on affine.**
- **Vérifications qualitatives (tests, ...).**
- **Intégrer la gestion des changements** (gestion des versions, maintien de la cohérence entre code et modèle, ...)

R.U.P. est développé par la société "**Rational Software**".

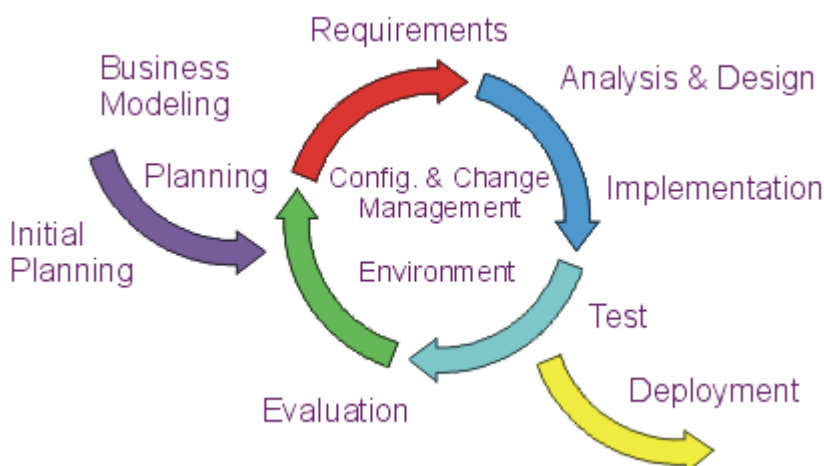
R.U.P. est concrètement constitué de:

- Un **important ensemble de pages "html"** constituant une **base de connaissance** et une **sorte de guide (démarche) sur la gestion de projet "orientée objet"**. Ces pages "html" détaillent essentiellement les activités à réaliser pour mener à bien les différentes étapes classiques (Conceptualisation, Analyse & conception, ...).
- Nb: la consultation de ces pages (depuis le site internet Rational Rose) nécessite une inscription préalable (une sorte de période d'évaluation gratuite est prévue).
- Quelques modèles de document (templates)
- Quelques conseils
- Des indications sur l'utilisation des outils de la société Rational (**Tools Mentors**)
- ...

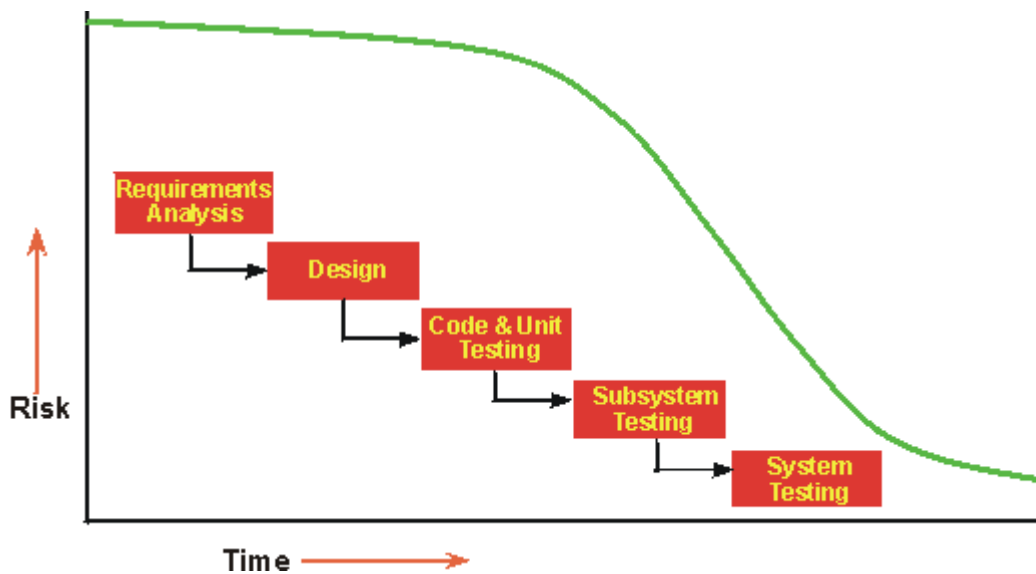
NB: La société "**Rational**" a été rachetée par **IBM**.

RUP fait maintenant partie de **RMC (Rational Method Composer)** qui s'intègre dans eclipse.

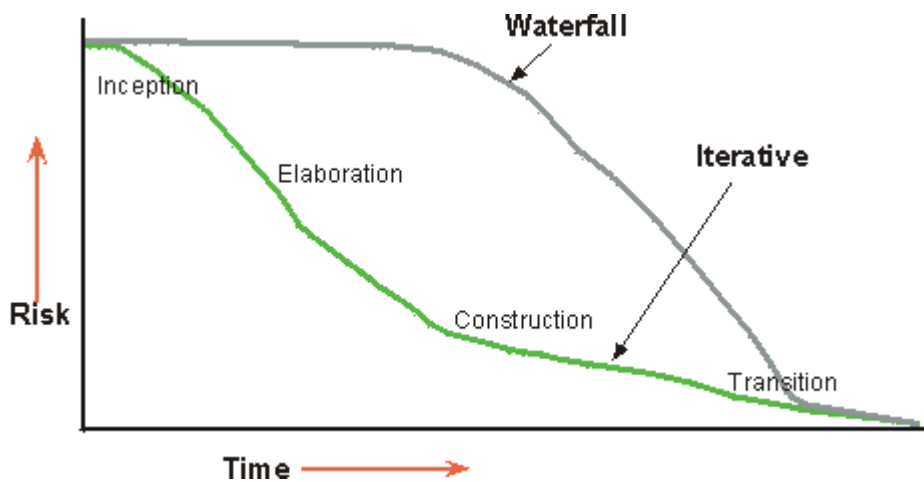
4. Approche itérative



Evolution des risques sans itération:



Evolution comparée des risques via une approche itérative:

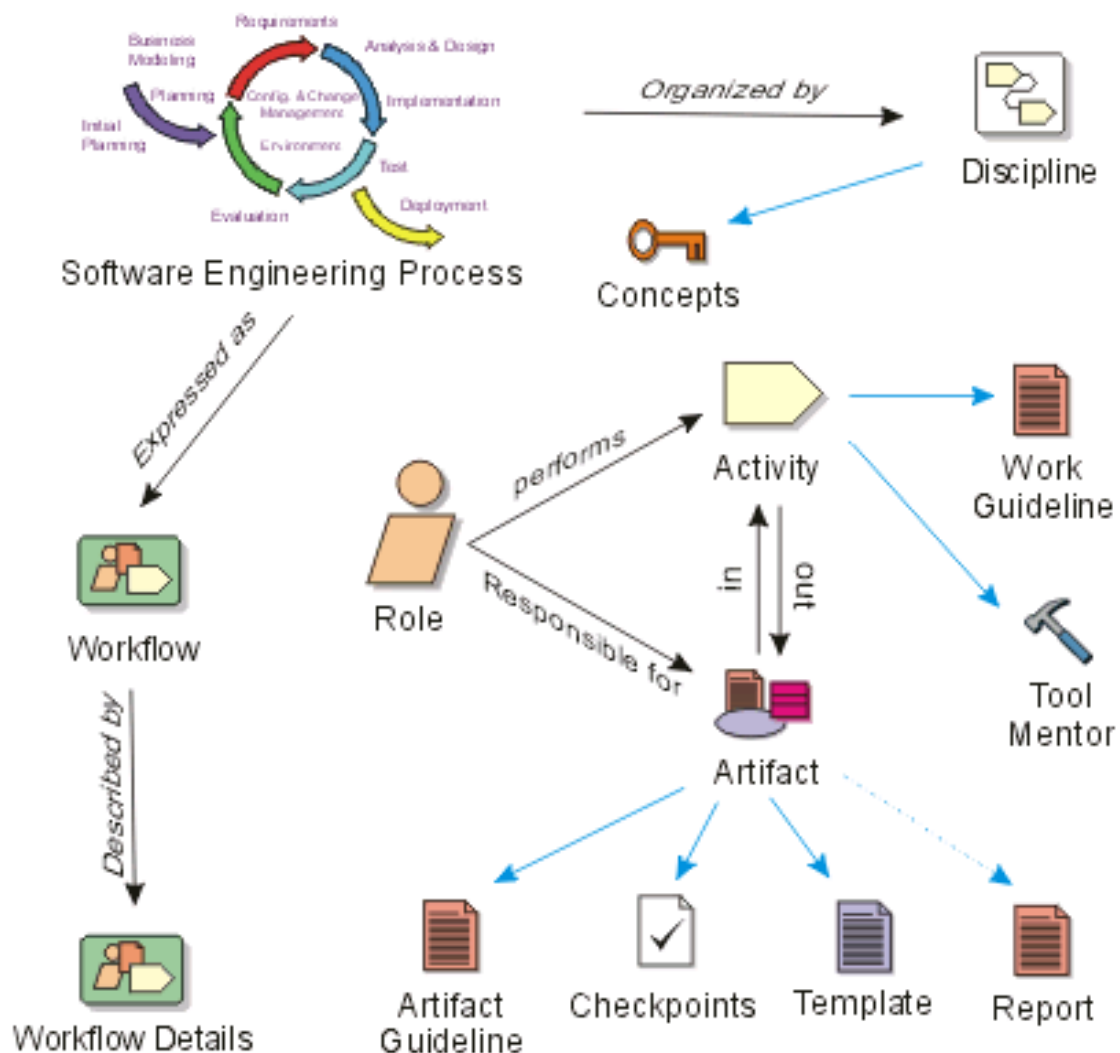


5. Vocabulaire

termes	définitions	exemples
Worker / Role (travailleur/acteur) (fonction)	Rôle ou fonction d'une personne (ou d'un groupe de personnes) travaillant sur un projet.	Concepteur, Architecte, Développeur,...
Activities (activités)	Activité (unité de travail) avec un objectif bien défini.	"Trouver les acteurs et les UC" , "Revoir la conception" , "Effectuer les tests de performance"
Artifact	Document / chose produite	"Modèle" , "Classe" , "Doc" , "Composant" , ...
Workflows	Enchaînements (séquences d'activités réalisées par différents acteurs)	"Capture des besoins" , "Analyse et conception" ,
Disciplines	Collection d'activités ...	Analyse, conception ,

- Qui (Who) ? ==> Workers / Roles

- **Comment (How) ? ==> Activities** (Dans quel contexte ? ==> **Disciplines**)
- **Quoi (What) ? ==> Artifact** (Avec quel outils ? ==> **Tool Mentor**)
- **Quand (When) ? ==> Workflows**



Disciplines de R.U.P. :



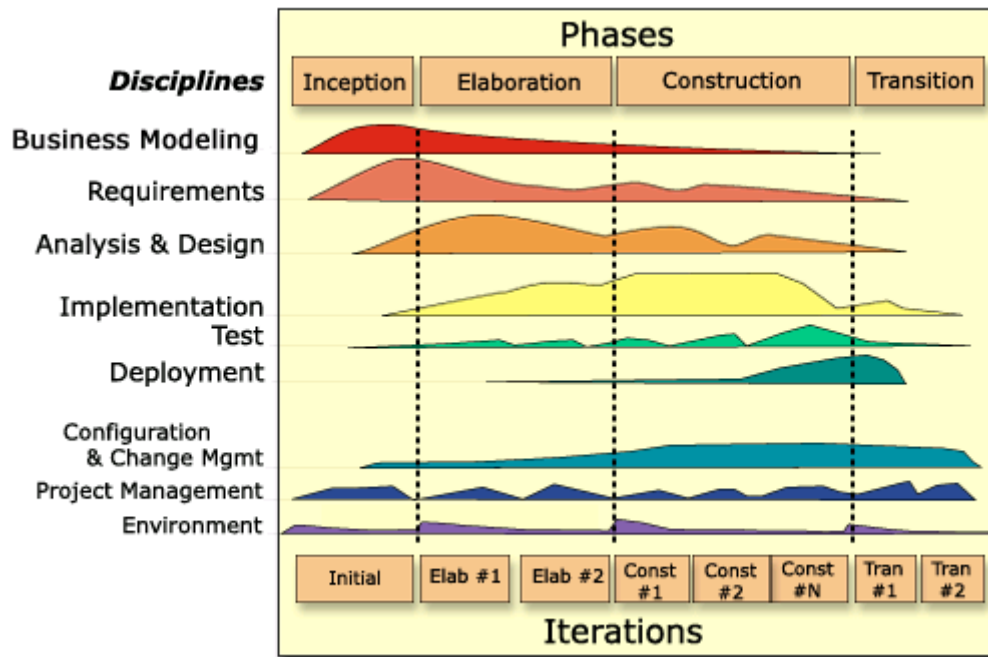
Disciplines de base (Core)

Disciplines d'encadrement

6. 4 grandes phases

L'élaboration d'un logiciel est schématisée par:

- Un ou plusieurs **cycle(s)**. Chaque nouveau cycle correspondant alors à une nouvelle génération du logiciel.
- Un cycle est décomposé en 4 grandes **phases**. La fin d'une phase est délimitée par une **borne** de fin de phase (**milestone**). Chaque borne est franchie via **une prise de décision** (est-on prêt à passer à la suite ?).
- Une phase peut éventuellement conduire à effectuer plusieurs **itérations** sur les différentes activités. Une itération s'achève généralement avec la génération d'un **prototype**.



6.1. Phase de conceptualisation / commencement (inception)

Objectifs:

- Vue d'ensemble sur le projet (besoins , caractéristiques clefs , principales contraintes).
- Une ébauche des diagrammes des Uses Cases (Dégrossi à hauteur de 10% - 25%).
- Un glossaire initial (termes, définitions).
- Une première version des processus métiers (avec analyse marché , prévisions sur les gains ,).
- Une première estimation des risques.
- Une ébauche de la planification du projet (phases, itérations).
- Un ou plusieurs Prototype(s) (éventuellement succin(s)).

Borne de fin de phase:

1. Crédibilité sur l'estimation des coûts , risques et du temps de développement ?
2. Pertinence du prototype (technicité , étendue, ...) ?
3. Compréhension claire des besoins ?
4. ...

6.2. Phase d'élaboration

Objectifs:

- Compléter et détailler le modèle des Uses Cases (ajout de diag. de séquences).
- Modélisation objet (Diagrammes = fruits de l'analyse).
- Description de l'architecture du logiciel (avec conception préliminaire et prototype).
- Une ré-estimation des risques et de la planification du projet.

Borne de fin de phase:

- Vision stable sur l'aspect fonctionnel du produit ?
- Architecture stable (prototype concluant) ?
- Planification suffisamment détaillée et précise ?
- Principaux risques écartés ?
- Accord des commanditaires sur les choix effectués (analyse, architecture) ?

6.3. Phase de construction

Objectifs:

- Concevoir , implémenter et tester le système par parties (itérations / incréments).
- Fabriquer un manuel "utilisateur" et une documentation technique .
- Intégration du logiciel (quasi complet) sur la bonne plate-forme (version bêta).

Borne de fin de phase:

- Le produit est-il assez mature et stable pour être déployé sur les postes des utilisateurs ?
- Commanditaires prêts à accepter le déploiement de la version bêta ?
- ...

6.4. Phase de transition

Objectifs:

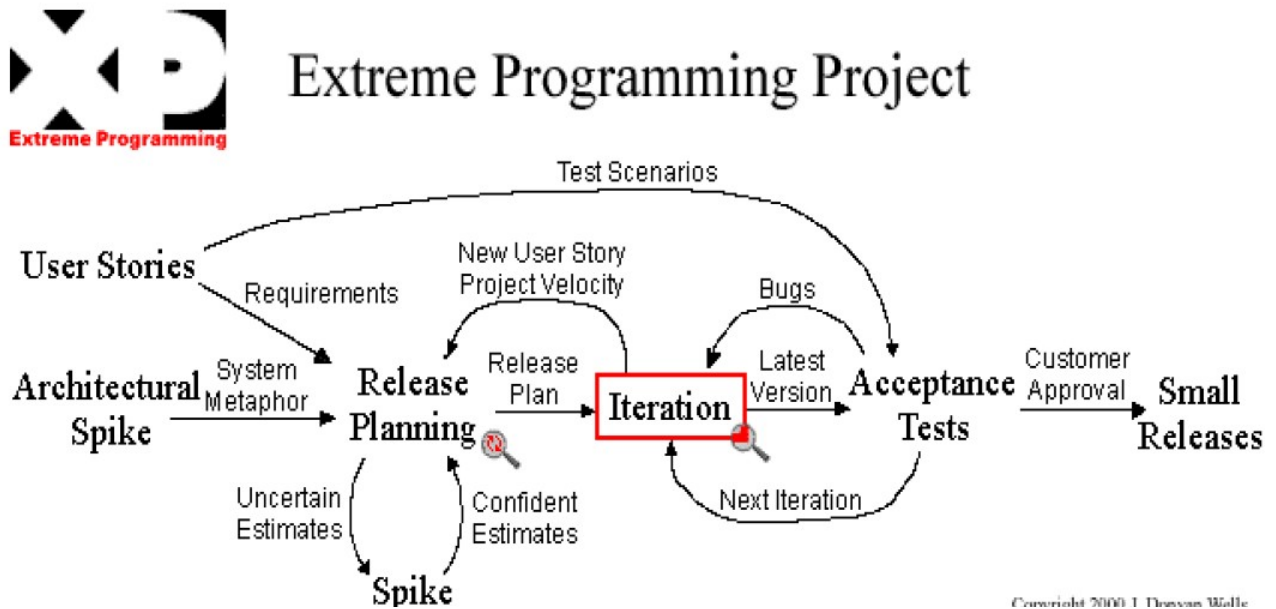
- Feed-back sur la version bêta , correction de bugs.
- Déploiement du nouveau logiciel en parallèle de l'ancien (si possible).
- Optimisations, maintenance applicative.
- Assistance vis à vis des utilisateurs.
- ... Packaging , commercialisation éventuelle, ...

Borne de fin de phase:

- Recette
- Acceptation (et maîtrise) du produit par les utilisateurs ?
- Acheter la version finale dans les délais et avec des coûts acceptables ?

7. XP (Extreme Programming)

Cycle de vie d'un projet XP:



Spike : transitoire , lancement , (==> [Ré-]estimation)

Planification adaptative

A partir d'un **découpage très fin** en "**User Stories**" (sorte de "*sous Use Cases*" auxquels on attache des priorités , des risques et des estimations de temps de développement) , on établit un planning temporaire avec des dates fixes (mais des contenus/livrables variables – pas tout à fait contractualisés)

==> Le planning initial est régulièrement remanié en fonction des :

- imprévus (Pb technique, ...) coté développement
- changements des besoins (nouvelle priorité) coté client

8. 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 !!!)

XXVII - Norme UML et MétaModèle

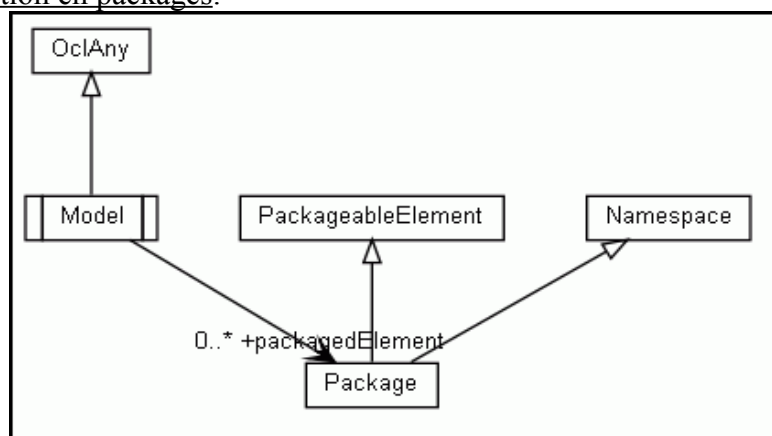
1. Quelques éléments du méta-modèle UML2

Le méta modèle UML est devenu très grand (et complexe) en version 2 .

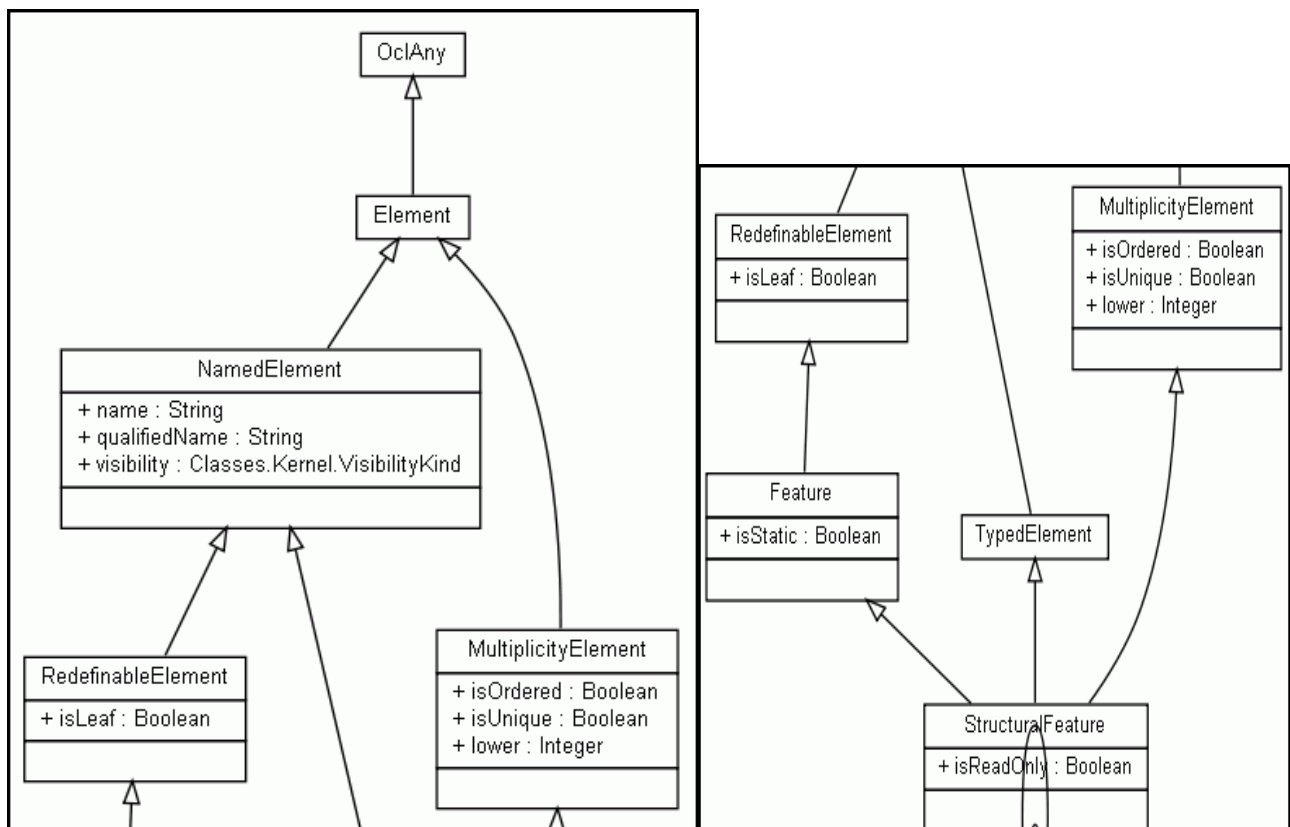
Il existe un programme gratuit "**UML2 MetaModel Viewer**" (de la société "*EmPowerTec*") qui permet de visualiser des parties choisies du méta modèle UML2 .

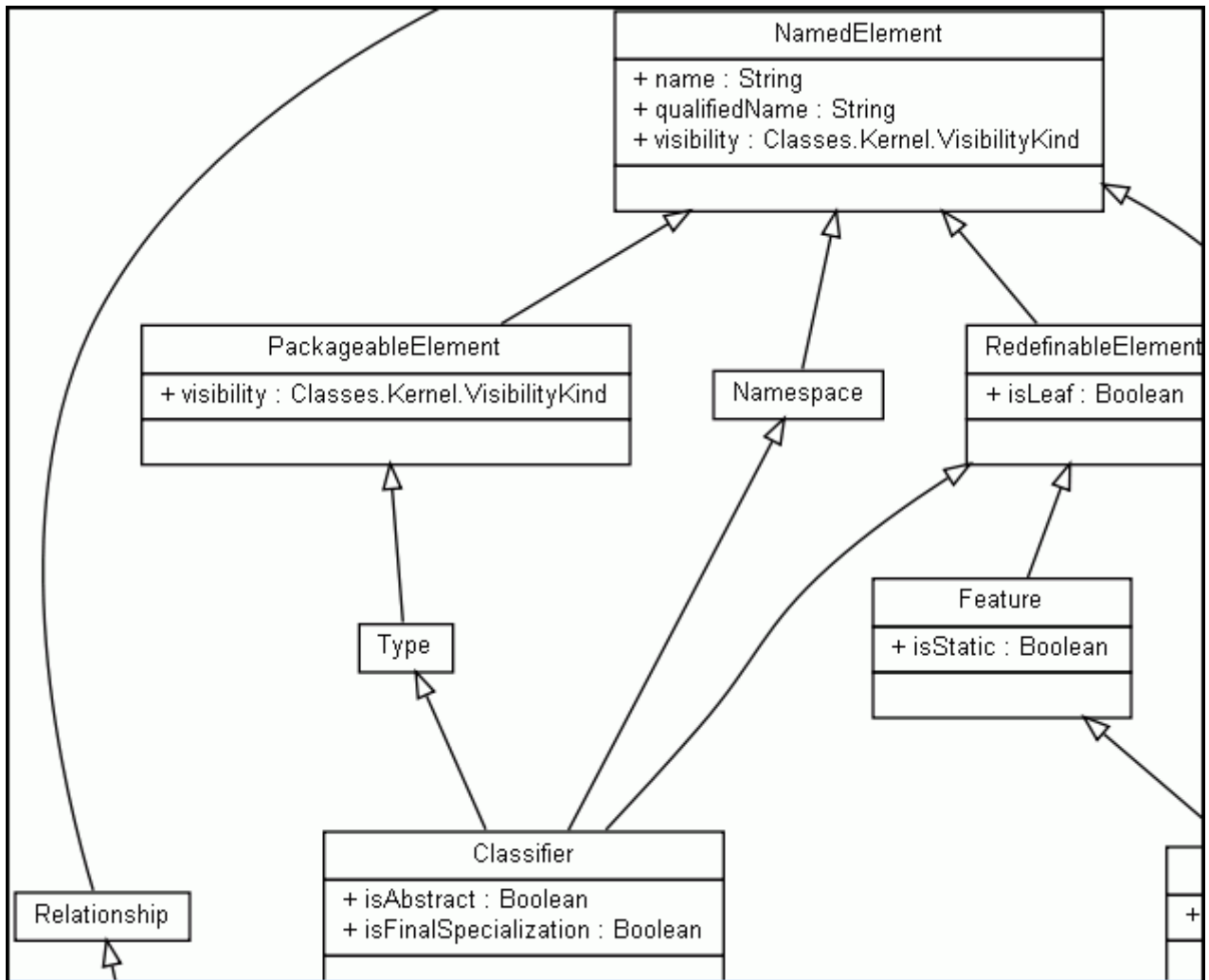
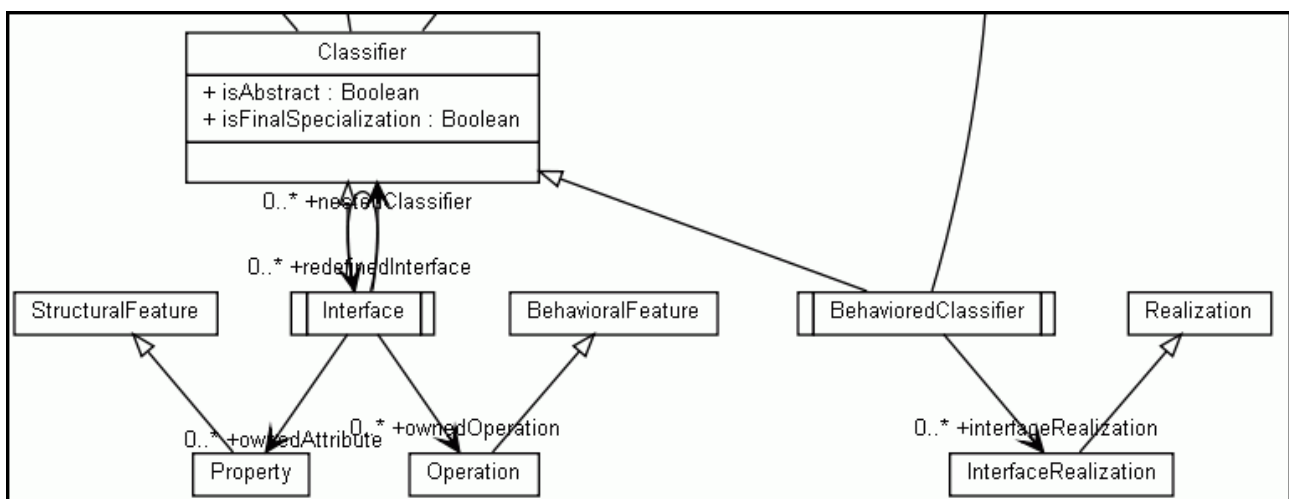
Les diagrammes suivants sont issus de ce produit.

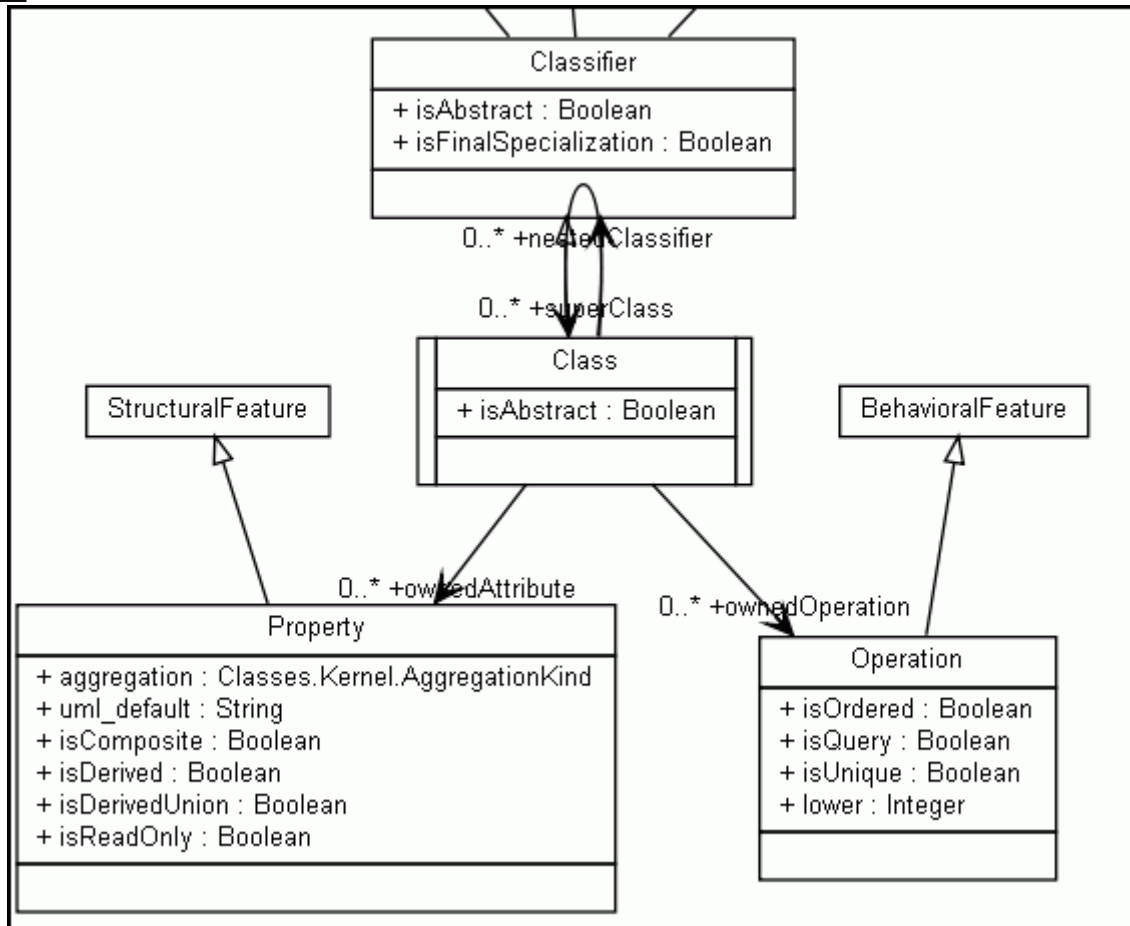
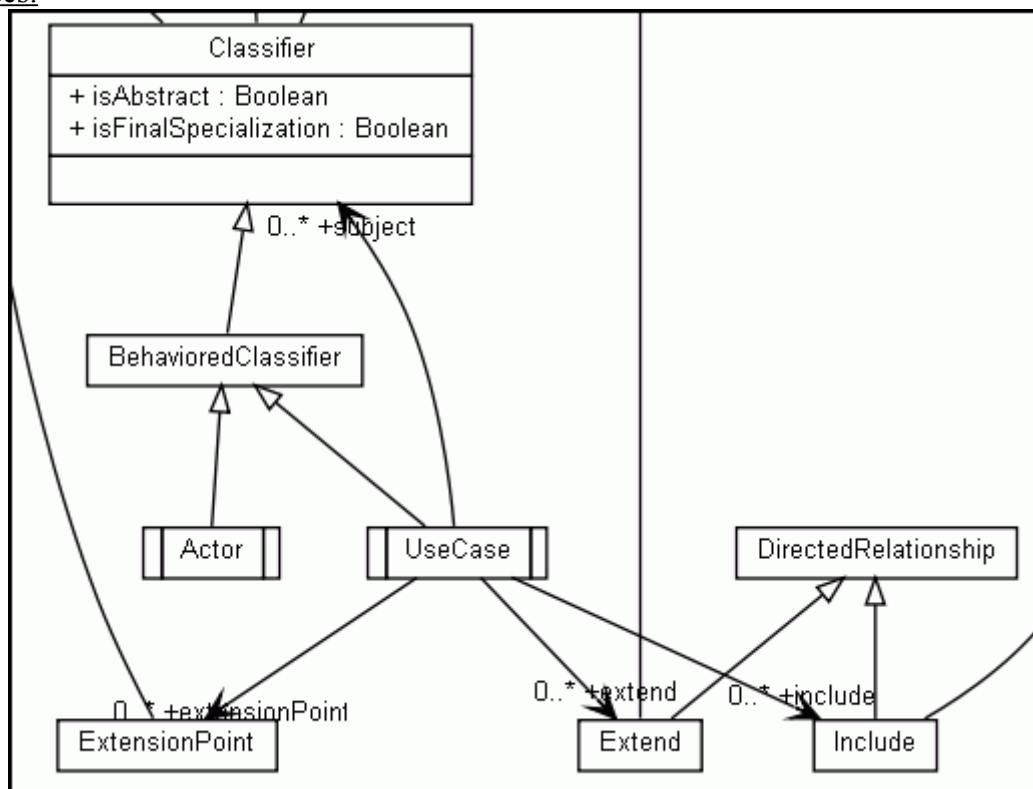
Modèle et organisation en packages:



Eléments et dérivés



Type & ClassifierInterfaces:

Classes:Uses Cases:

etc, etc (le méta-modèle UML2 complet est immense !)

XXVIII - Essentiel outil "Topcased UML"

1. Utilisation de TopCased UML

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

L'ergonomie de Topcased UML est "correcte/bonne" . Il y a mieux et moins bien dans des produits UML concurrents. Après environ une journée de "prise en main", on peut commencer à utiliser efficacement le produit ("*clic,clic*" et "*hop,hop*" plutôt que "*clic,undo*").

Soutenu entre autres par Airbus et Atos-Origin le projet Topcased UML est assez dynamique. Une nouvelle version apparaît régulièrement pour suivre l'évolution du standard UML et de la plateforme ECLIPSE.

L'un des principaux intérêts de l'outil Topcased UML tient dans le fait qu'en tant que plugin bien intégré à eclipse , on peut l'utiliser conjointement avec d'autres plugins importants (ex: accéléo_M2T , gendoc2 , ...).

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

- **créer/paramétrer des modèles UML** (diagrammes avec stéréotypes)
- **(re-)générer automatiquement de la documentation au format ".doc" ou ".odt"** (avec le plugin intégré gendoc2) pour produire des spécifications
- **(re-)générer automatiquement une bonne partie du code de l'application** (avec le plugin "accéléo_M2T" / MDA).

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

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

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

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

URL pour télécharger l'outil "Topcased UML":

<http://www.topcased.org/>



1.1. Intégration/installation de Topcased UML

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

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

- intégration pré-établie (prête à être téléchargée)
---> **Topcased RCP** (*Rich Client Platform*)
- intégration spécifique (à construire soit même en partant d'ECLIPSE et en y ajoutant un à un tous les plugins jugés utiles : Topcased_UML , accéléo_M2T , Maven , ...)

NB:

- Avec la version "pré-intégrée" dite "RCP" , on a déjà de quoi construire les modèles UML et générer la documentation associée (spécifications).
- Pour en plus pouvoir générer du code utile et le tester , il faut tout un tas de plugins supplémentaires (accéléo_M2T, WTP/JEE , Maven , SVN, ...) qui peuvent soit être placés dans un même et seul "gros IDE / super Eclipse customisé" ou bien être placés dans un second ECLIPSE . Si plusieurs "IDE ECLIPSE" sont utilisés, il suffit de recopier les fichiers ".uml" (et facultativement ".umldi") d'un eclipse à l'autre pour récupérer une copie des modèles UML à partir desquels du code doit être généré en mode MDA.

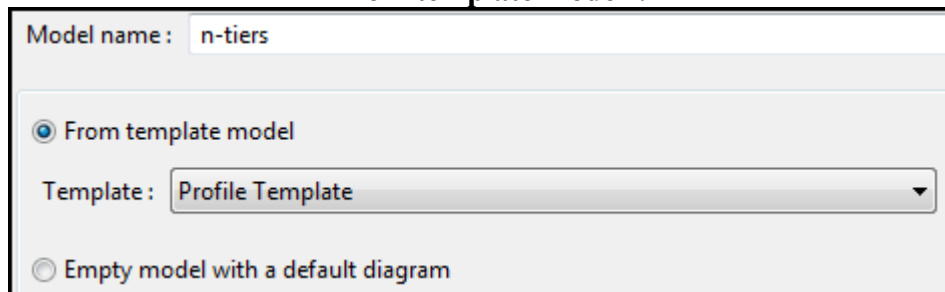
Dans tous les cas, le lancement de Topcased UML s'effectue en démarrant "**eclipse.exe**" (de la version RCP téléchargée ou bien spécifiquement construite).

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

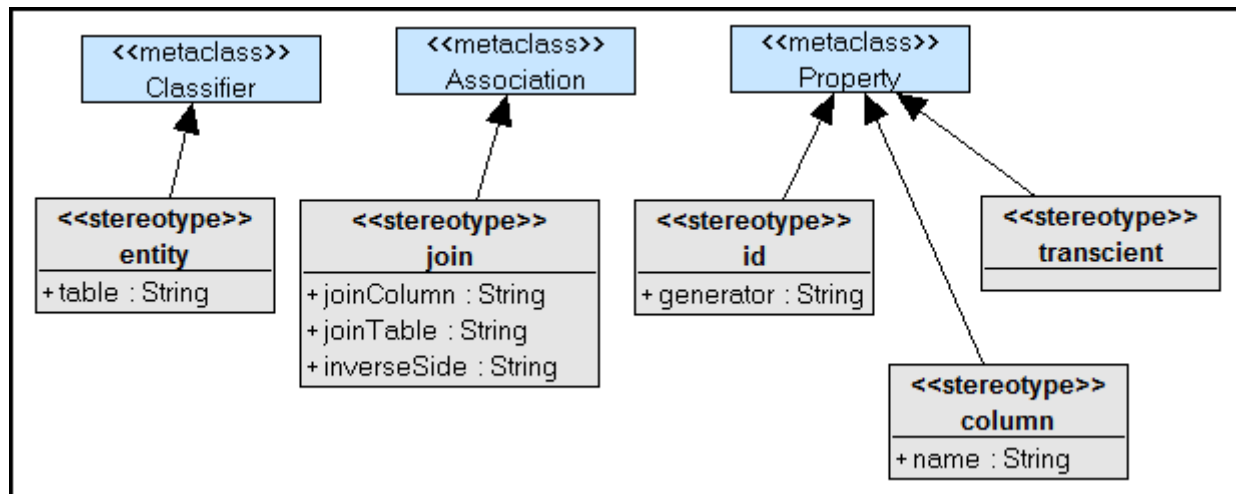
Pour placer des stéréotypes (ex: <<entity>> , <<id>>) dans un modèle UML , il faut d'abord les créer dans un fichier de type "**UML profile**".

Mode opératoire:

- Se placer dans la partie "**Models**" d'un projet eclipse de type "**Topcased project**"
- Créer un nouveau fichier via le menu "**New / UML model with Topcased**" en choisissant bien le nom du profil (ex: "n-tiers" ou "orm") et en sélectionnant "**Profile Template**" de "**From template model**":



- Créer de nouveaux **stéréotypes** (depuis la palette et en renseignant leurs **noms**).
- Placer et paramétrer des "**metaclass**" (ex: "Classifier" , "Property" , ...) .
- Relier les "**stéréotypes**" aux "**metaclass**" via des flèches d'extension pour indiquer "**sur quoi les stéréotypes seront applicables**".
- Bien sauvegarder le fichier généré.



NB: un stéréotype peut éventuellement comporter des propriétés.

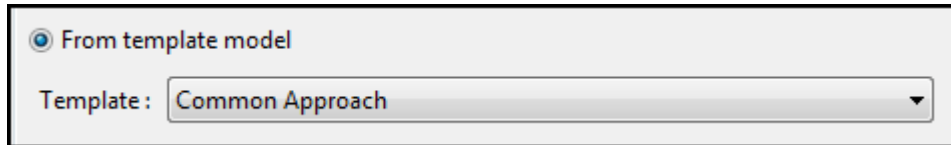
Quelques idées de stéréotypes:

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

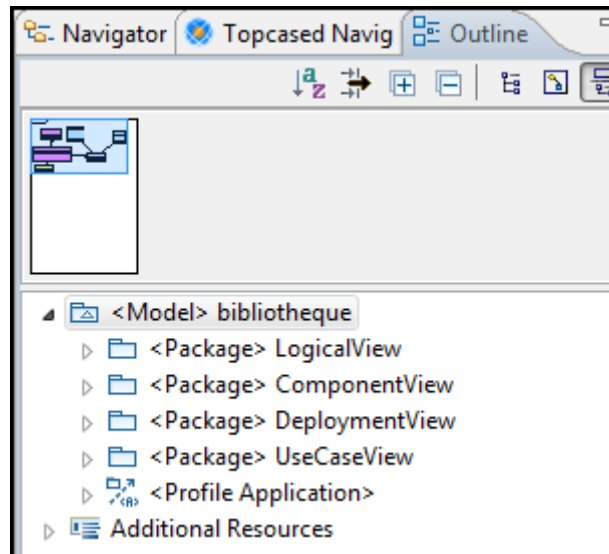
1.3. Création et initialisation d'un modèle UML (pour java)

Mode opératoire:

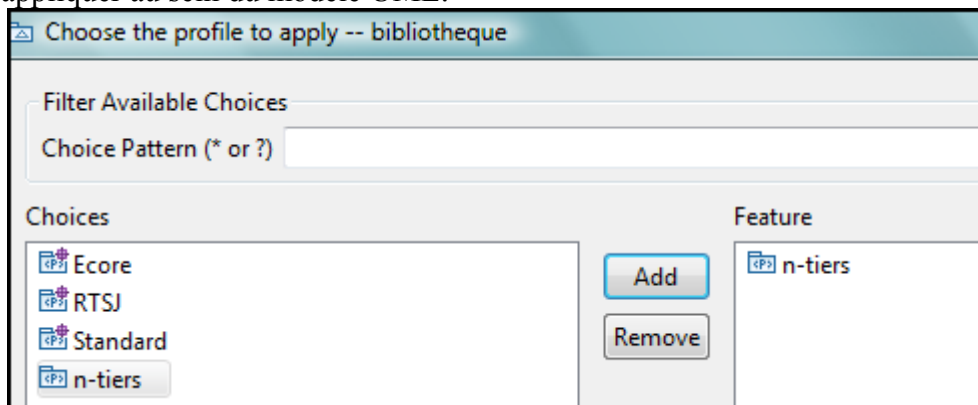
- Se placer dans la partie "**Models**" d'un projet eclipse de type "**Topcased project**"
- Créer un nouveau fichier via le menu "**New / UML model with Topcased**" en choisissant bien le nom du modèle (ex: "bibliotheque" ou "devises") et en sélectionnant "**Common Approach**" de "**From template model**":



- Se placer à la racine du modèle UML dans la vue "**outline**" :



- Activer le menu contextuel "**Load resources...**" et sélectionner/intégrer le (ou les) **fichier(s) de profile(s) UML** (ex: n-tiers.uml). Ceci permet d'enregistrer un chemin relatif entre le fichier du modèle UML et le fichier "profil" d'extension.
- Activer le menu contextuel "**Apply Profile...**" et sélectionner le (ou les) profile(s) à utiliser/appliquer au sein du modèle UML:



NB: "**Apply Profile**" sera éventuellement à re-déclencher ultérieurement s'il faut tenir compte d'une nouvelle version enrichie d'un fichier "UML profile".

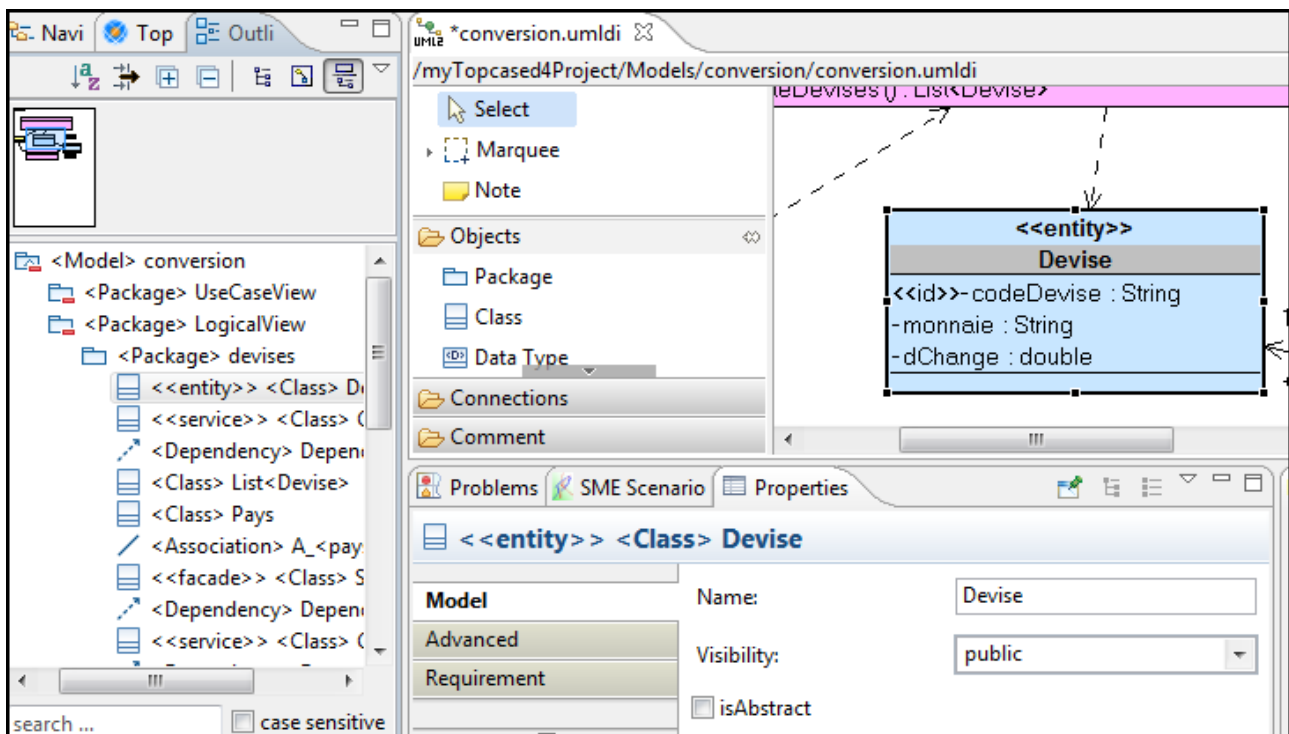
- Activer le menu contextuel "**Generate Primitive Type / Java**" pour générer si nécessaire les types de données élémentaires du langage Java (ex: double , int , ...).
- Bien (re-)sauvegarder le fichier du modèle UML.

1.4. Utilisation générale de l'outil TopCased UML

Chaque modèle UML est sauvegardé dans deux fichiers complémentaires:

- le fichier **".uml"** comporte tous les éléments significatifs du modèle UML (packages , classes ,). C'est à partir de ce fichier que l'on peut extraire les informations essentielles du modèle UML pour générer du code (via un outil MDA tel qu'accéléo_m2t par exemple)
- le fichier **".umldi"** comporte les coordonnées (x,y,...) des éléments internes des diagrammes UML.

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



De façon intuitive, la fenêtre "Outline" permet de naviguer dans l'arborescence du modèle UML, la fenêtre "Properties" permet de fixer les propriétés de l'élément sélectionné, une palette permet de choisir le type d'élément à ajouter au modèle.

Comme dans la plupart des outils UML:

- il y a une distinction importante entre les menus contextuels **"Delete from Model"** (*Shift+Del*) et **"Delete from diagram"** (*Del*).
- La palette est constituée de sous palettes (qui se cachent quelquefois mutuellement)
- Le "glisser/poser" est quelquefois possible.
- Ctrl-Z pour "undo"

Spécifiquement à l'outil "Topcased UML" :

- pas de "clic/glisser/lâcher" mais "clic pour sélectionner" dans la palette puis un "autre clic" pour créer un nouvel élément dans le diagramme à la position voulue.
- pas de menu contextuel "add attribute/property" ni "add operation/method" mais des éléments **"Property"** et **"Operation"** à récupérer dans la palette et à ajouter aux classes du diagramme.
- Souvent besoin de cacher (via Properties/**graphics**/hide) certains éléments secondaires.

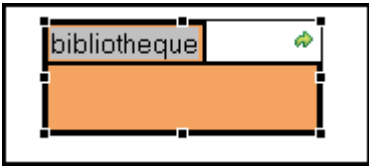
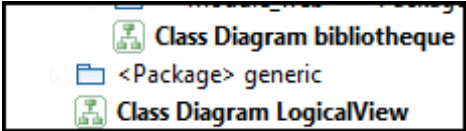
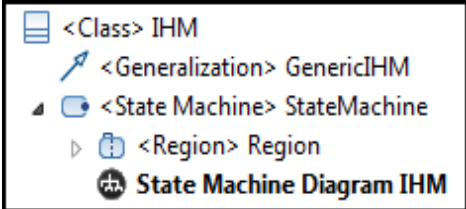
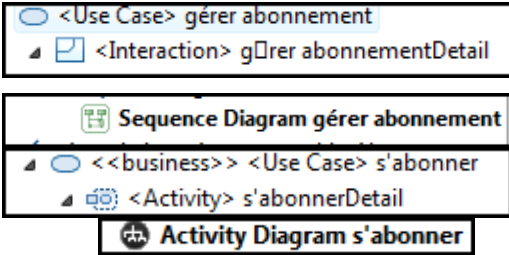
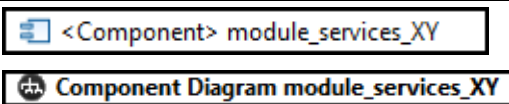
1.5. Organisation conseillée des packages et des diagrammes

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

Mode opératoire conseillé:

- Créer un élément UML (ex: package , classe, ...) dans un diagramme et le paramétrer.
- Effectuer un "double-clic" sur cet élément et ***choisir le type de "sous diagramme" à générer.***
- Par la suite (une fois la création/association effectuée) , un ***double-clic*** ultérieur permettra de ***naviguer d'un niveau vers un sous niveau*** (de détails).

Sous diagrammes classiques pour indiquer les détails:

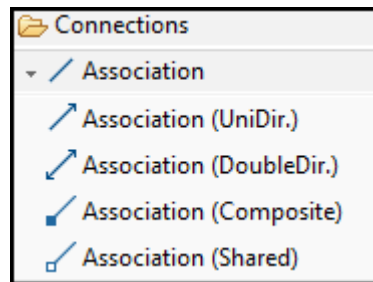
Eléments du modèle UML	diagramme(s) classique(s) associé(s) pour les détails	Exemple(s)
Package	Diagramme de classes (ou ...)	 
Classe	Diagramme d'états (state machine) ou diag. de structure composite	
Use Case	diagramme d'activités et/ou diagrammes de séquences	
Composant	Diagramme de (sous)composants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

Remarques importantes:

- Pour bien organiser un modèle UML, il faut réfléchir le plus tôt possible à la décomposition en différents packages.
- Il est toujours possible de renommer ou déplacer un package (après coup / par la suite).

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

Sous palette "connections" avec **plusieurs variantes pour les associations**:



Dans la fenêtre des propriétés:

* Paramétrer une opération (nom, type de retour,)

Model	Name:	getDeviseOfPays
Parameters	<input checked="" type="checkbox"/> Return Type:	<<entity>> <Class> Devise
Signature Parser		

* Paramétrer les paramètres d'une opération

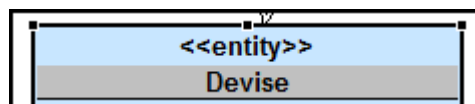
Model	Direction	Name	Type		Add
Parameters	return	return	Devise		Delete
Signature Parser	in	codePays	String		
Graphics					
Advanced					
Behavior					
Requirement					
Stereotypes					

Details of the Parameter	
Name :	codePays
Type :	<Primitive Type> String

```
+ getDeviseOfPays (codePays : String) : Devise
```

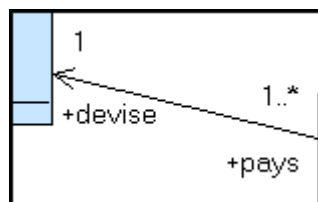
* Choix d'un (ou plusieurs) stéréotype(s) à appliquer:

Model	Available Stereotypes		Applied Stereotypes
Graphics	<div> <div><<S></div> <div>n-tiers::service</div> </div> <div> <div><<S></div> <div>n-tiers::ihm Bean</div> </div> <div> <div><<S></div> <div>n-tiers::stateless</div> </div> <div> <div><<S></div> <div>n-tiers::stateful</div> </div> <div> <div><<S></div> <div>n-tiers::facade</div> </div>	<div>Add ></div> <div>< Remove</div>	<div><<S></div> <div>n-tiers::entity</div>
Advanced			
Requirement			
Stereotypes			



* Paramétrer les extrémités ("first end", "second end") d'une association:

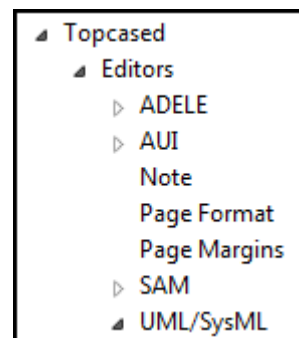
Model	Classifier:	Pays
First End	Role:	pays
Second End	<input type="checkbox"/> isNavigable	
Graphics	Association Type:	none
Advanced	Property Visibility:	public
Requirement	Multiplicity	
Stereotypes	Lower Bound:	1
Stereotype Attributes	Upper Bound:	*
Owned Rules		



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

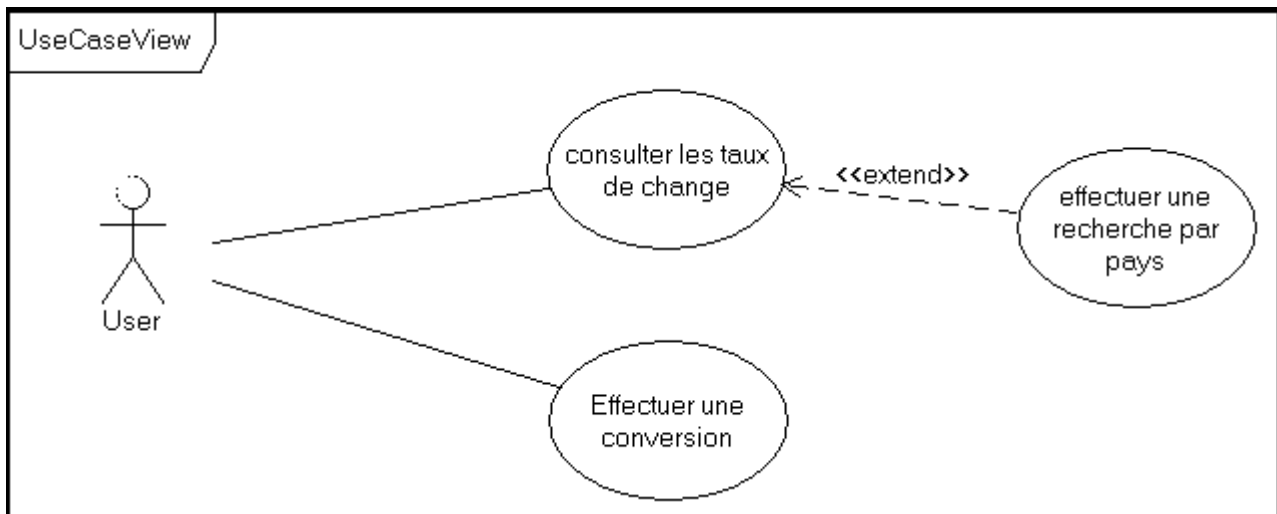
Graphics	Router:	Oblique
Advanced	Visible Elements	Hidden Elements
Requirement	Aa srcNameEdgeObject Aa srcPropertiesEdgeObject Aa srcCountEdgeObject Aa targetNameEdgeObject Aa targetPropertiesEdgeObject Aa targetCountEdgeObject Aa stereotypeEdgeObject	Aa middleNameEdge
Stereotypes	Hide >	< Show
Stereotype Attributes		
Owned Rules		

1.7. Préférences/options sur l'éditeur topcased UML



Depuis Navigator , racine du projet , clic droit / "**Properties**" et

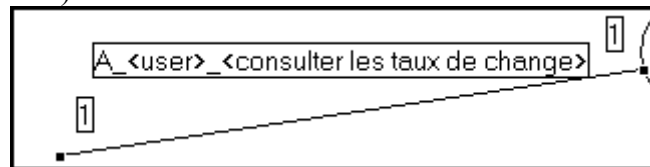
1.8. Edition d'un diagramme de Use Cases (spécificités)



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



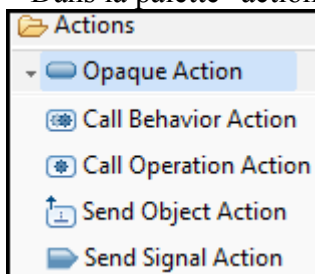
Si tout est visible (et illisible):



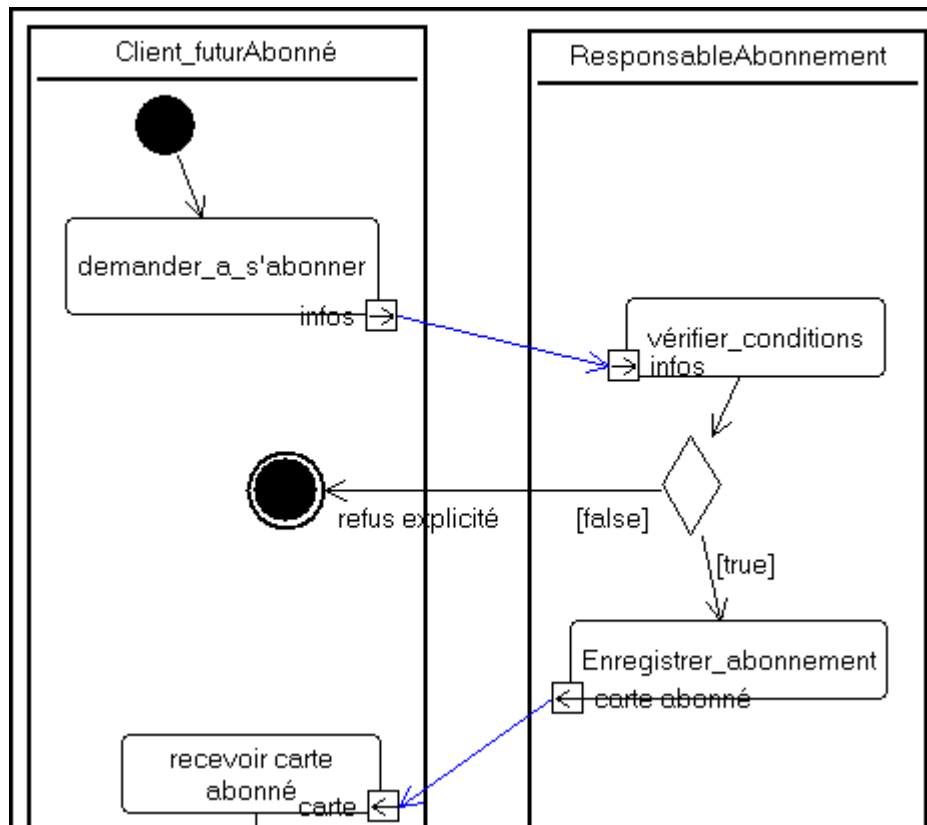
En cachant certaines parties non significatives , c'est mieux (plus lisible) !

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

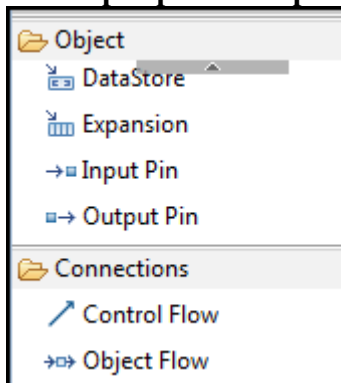
* Dans la palette "actions" , plusieurs variantes d'actions:



NB: Une "opaque action" pourra ultérieurement être transformée en un type d'action plus précis via le menu contextuel "**transform into ...**" de l'arborescence (outline).



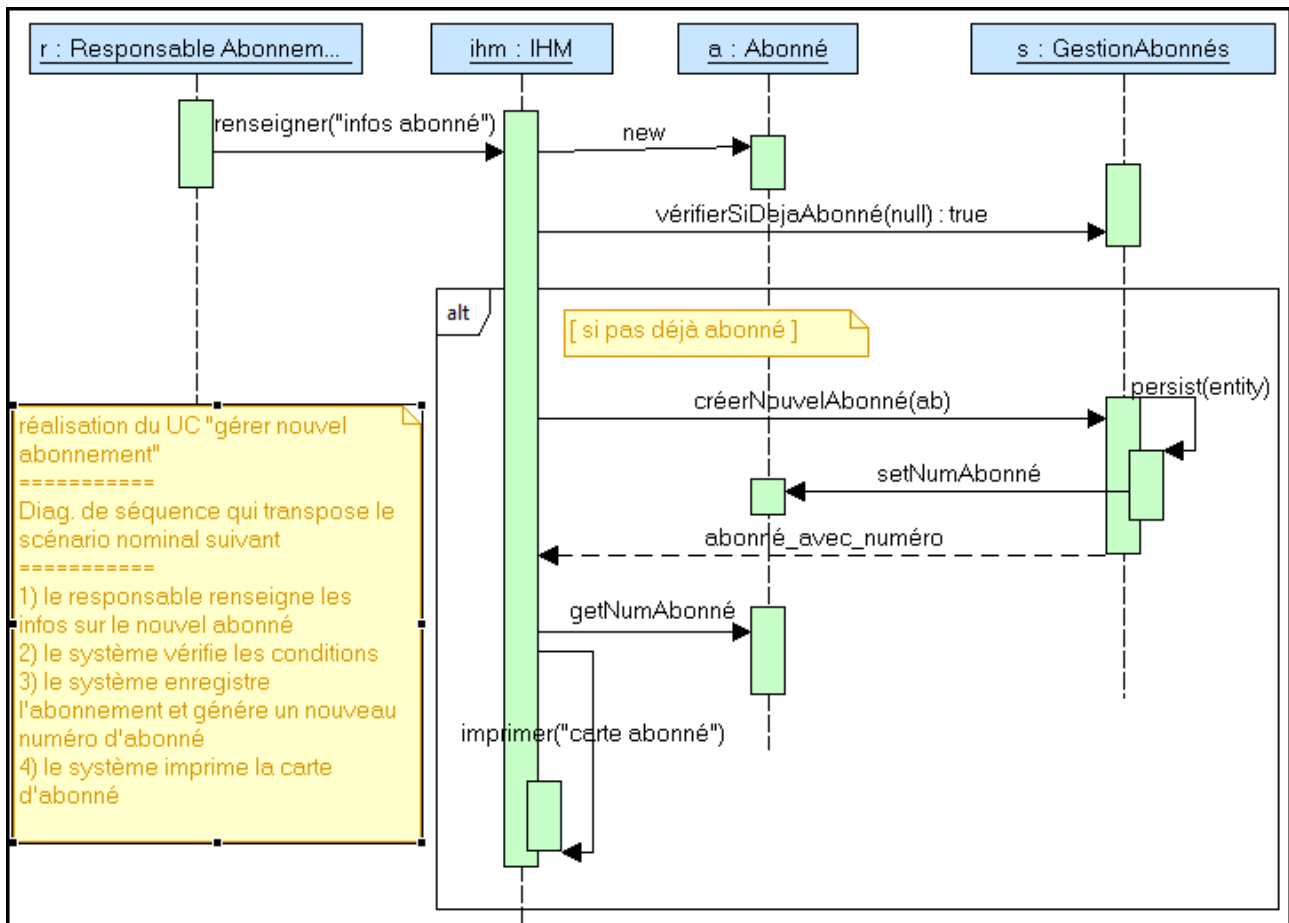
* Dans les versions très récentes d'UML , les "objects flow" nécessitent la mise en place préalable de "output pin" et "input pin" sur les actions:



-----> Une connexion de type "Object flow" pourra être directement placée entre un "output pin" et un "input pin" ou pourra mener vers un éventuel intermédiaire de flux de données (ex: DataStore , CentralBuffer, ...).

NB: Comme toujours, utiliser la sous partie "graphics" de la fenêtre des propriétés permet de cacher les éléments non significatifs et de gagner en lisibilité.

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



Mode opératoire:

- créer un diagramme de séquence (en tant que détail d'un "use case")
- placer des **lignes de vie ("LifeLine")** et préciser les **types représentés** (acteurs, classes, ...)

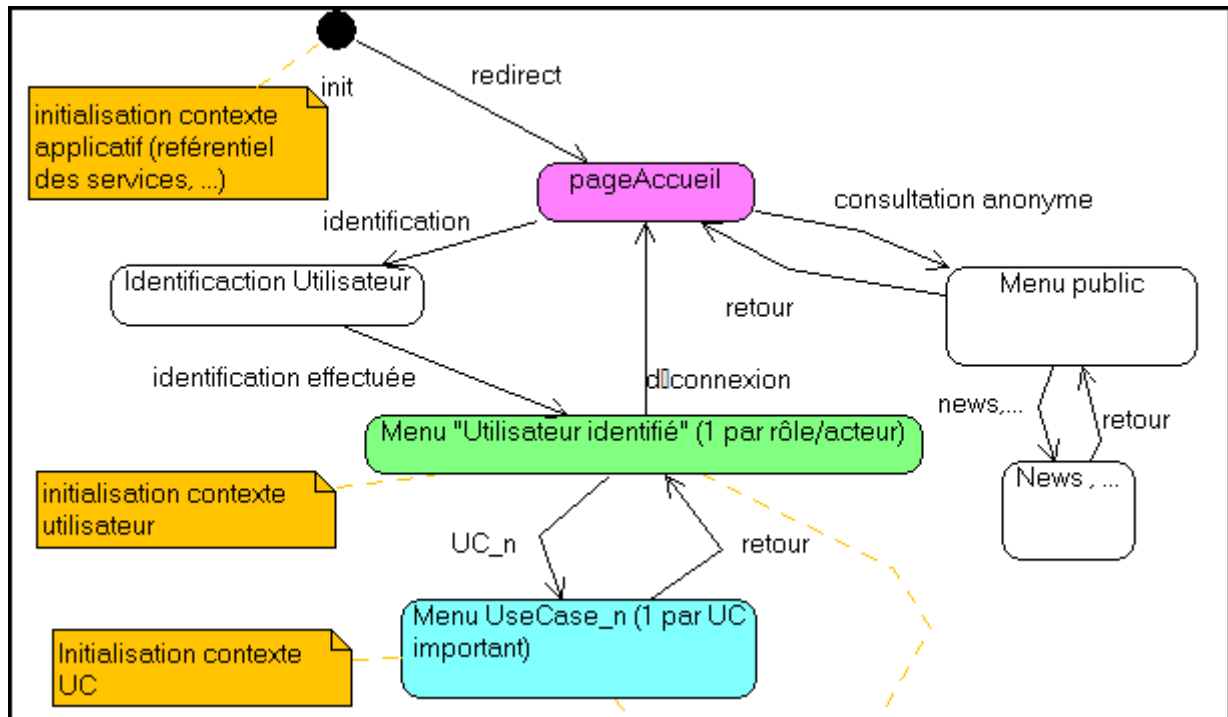
Model	Name:	a
Graphics	Visibility:	public
Advanced	Represents:	<<entity>> <Class> Abonné
Requirement		

- placer des **blocs d'exécution** sur les lignes de vie
- placer des **messages** entre un bloc d'exécution et un autre
- **paramétrer** les messages (*saisir un nom ou bien sélectionner une opération disponible au niveau de type d'objet qui reçoit le message*)
- si une opération sélectionnée en tant que message envoyé/reçu/traité comporte des paramètres de types simples (String, ...), on peut alors spécifier les valeurs de ces paramètres

Operation:	<Operation> renseigner (quelquechose : String)		
	Property Name	Type	Value
	quelquechose	String	infos abonné

NB: On peut également placer des fragments combinés (avec mot clef "alt" , "opt" , "loop" , ...) d'UML2 . Cependant, l'outil Topcased 4.2.1 ne gère pas encore parfaitement tous les paramétrages nécessaires.

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



* Paramétrage interne d'un état

Model	Name:	NEW_ONE
Internal Transition	Visibility:	public
Graphics	Entry Behaviour:	
Advanced	Exit Behavior:	
Requirement	Do Activity:	<Activity> saisir valeurs initiales
Stereotypes		
Stereotype Attributes		



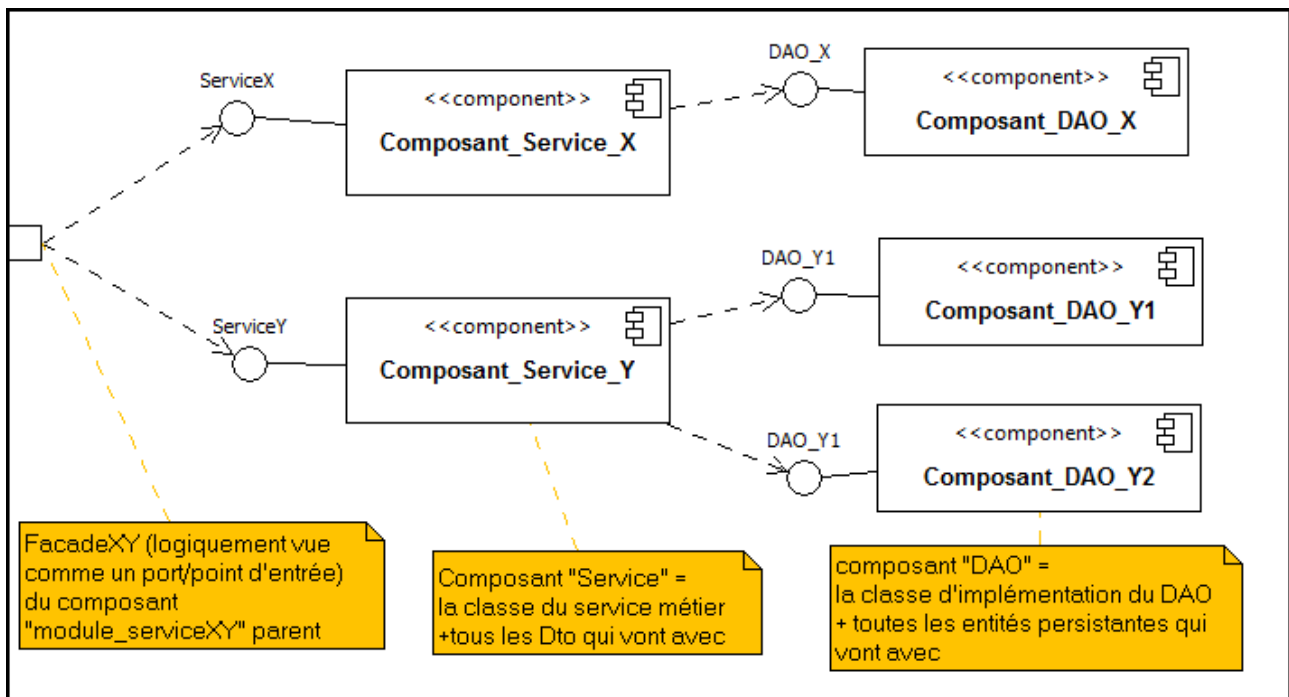
NB: Dans une version récente du produit (4.2.1), le choix d'une action/activité/traitement/comportement à associer à une activité via (entry: , do: , exit:) passe par une sélection d'une chose qui doit avoir été préalablement créée.

Mode opératoire (indirect) et qui fonctionne à peu près:

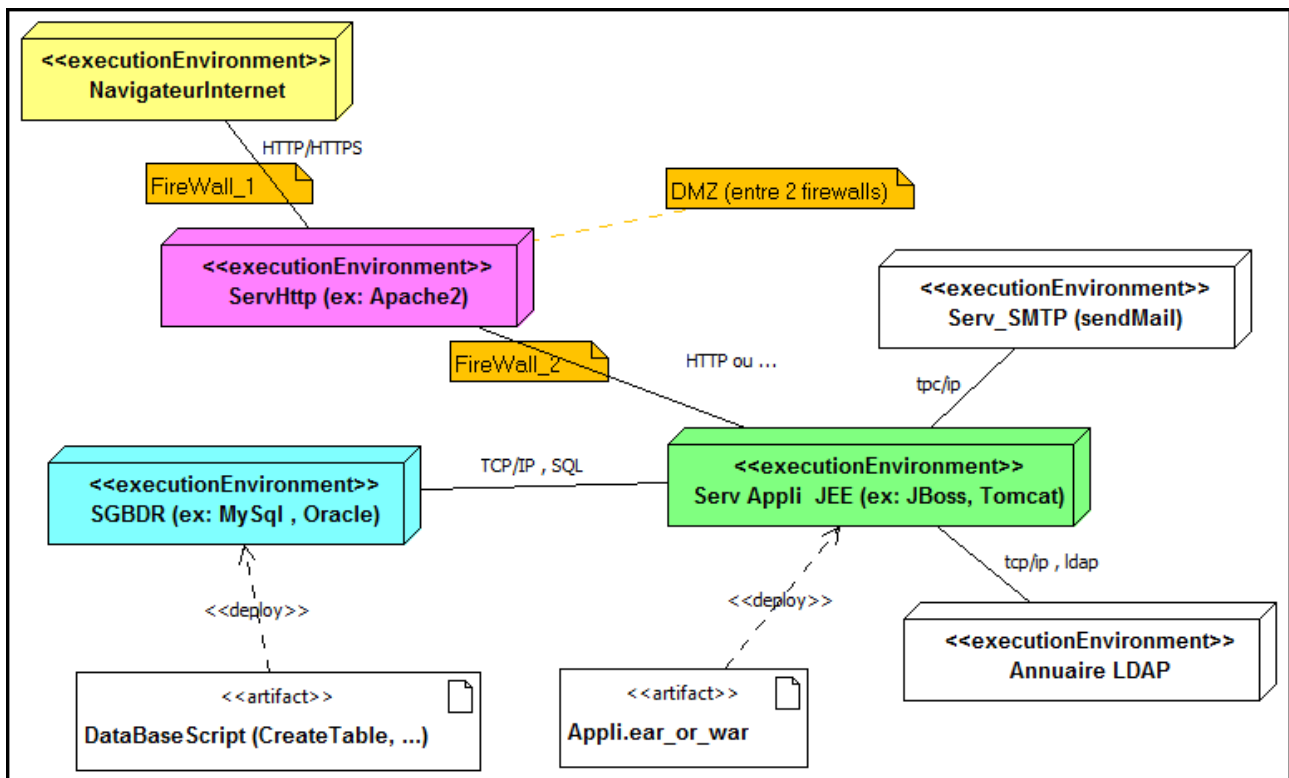
- se placer sur niveau parent "StateMachine" et créer un nouvel élément de traitement via le menu contextuel "Create child / Owned behaviour / Opaque Behaviour"
- renommer cet "opaque behaviour" avec un nom explicite d'action/activité
- retourner sur le paramétrage d'un des états du diagramme d'états.
- Sélectionner l'élément "behaviour" en tant qu'action à associer (entry/do/exit).

NB: En suivant ce procédé on remarque que l'élément "behaviour" à rattacher est devenu une simple chaîne de caractères après une fermeture/réouverture du modèle (.umldi, .uml).

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



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



Remarque: au sein de la version récente (4.2.1), les multiplicités fixées (1, *, ...) ne s'affichent malheureusement pas (<hide> par défaut).

1.14. Glisser/poser d'un élément externe pour y faire référence

En effectuant un **glisser/poser** d'une classe Cx d'un package p1 vers le diagramme de classes du package p2 **tout en maintenant enfoncée la touche "CTRL"** on y introduit **une référence externe avec l'indication (from p1)** qui sera "sans détails/sans contenu".

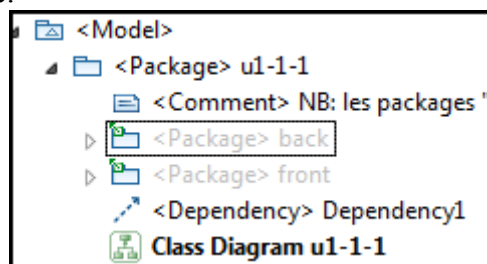
NB: Si l'on n'enfonce pas la touche "CTRL" durant le glisser/poser, on récupère alors tous les attributs et toutes les opérations comme détails dans une copie "Cx dans p2" malheureusement pas automatiquement synchronisée avec l'original "Cx de p1" lorsque sa structure évolue.

1.15. Décomposition d'un modèle UML en plusieurs sous fichiers

Attention: Cette **fonctionnalité avancée** n'est utile que sur un vrai projet de taille importante.
Les chemins relatifs doivent rester stables

	<p>Mode opératoire (dans un niveau "xxx"):</p> <ol style="list-style-type: none"> 1) créer un sous niveau yyyy (package yyy) 2) "double-clic" pour créer le diagramme de classes qui va avec 3) sélectionner le sous package yyyy et clic droit "Control" ajouter éventuellement le sous répertoire "yyy" devant yyy.uml (platform:/resource/tests/Models/yyyy.uml --> platform:/resource/tests/Models/yyyy/yyyy.uml) <p>----> résultat : le fichier xxx.uml (et son frère xxx.umldi) ont été modifiés et la sous partie "yyy" sélectionnée a été déplacée dans les nouveaux fichiers yyy/yyyy.uml (et yyy/yyyy.umldi).</p> <p>On peut ainsi directement travailler sur le fichier yyy.umldi si on ne travaille que sur cette partie. --> On peut envisager plusieurs versions de la sous partie "yyy". Si l'on ouvre le fichier de niveau principal "xxx.umldi", les modifications apportées à la partie yyy sont alors automatiquement stockées dans les sous fichiers "yyy" (en suivant les liens).</p>
--	---

Les parties annexes (dans d'autres fichiers) apparaissent en gris mais la navigation (essentiellement en mode lecture) reste possible.



2. Génération de documentation (gendoc2)

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

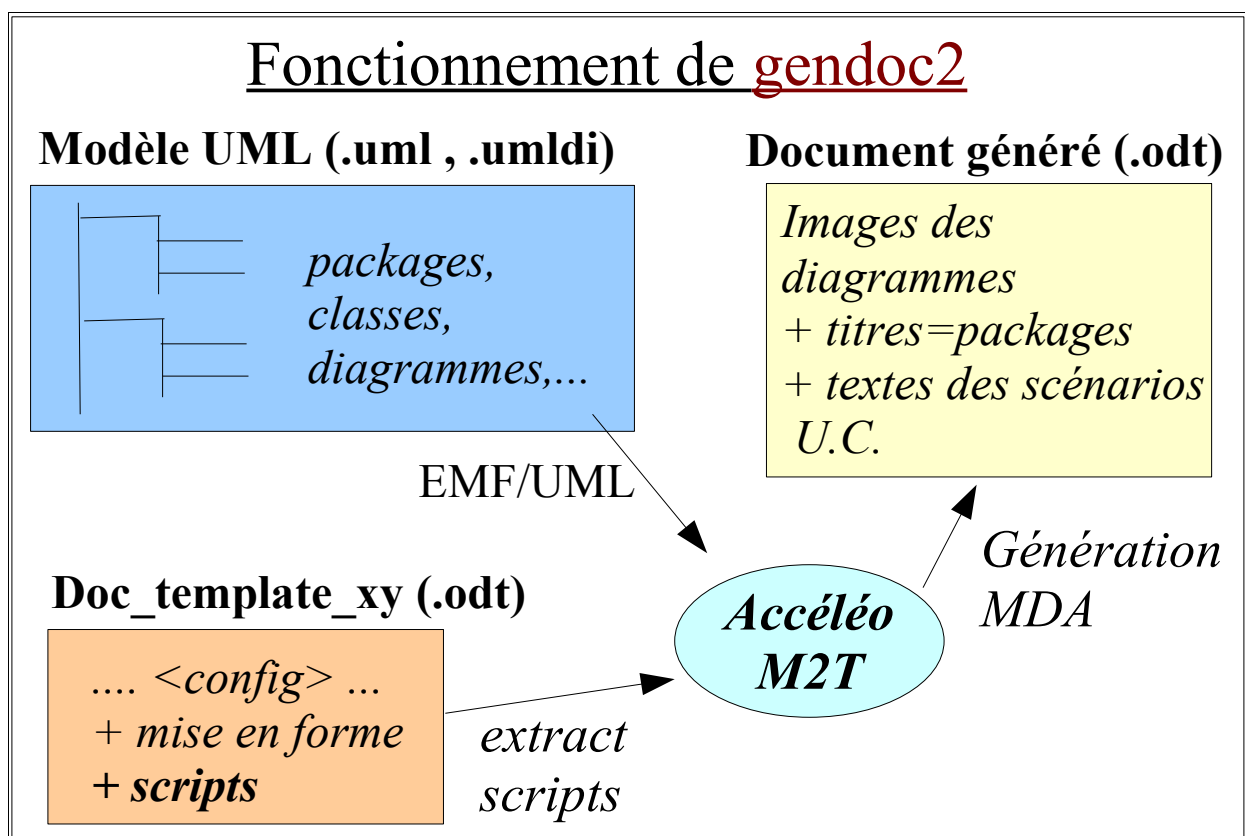
NB: à partir des formats ".odt" ou ".docx" , il est assez facile de **générer une version ".pdf"** .

Gendoc2 est déjà **intégré à Topcased_RCP 4.2.1** et prêt à l'emploi.

L'ancienne version "gendoc" utilisait en interne une ancienne version du générateur MDA "accéléo"
La nouvelle version "gendoc2" utilise en interne la nouvelle version 3 d'accéléo (accéléo_M2T).

NB: En combinant des **fichiers générés par gendoc2** avec des **fichiers statiques** , on peut assez rapidement produire des **spécifications** assez complètes de bonnes qualités et toujours cohérentes avec les dernières versions des modèles .

2.1. Principe de fonctionnement de gendoc2



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

- **se plaçant sur un fichier modèle "doc_template.odt"**
- **activant le menu contextuel "generate documentation"**.

NB2: le fichier généré a souvent besoin d'être réparé après avoir été généré --> répondre "oui" à "voulez vous réparer le document" .

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

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

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

Syntaxe et exemple:

```
<config>
<param key='workspace'
  value='D:\tp\Modelisation_UML\UML\Modeles_Topcased_rcp\myTopCasedWorkspace' />
<!-- UPDATE WORKSPACE , PROJECT , MODEL LOCATION -->
<param key='project' value='${workspace}\myTopcased4Project' />
<param key='appName' value='bibliotheque' />
<param key='model' value='${project}\Models\${appName}\bibliotheque.uml' />
<output path='${project}/Documentation/${appName}/Generated/analyseUml.odt' />
</config>
<context model='${model}' importedBundles='topcased' searchMetamodels='true'/>
```

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

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

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

2.3. Généralités sur les scripts de gendoc2

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

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

Ces instructions entre [] servent essentiellement à :

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

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

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

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

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

Exemple:

```
<context element='bibliotheque/UseCaseView' />
```

Expression des besoins fonctionnels (Uses Cases) ici en texte caché (open office ou word)

```
<gendoc>
```

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

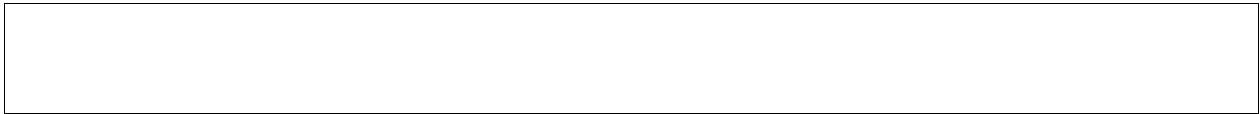
```
U.C. "[uc.name/]"
```

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

```
[/for]
```

```
[for (inter:Interaction|self.ownedElement->filter(Interaction) )]
```

```
[if (inter.getAllDiagrams()->size() > 0)]<image object='[inter.getDiagram()]' keepW='true'>
```



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

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

2.4. Scripts avec images/diagrammes

```
<gendoc>
```

```
[if (self.getAllDiagrams()->size() > 0)]<image object='[self.getDiagram()/]' keepW='true'>
```

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

```
</image>[/if]
```

```
[for (p:Package|self.ownedElement->filter(Package))]
```

```
[p.name/]
```

```
[if (p.getAllDiagrams()->size() > 0)]<image object='[self.getDiagram()/]' keepW='true'>
```



```
</image>[/if]
```

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

2.5. Document maître pour fédérer plusieurs fichiers générés

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

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

Ceci permet en outre :

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

XXIX - Annexe – Bibliographie, Liens WEB , outils

1. Bibliographie et liens vers sites "internet"

Site de référence (OMG)	http://www.uml.org/
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

2. Quelques outils UML (Editeurs , AGL)

Outils/AGL UML	Editeur	Open Source ?	Caractéristiques
Rational Rose --> Rational XDE ----> RSM (Rational Software Modeler)	Rational ----> IBM	non	Très bon produit (très complet) mais assez cher.
Together	--> Borland	non	?
Poseidon UML for Java .	Gentleware	non	version de base presque gratuite (anciennement gratuite) basée sur Argo UML . Bonnes fonctionnalités . Ergonomie moyenne / correcte.
Star UML		oui	Produit gratuit assez complet (très inspiré de Rational Rose) . <u>Avantage</u> : très bien dans l'état (intuitif , facile à utiliser) <u>Défaut</u> : n'a pas évolué depuis 2005 évolura plus (car développé dans ancien langage "Delphi").
Enterprise Architect	Sparx	non	assez complet , à tester
MagicDraw UML	MagicDraw	non	intègre très bien les normes récentes
Embarcadero	???	non	???
Objectteering UML	Softeam (fr)	non	ergonomie très moyenne
Eclipse UML	Omondo	oui	plugin UML pour eclipse (bien)
Topcased UML	Airbus ,	oui	Plugin UML pour eclipse (très bien mais un peut sembler compliqué au départ)
...			