MDE : **M**odel **D**riven **E**ngineering
IDM : **I**ngénierie **D**irigée par les **M**odèles

Younes LAKHRISSI

Plan du cours

- **Chapitre 1 :** Introduction et Contexte de l'IDM
- **Chapitre 2 : OCL** (Object Constrainte Language)
- **Chapitre 3 :** Méta-Modélisation
  - Notions de Modèle(M1) ; Méta-modèle(M2) ; Méta-méta-modèle(M3)
  - Extension du méta-modèle d'UML
  - TP 1
- **Chapitre 4 :** MDA (Model Driven Architecture)
  - Principes : CIM, PIM, PSM.
  - Outils MDA
- **Chapitre 5 :** Transformation de Modèles
  - Principe
  - Langages de transformation
  - TP2, TP3 et TP4

Chapitre I : Introduction

Contexte de l'IDM

Plan du chapitre I

- Chapitre I : Contexte de l'IDM

  - **1 - État des lieux et constats**
    - Types de logiciel
    - Nature du logiciel
    - Critères de la qualité logicielle

  - **2 - Génie Logiciel**
    - Objectifs
    - Technologies de production logiciel au cours de l'histoire

  - **3 - Limites du développement logiciel actuel**
    - Limites de la technologie Objet
    - Les nouvelles exigences

  - **4 - L'Ingénierie des Modèles : une nouvelle philosophie de production logiciel**
    - Définitions
    - Vision générale des standards de l'OMG Object Management Group

Les catégories de logiciel (1/3)

- 1 – Les systèmes embarqués
  - Systèmes exécutés dans du matériel électronique isolé
    - machine à laver, télévision, lecteur DVD, téléphone mobile, magnétoscope, four à micro-ondes, réfrigérateur, lecteur MP3, ...
  - Difficile à modifier
- 2 - Les systèmes temps réel
  - Réaction immédiate requise
  - Systèmes de contrôle et de surveillance
  - Manipulent et contrôlent le matériel technique
  - Environnement contraignant
- 3 - Les systèmes de matériel
  - Systèmes d'exploitation, …

Les catégories de logiciel (2/3)

- 4 - Les systèmes de traitement de données
  - Ils stockent, recherchent, transforment et présentent l'information aux utilisateurs
  - Grandes quantités de données avec des corrélations complexes, enregistrées dans les bases de données
  - Largement utilisés en administration des affaires
  - Fiabilité des résultats
  - Sécurité dans l'accès aux données
- 5 - Les systèmes distribués
  - Synchronisent la transmission, assurent l'intégrité des données et la sécurité, ...
  - Technologies utilisées : CORBA, DOM/DCOM/.NET, SOAP, EJB, …

Les catégories de logiciel (3/3)

- 6 - Les systèmes d'entreprise
  - Décrivent les buts, les ressources, les règles et le travail réel dans une entreprise
- 7 - Générique
  - Vendu sur le marché
    - un tableur, un outil de base de données
    - un outil de traitement de texte
    - …
- 8 - Sur mesure
  - Pour un client spécifique
- 9 - …

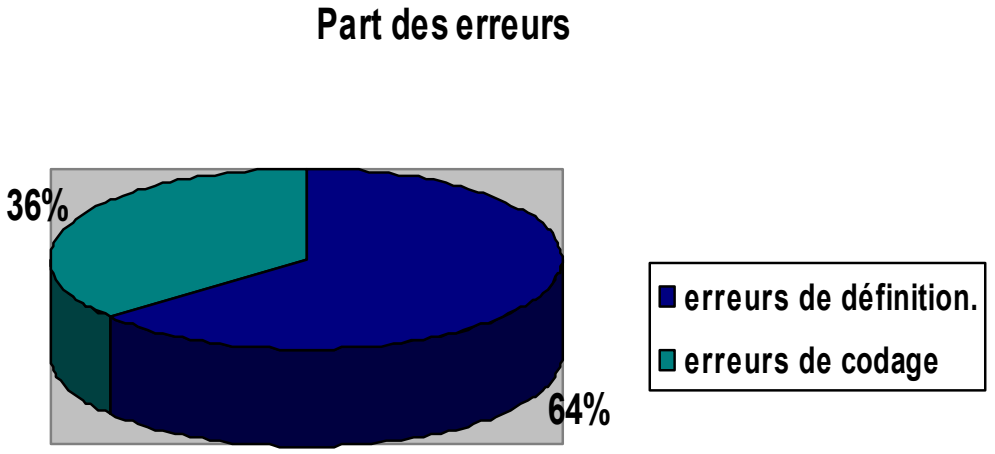
Les critères de qualité du logiciel

A considérer dès la phase de modélisation

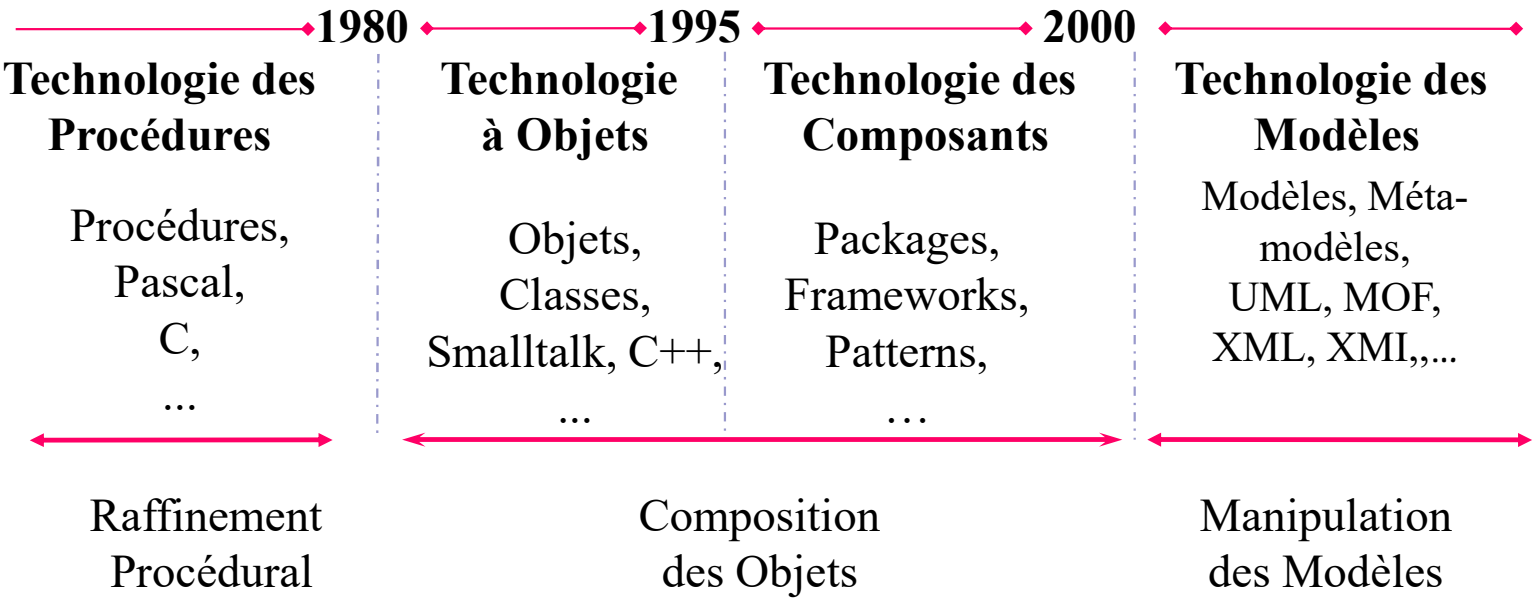
- **Validité** : Conformité d'un logiciel avec sa spécification
- **Robustesse** : Capacité à fonctionner même dans des conditions anormales
- **Extensibilité** : Facilité d'adaptation à des changements de spécifications
- **Réutilisabilité** : Capacité à être réutilisé en tout ou partie dans une nouvelle application
- **Compatibilité** : Facilité avec laquelle des composants logiciels peuvent être combinés
- **Efficacité** : Utilisation optimale des ressources matérielles
- **Portabilité** : Facilité de transfert dans différents environnements
- **Vérifiabilité** : Facilité à valider ; bien structuré et modulaire
- **Intégrité** : Aptitude à protéger les codes et les données
- **Ergonomie** : Facilité d'utilisation, Facilité d'apprentissage, bien documenté

Quelques statistiques

- Étude du gouvernement américain en 1979
  - Logiciels payés mais jamais livrés 40%
  - Logiciels livrés mais jamais utilisés 30%
  - Logiciels utilisés après modification 25%
  - Logiciels utilisés tel quel 5%



Évolution des technologies



Abandon de la Technologie Objets pour la Technologie Modèles (J. Bézivin)

Limites de la programmation objet (1/3)

- Les systèmes deviennent plus complexes
- Volume croissant
  - Des données
  - Du code
- Évolutivité croissante
  - De la partie métier (mondialisation, concentration, restructuration, …)
  - De la partie plate-forme d'exécution
- Hétérogénéité croissante
  - Des langages et des paradigmes
  - Des supports de données et des protocoles d'accès
  - Des systèmes et des plates-formes
  - Des technologies
- Le rythme d'arrivée des nouvelles technologies s'accélère
- Les vieilles technologies ne meurent pas, elles se cachent

Limites de la programmation objet (2/3)

- Spécifications mouvantes
  - Evolution des besoins, de la législation, des groupements de sociétés, des technologies, …)
- Besoin important d'Intégration des applications existantes
- Intégration de propriétés non-fonctionnelles complexes (sécurité, persistance, traçabilité, …)
- Multiplicité des points de vue sur le logiciels
  - Expression des besoins, analyse, conception, implantation, validation, évolution, maintenance, clients, …
- La complexité a atteint un tel niveau qu'il est hors de portée d'avoir une **vision globale** sur un système en évolution

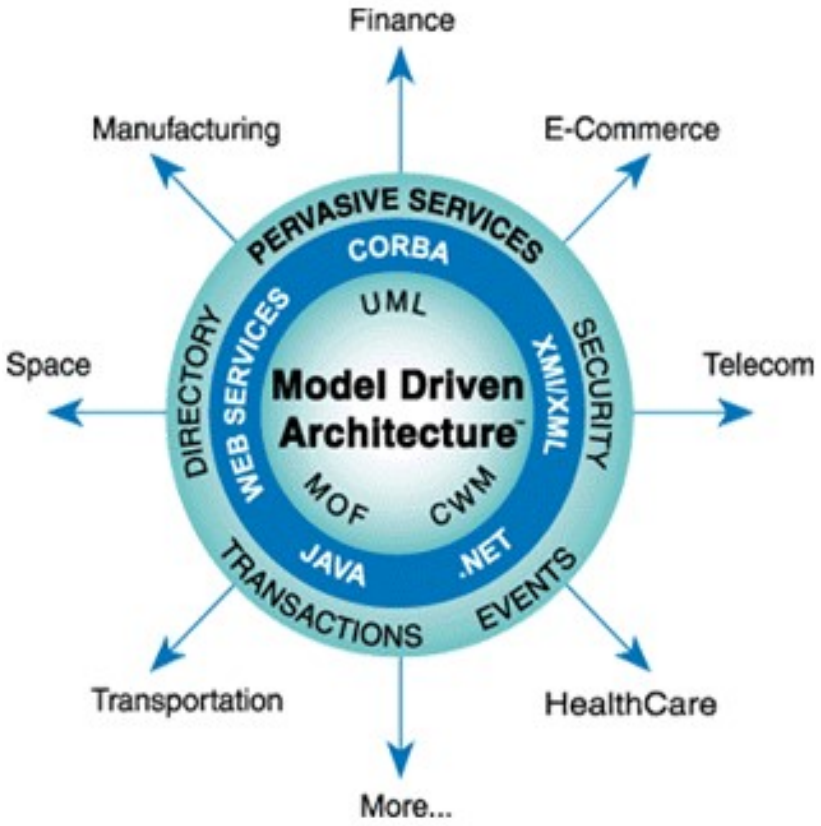
OMG : Object Management Group

- Parlons les mêmes langages et partageons les mêmes technologies !
  - → Les standard de l'OMG : UML, XMI, CWM, CORBA, IDL, …
- Expression de contraintes sur les modèles UML
  - → OCL
- Adaptabilité, extension d'UML
  - → Méta-modélisation
  - → Profils UML
- Interopérabilité
  - → MDA (CIM, PIM, PSM)
  - → QVT
- Réutilisation des solutions
  - → Composants, l'architecture CORBA

MDA : Vers l'interopérabilité

- Il n'y aura pas de consensus sur
  - Les plates-formes "hardware"
  - Les « operating systems »
  - Les protocoles network
  - Les langages de programmation
- Une solution : Se mettre d'accord à un niveau plus haut ! … MDA
- Considérer les services de modélisation
  - Une transformation est un service de modélisation au même titre que l'exécution de modèles ou la génération de tests
  - Interopérabilité de services

OMG - MDA


Finance
Manufacturing
E-Commerce
Space
Telecom
Transportation
HealthCare
More...
PERVASIVE SERVICES
CORBA
UML
Model Driven Architecture
MOF
CWM
JAVA
.NET
TRANSACTIONS
EVENTS
SECURITY
XMI/XML
WEB SERVICES
DIRECTORY

L’approche MDA

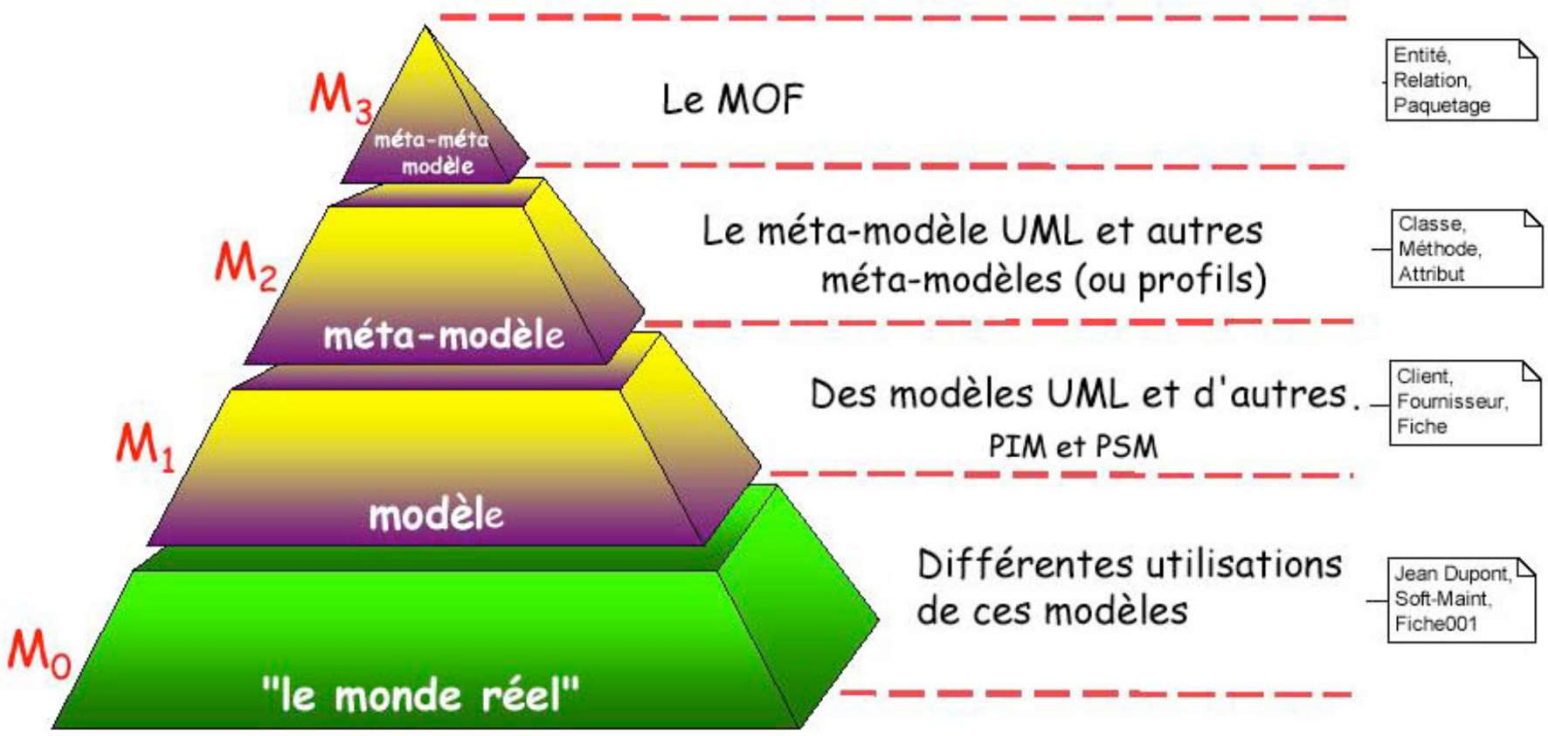
- Description
  - Le MDA est l'outil qui permet à une industrie de décrire ses fonctions indépendamment des implémentations
  - Penser l'application au niveau du modèle et laisser le soin de l'implémentation aux outils
  - Interopérabilité au niveau des modèles : Il s'agit d'avoir la possibilité d'écrire et de faire évoluer le modèle en fonction de l'organisation métier de l’application et non plus par les plate-formes.
    - Au niveau de l'organisation : PIM (Plateform Independant Model)
    - Au niveau des plate-formes : PSM (Plateform Specific Model).

L’approche MDA

- Une application complète de MDA : un PIM et un ou plusieurs PSM
  - Langage de description du MDA : UML.
  - L'application sera ensuite implémentée sur un large éventail de systèmes.
  - Les PSM peuvent communiquer entre eux en faisant intervenir plusieurs plate-formes pour échanger des données (CORBA par exemple)

- Avantages du MDA :
  - une architecture basée sur MDA est prête pour les évolutions technologiques.
  - plus grande facilité d'intégration des applications et des systèmes autour d'une architecture partagée.
  - une interopérabilité plus large permettant de ne pas être lié à une plate-forme.

Architecture à quatre niveaux



Chapitre II : OCL

Object Constraint Language

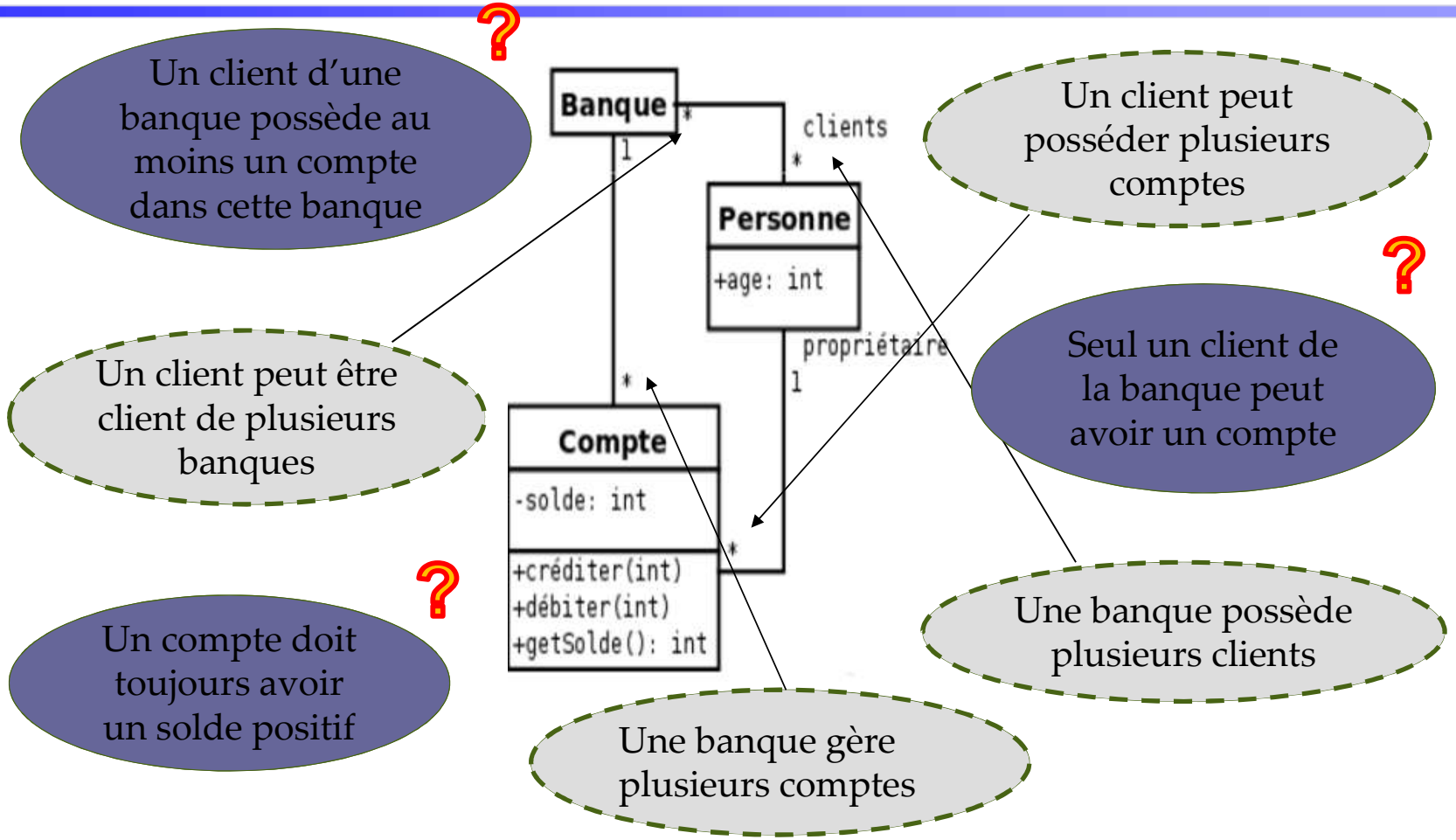
Contraintes UML

- Avec UML, on peut exprimer certaines formes de contraintes :
  - Contraintes de type : typage des propriétés, …
  - Contraintes structurelles :
    - les attributs dans les classes,
    - les différents types de relations entre classes (généralisation, association, agrégation, composition, dépendance),
    - la cardinalité et la navigabilité des propriétés structurelles,
    - …
  - Contraintes diverses : les contraintes de visibilité, les méthodes et classes abstraites, …
- Dans la pratique, toutes ces contraintes sont très utiles mais se révèlent insuffisantes.

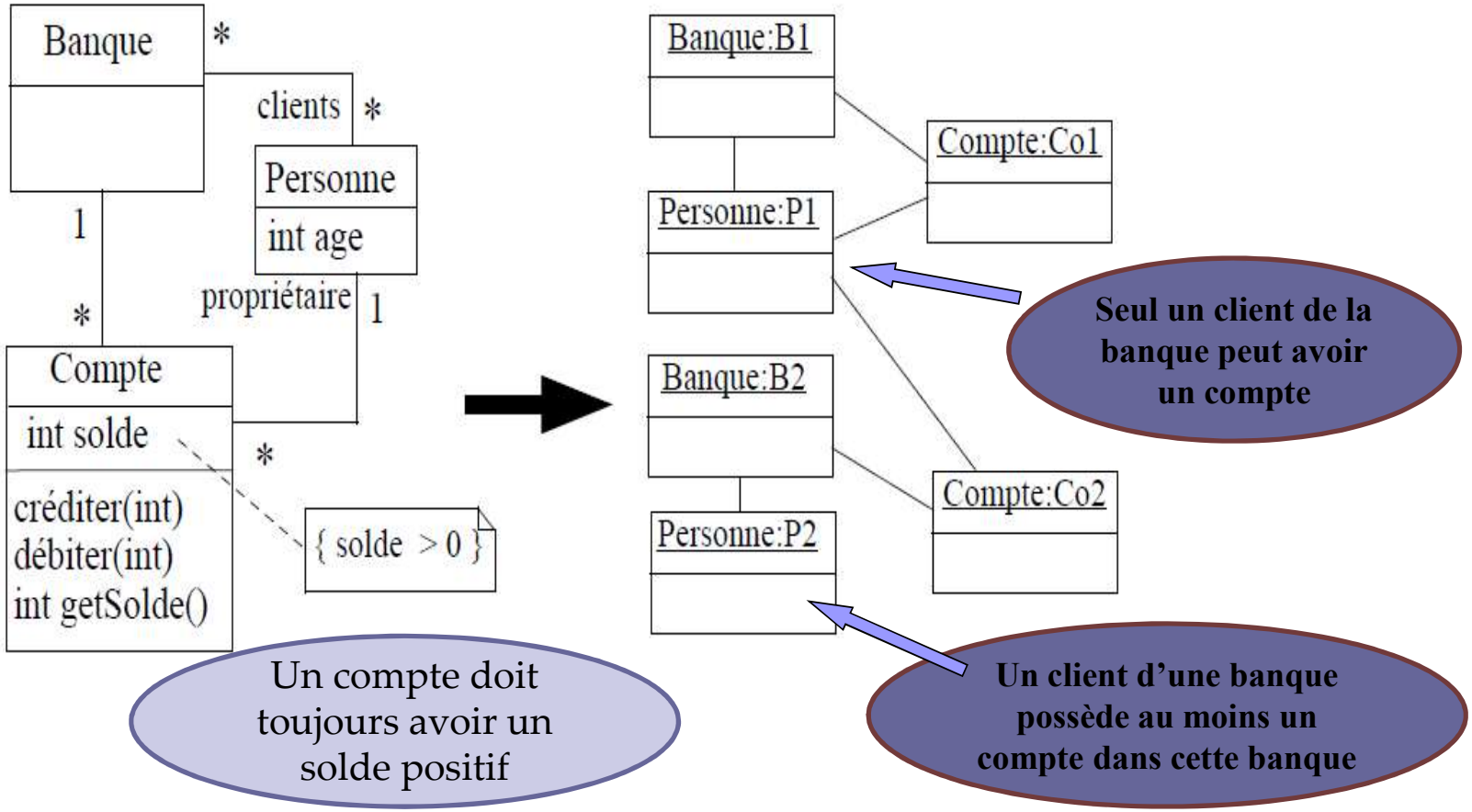
Exemple : Cahier des charges

- Développer un diagramme de classes qui respecte les contraintes suivantes :
  1. Un client peut être propriétaire de plusieurs comptes bancaires
  2. Un compte appartient à un seul client
  3. Une banque gère un ensemble de clients
  4. Le solde d'un compte doit être toujours supérieur à 1000 dh
  5. Une banque gère un ensemble de comptes
  6. Une personne peut être client dans plusieurs banques
  7. Un compte appartient à une seule banque
  8. Un client ne peut pas retirer de son compte une somme inférieure à 100 dh
  9. Un bonus de 500 dh est attribué au jeunes clients de moins de 23 ans
  10. Complément de la contrainte 1 : une personne entre 18 et 20 ans ne peut posséder qu'un seul compte

Exemple : Spécifications



Validité du diagramme de classe à travers le diagramme d'objets



OMG - OCL
(Object Constraint Language)

- Pour spécifier complètement une application :
  - Diagrammes UML seuls sont généralement insuffisants
  - Nécessité de rajouter des contraintes
- Comment exprimer ces contraintes ?
  - Langue naturelle, mais manque de précision → compréhension pouvant être ambigüe
  - Langage formel avec sémantique précise
- OCL : *Object Constraint Language*
  - Un langage formel pour l'expression de contraintes, standardisé par l'OMG.
  - Permet d'ajouter des descriptions
    - Précises
    - Non Ambigües
    - Evite les désavantages des langages formels

OCL
(Object Constraint Language)

- Description
  - Développé d'abord en 1995 par IBM, puis combiné à UML en 1997 ;
  - Version 2.0 conforme à UML 2 et au MOF 2.0
  - Langage de contraintes orienté-objet
  - S'applique sur les diagrammes UML
  - Il se veut simple à écrire ainsi qu'à comprendre.
  - OCL se veut un langage formel permettant de décrire des contraintes de façon déterministe.
  - Langage formel (mais simple à utiliser) avec une syntaxe, une grammaire, une sémantique (manipulable par un outil)
  - OCL est purement un langage interrogatif : en aucun cas il ne peut modifier le modèle auquel il se rapporte.
  - On parle de langage sans effet de bord, ou *side-effect free*.

OCL : Utilisation

- OCL permet principalement d'exprimer des contraintes sur l'état d'un objet ou d'un ensemble d'objets :
  - Des invariants qui doivent être respectés en permanence
  - Des pré et post-conditions pour une opération :
    - Précondition : doit être vérifiée avant l'exécution
    - Postcondition : doit être vérifiée après l'exécution
  - Gardes sur transitions de diagrammes d'états ou de messages de diagrammes de séquence/collaboration
  - Des ensembles d'objets destinataires pour un envoi de message
  - Des attributs dérivés
  - Des stéréotypes
  - ...

OCL : Contexte d'une contrainte

- Contexte
  - Une expression OCL est toujours définie dans un contexte qui définit sa portée .
  - Le contexte permet d'associer une contrainte à n'importe quel élément du modèle : package, classe, interface, composant, opération, attribut …
  - Un contexte peut aussi dénoter un sous-élément comme une opération ou un attribut.
  - Dans un contexte, le mot-clé **self** dénote l'objet courant:
    - 'self' est implicite dans toute expression OCL
    - Équivalent à `**this**' en Java ou c++

- Syntaxe :
  - **context** <élément>

OCL : Contexte d'une contrainte

- Pour faire référence à un élément de type opération d'une classe C, il faut utiliser les :: comme séparateur (comme C::op).

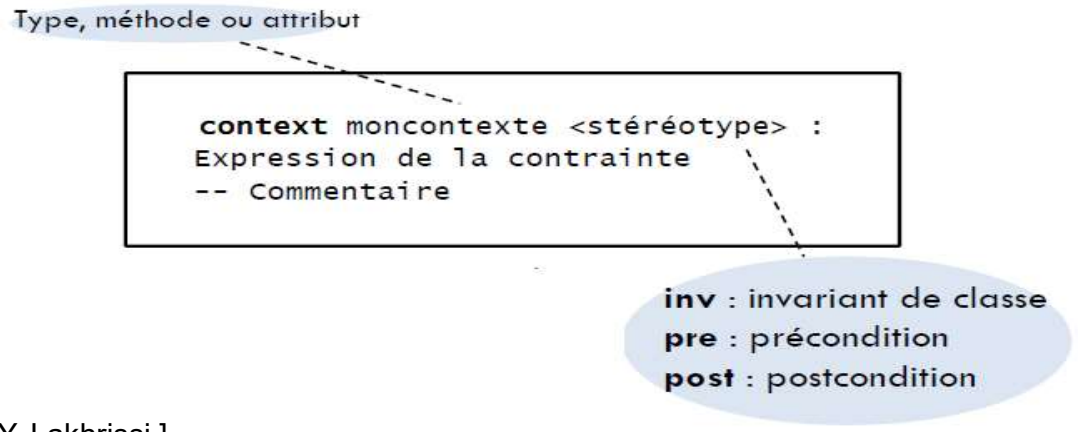
- Syntaxe pour préciser l'opération :
  - ***context ma_classe::mon_op(liste_param) : type_retour***

- Exemple :
  - Le contexte est la classe Compte :
    - ***context Compte***
    - L'expression OCL s'applique à la classe Compte, c'est-à-dire à toutes les instances de cette classe
  - Le contexte est l'opération getSolde() de la classe Compte:
    - ***context Compte::getSolde():integer***

Contrainte

- Une contrainte est la restriction d'une ou plusieurs valeurs d'un modèle UML

- Plusieurs types de contraintes :
  - **Invariant de Classe**
    - une contrainte qui doit être satisfaite par toutes les instances de la classe
  - **Precondition d'une opération**
    - une contrainte qui doit *être vérifiée* avant l' execution de l' opération
  - **Postcondition d'une opération**
    - une contrainte qui *doit être vérifiée* après l'execution de l' opération



Contrainte : Exemple(1/3)

-- Pour toutes les instances de la classe Compte, l'attribut solde doit toujours être positif

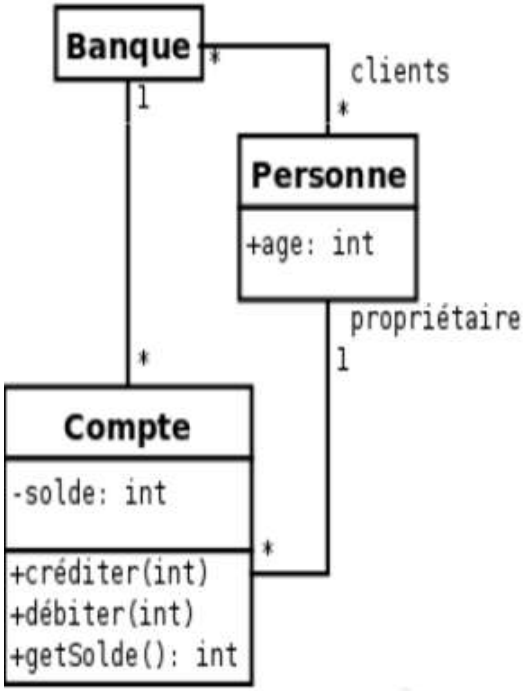
context Compte **inv**: self.solde > 0
context Compte::solde:Integer inv: self>0

-- Le propriétaire d'un compte doit avoir l'age>18

context Compte
inv: self.proprietaire.age>18

--La somme à débiter doit être positive (méthode débiter)

context Compte::débiter(s : Integer)
pre: s > 0 and s < solde
post: solde = solde**@pre** – s



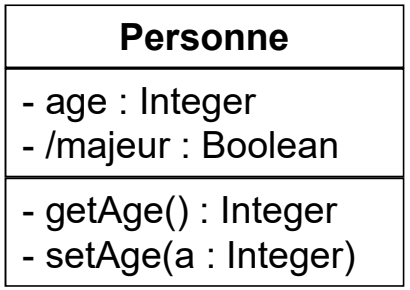
Remarque :

Dans la postcondition, deux éléments particuliers sont utilisables :

- Attribut **result** : référence la valeur retournée par l'opération
- mon_attribut**@ pre** : référence la valeur de mon_attribut avant l'appel de l'opération

Contrainte : Exemple(2/3)

context Personne::setAge(a : integer)
pre: (a <= 140)
and (a >=0)
and (a >= age)
post: age = a



context Personne **inv** ageBorné:
(age <= 140) and (age >=0)
-- l'âge ne peut dépasser 140 ans

Une contrainte peut être nommée par un label

context **p** : Personne **inv**:
(**p**.age <= 140) and (**p**.age >=0)

Un nom formel peut être donné à l'objet à partir duquel part l'évaluation

Contrainte : Exemple(3/3)

- Exemple 1 :
  - Le solde et la somme à débiter doivent être positifs pour que l'appel de l'opération « debiter » soit valide.
  - Après l'exécution de l'opération, l'attribut solde doit avoir pour valeur la différence de sa valeur avant l'appel et de la somme passée en paramètre.

context Compte::débiter(somme : Integer)
pre : somme > 0 and solde>somme
post : solde = solde@pre – somme

- Exemple 2 :
  - Le résultat retourné par la méthode getSolde doit être le solde courant

context Compte::getSolde() : Integer
post : result = solde

- Attention !
  - On ne décrit pas comment l'opération est réalisée mais des contraintes sur l'état avant et après son exécution.

Contrainte : init et derive

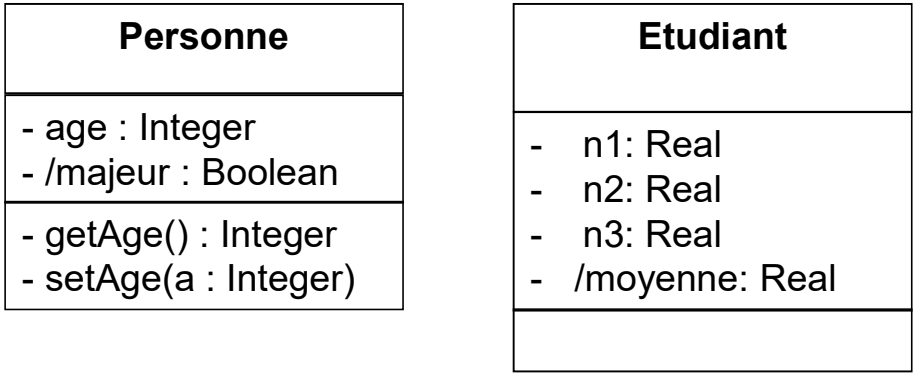
- ***init*** **:** indique la valeur initiale d'un attribut
  - Exemple :
    L'âge initial de toute Personne est égal à 0

- ***derive*** **:** indique la valeur dérivée d'un attribut
  - Exemple :
    L'attribut dérivé « majeur » égal à True automatiquement si l'âge est >=18

```
context Personne::age : Integer
init  : 0

context Personne::majeur : Boolean
derive  : age>=18

context Personne::moyenne: Real
derive : (n1+n2+n3)/3
```



Contrainte : Bilan

- Une expression est définie sur un contexte qui identifie :
- ***une cible*** : l'élément du modèle sur lequel porte l'expression OCL

T	Type (Classifier : Interface, Classe...)	**context** Employee
M	Opération/Méthode	**context** Employee::raiseWage(inc: Int)
A	Attribut ou extrémité d'association	**context** Employee::job : Job

- ***le rôle*** : indique la signification de cette expression (pré, post, invariant...) et donc contraint sa cible et son évaluation.

rôle	cible	signification	évaluation
inv	T	invariant	toujours vraie
pre	M	précondition	avant tout appel de M
post	M	postcondition	après tout appel de M
body	M	résultat d'une requête	appel de M
init	A	valeur initiale de A	création
derive	A	valeur de A	utilisation de A
def	T	définir une méthode ou un attribut	

Contrainte : Bilan

- Dans une expression OCL, on peut utiliser :
  - Types de base: String, Boolean, Integer, Real.
  - Éléments du modèle UML
    - attributs,
    - classes
    - opérations
  - Les associations du modèle UML
    - Y compris les noms de rôles
    - Conseil : utiliser des rôles dans UML pour simplifier OCL

Accès aux objets, navigation(1/2)

- Dans une contrainte OCL associée à un objet, on peut :
  - Accéder à l'état interne de cet objet (ses attributs)
  - Naviguer dans le diagramme : accéder de manière transitive à tous les objets (et leur état) avec qui il est en relation

- Nommage des éléments dans une expression OCL :

  - Attributs ou paramètres d'une opération : utilise leur nom directement

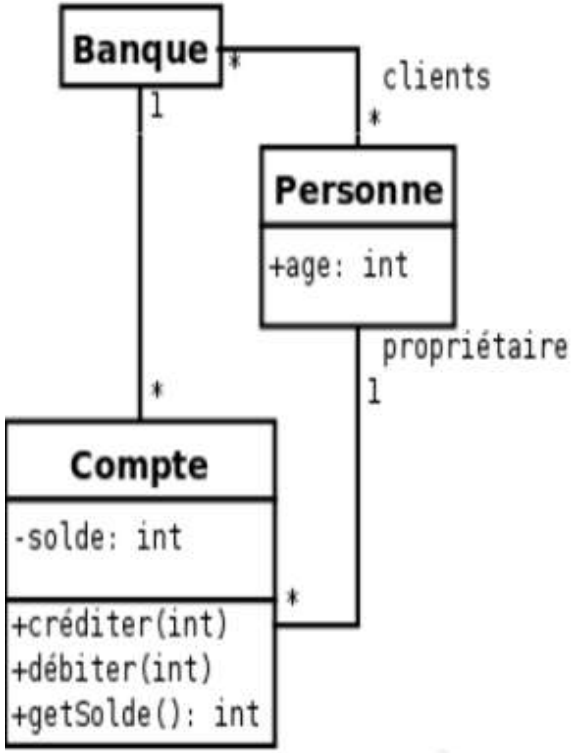
  - Objet(s) en association : utilise le nom de la classe associée (en minuscule) ou le nom du rôle d'association du coté de cette classe

  - Si cardinalité de 1 pour une association : référence un objet

  - Si cardinalité > 1 : référence une collection d'objets

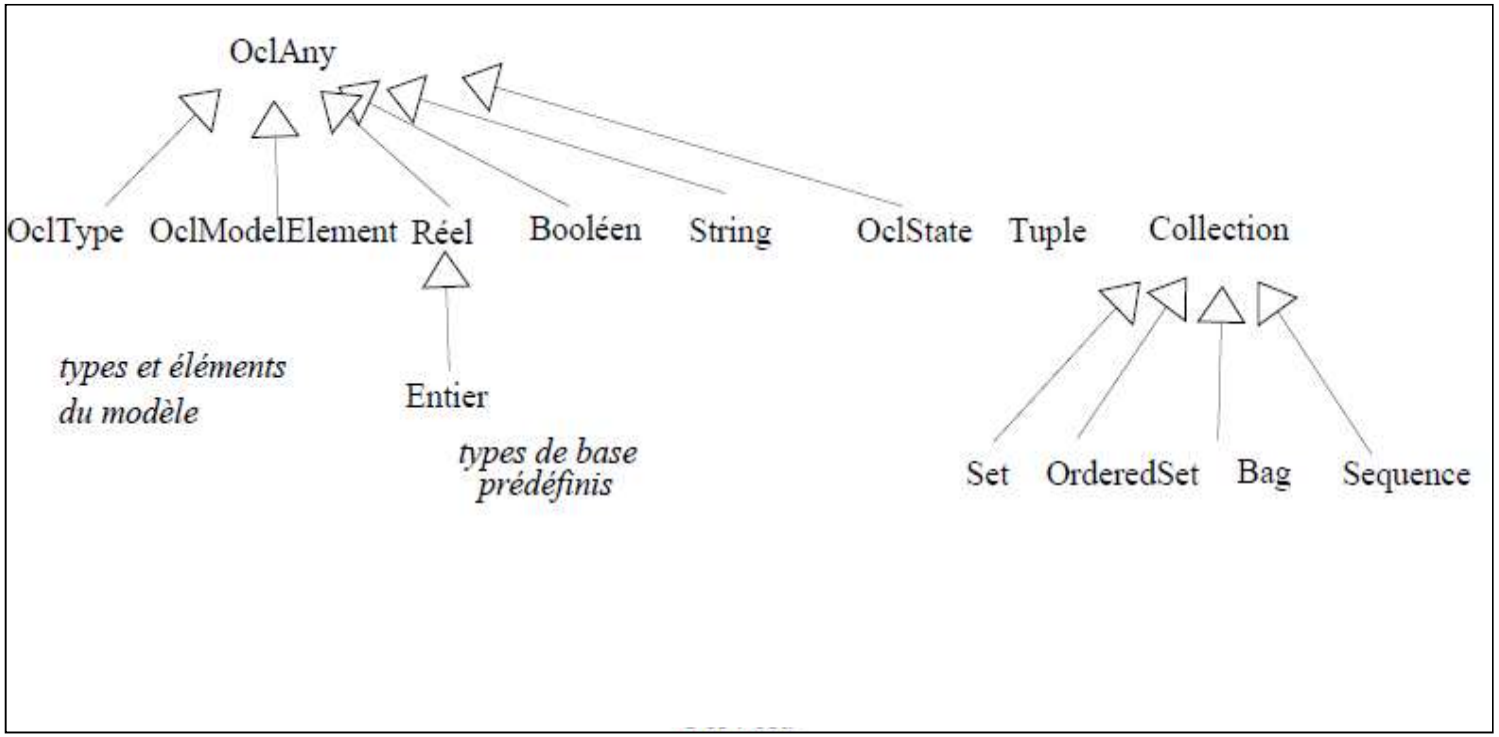
Accès aux objets, navigation(2/2)

- Exemples, dans contexte de la classe Compte
  - **solde** : attribut référencé directement
  - **banque** : objet de la classe Banque (référencé via le nom de la classe) associé au compte
  - **propriétaire** : objet de la classe Personne (référence via le nom de rôle d'association) associée au compte
  - **banque.clients** ➔ ensemble des clients de la banque : c'est une collection de personnes
  - **banque.clients.age** ➔ ensemble des âges de tous les clients de la banque : c'est une collection d'entiers



- - Le propriétaire d'un compte doit avoir plus de 18 ans
context Compte **inv**:
self.propriétaire.age >= 18

Types OCL


OclAny
OclType
OclModelElement
Réel
Booléen
String
OclState
Tuple
Collection
types et éléments du modèle
Entier
types de base prédéfinis
Set
OrderedSet
Bag
Sequence

Types OCL : Types de base

- Integer
  - 1, -2, 145
  - Opérateur : = <> + - * / abs div mod < > <= >=
- Real
  - 1.5, -123.4
  - Opérateur : = <> + - * / abs floor round < > <= >=
- String
  - 'bonjour'
  - s.concat(t), s.size(), s.toLower(), s.toUpper(), substring(,), replaceAll(,)…

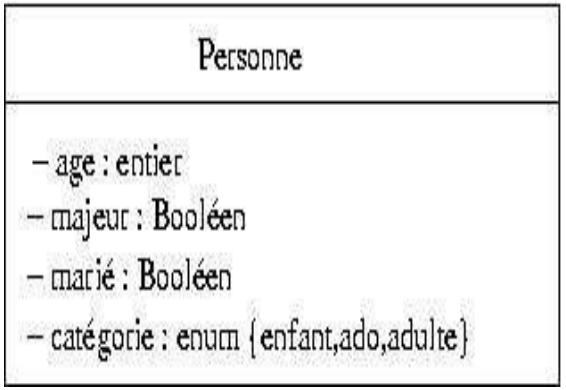
Types OCL : Types de base

- Booléen
  - Les constantes s'écrivent : true, false
  - Les opérateurs sont les suivants :
    - **= or xor and not**
    - b1 **implies** b2
    - **if** b **then** expression1 **else** expression2 **endif**

Exemple :

```
context Personne inv :
    marié implies majeur

context Personne inv :
   majeur = if age >=18 then
                        true
                else false
                endif
```

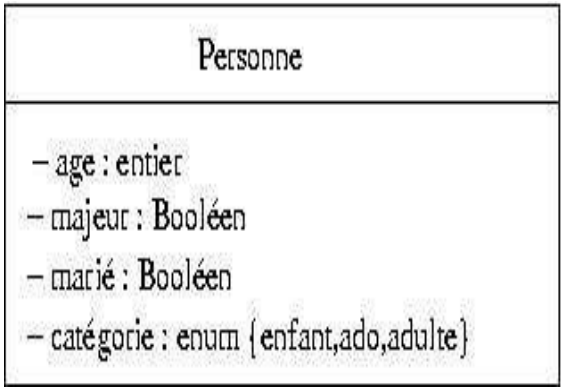


Types OCL : Type énuméré

- Leur syntaxe est la suivante :
  - ***<nom_type_enuméré>::valeur***
- Le nom du type est déduit de l'attribut de déclaration
  - catégorie => Catégorie
- Exemple :

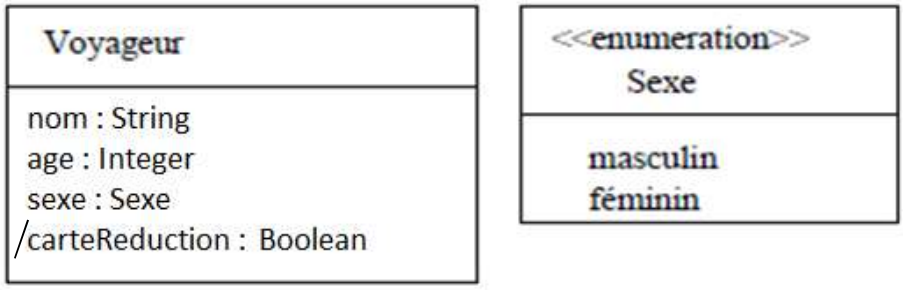
context Personne inv:

categorie **=** ***if*** *age <=12* ***then***
Categorie::enfant
else
if *age <=18* ***then***
Categorie::ado
else
Categorie::adulte
endif
endif



Types OCL : Types de base

- Exemple
  - Ecrivez pour le diagramme ci-dessous la contrainte qui caractérise l'attribut dérivé carteReduction. Un voyageur a droit à la carte de réduction si c'est une femme de plus de 60 ans ou un homme de plus de 65 ans.



Solution:

context Voyageur inv :
carteReduction = ((age >=65) or ((sexe=Sexe::féminin) and (age >=60)))

Cette contrainte peut également s'écrire avec derive.
context Voyageur::carteReduction : Boolean
derive: ((age >=65) or ((sexe=Sexe::féminin) and (age >=60)))

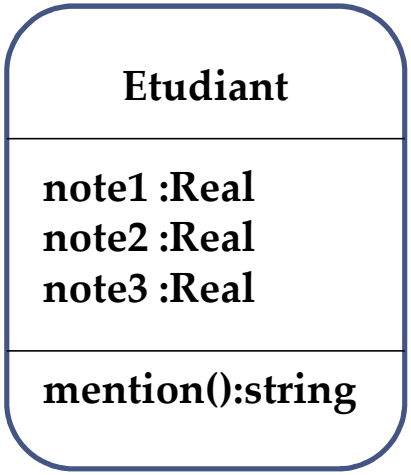
Types : Règle de précédence

- Ordre de précédence pour les opérateurs/primitives :
  - @pre
  - . et ->
  - not et -
  - * et /
  - + et -
  - if then else endif
  - >, <, <= et >=
  - = et <>
  - and, or et xor
  - Implies
- Les parenthèses permettent de changer cet ordre

Définition de variables

- Syntaxe pour définir une nouvelle variable :
  - **let** variable : type **=** expression1 **in** expression2
- Exercice
  - soit la classe **Etudiant**, disposant de 3 notes et munie d'une opération **mention** qui retourne la mention de l'étudiant sous forme d'une chaîne de caractères. Ecrivez les contraintes postcondition sur l'opération mention() en utilisant **let** pour calculer la moyenne.

```
context Etudiant::mention() : String
post :
let moyenne : Real =(note1+note2+note3)/3      in
result=
if (moyenne >= 16) then
            'très bien'
else if (moyenne >= 14) then
            'bien'
     else  'moins bien'
     endif
endif
```



Définition des opérations

- Si on veut définir une variable(opération) utilisable dans plusieurs contraintes de la classe, on peut utiliser la construction def.
  - Syntaxe : **def** : <déclaration> = <requête>

Exemple :

context Personne
def : ageCorrect(a :Real) :Boolean = a>=0 and a<=140

Exp1- context Personne inv :
ageCorrect(age) - - l'âge ne peut depasser 140 ans

Exp2- context Personne::setAge(a :integer)
pre : ageCorrect(a) and (a >= age)

Exercice : **réaliser le même exercice précédent en utilisant def au lieu de let**

Keywords

- and
- body
- context
- def
- derive
- else
- endif
- endpackage
- false
- if
- implies
- in
- init
- or
- package
- post
- pre
- self
- static
- then
- true
- xor
- inv
- invalid
- let
- not
- null
- Bag
- Boolean
- Collection
- Integer
- OclAny
- OclInvalid
- OclMessage
- OclVoid
- OrderedSet
- Real
- Sequence
- Set
- String
- Tuple
- UnlimitedNatural

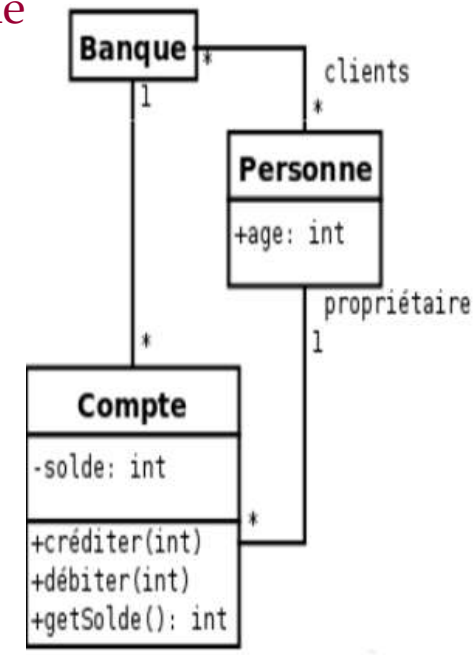
Définition de variables et d'opérations (Exercice)

- Utilisez le **Let ou Def** pour écrire une contrainte dans le contexte Personne qui spécifie qu'une personne majeure doit avoir de l'argent.
- sum() : fait la somme de tous les objets de l'ensemble

context Personne **inv** :
let argent : integer = compte.solde -> sum()
in **self.**age >= 18 **implies** argent > 0

context Personne **def** : argent() : integer =
self.compte.solde -> sum()

context Personne **inv** :
self.age >= 18 **implies** argent() > 0



Types OCL : Le type Collection

- Dans les expressions de navigation (a.b), le type du résultat dépend de la cardinalité de l'association :
  - Cardinalités simples : 0 ou 1, rend un objet.
  - Cardinalités multiples, rend une collection

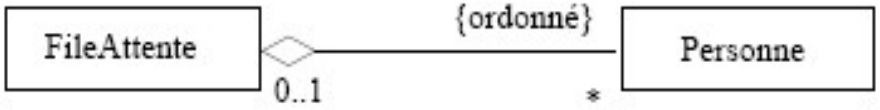
Types OCL : Le type Collection

- Les types suivants sont fournies pour les collections, tous sont des sous-types d'une classe abstraite Collection:
  - Set : La notion mathématique d'ensemble (pas de doublons, pas d'ordre)
    => Exemple : { 1, 4, 3, 5 }
  - OrderedSet : La notion mathématique d'ensemble ordonné
    => Exemple : { 2, 4, 6, 8 }
  - Bag : La notion mathématique de famille (doublons possibles, pas d'ordre)
    => Exemple : { 1, 4, 1, 3, 5, 4 }
  - Sequence : La notion mathématique de famille, et les éléments sont ordonnés
    => Exemple : { 1, 1, 3, 4, 4, 5 }

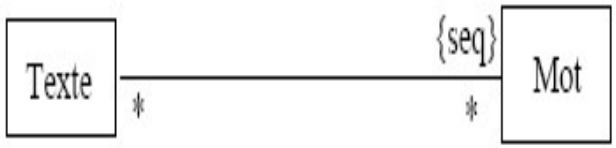
- Remarque : il est possible de définir des collections de collections,
  => Exemple : Set { 2, 4, Set {6, 8 }}

Types OCL : Le type Collection
Exemple : résultats de navigation

- **OrderedSet** :obtenu en naviguant vers une extremité d'association munie de la contrainte « ordonné »
  - Exemple : l'expression **self.personne** dans le contexte **FileAttente**



- **Sequence** :obtenu en naviguant vers une extrémité d'association munie de la contrainte **seq**,
  - Exemple: l'expression **self.mot** dans le contexte **Texte**.



Opérations sur les collections(1/2)

- Appel : Les opérations sur une collection sont mentionnées avec ->
  - **Syntaxe d'utilisation** : **objetOuCollection -> primitive**

- OCL propose un ensemble de primitives utilisables sur les ensembles :
  - **size() : integer** : retourne le nombre d'éléments de l'ensemble
  - **count(unObjet :T) : integer** : donne le nombre d'occurrences de unObjet.
  - **sum() : T** : Addition de tous les éléments de la collection (qui doivent supporter une opération + associative)
  - **isEmpty() : Boolean** : retourne vrai si l'ensemble est vide
  - **notEmpty() : Boolean** : retourne vrai si l'ensemble n'est pas vide
  - **includes(obj: T) : Boolean** : vrai si l'ensemble inclut l'objet obj
  - **excludes(obj:T) : Boolean** : vrai si l'ensemble n'inclut pas l'objet obj

Opérations sur les collections(2/2)

- **forAll(uneExpression :OclExpression):Boolean** :Vaut vrai ssi une expression est vraie pour tous les éléments de la collection.
- **exists(uneExpression):Boolean :**Vrai ssi au moins un élément de la collection satisfait uneExpression
- **isUnique(uneExpression):Boolean** : Vrai ssi une expression s'évalue avec une valeur distincte pour chaque élément de la collection.
- **one(uneExpression):Booleen** :Vrai ssi un et un seul élément de la collection vérifie uneExpression.
- **any(uneExpression):T** :Retourne n'importe quel élément de la collection qui vérifie uneExpression.
- **sortedBy(uneExpression):Séquence** :Retourne une séquence contenant les éléments de la collection tries par ordre croissant suivant le critère uneExpression
- **select(expression):Séquence** rend la sous collection d'objets satisfaisant l'expression
- **reject(expression):Séquence** rejette de la collection les objets satisfaisant l'expression

Exercice 1

- Il n'existe pas de clients de la banque dont l'âge est inférieur à 18 ans (not : prend la négation d'une expression)

  context Banque inv: not(self.clients -> exists (age < 18))

  context Banque inv: self.clients-> select(a | a.age<18)->isEmpty()

- Il n'existe pas deux instances de la classe Personne pour lesquelles l'attribut nom a la même valeur : deux personnes différentes ont un nom différent

  context Personne inv:

  Personne.allInstances() -> forAll(p1, p2 | p1 <> p2 implies p1.nom <> p2.nom)

  - Remarque : allInstances() : primitive s'appliquant sur une classe (et non pas un objet) et retournant toutes les instances de cette classe
  - **Personne.allInstances() -> isUnique(nom)**

- Si une personne possède au moins un compte bancaire, alors elle est cliente d'au moins une banque

  context Personne inv:

  self.compte -> notEmpty() implies self.banque -> notEmpty()

- Le propriétaire d'un compte doit être client de la banque

  context Compte inv :

  self.banque.clients -> includes (self.propriétaire)