

Université Mohammed Premier

École Nationale des Sciences Appliquées

Oujda



Formation : Génie Informatique
Niveau : 5ème année

Séance n°2

Object Constraint Language 2.x

OCL 2.x

Présenté par :
Redouane ESBAI

- 
- 
1. Pourquoi OCL ?
 2. Les principaux concepts d'OCL
 3. Exemple d'application sur modèle

Exemple d'application

➤ Application bancaire

- ❑ Des comptes bancaires
- ❑ Des clients
- ❑ Des banques

➤ Spécifications

- ❑ Un compte doit avoir un solde toujours positif
- ❑ Un client peut posséder plusieurs comptes
- ❑ Un client peut être client de plusieurs banques
- ❑ Un client d'une banque possède au moins un compte dans cette banque
- ❑ Une banque gère plusieurs comptes
- ❑ Une banque possède plusieurs clients

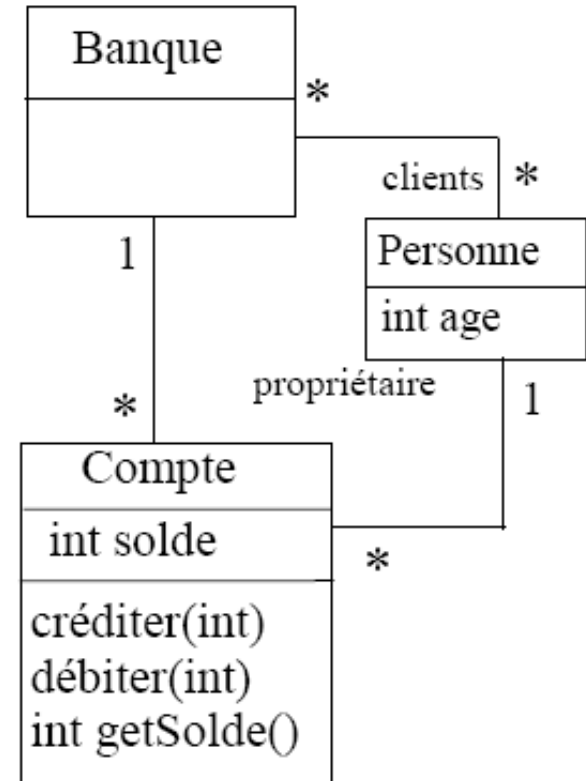
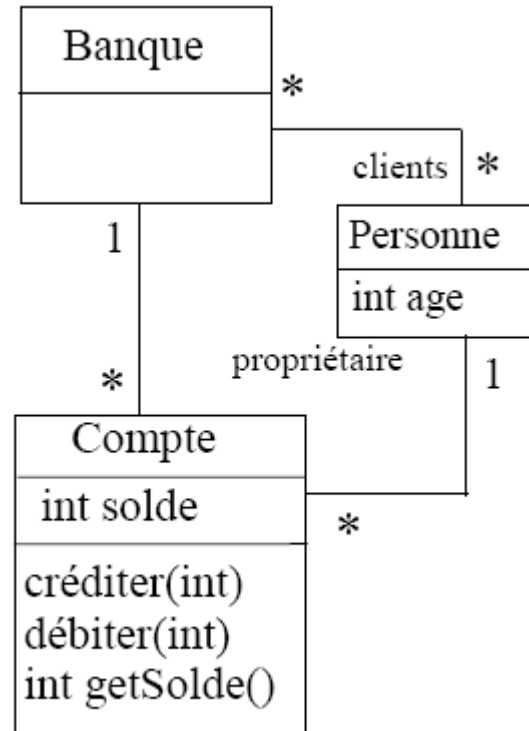


Diagramme de classe



Manque de précision

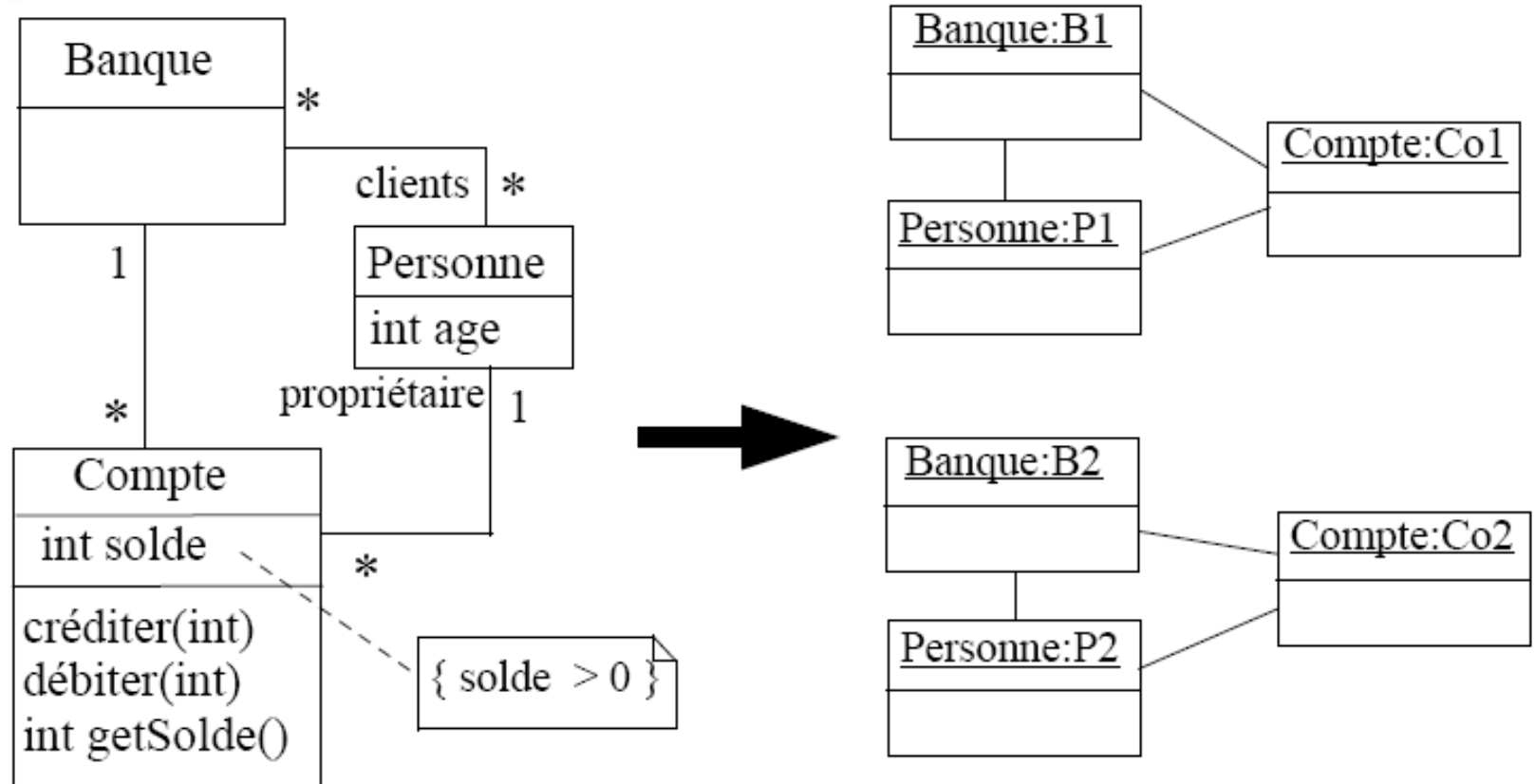
➤ Le diagramme de classe ne permet pas d'exprimer tout ce qui est défini dans la spécification informelle

➤ **Exemple**

❑ Le solde d'un compte doit toujours être positif : ajout d'une contrainte sur cet attribut

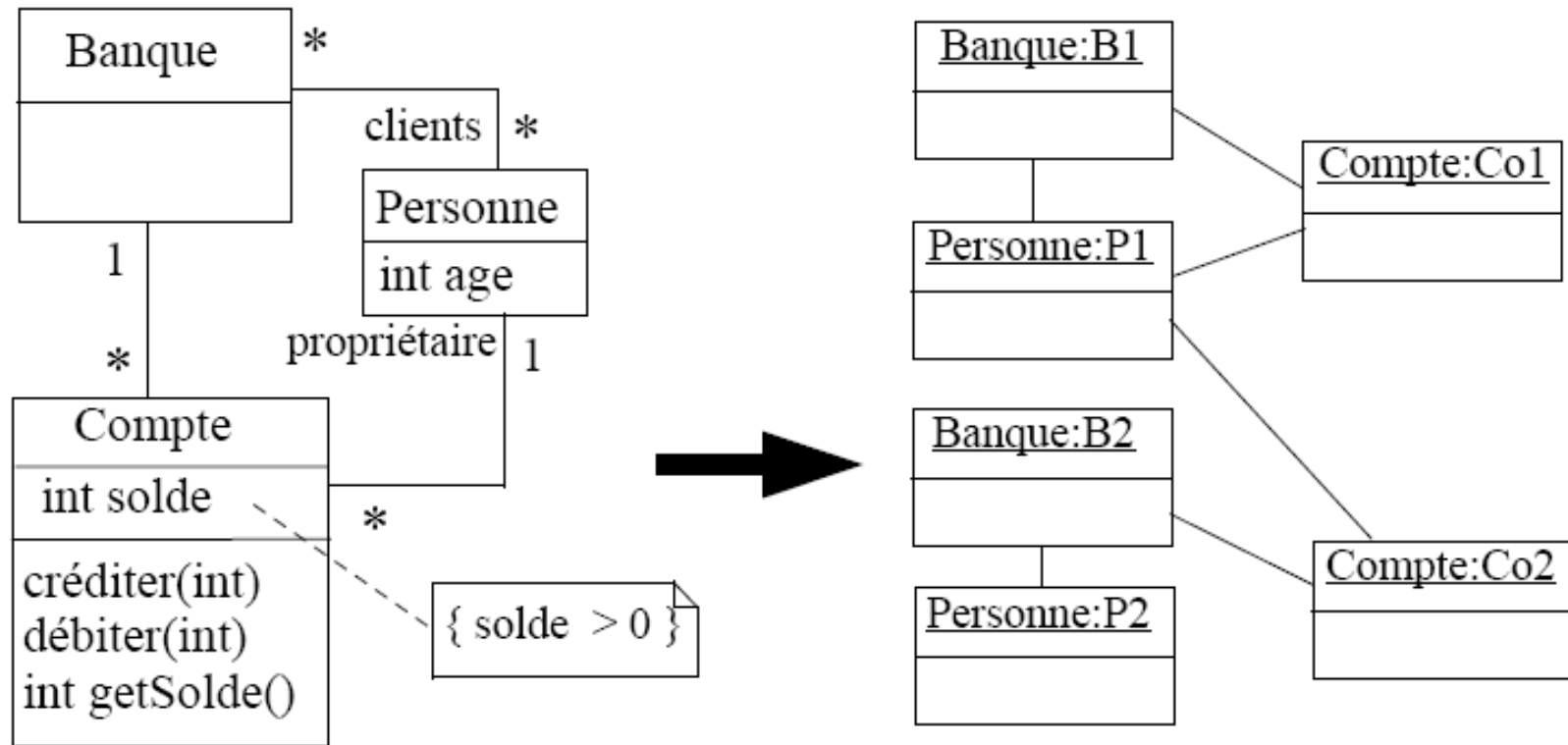
➤ Le diagramme de classe permet-il de détailler toutes les contraintes sur les relations entre les classes ?

Diagramme d'objets



- ◆ Diagramme d'instances valide vis-à-vis du diagramme de classe et de la spécification attendue

Diagramme d'objets



- ◆ Diagramme d'instances valide vis-à-vis du diagramme de classe mais ne respecte pas la spécification attendue
 - ◆ Une personne a un compte dans une banque où elle n'est pas cliente
 - ◆ Une personne est cliente d'une banque mais sans y avoir de compte

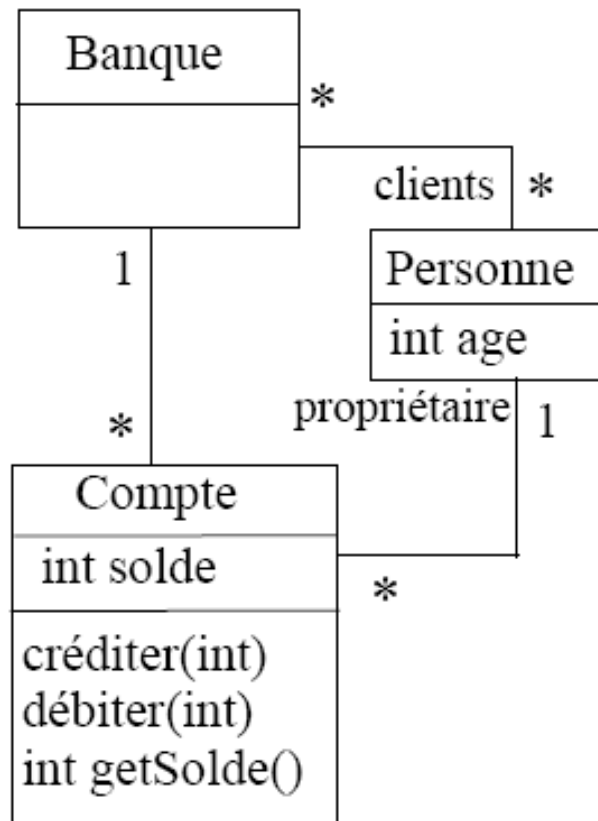
Diagrammes UML insuffisants

- ◆ 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 ambiguë
 - ◆ Langage formel avec sémantique précise : par exemple OCL
- ◆ OCL : Object Constraint Language
 - ◆ Langage de contraintes orienté-objet
 - ◆ Langage formel (mais simple à utiliser) avec une syntaxe, une grammaire, une sémantique (manipulable par un outil)
 - ◆ S'applique entre autres sur les diagrammes UML ou MOF

Le langage OCL

- ◆ Version courante : version 2.0
 - ◆ Peut s'appliquer sur tout type de modèle, indépendant d'un langage de modélisation donné
- ◆ OCL permet principalement d'exprimer deux types de 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
- ◆ Attention
 - ◆ Une expression OCL décrit une contrainte à respecter et non pas le « code » d'une méthode

Usage d'OCL sur l'application bancaire



context Compte

inv: solde > 0

context Compte : débiter(somme : int)

pre: somme > 0

post: solde = solde@pre - somme

context Compte

inv: banque.clients

→ includes (propriétaire)

- ◆ **Avantage d'OCL :** langage formel permettant de préciser clairement de la sémantique sur les modèles UML

OCCL : introduction

- Un langage formel d'expression de contraintes adapté aux diagrammes d'UML,
 - En particulier au diagramme de classes.
 - Existe depuis la version 1.3
-

Historique

- **Origine**
 - OCL a été développé à partir de 95 par Jos Warmer (IBM) sur les bases du langage IBEL (Integrated Business Engineering Language).
- **Première définition : IBM, 1997**
- **Formellement intégré à UML 1.1 en 1999**
- **OCL2.0 intégré dans la définition d'UML2.0 en 2003**
 - conforme à UML 2 et au MOF 2.0
 - Fait partie du catalogue de spécifications de l'OMG

Principes

- **OCCL : Langage déclaratif**
 - Les contraintes ne sont pas opérationnelles.
 - On ne peut pas invoquer de processus ni d'opérations autres que des requêtes
 - On ne décrit pas le comportement à adopter si une contrainte n'est pas respectée
- **OCCL : Langage sans effet de bord**
 - Les instances ne sont pas modifiées par les contraintes

Utilisation d'OCL dans le cadre d'UML

- ◆ OCL peut s'appliquer sur la plupart des diagrammes UML
- ◆ Il sert, entre autres, à spécifier des
 - ◆ Invariants sur des classes
 - ◆ Pré et postconditions sur des opérations
 - ◆ 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

Contexte

- ◆ Une expression OCL est toujours définie dans un contexte
 - ◆ Ce contexte est l'instance d'une classe
- ◆ Mot-clé : **context**
- ◆ Exemple
 - ◆ **context** Compte
 - ◆ L'expression OCL s'applique à la classe Compte, c'est-à-dire à toutes les instances de cette classe

Invariants

- ◆ Un invariant exprime une contrainte sur un objet ou un groupe d'objets qui doit être respectée en permanence
- ◆ Mot-clé : **inv**
- ◆ Exemple
 - ◆ **context** Compte
inv: solde > 0
 - ◆ Pour toutes les instances de la classe Compte, l'attribut solde doit toujours être positif

Pré et postconditions

- ◆ Pour spécifier une opération
 - ◆ Précondition : état qui doit être respecté avant l'appel de l'opération
 - ◆ Postcondition : état qui doit être respecté après l'appel de l'opération
 - ◆ Mots-clés : **pre** et **post**
- ◆ 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
- ◆ Syntaxe pour préciser l'opération
 - ◆ `context ma_classe::mon_op(liste_param) : type_retour`

Pré et postconditions

◆ Exemples

◆ **context** `Compte::débitier(somme : int)`

pre: `somme > 0`

post: `solde = solde@pre - somme`

◆ La somme à débiter doit être positive pour que l'appel de l'opération 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::getSolde() : int`

post: `result = solde`

◆ Le résultat retourné *doit* être le solde courant

◆ Attention

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

Accès aux objets, navigation

- ◆ 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
 - ◆ 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 côté 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

- ◆ Exemples, dans contexte de la classe Compte
 - ◆ solde : attribut référencé directement
 - ◆ banque : objet de la classe Banque (référence 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 associée au compte (référence par transitivité)
 - ◆ banque.clients.age : ensemble des âges de tous les clients de la banque associée au compte
- ◆ Le propriétaire d'un compte doit avoir plus de 18 ans
 - ◆ **context** Compte
 - ◆ **inv:** propriétaire.age \geq 18

Opérations sur objets et collections

- ◆ OCL propose un ensemble de primitives utilisables sur les collections
 - ◆ `size()` : retourne le nombre d'éléments de la collection
 - ◆ `isEmpty()` : retourne vrai si la collection est vide
 - ◆ `notEmpty()` : retourne vrai si la collection n'est pas vide
 - ◆ `includes(obj)` : vrai si la collection inclut l'objet *obj*
 - ◆ `excludes(obj)` : vrai si la collection n'inclut pas l'objet *obj*
 - ◆ `including(obj)` : la collection référencée doit être cette collection en incluant l'objet *obj*
 - ◆ `excluding(obj)` : idem mais en excluant l'objet *obj*
 - ◆ `includesAll(ens)` : la collection contient tous les éléments de la collection *ens*
 - ◆ `excludesAll(ens)` : la collection ne contient aucun des éléments de la collection *ens*
- ◆ Syntaxe d'utilisation : *objetOuCollection -> primitive*

Opérations sur objets et collections

- ◆ Exemples, invariants dans le contexte de la classe Compte
 - ◆ `propriétaire -> notEmpty()` : il y a au moins un objet `Personne` associé à un compte
 - ◆ `propriétaire -> size() = 1` : le nombre d'objets `Personne` associés à un compte est de 1
 - ◆ `banque.clients -> size() >= 1` : une banque a au moins un client
 - ◆ `banque.clients -> includes(propriétaire)` : l'ensemble des clients de la banque associée au compte contient le propriétaire du compte
 - ◆ `banque.clients.compte -> includes(self)` : le compte appartient à un des clients de sa banque
- ◆ `self`
 - ◆ pseudo-attribut référençant l'objet courant

Opérations sur objets et collections

◆ Autre exemple

- ◆ **context** Banque :: creerCompte(p : Personne) : Compte
post: result.ocllsNew() and
compte = compte@pre -> including(result) **and**
p.compte = p.compte@pre -> including(result)
- ◆ Un nouveau compte est créé. La banque doit gérer ce nouveau compte. Le client passé en paramètre doit posséder ce compte. Le nouveau compte est retourné par l'opération.

◆ OcclIsNew()

- ◆ Primitive indiquant qu'un objet doit être créé pendant l'appel de l'opération (à utiliser dans une postcondition)

◆ And

- ◆ Permet de définir plusieurs contraintes pour un invariant, une pré ou postcondition
- ◆ and = « et logique » : l'invariant, pré ou postcondition est vrai si toutes les expressions reliées par le « and » sont vraies

Relations ensemblistes entre collections

- ◆ union
 - ◆ Retourne l'union de deux collections
- ◆ intersection
 - ◆ Retourne l'intersection de deux collections
- ◆ Exemples
 - ◆ `(coll -> intersection(col2)) -> isEmpty()`
 - ◆ Renvoie vrai si les collections `col1` et `col2` n'ont pas d'élément en commun
 - ◆ `coll = col2 -> union(col3)`
 - ◆ La collection `col1` doit être l'union des éléments de `col2` et de `col3`

Opérations sur éléments d'une collection

- ◆ OCL permet de vérifier des contraintes sur chaque élément d'une collection ou de définir une sous-collection à partir d'une collection en fonction de certaines contraintes
- ◆ Primitives offrant ces services et s'appliquant sur une collection *col*
 - ◆ *select* : retourne le sous-ensemble de la collection *col* dont les éléments respectent la contrainte spécifiée
 - ◆ *reject* : idem mais ne garde que les éléments ne respectant pas la contrainte
 - ◆ *collect* : retourne une collection (de taille identique) construite à partir des éléments de *col*. Le type des éléments contenus dans la nouvelle collection peut être différent de celui des éléments de *col*.
 - ◆ *exists* : retourne vrai si au moins un élément de *col* respecte la contrainte spécifiée et faux sinon
 - ◆ *forAll* : retourne vrai si tous les éléments de *col* respectent la contrainte spécifiée (pouvant impliquer à la fois plusieurs éléments de la collection)

Opérations sur éléments d'une collection

- ◆ Syntaxe de ces opérations : 3 usages
 - ◆ `collection -> primitive(expression)`
 - ◆ La primitive s'applique aux éléments de la collection et pour chacun d'entre eux, l'expression *expression* est vérifiée. On accède aux attributs/rerelations d'un élément directement.
 - ◆ `collection -> primitive(elt : type | expression)`
 - ◆ On fait explicitement apparaître le type des éléments de la collection (ici *type*). On accède aux attributs/rerelations de l'élément courant en utilisant *elt* (c'est la référence sur l'élément courant)
 - ◆ `collection -> primitive(elt | expression)`
 - ◆ On nomme l'attribut courant (*elt*) mais sans préciser son type

Opérations sur éléments d'une collection

- ◆ Dans le contexte de la classe Banque
 - ◆ `compte -> select(c | c.solde > 1000)`
 - ◆ Retourne une collection contenant tous les comptes bancaires dont le solde est supérieur à 1000 €
 - ◆ `compte -> reject(solde > 1000)`
 - ◆ Retourne une collection contenant tous les comptes bancaires dont le solde n'est pas supérieur à 1000 €
 - ◆ `compte -> collect(c : Compte | c.solde)`
 - ◆ Retourne une collection contenant l'ensemble des soldes de tous les comptes
 - ◆ `(compte -> select(solde > 1000))
-> collect(c | c.solde)`
 - ◆ Retourne une collection contenant tous les soldes des comptes dont le solde est supérieur à 1000 €

Opérations sur éléments d'une collection

- ◆ **context** Banque
 - inv:** `not(clients -> exists (age < 18))`
 - ◆ Il n'existe pas de clients de la banque dont l'age est inférieur à 18 ans
 - ◆ **not**
 - ◆ Prend la négation d'une expression
- ◆ **context** Personne
 - inv:** `Personne.allInstances() -> forAll(p1, p2 | p1 <> p2 implies p1.nom <> p2.nom)`
 - ◆ 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
 - ◆ `AllInstances()`
 - ◆ Primitive s'appliquant sur une classe (et non pas un objet) et retournant toutes les instances de la classe référencée (ici la classe Personne)

Types OCL : types de base

- ◆ Types de base et exemples d'opérations associées
 - ◆ Integer
 - ◆ 1, -2, 145
 - ◆ *, +, -, /, abs()
 - ◆ Real
 - ◆ 1.5, -123.4
 - ◆ *, +, -, /, floor()
 - ◆ String
 - ◆ 'bonjour'
 - ◆ concat(), size(), substring()
 - ◆ Boolean
 - ◆ true, false
 - ◆ And, or, not, xor, not, implies, if-then-else
 - ◆ La plupart des expressions OCL sont de types Boolean
 - ◆ Notamment les expressions formant les inv, pre et post

Types OCL : types de collection

- ◆ 3 (4) types de collection d'objets
 - ◆ Set : ensemble au sens mathématique, pas de doublons, pas d'ordre
 - ◆ OrderedSet : idem mais avec ordre
 - ◆ Bag : comme un Set mais avec possibilité de doublons
 - ◆ Sequence : un Bag dont les éléments sont ordonnés
- ◆ Exemples :
 - ◆ { 1, 4, 3, 5 } : Set
 - ◆ { 1, 4, 1, 3, 5, 4 } : Bag
 - ◆ { 1, 1, 3, 4, 4, 5 } : Sequence
- ◆ Notes
 - ◆ Un collect() renvoie toujours un Bag

Conditionnelles

- ◆ Certaines contraintes sont dépendantes d'autres contraintes
- ◆ Deux formes pour gérer cela
 - ◆ **if *expr1* then *expr2* else *expr3* endif**
 - ◆ Si l'expression *expr1* est vraie alors *expr2* doit être vraie sinon *expr3* doit être vraie
 - ◆ ***expr1* implies *expr2***
 - ◆ Si l'expression *expr1* est vraie, alors *expr2* doit être vraie également.

Conditionnelles

- ◆ **context** Personne **inv**:
if age < 18
then compte -> isEmpty()
else compte -> notEmpty()
endif
- ◆ Une personne de moins de 18 ans n'a pas de compte bancaire alors qu'une personne de plus de 18 ans possède au moins un compte
- ◆ **context** Personne **inv**:
compte -> notEmpty()
implies banque -> notEmpty()
- ◆ Si une personne possède au moins un compte bancaire, alors elle est cliente d'au moins une banque

◆ Commentaire en OCL : utilisation de --

◆ Exemple

```
context Personne inv:  
  if age < 18  
    then compte -> isEmpty()  
    else compte -> notEmpty()  
  endif  
-- vérifie age de la personne  
-- pas majeur : pas de compte  
-- majeur : doit avoir  
-- au moins un compte
```

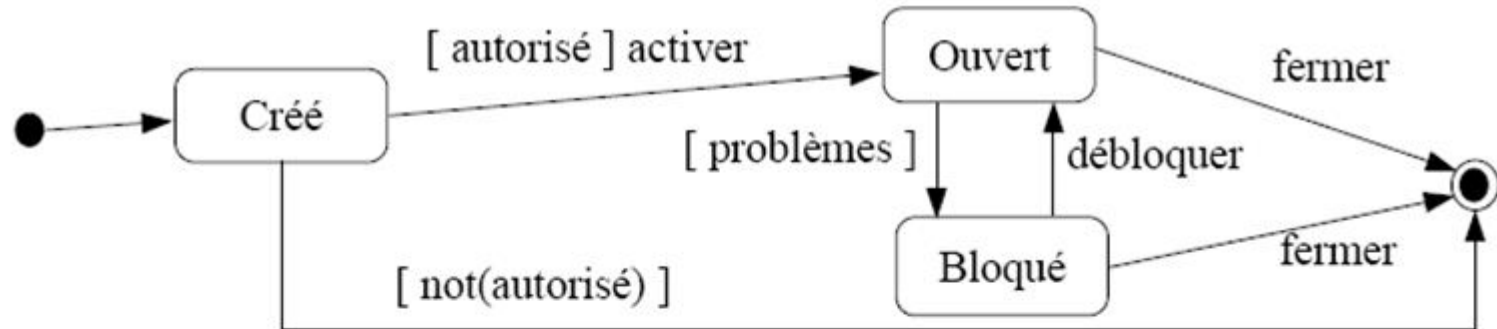
Variables

- ◆ Pour faciliter l'utilisation de certains attributs ou calculs de valeurs on peut définir des variables
- ◆ Dans une contrainte OCL : `let ... in ...`
 - ◆ **context** Personne
inv: `let argent = compte.solde -> sum() in`
`age >= 18 implies argent > 0`
 - ◆ Une personne majeure doit avoir de l'argent
 - ◆ `sum()` : fait la somme de tous les objets de la collection
- ◆ Pour l'utiliser partout : `def`
 - ◆ **context** Personne
def: `argent : int = compte.solde -> sum()`
 - ◆ **context** Personne
inv: `age >= 18 implies argent > 0`

Appels d'opération des classes

- ◆ Dans une contrainte OCL : accès aux attributs, objets ... « en lecture »
- ◆ Possibilité d'utiliser une opération d'une classe dans une contrainte
 - ◆ Si pas d'effets de bords (de type « query »)
 - ◆ Car une contrainte OCL exprime une contrainte sur un état mais ne précise pas qu'une action a été effectuée
- ◆ Exemple :
 - ◆ **context** Banque
 - inv:** compte -> forAll(c | c.getSolde() > 0)
 - ◆ getSolde() est une opération de la classe Compte. Elle calcule une valeur mais sans modifier l'état d'un compte

Liens avec diagrammes d'états



- ◆ Possibilité de référencer un état d'un diagramme d'états associé à l'objet
 - ◆ `oclInState(etat)` : vrai si l'objet est dans l'état *etat*.
- ◆ Exemple
 - ◆ **context** Compte :: débiter(somme : int)
pre: somme > 0 **and** self.oclInState(Ouvert)
 - ◆ L'opération débiter ne peut être appelée que si le compte est dans l'état ouvert

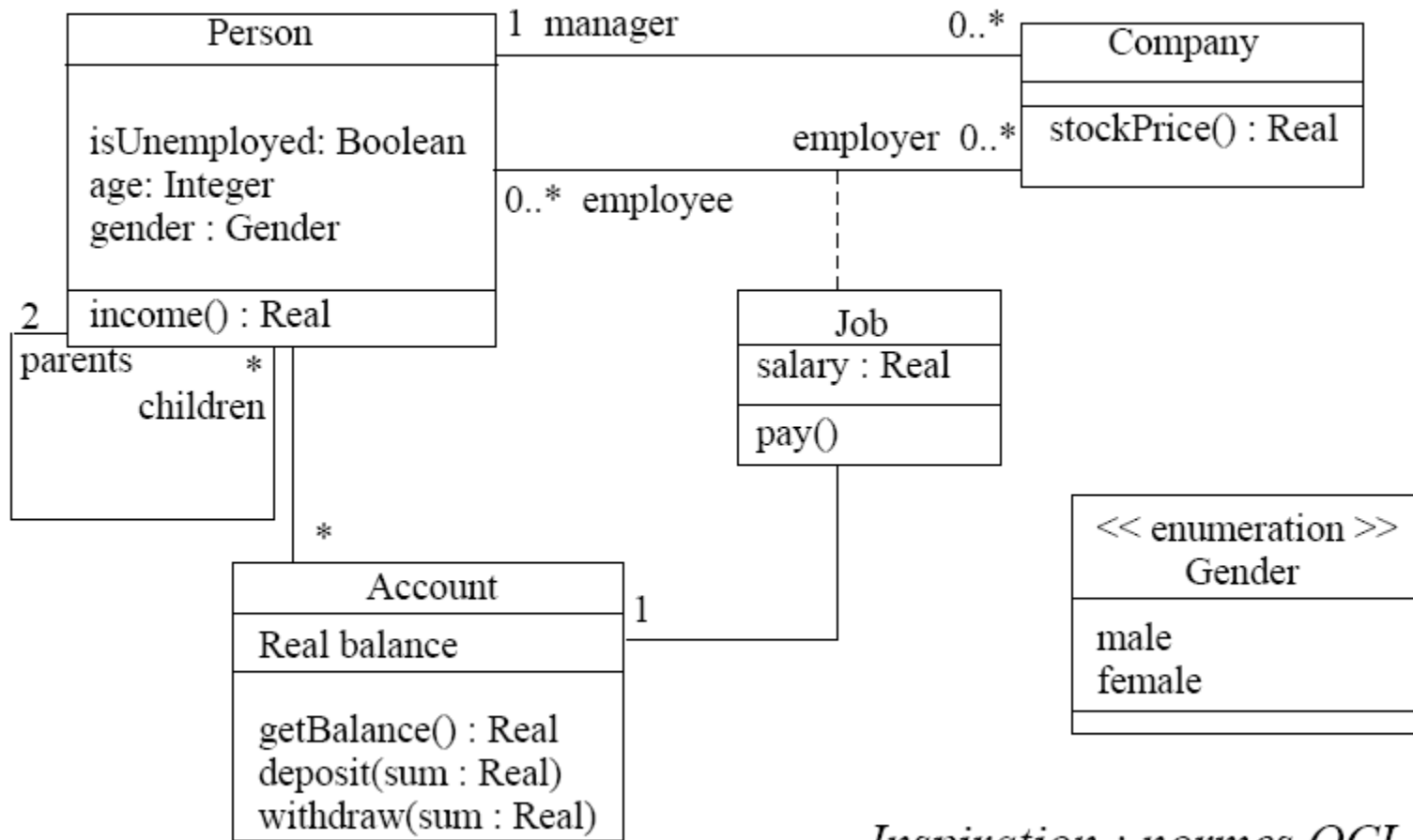
Propriétés

- ◆ Accède à la propriété d'un objet avec « . »
- ◆ Exemples
 - ◆ **context** Compte **inv**: self.solde > 0
 - ◆ **context** Compte **inv**: self.getSolde() > 0
- ◆ Accède à la propriété d'une collection avec « -> »
 - ◆ On peut utiliser « -> » également dans le cas d'un objet (= collection d'1 objet)

Accès aux attributs pour les collections

- ◆ Accès à un attribut sur une collection
 - ◆ Exemple dans contexte de Banque :
`compte.solde`
 - ◆ Renvoie l'ensemble des soldes de tous les comptes
- ◆ Forme raccourcie et simplifiée de
 - ◆ `compte -> collect (solde)`

Diagramme de classe



Inspiration : normes OCL

- 1- Dans une compagnie, un manager doit travailler et avoir plus de 40 ans. Le nombre d'employé d'une compagnie est non nul.
- 2- Une personne considérée comme au chômage ne doit pas avoir des revenus supérieurs à 500 €