OCL : Object Constraint Language

Une introduction

2 octobre 2019

Sommaire

Objectifs du langage

- Accompagner les diagrammes UML de descriptions :
  - précises
  - non ambiguës
    - vérifiables automatiquement sur les modèles
    - éventuellement compilables vers les langages
- Eviter les formalismes obscurs :
  - rester facile à écrire ...
  - et facile à lire,
  - utiliser la navigation des modèles

Historique

- Source d'inspiration : Syntropy (OMT + Z)
- 1997 : développement chez IBM (Jos Warmer)
- 1999 : intégration dans UML 1.1
- 2003 : OCL 2.0 dans UML 2.0
- 2014 : OCL 2.4, quelques précisions et corrections

Principes

Contrainte

- Expression à valeur booléenne qui s'attache à un élément UML
- Restriction ou informations complémentaires sur un modèle

Langage déclaratif

- Les contraintes ne sont pas opérationnelles
- Pas d'invocation de processus ni d'opérations autres que des requêtes
- Pas de description du comportement à adopter si une contrainte n'est pas respectée

Langage sans effet de bord

Les instances ne sont pas modifiées par les contraintes

Principes

Contextes d'utilisation

- Modèles
- Profils UML
- Méta-modèles

Types d'utilisation

- description d'invariants sur les classes et les types
- pré- / post-conditions sur les opérations et méthodes
- contraintes sur la valeur retournée
- règles de dérivation des attributs
- description de cibles pour messages et actions
- expression des gardes (diagrammes dynamiques)
- invariants de type pour les stéréotypes

La notion de contexte

Une contrainte OCL est liée à un *contexte*

$\rightarrow$ type, opération ou attribut

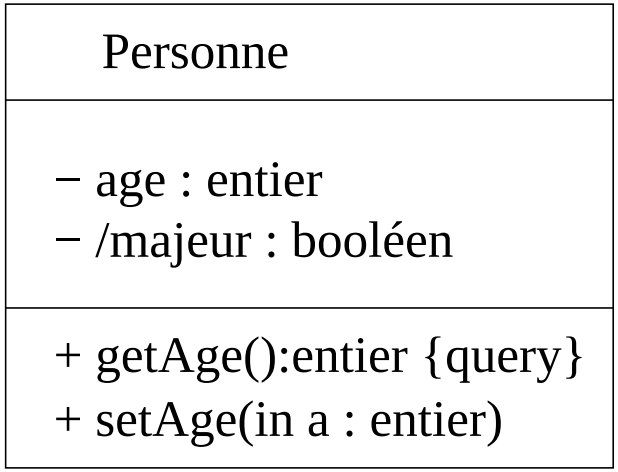
Contexte

```
context monContexte <stéréotype>:
    Expression de la contrainte
```

Le stéréotype peut prendre les valeurs suivantes :

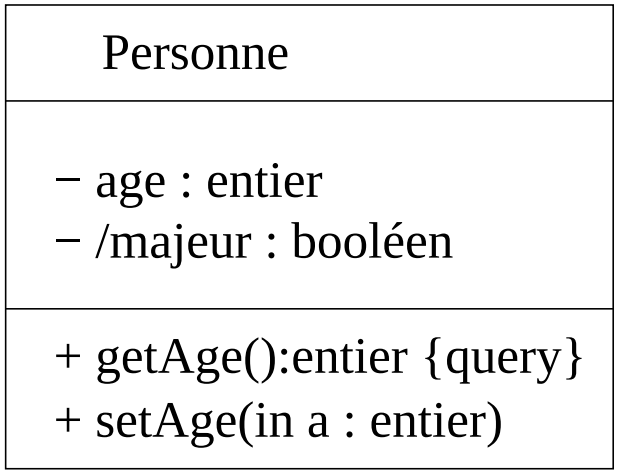
- `inv` invariant de classe
- `pre` précondition
- `post` postcondition
- `body` indique le résultat d'une opération `query`
- `init` indique la valeur initiale d'un attribut
- `derive` indique la valeur dérivée d'un attribut

Invariant de classe



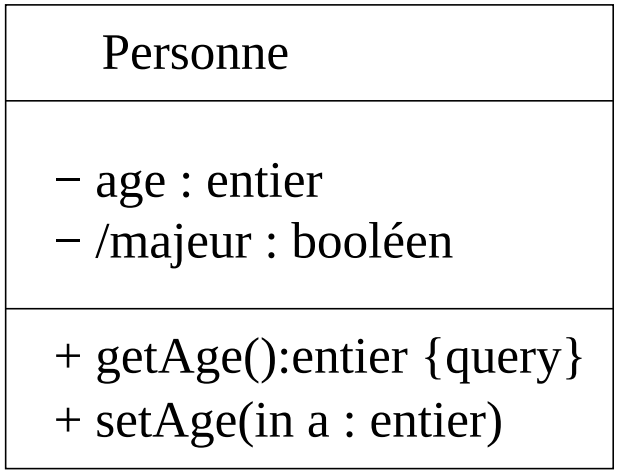

```
context Personne inv:
   (age <= 140) and (age >=0)
      - - l'âge est compris entre 0 et 140 ans
```

Pre/post conditions



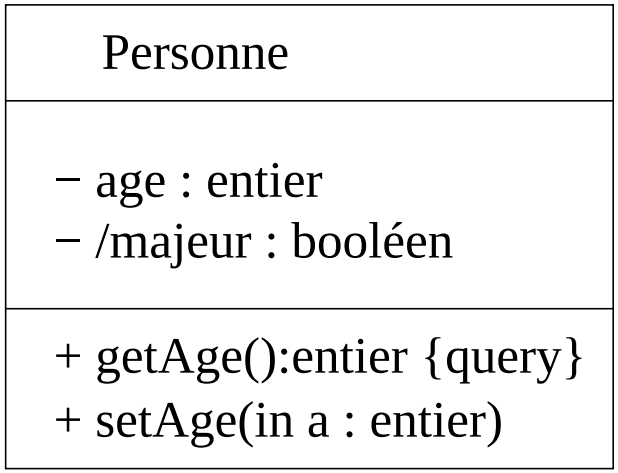

```
context Personne::setAge(a :entier)
   pre: (a <= 140) and (a >=0) and (a >= age)
   post: age = a - - on peut écrire également a=age
```

Body



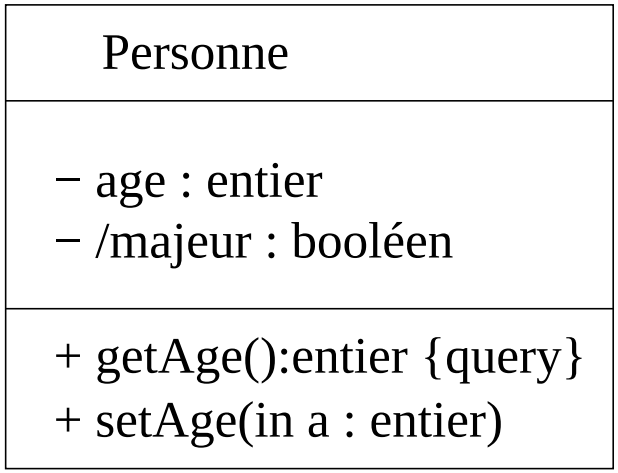

```
context Personne::getAge() :entier
   body: age
```

Initialisation



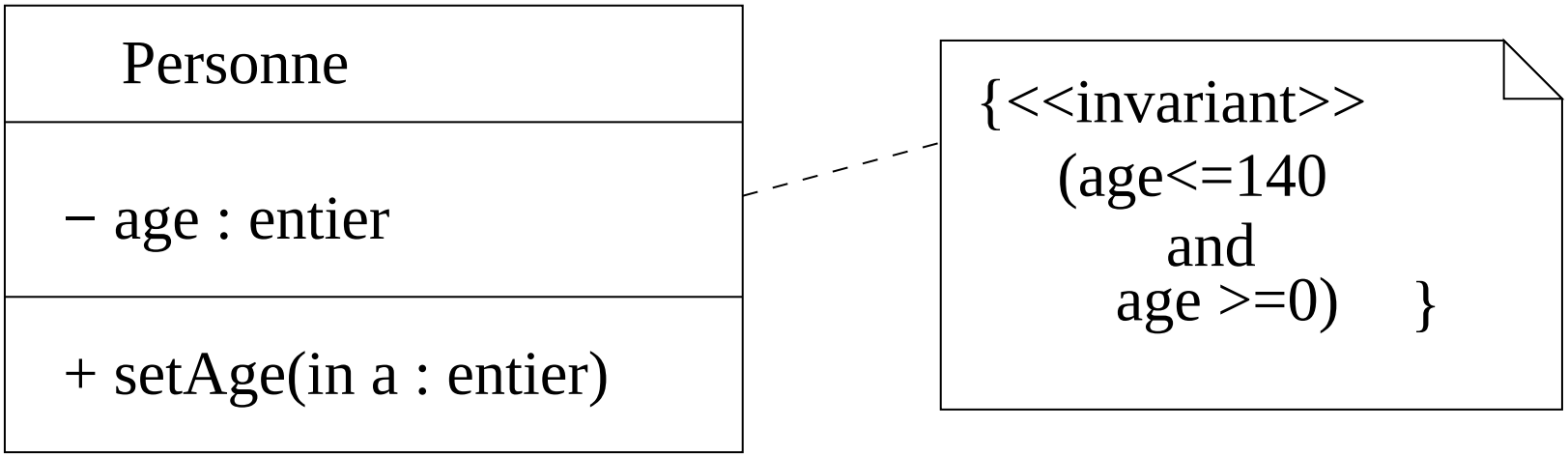

```
context Personne::age :entier
   init: 0
```

La notion de contexte

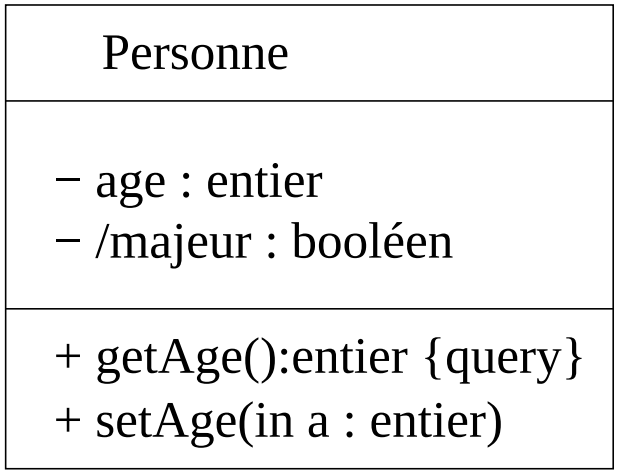



```
context Personne::majeur :booléen
   derive: age>=18
```

Version visuelle



Nommage de la contrainte

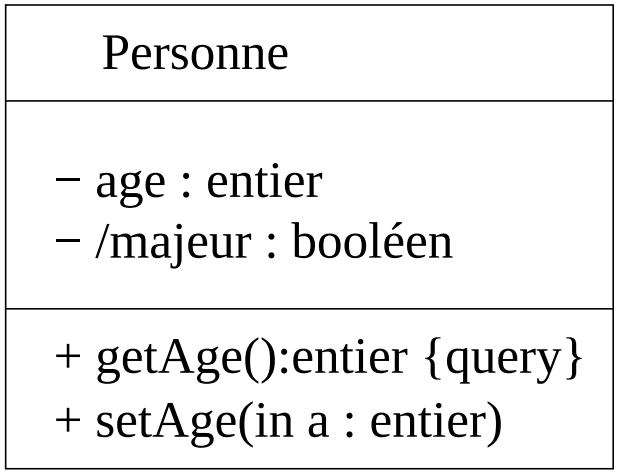


`context` Personne `inv` ageBorné :

(age <= 140) and (age >=0)

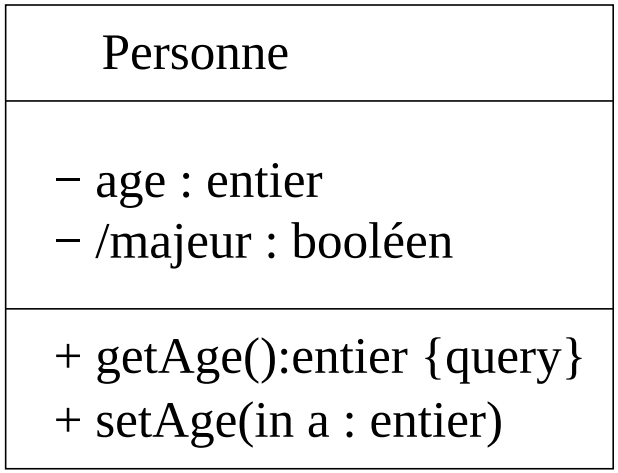
- - l'âge ne peut depasser 140 ans

Utilisation du mot-clef `self` pour désigner l'objet




```
context Personne inv:
    (self.age <= 140) and (self.age >=0)
    - - l'âge ne peut depasser 140 ans
```

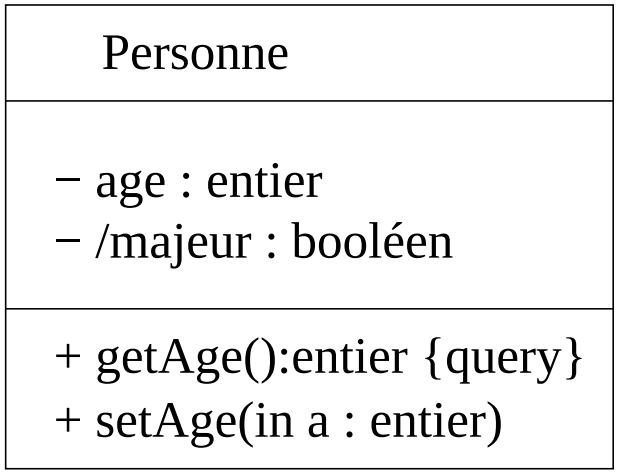
Utilisation du mot-clef `self` pour désigner l'objet



En ajoutant un attribut `mère` :
La mère d'une personne ne peut être cette personne elle-même et l'âge de la mère doit être supérieur à celui de la personne

```
context Personne inv:
    self.mère <> self and self.mère.age > self.age
```

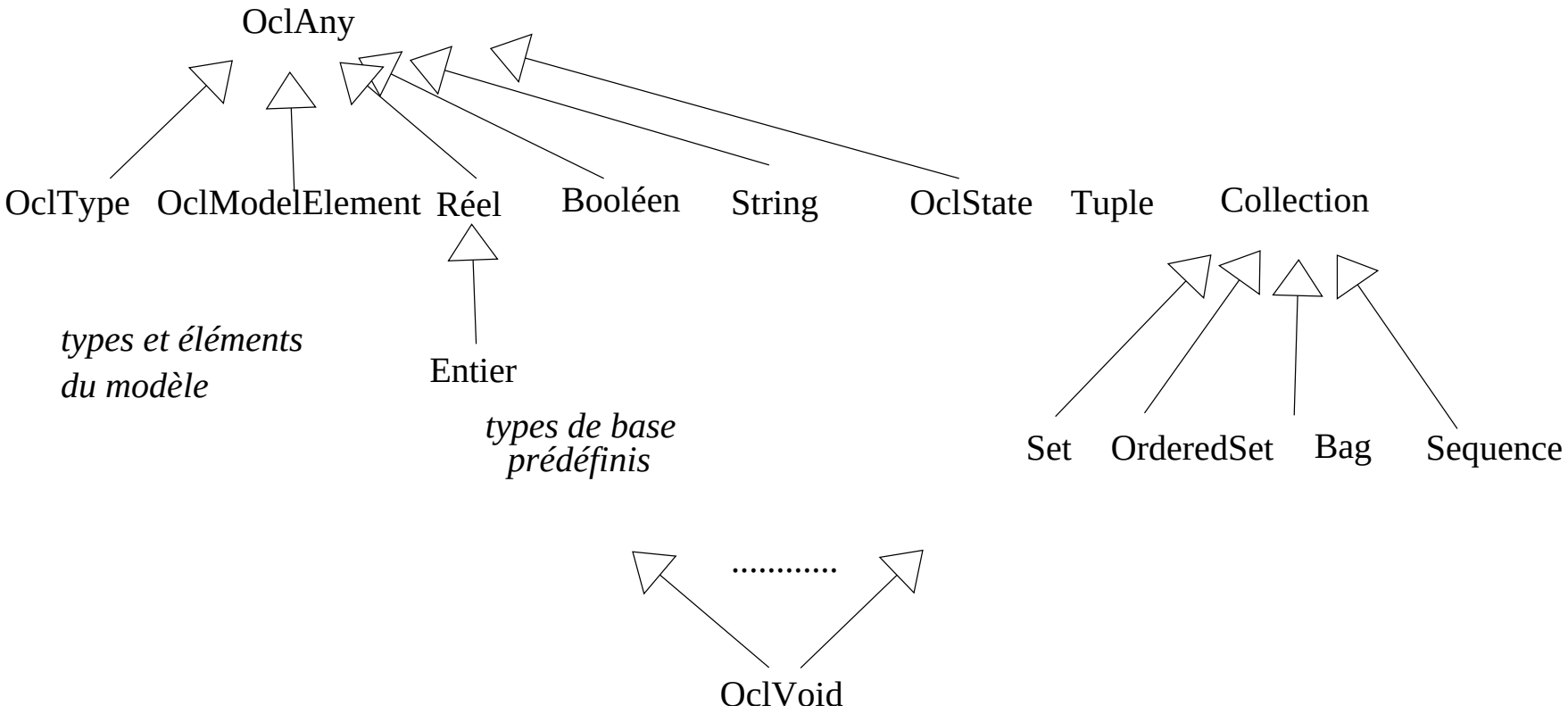
Utilisation d'un nom d'instance formel




```
context p :Personne inv:
    (p.age <= 140) and (p.age >=0)
    - - l'age ne peut depasser 140 ans
```

Sommaire

Hiérarchie des types



Types de base

Prédéfinis

- Entier (Integer)
- Réel (Real)
- String
- Booléen (Boolean)

Spéciaux

- OclModelElement (énumération des éléments du modèle)
- OclType (énumération des types du modèle)
- OclAny (tout type autre que Tuple et Collection)
- OclState (pour les diagrammes d'états)
- OclVoid sous-type de tous les types ; unique instance `null`

Entier

opérateurs = <> + - * / abs div mod max min < > <= >=
- est unaire ou binaire

Réel

opérateurs = <> + - * / abs floor round max min < > <= >=
- est unaire ou binaire

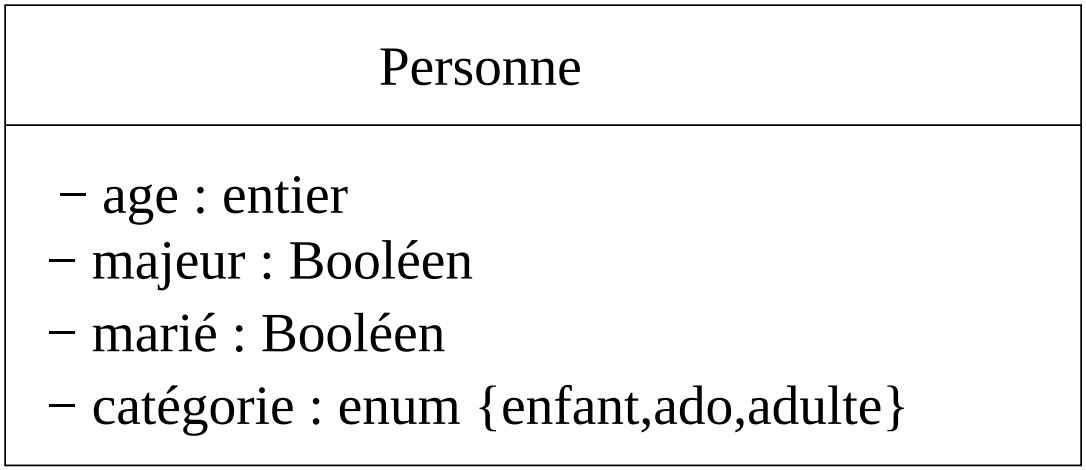
String

opérateurs = size() concat(String) toUpper() toLower()
substring(Entier, Entier)
Les chaînes de caractères constantes s'écrivent entre deux simples quotes :
'voici une chaîne'

Booléen

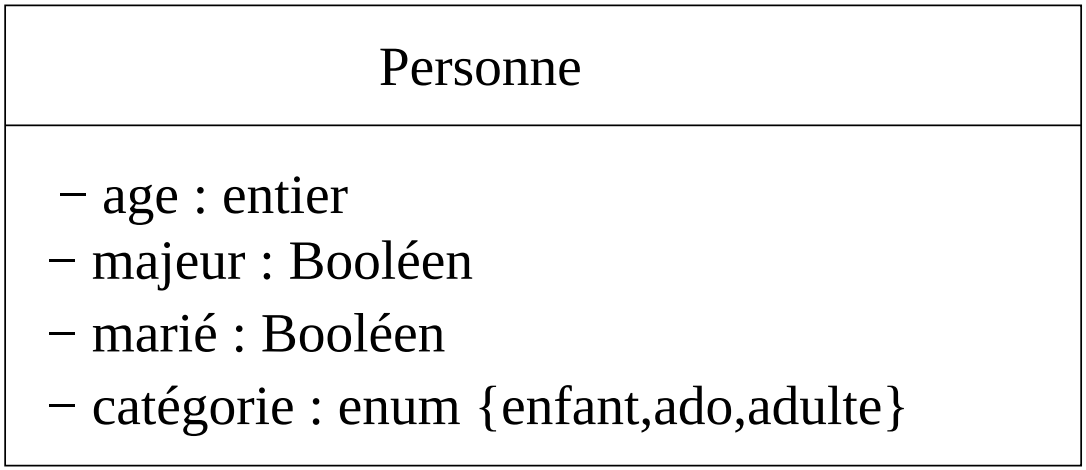
opérateurs = or xor and not
b1 implies b2
if b then expression1 else expression2 endif

Opérateurs booléens, exemples



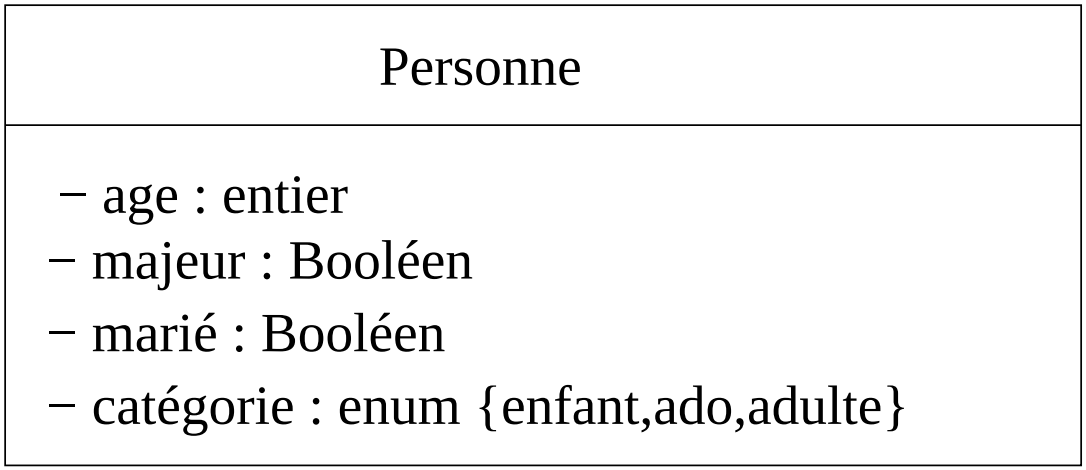

```
context Personne inv:
    marié implies majeur
```

Opérateurs booléens, exemples




```
context Personne inv:
    if age >=18 then majeur=vrai
    else majeur=faux
    endif
```

Opérateurs booléens, exemples



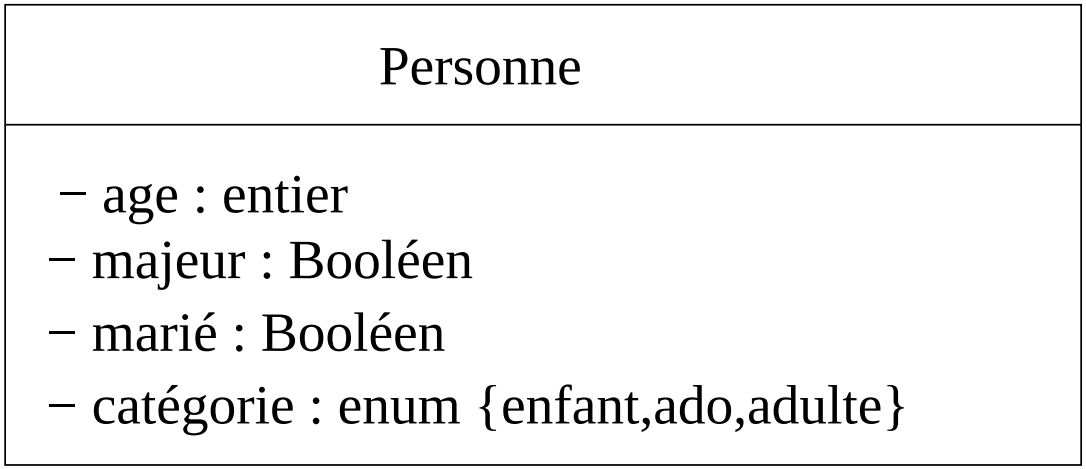

```
context Personne inv:
    majeur = age >=18
```

Précédence des opérateurs

. ->
not - unaire
* /
+ -
if then else
< > <= >=
<> =
and or xor
implies

Types énumérés

Leurs valeurs apparaissent précédées de #



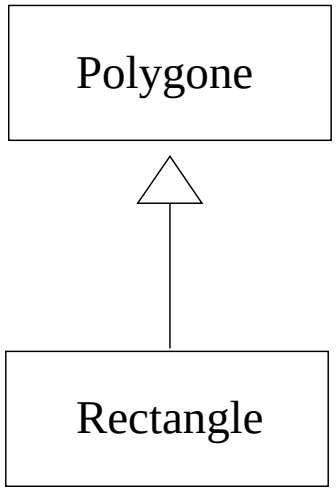

```
context Personne inv:
    if age <=12 then catégorie =#enfant
    else if age <=18 then catégorie =#ado
        else catégorie=#adulte
        endif
    endif
```

Types des modèles

Les types des modèles utilisables en OCL sont les « classificateurs (classifiers) », notamment les classes, les interfaces et les associations.

- oclAsType(t) (conversion ascendante ou descendante vers t)
  - la conversion ascendante sert pour l'accès à une propriété redéfinie
  - la conversion descendante sert pour l'accès à une nouvelle propriété
- oclIsTypeOf(t) (vrai si t est supertype direct)
- oclIsKindOf(t) (vrai si t est supertype indirect)

Types des modèles



p : Polygone
r : Rectangle

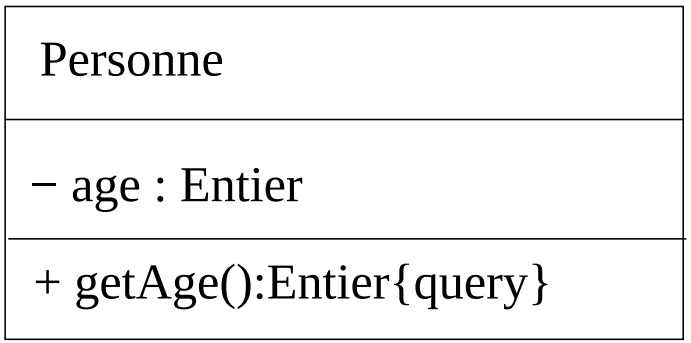
Figure – Polygones et rectangles

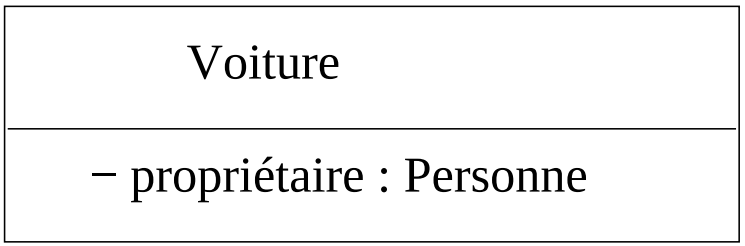
- p = r
- p.oclAsType(Rectangle)
- r.oclIsTypeOf(Rectangle) (vrai)
- r.oclIsKindOf(Polygone) (vrai)

Sommaire

Accès aux attributs

Opérateur d'accès noté '.'

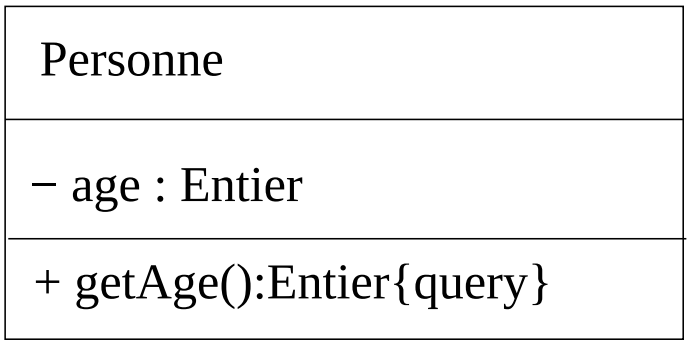


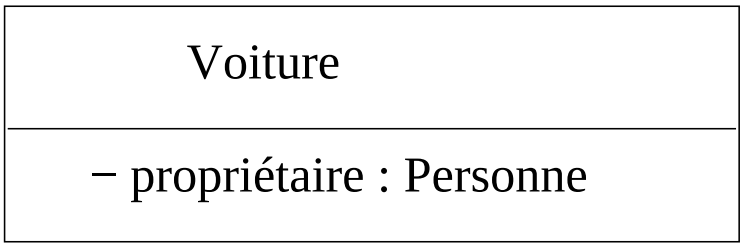



```
context Voiture inv propriétaireMajeur :
    self.propriétaire.age >= 18
```

Accès aux opérations query

Opérateur d'accès noté '.'



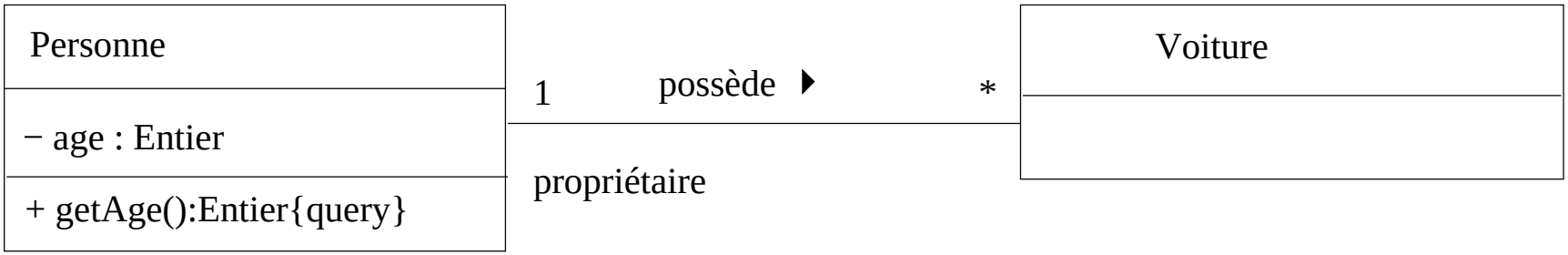



```
context Voiture inv:
    self.propriétaire.getAge() >= 18
```

Accès aux extrémités d'associations

La navigation le long des liens se fait en utilisant :

- soit les noms de rôles
- soit les noms des classes extrémités en mettant leur première lettre en minuscule, à condition qu'il n'y ait pas ambiguïté

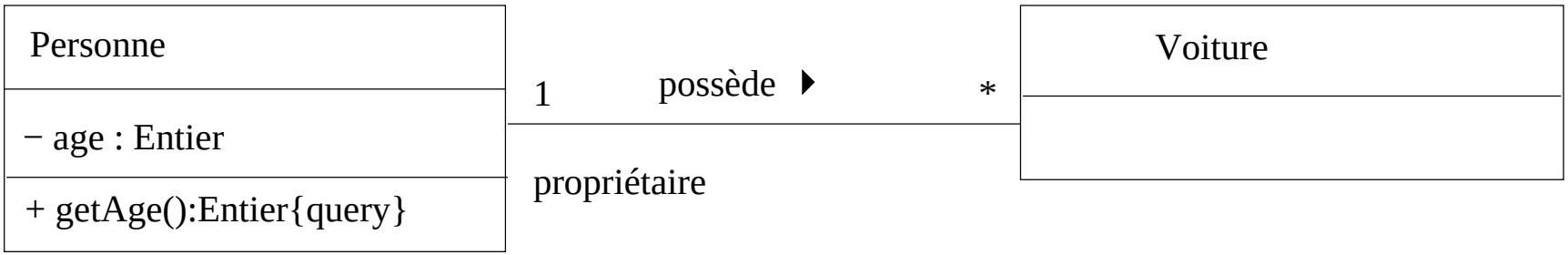



```
context Voiture inv:
    self.propriétaire.age >= 18
```

Accès aux extrémités d'associations

La navigation le long des liens se fait en utilisant :

- soit les noms de rôles
- soit les noms des classes extrémités en mettant leur première lettre en minuscule, à condition qu'il n'y ait pas ambiguïté

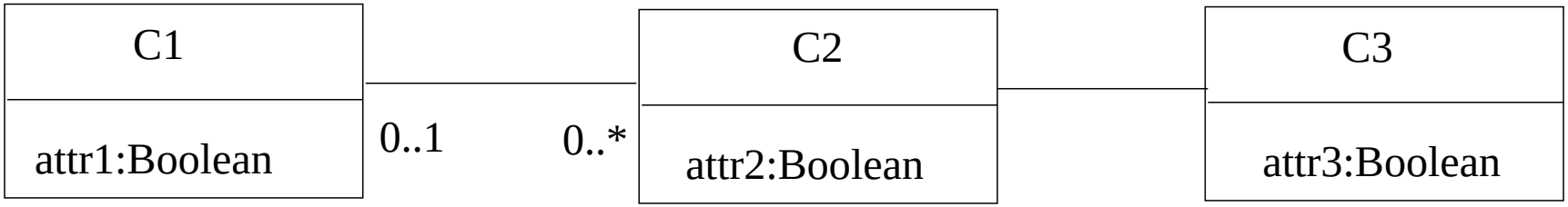



```
context Voiture inv:
    self.personne.age >= 18
```

Ambiguïté

Deux associations entre Voiture et Personne (possède et aConduit)

Influence du contexte

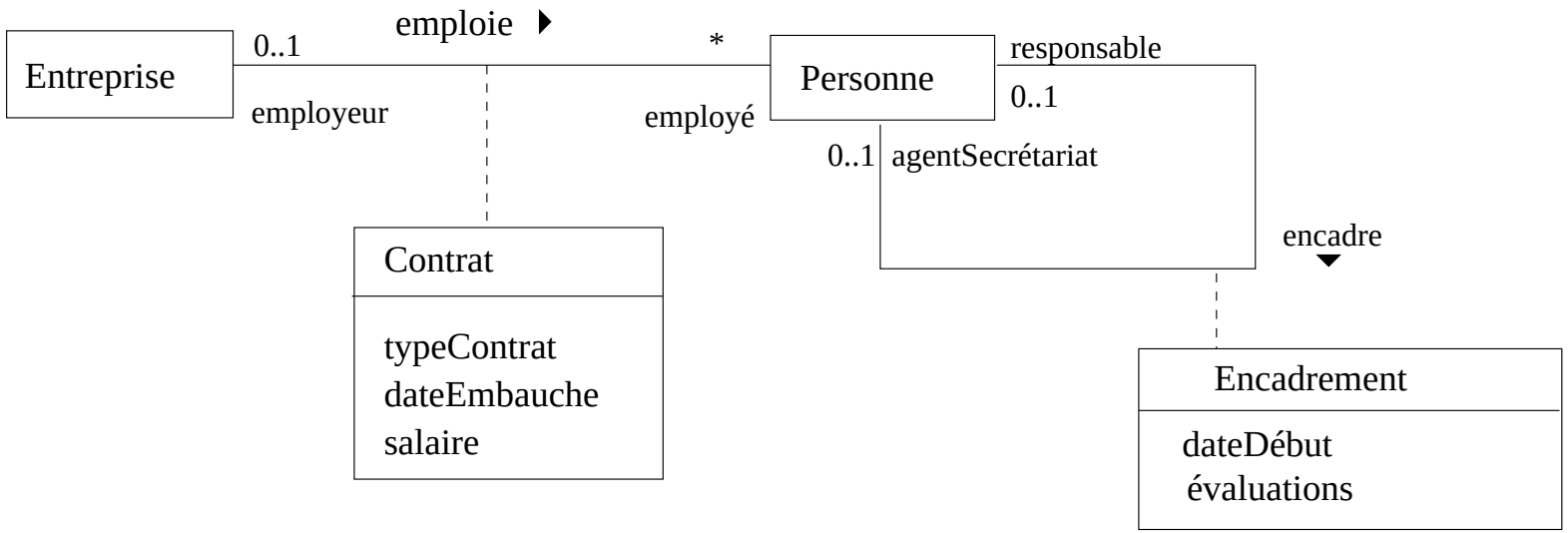



```
context C1 inv:
```
c2.attr2=c2.c3.attr3
- - pour des instances de C2 et C3 liées entre elles
- - et celle de C2 liée avec une instance de C1
- - les attributs attr2 et attr3 sont égaux

```
context C2 inv:
```
attr2=c3.attr3
- - pour des instances de C2 et C3 liées entre elles
- - les attributs attr2 et attr3 sont égaux

Navigation vers les classes association

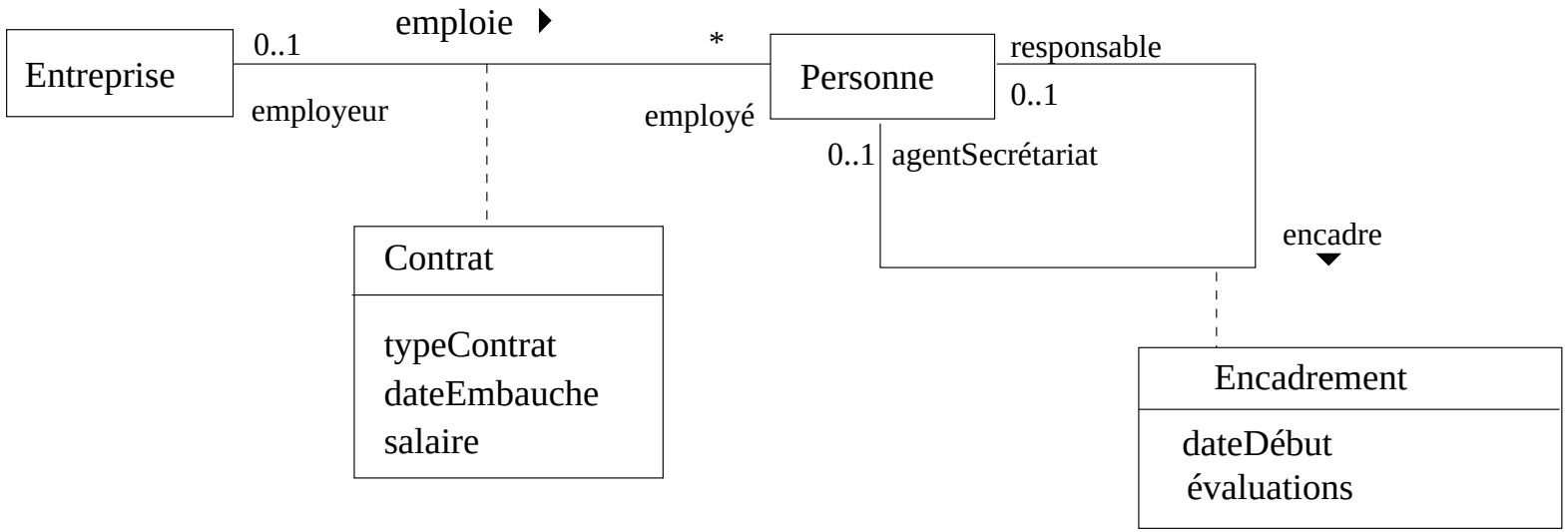
En utilisant le nom de la classe (premier caractère en minuscule)




```
context p :Personne inv:
   p.contrat.salaire >= 0
```

Navigation vers les classes association

En utilisant le nom de rôle opposé (obligatoire pour une association réflexive)

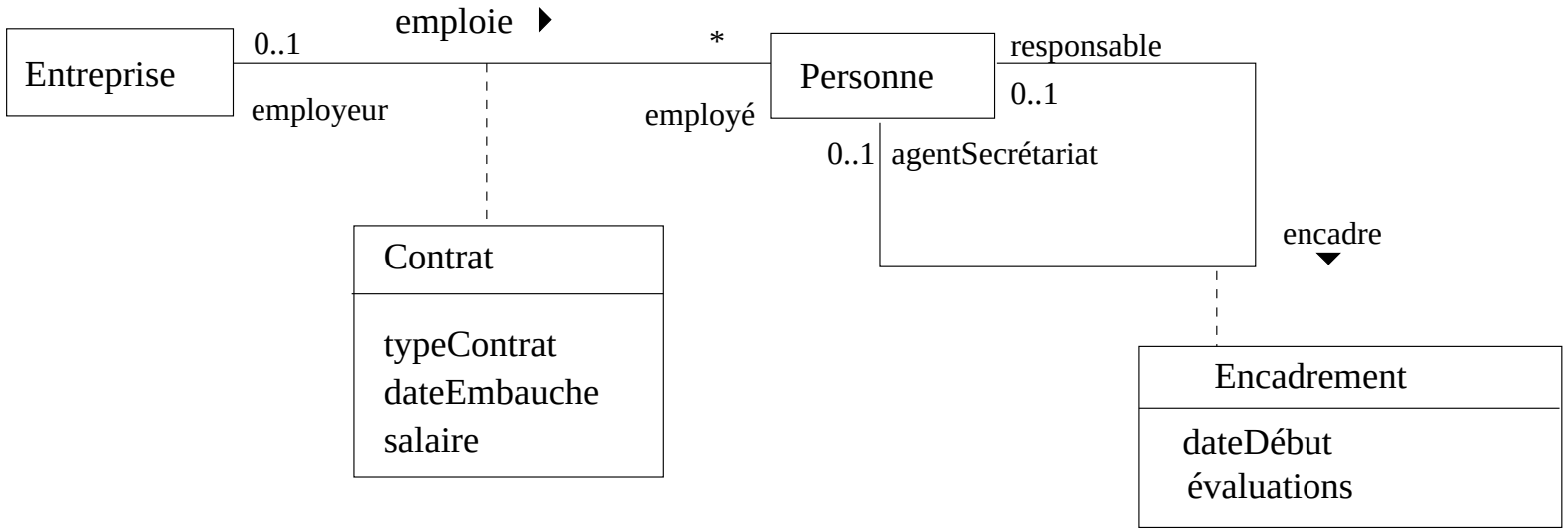



```
context p :Personne inv:
    p.contrat[employeur].salaire >= 0
```

Navigation depuis une classe association

Utilise les noms de rôles ou de classes

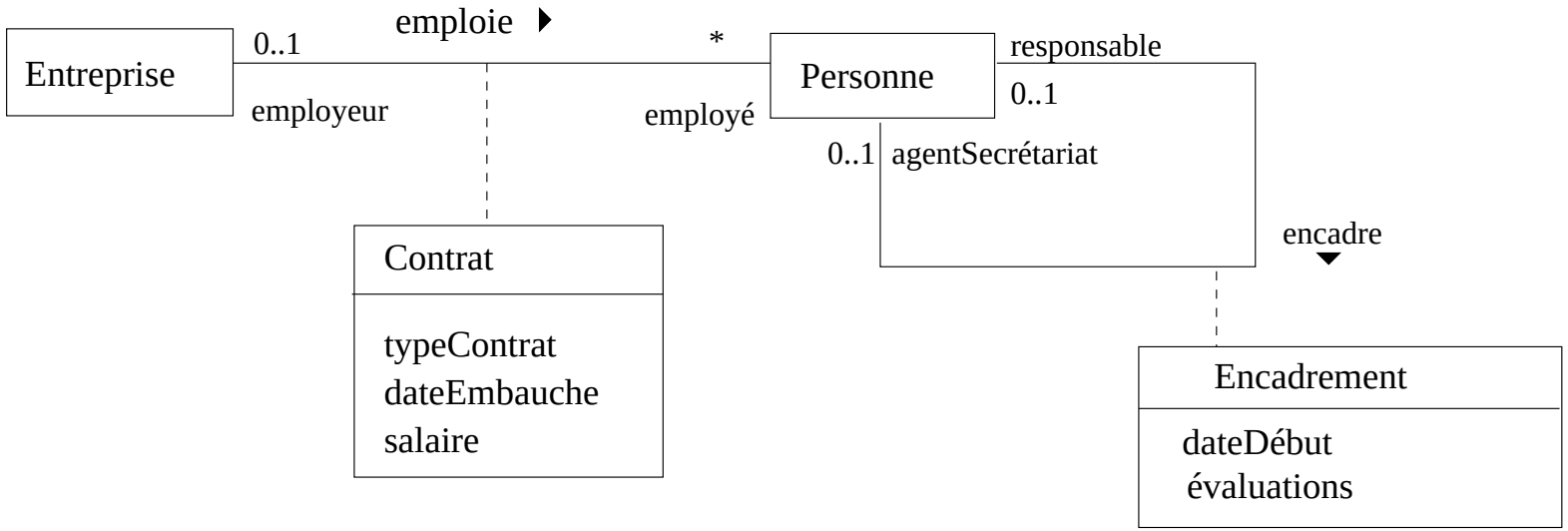
Navigation rôle vers classe ne peut donner qu'un objet !




```
context c :Contrat inv:
   c.employé.age >= 16
```

Navigation

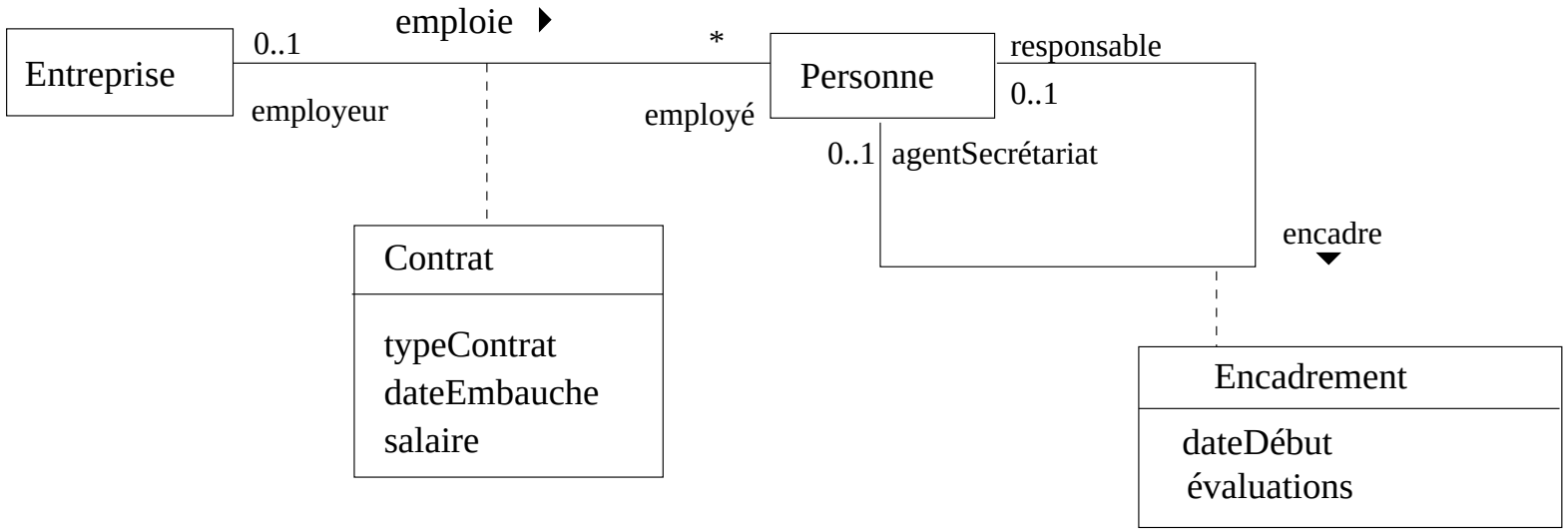
Le salaire d'un agent de secrétariat est inférieur à celui de son responsable



`context e :encadrement inv:`
e.responsable.contrat.salaire >= e.agentSecrétariat.contrat.salaire

Navigation

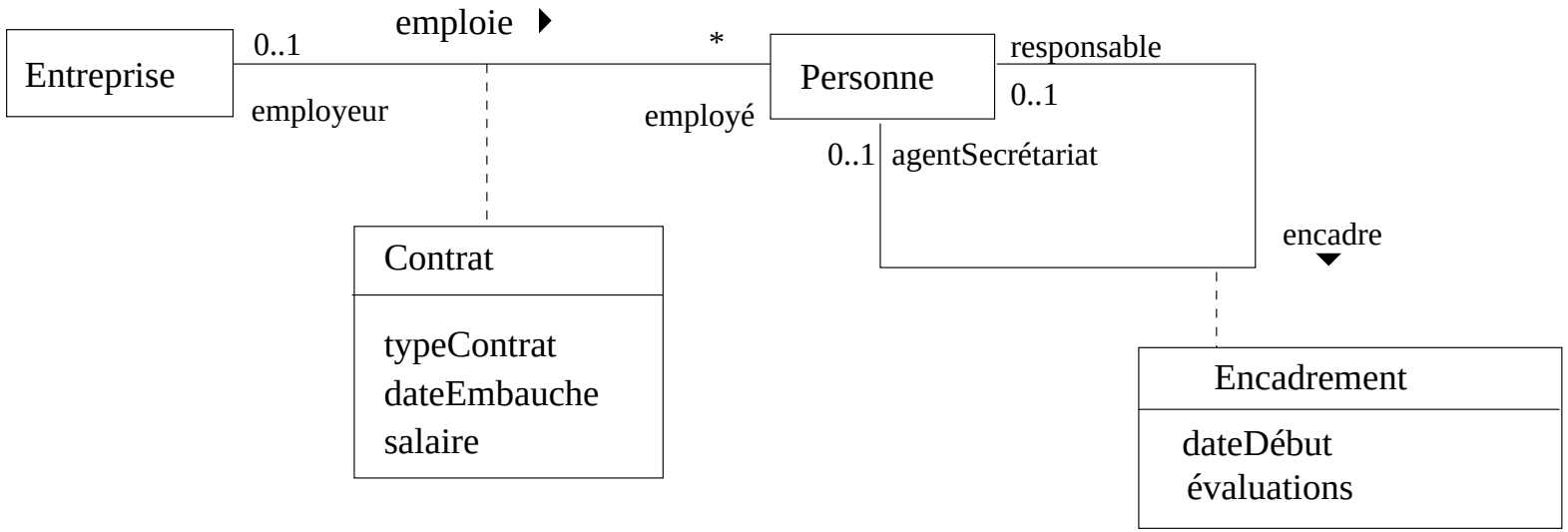
Un agent de secrétariat a un type de contrat "agentAdministratif"




```
context e :encadrement inv:
    e.agentSecrétariat.contrat.typeContrat='agentAdministratif'
```

Navigation

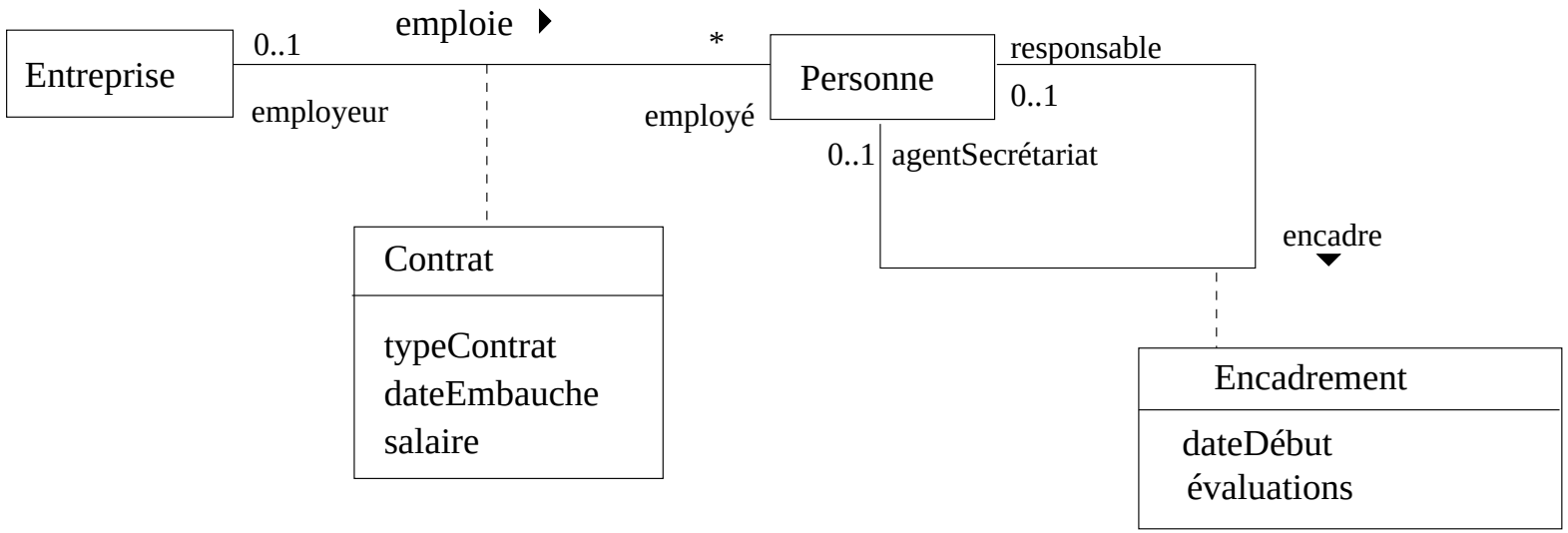
Un agent de secrétariat a une date d'embauche antérieure à la date de début de l'encadrement (on suppose que les dates sont des entiers)




```
context e :encadrement inv:
    e.agentSecrétariat.contrat.dateEmbauche <= e.dateDebut
```

Navigation (effet du contexte)

Un agent de secrétariat a une date d'embauche antérieure à la date de début de l'encadrement (on suppose que les dates sont des entiers)




```
context p :Personne inv:
   p.agentSecrétariat.contrat.dateEmbauche
         <= p.encadrement[agentSecrétariat].dateDebut
```

Ajouter que p.agentSecrétariat est valué (voir plus loin dans le cours)

Sommaire

Désignation de la valeur antérieure

Valeur d'une propriété avant l'exécution de l'opération :
suffixer le nom de la propriété avec `@pre`

```
context Personne::feteAnniversaire()
  pré: age < 140
  post: age = age@pre + 1
```

Définition de variables

```
let variable : type = expression1 in
expression2
```

Ex. avec l'attribut dérivé `impot` dans Personne :

```
context Personne inv:
let montantImposable : Réel = contrat.salaire*0.8 in
if (montantImposable >= 100000)
then impot = montantImposable*45/100
else if (montantImposable >= 50000)
    then impot = montantImposable*30/100
    else impot = montantImposable*10/100
    endif
endif
```

Définition de variables

Variable utilisable dans plusieurs contraintes de la classe, on peut utiliser la construction `def`

```
context Personne
def : montantImposable : Réel = contrat.salaire*0.8
```

Définition d'opération

```
context Personne
def: ageCorrect(a :Réel) :Booléen = a>=0 and a<=140
```

Usage pour simplifier :

```
context Personne::setAge(a :entier)
   pre: ageCorrect(a) and (a >= age)

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

Retour de méthode

L'objet retourné par une opération est désigné par `result`.

```
context Personne::getAge()
  post: result=age
```

Equivalent à l'utilisation de `body` lorsque la postcondition se résume à décrire la valeur du résultat

Objet créé dans une méthode

`oclIsNew` : utilisée dans une postcondition

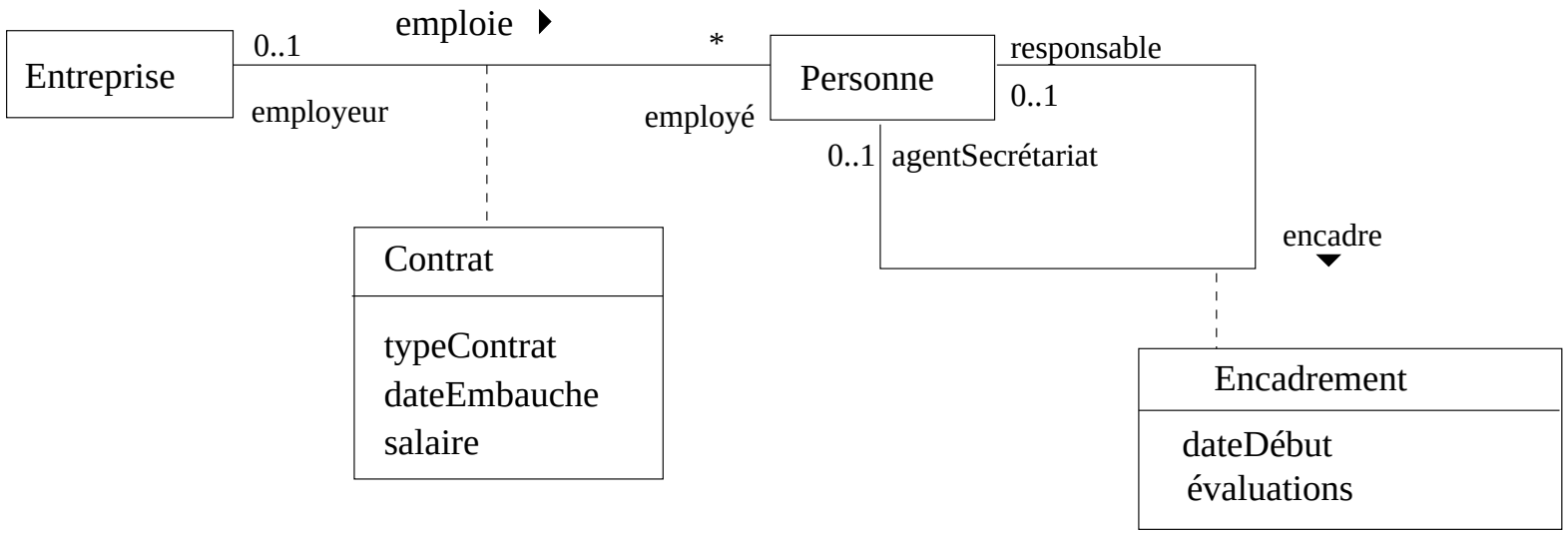
```
context Personne::donneNaissance() :Personne
  post: result.oclIsNew() and result.age=0
```

L'objet est créé pendant l'opération
l'objet n'existait pas au moment des préconditions

Existence d'une valeur pour une propriété

`isEmpty() / notEmpty()` :

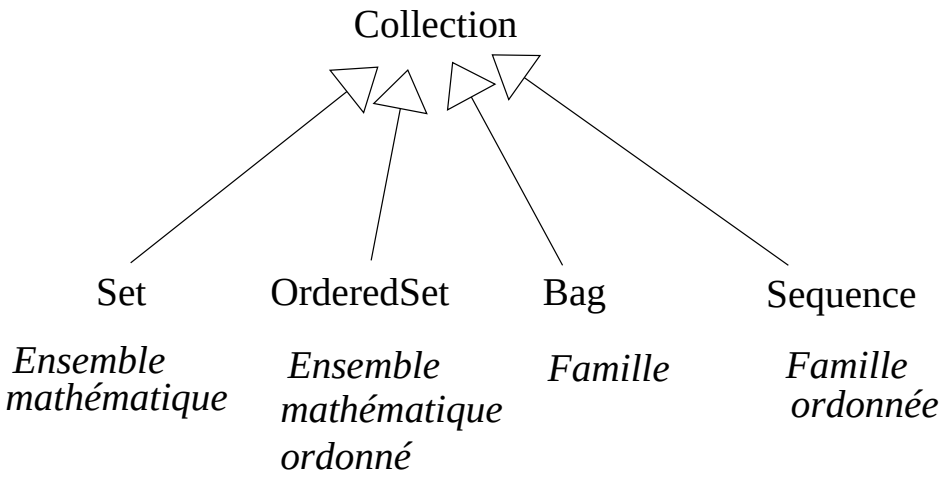
→ la collection des valeurs est vide / non vide




```
context p :Personne inv:
p.agentSecrétariat.notEmpty() implies
    p.agentSecrétariat.contrat.dateEmbauche
            <= p.encadrement[agentSecrétariat].dateDebut
```

Sommaire

Les types de collections



- `Collection` est un type abstrait
- `Set` notion mathématique d'ensemble
- `OrderedSet` notion mathématique d'ensemble ordonné
- `Bag` notion mathématique de famille (doublons)
- `Sequence` famille avec éléments ordonnés.

Définitions par des littéraux

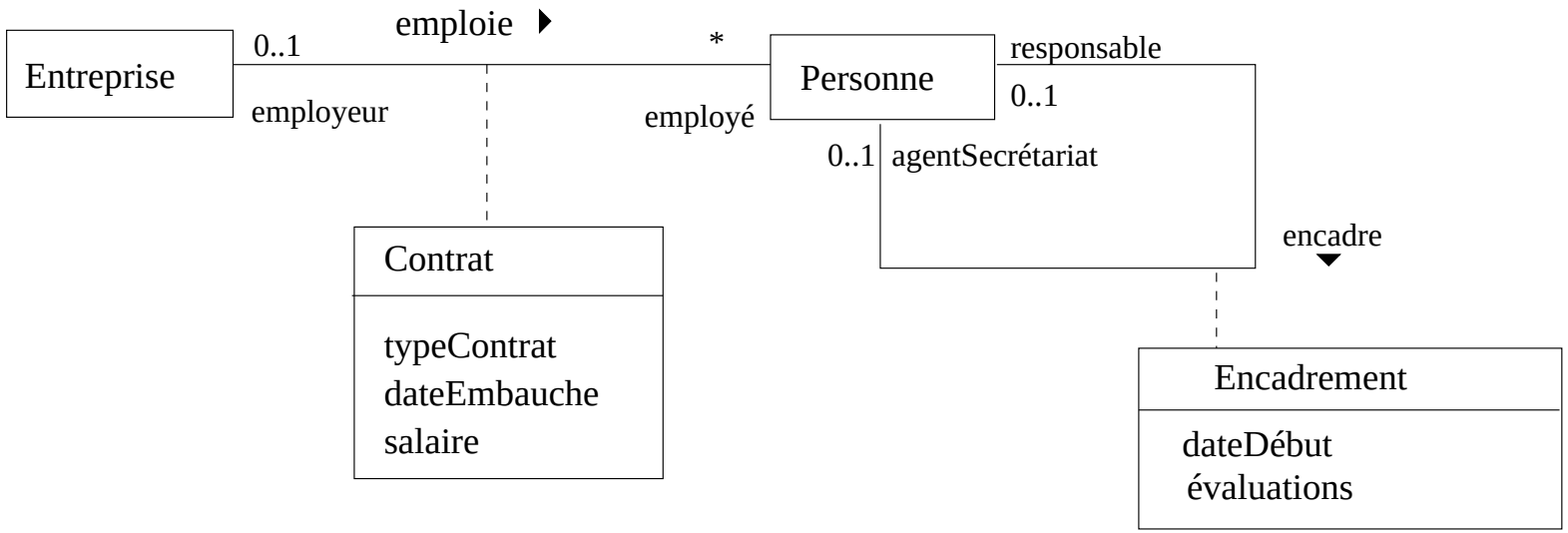
- `Set { 2, 4, 6, 8 }`
- `OrderedSet { 2, 4, 6, 8 }`
- `Bag { 2, 4, 4, 6, 6, 8 }`
- `Sequence { 'le', 'chat', 'boit', 'le', 'lait' }`
- `Sequence { 1..10 }` spécification d'un intervalle d'entiers

A partir d'OCL 2.0 : plusieurs niveaux d'imbrication possibles
`Set { 2, 4, Set {6, 8 }}`

Collection et navigation

Set

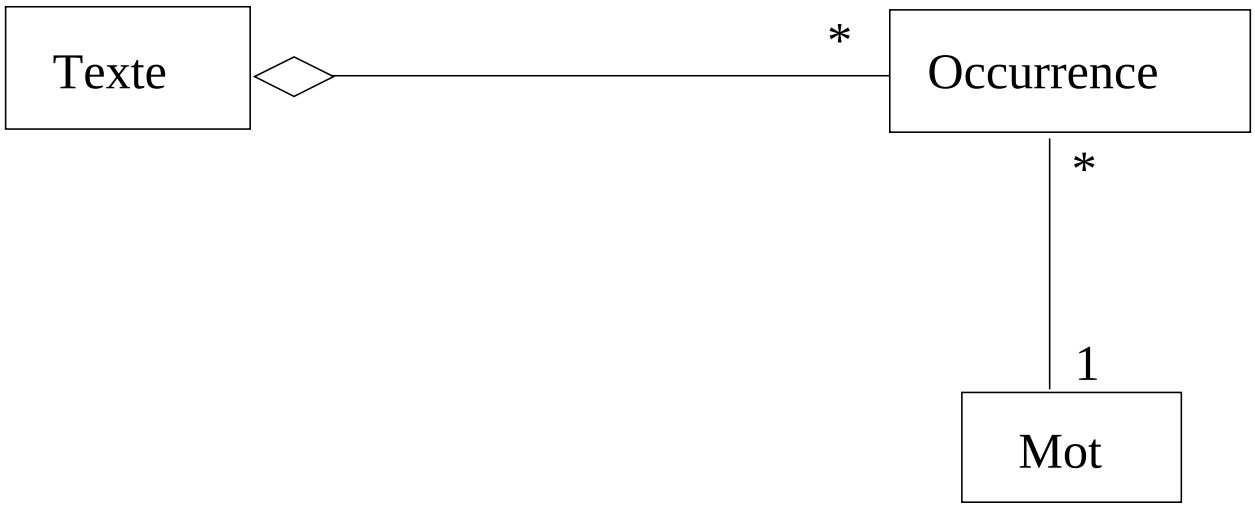
Navigation le long d'une association ordinaire (ne mentionant pas bag ou seq à l'une de ses extrémités)
`self.employé` dans le contexte `Entreprise`



Collection et navigation

Bag

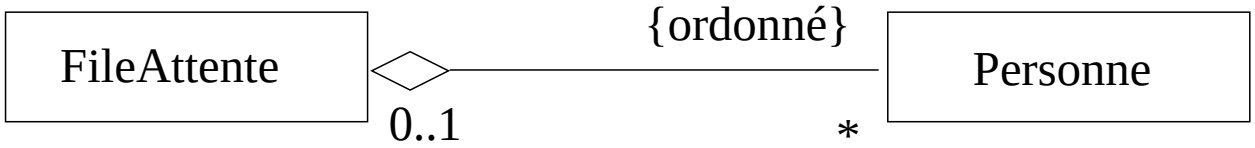
Navigation le long d'une association portant bag ou navigation multiple `self.occurrence.mot` dans le contexte `Texte`



Collection et navigation

OrderedSet

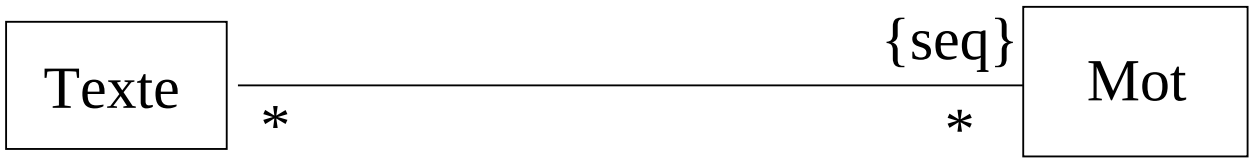
Navigation vers une extrémité d'association munie de la contrainte `ordonné` `self.personne` dans le contexte `FileAttente`



Collection et navigation

Sequence

Navigation vers une extrémité d'association munie de la contrainte seq `self.mot` dans le contexte `Texte`



Collection et navigation

Accès à un attribut multi-valué

- bag si une valeur peut être répétée plusieurs fois
- séquence si les valeurs sont de plus ordonnées
- set s'il n'y a jamais de doublon dans les valeurs
- orderedSet si les valeurs sont de plus ordonnées.

Opérations de `Collection`

Trois remarques :

- nous considérons des collections d'éléments de type `T`
- les opérations sur une collection sont généralement mentionnées avec `->`
- Les opérations qui prennent une expression comme paramètre peuvent déclarer optionnellement un itérateur,
  toute opération de la forme `operation(expression)` existe également sous deux formes plus complexes
  `operation(v | expression-contenant-v)`
  `operation(v : T | expression-contenant-v)`

Opérations sur tous les types de collections

isEmpty() :Boolean notEmpty() :Boolean

Permettent de tester si la collection est vide ou non. Ils servent en particulier à tester l'existence d'un lien dans une association dont la multiplicité inclut 0

size() :Entier

Donne la taille (nombre d'éléments)

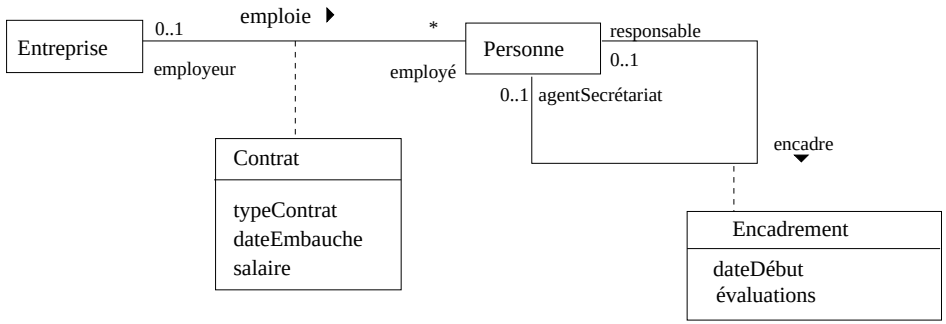
count(unObjet :T) :Entier

Donne le nombre d'occurrences de unObjet

Opérations sur tous les types de collections

sum() :T

Addition de tous les éléments de la collection (qui doivent supporter une opération + associative)




```
context e :Entreprise
    def: ageMoyen : Real = e.employé.age->sum() /
                           e.employé.age->size()
```

Opérations sur tous les types de collections

includes(o :T) :Boolean excludes(o :T) :Boolean

Retourne vrai (resp. faux) si et seulement si o est (resp. n'est pas) un élément de la collection

Soit une classe `Etudiant`, disposant d'un attribut `notes:Real[*]`, on peut définir l'opération `estEliminé():Boolean`.
Un étudiant est éliminé s'il a obtenu un zéro.

```
context e :Etudiant
  def: estEliminé() : Boolean = e.notes->includes(0)
```

Opérations sur tous les types de collections

includesAll(c :Collection(T)) :Boolean
excludesAll(c :Collection(T)) :Boolean

Retourne vrai (resp. faux) si et seulement si la collection contient tous les (resp. ne contient aucun des) éléments de c

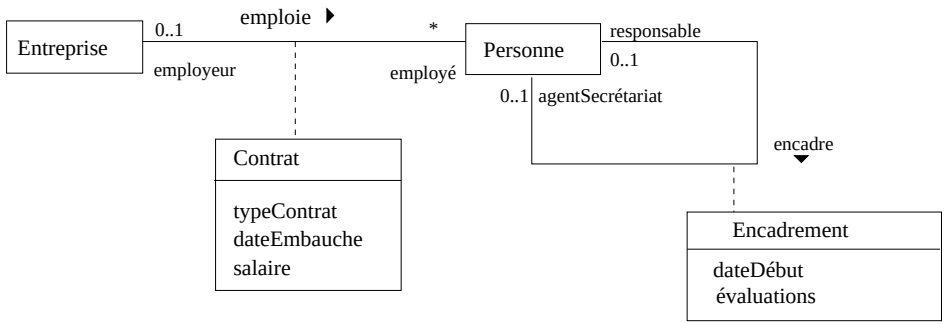
forAll(uneExpression :Boolean) :Boolean

Vaut vrai si et seulement si `uneExpression` est vraie pour tous les éléments de la collection

Opérations sur tous les types de collections

forAll

Dans une entreprise, une contrainte est que tous les employés aient plus de 16 ans.

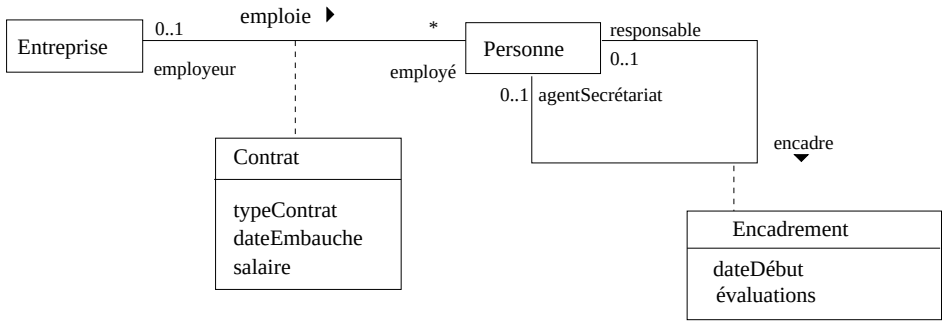



```
context e :Entreprise inv:
    e.employe->forAll(emp :Personne | emp.age >= 16)
```

Opérations sur tous les types de collections

forAll(t1,t2 :T | expression-contenant-t1-et-t2)

Variante permettant d'itérer avec plusieurs itérateurs sur la même collection : parcours des ensembles produits de la même collection



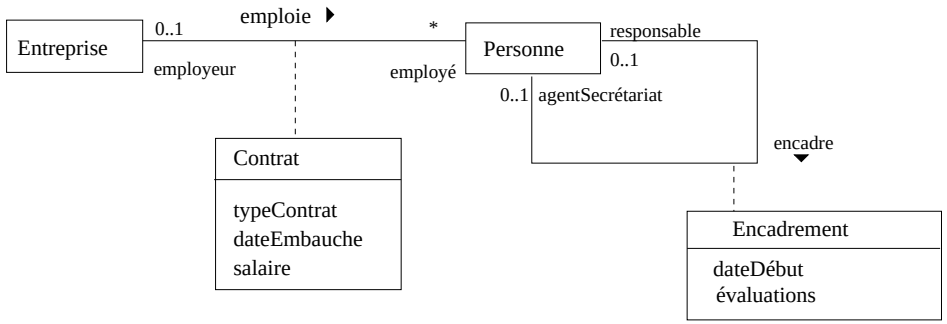
Dans une entreprise, deux employés différents n'ont jamais le même nom

```
context e :Entreprise inv:
   e.employe->forAll(e1,e2 :Personne |
         e1 <> e2 implies e1.nom <> e2.nom)
```

Opérations sur tous les types de collections

exists(uneExpression :Boolean)

Vrai si et seulement si au moins un élément de la collection satisfait `uneExpression`



Dans toute entreprise, il y a au moins une personne dont le contrat est de type "agent administratif".

```
context e :Entreprise inv:
    e.contrat->exists(c :Contrat |
        c.typeContrat='agent administratif')
```

Opérations sur tous les types de collections

iterate(i :T ; acc :Type = uneExpression | ExpressionAvecietacc)

Cette opération permet de généraliser et d'écrire la plupart des autres.
i est un itérateur sur la collection
acc est un accumulateur initialisé avec *uneExpression*
L'expression *ExpressionAvecietacc* est évaluée pour chaque *i* et son résultat est affecté dans *acc*. Le résultat de *iterate* est *acc*

Caractérisation du résultat d'une opération `masseSalariale` d'une entreprise

```
context Entreprise::masseSalariale() :Real post:
result = employé->iterate(p :Personne ; ms :Real=0 |
            ms+p.contrat.salaire)
```

Opérations sur tous les types de collections

Ecriture de `size` et `forAll` dans le contexte de la classe Collection

`context` Collection : :size() `post:`
result = self->iterate(elem ; acc :Integer=0| acc+1)

`context` Collection : :forAll(expr) `post:`
result = self->iterate(elem ; acc :Boolean=true| acc and expr)

Opérations sur tous les types de collections

isUnique(uneExpression :BooleanExpression) :Boolean

Vrai si et seulement si `uneExpression` s'évalue avec une valeur distincte pour chaque élément de la collection.

any(uneExpression :OclExpression) :T

Retourne n'importe quel élément de la collection qui vérifie `uneExpression`

one(uneExpression :OclExpression) :Booléen

Vrai si et seulement si un et un seul élément de la collection vérifie `uneExpression`

Opérations sur tous les types de collections

sortedBy(uneExpression) :Sequence

Retourne une séquence contenant les éléments de la collection triés par ordre croissant suivant le critère `uneExpression`. L'évaluation de `uneExpression` doit donc supporter l'opération $<$.

Post condition d'une méthode `salariésTriés` dans le contexte de la classe `Entreprise`.
Cette méthode retourne les employés ordonnés suivant leur salaire.

```
context Entreprise : : salariesTries() :OrderedSet(Personne) post:
result = self.employe->sortedBy(p | p.contrat.salaire)
```

Opérations sur tous les types de collections, définies spécifiquement sur chaque type

Exemple
`select`, commune aux trois types de collection concrets, n'est pas définie dans `Collection`, mais seulement dans les types concrets avec des signatures spécifiques :

```
select(expr:BooleanExpression): Set(T)
select(expr:BooleanExpression): OrderedSet(T)
select(expr:BooleanExpression): Bag(T)
select(expr:BooleanExpression): Sequence(T)
```

Dans un tel cas, nous présentons une généralisation de ces opérations

```
select(expr:BooleanExpression): Collection(T)
```

Opérations sur tous les types de collections, définies spécifiquement sur chaque type

select(expr :BooleanExpression) : Collection(T)
reject(expr :BooleanExpression) : Collection(T)

Retourne une collection du même type construite par sélection des éléments vérifiant (resp. ne vérifiant pas) `expr`

L'expression représentant dans le contexte d'une entreprise les employés âgés de plus de 60 ans serait la suivante :

```
self.employé->select(p:Personne|p.age>=60)
```

Opérations sur tous les types de collections, définies spécifiquement sur chaque type

collect(expr :BooleanExpression) : Collection(T)

Retourne une collection composée des résultats successifs de l'application de `expr` à chaque élément de la collection.

La manière standard d'atteindre les mots d'un texte s'écrit ainsi dans le contexte d'un texte :

```
self.occurrence->collect(mot)
self.occurrence->collect(o | o.mot)
self.occurrence->collect(o:Occurrence | o.mot)
```

Comme nous l'avons évoqué précédemment, cette notation admet un raccourci :

```
self.occurrence.mot
```

Opérations sur tous les types de collections, définies spécifiquement sur chaque type

including(unObjet :T) :Collection(T)
excluding(unObjet :T) :Collection(T)

Retourne une collection résultant de l'ajout (resp. du retrait) de `unObjet` à la collection

Post-condition d'une opération `embauche(p:Personne)` de la classe `Entreprise`.

```
context Entreprise : :embauche(p :Personne) post:
  self.employé=self.employé@pre->including(p)
```

Opérations sur tous les types de collections, définies spécifiquement sur chaque type

opération =

union(c :Collection) :Collection

Opérations de conversion

Chaque sorte de collection (set, orderedSet, sequence, bag) peut être transformée dans n'importe quelle autre sorte
Par exemple un `Bag` peut être transformé en :

- sequence par l'opération `asSequence():Sequence(T)` ; l'ordre résultant est indéterminé
- ensemble par l'opération `asSet():Set(T)` ; les doublons sont éliminés
- ensemble ordonné par l'opération `asOrderedSet():OrderedSet(T)` ; les doublons sont éliminés ; l'ordre résultant est indéterminé.

Le nombre de mots différents d'un texte est plus grand que deux :

```
context Texte inv:
self.occurrence->collect(mot)->asSet()->size() >= 2
```

Opérations propres à Set et Bag

- `intersection`

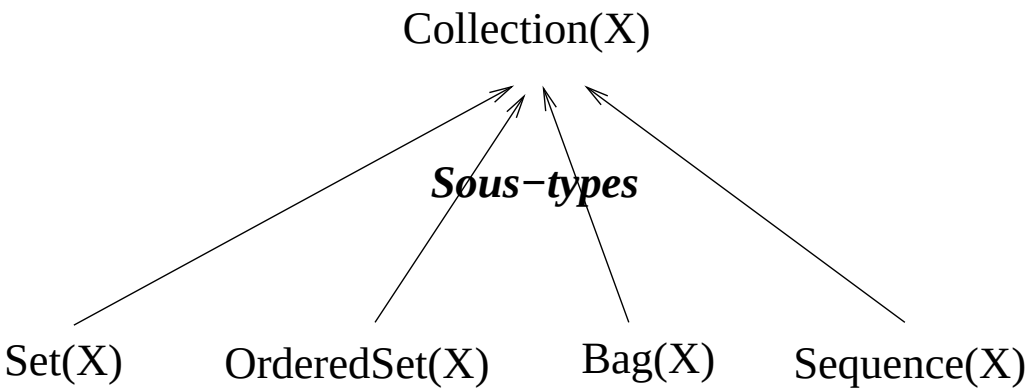
Opérations propres à `OrderedSet` et `Sequence`

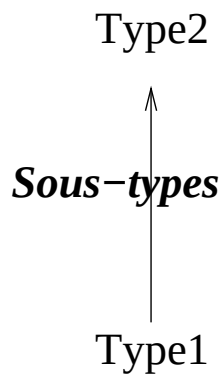
- ajout d'un objet à la fin
  - `append(unObjet : T): OrderedSet(T)` pour les ensembles ordonnés
  - `append(unObjet : T): Sequence(T)` pour les séquences
- ajout d'un objet au début
  - `prepend(unObjet : T): OrderedSet(T)` pour les ensembles ordonnés
  - `prepend(unObjet : T): Sequence(T)` pour les séquences
- objets d'un intervalle
  - `subOrderedSet(inf:Entier, sup:Entier): OrderedSet(T)` pour les ensembles ordonnés
  - `subSequence(inf:Entier, sup:Entier): Sequence(T)` pour les séquences

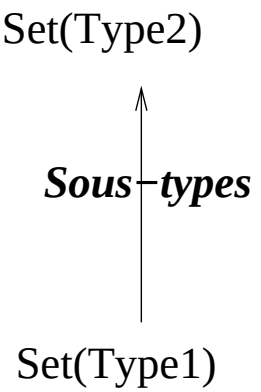
Opérations propres à `OrderedSet` et `Sequence`

- `at(inf:Entier): T`
- `first(): T`
- `last(): T`
- insertion d'un objet à un certain indice
  - `insertAt(i:Integer, o:T): OrderedSet(T)` pour les ensembles ordonnés
  - `insertAt(i:Integer, o:T): Sequence(T)` pour les séquences
- `indexOf(o:T): Integer`

Relation de « conformance de type »







Si la classe `Assiette` est sous-classe de `Vaisselle`, on peut en déduire :

- `Set(Assiette)` se conforme à `Collection(Assiette)`
- `Set(Assiette)` se conforme à `Set(Vaisselle)`
- `Set(Assiette)` se conforme à `Collection(Vaisselle)`

Relation de « conformance de type »

Les collections étant constantes, la conformance rejoint ici la substituabilité (sous-typage)
Prenons le cas d'un `Set(Assiette)`.

Considéré comme un `Set(Vaisselle)`, on peut lui appliquer l'opération
`including(object:Vaisselle):Set(Vaisselle)`
où `object` peut être tout à fait autre chose qu'une assiette.

Ce faisant on crée et retourne un `Set(Vaisselle)` (et non un `Set(Assiette)`), ce qui n'est pas problématique (il n'y a pas d'erreur de type, on n'a pas introduit un verre dans un ensemble d'assiettes).

Sommaire

Diagrammes d'états

Modélisation du comportement d'éléments UML
le plus souvent le comportement d'une instance d'une classe donnée

état

moment de la vie d'une instance déterminé par le fait qu'elle vérifie une condition particulière, est en train d'effectuer une opération ou attend un événement. Il est représenté en UML par une boîte aux angles arrondis.

transition

représente le passage d'un état à un autre. Ce passage est déclenché par un *événement*

Diagrammes d'états

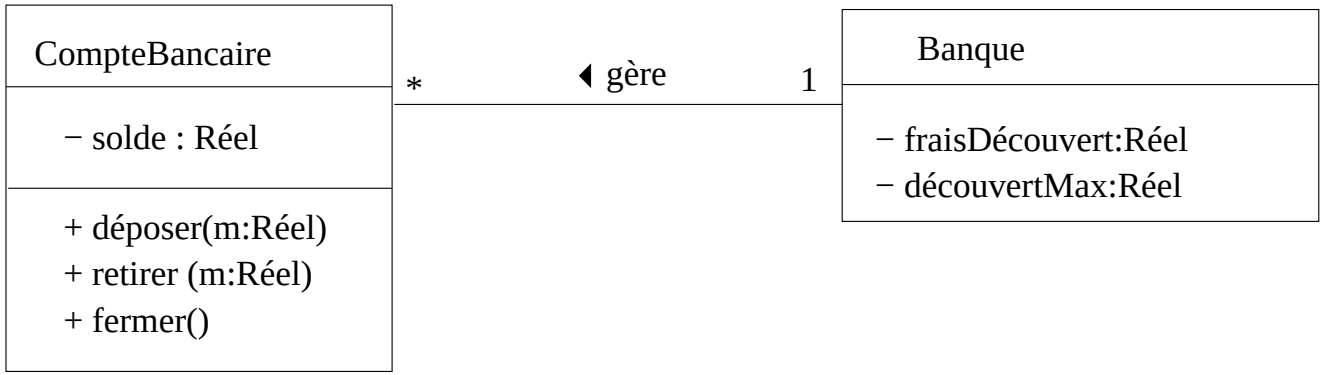
événement

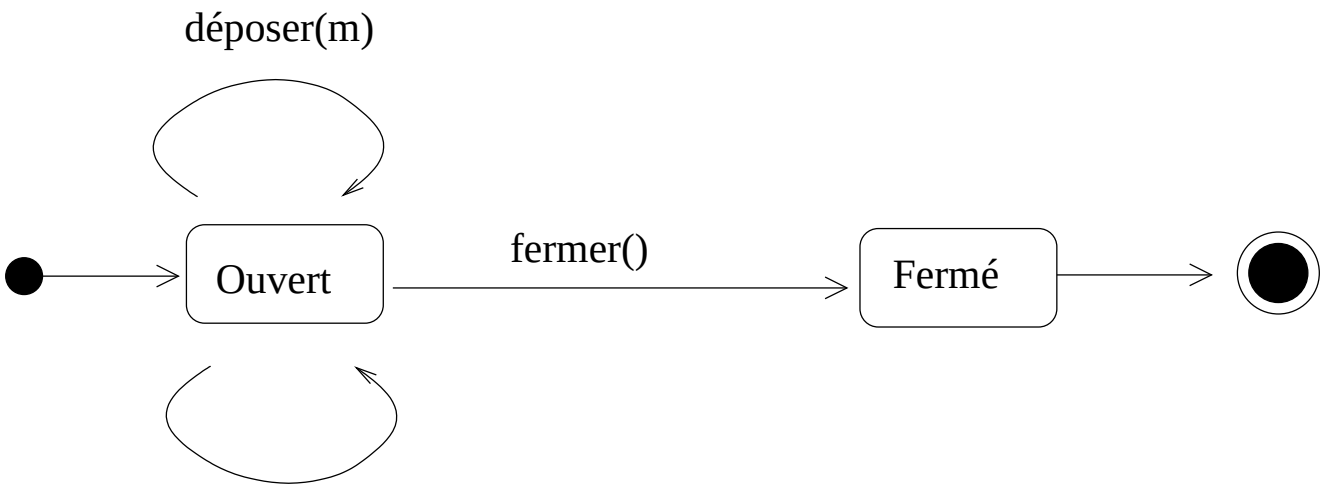
modification (une condition sur l'instance devient vraie), appel d'une opération, réception d'un signal asynchrone, fin d'une période de temps

garde

Condition associée à une transition, évaluée lorsqu'un événement survient

Diagramme de classes / diagramme d'états

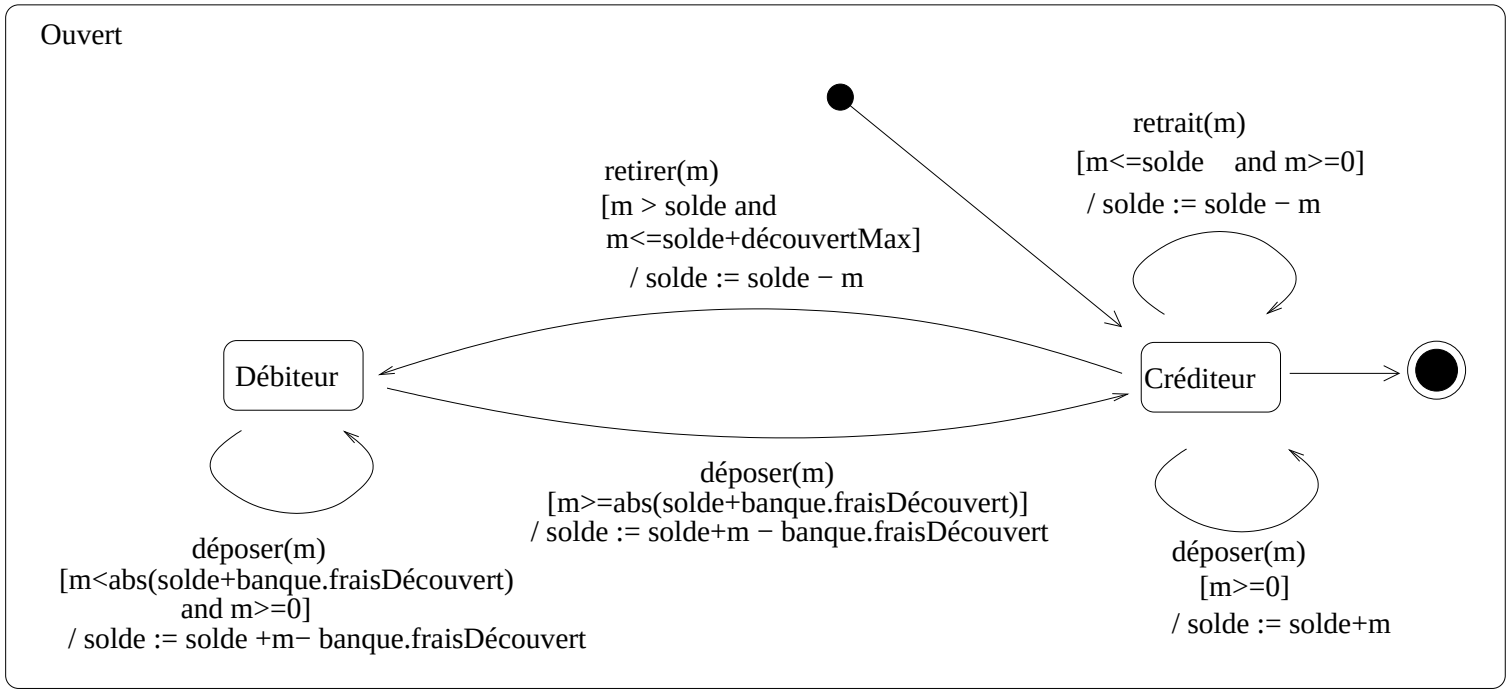




Diagrammes d'états / actions

Action

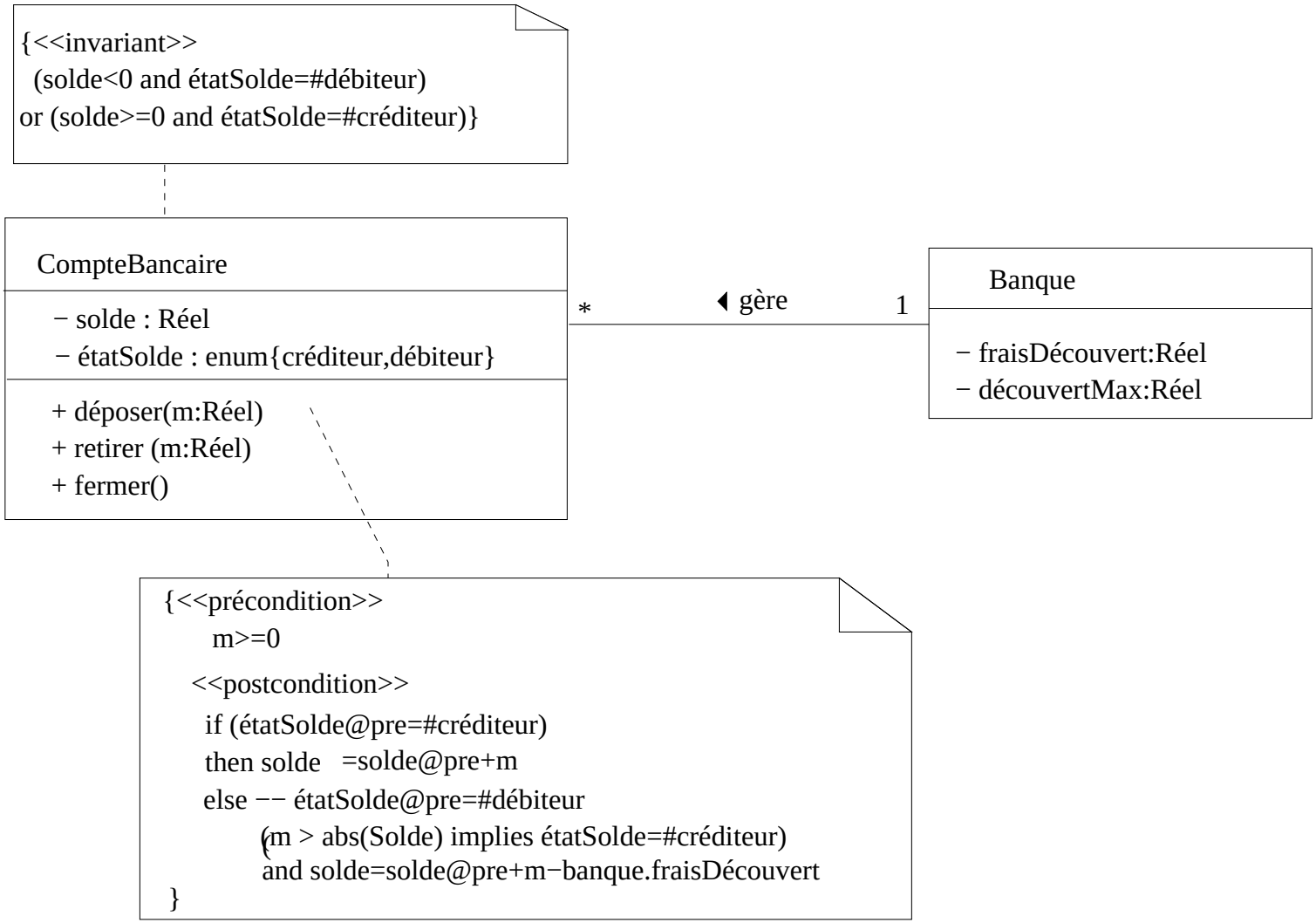
Etiquettent les transitions derrière le symbole /
OCL sert alors de langage de navigation avec le symbole :=
Contexte : celui de la classe associée au diag. d'états



Liaison diagramme de classes / diagramme d'états

- contraintes OCL dans des notes
- état = attribut `étatSolde` de `CompteBancaire`
- gardes du diagramme d'états = préconditions, conditions de la post-condition
- actions = associées à l'écriture de la post-condition

Liaison diagramme de classes / diagramme d'états

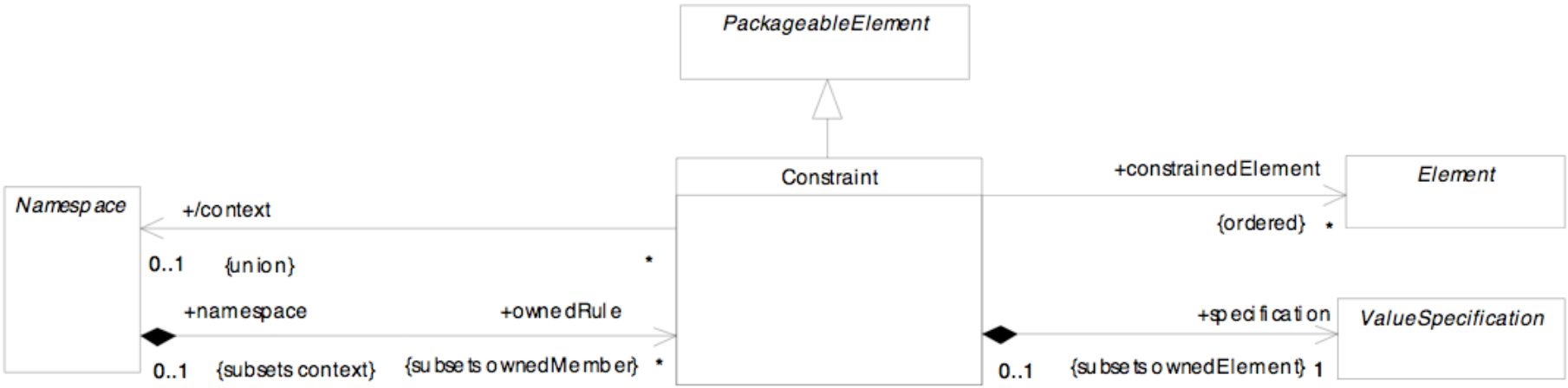


Sommaire

OCL dans le méta-modèle UML

- invariants sur les méta-classes
- extensions du méta-modèle à l'aide de stéréotypes
- les règles ne décrivent pas *toute* la sémantique UML. Une partie de cette sémantique reste écrite en langue naturelle, et une partie n'est pas décrite du tout
- En TD : de nombreux exemples

Contraintes dans le méta-modèle



Une contrainte ne peut pas s'appliquer à elle-même.

```
context Constraint inv:
not constrainedElement->include(self)
```

Ne peut s'exprimer en OCL :

- la spécification de valeur pour une contrainte est de type booléen
- l'évaluation d'une contrainte n'a pas d'effet de bord