U.M.L. 2
O.C.L. (Object Constraint Language) :
le langage de contraintes d'UML

Bernard ESPINASSE
Professeur à l'Université d'Aix-Marseille

2008

- **Besoin d'expression de contraintes en UML**
- **Introduction à OCL**
- **Spécification de contexte, invariants, pré et postconditions**
- **Opérations sur les types de base et les collections dans OCL**
- **Compléments divers**

Plan

1. Besoin d'expression de contraintes en UML
2. Introduction à OCL
3. Spécifications de base : contexte, invariants, conditions, pré et postconditions, accés et navigation, …
4. Opérations sur les types de base et les collections dans OCL
5. Compléments divers : variables, appels d'opérations, liens avec diagramme d'états, …

Sources du cours :

- *The Object Constraint Language Second Edition - Getting Your Models Ready for MDA de Jos Warmer et Anneke Kleppe, éditions Addison Wesley.*
- *la norme (pdf) : http://www.omg.org/cgi-bin/doc?formal/2006-05-01*
- *les cours de E. Cariou, J.M. Favre, S. Galland, Y. Lassus, M. Nebut, …*

1. Besoin d'expression de contraintes en UML

Besoin d'expression de contraintes en UML

Les différents diagrammes d'UML permettent d'exprimer certaines contraintes :

Graphiquement :

- contraintes structurelles (e.g. un attribut dans une classe)
- contraintes de types (e.g. sous-typage)
- contraintes diverses (e.g. composition, cardinalité, etc.)

Via des propriétés prédéfinies :

- sur des classes (e.g. {ordered})
- sur des rôles (e.g. {ordered})

=> C'est bien, mais loin d'être suffisant !

Besoin d'expression de contraintes en UML (1)

Soit une application banquaire portant sur des :

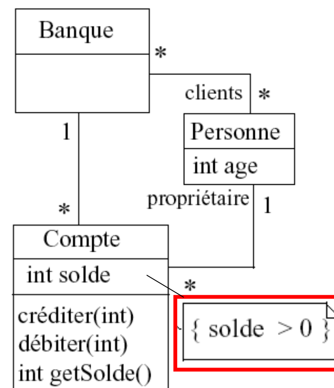
- comptes banquaires
- clients
- banques

Soit les contraintes suivantes à spécifier :

- 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

Besoin d'expression de contraintes en UML (2)

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

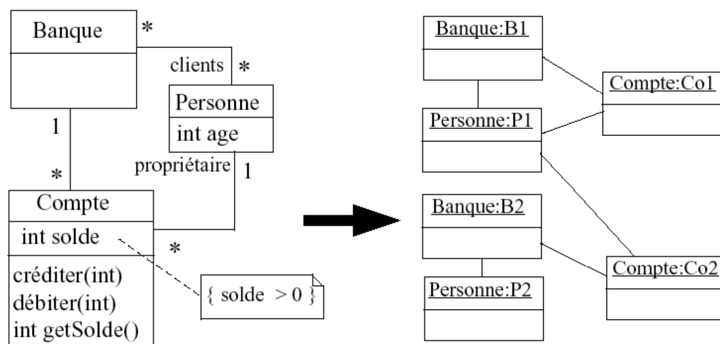
  => ajout d'une contrainte sur cet attribut

- ne permet pas de détailler toutes les contraintes sur les relations entre les classes ?

=> Intérêt des diagrammes d'instances ?

Besoin d'expression de contraintes en UML (3)

Diagramme de classes : **un diagramme d'instances valide :**



Ce diagramme d'instances bien que valide 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*

=> intérêt limité des diagrammes d'instances dans l'expression de contraintes

Besoin d'expression de contraintes en UML (4)

Ainsi pour spécifier complètement une application :

- Diagrammes UML seuls sont généralement insuffisants
- Nécessité de rajouter des contraintes exprimées par ailleurs

Comment exprimer ces contraintes ?

- **Langue naturelle** :
  - utilisation des notes en UML + texte libre mais manque de précision, compréhension
  - peut être ambiguë et imprécis, ne permet pas d'exprimer clairement des contraintes complexes
- **Langage formel avec sémantique précise** :

  => OCL (**Object Constraint Language)**

2. Introduction à OCL

OCL : Object Constraint Language

OCL (Object Constraint Language) est un langage :

- initialement développé par IBM, méthode Syntropy (1995)
- langage **formel** et **extensible** combiné **à UML** pour **l'expression de contraintes** orienté-objet (1997)
- défini de façon relativement **rigoureuse**, avec une **syntaxe précise** et **textuelle** (sans symbole étranges) et **simple à mettre en œuvre**
- permet principalement d'exprimer **2 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 :

(une expression OCL décrit une contrainte à respecter et pas le « code » d'une méthode)

Normalisation de OCL :

- **OCL version 1** : fait partie de la **norme UML 1.X de l'OMG** (Object Management Group)
- **OCL version 2.0** : spécification conforme à **UML 2.0**, mais norme à part entière de **l'OMG** pouvant s'appliquer sur d'autres modèles que ceux de UML

Caractéristiques, forces et faiblesses de OCL

- **Caractéristiques de OCL :**
  - langage d'expression **sous ensemble des langages fonctionnels**
  - basé sur la **théorie des ensembles** et la **logique des prédicats**,
  - repose sur le concept de **valeur** et **d'expression**
  - **sans effet de bord** et **fortement typé**
  - permet de **spécifier**, pas de programmer
  - **extensible**
- **Forces :**
  - d'être parfaitement **adapté et intégré à UML**
  - notation dite **intuitive** et **proche des langages de programmation**, moins rebutante qu'une notation mathématique
- **Faiblesses :**
  - pas toujours défini très proprement
  - pas vraiment lisible pour des contraintes complexes
  - pas aussi rigoureux qu'un langage de spécification comme Z ou B => pas de preuves possibles
  - puissance d'expression limitée

Utilisation de OCL dans les diagrammes UML

OCL permet de **spécifier des contraintes** ou **des expressions** associés à un **diagramme UML**

Contraintes pour :

- **contraindre** les **diagrammes UML**
- **définir** des **invariants** de classe, **pré-condition** de méthode, …
- **vérifier** potentiellement ces **contraintes** à **l'exécution**

Expressions pour :

- **indiquer** des **valeurs** ou des collections de valeurs
- **décrire** des **post-conditions**
- **spécifier des opérations** : par exemple de type ***query*** (OCL se rapproche alors de SQL : requètes portant sur un diagramme UML)
- **générer éventuellement du code** correspondant à l'évaluation d'une expression

Utilisation de OCL dans les diagrammes UML

Par la suite on parlera d'expressions au sens large.

Une expression OCL est toujours associée à un élément d'un diagramme : c'est le **contexte de la contrainte (*context*)**

Une expression OCL permet de spécifier :

- des **Invariants** sur des **classes** (*inv*)
- des **Pré-conditions** sur des **opérations** (*pre*)
- des **Post-conditions** sur des **opérations** (*post*)
- des **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**
- ...

Types et syntaxe générale des expressions OCL

Différents types :

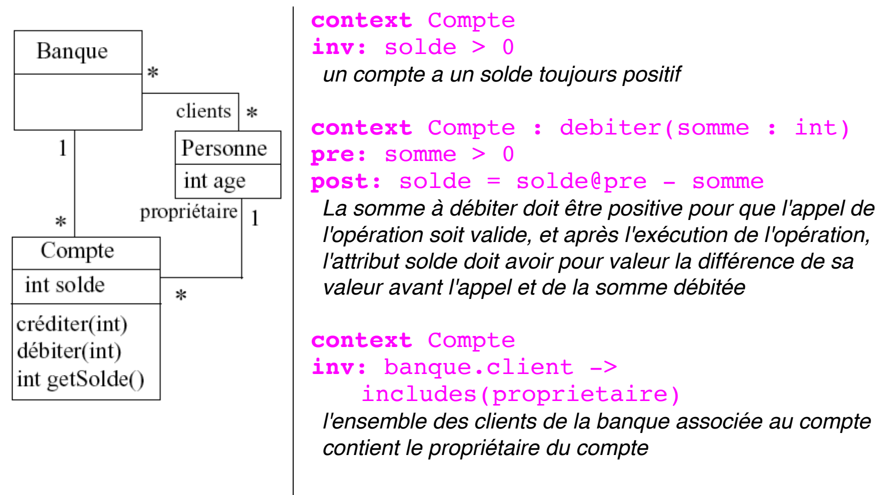
- **Types provenant d'UML** : `classes, association, ...`
- **Types de base** : `integer, real, boolean, string`
- **Types énumérés** : `{masculin, féminin}`
- **Types construits ou Collection**: `set(T), sequence(T), bag(T)`
- **Méta-types** : `OclType, OclAny, OclState, OclExpression`

Syntaxe générale des expressions OCL :

- `Constantes`
- `Identificateurs`
- **`self`**
- `expr op expr`
- `exprobjet . propobjet ( parametres )`
- `exprcollection -> propcollection ( parametres )`
- `package::package::element`
- **`if`** `cond` **`then`** `expr` **`else`** `expr` **`endif`**
- **`let`** `var : type` **`in`** `expr, …`

Exemple d'usage d'OCL avec UML

Diagramme de classe (application banquaire) :




```
context Compte
inv: solde > 0
```
un compte a un solde toujours positif

```
context Compte : debiter(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, et 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 débitée

```
context Compte
inv: banque.client ->
    includes(proprietaire)
```
l'ensemble des clients de la banque associée au compte contient le propriétaire du compte

OCL permet ainsi de préciser clairement de la sémantique sur les modèles UML

3. Spécifications de base :

- contexte
- invariants
- conditions et dépendances
- pré et postconditions
- accés et navigation, ...

Contexte d'une contrainte OCL

Contexte de la contrainte en OCL :

- précise **l'élément du modèle** auquel est associé la contrainte (classe, interface, type de données, composant), opération, attribut, …)
- mot-clé : **context**
- l'expression OCL est évaluée pour une instance particulière, appelée **instance contextuelle** (le type de cette instance : **type contextuel)**
- le mot-clé **self** réfère explicitement cette **instance** (il est souvent être omis)

```
context Compte inv : solde>0
```

L'expression OCL s'applique à la classe Compte, c.a.d. à toutes les instances de cette classe

```
context Personne inv : age>10
```

L'expression OCL s'applique à la classe Personne, c.a.d. à toutes les instances de cette classe

Association graphique d'une contrainte à son contexte :

- par une note UML reliée par un trait pointillé au contexte (cf exemple précédent)
- permet de visualiser directement le contexte (éviter surcharge du schéma)

Spécification d'Invariants : inv

Invariant :

- exprime une contrainte, un prédicat, associé à un objet (une classe, une association, …) ou groupe d'objets **devant être respectée à tout instant**
- mot-clé : **inv**
- son contexte est un **objet** pouvant être **nommé explicitement** ou **référencé** par **self**
- peut être **nommé**

```
context Compte
inv: solde > 0
```

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

```
context p : Personne inv ageOk : p.age < 110
```

L'invariant peut être nommé

```
context Personne
inv : age > 0 and self.age < 110
```

Pour toutes les instances de la classe Personne, l'attribut age doit être compris entre 0 et 110

```
context p : Personne
inv mariageLégal : p.marié implies p.age > 16
```

Pour toutes les instances de la classe Personne, un mariage légal nécessite un âge supérieur à 16 ans

Nommage des contraintes et commentaires

Nommage des contraintes :

```
context Compte
inv soldePositif: solde > 0

context Compte::debiter(somme : int)
pre sommePositive: somme > 0
post sommeDebitee: solde = solde@pre - somme
```

Commentaire en OCL : utilisation de --

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


Conditions dans les expressions OCL

2 formes pour exprimer des dépendances de contraintes :

1) `if expr1 then expr2 else expr3 endif`
   si l'expression expr1 est vraie alors expr2 doit être vraie
   sinon expr3 doit être vraie

2) `expr1 implies expr2`
   si l'expression expr1 est vraie, alors expr2 doit être vraie également
   si expr1 est fausse, alors l'expression complète est vraie

```
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 banquaire 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 banquaire, alors elle est cliente d'au moins une banque

Préconditions et Postconditions

Prédicats associés à une opération :

- **précondition** : spécifie l'état qui doit être respecté **avant** l' appel de l'opération
- **postcondition** : specifie l'état qui doit être respecté **après** l'appel de l'opération
- mots-cle : **pre :** et **post :**
- **self** désigne l'objet sur lequel l'opération à lieu

Syntaxe :

```
Context ma_classe :: mon_operation(liste_parametres) :
  type_retour
context Type::opération( param1 : Type1, …) : Type
pre nom1 : param1 < …
pre nom2 : …
post nom2 : … mon_attribut@pre … result > …
```

Dans la postcondition :

- l'attribut `result` : référence la valeur retournée par l'opération
- `mon_attribut@pre` : référence la valeur de « mon_attribut » avant l'appel de l'opération

Préconditions & Postconditions : exemples

```
context Personne::anniversaireArrive()
post : age = age@pre + 1
```

*Après l'exécution de l'opération, l'attribut age **doit** avoir prend la valeur antérieure +1*

```
context Compte::debiter(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
```

*Après l'exécution de l'opération, l'attribut result **doit** avoir pour valeur la valeur de solde*

Remarque : on ne décrit pas comment l'opération est réalisée mais des contraintes sur l'état avant et après son exécution

Accès aux propriétés dans les expressions

En OCL une propriété est un élément pouvant être : un attribut, un bout d'association, une opération ou une méthode de type requête

Accès à une propriété d'un objet : « . » *notation pointée*

```
context Compte inv: self.solde > 0
```

Accès à un attribut de la classe Compte :

```
context Compte inv: self.getSolde() > 0
```

Accès à une opération de la classe Compte :

Accés à une propriété d'une <u>collection</u> d'objets : « -> »

```
compte -> collect (solde)
```

Accès à un attribut sur un ensemble :

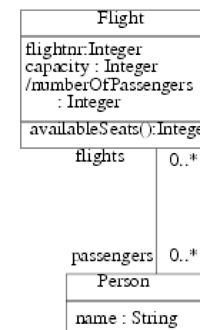
- retourne l'ensemble des soldes de tous les comptes
- Forme raccourcie et simplifiée de : `compte.solde`

Des règles permettent de mixer collection et objets

Accès aux propriétés des expressions OCL

Spécification d'une propriété: « :: » *notation 4 points*

- Attribut : `context TypeName::AttributeName : TypeAttribute`
- Opération :
  - `context Typename::operationName(param1 : Type1, ... )`
  - `context Typename::operationName(param1 : Type1, ... ):ReturnType`
- Extrémité d'association : `context Typename::operationName : Type`



Exemples dans le contexte de la classe Flight :

```
context Flight::capacity : Integer
```

accés à la propriété <u>capacity</u> de Flight

```
context Flight::availableSeats() : Integer
```

accés à l'opération <u>availableSeats</u>() de Flight

```
context Flight::passengers : Set(Person)
```

accés à l'extrémité d'association <u>passagers</u>

Accès aux objets, navigation

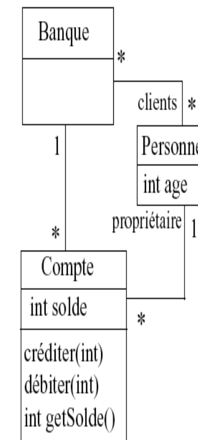
Dans une contrainte OCL associée à un objet, on peut :

- **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 lesquels 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 coté de cette classe
  - Si cardinalité de 1 pour une association : **référence un objet**
  - Si cardinalité > 1 pour une association : **référence une collection d'objets**

Accès aux objets, navigation : exemples



Exemples, dans le 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
- `proprietaire` : 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: proprietaire.age >= 18
```


4. Opérations sur les types de base et les collections

Opérations sur les types de base

Opérations sur Integer :
`+, -, *, div, mod, abs, max, min`

Opérations sur Real :
`+, -, *, /, floor, round, max, min`

Opérations sur String :
`s.size(),s1.concat(s2), s1.substring(i1,i2),s1.toUpper, s1.toLower()`

Opérations sur Boolean :
`not, and, or, xor, implies, if-then-else –endif`

L 'évaluation des opérateurs `or, and, if` est partielle :

- **true or x** est toujours **vrai**, même si x est indéfini
- **false and x** est toujours **faux**, même si x est indéfini

```
(age<40 implies salaire>10000) and (age>=40 implies salaire>20000)
if age<40 then salaire > 10000 else salaire > 20000 endif
salaire > (if age<40 then 10000 else 20000 endif)
```

Types énumérés : `enum { masculin, feminin }`

Utilisation d 'une valeur d 'un type énuméré : `#masculin`

Opérations sur type énuméré : `– =, <>`

Collections dans OCL

OCL permet de gérer 3 types de collections :

- **ensembles : Set(T)**
  - pas de répétition, pas d 'ordre
  - `Set { 1, 5, 10, 3 }`
  - très utilisés dans UML : extension d'une classe, extension ou extrémité d'une association, ...
- **listes : Sequence(T)**
  - répétitions possibles, les éléments sont ordonnés
  - `Sequence { 1, 5, 5, 10, 3, 1 }`
  - peu utilisés dans UML : le résultat d 'une association avec la contrainte {ordered}
- **sacs : Bag(T)**
  - répétitions possibles, pas d 'ordre
  - `Bag { 1, 5, 5, 10, 3, 1 }`
  - sont le résultat de navigations en OCL : le résultat de l 'opération `collect` est un sac, utile pour des opérations comme `sum`
    `Ex : employes->collect(salaire).sum`

Remarque :

- **pas de collections de collections dans OCL 1, mise à plat automatique** :
  `Set{ Set{1,2}, Set{3,1} } = Set{1,2,3}`
- **en OCL 2.0, possibilité de collections de collections et de tuples**

Opérations générales sur les collections

- **Opérations traditionnelles ensemblistes :**
  - `union` : retourne l'union de 2 collections
  - `intersection` : retourne l'intersection de 2 collections
  - `size(coll)` : 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)` : l'ensemble référencé doit être cet collection en incluant l'objet obj
  - `excluding(obj)` : idem mais en excluant l'objet obj
  - `includesAll(coll)` : l'ensemble contient tous les éléments de la collection coll
  - `excludesAll(coll)` : l'ensemble ne contient aucun des éléments de la collection coll

Opérations générales sur les collections

- **Opération de filtrage :** sélection et élimination selon un prédicat **:** `select, filter`
- **Opération d'image d'une fonction (d'une expression) :** `collect`
- **Quantificateurs :** `forall, exists`
- **Itérateur général :** `iterate`
- **Cardinalité :** `coll -> size`
- **Nombre d 'occurrences :** `coll -> count(elem)`
- **Vide :** `coll -> isEmpty`
- **Non vide :** `-> nonEmpty`
- **Somme des éléments :** `coll -> sum`

Opérations spécifiques aux ensembles

- **Union :** `ens -> union(ens)`

  Ex : `ens1 = ens2 -> union(ens3)`

  L'ensemble ens1 doit être l'union des éléments de ens2 et de ens3

- **Intersection :** `ens -> intersection(ens)`

  Ex : `(ens1 -> intersection(ens2)) -> isEmpty()`

  Les ensembles ens1 et ens2 n'ont pas d'élément en commun

- **Différence :** `ens1 - ens2`
- **Ajout d'un élément :** `ens -> including(elem)`
- **Suppression d'un élément :** `ens -> excluding(elem)`
- **Conversion Set vers Liste :** `ens -> asSequence`
- **Conversion Set vers Sac :** `ens -> asBag`

Opérations sur les éléments d'une collection

Primitives d'OCL permettant de :

- **vérifier** des **contraintes** sur **chaque élément** d'une **collection**
- **définir** une **sous-collection** d'une **collection** en fonction de certaines **contraintes**

Syntaxe générale :

1) ensemble -> primitive(expression)

- s'applique aux éléments de l'ens. et pour chacun d'entre eux, l'expression est vérifiée. On accède aux attributs/relations d'un élément directement :

  compte -> reject(solde > 1000)

  Retourne une collection contenant tous les comptes banquaires dont le solde n'est pas supérieur à 1000 €

2) ensemble -> primitive(elt | expression)

- nomme l'attribut courant (elt) mais sans préciser son type :

  compte -> collect(c | c.solde)

  Retourne une collection contenant l'ensemble des soldes de tous les comptes

3) ensemble -> primitive(elt : type | expression)

- fait explicitement apparaître le type des éléments de l'ensemble (ici type). On accède aux attributs/relations de l'élément courant en utilisant **elt** (réf. sur l'élément courant) :

  compte -> collect(c : Compte | c.solde)

  Retourne une collection contenant l'ensemble des soldes de tous les comptes

Opérations sur les éléments d'une collection

Primitives de filtrage d'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**

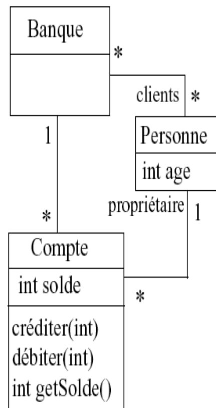
Primitives de navigation dans une collection col:

- **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.

Quantificateur universel et existentiel :

- coll -> **forAll**(cond): **retourne vrai si tous les éléments de** col **respectent la contrainte** cond spécifiée (pouvant impliquer plusieurs éléments de la collection) sinon faux
- coll -> **exists**(cond): **retourne vrai si au moins un élément de** col **respecte la contrainte** cond spécifiée et faux sinon

Exemples d'opérations sur les éléments d'une collection (1)



Exemples, dans le contexte de la classe Compte :

proprietaire -> **notEmpty()**

il y a au moins un objet Personne associé à un compte

proprietaire -> **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(proprietaire)**

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

Exemples d'opérations sur les éléments d'une collection (2)

Select et Reject :

compte -> **select**(c | c.solde > 1000)

*Retourne une collection contenant tous les **comptes banquaires dont le solde est supérieur à 1000 €***

compte -> **reject**(solde > 1000)

*Retourne une collection contenant tous **les comptes banquaires dont le solde n'est pas supérieur à 1000 €***

(compte -> **select**(solde > 1000)) -> **collect**(c | c.solde)

*Retourne une collection contenant tous les **soldes des comptes dont le solde est supérieur à 1000 €***

self.enfants ->**select**(age>10 and sexe = #masculin)

Retourne une collection contenant les enfants de moins de 10 ans et de sexe masculin

self.employé->**select**(age > 50)

self.employé->**select**(p | p.age>50)

Retourne une collection contenant les employés dont l'âge est supérieur à 50 ans, sans et avec renommage de variable et explicitation de son type

Exemples d'opérations sur les éléments d'une collection (3)

Collect = primitive de navigation dans une collection col :

- correspond à **l'image d'une fonction** (map, apply, ...)
- l'expression **est évaluée pour chaque élément**
- le **résultat est le Sac des résultats**, pas un ensemble !

```
compte -> collect( c : Compte | c.solde)
```

*Retourne une collection contenant l'ensemble des **soldes de tous les comptes***

```
self.enfants->collect(age)
```

Retourne une collection (un sac) contenant les âges des enfants, exemple Bag {10,5,10,7}

Simplification :

```
self.enfants->collect(age) <=> self.enfants.age
```

Rappel : le résultat est un sac (Bag) si l'on souhaite obtenir un ensemble :

```
self.enfants.age->asSet
```

Permet de naviguer :

```
self.enfants.voitures
```


Exemples d'opérations sur les éléments d'une collection (4)

Exist et AllInstance :

```
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*

```
self.clients->forall(age<18)
```

Retourne vrai si tous les clients ont un âge > 18 ans

```
self.clients->exists(sexe=#feminin)
```

Retourne vrai s'il existe un client de sexe féminin

```
context Personne
inv: Personne.allInstances() -> forAll(p1, p2 |
   p1 <> p2 implies p1.nom <> p2.nom)
```

- *Il n'existe pas 2 instances de la classe Personne pour lesquelles l'attribut nom a la même valeur : 2 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*

Exemples d'opérations sur les éléments d'une collection (5)

Exist et AllInstance :

Possible de nommer la variable :

```
self.enfants->forall( e | e.age < self.age - 14)
```

Retourne vrai si tous les enfants ont un âge > 14 ans

Possible d'expliciter son type :

```
self.enfants->forall( e : Personne | e.age < self.age - 14)
```

Retourne vrai si tous les enfants ont un âge > 14 ans

Possible de parcourir plusieurs variables à la fois :

```
self.enfants->forall( e1,e2 : Personne | e1 <> e2 implies
e1.prénom <> e2.prénom)
```

Retourne vrai si tous les enfants n'ont pas le même prénom

6. Compléments divers

- **variables,**
- **appels d'opérations,**
- **lien avec diagramme d'états, ...**

Définition de variables dans OCL

- **Pour faciliter l'utilisation de certains attributs ou calculs de valeurs :**
- **Dans une contrainte OCL :** `let ... in ...`

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

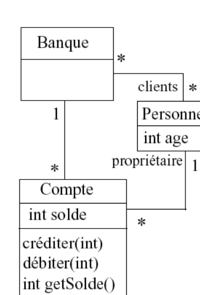
  - une personne majeure doit avoir de l'argent
  - sum() : fait la somme de tous les objets de l'ensemble
- **Pour l'utiliser partout :** `def`

```
context Personne
def: argent : int = compte.solde -> sum()

context Personne
inv: age >= 18 implies argent > 0
```


Appels d'opération de classe

- **Accès dans une contrainte OCL aux attributs, objets ... « en lecture » du diagramme de classe**
- **Possibilité d'appeler une opération d'une classe dans une contrainte :**



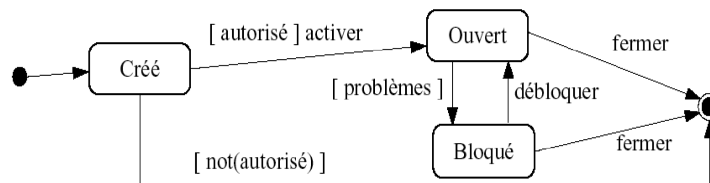

```
context Banque
inv: compte -> forAll( c |
 c.getSolde() > 0)
```

  - `getSolde()` est une opération de la classe Compte.
  - Elle calcule une valeur mais sans modifier l'état d'un compte

Remarque : 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

Liens avec Diagramme d'états

- **Soit un diagramme d'états associé à un objet d'un diagramme de classe :** ici état de l'objet Compte :



- **Possibilité de référencer un état de ce diagramme d'états dans une contrainte OCL :**

  `oclInState(etat)` : vrai si l'objet est dans l'état `etat`.
- **Pour sous-états :** `etat1::etat2` **si** `etat2` **est un état interne de** `etat1`
- **Ex 1 :** état de l'objet Compte :

```
context Compte :: debiter(somme : int)
pre: somme > 0 and self.oclInState(Ouvert)
```

L'opération debiter ne peut être appelée que si le compte est dans l'état ouvert

Liens avec Diagramme d'états (suite)

- **Ex 2 :** On ne peut pas avoir plus de 5 comptes ouverts dans une même banque **:**

```
context Compte :: activer()
pre: self.oclInState(Cree) and
    proprietaire.compte -> select( c |
    self.banque = c.banque) -> size() < 5
post: self.oclInState(Utilisable)
```

- **On peut aussi exprimer la garde** `[ autorise ]` **en OCL :**

```
context Compte
def: autorise : Boolean =
     proprietaire.compte -> select( c |
     self.banque = c.banque) -> size() < 5
```

Propriétés définies en OCL

- **Pour objets :**

  `oclIsTypeOf(type)` : l'objet est du type `type`

  `oclIsKindOf(type)` : l'objet est du type `type` ou un de ses sous-types

  `oclInState(etat)` : l'objet est dans l'état `etat`

  `oclIsNew()` : l'objet est créé pendant l'opération

  `oclAsType(type)` : l'objet est « casté » en type `type`

- **Pour ensembles :**

  `isEmpty(), notEmpty(), size(), sum()`

  `includes(), excludes(), includingAll() ...`

Règles de précédence en OCL

- **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**