Conception par contrats avec UML

OCL – Object Constraint Language

Gerson Sunyé
gerson.sunye@univ-nantes.fr

Université de Nantes

21 mars 2014

Plan

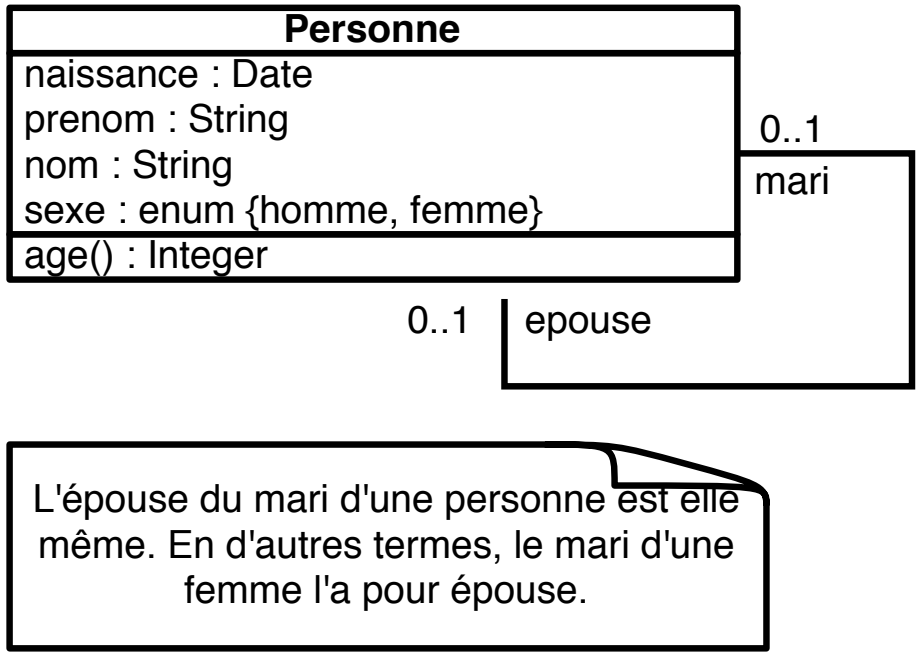
Plan

OCL – Object Constraint Language

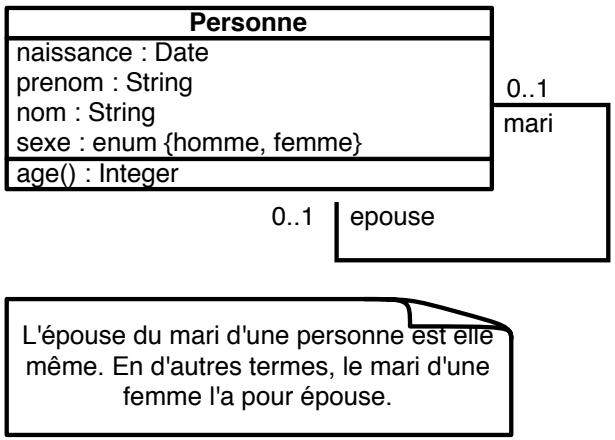
- Langage de description de contraintes de UML.
- Permet de restreindre une ou plusieurs valeurs d'un ou de partie d'un modèle.
- Utilisé dans les modèles UML ainsi que dans son méta-modèle (grâce aux stéréotypes).
- Formel, non ambigu, mais facile à utiliser (même par les non mathématiciens).
- Actuellement, dans la version 2.3.1.

Motivation

Peut-on rendre plus précis un diagramme UML?



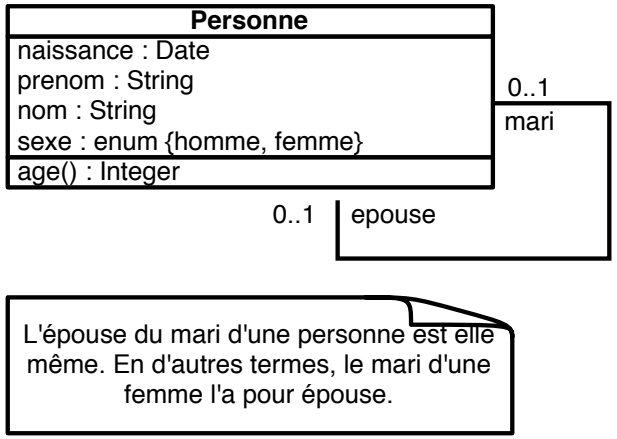
Motivation



- Les diagrammes UML manquent parfois de précision.
- Le langage naturel est souvent ambigu.

Conception par contrats avec UML

- Ajout de *contraintes* à des éléments de modélisation.



context Personne
inv: **self**.epouse→**notEmpty**() **implies self**.epouse.mari = **self and**
self.mari→**notEmpty**() **implies self**.mari.epouse = **self**

Plan

Principes du langage

Une expression OCL est une expression sans effet de bord, portant sur les éléments suivants :

1. Types de base :**Integer**, **Real**, **Boolean**, **String**, **UnlimitedNatural** ;
2. Collections d'éléments ;
3. Éléments du contexte (le modèle UML associé).

Appel d'opérations

L'appel d'une opération sur un élément se fait grâce à la notation pointée:

$\texttt{'Nantes'}.\textbf{substring}(1,3) = \texttt{'Nan'}$

L'appel d'une opération sur une collection d'éléments se fait grâce à une flèche ($\rightarrow$):

$\{1,\ 2,\ 3,\ 4,\ 5\}\rightarrow\textbf{size}() = 5$

Types de base

Type	Valeurs
Boolean	true, false
Integer	1, -5, 2, 34, 26524, ...
Real	1.5, 3.14, ...
String	'To be or not to be...'
UnlimitedNatural	0, 1, 2, 42, ..., *

Opérations sur les types de base

Type	Operations
Integer	=, *, +, -, /, abs(), div(), mod(), max(), min()
Real	=, *, +, -, /, abs(), floor(), round(), max(), min(), >, <, <=, >=, ...
String	=, size(), concat(), substring(), toInteger(), toReal(), toUpper(), toLower()
Boolean	or, xor, and, not, implies
UnlimitedNatural	*,+,/

Opérations simples sur les collections

- **isEmpty**() : vrai si la collection est vide.
- **notEmpty**() : vrai si la collection contient au moins un élément.
- **size**() : nombre d'éléments dans la collection.
- **count**(elem) : nombre d'occurrences de *elem* dans la collection.

Exemples:

$\{\}\rightarrow$**isEmpty**() $--$ *Vrai*
$\{1\}\rightarrow$**notEmpty**() $--$ *Vrai*
$\{1,2,3,4,5\}\rightarrow$**size**() $= 5$
$\{1,2,3,4,5\}\rightarrow$**count**(2) $= 1$

Opérations complexes sur les collections

Certaines opérations utilisent un itérateur (nommé each par convention), c'est à dire, une variable qui sera évaluée à chaque élément de la collection.

- **select**(expression −booleenne) : Sélectionne (filtre) un sous-ensemble de la collection.
- **collect**(expression) : Évalue une expression pour chaque élément de la collection.

Exemples:

$\{1,2,3,4,5\} \rightarrow$**select**(each | each > 3) $= \{4,5\}$

$\{$'a','bb','ccc','dd'$\} \rightarrow$**collect**(each | each.**toUpper**()) $=$ $\{$'A','BB', 'CCC','DD'$\}$

Vérification de propriétés sur des collections

- **forAll**(expression −booleene) : Vérifie que **tous** les éléments de la collection respectent l'expression.
- **exists**(expression −booleene) : Vérifie qu'**au moins un** élément de la collection respecte l'expression.

Exemples:

$\{1,2,3,4,5\} \rightarrow$ **forAll**(each | each $>$ 0 **and** each $<$ 10) −− *Vrai*
$\{1,2,3,4,5\} \rightarrow$ **exists**(each | each $=$ 3) −− *Vrai*

Plan

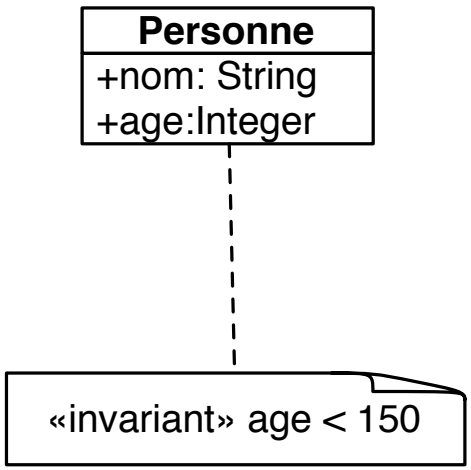
OCL et UML

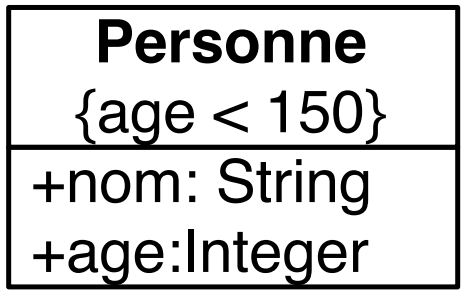
Une expression OCL peut aussi porter sur les éléments UML suivants:

- Classificateurs (classes, interfaces, composants, cas d'utilisation, etc.) et leurs propriétés : opérations sans effet de bord, attributs et rôles d'associations ;
- États (des machines d'états associées).

Notation graphique

Directement à l'intérieur d'un modèle ou dans un document séparé:





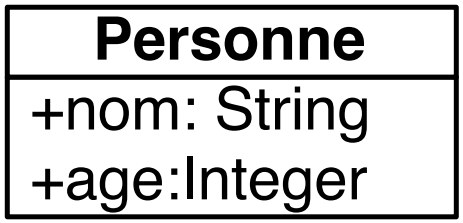
context Personne **inv**: **self**.age $<$ 150
context Personne **inv**: age $<$ 150

Notion de contexte

- Toute expression OCL est liée à un contexte spécifique, l'élément auquel l'expression est attachée.
- Le contexte peut être utilisé à l'intérieur d'une expression grâce au mot-clef "self".
  - Implicite dans toute expression OCL.
  - Similaire à celui de Smalltalk ou Python, au "this" de C++ et Java, ou au "Current" de Eiffel.

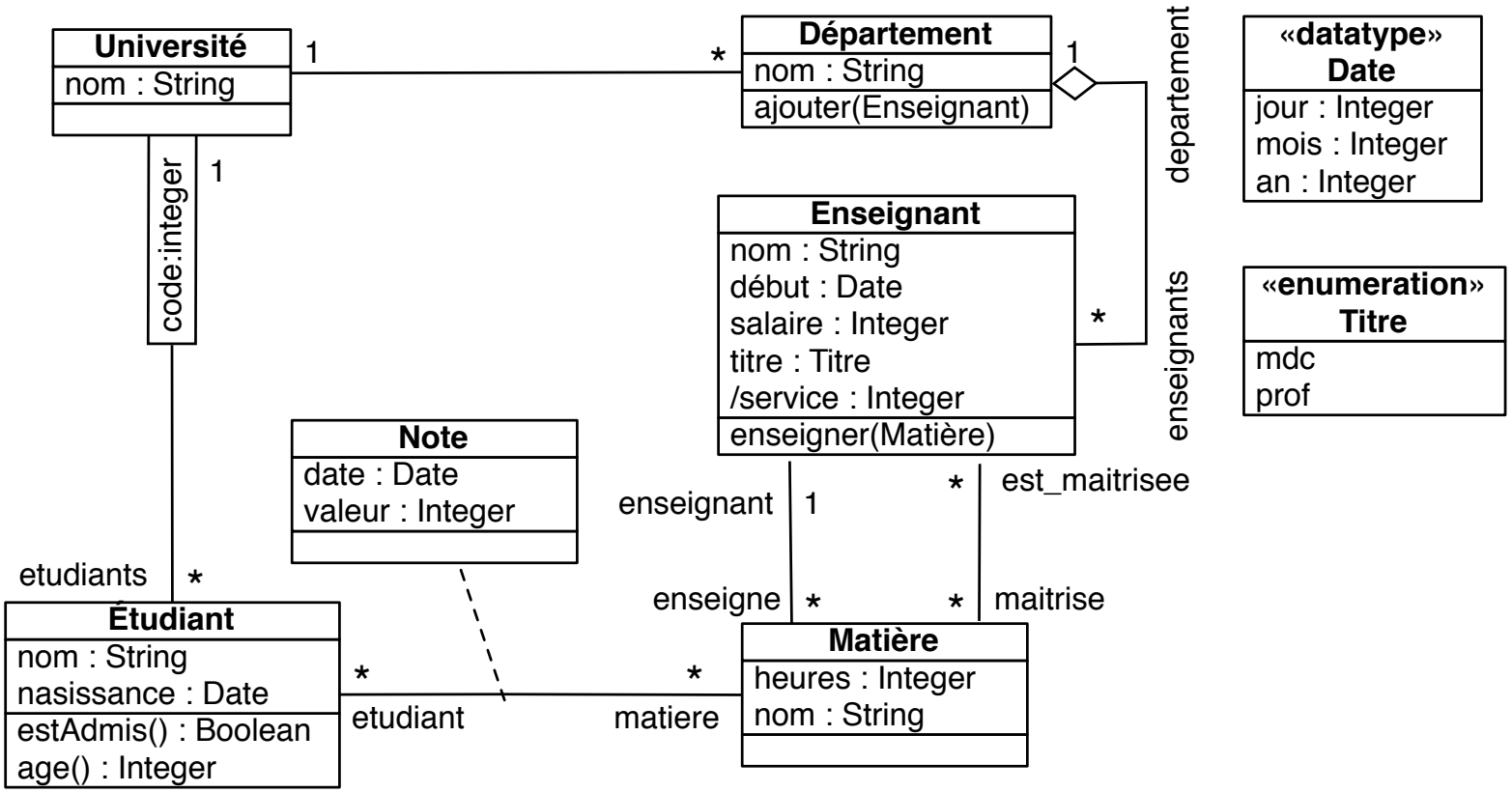
Propriétés du contexte

Le contexte permet l'accès aux propriétés de l'élément UML attaché, à l'intérieur d'une expression.



context Personne
−− *(...)*
self .nom.**size**() $>$ 1 **and**
self .age $\geq$ 0 **and**
self .age $\leq$ 150

Exemple de modèle



Propriétés: Attributs

On utilise la notation pointée:

- Attributs d'instance:

context Enseignant **inv**:
self . salaire $<$ 10

- Attributs de classe:

context Enseignant **inv**:
self . salaire $<$ Enseignant::salaireMaximum

Propriétés: Types énumérés

- Définition: enum{value1, value2, value3}
- Pour éviter les conflits de nom, on utilise le nom de l'énumération: Enum::val1

context Enseignant **inv**:
self . titre = Titre :: prof **implies**
self . salaire > 10

Propriétés: Opérations de *Query*

Notation pointée:

- Opérations d'instance:

context Etudiant **inv**:
self.age() > 16

- Opérations de classe:

context Etudiant **inv**:
self.age() > Etudiant::ageMinimum()

Propriétés: États

- Accessibles avec *oclInState()*:

context Departement::ajouter(e:Enseignant)
pre: e.**oclInState**(disponible)

pre:e.**oclInState**(indisponible :: en_vacances)
-- *etats imbriques*

Plan

Différentes expressions OCL

OCL peut spécifier :

1. Des invariants de classe ;
2. Les pré- et post-conditions d'une opération (ou d'une transition) ;
3. Le corps d'une opération.
4. Des nouvelles propriétés ;
5. L'initialisation d'attributs ;
6. Des propriétés dérivées.

Invariants de classe

- Dans un état **stable**, toute instance d'une classe doit vérifier les invariants de cette classe.
- Exemples:

context e : Etudiant **inv**: ageMinimum: e.age $>$ 16
context e : Etudiant **inv**: e.age $>$ 16
context Etudiant **inv**: **self**.age $>$ 16
context Etudiant **inv**: age $>$ 16

Spécification d'opérations

- Inspirée des types abstraits : une opération est composée d'une signature, de pré-conditions et de post-conditions.
- Permet de contraindre l'ensemble de valeurs d'entrée d'une opération.
- Permet de spécifier la sémantique d'une opération : ce qu'elle fait et non comment elle le fait.

Pré-condition

- Ce qui doit être respecté par le client (l'appelant de l'opération)
- Représentée par une expression OCL stéréotypée ≪ precondition ≫

context Departement::ajouter(e : Enseignant) : **Integer**
pre nonNul: **not** e.**oclIsUndefined**()

Post-condition

- Spécifie ce qui devra être vérifié après l'exécution d'une opération.
- Représentée par une expression OCL stéréotypée ≪ postcondition ≫:
- Opérateurs spéciaux:
  - **@pre**: accès à une valeur d'une propriété d'avant l'opération (old de Eiffel).
  - **result**: accès au résultat de l'opération.

Post-condition

context Etudiant::age() : **Integer**
post correct : **result** = (today − naissance).years()

context Typename::operationName(param1: type1, ...): Type
post: **result** = ...

context Typename::operationName(param1: type1, ...): Type
post resultOk: **result** = ...

Post-condition : valeurs précédentes

A l'intérieur d'une postcondition, deux valeurs sont disponibles pour chaque propriété:

- la valeur de la propriété avant l'opération.
- la valeur de la propriété après la fin de l'opération.

context Personne:: anniversaire ()
post: age = age**@pre** + 1

context Enseignant::augmentation(v : **Integer**)
post: **self**. salaire = **self**. salaire **@pre** + v

Corps d'une opération

- Spécification du corps d'une opération sans effet de bord.

context Universite :: enseignants() : **Set**(Enseignant)
body:
self.departements.enseignants→**asSet**()

Définitions

- Définition de nouveaux attributs et opérations dans une classe existante.

context Classe
  def: nomatt : type = expr
  def: nomop(...) : type = expr

Définitions

- Utile pour décomposer des expressions complexes, sans surcharger le modèle.

```
context Enseignant
    def: eleves() : Bag(Etudiants) =
        self.enseigne.etudiant
    inv: self.titre = Titre::prof implies self.eleves()
        →forAll(each | each.estAdmis())
        -- un professeur a toujours 100% de reussite

context Departement
    def: eleves():Set(Etudiants) =
        self.enseignants.enseigne.etudiant→asSet()
```

Valeur initiale

- Spécification de la valeur initiale d'un attribut ou d'un rôle (Association End).
- Le type de l'expression doit être conforme au type de l'attribut ou du rôle.

context Typename::attributeName: Type
init : -- *expression representant la valeur initiale*

context Enseignant:: salaire : **Integer**
init : 800

Propriétés dérivées

- Spécification de la valeur dérivée d'un attribut ou d'un rôle (Association End).

context Typename::assocRoleName: Type
derive: *−− expression representant la regle de derivation*

Propriétés dérivées

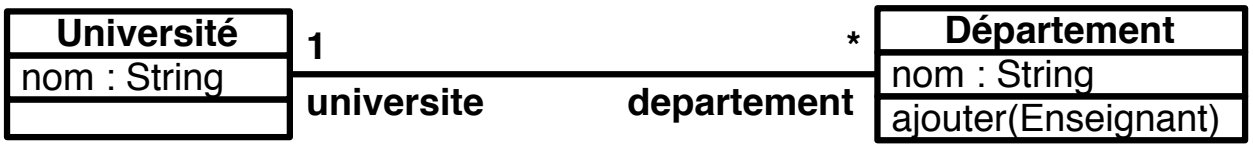
context Enseignant:: service : **Integer**
derive: **self** . enseigne . heures→**sum**()

context Personne:: celibataire : **Boolean**
derive: **self** . conjoint →**isEmpty**()

Plan

Rôles: navigation

Il est possible de naviguer à travers les associations, en utilisant le rôle opposé:



context Departement
inv: **self**. universite →**notEmpty**()

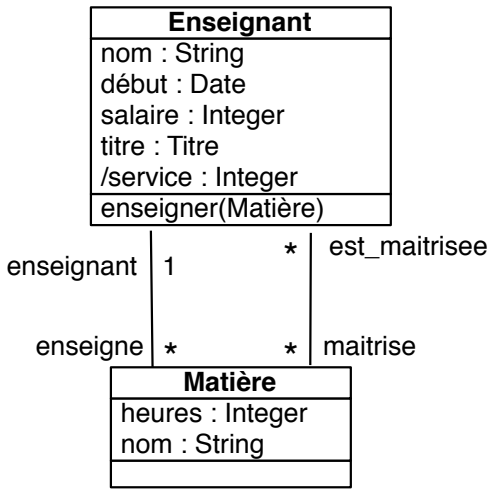
context Universite
inv: **self**.departement→ (...)

Rôles: cardinalités

Le type de la valeur de l'expression dépend de la cardinalité maximale du rôle. Si égal à 1, alors c'est un classificateur. Si > 1, alors c'est une collection.

context Matiere
-- un objet:
inv: **self**.enseignant.**oclInState**(disponible)

-- une collection (Set):
inv: **self**.est_maitrisee →**notEmpty**()



Rôles: noms

- Quand le nom de rôle est absent, le nom du type (en minuscule) est utilisé.
- Il est possible de naviguer sur des rôles de cardinalité 0 ou 1 en tant que collection:

context Departement **inv**: **self**.chef→**size**() = 1

context Departement **inv**: **self**.chef.age > 40

context Personne **inv**: **self**.epouse→**notEmpty**()
implies **self**.epouse.sexe = Sexe::femme

Rôles: navigation

- Il est possible de combiner des expressions:

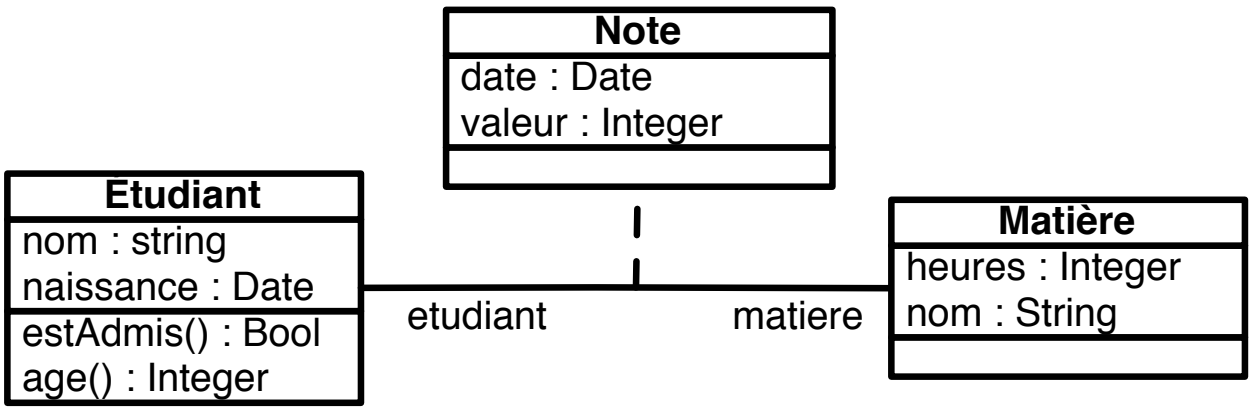
context Personne **inv**:
self .epouse→**notEmpty**() **implies self**.epouse.age ≥ 18 **and**

self .mari→**notEmpty**() **implies self**.mari.age ≥ 18

Classe-association

- On utilise le nom de la classe-association, en minuscules:

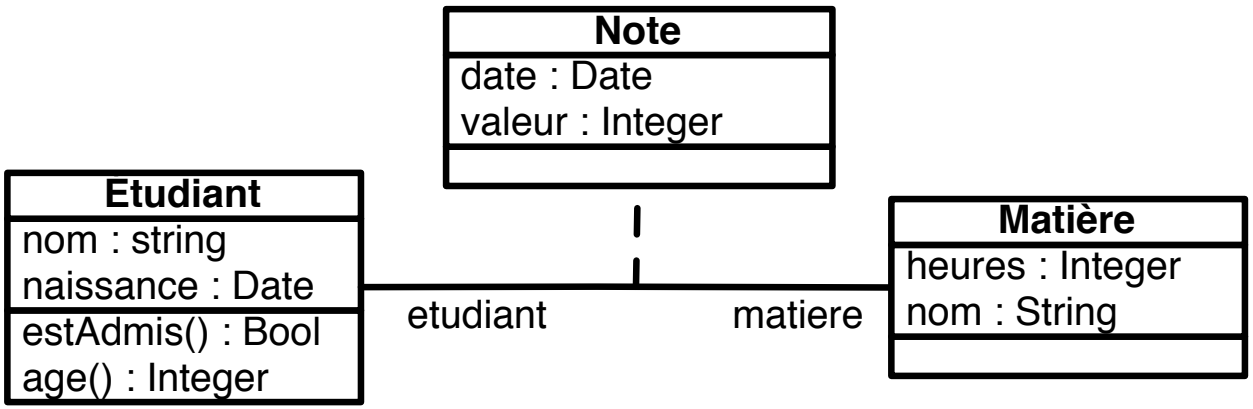
context Etudiant
inv:
—— *La moyenne des notes d'un étudiant est toujours supérieure à 4:*
self.note→average() > 4



Classe-association

- Il est possible de naviguer à partir de la classe-association en utilisant les noms de rôle et la notation pointée:

context Note **inv**:
self . etudiant . age() $\geq$ 18
self . matiere . heures $>$ 3



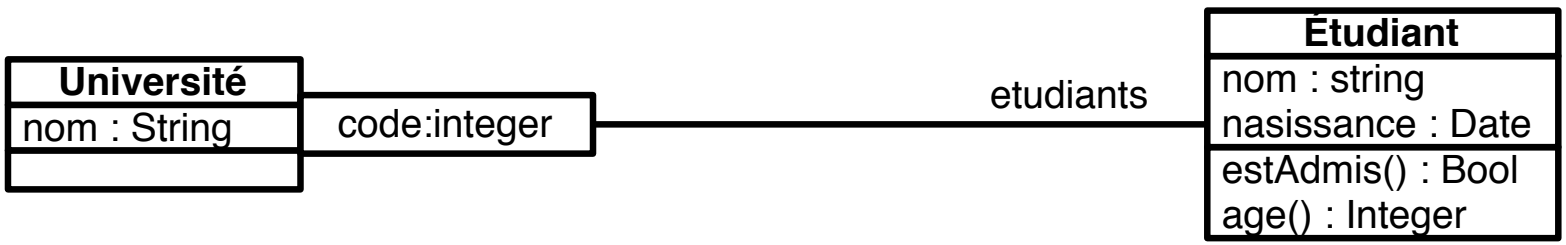
Associations qualifiées

- La valeur du qualificatif est mise entre crochets:

```
context Universite
    -- Le nom de l'étudiant 8764423 est "Martin".
    inv: self.etudiants[8764423].nom = "Martin"
```

- Quand la valeur n'est pas précisée, le résultat est une collection:

```
context Universite
    -- Il existe un étudiant dont le nom est "Martin"
    inv: self.etudiants->exists(each | each.nom = "Martin")
```



Plan

Sortes de Collection (1/2)

- Set: ensemble non ordonné.
  - Résultat d'une navigation.
  - $\{1, 2, 45, 4\}$
- OrderedSet: ensemble ordonné.
  - Résultat d'une navigation par un rôle ordonné (orné par une étiquette $\{ordered\}$).
  - $\{1, 2, 4, 45\}$

Sortes de Collection (2/2)

- Bag: multi-ensemble non ordonné.
  - Résultat de navigations combinées.
  - {1, 3, 4, 3}
- Sequence: multi-ensemble ordonné.
  - Navigation à travers un rôle ordonné {Ordered}
  - {1, 3, 3, 5, 7}
  - {1..10}

Opérations sur les collections

- **isEmpty**() : vrai si la collection est vide.
- **notEmpty**() : vrai si la collection contient au moins un élément.
- **size**() : nombre d'éléments dans la collection.
- **count**(elem) : nombre d'occurrences de *elem* dans la collection.

Opération sur les collections

- Select et Reject
- Collect
- Collect Nested
- For All
- Exists
- Closure
- Iterate

Select et Reject

collection →**select**(elem:T | bool−expr) : collection
collection →**reject**(elem:T | bool−expr) : collection

- Sélectionne (respectivement rejette) le sous-ensemble d'une collection pour lequel la propriété *expr* est vraie (respectivement fausse).

Select et Reject

Syntaxes possibles:

context Departement **inv**:

-- *sans iterateur*

self.enseignants→**select**(age > 50)→**notEmpty**()

self.enseignants→**reject**(age > 23)→**isEmpty**()

-- *avec iterateur*

self.enseignants→**select**(e | e.age > 50)→**notEmpty**()

-- *avec iterateur typé*

self.enseignants→**select**(e : Enseignant | e.age > 50)→
notEmpty()

Collect

collection →**collect**(expr) : collection

- Évalue l'expression *expr* pour chaque élément de la collection, et rend une autre collection, composée par les résultats de l'évaluation ;
- Le résultat est un multi-ensemble (**Bag**) ;
- Si le résultat de *expr* est une collection, le résultat ne sera pas une collection de collections. Les collections de collections sont automatiquement mises à plat.

Collect

Syntaxe:

```
context Departement:
    self.enseignants→collect(nom)
    self.enseignants→collect(e | e.nom)
    self.enseignants→collect(e: Enseignant | e.nom)

    -- conversion de Bag en Set:
    self.enseignants→collect(nom)→asSet()

    -- raccourci:
    self.enseignants.nom
```

Collect Nested

Opération similaire à **collect**, mais qui ne met pas à plat les collections de collections.

context Universite
self.departement→collectNested(enseignants)

Les collections de collections peuvent être mises à plat grâce à l'opération **flatten**() :

$$\mathbf{Set}\{\mathbf{Set}\{1, 2\}, \mathbf{Set}\{3, 4\}\} \rightarrow \mathbf{flatten}() = \mathbf{Set}\{1, 2, 3, 4\}$$

For All

collection $\rightarrow$ **forAll**(elem:T | bool−expr) : **Boolean**

Vrai si *expr* est vrai pour chaque élément de la collection.

For All

Syntaxe:

context Departement **inv**:
−− Tous les enseignants sont des MdC.
self.enseignants→**forAll**(titre = Titre::mdc)

self.enseignants→**forAll**(each | each.titre = Titre::mdc)

self.enseignants→**forAll**(each: Enseignants | each.titre = Titre::mdc)

For All

Produit cartésien:

context Departement **inv**:
  self.enseignants→**forAll**(e1, e2 : Enseignant |
    e1 <>e2 **implies** e1.nom <>e2.nom)

-- *equivalent à*
  self.enseignants→**forAll**(e1 | **self**.enseignants→
    forAll(e2 | e1 <>e2 **implies** e1.nom <>e2.nom))

Exists

collection →**exists**(boolean−expression) : **Boolean**

Rend vrai si *expr* est vraie pour au moins un élément de la collection.

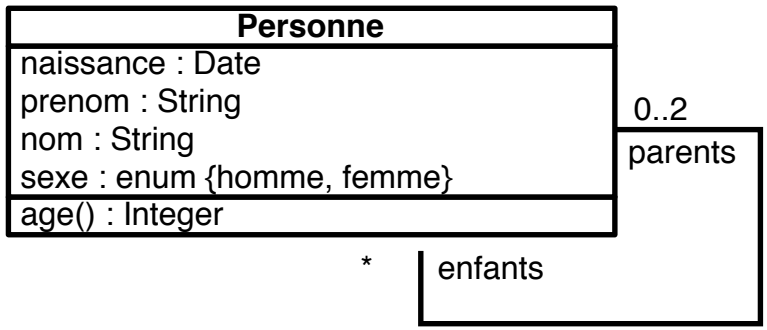
context: Departement **inv**:
    self.enseignants→**exists**(e: Enseignant |
        p.nom = 'Martin')

Closure (1/2)

source→**closure**(v : Type | expression −with−v)

- Evalue de façon récursive *expression-with-v* à l'ensemble *source* et additionne les résultats successifs à *source*.
- L'itération termine lorsque l'évaluation de *expression-with-v* est un ensemble vide.

Closure (2/2)



context Personne
def descendants() : **Set**(Personne) =
self . children →**closure**(children)

Iterate

collection →**iterate**(elem: T; reponse: T = <valeur> |
<expr−avec−elem−et−reponse>)

Opération générique (et complexe) applicable aux collections.

Iterate

```
context Departement inv:
    self.enseignants→select(age > 50)→notEmpty()

    -- expression équivalente:
    self.enseignants→iterate(e: Enseignant;
        answer: Set(Enseignant) = Set {} |
            if e.age > 50 then answer.including(e)
            else answer endif) → notEmpty()
```

Autres opérations sur les Collections

- **includes**(elem), **excludes**(elem) : vrai si *elem* est présent (resp. absent) dans la collection.
- **includesAll**(coll), **excludesAll**(coll) : vrai si tous les éléments de *coll* sont présents (resp. absents) dans la collection.
- **union**(coll), **intersection**(coll) : opérations classiques d'ensembles.
- **asSet**(), **asBag**(), **asSequence**() : conversions de type.

Plan

Tuples (N-Uplets)

Définition

Une N-Uplet est une séquence finie de objets ou *composantes*, où chaque composante est nommée. Les types des composantes sont potentiellement différents.

Exemples :

Tuple {nom:**String** = 'Martin', age:**Integer** = 42}
Tuple {nom:'Colette', notes:**Collection**(**Integer**) = **Set**{12, 13, 9},
formation:**String** = 'Informatique'}

N-Uplets

Notation

Les types sont optionnels. L'ordre des composantes n'est pas important :

Expressions équivalentes :

Tuple {**name**: **String** = 'Martin,' age: **Integer** = 42}
Tuple {**name** = 'Martin,' age = 42}
Tuple {age = 42, **name** = 'Martin'}

N-Uplets

Les valeurs des composantes peuvent être spécifiées par des expressions OCL :

```
context Universite def:
 statistiques  : Set(Tuple(dpt:Departement, nbEtudiants:Integer,
                                admis: Set(Etudiants), moyenne:
                                    Integer)) =
     departement→collect(each |
       Tuple {dpt:Departement = each,
            nbEtudiants: Integer = each.eleves()→size(),
            admis: Set(Person) = each.eleves()→select(estAdmis()),
            moyenne: Integer = eleves().note→avg()
           }
      )
```

N-Uplets

Notation

Les composantes sont accessibles grâce à leurs noms, en utilisant la notation pointée :

Tuple {nom:**String**='Martin', age:**Integer** = 42}.age = 42

L'attribut statistiques définit précédemment peut être utilisé à l'intérieur d'un autre expression OCL :

context Universite **inv**:
statistiques →**sortedBy**(moyenne)→**last**().dpt.nom = '
Informatique'
-- *Le département d'informatique possède les meilleurs étudiants.*

Messages

Appel d'opérations et envoi d'événements

On utilise l'opérateur "^" (hasSent) pour spécifier qu'une communication a eu lieu.

context Subject::hasChanged()
post: observer^update(12, 14)

Messages

Jokers

context Subject::hasChanged()
post: observer^update(? : **Integer**, ? : **Integer**)

L'opérateur ≪?≫ indique que les valeurs des arguments ne sont pas connues.

Le type OclMessage

OCL introduit le type OclMessage. L'opérateur «^^»permet d'accéder à une séquence de messages envoyés.

observer^^update(?, ?)
-- *renvoie une séquence de messages envoyés.*

Valeurs des arguments

Il est possible d'accéder aux valeurs des arguments d'un message grâce aux noms des paramètres de l'opération ou du signal:

context Subject::hasChanged()
post: **let** messages : **Sequence**(OclMessage) =
observer^^update(? : **Integer**, ? : **Integer**) **in**
messages→**notEmpty**() **and**
messages→**exists**(m | m.i > 0 **and** m.j ≥ m.i)

Messages

Accès aux valeurs renvoyés

L'opérateur OclMessage::**result**() permet l'accès à la valeur renvoyée d'une opération (les signaux ne renvoient pas de valeur). L'opérateur OclMessage::hasReturned() retourne vrai si l'opération a renvoyé une valeur.

```
context Person::giveSalary(amount : Integer)
post: let message : OclMessage = company^getMoney(amount) in
      message.hasReturned()
      -- getMoney was sent and returned
      and
      message.result()
      -- the getMoney call returned true
```

Stéréotypes des contraintes

Plusieurs stéréotypes sont définis en standard dans UML:

- Invariants de classe: ≪ invariant ≫
- Pré-conditions: ≪ precondition ≫
- Post-conditions: ≪ postcondition ≫
- Définitions de propriétés: ≪ definition ≫

Package context

Il est possible de spécifier explicitement le nom du paquetage auquel appartient une contrainte :

```
package Package::SubPackage
context X inv:
    -- some invariant

context X::operation()
pre:
    -- some precondition
endpackage
```

Règles de conformité de types

Type	**Est conforme à**	**Condition**
Set(T1)	Collection(T2)	Si T1 est conforme à T2
Sequence(T1)	Collection(T2)	Si T1 est conforme à T2
Bag(T1)	Collection(T2)	Si T1 est conforme à T2
OrderedSet(T1)	Collection(T2)	Si T1 est conforme à T2
Integer	Real	

Propriétés prédéfinies

oclIsTypeOf(t : **OclType**):**Boolean**
oclIsKindOf(t : **OclType**):**Boolean**
oclInState(s : **OclState**):**Boolean**
oclIsNew():**Boolean**
oclIsUndefined():**Boolean**
oclIsInvalid ():**Boolean**
oclAsType(t : Type):Type
allInstances():**Set**(T)

Exemples:

context Universite
inv: **self**.**oclIsTypeOf**(Universite) −− *vrai*
inv: **self**.**oclIsTypeOf**(Departement) −− *faux*

Expression *Let*

Quand une sous-expression apparaît plus d'une fois dans une contrainte, il est possible de la remplacer par une variable qui lui sert d'alias :

context Person **inv**:
  let income : **Integer** = **self**.job.salary→**sum**() **in**
  if isUnemployed **then**
    income < 100
  else
    income ≥ 100
  endif

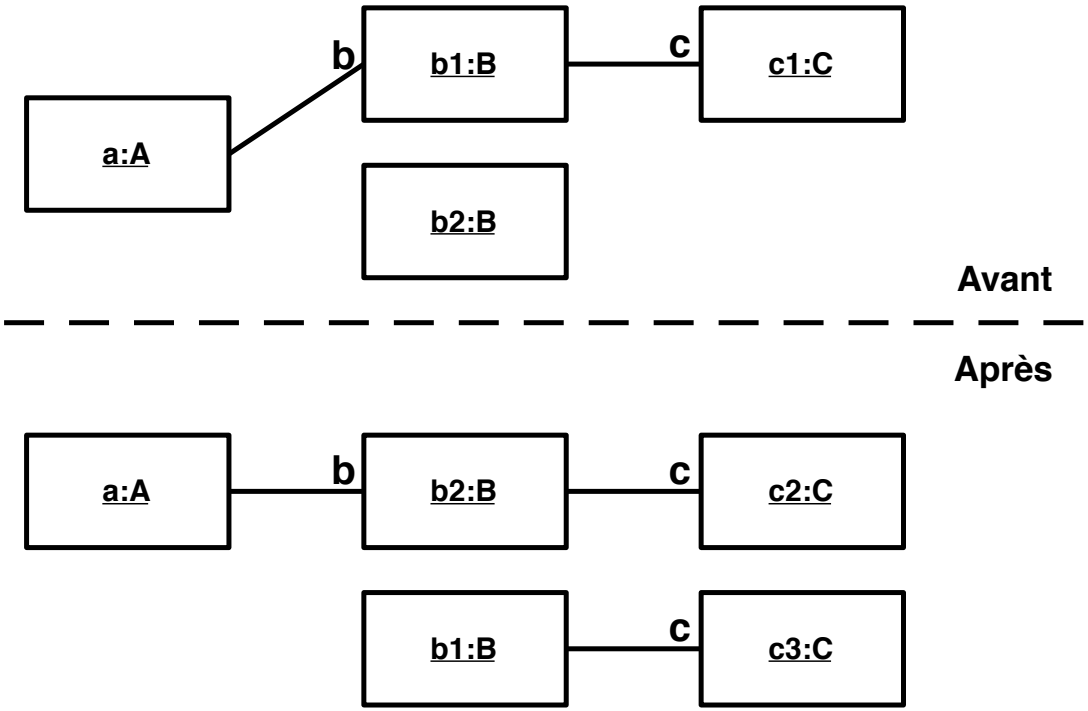
Valeurs précédentes (1/2)

Quand la valeur @pre d'une propriété est un objet, toutes les valeurs atteintes à partir de cet objet sont nouvelles :

a.b**@pre**.c
-- l'ancienne valeur de b, disons X,
-- et la nouvelle valeur de c de X

a.b**@pre**.c**@pre**
-- l'ancienne valeur de b, disons X,
-- et l'ancienne valeur de c de x.

Valeurs précédentes (2/2)



a.b**@pre**.c -- *la nouvelle valeur de b1.c, c3*
a.b**@pre**.c**@pre** -- *l'ancienne valeur de b1.c, c1*
a.b.c -- *la nouvelle valeur de b2.c, c2*

Héritage de contrats

Rappel: principe de substitution de Liskov (LSP) :

"Partout où une instance d'une classe est attendue, il est possible d'utiliser une instance d'une de ses sous-classes."

Héritage d'invariants

Conséquences du principe de substitution de Liskov sur les invariants :

- Les invariants sont toujours hérités par les sous-classes.
- Une sous-classe peut renforcer l'invariant.

Héritage de pré et post-conditions

Conséquences du LSP pour les pré et post-conditions :

- Une pré-condition peut seulement être assouplie (contrevariance).
- Une post-condition peut seulement être renforcée (covariance).

Plan

Conseils de modélisation

- Faire simple : les contrats doivent améliorer la qualité des spécifications et non les rendre plus complexes.
- Toujours combiner OCL avec un langage naturel : les contrats servent à rendre les commentaires moins ambigus et non à les remplacer.
- Utiliser un outil.

Rappels

La conception par contrats permet aux concepteurs de :

- Modéliser de manière plus précise ;
- Mieux documenter un modèle ;
- Rester indépendant de l'implémentation ;
- Identifier les responsabilités de chaque composant.

Applicabilité

- Génération de code :
  - assertions en Eiffel, Sather.
  - dans d'autres langages, grâce à des outils spécialisés : iContract, JMSAssert, jContractor, Handshake, Jass, JML, JPP, etc.
- Génération de tests mieux ciblés.

Références

- The Object Constraint Language – Jos Warmer, Anneke Kleppe.
- OCL home page – http://www.klasse.nl/ocl/
- OCL tools – http://www.um.es/giisw/ocltools
- OMG Specification v2.3.1 –http://www.omg.org/spec/OCL/Current/
- OMG UML 2.5 Working Group.

Outils

- ModelRun (Boldsoft). http://www.borland.com/company/boldsoft.html/products/modelru
- OCL Compiler (Cybernetic Intelligence GMBH). http://www.cybernetic.org/
- OCL Checker (Klasse Objecten)
- USE (Mark Richters). http://www.db.informatik.uni-bremen.de/projects/USE/
- Dresden OCL. http://dresden-ocl.sourceforge.net/
- Octopus (Warmer & Kleppe). http://octopus.sourceforge.net/