Conception par contrats avec UML

OCL – Object Constraint Language

Gerson Sunyé

gerson.sunye@univ-nantes.fr

LINA

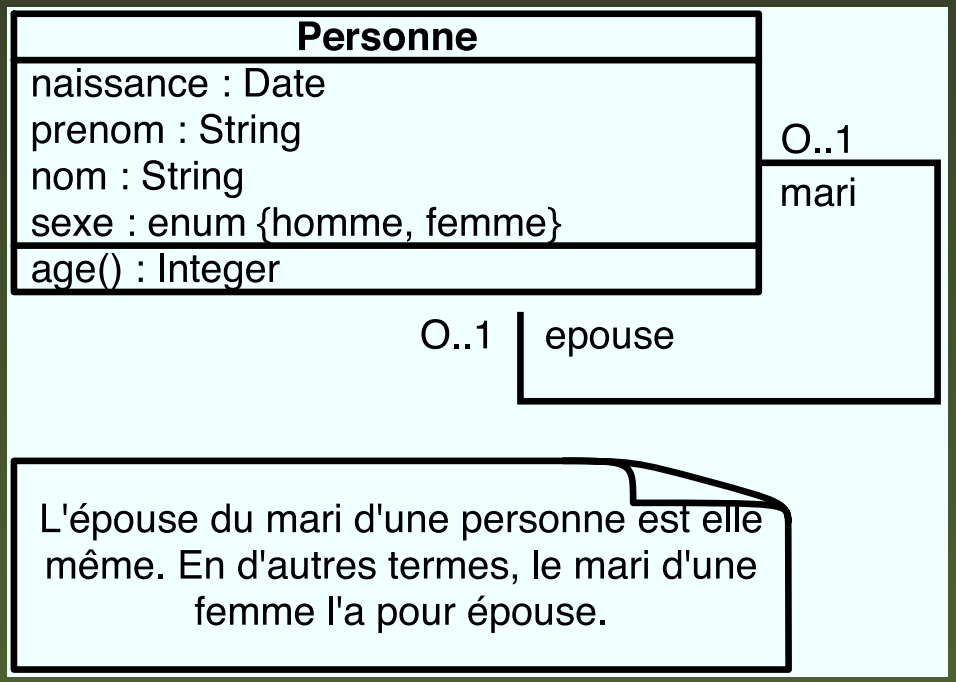
Plan

- **Introduction**
- Invariants, pré et post-conditions
- Spécification de propriétés
- Expressions OCL portant sur les Associations
- Concepts avancés
- Conclusion

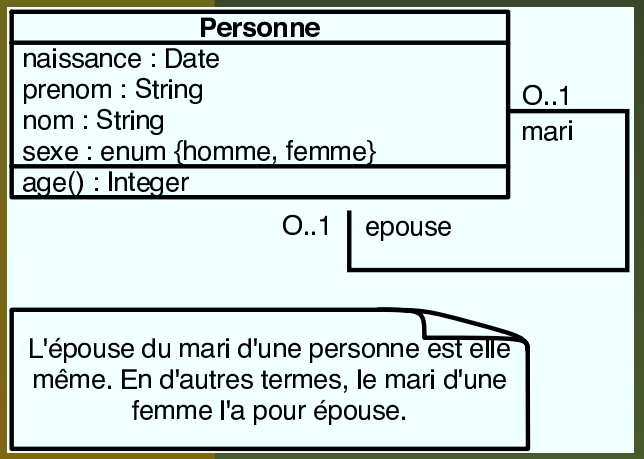
Introduction

Motivation

Peut-on rendre plus précis un diagramme UML?



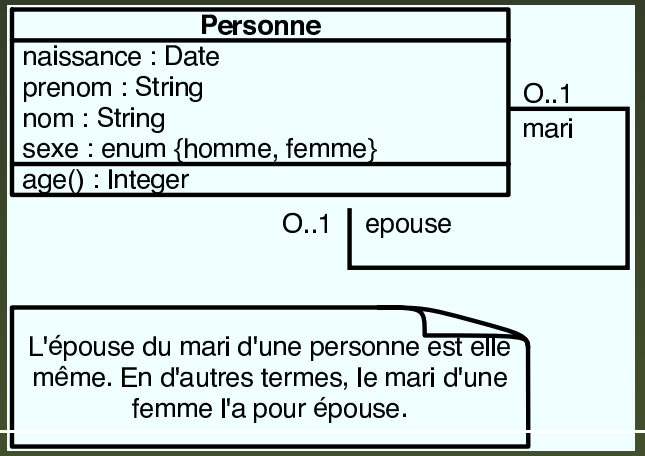
Motivation



- 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** }

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 (à travers les stéréotypes).
- Formel, non ambigu, mais facile à utiliser (même par les non mathématiciens).

OCL – Object Constraint Language

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

- Types de base: String, Boolean, Integer, Real;
- Classificateurs et ses propriétés: Attributs, Opération qui sont des "fonctions pures";
- Associations;
- États (des machines d’états associées).

Différentes sortes de contraintes (1/2)

- Invariants de classe.
  - Une contrainte qui doit être respectée par toutes les instances d'une classe.
- Pré-condition d'une opération (ou transition).
  - Une contrainte qui doit être toujours vraie **avant** l'exécution d'une opération.
- Post-condition d'une opération (ou transition).
  - Une contrainte qui doit être toujours vraie **après** l'exécution d'une opération.

Différentes sortes de contraintes (2/2)

- Définition de nouvelles propriétés.
  - Définition de nouveaux attributs ou opérations à l'intérieur d'une classe.
- Établir la valeur initiale d'un attribut.
- Définition de propriétés dérivées.
  - Préciser la valeur d'un attribut ou association dérivés.
- Spécifier le corps d'une opération.
  - Spécification d'une opération dans effet de bord (query).

Stéréotypes des contraintes

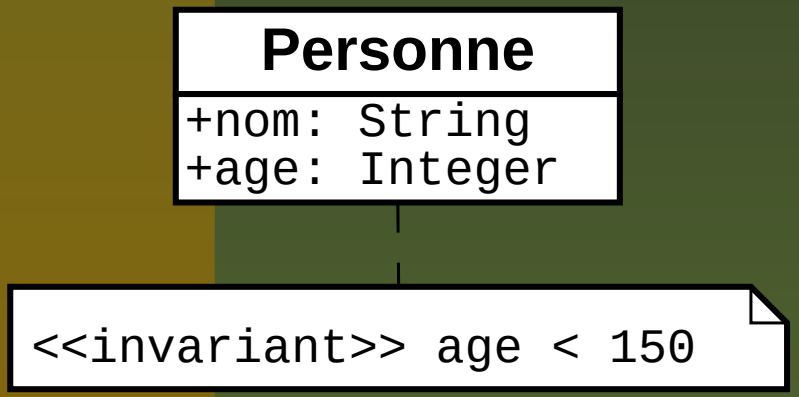
Trois stéréotypes sont définis en standard dans UML:

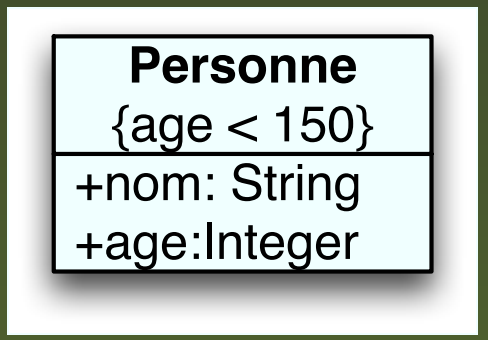
- Invariants de classe: «invariant»
- Pré-conditions: «precondition»
- Post-conditions: «postcondition»

Notation

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

- **context Personne inv: self.age < 150**
- **context Personne inv: age < 50**



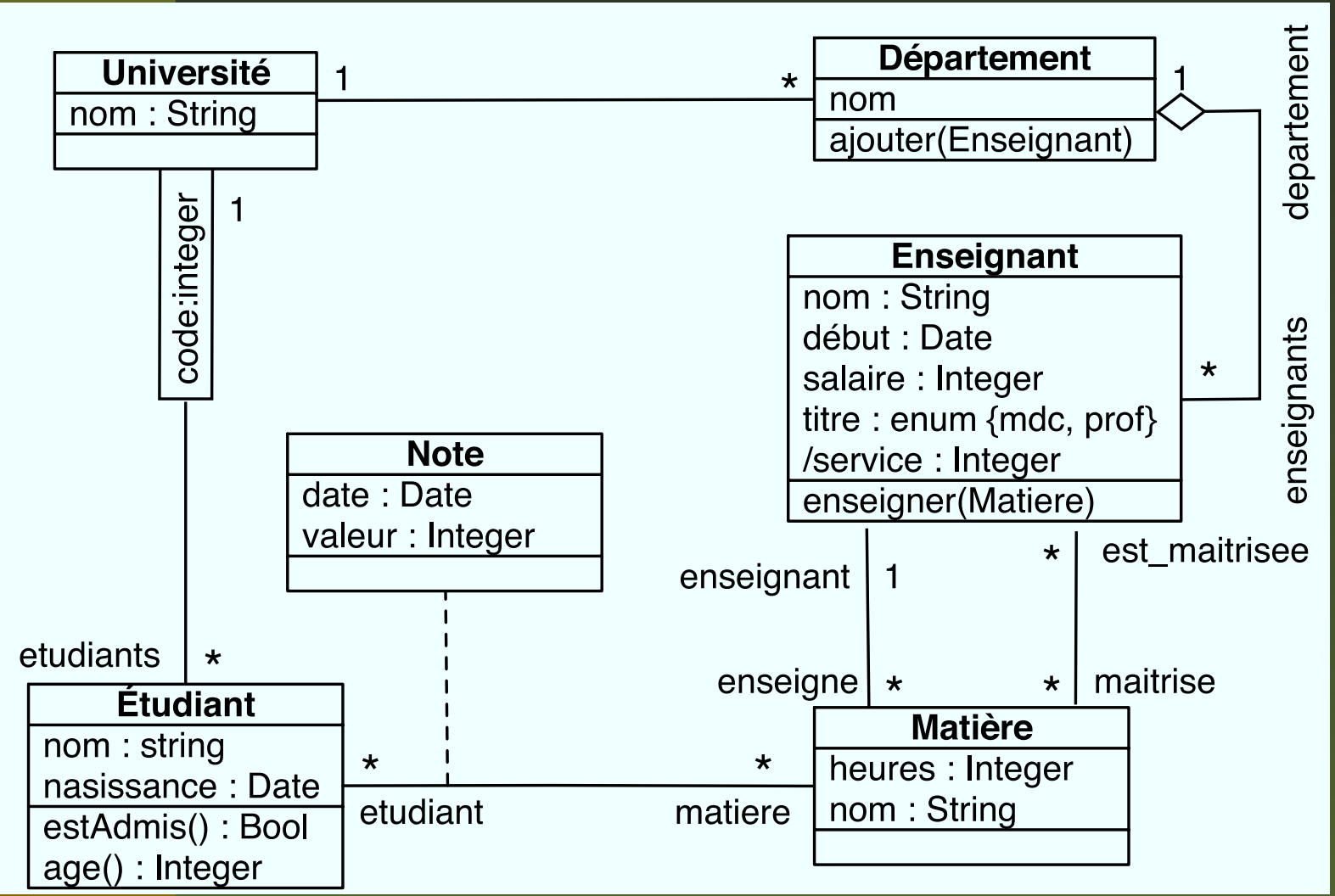


Plan

- Introduction
- **Invariants, pré et post-conditions**
- Spécification de propriétés
- Expressions OCL portant sur les Associations
- Concepts avancés
- Conclusion

Invariants, pré et post-conditions

Exemple de modèle



Contexte

- Toute contrainte OCL est liée à un contexte spécifique, l'élément auquel la contrainte 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.

Invariants de classe

- Dans un état stable, toute instance d'une classe doit vérifier les invariants de cette classe.
- Expression OCL stéréotypée «invariant».
- Exemples:

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

context e : Etudiant **inv**: e.age > 16

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

Éléments d'une expression OCL

Les éléments suivants peuvent être utilisés dans une expression OCL:

- Les types de base: String, Boolean, Integer, Real.
- Les Classificateurs et leurs propriétés:
  - Attributs d'instance et de classe.
  - Opérations de *Query* (i.e. sans effet de bord) d'instance et de classe.
  - États des machines d'états associées.
- Associations du modèle UML.

Types de base

type	values
Boolean	true, false
Integer	1, -5, 2, 34, 26524, . . .
Real	1.5, 3.14, . . .
String	‘To be or not to be. . . ’

Opérations sur les types de base

type	operations
Integer	=, *, +, -, /, abs, div, mod, max, min
Real	=, *, +, -, /, abs, floor, round, max, min, >, <, . . .
String	=, size, concat, toUpper, toLower, substring

Objets et propriétés

Une expression OCL peut faire référence à:

- Des classificateurs (types, classes, interfaces, associations, datatypes, cas d'utilisation, etc.).
- Des propriétés:
  - Un Attribut.
  - Une Opération dont *isQuery* est vrai.
  - Un État.
  - Un rôle d'association.

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
    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

Spécification d’opérations

- Inspirée des Types Abstraits: une opération est composée par une signature, des pré-conditions et des post-conditions.
- Permet de contraindre l’ensemble de valeurs d’entrée d’une opération.
- Permet de spécifier la sémantique d’une opération: c’est 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: e.notNil ()

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** = ...

Postcondition: 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
```

Body

- Spécifie le corps d'une opération sans effet de bord (query).

context Universite :: enseignants () : **Set**(Enseignant)

body: **self** .departements . enseignants →**asSet**()

Plan

- Introduction
- Invariants, pré et post-conditions
- **Spécification de propriétés**
- Expressions OCL portant sur les Associations
- Concepts avancés
- Conclusion

Spécification de propriétés

Définitions

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

contextClasse

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
```

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

- Introduction
- Invariants, pré et post-conditions
- Spécification de propriétés
- **Expressions OCL portant sur les Associations**
- Concepts avancés
- Conclusion

Expressions OCL portant sur les Associations

Rôles: navigation

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

context Departement

inv: **self** . universite . notNil ()

context Universite

inv: **self** .departement (...)

Rôles: navigation

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** .competent→**notEmpty**()

Rôles: navigation

- 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 $\geq$ 18 **and**

self.mari→**notEmpty**() **implies** **self**.mari.age $\geq$ 18

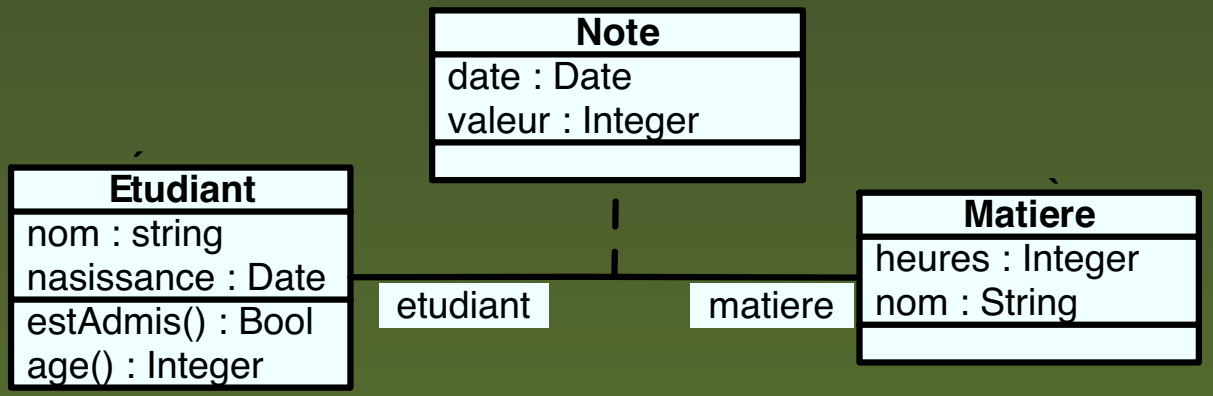
Association Classe

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

context Etudiant **inv**:

—— *Ensemble des notes:*

self .note



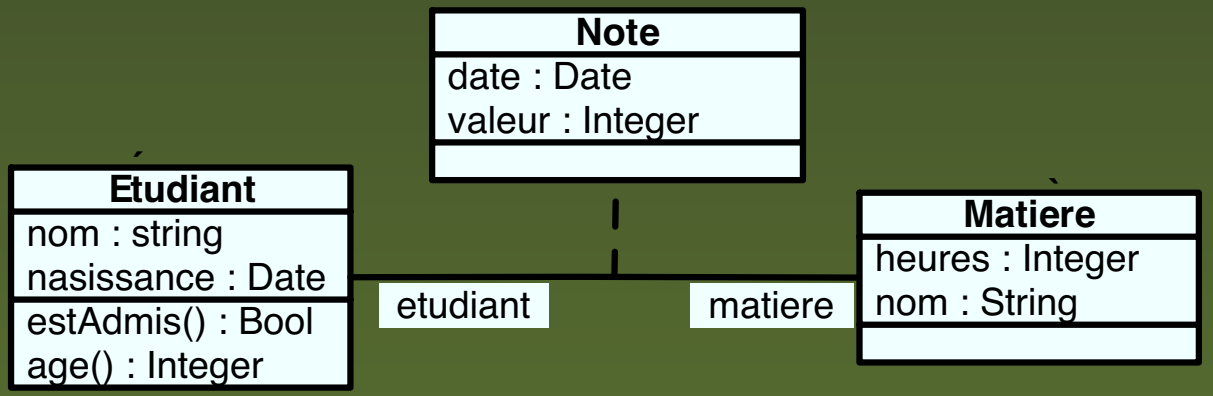
Association Classe

- Il est possible de naviguer 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 mis entre crochets:

  context Universite **inv**: **self** . etudiants [8764423]

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

  context Universite **inv**: **self** . etudiants

Sortes de Collection (1/2)

- Set: ensemble non ordonné.
  - Résultat d'une navigation.
  - {1, 2, 45, 4}
- OrderedSet: ensemble ordonné.
  - navigations combinées.
  - {1, 2, 4, 45}

Sortes de Collection (2/2)

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

Collections de collections

En UML 1.5, les collections de collections étaient mises à plat automatiquement. En UML 2.0 ce n'est plus le cas, la mise à plat doit être explicitement spécifiée.

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

Opérations sur les collections

- isEmpty() : vrai si la collection est vide.
- notEmpty() : vrai si la collection a 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
- ForAll
- Exists
- Iterate

Select et Reject

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

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

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

Select et Reject

Syntaxes possibles:

```
context Departement inv:
    self.enseignants→select(age > 50)→notEmpty()
    self.enseignants→reject(age < 23)→isEmpty()
context Departement inv:
    -- avec iterateur
    self.enseignants→select(e | e.age > 50)→notEmpty()
context Departement inv:
    -- 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*).

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

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:
    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.enseignant->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**

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

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

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()

context Departement inv:
    -- 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 (ou absent) dans la collection.
- includesAll(*coll*) : vrai si tous les éléments de *coll* sont dans la collection.
- union (*coll*), intersection (*coll*) : opérations classiques d'ensembles.
- asSet(), asBag(), asSequence() : conversions de type.

Plan

- Introduction
- Invariants, pré et post-conditions
- Spécification de propriétés
- Expressions OCL portant sur les Associations
- **Concepts avancés**
- Conclusion

Concepts avancés

Conformance de type

Type	Est conforme à
Set(T)	Collection(T)
Sequence(T)	Collection(T)
Bag(T)	Collection(T)
Integer	Real

Propriétés prédéfinies

oclIsTypeOf(t : **OclType**) : **Boolean**

oclIsKindOf(t : **OclType**) : **Boolean**

oclInState (s : OclState) : **Boolean**

oclIsNew() : **Boolean**

oclAsType(t : **OclType**) : instance de **OclType**

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 sert de 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 accédées à partir de cet objet sont nouvelles:

a.b**@pre**.c

– – *l'ancienne valeur de b, disons X,*

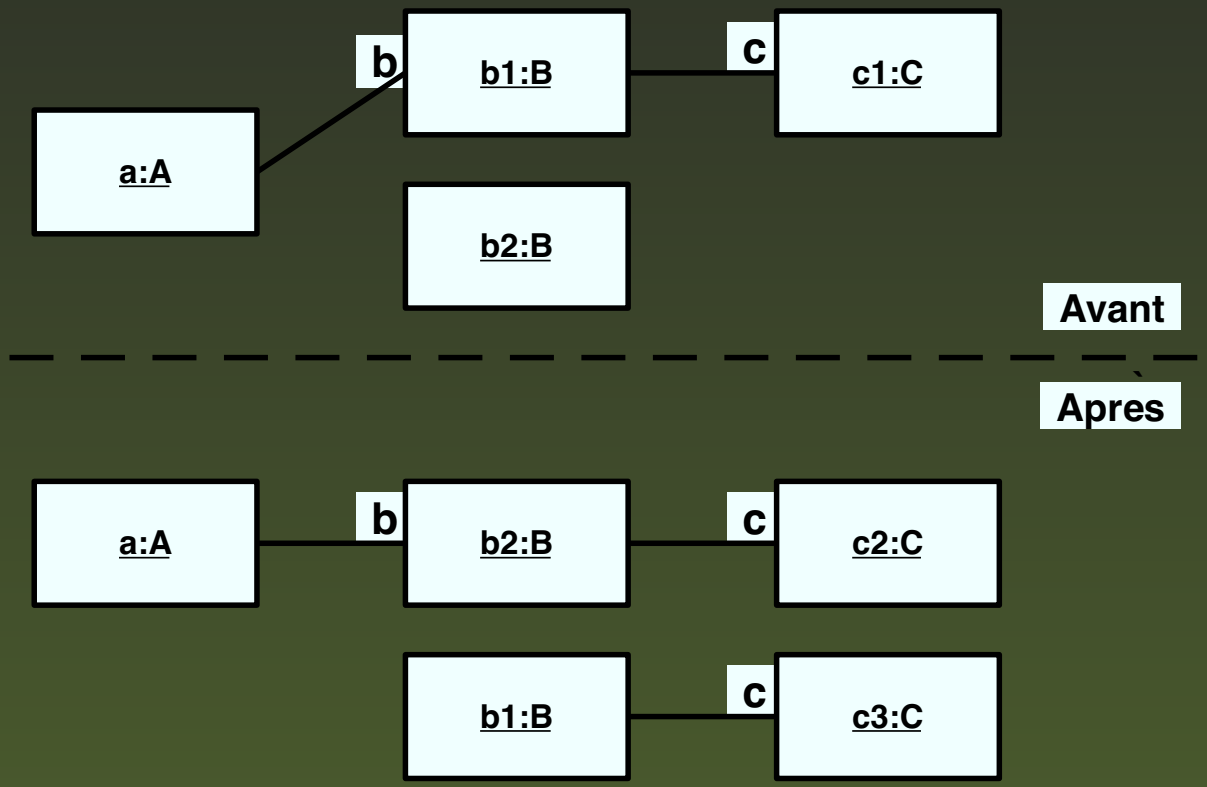
– – *et la nouvelle valeur de c de X*

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 pour les invariants:

- Les invariants sont toujours hérités par les sous-classes.
- Un 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

- Introduction
- Invariants, pré et post-conditions
- Spécification de propriétés
- Expressions OCL portant sur les Associations
- Concepts avancés
- **Conclusion**

Conclusion

Développement en cours

- Amélioration de la grammaire.
- Développement d’un méta-modèle (syntaxe abstraite).
- UML 2.0.

Autres nouveautés dans UML 2.0

- Tuples.
- Collections imbriquées (collectNested).
- Envoi de messages.

```
context Subject :: hasChanged()
  -- la post-condition doit asssurer que le message update()
  -- a ete envoye a tous les observateurs:
  post: observers →forAll( o | o^update() )
```

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 ou au patron Contract.
- Génération de tests mieux ciblés.

Références

- The Object Constraint Language – Jos Warmer, Anneke Kleppe.
- OCL home page – www.klasse.nl/ocl/
- OCL tools – www.um.es/giisw/ocltools
- OMG Specification v1.5.
- OMG UML 2.0 Working Group.