MGL7160 Méthodes formelles et semi-formelles

OCL partie I

Roger Villemaire

Département d'informatique
UQAM

14 janvier 2014

Plan

1. UML
2. OCL
   - Les types de base
   - Contraintes OCL

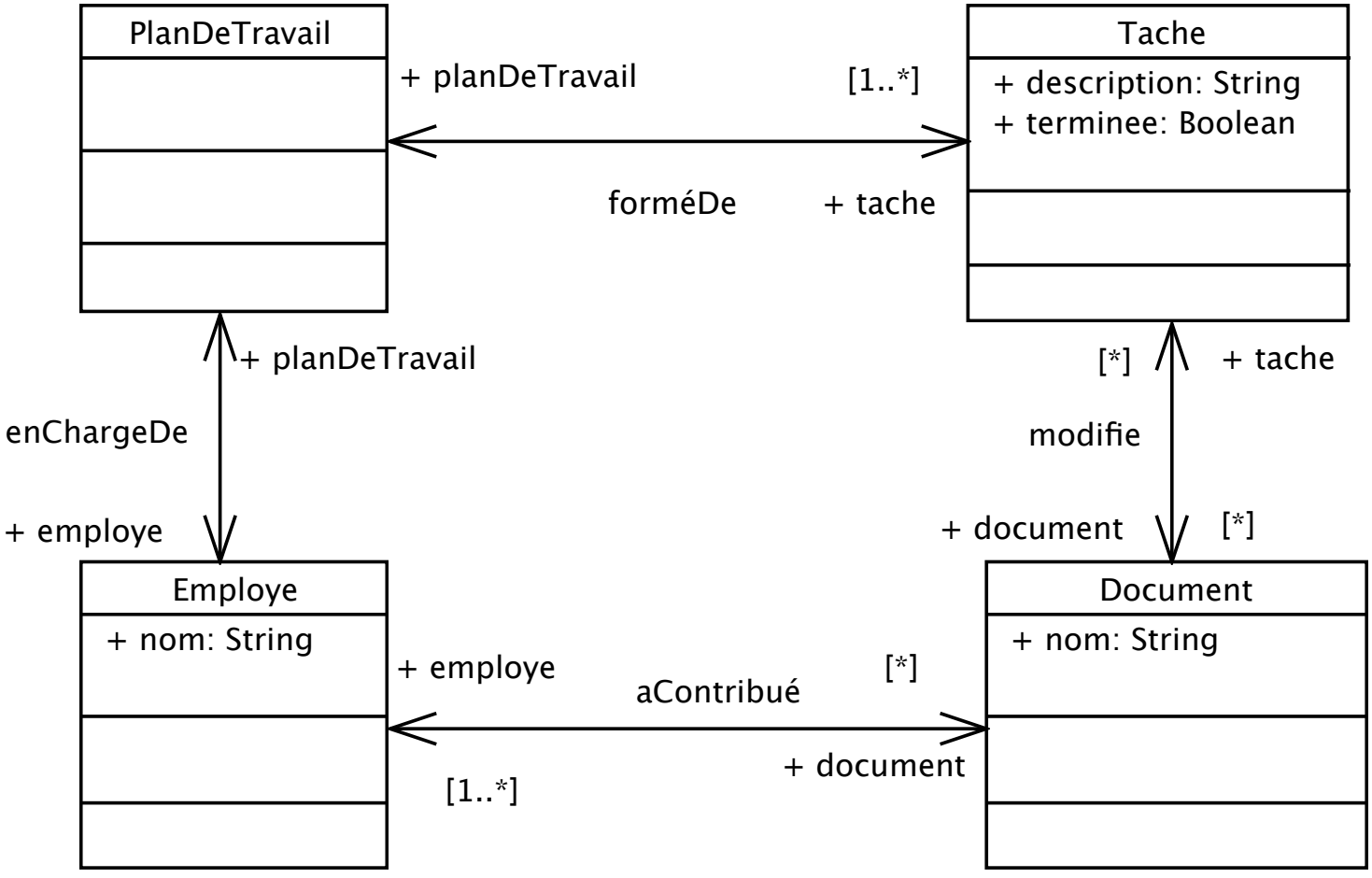
Unified Modeling Language

- Nous allons nous concentrer sur les diagrammes de classes UML qui permettent de :
  - décrire des classes avec leurs attributs (variables membres, champs) et opérations (méthodes).

Représentation graphique

- Un diagramme de classe UML est normalement représenté sous forme graphique.
  - Une classe est alors représentée par un rectangle contenant :
    - son nom,
    - certains de ses attributs,
    - ainsi que ses opérations.
  - Les autres attributs sont présents sous forme d'*associations* représentées par un trait joignant deux classes.

Exemple



Associations

- La représentation des associations met l'emphase sur les interactions (usages) des classes.
  - Une extrémité est déterminée par :
    - sa *multiplicité* (le nombre d'éléments associés),
    - son statut d'*unicité* (pas du doublons),
    - le fait d'être *ordonné* ou pas.
  - En pratique une extrémité d'une association est implémentée par un attribut de la classe *opposée*.
  - Plus précisément cet attribut est en général une collection d'éléments (en fonction de la multiplicité). Le type de collection est déterminé par l'unicité et le fait d'être ordonné ou non.

Limitation

- Cette vue simplifiée des diagrammes de classes UML sera suffisante pour nous.
- Bien qu'en général en UML on tienne compte des directions de *navigation* pour une association, nous allons toujours supposer des associations bidirectionnelles.
- Ceci n'est pas une véritable restriction puisque pour utiliser une extrémité dans une contrainte, celle-ci devrait être navigable.

Object Constraint Language

- Une norme de l'OMG (Object Management Group),
- En fait une partie de UML (Unified Modeling Language),
- Permet de décrire des *contraintes* sur les attributs de classes de diagrammes UML.

Comportement versus Contraintes

- En OCL on décrit des contraintes, c.-à-d. des propriétés qui doivent être satisfaites lors de l'exécution du système.
  - Il s'agit d'une approche *déclarative*, où on décrit ce qui devrait être vrai
  - et non une approche comportementale où on décrit comment on s'assure que ça le soit !
- OCL est donc un langage de description avec des primitives pour décrire l'état du système,
- et non un langage de programmation pour décrire le comportement du système.

Types de base OCL

- Boolean
  - true, false.
- Integer (qui est une sous-classe de Real)
  - 1, -5, 2, 34, 26524, ...
- Real
  - 1.5, 3.14, ...
- String
  - ’Ceci est une chaîne de caractères’, ’MGL7160’, ...

Quelques opérations

- Boolean : and, or, xor, not, implies, if-then-else-endif
  - true implies false = false,
  - if false then A else B endif = B
- Integer : *, +, -, /, div(), abs()
  - 5/2 = 2.5, 5.div(2) = 2
  - (-5).abs() = 5
- Real : *, +, -, /, floor()
  - 1.1 * 5.0 = 5.5
  - (5.2).floor() = 5.0
- String : concat(), +, size(), substring()
  - 'abc'.concat('def') = 'abcdef'
  - 'abc' + 'def' = 'abcdef'
  - 'abc'.size() = 3
  - 'abcdef'.substring(1,3) = 'abc'

Les collections

Collection	Ordre	Unicité
Set	non	oui
Bag	non	non
Sequence	oui	non
OrderedSet	oui	oui

- Set{0,1,2}
- Bag{0,0,1,1,1,2}
- Sequence{2,0,1,1,0,2}
- OrderedSet{2,0,1,3}
- Sequence{1..10} = Sequence{1..(6 + 4)}
  = Sequence{1,2,3,4,5,6,7,8,9,10}

Les tuplets

- Tuple {nom : String = ’Jean’, age : Integer = 35}
- Tuple {tailles : Collection(Integer) = Set{1, 3, 4},
  article : String = ’ecrou’,
  type : String = ’hex’}

Opérations sur les collections

- Les opérations sont toujours préfixées par ->
  - Set{0,1,2} -> size() = 3
  - Bag{0,0,1,1,1,2} -> isEmpty() = false
  - Sequence{2,0,1,1,0,2} -> notEmpty() = true
- sauf évidemment les opérations infixes, comme :
  - Bag{0,0,1,1,1,2} <> Bag{0,1,2} = true
  - Sequence{2,0,1,1,0,2} = Sequence{} = false

Inclusion et exclusion

- Set{0,1,2} -> includes(1) = true
- Bag{0,0,1,1,1,2} -> excludes(2) = false
- Sequence{2,0,1,1,0,2} -> includesAll(Set{2,0}) = true
- OrderedSet{2,0,1,3} -> excludesAll(Sequence{5,0}) = false

Itérations

- Bag{0,0,1,1,1,2} -> select(c : c > 0) = Bag{1,1,1,2}
- Sequence{2,0,1,1,0,2} -> reject(c : c > 1) = Sequence{0,1,1,0}
- Bag{1,2,0,1,2,3} -> collect(c : c.mod(2)) = Bag{1,0,0,1,0,1}
- Sequence{1,2,0,1,2,3} -> collect(c : c.mod(2)) = Sequence{1,0,0,1,0,1}
- Set{2,0,1,3} -> collect(c : c.mod(2)) = Bag{0,0,1,1}
- OrderedSet{2,0,1,3} -> collect(c : c.mod(2)) = Sequence{0,0,1,1}

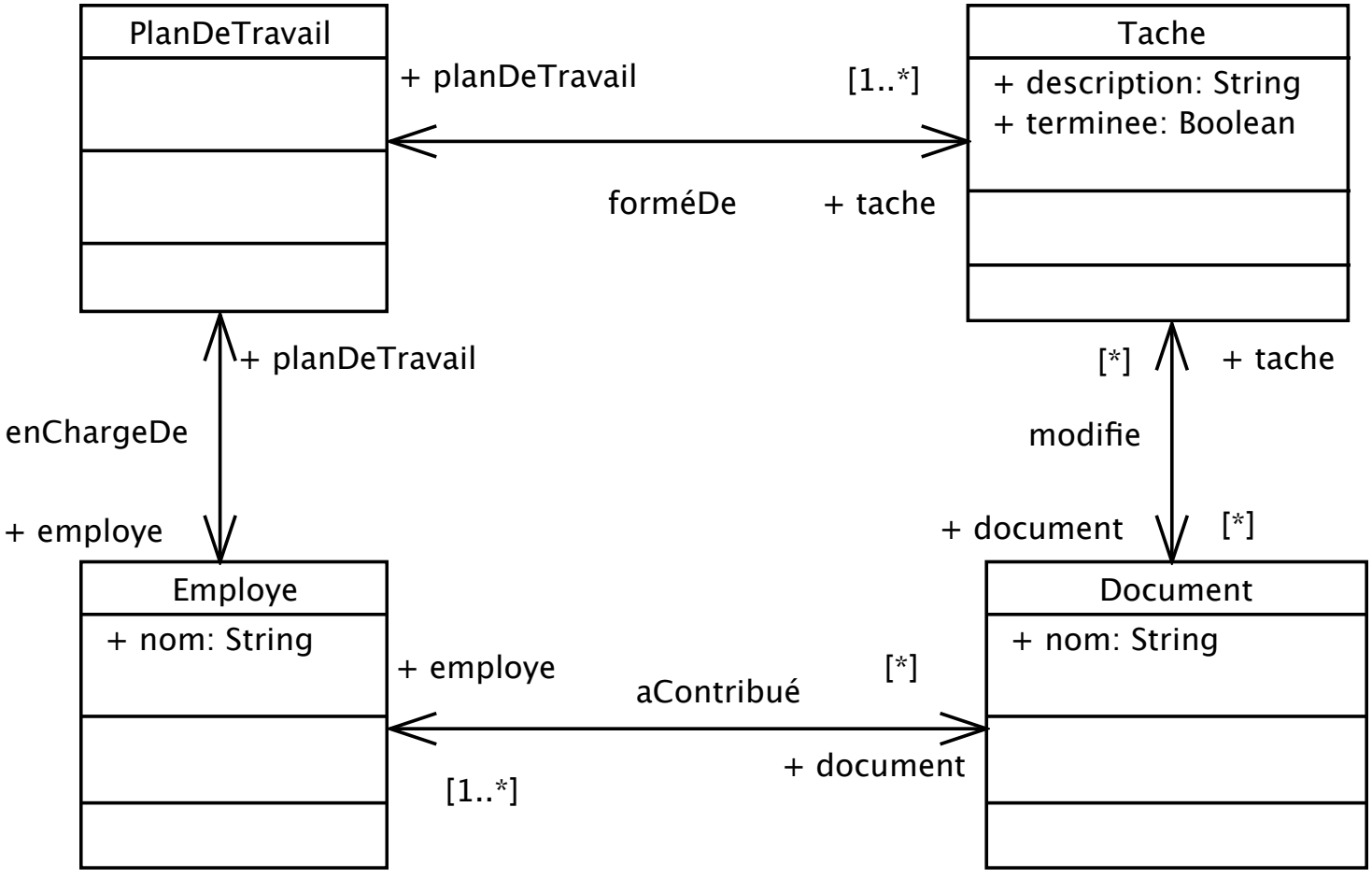
Invariants pré- post-conditions

- En pratique les expressions OCL servent principalement à spécifier des invariants, des pré- et des post-conditions.
  - invariant : propriété des instances d'une classe qui devrait *en principe* être toujours vérifiée.
  - pré-condition : propriété qui devrait toujours être vérifiée *avant* l'appel d'une opération.
  - post-condition : propriété qui devrait toujours être vérifiée *après* l'appel d'une opération.

Invariants

- Un invariant
  - est une propriété satisfaite par les instances *correctement construites* d'un classe.
  - décrit des contraintes sur les attributs de la classe qui doivent être maintenues.
  - permet de préciser le sens de la classe.

Exemple



Exemple d'invariants

- Le nom d'un Employé est toujours une chaîne non-vide :

```
context Employe
inv : nom <> ''
```

Exemple d'invariants

- Un Employé qui a plus de 10 Tâches dans son planDeTravail, doit avoir contribué à au moins 50 Documents.

```
context Employe
inv : planDeTravail.tache -> size() > 10
          implies document -> size() >= 50
```