OCL - Object Constraint Language

Laëtitia Matignon
laetitia.matignon@univ-lyon1.fr

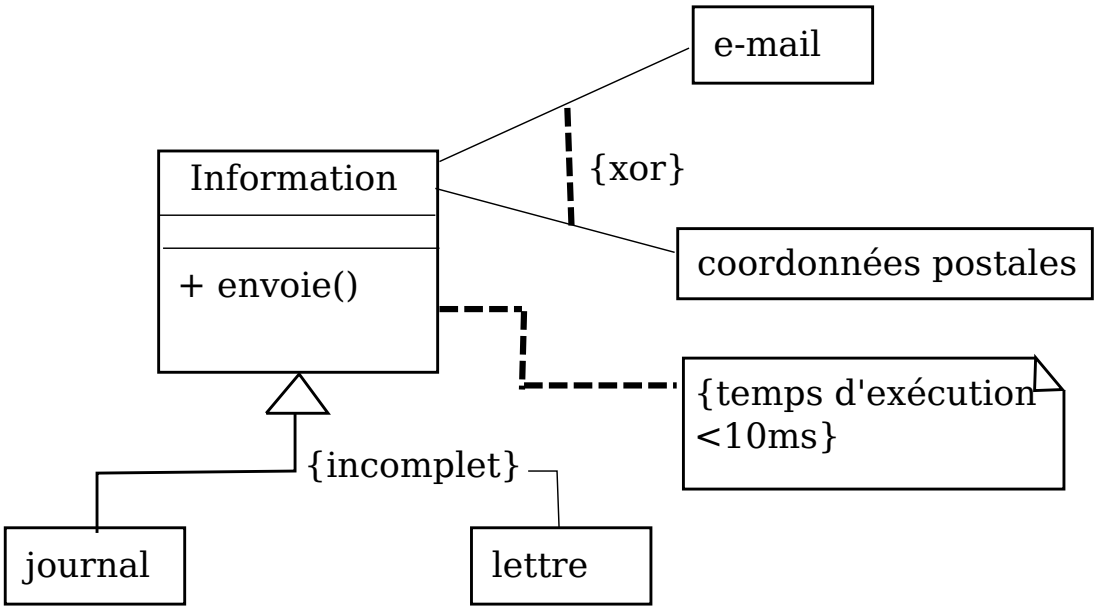
Département Informatique - Polytech Lyon
Université Claude Bernard Lyon 1
2012 - 2013

Plan du cours

Plan

Contraintes

- Relation entre éléments de modélisation : propriété qui doit être vraie
- Notation : $- - - - - - - - - \{contrainte\}$
  3 types de contraintes :
  - Contraintes prédéfinies : $disjoint$, $overlapping$, xor, ...
  - Contraintes exprimées en langue naturelle (commentaires)
  - Contraintes exprimées avec OCL (Object Constraint Language)
- Stéréotypes : ≪précondition≫, ≪postcondition≫

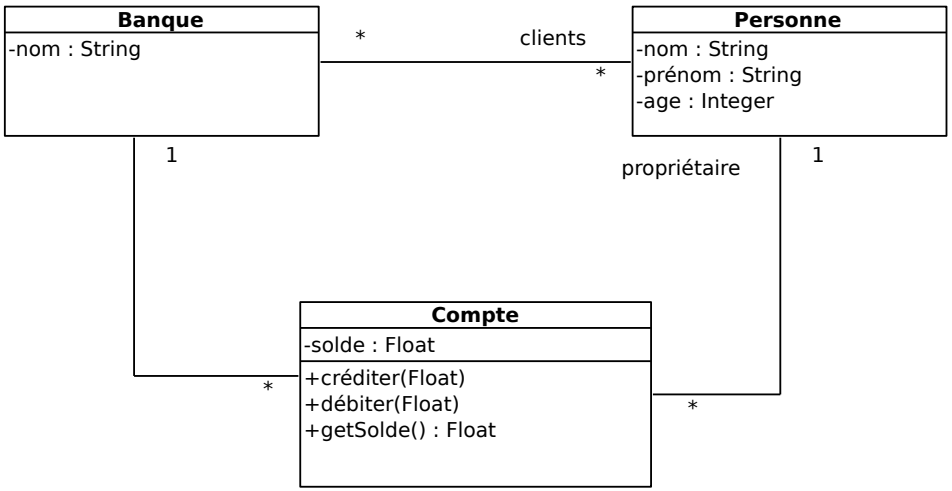


Pourquoi OCL ? Illustration par l'exemple

Application bancaire

Spécification

- 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



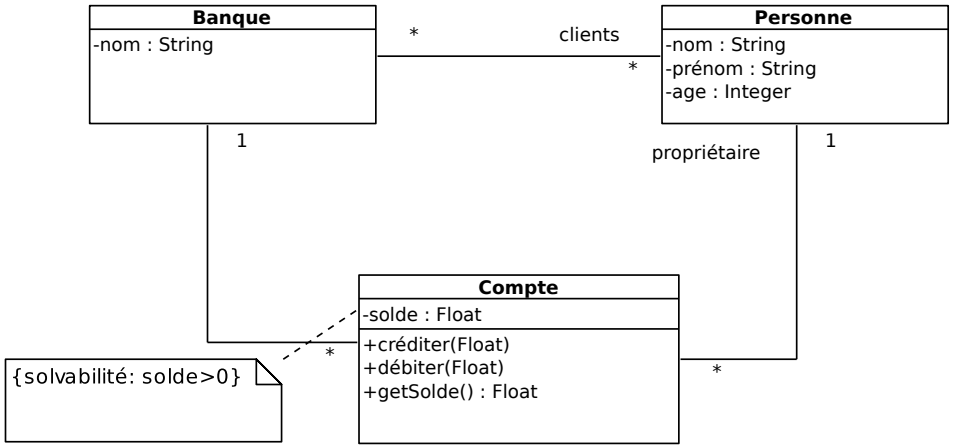
Quelles spécifications ne sont exprimées ?

Pourquoi OCL ? Illustration par l'exemple

Application bancaire

Spécification

- 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



Le diagramme de classe ne permet pas d'exprimer tout ce qui est défini dans la spécification

Pourquoi OCL ? Illustration par l'exemple

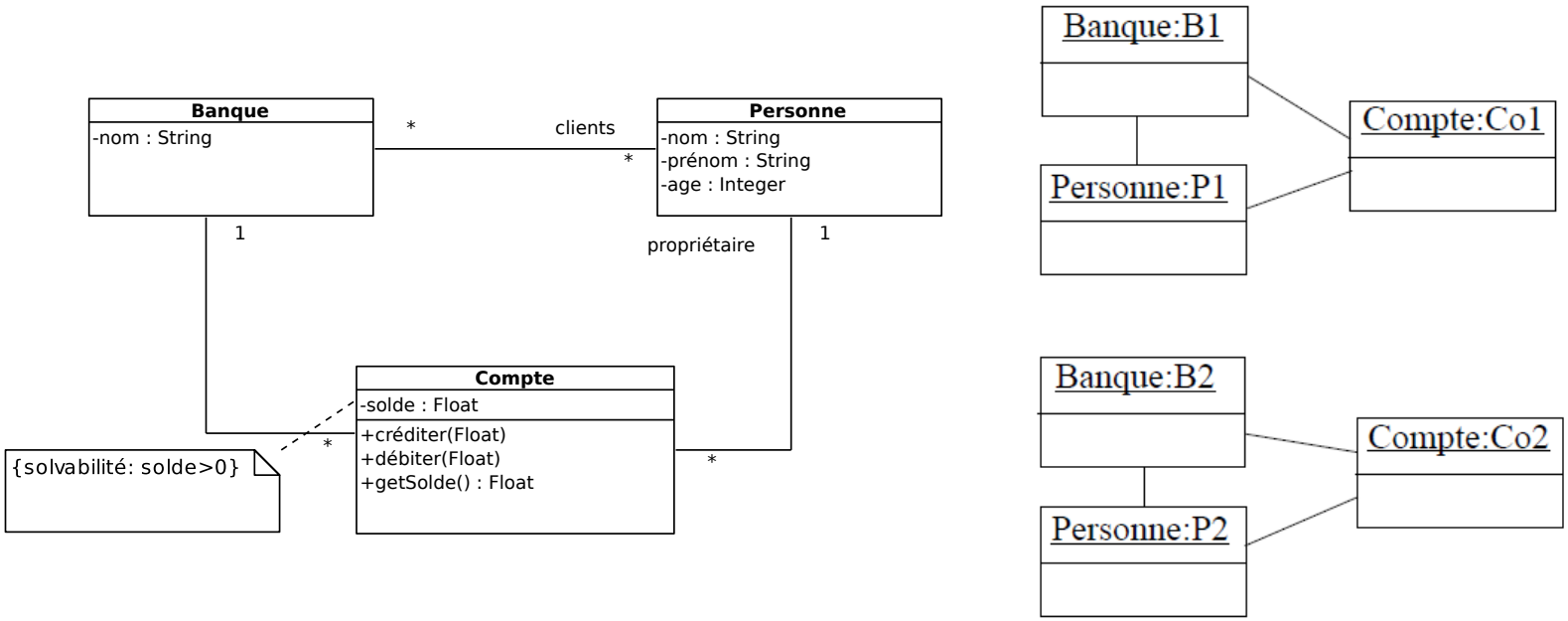
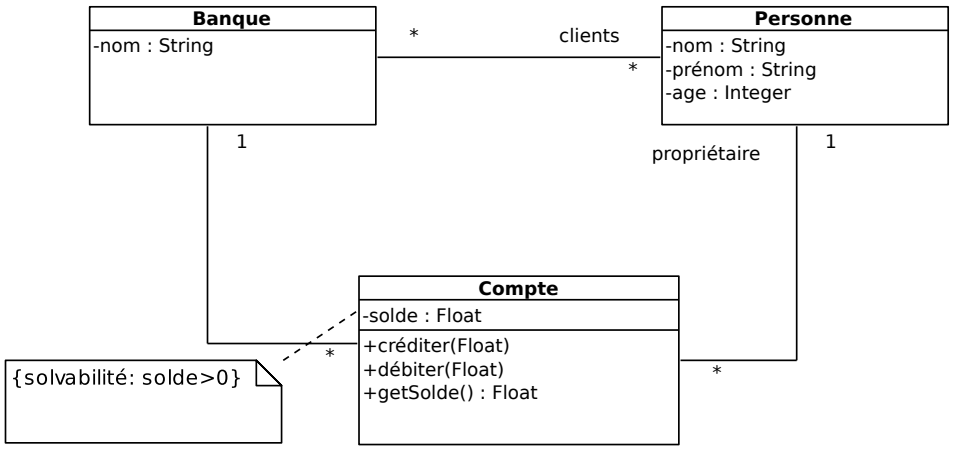


Diagramme d'objets cohérent avec le diagramme de classes et la spécification.

Pourquoi OCL ? Illustration par l'exemple



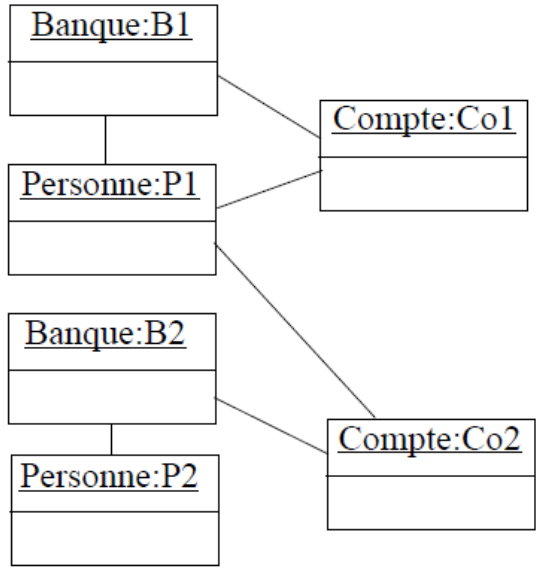


Diagramme d'objets cohérent avec le diagramme de classes mais qui ne respecte pas la spécification attendue.

- 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

OCL - Object Constraint Language

- Langage de description de contraintes d'UML
- Langage formel (mais simple à utiliser), non ambigü, grammaire précise
- Spécifier des information supplémentaires importantes qui n'ont pas pu l'être avec les éléments de modélisation d'UML
- Standardisé par l'OMG : spécification v2.0
  `http://www.omg.org/spec/OCL/`

Utilisation

Principe

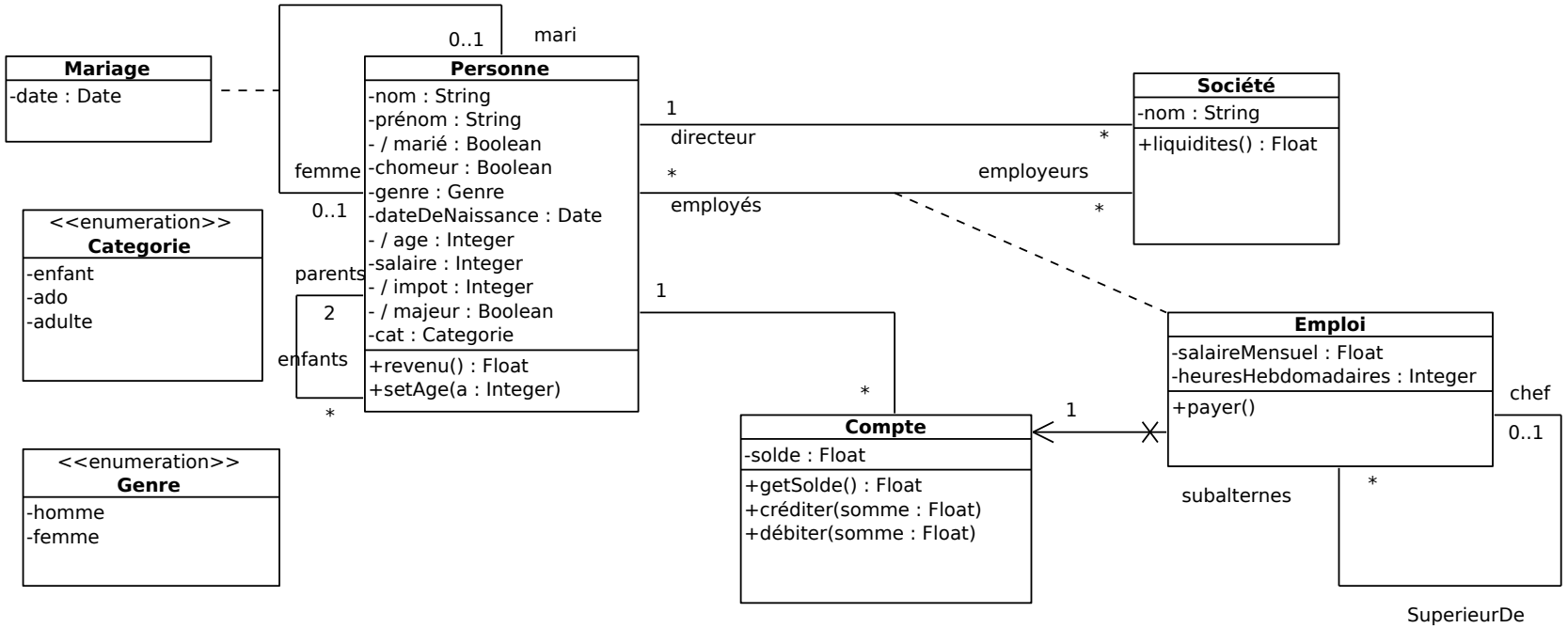
- Langage déclaratif : les contraintes ne sont pas opérationnelles.
- Langage sans effet de bord : les instances ne sont pas modifiées par les contraintes.
- Une expression OCL décrit une contrainte à respecter et pas une implémentation d'opération.
- OCL peut s'appliquer sur la plupart des diagrammes UML

Utilisation

Les contraintes OCL servent dans plusieurs situations :

- description d'invariants de classe
  - Une contrainte qui doit être respectée en permanence par toutes les instances d'une classe
- pré-condition et post-conditions à l'exécution d'une opération
  - Une contrainte qui doit être toujours vraie avant/après l'exécution d'une opération
- contraintes sur la valeur retournée par une méthode
- initialisation d'un attribut
- règles de dérivation des attributs
  - Préciser la valeur d'un attribut dérivé
- expression des gardes (conditions dans les diagrammes dynamiques)
- spécification du corps d'une opération
  - spécification d'une opération sans effet de bord (query)
- ...

Exemple de modèle



Plan

Notion de contexte

- Toute contrainte OCL est liée à un contexte spécifique, l'élément auquel la contrainte est attachée.
- Syntaxe :
  `context` monContexte
  `<stéréotype>` nomContrainte : Expression de la contrainte
- Le contexte peut être utilisé à l'intérieur d'une expression grâce au mot-clef `self`
- Exemple :
  - `context Compte` : l'expression OCL s'applique à la classe `Compte`, c'est-à-dire à toutes les instances de cette classe
  - `context Compte::getSolde() : Real` : l'expression OCL s'applique à la méthode `getSolde()`
  - `context Compte::créditer(somme : Real)` : l'expression OCL s'applique à la méthode `créditer`
  - `context Compte:solde : Real` : l'expression OCL s'applique à l'attribut `solde`

Invariants

- Un invariant exprime une contrainte sur un objet ou un groupe d'objets qui doit être respectée en permanence
- <stéréotype> : inv

```
context Compte
inv: solde > 0 -- solde d'un compte doit toujours être positif
context Personne
inv: (age <= 140) and (age >=0) -- l'âge est compris entre 0 et 140 ans
context Personne
inv : (self.age <= 140) and (self.age >=0)
context p:Personne
inv borneAge: (p.age <= 140) and (p.age >=0)
```

- . permet d'accéder à une caractéristique d'un objet (méthode, attribut, terminaison d'association)
- *self* objet désigné par le contexte

Pré-conditions & Post-conditions

- Spécifie une contrainte qui doit être vérifiée avant/après l'appel d'une opération
  - Contraindre l'ensemble des valeurs d'entrée d'une opération (`pre`)
  - Spécifier la sémantique d'une opération : ce qu'elle fait et non comment elle le fait (`post` concerne les effets de bord (modification d'un argument, d'un autre membre de la classe, etc.))
- `<stéréotype>` : `pre post`

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

Pré-conditions & Post-conditions

- Spécifie une contrainte qui doit être vérifiée avant/après l'appel d'une opération
  - Contraindre l'ensemble des valeurs d'entrée d'une opération (`pre`)
  - Spécifier la sémantique d'une opération : ce qu'elle fait et non comment elle le fait (`post` concerne les effets de bord (modification d'un argument, d'un autre membre de la classe, etc.))
- `<stéréotype>` : `pre post`
- Opérateurs spéciaux :
  `@pre` : accès à la valeur d'une propriété avant l'exécution de l'opération
  `result` : accès au résultat de l'opération

```
context Compte::débiter(somme : Real)
pre: somme > 0
post: solde = solde@pre - somme
context Compte::getSolde() : Real
post: result = somme
```

Retour d'une opération

- Spécifie le corps d'une opération
- Définit directement le résultat d'une opération
- <stéréotype> : body

```
context Compte::getSolde() : Real
body : solde
```

Valeur initiale et dérivée d'un attribut

- Spécifier la valeur initiale ou dérivé d'un attribut
- <stéréotype> : `init derive`

```
context Personne::age :entier
init : 0
context Personne::age : entier
derive : dateDeNaissance.getYear() - Date::current().getYear()
```

Définition de variables et d'opérations

- Définir des variables pour simplifier l'écriture de certaines expressions
- syntaxe : `let` variable : type = expression1 `in` expression2

```
context Personne
inv : let montantImposable : Real = 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 et d'opérations

- Définir des variables pour simplifier l'écriture de certaines expressions
- syntaxe : `let` variable : type = expression1 `in` expression2
- Définir des variables/opérations utilisables dans plusieurs contraintes de la classe
- syntaxe : `def` variable : type = expression1

```
context Personne
def : montantImposable : Real = salaire*0.8

context Personne
def : ageCorrect(a :Real) :Boolean = (a>=0) and (a<=140)
```

on peut réécrire l'invariant sur l'age :

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

Plan

Conception par contrats

Définition

- La programmation par contrat est un paradigme de programmation dans lequel le déroulement des traitements est régi par des règles ou contraintes appelées *assertions* ou contrats
- Le contrat est passé entre l'appelant d'une opération et l'appelé (celui qui exécute l'opération) par l'intermédiaire des contraintes
- Le but principal est de réduire le temps de débogage des programmes
- Librairie Java `jContractor`

Conception par contrats

3 types d'assertion

- Invariants, pré et postconditions permettent de faire une conception par contrat
  - Si l'appelant respecte les contraintes de la précondition alors l'appelé s'engage à respecter la post-condition
  - Si l'appelant ne respecte pas la précondition, alors le résultat de l'appel est indéfini
- Tester la satisfaction des assertions :
  $\rightarrow$ Avant l'appel de la méthode : confiance dans l'utilisateur
  $\rightarrow$ Dans le corps de la méthode : lever une exception
  $\rightarrow$ Précompilation
  $\rightarrow$ Outils Java 1.5 pour mettre en place des assertions : le mot-clé `assert`

- Pour l'exemple précédent : Si l'appelant de l'opération `débiter` passe une somme positive en paramètre, alors le compte est bien débité de cette somme

Plan

Types de base et opérations associées

Type	Exemples de valeurs	Opérateurs
Boolean	true ; false	and ; or ; xor ; not ; implies ; if-then-else-endif ; ...
Integer	1 ; -5 ; 2 ; 34 ; 26524 ; ...	* ; + ; - ; / ; abs() ; ...
Real	1,5 ; 3,14 ; ...	* ; + ; - ; / ; abs() ; floor() ; ...
String	“To be or not to be ...”	concat() ; size() ; substring() ; ...

Types et opérateurs prédéfinis dans les contraintes OCL.

Opérateurs conditionnels

Certaines contraintes sont dépendantes d'autres contraintes

- `if` expr1 `then` expr2 `else` expr3 `endif`

  Si l'expression `expr1` est vraie alors `expr2` doit être vraie sinon `expr3` doit être vraie

- 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
majeur=vrai
 else majeur=faux endif
```

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

Types de base et opérations associées

Définir plusieurs contraintes pour un invariant, une pré ou postcondition :

- `and` : « et logique » : l'invariant, pré ou postcondition est vrai si toutes les expressions reliées par le `and` sont vraies
- `or` et `xor`

Types énumérés

Utilisation d'une valeur d'une énumération

- `NomEnum::valeur`
- Ancienne notation : `#valeur`

```
context Personne
inv : if age <=12 then cat =Catégorie::enfant
else if age <=18 then cat =Catégorie::ado
else cat=Catégorie::adulte
endif endif
context Personne
inv: genre = Genre::femme
```

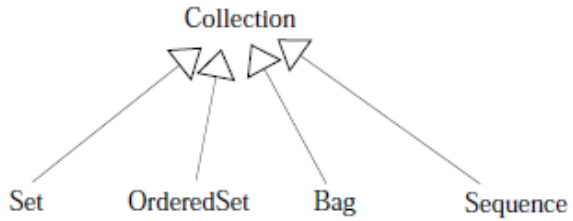
Opérations prédéfinies sur les objets

Types reliés par une relation de spécialisation

Prise en compte des spécialisations entre classes du modèle avec opérations OCL dédiées à la gestion des types :

- `oclIsKindOf(type)` : vrai si l'objet est du type `type` ou un de ses sous-types
- `oclIsTypeOf(type)` : vrai si l'objet est du type `type`
- `oclAsType(type)` : l'objet est « casté » en type `type`
- Exemple : une `classeB` hérite d'une `classeA` :
  - dans le contexte de `classeB`, `self.oclIsKindOf(ClasseB)` et `self.oclIsKindOf(ClasseA)` sont vraies ;
  - dans le contexte de `classeA`, `self.oclIsKindOf(ClasseB)` est fausse ;
  - dans le contexte `Société`, `directeur.oclIsTypeOf(Personne)` est vraie alors que `self.oclIsTypeOf(Personne)` est fausse.

Types Collection



Plusieurs sous-types du type abstrait Collection

- Set : ensemble non ordonné d'éléments uniques
- OrderedSet : ensemble ordonné d'éléments uniques ({`ordered`})
- Bag : collection non ordonnée d'éléments identifiables
- Sequence : collection ordonnée d'éléments identifiables.

Plan

Accès aux objets et navigation

Dans une contrainte OCL associée à un objet, on peut

- Accéder à l'état interne de cet objet (ses attributs) : `self.attribut`
- Accéder aux opérations de cet objet : `self.operation`
- Naviguer le long des liens : accéder de manière transitive à tous les objets ou groupe d'objets (et leur état) avec qui l'objet désigné par le contexte est en association
- syntaxe : nom de la classe associée avec minuscule (si pas d'ambiguïté) OU nom du rôle
- ambiguité $\rightarrow$ multiples associations entre objet désigné par le contexte et objet associé

  $\rightarrow$ association empruntée est réflexive

```
context Société
inv: self.personne.age >= 50 --ambigue
inv: self.directeur.age >= 50
```

Accès aux objets et navigation

Type du résultat

Objet référencé instance de la classe `X` et multiplicité de l'association du côté de cette classe X est :

- 1 : le type du résultat est `X` ;
- $*$ ou $O..n$: le type du résultat est `Set(X)` ;
- $*$ ou $O..n$ avec contrainte $\{$`ordered`$\}$: le type du résultat est `OrderedSet(X)` ;
- si l'on emprunte plusieurs associations/propriétés : le type du résultat peut être `Bag`
- Exemple : Dans le contexte de la classe `Société` :
  - `directeur` de type `Personne` ;
  - `employés` de type `Set(Personne)` ;
  - `employés.compte` de type `Bag(Compte)` ;
  - `employés.dateDeNaissance` de type `Bag(Date)`.

```
context Personne
inv: self.compte.solde->sum() > 0 -- résultat de type collection
```

Accès aux objets et navigation

Dans une contrainte OCL associée à un objet, on peut

- Naviguer vers une classe association : nom de la classe association en minuscule
- Naviguer à partir d'une classe association : nom des rôles (résultat est toujours un objet unique)

```
context Société
inv: self.emploi.salaireMensuel -- salaires de tous les employés
context Personne
inv: self.mariage[femme].date -- dates de mariage de toutes les femmes
--préciser par quelle extremité on emprunte l'association avec nom du
role entre crochet
context Emploi
inv: self.employés.age >= 18 -- produit un singleton
```

Plan

Opérations sur les Collections

- size() : retourne le nombre d'éléments de la collection
- count(obj) : retourne le nombre d'occurence d'un objet dans la collection
- sum() : addition de tous les éléments de la collection
- exists(uneExpression) : vaut vrai si et seulement si au moins un élément de la collection satisfait `uneExpression`.
- isEmpty()/notEmpty() : retourne vrai si la collection est/n'est pas vide (tester l'existence d'un lien dans une association dont la multiplicité inclut 0)
- Notation : `collection->operation()`

```
context Société
inv : self.employés->notEmpty() --une société a au moins un employé
context Société
inv : self.directeur->size()=1 --une société possède exactement un
directeur
context Société
inv : self.employés->exists(age>50) --une société doit posséder au moins
une personne de + de 50 ans
context Société
inv : not(self.employés->exists(age<18)) --n'existe pas d'employés dont
l'age <18
```

Opérations sur les Collections

- includes(obj)/excludes(obj) : vrai si la collection inclut/n'inclut pas l'objet `obj`
- including(obj)/excluding(obj) : la collection référencée doit être cette collection en incluant/excluant l'objet `obj`
- includesAll(ens)/excludesAll(ens) : la collection contient tous/ne contient aucun des éléments de la collection `ens`
- AllInstances() : retourne toutes les instances de la classe référencée
- forAll(elem :T|uneExpression) : vaut vrai si et seulement si `uneExpression` est vraie pour tous les éléments de la collection.
- Notation : `collection->operation()`

```
context Société
inv : self.employes->includes(self.directeur) --le directeur est
également un employe
context Personne
inv: Personne.allInstances() -> forAll(p1, p2 |
 p1 <> p2 implies p1.nom <> p2.nom)
```

Opérations sur les Collections

- union : Retourne l'union de deux collections
- intersection : Retourne l'intersection de deux collections
- Exemple :
  - `(col1 -> intersection(col2)) -> isEmpty()` : Renvoie vrai si les collections `col1` et `col2` n'ont pas d'élément en commun
  - `col1 = col2 -> union(col3)` : La collection `col1` doit être l'union des éléments de `col2` et de `col3`

- Générer une sous-collection en filtrant les éléments de la collection `self` :
  - select : retourne les éléments de la collection `self` dont les éléments respectent la contrainte spécifiée
  - reject : retourne tous les éléments de la collection `self` excepté ceux qui satisfont la contrainte spécifiée

```
context Société
inv: self.employés->select(age > 50)-> notEmpty()
context Personne
inv: compte -> select( c |c.solde > 1000)
inv: compte -> reject( c |c.solde > 1000)
```

Plan

Conclusion

Conseils de modélisation

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

Outils

- USE (Mark Richters) UML-based Specification Environment http ://sourceforge.net/projects/useocl/
- Octopus (Warmer & Kleppe) http ://octopus.sourceforge.net/
- Dresden OCL http ://dresden-ocl.sourceforge.net/