Notes de cours sur OCL
(*Object Constraint Language*)

INF3140
Hiver 2010

Table des matières

OCL : Chapitre I

Propositions, prédicats et quantificateurs

0. Introduction

Objectif : Présenter certains éléments de base de la notation OCL pour la **logique**

Note : Rappels de notions vues en INF1130 (Mathématiques pour informaticien)

1. Propositions = Valeurs et expressions booléennes

- Constantes (opérateurs 0-aires) :

  `true`, `false`

- Opérateur unaire :

  `not` (négation = non)

- Opérateurs binaires :

`and`	(conjonction = et)
`or`	(disjonction = ou)
`xor`	(ou exclusif)
`implies`	(implication)
`=`	(égalité)
`<>`	(inégalité)

- Opérateur ternaire (résultat de type `T`) :

  `if` *`Boolean`* `then T else T endif`

Priorité des opérateurs (ordre décroissant), en incluant les opérateurs arithmétiques :

```
not, -
*, /
+, -
if_then_else_endif
<, <=, >, >=
=, <>
and, or, xor
implies
```

Propriété d'évaluation partielle de certains opérateurs (*short-circuit*, appelé aussi *évaluation paresseuse*) :

- `(true or x) = true` — peu importe `x`
- `(false and x) = false` — peu importe `x`
- `(if true then x else y endif) = x` — peu importe `y`
- `(if false then x else y endif) = y` — peu importe `x`

Exemples :

```
(2 = 2 or false) = true
(3 = 3 or 2 = 1/0) = true
if 2 = 3 then (2/0).round() else 2+3 endif = 5
```

Tables de vérité = sémantique des opérateurs

P_1	P_2	P_1 `or` P_2
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

P_1	P_2	P_1 `xor` P_2
`true`	`true`	`false`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

P_1	P_2	P_1 `implies` P_2
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`true`
`false`	`false`	`true`

P_1	`if` P_1 `then` E_1 `else` E_2 `endif`
`true`	E_1
`false`	E_2

Interprétation de l'implication (`implies`)

Une implication sert à décrire une forme de *règle conditionnelle*, où la partie gauche (avant `implies`) indique la condition (appelée aussi «antécédent») ; la valeur de vérité de l'implication est alors déterminée en vérifiant *si la règle est respectée*.

Un étudiant a 100 % dans tous ses travaux et examens `implies` *il obtient A+ pour le cours*

= *«Si un étudiant a 100 % dans tous ses travaux et examens alors il obtient A+ pour le cours»*

Fait : Un étudiant peut avoir A+ même s'il n'a pas 100 % (`false implies true = true`)

L'alarme d'incendie sonne `implies` *on doit utiliser les escaliers*

= *«Si l'alarme d'incendie sonne alors on doit utiliser les escaliers»*

Fait : On peut prendre les escaliers même si l'alarme ne sonne pas

Remarque :
`(p implies q)` = `(if p then q else true endif)`

Équivalences logiques

`p and true`	$\equiv$	`p`	Identité
`p or false`	$\equiv$	`p`	
`p and false`	$\equiv$	`false`	Dominance
`p or true`	$\equiv$	`true`	
`p and p`	$\equiv$	`p`	Idempotence
`p or p`	$\equiv$	`p`	
`p and q`	$\equiv$	`q and p`	Commutativité
`p or q`	$\equiv$	`q or p`	
`not(not p)`	$\equiv$	`p`	Double négation
`(p and q) and r`	$\equiv$	`p and (q and r)`	Associativité
`(p or q) or r`	$\equiv$	`p or (q or r)`	
`p and (q or r)`	$\equiv$	`(p and q) or (p and r)`	Distributivité
`p or (q and r)`	$\equiv$	`(p or q) and (p or r)`	
`not(p and q)`	$\equiv$	`not p or not q`	De Morgan
`not(p or q)`	$\equiv$	`not p and not q`	
`p and not p`	$\equiv$	`false`	Contradiction
`p or not p`	$\equiv$	`true`	Tiers exclu
`p implies q`	$\equiv$	`not p or q`	

2. Prédicats, formes propositionnelles et quantificateurs

Prédicats et formes propositionnelles

Prédicat = *Fonction* qui retourne un résultat booléen

Exemples :

- ... `<=` ...
- ... est plus agé que ...
- ... est le fils de ... et ...

Forme propositionnelle = Énoncé logique (valeur booléenne) qui réfère à une ou plusieurs variables

Exemples :

- `x >= 0`
- `x <= y`
- `x` est plus agé que `y`
- `u` est le fils de *Jean* et `v`

Fait : Une forme propositionnelle (sans valeur de vérité) devient une *proposition* (avec valeur de vérité) lorsque les variables deviennent liées à des valeurs

Exemples d'expressions OCL (incomplètes) correspondant à des formes propositionnelles :

```
( x | x >= 0 )

( x: Integer | x >= 0 )


( x, y | x <= y )


( x: Personne, y: Personne | x.plusAgeQue(y) )


( u, v | u.estFilsDe(JEAN, v) )
```

Donc : Le type d'un identificateur peut être indiqué comme il peut être omis — dans ce dernier cas, c'est le contexte (collection sur laquelle s'applique la forme propositionnelle) qui détermine le type de l'identificateur.

Quantificateurs logiques

- Pour obtenir une proposition à partir d'une forme propositionnelle, il faut contraindre (on dit aussi "lier", en englais "*bind*") les variables.
- Divers quantificateurs logiques permettent de produire une proposition à partir d'une forme propositionnelle :
  - Quantificateur *universel* ⇒ tous les éléments ont la propriété :

    `forAll(` *forme propositionnelle* `)`
  - Quantificateur *existentiel* ⇒ il existe au moins un élément (mais il peut y en avoir plusieurs) qui a la propriété :

    `exists(` *forme propositionnelle* `)`
- Un quantificateur s'applique nécessairement à une **collection** :

```
col1->forAll( ... )
col2->exists( ... )
```

- Les principales collections OCL (chapitre ultérieur) :
  - `Set` = Ensemble
  - `Bag` = Sac (multi-ensemble)
  - `Sequence` = Séquence (liste)

Donc, la **forme générale** d'une expression logique avec quantificateur *quantif* sur une collection *coll* pour une expression (forme propositionnelle) *expr* est l'une des suivantes :

- Avec un identificateur v sans spécification de type :

  coll->quantif(v | *expr*)

- Avec un identificateur v de type T :

  coll->quantif(v: T | *expr*)

Note : La collection *coll* peut être une valeur (littérale), un identificateur référant à une collection ou une expression produisant une collection.

Quelques exemples de propositions (toutes vraies) :

```
Set{0..9}->forAll( x | 0 <= x and x <= 9 )

Set{0..9}->forAll( x | 0 <= x and x <= 124 )


Set{0..9}->exists( x | 9 <= x and x <= 23 )

Set{0..9}->exists( x | 0 <= x and x <= 23 )


Set{0,2,4,6,8}->forAll( x | x.mod(2) = 0 )

not Set{1,3,5,7,9}->exists( x | x.mod(2) = 0 )
```

Remarques :

- Ces expressions sont exprimées dans la notation de l'outil `USE` (développé à l'Université de Bremen, Allemagne).
- L'outil `use` est disponible sur les machines `rayons` (`rayon1.labunix.uqam.ca` et `rayon2.labunix.uqam.ca`) et sera utilisé dans les labos et dans les devoirs.
- Toutes ces expressions retournent `true`, donc il est inutile de l'indiquer. Si `e` est une expression booléenne, alors :
  - "`e`" et "`e = true`" ont la même valeur de vérité ;
  - "`not e`" et "`e = false`" ont la même valeur de vérité.

```
// Quantificateurs imbriques.
Set{0..9}->forAll( x |
             Set{0,2,4,6,8}->exists(y | y = x)
             xor
             Set{1,3,5,7,9}->exists(y | y = x) )


// Applications sur l'ensemble vide.
oclEmpty(Set(Integer))->forAll( x | x < 0 )

not Set{1..0}->exists( x | x >= 0 )

not Set{}->exists( x: Integer | x >= 0 )
```

Remarque Dans la dernière expression, la déclaration du type de `x` **est obligatoire** :

```
use> ?not Set{}->exists( x | x >= 0 )
<input>:1:23: Undefined operation `OclVoid.>(Integer)'.
use> ?Set{}
-> Set{} : Set(OclVoid)
use> ?Set{}->including(1)
-> Set{1} : Set(Integer)
```



```
// Autre quantificateur = one : Il existe un,
// et un seul, element qui satisfait la propriete.
Set{1..10}->one( x | x = 2 )

not Set{1..10}->one( x | x.mod(2) = 1 )


// Quantificateurs universel et existentiel avec
// plusieurs (2) arguments.
Set{0..9}->exists( x, y | x <= y )

Set{0..9}->forAll( x, y | x < y implies x <> 9 )
```

Remarques sur l'interprétation des quantificateurs

- Soit une expression de la forme suivante représentant une quantification universelle sur un ensemble d'entiers :

  `Set{`$e_1, e_2, \ldots, e_n$`}->forAll( x: Integer |` p`(x) )`

  Signification :

  $p(e_1)$ `and` $p(e_2)$ `and` $\ldots$ `and` $p(e_{n-1})$ `and` $p(e_n)$

  Or :

  $p(e_1)$ `and` $p(e_2)$ `and` $\ldots$ `and` $p(e_{n-1})$ `and` $p(e_n)$
  `=`
  `true and` $p(e_1)$ `and` $p(e_2)$ `and` $\ldots$ `and` $p(e_{n-1})$ `and` $p(e_n)$

  Donc :

  `Set{}->forAll( x: Integer |` p`(x) )`

- Soit une expression de la forme suivante représentant une quantification existentielle sur un ensemble d'entiers :

  `Set{`$e_1, e_2, \ldots, e_n$`}->exists( x: Integer |` p`(x) )`

  Signification :

  $p(e_1)$ `or` $p(e_2)$ `or` $\ldots$ `or` $p(e_{n-1})$ `or` $p(e_n)$
  `=`
  `false or` $p(e_1)$ `or` $p(e_2)$ `or` $\ldots$ `or` $p(e_{n-1})$ `or` $p(e_n)$

  Donc :

  `not Set{}->exists( x: Integer |` p`(x) )`

3. Introduction à l'outil `use`

L'outil `use` *(A UML-based Specification Environment)* permet d'analyser et manipuler des spécifications OCL.
Exemples simples **d'utilisation interactive** :

```
$ use -nogui
use version 2.5.0, Copyright (C) 1999-2009 University of Bremen
use> ? Set{0..9}
-> Set{0,1,2,3,4,5,6,7,8,9} : Set(Integer)
use> ? Set{0..9}->forAll( x | x <= 10 )
-> true : Boolean
use> ? Set{1..0}->forAll( x | x = 2 )
-> true : Boolean
use> \
> ? let pairs: Set(Integer) = Set{0,2,4,6,8} in
> pairs->forAll( x | x. mod(2) = 0 )
> -> true : Boolean
use> \
> ? let pairs: Set(Integer) = Set{0,2,4,6,8} in
> let impairs: Set(Integer) = Set{1,3,5,7,9} in
> pairs->forAll( x | impairs->exists( y | x = y ) )
> -> false : Boolean
```

Remarques :

- L'opération `use` «?» permet d'évaluer une expression.
- L'opération `use` «\» permet d'évaluer une expression écrite sur plusieurs lignes — terminée avec «^D».
- L'expression OCL «`let... in...`» permet d'introduire un identificateur local, en déclarant son type et sa **valeur**.

OCL : Chapitre II

Les types de base et les collections OCL

0. Introduction

Dans ce chapitre, nous allons voir les principaux types disponibles dans le langage OCL : types de base et collections.

Comme on le verra plus tard, les collections serviront à modéliser les relations (liens) entre entités pour lesquelles la multiplicité est supérieure à 1.

Le langage OCL fournit quatre types de collections :

- `Set`
- `Bag`
- `Sequence`
- `OrderedSet`

Ces identificateurs peuvent être utilisés comme *constructeurs de types* (ici, à droite de « : ») — puisqu'on peut indiquer un nouveau type en fournissant un type approprié en argument — mais aussi comme *constructeurs de valeurs* (à gauche de « : ») :

```
Set{true}: Set(Boolean)
Set{10,20,30}: Set(Integer)
Set{Set{'a','c'},Set{'xxx',''}}: Set(Set(String))
Set{Bag{1,1,1},Bag{2,3},Bag{2..1}}: Set(Bag(Integer))
```

Remarque : Dans ce qui suit, nous nous restreindrons aux trois premiers types de collections, qui sont les seuls disponibles dans l'outil `USE`.

1. Les types de base

Les types de base en OCL sont les suivants :

- `Boolean`
- `Integer`
- `Real`
- `String`

On a déjà vu les types `Boolean` et `String` («Propositions, prédicats et quantificateurs», Exercice #1).

Les types `Integer` et `Real` possèdent les opérateurs habituels :
`=, <>, <, >, <=, >=, +, -, *, /`

Les méthodes suivantes sont aussi disponibles :

`a.mod(b)`	Reste de la division entière (résultat `Integer`)
`a.div(b)`	Division entière (résultat `Integer`)
`a.abs()`	Valeur absolue
`a.max(b)`	Maximum parmi `a` et `b`
`a.min(b)`	Minimum parmi `a` et `b`
`a.round()`	Valeur arrondie (résultat `Integer`)
`a.floor()`	Valeur tronquée (résultat `Integer`)

Notation pour les propriétés et appels de méthodes

- `x.mthd(...)` : Méthode (ou propriété) d'un **objet** `x`.
- `c->mthd(...)` : Méthode (ou propriété) d'une **collection** `c`.

2. Opérations applicables à toutes les collections vs. à certaines collections seulement

De nombreuses opérations s'appliquent à n'importe quel type de collections, par exemple :

- `size()`
- `isEmpty()`
- `notEmpty()`
- `count( elem )`

- `includes( elem )`
- `includesAll( col )`
- `excludes( elem )`
- `excludesAll( col )`

- `including( elem )`
- `excluding( elem )`
- `union( col )`

- `exists( ... )`
- `forAll( ... )`
- `one( ... )`
- `isUnique( ... )`

- `any( ... )`

- `select( ... )`
- `reject( ... )`
- `collect( ... )`
- `iterate( ... )`

D'autres opérations ne s'appliquent que sur des collections avec des éléments d'un type particulier, par exemple :

- `sum()` : collections de nombres.
- `sortedBy( ... )` : collections d'éléments pouvant être comparés entre eux (avec «<»).
- `flatten()` : collections dont les éléments sont eux-mêmes *des collections*.

Finalement, certaines opérations ne s'appliquent que sur un type particulier de collections, par exemple, les opérations suivantes ne s'appliquent que sur des `Sequences` : `at()`, `first()`, `last()`, `prepend()`, `append()`, `subSequence()`.

Remarque : Dans le présent document, certaines opérations ne sont pas nécessairement illustrées pour chacun des types de collection. Toutefois, une fois qu'on a compris ce que fait une opération pour un type de collection, il est généralement clair ensuite ce que signifie cette opération dans le contexte d'un autre type de collections.

3. Les ensembles

3.1 Principale caractéristique

L'ordre et le nombre d'occurrence ne sont pas significatifs.

```
Set{1,1,1} = Set{1}

Set{1..2} = Set{2,1}

Set{1,2,1,1..2} = Set{2,1}
```

3.2 Opérations

- `s->size()` : Retourne le nombre d'éléments (distincts) de l'ensemble `s`

```
Set{1}->size() = 1

Set{1,1,1}->size() = 1

Set{1,2,1}->size() = 2
```

- `s->including( x )` : Retourne un nouvel ensemble qui inclut tous les éléments de `s` en plus, possiblement, de `x`.

```
Set{1,2}->including(1) = Set{1,2}

Set{1,2}->including(3)->including(4) = Set{1..4}
```

- `s->union( s2 )` : Retourne un nouvel ensemble qui contient tous les éléments de `s` ainsi que tous ceux de `s2`.

```
Set{0..3}->union( Set{2..5} ) = Set{0..5}
```


- `s->intersection( s2 )` : Retourne un nouvel ensemble qui contient tous les éléments qui sont dans `s` **et** qui sont **aussi** dans `s2`.

```
Set{1..3}->intersection( Set{-1..2} ) = Set{1..2}
```

- `s - s2` : Retourne un nouvel ensemble qui contient tous les éléments qui sont dans `s` mais qui ne sont pas dans `s2`.

```
Set{1..100} - Set{-10..20} = Set{21..100}
```

- `s->isEmpty()` : Détermine si l'ensemble s est vide.
  `s->notEmpty()` : Détermine si l'ensemble s n'est pas vide.

```
Set{1..0}->isEmpty()

oclEmpty(Set(Integer))->isEmpty()

Set{}->isEmpty()

not Set{1,2}->isEmpty()

Set{1,2}->notEmpty()
```

- `s->includes( x )` : Détermine si l'ensemble `s` contient l'élément `x` — si `x` appartient à `s`.

```
Set{1..3}->includes( 2 )

not Set{1,2}->includes( 3 )

not Set{2..1}->includes(2)
```

- `s->includesAll( s2 )` : Détermine si l'ensemble `s` contient chacun des éléments de `s2` — si `s2` est un sous-ensemble de `s`.

```
Set{1..3}->includesAll( Set{1,2} )

not Set{1,2}->includesAll( Set{2,3} )
```


- `s->excluding( x )` : Retourne un ensemble avec les mêmes éléments que `s`, sauf possiblement pour `x`.

```
Set{1..10}->excluding(4) = Set{1..3}->union( Set{5..10} )
```

- `s->sum()` : Retourne la somme des éléments de `s`, qui doivent être des nombres.

```
Set{-10..10}->sum() = 0

Set{1..100}->sum() = 5050
```

- `s->count( x )` : Retourne le nombre d'occurrences de `x` dans `s` (0 ou 1!).

```
Set{-2..10}->count( 0 ) = 1

Set{1..10}->count( 0 ) = 0
```

- `s->select( p )` : Retourne l'ensemble des éléments de `s` qui satisfont le prédicat `p`.

```
Set{0..10}->select( x | x.mod(2) = 0 )
   = Set{0,2,4,6,8,10}

Set{0..10}->select( x | x.mod(2) = 0 and x >= 10 )
   = Set{10}

Set{0..10}->select( x | x.mod(2) = 0
                        and x >= 999 )->isEmpty()
```

- `s->reject( p )` : Retourne l'ensemble des éléments de `s` qui ne satisfont pas le prédicat `p`.

```
Set{0..10}->reject( x | x.mod(2) = 0 )
   = Set{1,3,5,7,9}

Set{0..100}->reject( x | x.mod(2) = 0 and x >= 100 )
   = Set{0..99}

Set{0..9}->reject( x | x.mod(2) = 0 and x >= 999 )
   = Set{0..9}
```

Remarque : Dans le cas de `select` et `reject`, on parle parfois d'opérations de **filtrage** — filtrage de sélection ou filtrage d'élimination (en fonction d'un prédicat).

- `s->any( p )` : Retourne *un élément arbitraire* (n'importe lequel) de l'ensemble `s` qui satisfait le prédicat `p` :

```
Set{1..100}->any( x | x = 100 ) = 100

Set{1..4}->any( y | y.mod(2) = 0 ) = 2
or
Set{1..4}->any( x | x.mod(2) = 0 ) = 4


Set{1..100}->any( e | e > 100 ).isUndefined
```

Note : L'opération `any` s'applique aussi bien aux ensembles qu'aux sacs ou aux séquences.

- `s->flatten()` : Transforme une collection de collections de `T` en une collection de `T`.

```
Set{Set{3,4},Set{4,5},Set{5,6},Set{6,7}}->flatten()
   = Set{3..7}

Set{Set{},Set{10,11,12},Set{12, 14}}->flatten()
   = Set{10,11,12,14}

Set{Set{Set{1..0},Set{10,12},Set{12, 14}},
    Set{Set{10}}}->flatten()
   = Set{Set{},Set{10},Set{12,14},Set{10,12}}
```

Remarque : Dans ce dernier exemple, la valeur «`Set{1..0}`» a été utilisée pour dénoter un ensemble vide. On aurait aussi pu utiliser simplement «`Set{}`».

Dans certains contextes, il n'est pas possible d'écrire directement et simplement un ensemble (ou sac, ou séquence) vide tel que «`Set{}`» parce que l'information de type est insuffisante. Exemple du chapitre précédent :

```
not Set{}->exists( x | x >= 0 )
```

Par contre, il est toujours possible d'utiliser `oclEmpty`, et ce en indiquant le type approprié, par exemple :

- `oclEmpty(Set(Integer))`
- `oclEmpty(Set(String))`
- `oclEmpty(Bag(Sequence(Integer)))`
- Etc.

4. Les sacs

4.1 Principale caractéristique

L'ordre n'est pas significatif mais le nombre d'occurrence l'est.

```
Bag{1,1,1} <> Bag{1}

Bag{1..2} = Bag{2,1}

Bag{1,1..2} = Bag{1,2,1}

Bag{1,2} <> Bag{2,1,1}

Bag{1,2,1,1,2} = Bag{1,1,1,2,2}

Bag{5..2} = oclEmpty(Bag(Integer))

Bag{5..2} = Bag{}
```


4.2 Opérations

- `s->size()` : Retourne le nombre d'éléments du sac `s`

```
Bag{1}->size() = 1

Bag{1,1,1}->size() = 3

Bag{1,2,1}->size() = 3
```

- `s->including( x )` : Retourne un nouveau sac qui inclut tous les éléments de `s` en plus de `x`.

```
Bag{1,2}->including(1) = Bag{1,1,2}

Bag{1,2}->including(3)->including(1) = Bag{1,1,2,3}
```

- `s->union( s2 )` : Retourne un nouveau sac qui contient tous les éléments de `s` ainsi que tous ceux de `s2`.

```
Bag{1,2,3}->union( Bag{0,1,2,4,5} )
    = Bag{0,1,1,2,2,3,4,5}
```

- `s->intersection( s2 )` : Retourne un nouveau sac qui contient tous les éléments qui sont dans **`s` et** qui sont aussi dans `s2`.

```
Bag{1,1,1,1,3}->intersection( Bag{1,1,3,3,5} )
   = Bag{1,1,3}
```

- `s->isEmpty()` : Détermine si le sac `s` est vide.

```
oclEmpty(Bag(Integer))->isEmpty

not Bag{1,2}->isEmpty()
```

- `s->includesAll( s1 )` : Détermine si le sac `s` contient chacun des éléments de `s2` (indépendamment du nombre d'occurrences).

```
Bag{1,2,3}->includesAll( Bag{1,2} )

Bag{1,2,3}->includesAll( Bag{1,2,2} )

not Bag{1,2}->includesAll( Bag{2,3} )
```

- `s->excluding( x )` : Retourne un sac avec les mêmes éléments que `s`, sauf possiblement pour `x` dont toutes les occurrences sont éliminées.

```
Bag{1,1,1,2,2,3}->excluding(1) = Bag{2,2,3}
```

- `s->sum()` : Retourne la somme des éléments de `s`, qui doivent être des nombres.

```
Bag{1,1,1,1}->sum() = 4

Bag{1..100}->sum() = 5050
```

- `s->asSet()` : Retourne l'ensemble des éléments distincts de `s`.

```
Bag{1,1,1,2,2}->asSet() = Set{1,2}
```


- `s->count( x )` : Retourne le nombre d'occurrences de `x` dans `s`.

```
Bag{1,1,1,2,2,3}->count( 1 ) = 3

Bag{1,1,1,2,2,3}->count( 3 ) = 1

Bag{1..100}->count( 0 ) = 0
```

- `s->collect( x | e )` : Retourne le sac des valeurs obtenues en évaluant l'expression `e` pour les divers éléments `x` de `s` — avec un aplatissement (`flatten`) implicite.

```
Bag{10,20,30}->collect( x | 2 * x + 1 )
   = Bag{21,41,61}

Bag{0..9}->collect( y | y.mod(2) )
   = Bag{0,1,0,1,0,1,0,1,0,1}

Bag{0..100}->collect( i | i.mod(10) )->asSet()
   = Set{0..9}

Bag{1..4}->collect( x | Bag{2..3}->collect( y | x + y ) )
   = Bag{3,4,4,5,5,6,6,7}
```

Note : L'opération `collect`, même lorsqu'appliquée à un ensemble, retourne un sac.

```
Set{10,20,30}->collect( x | 2 * x + 1 )
   = Bag{21,41,61}

Set{0..9}->collect( x | x.mod(2) )
   = Bag{0,1,0,1,0,1,0,1,0,1}
```

Note : La collection retournée en résultat du `collect` peut être d'un type différent.

```
Bag{'abc','def','','bonjour'}->collect( x | x.size() )
   = Bag{0,3,3,7}

Bag{1..3}->collect( x | x.mod(2) = 0 )
   = Bag{false,true,false}
```

Note : Dans certains cas, la déclaration de l'argument peut être omise.

```
Bag{'abc','def','','bonjour'}->collect( size() )
   = Bag{0,3,3,7}

Bag{1..3}->collect( mod(2) = 0 )
   = Bag{false,true,false}
```


- `s->isUnique( e )` : Retourne `true` si l'expression `e`, lorsqu'elle est évaluée pour chaque valeur de la collection `s`, retourne toujours *une valeur distincte*.

```
Bag{1,2,3,4}->isUnique( x | x )

Bag{1,2,3,4}->isUnique( y | 2 * y )

not Bag{1,2,3,4}->isUnique( mod(2) )

not Bag{1,2,2,3}->isUnique( z | z )

Bag{ Bag{1}, Bag{2,3}, Bag{} }->isUnique( size() )

not Bag{ Bag{1}, Bag{1,2,3}, Bag{1,2} }
       ->isUnique( excluding(2) )
```

Note :

```
s->isUnique( e )
  = ( s->size() = s->collect( e )->asSet()->size() )
```

- `s->collectNested( e )` : Retourne le sac des valeurs obtenues en évaluant l'expression `e` pour les divers éléments `x` de `s`, mais en préservant la structure si ces valeurs sont elles-mêmes des collections, donc sans aplatissement implicite comme avec `collect`.

```
Bag{1..3}->collect( x | Bag{x} )
  = Bag{1..3}

Bag{1..3}->collectNested( x | Bag{x} )
  = Bag{Bag{1}, Bag{2}, Bag{3}}


Bag{1..4}->collect( x | Bag{x.mod(2)} )
  = Bag{0,0,1,1}

Bag{1..4}->collectNested( x | Bag{x.mod(2)} )
  = Bag{Bag{0},Bag{0},Bag{1},Bag{1}}
```


5. Les séquences

5.1 Principale caractéristique

L'ordre et le nombre d'occurrences sont significatifs.

```
Sequence{1,1,1} <> Sequence{1}

Sequence{1,2} <> Sequence{2,1}

Sequence{1,1,2} <> Sequence{1,2,1}

Sequence{1,2,1,1,2} <> Sequence{1,1,1,2,2}
```

5.2 Opérations

- `s->size()` : Retourne le nombre d'éléments de la séquence `s`.

```
Sequence{1}->size() = 1

Sequence{1,1,1}->size() = 3

Sequence{1,2,1}->size() = 3
```

- `s->prepend( x )` : Retourne une nouvelle séquence qui inclut tous les éléments de `s` en plus de `x` ajouté au début.

```
Sequence{1,2}->prepend(1)
   = Sequence{1,1,2}

Sequence{1,2}->prepend(3)->prepend(1)
   = Sequence{1,3,1,2}
```

- `s->append( x )` : Retourne une nouvelle séquence qui inclut tous les éléments de `s` en plus de `x` ajouté à la fin.

```
Sequence{1,2}->append(1)
   = Sequence{1,2,1}

Sequence{1,2}->append(3)->append(1)
   = Sequence{1,2,3,1}
```

- `s->including( x )` : Identique à `s->append( x )`.
- `s->union( s2 )` : Retourne une nouvelle sequence qui contient tous les éléments de `s` suivis de tous ceux de `s2`.

```
Sequence{1,2,3}->union( Sequence{0,1,2,4,5} )
   = Sequence{1..3,0..2,4..5}
```

- `s->isEmpty()` : Détermine si la séquence `s` est vide.

```
oclEmpty(Sequence(Integer))->isEmpty()

not Sequence{1,2}->isEmpty()
```

- `s->includesAll( s2 )` : Détermine si la séquence `s` contient chacun des éléments de `s2` (indépendamment du nombre d'occurrences et de la position).

```
Sequence{1,2,3}->includesAll( Sequence{1,2} )

Sequence{1,2,3}->includesAll( Sequence{1,2,2} )

not Sequence{1,2}->includesAll( Sequence{2,3} )
```


- `s->asSet()` : Retourne l'ensemble des éléments distincts de `s`.

```
Sequence{1,1,1,2,2}->asSet() = Set{1,2}
```

- `s->asBag()` : Retourne le sac des éléments de `s`.

```
Sequence{1,1,1,2,2}->asBag() = Bag{1,1,1,2,2}
```

- `s->count( x )` : Retourne le nombre d'occurrences de `x` dans `s`.

```
Sequence{1,1,1,2,2,3}->count( 1 ) = 3

Sequence{1,1,1,2,2,3}->count( 3 ) = 1

Sequence{1..10}->count( 0 ) = 0
```

- `s->at( i )` : Retourne le ième élément de `s`. Le premier index est 1. Retourne `Undefined` si l'index n'est pas valide.

```
Sequence{1..99}->at( 3 ) = 3

Sequence{1..99}->at( 0 ).isUndefined()
```

- `s->first()` : Retourne le premier élément de `s`.

```
Sequence{1..99}->first() = 1

Sequence{1..99}->first() = Sequence{1..99}->at( 1 )
```

- `s->last()` : Retourne le dernier élément de `s`.

```
Sequence{1..99}->last() = 99

Sequence{1..99}->last()
   = Sequence{1..99}->at( Sequence{1..99}->size() )
```

- `s->subSequence( debut, fin )` : Retourne la sous-séquence pour les index indiqués.

```
Sequence{1..10}->subSequence( 3, 6 ) = Sequence{3..6}

Sequence{1..10}->subSequence( 3, 3 ) = Sequence{3}

Sequence{1..10}->subSequence( 3, 2 )->isUndefined()
```

- `s->excluding( x )` : Retourne une séquence avec les mêmes éléments que `s`, sauf possiblement pour `x` dont *toutes les occurrences sont éliminées*.

```
Sequence{1,1,2,2,1,1,3,1}->excluding(1)->excluding(2)
  = Sequence{3}
```

- `s->select( p )` : Retourne la (sous-)séquence des éléments de `s` qui satisfont le prédicat `p`.

```
Sequence{0..10}->select( x | x.mod(2) = 0 )
   = Sequence{0,2,4,6,8,10}

Sequence{0..100}->select( x | x.mod(2) = 0 and x >= 100 )
   = Sequence{100}
```


- `s->reject( p )` : Retourne la (sous-)séquence des éléments de `s` qui ne satisfont pas le prédicat `p`.
- `s->collect( e )` : Retourne la séquence des valeurs obtenues en évaluant l'expression `e` pour les divers éléments de `s`.

```
Sequence{10,20,30}->collect( x | 2 * x + 1 )
   = Sequence{21,41,61}

Sequence{0..9}->collect( y | y.mod(2) )
   = Sequence{0,1,0,1,0,1,0,1,0,1}

Sequence{0..100}->collect( x | x.mod(10) )->asSet()
   = Set{0..9}

Sequence{1..4}->collect( x | Sequence{2..3}
              ->collect( y | x + y ) )
   = Sequence{3,4,4,5,5,6,6,7}
```

- `s->sortedBy( e )` : Retourne les éléments de `s` triés en fonction de la valeur produite par `e` — les éléments du type de `e` doivent pouvoir être comparés avec une méthode «<». Le résultat est toujours une `Sequence`.[1]

```
Sequence{10,6,2,1,3}->sortedBy( x | x )
  = Sequence{1,2,3,6,10}

Sequence{10,-6,4,-1,-5,9}->sortedBy( abs() )
  = Sequence{-1,4,-5,-6,9,10}

Sequence{10,6,2,1,3,7,9,4}->sortedBy( mod(3) )
  = Sequence{6,3,9,10,1,7,4,2}

Sequence{ Set{1..3}, Set{}, Set{8,8,8} }->sortedBy( size(
  = Sequence{Set{},Set{8},Set{1,2,3}}

Bag{ 'xyz', 'def', 'allo' }->sortedBy( x | x )
  = Sequence{'allo','def','xyz'}

Bag{ 'xyz', 'de', 'allo' }->sortedBy( size() )
  = Sequence{'de','xyz','allo'}

Set{20,10,30}->sortedBy( e | e )
  = Sequence{10,20,30}
```

[1] Par contre, en OCL 2.0 (plutôt que dans l'outil `use`), le résultat de `sortedBy` lorsqu'appliqué à un `Set` produit plutôt un `OrderedSet`, et non une `Sequence`.

6. La méthode `iterate`

- La méthode `iterate` permet d'effectuer des traitements itératifs arbitraires sur des collections.
- C'est une méthode qui s'applique à tous les types de collection.
- La méthode `iterate` est générale et permet en fait de réaliser n'importe quelle autre opération permettant de traiter tous les éléments d'une collection, par exemple, `sum()`, `select()`, `collect()`, etc.

- La syntaxe pour la forme la plus courante (où le type de `x` est omis) est la suivante, où `expr` est une expression qui peut utiliser `x` et `res` :

```
col->iterate( x;
              res: TypeRes = valInitiale |
              expr )
```

- L'interprétation (procédurale) d'une telle expression `iterate` est alors la suivante :

```
res ← valInitiale
POUR chaque x appartenant à col FAIRE
  res ← expr
FIN
RETOURNER res
```

Quelques exemples :

```
Set{10,20,30}
   ->iterate( x; tot: Integer = 0 |
              x + tot )
  = 60


Set{1,2,3}
   ->iterate( e; tot: Integer = 12 |
               10 * e + tot )
  = 72


Set{1..10,98,24}
   ->iterate( v; max: Integer = 0 |
              if v > max then v else max endif )
  = 98


Set{10,20,30}
   ->iterate( x; b: Bag(Integer) = Bag{} |
              b->including(x)->including(x) )
  = Bag{10,10,20,20,30,30}
```


D'autres exemples :

```
Sequence{1..10}
   ->iterate( x; nb: Integer = 0 |
              if x.mod(2) = 0
                then nb+1
                else nb
              endif )
  = 5


Sequence{1..10}
   ->iterate( x; seq: Sequence(Integer) = Sequence{} |
              if x.mod(2) = 1
                then seq->including(x)
                else seq
              endif )
  = Sequence{1,3,5,7,9}
```

Exercices :

- Quelles expressions peuvent s'exprimer plus simplement et de quelle façon?
- Comment peut-on exprimer la méthode `exists` à l'aide de la méthode `iterate`?

Équivalent de `iterate` en Ruby : `inject`
Plusieurs langages de programmation modernes possèdent une construction semblable au `iterate` d'OCL.

En Ruby, par exemple, on peut écrire les expressions suivantes, semblables à certaines des expressions présentées plus haut, où la valeur initiale est spécifiée comme argument (entre parenthèses) de l'appel à `inject` et où les variables sont déclarées dans l'ordre inverse d'OCL :

```
[10,20,30].inject(0) { |tot, x| x + tot }
```

```
[1,2,3].inject(12) { |tot, e| 10*e + tot }
```

```
[1,2,3,4,5,6,7,8,9,10,98,24].inject(0) { |max, v|
   if v > max then v else max end
}
```

```
[1,2,3,4,5,6,7,8,9,10].inject([]) { |seq, x|
  if x % 2 == 1 then seq.push(x) else seq end
}
```


7. Autres éléments OCL 2.0

Note : Tous les éléments OCL 2.0 qui suivent sont maintenant supportés dans l'outil USE (version 2.5.0).

- `OrderedSet` :

  Dans un `Set`, ni le nombre d'occurrences, ni l'ordre ne sont significatifs.

  Dans un `Bag`, le nombre d'occurrences est significatif mais l'ordre ne l'est pas.

  Dans un `OrderedSet`, c'est l'ordre qui est significatif mais pas le nombre d'occurrences :

  ```
  OrderedSet{1,2} <> OrderedSet{2,1}
  OrderedSet{10,8,5,6} <> OrderedSet{5,8,6,10}
  OrderedSet{1,2} = OrderedSet{1,1,2,2,1,1}
  OrderedSet{2,2,2,3,3,3,1,1,1,2} = OrderedSet{2,3,1}
  ```

- `Tuple` : Les *tuples* correspondent à la notion *d'enregistrements* — `RECORD` en Pascal/Ada, `struct` en C/C++.

  Définitions de types tuples — l'ordre de déclaration des champs *n'est pas significatif* :

```
Tuple( x: Integer, y: Integer )
Tuple( reel: Real, imag: Real )
Tuple( numero: Integer, rue: String, ville: String )
```

  Définitions de valeurs :

```
Tuple{ y=10, x=-12 }
Tuple{ reel=0.0, imag=1.0 }
Tuple{ numero=23, rue='Berri', ville='Laval' }
```

  Sélection d'un champ :

```
Tuple{ x=10, y=-12 }.x = 10
Tuple{ reel=0.0, imag=1.0 }.imag > 0.0
```


- Opérations prédéfinies sur tous les objets
  - `oclIsTypeOf(t: OclType): Boolean`
    Retourne `true` si et seulement si le type de l'objet sur lequel est appliquée l'opération est **identique** au type (`t`) passé en paramètre.
  - `oclIsKindOf(t: OclType): Boolean`
    Retourne `true` si et seulement si le type (`t`) passé en paramètre est le même type ou un type parent (super-type) de l'objet sur lequel l'opération est invoquée. En d'autres mots, retourne `true` si le type de l'objet est du même type ou est un sous-type de `t`.

OCL : Chapitre III

Modélisation conceptuelle avec UML et USE

0. Introduction

L'objectif de ce chapitre n'est pas que vous soyez capable de concevoir et d'écrire des diagrammes de classes — vous le ferez dans le cours INF5151. L'objectif est plutôt que vous soyez capables de **lire et comprendre** de tels diagrammes et de les **traduire dans la notation (textuelle) de l'outil USE**.

1. La notation UML

UML

= *Unified Modeling Language*

= notation graphique très (!) populaire pour la modélisation conceptuelle, l'analyse et la conception orientée-objets — notation devenue la *lingua franca* moderne

. . . a mis fin à la "guerre des méthodes" :

Combine des éléments des méthodes et notations des trois *gourous* (*los tres amigos*) : Booch (Booch OOD), Jacobson (Objectory), Rumbaugh (OMT)

2. Les diagrammes de classes d'UML

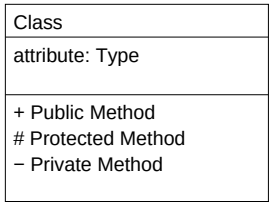
Notation graphique permettant de représenter :

- les entités (les objets)
- les attributs des entités (les variables d'instance)
- les relations entre entités (les liens entre objets)
- les dépendances
- les liens d'héritage (les sous-classes)

2.1 Entités et attributs

- Une entité — on dit aussi une classe — représente une collection d'objets ou de valeurs, donc dénote un *type*. Une telle entité est représentée graphiquement par une boîte rectangulaire, le nom de cette entité apparaissant dans la partie supérieure. Par convention, le nom d'une classe débute par une Majuscule.
- Un attribut est une caractéristique, un élément d'information décrivant une entité. Un attribut est généralement une unité atomique (indivisible) d'information. Dans un diagramme de classes, les attributs apparaissent dans le rectangle juste en dessous du nom de la classe.
- Des méthodes peuvent aussi être indiquées, dans le rectangle sous celui des attributs.

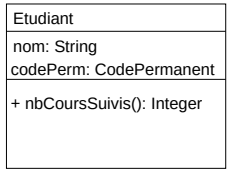
Description graphique d'une entité (forme générale) :



Remarque : Dans les semaines qui viennent, nous mettrons l'accent sur la modélisation conceptuelle (étape d'analyse) et nous nous restreindrons surtout aux entités et attributs, donc nous ne traiterons pas (ou très peu) les opérations et méthodes. Ces dernières seront vues ultérieurement, par le biais de la notion de **contrat**.

Exemple : Une classe pour `Etudiant` avec attributs `nom` et `codePerm` et méthode `nbCoursSuivis`

Représentation graphique :



Représentation en Java :

```
class Etudiant {
   private String nom;
   private CodePermanent codePerm;
   ...

   public Integer nbCoursSuivis() {
     return ...
   }
}
```

Représentation dans la notation de l'outil `USE` :

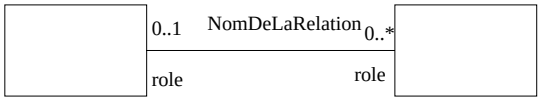
```
class Etudiant
attributes
  nom: String
  codePerm: CodePermanent
operations
  nbCoursSuivis(): Integer
end
```

2.2 Relations entre entités

- Une relation représente une association, un lien existant dans le problème entre des entités.
- Un nom de relation est généralement décrit par un *verbe*, alors qu'un nom de type ou d'attribut est généralement décrit par un *substantif*.
- On peut indiquer la cardinalité d'une relation, ce qui spécifie une contrainte sur le nombre d'entités pouvant être associées à l'autre entité. La cardinalité "$*$" dénote un nombre entier arbitraire.
- À une relation peuvent aussi être associés un ou deux *rôles*. Un rôle est indiqué par un identificateur qui apparaît près d'une entité et décrit le rôle de l'entité relativement à *l'autre* entité de la relation.

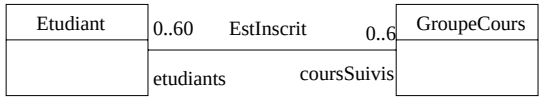
  Un rôle peut être vu comme une forme d'attribut *dérivé* dont la valeur est, souvent, une collection, i.e., les divers objets en lien avec l'entité via la relation.

Description graphique d'une relation (forme générale) :



Exemple : Une relation entre `Etudiant` et `GroupeCours` — un étudiant peut être inscrit à au plus six (6) cours, et chaque groupe-cours ne peut contenir qu'au plus 60 étudiants.

Représentation graphique :



Représentation en Java :

```
class Etudiant {
   private Set<GroupeCours> coursSuivis;
   ...

   public Integer coursSuivis() {
     return coursSuivis;
   }
}
```

Représentation `USE` :

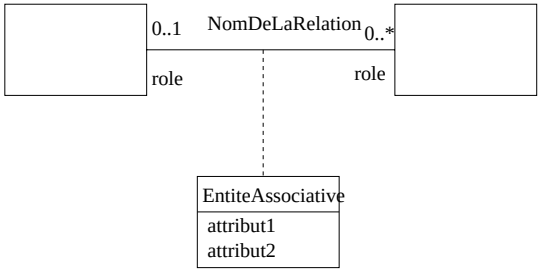
```
association EstInscrit between
  Etudiant[0..60] role etudiants
  GroupeCours[0..6] role courSuivis
end
```

2.3 Liens de dépendance et entités associatives

- Une dépendance — représentée par une ligne pointillée — représente un type particulier de relation, où l'existence d'une entité ou relation dépend d'une autre entité ou relation.
- Un cas souvent rencontré est celui d'une *entité associative* : une telle entité est liée à une relation parce qu'elle vise à *décrire* cette relation.

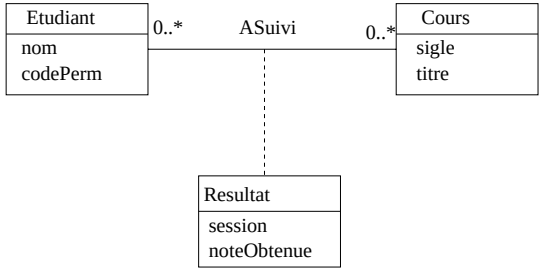
  Alors qu'une entité peut exister en dehors de toute relation, une entité associative n'existe que lorsque des entités appropriées sont en relation. À chaque entité associative sont donc associés des attributs décrivant la relation et des attributs (implicites) indiquant les instances pour lesquelles la relation existe.

Description graphique d'une entité associative (forme générale) :



Exemple : Une relation entre `Etudiant` et `Cours` donnant lieu à une entité associative : lorsqu'un étudiant a suivi un cours, alors on doit savoir quelle note il a obtenue et quand il a suivi le cours.

Représentation graphique :



Représentation `USE` :

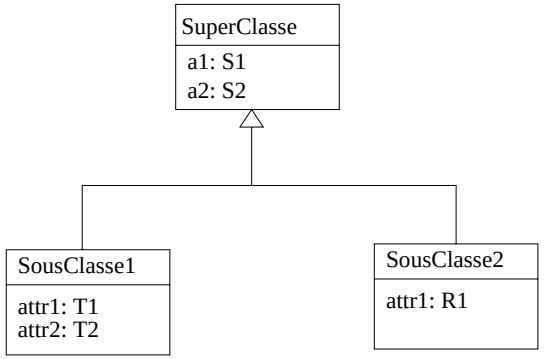
```
associationclass Resultat between
   Etudiant[0..*]
   Cours[0..*]
attributes
   session: Session
   noteObtenue: Integer
end
```

2.4 Liens d'héritage

- Une *relation d'héritage* — représentée par la présence d'un triangle près de la super-classe — représente un autre type particulier de relation.

  Note : On parle aussi de relation de *spécialisation* : les entités des sous-classes sont des spécialisations de celles de la super-classe. En d'autres mots, les entités des types des sous-classes *sont aussi des entités* du type de la super-classe.
- Une entité de la sous-classe possède tous les attributs (et opérations, si présentes) qui lui sont directement associés *ainsi que tous les attributs associés à la super-classe*.
- Un cas souvent rencontré est lorsque la super-classe est *abstraite* (nom en *italiques*). Dans ce cas, il n'est pas possible d'avoir des objets de la super-classe : les seules instances possibles sont celles des sous-classes concrètes.

Description graphique d'une relation d'héritage (forme générale avec une super-classe et deux sous-classes, qui sont toutes des classes concrètes) :

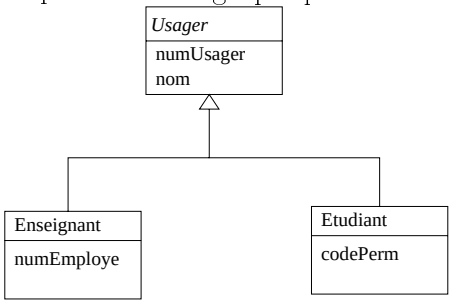


Pour un tel modèle (sans classe abstraite), on a alors :

- Les objets de type `SuperClasse` possèdent des attributs `a1` et `a2` (types `S1` et `S2`).
- Les objets de type `SousClasse1` possèdent des attributs `a1` et `a2` (types `S1` et `S2`) *ainsi que des attributs* `attr1` et `attr2` (types `T1` et `T2`).
- Les objets de type `SousClasse2` possèdent des attributs `a1` et `a2` (types `S1` et `S2`) *ainsi qu'un attribut* `attr1` (types `R1`).

Exemple : Une relation d'héritage (de spécialisation) entre, d'une part, `Usager` (d'une bibliothèque universitaire) et, d'autre part `Etudiant` et `Enseignant`, les deux types d'usager n'ayant pas les mêmes droits d'emprunt. Ici, la classe ***`Usager`*** est abstraite, puisqu'un ***`Usager`*** abstrait n'existe pas : soit c'est un `Enseignant`, soit c'est un `Etudiant`.

Représentation graphique :



Représentation `USE` :

```
abstract class Usager
attributes
  numUsager: NumeroUsager
  nom: String
end

class Enseignant < Usager
attributes
  numEmploye: NumeroEmploye
end

class Etudiant < Usager
attributes
  codePerm: CodePermanent
end
```


3. Les diagrammes d'objets

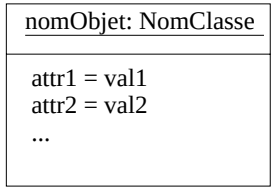
Diagramme de classes vs. diagramme d'objets :

- Un diagramme **de classes** spécifie un *modèle* d'un problème, d'un domaine d'application.

  Un tel modèle décrit les entités et relations qui existent de façon *abstraite* = **les concepts clés** du problème, du domaine d'application.

- Un diagramme **d'objets** représente des *instances concrètes* du modèle abstrait. Les éléments du diagramme représentent donc des objets spécifiques qui définissent l'état d'un *système* particulier (décrit par le *modèle*).

Description graphique d'un objet (forme générale) :



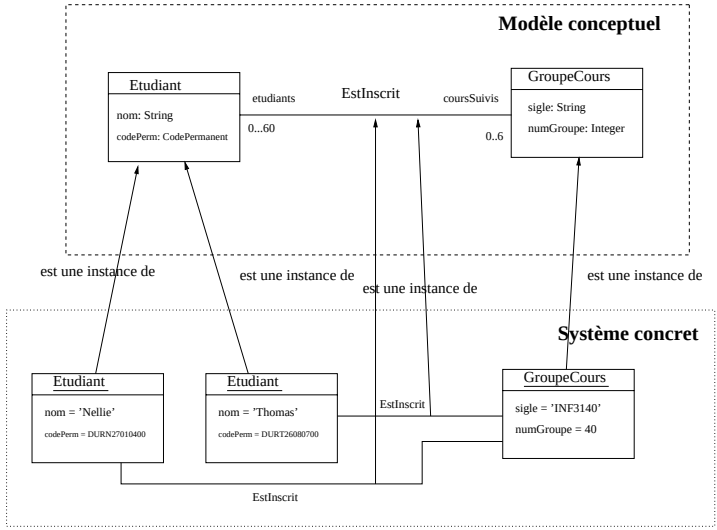
Donc :

- Le nom de l'objet et de la classe sont soulignés.
- Les valeurs spécifiques des attributs de l'objet sont (parfois) indiquées.
- Le nom spécifique de l'objet est parfois omis, mais jamais le nom de la classe.

Analogie :

- Diagramme de classes = **schéma** décrivant la structure d'une base de données
- Diagramme d'objets = **contenu effectif** d'une base de données, qui obéit aux contraintes du schéma

Exemple : Illustration d'un système avec deux objets `Etudiant` et un objet `GroupeCours`, où les deux étudiants sont inscrits dans le groupe-cours, montrant la correspondance entre les objets concrets du système et les entités conceptuelles du diagramme de classes.

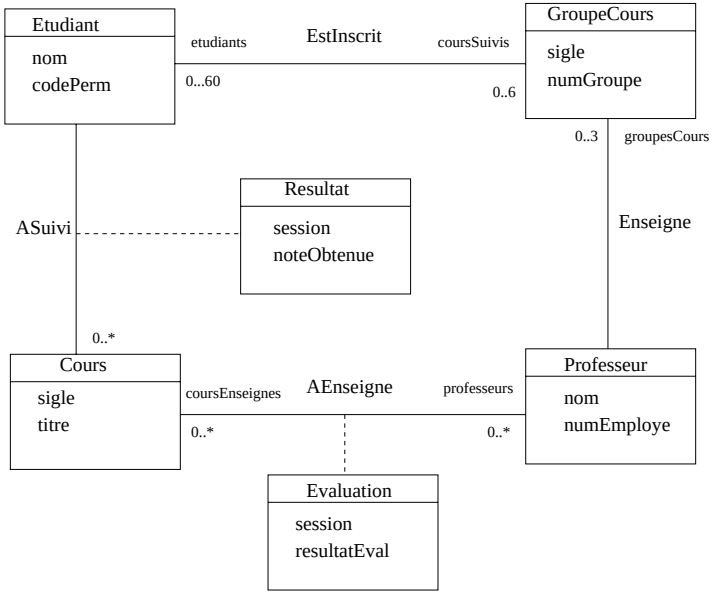


4. Un exemple combinant les divers éléments pour étudiants, cours, etc.

Un modèles conceptuel et des objets concrets pour des étudiants, cours, groupes-cours et professeurs, avec entités associatives.

4.1 Modèle conceptuel

Diagramme de classes



Représentation dans la notation de l'outil USE

```
model Cours

//
// Entites.
//

class Etudiant
attributes
  nom: String
  codePerm: CodePermanent
end

class Cours
attributes
  sigle: String
  titre: String
end

class GroupeCours
attributes
  sigle:  String
  numGroupe: Integer
end

class Professeur
attributes
  nom:  String
  numEmploye: NumEmploye
end
```

```
//
// Relations simples.
//

association Enseigne between
  Professeur[1]
  GroupeCours[0..3] role groupesCours
end

association EstInscrit between
  Etudiant[0..60] role etudiants
  GroupeCours[0..6] role coursSuivis
end


//
// Entites associatives.
//
associationclass Resultat between
   Cours[0..*]
   Etudiant[0..*]
attributes
   session: Session
   noteObtenue: Integer
end

associationclass Evaluation between
   Cours[0..*] role coursEnseignes
   Professeur[0..*] role professeurs
attributes
   session: Session
   resultatEval: Integer
end
```

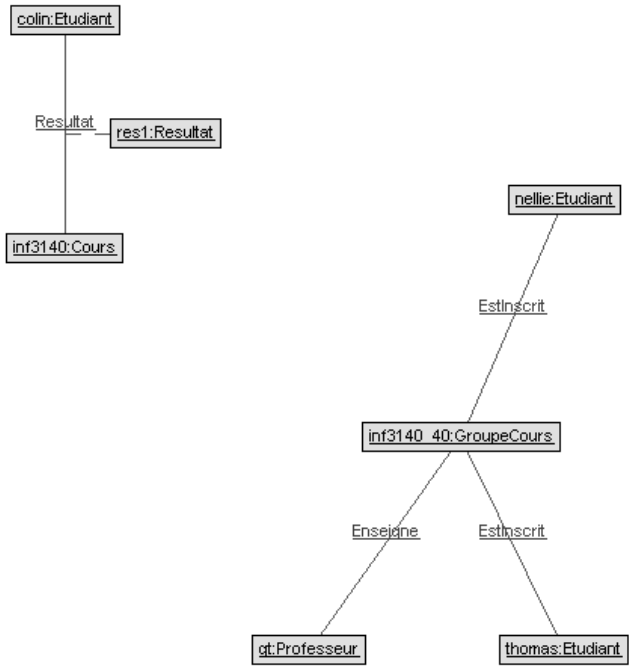


```
//
// Types primitifs (sans attributs).
//

class CodePermanent end

class NumEmploye end

class Session end
```

4.2 Objets concrets

Diagramme d'objets

Note : Diagramme produit par l'outil `USE`



Représentation dans la notation de l'outil `USE`

```
!create cp1, cp2: CodePermanent

!create nellie, thomas: Etudiant
!set nellie.nom := 'Nellie Durocher'
!set thomas.nom := 'Thomas Durocher'
!set nellie.codePerm := cp1
!set thomas.codePerm := cp2

!create inf3140_40: GroupeCours
!set inf3140_40.sigle := 'INF3140'
!set inf3140_40.numGroupe := 40

!create gt: Professeur
!set gt.nom := 'Guy Tremblay'

!insert (nellie,inf3140_40) into EstInscrit
!insert (thomas,inf3140_40) into EstInscrit
!insert (gt,inf3140_40) into Enseigne

!create colin: Etudiant
!set colin.nom := 'Colin Garceau'

!create inf3140: Cours
!set inf3140.sigle := 'INF3140'
!set inf3140.titre := 'Specification formelle'

!create res1: Resultat between (colin, inf3140)
!set res1.noteObtenue := #A
```

Note : Pour simplifier le diagramme, les objets `cp1` et `cp2` (`CodePermanent`) ne sont pas indiqués sur la figure précédente.

4.3 Comment vérifier qu'un système concret est fidèle au modèle conceptuel

Lorsqu'on définit un système concret d'objets — une collection d'objets concrets et les liens qui les unissent — il est important de s'assurer que le système ainsi créé est bien **une instance valide** du modèle conceptuel.

L'outil `USE` permet d'effectuer une telle vérification.

Supposons :

- Le modèle conceptuel est défini dans le fichier `cours.use`
- Le système concret est défini dans le fichier `cours.cmd`

Différentes utilisations possibles (mode «ligne de commandes» Unix) :

1. On vérifie la syntaxe et les types du modèle conceptuel, puis on termine l'exécution :

```
$ use -nogui -c cours.use
$
```

2. On vérifie la syntaxe et les types du modèle conceptuel, on fait afficher le code compilé, puis on termine l'exécution :

```
$ use -nogui -cp cours.use
model Cours
enum NoteLitterale { A, B, C, D, E };

class CodePermanent
end

class Cours
.
.
.
$
```

3. On vérifie le modèle et la structure du système (respect du modèle), puis on termine l'exécution :

```
$ use -nogui cours.use <cours.cmd
$
```

4. On vérifie le modèle et la structure du système, puis on exécute diverses commandes, entre autres, des requêtes :

```
$ use -nogui cours.use cours.cmd
use version 2.4.0, Copyright (C) 1999-2008 Univers
cours.cmd> !create cp1, cp2: CodePermanent
cours.cmd>
cours.cmd> !create nellie, thomas: Etudiant
...
cours.cmd>
use> ?inf3140_40.professeur
-> @gt : Professeur
use> ?inf3140_40.professeur.nom
-> 'Guy Tremblay' : String
use> ?inf3140_40.etudiants
-> Set{@nellie,@thomas} : Set(Etudiant)
use> ?inf3140_40.etudiants.nom
-> Bag{'Nellie Durocher','Thomas Durocher'} : Bag
```


OCL : Chapitre IV

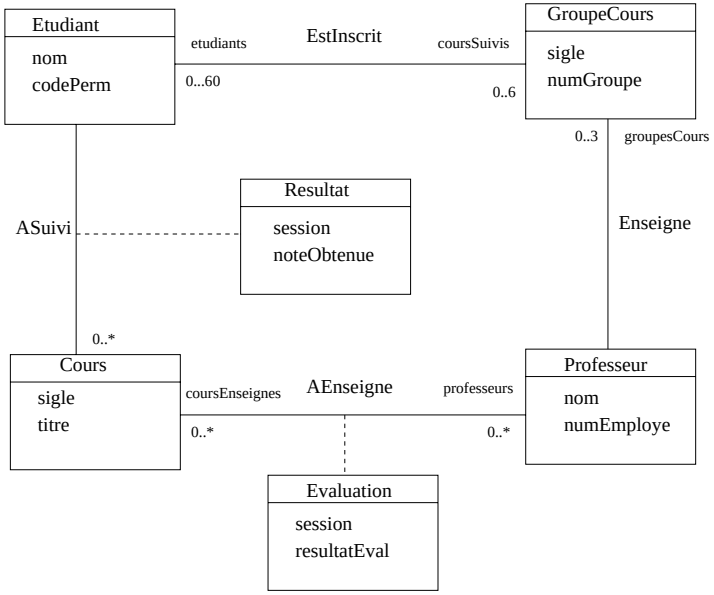
Spécification de contraintes et requêtes sur des diagrammes de classes UML

0. Introduction

Objectif : Dans le présent chapitre nous allons voir comment spécifier des **contraintes**, exprimées en OCL, s'appliquant sur des entités et relations décrites par des diagrammes de classes UML.

1. Contexte d'une expression et navigation

Soit le diagramme de classes suivant :



Une **expression de navigation** permet, à partir d'un point d'ancrage appelé un **contexte**, de naviguer dans un diagramme de classes pour sélectionner des attributs, collections, liens, etc., associés à ce contexte.

Exemple — pour le diagramme de classes précédent :

```
context GroupeCours
  ...
  sigle
  ...
  numGroupe
  ...
  etudiants
  ...
  professeur.nom
```

Dans ce contexte, on a alors les interprétations suivantes de ces identificateurs ou expressions :

- `sigle` : l'attribut `sigle` du groupe-cours.
- `numGroupe` : l'attribut `numGroupe` du groupe-cours.
- `etudiants` (rôle *explicite*) : la *collection* d'`Etudiant`s (multiplicité supérieure à 1) en relation avec le groupe-cours.
- `professeur` (rôle *implicite*) : le `Professeur` (unique : multiplicité 1) qui `enseigne` le groupe-cours.
- `professeur.nom` : le `nom` du `Professeur` qui `enseigne` le groupe-cours.

2. Les différents rôles

- **Rôle explicite**

  = nom indiqué sur l'extrémité d'une relation

  ⇒ utilisable comme un attribut de l'entité, produisant une collection si multiplicité > 1

  Exemples :

  * `GroupeCours` : `etudiants`
  * `Professeur` : `coursEnseignes`, `groupesCours`
  * `Etudiant` : `coursSuivis`

- **Rôle implicite**

  = aucun nom de rôle n'est indiqué sur la relation

  ⇒ le nom de la classe associée, mais avec une minuscule, peut être utilisé comme un nom de rôle

  ⇒ produit aussi une collection si multiplicité > 1

  Exemples :

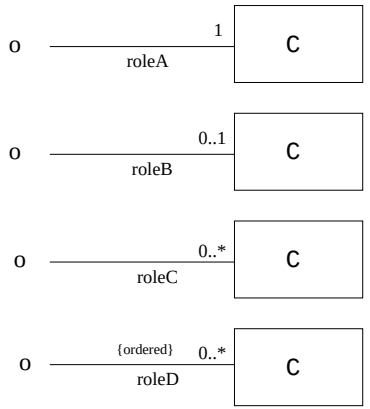
  * `GroupeCours` : `professeur`
  * `Etudiant` : `cours`, `resultat`
  * `Resultat` : `etudiant`, `cours`
  * `Evaluation` : `cours`

Remarques :

- Un nom de classe débute toujours par une `Majuscule`
- Une entité associative possède aussi des rôles implicites

3. Cardinalités et types des rôles

Soit les rôles suivants, pour diverses relations avec la classe `C` toutes de cardinalités (ou type) différentes :



Soit `o` un objet en lien avec des objets de type `C` via ces différentes relations. Alors, les types des résultats retournés par les différents rôles sont les suivants :

- `o.roleA: C`
- `o.roleB: C`
- `o.roleC: Set(C)`
- `o.roleD: Sequence(C)`

Remarque : En OCL 2.0, le type de `roleD` serait plutôt `OrderedSet(C)`.

Cas particulier d'un rôle de cardinalité «1» ou «0..1»

Lorsqu'un rôle est de cardinalité «1» ou «0..1» — donc représente (au plus) un élément — une conversion implicite sous forme de collection *singleton* est possible.

Par exemple, pour le rôle `roleB` indiqué plus haut (idem pour `roleA`), les expressions suivantes seraient valides et permettraient par exemple (trois premiers cas) de déterminer si l'objet `o` est, ou non, en relation avec un objet de classe `C` :

```
o.roleB->size() = 1
o.roleB->notEmpty()
not b1.roleB.isUndefined()
```


4. L'identificateur `self`

L'identificateur "`self`" réfère toujours à l'objet (arbitraire) associé au contexte courant.

Les références indiquées plus haut sont donc équivalentes aux suivantes :

```
context GroupeCours
  ...
  self.sigle
  ...
  self.numGroupe
  ...
  self.etudiants
  ...
  self.professeur.nom
```

5. Divers exemples d'expression de navigation

- Le nombre de cours suivis par un étudiant :

```
context Etudiant
  ...
  coursSuivis->size()
  ...
```

- Les noms de tous les professeurs qui enseignent l'un des cours suivis par un étudiant :

```
context Etudiant
  ...
  coursSuivis->collect( c | c.professeur.nom )
  ...
```


Raccourci pour les collections

Il existe un raccourci permettant d'écrire l'expression précédente :

```
coursSuivis.professeur.nom
```

De façon générale, si `col` est une collection et `a` un attribut des éléments de cette collection, alors on a l'équivalence suivante :

```
col->collect( c | c.a )
=
col.a
```

- La collection (sac) de tous les sigles possibles de cours pour des cours suivis par les divers étudiants inscrits dans les groupes-cours enseignés par un professeur :

```
context Professeur
  ..
  groupesCours.etudiants.cours.sigle
  ...
```

Note : L'expression précédente est équivalente à la suivante :

```
groupesCours
  ->collect( etudiants )
  ->collect( cours )
  ->collect( sigle )
```

Or, pour un `Professeur`, le rôle explicite `groupesCours` retourne une collection de `GroupeCours`, et pour un `GroupeCours` le rôle explicite `cours` retourne une collection de `Cours`. On semble donc obtenir une collection... de collections de `Cours`. Toutefois, grâce à la règle du `flatten` *implicite* pour l'opération `collect`, on obtient bien une collection de `Cours`, ce qui permet d'obtenir une collection (sac) de `Sigle` pour l'expression globale.

- Étant donné un professeur, les étudiants qui suivent l'un des cours qu'il enseigne et qui ont le même nom que lui :

```
context Professeur
  ...
  groupesCours
    .etudiants
    ->select( e | e.nom = nom )
    ->asSet()
  ...
```

Autre façon d'exprimer la même expression, illustrant le rôle de `self` lorsqu'une opération sur une collection ne déclare pas explicitement d'argument :

```
  ...
  groupesCours
    .etudiants
    ->select( nom = self.nom )
    ->asSet()
```

6. Contraintes et invariants

Les expressions OCL peuvent être utilisées pour spécifier, textuellement, diverses contraintes.

La forme de contrainte la plus importante est l'**invariant**, qui décrit une propriété qui doit être vraie "en tout temps" :

> *An* invariant *is a constraint that should be true for an object during its complete lifetime. Invariants often represent rules that should hold for the real-life objects after which the software objects are modeled. (Warmer & Kleppe, 2003)*

On peut utiliser des contraintes OCL pour formaliser certaines propriétés pouvant s'exprimer graphiquement — par exemple, cardinalité d'une relation — mais surtout pour formaliser des propriétés qui ne *peuvent pas* s'exprimer graphiquement

6.1 Invariants sur la cardinalité de relations

- Cardinalité de la relation `estInscrit` pour le rôle `coursSuivis` :

```
context Etudiant
inv NbCoursSuivisOk:
   0 <= coursSuivis->size()
   and
   coursSuivis->size() <= 6
```

Remarques :

  - Le mot-clé `inv` indique que l'expression booléenne qui suit «:» décrit un `inv`ariant qui s'applique au contexte indiqué.
  - L'identificateur entre "`inv`" et ":" est optionnel : il sert à donner un nom, significatif, à l'invariant (utile lors des vérifications).

- Cardinalité de la relation `estInscrit` pour le rôle `etudiants` :

```
context GroupeCours
inv NbEtudiantsOk:
   0 <= etudiants->size()
   and
   etudiants->size() <= 60
```

Remarque :

- Il est souvent possible, et utile (noms significatifs), de décomposer un invariant complexe (contenant un ou plusieurs `and`) en plusieurs invariants indépendants.

Exemple :

```
context Etudiant
inv NbCoursSuivisNonNegatif:
   0 <= coursSuivis->size()
inv NbCoursSuivisInferieurAuMax:
   coursSuivis->size() <= 6
```


6.2 Invariants sur des attributs

- Un numéro de groupe est toujours supérieur à 0 (`Integer` comprend 0 et les négatifs!)

```
context GroupeCours
inv NumGroupePositif:
  numGroupe > 0
```

Note : Avec `USE`, on pourrait aussi écrire :

```
class GroupeCours
attributes
  sigle:  String
  numGroupe: Integer
constraints
  inv NumGroupePositif:
    numGroupe > 0
end
```

- Un résultat d'évaluation est plus grand ou égal à 0

```
associationclass Evaluation between
  ...
constraints
   inv ResultatNonNegatif:
     resultatEval >= 0
end
```

6.3 Invariants spécifiant *l'unicité* d'identifiants

Il arrive fréquemment qu'un attribut serve d'*identifiant*, donc sert à identifier *de façon unique* un objet. Exemples typiques : numéro d'employé et code permanent.

- ```
  context Professeur
  inv NumEmployeUnique:
    Professeur.allInstances->forAll( p |
             p <> self
             implies
             p.numEmploye <> numEmploye )
  ```
- ```
  context Etudiant
  inv CodePermanentUnique:
    Etudiant.allInstances->forAll( e |
             e <> self
             implies
             e.codePerm <> codePerm )
  ```

Remarque : Soit une nom de classe `C`. Alors, l'expression `C.allInstances` permet d'obtenir la collection de *toutes* les instances de cette classe, i.e., une collection qui comprend tous les objets de cette classe (et sous-classes) qui existent au moment de l'appel.

Remarque : Les contraintes qui précédent peuvent être exprimées plus simplement à l'aide de l'opération `isUnique` :

- ```
  context Professeur
  inv NumEmployeUnique:
    Professeur.allInstances->isUnique( numEmploye )
  ```
- ```
  context Etudiant
  inv CodePermanentUnique:
    Etudiant.allInstances->isUnique( codePerm )
  ```

The parameter of [the `isUnique`*] operation is usually a feature of the type of the elements of the collection. The result is either true or false. The operation will loop over all elements and compare the values by calculating the parameter expression for all elements. If none of the values is equal to another, the result is true; otherwise the result is false. (Warmer & Kleppe, 2003)*

6.4 Invariant spécifiant des contraintes inter-entités

- Un étudiant ne peut pas suivre un cours dont le professeur a le même nom que l'étudiant :

```
context Etudiant
inv PasLieAuProf2:
  coursSuivis.professeur.nom
       ->forAll( n | n <> nom )
```

7. Invariants complexes, type `enum` et `let_in_`

- Si un étudiant est inscrit à un cours, alors s'il a déjà suivi ce cours, sa note était nécessairement *inférieure* à `C` — donc égale à `D` ou `E` :

```
context Etudiant
inv NoteOkPourCoursDejaSuivi:
 coursSuivis->forAll( gc |
   Resultat.allInstances->forAll( r |
     r.cours.sigle = gc.sigle
     and
     r.etudiant = self
     implies
     (r.noteObtenue = #D or r.noteObtenue = #E) )
```

Pour cet exemple, on suppose :

– Que le type énuméré suivant a été défini (pour simplifier, on ignore les `A+`, `A-`, etc.) :

```
enum NoteLitterale {A, B, C, D, E}
```

– Que l'attribut `noteObtenue` est déclaré ainsi :

```
noteObtenue: NoteLitterale
```

Remarque En OCL 2.0, une valeur du type énuméré `NoteLitterale` telle que "`#C`" serait plutôt notée "`NoteLitterale::C`".

Autre version de la même contrainte, avec `let...in...` :

-
```
context Etudiant
inv NoteOkPourCoursDejaSuivi2:

  let siglesCoursInscrits : Bag(String)
      = coursSuivis.sigle in

  let coursDejaSuivis : Set(Cours)
      = cours->select(
           siglesCoursInscrits->includes(sigle) )

    coursDejaSuivis.resultat.noteObtenue
         ->forAll( n | n = #D or n = #E )
```

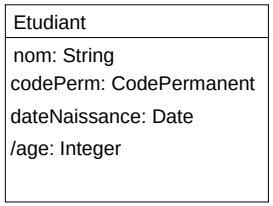
Remarque : La construction `let...in...` permet d'introduire un identificateur *local*, ce qui est souvent utile pour décomposer les expressions complexes en sous-expressions plus facilement compréhensibles.

8. Spécification d'attributs dérivés

La notation OCL standard permet de définir des *attributs dérivés* :

- Un attribut dérivé est un élément d'information *qui peut être calculé* (donc *dérivé*) à partir d'autres attributs.
- Un attribut dérivé est indiqué, sur le diagramme de classes, en le précédant d'une barre oblique.

Exemple : puisque l'âge peut être déterminé à partir de la date de naissance :



En OCL standard, un tel attribut dérivé peut être spécifié comme suit, en supposant qu'il existe une méthode `nombreAnneeDepuis` associée au type `Date` :

```
context Etudiant::age
derive: Date.nombreAnneesDepuis( dateNaissance )
```

Note : L'outil `USE` ne supporte pas le mot-clé `derive` :(

9. Spécification de requêtes (op. auxiliaires)

OCL standard permet de spécifier des *opérations auxiliaires* — qui ne sont pas des méthodes publiques — à l'aide du mot-clé "`def`".

Une telle opération doit nécessairement être une *requête* (en anglais *query*, on dit aussi un *observateur*) : une requête *observe* l'état du système sans le modifier.

Exemples (notation OCL standard) :

- Une opération qui détermine si un étudiant est à temps-plein (inscrit à quatre cours ou plus) :

```
context Etudiant::estATempsPlein(): Boolean
def: coursSuivis->size() >= 4
```

- Une opération qui détermine si un étudiant a déjà suivi un certain cours avec succès :

```
context Etudiant::aSuiviAvecSucces
       ( sigle: String ): Boolean
def: resultat->exists( r | r.cours.sigle = sigle
                           and
                           r.noteObtenue <> #E )
```

Mais : L'outil `USE` ne supporte pas le mot-clé `def` :(

Par contre, on peut définir des opérations (**requêtes**) qui sont associées *directement* à une expression.

Exemples (notation de l'outil `USE`) :

```
class Etudiant

attributes
  nom: String
  codePerm: CodePermanent

operations

  estATempsPlein(): Boolean
    = coursSuivis->size() >= 4

  aSuiviAvecSucces( sigle: String ): Boolean
    = resultat->exists( r | r.cours.sigle = sigle
                            and
                            r.noteObtenue <> #E )
end
```

Une fois définies, de telles opérations peuvent ensuite être utilisées dans les contraintes, ce qui facilite leur lecture et écriture (abstraction procédurale).

10. Vérification des contraintes

Étant donné un système d'objets concrets, l'outil `USE` permet de vérifier que ces objets, et leurs relations, satisfont les contraintes du modèle.

Soit le système concret suivant (fichier `cours.cmd`), lequel est une instance du modèle abstrait de la page (fichier `cours.use`) avec les diverses contraintes présentées plus haut :

```
!create cp1, cp2: CodePermanent

!create nellie, thomas: Etudiant
!set nellie.nom := 'Nellie Durocher'
!set thomas.nom := 'Thomas Durocher'
!set nellie.codePerm := cp1
!set thomas.codePerm := cp2

!create inf3140_40: GroupeCours
!set inf3140_40.sigle := 'INF3140'
!set inf3140_40.numGroupe := 40

!create gt: Professeur
!set gt.nom := 'Guy Tremblay'

!insert (nellie,inf3140_40) into EstInscrit
!insert (thomas,inf3140_40) into EstInscrit
!insert (gt,inf3140_40) into Enseigne

!create colin: Etudiant
!set colin.nom := 'Colin Garceau'

!create inf3140: Cours
!set inf3140.sigle := 'INF3140'
!set inf3140.titre := 'Specification formelle'

!create res1: Resultat between (colin, inf3140)
!set res1.noteObtenue := #A
```


Exemple d'exécution permettant de vérifier que le système décrit par `cours.cmd` satisfait le modèle et les contraintes de `cours.use` :

```
$ use -nogui cours.use cours.cmd
use version 2.4.0, Copyright (C) 1999-2008 University of Bremen
cours.cmd> !create cp1, cp2: CodePermanent
cours.cmd>
cours.cmd> !create nellie, thomas: Etudiant
cours.cmd> !set nellie.nom := 'Nellie Durocher'
.
.
cours.cmd> !create res1: Resultat between (colin, inf3140)
cours.cmd> !set res1.noteObtenue := #A
cours.cmd>
use> check
checking structure...
checking invariants...
checking invariant (1) `Etudiant::CodePermanentUnique': OK.
checking invariant (2) `Etudiant::CodePermanentUnique2': OK.
checking invariant (3) `Etudiant::CoursDistincts': OK.
checking invariant (4) `Etudiant::NbCoursSuivisInferieurAuMax': OK.
checking invariant (5) `Etudiant::NbCoursSuivisNonNegatif': OK.
checking invariant (6) `Etudiant::NbCoursSuivisOk': OK.
checking invariant (7) `Etudiant::NoteOkPourCoursDejaSuivi': OK.
checking invariant (8) `Etudiant::NoteOkPourCoursDejaSuivi2': OK.
checking invariant (9) `Etudiant::PasLieAuProf': OK.
checking invariant (10) `Etudiant::PasLieAuProf2': OK.
checking invariant (11) `Evaluation::EvaluationNonNegative': OK.
checking invariant (12) `GroupeCours::NbEtudiantsOk': OK.
checking invariant (13) `GroupeCours::NumGroupePositif': OK.
checking invariant (14) `GroupeCours::NumGroupePositif1': OK.
checking invariant (15) `Professeur::NbCoursEnseignesOk': OK.
checking invariant (16) `Professeur::NumEmployeUnique': OK.
checking invariant (17) `Professeur::NumEmployeUnique2': OK.
checking invariant (18) `Professeur::PasLieAuxEtudiants': OK.
checking invariant (19) `Professeur::PasLieAuxEtudiants2': OK.
checked 19 invariants in 0.016s, 0 failures.
use>
```

Autre exemple d'exécution permettant de vérifier que le système décrit par `cours.cmd` satisfait le modèle et les contraintes de `cours.use` :

```
$ use -nogui -qv cours.use cours.cmd
checking structure...
checking invariants...
checking invariant (1) 'Etudiant::CodePermanentUnique': OK.
checking invariant (2) 'Etudiant::CodePermanentUnique2': OK.
checking invariant (3) 'Etudiant::CoursDistincts': OK.
checking invariant (4) 'Etudiant::NbCoursSuivisInferieurAuMax': OK.
checking invariant (5) 'Etudiant::NbCoursSuivisNonNegatif': OK.
checking invariant (6) 'Etudiant::NbCoursSuivisOk': OK.
checking invariant (7) 'Etudiant::NoteOkPourCoursDejaSuivi': OK.
checking invariant (8) 'Etudiant::NoteOkPourCoursDejaSuivi2': OK.
checking invariant (9) 'Etudiant::PasLieAuProf': OK.
checking invariant (10) 'Etudiant::PasLieAuProf2': OK.
checking invariant (11) 'Evaluation::EvaluationNonNegative': OK.
checking invariant (12) 'GroupeCours::NbEtudiantsOk': OK.
checking invariant (13) 'GroupeCours::NumGroupePositif': OK.
checking invariant (14) 'GroupeCours::NumGroupePositif1': OK.
checking invariant (15) 'Professeur::NbCoursEnseignesOk': OK.
checking invariant (16) 'Professeur::NumEmployeUnique': OK.
checking invariant (17) 'Professeur::NumEmployeUnique2': OK.
checking invariant (18) 'Professeur::PasLieAuxEtudiants': OK.
checking invariant (19) 'Professeur::PasLieAuxEtudiants2': OK.
checked 19 invariants in 0.016s, 0 failures.
$
```


OCL : Chapitre V

Spécification de contrats pour des opérations sur des classes UML

0. Introduction

Remarques :

- Dans ce chapitre nous allons voir comment spécifier des opérations, et ce en utilisant des **contrats** exprimés en OCL.

  Par la suite, nous verrons comment spécifier des contrats semblables en Java (avec l'outil `Modern JASS`).

- Dans ce qui suit, nous allons utiliser le terme «composant logiciel» pour dénoter une entité logicielle quelconque : programme, système, sous-système, module, classe, etc. Le contexte déterminera de quel(s) genre(s) d'entité(s) il s'agit.

  > **Composant logiciel** : Logiciel, programme ou élément d'un logiciel ou d'un programme qui constitue un module indépendant utilisé comme élément d'un système plus complexe et qui est spécialement conçu pour fonctionner sans problèmes avec d'autres logiciels ou programmes.
  >
  > `http://www.granddictionnaire.com`

1. Spécifications fonctionnelles

Une **spécification fonctionnelle** vise à décrire le **comportement** d'un composant logiciel :

> *A functional specification [...], in systems engineering and software development, is the set of documentation that describes the requested behavior of an engineering system. The documentation typically describes what is needed by the system user as well as requested properties of inputs and outputs [...].*
>
> `http://en.wikipedia.org/wiki/Functional_specification`

- À l'étape d'analyse et spécification, on décrit le comportement (global) du logiciel qui vise à satisfaire les besoins et exigences du client.
- À l'étape de conception (architecturale ou détaillée), on décrit le comportement des divers sous-systèmes, modules, classes, méthodes, etc.

Spécification boîte noire: Quoi? vs. Comment?

On considère généralement préférable de décrire le comportement d'un composant (QUOI?) sans décrire la façon dont ce comportement est réalisé (COMMENT?)

- «Quoi?» = Que doit faire le composant? Comment se comporte-t-il?
- «Comment?» = De quelle façon le composant est-il mis en oeuvre?

Donc, on vise à définir des spécifications de type **boîte noire** (on ne voit rien de ce qui se passe à l'intérieur du composant) :

> **Boîte noire** : Élément logiciel ou matériel dont le fonctionnement est connu et documenté, mais dont la structure interne est inconnue.
>
> `http://www.granddictionnaire.com`

Autre façon d'exprimer l'idée de boîte noire :

- On décrit les concepts et les interfaces nécessaires pour **utiliser** un composant logiciel — d'un point de vue **externe**, donc du point de vue des **clients** (utilisateurs) du composant.

On parle aussi de «**spécification comportementale**».

2. Spécifications comportementales abstraites

Les spécifications fonctionnelles que nous étudierons seront des spécifications **abstraites**, en ce sens qu'elles seront *indépendantes des détails des entrées/sorties concrètes.*

Donc : on se concentre sur les **concepts** plutôt que sur les détails de syntaxe de commandes ou d'interface personne–machine

Exemple : On veut décrire le comportement d'une opération qui permet d'imprimer un fichier sur une imprimante cible, mais en faisant abstraction des détails exacts de l'interface personne–machine :

```
OPERATION imprimerFichier( fich: NomDeFichier, imp: NomImprimante )
   NECESSITE  Le fichier fich existe ET l'imprimante imp existe.
   EFFETS     Le fichier fich a ete imprime sur l'imprimante.
   EXCEPTION  Le fichier ou l'imprimante n'existe pas.
```

Contre-exemple :

`lpr -P` *`nomImprimante nomDeFichier`*

Contre-exemple :

> «Pour imprimer un fichier, il faut cliquer sur l'icône représentant le fichier puis la déplacer sur l'icone représentant l'imprimante.»

3. Pré/postcondition d'une opération

Soit une opération `op` ayant la signature suivante :

```
op( a1: T1, ..., ak: Tk ): R
```

- Une **précondition** de l'opération `op` est une *contrainte* (une expression booléenne) qui doit être vraie au moment où l'opération débute son exécution.
- Une **postcondition** de l'opération `op` est une contrainte (une expression booléenne) qui doit être vraie au moment où l'opération s'apprête à terminer son exécution et à retourner son résultat (de type `R`).

Remarques additionnelles :

- La contrainte associée à une précondition va généralement faire référence à un ou plusieurs arguments.

  Elle peut aussi, le cas échéant, faire référence à des attributs ou rôles de l'entité à laquelle est associée l'opération (contexte).
- La contrainte associée à une postcondition va généralement faire référence au résultat retourné par l'opération.

  Elle peut aussi, le cas échéant, faire référence aux arguments et/ou à des attributs ou rôles.

  Finalement, elle peut aussi faire référence aux valeurs que certaines variables avaient **avant l'exécution de l'opération** (opération avec modification de l'état).

4. Relations clients/fournisseurs et contrats

Décomposer un programme en composants «relativement indépendants» entraine la nécessité pour ces composants de *communiquer entre eux* — s'ils étaient *complètement indépendants*, il n'y aurait pas de collaboration : (

Lorsqu'un un composant communique avec un autre composant, on a généralement une relation de type **client/fournisseur** qui s'établit : un composant (le client) appelle une méthode (une routine) d'un autre composant (le fournisseur) pour que ce dernier lui rende **un service**.

Pour qu'un programme soit correct et robuste, il faut établir des **contraintes** sur les communications et les relations entre clients/fournisseurs : c'est le rôle des **contrats**.

Un contrat implique une entente entre deux parties : chaque partie a des **obligations** face au contrat et, en retour, en retire des **bénéfices** :

> *The definition of* contract *in the design by contract principle is derived from the legal notion of a contract: a univocal, lawful agreement between two parties in which both parties accept obligations, and on which both parties can ground their rights. (Warmer & Kleppe, 2003)*

Fait : Les pré/postconditions d'une opération représentent le **contrat** que doit satisfaire l'opération :

	Obligation	**Bénéfice**
Client	Doit satisfaire la précondition	Est assuré que la postcondition sera satisfaite
Fournisseur	Doit satisfaire la postcondition	Est assuré que la précondition sera satisfaite

Une spécification d'une opération sera donc vue comme un **contrat** : un composant sera considéré **correct** si son comportement est bien celui décrit par la spécification, donc s'il **remplit** le contrat exprimé par cette spécification.

Exemple = Un contrat pour une fonction qui calcule la valeur maximum parmi un ensemble de nombres :

	Obligation	**Bénéfice**
Client	Doit fournir un ensemble non vide	Obtiendra une valeur qui fait partie de l'ensemble et qui est plus grande ou égale à tous les autres nombres
Fournisseur	Doit trouver la valeur appropriée	Est assuré qu'il existe au moins un nombre dans l'ensemble

Heuristique pour l'approche par contrat = le fournisseur devrait être «*paresseux*» :

- Une routine devrait être *stricte* sur ce qu'elle est prête à accepter
  (précondition forte) ;
- Une routine devrait en promettre le moins possible sur ce qu'elle va retourner comme résultat
  (postcondition faible).

Design-by-contract (DBC) style Eiffel (Bertrand Meyer) = utilisation de pré/postconditions dans la conception **et la programmation** :

⇒ Les assertions sont évaluées durant l'exécution

- Violation d'une précondition ⇒ erreur dans le code *du client* ⇒ le service demandé *ne devrait pas* être exécuté.
- Violation d'une postcondition ⇒ erreur dans le code *du fournisseur*.

Note : On verra ultérieurement comment utiliser l'approche DBC en Java avec l'outil `Modern JASS`.

5. Requêtes vs. commandes

- Une **requête** est une opération qui **observe** l'état du système sans le modifier.
- Une **commande** est une opération qui **modifie** l'état du système.

Remarque :

- On suggère généralement — bien que ce ne soit pas toujours possible ou approprié — de définir une opération de façon à ce qu'elle ne combine pas ces deux aspects :
  - Soit elle observe l'état et retourne un résultat, mais sans le modifier.
  - Soit elle modifie l'état, mais sans retourner un résultat.

Les modifications d'état effectuées par une commande peuvent être de différents types :

- Modifier la valeur associée à un attribut d'un objet.
- Créer un nouvel objet (une nouvelle instance).
- Détruire un objet.
- Créer un nouveau lien entre objets.
- Détruire un lien entre objets.

Remarque sur les liens entre préconditions et invariants dans le cas des commandes :
Pour les opérations qui modifient l'état du système (les commandes), il arrive fréquemment qu'une ou plusieurs préconditions servent essentiellement **à assurer que l'invariant soit préservé**.

En d'autres mots, si une commande s'exécute dans un état du système où l'invariant est vrai, alors on doit indiquer les préconditions appropriées pour qu'après l'exécution de la commande, l'invariant soit encore vrai.

6. Opérations publiques vs. auxiliaires

Dans ce qui suit, nous allons suivre la convention suivante (associée à l'utilisation de l'outil `USE`) :

Opérations publiques:

- Les fonctionalités offertes — on dit aussi *fournies* ou *exportées* — par un composant logiciel seront spécifiées à l'aide d'opérations avec pré/postconditions.
- De telles opérations seront considérées comme des **opérations publiques**.
- Les opérations publiques peuvent définir des **requêtes** (observation de l'état), mais aussi des **commandes** (modification de l'état).

Opérations auxiliaires :

- Les opérations auxiliaires sont définies directement par une expression OCL avec «=», par ex. :

```
operations
  nbCoursSuivis(): Integer = coursSuivis->size()
```

- Les opérations auxiliaires définissent uniquement des requêtes (observation) *et ne représentent pas des fonctionalité offertes par un composant logiciel.*
- De telles opérations seront donc considérées comme des **opérations privées**.

Propriétés de l'outil `USE` :

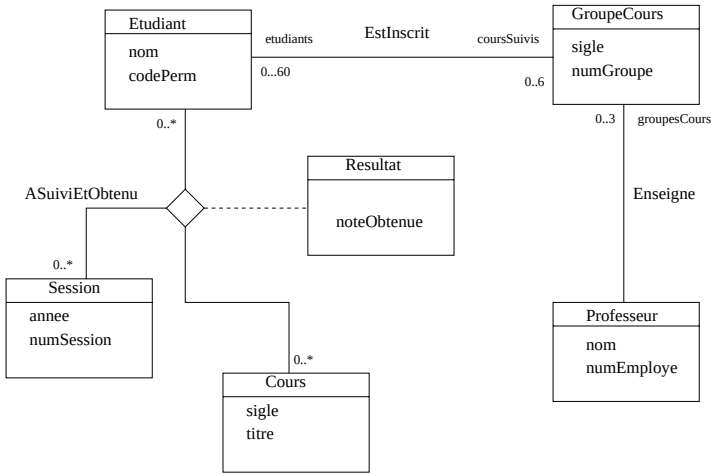
- Une opération auxiliaire peut être utilisée dans les contraintes (invariant, pré/postconditions).
- Une opération publique *ne peut pas* être utilisée dans les contraintes.
- Une opération auxiliaire, en plus d'être définie par une équivalence (avec «=»), peut aussi comporter des pré/postconditions.

Donc, en résumé :

Opération	Définie avec	Requête	Commande	Utilisable dans contraintes	Fonctionnalité exportée
Publique	`pre/post` uniquement	Oui	Oui	Non	Oui
Auxiliaire	`=`	Oui	Non	Oui	Non

7. Exemples : Des requêtes pour le modèle des étudiants, cours, professeurs

Soit le diagramme de classes suivant, pour des étudiants, cours, professeurs, etc. :



Quelques remarques :

- Dans cet exemple, on utilise une relation **ternaire** — donc entre trois (3) entités. À cette relation est associée une information, décrite par l'entité associative `Resultat`, laquelle est spécifiée comme suit (notation `USE`) :

```
associationclass Resultat between
   Etudiant[0..*]
   Cours[0..*]
   Session[0..*]
attributes
   noteObtenue: Integer
end
```

  Ceci signifie donc que l'on aura un `Resultat` à chaque fois qu'on voudra prendre note qu'un `Etudiant` a suivi un `Cours` à une certaine `Session`.

  Note : Une telle approche avec relation ternaire est nécessaire avec `USE` parce que, pour une relation binaire comme celle présentée dans un exemple précédent où l'entité `Resultat` possédait un attribut `session`, il ne peut y avoir *qu'un seul et unique* lien pour chaque paire d'objets en relation. Ceci rend donc impossible d'avoir plusieurs résultats, où chaque résultat contiendrait l'information sur la session.

- Pour simplifier, on va supposer que l'attribut `noteObtenue` d'une entité `Resultat` est de type `Integer` (plutôt qu'une valeur littérale d'un type énuméré comme dans un exemple précédent). Plus précisément, on se restreint aux valeurs comprises entre `0` et `100`, et ce en spécifiant l'invariant «`0 <= noteObtenue and noteObtenue <= 100`».

- On va supposer qu'on peut comparer deux entités de type `Session` comme suit :

```
class Session
  ...
  operations
     vientAvant( s: Session ): Boolean = ...
```


7.1 Des requêtes sans précondition

- Une opération publique qui retourne, pour un groupe-cours, le nombre d'étudiants inscrits dans le groupe :

```
class GroupeCours
  ...
operations
  ...
  nbEtudiants(): Integer
    post: result = etudiants->size()
  ...
```

  Quelques points à souligner :

  – L'identificateur `result` représente, dans la `post`-condition, le résultat produit par la requête — i.e., la valeur retournée par la fonction, dont le type est celui indiqué après «`():`».
  – Si la précondition est absente, alors elle est équivalente à `true`.

  La spécification ci-haut est donc équivalente à la suivante :

```
nbEtudiants(): Integer
  pre:  true
  post: result = etudiants->size()
```

- Une opération qui retourne, pour un étudiant, l'ensemble des sigles de cours dans lesquels il est inscrit :

```
class Etudiant
  ...
operations
  ...
  coursSuivis(): Set(String)
    post: result = coursSuivis.sigle->asSet()
  ...
```

Remarque :

— Les pré/postconditions peuvent être indiquées directement dans la classe. Toutefois, elles peuvent aussi être indiquées à l'extérieur de la classe. Dans ce cas, la signature doit être présente tant dans la classe que dans la spécification :

```
class Etudiant
  ...
operations
  ...
  coursSuivis(): Set(String)
  ...
end

...

context Etudiant::coursSuivis(): Set(String)
  post: result = coursSuivis.sigle->asSet()
```


7.2 Des requêtes avec précondition

Pour certaines requêtes, il peut parfois être impossible de produire un résultat «sensé» si certaines conditions ne sont pas satisfaites. Dans un tel cas, ces conditions sont indiquées comme **préconditions** de l'opération.

- Une opération publique qui, pour un étudiant, retourne le meilleur résultat qu'il ait jamais obtenu. Pour qu'une telle opération fonctionne, il faut que l'étudiant ait déjà suivi au moins un cours :

```
context Etudiant::meilleurResultat(): Resultat
  pre ASuiviDesCours:
    resultat->notEmpty()

  post EstUnDesResultatsObtenus:
    resultat->includes( result )
  post EstLeMeilleurResultat:
    resultat->forAll( r |
                result.noteObtenue >= r.noteObtenue )
```

Quelques remarques :

- Dans une pré/postcondition, tout comme dans un invariant, on peut inclure un identificateur (entre «`pre`» et «:» ou entre «`post` et «:») pour nommer la condition (utilisé dans les messages de vérification).
- Une pré/postcondition peut être divisée en plusieurs clauses (comme pour les invariants). La condition complète est alors obtenue par la conjonction (`and`) des conditions.

- Une opération publique qui, pour un étudiant, retourne le dernier résultat obtenu pour un sigle de cours donné. Pour qu'une telle opération fonctionne, il faut évidemment que l'étudiant ait déjà suivi un cours portant ce sigle :

```
context Etudiant::
        dernierResultatPourCours( sigle: String ): Resultat
    pre ASuiviLeCours:
      resultat.cours.sigle->includes( sigle )

    post EstUnDesResultatsObtenusPourLeCours:
      result.cours.sigle = sigle
    post EstUnDesResultatsDeLEtudiant:
      result.etudiant = self
    post EstLeDernierResultatPourCeCours:
      resultat
         ->select( r | r.cours.sigle = sigle and r <> resul
         ->forAll( r | r.session.vientAvant( result.session
```

Quelques remarques :

- La condition `EstLeDernierResultatPourCeCours` aurait aussi pu être écrite comme suit :

```
post EstLeDernierResultatPourCeCours:
  resultat->forAll( r | r.cours.sigle = sigle and r <> result
                        implies
                        r.session.vientAvant( result.session )
```

- Question : Que se passe-t-il si on omet la condition `EstUnDesResultatsObtenus-PourLeCours`?

7.3 Mise en oeuvre d'opérations avec préconditions

En OCL, il n'est pas possible de spécifier des exceptions. On ne peut donc spécifier que le comportement «normal», en spécifiant explicitement la ou les préconditions appropriées.

Par contre, dans la plupart des **langages de programmation** modernes, il est possible de spécifier des exceptions. Dans un langage de programmation, la mise en oeuvre d'une opération avec précondition peut alors prendre différentes formes.

Exemple : Soit la méthode `meilleurResultat`, qu'on veut mettre en oeuvre en Java. On va supposer que la classe (Java) `Etudiant` possède un attribut (une variable d'instance) `lesResultats`, lequel réfère à la collection des résultats obtenus par l'étudiant dans les divers cours qu'il a déjà suivis. On suppose aussi qu'un `Iterator` approprié est disponible.

Mise en oeuvre sans traitement de la précondition

Dans cette mise en oeuvre, si la précondition n'est pas satisfaite (`lesResultats` est vide), alors aucune mesure spéciale n'est prise par la méthode. Donc, en se fondant sur le code ci-bas, on peut alors supposer qu'une **erreur d'exécution** surviendra au moment où l'appel à la méthode `first()` sera effectué :

```
class Etudiant {
  ...
  public Resultat meilleurResultat() {
    Resultat r = lesResultats.first();
    for( Resultat r1: lesResultats ) {
      if( r1.noteObtenue() > r.noteObtenue() ) {
         r = r1;
      }
    }
    return r;
  }
  ...
}
```

Remarque : Cette technique de mise en oeuvre est évidemment la plus simple... mais la moins intéressante parce que non robuste, et non explicite quant au comportement attendu : le comportement dépend de l'algorithme (comment?) utilisé pour la mise en oeuvre.

Mise en oeuvre avec assertion pour la précondition

Dans une telle mise en oeuvre, si la précondition n'est pas satisfaite, alors une erreur d'assertion sera signalée (si l'option «`-ea`» a été activée), et ce dès le début de l'exécution de la méthode :

```
public Resultat meilleurResultat() {
  assert lesResultats != null
         &&
         lesResultats.length() > 0
         : "Precondition non satisfaite: " +
           "Il n'y a aucun resultat pour l'etudiant!?";

  Resultat r = lesResultats.first();
  for( Resultat r1: lesResultats ) {
    if( r1.noteObtenue() > r.noteObtenue() ) {
       r = r1;
    }
  }
  return r;
}
```

Remarque : C'est l'approche utilisée par plusieurs langages modernes qui permettent la spécification de pré/postconditions — Eiffel, iContract, Modern JASS, etc. : Le programmeur écrit une précondition, laquelle est ensuite transformée en une assertion au début de la routine. Le comportement attendu est donc clair et explicite, indépendant de l'algorithme utilisé pour la mise en oeuvre.

Mise en oeuvre avec spécification d'une exception

Dans une telle mise en oeuvre, si la précondition n'est pas satisfaite, alors une exception est signalée :

```
public Resultat meilleurResultat()
       throws ResultatInexistantException {
  if ( lesResultats == null || lesResultats.length() = 0 )
     throw new ResultatInexistantException();
  }

  Resultat r = lesResultats.first();
  for( Resultat r1: lesResultats ) {
    if( r1.noteObtenue() > r.noteObtenue() ) {
       r = r1;
    }
  }
  return r;
}
```

Remarque : Dans plusieurs approches de spécification (par exemple, VDM, JML), une telle mise en oeuvre correspond en fait à une spécification d'un contrat «multi-clauses» :

- Précondition 1 : la liste des résultats n'est pas vide

  Postcondition 1 : le résultat retourné est le meilleur parmi ceux présents pour le sigle de cours

- Précondition 2 : la liste des résultats est vide

  Postcondition 2 : l'exception `ResultatInexistantException` est signalée

8. Exemples : Des commandes pour le modèle des étudiants, cours, professeurs

8.1 Commandes modifiant des attributs

- Une opération publique pour corriger la note associée à un résultat :

```
context Resultat::corrigerNote( nouvelleNote: Integer )
  pre NouvelleNoteValide:
    0 <= nouvelleNote and nouvelleNote <= 100

  post NoteMiseAJour:
    noteObtenue = nouvelleNote
```

- Une opération publique pour augmenter la note associée à un résultat :

```
contexte Resultat::augmenterNote( increment: Integer )
  pre ValeurAjouteeValide:
    increment > 0 and noteObtenue + increment <= 100

  post NoteMiseAJour:
    noteObtenue = noteObtenue@pre + increment
```

Remarques :

- On suppose ici qu'on veut augmenter la note d'un certain `increment`, positif et tel que la note résultante ne dépasse pas 100.
- La valeur de `noteObtenue` *après l'exécution de la commande* dépend de sa valeur *avant l'éxécution*. Dans une postcondition, le suffixe «`@pre`» permet de référer à un attribut (ou rôle) tel qu'il était **avant** la modification d'état.
- L'expression d'une postcondition est une expression booléenne, pas une «affectation». La postcondition `NoteMiseAJour` suivante aurait été logiquement équivalente à celle indiquée plus haut :
  `noteObtenue - noteObtenue@pre = increment`.

8.2 Commandes créant/détruisant des liens entre objets

- Étant donné un groupe-cours, une opération pour inscrire un étudiant dans ce groupe-cours :

```
context GroupeCours::inscrire( e: Etudiant )
  pre PasDejaInscrit:
    not etudiants->includes(e)
  pre ResteDeLaPlace:
    etudiants->size() < 60

  post UnEtudiantDePlus:
    etudiants->size() = etudiants@pre->size() + 1
  post EstMaintenantInscrit:
    etudiants->includes(e)
```

Remarques :

- Pour inscrire un étudiant, il ne faut pas que l'étudiant soit déjà inscrit et il faut qu'il reste de la place dans le groupe (pour préserver l'invariant sur le nombre maximum d'étudiants).
- Après avoir inscrit l'étudiant, il y a un étudiant de plus inscrit dans le cours et cet étudiant fait partie des étudiants associés au groupe-cours.

– Pour qu'une postcondition telle que la précédente soit suffisante, il faut utiliser une règle implicite — appelée «*frame condition*», qu'on retrouve dans plusieurs langages de spécification — qui dit que **les seules choses qui ont changé suite à l'exécution d'une opération sont celles qui sont indiquées *explicitement* dans la postcondition ; tout le reste est inchangé.**

– Une autre façon d'exprimer les postconditions, sans utiliser la *frame condition*, pourrait être d'utiliser la condition suivante, qui est plus explicite quant à la modification de la relation `EstInscrit` (via spécification du rôle `etudiants`) :

```
post EstMaintenantInscrit: etudiants = etudiants@pre->inclu
```

On aurait aussi pu écrire ce qui suit, même si moins «naturel» :

```
post EstMaintenantInscrit: etudiants->includes(e) and
etudiants->excluding(e) = etudiants@pre
```


- Étant donné un groupe-cours, une opération pour désinscrire un étudiant de ce groupe-cours (annulation d'inscription) :

```
context GroupeCours::desinscrire( e: Etudiant )
  pre DejaInscrit:
    etudiants->includes(e)

  post UnEtudiantDeMoins:
    etudiants->size() = etudiants@pre->size() - 1
  post NEstPlusInscrit:
    not etudiants->includes(e)
```

Remarques :

– Autre façon de spécifier la postcondition, sans utiliser la *frame condition* :

```
post NEstPlusInscrit: etudiants = etudiants@pre->excluding
```

8.3 Commandes créant un nouvel objet

- Une commande qui, pour un étudiant, indique la note qu'il a obtenue dans un cours à une session donnée, retournant aussi l'objet `Resultat` ainsi créé :

```
context Etudiant::
        ajouterResultatPour( c: Cours, s: Sessi
                             note: Integer ): Re
   pre NoteValide:
     0 <= note and note <= 100
   pre PasDejaUnResultatPourCetteSession:
     resultat
       ->select( session = s and cours = c )
       ->isEmpty()

   post ResultatOk:
     resultat
       ->select( session = s and cours = c )
       ->includes(result)
     and
     result.noteObtenue = note
   post NouveauResultat:
     result.oclIsNew()
```


Remarques :

- Pour que l'opération puisse s'exécuter correctement, il faut que la note indiquée soit valide et qu'il n'y ait pas déjà un résultat pour le cours et la session indiqués.
- Cette opération est une commande puisque, bien qu'elle retourne un résultat (de type `Resultat`), ce résultat est un nouvel objet qui n'existait pas dans l'état initial, donc qui ne pouvait pas être observé — on parle parfois de «constructeur» pour une telle opération.
- Un appel tel que «`o.oclIsNew()`» retourne `true` si et seulement si l'objet `o` existe dans l'état **final** (après l'exécution de l'opération) **mais** *n'existait pas dans l'état* **initial** (avant l'exécution de l'opération).
  - Formellement :
    `o.oclIsNew() = o@pre.isUndefined() and not o.isUndefin`
  - Formellement (bis), où `o` est de type `T` :
    `o.oclIsNew() = T.allInstances@pre->excludes(o)`
    `and T.allInstances->includes(o)`

- Une commande qui crée un nouveau professeur à partir de son nom (qui peut ne pas être unique), en générant automatiquement un numéro d'employé approprié :

```
context Professeur::creerProf( nom: String ): Professeur
  post NouveauProf :
    result.oclIsNew()
  post InformationOkEtValide :
    result.nom = nom
    and
    Professeur.allInstances@pre
              .numEmploye
              ->excludes( result.numEmploye )
```

Remarques :

– L'attribut «`@pre`» peut être utilisé pour une collection, y compris `allInstances`.

– Dans le contexte d'un langage de programmation, l'opération précédente serait une **méthode de classe** (en Java, on dirait une méthode *statique*).

Bien que la notion de méthode de classe existe aussi en OCL, elle n'est pas supportée dans l'outil `USE`. L'utilisation d'une méthode telle que la précédente dans une commande `USE` pourrait toutefois se faire via une instance spécifique de la classe `Professeur` (un objet arbitraire, ou *bidon*).

– Une autre solution possible en USE (pour simuler les constructeurs) serait d'introduire une classe additionnelle avec une *méthode de fabrication* appropriée, par exemple :

```
class FabriqueDeProfesseurs
operations
   creerProf( nom: String ): Professeur
   post NouveauProf :
     result.oclIsNew()
   post InformationOkEtValide :
     result.nom = nom
      and
      Professeur.allInstances@pre
                .numEmploye
                ->excludes( result.numEmploye )
end
```

9. Heuristiques de Fox

Fox (*«Introduction to Software Engineering Design—Processes, Principles, and Patterns with UML2»*, Addison-Wesley, 2007) présente les *heuristiques* suivantes pour la spécification d'assertions :

- Préconditions :
  - *Spécifier des restrictions sur les arguments*, par exemple, pour indiquer qu'un argument ne peut pas être `null`, qu'une certaine relation doit exister entre deux arguments, pour assurer que l'exécution d'une commande avec les arguments reçus préservera l'invariant.
  - *Spécifier des conditions qui doivent avoir été établies*, par exemple, pour assurer qu'une autre opération a bien été appelée avant l'appel à l'opération courante, qu'un fichier a bien été ouvert.
  - *Spécifier une précondition vide*, i.e., `true` si aucune condition n'est préalablement requise.

- Postconditions :
  - *Spécifier des relations entre arguments et résultats*, donc décrire les liens entre les entrées et sorties de l'opération.
  - *Spécifier des restrictions sur les résultats*, par exemple, intervalle de valeurs possibles.
  - *Spécifier des modifications aux paramètres*, par exemple, des changements d'état effectués sur les objets reçus en argument.
- Invariants :
  - *Spécifier des restrictions sur les attributs*, par exemple, intervalle de valeurs possibles.
  - *Spécifier des relations entre les attributs*, par exemple, la taille d'un attribut collection doit être inférieure à la valeur d'un attribut indiquant la taille maximum.

OCL : Chapitre VI

Spécification de fonctions

0. Introduction

Remarques :

- Dans ce chapitre nous allons voir comment spécifier des **fonctions** et des **bibliothèques de fonctions**, et ce en utilisant des **contrats** exprimés en notation USE/OCL.
- Nous verrons aussi que certaines opérations s'expriment plus simplement à l'aide de spécifications **implicites**, plutôt qu'avec des spécifications **explicites**.

1. Fonction pure vs. opération avec effet de bord

Soit `o` un objet et `m` une méthode de `o`.

- On dit que `m` est une **fonction pure** — fonction au sens mathématique — si `m` n'a aucun effet sur son environnement (entre autres, ne modifie aucun attribut de `o`) et si le résultat retourné est déterminé uniquement par les attributs de `o` et par les valeurs reçues en arguments à l'appel de `m`.

  Autrement, on dit que `m` est une **opération avec effet de bord** — on dit aussi une méthode **impure**.
- `m` est dite **idempotente** si elle a le même «effet» qu'on l'exécute une, deux ou plusieurs fois de suite (exécutions immédiatement consécutives).

Faits :

- Une fonction pure est nécessairement idempotente.
- Une méthode idempotente n'est pas toujours pure.
- Si `m` est une fonction pure, alors :

```
o.m() = o.m()
```

- Si `m` est pure et retourne un nombre, alors :

```
o.m() + o.m() = 2 * o.m()
```

Exemples/contre-exemples en Java :

```
class PureImpure {
    private int x = 0;

    public int m1() {
        return x;
    }

    public int m2( int y ) {
        return 2 * x + y;
    }

    public int m3() {
        x += 1;
        return x;
    }

    public void m4() {
        x = 0;
    }
}
```

- `m1` et `m2` sont des fonctions pures et idempotentes.
- `m3` n'est ni pure, ni idempotente.
- `m4` est idempotente, mais pas une fonction pure.

2. Bibliothèques de fonctions

- Il est possible, dans un langage orienté objets, qu'une classe ne définisse aucun objet mais serve uniquement à définir des fonctions. Une telle classe définit donc une sorte de **bibliothèque de fonctions**.
- Sauf exception (voir plus bas), les fonctions d'une bibliothèque de fonctions sont généralement des fonctions pures.

  De plus, ces fonctions ne sont habituellement pas des méthodes d'instance, mais sont plutôt des méthodes de classe (méthodes statiques).
- Exemple : la classe `java.lang.Math` définit un ensemble de fonctions mathématiques ($\approx$ 60 méthodes).
  - Toutes ces fonctions sont des méthodes `static`.
  - Toutes ces méthodes sont des fonctions pures, à l'exception de la méthode `random()`.
  - La méthode `random()` permet de générer des nombres pseudo-aléatoires, donc doit retourner un résultat différent à chaque appel :

    ```
    public static double random();

    // Returns a double value with a positive sign,
    // greater than or equal to 0.0 and less than 1.0.
    ```

3. Méthodes de classe en OCL/USE

- La documentation Java définit la notion de *méthode de classe* comme suit :[2]

  Class Methods
  The Java programming language supports static methods as well as static variables. Static methods, which have the static modifier in their declarations, should be invoked with the class name, without the need for creating an instance of the class [...]

- En OCL/USE, la notion de méthode de classe (méthode statique) n'est pas disponible. Toute méthode doit nécessairement être une méthode d'instance, donc doit être rattachée explicitement à une instance d'une classe appropriée.

- Fait : Une méthode de classe peut être simulée par une méthode d'instance d'une classe ne possédant pas d'attribut et telle que la méthode ne dépend pas de l'instance sur laquelle la méthode s'applique.

  Une telle méthode, qui ne dépend que des arguments qu'elle reçoit en argument (puisque ni la classe ni l'instance ne possèdent d'attributs), définit alors une fonction pure.

[2] http://java.sun.com/docs/books/tutorial/java/java00/classvars.html

Exemple :

- Une classe regroupant diverses fonctions :

```
class Math
  operations
    signe( x: Integer ): Integer
      post ResultatNegatif:
        x < 0 implies result = -1
      post ResultatNul:
        x = 0 implies result =  0
      post ResultatPositif:
        x > 0 implies result = +1

    somme( x: Integer, y: Integer ): Integer
      post ResultatEstSomme:
        result = x + y

    produit( x: Integer, y: Integer ): Integer
      post ResultatEstProduit:
        result = x * y
    ...
```

- Utilisation des fonctions (fichier de tests) :

```
!create biblioMath1, biblioMath2: Math

!openter biblioMath1 signe( -23 )
!opexit -1

!openter biblioMath2 signe( -23 )
!opexit -1

!openter biblioMath2 produit( 2, 34 )
!opexit 68
...
```

4. Spécifications explicites vs. implicites

- Une fonction pure n'ayant aucun effet de bord, il s'agit donc d'une forme de **requête**.
- Fait : En OCL/USE, il est possible de spécifier une requête soit à l'aide de pré/post-conditions, soit à l'aide d'une expression OCL d'un type approprié.
- Selon la «façon» dont est spécifiée l'opération, on parle alors de deux formes de spécification :
  - Spécification **explicite** = On donne directement une expression qui indique la valeur du résultat.
  - Spécification **implicite** = On spécifie diverses **propriétés** qui sont satisfaites par le résultat, sans donner explicitement sa valeur.

    Dans ce dernier cas, **on n'a donc pas** une expression style :

    «`result = ...`»

- Fait : En OCL/USE, les spécifications explicites peuvent être écrites soit avec une définition — avec le symbole «=» suivi d'une expression — soit avec une postcondition.
- Fait : En OCL/USE, les spécifications explicites **sont exécutables** et **constructives** — donc si on les évalue, elles produisent explicitement le résultat approprié, contrairement aux spécifications implicites qui font simplement décrire les propriétés du résultat sans le construire.

Exemple :

- Trois variantes d'une fonction qui retourne la valeur maximum parmi ses trois arguments.
  - `max1` : spécifiée de façon *explicite* avec une expression.
  - `max2` : spécifiée de façon *explicite* avec une postcondition.
  - `max3` : spécifiée de façon *implicite* avec une postcondition.

```
class Math
   ...
   max1( x1: Integer, x2: Integer, x3: Integer ): Integer =
      x1.max(x2).max(x3)

   max2( x1: Integer, x2: Integer, x3: Integer ): Integer
     post PlusGrandDesTrois:
       result = x1.max(x2).max(x3)

   max3( x1: Integer, x2: Integer, x3: Integer ): Integer
      post EstUnDesArguments:
        Set{x1,x2,x3}->includes(result)
      post EstPlusGrand:
        Set{x1,x2,x3}->forAll( x | x <= result )
   ...
end
```

Question : Quel est un avantage de la troisième forme?

5. Des fonctions mathématiques

La classe qui contient les fonctions :

```
class Math
operations
  racineCarreeEntiere( x: Integer ): Integer

  racineCarree( x: Real, precision: Real ): Real
end
```

- Une fonction pour calculer, approximativement, la racine carrée entière d'un nombre entier — retourne le plus grand entier qui, lorsque mis au carré, ne dépasse pas `x` :

```
context Math::
  racineCarreeEntiere( x: Integer ): Integer
   pre ArgumentNonNegatif:
     x >= 0

   post RacinePositive:
     result >= 0
   post AuCarreNeDepassePasArgument:
     result * result <= x
   post PlusGrandPossible:
     x < (result + 1)*(result+1)
```

- Une fonction pour calculer, approximativement, la racine carrée d'un nombre réel :

```
context Math::
  racineCarree( x: Real, precision: Real ): Real
   pre XNonNegatif:
     x >= 0.0
   pre PrecisionPositive:
     precision > 0.0

   post RacinePositive:
     result >= 0.0
   post BonneRacine:
     approxEgal( result*result, x, precision )



  --
  -- Operation auxiliaire: Definie dans la classe ci-haut..
  --
  approxEgal( x1: Real, x2: Real, precision: Real ): Boole
    = (x1 - x2).abs() <= precision
```

Fait : Le style *implicite* permet souvent de spécifier/décrire ce qui doit être produit comme résultat (quoi?), sans décrire la façon dont ce résultat sera calculé (comment?).

Question : Pourrait-on spécifier `racineCarree` de façon explicite? Si oui, de quelle façon?

6. Des fonctions sur des séquences d'entiers

La classe qui contient les fonctions :

```
class FonctionsSequence
operations
  ...
end
```

- Une fonction qui retourne l'élément minimum de la séquence d'entiers reçue en argument.
  - Spécification implicite :

```
minimum( seq: Sequence(Integer) ): Integer
 pre PasVide:
   seq->notEmpty()

 post EstElementDeLaSequence:
   seq->includes(result)
 post EstPlusPetitOuEgalAuxAutres:
   seq->forAll( x | result <= x )
```

  - Spécification explicite :

```
minimum( seq: Sequence(Integer) ): Integer =
  seq->iterate( x; leMin: Integer = seq->any(true) |
       if x < leMin then x else leMin endif )
```

- Une fonction qui produit une version triée (en ordre croissant) de la séquence d'entiers reçue en argument.
  - Spécification implicite :

```
trier( seq: Sequence(Integer) ): Sequence(Integer)
 post MemeLongueur:
   result->size() = seq->size()
 post MemesOccurrences:
   result
     ->forAll( x | result->count(x) = seq->count(x) )
 post EstEnOrdre:
   Set{1..result->size()-1}
      ->forAll( i | result->at(i) <= result->at(i+1) )
```

  - Spécification explicite :

```
trier( seq: Sequence(Integer) ): Sequence(Integer) =
  seq->iterate( x; res: Sequence(Integer) = Sequence{} |
       inserer(x, res) )

inserer( x: Integer, seq: Sequence(Integer) ):Sequence(Integer)
  = if seq->isEmpty() or x < seq->first()
     then seq->prepend(x)
     else inserer( x,tail(seq) )->prepend(seq->first())
    endif

 tail( seq: Sequence(Integer) ): Sequence(Integer) =
   if seq->size <= 1
     then oclEmpty(Sequence(Integer))
     else seq->subSequence(2,seq->size())
   endif
```


- Une fonction qui retourne un des index (n'importe lequel) où apparaît, dans la séquence, l'élément indiqué.
  - Spécification implicite :

```
indexElement( seq: Sequence(Integer), elem: Integer ): I
 pre PresentDansLaSequence:
   seq->includes(elem)

 post IndexDonneLElement:
   seq->at(result) = elem
```

  - Spécification explicite :

```
indexElement( seq: Sequence(Integer), elem: Integer ):
            Integer
  = Set{1..seq->size()}->any( i | seq->at(i) = elem )
```

- Une fonction qui retourne un des index (n'importe lequel) où apparaît, dans la séquence, l'élément maximum de la séquence.
  - Spécification implicite :

```
indexMaximum( seq: Sequence(Integer) ): Integer
 pre PasVide:
   seq->notEmpty()

 post IndexDonneElementMaximum:
   seq->forAll( x | seq->at(result) >= x )
```

  - Spécification explicite :

```
indexMaximum( seq: Sequence(Integer) ): Integer =
  Set{1..seq->size()}->iterate(
     i;
     res: Tuple(leMax: Integer, pos: Integer)
        = Tuple{leMax=seq->first(), pos=1}
     |
     if seq->at(i) >= res.leMax
       then Tuple{leMax=seq->at(i), pos=i}
       else res
     endif
  ).pos
```


OCL : Chapitre VII

Spécification de types abstraits : Classes d'objets vs. collections de valeurs

0. Introduction : Types abstraits et modélisation abstraite

- Une classe, que ce soit en OCL ou en Java, définit une forme de **type**.

  Un type permet de dénoter un ensemble d'entités (d'objets, de valeurs) ainsi que les opérations permises sur ces entités.
- On utilise le terme **type abstrait** lorsqu'on décrit et spécifie un type en faisant abstraction de sa mise en œuvre effective — donc lorsqu'on s'intéresse aux propriétés/relations des/entre les opérations, plutôt qu'à leurs détails de mise en œuvre.
- Exemple : Un objet `Pile` possède la propriété LIFO *(Last-In/First-Out)*, pouvant être décrite comme suit :
  - `p.empiler(x).sommet() = x`
  - `p.sommet() = x`
    `implies`
    `p.empiler(y).depiler().sommet() = x`

- Une approche possible pour la spécification de types abstraits est l'approche dite de **modélisation abstraite**.

  Dans une telle approche, on décrit tout d'abord (de façon explicite, bien qu'abstraite) l'«état» (le contenu, les attributs) des entités : c'est ce qu'on appelle **le modèle abstrait**.

  Ensuite, on décrit «l'effet» de chacune des opérations du type sur le modèle abstrait, en utilisant des contrats spécifiés avec des pré/postconditions.

- On dit que la description de l'état (du modèle) est *abstraite* dans la mesure où on utilise des types de données de haut niveau, types qui représentent des structures mathématiques (donc des **collections de valeurs**), par exemple, ensembles, séquences, dictionnaires, etc.

- Les langages de spécification formelle les plus connus utilisant l'approche de modélisation abstraite sont VDM et Z.

  C'est aussi une approche utilisée par certains langages de programmation qui supportent la spécification de contrats, par ex., utilisent cette approche : Java/JML.

- Dans le présent chapitre nous allons voir, à l'aide d' exemples exprimés principalement *en OCL*, comment spécifier des types abstraits en utilisant l'approche de modélisation abstraite.

  Nous verrons aussi que, même si on utilise des classes pour spécifier des types, ces classes *ne représentent pas nécessairement des objets*.

- **Note** : Les spécifications de types abstraits que nous allons examiner peuvent être utilisées lors de l'étape d'analyse, mais aussi (et surtout) aux étapes de conception et de construction des logiciels.

1. Objets vs. valeurs

- On retrouve deux genres de types abstraits :

Types mutables	=	Classes d'objets
Types immuables	=	Collection de valeurs

1.1 La notion d'objet = Entité mutable

Un (vrai!) **objet** — au sens *conceptuel* — est caractérisé par les propriétés suivantes :

- Un objet est une entité qui possède **une identité fixe** — son *nom* ou, en termes de mise en œuvre, son adresse.
- Un objet possède un **cycle de vie** : à un certain moment, l'objet est créé (naissance) ; l'objet peut parfois ensuite être détruit (mort).
- Un objet possède un état interne, une *mémoire*, **qui évolue dans le temps ⇒ entité mutable**.
- Un objet peut recevoir des messages et faire certains traitements en réponse à la réception de ces messages — donc un objet fournit des opérations.
- Un objet peut envoyer des messages à d'autres objets — un objet peut utiliser les opérations fournies par d'autres objets.

1.2 La notion de valeur : Entité immuable

Une **valeur** — au sens *conceptuel* — possède les caractéristiques suivantes :

- Une valeur est une entité qui existe *de tout temps*.

  Exemple : Au sens mathématique, le nombre 4 ou l'ensemble {4,6,9} existent de tout temps, ne naissent pas et ne meurent pas.
- Une valeur est une entité *qui ne change pas*.

  Exemple : Le fait de calculer «2+3» ne modifie ni 2, ni 3, et ne crée pas un nouveau 5.
- Une valeur est caractérisée uniquement par ses attributs, et non par son identité — en fait, la notion d'identité ne s'applique pas aux valeurs.

  Exemple : Les valeurs 5, 3+2, 25/(3+2) sont toutes égales et identiques, *parce qu'indistinguables*. Les ensembles {2,3,9}, {8+1,3,3–1}, {2,0+3,2,2,3*3} sont tous égaux et identiques, *parce qu'indistinguables*.

Les **valeurs** représentent donc des **entités immuables** :

> Immuable (adj.) Qui reste identique à soi-même ; qui ne peut éprouver aucun changement. V. Constant, durable, inaltérable, invariable. (Le Petit Robert)

1.3 Objets ou valeurs?

Quand doit-on utiliser des objets vs. des valeurs?

- Objets :

  On utilise des objets pour représenter des entités **du monde réel** (du domaine d'application) qui ont un comportement **dynamique** — on parle souvent dans ce cas d'*objets d'affaires* :

  – Comptes bancaires (et leurs soldes).
  – Boites vocales (et leurs messages).
  – Emprunts faits par un uager d'une bibliothèque.
  – Etc.
  – . . .

- Valeurs :

  On utilise des valeurs pour représenter des entités **immuables**, souvent associées à des notions mathématiques :

  – Nombres.
  – Dates.
  – Valeurs monétaires.
  – Chaînes de caractères — mais voir plus loin.
  – Etc.

Fait : Les méthodes d'un type qui définit une collection de valeurs **sont toutes des fonctions pures**.

1.4 Objets vs. valeurs : Exemples des chaînes en Java

Certaines notions peuvent parfois être modélisées et/ou mises en œuvre tant par des objets que par des valeurs. Un exemple est celui des chaînes de caractères en Java :

- `String` ⇒ valeurs
- `StringBuffer` ⇒ objets

Note : Plus précisément, dans le cas des `String` Java, certains auteurs utilisent aussi le terme *objet immuable* (sic) — en anglais, *value object* ou *immutable object*. Ceci s'explique par le fait que, pour des raisons de mise en œuvre, les valeurs chaînes de Java peuvent effectivement être créées (allouées, via des `new`) de façon dynamique (caractéristique d'un objet).

Différences de comportement entre `String` et `StringBuffer` :

```
// String => Valeur
String s = "";
assert s.equals( "" );
s.concat( "abc" );
s.concat( "def" );
assert s.equals( "" );

s = s.concat( "abc" );
s = s.concat( "def" );
assert s.equals( "abcdef" );
assert s != "abcdef";

String autre = s;
s = s.concat( "000" );
assert s.equals( "abcdef000" );
assert autre.equals( "abcdef" );
```

```
// StringBuffer => Objet.
StringBuffer buf = new StringBuffer( 10 );
assert buf.toString().equals( "" );
buf.append( "abc" );
buf.append( "def" );
assert buf.toString().equals( "abcdef" );

StringBuffer autreBuf = buf;
buf.append( "000" );
assert buf.toString().equals( "abcdef000" );
assert autreBuf.toString().equals( "abcdef000" );
assert autreBuf == buf;
```


Signatures des méthodes `concat()` vs. `append()` :

```
// String::concat
String concat(String str)
Concatenates the specified string to the end of this string
```

```
// StringBuffer::append
StringBuffer append(String str)
Appends the specified string to this character sequence.
```

Donc :

- Un `String` étant une **valeur**, l'opération retourne un résultat *et ce sans modifier la chaîne initiale.*
- Un `StringBuffer` étant un **objet**, l'opération modifie l'objet et retourne comme résultat l'*identité* (inchangée) de cet objet.

```
String s = "";
assert s != s.concat( "abc" );
assert ! s.equals( s.concat( "abc" ) );
```

```
StringBuffer buf = new StringBuffer( 10 );
assert buf == buf.append( "abc" );
```

2. Des objets pour des comptes bancaires (simples)

```
class Compte
attributes
  numero: NumeroDeCompte
  client: Client
  solde: Montant

operations
  deposer( montant: Montant )
     post SoldeAugmente:
       solde.valeur = solde@pre.valeur + montant.valeur

  retirer( montant: Montant )
     pre SoldeSuffisant:
       solde.valeur >= montant.valeur
     post SoldeDiminue:
       solde.valeur = solde@pre.valeur - montant.valeur

  solde(): Montant
    post SoldeOk:
      result = solde

constraints
  inv NumeroUnique: Compte.allInstances->isUnique( numero )
end
```

Remarque : Il s'agit bien d'**objets** parce que les postconditions des contrats décrivent des *changements d'état*, et ce en spécifiant les relations qui existent entre l'état après l'opération et l'état avant (utilisation de `@pre`).

Un objet étant une entité avec un cycle de vie, il doit donc pouvoir être créé (naissance ⇒ `oclIsNew()`) :

```
class Banque
operations
  creerCompte( c: Client ): Compte
    post ClientDefini:
      result.client = c
    post SoldeNul:
      result.solde.valeur = 0.0
    post NumeroDistinctDesAutres:
      NumeroDeCompte.allInstances@pre->excludes(result.numero)
    post NouveauCompte:
      result.oclIsNew()
end

--
-- Types primitifs (atomiques).
--
class NumeroDeCompte end

class Montant
attributes
  valeur: Real
constraints
  inv JamaisNegatif: valeur >= 0.0
end

class Client end
```

Remarque : La classe `Montant` définit des entités qui sont des **valeurs** : Aucune opération ne permet de modifier l'état.

3. Des valeurs pour des fractions numériques

```
class Fraction
attributes          -- Note: Noms courts pour mise en page
  num:   Integer    -- => Numerateur
  denom: Integer    -- => Denominateur

operations
  plus( f: Fraction ): Fraction
    post: result.num   = num * f.denom + f.num * denom and
          result.denom = denom * f.denom

  fois( f: Fraction ): Fraction
    post: result.num   = num * f.num and
          result.denom = denom * f.denom

  inverse(): Fraction
    pre : num <> 0
    post: result.num * num = result.denom * denom

  egale( f: Fraction ): Boolean
    post: result = (num * f.denom = f.num * denom)

constraints
  inv DenominateurJamaisNul: denom <> 0
end
```


Remarque : Il s'agit bien de **valeurs** parce que les post-conditions des contrats ne décrivent pas de *changement d'état* (pas de référence à `@pre`). Les postconditions décrivent uniquement le résultat (identificateur `result`) **en fonction** des attributs des deux valeurs reçues en argument (`self` et `f`). De plus, `oclIsNew()` n'est pas utilisé.

Remarque : On peut aussi voir les choses «d'un autre point de vue», à savoir *en termes de contraintes à respecter lorsqu'on spécifie un type immuable.*
Une spécification d'un type immuable :

- Ne doit jamais utiliser `@pre`.
- Ne doit jamais utiliser `oclIsNew()`.
- Doit décrire de façon *complète et détaillée* les attributs du résultat retourné, peu importe son type.

Ce dernier point est important, et découle de ce que la propriété de *frame condition* **ne s'applique qu'à des objets**. Cette propriété stipule qu'on peut supposer que les attributs non spécifiés d'un objet sont inchangés — donc pas de différence entre avant et après, si rien n'est indiqué. Mais dans le cas d'un type immuable, **il n'y a pas d'avant vs. après, il y a simplement des arguments et un résultat.**

4. Des objets et valeurs pour des piles non bornées

- On dit d'un type de données modélisant une forme de collection — regroupement d'éléments du même type, par exemple, ensembles, séquences, piles — qu'il définit des collections **bornées** lorsqu'il existe une limite *a priori* sur le nombre d'éléments que peut contenir une collection.

  Lorsqu'aucune limite n'existe *a priori* (sauf, évidemment, l'espace mémoire disponible sur la machine ;), on dit alors que le type est **non borné**.

- Certains types abstraits — par exemple, certains types pour des *collections* — peuvent être définis soit sous forme d'objets, soit sous forme de valeurs.
  - Exemples, tirés de la section 7 des notes de cours, chapitre «Assertions et tests» :
    * Bibliothèque `OclCollections` : Valeurs.
    * Bibliothèque `java.util` : Objets.
  - Exemples, présentés plus en détails dans les sections qui suivent :
    * Piles sous forme de valeurs : et
    * Piles sous forme d'objets : et

4.1 Des objets pour des piles non bornées

```
abstract class Pile
attributes
  elems: Sequence(T)      -- Les elements empiles.

operations
  estVide(): Boolean =
    elems->size() = 0

  empiler( x: T )
    post ElementAjouteAuSommet :
      elems = elems@pre->prepend(x)

  depiler()
    pre PilePasVide :
      not estVide()
    post AncienSommetRetire :
      elems@pre = elems->prepend( elems@pre->first() )

  sommet(): T
    pre PilePasVide :
      not estVide()
    post SommetEstLElementAuDebut :
      result = elems->first()
end

class PileNonBornee < Pile
end
```

4.2 Des valeurs pour des piles non bornées

```
abstract class Pile
attributes
  elems: Sequence(T)

operations
  estVide(): Boolean =
    elems->size() = 0

  empiler( x: T ): Pile
    post ElementAjouteAuSommet :
      result.elems = elems->prepend( x )

  depiler(): Pile
    pre PilePasVide :
      not estVide()
    post AncienSommetRetire :
      result.elems->prepend( sommet() ) = elems

  sommet(): T = elems->first()
    pre PilePasVide :
      not estVide()
    post SommetEstLElementAuDebut:
      result = elems->first()
end

class PileNonBornee < Pile
end
```


Remarque : On a pu écrire `sommet` tel qu'indiqué plus haut puisque l'outil USE permet que des préconditions soient indiquées même pour des opérations auxiliaires (définies directement avec un «=»).

L'intérêt d'avoir une opération auxiliaire définie de cette façon est qu'elle peut ensuite être utilisée dans les pré/postconditions des autres opérations, par exemple :

```
depiler(): Pile
  pre PilePasVide :
    not estVide()
  post AncienSommetRetire :
    result.elems->append( sommet() ) = elems
```

5. Des objets et valeurs pour des piles bornées

- Une pile est dite *bornée* lorsqu'elle ne peut contenir qu'un nombre limité d'éléments.
- Le fait de fixer un nombre maximum d'éléments implique l'ajout d'un **invariant** : n'importe quelle pile du type doit satisfaire la propriété de n'avoir jamais plus d'éléments que le nombre maximum permis.
- De façon générale, la présence d'un invariantimplique quesi une opération peut avoir pour effet de modifier des propriétés qui sont liées à l'invariant, alors cette opération doit avoir une **précondition** qui assure que **l'invariant sera préservé** : si l'invariant est vrai avantl'opération, alors **il sera encore vrai après** l'exécution de l'opération.

- Exemple : Dans notre cas (piles bornées), ceci signifie qu'une opération qui augmente le nombre d'éléments (i.e., `empiler`) doit avoir une précondition appropriée.

5.1 Des objets pour des piles bornées

```
class PileBornee < Pile
attributes
  tailleMax: Integer  -- Nombre max. d'elements empilables.

operations
  empiler( x: T )
    pre PilePasDejaPleine:
      elems->size() < tailleMax
    post ElementAjouteAuSommet :
      elems = elems@pre->prepend( x )

constraints
  inv NombreElementsEmpilesPasSuperieurATailleMax:
    elems->size() <= tailleMax
end


class FabriqueDePiles
operations
  ...
  pileBornee( tailleMax: Integer ): PileBornee
    post PileEstVideBorneeEtNouvelle:
      result.estVide()
      and
      result.tailleMax = tailleMax
      and
      result.oclIsNew()
end
```

Remarque : Objets ⇒ Présence d'une méthode pour *donner naissance* à de **nouveaux** objets : ici, la méthode de fabrication `pileBornee`.

5.2 Des valeurs pour des piles bornées

```
class PileBornee < Pile
attributes
  tailleMax: Integer

operations
  empiler( x: T ): PileBornee
    pre PilePasDejaPleine :
      elems->size() < tailleMax
    post ElementAjouteAuSommet :
      result.elems = elems->prepend( x )

constraints
  inv NombreElementsEmpilesPasSuperieurATailleMax:
    elems->size() <= tailleMax
end

class FabriqueDePiles
operations
  ...
  pileBorneeVide( tailleMax: Integer ): PileBornee
    post:
      result.estVide() and result.tailleMax = tailleMax
end
```

Remarques :

- Tous les attributs de la pile retournée en résultat par une opération doivent être *spécifiés explicitement*, puisque la propriété de *frame condition* ne s'applique pas à des types immuables. Il faut donc spécifier, pour `result`, quels sont les éléments présents sur la pile (`elems`) ainsi que sa `tailleMax`. Et il faut donc aussi redéfinir `depiler`.

- Le fait de définir un type immuable implique la présence d'une méthode pour dénoter une pile vide (dans la classe `FabriqueDePiles`), et cette pile n'est pas un *nouvel objet*, donc pas de `oclIsNew()` — 2+3 ne produit pas un nouveau 5!

5.3 Comparaison entre les différentes sortes de piles

En résumé (tableau synthèse) :

	Piles non bornées	**Piles bornées**

Type mutable (objets)	`empiler( x: T )`
	`  post ElementAjouteAuSommet :`
	`  elems`
	`  = elems@pre->prepend(x)`
		`empiler( x: T )`
		`  pre PilePasDejaPleine:`
		`  elems->size() < tailleMax`
		`  post ElementAjouteAuSommet :`
		`  elems`
		`  = elems@pre->prepend( x )`

Type immuable (valeurs)	`empiler( x: T ): Pile`
	`  post ElementAjouteAuSommet :`
	`  result.elems`
	`  = elems->prepend( x )`
		`empiler( x: T ): PileBornee`
		`  pre PilePasDejaPleine :`
		`  elems->size() < tailleMax`
		`  post ElementAjouteAuSommet :`
		`  result.elems`
		`  = elems->prepend( x )`

6. Une autre approche de spécification des types abstraits

L'approche de modélisation abstraite n'est pas la seule approche qu'on peut utiliser pour spécifier des types abstraits :

- Pour les *collections de valeurs*, on peut aussi utiliser des approches dites de «spécifications algébriques» (e.g., Larch).
- Pour des *classes d'objets*, on peut aussi utiliser le style de spécification proposé par Bertrand Meyer dans son approche DBC (*Design By Contract*) — approche décrite plus en détails dans ce qui suit.

L'approche DBC repose sur la distinction importante entre *requêtes* et *commandes* :

- **Requête** : Opération qui **observe** l'état d'un objet **sans le modifier** — opération appelée aussi **observateur**.

  Donc : Une requête devrait être une fonction pure.
- **Commande** : Opération qui **modifie** l'état d'un objet, **sans retourner de résultat** — opération appelée aussi **mutateur**.

6.1 Approche style DBC : Effets sur les requêtes primitives

Notion de **requête dérivée** vs. **requête primitive** : :

- Une requête **dérivée** est une requête qui peut s'exprimer *à l'aide d'autres requêtes plus simples.*

  Exemple : Supposons un type pour des objets piles qui définit une requête `taille()`, laquelle retourne le nombre d'éléments empilés. La requête `estVide()` serait alors une requête dérivée, puisque :

  ```
  estVide() = (taille() = 0)
  ```

- Une requête **primitive** est une requête qui ne peut pas s'exprimer *à l'aide d'autres requêtes plus simples.*

Stratégie de spécification DBC :

1. On identifie tout d'abord les **requêtes primitives**.
2. On spécifie ensuite le comportement des **requêtes dérivées** et des **commandes** en fonction de leurs effets sur les requêtes primitives.

Plus précisément :

- **Pour chaque requête primitive**, on déclare sa signature (nom de la requête, types des arguments, type du résultat) mais sans décrire de façon formelle son interprétation (sa signification, son effet, sa valeur) — possiblement en utilisant *un simple commentaire.*
- **Pour chaque requête dérivée et chaque commande**, on déclare sa signature et on spécifie son comportement par une série de clauses, chaque clause décrivant généralement l'effet de l'opération **sur une des requêtes primitives**.

Remarque : Les exemples qui suivent sont écrits en OCL/USE, et ce pour éviter d'introduire une nouvelle notation. Il faut toutefois noter qu'**il ne s'agit pas de spécifications légales en OCL/USE**, puisque le style DBC n'est pas directement supporté — le suffixe `@pre`, en OCL/USE, ne peut s'appliquer qu'à un attribut, pas à une opération.
On verra ultérieurement comment utiliser cette approche en Java avec Modern JASS.

6.2 Des objets pour piles bornées

```
class PileBornee
operations

  // Requetes primitives.

  tailleMax(): Integer
    // post: result = Nombre max. d'elements pouvant etre empi

  nbElements(): Integer
    // post: result = Le nombre d'elements empiles

  elem( i: Integer ): T
    pre: 1 <= i <= nbElements()
    // post: result = Le ieme element empile sur la pile


  // Requetes derivees.

  estVide(): Boolean =
    post:
     result = (nbElements() = 0)

  sommet(): T
    pre PilePasVide :
      not estVide()
    post SommetEstDernirElementEmpile :
      result = elem( nbElements() )
```



```
  // Commandes.

  empiler( x: T )
    pre PilePasDejaPleine:
      nbElements() < tailleMax()
    post UnElementDePlus :
      nbElements() = nbElements()@pre + 1
    post ElementAjouteAuSommet :
      elem( nbElements() ) = x

  depiler()
    pre PilePasVide :
      not estVide()
    post AncienSommetRetire :
      nbElements() = nbElements()@pre - 1


constraints
  inv TailleMaxNonNulle:
    0 < tailleMax()
  inv NombreElementsEmpilesPasSuperieurATailleMax:
    nbElements() <= tailleMax()
end
```

6.3 Des objets pour files FIFO non bornées

Remarque : FIFO = *First-In/First-Out*

```
class FileNonBornee

operations
  // Requetes primitives.

  nbElements(): Integer

  elem( i: Integer ): T


  // Requetes derivees.

  estVide(): Boolean
    post: result = (nbElements() = 0)

  tete(): T
    pre FilePasVide :
      not estVide()
    post TeteEstElementAuDebut :
      result = elem( 1 )

  queue(): T
    pre FilePasVide :
      not estVide()
    post QueueEstElementALaFin :
      result = elem( nbElements() )
```



```
  // Commandes.

  ajouter( x: T )
    post UnElementDePlus :
      nbElements() = nbElements()@pre + 1
    post ElementAjouteALaFin:
      queue() = x

  retirer()
    pre FilePasVide :
      not estVide()
    post UnElementDeMoins :
      nbElements() = nbElements()@pre - 1
    post PremierElementRetire :
      Set{1..nbElements()}->forAll( i |
        elem(i) = elem(i+1)@pre )

end
```

6.4 Des objets pour des dictionnaires

Un type de données souvent utilisé est le **dictionnaire** (*map* en anglais).

Un dictionnaire permet d'associer une **définition** (disons de type `D`) à une **clé** (disons de type `C`) et permet ensuite, à partir de cette *clé*, de retrouver *directement* la *définition associée*.

Un dictionnaire `dict` peut être vu comme une forme de fonction :

$$\texttt{dict} : \texttt{C} \rightarrow \texttt{D}$$

Un dictionnaire `dict` peut aussi être vu comme une forme de *table*, une sorte de tableau généralisé où les index sont d'un type arbitraire (pas juste des entiers) :

```
dict: ARRAY[C] OF D
```


6.4.1 Exemple Ruby d'utilisation d'un dictionnaire (`Hash`)

```
require 'test/unit'

class TestAnalyseur < Test::Unit::TestCase

  def test_dictionnaire()
    d1 = Hash.new              # Creation d'un dictionnaire vide

    d1[""] = 0                 # Association de definitions a des cles
    d1["abc"] = 3
    d1["0123456789"] = 10

    assert_equal 3, d1.size()  # Nombre de cles

    assert d1.has_key?( "" )   # Cles definies?
    assert d1.has_key?( "abc" )
    assert d1.has_key?( "0123456789" )

    assert_equal ["", "0123456789",  "abc"], d1.keys.sort  # Les diverses

    assert_equal  0, d1[""]    # Obtention des definitions
    assert_equal  3, d1["abc"]
    assert_equal 10, d1["0123456789"]

    d1.delete( "" )            # Suppression d'une cle existante
    assert !d1.has_key?( "" )
    assert_equal 2, d1.size()

    d1["abc"] = 923            # Redefinition d'une cle
    assert d1.has_key?( "abc" )
    assert_equal 2, d1.size()
    assert_equal 923, d1["abc"]
  end
end
```

6.4.2 Exemple Java d'utilisation d'un dictionnaire (`Map`)

```
import java.util.*;

public class TesterMap {
    @Test public void testDictionnaire() {
      Map<String,Integer> d1  = new HashMap<String,Integer>();

      d1.put( "", 0 );
      d1.put( "abc", 3 );
      d1.put( "0123456789", 10 );

      assertEquals( 3, d1.size() );
      assertTrue( d1.containsKey("") );
      assertTrue( d1.containsKey("abc") );
      assertTrue( d1.containsKey("0123456789") );

      assertEquals( 3, d1.keySet().size() );
      assertTrue( d1.keySet().contains("") );
      ...

      assertEquals(  (Integer) 0, d1.get("") );
      assertEquals(  (Integer) 3, d1.get("abc") );
      assertEquals( (Integer) 10, d1.get("0123456789") );

      d1.remove( "" );
      assertFalse( d1.containsKey( "" ) );
      assertEquals( 2, d1.size() );

      d1.put( "abc", 923 );
      assertTrue( d1.containsKey( "abc" ) );
      assertEquals( 2, d1.size() );
      assertEquals( (Integer) 923, d1.get("abc") );
    }
}
```


6.4.3 Spécification pseudo-OCL d'un dictionnaire (`Dictionnaire`)

Le type abstrait `Dictionnaire`, présenté à la page suivante, spécifie une classe d'objets pour des dictionnaires dans le style DBC.

Exemples d'utilisation — en supposant une mise en œuvre Java avec une classe `Dictionnaire` :

```
@Test void exempleDictionnaire () {
  Dictionnaire<String,Integer> d1 = new Dictionnaire<String,Integer>();

  d1.definir( "", 0 );
  d1.definir( "abc", 3 );
  d1.definir( "0123456789", 10 );

  assertEquals( 3, d1.nbCles() );

  assertTrue( d1.estDefinie( "" ) );
  assertTrue( d1.estDefinie( "abc" ) );
  assertTrue( d1.estDefinie( "0123456789" ) );

  assertEquals(  0, d1.definitionPour( "" ) );
  assertEquals(  3, d1.definitionPour( "abc" ) );
  assertEquals( 10, d1.definitionPour( "0123456789" ) );

  d1.supprimer( "" );
  assertFalse( d1.estDefinie( "" ) );
  assertEquals( 2, d1.nbCles() );

  d1.definir( "abc", 923 );
  assertEquals( 2, d1.nbCles() );
  assertEquals( 923, d1.definitionPour( "abc" ) );
}
```

```
// Specification pseudo-OCL d'un Dictionnaire
// utilisant l'appproche DBC.

class Dictionnaire

operations

  // Requetes primitives.

  nbCles(): Integer
    // post: result = Nombre de cles definies.

  estDefinie( cle: C ): Boolean
    // post: result = cle est-elle definie?

  definitionPour( cle: C ): D
    pre CleEstDefinie:
       estDefinie(cle)
   // post: result = la definition associee a cle.


  // Requete derivee.
  cles(): Set(C)
    post TailleEgaleNombreDeCles:
      result->size() = nbCles()
    post ContientToutesLesClesDefinies:
      result->forAll( c | estDefinie(c) )
```



```
  // Commandes

  definir( cle: C, defn: D )
    post UneCleDePlusSiPasDejaDefinie:
      nbCles() = if estDefinie(cle)@pre
                   then nbCles()@pre
                   else nbCles()@pre + 1
                 endif
    post EstAssurementDefinie:
      estDefinie(cle)
    post BonneDefinitionPourCle:
      definitionPour(cle) = defn

  supprimer( cle: C )
    pre CleEstDefinie:
      estDefinie(cle)
    post UneCleDeMoins:
      nbCles() = nbCles()@pre - 1
    post ClePasDefinie:
      not estDefinie(cle)
end
```