# OCL and its use

Noël Plouzeau



## Precise specifications are a must

- Without precise specifications you cannot
  - write a correct implementation
  - test your implementation
  - match the product owner’s needs

# What is OCL

- A mathematical notation (predicate logic) with a plain programming language syntax
- Provides an unambiguous definition for predicates, preconditions, postconditions, invariants

# Origin of the OCL notation

- Came from the research work of Anneke Kleppe and Jos Warner at IBM Research
- It was included in the UML standard to enhance precision of the UML
  - european school/american school

# Principles of the OCL

- No greek characters
- Looks like a functional language
- No side effects (no assignments...)
- Based on the map/filter/reduce paradigm

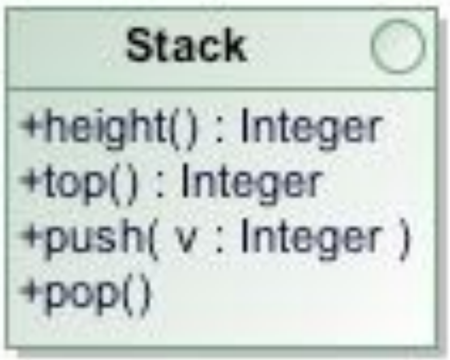
# Where is it useful?

- Preconditions of operations
- Postconditions of operations
- Type invariants
- Guards of transitions, message sending, etc
- Assertions

# When is it useful?

- At all stages of a design process
  - Problem domain analysis
  - Requirements
  - Design
  - Test
  - Implementation
- Again, this is **not** a programming language

# A first example



# OCL specification

- context Stack::pop()
  - pre stack_not_empty: height() > 0
  - post : -- A bit more complex, see later

# Predefined types

- Mathematical types
  - Integer
  - Real -- A real Real, not a floating point
  - Boolean

# Predefined types (cont'd)

- Collection(T)
  - The top type of composite structures
  - No notion of uniqueness or order
  - Many basic operations are defined for Collection

# Subtypes of Collection

- Set(T)
  - can store an object at most once, no order relation
- Bag(T)
  - can store an object several times, no order relation

# Subtypes of Collection (cont’d)

- OrderedSet(T)
  - can contain an object at most once, order relation
- Sequence(T)
  - can contain an object several times, order relation

# Test operations in Collection

- isEmpty(), notEmpty()
  - test wrt the empty collection
- size()
  - cardinality

# Test operations in Collection (cont'd)

- includes(e)
  - classical belongs to mathematical operation
- includesAll©
  - classical inclusion mathematical operation

# Construction operations in Collection (cont'd)

- including(e)
  - classical union operation with a singleton
- includingAll(c)
  - classical union operation with another collection

# Filter, map and reduce

- Classical concepts to work on collections
- Basic operations in functional programming (e.g. ML)
- They are applied on a input collection
- They produce an output collection
- The input is not changed (no side effects)

# Concept of filter

- Takes a collection and a predicate function as input
- Returns the collection of elements for which the function evaluates to true

# Filter operations in OCL

- select(x:T| expression):Collection(T)
  - *evaluation of* Set(Integer){1,2,4,6}->select(x|x < 3)
  - *gives* Set(Integer){1,2}
- reject(x:T| expression):Collection(T)
  - *is equivalent to* select(x:T| not expression):Collection(T)

# Concept of map

- Take a collection c  and a function f as inputs
- Compute f(e) for each e in c
- Return a collection of these results

# Map operations in OCL

- collect(x:T | expression_type_2):Collection(type_2)
- Example
  - *evaluation of* Sequence(Integer){4,2} ->collect(x:Integer| x*x-2)
  - *gives* Sequence(Integer){14,2}

# Concept of Reduce

- Filter and map work on each collection element separately
  - One needs a different concept to combine elements from the collection
- Reduce takes a collection c and a binary function as inputs
- The function is evaluated for each element, with each result being reused as the first parameter for the next evaluation

# Predefined reduce in OCL

- forAll(x:T | expression_bool):Boolean
  - reduces with f(x,y) = x and y
- exists(x:T | expression_bool):Boolean
  - reduces with f(x,y) = x or y

# General reduce in OCL

- iterate(x:T; acc : T2 = v0 | expr):T2
  - the expr is of type T2, and contains references to x and acc
  - expr is evaluated for each x, with acc bound to the previous evaluation result
  - the whole value is given by acc

# Example of reduce

- let p = Set(String){"Welcome","Neo"} in
- p->iterate(s:String ; acc:Integer=0 |
- acc + s->size())
- *gives* 10

# Special forms of reduce

- isUnique(x:T | expr_type_T2):Boolean
  - returns true if and only if all computed expressions differ
- any(x:T | expr):T
  - returns one item for which expr evaluates to true (non deterministic)

# Special forms of reduce (cont'd)

- one(x:T | expr):Boolean
- returns true if and only if un seul élément donne true pour l'évaluation de l'expression

# A few examples

- On the blackboard