Model-Driven Software Engineering (MDSE)

Bahman Zamani, Ph.D.
bahmanzamani.com

Department of Computer Engineering
Faculty of Engineering
University of Isfahan

Slides are prepared by Dr. Abdelwahab Hamou-Lhadj and are used by permission

Why Do We Need OCL?

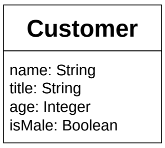
- We need precision (formality) to achieve MDA-like modelling
- The mechanisms seen so far for xUML are good, but they are still not sufficient
- *OCL is one solution (among many), standardized, and integrated to UML*

More About OCL?

- OCL is a textual language of typed expressions based on mathematical sets and logic
- Various other formal languages have also been used in software engineering
  - Most such languages have proven difficult for the average developer to use
  - The mathematical symbols are not well known and use obscure fonts
- OCL was developed by IBM (1995); (Formally defined 1997.)
  - It emphasize precision and simplicity. There is no use of special mathematical symbols

What is a Constraint?

- A constraint specifies a restriction on one or more values of (part of) an object-oriented model or system
- A constraint is a valid OCL expression of type Boolean
- Simple constraints on attributes of a class:



Customer

name: String
title: String
age: Integer
isMale: Boolean

- **`age >= 18 and age < 66`**
- **`title = if isMale then 'Mr.' else 'Ms.' endif`**
- **`name.size < 100`**

OCL: Declarative and Typed

- **Declarative**
  - States what should be done, not how
    — Implementation independent
  - Expressions have no side effects
    — Evaluation does not change the system state.
- **Strongly typed**
  - Each OCL expression has a type and evaluates to a value or to an object within the system:
    — A constraint is a valid OCL expression of type *Boolean*

Places to use OCL in UML models

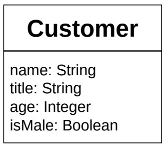
- **Invariants**
  - Constraint on a **class** or type that must **always** hold
  - Also applicable to types and stereotypes
- **Precondition**
  - Constraint that must hold **before** the execution of an **operation**
- **Postcondition**
  - Constraint that must hold **after** the execution of an **operation**
- **Guard**
  - Constraint on the **transition** from one state to another
  - OCL is not limited to class diagrams!

Types in OCL

- **Predefined types:**
  - Basic types: Integer, Real, String and Boolean
  - Collection types: Set and Sequence (there are others but they are outside the scope of this course)
- **User-defined model types:**
  - Enumeration and all classes

Context of an OCL expression

- Every OCL expression is bound to a specific **context**, where **self** can be used as a reference to this context.



“A customer must be between 18 and 66 years old”

context Customer

inv: self.age >= 18 and self.age < 66

Standard Operations for Real and Integer types

Operation	**Notation**	**Result type**
equals	a = b	Boolean
not equals	a <> b	Boolean
less	a < b	Boolean
more	a > b	Boolean
less or equal	a <= b	Boolean
more or equal	a >= b	Boolean
plus	a + b	Integer or Real
minus	a - b	Integer or Real
multiply	a * b	Integer or Real
divide	a / b	Real
modulus	a.mod(b)	Integer
integer division	a.div(b)	Integer
absolute value	a.abs()	Integer or Real
maximum	a.max(b)	Integer or Real
minimum	a.min(b)	Integer or Real
round	a.round()	Integer
floor	a.floor()	Integer

Standard Operations for the Boolean Type

Operation	Notation	Result type
or	a or b	Boolean
and	a and b	Boolean
exclusive or	a xor b	Boolean
negation	not a	Boolean
equals	a = b	Boolean
not equals	a <> b	Boolean
implication	a implies b	Boolean
if then else	if a then b1 else b2 endif	type of b

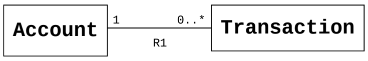
Standard Operations for the String Type

Operation	Expression	Result type
concatenation	s.concat(string)	String
size	s.size()	Integer
to lower case	s.toLower()	String
to upper case	s.toUpper()	String
substring	s.substring(int, int)	String
equals	s1 = s2	Boolean
not equals	s1 <> s2	Boolean

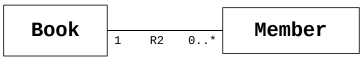
Collection Types and Navigation in OCL Expressions

- If self is class **C**, with attribute **a** then
  - **self.a** evaluates to the **object** stored in **a**.
- If **C** has a **1..*** association called **R1** to another class **D**
  - **self.R1** returns to a **Set** whose elements are of type **D**
  - if **R1** is {ordered} then a **Sequence** is returned
- If **D** has attribute **b** then
  - **self.R1.b** evaluates to the set (or sequence if {ordered is used}) of all the b’s belonging to **D**

Navigating Associations

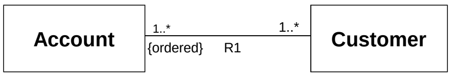


`self.R1` returns a **set** of transactions if we are in the context Account



`self.R2` returns a **set** of members if we are in the context Book

Navigating to Ordered Collections



The key word {ordered} is a predefined constraint in UML, which means the collection is ordered

self.R1 returns a sequence of accounts in the context Customer

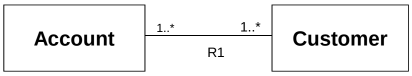
OCL Functions on Collection Types

- With collection types, an OCL expression
  - states a fact about all objects in the collection, or
  - states a fact about the collection itself, e.g. the size of the collection.
  - Syntax: **`collection->function`**

Standard Operations on all Collection Types

Operation	Description
size()	The number of elements in the collection
count(object)	The number of occurences of object in the collection.
includes(object)	True if the object is an element of the collection.
includesAll(collection)	True if all elements of the parameter collection are present in the current collection.
excludes(object)	True if the object is *not* an element of the collection.
excludesAll(collection)	True if all elements of the parameter collection are *not* present in the current collection.
isEmpty()	True if the collection contains no elements.
notEmpty()	True if the collection contains one or more elements.

Example using size() and notEmpty()



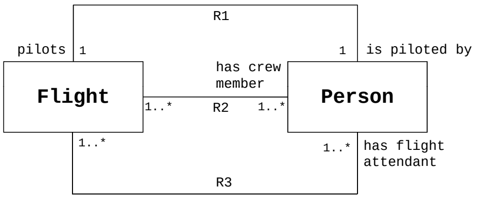
Each account can have at most 2 customers

```
context Account
inv: self.R1->size() <= 2
```

An account must be assigned to at least one customer

```
context Account
inv: self.R1->notEmpty()
```

Example Using includes and includesAll



Example Using includes and includesAll (cont.)

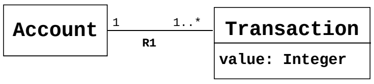
```
A pilot is a member of the crew
      context Flight
      inv: self.R2->includes(self.R1)


A11 flight attendants are also crew members
      context Flight
      inv: self.R2->includesAll(self.R3)
```

Select, Reject, Collect, and Sum

- collection->select(condition): creates a subcollection that contains objects that satisfy the condition
- collection->reject(condition): creates a subcollection that contains objects that do not satisfy the condition
- collection->collect(object.attribute): creates a collection of equal size with objects containing the attribute specified
- collection->sum(): returns the sum of the elements of the collection; these elements must be numerical

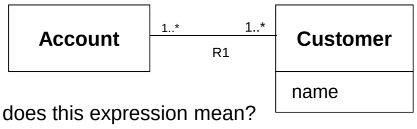
Examples



- Assume we are in the context Account, what is returned by each of the following expressions?
  - self.R1 -> select(value > 500)
  - self.R1 -> reject(value > 500)
  - self.R1 -> sum(self.R1->collect (value))
  - self.R1.value -> sum()

forAll Expression

- Evaluating some expression on every element of a collection
- Syntax: collection->forAll (boolean-expression)



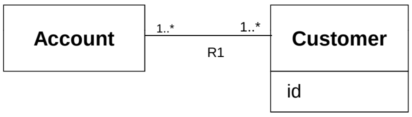
- What does this expression mean?

  context Account

  inv: self.R1->forAll (name = 'Jack')

forAll with two variables

- Considers each pair in the *Cartesian product* of employees



context Account

inv: self.R1->forAll(e1, e2 : Customer |
e1 <> e2 implies e1.id <> e2.id)

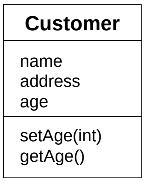
Adding Preconditions and Postconditions to Operations

- **Preconditions:**
  - are predicates associated with a specific operation must be true before the operation is invoked.
- **Postcondition:**
  - are predicates associated with a specific operation
  - must be true after an operation is invoked

Example of Using pre and post Conditions

- Precondition and postcondition:

```
context Customer::setAge(a: integer)
  pre: a > 1
  post: age = age@pre + 1
```



- In OCL, we use the **`@pre`** suffix to indicate that we are referring to a value at the start of an operation

The return Keyword

- The keyword return can be used in a post condition to indicate the return value of an operation:

```
context Customer::getAge(): integer
post: return = self.age
```

oclIsTypeOf and oclIsKindOf

- oclIsTypeOf(t : OclType) : Boolean
  - true if the type of self and t are the same.
  - E.g.

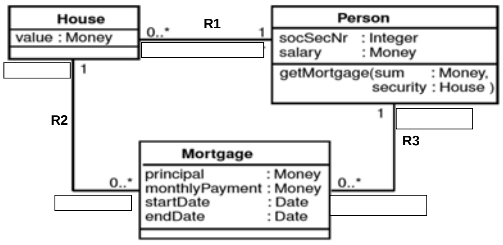
    context Account

    inv: self.oclIsTypeOf(Account) -- is true

    inv: self.oclIsTypeOf(Customer) -- is false

- oclIsKindOf(t : OclType) : Boolean
  - true if t is either the direct type or one of the supertypes of an object

Case Study



Write OCL Constraints:

- The start date for any mortgage must be before the end date.
- A person may have a mortgage on a house only if that house is owned by himself or herself (one cannot get a mortgage on the house of one’s neighbor or friend!)
- A person may have a mortgage on a house only if that house is owned by himself or herself (one cannot get a mortgage on the house of one’s neighbor or friend!). Second example, with Person as context.
- The social security number of all persons must be unique. Use house a context. You can also use Person but you will need to use self.allInstace() that returns all instances of Person.
- A new mortgage will be allowed only when the person’s income is sufficient. (assume that the yearly payment must be less than 30% of the salary)
- A new mortgage will be allowed only when the counter value of the house is sufficient

References

- “The Object Constraint Language (2nd edition), Getting Your Models Ready for MDA”

  by Jos Warmer and Anneke Kleppe,

  Addison Wesley, 2003

- UML 2.0 OCL : http://www.omg.org/docs/ptc/03-10-14.pdf
- OCL Center: http://www.klasse.nl/ocl/index.html
- USE Demo:

  http://cserg0.site.uottawa.ca/seg/bin/view/CSI5112/UsingUseForOCL