



HY 351: Ανάλυση και Σχεδίαση Πληροφοριακών Συστημάτων CS 351: Information Systems Analysis and Design

HY351:
Ανάλυση και Σχεδίαση Πληροφοριακών Συστημάτων
Information Systems Analysis and Design



OCL: Object Constraint Language



Γιάννης Τζιτζίκας

Διάλεξη : 14
Ημερομηνία : 11-12-2006



Διάρθρωση

- Στόχοι της OCL
- Γιατί να χρησιμοποιήσουμε την OCL
- Παρουσίαση της OCL
- Διαβεβαιώσεις και Γλώσσες Προγραμματισμού
 - (Assertions and Programming Languages)



Τι είναι η OCL (Object Constraint Language)?

- Μια τυπική γλώσσα (formal language) για την προδιαγραφή περιορισμών (constraints) σε αντικειμενοστρεφή μοντέλα
- Είναι δηλωτική (declarative) (περιγράφει το *τι* αντί του *πως*)
- Είναι μια γλώσσα με τύπους (typed)
 - Και πιο φιλική από άλλες τυπικές γλώσσες

Περιορισμοί

- Κάποιοι περιορισμοί μπορούν να εκφραστούν γραφικά (π.χ. η πολλαπλότητα των συσχετίσεων, partition subclasses, κλπ).
- Για κάποιους άλλους αυτό δεν είναι εύκολο, π.χ.:
 - Περιορισμοί που εμπλέκουν >2 κλάσεις
 - Περιορισμοί που εμπλέκουν τιμές γνωρισμάτων (και συνδυασμούς αυτών)
 - Προϋποθέσεις και Μετα-συνθήκες (pre/post-conditions) λειτουργιών

Η OCL μπορεί να τους εκφράσει με τυπικό τρόπο



Γιατί να γράφουμε περιορισμούς σε OCL; Why to write OCL constraints?

Γιατί να εκφράσουμε ρητώς τέτοιους περιορισμούς;

Γιατί κάνουν τα μοντέλα μας πιο ακριβή

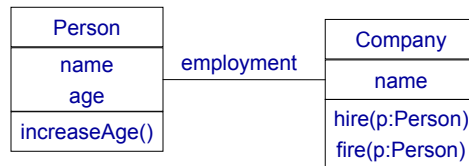
- Όστε να τα καταλαβαίνουμε καλύτερα
- Όστε οι προγραμματιστές να τα υλοποιούν (σωστά)
- Όστε να μπορούμε να έχουμε μια τυπική επαλήθευση (formal validation) του μοντέλου πριν την υλοποίηση

Μπορούν να μεταφραστούν σε «διαβεβαιώσεις» (assertions) στις γλώσσες προγραμματισμού

- Μερικά εργαλεία CASE προσφέρουν τέτοιες μεταφραστικές και επαληθευτικές υπηρεσίες



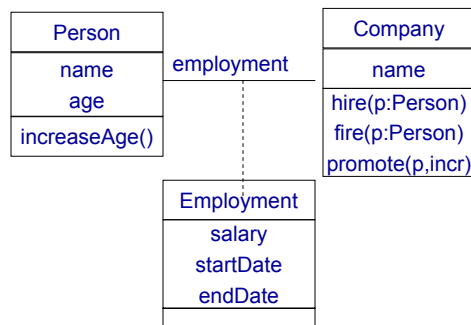
Τα διαγράμματα κλάσεων δεν είναι πολύ ακριβή



- Μπορεί ένας ανήλικος να εργαστεί σε μια εταιρία;
- Μπορεί μια εταιρία να προσλάβει ένα άτομο που είναι ήδη εργαζόμενος της;



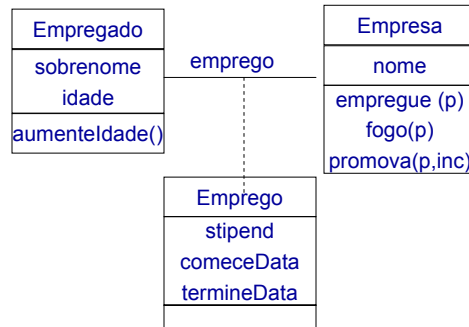
Τα διαγράμματα κλάσεων δεν είναι πολύ ακριβή (II)



- Μπορεί ένα πρόσωπο να αρχίσει να εργάζεται πριν τη γέννησή του;
- Μπορεί μια προαγωγή να μειώσει το μισθό ενός εργαζομένου;
- Υπάρχει κάποιο κατώτερο όριο στους μισθούς για αυτούς που εργάζονται στην εταιρία πάνω από 10 έτη;



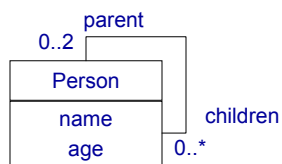
Τα διαγράμματα κλάσεων δεν είναι πολύ ακριβή (III) ... Η σπουδαιότητα της γνώσης υποβάθρου



- Θα μπορούσατε να αναπτύξετε ένα σύστημα αν τα διαγράμματα κλάσεων ήταν σε μια γλώσσα που δεν γνωρίζατε (π.χ. στα Ισπανικά);



Τα διαγράμματα κλάσεων δεν είναι πολύ ακριβή (IV)



Επιτρεπτό διάγραμμα
αντικειμένων
(object diagram)



Object Constraint Language (OCL)

- OCL is a formal language used to describe expressions on UML models.
- OCL expressions typically specify invariant conditions that must hold for the system being modeled.
- They also specify queries over objects described in a model.
- OCL is a typed language, so that each OCL expression has a type. To be well formed, an OCL expression must conform to the type conformance rules of the language. For example, you cannot compare an Integer with a String.
- When OCL expressions are evaluated, they do not have side effects; i.e. their evaluation cannot alter the state of the corresponding executing system.
 - However, OCL expressions could be used to specify operations / actions that, when executed, do alter the state of the system.



Πότε τη χρησιμοποιούμε; Where to use OCL?

OCL can be used for a number of different purposes:

- To specify invariants on classes and types in the class model
- To describe pre- and post conditions on Operations and Methods
- To specify derivation rules for attributes for any expression over a UML model.
- To describe Guards in State Diagrams
- To specify target (sets) for messages and actions
- To specify type invariant for Stereotypes
- As a query language

UML modelers can use OCL to

- to specify application-specific constraints in their models.
- to specify queries on the UML model, which are completely programming language independent



Οι κυριότεροι τύποι περιορισμών της OCL The main types of OCL Constraints

- **Invariants on classes** (αναλλοίωτες συνθήκες στις κλάσεις)
 - συνθήκες που πρέπει να ικανοποιούνται από όλα τα στιγμιότυπα μιας κλάσης
 - Π.χ.. salary > 1000 Euro
- **pre-conditions on operations** (προ-συνθήκες στις λειτουργίες)
 - συνθήκες που πρέπει να ικανοποιούνται πριν την εκτέλεση μιας λειτουργίας
 - Π.χ. η λειτουργία «Απόλυση()» μπορεί να εκτελεστεί μόνο σε έναν άτομο που έχει ήδη προσληφθεί
- **post-conditions on operations** (μετα-συνθήκες στις λειτουργίες)
 - συνθήκες που πρέπει να ικανοποιούνται μετά την εκτέλεση μιας λειτουργίας
 - Π.χ. μετά την εκτέλεση της «Ανάληψη(ποσό)» το υπόλοιπο του τραπεζικού λογαριασμού πρέπει να έχει μειωθεί κατά «ποσό».



Πως μπορούμε να προδιαγράψουμε έναν περιορισμό; How we can specify a constraint?

- Declaration of the context of a constraint by referencing the model element that a constraint applies to
- Declaration of the type of a constraint (*inv*, *pre*, *post*)
- Expressing the desired condition by referencing properties of model elements and using various operations that are supported.



Δήλωση Συφραζομένων (Συνκειμένων) Context Declaration

- Προσδιορίζει το στοιχείο στο οποίο αφορά ο περιορισμός
- Το context μπορεί να είναι
 - a class (for invariants)
 - an operation (for pre/post-conditions)

- Παράδειγμα:

Employee
name
age
salary
SetAge(a)
SetSalary(s)

Context Employee **inv:** self.salary > 1000

Context Employee::SetSalary(salary) **pre:** salary > 1000

Δεν είναι ισοδύναμα. Γιατί;



Ονόματα και σχόλια περιορισμών Constraint names and comments

Employee
name
age
salary
SetAge(a)
SetSalary(s)

Context Employee::SetAge (age)
pre: age > 0

Context Employee::SetAge (age)
pre positive_age : age > 0


Optional constraint name

Allowing the constraint to be referenced by name.

Context Employee::SetAge (age)
pre positive_age : age > 0
-- the age should always be positive

Comment

Everything immediately following the two dashes up to and including the end of line is part of the comment.

 **self**

Employee
name
age
salary
SetAge(a)
SetSalary(s)

Context Employee
inv: self.salary > 1000


Context Employee
inv: salary > 1000

Context e: Employee
inv: e.salary > 1000

In most cases, the keyword self can be dropped because the context is clear. As an alternative for self, a different name can be defined playing the part of self.

equivalent

U. of Crete, Information Systems Analysis and Design Yannis Tzitzikas, Fall 2005 15

 **Επιλογείς (πώς να αναφέρουμε στοιχεία)**
Selectors (how we reference elements)

Person	0..*	employment	0..1	Company
name	employee		employer	name
age		president		stockPrice()
	1		0..1	

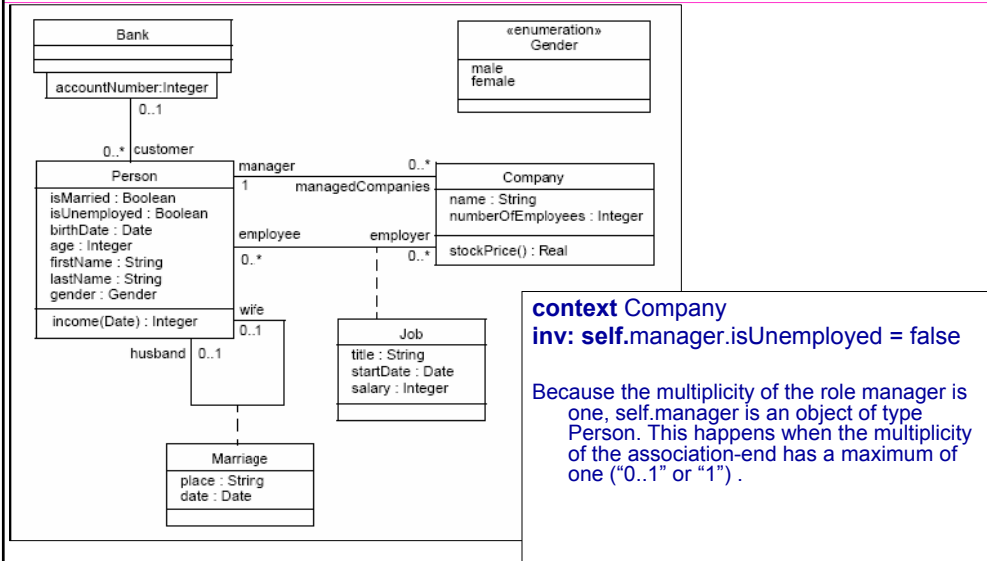
self.age // returns the age of a particular person
self.employment // returns the employer (company) of a person
self.employer // as before

self.employment // returns the set of all employees of a company
self.president // returns the singleton with the president of a company
self.stockPrice() // returns the value this method would return

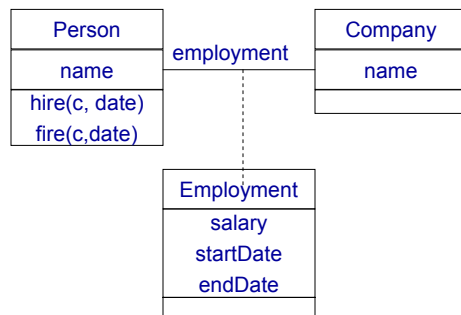
U. of Crete, Information Systems Analysis and Design Yannis Tzitzikas, Fall 2005 16



Selectors (how we reference elements)



Selectors Referencing Association Classes



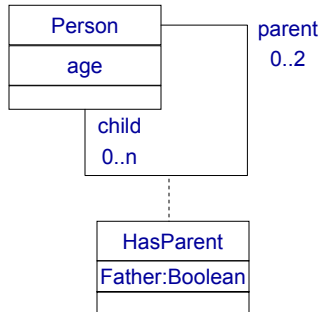
The salary should be > 1000

Context Person inv: self.employment.salary > 1000

We use dot and the name of the association class starting with a lowercase letter



Selectors Referencing Recursive Associations



The age of a children should be less than the age of its parents.

Here the name of the association class alone is not enough.

We need to distinguish the direction in which the association is navigated.

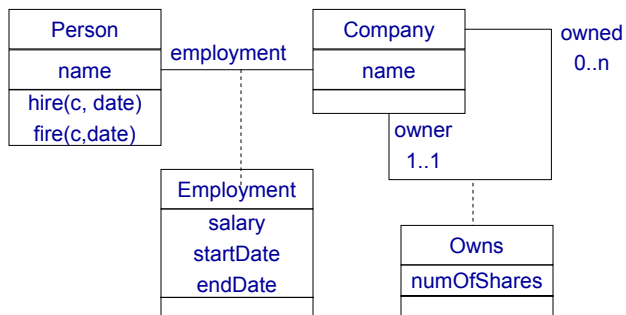
To make the distinction, the **rolename** of the direction in which we want to navigate is added to the association class name, enclosed in square brackets.

Context Person **inv:** self.hasParent[parent].age > self.age

self.hasParent.age is invalid



Selectors Referencing Recursive Associations



Let c be a company. The name of the company that owns c should be different than the name of c.

Context Company **inv:** self.owns[owner].name <> self.name



Πράξεις Operations

- Boolean Operations

- **and** // \wedge
- **or** // \vee
- **not** // \neg
- **implies** // \rightarrow
- **xor**

- Comparison operations

- $<$, $>$, $<=$, $>=$, $<>$, $==$

- Arithmetic

- $+$, $-$, $*$, $/$, **abs()**, **div**, **floor()**, **round()**

- String operations

- **concat**(s1, s2), **toUpper**(s),

- Nil

- if an attribute **attr** of an object **obj** has no value then **obj.attr** returns **nil**

- Empty

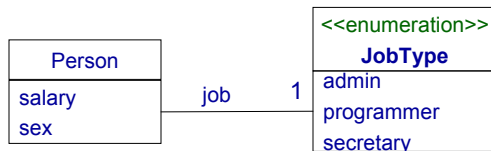
- if there are no associated objects to an object **obj** through an association **assoc** then **obj.assoc** returns the empty bag $\{\}$.

- Nil $<>$ Empty



Αναφορά σε απαριθμητούς τύπους Referring to enumerations

- Enumerations



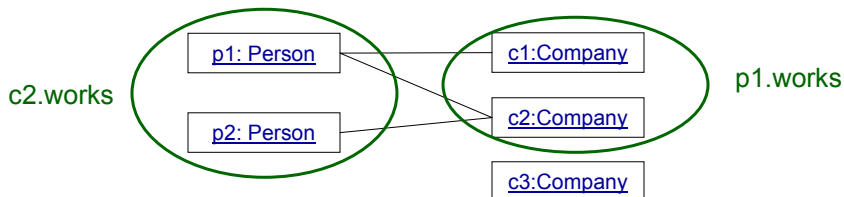
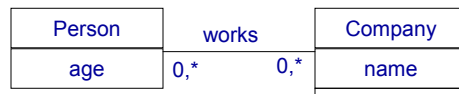
Context Person **inv:** self.job=JobType::admin **implies** self.salary > 10.000

Context Person **inv:** self.name="Yannis" **implies** self.sex::Male



Συλλογές στην OCL Collections in OCL

- Allow us to refer to the objects that are referred using associations (typically in those with upper multiplicity > 1)

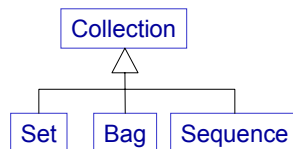


Collections in OCL (II)

Single navigation of an association results in a **Set**,
combined navigations in a **Bag**, and
navigation over associations adorned with {ordered} results in an **OrderedSet**.



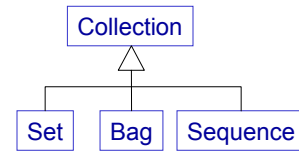
Collection is an abstract type, with the concrete collection types (Set, Sequence, and Bag) as its subtypes.





Αντικείμενα και Συλλογές Objects and Collections

- **Objects**
 - are instances of classes, including the predefined ones (e.g. Integer)
- **Sets**
 - a “set” of objects
 - example: Set { p1, p2}
- **Bag**
 - duplicates allowed
 - example: Bag { p1, p1, p1, p2, p1}
- **Sequence**
 - is a bag of ordered elements
 - example: Sequence {p1, p2, p3, p1 } // <p1, p2, p3, p1>



Πράξεις σε Συλλογές Collection Operations

The type Collection defines a large number of predefined operations to enable the modeler to manipulate collections.

As OCL is an expression language, collection operations never change collections (rather than changing the original collection they project the result into a new one).

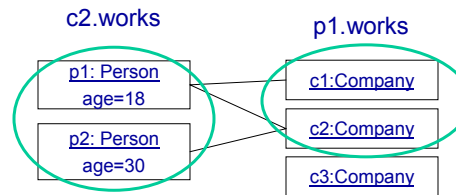
- `c1->Size()` // number of elements of c1
- `c1->count(elem)` // counts the number of occurrences of *elem* in c1
- `c1->includes(elem)` // checks if *elem* is member of c1
- `c1->includesAll(coll)` // checks if *coll* is contained in c1
- `c1->excludes(elem)` // returns True if *elem* is not member of in c1
- `c1->isEmpty()` // checks if c1={}

- `c1->forAll(expr)` // returns True if *expr* is true for all elements of c1
- `c1->exists(expr)` // returns True if *expr* is true for at least one element of c1
- `c1->select (expr)` // returns the elements of c1 that satisfy *expr*
- `c1->reject (expr)` // returns the elements of c1 that do not satisfy *expr*
- **SET OPERATIONS:**
 - `c1->union(c2)`, `c1->intersection(c2)`, `c1-c2`



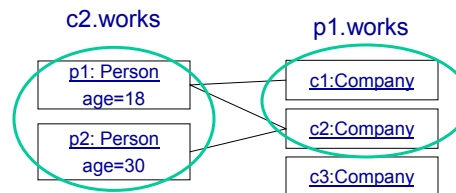
Παραδείγματα Collection Operations: Examples

- `p1.works->size()` is 2
- `p1.works->count(c3)` is 0
- `p1.works->includes(c2)`
 - is True
- `p1.works->includes(c3)` is False
- `c2.works->includesAll(c1.works)`
 - is True
- `c1.works->includesAll(c2.works)`
 - is False
- `c3.works->isEmpty()`
 - is True



Collection Operations: Examples (II)

- `c2.works->forall{ x | x.age>20 and x.age < 70}`
 - is False
- `c2.works->exists{ x | x.age>20 and x.age < 70}`
 - is True
- `c2.works->select{ x | x.age>20 and x.age < 70}`
 - will return {p2}
- `p1.works->intersection(p2.works)`
 - will return {c2}
- `p1.works - p2.works`
 - will return {c1}

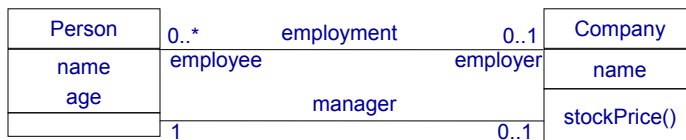




Ένα αντικείμενο μπορεί να αντιμετωπιστεί ως μονοσύνολο
 A single object can be treated as a singleton

A single object can be used as a Set as well. It then behaves as if it is a Set containing the single object. The usage as a set is done through the arrow followed by a property of Set.

context Company
inv: self.manager->size() = 1



Select / Reject

The reject operation is available in OCL for convenience, because each reject can be restated as a select with the negated expression. Therefore, the following two expressions are identical:

collection->reject(v : Type | boolean-expression-with-v)
collection->select(v : Type | not (boolean-expression-with-v))

The collection of all the employees who are not married is empty:

context Company
inv: self.employee->reject(isMarried)->isEmpty()



Collect operation

The select and reject operations always result in a sub-collection of the original collection.

When we want to specify a collection which is derived from some other collection, but which contains different objects from the original collection (i.e., it is not a sub-collection), we can use a collect operation.

The collect operation uses the same syntax as the select and reject and is written as one of:

collection->collect(v : Type | expression-with-v)

collection->collect(v | expression-with-v)

collection->collect(expression)

The value of the reject operation is the collection of the results of all the evaluations of expression-with-v.

An example: specify the collection of birthDates for all employees in the context of a company. This can be written in the context of a Company object as one of:

self.employee->collect(birthDate)

self.employee->collect(person | person.birthDate)

self.employee->collect(person : Person | person.birthDate)



Collect (2)

Shorthand for Collect

Because navigation through many objects is very common, there is a shorthand notation for the collect that makes the OCL expressions more readable.

Instead of

self.employee->collect(birthdate)

we can also write:

self.employee.birthdate

In general, when we apply a property to a collection of Objects, then it will automatically be interpreted as a collect over the members of the collection with the specified property. For any property name that is defined as a property on the objects in a collection, the following two expressions are identical:

collection.propertyname

collection->collect(propertyname)

and so are these if the property is parameterized:

collection.propertyname (par1, par2, ...)

collection->collect (propertyname(par1, par2, ...))



Collect (3)

When the source collection is a **Set** the resulting collection is not a Set but a **Bag**.

If the source collection is a **Sequence** or an **OrderedSet**, the resulting collection is a **Sequence**.

When more than one employee has the same value for birthDate, this value will be an element of the resulting Bag more than once.

The Bag resulting from the collect operation always has the same size as the original collection.

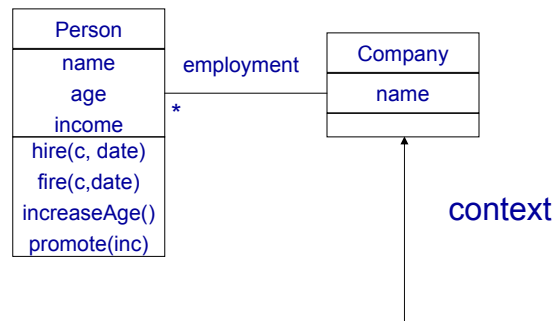
It is possible to make a Set from the Bag, by using the **asSet** property on the Bag.
Example:

```
self.employee->collect( birthDate )->asSet()
```

Results in the Set of different birthDates from all employees of a Company



Examples with Bags and other operations



- **employment.age** is a bag
- **employment.income** is a bag
- **employment.income->asSet()** returns all distinct incomes of the employees



Παραδείγματα με Αναλλοίωτες Συνθήκες Examples of Invariants (using collection operations)



All persons should have positive age

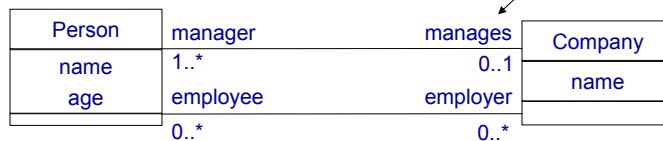
Context Person **inv:** self.age > 0

All persons that work for a company should be adults

Context Company **inv:** self.employment->forall(x | x.age > 18)



Examples of Invariants (using collection operations)



A person can be a manager of only one company

All companies should have managers that are not employers of other companies

Context Company **inv:** not (self.manager->exists(x) | x.employer->exists(y|y<>self))

Context Company **inv:** self.manager.employer->forall(x | x = self)



Another example



Context Person

inv: self.parent->excludes(self) and self.children->excludes(self)



Forall

context Company

inv: self.employee->forAll(age <= 65)

inv: self.employee->forAll(p | p.age <= 65)

inv: self.employee->forAll(p : Person | p.age <= 65)

These invariants evaluate to true if the age property of each employee is less or equal to 65.

The forAll operation has an extended variant in which more than one iterator is used.

Both iterators will iterate over the complete collection.

Effectively this is a forAll on the Cartesian product of the collection with itself.

context Company **inv:**

self.employee->forAll(e1, e2 : Person | e1 <> e2 implies e1.firstname <> e2.firstname)

This expression evaluates to true if the forenames of all employees are different.

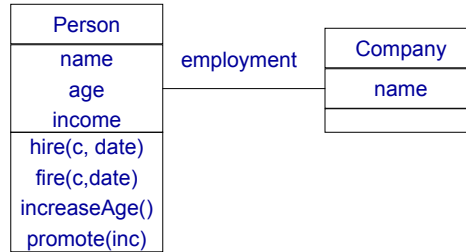
It is semantically equivalent to:

context Company **inv:**

self.employee->forAll(e1 | self.employee->forAll(e2 | e1 <> e2 implies e1.firstname <> e2.firstname))



Examples of Constraints (using collection operations) Pre/Post-Conditions



Context Person::hire(c:Company)
pre: not employment->includes(c)
post: employment->includes(c)

Context Person::fire(c:Company)
pre: employment->includes(c)
post: not employment->includes(c)

Context Person::increaseAge ()
post: age = age@pre + 1

@pre: the value of an attribute/association
 before the execution of the operation

Context Person::Promote (inc) **post:** self.income = income@pre * (1+inc)



@Pre

When the pre-value of a property evaluates to an object, all further properties that are accessed of this object are the new values (upon completion of the operation) of this object.

a.b@pre.c -- takes the old value of property b of a, say object18,
 -- and then the new value of c of object18.

a.b@pre.c@pre -- takes the old value of property b of a, say object18
 -- and then the old value of c of object18.

The '@pre' postfix is allowed only in OCL expressions that are part of a Postcondition.

Asking for a current property of an object that has been destroyed during execution of the operation results in OclUndefined. Also, referring to the previous value of an object that has been created during execution of the operation results in OclUndefined.



Post-conditions Result, out-parameters

The reserved word **result** denotes the result of the operation, if there is one.

```
Context Person::getIncome(d:Date): Integer  
post: result = 1000
```

The right-hand-side of this definition may refer to the operation being defined (i.e., the definition may be recursive) as long as the recursion is not infinite.

When the operation has no out or in/out parameters (like in this example), then the type of result is the return type of the operation (here Integer).

When the operation has out or in/out parameters, the return type is a **Tuple**.

The postcondition for the income operation with an out parameter *bonus* could be:

```
Context Person::getIncome(d:Date, bonus:Integer): Integer  
post: result = Tuple{bonus=300, result=1000}
```

The return type of operation calls is Tuple(bonus: Integer, result: Integer).



Post-conditions Result, out-parameters (2)

```
Context Person::getIncome(d:Date, bonus:Integer): Integer  
post: result = Tuple{bonus=300, result=1000}
```

The out parameters need not be included in the operation call (we have to provide values only for the in or in/out parameters).

Let *Yannis* be an object of the class *Person*, and let *d1* be a *Date*.
Then, **Yannis.getIncome(d1)** is a valid operation call.

The type of the result of this operation call is Tuple(bonus: Integer, result: Integer).

We can access these values using the names of the out parameters, and the keyword *result*, for example:

Yannis.getIncome(d1).bonus = 300 and
Yannis.getIncome(d1).result = 1000



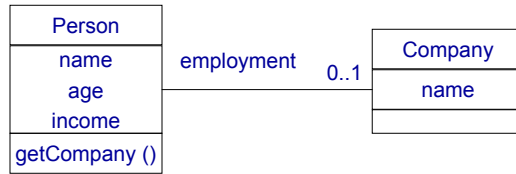
Body: Indicating the result of a query operation

An OCL expression may be used to indicate the result of a query operation.

The expression must conform to the result type of the operation.

Like in the pre/post-conditions, the parameters may be used in the expression.

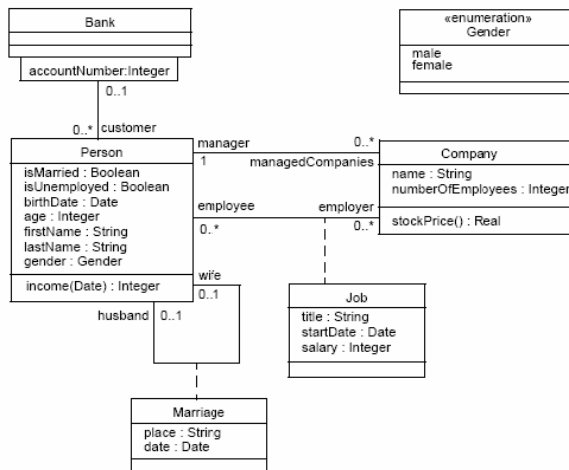
Pre/post-conditions, and body expressions may be mixed together after one operation context.



Context Person::getCompany():Company
pre: self.employment->size(>)>0
body: self.employment



Body: Indicating the result of a query operation (II)



Context Person::getCurrentSpouse():Person
pre: self.isMarried = true
body: self.marriages->select(m | m.ended = false).spouse



Αρχικά και Παραγόμενα Γνωρίσματα Initial and Derived Attributes

An OCL expression may be used to indicate the initial or derived value of an attribute or association end.

context Typename::attributeName: Type
init: -- some expression representing the initial value

context Typename::assocRoleName: Type
derive: -- some expression representing the derivation rule

The expression must conform to the result type of the attribute.

If the context is an association end the expression must conform to the classifier at that end when the multiplicity is at most one, or Set or OrderedSet when the multiplicity may be more than one. Initial, and derivation expressions may be mixed together after one context.

```
Context Person::income: Integer
init: parents.income->sum()*1%      -- pocket allowance
derive: if underAge
      then parents.income->sum()*1% -- pocket allowance
      else job.salary                -- income from regular job
endif
```



Let Expressions

Sometimes a sub-expression is used more than once in a constraint.

The **let expression** allows one to define a variable which can be used in the constraint.

```
context Person inv:
let income : Integer = self.job.salary->sum() in
if isUnemployed then
  income < 100
else
  income >= 100
endif
```

A let expression may be included in any kind of OCL expression. It is only known within this specific expression.



«definition» expressions

The Let expression allows a variable to be used in one OCL expression.
To enable reuse of variables/operations over multiple OCL expressions we can use the stereotype «definition».

All variables and operations defined in the «definition» constraint are known in the same context as where any property of the Classifier can be used.

The syntax of the attribute or operation definitions is similar to the Let expression, but each attribute and operation definition is prefixed with the keyword 'def'.

context Person

def: income : Integer = self.job.salary->sum()

def: nickname : String = 'Little Red Rooster'

def: hasTitle(t : String) : Boolean = self.job->exists(title = t)

The names of the attributes / operations in a let expression may not conflict with the names of respective attributes/ associationEnds and operations of the Classifier.



Re-typing or casting

In some circumstances, it is desirable to use a property of an object that is defined on a subtype of the current known type of the object. Because the property is not defined on the current known type, this results in a type conformance error.

When it is certain that the actual type of the object is the subtype, the object can be re-typed using the operation **oclAsType(OclType)**. This operation results in the same object, but the known type is the argument OclType.

When there is an object obj of type Type1 and Type2 is a subtype of Type1, then it is allowed to write:

obj1.oclAsType(Type2) --- evaluates to object with type Type2



Accessing overridden properties of supertypes

Whenever properties are redefined within a type, the properties of the supertypes can be accessed using the `oclAsType()` operation.

Whenever we have a class B as a subtype of class A, and a property p1 of both A and B, we can write:

context B inv:

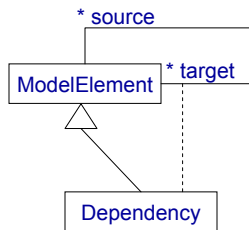
self.oclAsType(A).p1

-- accesses the p1 property defined in A

self.p1

-- accesses the p1 property defined in B

In this model fragment there is an ambiguity with the OCL expression on Dependency:



context Dependency inv:

self.source <> self

This can either mean normal association navigation, which is inherited from ModelElement, or it might also mean navigation through the dotted line as an association class. Both possible navigations use the same role-name, so this is always ambiguous. Using `oclAsType()` we can distinguish between them with:

context Dependency

inv: self.oclAsType(Dependency).source->isEmpty()

inv: self.oclAsType(ModelElement).source->isEmpty()



Predefined properties on all objects

There are several properties that apply to all objects, and are predefined in OCL.

- **oclIsTypeOf (t: OclType) : Boolean**

- returns true if the type of **self** and **t** are the same, e.g.

context Person

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

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

- **oclIsKindOf (t: OclType) : Boolean**

- The `oclIsTypeOf` deals with the direct type of an object. The `oclIsKindOf` property determines whether **t** is either the direct type or one of the supertypes of an object.

- **oclInState (s: OclState) : Boolean**

- will be discussed later on

- **oclIsNew () : Boolean**

- It returns true if, used in a postcondition, the object is created during performing the operation. i.e., it didn't exist at precondition time.

- **oclAsType (t : OclType) : instance of OclType**

- *we have discussed this already*



.allInstances()

All properties discussed until now in OCL are properties on instances of classes. The types are either predefined in OCL or defined in the class model. In OCL, it is also possible to use features defined on the types/classes themselves. These are, for example, the class-scoped features defined in the class model. Furthermore, several features are predefined on each type.

A predefined feature on classes, interfaces and enumerations is **allInstances**, which results in the Set of all instances of the type in existence at the specific time when the expression is evaluated.

Example

We want to make sure that all instances of Person have unique names:

```
context Person
inv: Person.allInstances()->forAll(p1, p2 | p1 <> p2 implies p1.name <> p2.name)
```

The Person.allInstances() is the set of all persons that exist in the system at the time that the expression is evaluated and is of type Set(Person).



Type conformance rules

Type conformance rules:

- Type1 conforms to Type2 when they are identical or when Type1 is a subtype of Type2 (standard rule for all types).
- Collection(Type1) conforms to Collection(Type2), when Type1 conforms to Type2. This is also true for Set(Type1)/ Set(Type2), Sequence(Type1)/ Sequence(Type2), Bag(Type1)/Bag(Type2)
- The types Set (X), Bag (X) and Sequence (X) are all subtypes of Collection (X).

Type conformance is transitive: if Type1 conforms to Type2, and Type2 conforms to Type3, then Type1 conforms to Type3 (standard rule for all types).

For example, if Bicycle and Car are two separate subtypes of Transport:

```
Set(Bicycle) conforms to Set(Transport)
Set(Bicycle) conforms to Collection(Bicycle)
Set(Bicycle) conforms to Collection(Transport)
```

However

```
Set(Bicycle) does not conform to Bag(Bicycle), nor the other way around.
They are both subtypes of Collection(Bicycle) at the same level in the hierarchy.
```

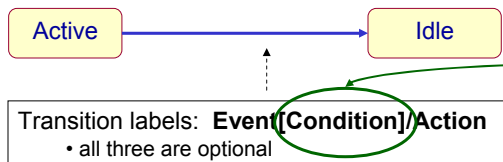


Χρήση εκφράσεων OCL στα μοντέλα UML (πέραν των διαγραμμάτων κλάσεων)

Use of OCL expressions in UML models
(apart from class diagrams)



OCL και Διαγράμματα Καταστάσεων OCL and State Diagrams



- **Event:**
 - if nil then when the task is completed we continue
- **Condition**
 - logical condition (transition occurs if its value is True)
 - the guards of transitions from a state must be mutually exclusive so that to have a unique next state
- **Action**
 - processes that occur quickly and are not interruptible

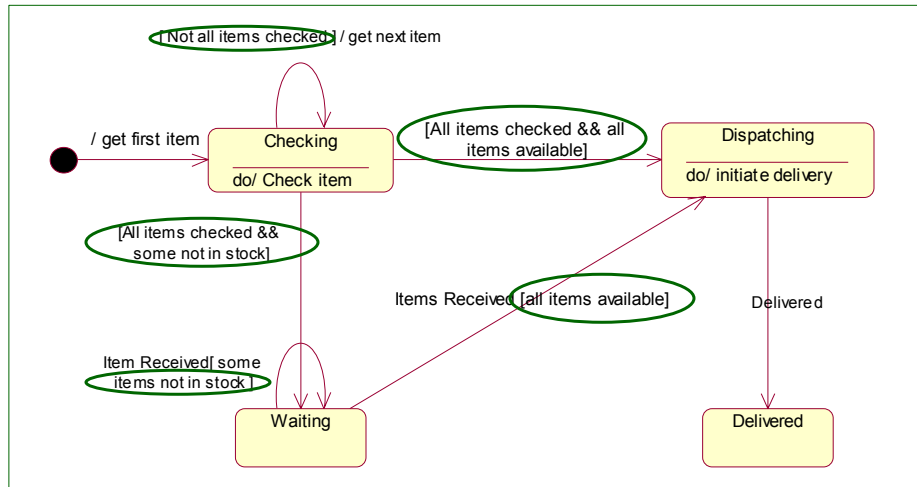
OCL expression

An OCL expression acting as value of a guard is of type Boolean.

The expression is evaluated at the moment that the transition attached to the guard is attempted



OCL and State Diagrams (II)



OCL and State Diagrams (III)

Predefined properties on all objects

oclInState (s : OclState) : Boolean

oclInState (s : OclState) : Boolean

This operation returns true if the object is in the state s.

Values for s are the names of the states in the statemachine(s) attached to the Classifier of object. For nested states the statenames can be combined using the double colon.



Here the values for s can be

- On
- Off
- Off::Standby
- Off::NoPower.

If the classifier of object has the above associated statemachine valid OCL expressions are:

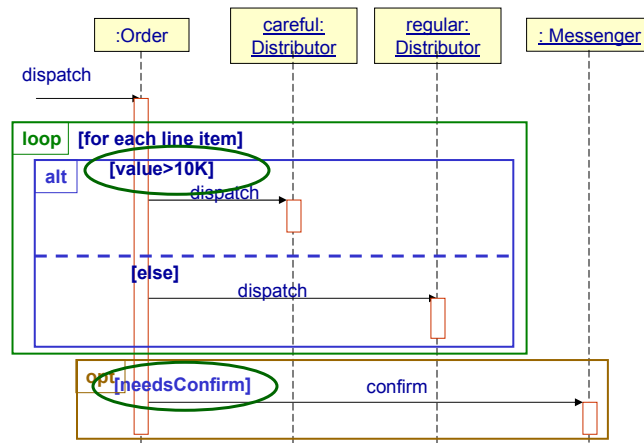
- object.oclInState(On)
- object.oclInState(Off)
- object.oclInState(Off::Standby)
- object.oclInState(Off::NoPower)

If there are multiple statemachines attached to the object's classifier, then the statename can be prefixed with the name of the statemachine containing the state and the double colon '::', as with nested states.



ΟCL και Διαγράμματα Αλληλεπίδρασης OCL and Interaction Diagrams

OCL can be used for expressing the conditions under which a message (in a sequence or communication diagram) is sent



Ποια εργαλεία CASE υποστηρίζουν την OCL και πως; Which UML CASE Tools support OCL and how?

- We can attach OCL constraints to our diagrams using an appropriate stereotype and a dashed line should connect it to its contextual element
- OCL constraints are exchanged using XMI
- Tools that support OCL
 - ArgoUML allows expressing them
 - OCL Evaluator (a tool for editing, syntax checking & evaluating OCL)
 - Octopus OCL 2.0 Plug-in for Eclipse



Assertions and Programming Languages



Assertions and Programming Languages

- Assertion techniques (preconditions, postconditions, invariants)
- History of assertion techniques:
 - Hoare 1972
 - Meyer 1971 (he proposed the idea Design by Contract)
- Assertions support in Programming Language:
 - Eiffel supports them
 - In Java it is also possible (e.g. using JAF)



Techniques for adding Assertion Support in a PL

- **Built in**
 - Syntactic correctness of assertions is checked by the compiler
 - The runtime environment performs the runtime assertion checks
- **Preprocessing**
 - Formulate assertions separate from the program or include the assertions as comments. A preprocessor translates the assertions to program code
 - Pros : separation (separation of programmatic logic from contracts)
 - Con: the original program code is modified (e.g. the line numbers of compiler errors do not fit the line numbers of the program)
- **Metaprogramming**
 - Traditionally this is possible only in dynamically typed and interpreted languages
 - Programs that have the possibility to reason about themselves have so called **reflective capabilities** (Java has a reflection API)
 - The main advantage of metaprogramming approaches is that no specialized preprocessor has to be used but the native compiler. Nevertheless a specialized runtime environment has to be used to enable assertion checking



Assertions and Java

- "An *assertion* is a statement containing a boolean expression that the programmer believes to be true at the time the statement is executed".
- It is a facility provided within the java programming language to test the correctness or assumptions made by your program. Assertions are checks provided within the system to ensure the smooth running of the program.
- **Why Assertions?**
- *Why we need another level of checking when exceptions can do the job?*
- Exceptions are primarily used to handle unusual (abnormal) conditions arising during program execution.
 - They do not guarantee smooth or correct execution of the program.
- **Assertions are used to specify conditions that a programmer assumes are true.**
 - If a programmer can swear that the value being passed into a particular method is positive no matter what a calling client passes, it can be documented using an assertion to state it. Assertions help state scenarios that ensure the program is running smoothly. Assertions can be efficient tools to ensure correct execution of a program. They also improve the confidence about the program.
 - We can turn them off



Java Assertion Facility (JAF)

Syntax `assert expression1;`

The expression is the one we wish to assert as true. If the assumption fails, the expression evaluates to be false which means the assertion failed. In case the expression succeeds the program continues normally.

When an assertion fails the program throws an `AssertionError` on to the stack trace.

Examples:

```
assert i<0;
assert (!myString.equals(""));
```

Java Assertion Facility (JAF)
(builtin since Java 1.4)



Java Assertion Facility (JAF)

Syntax `assert expression1 : expression2;`

The first argument takes a Boolean expression, while the second expression would be the resulting action to be taken if the assertion fails. The *Expression2* should be a value and can also be a result of executing a function. The compiler would throw an error if the second expression returns a void value.

When an assertion fails the program throws an `AssertionError` on to the stack trace. The program creates an object `AssertionError` with the return type of *Expression2*. The overloaded `AssertionError` constructor would then convert the returned data type into `String` and dump it on the stack trace with a meaningful message.

Examples:

```
assert age>0 : "The value of age cannot be negative" +age;
assert ((i/2*23-12)>0):checkArgumentValue();
assert isParameterValid():throwIllegalParameterError();
    In the last example the method checkArgumentValue() must return a value
```




OCL Constraints and Java

Context Account:withdraw(amount:Real)

pre: amount <= balance

post: balance = balance@pre - amount

Context Account:getBalance():Real

post: result = balance

```

class Account {
    private float balance = 0 ;
    public void withdraw(float amount){
        assert amount<= balance;
        balance = balance - amount;
    }

    public float getBalance(){
        return balance;
    }
}

```



OCL Constraints in Java (2)

Context Employee:SetAge (age)

pre: age > 0

```

class Employee {
    public void SetAge(int age){
        assert age>0;
        this.age = age;
    }
}

```

Design by contract

```

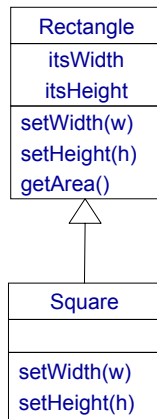
class Employee {
    public void SetAge(int age) throws ArgumentException {
        if (age<=0) {
            throw new ArgumentException("negative age");
        }
        this.age = age;
    }
}

```

Defensive Programming
(throwing exceptions)



Example: Rectangle and Square



```

class Rectangle {
public:
    virtual void setWidth(double w) {itsWidth=w;}
    virtual void setHeight(double h){itsHeight=h;}
    double getArea() {return itsHeight * itsWidth;}
private:
    double itsWidth;
    double itsHeight;
};
  
```

```

class Square: public Rectangle {
public:
    virtual void setWidth(double w);
    virtual void setHeight(double h);
};
  
```

```

void Square::setWidth(double w)
{Rectangle::setWidth(w); Rectangle::setHeight(w); }
void Square::setHeight(double h)
{Rectangle::setWidth(h); Rectangle::setHeight(h); }
  
```



Example: Rectangle and Square (II)

```

class Rectangle {
public:
    virtual void setWidth(double w) {itsWidth=w;}
    virtual void setHeight(double h){itsHeight=h;}
    double getArea() {return itsHeight * itsWidth;}
private:
    double itsWidth;
    double itsHeight;
};
  
```

```

class Square: public Rectangle {
public:
    virtual void setWidth(double w);
    virtual void setHeight(double h);
};
  
```

```

void Square::setWidth(double w)
{Rectangle::setWidth(w); Rectangle::setHeight(w); }
void Square::setHeight(double h)
{Rectangle::setWidth(h); Rectangle::setHeight(h); }
  
```

```

void g(Rectangle* r)
{
    r->setWidth(5);
    r->setHeight(4);
    assert(r->getArea()==20);
}
  
```

It will function correctly if **r** is a rectangle.

It will not function correctly if **r** is a square

The class Square actually violates an "invariant" of the class Rectangle, specifically the width-height independence.



Example: Rectangle and Square (III)

This could be expressed in OCL with

a post condition of `setWidth`: *i.e. the height is the old value of height;*

and a post condition of `setHeight` *i.e. the width is the old value of width.*

Context Rectangle:setWidth(w)

post: itsWidth = w and
itsHeight = itsHeight@pre

Context Rectangle:setHeight(w)

post: itsHeight = h and
itsWidth = itsWidth@pre

[Meyers]:

When we override a method A with a method B

the precondition of B should be that of A or a weaker condition, and

the postcondition of B should be that of A or a stronger (more strict) condition.

This reveals the problem in our example: the postcondition of `Square:setWidth` is weaker (although it should be stronger according to the above rule).

So, if for example we had copied the postconditions of the `Rectangle`'s methods to the methods of `Square`, we would have seen the problem while testing the class `Square`.



Where things go

- At the beginning each application was dependent on the hardware of the machine (machine code programming)
- With compilers (i.e. programming languages) each application can be compiled for different machines and OSs assuming there is a compiler for them (one compiler is needed, thousands of applications exploit it)
- With Java and bytecodes even the compilation is somehow "bypassed"
- With UML and OCL the specification of an application can be independent even from the PL



Χρήσιμες αναφορές

- Που μπορείτε να βρείτε την προδιαγραφή (specification) της UML 2.0 OCL
 - <http://www.omg.org/cgi-bin/doc?ptc/2005-06-06>
 - http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML
- Εργαλεία
 - OCL Evaluator (a tool for editing, syntax checking & evaluating OCL)
 - Octopus OCL 2.0 Plug-in for Eclipse
- Μια διαδικτυακή πύλη για την OCL
 - <http://www-st.inf.tu-dresden.de/ocl/>
- Ένα σχετικό βιβλίο
 - J. Warmer and A. Kleppe, "The Object Constraint Language: Precise Modeling with UML", Addison-Wesley 1999.
- Άλλες πηγές
 - http://en.wikipedia.org/wiki/Object_Constraint_Language
 - <http://www.omg.org/docs/ad/99-12-05.pdf>
 - <http://www.brucker.ch/projects/hol-ocl/>
 - <http://www.eclipse.org/articles/Article-EMF-Codegen-with-OCL/article.html> (for EMF)