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

Πανεπιστήμιο Κρήτης, Φθινόπωρο 2005

Φροντιστήριο 6

Θέμα : Object Constraint Language

Ημερομηνία : 5 Δεκεμβρίου 2005

Outline

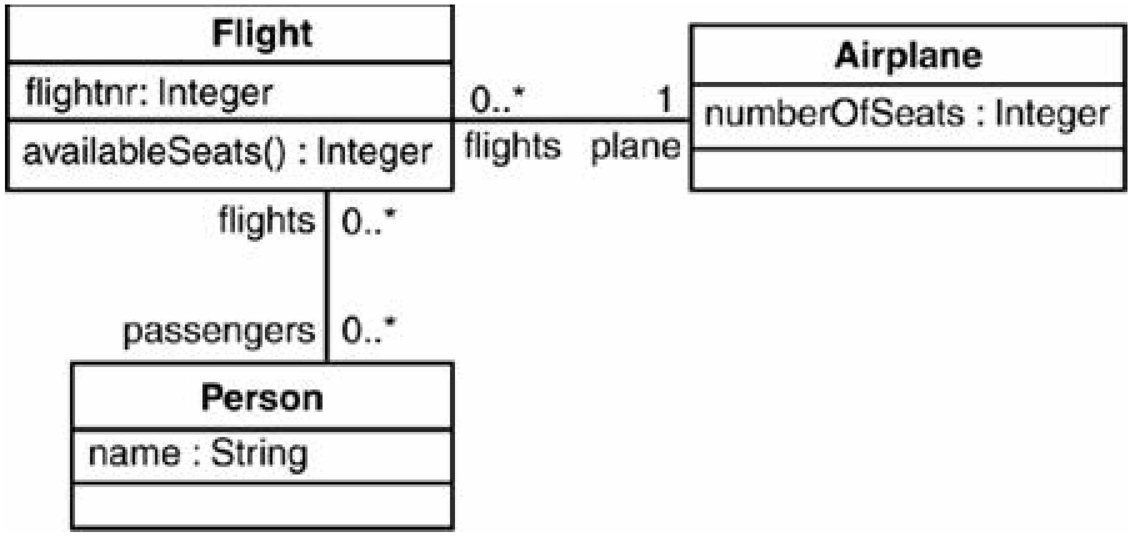
- What is OCL?
- OCL Basics
- OCL Tools
- Useful Links

What is OCL?

- OCL is a language that can express additional and necessary information about the models used in object-oriented modeling
- Much more information can be included in the model using the combination of OCL and UML than through the use of UML alone
- Many essential aspects of the system cannot be expressed in a UML diagram.
- OCL is a precise language that is easy for people and doesn't use any mathematical symbols
- It is a typed language, so that each OCL expression has a type
- OCL is a declarative language. It specifies what should be calculated, but not how to calculate the value of an expression.

Why Combine UML and OCL?

- For instance, we have the following UML model :



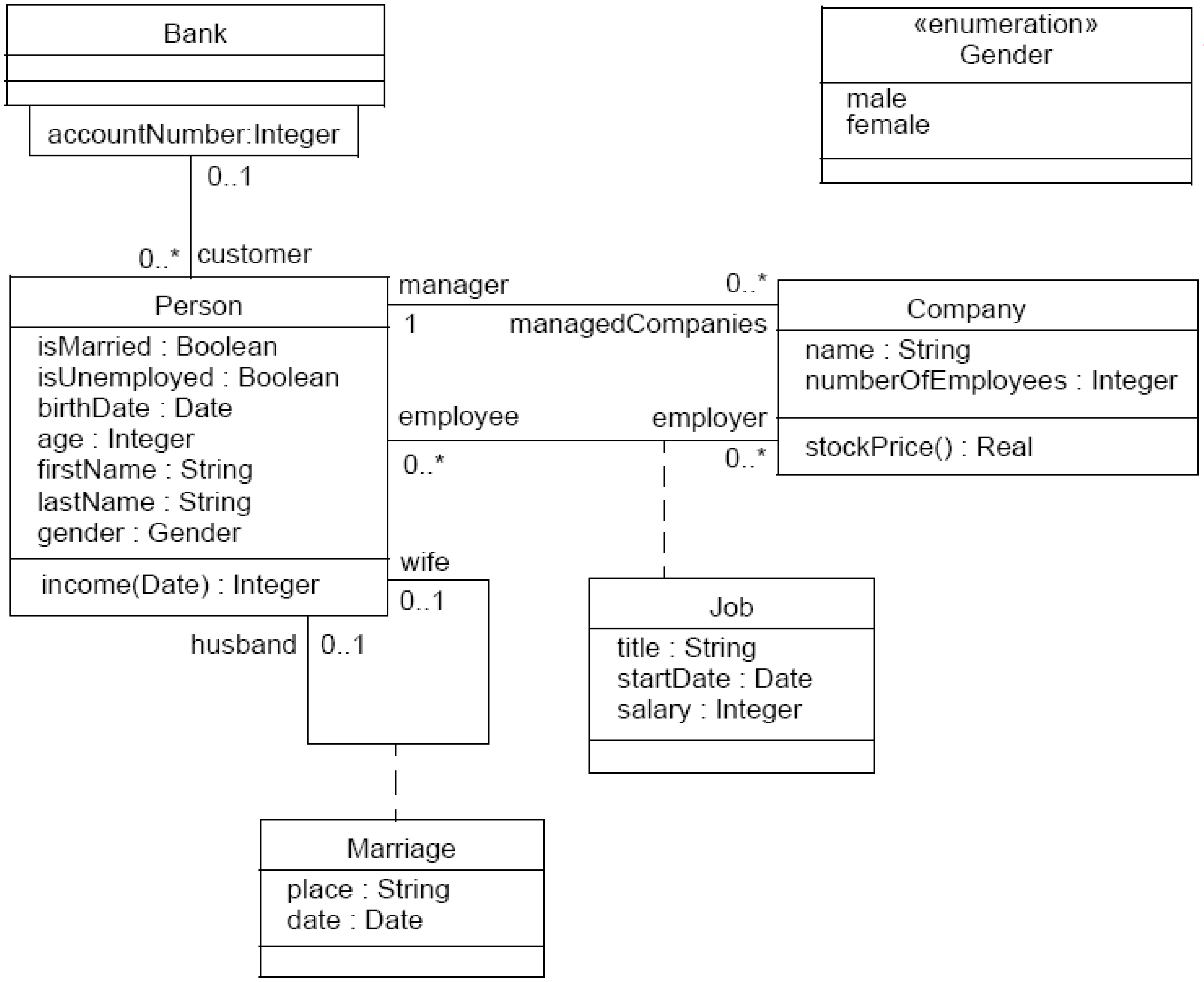
- There is an association between class Flight and class Person, which indicates that the number of passengers on a flight is unlimited
- In reality, the number of passengers will be restricted to the number of seats on the airplane that is associated with the flight
- The correct way to specify the multiplicity is to add to the diagram the following OCL constraint

  context Flight inv: passengers->size() <= plane.numberOfSeats

Where can OCL Expressions be used?

- OCL expressions can be used in a number of places in UML models.
- The class diagram may benefit from OCL expressions invariants, derivation rules, preconditions and postconditions.
- The interaction diagram and activity diagram can be improved by specifying instances and actual parameter values, and by stating conditions.
- The statechart may be augmented with guards
- In a use case, the pre- and postconditions can be written using OCL.

Class Diagram Example



Invariants

- An *invariant* is a constraint that states a condition that must always be met by all instances of the class, type, or interface (context).
- An invariant is described using an expression that evaluates to true if the invariant is met.
- The *context definition* of an OCL expression specifies the model entity for which the OCL expression is defined.
- The following OCL expression would specify an invariant that the number of employees must always exceed 50:

  context *Company*

  inv*: self.numberOfEmployees > 50*
- *self* is an instance of type Company (optional here)
- The label *inv:* declares that the constraint is an «invariant» constraint.

AssociationEnds and Navigation

- Starting from a specific object, we can navigate an association on the class diagram to refer to other objects
- To do so, we navigate the association by using the opposite association-end:

  context Company

  inv: self.manager.isUnemployed = false

  context Company

  inv: self.employee->notEmpty()
- If the multiplicity of the association-end has a maximum of one (“0..1” or “1”), then the value of this expression is an object
- In the first invariant self.manager is a Person, because the multiplicity of the association is one.
- In the second invariant self.employee will evaluate in a Set of Persons.
- A Set is a subtype of a *Collection*.

Collections

- Collection is a predifined abstract type in OCL that plays important role in OCL expressions.
- OCL distinguishes three different collection types: Set, Sequence, and Bag
- A Set is like a mathematical set that it does not contain duplicate elements.
- A Bag is like a set, which may contain duplicates
- A Sequence is like a Bag in which the elements are ordered.
- They have a large number of predefined operations on them.
- An operation of the collection itself is accessed by using an arrow '->' followed by the name of the operation.

Collection Operations(1/3)

- The following example is in the context of a person:

  context Person

  inv: self.employer->size() < 3

  This applies the size operation on the Set self.employer, which results in the number of employers of the Person self.
- context Person

  inv: self.employer->isEmpty()
- This applies the *isEmpty* operation and evaluates to true if the set of employers is empty and false otherwise.
- The parameter of *select* operation has a special syntax that enables one to specify which elements of the collection we want to select.
- As an example, the following OCL expression specifies the collection of all the employees older than 50 years :

  context Company inv: self.employee->select(age > 50)

Collection Operations(2/3)

- The *reject* operation is identical to the select operation, but with reject we get the subset of all the elements of the collection for which the expression evaluates to False.
- As an example, specify that the collection of all the employees who are not married

  context Company

  inv: self.employee->reject(isMarried)
- The forAll operation in OCL allows specifying a Boolean expression, which must hold for all objects in a collection
- This forAll expression results in a Boolean. For example, in the context of a company:

  context Company

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

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

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

Collection Operations(3/3)

- 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.
- An example: specify the collection of birthDates for all employees in the context of a company:

  self.employee->collect(birthDate)

  or

  self.employee.birthdate
- The *exists* operation in OCL allows you to specify a Boolean expression which must hold for at least one object in a collection:
- For example, in the context of a company:

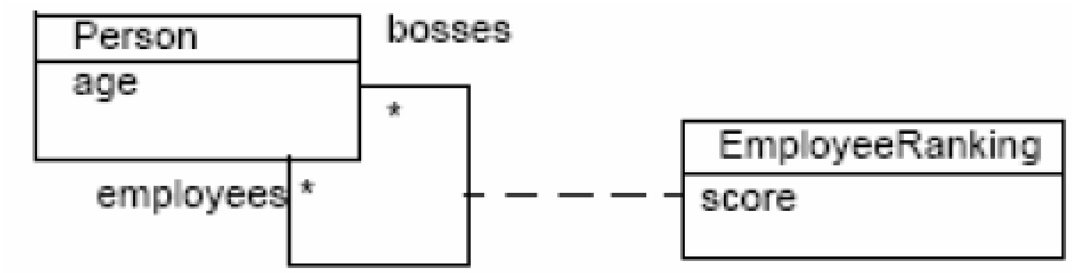
context Company inv: self.employee->exists(forename = 'Jack')

- These expressions evaluate to true if the forename attribute of at least one employee is equal to ‘Jack.’

Navigation to Association Classes (1/2)

- To specify navigation to association classes (Job and Marriage in the example), OCL uses a dot and the name of the association class starting with a lowercase character:

  context Person inv:self.job
- The sub-expression self.job evaluates to a Set of all the jobs a person has with the companies that are his/her employer.
- In case of a recursive association, that is an association of a class with itself, the name of the association class alone is not enough.
- Take the following model as an example:



- When navigating to an association class such as employeeRanking there are two possibilities depending on the direction (towards the employees or the bosses end.

Navigation to Association Classes (2/2)

- 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
- In the expression

  context Person

  inv: self.employeeRanking[bosses]->sum() > 0

  the self.employeeRanking[bosses] evaluates to the set of EmployeeRankings belonging to the collection of bosses.
- the following example is invalid:

  context Person

  inv:self.employeeRanking->sum() > 0 -- INVALID

Enumerations and Let Expressions

- *Enumerations* are Datatypes in UML.
- An enumeration defines a number of enumeration literals, that are the possible values of the enumeration.
- Within OCL one can refer to the value of an enumeration.
- When we have Datatype named Gender in the example model with values 'female' or 'male' they can be used as follows:

  context Person inv: gender = Gender::male
- The *let* expression allows one to define a variable which can be used in the constraint.
- The following example defines the variable *income*

  context Person inv:

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

  if isUnemployed then income < 100 else income >= 100 endif
- *If Expressions* are defined in the following manner:

  if <boolean OCL expression> then <OCL expression> else <OCL expression> endif

Defining New Attributes and Operations

- The Let expression allows a variable to be used in one OCL expression.
- To enable reuse of variables/operations over multiple OCL expressions one can use a constraint through *definition* expressions, in which helper variables/operations are defined.
- This definition constraint must be attached to a context and may only contain variable and/or operation definitions, nothing else.
- 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’ as shown below.
- *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)

Preconditions and Postconditions

- A *precondition* to an operation is a restriction that must be true at the moment that the operation is going to be executed.
- A *postcondition* to an operation is a restriction that must be true at the moment that the operation has just ended its execution.
- The context declaration in OCL uses the *context* keyword, followed by the type and operation declaration.
- *Context Person::hire(c:Company)*

  pre: not employment->includes(c)
- The reserved word *result* denotes the result of the operation, if there is one. It can be used in the postcondition only.
- The following OCL expression specify a postcondition that the execution of the *income* operation returns 5000

  context *Person::income(d : Date) : Integer*

  post*: result = 5000*

Previous Values in Postconditions

- In a postcondition, the expression can refer to values at two moments in time:

  i) The value at the start of the operation

  ii) The value upon completion of the operation
- To refer to the value of a property at the start of the operation, one has to postfix the property name with the keyword '@pre':

  context Person::birthdayHappens()

  post: age = age@pre + 1
- The term age refers to the value of the attribute after the execution of the operation.
- The term age@pre refers to the value of the attribute age at the start of the operation.
- If the property has parameters, the '@pre' is postfixed to the propertyname, before the parameters.

  context Company::hireEmployee(p : Person)

  post: stockprice() = stockprice@pre() + 10
- The '@pre' postfix is allowed only in OCL expressions that are part of a Postcondition.

Tuples

- It is possible to compose several values into a tuple.
- A tuple consists of named parts, each of which can have a distinct type.
- Some examples of tuples are:

  Tuple {name: String = ‘John’, age: Integer = 10}
- When an 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).

Messaging in Postconditions (1/2)

- Another thing allowed only in postconditions is specifying that communication has taken place.
- This can be done using the hasSent ('^') operator:

  context Subject::hasChanged()

  post: observer^update(12, 14)
- The observer^update(12, 14) results in true if an update message with arguments 12 and 14 was sent to the observer object during the execution of the operation hasChanged().
- update() is either an operation defined in the class of observer, or a signal specified elsewhere in the UML model.
- The argument(s) of the message expression (12 and 14 in this example) must conform to the parameters of the operation/signal definition.
- During execution of an operation many messages may have been sent that represent calls to the same operation, or send signals according to the same signal definition.

Messaging in Postconditions (2/2)

- Any operation call or signal being sent is virtually wrapped in an instance of OclMessage, a special type defined by OCL
- One can obtain access to all OclMessages that wrap a matching call or signal, through the message operator (denoted as ^^).
- A call or signal matches when the operation name and the argument types given after the message operator correspond to the formal definition of the operation or signal, as shown in the following example:

  observer^^update(12, 14)
- This expression results in the sequence of messages sent that match update(12, 14) being sent by the contextual instance to the object called observer during the execution of the operation.
- Each element of the sequence is an instance of OclMessage.
- The following postcondition is semantically equal to the postcondition in the previous slide

  context Subject::hasChanged()

  post: observer^^update(12, 14)->notEmpty()

OCL Tools (1/2)

- Support tools aimed at making this language easier to use are becoming available.
- These tools, both commercial and freely available ones, are capable of supporting and handling OCL expressions.
- The **OCL Compiler** (OCLCUD), by the University of Dresden, has been created in Java. There are two ways of working with the parser. OCLCUD can be used independently as the ocl compiler demo applet or as part of Argo/UML. The main characteristics of this analyser are the syntactical and semantic checking of OCL expressions and the possibility of generating Java code and SQL from the constraints written in OCL.

OCL Tools (2/2)

- **Octopus** is an Eclipse plugin, which is able to check the syntax of OCL expressions, as well as the types and correct use of model elements like association roles and attributes.
- **OCLE** version 2.0 is a UML CASE Tool offering full OCL support both at the UML metamodel and model level. You can use UML models saved in XMI 1.0 or 1.1 versions, regardless of the tools and parsers used in producing and transferring models.
- EmPowerTec offers an **OCL-AddIn for Rational Rose** that offers comprehensive support for OCL 2.0 including full syntactic and semantic checking. It provides everything for using OCL efficiently and productive in Rational Rose models.

Useful Links

- OCL 2.0 Specification:
  www.omg.org/docs/ptc/03-10-14.pdf
- Klasse Objecten OCL Center :
  www.klasse.nl/ocl/
- Dresden OCL Toolkit :
  dresden-ocl.sourceforge.net/
- ArgoUML
  argouml.tigris.org/
- OCLE 2.0 - Object Constraint Language Environment :
  lci.cs.ubbcluj.ro/ocle/
- Octopus OCL Tool
  www.klasse.nl/english/research/octopus-intro.html
- OCL-AddIn for Rational Rose
  www.empowertec.de/products/rational-rose-ocl.htm