

UML Object Constraint Language



A constraint is a restriction on one or more values of (part of) an object-oriented model or system.
UML OCL as specification language for
Building complete models with OCL

Dr. Bandu B. Meshram,
PhD(Computer Engineering),
LLM (Constitutional Law), CHFI
Professor and Former Head
Department of Computer Technology,
VJTI, Matunga,Mumbai-19
bbmeshram@ce.vjti.ac.in (9022358788)

OCL is a modeling language

- OCL is a typed language, so that each OCL expression has a type.
- An OCL expression must conform to the type conformance rules of the language.
- For example, you cannot compare an Integer with a String.

Where to Use OCL?

OCL can be used for a number of different purposes:

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

UML constraint

A constraint is a restriction on one or more values of (part of) an object-oriented model or system.

- A constraint is formulated on the level of classes, but its semantics is applied on the level of objects.

CONSTRAINTS IN CLASS DIAGRAM

- Construct class diagram with constraints
- For a constraint that applies to a **single element** (such as a **Class or an association path**), the constraint string may be placed near the symbol for the element, preferably near the name, if any.

Different kinds of constraints

“A constraint is a restriction on one or more values of (part of) an object-oriented model or system.”

Different kinds of constraints

- **Class invariant** ,a constraint that must always be met by all instances of the class.
- **Precondition of an operation** ,a constraint that must always be true BEFORE the execution of the operation
- **Postcondition of an operation** ,a constraint that must always be true AFTER the execution of the operation

Class Invariant

Invariant – 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.

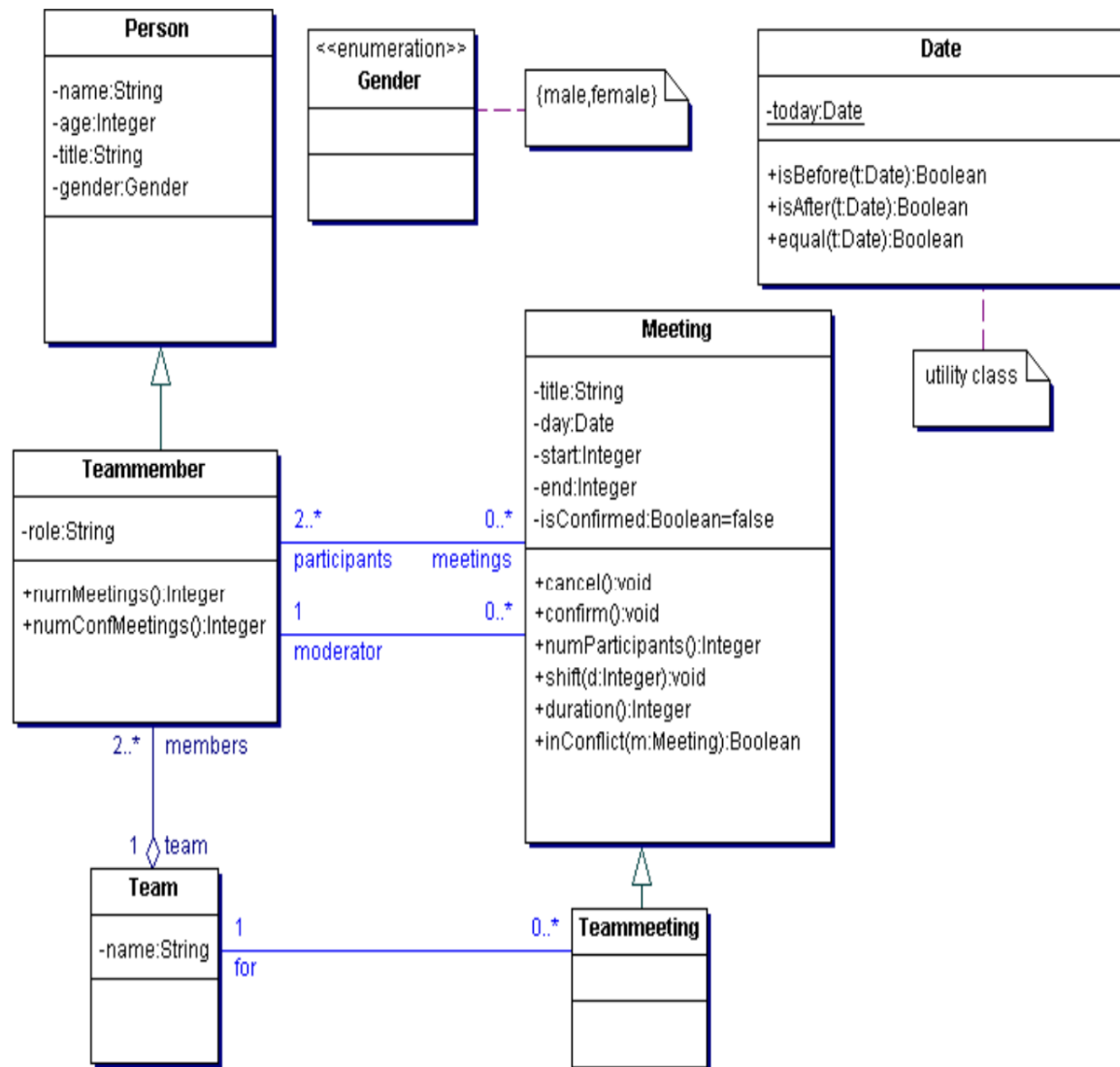
Syntax

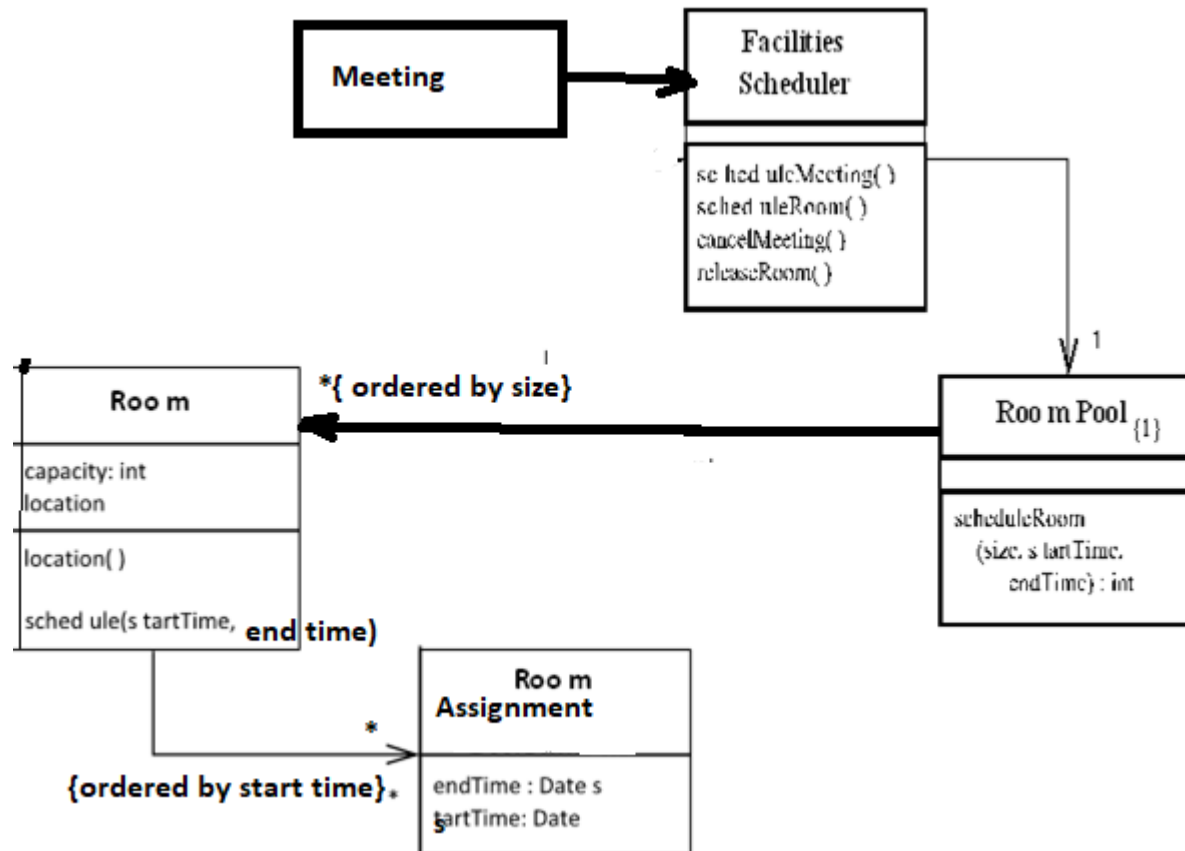
context < CLASSIFIER>

Inv [<constraint name>]: < Boolean OCL expression>

CASE STUDY- schedule meetings and meeting rooms

Design a system to schedule meetings and meeting rooms. A user can use this system simply to request a room of a given size for a given period of time. In addition, a user can request that an existing meeting (already defined in the system with a set of attendees) be scheduled at with a particular starting time and ending time. A user can cancel any scheduled meeting or any room assignment up until the point at which the meeting or assignment begins. When a meeting is scheduled, an electronic message about that meeting must be sent to each attendee. Likewise, when a meeting is canceled, each attendee must be informed by electronic mail about the cancellation. A user must also be able to define or alter a meeting. When defining the meeting, the user provides a list of attendees. The user may alter a meeting definition by adding attendees to or removing attendees from the meeting. A user may also remove an entire meeting definition. Note that adding or removing attendees has no effect on scheduled instances of that meeting (unless the last attendee is removed from a meeting, in which case future scheduled occurrences of that meeting should be canceled). A result of removing a meeting, on the other hand, is that all scheduled instances of that meeting must be canceled.





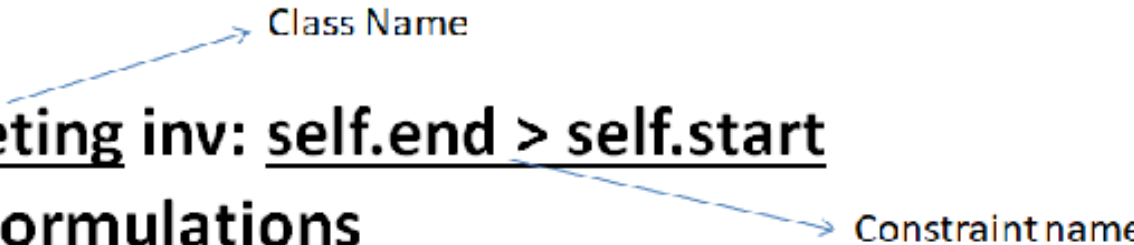
Invariant – Examples

context Meeting inv: self.end > self.start

Equivalent Formulations

context Meeting inv: end > : end > start

– "self" always refers to the object identifier from which the constraint is evaluated.



context Meeting inv **startEndConstraint**: self.end > self.start



-- **Names** can be given to the constraint

Precondition

Precondition– Constraint that must be true just prior to the execution of an operation

Syntax

```
context < classifier> :: (<operation>) (<parameters>)  
pre :[ <constraint name>]: < Boolean OCL expression>
```

Precondition - Examples context

1.context Meeting::shift(d:Integer) pre: self.isConfirmed = false

2.context Meeting::shift(d:Integer) pre: d>0 pre: d>0

Both preconditions are combined below:

3.context Meeting::shift(d:Integer) pre: self.isConfirmed = false and d>0

Postcondition

- Post condition – Constraint that must be true just after to the execution of an operation

Post conditions are the way how the actual effect of an operation is described in OCL.

Syntax

- **context** <classifier>::<operation> (<parameters>)

post [<constraint name>]: <Boolean OCL expression>

Postcondition - Examples

- **context Meeting::duration():Integer post: result = self.end – self.start**
- keyword *result* refers to the result of the operation **duration()** of class **Meeting**.
- **context Meeting::confirm() post: self.isConfirmed = true**

Postcondition – Examples (cont.)

- **context Meeting::shift(d:Integer)**

post: start = start@pre + d and end = end@pre + d

start@pre indicates a part of an expression which is to be evaluated in the original state before execution of the operation

- *start* refers to the value upon completion of the operation
- @pre is only allowed in postconditions

Postcondition – Examples (cont.)

- **messaging** only in postconditions is specifying that communication has taken place

hasSent (“^”) operator

context Subject::hasChanged()

post: observer^update(2,4)

/* standard **observer pattern**:

results in true if an update message with arguments 2 and 4 was sent to the observer object during execution of the operation *hasChanged()*

*/

Building OCL Expressions <OCL expression> (1)

- **Data types in OOPL**

type, storage , range

- **Boolean expressions**_True/false
- **Standard library** of primitive types and associated operations
- – **Basic types** (Boolean, Integer, Real, String) Classic basic primitive types may include:
 - Character (character, char);
 - Integer (integer, int, short, long, byte) with a variety of precisions;
 - Floating-point number (float, double, real, double precision);
 - Fixed-point number (fixed) with a variety of precisions and a programmer-selected scale.
- **Derived types**
- They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

Standard library

Standard library of primitive types and associated operations

List of data types of the Standard Libraries

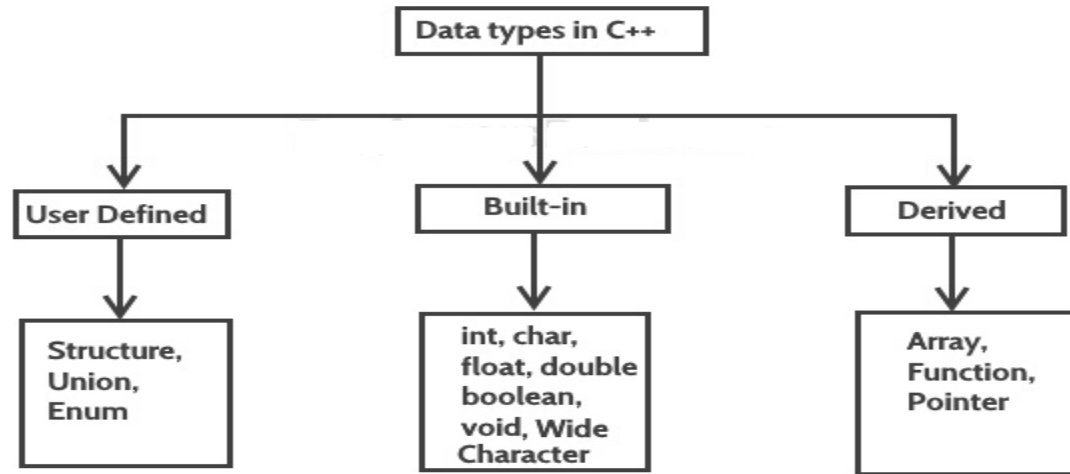
- Base Class **Library**.
- Runtime Infrastructure **Library**.
- Network **Library**.
- Reflection **Library**.
- XML **Library**.
- Extended Array **Library**.
- Extended Numerics **Library**.
- Parallel **Library**.

Collection types

- – **Collection types:**
 - Collection
 - Set
 - Ordered Set (only OCL2)
 - Bag
 - Sequence
- **Home work:** Learn how Boolean expression, primitive types and collection types are used in C++/java. Illustrate with suitable examples.

Attribute -Data types

function Return Type



In computer programming, the **return type** (or result **type**) defines and constrains the data **type** of the **value returned** from a subroutine or **method**.

In many programming languages (especially statically-typed programming languages such as **C**, **C++**, Java) the **return type** must be explicitly specified when declaring a function.

User defined types (OCLType)

Class type (Model type):

- Classifier in a class diagram (implicitly defined)
- Generalisation among classifiers leads to **Supertypes**
- A class has the following **Features**:

Attributes (start)

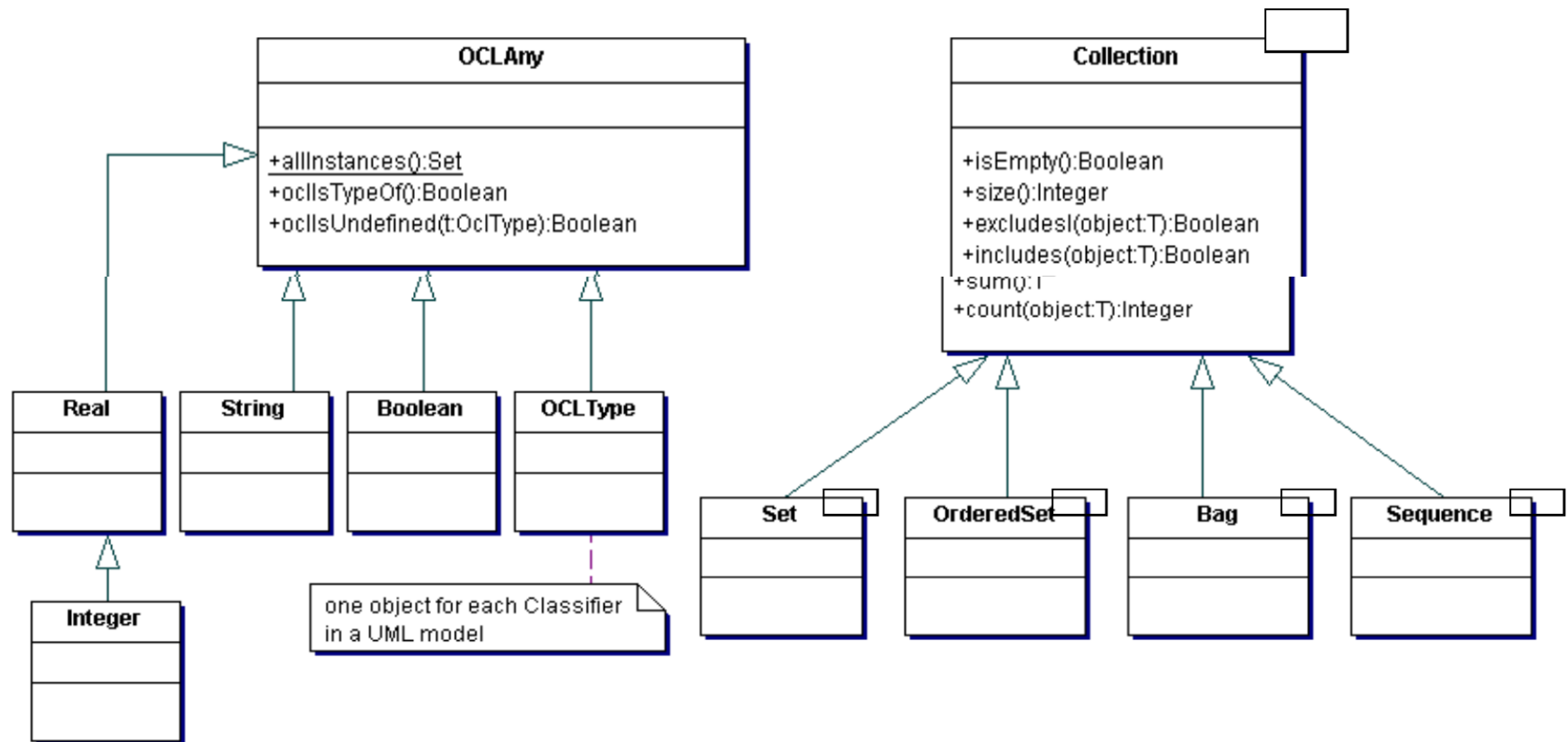
Operations (duration())

Class attributes (Date::today)

Class operations

- VOID MAIN(), int main(); int main(int, char*[]);
- My question is simple, would something like the following be legal?
- int main(const unsigned int, const char* const* argv);
 - Association ends (navigation expressions)-qualifier association
- **Enumeration type** (Gender, Gender::male)

OCL Type Hierarchy



STACK MODELLING

SPECIFY IN CLASS DIAGRAM

```
#include class stack
//Declaring a class stack
{
    int TotalElements;
    //variable for storing total elements in the stack
    int *array;
    //pointer to the array int top; //variable top to store address of top element in the
    stack
    public: stack();
    void ReadData();
    int IsEmpty();
    int IsFull();
    void push(int);
    int pop();
    void DisplayStack()
};;
```


OCL Constraints and Inheritance

- Constraints are inherited.
- **Liskov's Substitution Principle**
 - Wherever an instance of a class is expected, one can always substitute an instance of any of its subclasses.
- An **invariant** for a superclass is inherited by its subclass. A subclass may strengthen the invariant but cannot weaken it.
- A **precondition** may be weakened but not strengthened in a redefinition of an operation in a subclass.
- A **postcondition** may be strengthened but not weakened in a redefinition of an operation in a subclass.

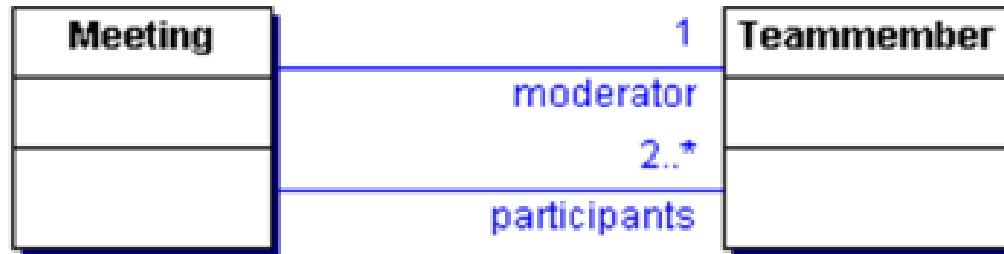
Navigation Expressions

Association ends (role names) are be used to „navigate“ from one object in the model to another object.

Navigations are treated as attributes (*dot.dot-Notation ..**).

- The type of a navigation expression is either a
 - **User defined type**
 - (association end with multiplicity at most 1)
 - **Collection**
 - (association end with multiplicity > 1)

- Navigation Expressions - Examples

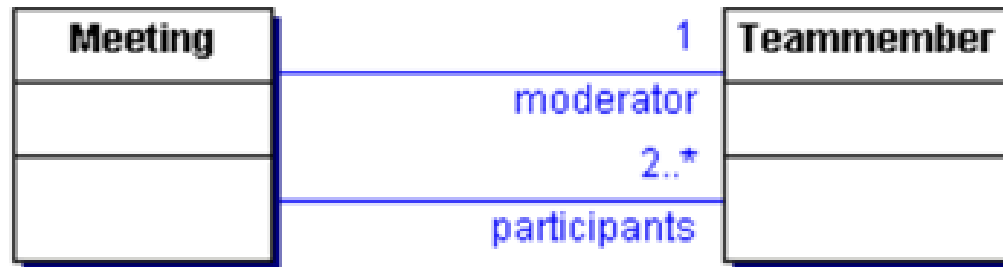


User defined type

– Navigation from Meeting to moderator results in type Teammember

context Meeting

inv: self.moderator.gender = Gender::female



- **Collection**

- Navigation von Meeting to participants results in type Set(Teammember)

- **context Meeting inv: self->collect(participants)->size()>=2**

or with **shorthand** notation:

- **context Meeting inv: self.participants->size()>=2**

Derivation Rule

- **Derived attribute** (size)

context Team::size: Integer derive: members->size()

- **Derived association** (conflict)

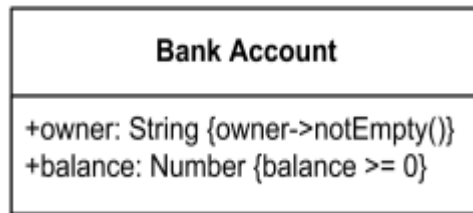
- – defines a set of meetings that are in conflict with each other

- **context Meeting::conflict:Set(Meeting)**

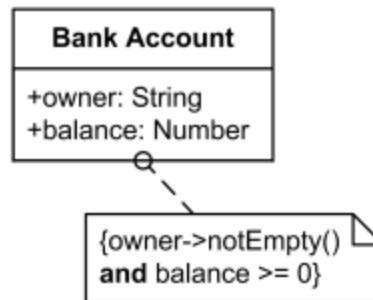
derive: select(m | m<>self and self.inConflict(m))

*Bank account **attribute constraints***

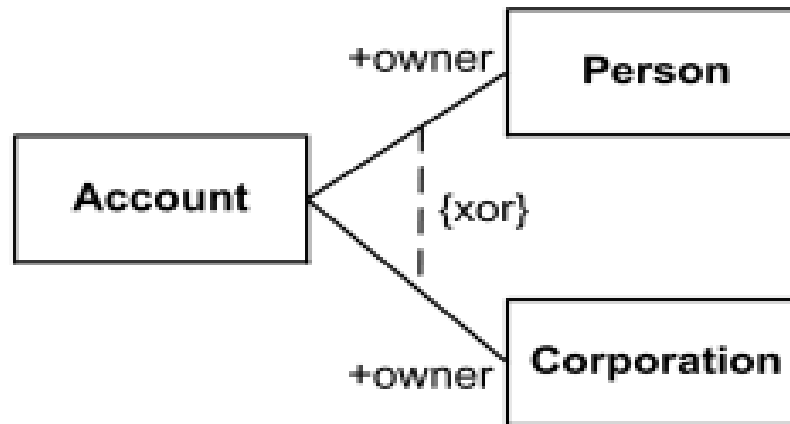
- non empty owner and positive balance



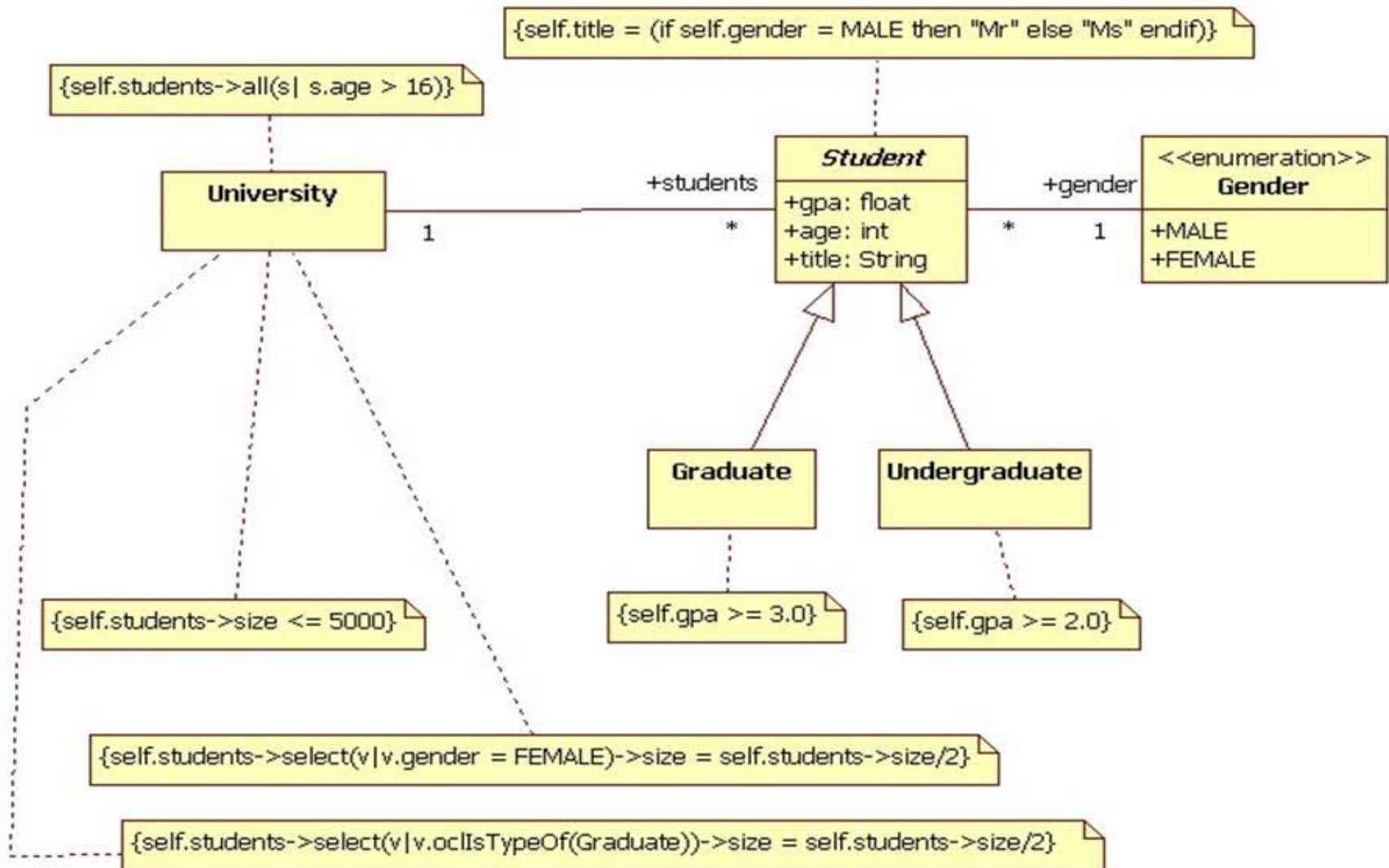
The constraint string may be placed in a **note symbol** (same as used for **COMMENT**) and attached to each of the symbols for the constrained elements by a dashed line.



- *Category association : Account owner is either Person or Corporation, **{xor}** is predefined UML constraint.*



We can represent constraints in a class Diagram as notes surrounded by curly braces:
Object class language(OCL)



Proposed Constraints Table for class diagram

CLASS NAME	CONSTRAINTS
UNIVERSITY	1.Class Constraints 2.Attribute constraints 3.Operation constraints 4. Association Constraints 5. Security Constraints 6. Nonfunctional constraints
STUDENT	
GRADUATE	
UNDERGRADUATE	

Example of a static UML Model Problem LOYALTY PROGRAM story

- A company handles loyalty programs for companies that offer their customers various kinds of bonuses. Often, the extras take the form of bonus points or air miles, but other bonuses are possible. Anything a company is willing to offer can be a service rendered in a loyalty program.
- Every customer can enter the loyalty program by obtaining a membership card .The objects of class Customer represent the persons who have entered the program.
- A membership card is issued to one person, but can be used for an entire family or business. Loyalty programs can allow customers to save bonus points, with which they can buy services from program partners. A loyalty account is issued per customer membership in a loyalty program

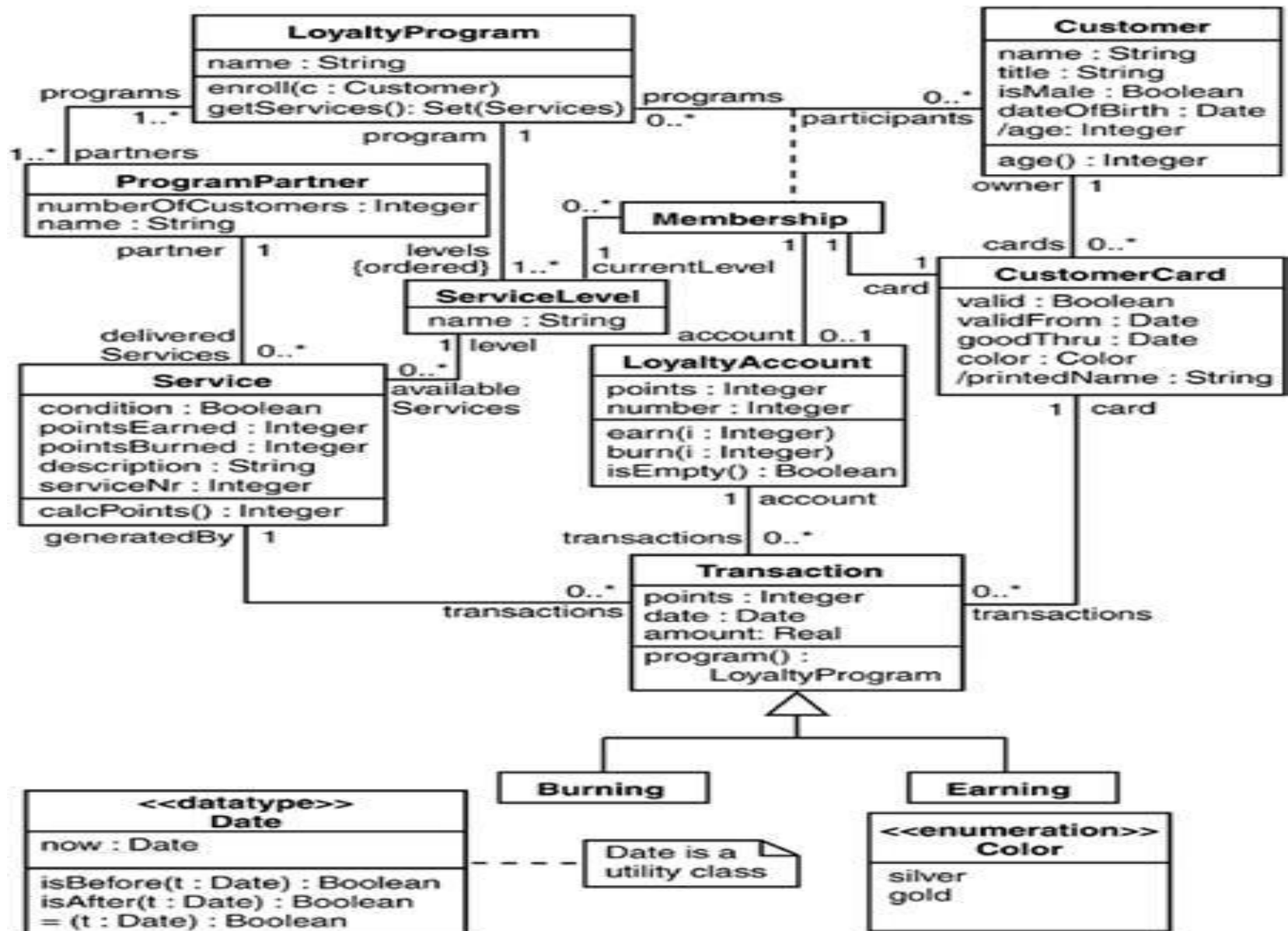
-Transactions on loyalty accounts involve various services provided by the program partners and are performed per single card. There are two kinds of transactions Earning and burning. Membership durations determine various levels of services.

Earn and Burn programs reward spend and repeat buying by incentivising customers to gather points rather than provide instantly redeemable offers.

Case study static UML Model LOYALTY PROGRAM

- A company handles loyalty programs (class **LoyaltyProgram**) for companies (class **ProgramPartner**) that offer their customers various kinds of bonuses. Often, the extras take the form of bonus points or air miles, but other bonuses are possible. Anything a company is willing to offer can be a service (class **Service**) rendered in a loyalty program.
- Every customer can enter the loyalty program by obtaining a membership card (class **CustomerCard**). The objects of class Customer represent the persons who have entered the program.
- A membership card is issued to one person, but can be used for an entire family or business. Loyalty programs can allow customers to save bonus points (class **loyaltyAccount**) , with which they can buy services from program partners. A loyalty account is issued per customer membership in a loyalty program (association class **Membership**). Transactions (class **Transaction**) on loyalty accounts involve various services provided by the program partners and are performed per single card. There are two kinds of transactions **Earning and burning**. Membership durations determine various levels of services (class **serviceLevel**).
- Construct class diagram with constraints

Specify Access Specifier in the class diagrams



REFERENCES

- [1] , , pp , j g g Warmer, J., Kleppe, A.: The Object Constraint Language. Precise Modeling with UML. Addison-Wesley, 1999
- [2] Warmer, J., Kleppe, A.: The Object Constraint Language Second Edition. Getting Your Models Ready For MDA. Addison-Wesley, 2003
- [3] OMG UML specification,
[www.omg.org/technology/documents/modeling spec catalo
www.omg.org/technology/documents/modeling spec catalo
g.htm#UML](http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML)
- [4] OMG UML 2.0 OCL,
[www omg org/technology/documents/formal/ocl htm
www.omg.org/technology/documents/formal/ocl.htm](http://www.omg.org/technology/documents/formal/ocl.htm)
- [5] Heinrich Hußmann: Formal Specification of Software Systems. Course, 2000, Technische Universität Dresden