OCL – The Object Constraint Language in UML

OCL website:

http://www.omg.org/uml/

Textbook: “The Objection Constraint Language: Precise Modeling with UML”, by Jos Warmer and Anneke Kleppe

This presentation includes some slides by: Yong He, Tevfik Bultan, Brian Lings, Lieber.

History

- First developed in 1995 as IBEL by IBM's Insurance division for business modelling
- IBM proposed it to the OMG's call for an object-oriented analysis and design standard. OCL was then merged into UML 1.1.
- OCL was used to define UML 1.2 itself.

Companies behind OCL

- Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, Softeam

UML Diagrams are NOT Enough!

- We need a language to help with the spec.
- We look for some “add-on” instead of a brand new language with full specification capability.
- Why not first order logic? – Not OO.
- OCL is used to specify constraints on OO systems.
- OCL is not the only one.
- But OCL is the only one that is standardized.

OCL – fills the missing gap:

- Formal specification language → implementable.
- Supports object concepts.
- “Intuitive” syntax – reminds OO programming languages.
- But – OCL is not a programming language:
  - No control flow.
  - No side-effects.

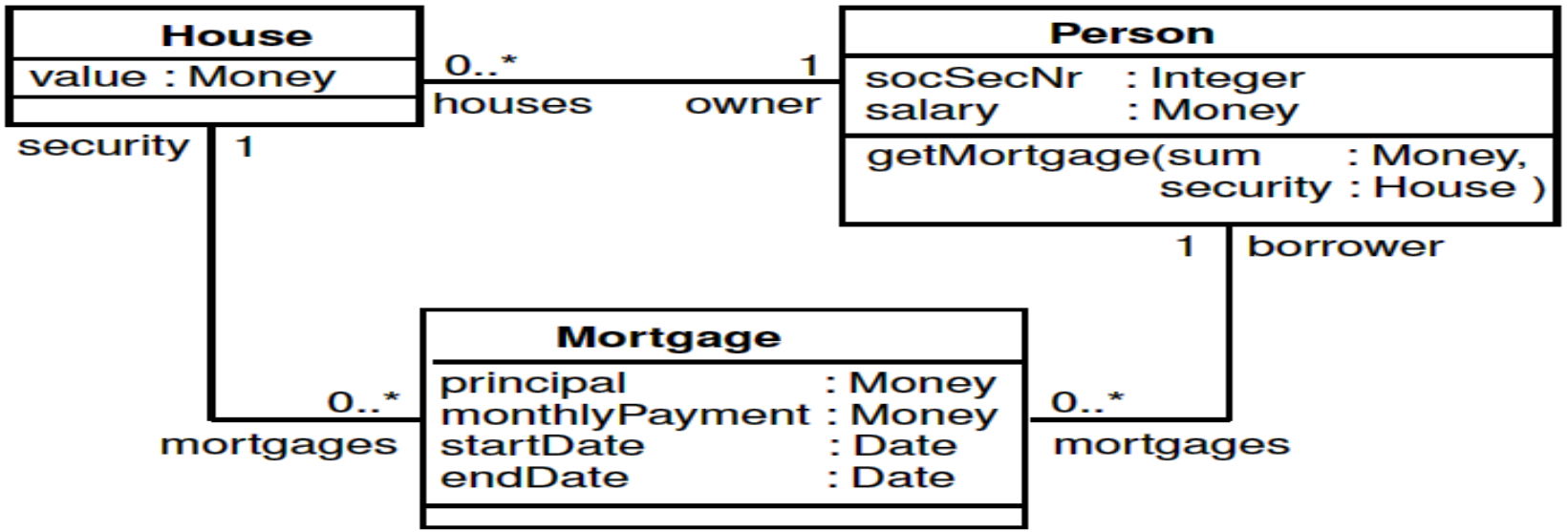
Advantages of Formal Constraints

- Better documentation
  - Constraints add information about the model elements and their relationships to the visual models used in UML
  - It is way of documenting the model
- More precision
  - OCL constraints have formal semantics, hence, can be used to reduce the ambiguity in the UML models
- Communication without misunderstanding
  - UML models are used to communicate between developers, Using OCL constraints modelers can communicate unambiguously

Where to use OCL?

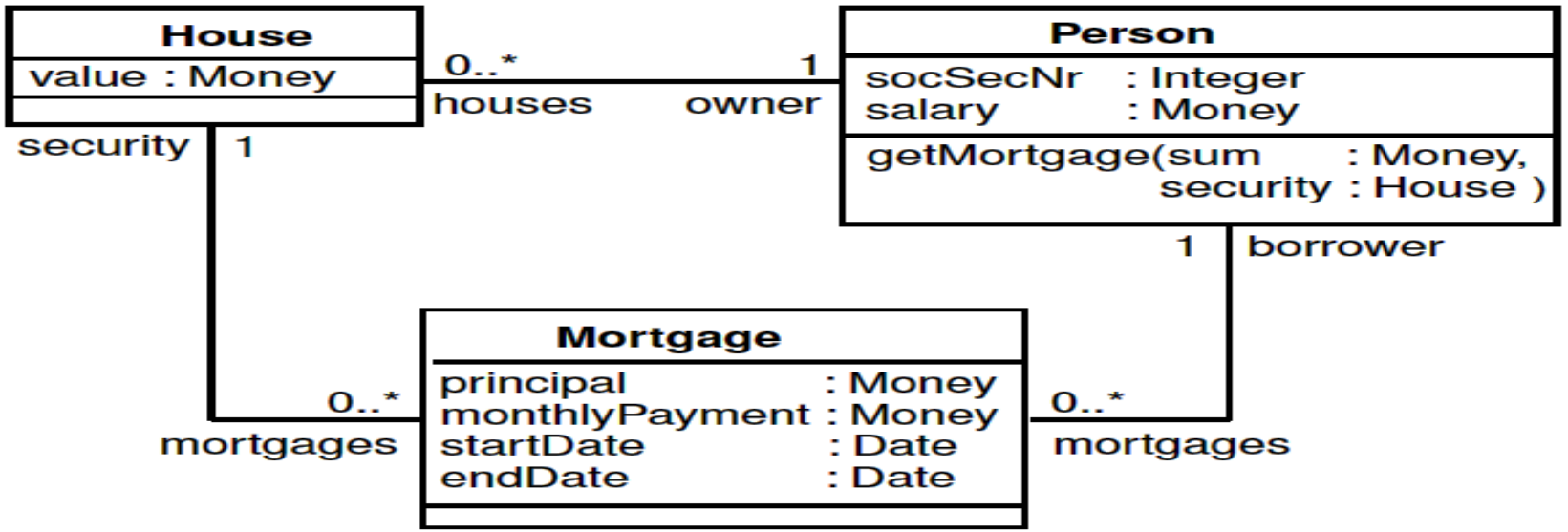
- Specify invariants for classes and types
- Specify pre- and post-conditions for methods
- As a navigation language
- To specify constraints on operations
- Test requirements and specifications

Example: A Mortgage System



1. A person may have a mortgage only on a house he/she owns.
   The start date of a mortgage is before its end date.

OCL specification of the constraints:



1. **context** *Mortgage*
   invariant: *self.security.owner = self.borrower*

   context *Mortgage*
   invariant: *security.owner = borrower*

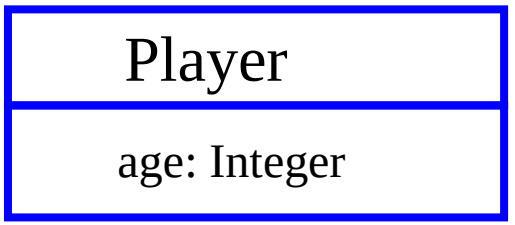
2. **context** *Mortgage*
   invariant: *self.startDate < self.endDate*

   context *Mortgage*
   invariant: *startDate < endDate*

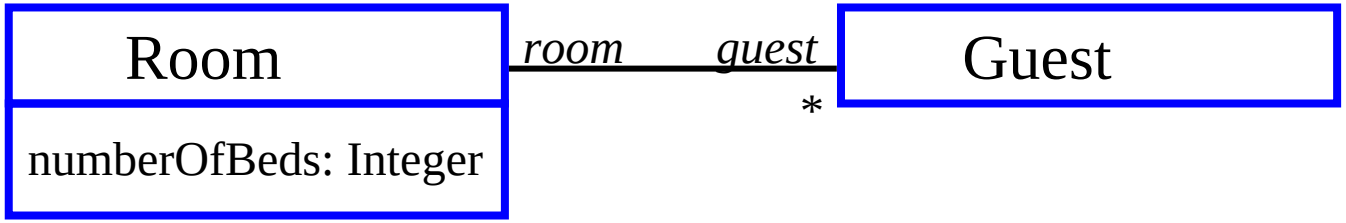
More Constraints Examples

- All players must be over 18.

context Player **invariant:**
self.age >=18



- The number of guests in each room doesn't exceed the number of beds in the room.



context Room **invariant:**
guests -> size <= numberOfBeds

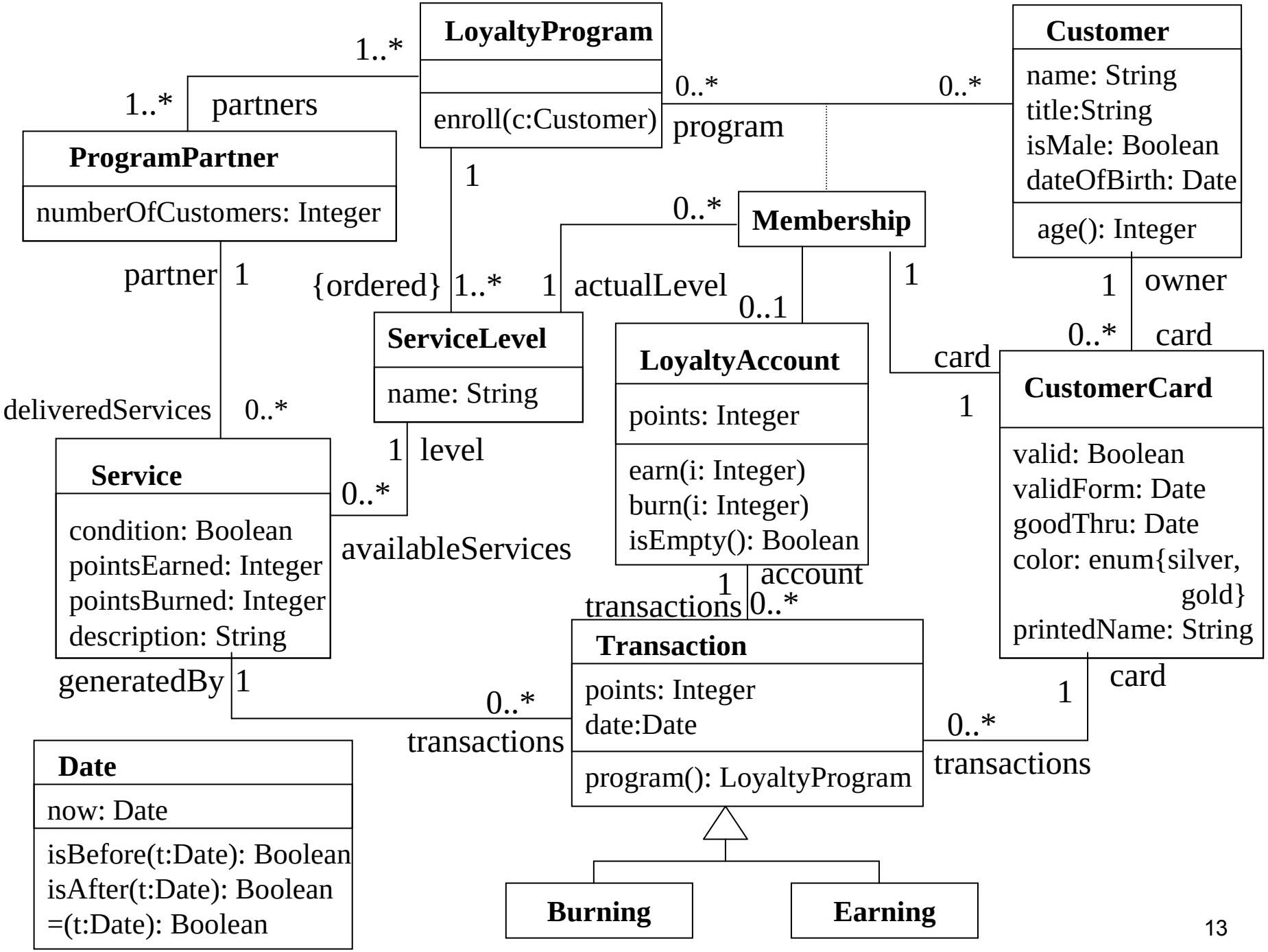
Constraints (invariants), Contexts and Self

- A constraint (invariant) is a boolean OCL expression – evaluates to true/false.
- Every constraint is bound to a specific type (class, association class, interface) in the UML model – its context.
- The context objects may be denoted within the expression using the keyword 'self'.
- The context can be specified by:
  - Context <context name>
  - A dashed note line connecting to the context figure in the UML models
- A constraint might have a name following the keyword **invariant**.

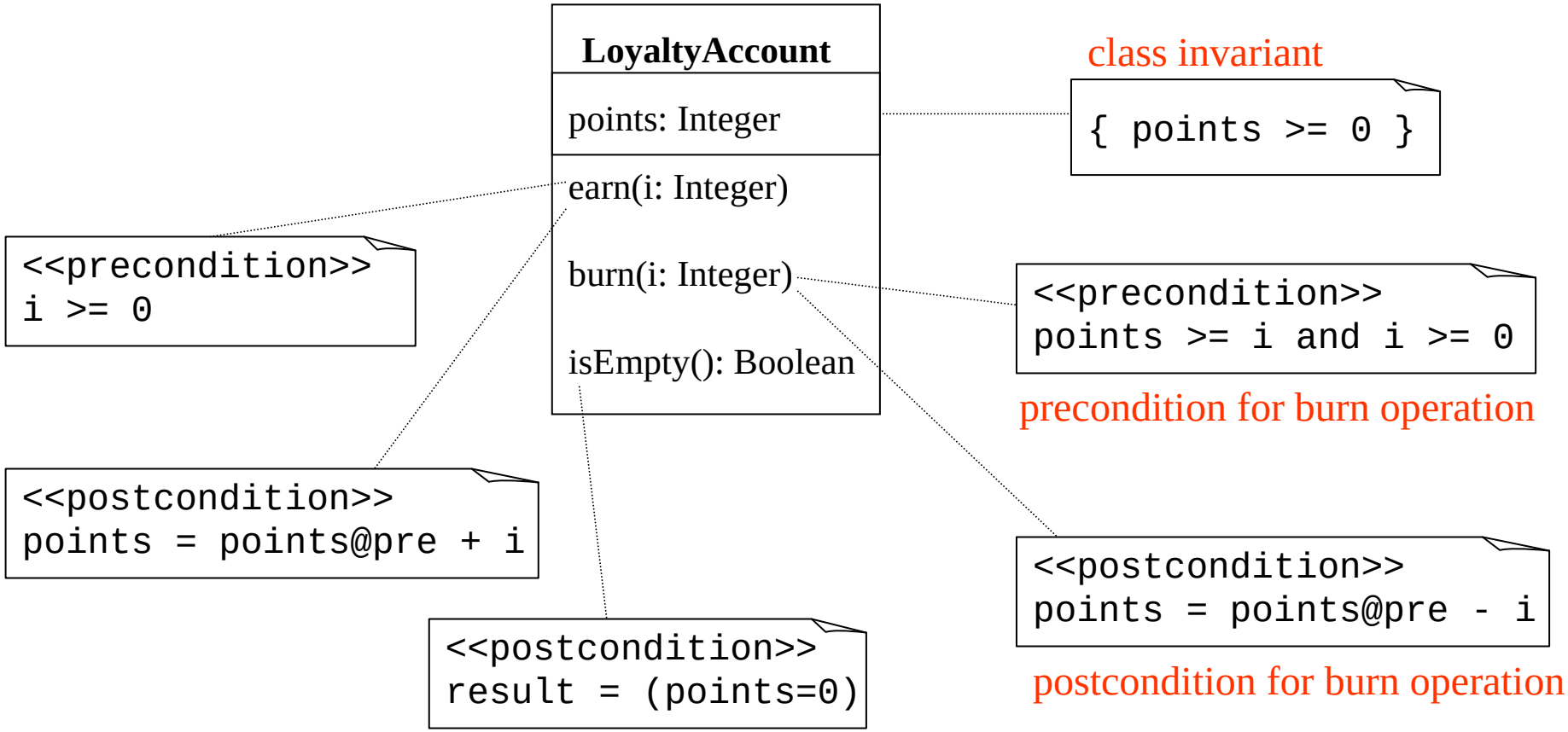
Example of a static UML Model

Problem story:

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).


LoyaltyProgram
enroll(c:Customer)
Customer
name: String
title:String
isMale: Boolean
dateOfBirth: Date
age(): Integer
ProgramPartner
numberOfCustomers: Integer
Membership
ServiceLevel
name: String
LoyaltyAccount
points: Integer
earn(i: Integer)
burn(i: Integer)
isEmpty(): Boolean
CustomerCard
valid: Boolean
validForm: Date
goodThru: Date
color: enum{silver, gold}
printedName: String
Service
condition: Boolean
pointsEarned: Integer
pointsBurned: Integer
description: String
Transaction
points: Integer
date:Date
program(): LoyaltyProgram
Date
now: Date
isBefore(t:Date): Boolean
isAfter(t:Date): Boolean
=(t:Date): Boolean
Burning
Earning
1..*
partners
1..*
0..*
program
0..*
1
0..*
1
{ordered}
1..*
1
actualLevel
0..1
1
owner
1
0..*
card
card
1
partner
1
deliveredServices
0..*
1
level
0..*
availableServices
1
account
transactions
0..*
generatedBy
1
0..*
transactions
1
card
0..*
transactions

Using OCL in Class Diagrams



Invariants on Attributes

- Invariants on attributes:

```
context Customer
invariant agerestriction: age >= 18

context CustomerCard
invariant correctDates: validFrom.isBefore(goodThru)

The type of validFrom and goodThru is Date.
isBefore(Date):Boolean is a Date operation.
```

- The class on which the invariant must be put is the invariant context.

- For the above example, this means that the expression is an invariant of the Customer class.

Invariants using Navigation over Association Ends – Roles (1)

Navigation over associations is used to refer to associated objects, starting from the context object:

```
context CustomerCard
invariant: owner.age >= 18
```

owner → a *Customer* instance.
owner.age → an *Integer*.
Note: This is not the "right" context for this constraint!

If the role name is missing – use the class name at the other end of the association, starting with a lowercase letter.
Preferred: Always give role names.

Invariants using Navigation over Association Ends – Roles (2)

```
context CustomerCard
invariant printedName:
printedName =
owner.title.concat(‘ ‘).concat(owner.name)
```

printedName → a String.
owner → a Customer instance.
owner.title → a String.
owner.name → a String.
String is a recognized OCL type.
concat is a String operation, with the signature *concat*(*String*): *String*.

Invariants using Navigation from Association Classes

Navigation from an association class can use the classes at the association class end, or the role names. The context object is the association class instance – a tuple.

"The owner of the card of a membership must be the customer in the membership":

```
context Membership
invariant correctCard: card.owner = customer
```

Invariants using Navigation through Association Classes

Navigation from a class through an association class uses the association class name to obtain all tuples of an object:

"The cards of the memberships of a customer are only the customer's cards":

```
context Customer
invariant correctCard:
cards->includesAll(Membership.card)
```

This is exactly the same as the previous constraint:

"The owner of the card of a membership must be the customer in the membership":

```
context Membership
invariant correctCard: card.owner = customer
```

The Membership correctCard constrain is better!

Invariants using Navigation through Associations with "Many" Multiplicity

Navigation over associations roles with multiplicity greater than 1 yields a Collection type. Operations on collections are accessed using an arrow ->, followed by the operation name.

"A customer card belongs only to a membership of its owner":

```
context CustomerCard
invariant correctCard:
owner.Membership->includes(membership)
```

owner → a *Customer* instance.
owner.Membership → a set of *Membership instances*.
membership → a *Membership instance*.
includes is an operation of the OCL *Collection* type.

Navigating to collections

context *Customer*
`account` produces a **set** of Accounts

context *Customer*
`account.transaction` produces a **bag** of transactions

If we want to use this as a set we have to do the following

```
account.transaction -> asSet
```

Navigation to Collections

"The partners of a loyalty program have at least one delivered service":

```
context LoyaltyProgram
invariant minServices:
partners.deliveredservices->size() >= 1
```

"The number of a customer's programs is equal to that of his/her valid cards":

```
context Customer
invariant sizesAgree:
Programs->size() = cards->select(valid=true)->size()
```

Navigation to Collections

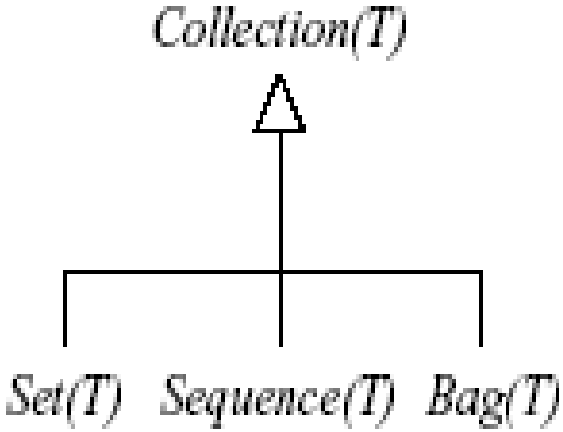
"When a loyalty program does not offer the possibility to earn or burn points, the members of the loyalty program do not have loyalty accounts. That is, the loyalty accounts associated with the Memberships must be empty":

```
context LoyaltyProgram
invariant noAccounts:
partners.deliveredservices->
  forAll(pointsEarned = 0 and pointsBurned = 0)
     implies Membership.account->isEmpty()

and, or, not, implies, xor are logical connectives.
```

The OCL Collection types

- Collection is a predefined OCL type
  - Operations are defined for collections
  - They never change the original
- Three different collections:
  - Set (no duplicates)
  - Bag (duplicates allowed)
  - Sequence (ordered Bag)
- With collections type, an OCL expression either 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->operation



Collection Operations

<collection> → size

→ isEmpty

→ notEmpty

→ sum ( )

→ count (object)

→ includes (object)

→ includesAll (collection)

Collections cont.

<collection> → select (e:T | <b.e.>)

→ reject (e:T | <b.e.>)

→ collect (e:T | <v.e.>)

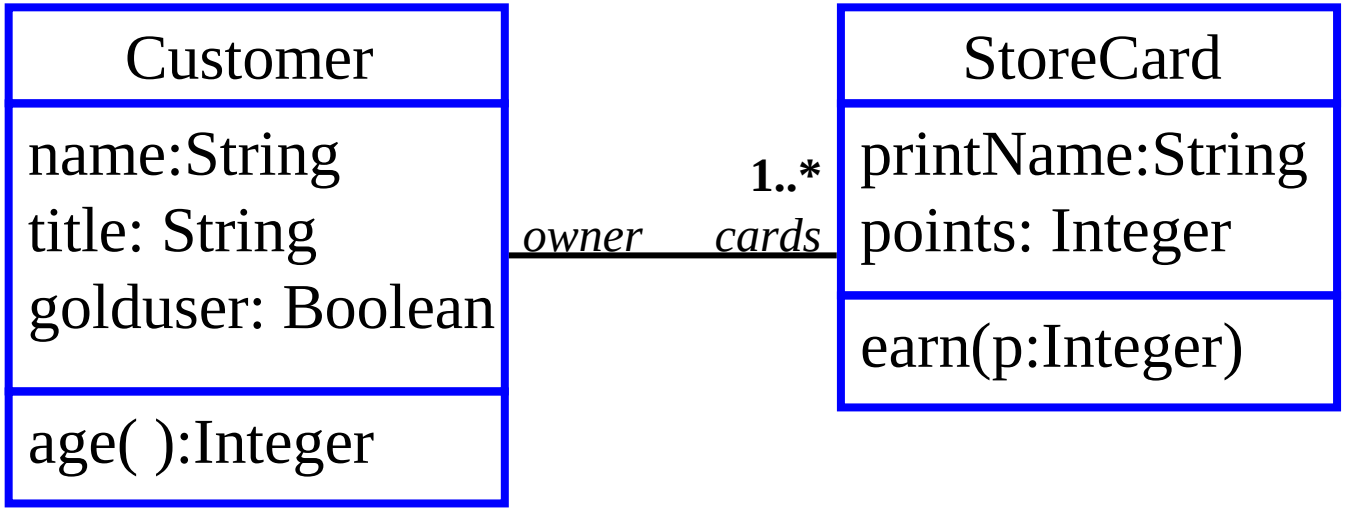
→ forAll (e:T* | <b.e.>)

→ exists (e:T | <b.e.>)

→ iterate (e:T_1; r:T_2 = <v.e.> | <v.e.>)

b.e. stands for: boolean expression
v.e. stands for: value expression

Changing the context

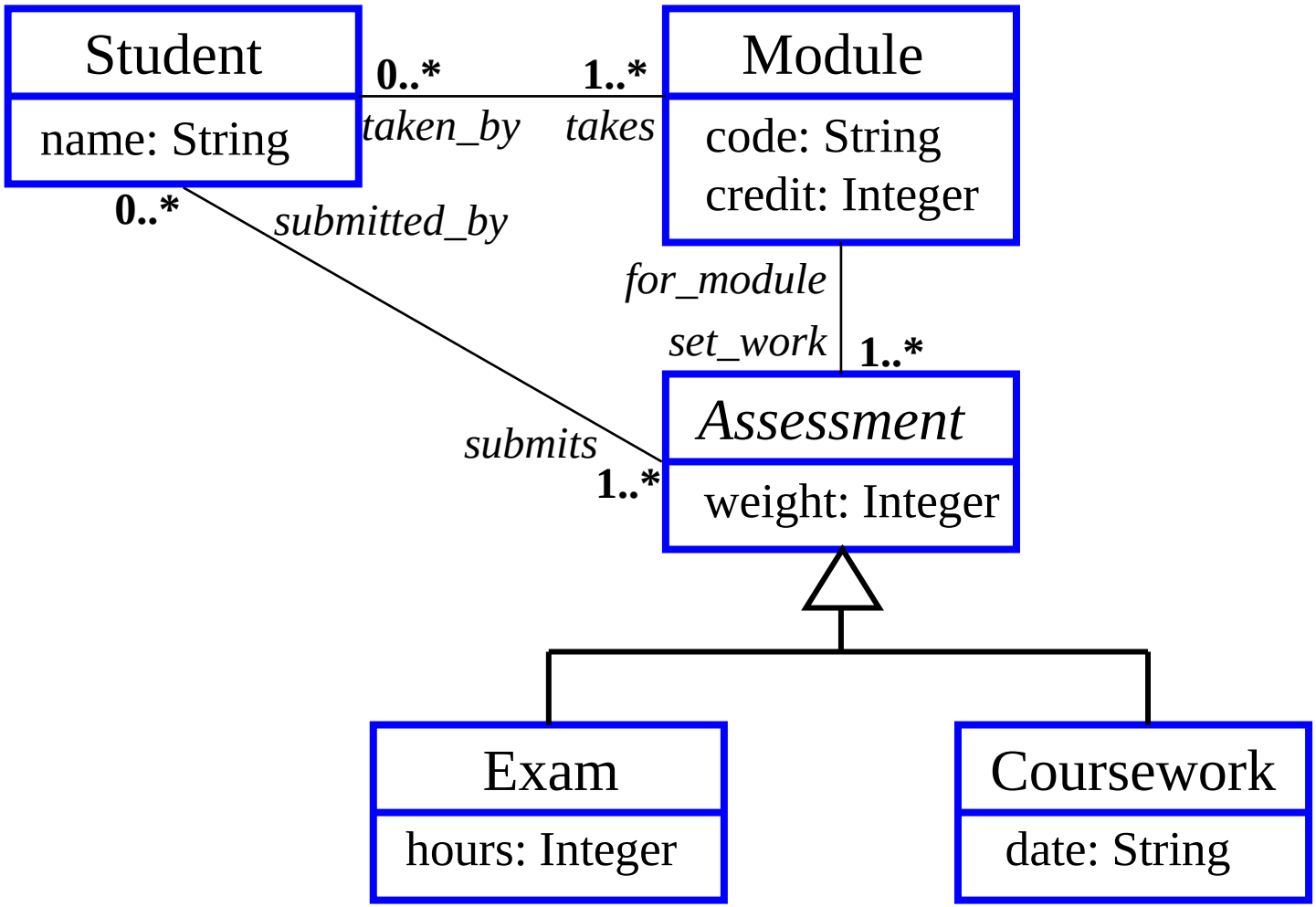


context *StoreCard*
invariant: *printName = owner.title.concat(owner.name)*

context *Customer*
cards →forAll (
printName = owner.title.concat(owner.name))

Note switch of context!

Example UML diagram



Constraints

a) Modules can be taken iff they have more than seven students registered
b) The assessments for a module must total 100%
c) Students must register for 120 credits each year
d) Students must take at least 90 credits of CS modules each year
e) All modules must have at least one assessment worth over 50%
f) Students can only have assessments for modules which they are taking

Constraint (a)

a) Modules can be taken iff they have more than seven students registered

Note: when should such a constraint be imposed?

context *Module*

invariant: *taken_by*$\rightarrow$*size* > *7*

Constraint (b)

b) The assessments for a module must total 100%

context *Module*

invariant:

set_work.weight→*sum( ) = 100*

Constraint (c)

c) Students must register for 120 credits each year

context *Student*

invariant: *takes.credit*→*sum*( ) = *120*

Constraint (d)

d) Students must take at least 90 credits of CS modules each year

context *Student*

invariant:

takes $\rightarrow$

select(*code.substring*(*1,2*) = *'CS'*).credit$\rightarrow$sum( ) >= 90

Constraint (e)

e) All modules must have at least one assessment worth over 50%

context *Module*

invariant: *set_work→exists(weight > 50)*

Constraint (f)

f) Students can only have assessments for modules which they are taking

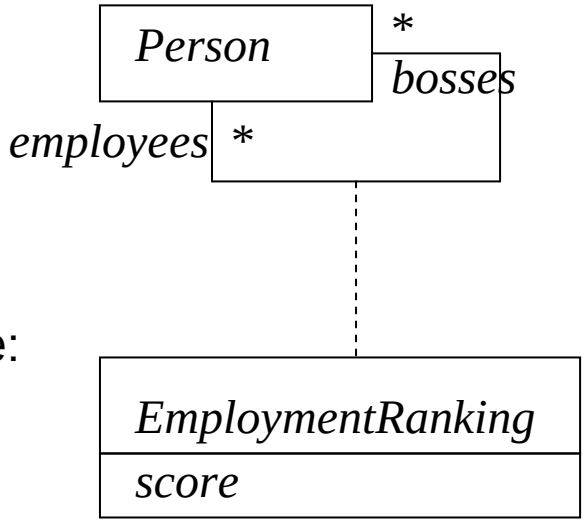
context *Student*

invariant:

takes→includesAll(submits.for_module)

Invariants using Navigation through Cyclic Association Classes

- Navigation through association classes that are cyclic requires use of roles to distinguish between association ends:
  `object.associationClass[role]`



- The accumulated score of an employee is positive:

```
context Person
invariant:
 employeeRanking[bosses].score->sum()>0
```

- Every boss must give at least one 10 score:

```
context Person
invariant:
 employeeRanking[employees]->exists(score = 10)
```

Classes and Subclasses

- Consider the following constraint

```
context LoyaltyProgram
invariant:
partners.deliveredServices.transaction.points->sum() < 10,000
```

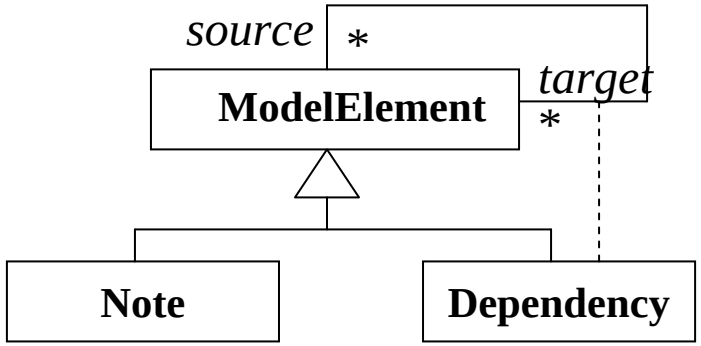
- If the constraint applies only to the *Burning* subclass, we can use the operation oclType of OCL:

```
context LoyaltyProgram
invariant:
partners.deliveredServices.transaction
   ->select(oclType = Burning).points->sum() < 10,000
```

Classes and Subclasses

"The target of a dependency is not its source"

```
context Dependency
invariant: self.source <> self
```



Is ambiguous:
Dependency is both
a *ModelElement* and an Association class.

```
context Dependency
invariant: self.oclAsType(Dependency).source <> self
invariant:
    self.oclAsType(ModelElement).source -> isEmpty()
```

OCL Constraints

- *A constraint is a restriction on one or more values of (part of) an object model/system.*
- Constraints come in different forms:
  - invariant
    - constraint on a class or type that must always hold
  - pre-condition
    - constraint that must hold before the execution of an op.
  - post-condition
    - constraint that must hold after the execution of an op.
  - guard
    - constraint on the transition from one state to another.

We study only class constraints (invariants).

OCL Expressions and Constraints

- Each OCL expression has a type.
- Every OCL expression indicates a value or object within the system.
  - *1+3* is a valid OCL expression of type *Integer*, which represents the integer value 4.
- An OCL expression is valid if it is written according to the rules (formal grammar) of OCL.
- A constraint is a valid OCL expression of type Boolean.

Combining UML and OCL

- Without OCL expressions, the model would be severely underspecified;
- Without the UML diagrams, the OCL expressions would refer to non-existing model elements,
  - there is no way in OCL to specify classes and associations.
- Only when we combine the diagrams and the constraints can we completely specify the model.

Elements of an OCL expression that is associated with a UML model

- basic types: String, Boolean, Integer, Real.
- classes from the UML model and their attributes.
- enumeration types from the UML model.
- associations from the UML model.

What is OCL Anyway?

- A textual specification language
- A expression language
- Is side-effect-free language
- Standard query language
- Is a strongly typed language
  - so expressions can be precise
- Is a formal language
- Is part of UML
- Is used to define UML
- Is Not a programming language
- The OCL is *declarative* rather than imperative
- Mathematical foundation, but no mathematical symbols
  - based on set theory and predicate logic
  - has a formal mathematical semantics

What did People Say?

- OCL is too implementation-oriented and therefore not well-suited for conceptual modelling. Moreover, it is at times unnecessarily verbose, far from natural language.
  - Alloy modelling language
- The use of operations in constraints appears to be problematic
  - An operation may go into an infinite loop or be undefined.
- Not stand alone language
- OCL is a local expression. (Mandana and Daniel)
- I would rather use plain English (Martin Fowler)

References

- The Amsterdam Manifesto on OCL
  - In Object Modeling with the OCL (LNCS2263) p115-149
- The Object Constraint Language, Precise Modeling with UML, Addison-Wesley, 1999.
- The Object Constraint Language, Precise Modeling with UML 2nd
- Response to the UML 2.0 OCL RfP (ad/2000-09-03) Revised Submission, Version 1.6 January 6, 2003
- Some Shortcomings of OCL, the Object Constraint Language of UML
  - Mandana Vaziri and Daniel Jackson,1999
- http://www.klasse.nl/english/uml/ UML CENTER
- Informal formality? The Object Constraint Language and its application in the UML metamodel
  - Anneke Kleppe, Jos Warmer, Steve Cook
- A Pratical Application of the Object Constraint Language OCL
  - Kjetil M°age
- The UML's Object Constraint Language: OCL Specifying Components, JAOO Tutorial – September 2000
  - Jos Warmer & Anneke Kleppe