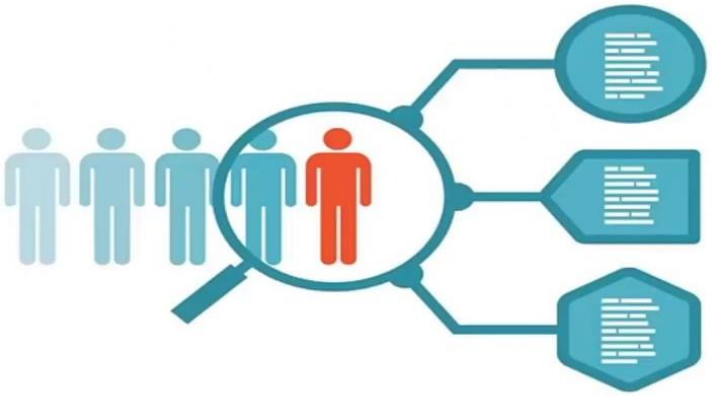
Software Engineering-2
OCL

By:

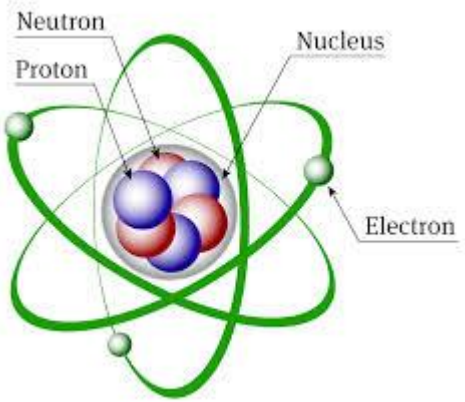
Dr. Salwa Osama

Any model

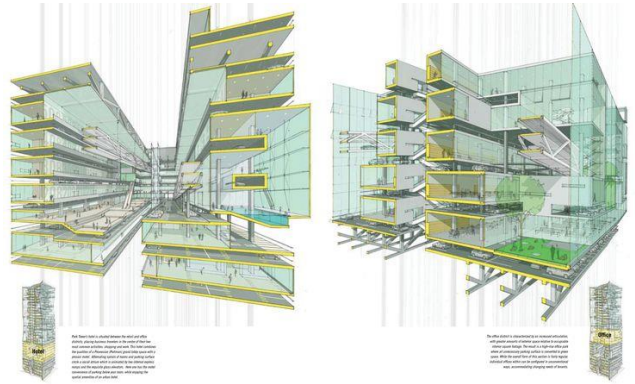
- Any model is splits into three types:
  1. Predictive model: Analyze the past for the future
  2. Descriptive model: creating the relationship in the data - grouping
  3. Perspective model: decision based on all the elements -prescribing



Predictive model



Descriptive model



Perspective model

Model Driven Engineering

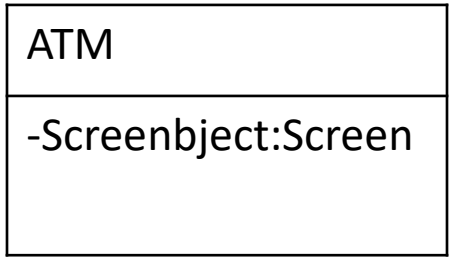
- Using Perspective models as programs
- Engineering domain-specific languages for capturing such models:
  - Precise abstract syntax.
  - Supporting graphical/textual modeling tool
- Expressing and checking validity constrains for models.
- Analyzing and simulating models
- Transformation models into:
  - Other types of models
  - Software products

Why Model Driven Engineering?

- When the abstractions provided by implementation-level technologies are not satisfactory
  - Engineers need to copy/past similar boilerplate content/code too often
  - To make change engineers need to modify several inter-related artefacts in a similar way.
- When reasoning about properties of the system is too hard/expensive at the implementation level.

Example1: Boilerplate code

```
public class ATM {

        private String Screen;

        public String getScreen() {
                // TODO - implement ATM.getScreen
                throw new UnsupportedOperationException();
        }

        /**
         *
         * @param Screen
         */
        public void setScreen(String Screen) {
                // TODO - implement ATM.setScreen
                throw new UnsupportedOperationException();
        }

}
```



Using Visual Paradigm

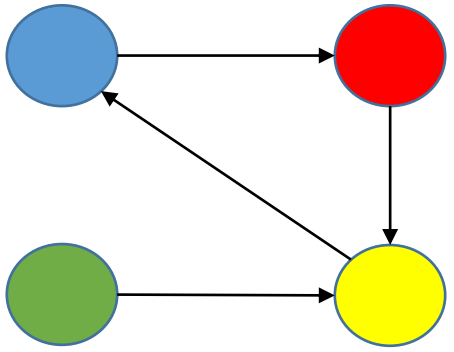
Example2: Property analysis and verification

- The following code controls the change of colors in tree lights.

```
switch(color){
        case "blue":{color="red"; break;}
        case "green":{color="yellow"; break;}
        case "yellow":{color="blue"; break;}
        case "red":{color="yellow"; break;}
    }
```

Example2: Property analysis and verification

- The following code controls the change of colors in tree lights.
- The code in previous slide can be trivially from this model.
- For larger state machine models we probably need automated reachability analysis



Modeling Languages

- Large number of off-the-shelf modeling languages
- Each language focuses on specific class of domains, problems and systems
  - UML (object oriented systems)
  - Simulink (control systems)
  - Archimate (enterprise architecture)
  - BPMN (business modeling)
  - ER (rational databases)

Domain-Specific Languages

- Often models are useful for the problem at hand but existing language lack appropriate abstraction
- Example; Organizing conferences

Conference Organization

- A conference runs over a number of days
- On every day, there are several talks organized in (potentially parallel) tracks.
- There are breaks between tracks (e.g. for lunch, coffee etc.)
- Each talk can be delivered by one or more speakers
- Each talk have a pre-defined duration

Artefacts Involved

Booklet

Website

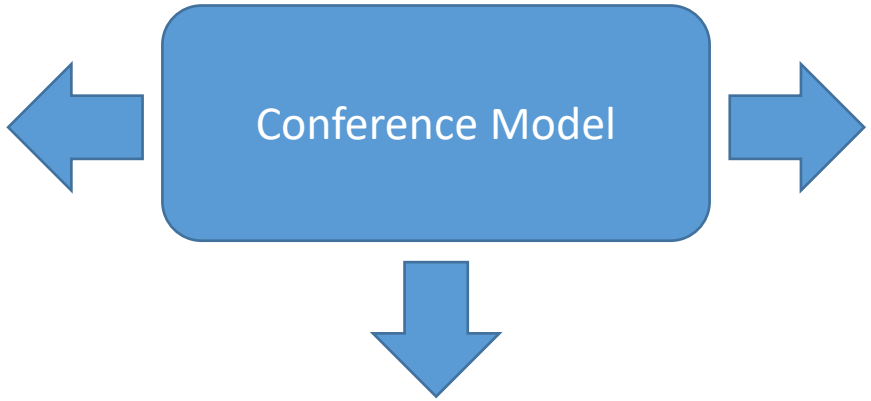
Track Posters

Challenges

- Consistency/maintainability
  - Same content appears in different artefacts
- Correctness
  - Parallel tracks must be located in different rooms
  - The total duration of the talks of track must not exceed the duration of the tracks
  - Breaks must not overlap with tracks.

Domain-specific models to the rescue

Conference modeling language



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

- OCL is The Object Constraint Language in UML
- First developed in 1995 as IBEL by IBM's Insurance division for business modelling.
- 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

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

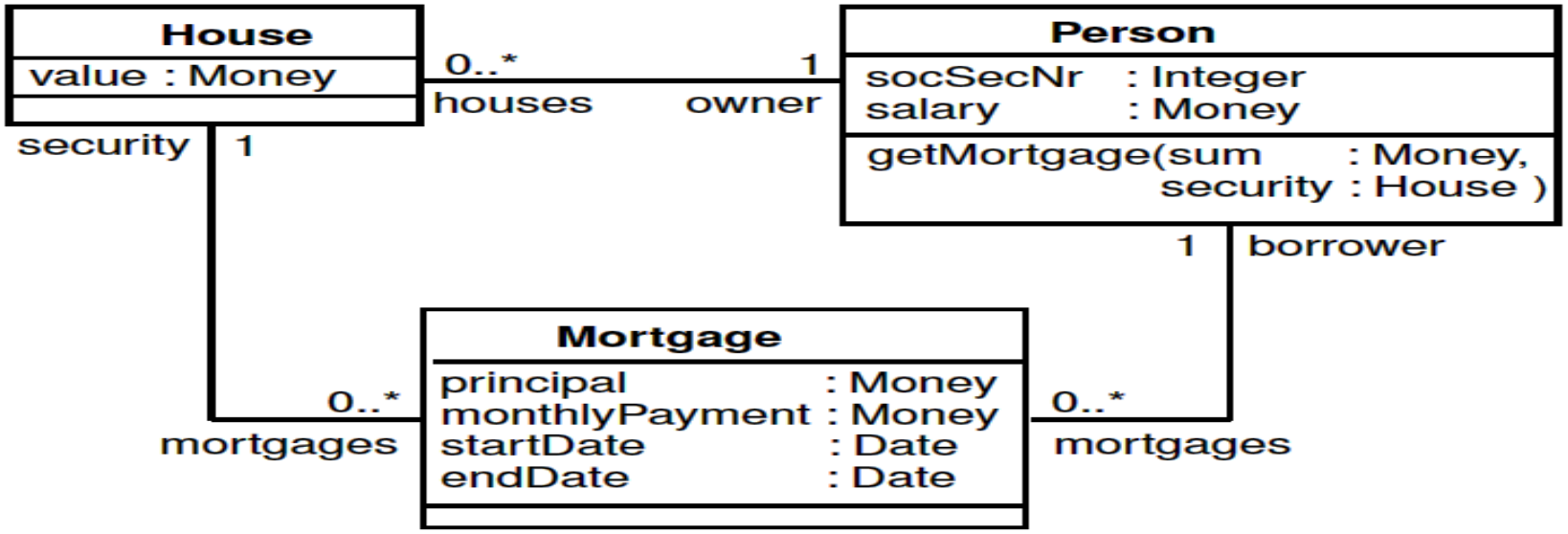
Advantages of Formal Constraints

- 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?

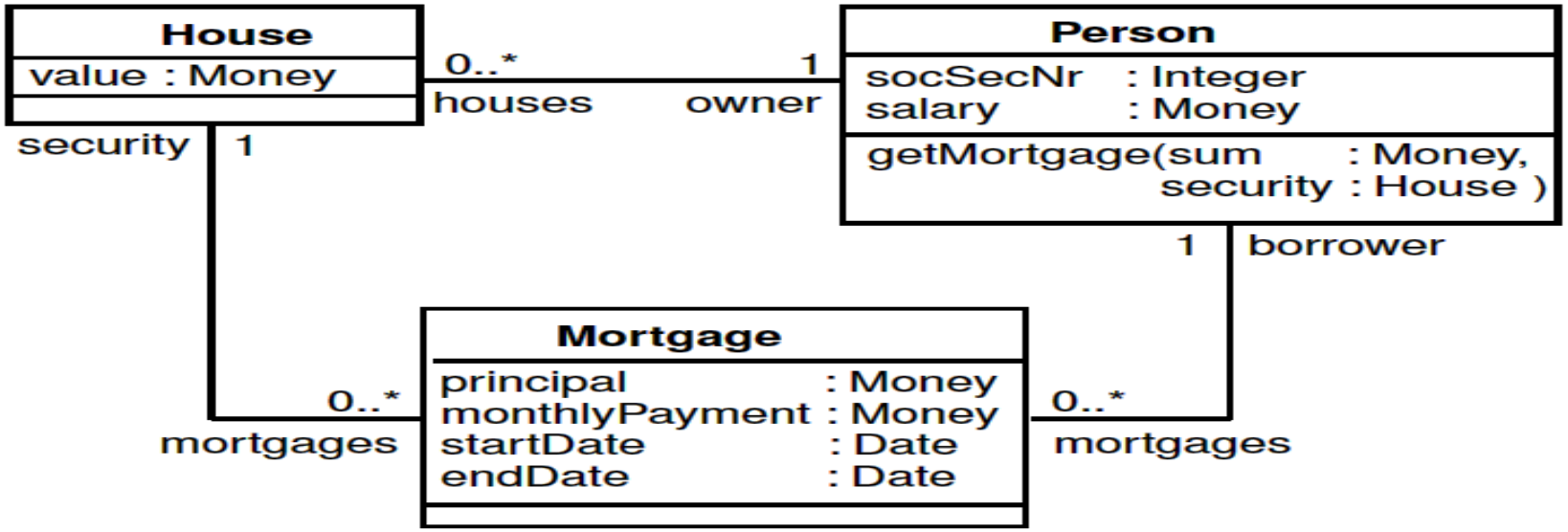
- Specify invariants for classes and types
- Specify pre- and post-conditions for methods
- As a navigation language
- To specify constraints on operations
- Test requirements and specifications

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*

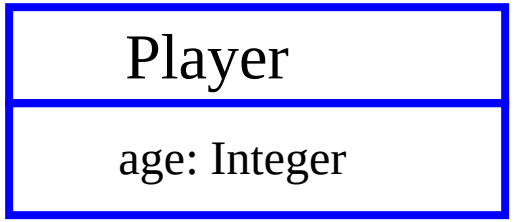
2. **context** *Mortgage*
   invariant: *self.startDate < self.endDate*

context *Mortgage*
invariant: *startDate < endDate*

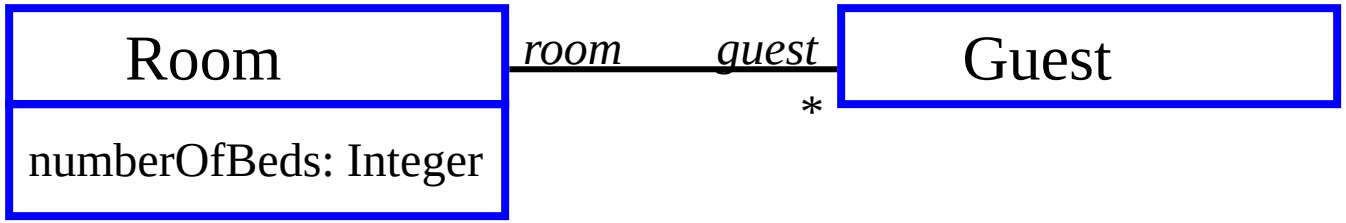
More Constraints Examples

- 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

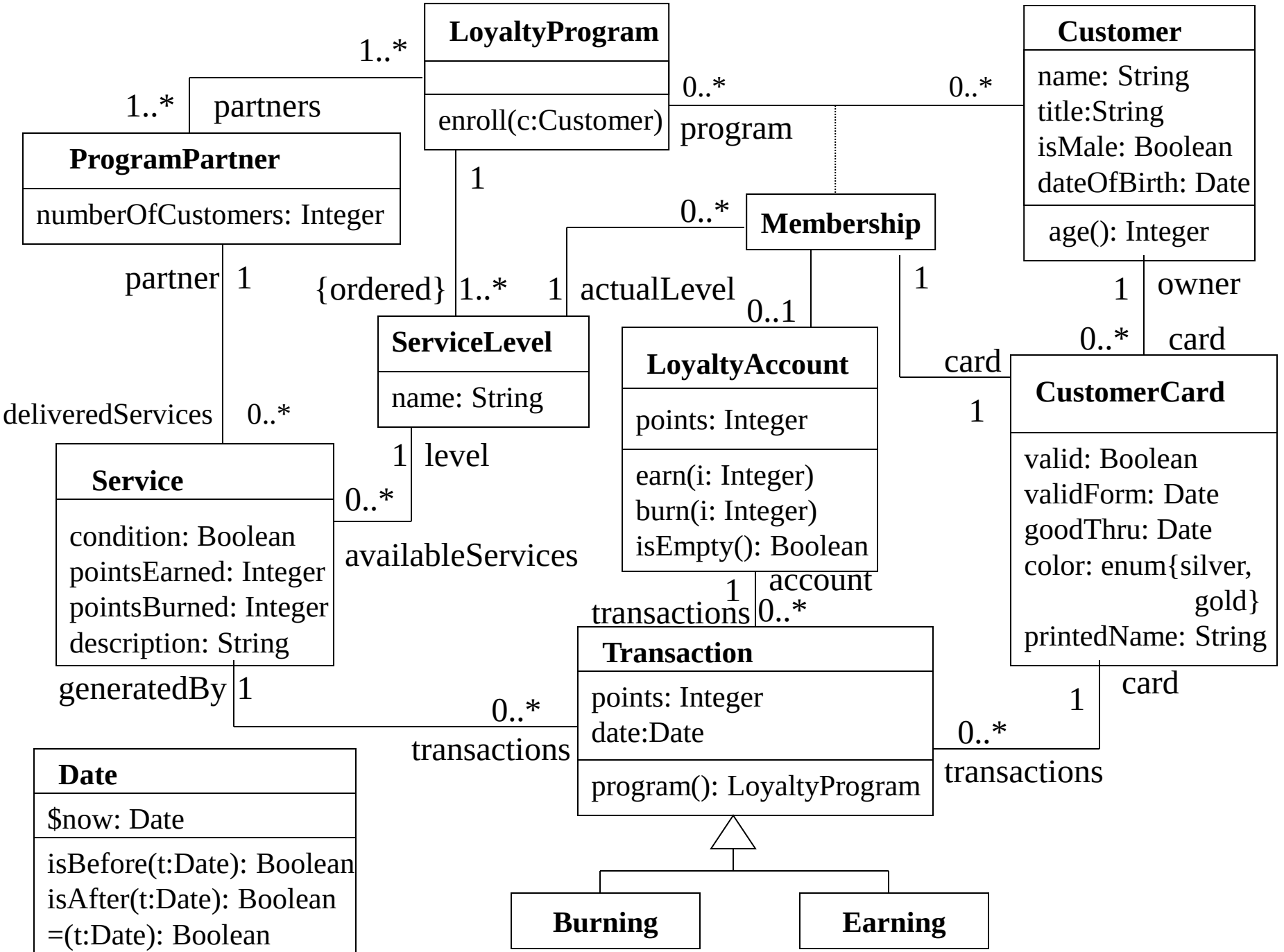
Constraints (invariants), Contexts and Self

- 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**.

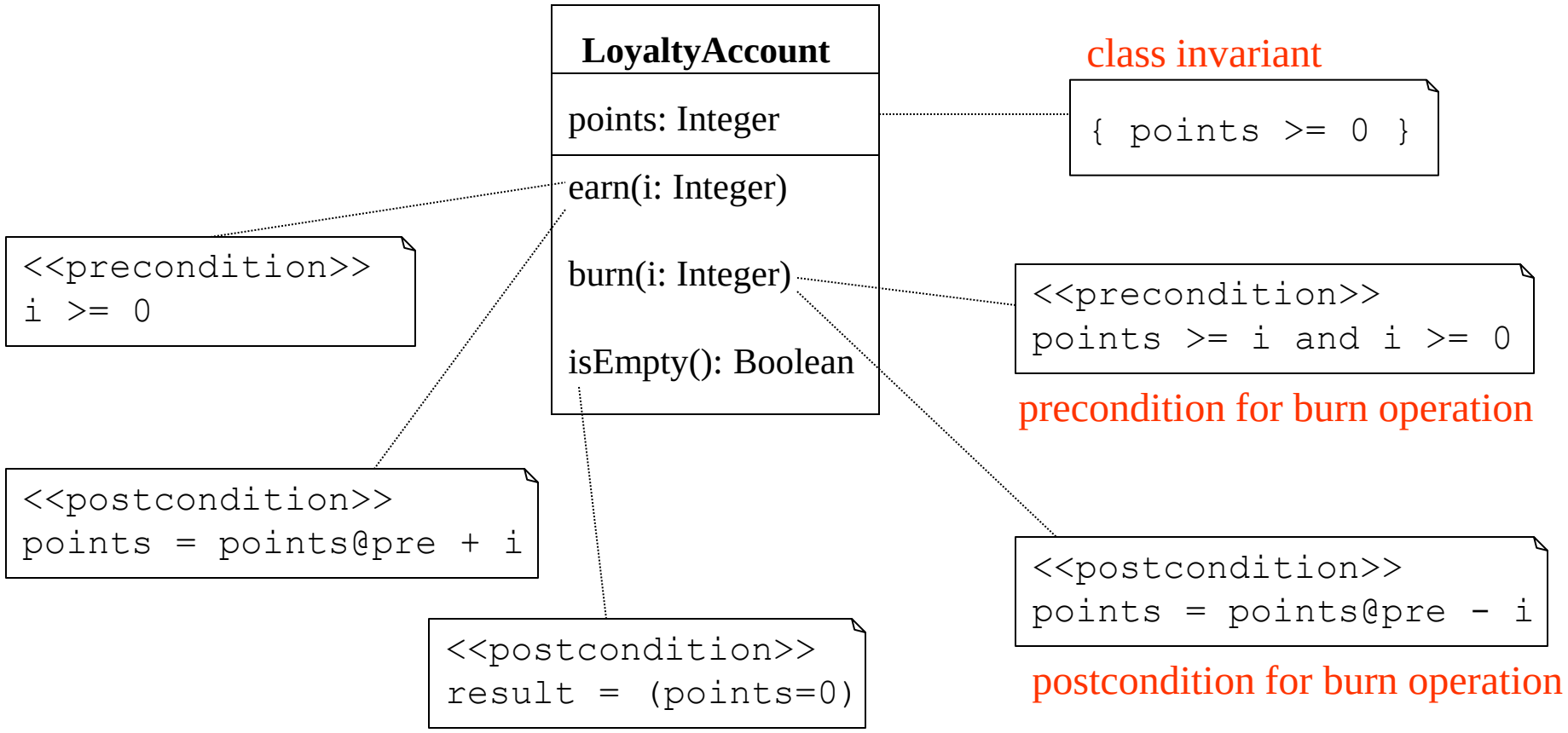
Example of a static UML Model

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
owner
1
0..*
card
card
1
{ordered}
1..*
1
actualLevel
0..1
partner
1
deliveredServices
0..*
1
level
0..*
availableServices
1
account
transactions
0..*
generatedBy
1
0..*
transactions
0..*
1
card
transactions

Using OCL in Class Diagrams



Invariants on Attributes

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

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

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

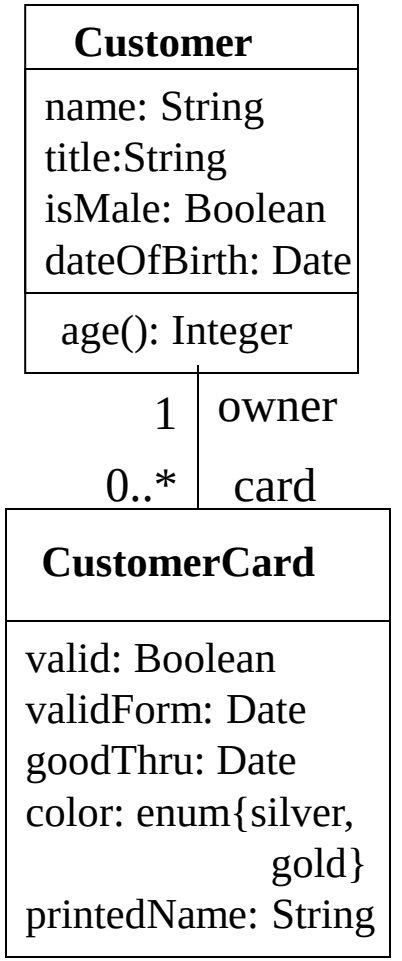
```
context CustomerCard
invariant correctDates: validFrom.isBefore(goodThru)

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

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
```

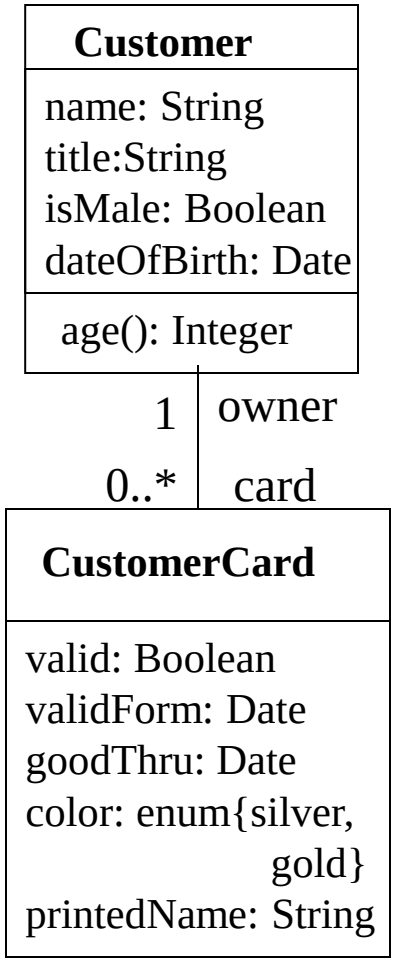
owner → a *Customer* instance.
owner.age → an *Integer*.



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 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)
```

a set of *Membership instances*.
membership → *owner* → a *Customer* instance.
owner.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

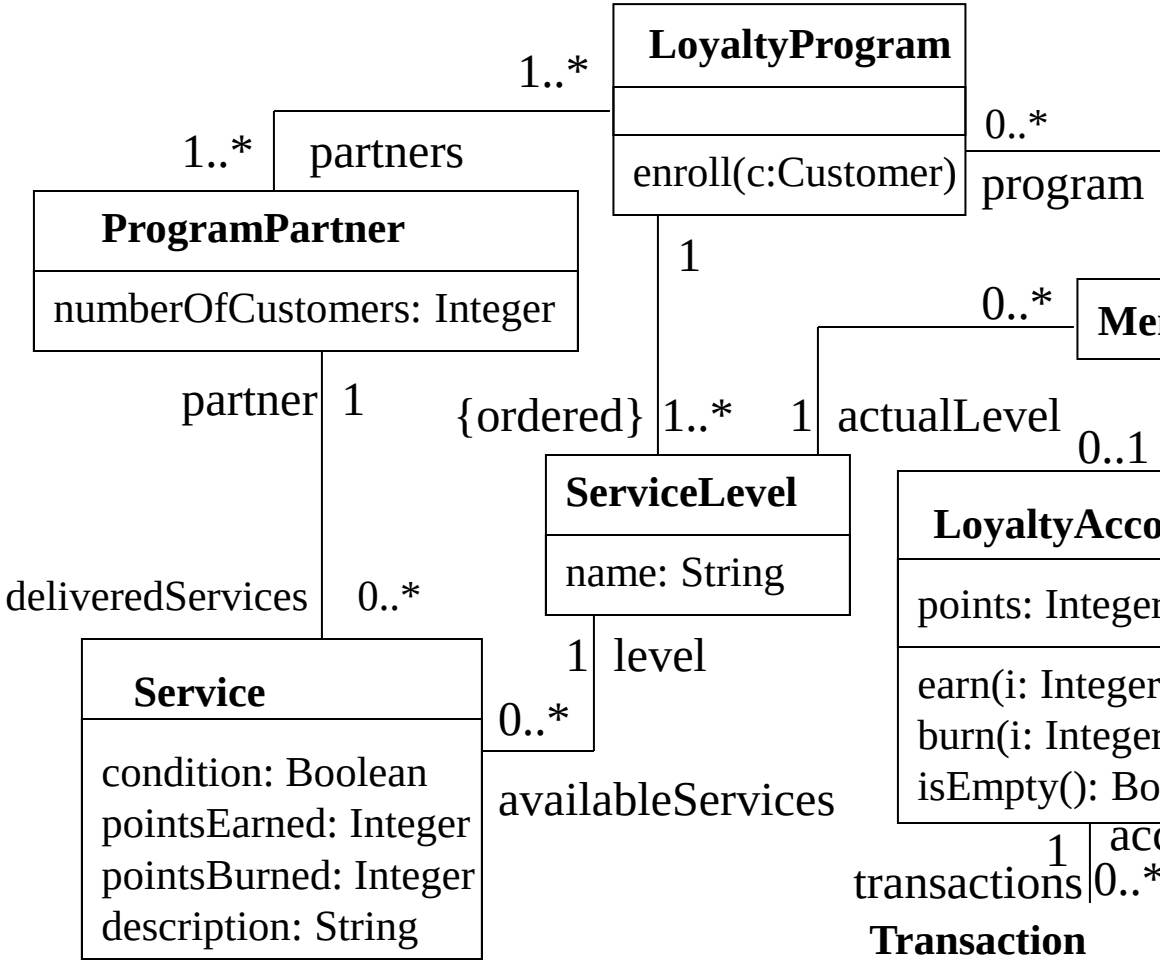
If we want to use this as a set we have to do the following

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

Navigation to Collections

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

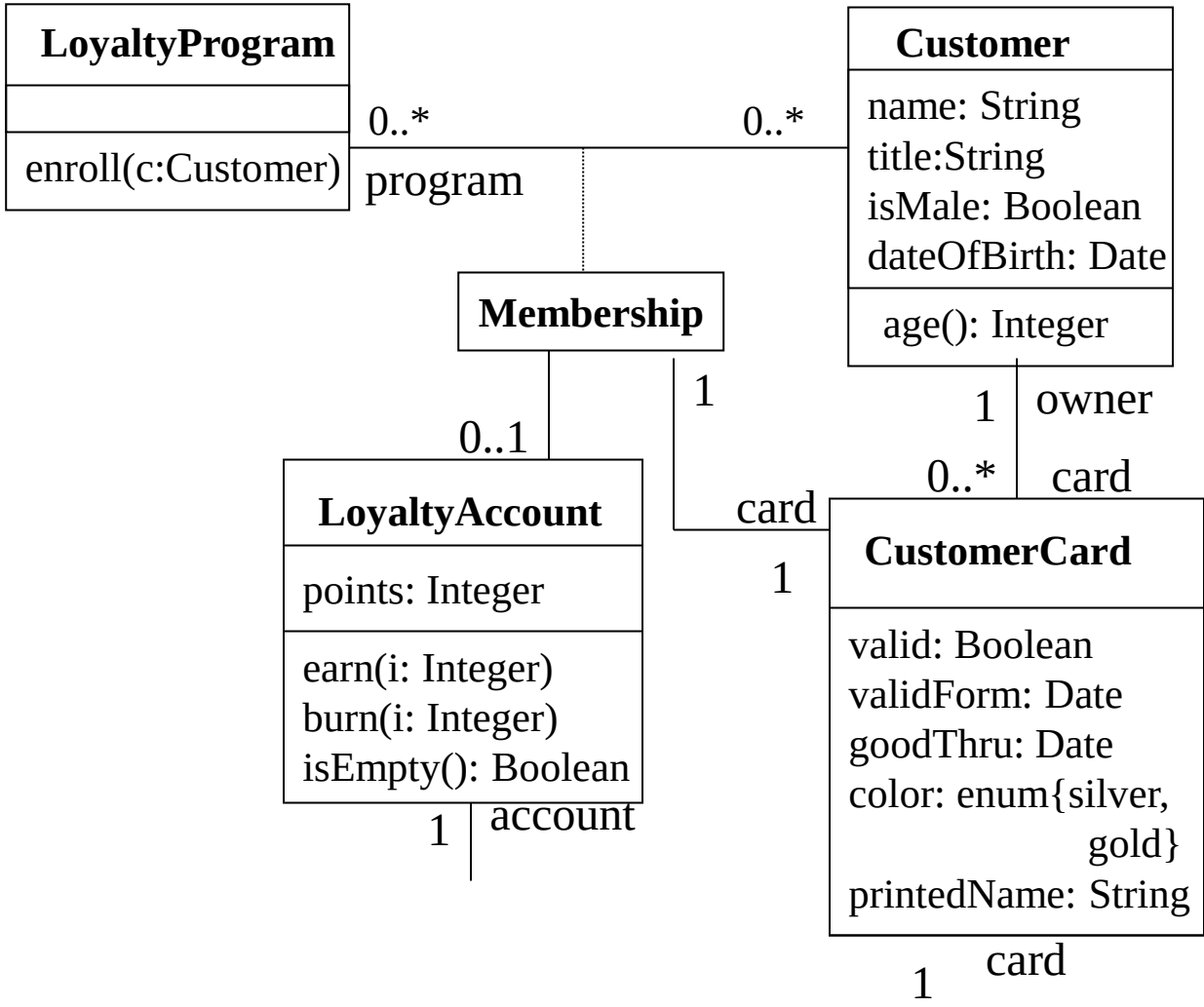
```
context LoyaltyProgram
invariant minServices:
partners.deliveredservices
  ->size() >= 1
```



Navigation to Collections

"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()
```

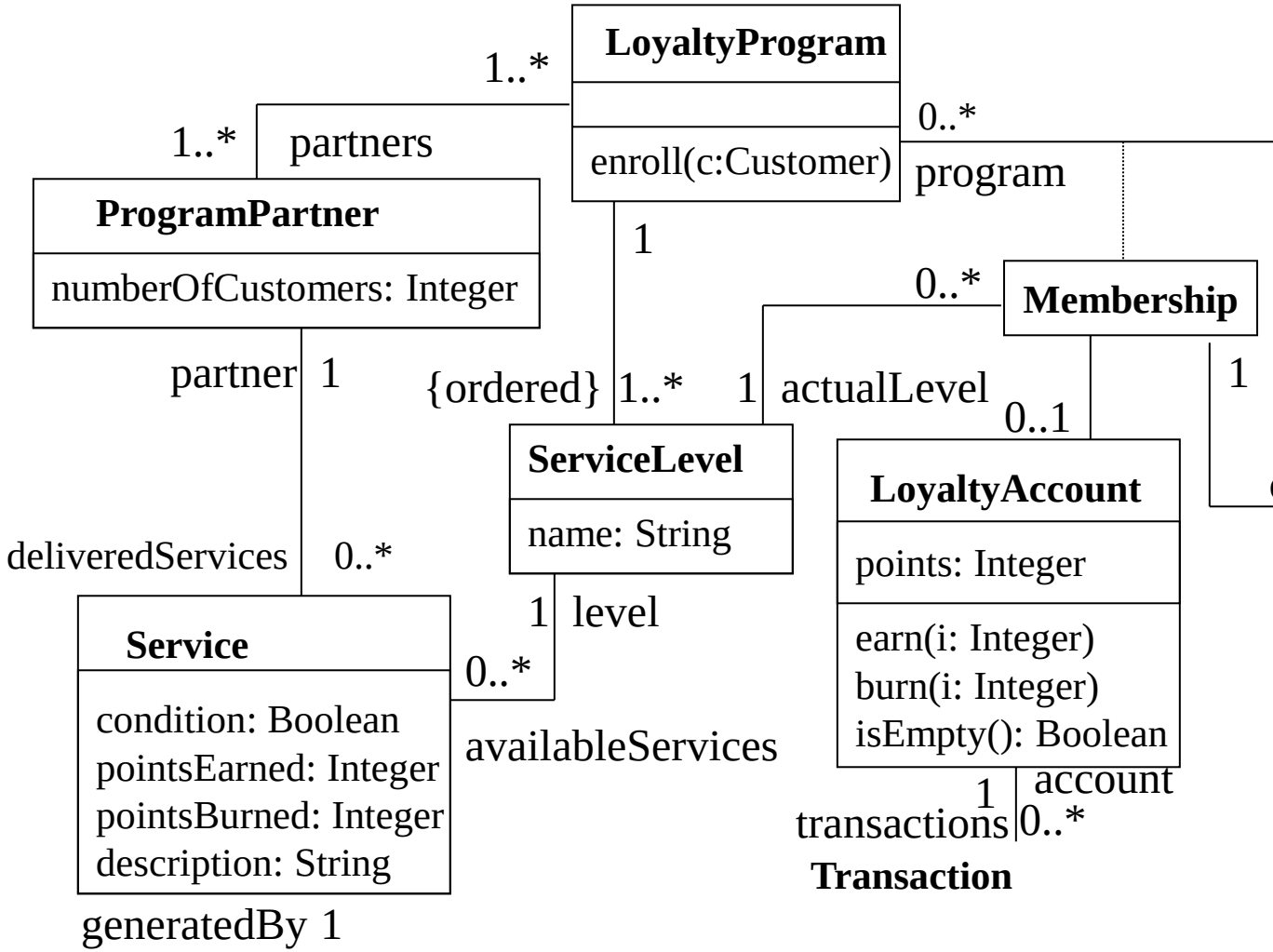


Navigation to Collections

"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":

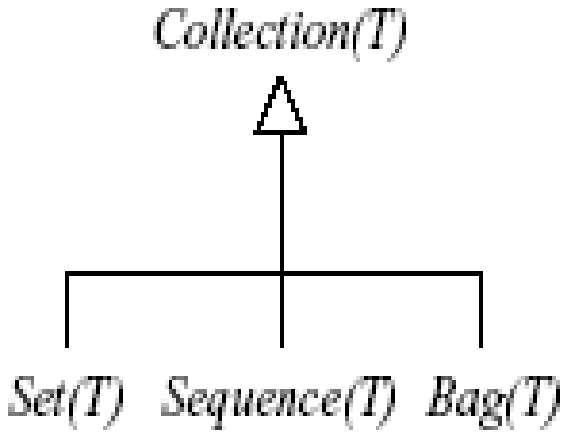
```
context LoyaltyProgram
invariant noAccounts:
partners.deliveredservices->
   forAll(pointsEarned = 0 and
   pointsBurned = 0)
      implies
   Membership.account-
   >isEmpty()
```

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



The OCL Collection types

- Collection is a predefined OCL type
  - Operations are defined for collections
- 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.>)

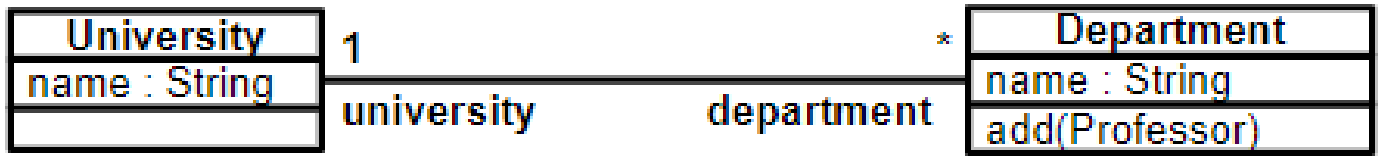
→ forAll (e:T* | <b.e.>)

→ exists (e:T | <b.e.>)

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

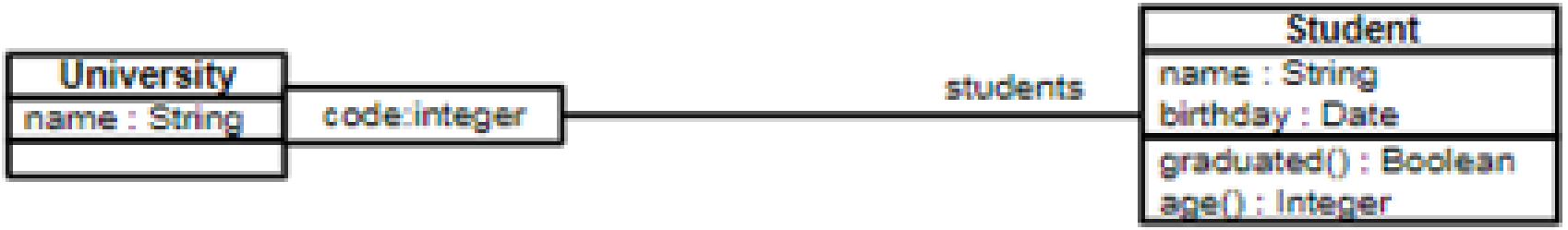
b.e. stands for: boolean expression
v.e. stands for: value expression

Examples




```
context Department
    -- A department's university should not be null.
    inv: not self.university.oclIsUndefined()

context University
    -- A university must have at least one department
    inv: self.department->notEmpty()
```

Examples




```
context University
    -- The name of student 8764423 must be 'Martin'.
    inv: self.students[8764423].name = 'Martin'
```

```
context University
    -- There is at least one student named 'Martin':
    inv: self.students->exists(each | each.name = 'Martin')
```

Example

- The collection of all the employees older than 50 years is not empty

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

self.employee->select(p | p.age > 50)->notEmpty()
```

- As an example, specify that the collection of all the employees who are not married is empty:

```
self.employee->reject( isMarried )->isEmpty()
```

Examples

- *specify the collection of birthDates for all employees in the context of a company*

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

- *An important issue here is that when the source collection is a Set the resulting collection is not a Set but a Bag.*

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

Examples

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

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.*

Examples

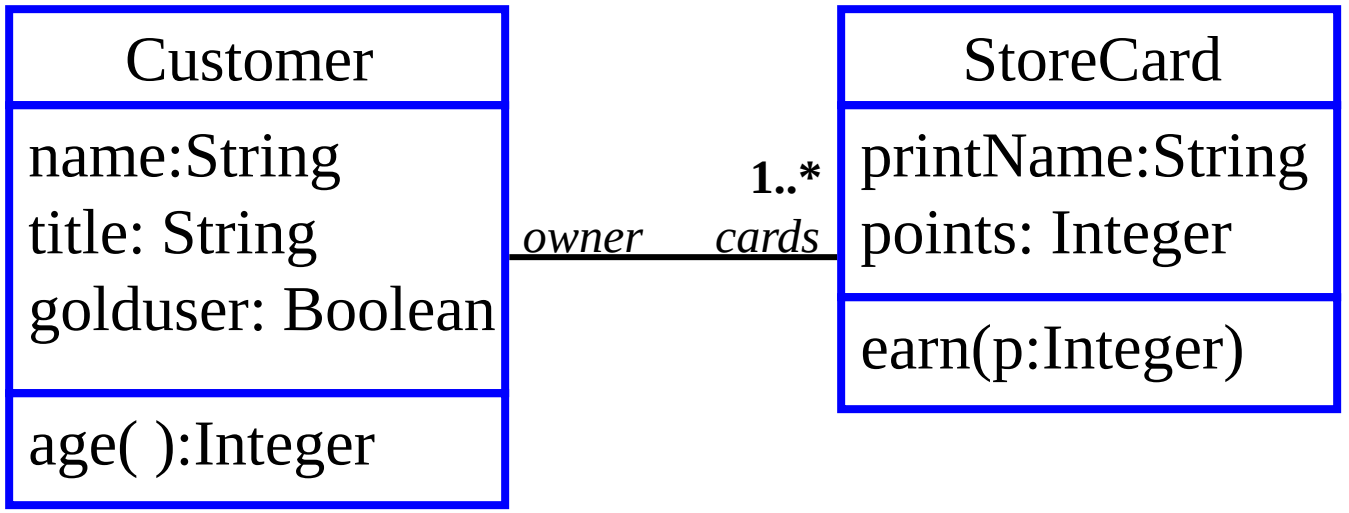
context Company inv:

self.employee->exists(forename = 'Jack’)

self.employee->exists(p | p.forename = 'Jack’)

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

Changing the context

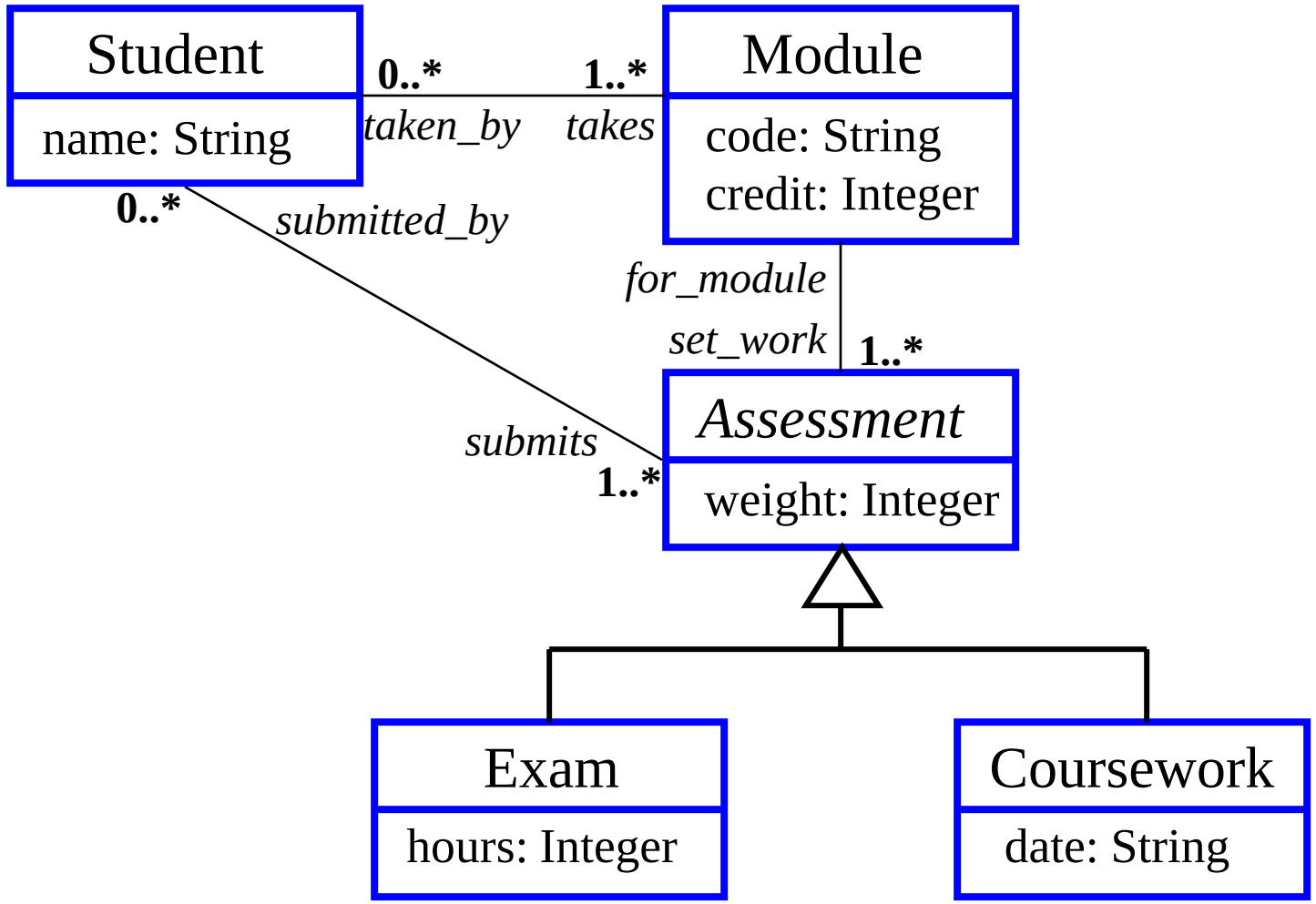


context *StoreCard*
invariant: *printName = owner.title.concat(owner.name)*

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


Let's PLAY Together

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

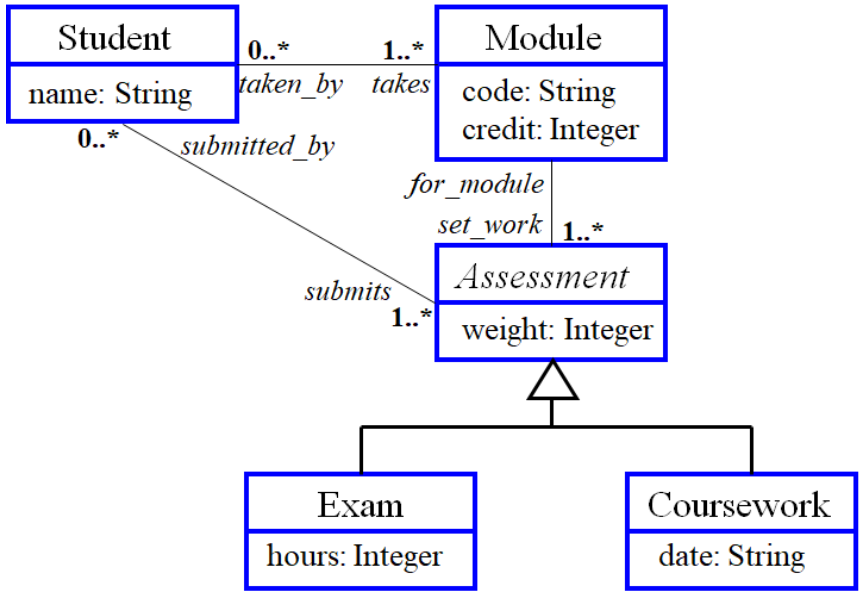
e) All modules must have at least one assessment worth over 50%

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

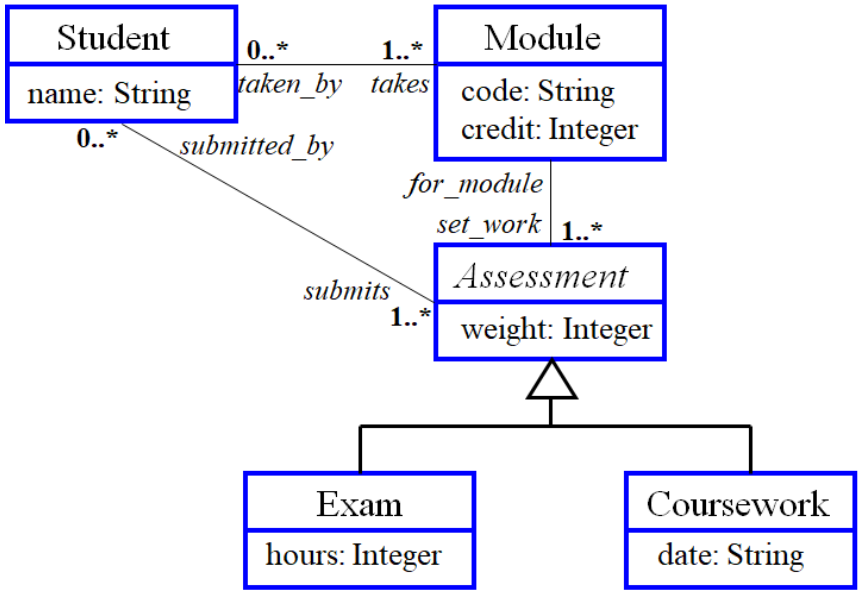
context *Module*
invariant: *taken_by→size > 7*



Constraint (b)

b) The assessments for a module must total 100%

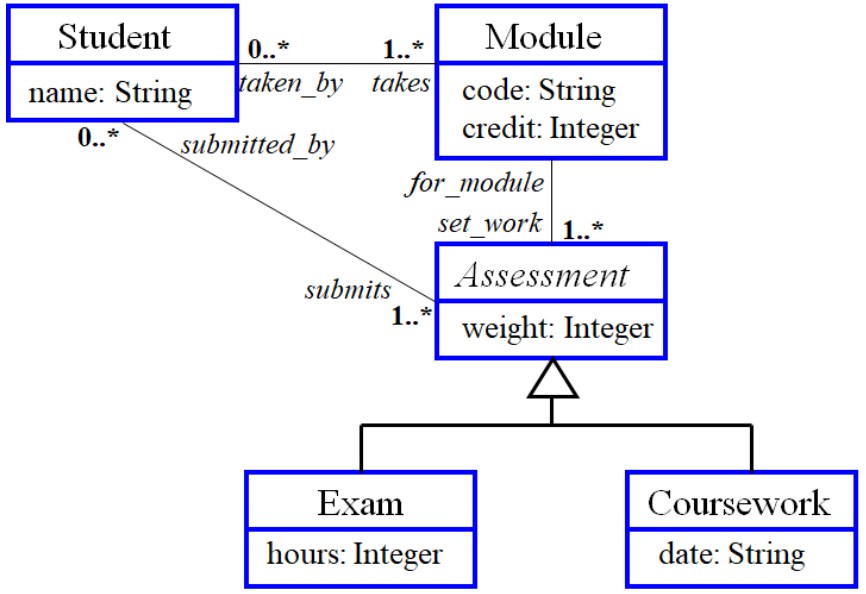
context *Module*
invariant:
set_work.weight→sum( ) = 100



Constraint (c)

c) Students must register for 120 credits each year

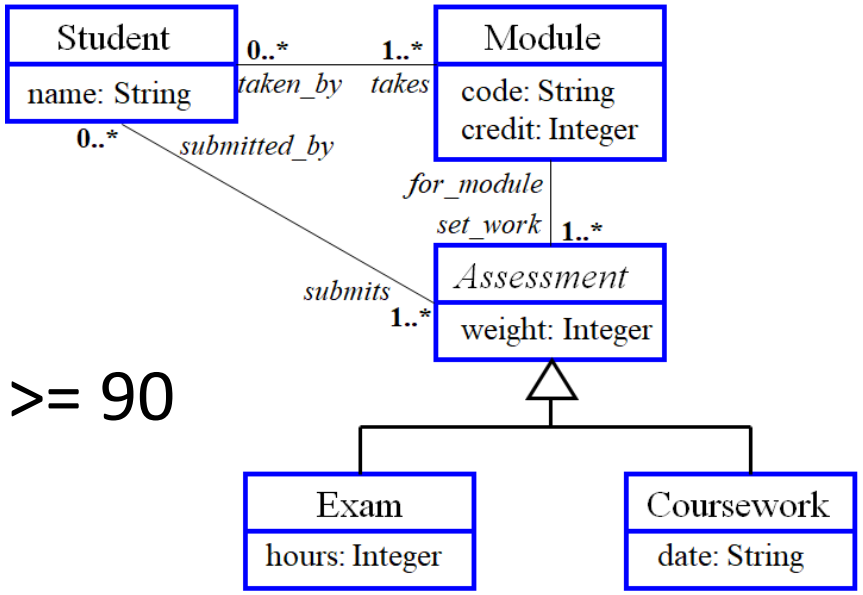
context *Student*
invariant: *takes.credit→sum( ) = 120*



Constraint (d)

d) Students must take at least 90 credits of CS modules each year, code start with "CS"

Select
Sum



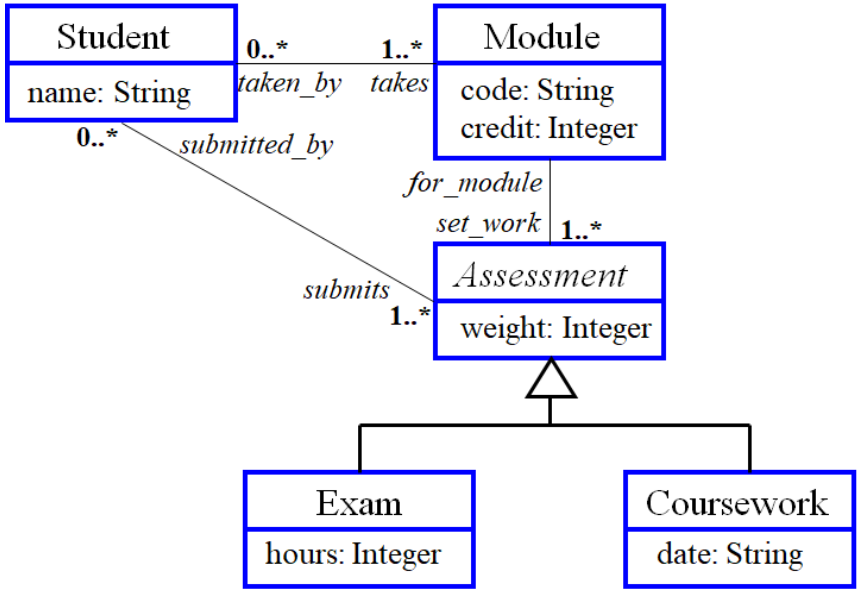
context *Student*
invariant:
takes →
select(*code.substring*(*1,2*) = *'CS'*).credit→sum( ) >= 90

Constraint (e)

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

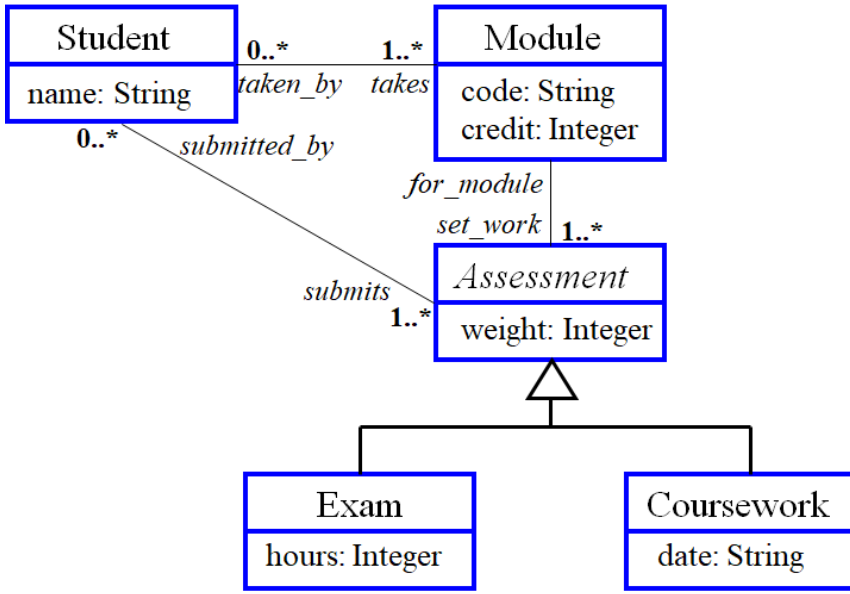
context *Module*

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



Constraint (f)

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



context *Student*
invariant:
takes→includesAll(submits.for_module)

Pre, Post Conditions

context Company::hire(p : Person)

pre hirePre1: p.isDefined()

pre hirePre2: employee->excludes(p)

post hirePost: employee->includes(p)

context Company::fire(p : Person)

pre firePre: employee->includes(p)

post firePost: employee->excludes(p)

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 X
    - constraint on the transition from one state to another.

THANK YOU!