Métodos de Desenvolvimento de Software
Software Development Methods
(MDS)
2016/2017

Object Constraint Language (OCL)

Negation and If

a	not a
true	false
false	true

```
if <booleanExpression> then
  <oclExpression1>
 else
  <oclExpression2>
endif
```

Structured Objects: Tuples

Tuple { partName1:partType1 = value1, partName2:partType2 = value2, ... }

Tuple { title:String = ‘UML 2 and the Unified Process’, publisher:String = ‘Addison Wesley’ }

Tuple { title:String = 'UML 2 and the Unified Process', publisher:String = 'Addison Wesley' }.publisher

Collection Types: Set, OrderedSet, Bag, and Sequence

OCL collection	Ordered	Unique (no duplicates allowed)	Association end properties
Set	No	Yes	{ unordered, unique } – default
OrderedSet	Yes	Yes	{ ordered, unique }
Bag	No	No	{ unordered, nonunique }
Sequence	Yes	No	{ ordered, nonunique }

Collection Operations


```
aCollection->collectionOperation( parameters... )
```

Collection Operations

- <collection> → size
  - → isEmpty
  - → notEmpty
  - → sum ( )
  - → count (object)
  - → includes (object)
  - → includesAll (collection)
  - → excludes (object)
  - → excludesAll (collection)

Collection operations - description

size() - number of elements in the collection self

includes() - information of whether an object is part of a collection

excludes() - information of whether an object isn't part of a collection

sum() - addition of all elements of a collection

sum() elements must be of a type supporting the + operation.

count() - number of times that object occurs in the collection self

includesAll() - information of whether all objects of a given collection are part of a specific collection

excludesAll() - information of whether none of the objects of a given collection are part of a specific collection

isEmpty() - information of whether a given collection is empty

notEmpty() - The information if a collection is not empty.

Collection operations - conversions

X(T)::asSet() : Set(T)
X(T)::asOrderedSet() : OrderedSet(T)
X(T)::asBag() : Bag(T)
X(T)::asSequence() : Sequence(T)

Converts a collection from one type of collection to another

When a collection is converted to a Set, duplicate elements are discarded

When a collection is converted to an OrderedSet or a Sequence, the original order (if any) is preserved, else an arbitrary order is established

Collections Access Operations

OrderedSet(T)::first() : T Sequence(T)::first() : T	Returns the first element of the collection
OrderedSet(T)::last() : T Sequence(T)::last() : T	Returns the last element of the collection
OrderedSet::at(i) : T Sequence::at(i) : T	Returns the element at position i
OrderedSet::indexOf(T) : Integer	Returns the index of the parameter object in the OrderedSet

Collection Selection Operations

X(T)::union(y : X(T)) : X(T)

Set(T)::intersection(y : Set(T)) : Set(T)

OrderedSet(T)::intersection(y : OrderedSet(T)) : OrderedSet(T)

Set(T)::symmetricDifference(y : Set(T)) : Set(T)

OrderedSet(T)::symmetricDifference(y : OrderedSet(T)) : OrderedSet(T)

Set(T)::-(y : Set(T)) : Set(T)

OrderedSet(T)::-(y : OrderedSet(T)) : OrderedSet(T)

X(T)::product(y : X(T2)) : Set(Tuple(first : T, second : T2))

X(T)::including(object : T) : X(T)

X(T)::excluding(object : T) : X(T)

Collection Selection Operations (cont.)

Sequence(T)::subSequence(i : Integer, j : Integer) : Sequence(T)

OrderedSet::subOrderedSet (i : Integer, j : Integer) : OrderedSet(T)

OrderedSet(T)::append(object : T) : OrderedSet(T)
Sequence(T)::append(object : T) : Sequence(T)

OrderedSet(T)::prepend(object : T) : OrderedSet(T)
Sequence(T)::prepend(object : T) : Sequence(T)

OrderedSet(T)::insertAt(index : Integer, object : T) : OrderedSet(T)
Sequence(T)::insertAt(index : Integer, object : T) : Sequence(T)

Collections Iteration


```
aCollection->iteratorOperation( <iteratorVariable> :<Type> |
                                <iteratorExpression>
                              )
```

Collections Iteration - Boolean

<collection>

→ forAll (e:T* | <b.e.>)

→ exists (e:T | <b.e.>)

b.e. stands for: boolean expression

The result of these Iterations is either **true** or **false**

Collections Iteration - Boolean

X(T)::exists(i : T | iteratorExpression) : Boolean	Returns true if the iteratorExpression evaluates to true for at least one value of i, else returns false
X(T)::forAll(i : T | iteratorExpression) : Boolean	Returns true if the iteratorExpression evaluates to true for all values of i, else returns false
X(T)::forAll(i : T, j : T ..., n : T | iteratorExpression) : Boolean	Returns true if the iteratorExpression evaluates to true for every { i, j ... n } Tuple, else returns false The set of { i, j ... n } pairs is the Cartesian product of the target collection with itself

Collections Iteration: Selection

<collection>

→ select (e:T | <b.e.>)

→ reject (e:T | <b.e.>)

→ collect (e:T | <v.e.>)

→ any (e:T | <b.e.>)

b.e. stands for: boolean expression

v.e. stands for: value expression

The result of these Iterations is a collection of elements

Collections Iteration: Selection

X(T)::any(i : T | iteratorExpression) : T	Returns a random element of the target collection for which iteratorExpression is true
X(T)::collect(i : T | iteratorExpression) : Bag(T)	Returns a Bag containing the results of executing iterator-Expression once for each element in the target collection
X(T)::select(i : T | iteratorExpression) : X(T)	Returns a collection containing those elements of the target collection for which the iteratorExpression evaluates to true
X(T)::reject(i : T | iteratorExpression) : X(T)	Returns a collection containing those elements of the target collection for which the iteratorExpression evaluates to false

Result is collection of Objects of the type of the Set

Result is a Bag of Objects of any type

Collections: Iterate operation


```
aCollection->iterate( <iteratorVariable> : <Type>
                      <resultVariable> : <ResultType> = <initializationExpression> |
                          <iteratorExpression>
                      )
```

Example:

```
Set{1,2,3,4,5}->iterate(x:Integer; y:Set(Integer)=Set{0}|
                                                    y->including(x))
```

Collection Operations: examples

specifies that the collection of all the employees older than 50 years is not empty.

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

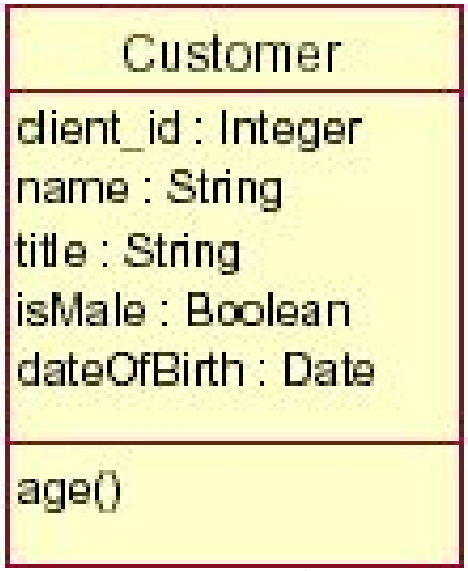
The ***self.employee*** is of type **Set(Person)**. The ***select*** takes each person from *self.employee* and evaluates *age > 50* for this person. If this results in *true*, then the person is in the result Set.

specifies that the collection of all the employees who are **not** married is empty.

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

Expressing uniqueness constraints

- Constraint: customer identifiers should always be unique



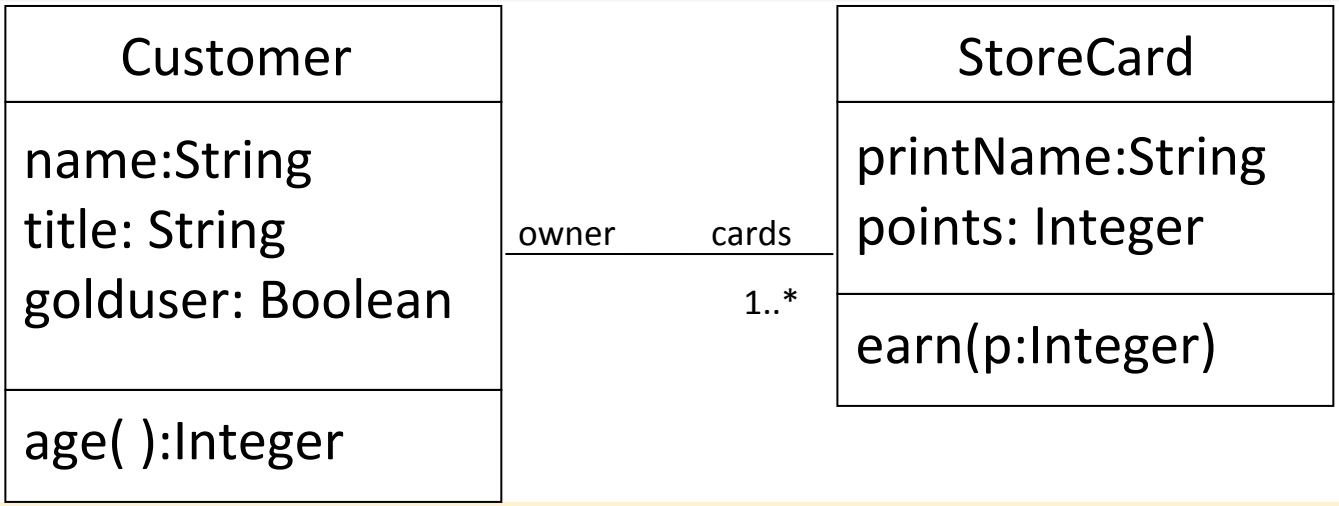
returns all instances of a given type

Context Customer inv:

Customer.allInstances->forAll(c1, c2: Customer | c1 <> c2 implies c1.client_id <> c2.client_id)

returns all instances of type Customer

Changing the context

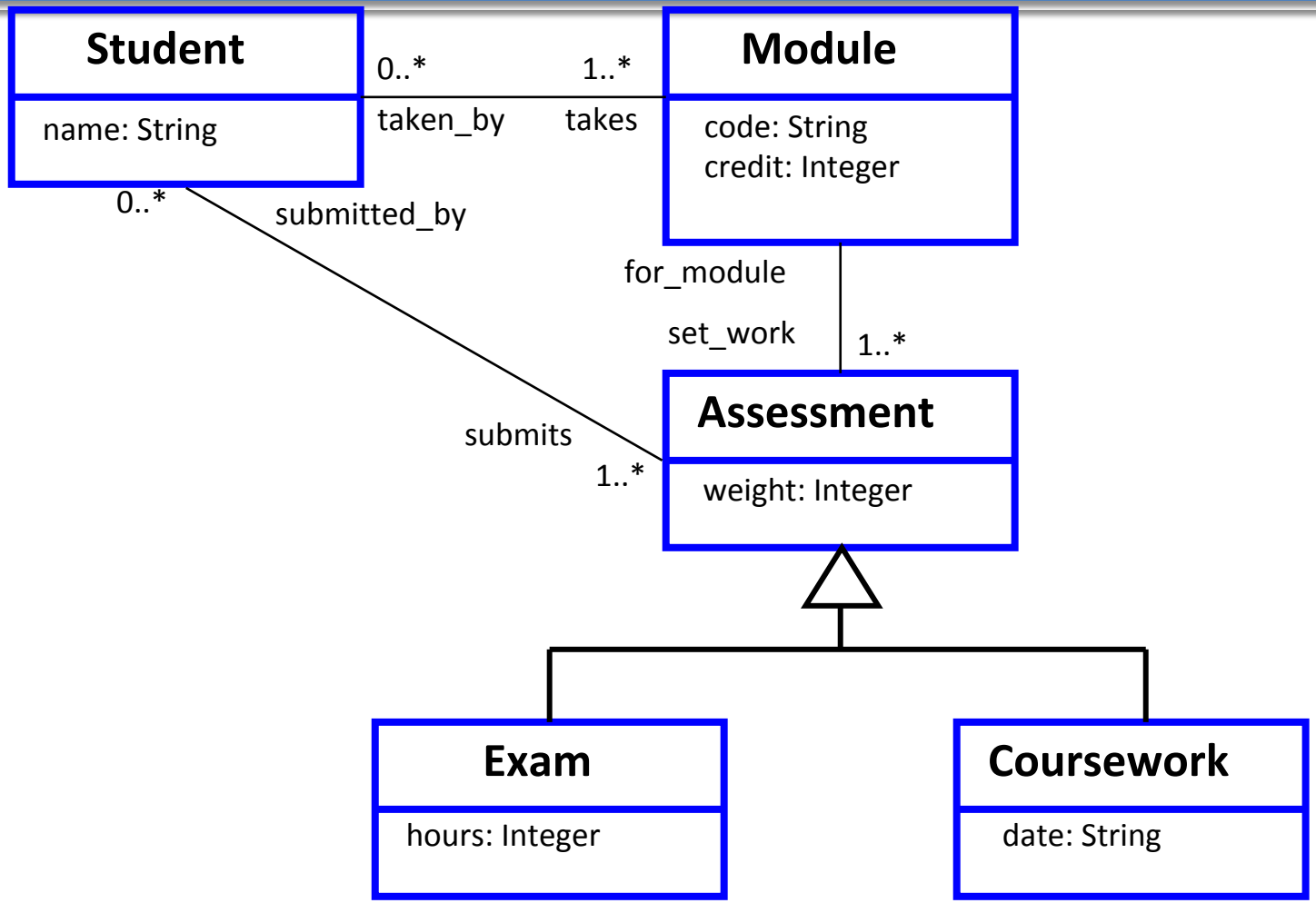


context StoreCard
inv: printName = owner.title.concat(owner.name)

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

Note switch of context!

Example UML diagram



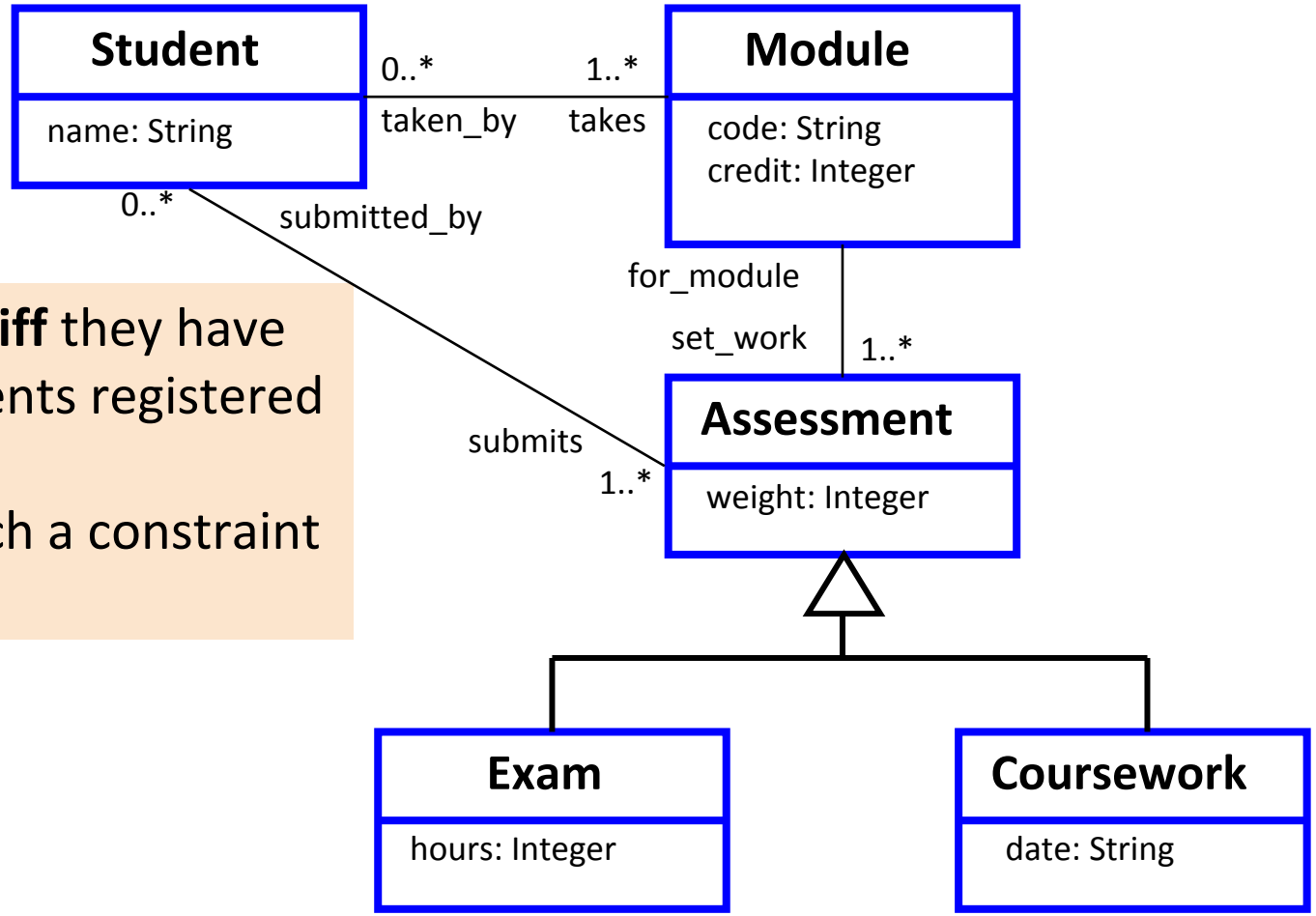
Constraints

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

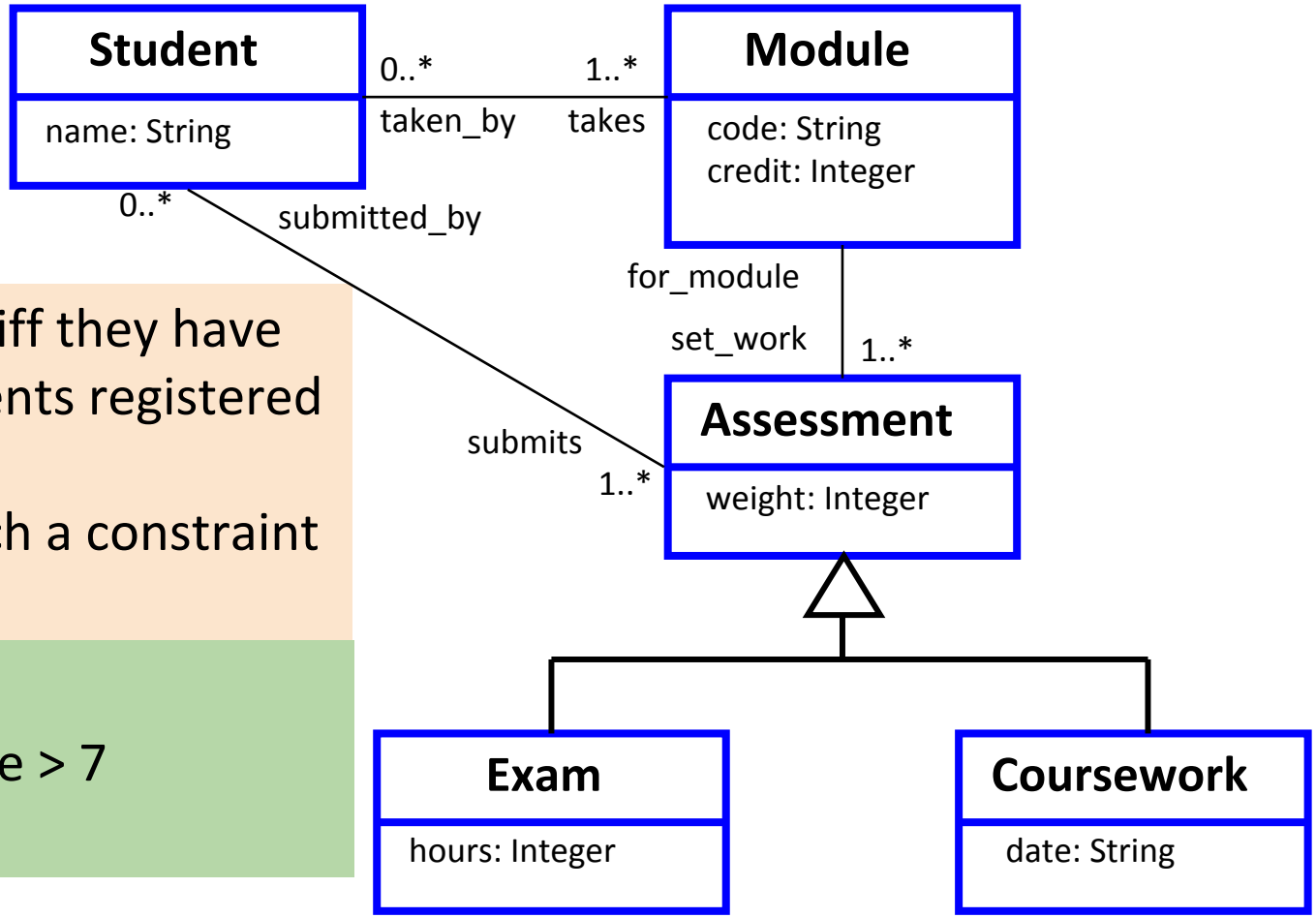
Constraint (a)

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

Note: when should such a constraint be imposed?



Constraint (a)



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

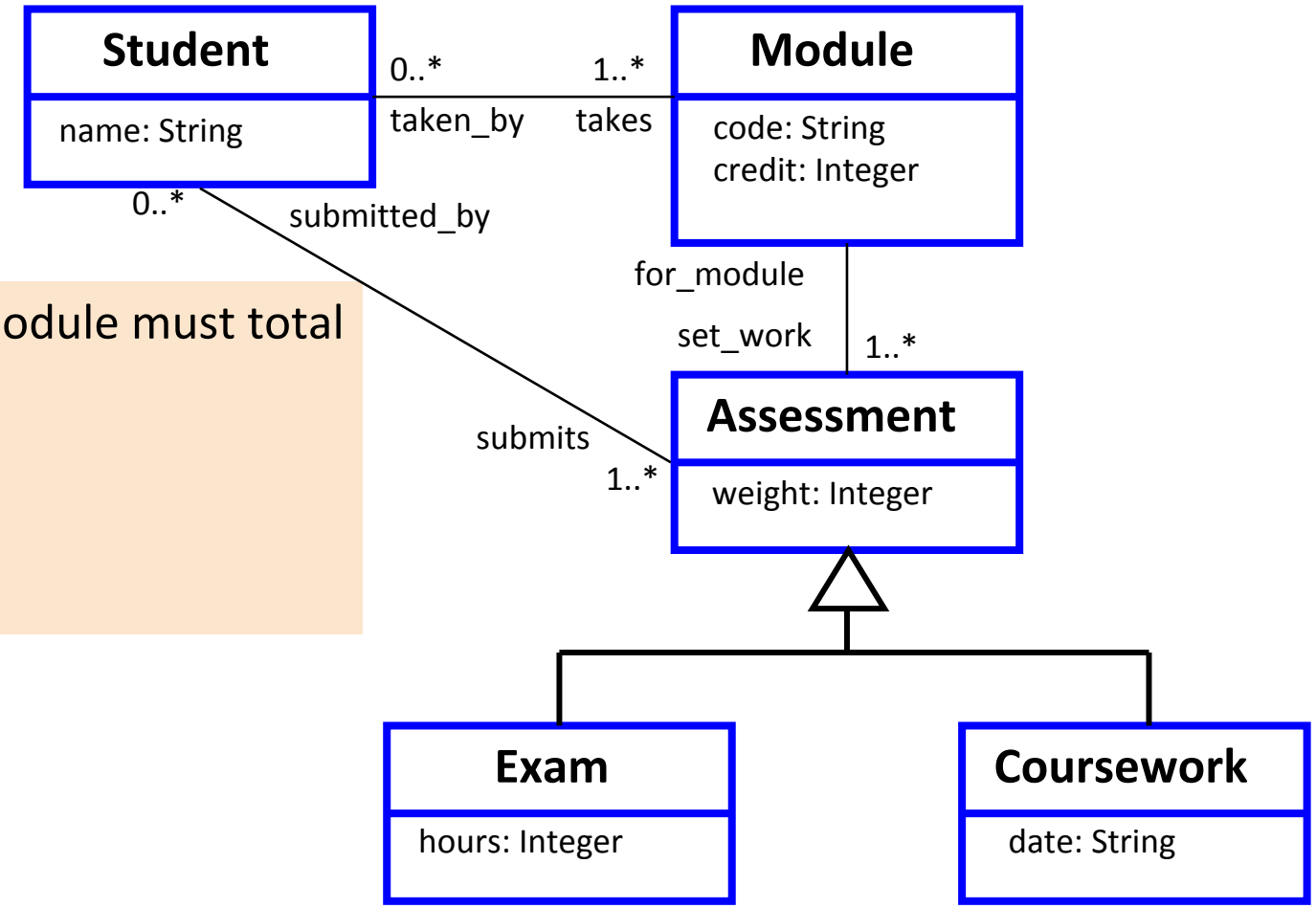
Note: when should such a constraint be imposed?

context Module

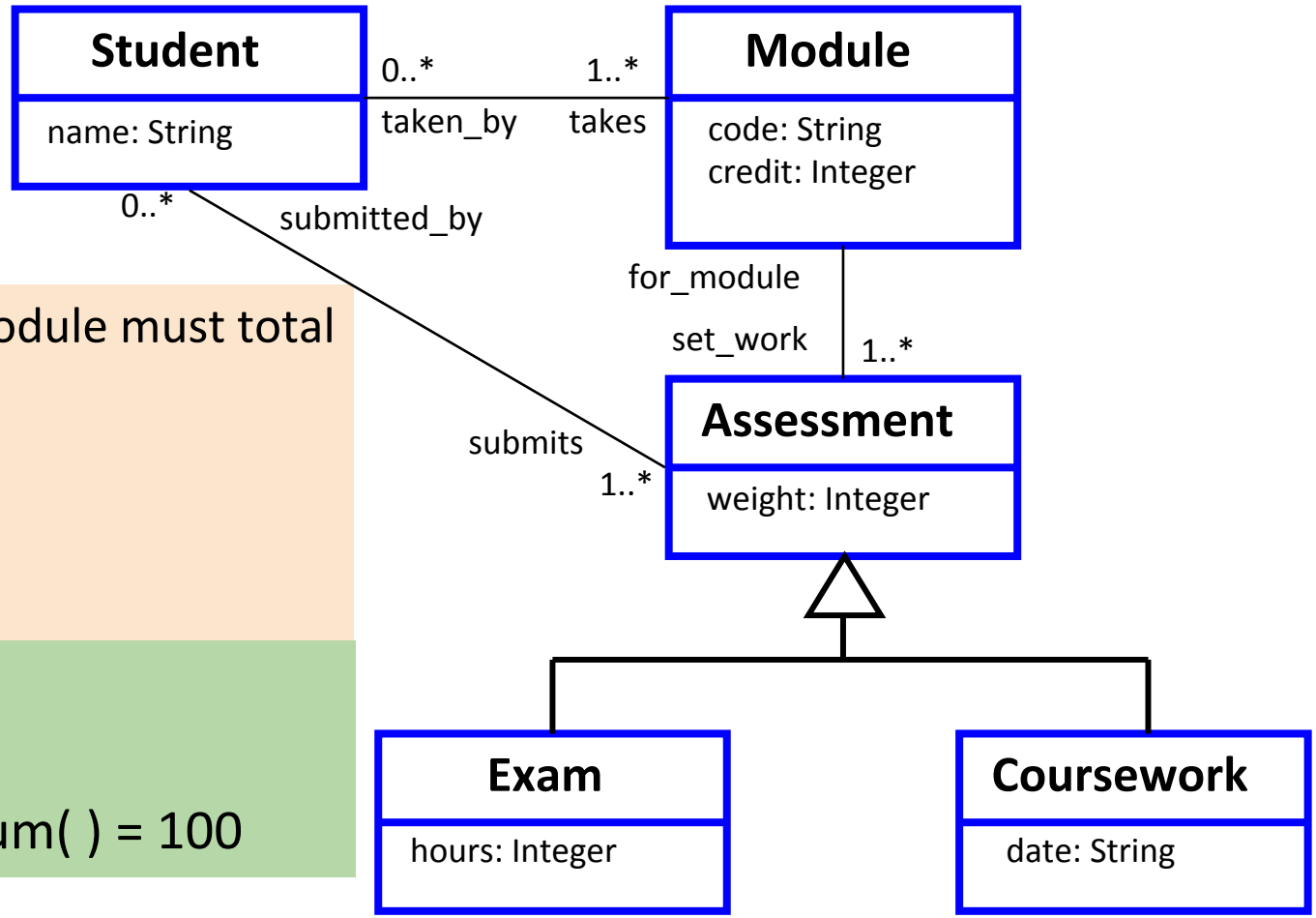
inv: taken_by→size > 7

Constraint (b)

The assessments for a module must total 100%



Constraint (b)

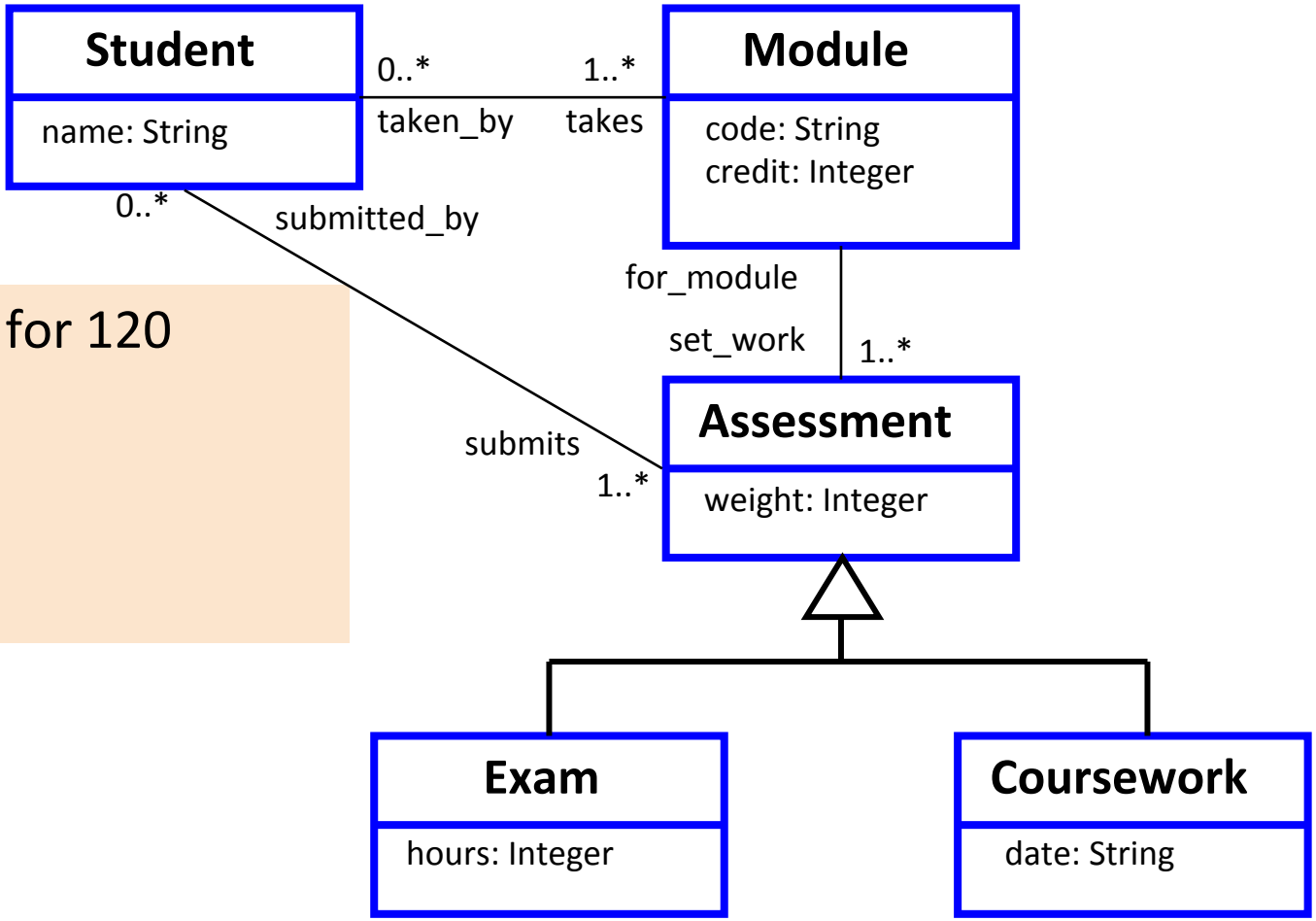


The assessments for a module must total 100%

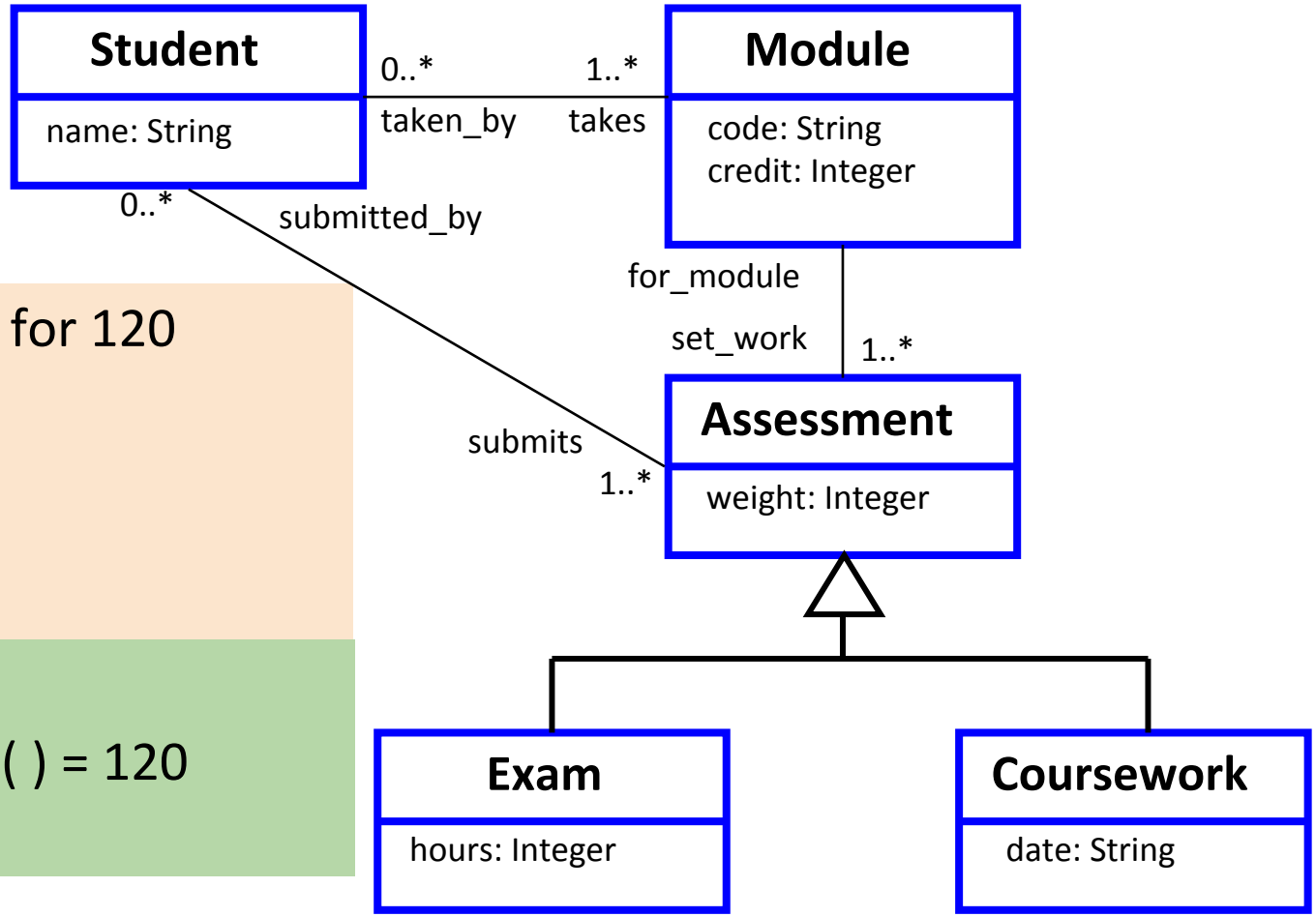
context Module
inv:
set_work.weight→sum( ) = 100

Constraint (c)

Students must register for 120 credits each year



Constraint (c)

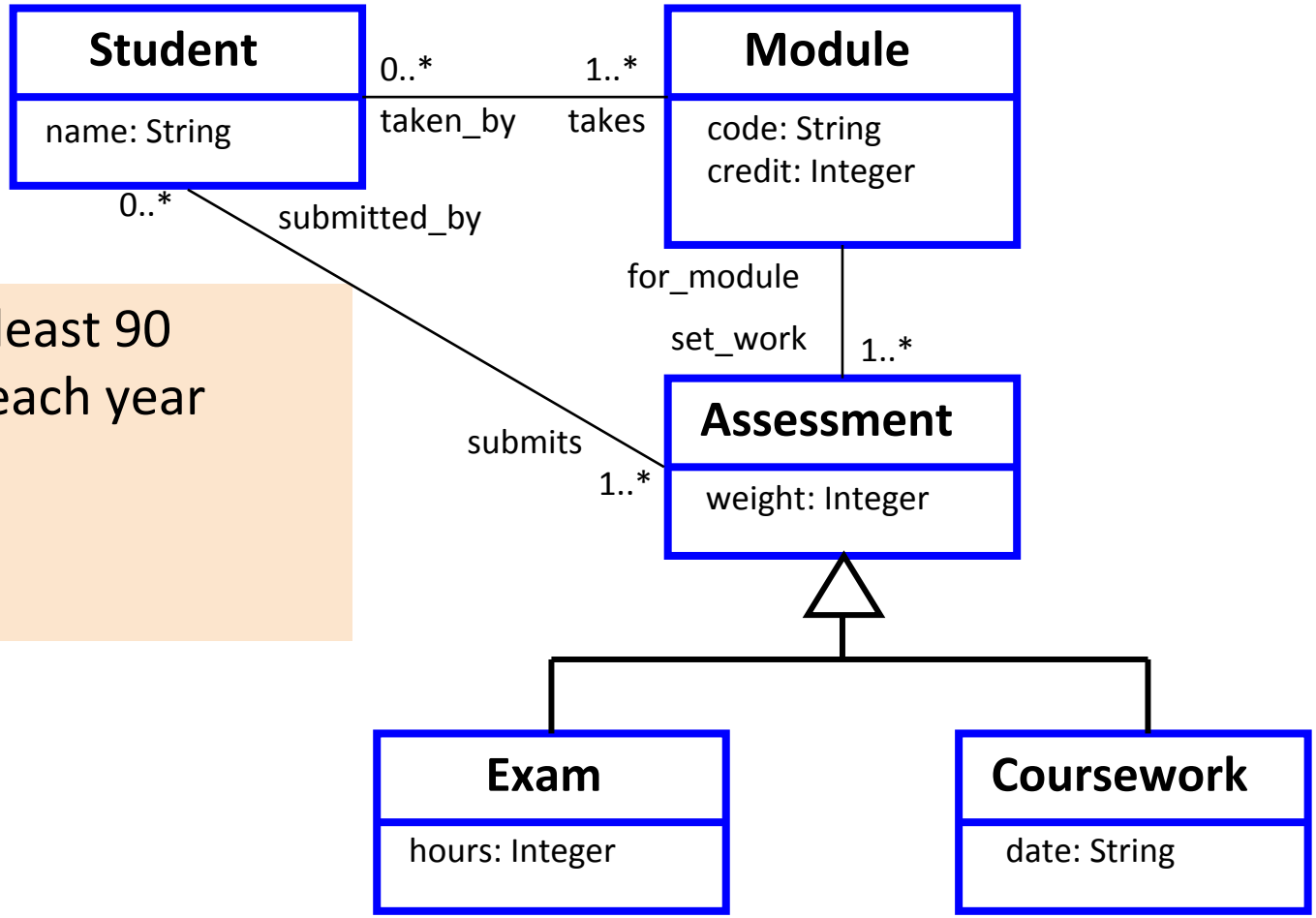


Students must register for 120 credits each year

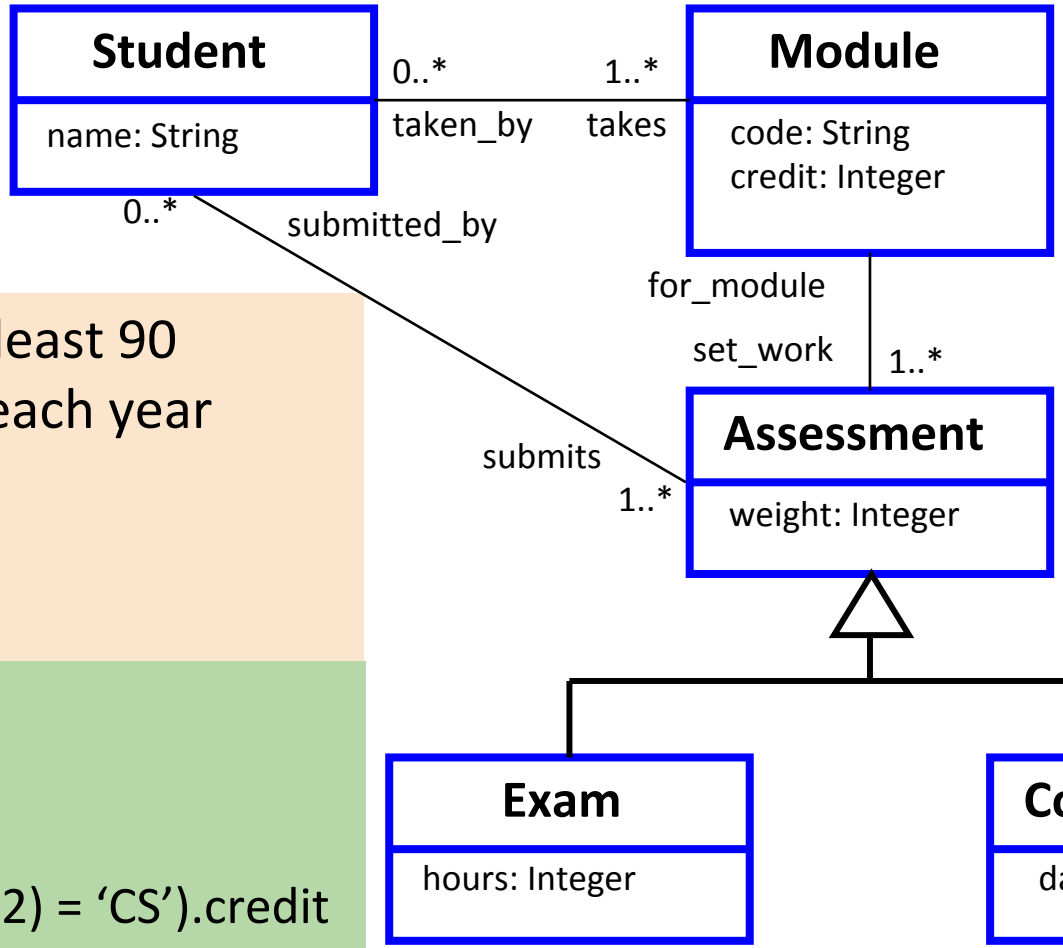
context Student
inv: takes.credit→sum( ) = 120

Constraint (d)

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



Constraint (d)

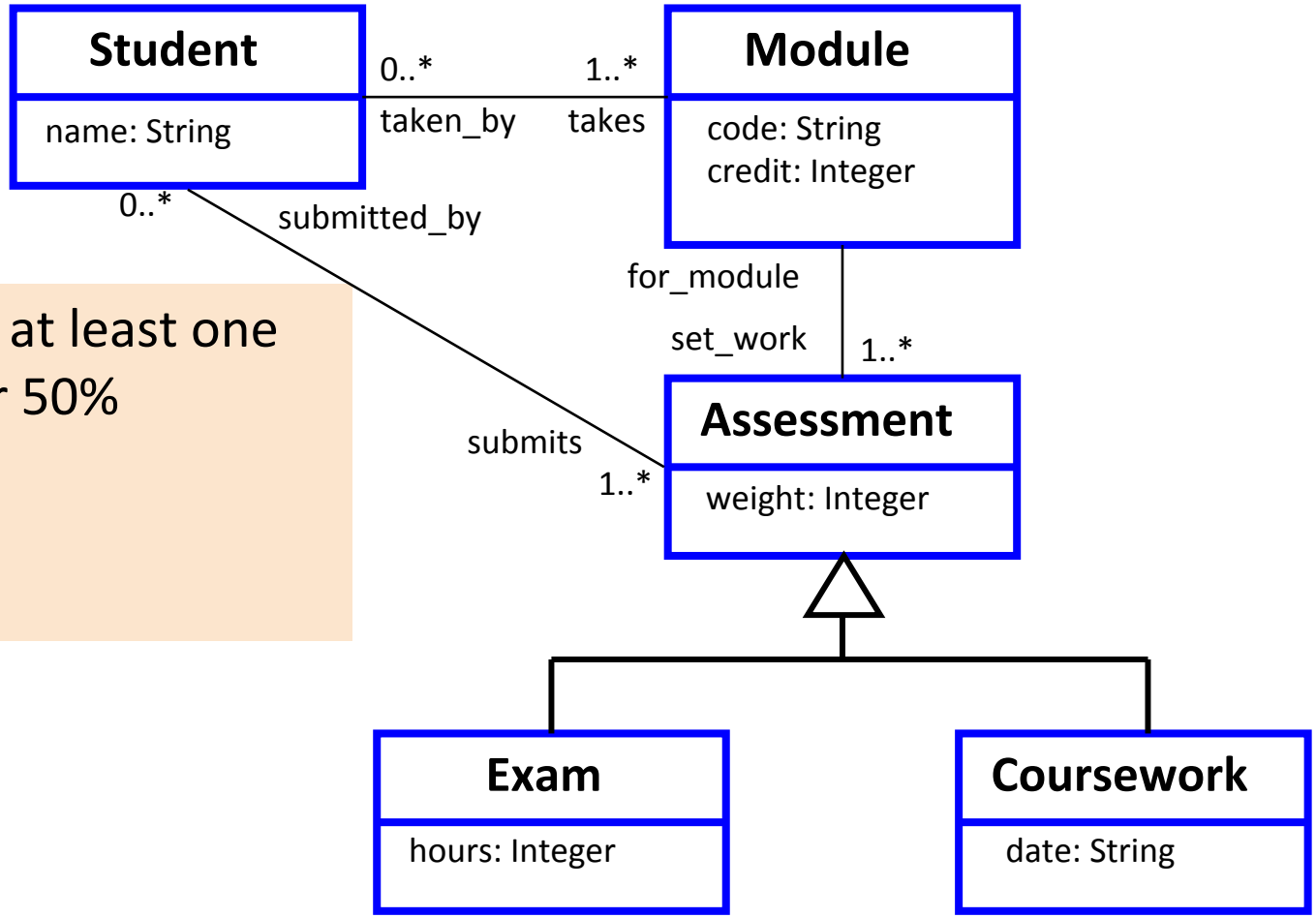


Students must take at least 90 credits of CS modules each year

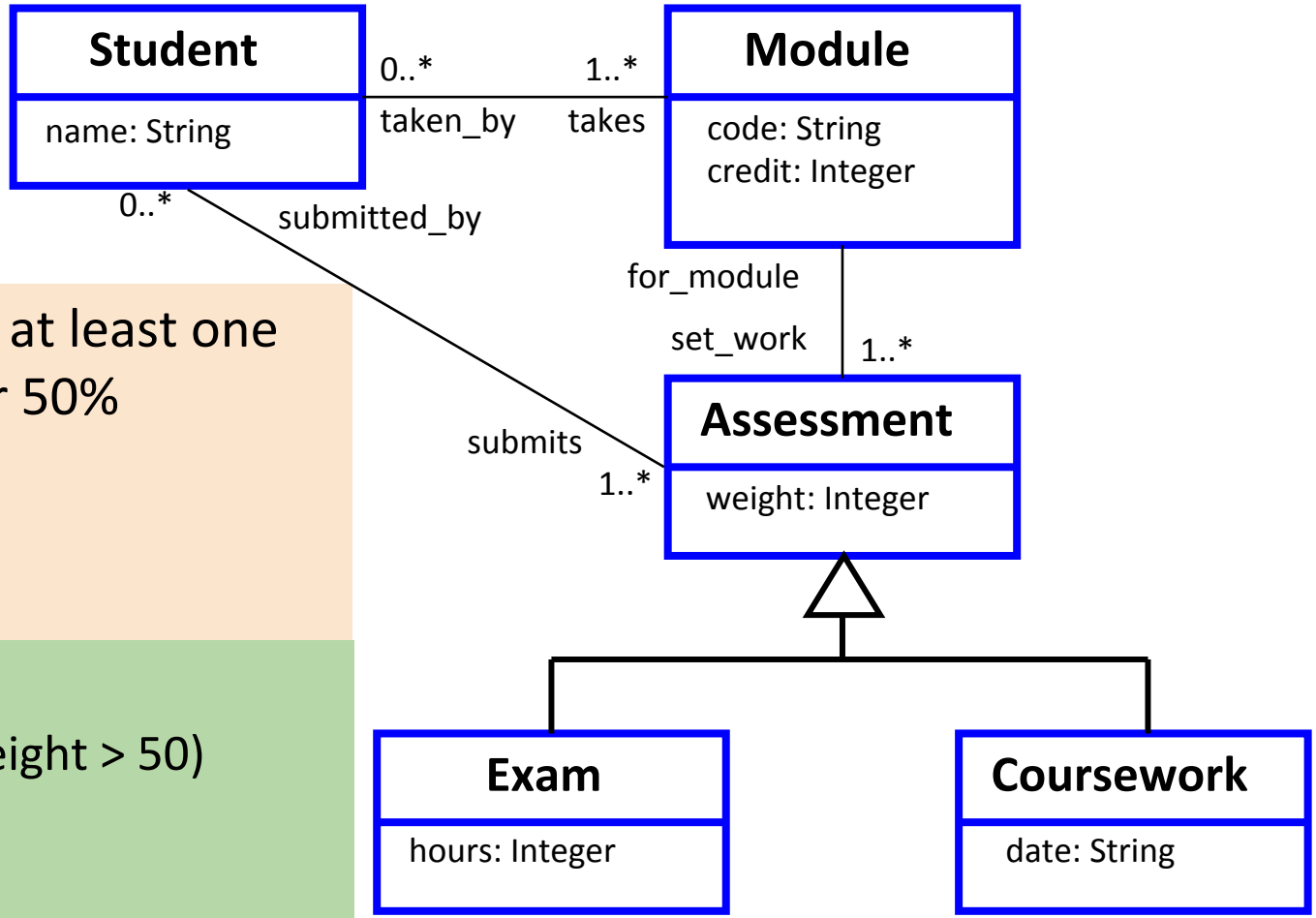
context Student
inv:
takes →select(
code.substring(1,2) = 'CS').credit
→sum( ) >= 90)

Constraint (e)

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



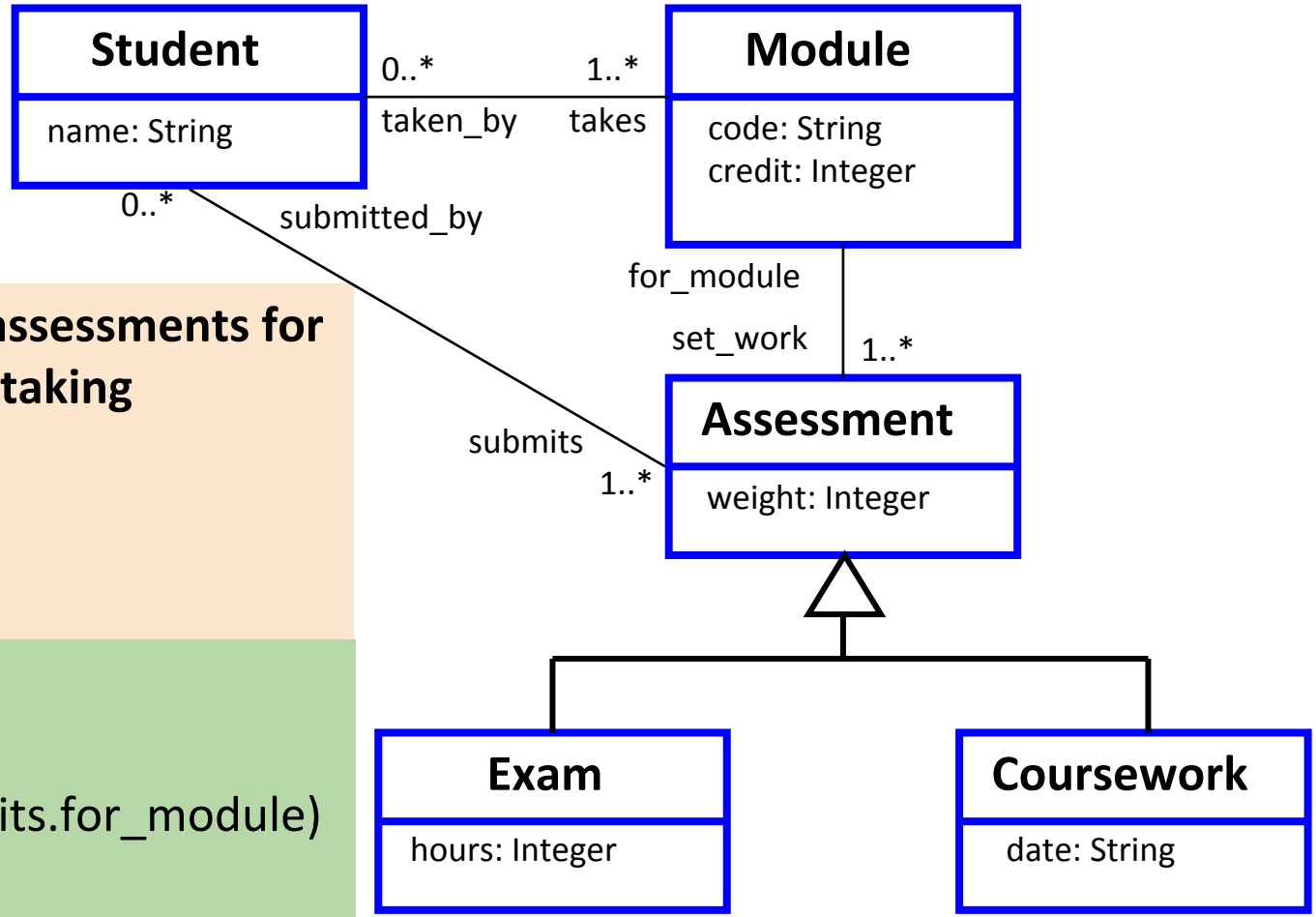
Constraint (e)



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

context Module
inv: set_work→exists(weight > 50)

Constraint (f)



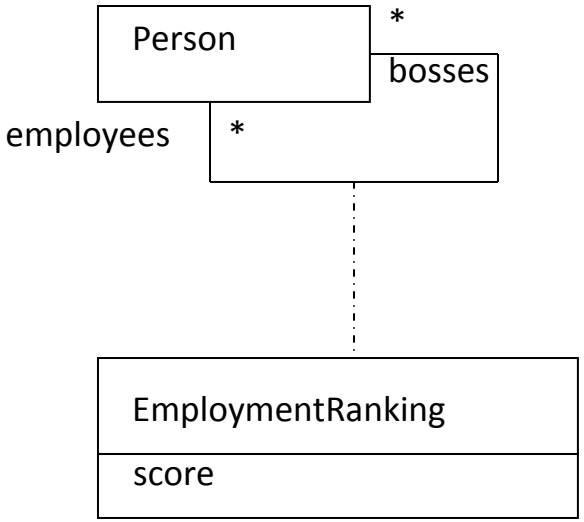
Students can only have assessments for modules which they are taking

context Student
inv:
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]



Every boss must give at least one 10 score:

context Person
inv:
employmentRanking[employees]->exists(score = 10)

The accumulated score of an employee is positive:

context Person
inv:
employmentRanking[bosses].score->sum()>0

Due to unary association, we need to state the direction of the navigation

Classes and Subclasses

Consider the following constraint

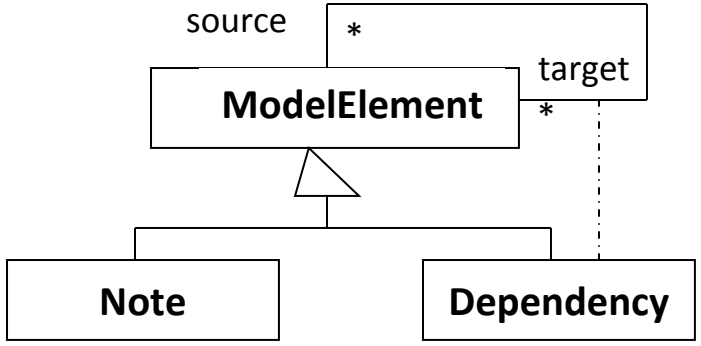
context LoyaltyProgram
inv:
partners.deliveredServices.transaction.points->sum() < 10,000

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

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

Classes and Subclasses

“The target of a dependency is not its source”



context Dependency
inv: self.source <> self

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

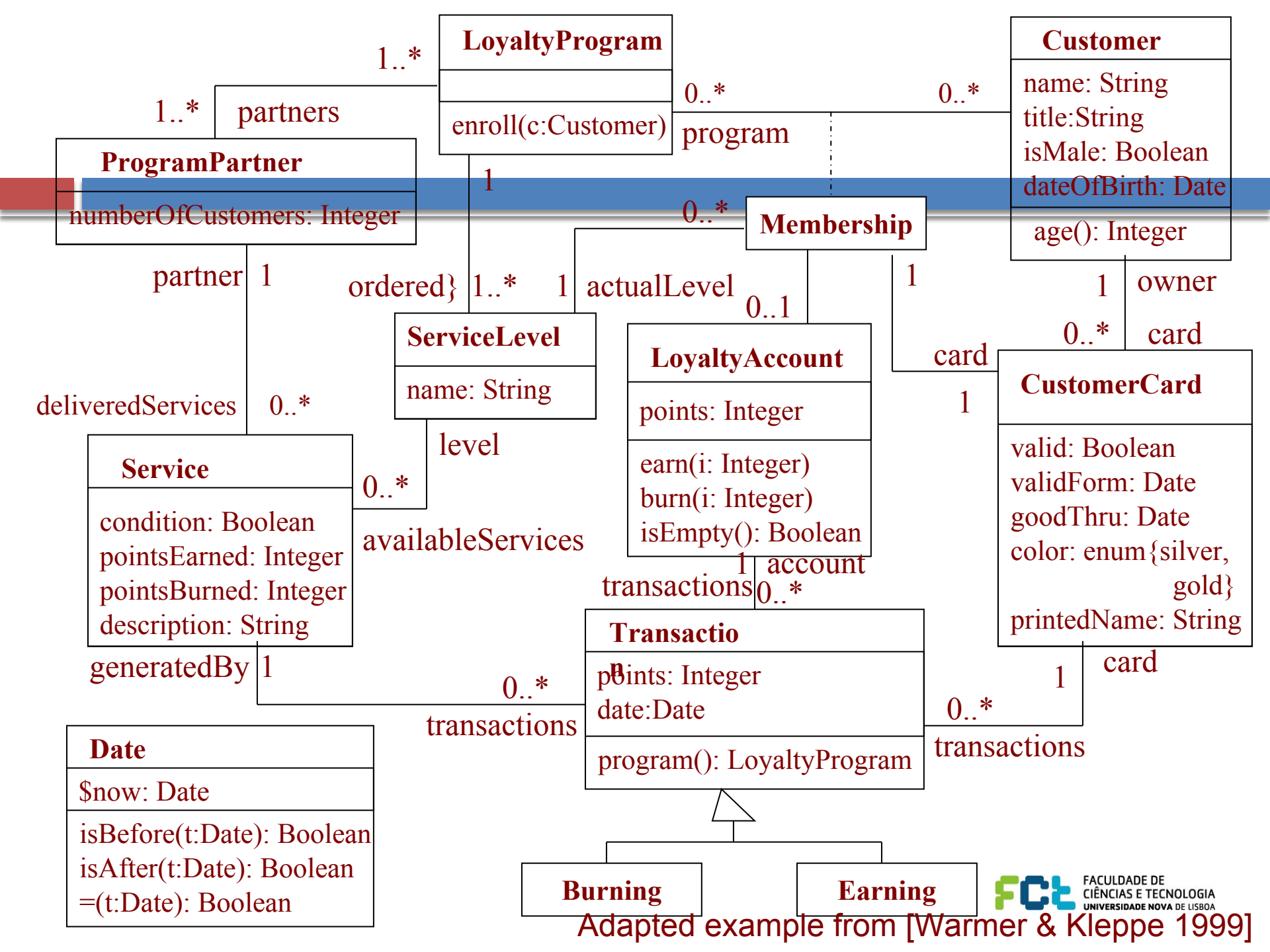
context Dependency
inv: self.oclAsType(Dependency).source <> self
inv: self.oclAsType(ModelElement).source -> isEmpty()

References

[1] OCL website: http://www.omg.org/uml/
[2] The Object Constraint Language, Precise Modeling with UML, Jos Warmer and Anneke Kleppe, Addison-Wesley, 1999.
[3] The UML's Object Constraint Language: OCL Specifying Components, JAOO Tutorial – September 2000, Jos Warmer & Anneke Kleppe
[4] http://www.db.informatik.uni-bremen.de/projects/USE-2.3.1/

Example of a static UML Model

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).



Adapted example from [Warmer & Kleppe 1999]

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Invariants on Attributes

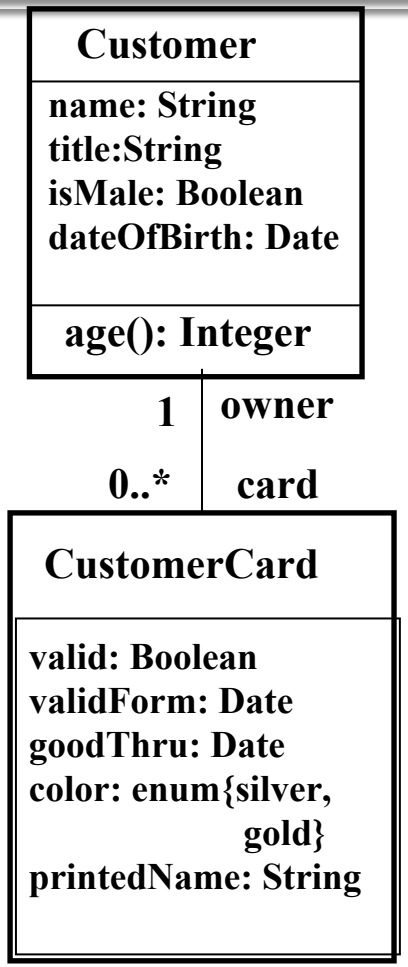
- Invariants on attributes:

Named invariant

```
context Customer
   invariant agerestriction: age >= 18
context CustomerCard
   invariant correctDates:
   validFrom.isBefore(goodThru)

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.

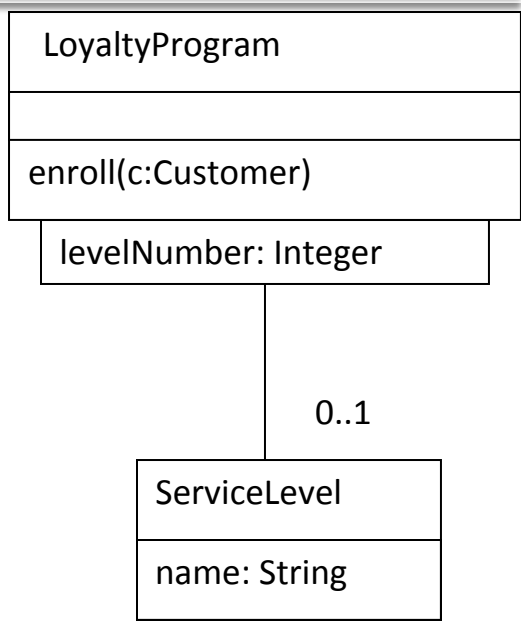


Invariants using Navigation through Qualified Association

To navigate qualified associations you need to index the qualified association using a qualifier

object.navigation[qualifierValue, ...]

If there are multiple qualifiers their values are separated using commas



Examples

context LoyaltyProgram

inv: serviceLevel[1].name = 'basic'

context LoyaltyProgram

inv: serviceLevel->exists(name = 'basic')