In the name of God

Object Constraint language

Advanced topics in Software Engineering

Leila Samimi-Dehkordi
PhD. Student
Department of Software Engineering
Faculty of Computer Engineering
University of Isfahan
Email: *l.samimi@gmail.com*

These slides are prepared for Master student of Islamic Azad University of Shahrekord

Introduction

- OCL is a formal language used to describe expressions on UML models.
- When the OCL expressions are evaluated, they do not have side effects.
  - Their evaluation cannot alter the state of the corresponding executing system

More about OCL

- OCL is a declarative programming language, with well defined syntax and semantics.
  - States what should be done, not how
    - Implementation independent
  - Expressions have no side effects
    - Evaluation does not change the system state.
- OCL is strongly typed.
  - Each OCL expression has a type and evaluates to a value or to an object within the system:
    - A constraint is a valid OCL expression of type Boolean

OCL types

- Basic types
  - Boolean
  - Integer
  - Real
  - String
- Collection types:
  - Collection: is an abstract type for others
  - Set: without duplicate elements
  - Ordered Set (only OCL2): a set in which the elements are ordered by their position
  - Bag: a set that may contain duplicate elements
  - Sequence: a bag in which the elements are ordered

Collections have a set of predefined operations
They are accessed using the -> notation

OCL Functions on Collection Types

- With collection types, an OCL expression
  - 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->function**

Example for OCL functions


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

or with **shorthand** notation:

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

Standard Operations on all Collection Types

Operation	Description
size()	The number of elements in the collection
count(object)	The number of occurences of object in the collection.
includes(object)	True if the object is an element of the collection.
includesAll(collection)	True if all elements of the parameter collection are present in the current collection.
excludes(object)	True if the object is *not* an element of the collection.
excludesAll(collection)	True if all elements of the parameter collection are *not* present in the current collection.
isEmpty()	True if the collection contains no elements.
notEmpty()	True if the collection contains one or more elements.

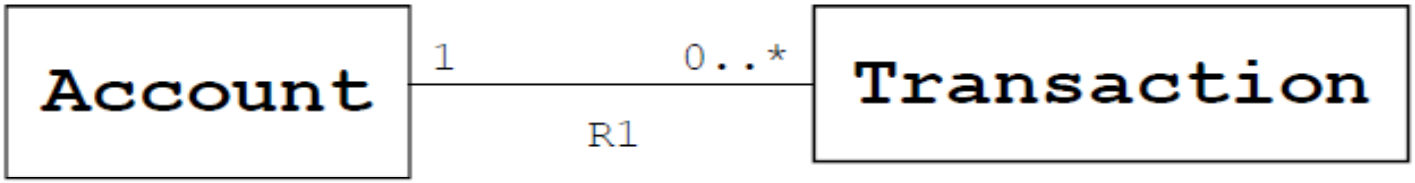
User defined types

- Class type (Model type):
  - A class has the following Features:
    - Attributes (start)
    - Operations (duration())
    - Class attributes (Date::today)
    - Class operations
    - Association ends (“navigation expressions“)
- Enumeration type (Gender, Gender::male)

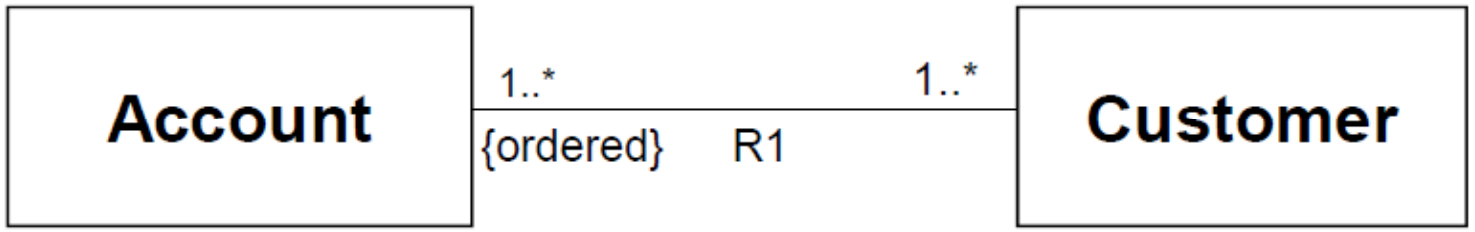
Navigation

- If self is class **C**, with attribute **a** then
  - **self.a** evaluates to the object stored in a.
- If **C** has a **1..*** association called **R1** to another class **D**
  - **self.R1** returns to a Set whose elements are of type **D**
  - if **R1** is {ordered} then a Sequence is returned
  - If **D** has attribute **b** then
  - **self.R1.b** evaluates to the set (or sequence if {ordered is used}) of all the **b's** belonging to **D**

Example for Navigation



self.R1 returns a **set** of transactions if we are in the context Account



The key word {ordered} is a predefined constraint in UML, which means the collection is ordered

self.R1 returns a sequence of accounts in the context Customer

Constraints

- Constraints express invariants over classes.
- They can also be used to express Pre-condition:
  - Constraint that must hold before the execution of an Operation
- They can also be used to express Post- condition:
  - Constraint that must hold after the execution of an Operation
- Since OCL is not limited to class diagrams, we can define constraint on the transition from one state to another.
  - These constraints are named Guards.

Invariants

- An invariant is a constraint that should be true for an object during its complete lifetime.
- A usual invariant constraint has the following structure:

  context *ClassName*

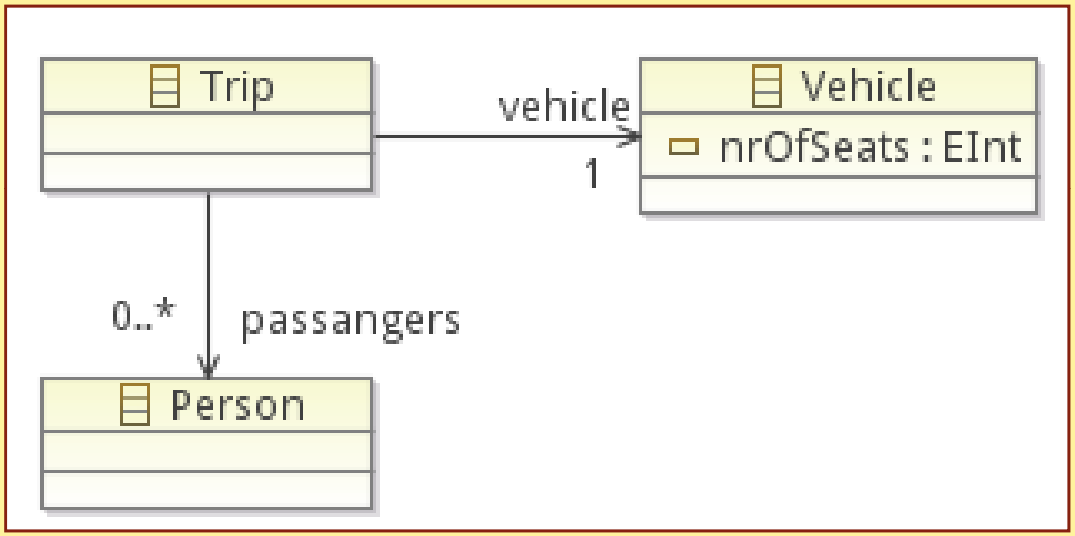
  inv: *OCL-expression*

```
context Account
  inv: balance >= 0
```

- Such an invariant will apply to all objects of the given class.
- The expression must be a boolean expression (a predicate).

Example 1

- We are interested in describing trips that combine a number of persons (passengers) in a vehicle.
- A natural integrity constraint is that the vehicle associated with the trip needs to be large enough
- to accommodate all the involved passengers.
- This constraint cannot be expressed directly in the diagrammatic language of class diagrams.

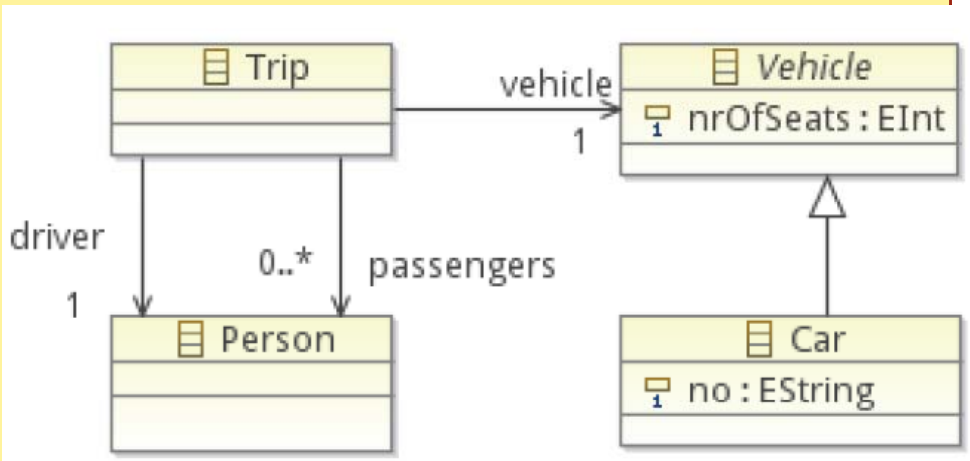


context Trip

inv: passengers->size() <= vehicle.nrOfSeats

Example 1(cont.)

- We want to add cars, and drivers to our model.
- Further, we would like to make sure that a driver is listed on the passenger list.




```
context Trip
inv: passengers->includes(driver)
```

or

```
context Trip
inv: self.passengers->includes(self.driver)
```

Pre/Post-conditions

- Pre-condition: Constraint that must be true just prior to the execution of an operation.
- Pre-Condition syntax:

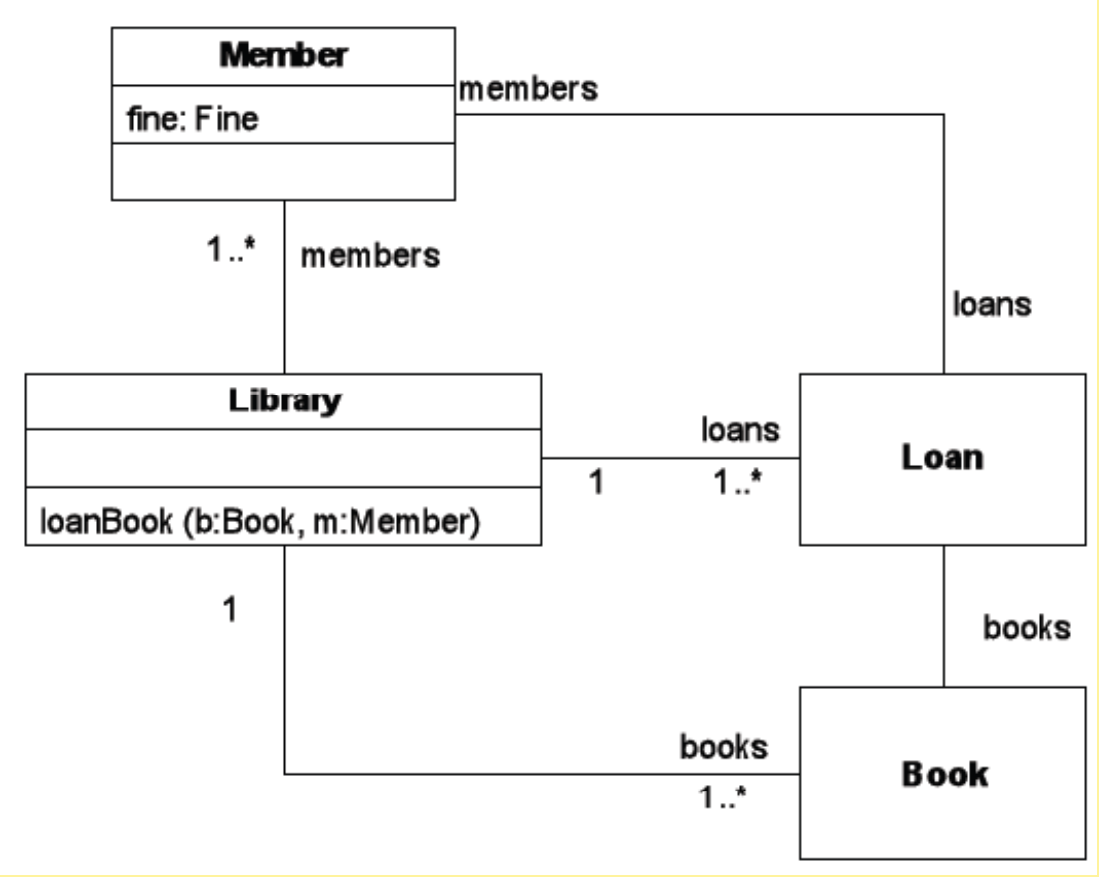
```
context <classifier>::<operation> (<parameters>)
pre [<constraint name>]:
<Boolean OCL expression>
```

- Post-condition: Constraint that must be true just after to the execution of an operation.
- Post-Condition syntax:

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

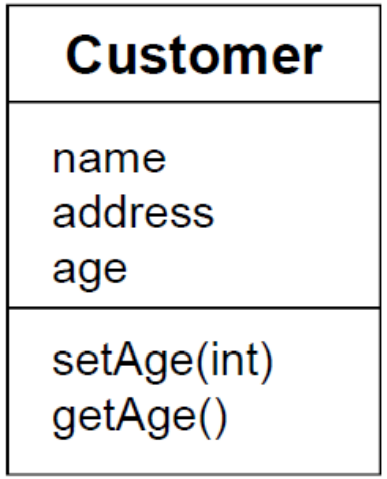
Example for pre/post-conditions


```
context Library::loanBook(b:Book, m:Member)
    pre: m.fine <= 0
    post: self.loans->exists(lo | lo.books = b and lo.members = m)
```



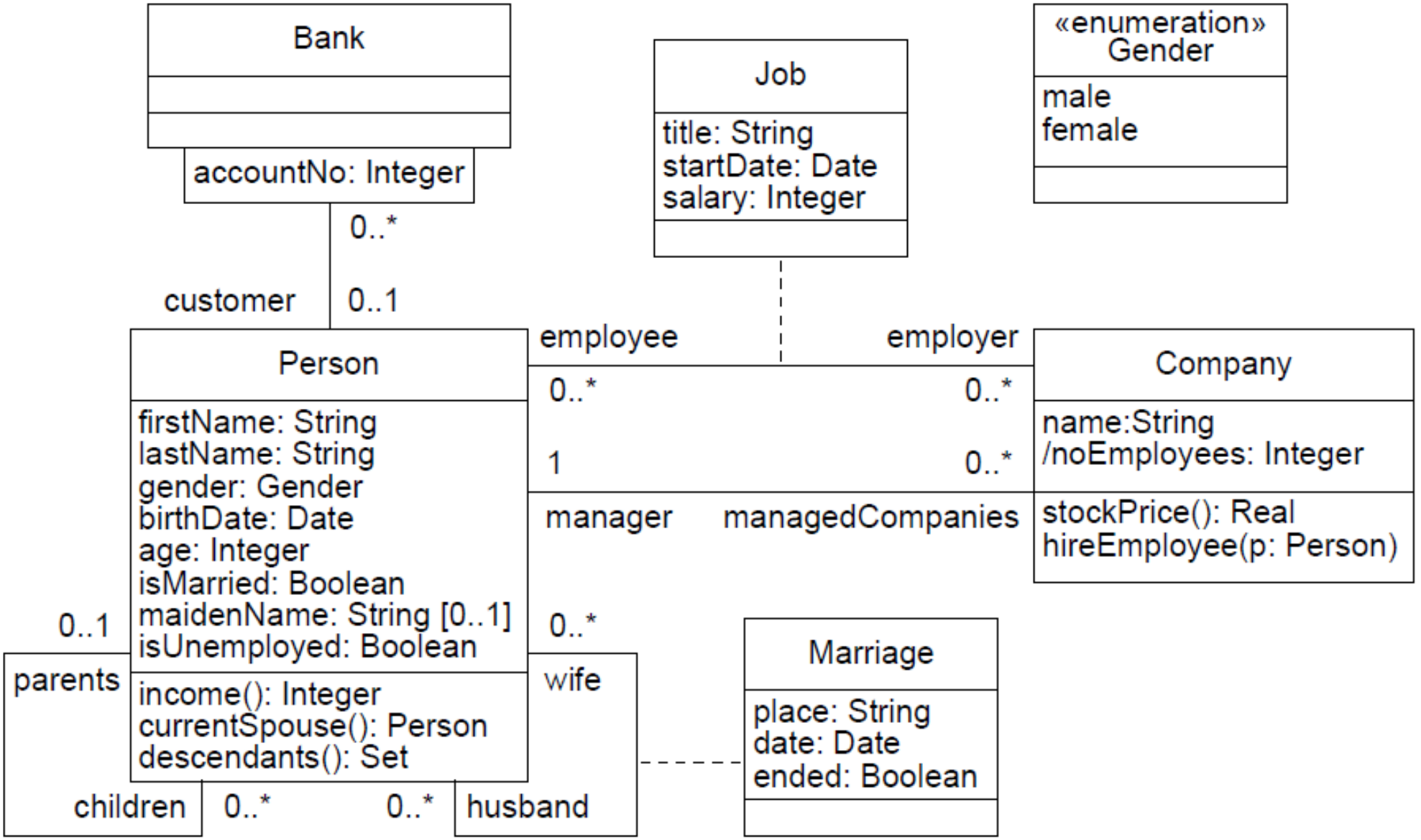
@pre

- In OCL, we use the @pre suffix to indicate that we are referring to a value at the start of an operation




```
context Customer::setAge(a: integer)
  pre: a > 1
  post: age = age@pre + 1
```

Class Diagram (Example)



Body Expression

- Used to indicate the result of a query operation
  - Example 1: Income of a person is the sum of the salaries of her jobs

    context Person::income(): Integer

    body: self.job.salary->sum()
- Expression must conform to the result type of the operation
  - Example 2: A method that obtains the direct and indirect descendants of a person

    context Person::descendants(): Set

    body: result = self.children->union(self.children->collect(c | c.descendants()))

Body expression (cont.)

- Pre and post-conditions, and body expressions may be mixed together after one

```
operation context
context Person::income(): Integer
pre: self.age >= 18
body: self.job.salary->sum()
post: result < 5000
```

Let expression

- Allows to define a variable that can be used in a constraint:

```
context Person inv:
let numberJobs: Integer = self.job->count() in
if isUnemployed then
numberJobs = 0
else
numberJobs > 0
endif
```

- A let expression is only known within its specific expression.

Definition Expressions

- Enable to reuse variables or operations over multiple expressions
- Must be attached to a classifier and may only contain variable and/or operation definitions:

```
context Person
def: name: String = self.firstName.concat(' ').concat(lastName)
def: hasTitle(t: String): Boolean = self.job->exists(title = t)
```

- Names of the attributes/operations in a def expression must not conflict with the names of attributes/association ends/operations of the classifier.

Initial and Derived Values

- Used to indicate the initial or derived value of an attribute or association end.
- Attribute isMarried in Person is initialized to false

```
context Person::isMarried: Boolean
init: self.isMarried = false
```

- Attribute noEmployees in Company is a derived attribute

```
context Company::noEmployees: Integer
derive: self.employee->size()
```

Predefined Properties on All Objects

- Several properties apply to all objects
  - **oclIsTypeOf(t: Type): Boolean** is true if the type of **self** and **t** are the same
  - **oclIsKindOf(t: Type): Boolean** is true if **t** is a direct/indirect type of **self**
  - **oclInState(s: State): Boolean** is true if **self** is in the state **s**
  - **oclIsNew: Boolean**, in a postcondition, is true if **self** has been created while performing the operation
- Example

```
context Person
inv: self.oclIsTypeOf(Person) -- is true
inv: self.oclIsTypeOf(Company) -- is false
```

Class Features

- Features of a class, not of its instances
- They are either user-defined or predefined.
- Predefined:
  - Predefined feature allInstances holds on all types
  - There are at most 100 persons

```
context Person inv:
Person.allInstances()->size() <= 100
```

- User-defined:
  - A user-defined feature averageAge of class Person

```
context Person inv:
Person.averageAge =
    Person.allInstances()->collect(age)->sum()/
    Person.allInstances()->size()
```

Select Operation on a Collection

- Obtains the subset of elements of a collection satisfying a Boolean expression
- Alternative expressions for the select operation
  - collection->select(Boolean-expression)
  - collection->select(v | Boolean-expression-with-v)
  - collection->select(v: Type | Boolean-expression-with-v)
- Example: A company has at least one employee older than 50

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

Reject Operation on a Collection

- Obtains the subset of all elements of the collection for which a Boolean expression evalutes to False
- Alternative expressions for the reject operation:
  - collection->reject(Boolean-expression)
  - collection->reject(v | Boolean-expression-with-v)
  - collection->reject(v: Type | Boolean-expression-with-v)
- Example: The collection of employees of a company who have not at least 18 years old is empty:

```
context Company inv:
self.employee->reject(age>=18)->isEmpty()
```

- A reject expression can always be restated as a select with the negated expression.

Collect Operation on a Collection

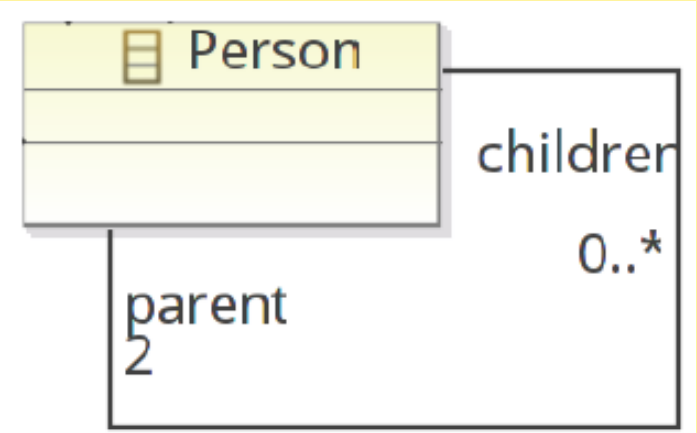
- Derives a collection from another collection, but which contains different objects from the original collection
- Alternative expressions for the collect operation:
  - collection->collect(expression)
  - collection->collect(v | expression-with-v)
  - collection->collect(v: Type | expression-with-v)
- Example: Collect of birth dates for all employees in the context of a Company object
  - self.employee->collect(birthDate)
  - self.employee->collect(p | p.birthDate)
  - self.employee->collect(p:Person | p.birthDate)
- Resulting collection above is a Bag.
  - some employees may have the same birth date

ForAll Operation on a Collection

- Specifies a Boolean expression that must be true for all elements in a collection
- Alternative expressions for the forall operation
  - collection->forAll(Boolean-expression)
  - collection->forAll(v | Boolean-expression-with-v)
  - collection->forAll(v: Type | Boolean-expression-with-v)
- Example: The age of each employee is less than or equal to 65
  - context Company
  - inv: self.employee->forAll(age <= 65)
  - inv: self.employee->forAll(p | p.age <= 65)
  - inv: self.employee->forAll(p: Person | p.age <= 65)
- More than one iterator can be used in the forAll operation
- All instances of persons have unique names
  - context Person inv:
  - Person.allInstances()->forAll(p1, p2 |p1 <> p2 implies p1.name <> p2.name)

forAll Example 2

- A person can have zero or more children, and a child has exactly two parents:
- It is expected that for a given parent, the parent is included in the set of parents of each of its children:




```
context Person
inv: self.children->forAll(c | c.parent->includes(self))
```

```
context Person
inv: self.parent->forAll(p | p.children->includes(self))
```

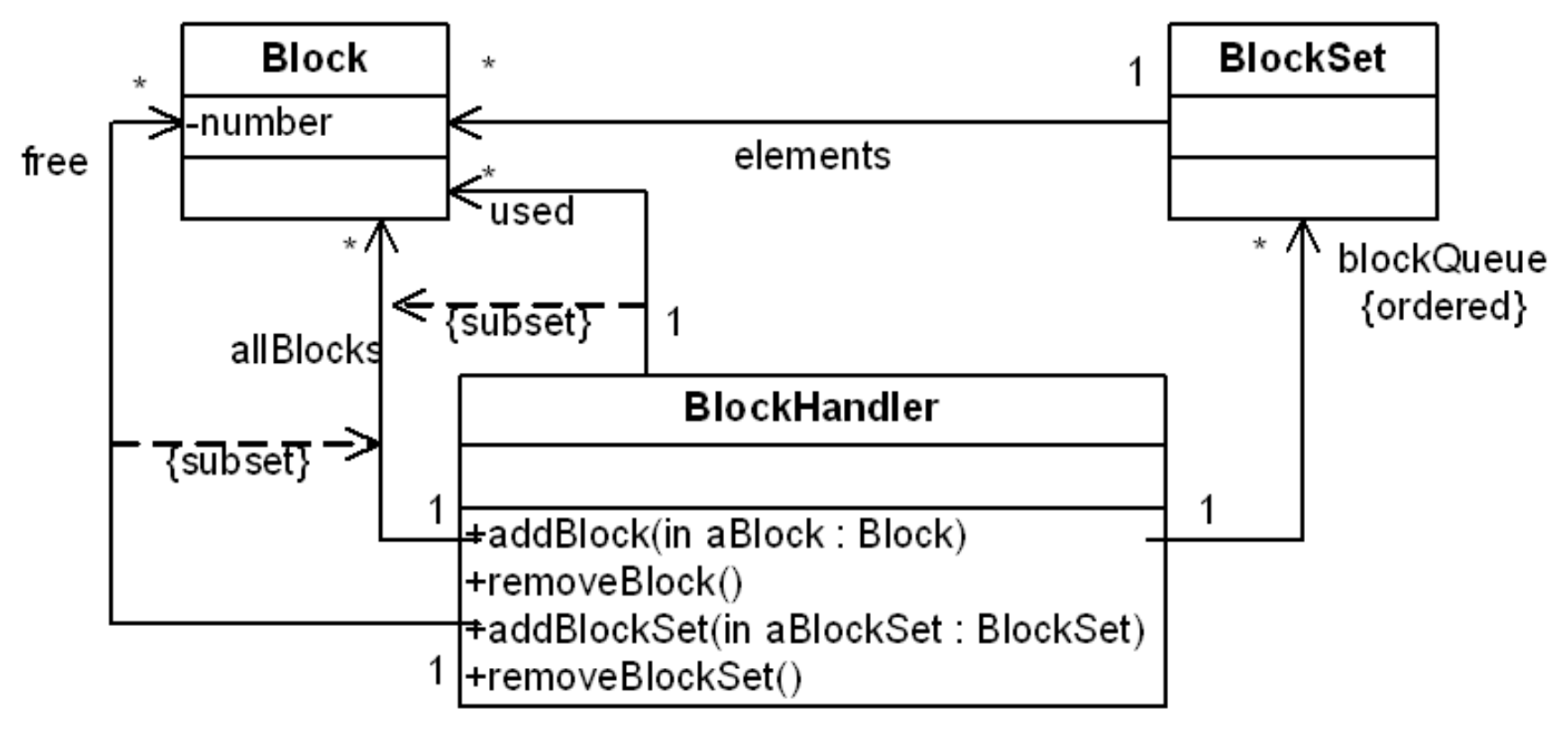
Exists Operation on a Collection

- Specifies a Boolean expression that must be true for at least one element in a collection.
- Alternative expressions for the exists operation:
  - collection->exists(Boolean-expression)
  - collection->exists(v | Boolean-expression-with-v)
  - collection->exists(v: Type | Boolean-expression-with-v)
- Example: The firstName of at least one employee is equal to `Jack'

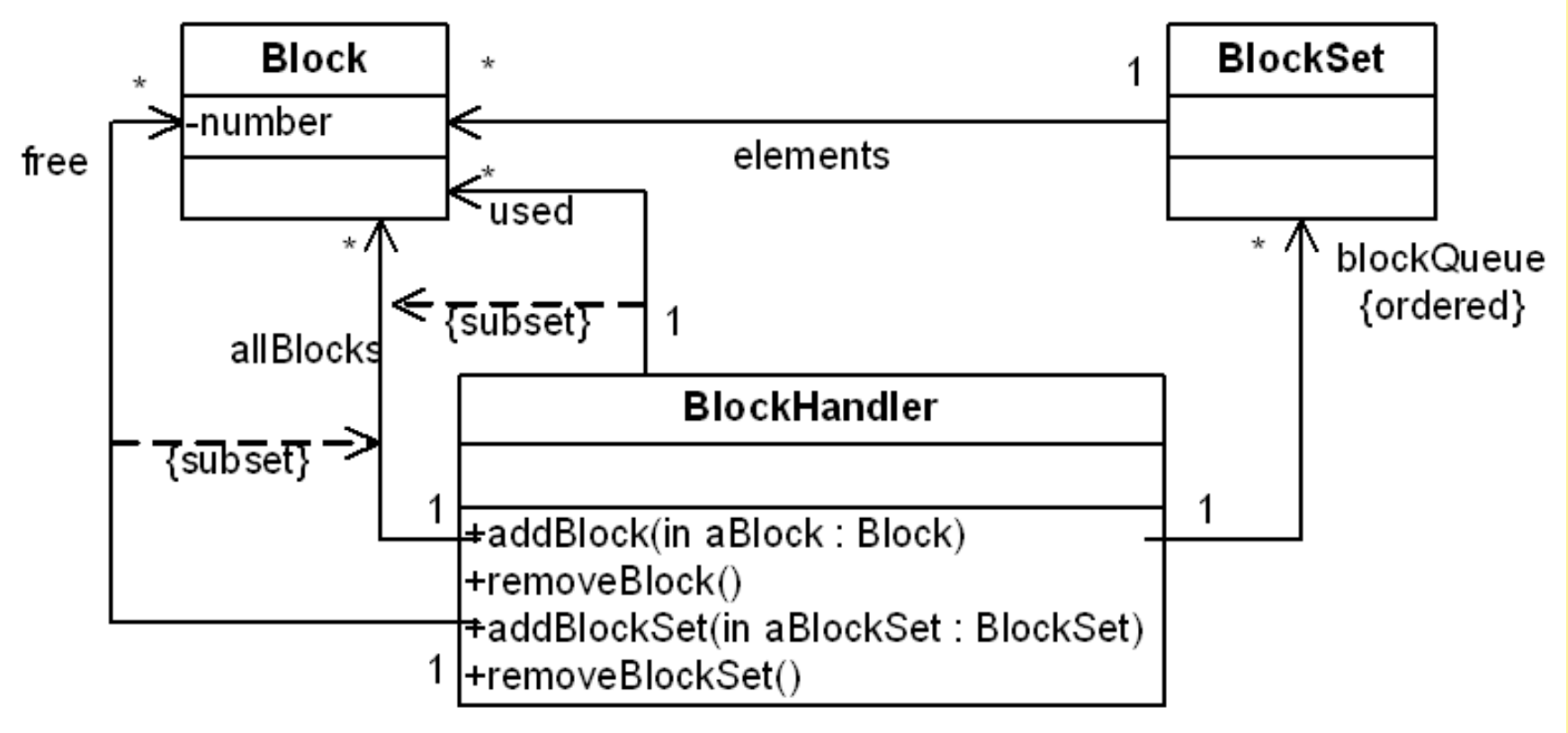
```
context Company
    inv: self.employee->exists(firstName = `Jack')
    inv: self.employee->exists(p | p.firstName = `Jack')
    inv: self.employee->exists(p: Person | p.firstName = `Jack')
```

Exercise 1

- Operation **removeBlockSet()** removes all blocks in the first element of **blockQueue** from the used blocks collection and adds them to the free blocks collection. Note that in order to remove a block set, the block queue should not be empty. Operation addBlockSet() adds a block set to the block queue and can add a block set if all its elements are currently used.



- No block will be marked as both unused and used



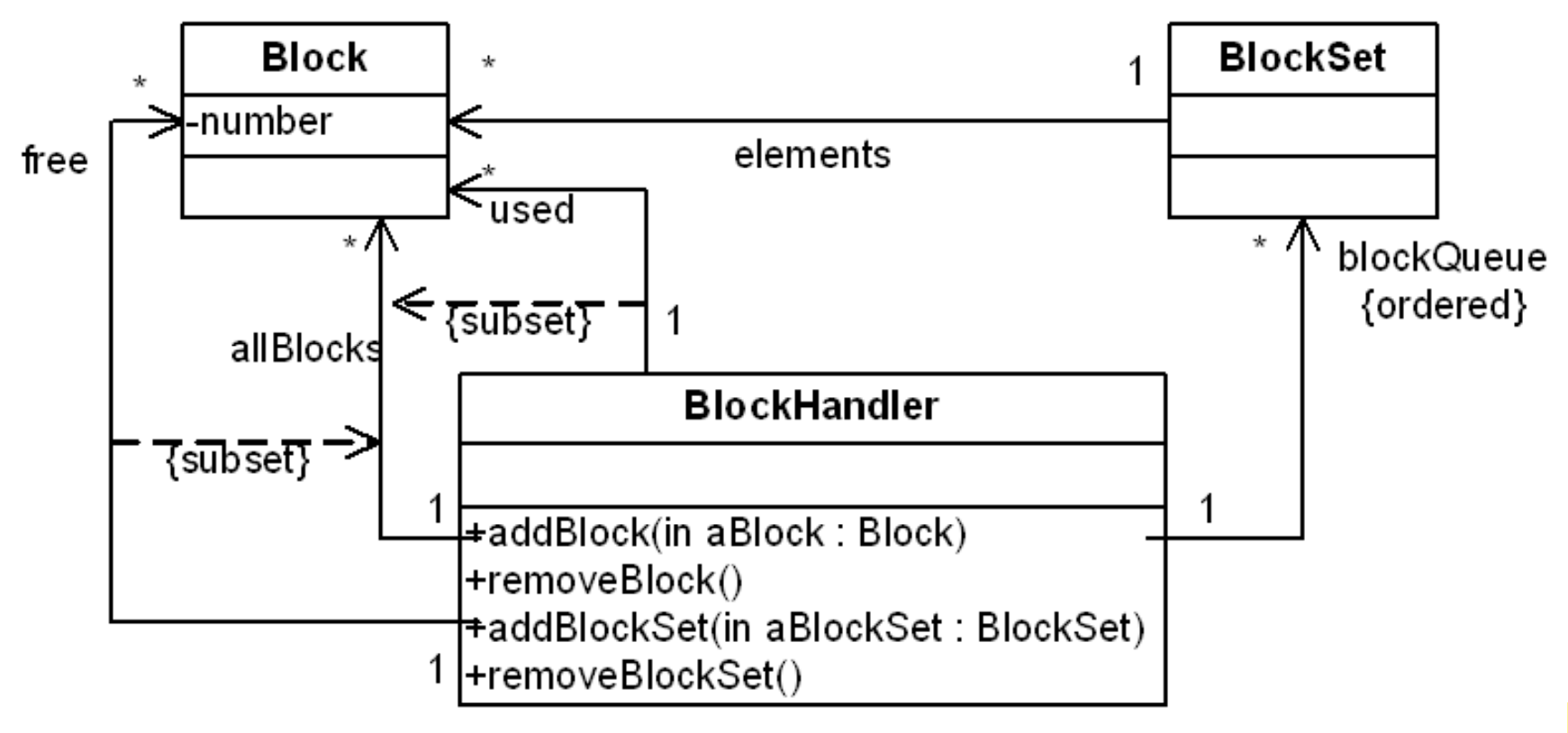
1.1 answer

No block will be marked as both unused and used

context BlockHandler **inv:**

(self.used->intersection(self.free)) ->isEmpty()

- All the sets of blocks held in the queue will be subsets of the collection of currently used blocks.



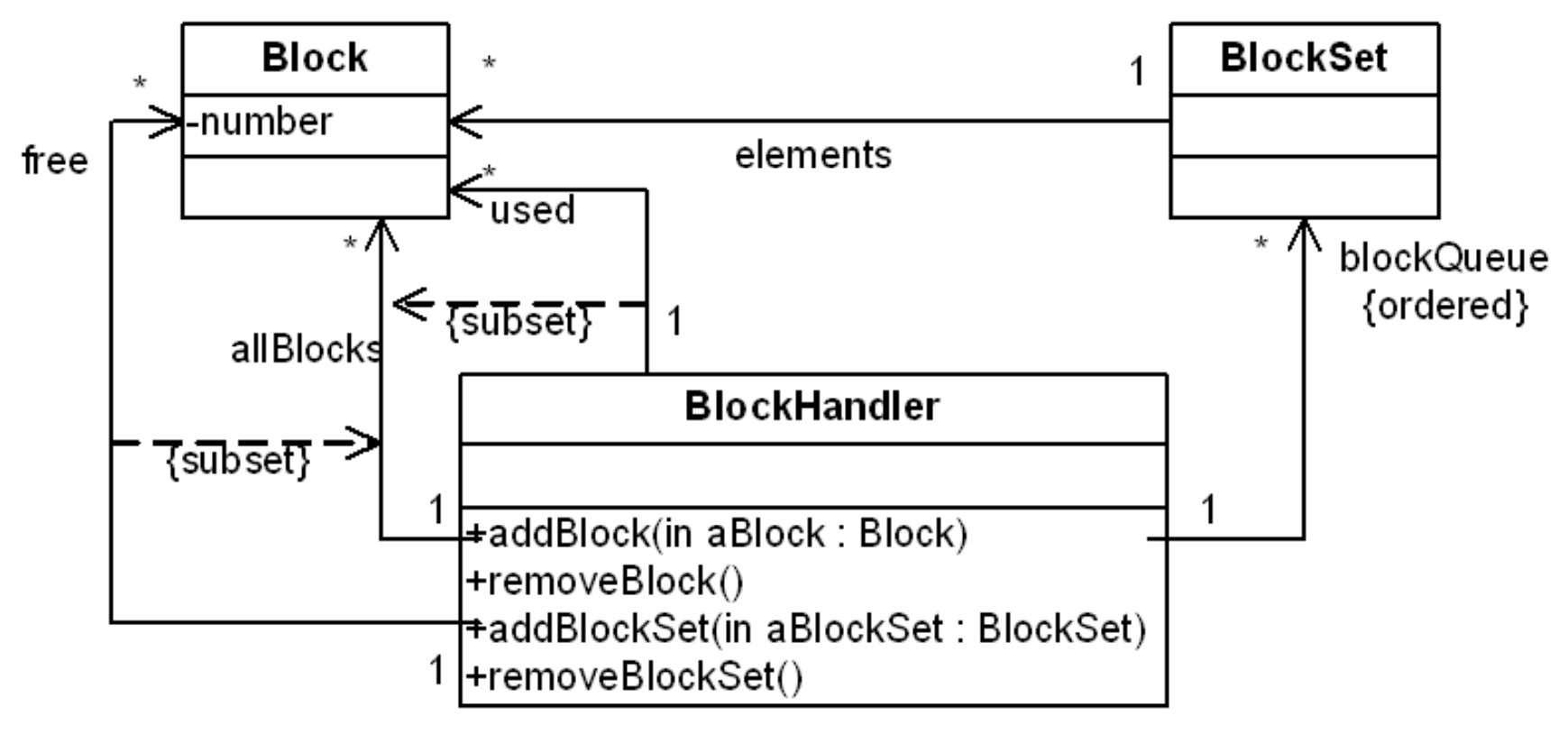
1.2 answer

- All the sets of blocks held in the queue will be subsets of the collection of currently used blocks.

  context BlockHandler **inv:**

  blockQueue->forAll(aBlockSet|used ->
  includesAll(aBlockSet.elements))

- No elements of the queue will contain the same block numbers



1.3 answer

- No elements of the queue will contain the same block numbers.

  context BlockHandler **inv**:

  blockQueue->forAll(blockSet1, blockSet2 |

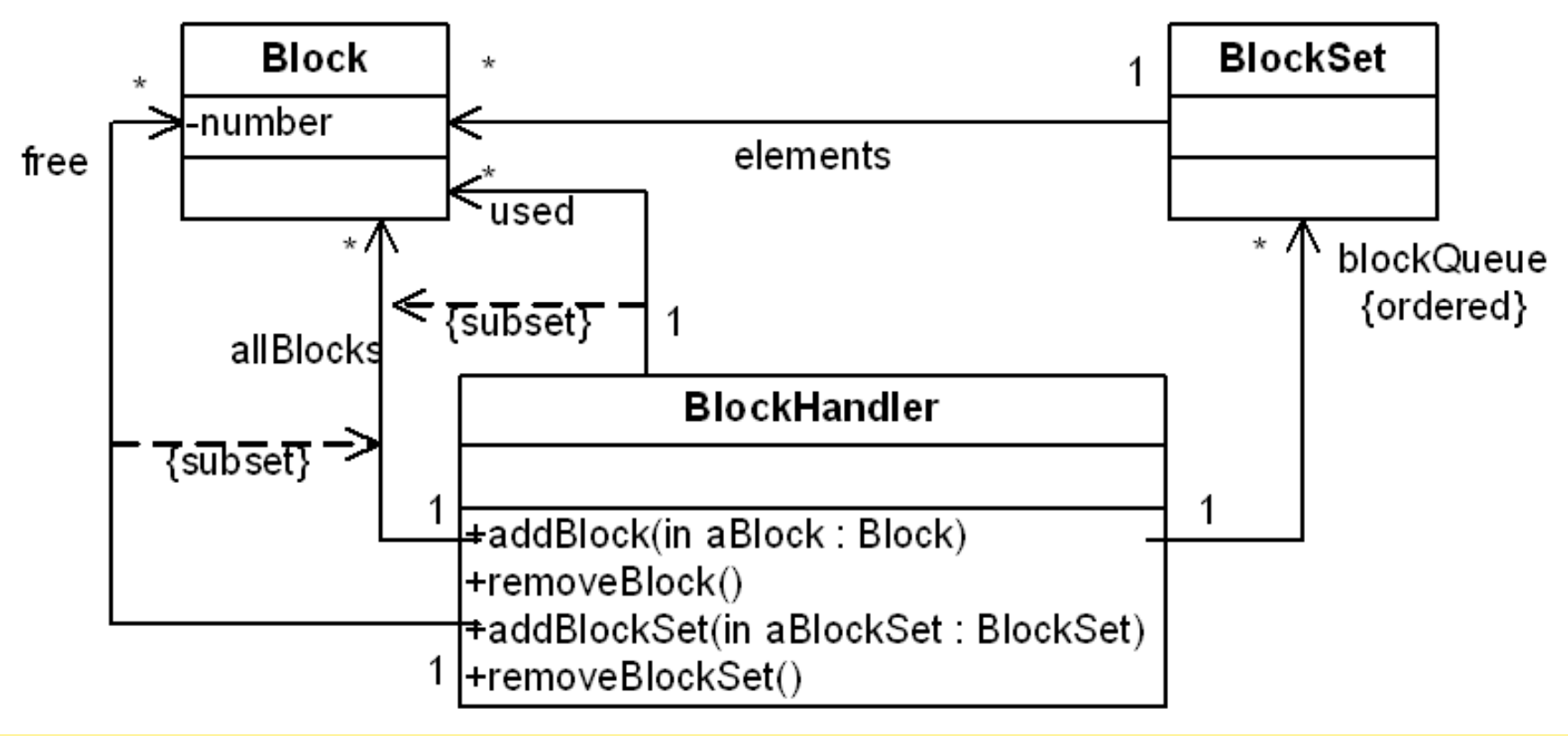
  blockSet1 <> blockSet2 implies

  blockSet1.elements.number->

  excludesAll(blockSet2.elements.number))

- The expression before implies is needed to ensure we ignore pairs where both elements are the same Block.

- The collection of used blocks and blocks that are unused will be the total collection of blocks that make up files.



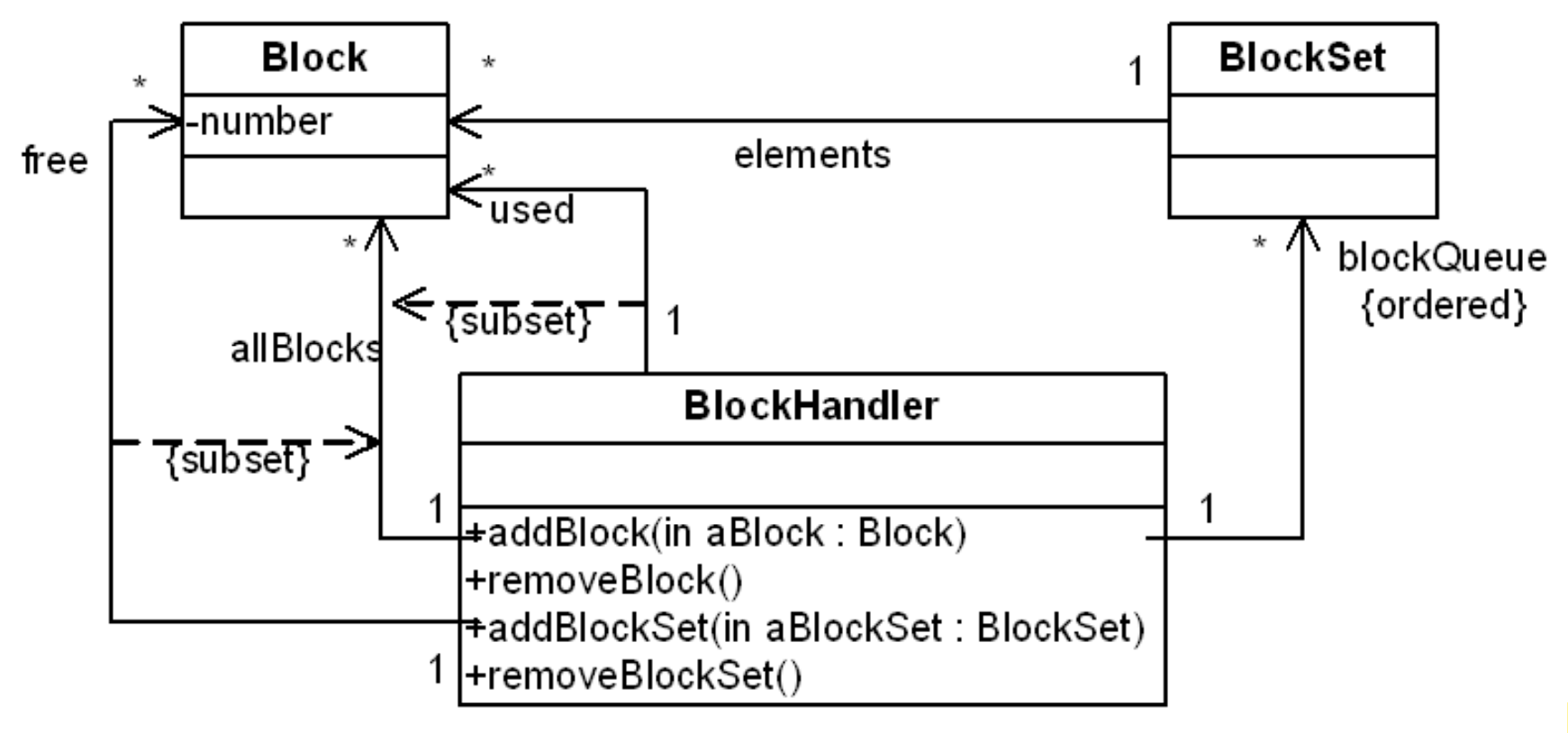
1.4 answer

- The collection of used blocks and blocks that are unused will be the total collection of blocks that make up files.

  context BlockHandler **inv**:

  allBlocks = used->union(free)

- The collection of unused blocks will have no duplicate block numbers.



1.5 answer

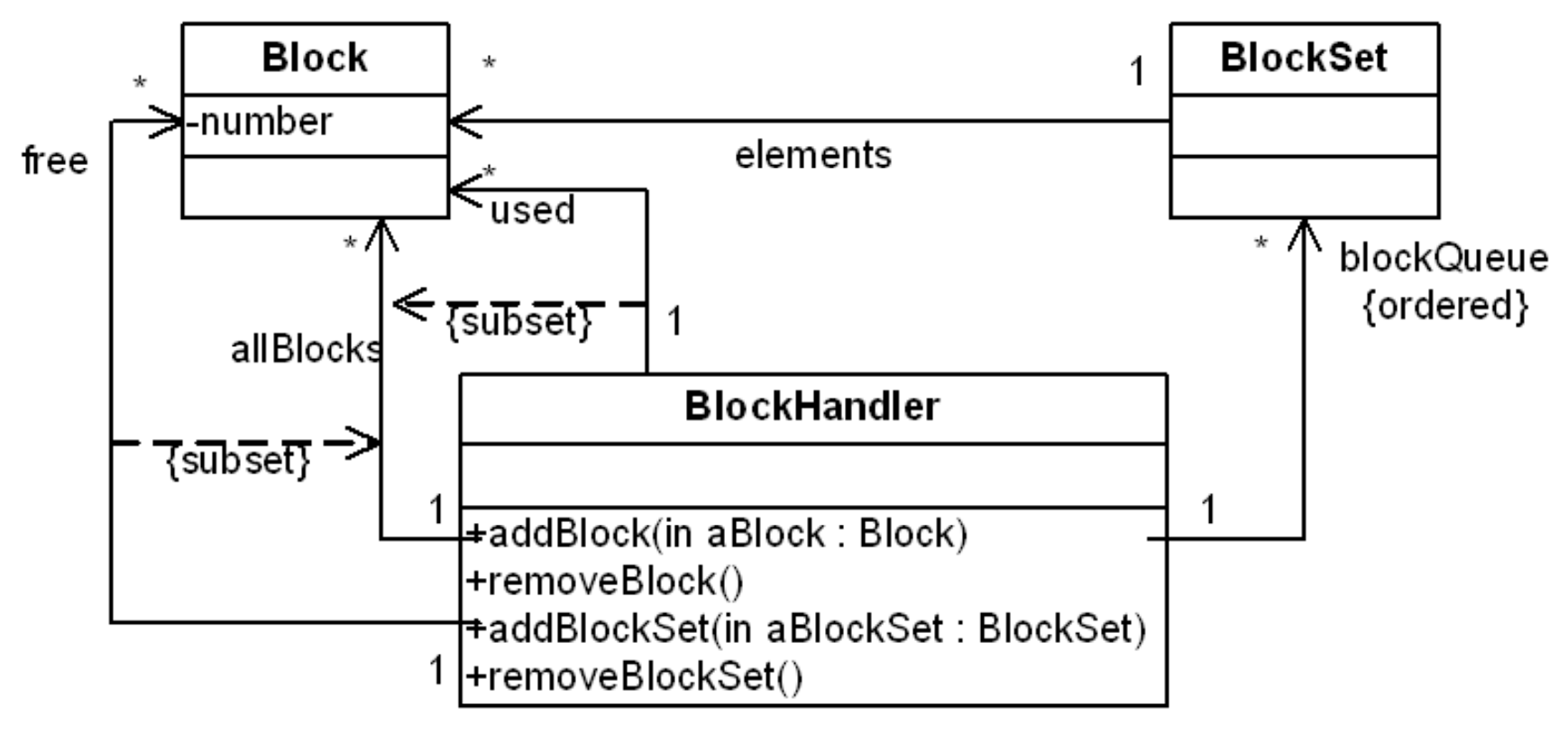
- The collection of unused blocks will have no duplicate block numbers.

  context BlockHandler **inv**:

  free->forAll(b1, b2 | b1 <> b2 implies

  b1.number <> b2.number)

- The collection of used blocks will have no duplicate block numbers.



1.6 answer

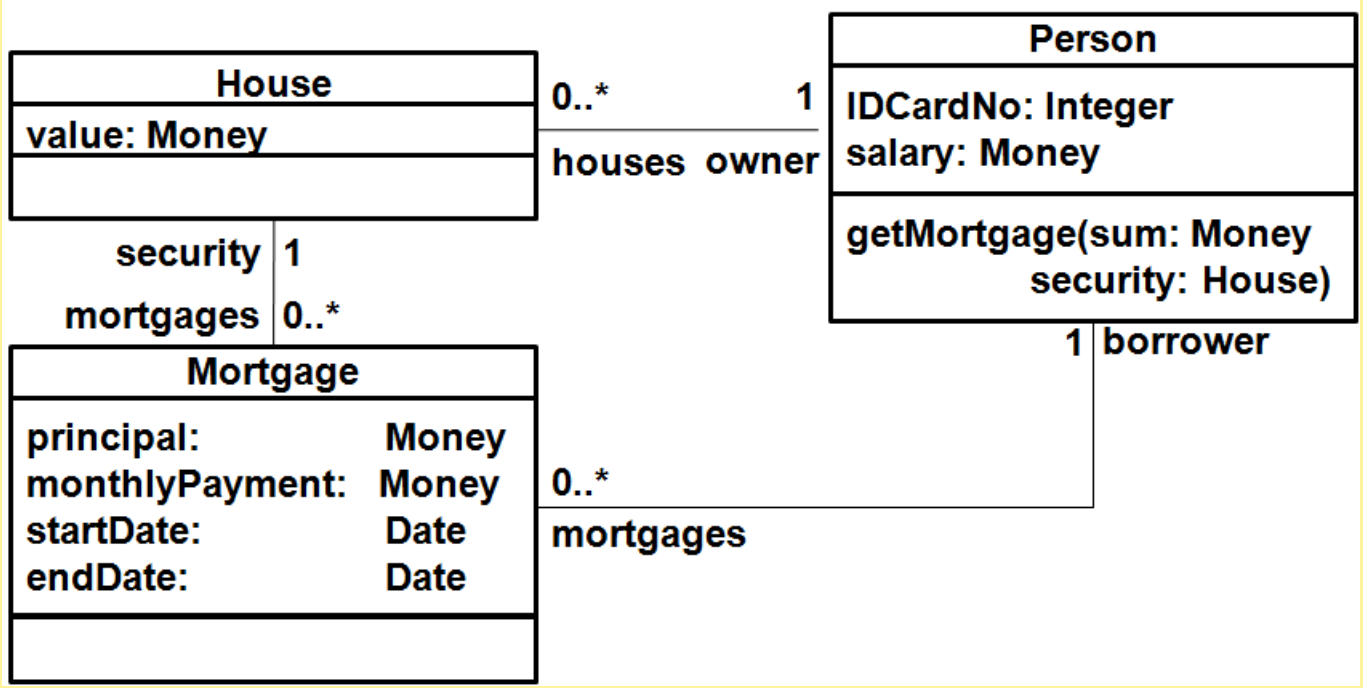
- The collection of used blocks will have no duplicate block numbers.

  context BlockHandler **inv**:

  used-> forAll(b1, b2 | b1 <> b2 implies b1.number <> b2.number)

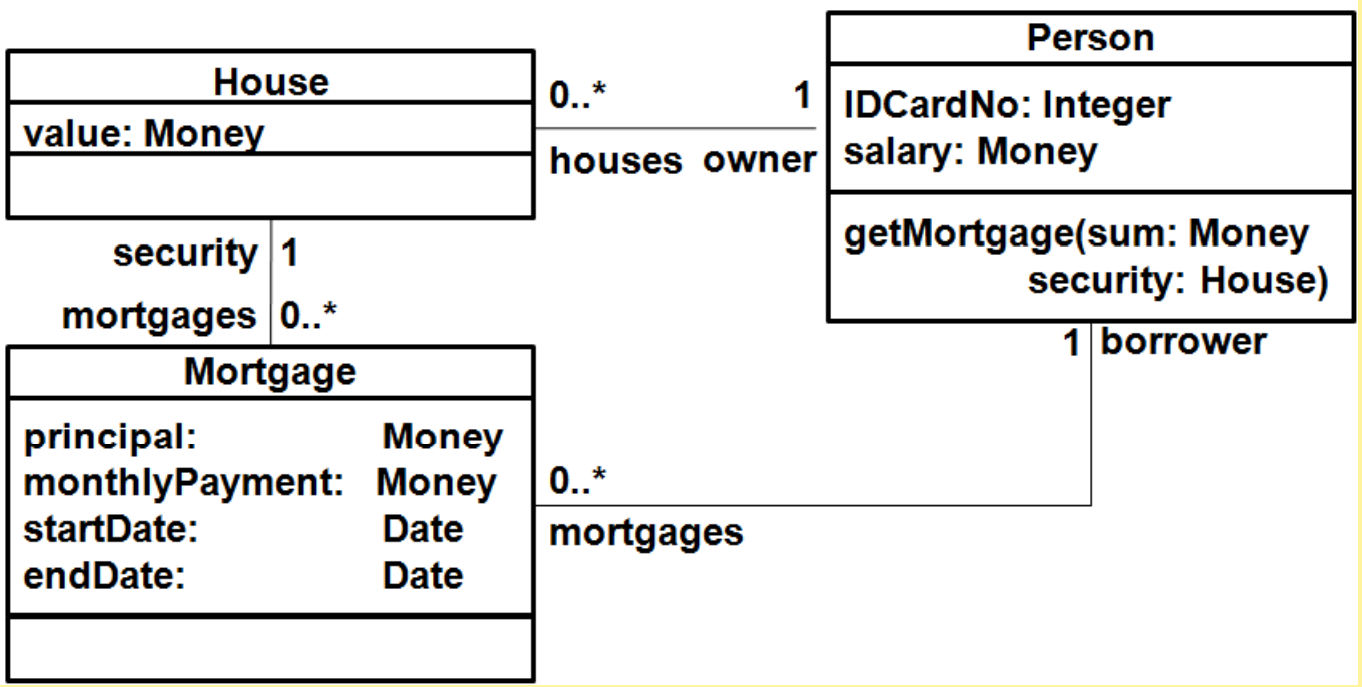
Case study 2

- A person may have a mortgage on a house only if that house is owned by him- or herself;
  - one cannot obtain a mortgage on the house of one's neighbor or friend.
- The start date for any mortgage must be before the end date.
- The ID card number of all persons must be unique.



Case study 2

- A new mortgage will be allowed only when the person's income is sufficient.
- A new mortgage will be allowed only when the counter-value of the house is sufficient.



Answers

context Mortgage
inv: security.owner = borrower

context Mortgage
inv: startDate < endDate

context Person
inv: Person::allInstances()->isUnique(socSecNr)

context Person::getMortgage(sum : Money, security : House)
pre: self.mortgages.monthlyPayment->sum() <= self.salary * 0.30

context Person::getMortgage(sum : Money, security : House)
pre: security.value >= security.mortgages.principal->sum()

Any Questions?