# OCL and Concept Model

Jörg Kienzle & Alfred Strohmeier

# Overview

- OCL
  - History and Goal
  - Constraints
  - OCL Types
  - Base Types & Operations
  - Collection Types & Operations
  - Navigating UML Diagrams
  - Conformance Rules
- Concept Model
  - Building the Concept Model
  - Actors and the Concept Model

# OCL History and Goal

- First version developed at IBM in 1995
- Standardized by OMG in November 1997 as part of the UML 1.1 standard
- Current version: OCL 2.4 (as part of UML 2.4)
- Language for specifying invariants, preconditions, postconditions and other constraints
- Easy to learn, easy to use, easy to understand
- Formal language for UML users

# Constraint Definition

- A constraint is a restriction on one or more values of (part of) an object-oriented model or system.
- Assertion on a model
- Static
  - Invariant
- Dynamic
  - Pre- / postconditions

# Constraint Advantages

- Better Documentation
  - Constraints add to visual models information about the model elements and their relationships
- Improved Precision
  - Constraints can be more detailed than visual models
  - Constraints can be verified for coherence or type-checked
- Communication without Misunderstanding
  - Constraints are unambiguous

# Declarative Language

- OCL constraints are declarative
  - Predicates / Assertions / Truths
- Constraints have no side effects
  - The system state does not change due to the evaluation of a constraint!
- Advantages
  - The modeller does not have to decide what happens when a constraint is broken (to be addressed in a later phase)
  - Constraint evaluations are atomic
  - Constraints can be evaluated as many times as necessary

# OCL Types

- Predefined standard types
  - Boolean
  - Integer
  - Real
  - String
- Predefined collection types
  - Collection
  - Set
  - Bag
  - Sequence
- User-defined types
  - All types from a UML model (the one that defines the context) can also be used.

# Value Types and Object Types

- Value Types
  - Their instances never change value
  - The value is the instance
  - Example: Integer “1”
  - All pre-defined OCL types are value types
- Object Types
  - Instances of classes
  - Instances are object, can change the value of their attributes
  - Example: the person “Jörg”
  - User-defined types are object types

# Boolean

- Predefined operations (order of precedence)
  - **not**
  - **if** … **then** … **else** … **endif**
  - **=**, **<>**
  - **or**, **and**, **xor**
  - **implies**

# Integer and Real

- Predefined operations (order of precedence)
  - **.abs**, **.max**, **.min**
  - For Integers: **.div**, **.mod**
  - For Reals: **.floor**, **.round**
  - **-** (unary minus)
  - *****, **/**
  - **+**, **-**
  - **<**, **>**, **<=**, **>=**
  - **=**, **<>**
- Mathematical definition
  - Integer is a subtype of real
  - There are no maximum integer / real values

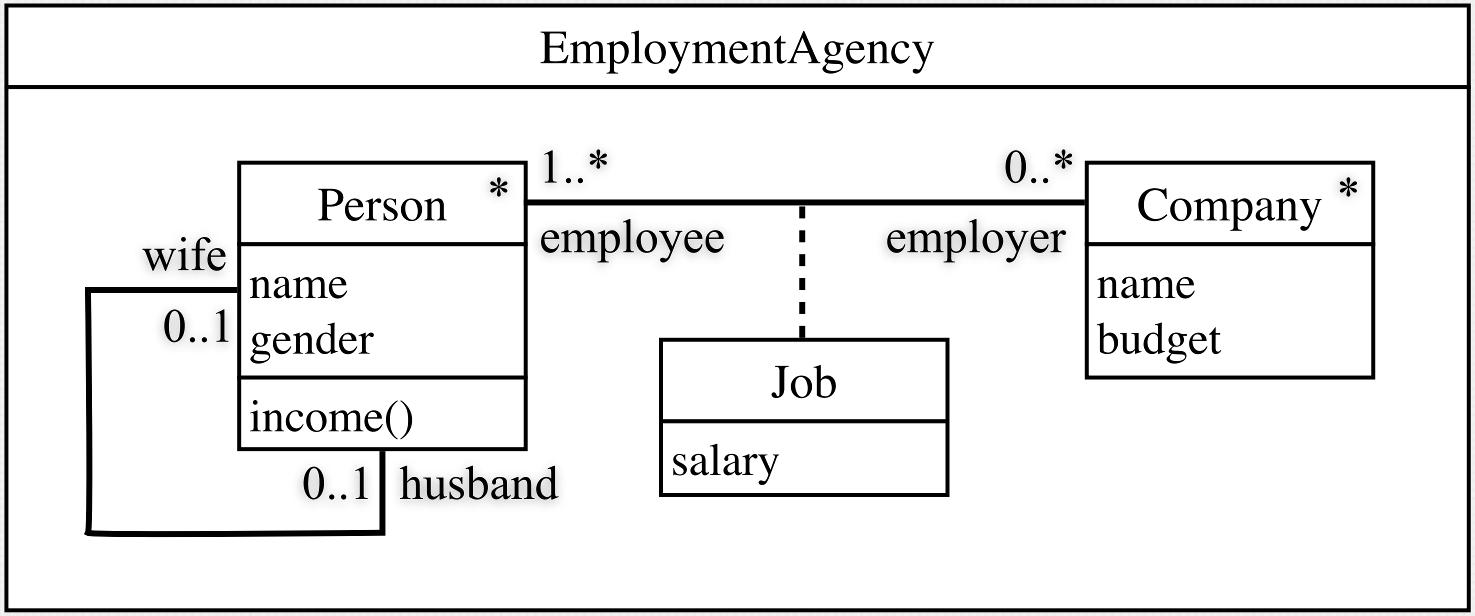
# String

- Sequence of characters
- Literals written in single quotes
  - ‘apple’, ‘macintosh’
- Predefined operations
  - **toUpper**
  - **toLower**
  - **size**
  - **substring**(int, int)
  - **concat**(String)
  - **=**, **<>**

# OCL Constraints

- An OCL constraint is a valid OCL expression of type Boolean
- OCL expressions can refer to types, classes, association classes, interfaces, and datatypes, and to properties of objects.
- A property is one of the following
  - an Attribute
  - an Operation with `isQuery` being true
    - `isQuery` is true for Observers (read-only operations)
    - This rule guarantees that OCL expressions have no side-effects
  - an AssociationEnd
- Properties can be accessed via the dot operator.
- OCL expressions are evaluated / read from left to right

# Example UML Class Diagram



# Context and self

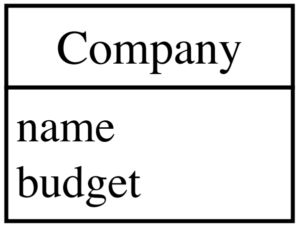
- OCL allows a developer to “navigate” through a UML diagram and express constraints.
- Each expression is written in the context of an instance of a specific type. The reserved word **self** refers to this instance.
- Example

**context** Company

    **inv**: **self**.budget > 50

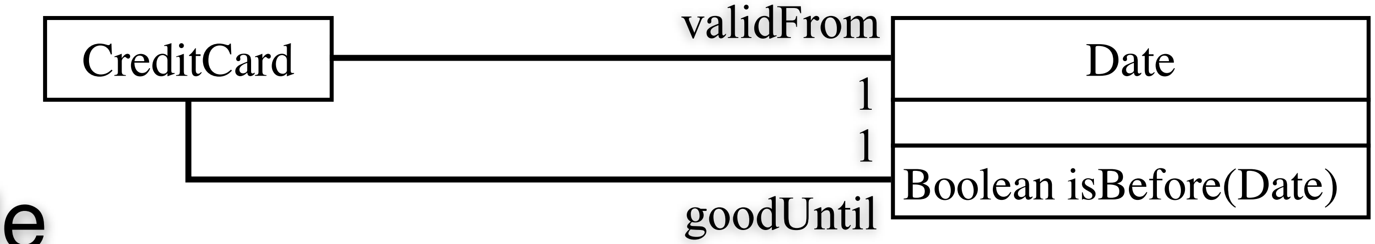
**context** c: Company

    **inv**: c.budget > 50



# Using Operations without Side-Effects

- You can use operations of classes in OCL expressions, provided that they have no side-effects

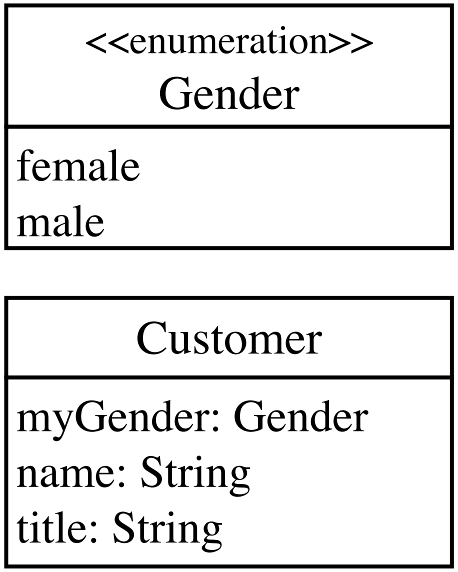


- Example
  **context** CreditCard
      **inv**: **self**.validFrom.isBefore(**self**.goodUntil)

# Enumeration Type

- Defined by UML class with stereotype `<<enumeration>>`
  - Literals: `class::value`
- Predefined operations
  - =, <>
- Example

```
context Customer
      inv: self.myGender = Gender::male
            implies self.title = 'Mr.'
```



# OCL Collections

- Collection is the abstract supertype of all collection types in OCL: Set, Bag and Sequence.
- A Set is a mathematical set. It does not contain duplicate elements.
- A Bag is like a set, but may contain duplicates (i.e., the same element may be in a bag twice or more).
- A Sequence is like a bag in which the elements are ordered. Both bags and sets have no order defined on them.
- Operations on collections are applied using the "→" operator

# Collection Examples

- Sets
  - Set {1, 2, 5, 88}
  - Set {‘apple’, ‘orange’, ‘strawberry’}
- Sequences
  - Sequence {1, 3, 45, 2, 3}
  - Sequence {1..10}
- Two identical bags
  - Bag {1, 3, 45, 2, 3}
  - Bag {1, 2, 3, 3, 45}

# Navigating UML Diagrams (1)

- It is possible to navigate over associations by using the association role name, if it exists, or the corresponding class name in lowercase letters
- If the multiplicity of the association end is 0..1 then the expression results in an object (or oclVoid). However, such an expression can be treated like it results in a set as well.
- Example

  **context** Person

  ```
      inv: self.wife→notEmpty() implies
        self.wife.gender = Gender::female
  ```

# Operations on Collections

- A collection operation never changes the collection
  - but results in a new one!
- Operations
  - Integer **size**()
  - Boolean **isEmpty**()
  - Boolean **notEmpty**()
  - Integer **count**(Object o) : the number of occurrences of object o in the collection
  - Boolean **includes**(Object o) : true if object o is an element of the collection
  - Boolean **includesAll**(Collection c) :
    - true if collection c is a subset of the current collection
  - OCLAny **any**(boolean expression e):
    - selects one object that satisfies the expression e at random
  - SummableType **sum**() :
    - calculates the sum of all elements (integer / real) in the collection
  - Sequence **sortedBy**(expression) :
    - produces a sorted sequence containing the elements of the original set

# Collection Operation Examples (1)

- **context** Person
  **inv**: **self**.wife→**notEmpty**() **implies**
  **self**.wife.age ≥ 18
  **and self**.husband→**notEmpty**() **implies**
  **self**.husband.age ≥ 18
- **context** Company
  **inv**: **self**.employee→**size**() ≤ 50

# Operations with Variant Meaning

- **=**
  - True if all elements in the two collections are the same
  - For two bags, the number of times an element is present must also be the same
  - For two sequences, the order of elements must also be the same
- Collection **union**(Collection)
  - Works also for combining bags and sequences
- Collection **intersection**(Collection)
  - Works also for intersecting bags and sequences
- **including**(Object o)
  - Returns new collection including object o
  - For sets, the object is only added if it is not present in the set already
  - For sequences, the object is added at the end
- **excluding**(Object o)
  - Returns new collection where all occurrences of object o were removed

# Operations on Sets

- Set **minus**(Set s)
  - The resulting set contains all elements that are in the current, but not in the set s
- Set **symmetricDifference**(Set s)
  - The resulting set contains all elements that are in the current or the set s, but not in both

# Operations on Sequences

- Object **first**(), Object **last**()
  - Returns the first, rsp. last element of the sequence
- Object **at**(Integer)
  - Returns the element at the desired position
- Sequence **append**(Object o),
  Sequence **prepend**(Object o)
  - Return a new sequence where object o was added as the last, rsp. first element of the sequence

# Iterate Over Collections

- Boolean **exists**(boolean expression) :
  true if expression is true for at least one element of the collection
- Boolean **one**(boolean expression) :
  true if expression is true for exactly one element of the collection
- Set **select**(boolean expression) :
  pick all objects for which the expression evaluates to true
- Set **reject**(boolean expression) :
  exclude all objects for which the expression evaluates to true
- Bag/Sequence **collect**(expression) :
  computes expression for each element, and puts all results in a bag (or in a sequence, if applied to a sequence)
- Boolean **isUnique**(expression) :
  true if expression is unique for each element
- Boolean **forAll**(boolean expression) :
  true if for all elements in the collection, expression is true
- OCLAny **iterate**(expression) :
  expression is evaluated for every element of the collection. The result depends on the expression

# Collection Operation Examples (2)

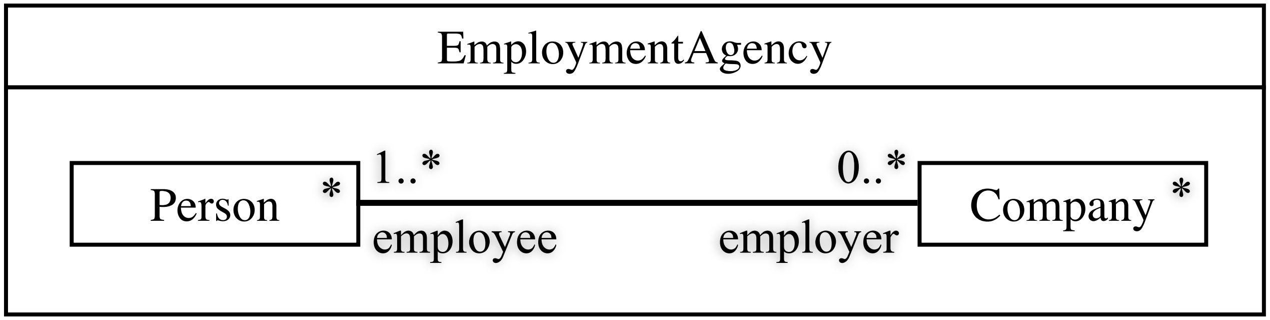
- **context**: EmploymentAgency
  - **self**.person→**collect**(name)
    - (shorthand: self.person.name)
  - **self**.company→**select**(c: Company | c.budget > 100,000);
  - **self**.person→**select**
    (p: Person ѣ   p.name = ‘Jörg’ **and**
    p.employer.name = ‘McGill’);

# Automatic Flattening

- Collections are automatically flattened
  - A collection never contains collections, only simple objects
- Example of identical sets
  - Set { Set {1, 2}, Set {3, 4}, Set {5, 6} } =
  - Set {1, 2, 3, 4, 5, 6}
- Converting between collections
  - **asSet**() : transforms the collection into a set
  - **asBag**() : transforms the collection into a bag
  - **asSequence**() :  transforms the collection into a sequence

# Navigating UML Diagrams (2)

- Single navigation results in a Set, combined navigations in a Bag, and navigation over associations annotated with {ordered} results in a Sequence.



- Example

**context** EmploymentAgency
  **inv** : **self**.person.employer→**size**()    **self**.company→size()

What goes here?
=, <>, <, >?

# Navigating Association Classes

- Navigation from an object to association class instances also uses the dot-operator
  - p.job
  - The above expression evaluates to all the jobs a person p has with the companies that are his/her employer. Note that the name of the association class, in lowercase, is used to show the role for navigation.
- It is also possible to start navigating from the association class to linked objects:

**context** Job

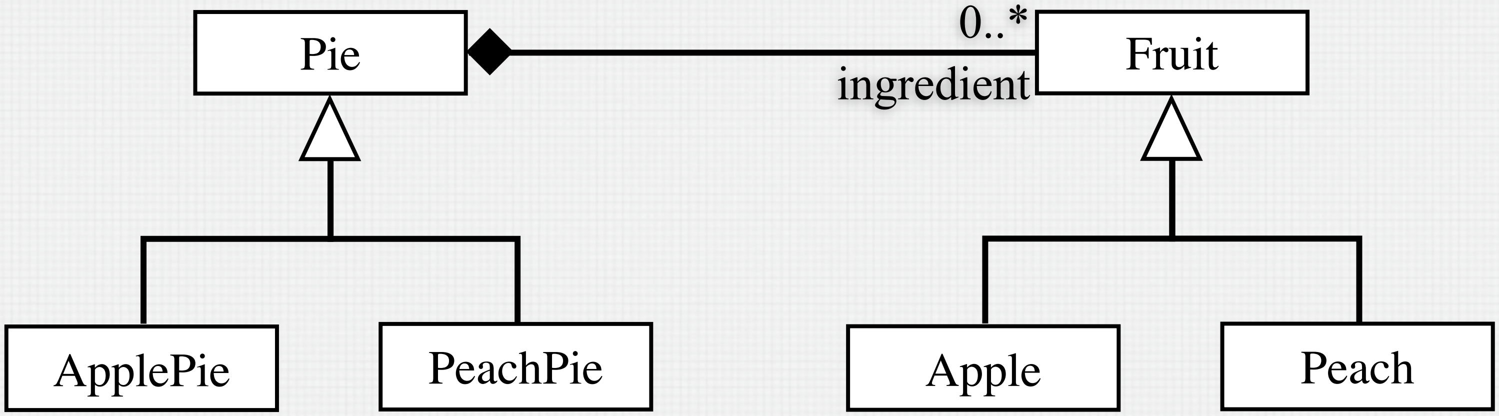
    **self**.employer.budget ...

    **self**.employee.age ...

# Operations on Every OCL Type

- **OclAny** is the supertype of all OCL types
- Predefined operations
  - **=**
  - **<>**
  - **oclType()** : get the OclType of the object
    - It is then possible to access the meta-level of OCL, e.g. query the name, attributes, associationEnds, operations, supertypes, instances of the type
  - **oclIsKindOf**(OclType t) :
    - true if the object is of type t or a subclass
  - **oclIsTypeOf**(OclType t) :
    - true if the object is of type t
  - **oclAsType**(OclType t) :
    - typecast the object to OclType t
  - **oclIsUndefined**() :
    - returns true if the argument is undefined
- Operations on (user-defined) classes
  - **.allInstances**() : get the set of all the objects of a given class

# IsKindOf Example



**context** ApplePie
  **inv**: **self**.ingredient→**forAll**(**oclIsKindOf**(Apple))

# OCL Operator Precedence

- **@pre**
- dot and arrow operations: ‘**.**’ and ‘→’
- unary **not** and unary ‘**-**’
- ‘*****’ and ‘**/**’
- ‘**+**’ and binary‘**-**’
- **if** ... **then** ... **else** ... **endif**
- ‘**<**’, ‘**>**’, ‘**≤**’, ‘**≥**’
- ‘**=**’, ‘**<>**’
- **and**, **or** and **xor**
- **implies**

# Conformance Rules

- OCL is a typed language
  - **OclAny** is the supertype of all OCL types
- Conformance rule
  - Type1 conforms to Type2 if an instance of Type1 can be substituted at each place where an instance of Type2 is expected.
  - Works for identical types, subtypes
    - Integer is a subtype of Real
- **Collection**(Type1) conforms to **Collection**(Type2), iff Type1 conforms to Type2
- Sequences, Bags, and Sets do not conform to each other
- Example
  - **Set**(ApplePie) conforms to **Collection**(ApplePie)
  - **Bag**(ApplePie) does not conform to **Set**(ApplePie)

# Errors

- If the type matching rules are not satisfied, an OCL expression is erroneous, e.g. the parser will output a conformance error
- Even if the type matching rules are satisfied, there can be problems in the following cases
  - There is a contradiction
    e.g. the a constraint states self.age = 33 and self.age < 18
  - An expression can yield an undefined value

# Undefined Expressions

- Some expressions, when evaluated, have an undefined value
- Examples
  - Getting the first element of an empty collection
  - An expression that dereferences an empty set and the expected result is not a set. This case also includes dereferencing after navigation to an association end of multiplicity 0..1 when there is no object having the role.
- There is an implicit "undefined value" for all types
  - **oclIsUndefined**() is an operation on clanky that returns true if the argument is undefined
- An expression where one of the parts is undefined is undefined
  - Exceptions
    - True **or undefined**
    - False **and undefined**
    - False **implies undefined**
    - **Undefined implies** True

# Let Expression

- Sometimes a sub-expression is used more than once in an expression
  - Declare a typed “Variable”

```
context Person
inv : let income : Integer = self.job.salary→sum() in
        if self.isUnemployed() then
           income < 100
        else
           income >= 100
        endif
```

# OCL Functions

- A function may be used to encapsulate a calculation.
- Functions have no side effects

  “**context**” Class:: FunctionName
  “(“ [ParameterList] “)” “:” TypeName
  “**post**:” result = Expression “;”

- or

  “**context**” Classname
  “**def**:” FunctionName “(“ [ParameterList] “)” “:” TypeName = Expression “;”

- The expression must evaluate to a value of type TypeName

# OCL Function Example (1)

- A function that counts the number of female employees for a given company

```
context Company::countFemEmployees() : Integer
post:
   result = self.employee→select
         (p | p.gender = Gender::female)→size()
```

Reserved word to refer to the result of the operation

# OCL Function Example (2)

- A function that calculates the sum of all the salaries paid to employees

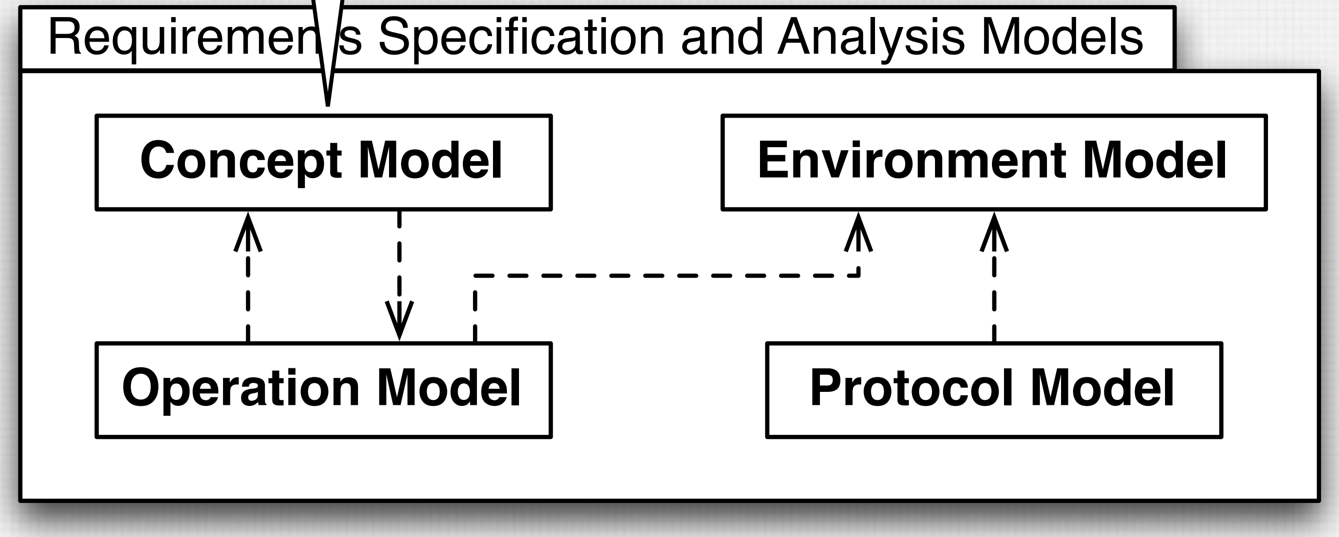
```
context EmploymentAgency
def sumOfSalaries(c : Company) : Real
    = c.job→collect(salary)→sum()
```

# RE SPECIFICATION PHASE REMINDER

- Purpose
  - To produce a complete, consistent, and unambiguous description of
    - the problem domain and
    - the functional requirements of the system.
- Models are produced, which describe
  - Structural Properties
    - Environment Model
      - Defines the system's interface, i.e. the boundaries of the system and the operations that can be performed on the system.
    - Concept Model
      - Defines the static structure of the information in the system, i.e. the concepts hat exist in the system, and the relationships between them
  - Behavioural Properties (see next lecture)
  - The models concentrate on describing what a system does, rather than how it does it.

# Fondue Models: Requirements Spec.

UML Class Diagram, describing the conceptual state of the system



# Requirements Specification Process

- From Requirements Elicitation:
  - Use Case Model
  - Domain Model

1. Develop the Environment Model
   - Identify actors, messages and system operations
2. Produce the Concept Model
   - By adding the system boundary to the Domain Model
3. Develop the Protocol Model (next lecture)
4. Develop the Operation Model (next lecture)
   - Update Concept Model if needed
5. Check the requirements models for consistency and completeness

# Concept Model (1)

- The Concept Model contains the set of classes and associations modelling the system's conceptual state
  - Classes and relationships from the Domain Model can specify concepts belonging to the system itself, as well as to its environment.
  - The Concept Model only contains the classes and relationships of the Domain Model that relate to the system to be built.
- The Concept Model is built by delimiting in the Domain Model what is inside of the system from what is outside of it. The Concept Model is therefore formed by excluding all the objects, classes and relationships that belong exclusively to the environment.

# Concept Model (2)

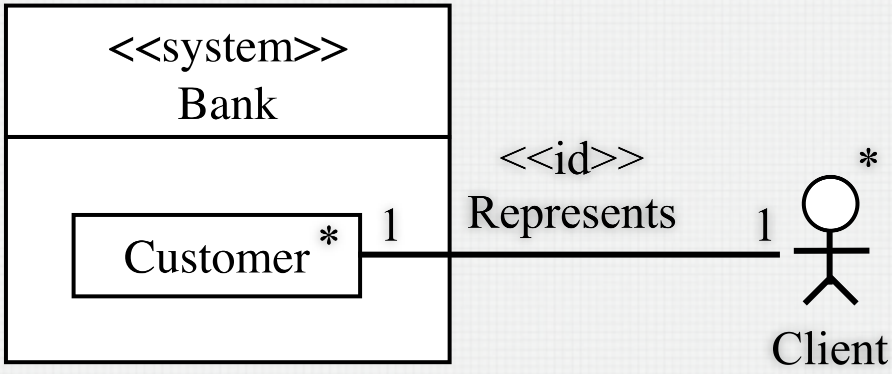
- Actors, for example, belong to the environment. Associations connecting them to the system correspond to communication paths between actors and the system.
  - When a class is outside of the system boundary, and if its instances interact with the system, then these instances are in fact actors.
- If an association is in the Concept Model, then all connected classes must also be in the model (well-formed class model)
  - Hence actors that interact with the system directly are included in the Concept Model
- If everything is in the system, then the system is a simulation model (no interaction with the environment).

# Concept Model (3)

- The Concept Model is a class diagram, where the system is shown explicitly as a composite class (stereotyped <<system>>) with a single instance, using graphical nesting of all the entities belonging to the system. As a consequence, all classes in the system get an explicit multiplicity that shows the number of instances within the system.
- Actors are modelled like classes belonging to the environment, together with their multiplicity in the environment, as viewed by the system.
    - Associations (often unnamed) depict the flows of messages between the system and the actors.

# Concept Model (4)

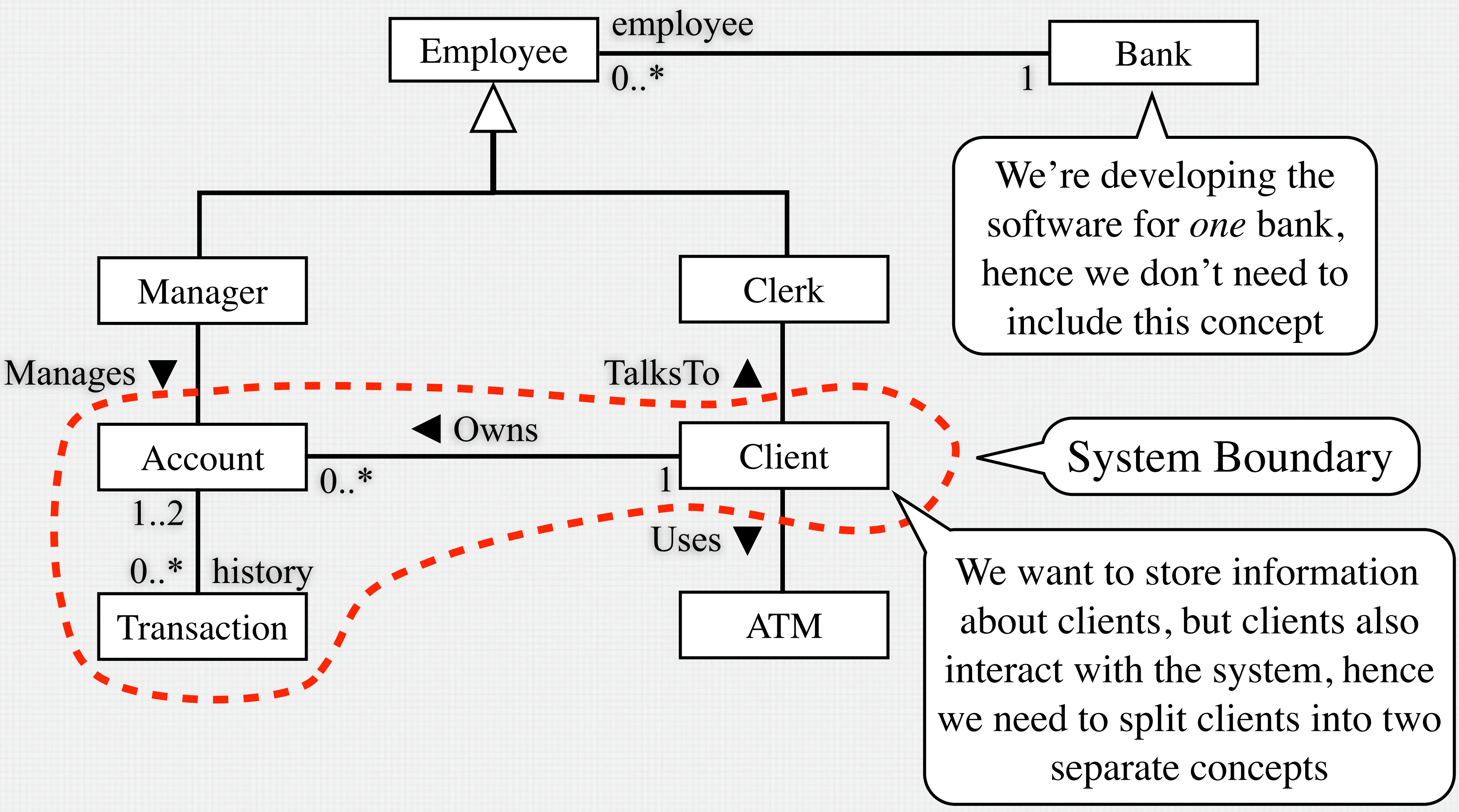
- Actors may be mirrored in the system by classes; this happens especially when the system needs to record state information about the actor.
    - In order to send a message to an actor, for example, it is often necessary to identify the actor using its representation in the system. Fondue defines an association stereotype <<id>> that can be used, and only used, to connect a class instance to an actor instance for this purpose. The multiplicity of an <<id>> association is exactly 1 for the role of the actor.



# Concept Model (5)

- The Concept Model must contain all conceptual system state needed in order to provide the required system functionality
  - Often, new concepts need to be added to the Concept Model once the behavioural specification models (Operation Model) are being developed
  - The Concept Model should not contain state that is not needed to provide the required system functionality

# Concept Model Example (1)

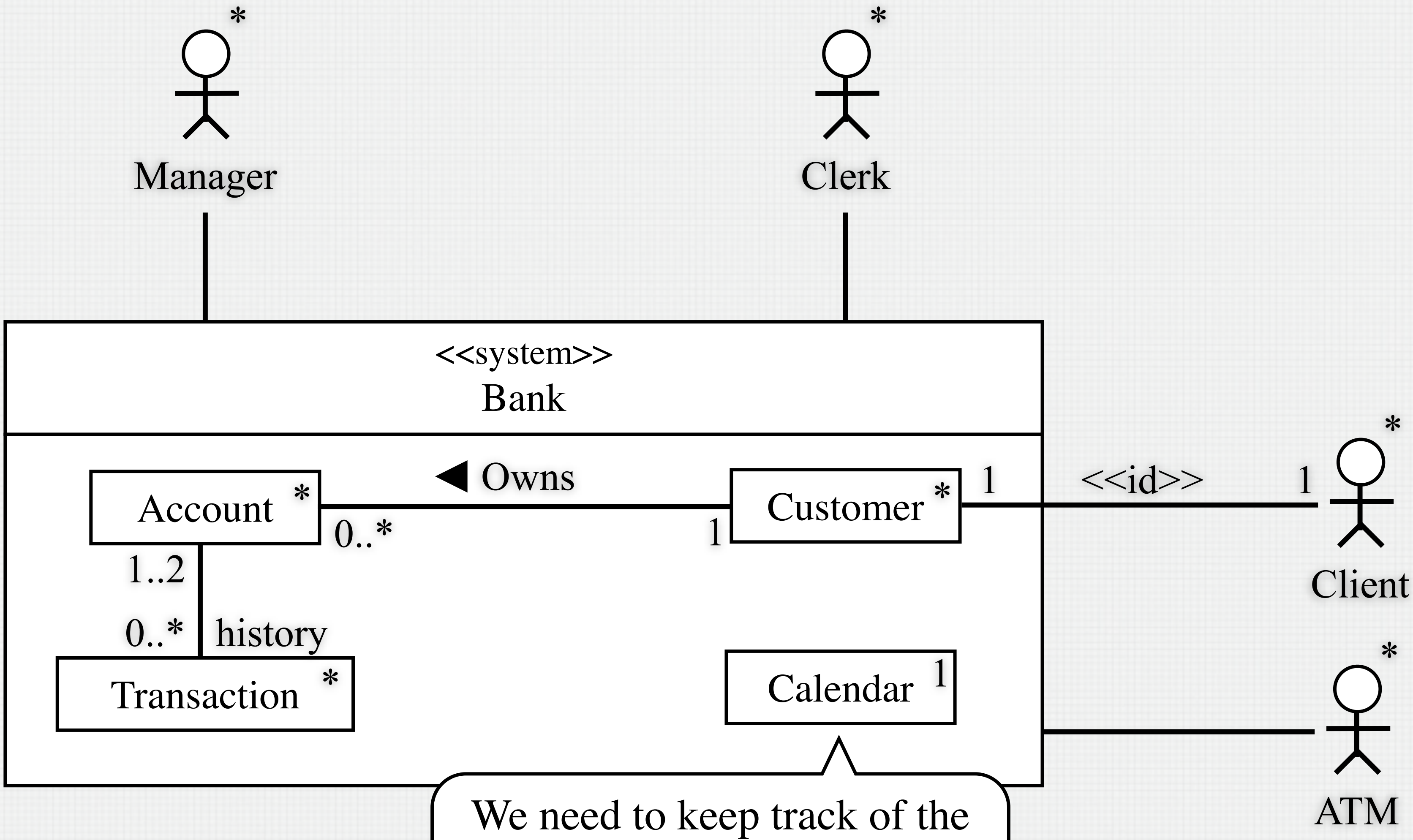


We're developing the software for *one* bank, hence we don't need to include this concept

System Boundary

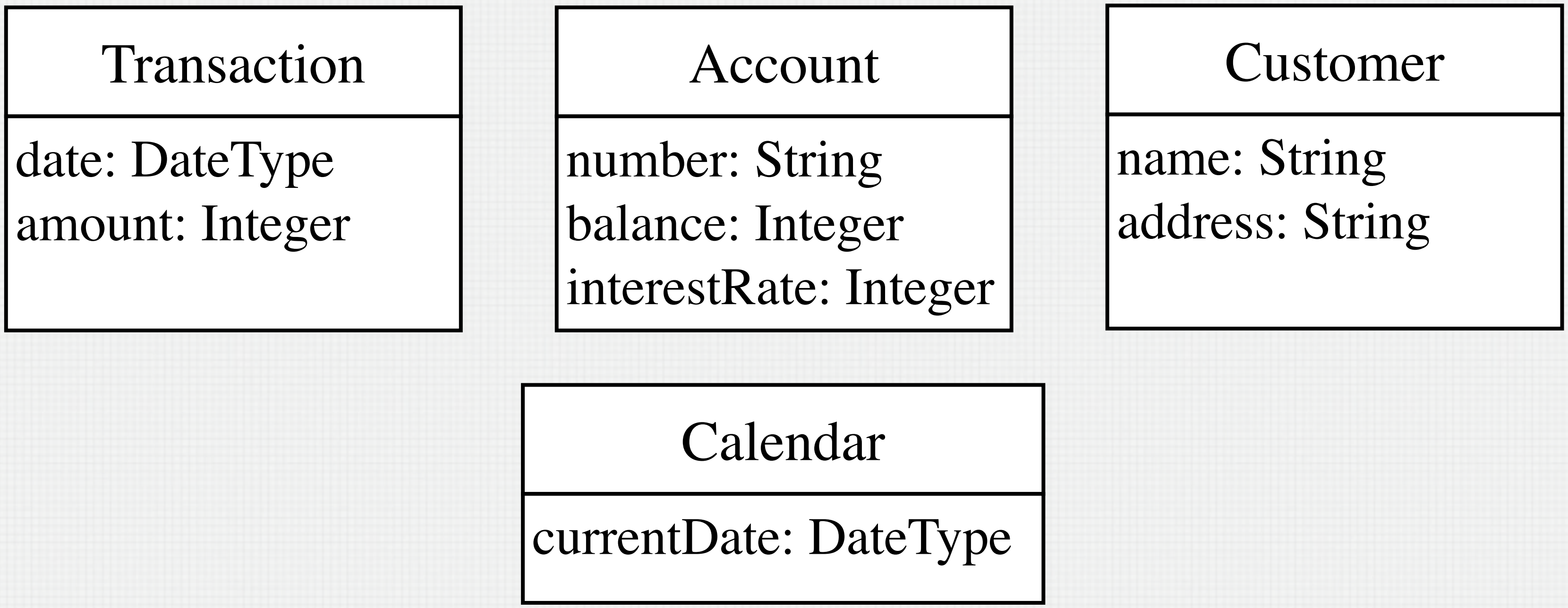
We want to store information about clients, but clients also interact with the system, hence we need to split clients into two separate concepts

# Concept Model Example (2)



We need to keep track of the current date in order to correctly initialize the date fields of the Transaction class

# Concept Model Example (3)



Transaction

date: DateType
amount: Integer

Account

number: String
balance: Integer
interestRate: Integer

Customer

name: String
address: String

Calendar

currentDate: DateType

+ OCL Constraints, if needed

# Questions?

# References

- [1] Jos Warmer and Anneke Kleppe: The Object Constraint Language - Precise Modeling with UML. Object Technology Series, Addison Wesley, 1999. ISBN 0-201-37940-6
- [2] Jos Warmer and Anneke Kleppe: The Object Constraint Language, Second Edition - Getting Your Models Ready for MDA. Object Technology Series, Addison Wesley, 2003. ISBN 0-321-17936-6

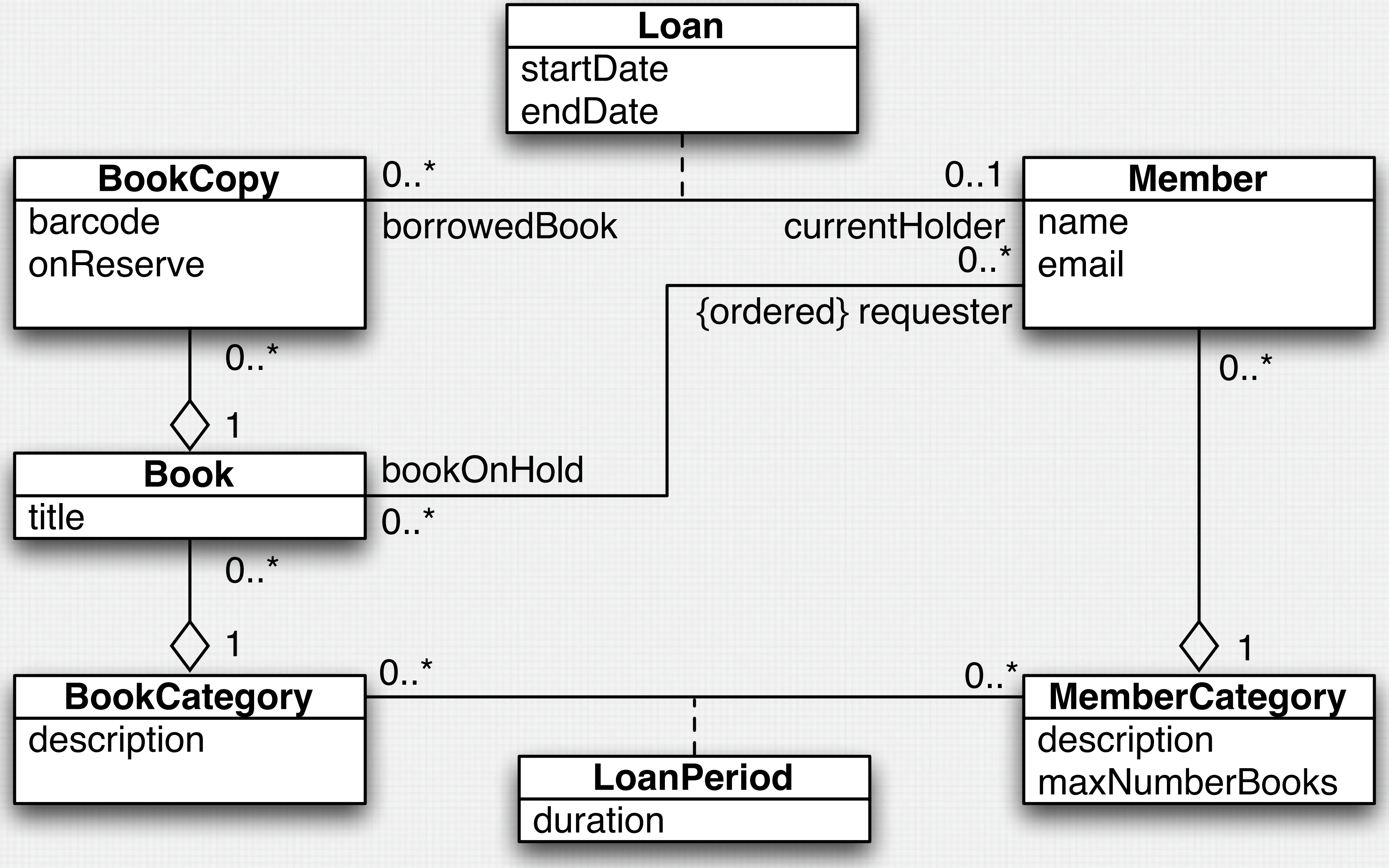
# Train Depot Questions (1)

- Develop a Domain Model that models the following situation:
  - A train is composed of train engines and cars.
  - Train engines and cars have a certain weight (measured in steps of 1 kg).
  - A car has a current load and a maximum carrying capacity (also measured in steps of 1 kg).
  - Train engines can pull up to a certain amount of kg (traction).

# Train Depot Questions (2)

- Write the following constaints and functions in OCL:
  (if your model already models that constraint, then just write: “Is covered by model”)
  1. The current load of a car cannot exceed its capacity.
  2. The length of a train should not exceed 25 train units, i.e. cars or train engines.
  3. Every train must have at least one train engine.
  4. Write an OCL function that computes the total weight of an empty train (You do not have to check that the train is empty).
  5. Write an OCL function that computes the total traction strength of a train.
  6. The total weight of a train plus the load in the cars cannot exceed the total traction strength of the locomotives of the train. (You are allowed to use the functions declared above.)
  7. Write an OCL function that computes the available load of a train, respecting all invariants mentioned above (i.e. enough room, engines are strong enough). (You are allowed to use the functions declared above.)

# Library Domain Model



# Library OCL Question (1)

- Write the following constraints and functions in OCL:
  (if your model already models that constraint, then just write: “Is covered by model”)

1. A book cannot be borrowed by more than one member.
2. The number of books on loan for a given member does not exceed the maximum number of books on loan allowed for his category.
3. A given member cannot be twice on the waiting list for the same book.
4. A member is not allowed to place holds on more than 5 books in each category.
5. Every book category must have a maximum length of loan period defined for every member category.
6. A book that is on reserve can not be on loan.

# Library OCL Question (2)

1. Write a function that calculates, for a given member, how long he is allowed to borrow a given book.
2. Write a function that calculates, for a given member, when (date) he has to go to the library the next time (because one of his books on loan has to be returned).

# Elevator Environment and Concept Model

- You are to devise the Environment Model and the Concept Model for the Elevator system based on the Use Case Model. Remember that:
  - There is only one elevator cabin, which travels between the floors.
  - There is a single button on each floor to call the lift.
  - Inside the elevator cabin, there is a series of buttons, one for each floor.
  - Requests are definitive, i.e., they cannot be cancelled, and they persist; thus they should eventually be serviced.
  - The arrival of the cabin at a floor is detected by a sensor.
  - The system may ask the elevator to go up, go down or stop. In this example, we assume that the elevator's braking distance is negligible.
  - The system may ask the elevator to open its door. The system will receive a notification when the door is closed. This simulates the activity of letting people on and off at each floor.
  - The door closes automatically after a predefined amount of time. However, neither this function of the elevator nor the protection associated with the door closing (stopping it from squashing people) are part of the system to realize.

# Take Lift Use Case (1)

**Use Case**: Take Lift

**Scope**: Elevator Control System

**Level**: User Goal

**Intention in Context**: The User intents to go from one floor to another.

**Multiplicity**: The System has a single lift cabin that may service many users at any one time.

**Primary Actor**: User

**Main Success Scenario**:

1. *User* enters lift.
2. *User* exits lift at destination floor.

**Extensions**:

1a. *User* fails to enter lift; use case ends in failure.

# Enter Life Use Case (1)

**Use Case**: Enter Lift

**Scope**: Elevator Control System

**Level**: Subfunction

**Intention in Context**: The User intends to enter the cabin at a certain floor.

**Primary Actor**: *User*

**Secondary Actors**: *Floor Sensor, Motor, Door*

**Main Success Scenario**:

1. *User* requests *System* for lift;
2. *System* acknowledges request to *User*.
3. *System* requests *Motor* to go to source floor.

*Step 4 is repeated until System determines that the source floor of the User has been reached*

4. *Floor Sensor* informs *System* that lift has reached a certain floor.
5. *System* requests *Motor* to stop;
6. *Motor* informs *System* that lift is stopped.
7. *System* requests *Door* to open;

*User enters lift at source floor.*

# Enter Life Use Case (2)

**Extensions**:

3a. *System* determines that another request has priority:

3a.1. *System* schedules the request; use case continues at step 2.

3b. *System* determines that the cabin is already at the requested floor. 3b.1a *System* determines that the door is open; use case ends in success.

3b.1b *System* determines that the door is closed; use case continues at step 7.

# Exit Life Use Case (1)

**Use Case**: Enter Lift

**Scope**: Elevator Control System

**Level**: Subfunction

**Intention in Context**: The User intends to leave the cabin at a certain floor.

**Primary Actor**: *User*

**Secondary Actors**: *Floor Sensor, Motor, Door*

**Main Success Scenario**:

*Steps 1 and 2 can happen in any order.*

1. *User* requests *System* to go to a floor.
2. System acknowledges request to *User*.
3. *Door* informs *System* that it is closed.
4. *System* requests *Motor* to go to destination floor.

*Step 5 is repeated until System determines that the destination floor of the User has been reached.*

5. *Floor Sensor* informs *System* that lift has reached a certain floor.
6. *System* requests *Motor* to stop.
7. *Motor* informs *System* that lift is stopped.
8. *System* requests *Door* to open.
9. *User exits lift at destination floor.*

# Exit Lift Use Case (2)

**Extensions**:

(3-5)||a. *User* requests *System* to go to a different floor;

(3-5)||a.1 *System* schedules the request; use case continues at the same step.

4a. *System* determines that another request has priority.

4a.1. *System* schedules the request; use case continues at step 4.

9a. *System* determines that there are additional requests pending; use case continues at step 3.