

## Employment

EmploymentAdministration											
numberOfLowPaidJobs: Integer											
<table border="1"> <tr> <td colspan="2">Job</td><td>*</td><td></td></tr> <tr> <td>salary: Integer</td><td>id: Integer</td><td></td><td></td></tr> </table>				Job		*		salary: Integer	id: Integer		
Job		*									
salary: Integer	id: Integer										
Person	*	*	Company								
isWellPaid: Boolean		0..1	salariesToBePaid: Integer								
			*								

## Employment

For the given Concept Model write the following constraints (invariants) using OCL :

- The attribute salariesToBePaid in class Company represents the amount of money to be paid as salaries by the company
- The attribute numberOfLowPaidJobs represents the number of jobs with salary less than 1000
- A person is well paid if and only if he/she has a job and the salary of this job is greater than the average salary of all jobs
- Job id is
  - unique in the system
  - unique for the jobs within a company

## Employment

- The attribute salariesToBePaid in class Company represents the amount of money to be paid as salaries by the company

Version 1:

context Company inv:

self.salariesToBePaid= self.job->collect(salary)->sum()

Version 2:

context Company inv:

self.salariesToBePaid= self.job->collect(salary)  
->iterate(it; res: Integer=0 | res+it)

Version 3:

context Company inv:

self.salariesToBePaid= self.job->iterate(it; res: Integer=0 | res+it.salary)

## Employment

- The attribute numberOfLowPaidJobs represents the number of jobs with salary less than 1000

context EmploymentAdministration inv:

self.numberOfLowPaidJobs= self.job->select(salary<1000)->size()

- A person is well paid if and only if he/she has a job and the salary of this job is greater than the average salary of all jobs

context Person inv:

self.isWellPaid= self.job->notEmpty() and  
self.job.salary

>

Job.allInstances()->collect(salary)->sum()/Job.allInstances()->size()

## Employment

- Job id is:
  - unique in the system

**Version 1:**

```
context EmploymentAdministration inv:
  self.job->forall(j1, j2 | j1.number=j2.number implies j1=j2)
```

**Version 2:**

```
Job.allInstances->forall(j1, j2 | j1.number=j2.number implies j1=j2)
```

**Version 3:**

```
context j1, j2: Job inv:
  j1.number=j2.number implies j1=j2
```

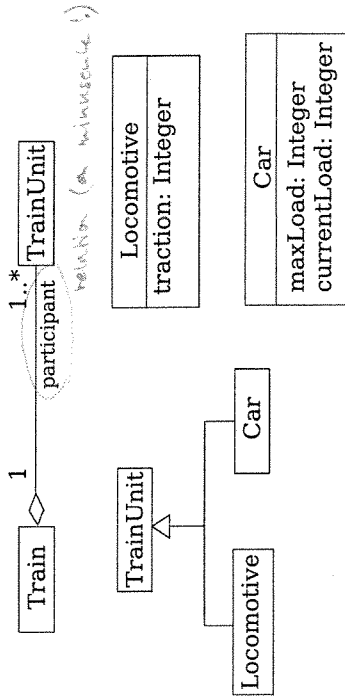
  - unique for the jobs within a company

**context Company inv:**

```
self.job->forall(j1, j2 | j1.number=j2.number implies j1=j2)
```

## Train Depot

We consider a train depot. A train is composed of "train units", each unit being either a locomotive or a car. The following Domain Model describes this situation.



## Train Depot: Constraints

- Write constraints using the OCL.
  - The current load of a car cannot exceed its capacity.
  - A train cannot have more than 25 train units.
  - One train unit in the train must be a locomotive.
  - The total load of the cars cannot exceed the total traction strength of the locomotives of the train.
  - Propose a function that computes the available traction strength of a train.
  - Propose a function that computes the available load of a train

## Train Depot: Constraints

- The current load of a car cannot exceed its capacity.
 

**context:** Car

**inv:** self.currentLoad <= self.maxLoad
- A train cannot have more than 25 train units.
 

**context:** Train

**inv:** self.participant -> size () <= 25

*set of Train Unit!*
- One train unit in the train must be a locomotive.
 

**context:** Train

**inv:** self.participant -> exists (u | uoclIsTypeOf (Locomotive))

*remember this syntax!*

## Train Depot: Constraints

1.4. The total load of the cars cannot exceed the total traction strength of the locomotives of the train.

context Train

inv: *set of TrainUnits*

*only cars in boolean*

*{car1, car2, car3}*

(self.participant -> select (u | u.ocListTypeOf (Car))  
-> collect (a | a.ocListTypeOf (Car).currentLoad) -> sum ())

<=

*{car1, car2, car3}*

(self.participant -> select (u | u.ocListTypeOf (Locomotive))  
-> collect (a | a.ocListTypeOf (Locomotive).traction) -> sum ())

*Why do we need ocListTypeOf()*

## Train Depot: Constraints

1.5. Propose a function that computes the available traction strength of a train.

context Train def:

let availableStrength():Integer =

(self.participant -> select (u | u.ocListTypeOf (Locomotive))  
-> collect (a | a.ocListTypeOf (Locomotive).traction) -> sum ())

(self.participant -> select (u | u.ocListTypeOf (Car))

-> collect (a | a.ocListTypeOf (Car).currentLoad) -> sum ())

## Train Depot: Constraints

1.6. Propose a function that computes the available load of a train

context Train def:

let availableLoad(): Integer =

self.availableStrength().min(  
(self.participant -> select (u | u.ocListTypeOf (Car))

-> collect (a | a.ocListTypeOf (Car).maxLoad) -> sum ())

(self.participant -> select (u | u.ocListTypeOf (Car))

-> collect (a | a.ocListTypeOf (Car).currentLoad) -> sum ())

)

## Train Depot: Schemas

For the Train Depot problem you are asked to provide the following Operation Schemas:

2. Operation Schemas

*→ can only be applied to a system*

Note: The constraints 1.1, 1.2 and 1.3 are system invariants.

2.1. Change the load of a car (as a result of loading or unloading it). The new weight is a parameter of the operation.

2.2. Add a car to a train.

2.3. Transfer one train unit from one train to another one.

2.4 Compute the total load weight of a train and communicate it to the driver of the train. What are the consequences for the class model?

## Train Depot: Schemas

2.1. Change the load of a car.

**Operation:** Depot :: changeLoad (target: Car, newLoad: Integer);

**Pre:** newLoad <= target.maxLoad;

**Post:** target.currentLoad = newLoad;

Note: If this operation is applied to a Car of a Train, the traction strength of the locomotives might be exceeded, but this constraint was not defined as being a system invariant.

if pre-condition there is a false message

## Train Depot: Schemas

2.2. Add a car to a train.

**Operation:** Depot :: addCar (unit: Car, to: Train);

**Pre:** to.participant -> size() < 25;

**Post:** to.participant -> includes(unit);

we want to end  
in boolean-expression

boolean

Note: The precondition ensures that the invariant 1.2 is preserved

## Train Depot: Schemas

2.3. Transfer one train unit from one train to another one.

**Operation:** Depot::transfer (from: Train, unit: TrainUnit, to: Train);

**Pre:** from.participant -> includes(unit) and

(unit.locIstType (Locomotive) implies  
from.participant ->

select (u | u.locIstType (Locomotive)) -> size() > 1) and

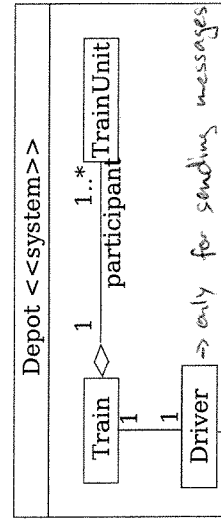
to.availableLoad() >= unit.currentLoad and

to.participant -> size() < 25;

**Post:** from.participant -> excludes(unit) and

to.participant -> includes(unit);

## Train Depot: Concept Model (partial)



<<id>> Represents

1

Schema 2.4

We must add the concept of the driver of the train to the class model, and an <<id>> association from this driver to the physical person who is the driver.

## Train Depot: Schemas

2.4 Compute the total load weight of a train and communicate it to the driver of the train.

message type Weight (load: Integer);

Operation: Depot::computeLoad (target: Train);

Messages: Person::[weight];

Pre: true;

Post: target.driver.person^weight (

target.participant-> select (u | u.occlIsTypeOf (Car))

-> collect (a | a.occlIsTypeOf (Car).currentLoad) -> sum ();

*-> we don't have an environment model...*

*post-condition is never true !!!*

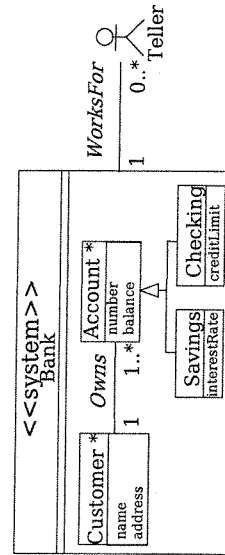
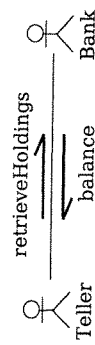
## Retrieve Holdings

### Statement

Let's consider the system operation which retrieves the holdings of a customer's accounts from the Bank system, i.e. the sum of all the balances of her/his accounts. The main effect of the operation is to notify the amount of the holdings to the teller of the bank.

1. Provide a schema for this system operation.

## Retrieve Holdings



## Retrieve Holdings

Message declaration: balance (amount: Integer);

Operation:  
Description:

Bank::retrieveHoldings (name: String);  
Results in the sum of the balances of all the accounts for a given customer being sent to the teller;

Messages:  
Aliases:

Teller::[balance];

cust:Customer Is self.customer -> any(c|c.name=name);  
total: Integer Is cust.account.balance -> sum ();

Pre: self.customer -> one(c|c.name=name);

Post: sender^balance (total);

## Monthly Verification

### Statement

"When monthly verification is performed, the owners of all accounts having a negative balance are warned."

Provide an operation schema for the verification system operation. What are the necessary changes that have to be made to the given bank concept model? Show them on the given concept model.

## Monthly Verification

**Message declaration:** warning (balance: Integer);

**Operation:** Bank::verify ();  
**Description:** Monthly verification of all bank accounts; a warning message is sent to the client if the balance is negative;

**Messages:** Client::!warning;;  
**Aliases:** overdrawnAccounts: Set (Account) **Is** self.account->select(a | a.balance<0);  
**Pre:** true;  
**Post:** overdrawnAccounts -> forall (a | a.customer.client^warning (a.balance));

## Monthly Verification

