Tutorial: OCL: tutor notes

Notes

Here are some sample answers, but in most cases there are other correct answers. If you are in doubt as to whether an answer is correct, feel free to ask me.

These questions refer to the following diagram extracted from the OCL specification. Some remarks:

- The role name `managedcompanies` is a bit weird – normally, role names, like class names, are given in the singular, so `managedcompany` would be more normal.
- Note the association classes. Association classes are examinable, but how they are referred to in OCL is not. So make sure you understand the meaning of the diagram, but for purposes of this tutorial exercise, treat these as ordinary associations.
- The small rectangle on the bottom of Bank is another bit of UML that we didn't talk about and that is not examinable: it's a qualified association. In the words of the UML standard, "In the case of multiplicity 0..*, it has no real semantic consequences but suggests an implementation that facilitates easy access of sets of associated instances linked by a given qualifier value." You may have noticed that what it does do is to affect the meaning of multiplicities on the association: the diagram does not mean to show that any Bank has 0 or 1 customers, but rather, that it has 0 or 1 customers *with a given account number.*

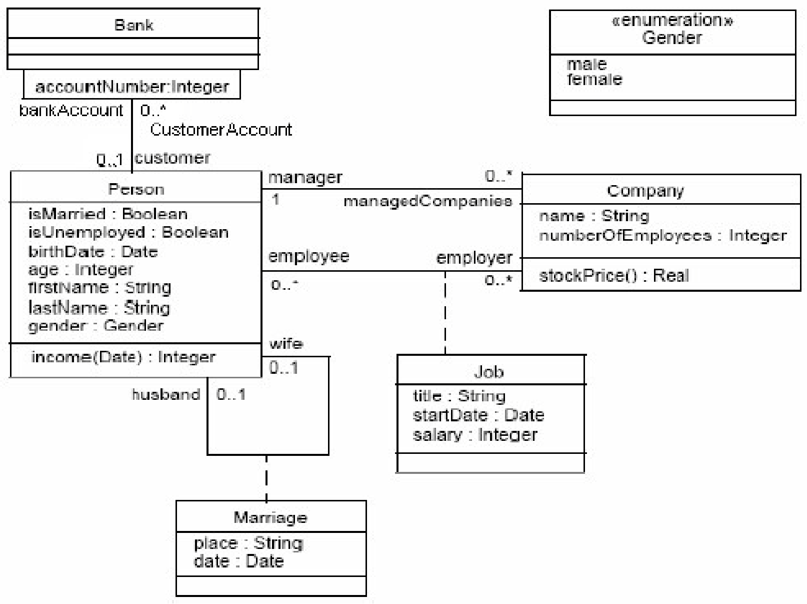

Figure 7.1 - Class Diagram Example

1 Question 1

Translate into English:

1. In the context of a Person:

   ```
   isMarried implies age > 15
   ```

   If the person is married then their age is over 15.

2.
   ```
   context Company inv:
   numberOfEmployees = employee->size()
   ```

   It is a class invariant of Company that the numberOfEmployees attribute correctly records the number of Person objects linked by an employee role to this Company object.

3.
   ```
   context Person::income(d:Date) : Integer
   pre: d.laterThan(self.birthDate)
   post: if age < 18
           then result < 100
           else result < 200
         endif
   ```

   A precondition of the income operation of Person is that the argument date should be later than the birth date of the receiving Person. (The constraint only makes sense if type Date supports a query operation laterThan taking another Date as argument.) A postcondition is that the result of the operation should be less than 100 if the age of the recipient person is less than 18, otherwise less than 200.

4. In the context of `bigBank : Bank`:

   ```
   bigBank.customer -> collect(p : Person | p.managedcompanies)
   -> asSet() -> size() >= 3
   ```

   bigBank's customers include the managers of at least 3 companies.

   What is the difference between this and

   ```
   bigBank.customer -> collect(p : Person | p.managedcompanies)
   -> size() >= 3
   ```

   ?

   No difference, because the diagram states that each company has exactly one manager anyway, so the bag of Company instances whose size we take in the first version cannot contain any Company with multiplicity greater than 1, so turning that bag into a set makes no difference to the size of the collection. If we had done the same thing with the employer role instead of the managedcompanies role, it would have made a difference: the first version would have counted the number of different companies for which the bank's customers work, whereas the second counts the companies employing the bank's customers with multiplicity (i.e., counted the total number of employments held by the bank's customers).

2 Question 2

Translate into OCL:

1. The length of a person's first name is always less than 20 characters, and so is the length of their last name.

```
context Person inv:
firstName.size() < 20 and lastName.size() < 20
```

2. Anyone who manages a company is an employee of that company. (You could write this in context Person – making it an invariant of Person – or in context Company – making it an invariant of Company. Try it both ways.)

   ```
   context Person inv:
   self.managedcompanies->forAll (c | self.employer->includes(c))
   ```

   ```
   context Company inv:
   let p = self.manager in
   self.employee->includes(p)
   ```

3. Every company has a male employee.

   ```
   context Company inv:
   self.employee->exists(e|e.gender = Gender::male)
   ```

4. It is a class invariant of Person that nobody can have more than 5 bank accounts.

   ```
   context Person inv:
   self.bankAccount->size() <= 5
   ```

5. Nobody can have two employments with companies that have identical names.

   ```
   context Person inv:
   self.employer.name->size() = self.employer.name->asSet->size()
   ```

   There are other ways to do this, but this is a handy idiom; it says "discarding duplicates from the bag of names of companies someone works for actually doesn't discard anything" i.e. there were no duplicates in the first place.