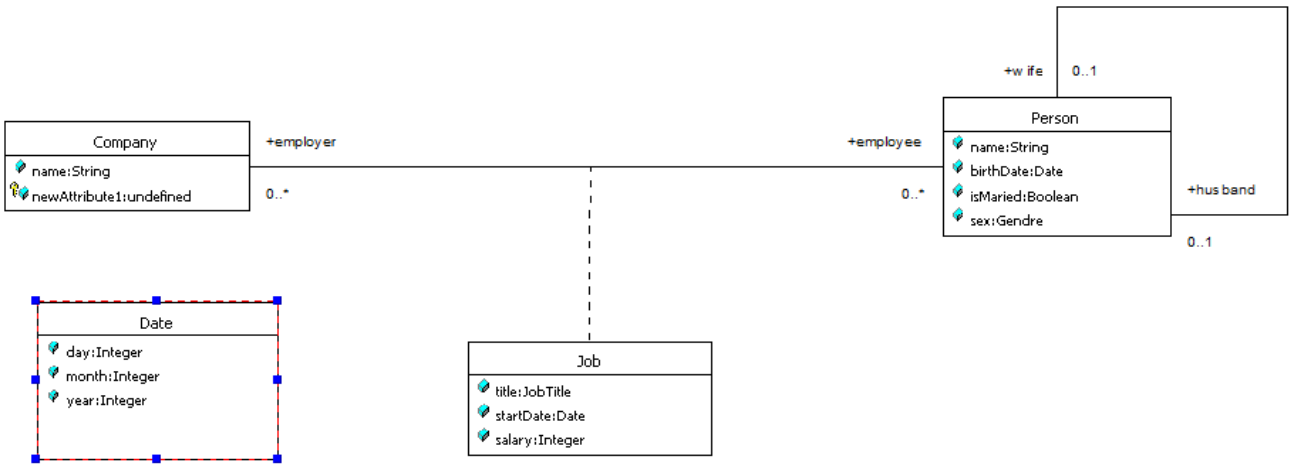
MIE – the 26th of May 2020

Working time 2h!

I. A person may be employed by companies. An employed person has a job at each employer. Any job is characterized by a title {designer, programmer, tester, other}, a start date and a salary. A person has a name, a birth date, a gender {female, male}, an information confirming that he/she is married. Each company is characterized by its name and maximum numberOfEmployee. Each married person knows the identity of the partner (wife or husband).

a. please represent by means of a UML class diagram a model complying with the above-mentioned description; 2.5pt

b. using OCL, please specify in the appropriate context an invariant checking that any employee of a company having a title different from designer cannot have a salary bigger than any other employee from the same company having the title designer; 1.5pt

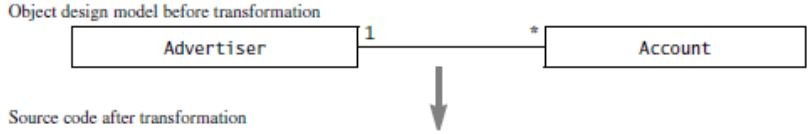



```
context Person
    inv isMaried:
        self.isMaried implies
           if sex = Gendre::female
               then self.husband.sex = Gendre::male
               else self.wife.sex =  Gendre::female
           endif

context Company
    inv designersSalary:
        let lowestDesignerSalary:Integer = self.job->select(j:Job | j.title = JobTitle::designer).salary->sortedBy(s:Integer| s)->first
        let highestNonDesignersSalary:Integer = self.job->reject(j:Job | j.title = JobTitle::designer).salary->sortedBy(s:Integer | s)->last in
        lowestDesignerSalary > highestNonDesignersSalary

context Company
    def numOfEmployedFamilies:
        let numOfEmployedFamilies:Integer = self.employee->select (e | e.isMaried and e.sex = Gendre::male)->select( e | e.husband.employer->includes(self))->size
```

II. Analyze the UML model above and explain using logical argues if the associated Java code was automatically generated or not. 1pt



Source code after transformation

```
public class Advertiser {
    private Set accounts;
    public Advertiser() {
        accounts = new HashSet();
    }
    public void addAccount(Account a) {
        accounts.add(a);
        a.setOwner(this);
    }
    public void removeAccount
            (Account a) {
        accounts.remove(a);
        a.setOwner(null);
    }
}
```

```
public class Account {
    private Advertiser owner;
    public void setOwner
            (Advertiser newOwner) {
        if (owner != newOwner) {
            Advertiser old = owner;
            owner = newOwner;
            if (newOwner != null)
              newOwner.addAccount(this);
            if (old != null)
              old.removeAccount(this);
        }
    }
}
```

As we may notice, the names of associationEnds are missing in this diagram. By default, the name of a missing associationEnd is the name of the class connecting the association with a small letter. In this case, in the Account class, the name of the opposite associationEnd is advertiser. So, each code generator will produce the declaration **private** Advertiser advertiser and not **private** Advertiser owner. The conclusion is that the presented code cannot be automatically generated. The code does not comply with the model.

III. The Figure 4 draw above presents by means of a diagram the behavior of a turnstile width diagnostic mode.

a. Please mention the type of the diagram, and all the components represented mentioning at the beginning the component type and after the list of all instances from the diagram 1.5 pt

- concrete states: Violation, Locked, Unlocked, TestCoin, TestPass
- composed states: NormalMode, DiagnosticMode
- pseudo states: 2 input states (one in each composed state) and a history state marked with H

b. In an informal manner (using your own words), please describe how the component behave 1.5 pt

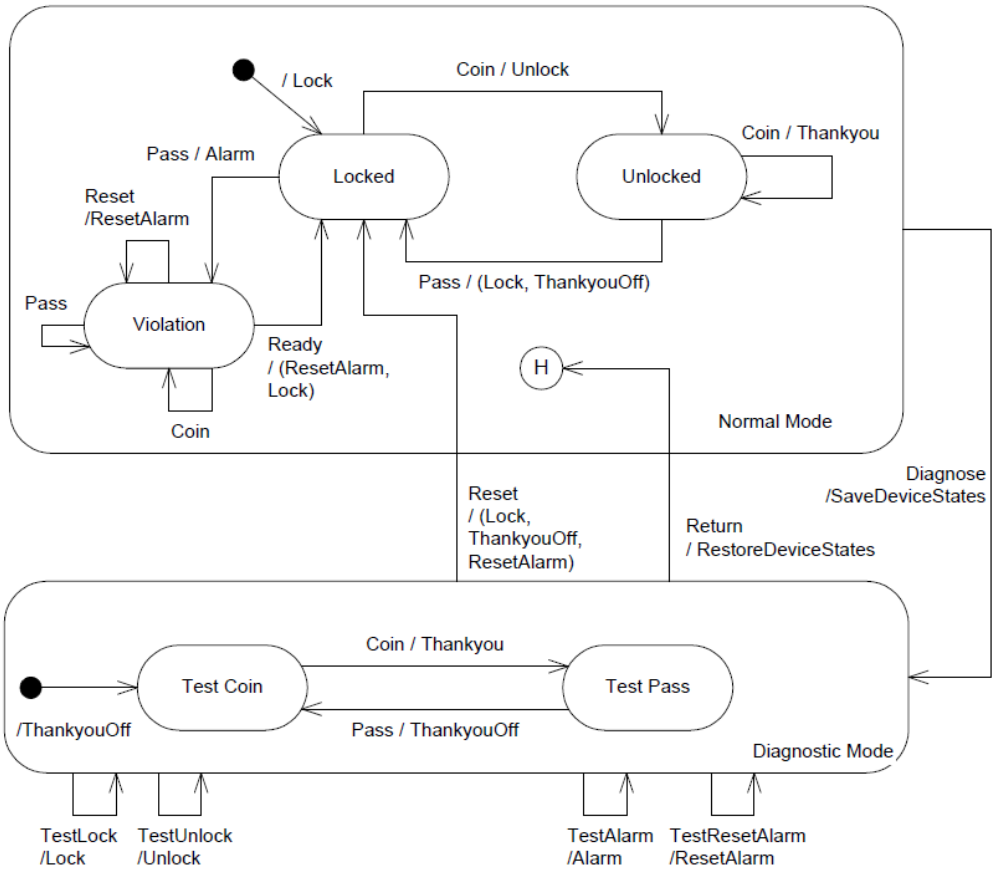


Figure 4: Turnstile with Diagnostic Mode.

The UML concepts represented in this diagram are:
1. **composed states**: NormalMode and DiagnosticMode
2. **concrete states**: Violation, Locked, Unlocked, TestCoin and TestPassReady
3. pseudo states: a. 2 **input states** (one in each composed state) and,
b. a **history state** in NormalMode

4. **transitions**: between two different states or in the same state as those triggered by stimuli: Pass, Reset, Coin
5. **events** (stimuli): Pass, Reset, Coin, Diagnose, Return, TestLock, TestUnlock, TestAlarm and TestResetAlarm
6. **actions**: Alarm, Lock, Unlock, Thankyou a.s.o.

After powering on the system, the action Lock will be executed, and the system will be in the state Locked. In this state, the events that will trigger a transition are: Coin and Pass. Coin trigger the execution of Unlock action and the transition in the state Unlock. Pass will trigger the execution of the Alarm action and the transition in the state Violation. The behavior corresponding to the remained concrete states: Violation, Unlocked, TestCoin and TestPassReady is similar.
If the system is in the composed state NormalMode (that is in one of its inner concrete states) and a Diagnose event will appear, then, a transition will be executed: from the inner concrete state of

NormalMode in the inner concrete state TestCoin of the composed state DiagnosticMode. The semantics of the auto-transitions specified for the composed state DiagnosticMode is that the system will remain in the concrete state in which it is and, the corresponding action of each event will be executed. If the system is in the composed state DiagnosticMode and the Return event will appear, then, the ReturnDeviceStates will be executed and the system will transit in the last concrete state of NormalMode in which the system was before to transit toward TestCoin. This happen due to the pseudostate History.

It is a State Transition Diagram (STD), which is used to specify the behavioral view of reactive objects.

IV. Analyzing the SimpleWatch use case diagram below, it is easy to notice that some usual relationships were not specified. Please includes these relationships and justify why these(this) are(is) needed. 1pt

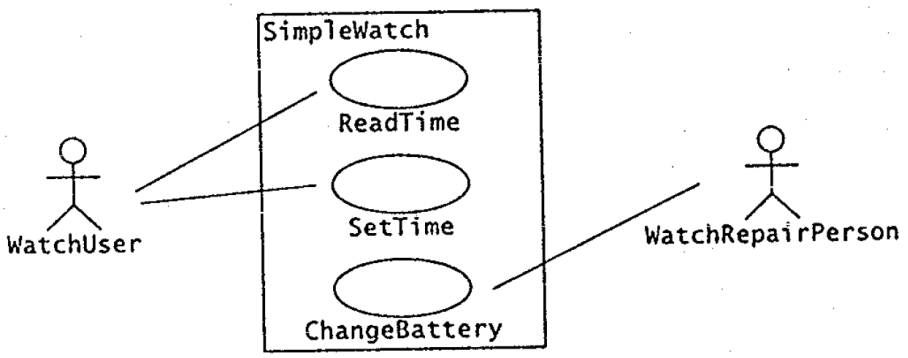


Figure 1 - An incomplete SimpleWatch use case diagram

It is noticeably clear that any WatchRepairPerson is also a WatchUser. So, the missing relationship is an inheritance relationship (specialization) between the WatchRepairPerson and the WatchUser. In this manner, the WatchRepairPerson will inherit the associations towards ReadTime and SetTime UseCases being able to access all the functionalities