Lab 7.
Prototyping and Testing

Question 1 – Car Park System

Consider the design class model and the atomic use case specifications for the Car Park System, given below. The specifications cover the following atomic use cases:

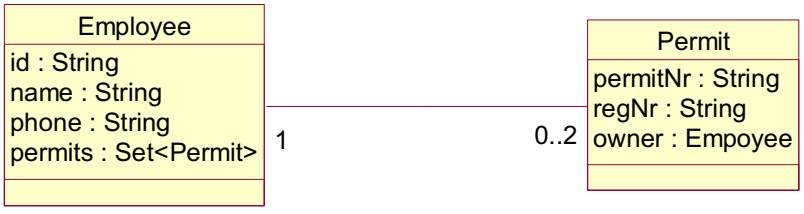
1. Add a new employee
2. Change an employee's phone number
3. Add a permit
4. Delete an employee
5. Terminate a permit
6. Replace a permit
7. Given a permit number, get the details (id, name, phone number) of the owner of the permit.
8. Get details for each of the permits. For each permit, get the following details: the permit number, the employee's id, name and phone number, and the car's registration number

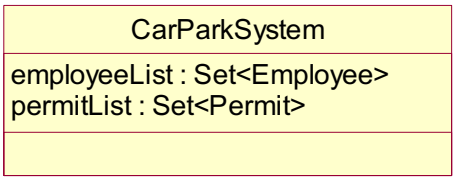
Tasks 1 - 8

For each of the use cases,

- Prototype the atomic use case
- Design the test cases and use them to test your implementation of the atomic use case

Make sure you understand the relationships between the atomic use case specifications, the programs, the test cases and the test results.





1. Add an employee

```
in:
    id? : String
    name?: String
    phone?: String

out:
    NONE

pre:
    // id? is new
    not exists e in employeeList |  e.id = id?

post:
    // create a new employee
    let e = new Employee(id?, name?, and phone?) |
        e.id = id?
        e.name = name?
        e.phone = phone?
        e.permits = new Set<Permit>( )

    // add the employee to the set of all employees
    add e to employeeList
```

2. Change an employee's phone number

```
in:
    id? : String
    phone? : String

out:
    NONE

pre:
    // id? exists
    exists e in employeeList |  e.id = id?

post:
    // retrieve the employee
    let e = element in employeeList | e.id = id?

    // change the employee's phone number
    set e.phone = phone?
```

3. Add a permit

```
in:
    permitNr?: String
    regNr?: String
    id? : String
out:
    NONE
pre:
    // permitNr? is new
    not exists p in permitList | p.permitNr = permitNr?

    // registration number is new
    not exists p in permitList | p.regNr = regNr?

    // id? exists and the employee has less than 2 permits
    exists e in employeeList |
          e.id = id?
          size (e.permits) < 2
post:
    // retrieve the owner
    let owner = element in employeeList | e.id = id?

    // create a new permit
    let p = new Permit(permitNr?, regNr?, owner ) |
            p.permitNr = permitNr?
            p.regNr = regNr?
            p.owner = owner

    // update the set of permits of the owner
    add p to owner.permits

    // add new permit to the set of all permits
    add p to permitList
```

4. Delete an Employee

```
in:
    id? : String
out:
    NONE
pre:
    // id? exists
    exists e in employeeList |  e.id = id?

post:
    // retrieve the employee
    let e = element in employeeList | e.id = id?

    // delete the employee and all of his or her permits, if any
    set permitList = permitList MINUS  e.permits
    remove e from employeeList
```

5. Terminate a permit

```
in:
    permitNr?: String

out:
    NONE

pre:
    // permitNr? exists
    exists p in permitList | p.permitNr = permitNr?
post:

    // retrieve the permit
    let p = element in permitList | p.permitNr = permitNr?

    // delete the permit
    remove p from p.owner.permits
    remove p from permitList
```

6. Replace Permit

```
in:
    oldPermitNr?: String
    newPermitNr?: String
    regNr?: String
    area?: String

out:
    NONE

pre:

    // oldPermitNr? exists
    exists p in permitList | p.permitNr = oldPermitNr?

    // newPermitNr? is new
    not exists p in permitList | p.permitNr = newPermitNr?

    // registration number is new
    not exists p in permitList | p.regNr = regNr?
```

```
post:

    // retrieve the old permit and the owner
    let oldPermit = element in permitList |
          oldPermit.permitNr = oldPermitNr?

    let owner = oldPermit.owner

    // create a new permit
    let p = new Permit(permitNr?, regNr?, area?, owner ) |
            p.permitNr = permitNr?
            p.regNr = regNr?
            p.area = area?
            p.owner = owner

    // update the set of permits of the owner
    remove oldPermit from owner.permits
    add p to owner.permits

    // update the set of all permits
    remove oldPermit from permitList
    add p to permitList
```

7. Given a permit number, get the details of the employee who has the permit. Include the employee's id, name and phone number

```
in:
    permitNr?: String

out:
    result! // Do not need to specify the type.
            // Determined by expressions in the postcondition

pre:
    // permitNr? exists
    exists p in permitList | p.permitNr = permitNr?

post:
    // retrieve the permit and the owner
    let p = element in permitList | p.permitNr = permitNr?
    let owner = p.owner

    // retrieve permit number, employee's name and phone number
    let result! = ( owner.id, owner.name, owner.phone )
```

Note: Alternatively, and equivalently, we can have three output items

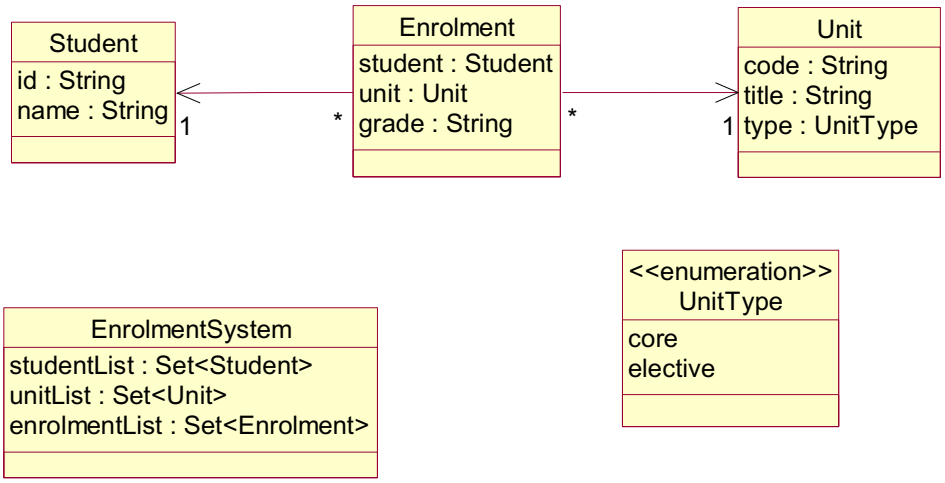
```
in:
    permitNr?: String

out:
    id!
    name!
    phone!

pre:
    // as above

post:
    // retrieve permit number, employee's name and phone number
    let p = element in permitList | p.permitNr = permitNr?
    let owner = p.owner
    id! =  owner.id
    name! = owner.name
    phone! = owner.phone
```

8. List all the permits. Include the following details: the permit number, the employee's id, name and phone number, and the car's registration number

```
in:
    NONE

out:
    result! // postcondition shows that we return a list of tuples
            // (i.e. a list of records)

pre:
    // the set of permits is not empty
    size( permitList ) > 0

post:

    // for each permit return permit nr, employee id, phone nr,
    // and reg nr
    let result! = new List()
    for each p in permitList do
    {
        let e = p.owner
        add ( p.permitNr, e.id, e.name, e.phone, p.regNr ) to result!
    }
```

Question 2 - Subject Enrolment

Assume that we have the following design class model for a simplified student record system. Suppose each student must enrol in at least 2 core subjects.



Consider the atomic use case specification for the use case to *enter the enrolment of a student*. (Note precondition 2, which requires that the student have no subjects enrolled.)

```
In:
    id?: String
    unitCodes?: List<String>

Out:
    None

Pre:
    // pre1: id? exists
    exists c in studentList | c.id = id?

    // pre2: student has not enrolled yet
    let units = set of e in enrolmentList |
        e. student.id = id?
    then
        size(units) = 0

    // pre3: all unit codes in the input list are distinct
    for each 1 in 1.. size(unitCodes?) then
        for each j in 1..size(unitCodes?) then
            if i ≠j then unitCodes?[i] ≠ unitCodes?[j]
```

```
    // pre4: each unit code in the input list exists
    for each code in unitCodes? then
        exists u in unitList | u.code = code


    // pre5: there are at least 2 core units
    let coreUnits = set of u in unitList |
        u.code in unitCodes? and u.type = "core"
    then
        size( coreUnits ) >= 2
```

Post:

```
    // retrieve the student
    Let student = element in studentList | student.id = id?

    For each :code in unitCodes? do
    {
        // retrieve the unit
        Let unit = element in unitList | unit.code = :code

       // create the enrolment
       Let enrolment = new Enrolment(student, unit) |
            enrolment.student = student
            enrolment.unit = unit
            grade = null

       // and add the enrolment to the set of all enrolments
        add enrolment to enrolmentList
    }
```

Implement the above atomic use case and design the test cases to test it thoroughly.

Of course, you will need to implement a number of classes. For each class, include enough details in order to implement and test the atomic use case.

■