Software Engineering

Books or notes are **not** allowed.
Write only on these sheets. **Concise** and **readable** answers please.

Surname, name, matricola _____

Car parking
In most cities owners of cars have to pay to park their car in the roads. The parking fee can be of two types

- One off. The owner pays a fee for a certain duration of park, in a certain place. The fee depends on the duration, and on the area of the city (ex central area 2 euro/hour, other areas 1 euro /hour). It also depends on the day (ex free parking on Sunday) and on the part of the day (ex reduced 50% fee parking from 8pm to 6 am). The fee is attached to one and only one car (identified by its tag).
- Pass. The owner pays a fee that covers a longer period (ex week or month) in a specific area only (only one area). This allows to park any time and for any duration in that area. The fee is attached to one and only one car (identified by its tag).

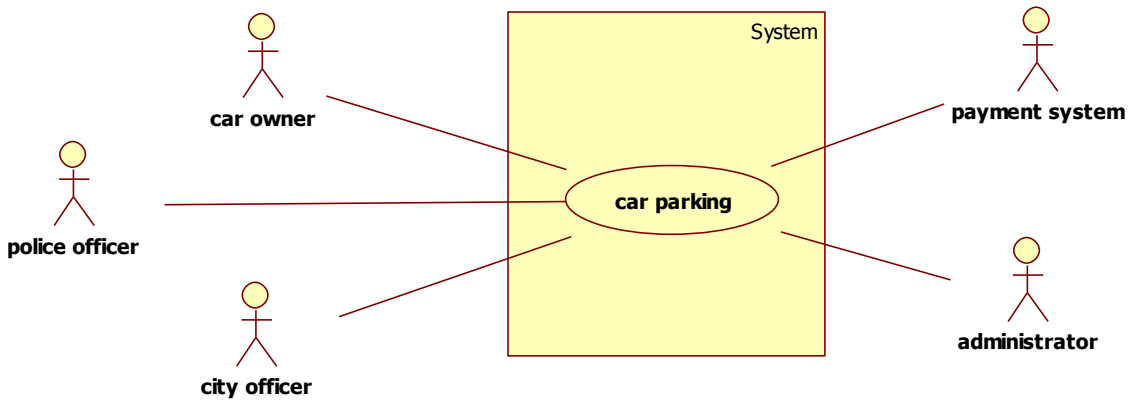
In the following you should analyze and model the client server application that supports the collection of fees and enforcing of payments.

You should consider

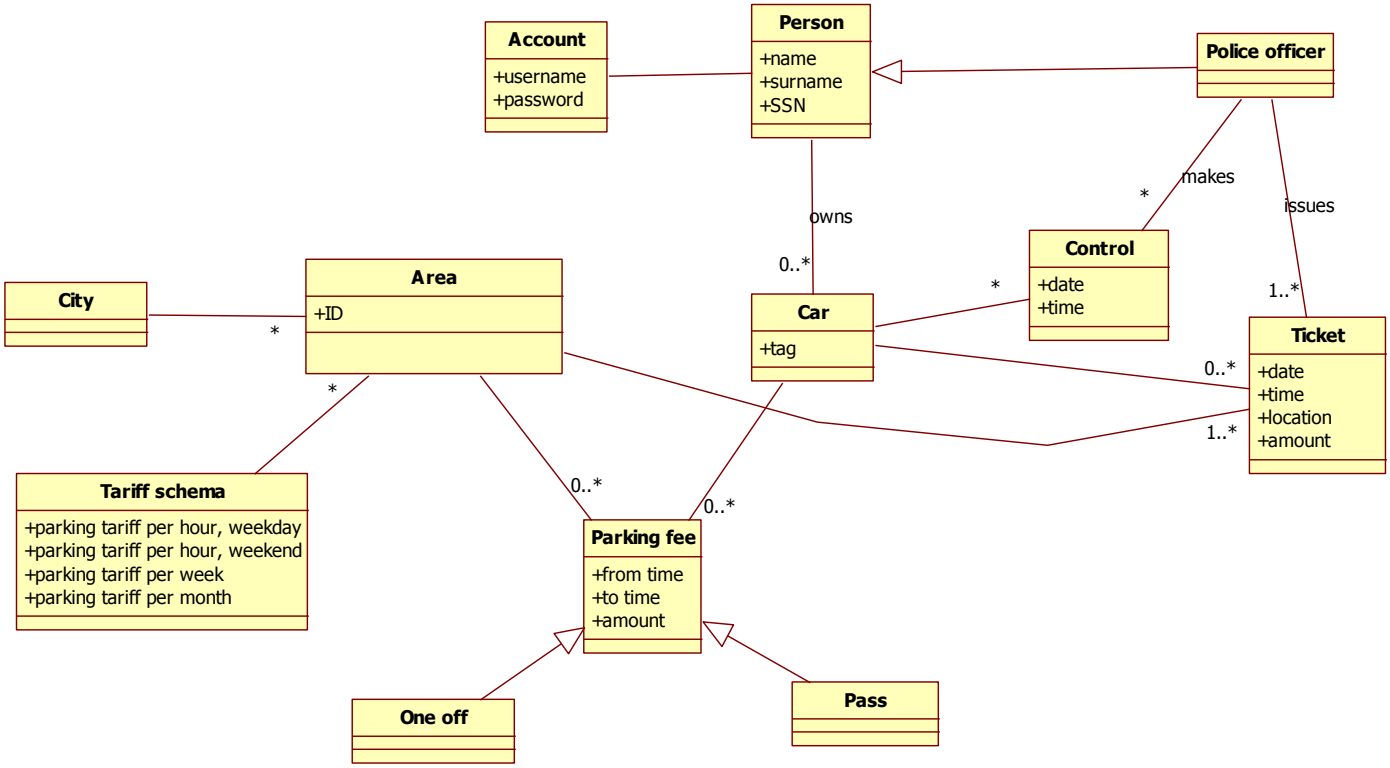
- Payment by car owners using smartphones. Car owners define an account on the application, attach a credit card to it, and pay the fee (one off or pass). There is no need to print or show anything on the car.
- Control of payment by police officers. Police officers, using a smartphone, can check that a certain car parked in a certain place has paid the correct fee.
- Issue of ticket. If car is parked and no fee has been paid, a police officer can issue a ticket.
- Monitor. A city officer can check a number of statistics: fees collected per area and per period, fees collected per car, controls operated per police officer, tickets issued per period, and so no.

1 – a. Define the **context diagram** (including relevant interfaces)

Actor	Physical interface	Logical interface
car owner	PC / smartphone screen + keyboard	GUI
Police officer	PC / smartphone screen + keyboard	GUI
City officer	PC / smartphone screen + keyboard	GUI
Admin	PC screen + keyboard	GUI
Payment system	Internet connection	API



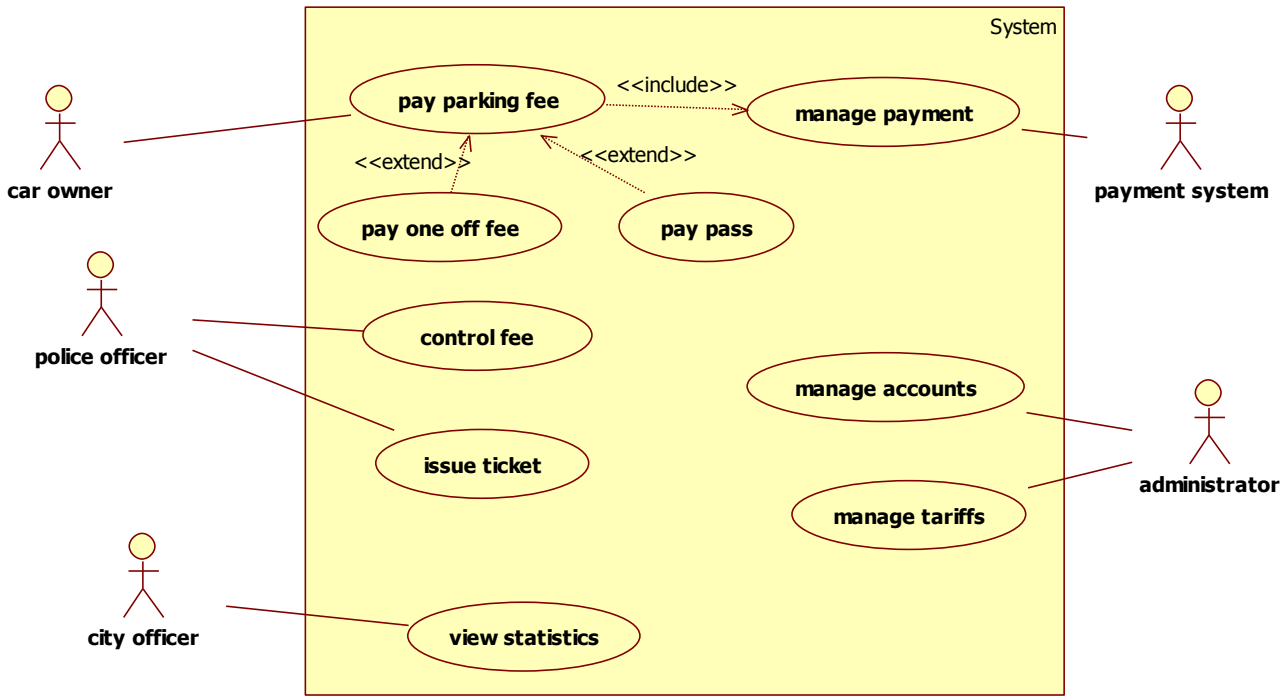
1-b Define the **glossary** (key concepts and their relationships) (UML class diagram) for the application



Usually the parking place (the specific few square meters area dedicated to one car) is not identified in cities, so there is no corresponding class.
Area is used in many parts (attached to Parking fee, Ticket, Tariff schema) so it must be a class.
Area is attached to Parking fee. By inheritance both One off and Pass are attached to Area
The tariff schema can be quite complex so better to represent it with a class (instead of spreading the related information in Area or Parking fee).
Statistics will become operations (computations) so they are not represented as classes. However for each statistic the required data must be available, and represented as class or attribute (see for instance the 'Control' class that represents a control operated by a police officer, required to compute the statistic 'controls operated by police officer')

1-c Draw the Use Case Diagram for the application. For each Use Case give a short description here

Use Case ID	Description
1	Pay parking fee. Select type (one off or pass), select area, duration, start time, select car (tag), manage payment.
2	Control fee. Insert car tag, area, query if car has right to park at this point in time in the area.
3	Issue ticket. Insert car tag, issue ticket for violation of parking regulation at this point in time in the area.
4	View statistics
5	Manage accounts. Create, delete, modify accounts
6	Manage tariffs. Read, modify parking fee tariff for selected area and time period.
7	Manage payment. Define amount to be paid, payment method, supervise payment until success or exception



Reminder

The use cases should provide a value to an actor (ex login logout insert credit card number are lower level actions that do not provide value)

UC are not flow charts (don't try to represent a sequence of actions, scenarios should be used for that)

1-D List the **NON functional requirements** that you deem important for the application

ID	Description
1	Usability: user regularly using a smartphone for at least one year should be able to use the system in < 10min the first time, with no training
2	Efficiency: all functions should complete in <0,5 sec
3	Privacy: data of user should not be visible to other users
4	Portability: client app should run on Ios and Android; web app should run on IE from version x, Chrome for version y
5	Domain: currency is euro

1-e Describe below the scenario specific to a car owner who pays a one-off fee

Precondition: user has a car, has an account

Postcondition: one off fee for certain amount, certain area, has been attached to car c starting at time t

Step	Description
1	User logs in account
2	User selects one off fee payment, selects area, day, start time, end time. Select tag of car
3	User selects a payment method
4	User performs payment
5	System records one off fee and attaches it to car tag

2 (7 points) -Define black box tests for the following function, using equivalence classes and boundary conditions.

A railway company offers the possibility to people under 15 to travel free. The offer is dedicated to groups from 2 to 5 people travelling together.

For being eligible to the offer, at least a member of the group must be at least 18 years old. If this condition applies, all the under 15 members of the group travel free, and the others pay the Base Price.

The function computeFee receives as parameters basePrice (the price of the ticket), n_passengers (the number of passengers of the group), n_over18 (the number of passengers at least 18 old), n_under15 (the number of passengers under 15 years old). It gives as output the amount that the whole group has to spend. It gives an error if groups are composed of more than 5 persons.

double computeFee(double basePrice, int n_passengers, int n_over18, int n_under15);

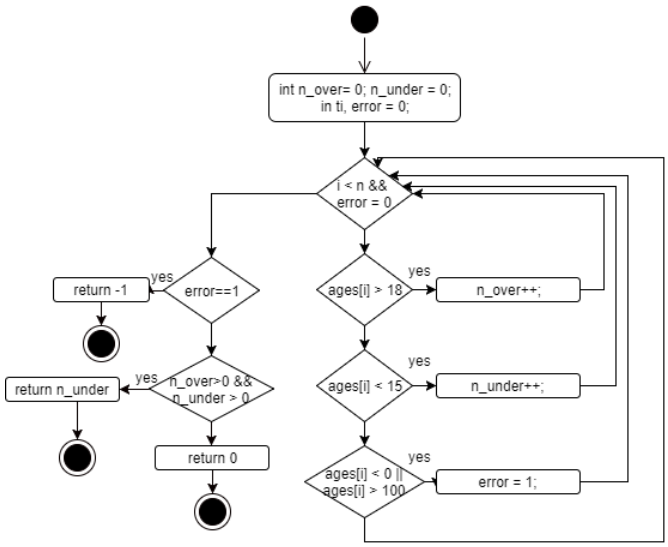
Examples:
computeFee(20.0, 3, 0, 1) -> 60.0;
computeFee(30.0, 5, 1, 2) -> 150.0.

basePrice	n_passengers	n_over18	n_under15	n_passengers >= n_over18 + n_under15	V/I	TC
[mindouble, 0]	*	*	*	*	I	(-10.0, 3, 1, 1) -> error B(**0**, 3, 1, 1) -> error
*	[minint, 1]	*	*	*	I	(10.0, -5, 2, 2) -> error B (10.0, **1**, 2, 2) -> error
*	*	[minint, -1]	*	*	I	(10.0, 5, -2, 2) -> error B(10.0, 5, **-1**, 2) -> error
*	*	*	[minint, -1]	*	I	(10.0, 5, 2, -2) -> error B(10.0, 5, 2, **-1**) -> error
(0, maxdouble)	[2, 5]	0	[0, 5]	no	I	(10.0, 3, 0, 4) -> error
"	"	"	"	yes	V	(10.0, 2, 0, 2) -> 2*10.0 = 20.0 B(**0.00001**, 3, 0, 2) -> 3*0.000001 B(10.0, **2**, 0, 2) -> 2*10.0 B(10.0, 3, 0, **0**) -> 3*10.0 B(10.0, **5**, 0, **5**) -> 5*10.0
"	"	[1, 5]	[0, 5]	no	I	(10.0, 2, 2, 3) -> error
"	"	"	"	yes	V	(10.0, 4, 2, 2) -> 2*10.0 = 20.0 B(10.0, 4, **1**, 2) -> 2*10.0 = 20.0 B(10.0, 5, **5**, 0) -> 5*10.0 = 50.0
*	[6, maxint]	*	*	*	I	(10.0, 7, 2, 2,) -> error B(10.0, ,2, 2) -> error
*	*	[6, maxint]	*	*	I	(10.0, 2, 7, 2) -> error B(10.0, 2, **6**, 2) -> error
*	*	*	[6, maxint]	*	I	(10.0, 2, 2, 7) -> error B(10.0, 2, 2, **6**) -> error

3 (7 points) – For the following function define the control flow graph, and define test cases to obtain the highest possible node coverage, edge coverage, multiple condition coverage, loop coverage, path coverage. For the test cases, **write only the input value**.

```
1 int main(int ages[5]) {
2     int n_over, n_under;
3     int i, error = 0;
4
5     for (i = 0; i < 5; i++) {
6           if (ages[i] > 18)
7                 n_over++;
8           else if (ages[i] < 15)
9                 n_under++;
10          else if (ages[i] < 0 || ages[i] > 100)
11                error = 1;
12    }
13
14 if (error == 1)
15    return 1;
16 if (n_over > 0 && n_under > 0)
17    return n_under;
18 else
19    return 0;
20 }
```

Write control flow graph here



Coverage type	Number of test cases needed to obtain 100% coverage	Coverage obtained with test cases defined (%)	Test cases defined
Node	Not feasible	11/13 = 84% *	T1, T2
Edge	Not feasible	16/20= 80% *	T1, T2
Multiple condition line 16	4	100%	T1,T2, T3, T4
Loop line 5	3	33% *	Only multiple iteration feasible (any test case)
Path	3^5		

Write test case ID (T1, T2 ..) in the rightmost column, and test cases here

T1: {20, 20, 10, 10, 10}; return n_under; multiple condition line 16: TT
T2: {10, 10, 10, 10, 10}; return 0; multiple condition line 16: FT
T3: {20, 20, 20, 20, 20}; return 0; multiple condition line 16: TF
T4: {16, 16, 16, 16, 16}; return 0; multiple condition line 16: FF

* The obtainable coverage is influenced by the presence of dead code in the tested function (line 11 and 15 are not reachable). Solutions considering full coverage (and hence three test cases for node and edge coverage, and 66% loop coverage) have not been considered wrong.

4 (1 points) – In the context of verification and validation, describe Weinberg's law

The creator of a program is unsuitable to test it – for emotional attachment to its creature the programmer tends to overlook defects in it.

5 (1 point) – In the context of change control, describe the lock-modify-unlock technique, its pros and cons.

See slides

6 (1 point) — In the context of configuration management, explain what is a baseline and when is it used.

A baseline is a configuration (== a set of configuration items) in a stable form (ex compiles, links, passes all regression tests).
Not all configurations are baselines
Used to delivery an application internally / externally, while development continues on next version.

7 (1 point) – Describe the waterfall process, its pros and cons.

Activities (requirement design implementation unit test intergration test system test) are done in sequence (activity i+1 starts only after activity i is completed).
Document oriented.
Pro: easy structure of activities; agreement on design allows to allocate tasks to many, distributed workers/companies
Con: delivery to customer and validation of requirements and system happen very late. Changes require to restart the process, slowness and lack of flexibility.

8 (1 point) – Describe the singleton design pattern, and when it can be used.

Creational pattern. Ensures that only one instance of the class is created. Ex in an operating system the load balancer must be unique, and could be implemented by a singleton.