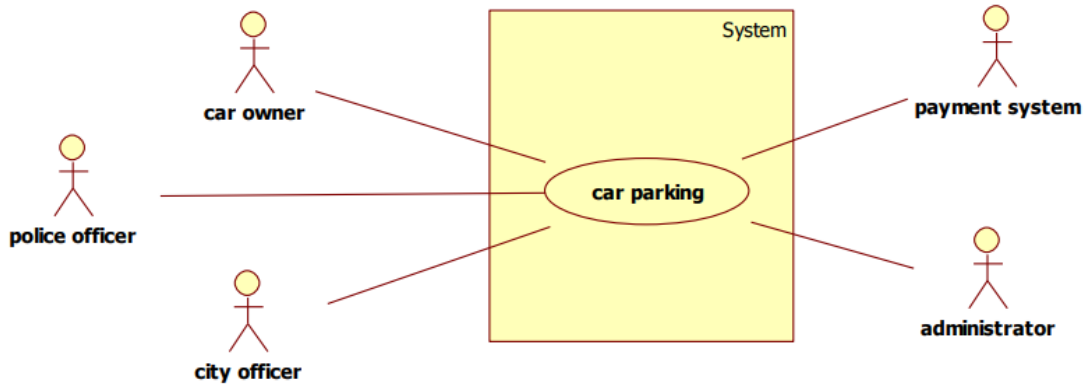
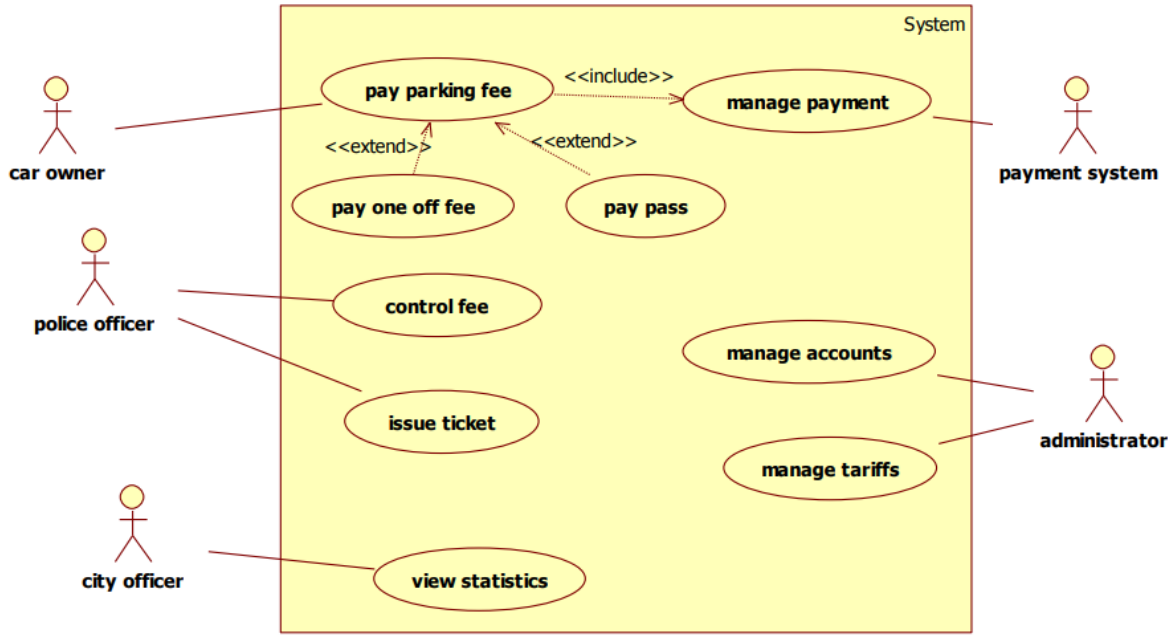
Q1.

a.


System
car owner
payment system
police officer
car parking
administrator
city officer

b.

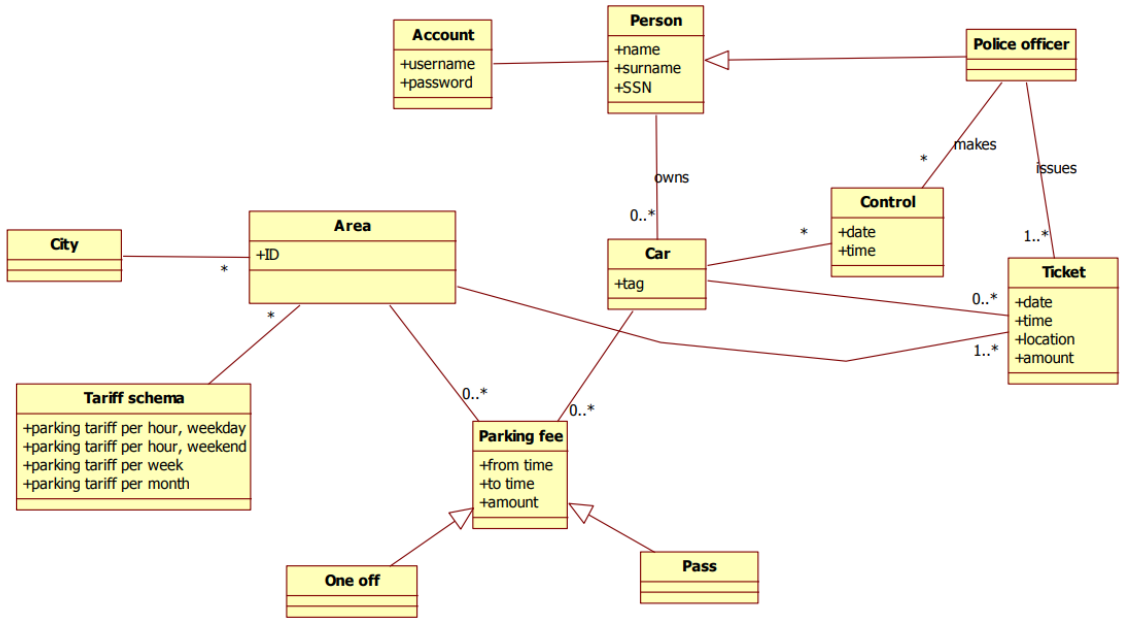

System
pay parking fee
<<include>>
manage payment
car owner
payment system
<<extend>>
<<extend>>
pay one off fee
pay pass
police officer
control fee
manage accounts
administrator
issue ticket
manage tariffs
city officer
view statistics

C -d

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

E



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.

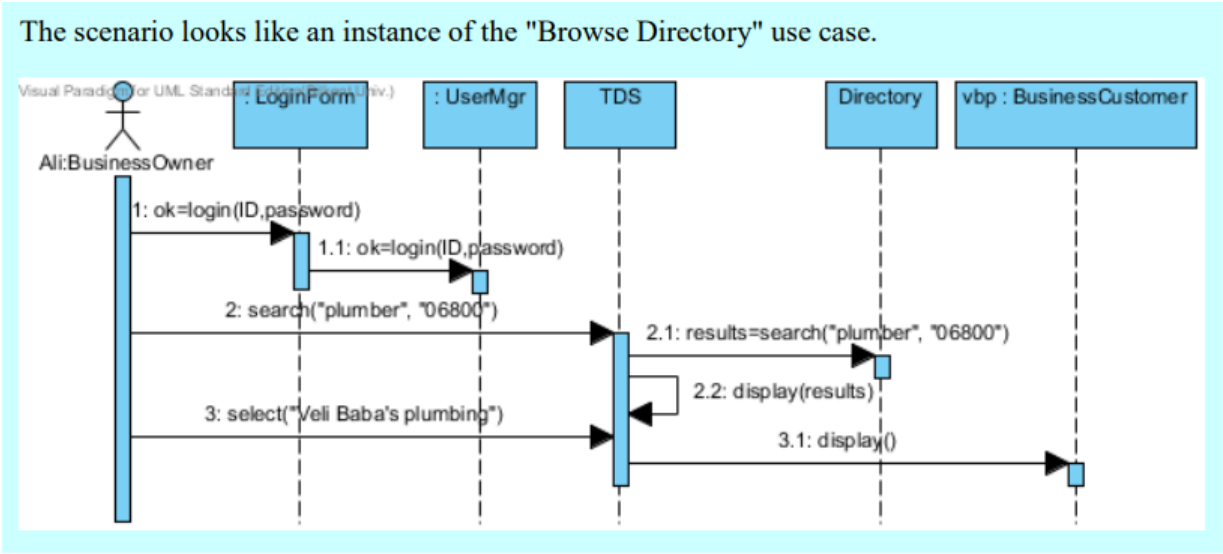
The tariff schema can be quite complex so better to represent it with a class (instead of spreading the related information in Area or Parking fee).

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'

Q2

A - Can easily be converted to SSD for browse Directory usecase

b

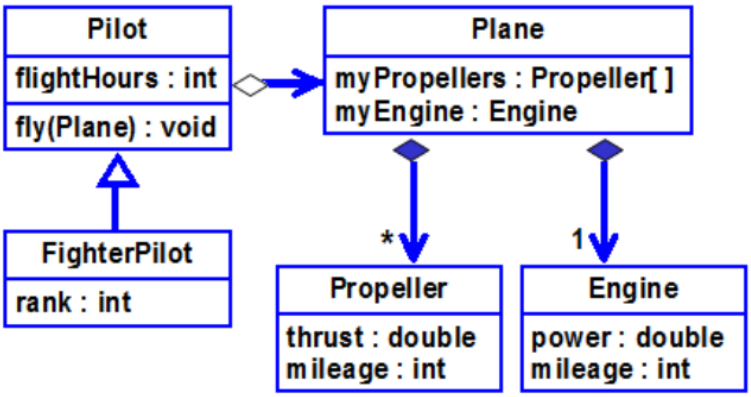


Q3

a.

3-b

Solution:



Marking:

Class identification → 2
Inheritance → 1/2
Cardinality → 1
Attributes enforcing cardinality constraints → 2
Composition relation → 2
Methods → 2
Pilot uses Plane → 1/2

3c

31. (1 point) What is the relationship between `Vertebrate` and `Animal`?

    `Vertebrate` is a subclass of `Animal`.

32. (1 point) What is the relationship between `Snake` and `Animal`?

    `Snake` is a subclass of `Animal`.

33. (2 points) What is the **association type** between `Mongoose` and `Snake`?

    Dependency association.

34. (1 point) According to the class diagram, can 2 mongooses eat 1 snake?

    No. The relationship between `Mongoose` and `Snake` is one-to-zero-or-more.

35. (3 points) Identify one thing wrong with this diagram.

    Because `Animal` is an interface, the relationship from `Vertebrate` should be dashed.

Q4

(a)

1.observer
2.Façade
3.Adapter
4.Singleton
5.Factory

(b)

```
lass Singleton {
   private static volatile Singleton instance = null;

   // Private constructor to prevent instantiation from outside
   private Singleton() {
      // Initialization code, if needed
   }
```

```
// Public method to get the instance of the Singleton class
public static Singleton getInstance() {
    if (instance == null) {
        synchronized (Singleton.class) {
            if (instance == null) {
                instance = new Singleton();
            }
        }
    }
```

Q5.

- **Singleton Pattern:**
  - Singleton class: **`Deadbeef`**
  - Roles: Ensures a single instance of the class and provides a global point of access.
- **Factory Method Pattern:**
  - Factory Interface: **`Dead`**
  - Concrete Product: **`Beefdead`**
  - Concrete Creator: **`Deadbeef`**
  - Roles: Defines a factory method (**`dead1`**) in the interface, and the implementing class (**`Deadbeef`**) creates an object of the concrete product (**`Beefdead`**) using this method.
- **Observer Pattern:**
  - Subject Interface: **`Beef`**
  - Concrete Subject: **`Beefdead`**
  - Observer Interface: **`Foo`**
  - Concrete Observer: **`Bar`**
  - Roles: **`Beef`** maintains a list of observers (**`Foo`** objects), and when specific events occur (calling **`beef2`**), it notifies and triggers the observers' actions (**`foo1`**). **`Bar`** is an implementation of the observer.