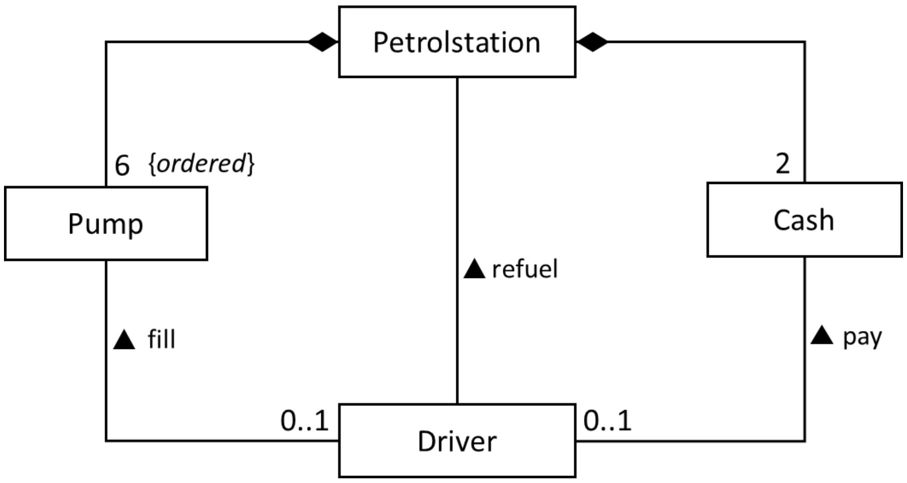
## 9. *Collections for 1-n relationships*

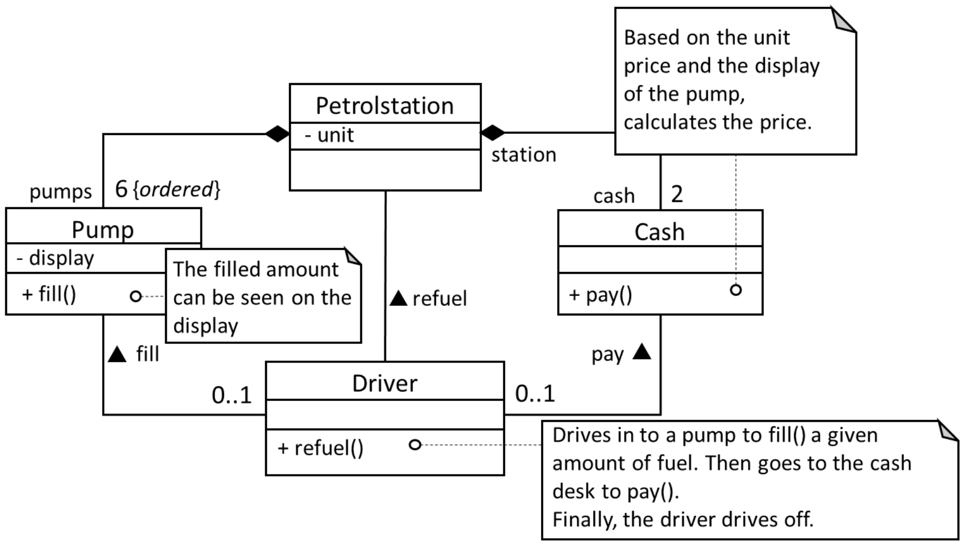
Task: Class diagrams with detailed definitions of the methods. Many times a method processes a collection that implements a '1-*' relationship.

1. On a petrol station, there are 6 pumps and 2 cash desks. When a driver drives in, he decides at which pump to fill (it is a message sent to the station, which checks if the given pump exists). After getting to the pump, the driver fills the vehicle with a given amount of fuel, then goes to pay at the cash desk (the cash desk checks the amount of fuel filled by the driver, calculates the price, and clears the display of the pump). Finally, the driver drives off. One driver can use only one pump and only one cash desk. This task is going to be solved again on a later lecture with parallel blocks and awaits.

First, let us see the main structure of the program:

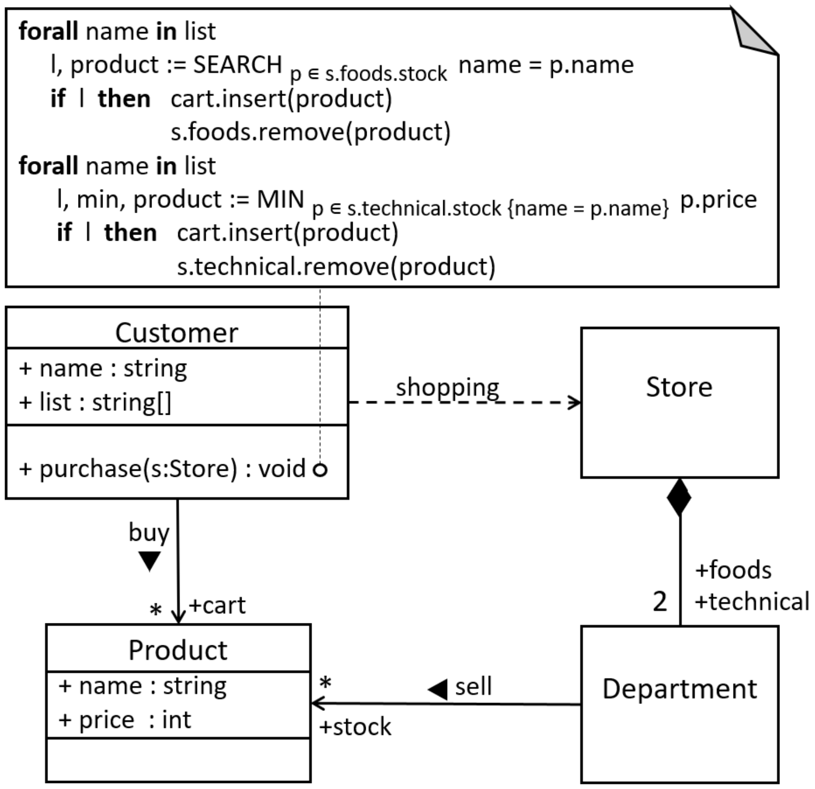


Then, let us consider what procedure refuel() should do. For that, let us introduce some new attributes and methods:


Petrolstation
- unit
station
pumps
6 {ordered}
Pump
- display
+ fill()
The filled amount can be seen on the display
Based on the unit price and the display of the pump, calculates the price.
cash
2
Cash
+ pay()
▲ refuel
▲ fill
pay ▲
0..1
Driver
+ refuel()
0..1
Drives in to a pump to fill() a given amount of fuel. Then goes to the cash desk to pay().
Finally, the driver drives off.

Homework: decide the visibility and the owner of the role names, define some getters and setters, and introduce some new methods (e.g. to access a pump of a given index from the Cash_desk class so that its fill() method could be called).

2. A small-town store has a food and a technical department. The customers have a shopping list, which contains the name of the products they would like to buy. They buy all the products on their list they can find, but they do this differently at the two departments. At the food department, they just put the first product they find into their cart. At the technical department, they always buy the cheapest product from among products with the same name. Simulate the purchasing process.



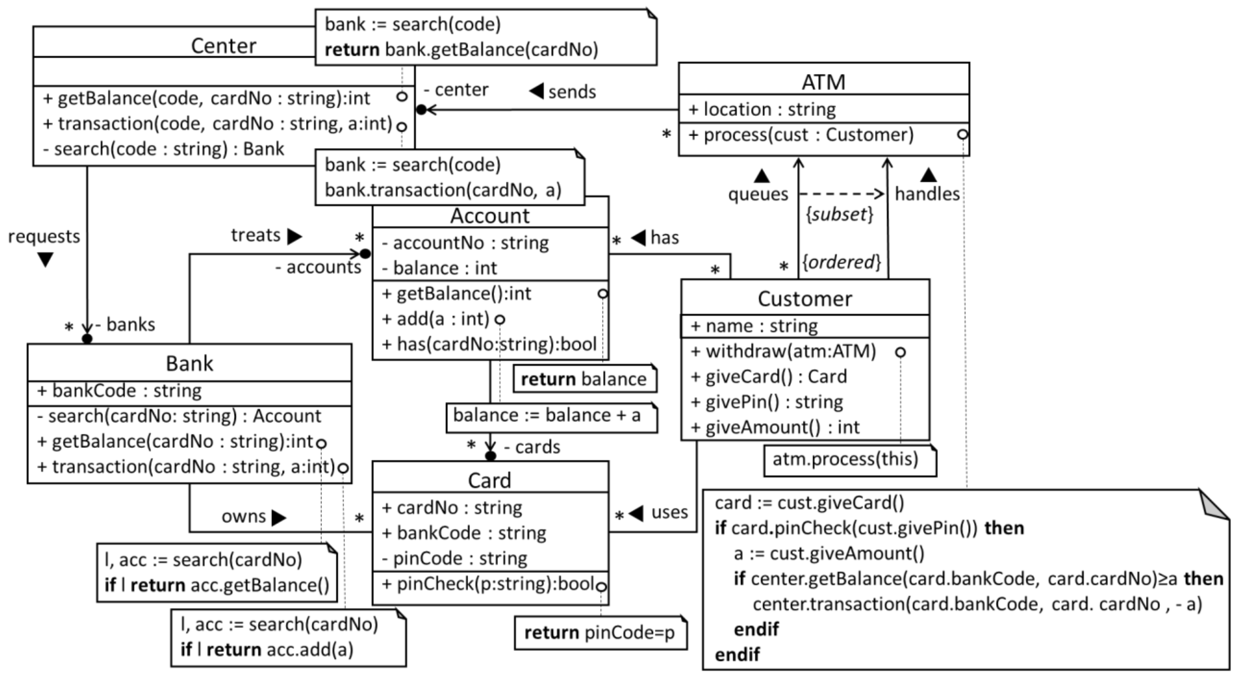
```
forall name in list
    l, product := SEARCH$_{p \in s.foods.stock}$ name = p.name
    if l then cart.insert(product)
              s.foods.remove(product)
forall name in list
    l, min, product := MIN$_{p \in s.technical.stock \{name = p.name\}}$ p.price
    if l then cart.insert(product)
              s.technical.remove(product)
```

The behaviour of the customer is implemented by the purchase() method of the Customer class. It contains a linear and a conditional maximumsearch.

We assume that all of the collections (cart, foods.stock, and technical.stock, each of which are Collection[Product]) are enumerable, and have an insert(), a remove(), and a clear() method.

There are three „name" variables that denote different things: the attribute of Customer, the attribute of Product, and the local variables used in the forall loops of purchase().

3. Clients are standing in line at an ATM to withdraw money from their accounts. The clients have debit cards, each of which has a PIN code and an account. The clients can withdraw money one after another by giving their card and code. If the given code is the same as the code of the card, the ATM gives out the money, as long as subtracting the amount from the balance yields a positive number. To check this, the ATM can query the balance of the client from a center by sending the data of the card to the center. The ATM can also send a report about the transaction to the center so the bank can subtract the amount from the account. Simulate the process of withdrawing from the ATM.



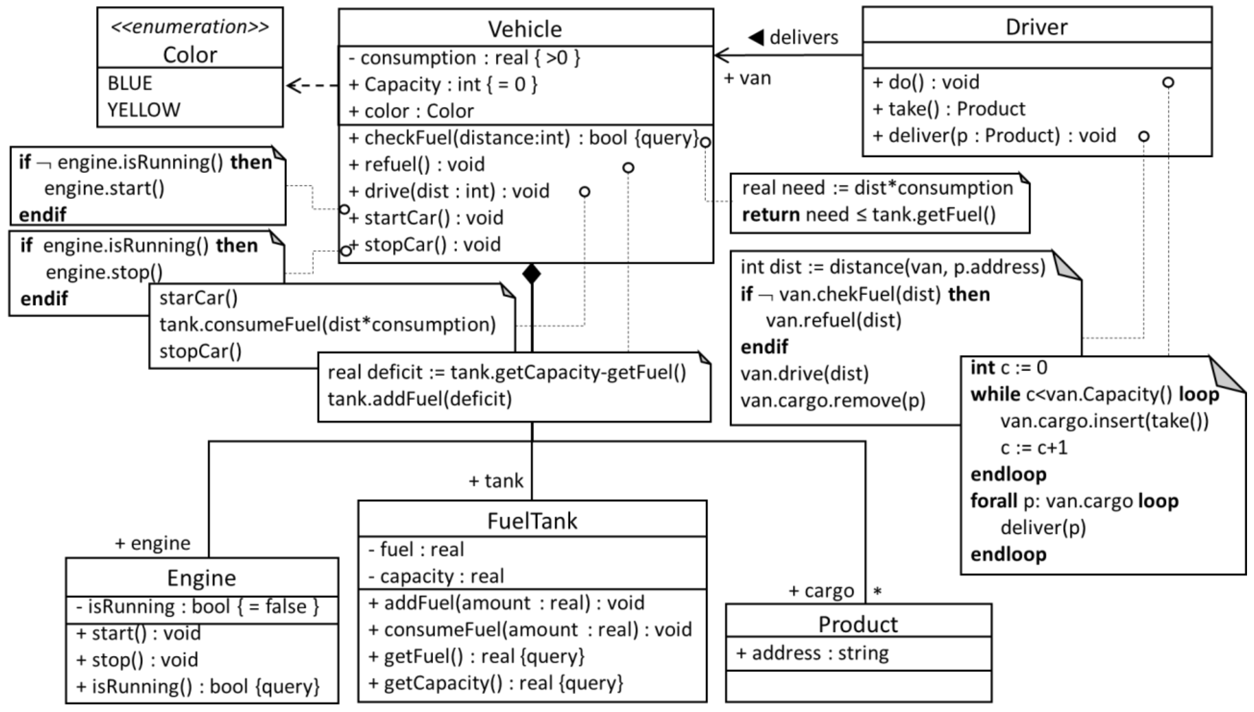
The solution starts by calling atm.process(), which is called by the withdraw() method of the client at the ATM. The process is as follows:

```
the ATM asks for the card from the client
it asks for the PIN
if the PIN is the same as the PIN of the card, then
    it asks for the amount to withdraw
    checks that the client has the funds to withdraw
    if the amount is not greater than the balance, then
        it notifies the bank about the transaction. The bank subtracts the amount from the
    balance.
```

Withdrawing money is realized along the sends, treats, owns associations, so we should make navigation easier by using role names. We should put getters, setters, and other helper methods inside the classes along the navigation route. The other relationships (has, uses, queues) are less important.

The giveXxx() methods are asking the client for some data: the card, the PIN code, the amount. The physical interface for this is provided by the ATM, which we don't detail. The pinCheck() method of Card does the checking of the PIN. The getBalance() and transaction() methods of Center and Bank serve the queries sent to the bank; they will activate the getBalance() and add() methods of the Account class in the end.

4. A package delivery driver loads as many products into his van as he can (considering the capacity of the van). The address of the destination is written on each product, so the distance (in kilometers) between the driver and the destination can be computed. The destinations are always gas stations (the PickPack points there), so the driver can refuel at each destination. The driver has to deliver all of the products that were put into his van. Delivering a poduct consists of the following steps: (1) the driver checks whether there is enough fuel to reach the destination. (2) if there is not enough fuel, the driver refuels the van. (3) the driver starts the engine, drives to the destination (during which the van consumes fuel according to its consumption rate), stops the van, and unloads the product. The van of the driver can be blue or yellow. The van has an engine, a fuel tank, and a hold to carry the products. The engine can be started and stopped. We can refuel the fuel tank up to its capacity. We know the consumption of the van, given in liters/km.


<<enumeration>>
Color
BLUE
YELLOW
Vehicle
- consumption : real { >0 }
+ Capacity : int { = 0 }
+ color : Color
+ checkFuel(distance:int) : bool {query}
+ refuel() : void
+ drive(dist : int) : void
+ startCar() : void
+ stopCar() : void
◀ delivers
+ van
Driver
+ do() : void
+ take() : Product
+ deliver(p : Product) : void
if ¬ engine.isRunning() then
engine.start()
endif
if engine.isRunning() then
engine.stop()
endif
starCar()
tank.consumeFuel(dist*consumption)
stopCar()
real deficit := tank.getCapacity-getFuel()
tank.addFuel(deficit)
real need := dist*consumption
return need ≤ tank.getFuel()
int dist := distance(van, p.address)
if ¬ van.chekFuel(dist) then
van.refuel(dist)
endif
van.drive(dist)
van.cargo.remove(p)
int c := 0
while c<van.Capacity() loop
van.cargo.insert(take())
c := c+1
endloop
forall p: van.cargo loop
deliver(p)
endloop
+ engine
+ tank
+ cargo
*
Engine
- isRunning : bool { = false }
+ start() : void
+ stop() : void
+ isRunning() : bool {query}
FuelTank
- fuel : real
- capacity : real
+ addFuel(amount : real) : void
+ consumeFuel(amount : real) : void
+ getFuel() : real {query}
+ getCapacity() : real {query}
Product
+ address : string

First, we create the classes with the most important attributes and their getters and setters. The associations of the classes only appear in rudimentary form. After this we describe the behavior of the driver starting from its do() method, and continue by partitioning it into different methods. During the partitioning we have to decide the steps each method consists of, and the classes they are part of. This is when the directions and role names of the associations are formed.