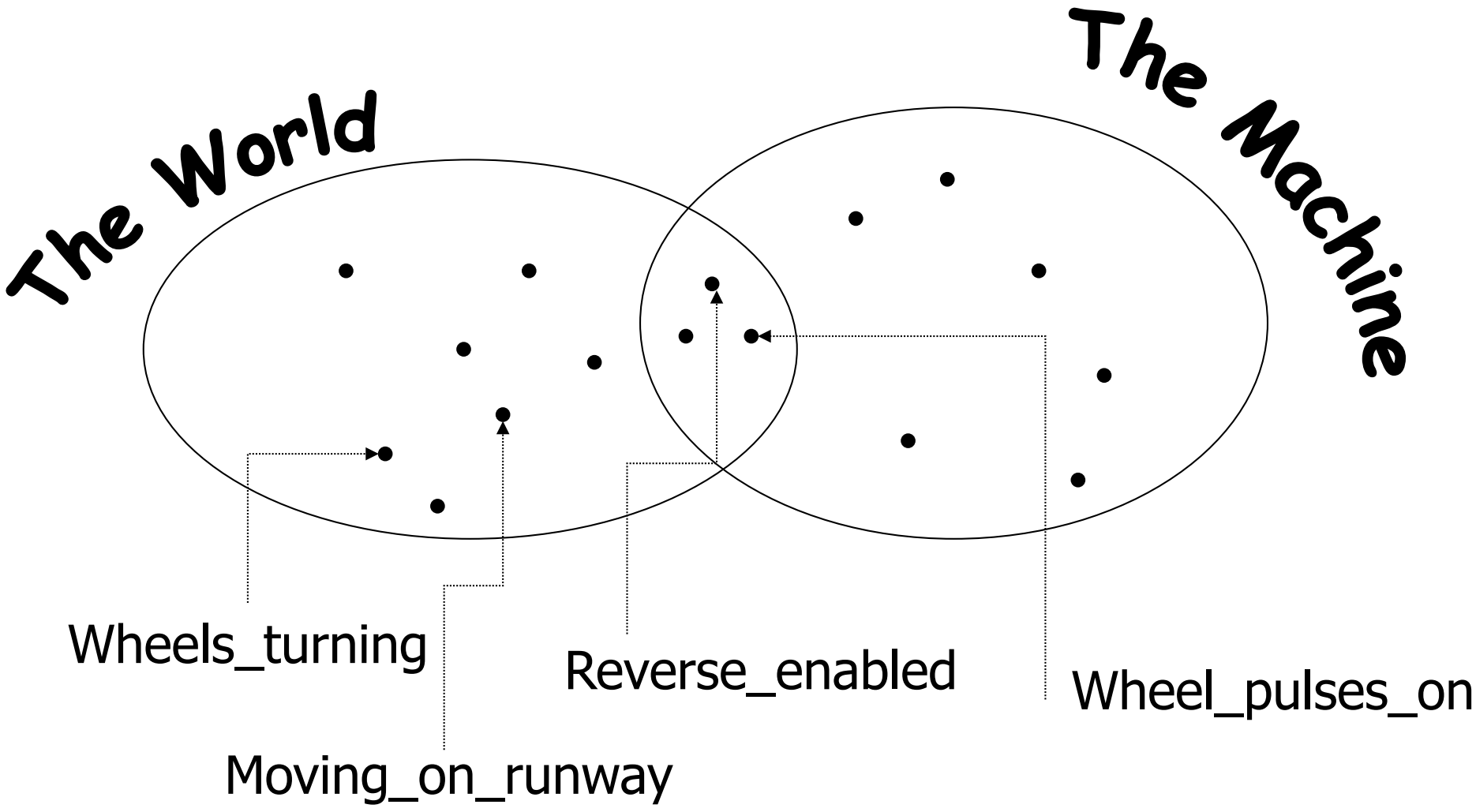
Politecnico
di Milano

Alloy, the world and the machine, and requirements – the Airbus case

Example – Airbus A320 Braking Logic



Modeling the Airbus braking logic with Alloy


```
abstract sig Bool {}
one sig True extends Bool {}
one sig False extends Bool {}

abstract sig AirCraftState {}
one sig Flying extends AirCraftState {}
one sig TakingOff extends AirCraftState {}
one sig Landing extends AirCraftState {}
one sig MovingOnRunaway extends AirCraftState {}
```


… for landing, we are not considering the movement due to takeoff as it is not relevant to our analysis

Modeling the Airbus braking logic with Alloy


```
sig Wheels {
        retracted: Bool,
        turning: Bool
}{turning = True implies retracted = False}

sig Aircraft {
        status: one AirCraftState,
        wheels: one Wheels,
        wheelsPulsesOn: one Bool,
        reverseThrustEnabled: one Bool
}{status = Flying implies wheels.retracted = True}
```

Modeling the Airbus braking logic with Alloy


```
fact domainAssumptions {
all a: Aircraft | a.wheelsPulsesOn = True
                  <=> a.wheels.turning = True
all a: Aircraft | a.wheels.turning = True
                  <=> a.status = MovingOnRunaway}

fact requirement {
all a: Aircraft | a.reverseThrustEnabled = True
                  <=> a.wheelsPulsesOn = True}

assert goal {
all a: Aircraft | a.reverseThrustEnabled = True
                  <=> a.status = MovingOnRunaway}
check goal
```

No counterexamples are found!

But note that, still, this is the wrong model of our world: the spec is internally coherent, but it does not correctly represent the world

Politecnico
di Milano

Alloy, the world and the machine, and requirements – the LuggageKeeper case

See also exam of February 16th, 2018

Informal description

- The company *TravelSpaces* decides to help tourists visiting a city in finding places that can keep their luggage for some time. The company establishes agreements with small shops in various areas of the city and acts as a mediator between these shops and the tourists that need to leave their luggage in a safe place.
- To this end, the company wants to build a system, called *LuggageKeeper*, that offers tourists the possibility to: look for luggage keepers in a certain area; reserve a place for the luggage in the selected place; pay for the service when they are at the luggage keeper; and, optionally, rate the luggage keeper at the end of the service.

(some) possible world and machine phenomena

- World phenomena
  - Some users own various pieces of luggage.
  - Some users carry around various pieces of luggage.
  - Some pieces of luggage are safe
  - Some pieces of luggage are unsafe.
  - Small shops store the luggage in lockers.
- Shared phenomena
  - Some lockers are opened with an electronic key.
  - Some users hold various electronic keys.

Alloy signatures

```
abstract sig Status{}
one sig Safe extends
Status{}
one sig Unsafe
        extends Status{}

sig Luggage{
  luggageStatus : one
Status
}

sig EKey{}
```

```
sig User{
  owns : set Luggage,
  carries : set Luggage,
  hasKeys : set EKey
}

sig Locker{
  hasKey : lone EKey,
  storesLuggage : lone Luggage
}

sig Shop{
  lockers : some Locker
}
```

Various constraints could be added
(e.g., the owner of a luggage is unique)

A domain assumption

- any piece of luggage is safe if, and only if, it is with its owner, or it is stored in a locker that has an associated key, and the owner of the piece of luggage holds the key of the locker

```
fact DAsafeLuggages {
  all lg : Luggage |
    lg.luggageStatus in Safe
    iff
    all u : User |
        lg in u.owns
        implies
        ( lg in u.carries
          or
          some lk : Locker | lg in lk.storesLuggage and
                             lk.hasKey != none and
                             lk.hasKey in u.hasKeys )
}
```

A requirement

- a key opens only one locker

```
fact requirement {
  all ek : EKey |
    no disj lk1, lk2: Locker |
       ek in lk1.hasKey and ek in lk2.hasKey
}
```

A goal

- for each user all his/her luggage is safe

```
pred goal {
  all u : User |
    all lg : Luggage |
        lg in u.owns
        implies
        lg.luggageStatus in Safe
}
```

Operation GenKey

- Given a locker that is free, GenKey associates with it a new electronic key

```
pred GenKey[lk, lk' : Locker] {
  //precondition
  lk.hasKey = none
  //postcondition
  lk'.storesLuggage = lk.storesLuggage
  one ek : EKey | lk'.hasKey = ek
}
```

Politecnico
di Milano

Milk Vending Machine

Identification of world and machine phenomena
From the exams of July 10th 2019 and June 25th 2019

Problem description

- A milk vending machine accepts 3 types of coins: 25¢ (cents), 50¢, and 1$. It accepts coins only in ascending order, for example, after inserting a 50¢ coin, it can only accept 50¢ or 1$ coins and it cannot accept 25¢ coins until the process is restarted by asking for the remaining change.
- Every time the amount of money in the machine reaches (or surpasses) 1$, it produces a bottle of milk, subtracts 1$ from the amount of money that is in the machine, and leaves the rest in the machine.

Problem description

- At any time, the user can ask for the money still in the machine (and not used to buy a bottle of milk) to be returned (this can occur even if there is no money remaining in the machine); the effect of this is that the process is restarted, so smaller coins can be introduced again.
- The vending machine also accepts fidelity cards. When the user inserts a fidelity card, he/she can buy a bottle of milk for 75¢ instead than 1$. Thus, if the machine receives the fidelity card and the current amount of money is equal to or surpasses 75¢, it produces the bottle of milk, subtracts 75¢ from the amount of money that is in the machine, and leaves the rest in the machine.

The task

- Referring to the Jackson-Zave distinction between the world and the machine, identify world, machine and shared phenomena. For these last ones specify which part (world or machine) is controlling them

Phenomenon	Shared	Who controls it
User wants to buy some milk	N	W
User inserts a coin in the machine	Y	W
The machine compares the inserted coin with the last received one	N	M
The machine rejects the inserted coin	Y	M
The machine accepts the inserted coin	Y	M
User inserts a fidelity card	Y	W
The machine checks and accepts the fidelity card	Y	M
The machine sees that amount needed to buy a bottle of milk is reached	N	M
The machine delivers the bottle of milk	Y	M
The machine updates the current amount of money	Y	M
The user goes home with the milk	N	W
The user wants to receive the money back	N	W
The user asks for the money back	Y	W
The machine delivers the amount of money to the user	Y	M
The machine resets the money count	N	M
The operator sets the current number of bottles in the machine	Y	W
A milk sensor signals the milk in the machine is finishing	Y	W
The machine decreases the counter of the current number of bottles	N	M
The machine goes out of service	Y	M

Model in Alloy the coins and the amount in the vending machine


```
abstract sig Coin {}
one sig Quarter extends Coin {} // a 25¢ coin
one sig Half extends Coin {} // a 50¢ coin
one sig Dollar extends Coin {} // a 1$ coin

abstract sig AmountMoney {} // collected amount
one sig Tot0 extends AmountMoney {} // amount of 0$
one sig Tot25 extends AmountMoney {} // amount of 25¢
one sig Tot50 extends AmountMoney {} // amount of 50¢
one sig Tot75 extends AmountMoney {} // amount of 75¢
```

computeRest function

computeRest takes as input an amount of money am and a coin c and, if the sum of am and the coin value is greater than or equal to 1$, it returns the rest of the sum with respect to 1$; otherwise, if the sum is less than 1$, it simply returns the sum.

```
fun computeRest[ am : AmountMoney, c : Coin ] : AmountMoney {
        { am' : AmountMoney |
            (am = Tot0 and c = Quarter and am' = Tot25) or
            (am = Tot0 and c = Half and am' = Tot50) or
            (am = Tot25 and c = Quarter and am' = Tot50) or
            (am = Tot25 and c = Half and am' = Tot75) or
            (am = Tot50 and c = Quarter and am' = Tot75) or
            (am = Tot50 and c = Half and am' = Tot0) or
            (am = Tot75 and c = Quarter and am' = Tot0) or
            (am = Tot75 and c = Half and am' = Tot25) or
            (c = Dollar and am' = am)
        }
}
```

Other Alloy elements

- Formalize in Alloy the following elements:
  - A signature VendingMachineState describing the state of the vending machine.
  - A predicate leqCoin that takes as input two coins, c1 and c2, and returns true if c1 is less than, or equal to, c2, false otherwise.
  - A predicate describing operation insertCoin, which, given a vending machine, takes the value of the inserted coin, and changes the state of the vending machine accordingly.
  - A predicate describing operation getMoneyBack, which, given a vending machine, changes its state and produces an AmountMoney with the amount of money that was present in the machine.

VendingMachineState and leqCoin


```
sig VendingMachineState {
  money: AmountMoney,
  minCoin : Coin
}

pred leqCoin[ c1, c2 : Coin ] {
  c1 = Quarter or
  c1 = Half and not (c2 = Quarter) or
  c1 = Dollar and c2 = Dollar
}
```

insertCoin and getMoneyBack


```
pred insertCoin[ s, s': VendingMachineState, c : Coin ]
{
  leqCoin[s.minCoin, c] and
  s'.money = computeRest[s.money, c] and
  s'.minCoin = c
}

pred getMoneyBack[ s, s': VendingMachineState, res :
AmountMoney ] {
  s'.money = Tot0 and
  s'.minCoin = Quarter and
  res = s.money
}
```

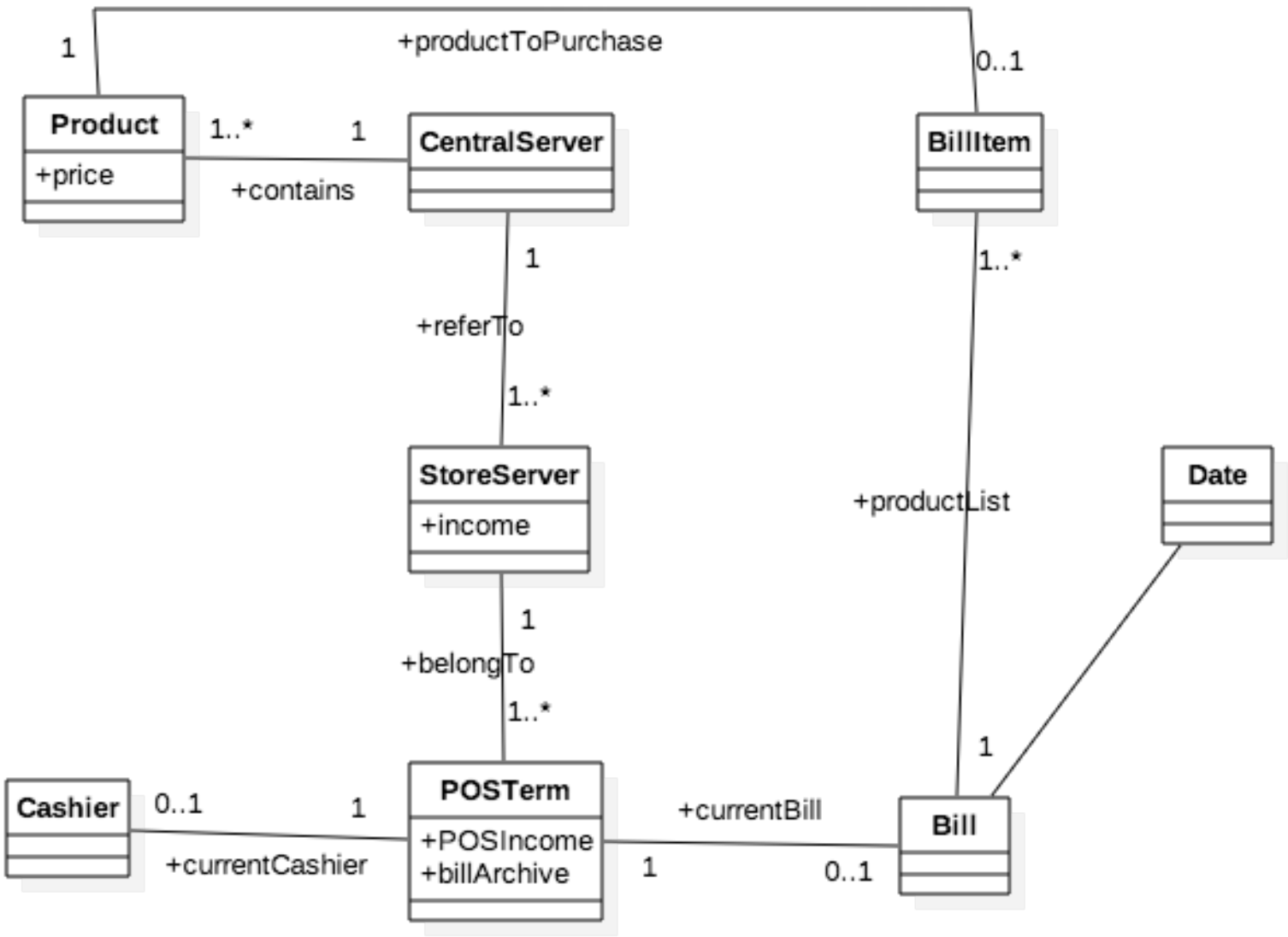
Politecnico
di Milano

POS: Reasoning on how to combine UML and Alloy

POS

- A supermarket wants to build a new POS (Point Of Sale) system to support the work of cashiers
- The next slide describes in UML the domain model.
- Translate this model into an Alloy specification. For simplicity, assume that the price of each product is equal to one.
- Specify in Alloy the following operations
  - Add an item to a bill
  - Cashier login

POS – class diagram



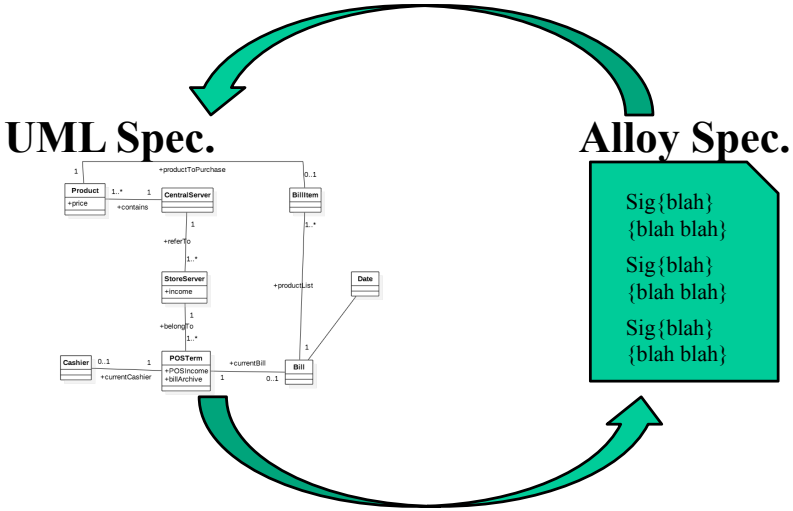
POS - signatures


```
open util/integer as integer

sig Date {}
sig Cashier {}

sig Product {
  price: Int
}
{price = 1}

sig CentralServer {
  contains: some Product
}

sig Store {
  refersTo: one CentralServer,
  runningIncome: Int
}
```

POS – more signatures

```
sig BillItem {
  productToPurchase: Product,
  quantity: Int
}{quantity > 0}

sig Bill {
  productsList: set BillItem,
  price: Int,
  date: Date
}{
  #productsList > 0
  price >= 0
}

sig POSTerm {
  belongsTo: one Store,
  POSrunningIncome: Int,
  currentBill: lone Bill,
  billArchive: set Bill,
  currentCashier: lone Cashier
}
```

"precise specification of structural feats…"



POS - facts

```
fact cashierUsesOnePOSAtATime {
  no c: Cashier | some t1, t2: POSTerm |
         t1 != t2 and c in t1.currentCashier and
         c in t2.currentCashier
}

fact billCanBeAssociatedWithOnlyOnePOSTerm {
  no b: Bill | some t1, t2: POSTerm |
     t1 != t2 and
      (b in t1.currentBill or b in t1.billArchive)
      and
      (b in t2.currentBill or b in t2.billArchive)
}
```

POS - operations

```
pred addItemToBill[b, b': Bill, i: BillItem]
{
  (b'.productsList = b.productsList + i) &&
  (b'.price = add[b.price, mul[i.productToPurchase.price,i.quantity]]) &&
  (b'.date = b.date)
}
```

addition multiplication

```
pred loginCashier[p, p': POSTerm, c: Cashier]
{
  // precondition
  #p.currentCashier = 0 &&
  // postcondition
  (p'.currentCashier = p.currentCashier + c) &&
  (p'.belongsTo = p.belongsTo) &&
  (p'.POSrunningIncome = p.POSrunningIncome) &&
  (p'.currentBill = p.currentBill) &&
  (p'.billArchive = p.billArchive)
}
```

POS – commands to run


```
pred show {
 #POSTerm > 1
}

run show for 3 but 3 Product, 2 Store,
                    2 POSTerm, 2 Bill
run addItemToBill
run loginCashier
```

Politecnico
di Milano

More on Alloy + Requirements Engineering: the Electric vehicle sharing

Electric vehicle sharing

A company operates a system for sharing electric vehicles. The vehicles can be cars, scooters and electric bicycles. The company provides its users with the charging stations for vehicles, and with a number of vehicles. Each user can either rent a vehicle (only one at a time), or use the charging stations provided by the company to recharge his vehicle. The company handles bookings for both vehicles and charging stations.

A. Define an Alloy model for the above domain. Specifically, define the facts modeling the following constraints:

- A vehicle cannot be used by multiple users simultaneously.
- A charging station has a finite capacity (defined by the number of electrical outlets available). The number of vehicles using the station cannot exceed this capacity.
- A user with a disability cannot use scooters or bicycles.
- A vehicle in charge cannot be (at the same time) used by a user.

B. Define the predicate acquireVehicle that, given a vehicle, models its acquisition by a user.

What do we do now?

Step. 1 – identify signatures

A **company** operates a system for sharing electric vehicles. The **vehicles** can be cars, scooters and electric bicycles. The company provides its users with the **charging stations** for vehicles, and with a number of vehicles. Each user can either rent a vehicle (only one at a time), or use the charging stations provided by the company to recharge his vehicle. The company handles bookings for both vehicles and charging stations.

A. Define an Alloy model for the above domain. Specifically, define the facts modeling the following constraints:

- A vehicle cannot be used by multiple users simultaneously.
- A charging station has a finite capacity (defined by the number of electrical outlets available). The number of vehicles using the station cannot exceed this capacity.
- A user with a disability cannot use scooters or bicycles.
- A vehicle in charge cannot be (at the same time) used by a user.

B. Define the predicate acquireVehicle that, given a vehicle, models its acquisition by a user.

Signatures


```
one sig Company {
  stations: set ChargingStation,
  vehicles: set Vehicle
}

sig Vehicle {}

sig ChargingStation {
  capacity: some Plug,
  chargingVehicles: set Vehicle
}
{#chargingVehicles <= #capacity}
```

Signatures


```
one sig Company {
  stations: set ChargingStation,
  vehicles: set Vehicle
}

sig Vehicle {}

sig ChargingStation {
  capacity: some Plug,
  chargingVehicles: set Vehicle
}
{#chargingVehicles <= #capacity}
```

Are we done here???

Step. 1b – relate signatures together

A company operates a system for sharing electric vehicles. The **vehicles** can be **cars**, **scooters** and **electric bicycles**. The company provides its users with the charging stations for vehicles, and with a number of vehicles. Each user can either rent a vehicle (only one at a time), or use the charging stations provided by the company to recharge his vehicle. The company handles bookings for both vehicles and charging stations.

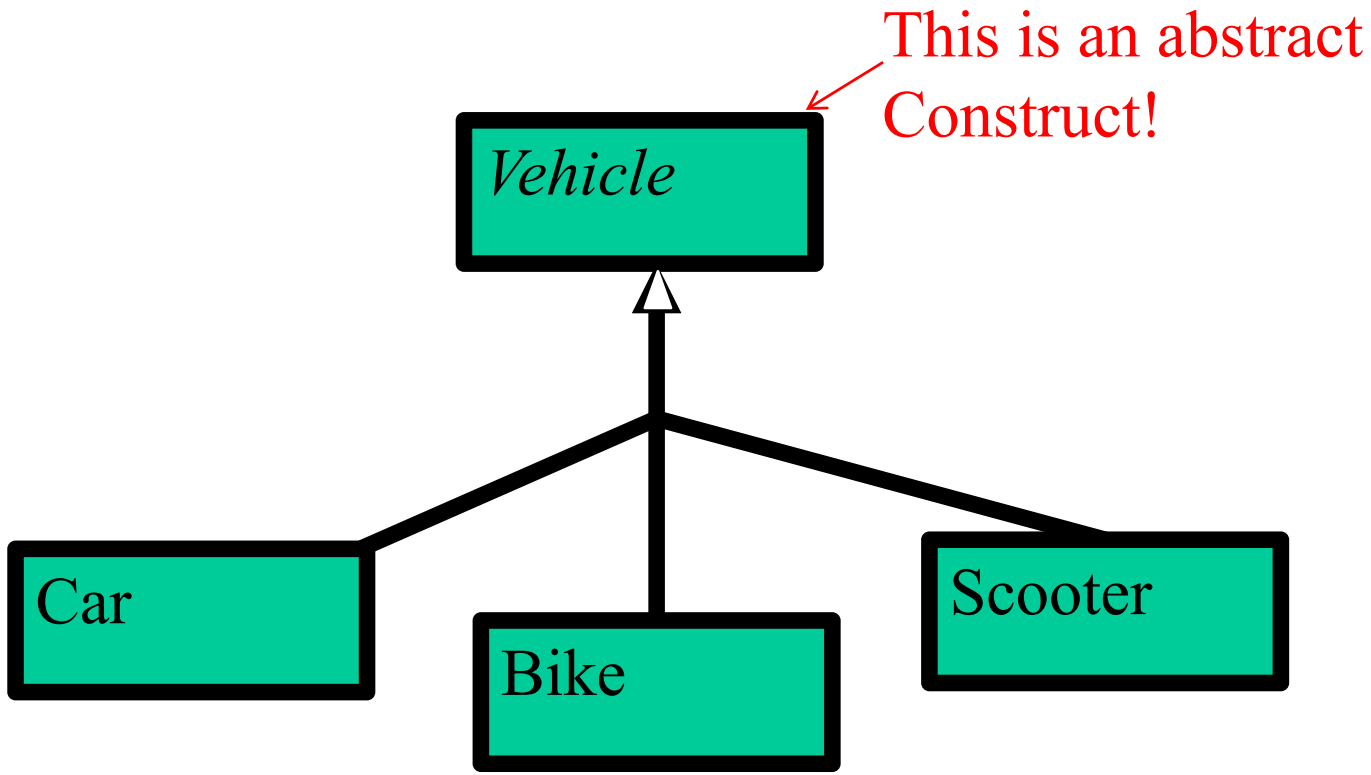
A. Define an Alloy model for the above domain. Specifically, define the facts modeling the following constraints:

- A vehicle cannot be used by multiple users simultaneously.
- A charging station has a finite capacity (defined by the number of electrical outlets available). The number of vehicles using the station cannot exceed this capacity.
- A user with a disability cannot use scooters or bicycles.
- A vehicle in charge cannot be (at the same time) used by a user.

B. Define the predicate acquireVehicle that, given a vehicle, models its acquisition by a user.

Step. 3 – relate signatures together

- Arguably, this relation holds:



Signatures

- The correct signatures are, therefore:

```
abstract sig Vehicle {}
sig Car extends Vehicle{}
sig Scooter extends Vehicle{}
sig Bike extends Vehicle{}
```

Step. 2 – identify more signatures!

A **company** operates a system for sharing electric vehicles. The **vehicles** can be cars, scooters and electric bicycles. The company provides its **users** with the **charging stations** for vehicles, and with a number of vehicles. Each user can either rent a vehicle (only one at a time), or use the charging stations provided by the company to recharge his vehicle. The company handles bookings for both vehicles and charging stations.

A. Define an Alloy model for the above domain. Specifically, define the facts modeling the following constraints:

- A vehicle cannot be used by multiple users simultaneously.
- A charging station has a finite capacity (defined by the number **of electrical outlets** available). The number of vehicles using the station cannot exceed this capacity.
- A user with a **disability** cannot use scooters or bicycles.
- A vehicle in charge cannot be (at the same time) used by a user.

B. Define the predicate acquireVehicle that, given a vehicle, models its acquisition by a user.

Hint: some signatures are often implicit!

Signatures


```
sig Disability{}
sig User {
  ownedVehicles: set Vehicle,
  usedVehicle: lone Vehicle,
  disability: lone Disability
}
{
  disability != none implies
  (usedVehicle & Scooter = none and
  usedVehicle & Bike = none)
}
```

Signatures


```
sig Disability{}
sig User {
  ownedVehicles: set Vehicle,
  usedVehicle: lone Vehicle,
  disability: lone Disability
}
{
  disability != none implies
  (usedVehicle & Scooter = none and
  usedVehicle & Bike = none)
}
sig Plug{}
```

Don’t forget the electrical outlet!

And now… the facts!

- What do I need to check?
  - Let's take a look at the scenario description…

Electric vehicle sharing

A company operates a system for sharing electric vehicles. The vehicles can be cars, scooters and electric bicycles. The company provides its users with the charging stations for vehicles, and with a number of vehicles. Each user can either rent a vehicle (only one at a time), or use the charging stations provided by the company to recharge his vehicle. The company handles bookings for both vehicles and charging stations.

A. Define an Alloy model for the above domain. Specifically, define the facts modeling the following constraints:

- A vehicle cannot be used by multiple users simultaneously.
- A charging station has a finite capacity (defined by the number of electrical outlets available). The number of vehicles using the station cannot exceed this capacity.
- A user with a disability cannot use scooters or bicycles.
- A vehicle in charge cannot be (at the same time) used by a user.

B. Define the predicate acquireVehicle that, given a vehicle, models its acquisition by a user.

Electric vehicle sharing

A company operates a system for sharing electric vehicles. The vehicles can be cars, scooters and electric bicycles. The company provides its users with the charging stations for vehicles, and with a number of vehicles. **Each user can either rent a vehicle (only one at a time)**, **or use the charging** stations provided by the company to recharge his vehicle. The company handles bookings for both vehicles and charging stations.

A. Define an Alloy model for the above domain. Specifically, define the facts modeling the following constraints:

- **A vehicle cannot be used by multiple users simultaneously**.
- A charging station has a finite capacity (defined by the number of electrical outlets available). **The number of vehicles using the station cannot exceed this capacity.**
- **A user with a disability cannot use scooters or bicycles.**
- **A vehicle in charge** can**not** be (at the same time) **used by a user**.

B. Define the predicate acquireVehicle that, given a vehicle, models its acquisition by a user.

Facts and predicates


```
fact noTwoUsersExploitingAVehicle {
  no disjoint u1, u2: User |
     u1.usedVehicle = u2.usedVehicle
}
fact noChargingVehicleInUse {
  no c: ChargingStation, u: User |
     c.chargingVehicles & u.usedVehicle != none
}
pred compatibleVehicle[u: User, v: Vehicle]{
  u.disability != none implies not
  (v in Scooter or v in Bike)
}
```

Electric vehicle sharing

A company operates a system for sharing electric vehicles. The vehicles can be cars, scooters and electric bicycles. The company provides its users with the charging stations for vehicles, and with a number of vehicles. **Each user can either rent a vehicle (only one at a time)**, **or use the charging** stations provided by the company to recharge his vehicle. The company handles bookings for both vehicles and charging stations.

A. Define an Alloy model for the above domain. Specifically, define the facts modeling the following constraints:

- **A vehicle cannot be used by multiple users simultaneously**.
- A charging station has a finite capacity (defined by the number of electrical outlets available). **The number of vehicles using the station cannot exceed this capacity.**
- **A user with a disability cannot use scooters or bicycles.**
- **A vehicle in charge** cannot be (at the same time) **used by a user**.

B. Define the predicate acquireVehicle that, given a vehicle, models its acquisition by a user.

So… More Predicates


```
pred freeVehicle[v: Vehicle] {
    all c: ChargingStation, u: User |
        not (v in c.chargingVehicles) and
        not (v in u.usedVehicle) and
        not (v in u.ownedVehicles)
}
pred acquireVehicle[u: User, v: Vehicle, u':
User] {
    compatibleVehicle[u, v] and
    (v in u.ownedVehicles or freeVehicle[v]) and
    u.usedVehicle = none
    u'.ownedVehicles = u.ownedVehicles and
    u'.disability = u.disability and
    u'.usedVehicle = v
}
```

And a few Facts!

- We need to model what shall never happen by design!

```
fact noTwoOwners {
  no disjoint u1, u2: User |
  u1.ownedVehicles & u2.ownedVehicles != none

  no u: User |
  u.ownedVehicles & Company.vehicles != none
}
```

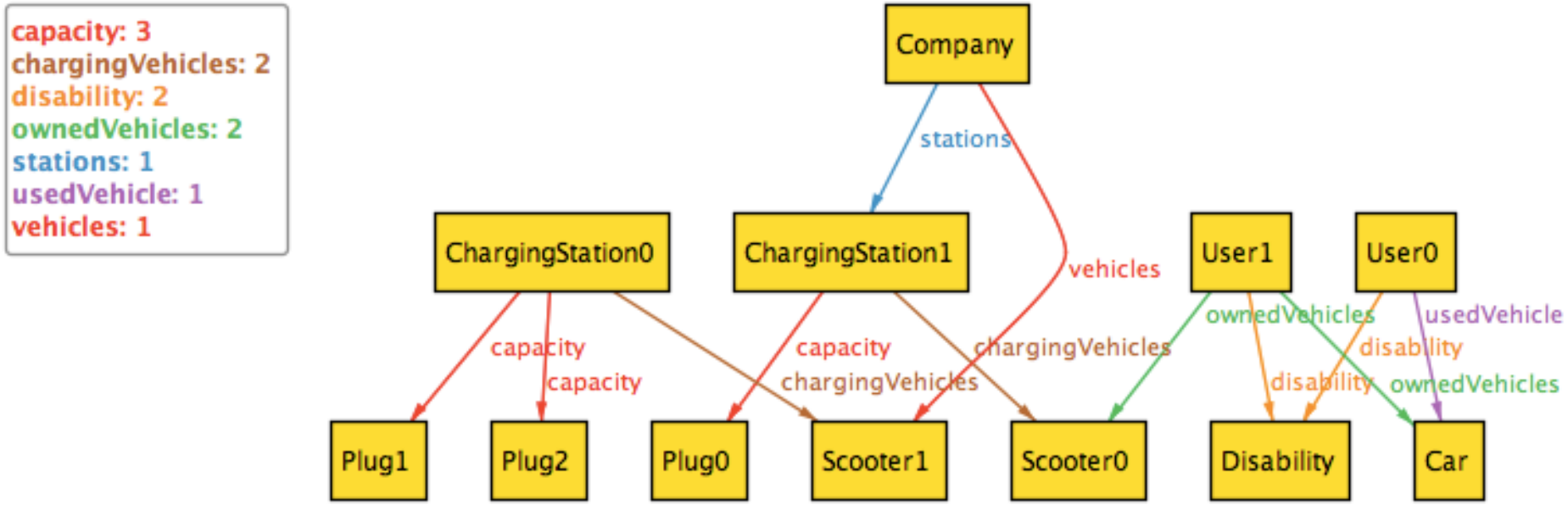
Hint: Do not forget any structural integrity facts that may remain implicit in the specs but are needed for the model to properly reflect reality…

Finally, we use the tool…


```
pred show(){}

run show

run freeVehicle
run compatibleVehicle
run acquireVehicle
```

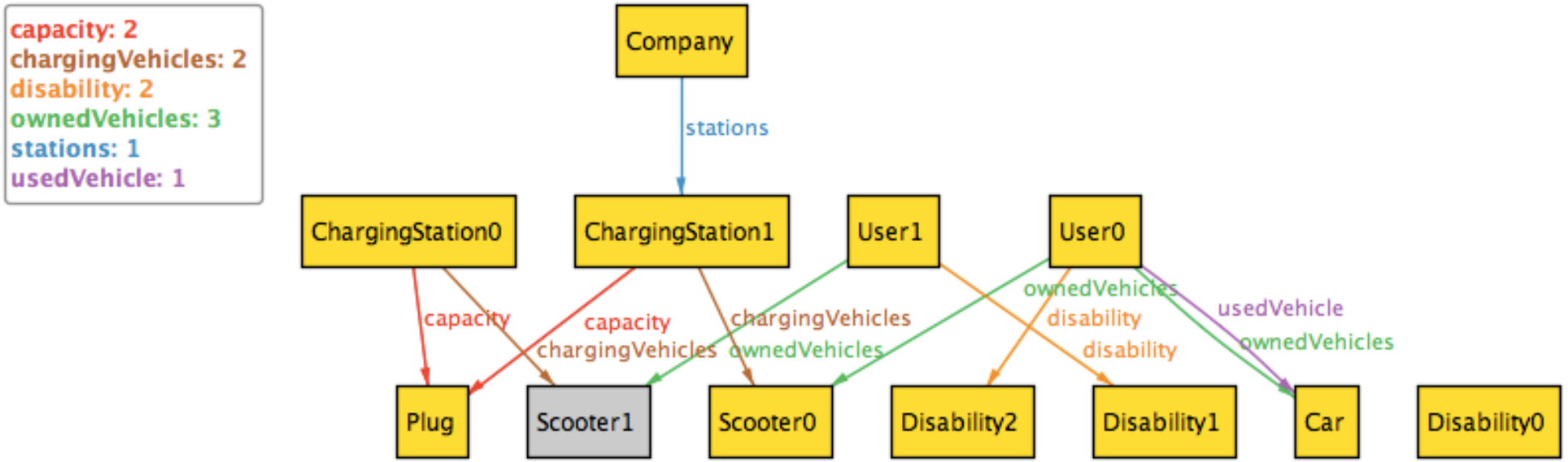
Hint: Do not forget the execution clauses and the appropriate scope… these are critical parts of the specs!!!

Analysis of possible worlds



- A user should not use another person's car

More on analysis of possible worlds



- A plug should belong to a single charging station

A user should not use another person's car


```
fact ownedVehiclesCannotBeUsedByOthers {
   all v: Vehicle, u: User |
       v in u.ownedVehicles implies
          (no u2: User | u2!=u and
              v in u2.usedVehicle)
}
```

A plug should belong to a single charging station


```
fact noSharedPlugsBetweenStations {
      all p: Plug | #capacity.p <=1
}
```

Politecnico
di Milano

One more case: taxi sharing

(a critical analysis of a pre-existing specification)

Taxi sharing (1)

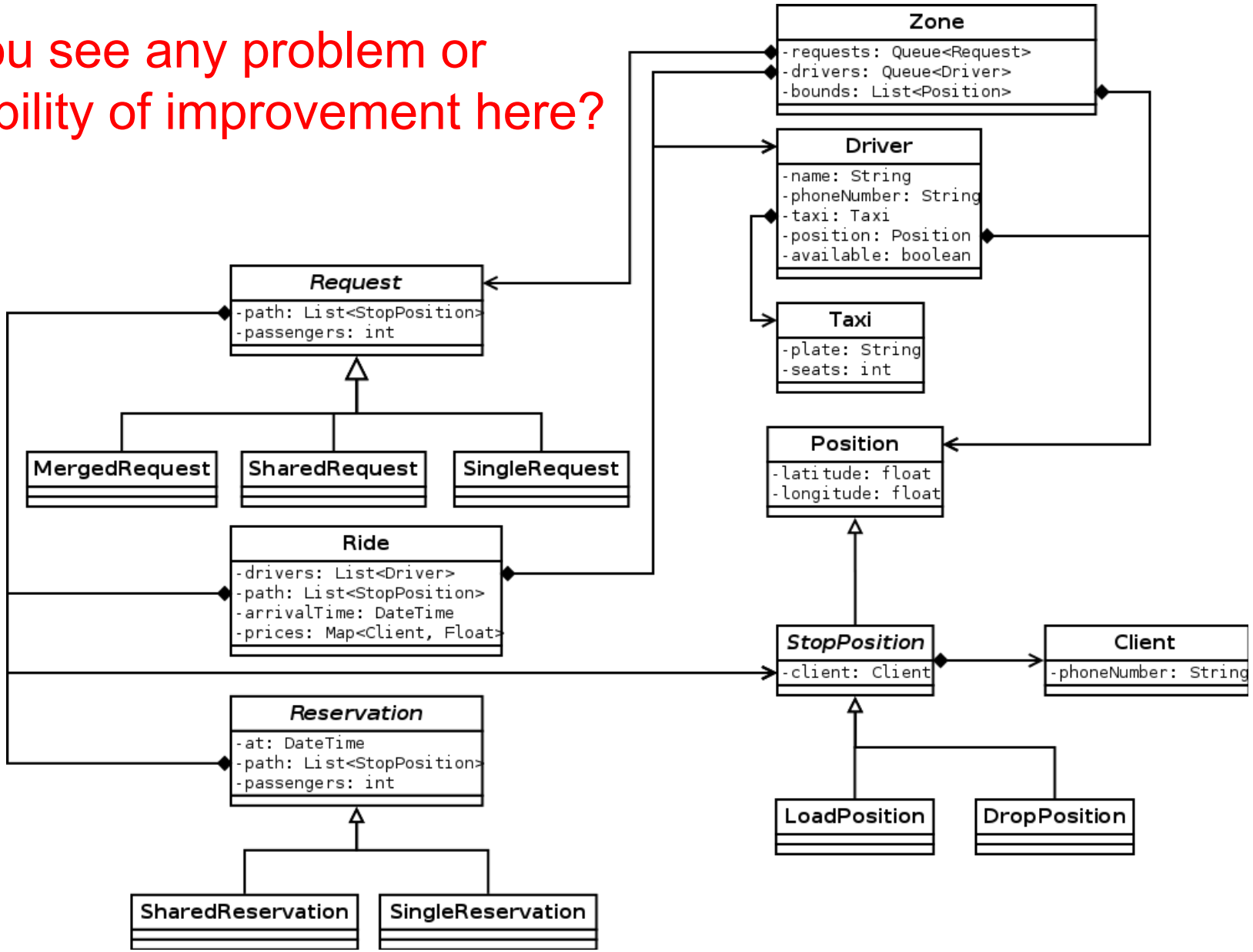
- We will design and implement myTaxiService, which is a service based on a mobile application and a web application, with two different targets of people:
  - (1) **taxi drivers**;
  - (2) **clients**;
- The system allows clients to reserve a **taxi** via mobile or web app, using the GPS position to identify the client's **zone** (but the client can insert it manually) and find a taxi in the same zone. On the other side the mobile app allows taxi drivers to accept or reject a **ride** request and to communicate automatically their position (so the zone).
- The clients are not registered since the company wants a system that can be used immediately and it is worried that, if there is a registration, many potential clients will not use the app. So the clients must insert their **name** and **phone number** each time (this is faster than creating an account and logging each time).

Taxi sharing (2)

- The system includes extra services and functionalities such as taxi **sharing** and **reservations**.
- With taxi sharing the client gives his/her availability to share the ride with other people. If there are others that give the same availability and have compatible destinations, then the system defines a plan for the shared ride and sends it to the taxi driver assigned to it.
- Reservations are requests for future rides that should occur at least three hours after the reservation is issued.

Class diagram

Do you see any problem or possibility of improvement here?



Class diagram

- Names and cardinality of associations are missing
- Not all associations should be inclusions
- The client is not associated with requests and reservations

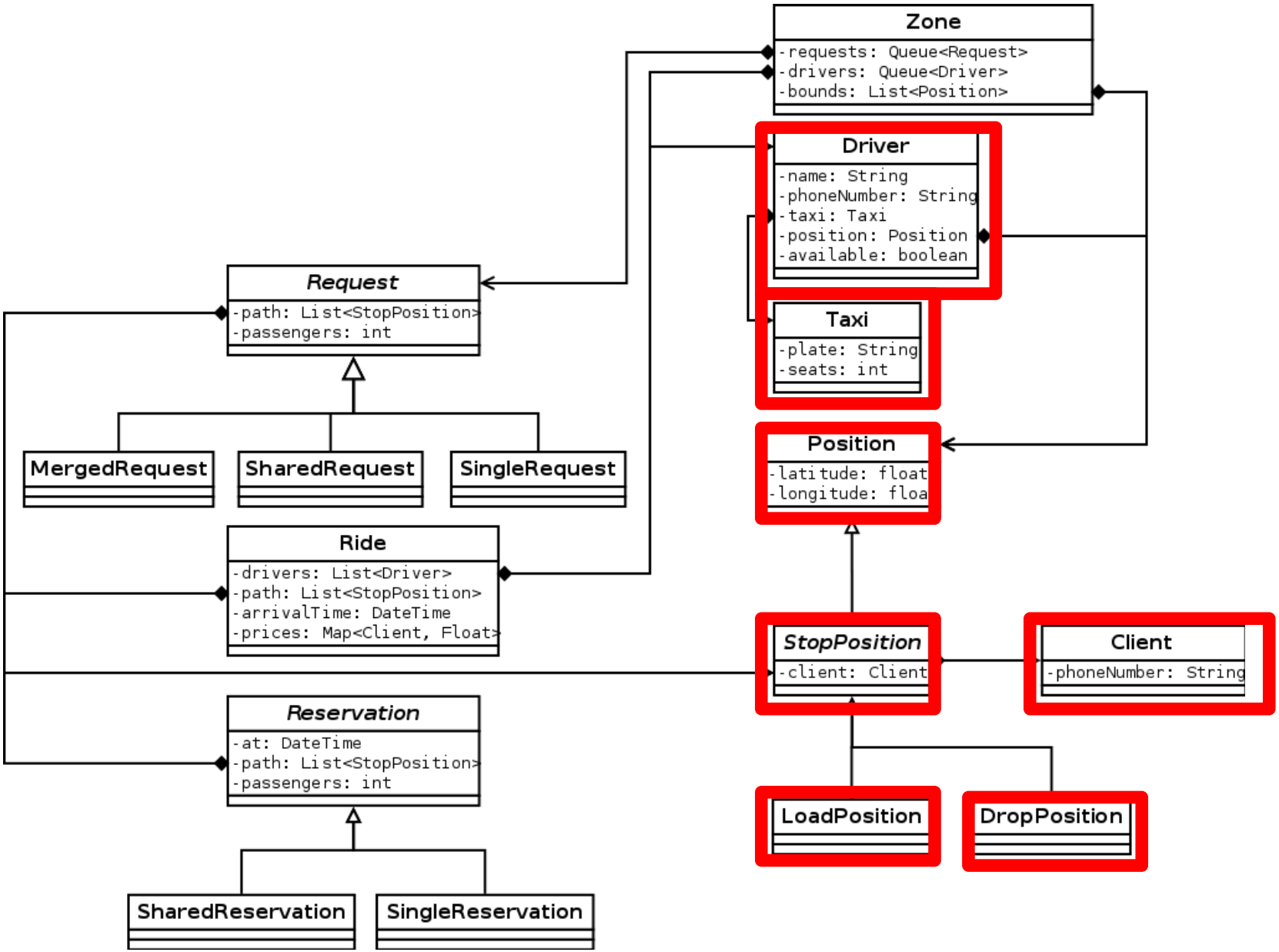
Alloy: signatures


```
// this is a support clause
      calling for additional libraries
open util/boolean
sig Taxi{
  code: Int,
  seats: Int}
{code > 0 seats > 0 seats <= 4}

sig Driver {
  taxi: one Taxi,
  available: one Bool}

sig PhoneNumber { }
```

Alloy: signatures


```
sig Client { phoneNumber: PhoneNumber }
abstract sig Position {
  latitude: Int,
  longitude: Int
}
abstract sig StopPosition extends Position {
  client: one Client
}
sig LoadPosition extends StopPosition { }
sig DropPosition extends StopPosition { }
```

Class diagram and signatures



More signatures

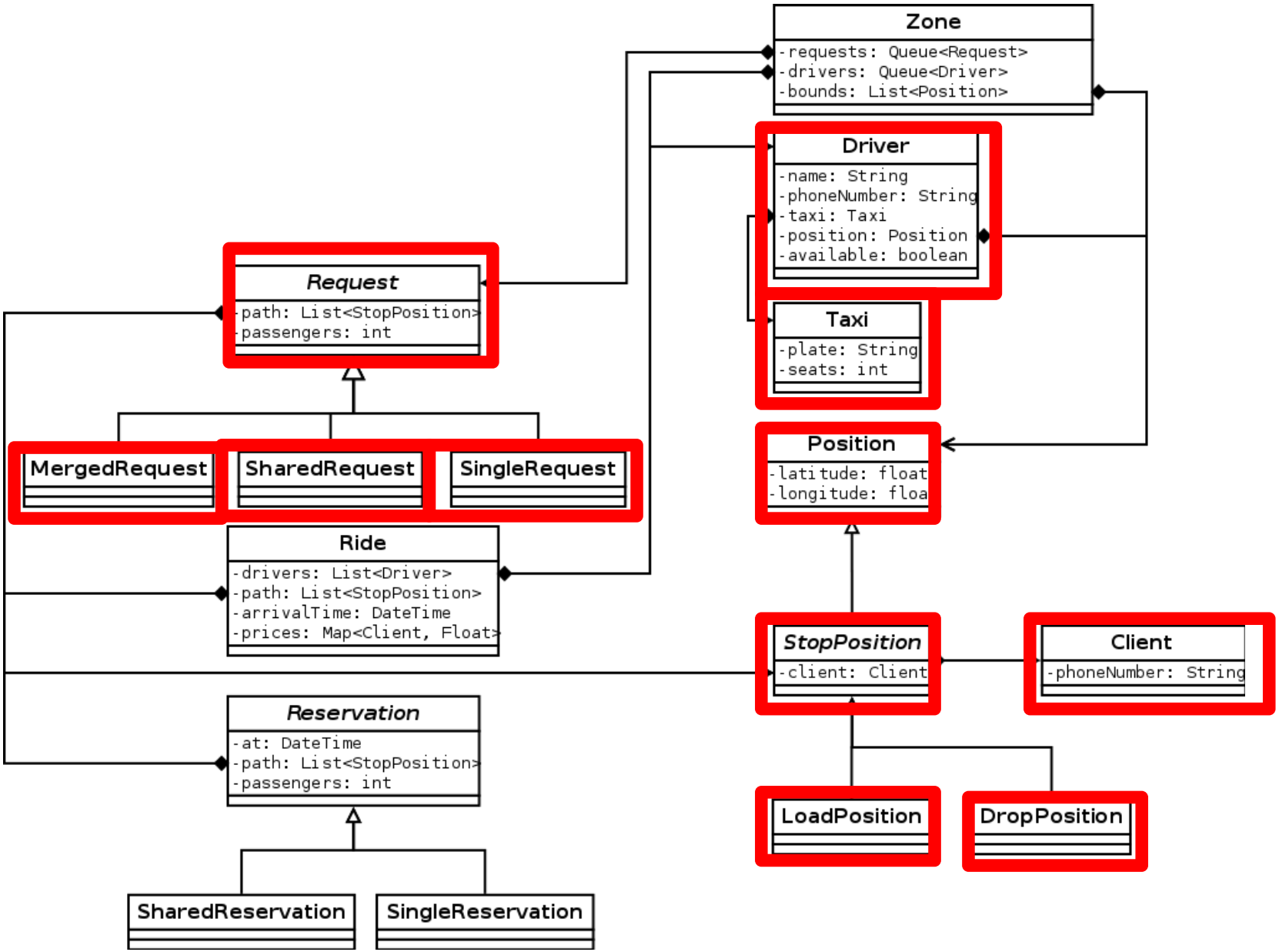
```
sig Path {
  positions: seq Position }
{
  #positions >= 2
}
abstract sig Request {
  path: Path, passengers: Int
}
{
  passengers > 0
}
sig SingleRequest extends Request { }
sig SharedRequest extends Request { }
sig MergedRequest extends Request { }
```

See next slide

seq

- `seq`: new operator used for declaring a field as a sequence of atoms
- If `p: Path`:
  - `p.positions` is a binary relation from `Int` to `Position`
  - Access to the elements of the sequence looks syntactically similar to access to arrays:
    - `p.positions[0]` is the position corresponding to the integer 0 in the sequence -> 0 is one of the indexes of the sequence
  - `p.positions.inds` is the set including all indexes in the sequence
  - `univ.(p.positions)` is the set including all elements in the sequence
  - `(p.positions).hasDups` is true if there are duplicates in the sequence

Class diagram... cont'd



More signatures


```
sig DriverQueue {
  dq: seq Driver
}{
   not dq.hasDups
}
sig RequestQueue {
  rq: seq Request
}{   not rq.hasDups
}
```

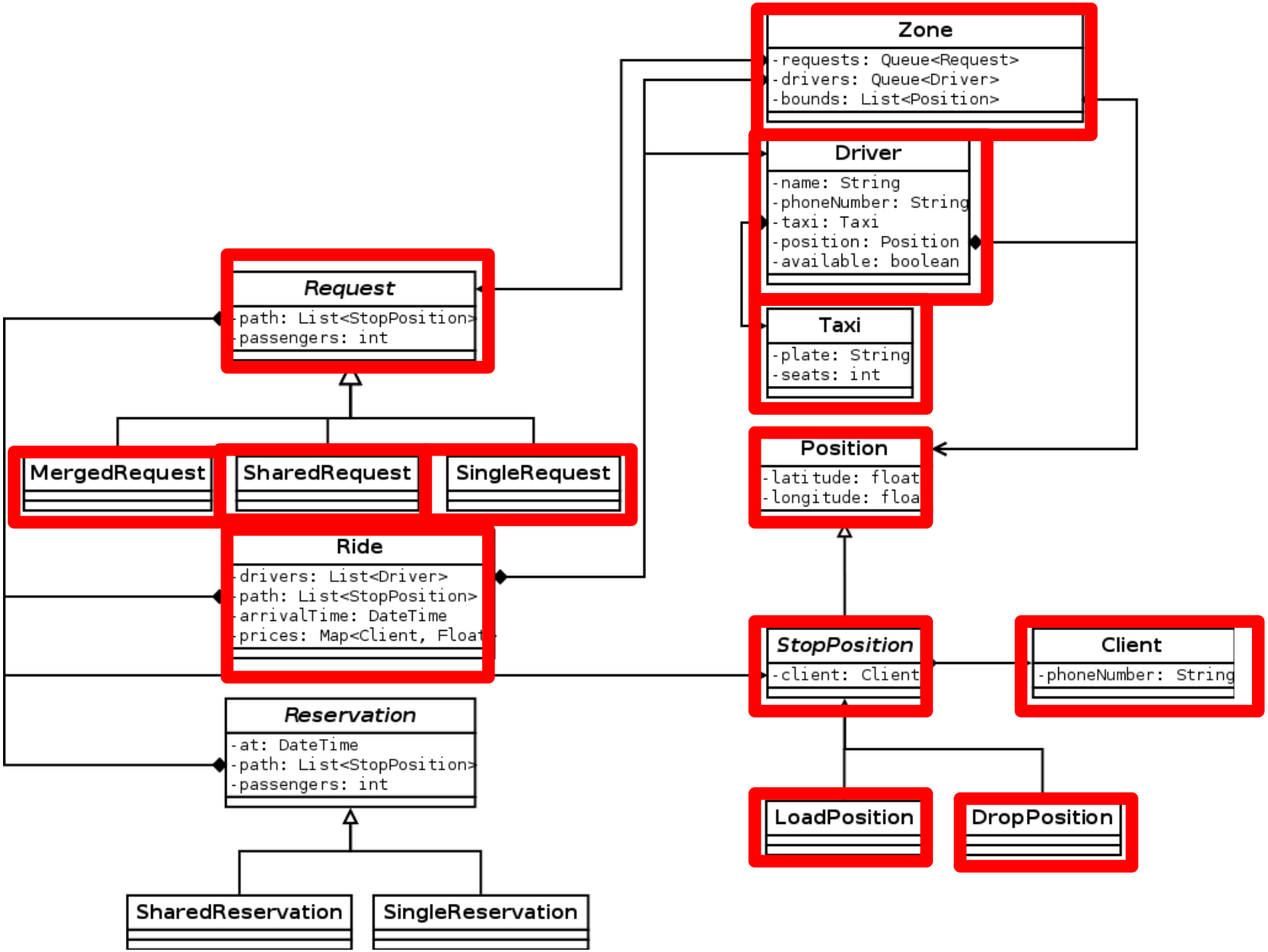
More signatures


```
sig Zone {
  drivers: one DriverQueue,
  requests: one RequestQueue,
  bounds: seq Position
}{
  #bounds > 2
  not bounds.hasDups
}
```

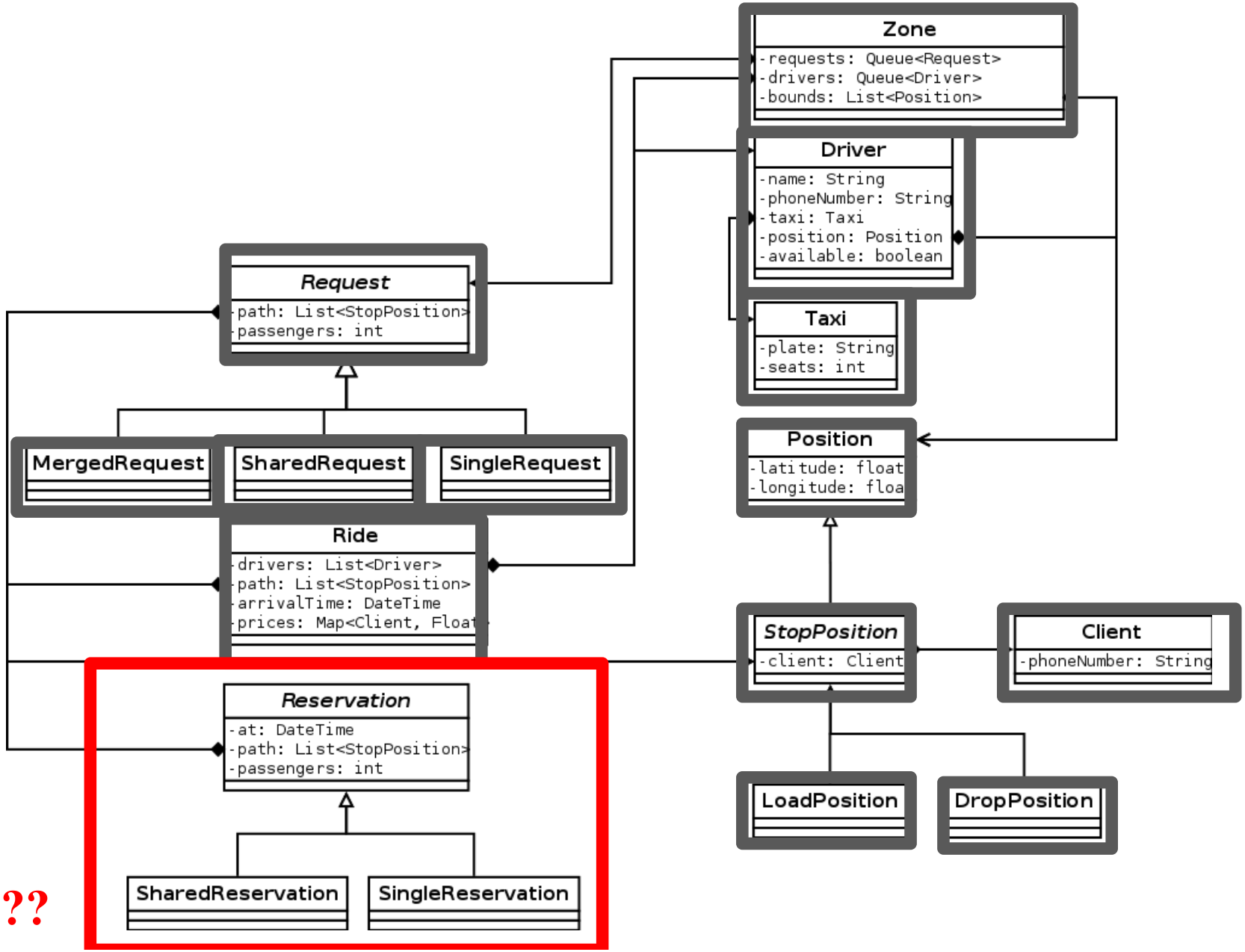
More signatures


```
sig Ride {
  drivers: some Driver,
  path: Path,
  prices: Client -> Int
}
one sig TaxiCentral {
  drivers: some Driver,
  clients: some Client,
  zones: some Zone
}
```

Class diagram... cont'd



Class diagram... cont'd



How about these???

Some facts


```
fact codeAreUnique {
  all disj t1,t2: Taxi | t1.code != t2.code }

fact allTaxiAreOwned { Driver.taxi = Taxi }

fact taxiHaveOnlyOneDriver {
   all disj d1, d2:Driver | d1.taxi != d2.taxi }

fact pathPositionsAreUnique {
  all p: Path | not p.positions.hasDups
}
```

Some facts

```
pred complement[drop, load: StopPosition] {
  drop != load
  drop.client = load.client
  ((drop in DropPosition) <=> (load in LoadPosition))
}
```

The set that includes all indexes of the sequence

```
fact pathClientHasALoadAndADrop {
all p: Path | all i1: p.positions.inds |
one i2: p.positions.inds | i2 != i1 and
let p1 = p.positions[i1] | let p2 = p.positions[i2] |
p1.complement[p2] and not ( one i3: p.positions.inds |
i3 != i2 and i3 != i1 and let p3=p.positions[i3] |
p3.complement[p1] or p3.complement[p2])
}
```

Some facts


```
fact pathStartWithLoad {
all p: Path | p.positions.first in
LoadPosition
}
fact pathEndWithDrop {
all p: Path | p.positions.last in
DropPosition
}
```

Some assertions


```
assert clientGetInAndGetOut {
  all p: Path| all i1: p.positions.inds |
      one i2: p.positions.inds |
      i2 != i1 and let p0=p.positions[i1]
      | let p1=p.positions[i2] |
        p0.complement[p1]
}

assert pathSameNumberOfLoadAndDrop {
  all p: Path | #(p.positions.elems & DropPosition) =
      #(p.positions.elems & LoadPosition)
}
```

Final comments on the Alloy spec

- The focus is mainly on defining a path as a sequence of positions
- Some parts of the initial description are not modeled at all
  - This is not necessarily a problem: we may want to build the formal model to focus on a specific aspect
  - … but it needs to be explained and justified
- There is a mismatch between the UML model and the Alloy spec
  - Same consideration as above
- Are all details in the Alloy model useful?
  - Do we need to spend time on the taxi code?
  - Number of seats?
  - What about latitude and longitude?

Politecnico
di Milano

More on Alloy + Requirements Engineering: the BLAA example

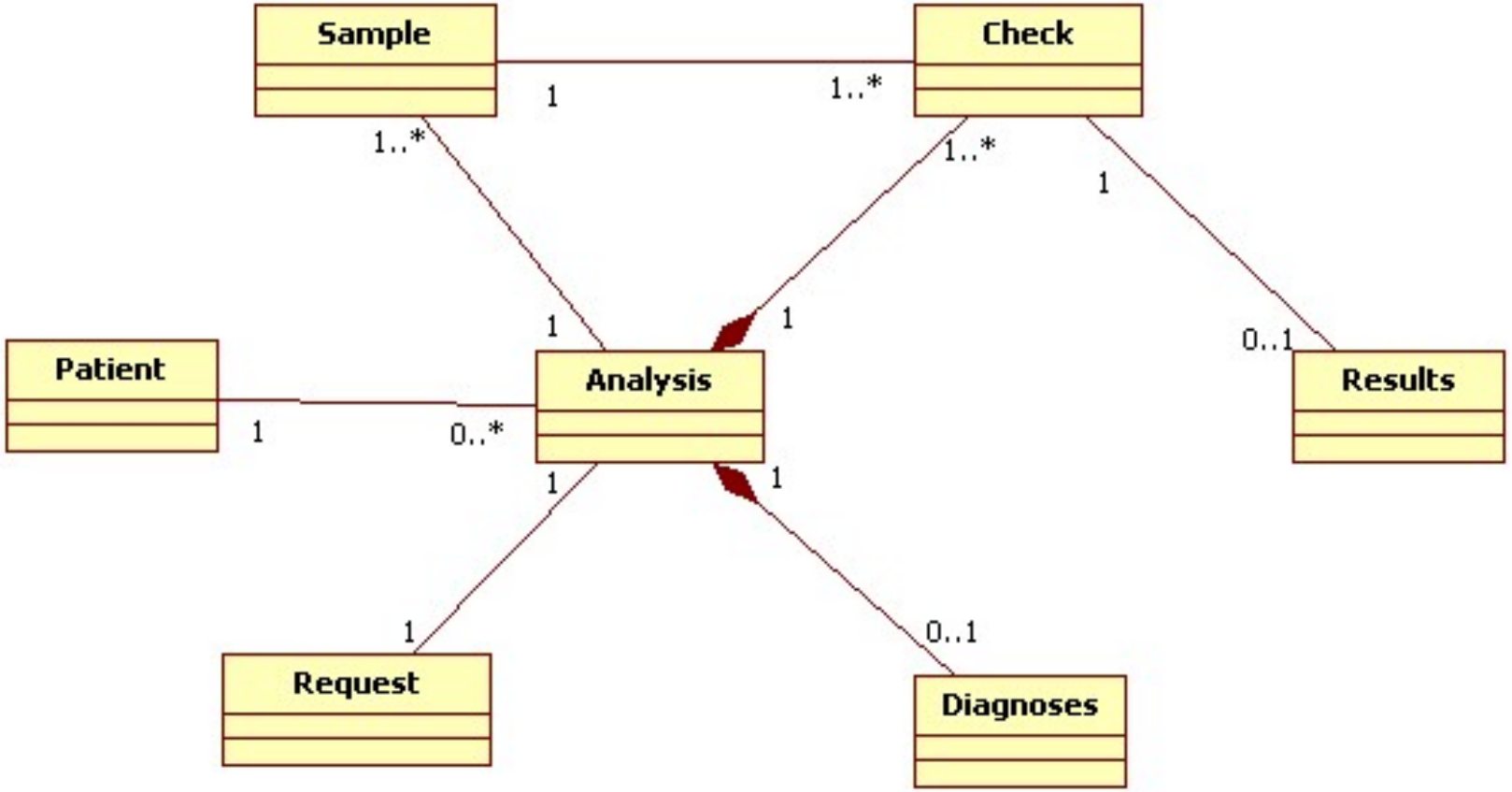
TO BE FIXED
UML DIAGRAM DOES NOT FULLY CORRESPOND TO THE PROBLEM DESCRIPTION
SOME ALLOY FACTS ARE NOT REALLY MEANINGFUL

BLAA

The system, called BLAA, automates the main functions of the lab. These are the main roles involved in the system and the functionalities to support:

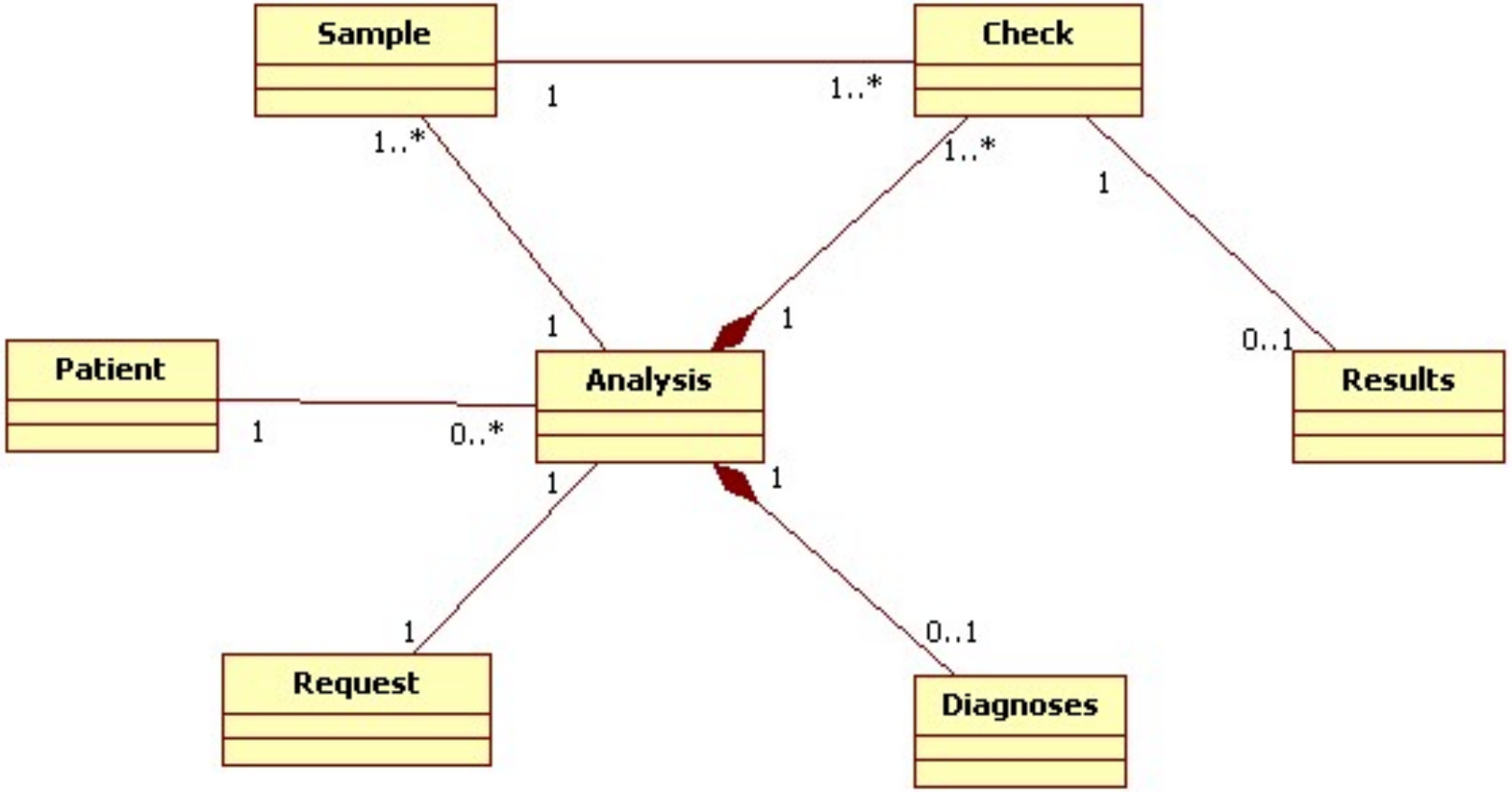
- ***Registration and checkout desk personnel****:* When a patient asks the acceptance desk clerk (ADC) to be accepted for a test, ADC enters in the system the personal data of the patient and the date when the results will be available. ADC is also in charge of delivering the report with the results to the patient.
- ***Nurse****:* Uses the system to obtain the list of different checks required by a patient. As a consequence, the nurse determines the number of blood samples to take and enters this data into the system. Then the system provides the codes to be written on the test-tubes containing the blood samples.
- ***Lab analyst****:* performs the analysis of the blood samples. Visualizes the data associated with each sample to determine the kind of analysis to perform, and enters in the system the outcomes of the analysis.
- ***Doctor****:* visualizes the results of the analysis for a given patient and provides a diagnosis.

BLAA – class diagram



BLAA – class diagram

First off… is this diagram complete?



BLAA

The system, called BLAA, automates the main functions of the lab. These are the main roles involved in the system and the functionalities to support:

- ***Registration and checkout desk personnel:*** When a patient asks the acceptance desk clerk (ADC) to be accepted for a test, ADC enters in the system the personal data of the patient and the date when the results will be available. ADC is also in charge of delivering the report with the results to the patient.
- ***Nurse:*** Uses the system to obtain the list of different checks required by a patient. As a consequence, **the nurse determines the number of blood samples to take and enters this data into the system**. Then the system **provides the codes** to be written on the test-tubes containing the blood samples.
- ***Lab analyst:*** performs the analysis of the blood samples. Visualizes the data associated with each sample to determine the kind of analysis to perform, and enters in the system the outcomes of the analysis.
- ***Doctor:*** visualizes the results of the analysis for a given patient and provides a **diagnosis**.

BLAA - signatures


```
sig Data {}
sig Result {}
sig Diagnosis {}

sig Patient {
   healtCard: one Int,
   patientCode: one Int
}

sig Request {
   issue_data: one Data,
   delivery_data: one Data
}
```

BLAA – more signatures


```
sig Sample {
    sampleNum: one Int,
    sampleCode: one Int
}{
    sampleNum > 0
}

sig Check{
    sample: one Sample,
    results:  lone Result
}

sig Analysis {
    request: one Request,
    checks: some Check,
    diagnosis: lone Diagnosis,
    patient: one Patient,
    samples: some Sample
}
```

BLAA - facts


```
fact NoDiagnosisWithoutAnalysis{
    (all d: Diagnosis {one a: Analysis | d = a.diagnosis})
}
fact {
    //No patient without analysis
    (all p: Patient { one a: Analysis | p = a.patient})

    //No Sample without Analysis
    (all s: Sample {one a: Analysis | s in a.samples})

    //No sample with the same SampleCode
    (no s1, s2: Sample | s1 != s2 and s1.sampleCode = s2.sampleCode)

    //No patients with the same PatientCode
    (no p1, p2: Patient | p1 != p2 and p1.patientCode = p2.patientCode)

    //No check without an Analysis
    (all c: Check {one a: Analysis | c in a.checks})

    //No results without a Check
    (all r: Result {one c: Check | r in c.results})
}
```

HOMEWORK: DEFINE MISSING FACTS AND CHECK THEM

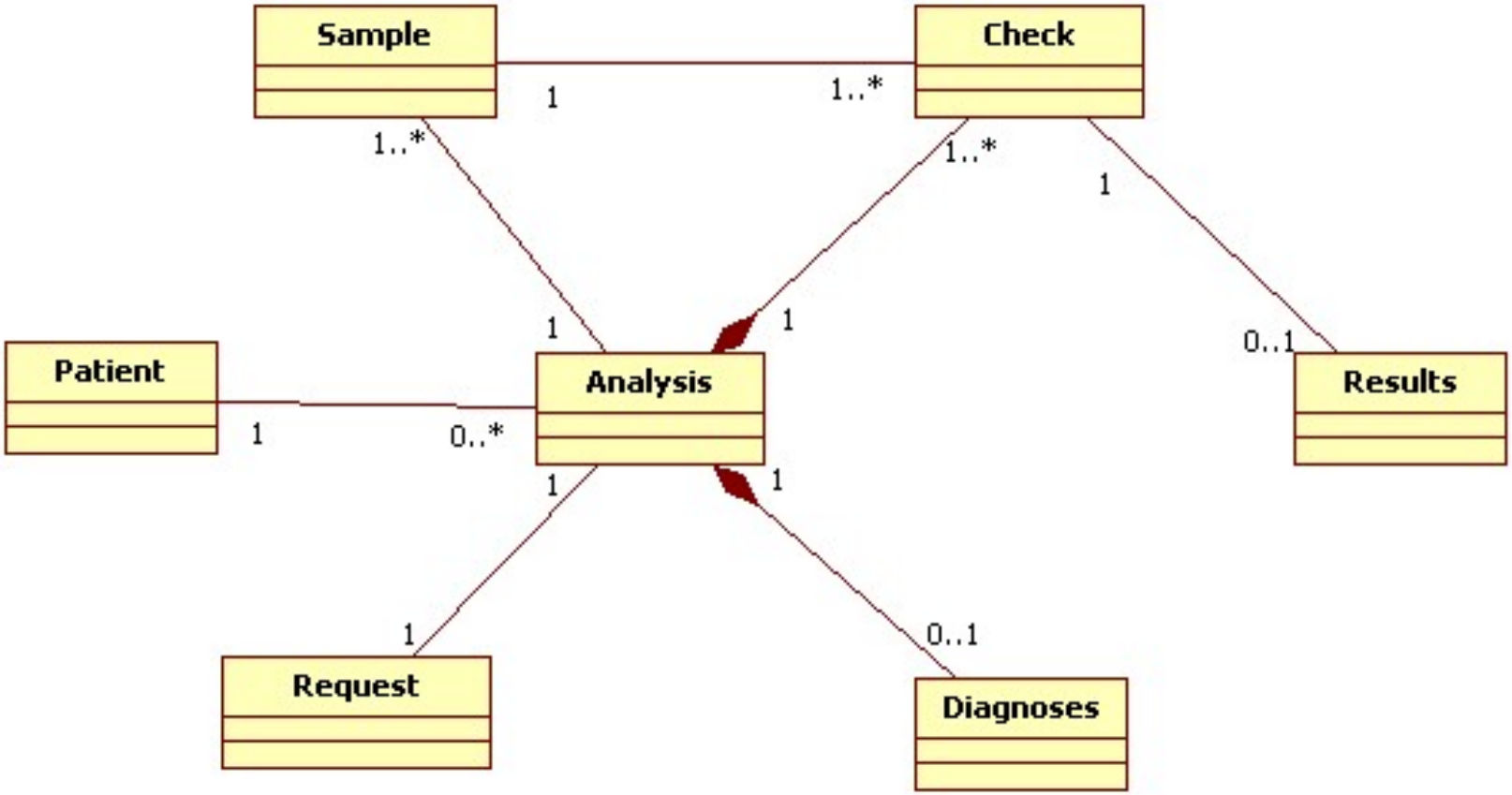
BLAA - facts

```
fact{
        //Two Analysis have different diagnosis
        (no a1, a2: Analysis, d: Diagnosis | a1 != a2 and a1.diagnosis = d and a2.diagnosis = d)

        //Two Diagnosis without Analysis
        (all d: Diagnosis {one a: Analysis | d = a.diagnosis})

        //No request without an Analysis
        (all r: Request {one a: Analysis | r = a.request})

        //No patient without analysis
        (all p: Patient { one a: Analysis | p = a.patient})

        //No Sample without Analysis
        (all s: Sample {one a: Analysis | s in a.samples})

        //No sample with the same SampleCode
        (no s1, s2: Sample | s1 != s2 and s1.sampleCode = s2.sampleCode)

        //No patients with the same PatientCode
        (no p1, p2: Patient | p1 != p2 and p1.patientCode = p2.patientCode)

        //No check without an Analysis
        (all c: Check {one a: Analysis | c in a.checks})

        //No results without a Check
        (all r: Result {one c: Check | r in c.results})
}
```

HOMEWORK: DEFINE MISSING FACTS AND CHECK THEM ***…How?***

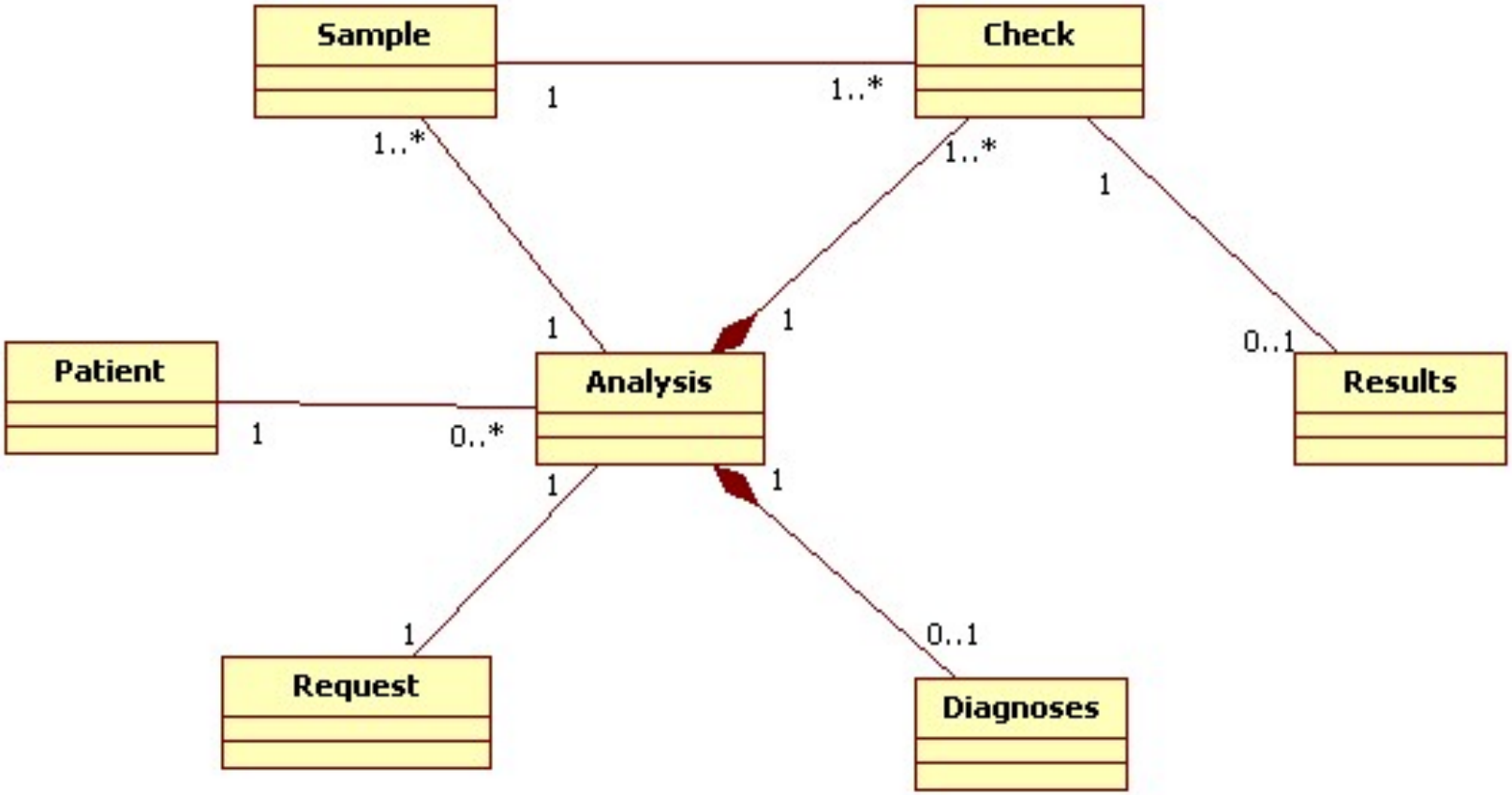
BLAA – class diagram

1. **Scan the model – find unconstrained elements;**



BLAA – class diagram

1. Scan the model – find unconstrained elements;
2. **Elaborate constraints of the element AND with respect to related elements;**



BLAA… **for example…**

The system, called BLAA, automates the main functions of the lab. These are the main roles involved in the system and the functionalities to support:

- ***Registration and checkout desk personnel****:* When a patient asks the acceptance desk clerk (ADC) to be accepted for a test, ADC enters in the system the personal data of the patient and the date when the results will be available. ADC is also in charge of delivering the report with the results to the patient.
- ***Nurse****:* Uses the system to obtain the list of different checks required by a patient. As a consequence, the nurse determines the number of blood samples to take and enters this data into the system. Then the system provides the codes to be written on the test-tubes containing the blood samples.
- ***Lab analyst****:* performs the analysis of the blood samples. Visualizes the data associated with each sample to determine the kind of analysis to perform, and enters in the system the outcomes of the analysis.
- ***Doctor****:* visualizes the results of the analysis for a given patient and provides a diagnosis.

Do we have all the details in the class diagram?

BLAA… **for example…**

The system, called BLAA, automates the main functions of the lab. These are the main roles involved in the system and the functionalities to support:

- ***Registration and checkout desk personnel****:* When a patient asks the acceptance desk clerk (ADC) to be accepted for a test, ADC enters in the system the personal data of the patient and the **date when the results** will be available. ADC is also in charge of delivering the report with the results to the patient.
- ***Nurse****:* Uses the system to obtain the list of different checks required by a patient. As a consequence, the nurse determines **the number of blood samples** to take and enters this data into the system. Then the system provides the **codes** to be written on **the test-tubes** containing the blood samples.
- ***Lab analyst****:* performs the analysis of the blood samples. Visualizes the data associated with each sample to determine the **kind of analysis** to perform, and enters in the system the outcomes of the analysis.
- ***Doctor****:* visualizes the results of the analysis for a given patient and provides a **diagnosis**.

Do we have all the details in the class diagram?

The goal of any Alloy Exercise

"precise specification of structural feats…"