Politecnico
di Milano

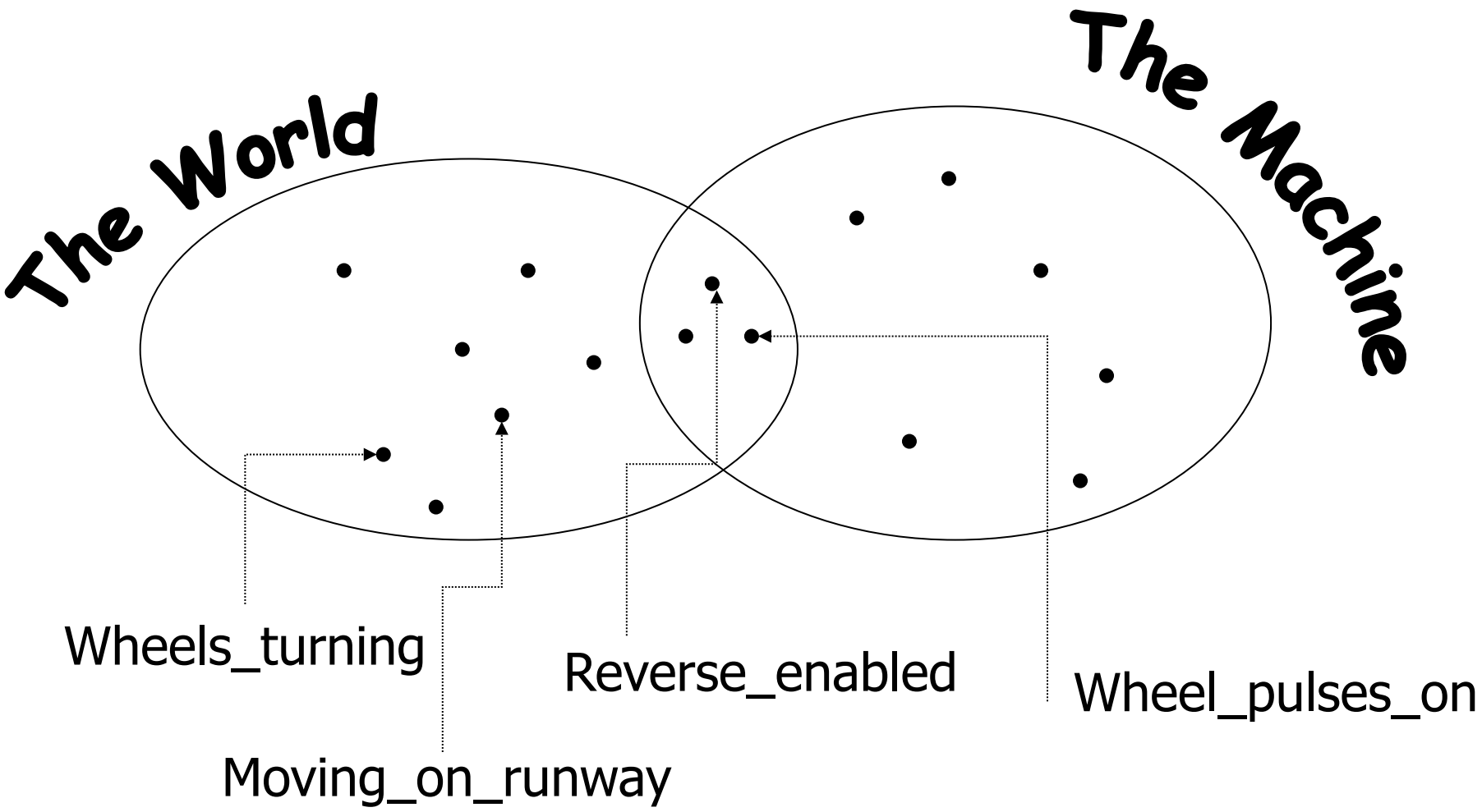
Alloy + Requirements Engineering

More fun with formal specs!

First… a bit of recap! Alloy and requirements!

- Are the previous examples describing requirements?
- What about the Line example?
  - We are modeling the structure of geometric figures and
  - Some constraints on this structure
  - Some operations
  - A constraint on one of the operations: the last point of the last segment of the line and the first point of the segment must coincide
  - Where are requirements?

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 {}
sig Flying extends AirCraftState {}
sig TakingOff extends AirCraftState {}
sig Landing extends  AirCraftState {}
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 Weels {
        deployed: Bool,
        turning: Bool
}{turning = True implies enabled = True}

sig Aircraft {
        status: one AirCraftState,
        weels: one Weels,
        weelsPulsesOn: one Bool,
        reverseThrustEnabled: one Bool
}{status = Flying implies Weels.deployed = False}
```

Modeling the Airbus braking logic with Alloy

fact domainAssumptions {

all a: Aircraft | a.weelsPulsesOn = True <=> a.weels.turning = True

all a: Aircraft | a.weels.turning = True <=> a.status = MovingOnRunaway}

fact requirement {

all a: Aircraft | a.reverseThrustEnabled = True <=> a.weelsPulsesOn = True}

assert goal {

all a: Aircraft | a.reverseThrustEnabled = True <=> a.status = MovingOnRunaway}

check goal

No counter examples are found!

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

Sample from Classwork

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: set Plug,
  chargingVehicles: Vehicle,

}
{#chargingVehicles <= #capacity}

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

}

Are we done here???

sig Vehicle {}

- sig ChargingStation { capacity: set Plug,
  chargingVehicles: Vehicle,

}

{#chargingVehicles <= #capacity}

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.

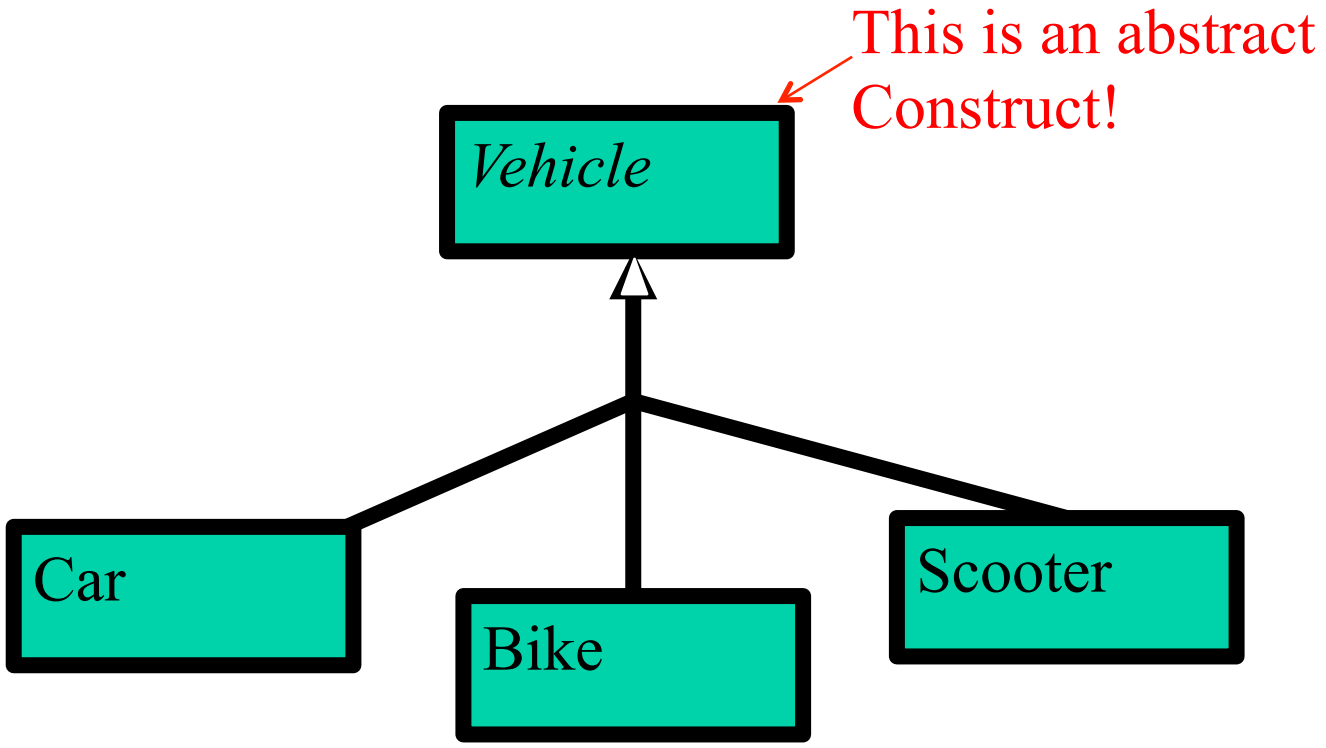
A. Define an Alloy model for the above domain. Specifically, define the facts modeling the following constraints:

- 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 is, 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 User {

ownedVehicles: set Vehicle, usedVehicle: lone Vehicle, disability: Disability

}

{

disability != none implies (usedVehicle & Scooter = none and usedVehicle & Bike = none)

}

- sig Disability{}

Signatures

- sig User {

ownedVehicles: set Vehicle, usedVehicle: lone Vehicle, disability: Disability

}

{

disability != none implies (usedVehicle & Scooter = none and usedVehicle & Bike = none)

}

- sig Disability{}
- sig Plug{}

Don’t forget the electrical outlet!

And now... the facts!

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

Sample from Classwork

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.

Sample from Classwork

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.

Facts

- fact noChargingVehicleInUse {
  no c: ChargingStation, u: User | c.chargingVehicles
  & u.usedVehicle != none

}

Sample from Classwork

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.

Facts

- fact noChargingVehicleInUse {
  no c: ChargingStation, u: User | c.chargingVehicles
  & u.usedVehicle != none
}
- fact noTwoUsersExploitingAVehicle {
  no disjoint u1, u2: User | u1.usedVehicle =
  u2.usedVehicle
}

Sample from Classwork

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.

Facts

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

Sample from Classwork

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.

Facts

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

Are we done?

- Moment of suspense...

Facts

- No!
  - We need to model what *can* happen, in our case...
  - And the connected conditions

Sample from Classwork

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)

}

- pred acquireVehicle[u: User, v: Vehicle, u': User]
  { compatibleVehicle[u, v] and
  freeVehicle[v] and
  u.usedVehicle = none
  u'.ownedVehicles = u.ownedVehicles and
  u'.disability = u.disability and u'.usedVehicle = v

}

Hint: remember there is usually a part A and a part B;

So… More Predicates

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

Hint: check your time!

And a few Facts!

- and, we need to model what shall never happen by design!

- fact noTwoOwners {

```
no disjoint u1, u2: User | u1.ownedVehicles &
u2.ownedVehicles != none and
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!!!

Thanks for your endurance…

- Break?

Coming up next… Examples from previous projects!

General scientific rule, $E=MC^2$ ➔ Exam = $(MoreCases)^2$

- Scan through and study project spec from last year!
  - Identify the UML models and supporting “views” for the specification;
  - Study the Alloy specification provided;
  - Consider their result to evaluate the completeness and correctness of your own;

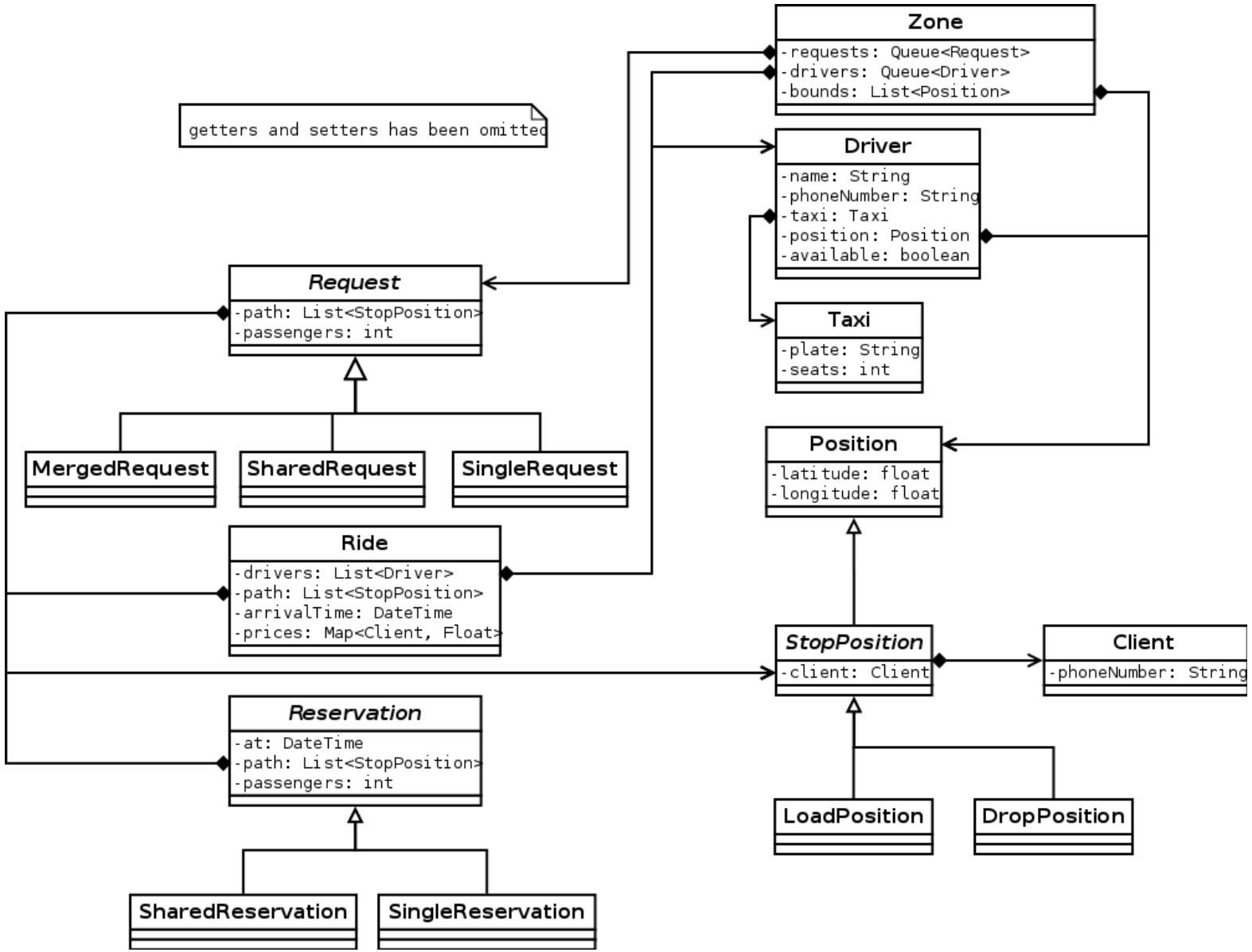
For Example...

- We will project and implement myTaxiService, which is a service based on mobile application and 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 GPS position to identify client's zone (but the client can insert it manually) and find 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 quick system so if there is a registration a lot of clients won't 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).
- The system includes extra services and functionalities such as taxi sharing.
- The main purpose of the system is to be more efficient and reliable than the existing one in order to decrease costs of the taxi management and offer a better service to the clients.

For Example...

- We will project and implement myTaxiService, which is a service based on mobile application and 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 GPS position to identify client's **zone** (but the client can insert it manually) and find 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 quick system so if there is a registration a lot of clients won't 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).
- The system includes extra services and functionalities such as taxi **sharing**.
- The main purpose of the system is to be more efficient and reliable than the existing one in order to decrease **costs** of the taxi management and offer a better service to the clients.

For Example… cont'd



Signatures

- open util/boolean // this is a support clause calling for additional libraries
- sig Taxi{

code: Int,

seats: Int}

{code > 0 seats > 0 seats <= 4}

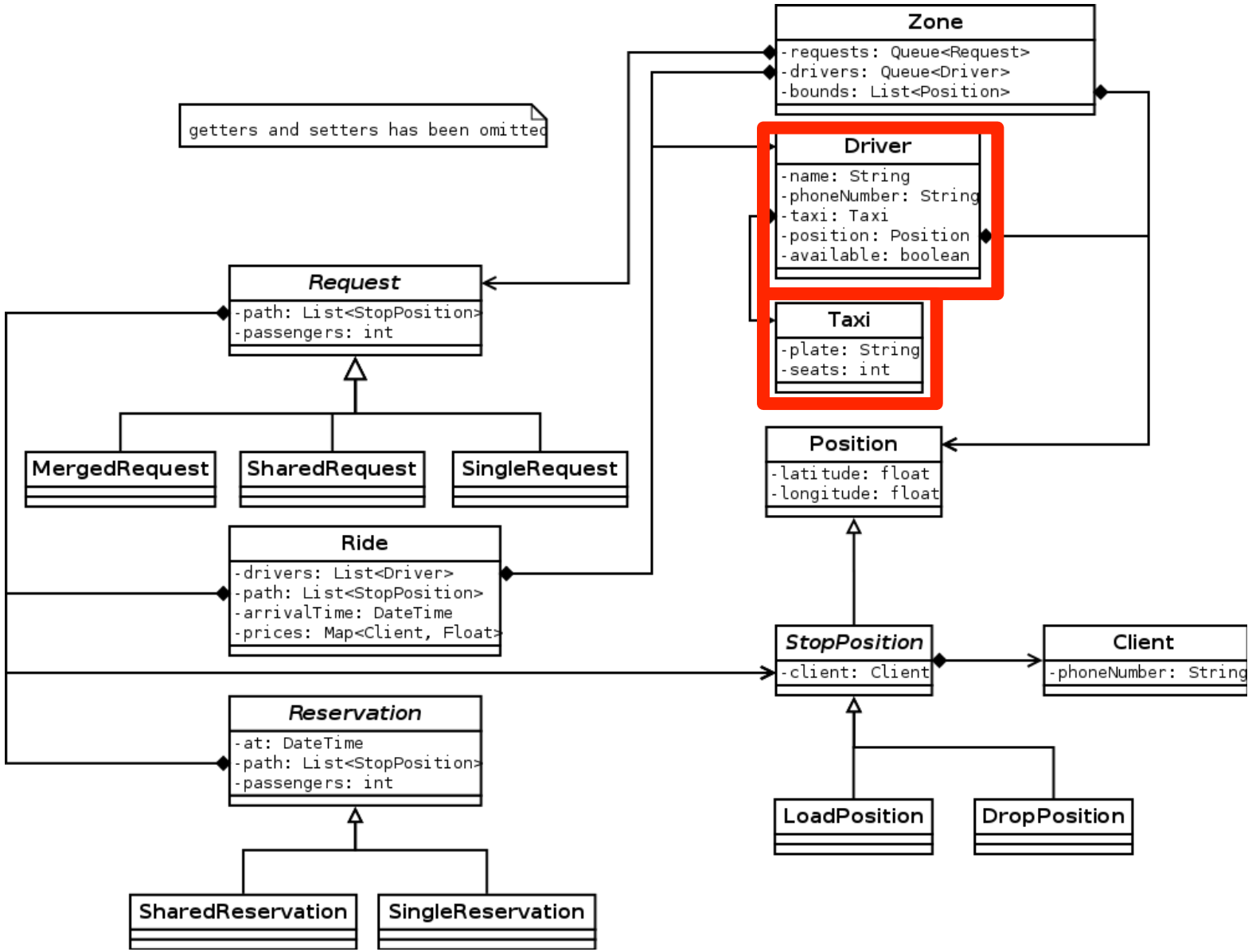
sig Driver {

taxi: one Taxi,

available: one Bool}

sig PhoneNumber { }

For Example… cont'd



Signatures

- sig Client { phoneNumber: PhoneNumber }
- sig Position {

latitude: Int, // should be float

longitude: Int // should be float

}

abstract sig StopPosition extends Position {

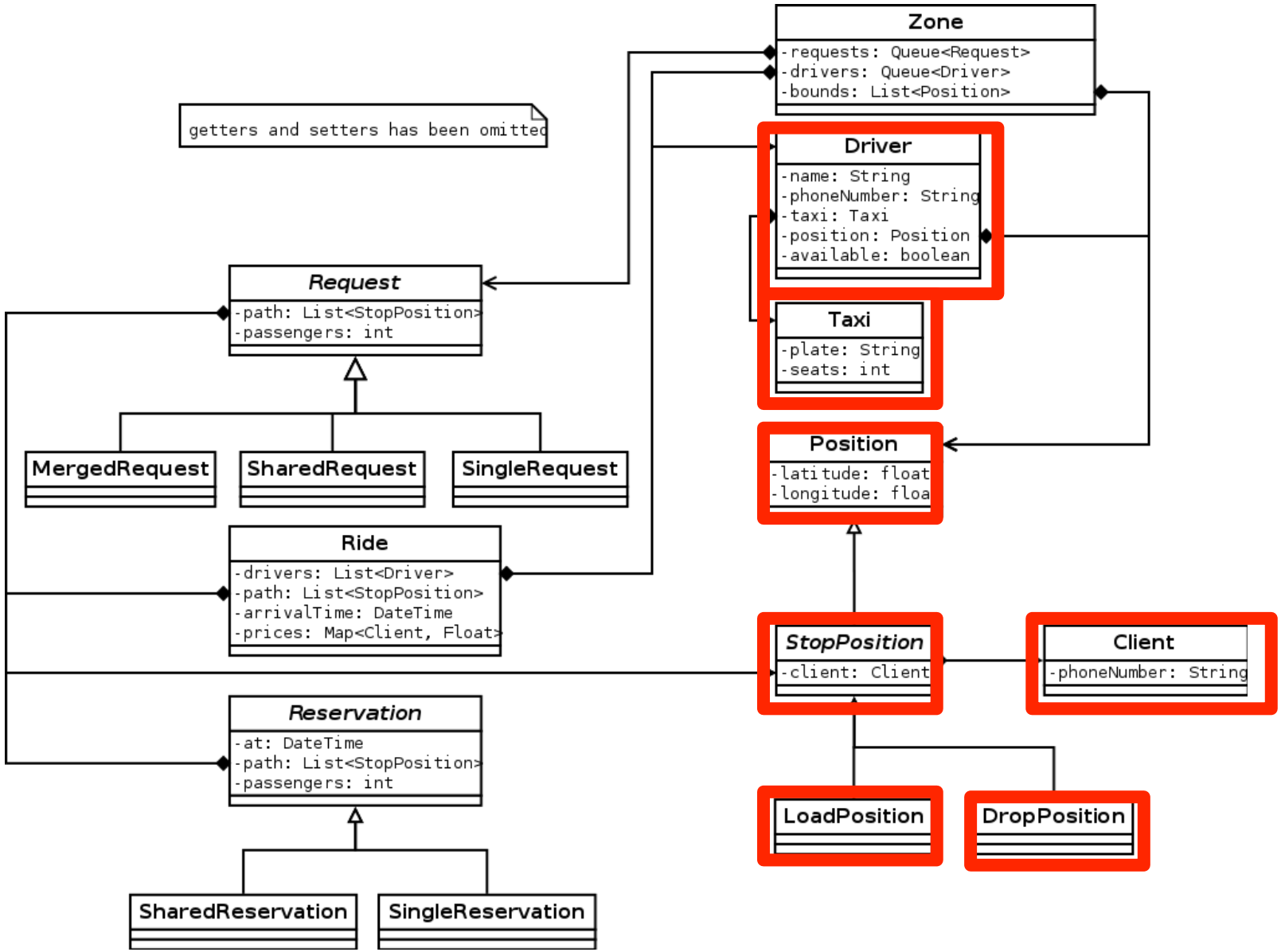
client: one Client

}

sig LoadPosition extends StopPosition { }

sig DropPosition extends StopPosition { }

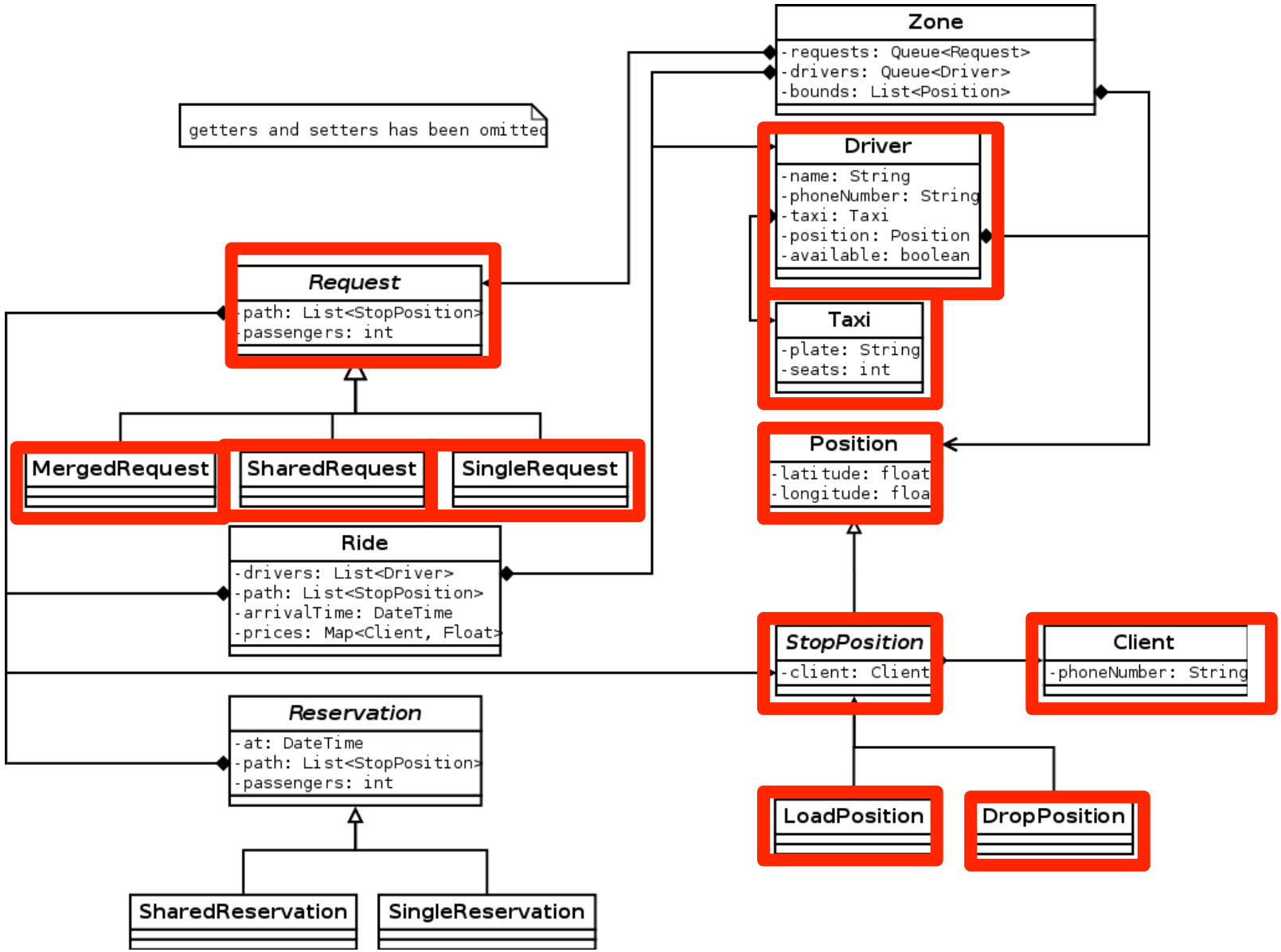
For Example… cont'd



Signatures


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

For Example… cont'd

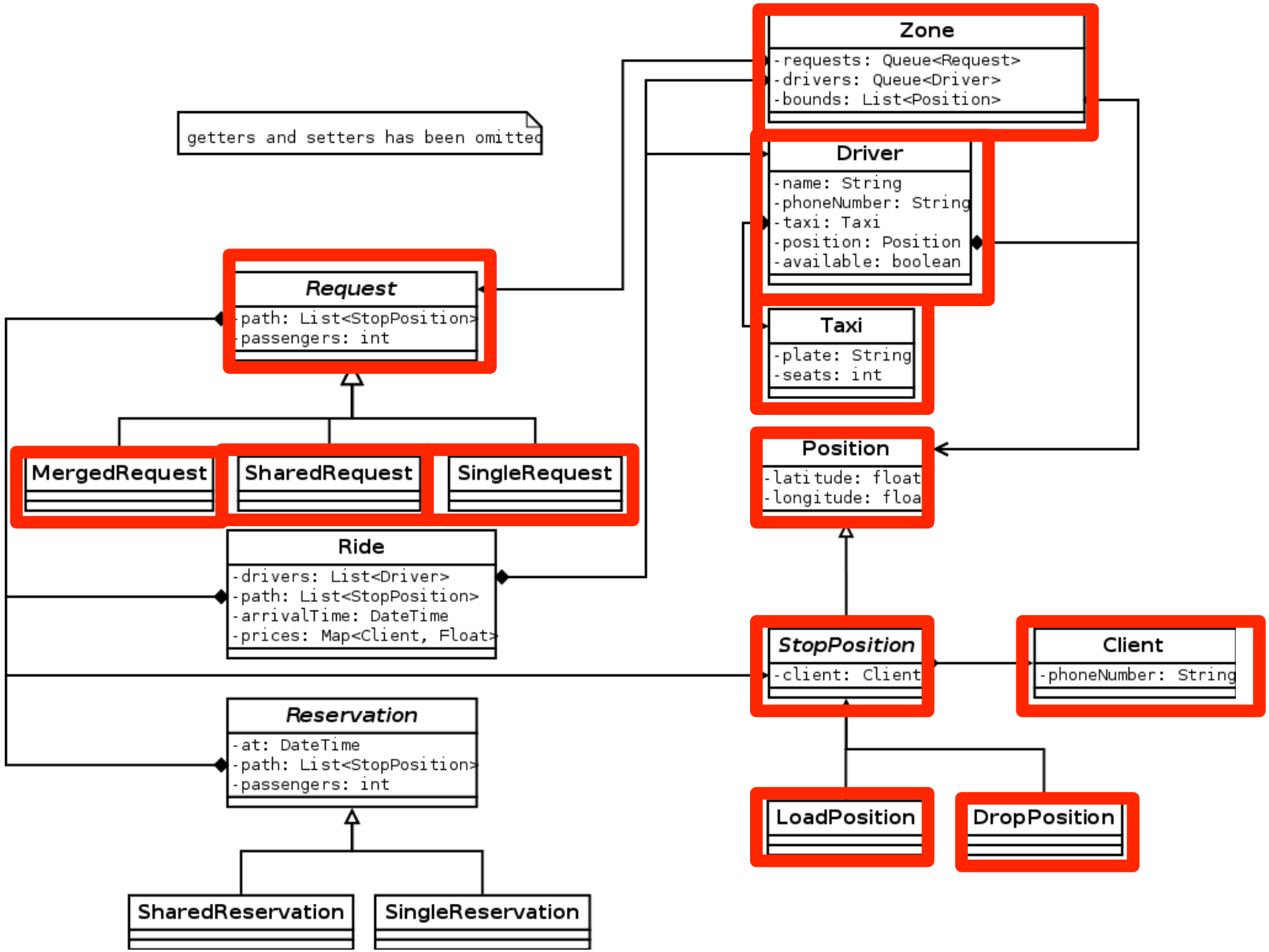


Signatures


```
abstract sig Queue { s: seq univ }
sig DriverQueue extends Queue { }{
s.elems in Driver
}
sig RequestQueue extends Queue { }{
s.elems in Request
}

sig Zone {
drivers: one DriverQueue,
requests: one RequestQueue,
bounds: seq Position
}{
#bounds > 2
all p: bounds.elems | not p in StopPosition not bounds.hasDups
}
```

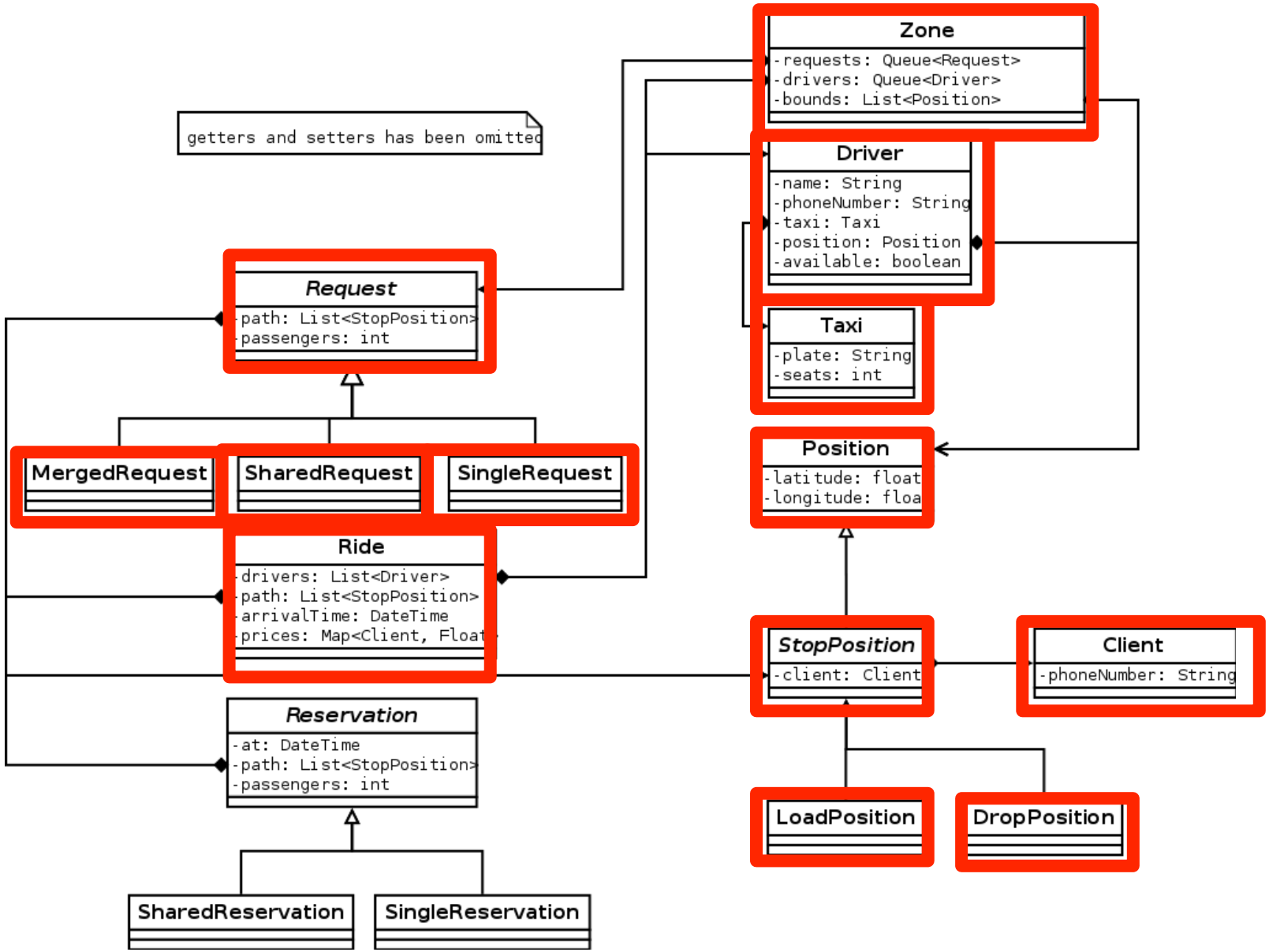
For Example… cont'd



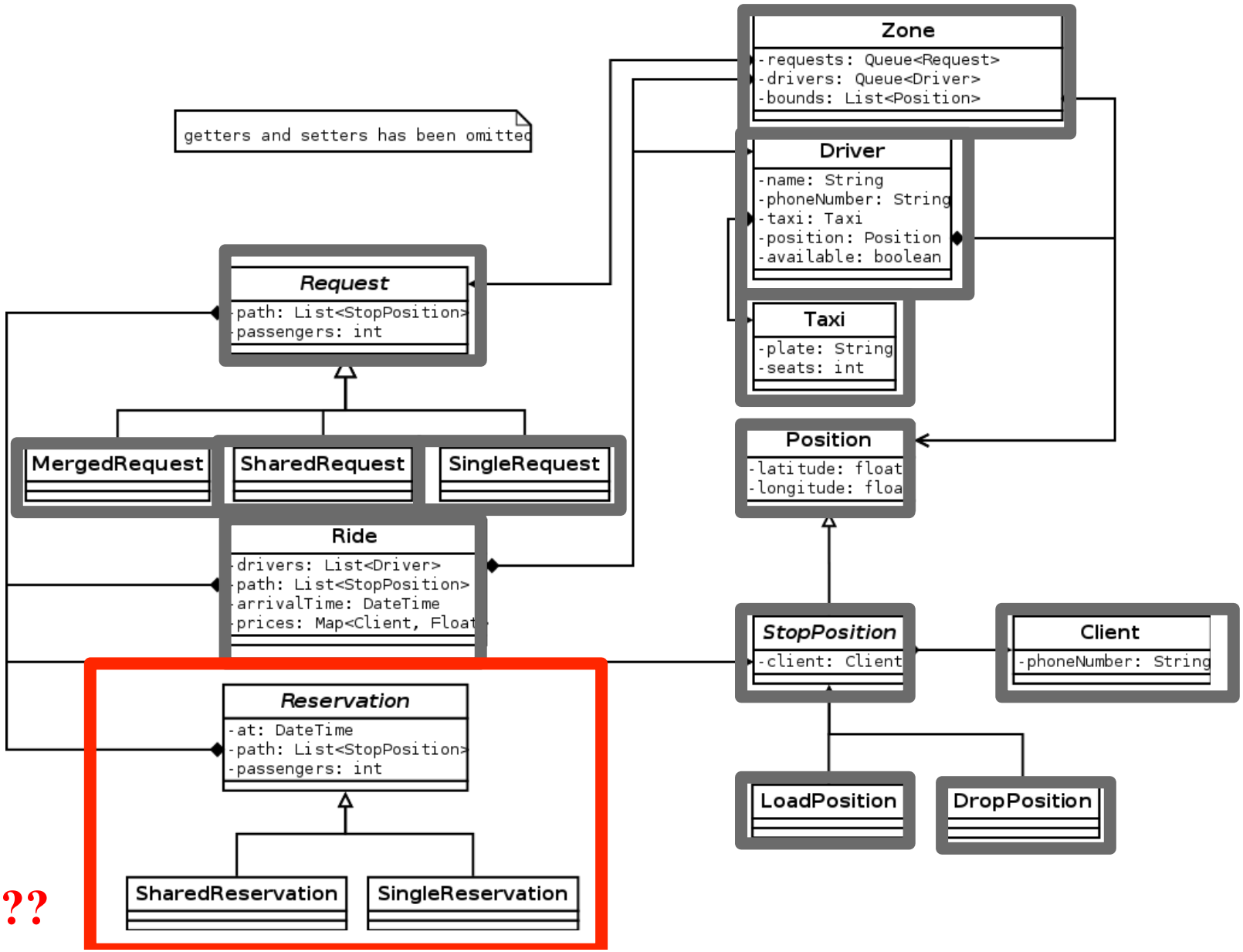
Signatures

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

For Example… cont'd



For Example… cont'd



How about these???

Signatures

```
abstract sig Queue { s: seq univ }
sig DriverQueue extends Queue { }{
s.elems in Driver
}
sig RequestQueue extends Queue { }{
s.elems in Request
}

sig Zone {
drivers: one DriverQueue,
requests: one RequestQueue,
bounds: seq Position
}{
#bounds > 2
all p: bounds.elems | not p in StopPosition not bounds.hasDups
}
```

Could be this??? I (the teacher) cannot really be sure…

This is (almost) a mistake!!!

Facts

- **Consistency clauses:**
  - fact codeAreUnique { all t1,t2: Taxi | (t1 != t2) => t1.code != t2.code }
  - fact allTaxiAreOwned { Driver.taxi = Taxi }
  - fact taxiHaveOnlyOneDriver { all d1, d2:Driver | d1 != d2 => d1.taxi != d2.taxi }
  - …
- **Model properties verification:**
  - assert pathPositionsAreEven { all p:Path| rem[#p.positions, 2] = 0 }
  - 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] }
  - …

And finally…

- run show for 6
- check pathPositionsAreEven
- check clientGetInAndGetOut
- check pathSameNumberOfLoadAndDrop
- check getInverseOfAdd
- run add for 5
- run get for 5

Choose boundaries and Execution clauses wisely!

7 commands were executed. The results are:
#1: **Instance found.** show is consistent.
#2: No counterexample found. pathPositionsAreEven may be valid.
#3: No counterexample found. clientGetInAndGetOut may be valid.
#4: No counterexample found. pathSameNumberOfLoadAndDrop may be valid.
#5: No counterexample found. getInverseOfAdd may be valid.
#6: **Instance found.** add is consistent.
#7: **Instance found.** get is consistent.

- Any Questions?