Politecnico
di Milano

Alloy

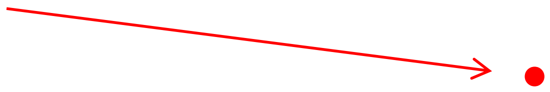
Exercises (and a few Hints ☺)

Line

- We wish to specify a simple fragment of geometry
- Points are defined by a signature
  - `sig Point {}`
- Starting from Point, you are to specify:
  1. A Segment; i.e., a pair of points
  2. A Line; i.e., a sequence of connected segments
  3. A Polygon; i.e., a closed Line
  4. Assuming that a predicate `intersect` is defined for two segments, write a predicate `linesIntersect` that checks intersection of two lines
  5. Specify an operation `add` that takes a line and a segment and produces a new line where the segment is appended as a last segment; of course the last point of the last segment of the line and the first point of the segment must coincide

Line - signatures

sig Point{}



Remember: Fields **are** Relations!

- Alloy fields define relations among the signatures
  - Similar to a field in an object class that establishes a relation between objects of two classes
  - Similar to associations in OCL

Line - signatures

```
sig Point{}

sig Segment {
  start: one Point,
  end: one Point
}{???}
```

Is this enough? What is the condition that must always be true?

Line - signatures

```
sig Point{}

sig Segment {
  start: one Point,
  end: one Point
}{start!=end}
```

- Alternative formulation

```
sig Segment {
  start: one Point,
  end: one Point
}
fact {
 all s: Segment | s.start!=s.end}
```

Line

- We wish to specify a simple fragment of geometry
- Points are defined by a signature
  - `sig Point {}`
- Starting from Point, you are to specify:
  1. A Segment; i.e., a pair of points
  2. **A Line; i.e., a sequence of connected segments**
  3. A Polygon; i.e., a closed Line
  4. Assuming that a predicate `intersect` is defined for two segments, write a predicate `linesIntersect` that checks intersection of two lines
  5. Specify an operation `add` that takes a line and a segment and produces a new line where the segment is appended as a last segment; of course the last point of the last segment of the line and the first point of the segment must coincide

Line - signatures

```
sig Line {
  startSeg: one Segment,
  endSeg: one Segment,
  intermediateSeg: set Segment
}
```

My definition…

```
{
```

What do I have to do here?

```
??????????

}
```

Line - signatures

```
sig Line {
  startSeg: one Segment,
  endSeg: one Segment,
  intermediateSeg: set Segment
}
{
  no x: intermediateSeg | startSeg=x or endSeg=x




}
```

There is no intermediate that is also a start or end

Line - signatures

```
sig Line {
  startSeg: one Segment,
  endSeg: one Segment,
  intermediateSeg: set Segment
}
{
  no x: intermediateSeg | startSeg=x or endSeg=x
  lone x: intermediateSeg | startSeg.end=x.start
}
```

There is zero or one intermediate whose start tails the end of startSeg

Line - signatures

```
sig Line {
  startSeg: one Segment,
  endSeg: one Segment,
  intermediateSeg: set Segment
}
{
  no x: intermediateSeg | startSeg=x or endSeg=x
  lone x: intermediateSeg | startSeg.end=x.start
  lone x: intermediateSeg | endSeg.start=x.end




}
```

There is zero or one intermediate whose end leads to the start of endSeg

Line - signatures

```
sig Line {
  startSeg: one Segment,
  endSeg: one Segment,
  intermediateSeg: set Segment
}
{
  no x: intermediateSeg | startSeg=x or endSeg=x
  lone x: intermediateSeg | startSeg.end=x.start
  lone x: intermediateSeg | endSeg.start=x.end
  no x: intermediateSeg | startSeg.start=x.end




}
```

There is no intermediate whose end is the starting point of startSeg

Line - signatures

Full signature… can anyone tell me what's wrong with this?

```
sig Line {
  startSeg: one Segment,
  endSeg: one Segment,
  intermediateSeg: set Segment
}
{
  no x: intermediateSeg | startSeg=x or endSeg=x
  lone x: intermediateSeg | startSeg.end=x.start
  lone x: intermediateSeg | endSeg.start=x.end
  no x: intermediateSeg | startSeg.start=x.end
  no x: intermediateSeg | endSeg.end=x.start
  all x: intermediateSeg | x.end = endSeg.start or one x1: intermediateSeg |
     x.end = x1.start
  all x: intermediateSeg | x.start = startSeg.end or one x1: intermediateSeg |
     x.start = x1.end

}
```

Line - signatures

```
sig Line {
  startSeg: one Segment,
  endSeg: one Segment,
  intermediateSeg: set Segment
}
{
  no x: intermediateSeg | startSeg=x or endSeg=x
  lone x: intermediateSeg | startSeg.end=x.start
  lone x: intermediateSeg | endSeg.start=x.end
  no x: intermediateSeg | startSeg.start=x.end
  no x: intermediateSeg | endSeg.end=x.start
  all x: intermediateSeg | x.end = endSeg.start or one x1: intermediateSeg |
     x.end = x1.start
  all x: intermediateSeg | x.start = startSeg.end or one x1: intermediateSeg |
     x.start = x1.end
 #intermediateSeg = 0 implies (startSeg = endSeg or
     startSeg.end=endSeg.start)
}
```

Line - signatures

We need to exhaust all possible cases for the specs to be **complete… Correctness check** comes from **checking facts!**

```
sig Line {
  startSeg: one Segment,
  endSeg: one Segment,
  intermediateSeg: set Segment
}
{
  no x: intermediateSeg | startSeg=x or endSeg=x
  lone x: intermediateSeg | startSeg.end=x.start
  lone x: intermediateSeg | endSeg.start=x.end
  no x: intermediateSeg | startSeg.start=x.end
  no x: intermediateSeg | endSeg.end=x.start
  all x: intermediateSeg | x.end = endSeg.start or one x1: intermediateSeg |
     x.end = x1.start
  all x: intermediateSeg | x.start = startSeg.end or one x1: intermediateSeg |
     x.start = x1.end
  #intermediateSeg = 0 implies (startSeg = endSeg or
     startSeg.end=endSeg.start)
}
```

Line

- We wish to specify a simple fragment of geometry
- Points are defined by a signature
  - `sig Point {}`
- Starting from Point, you are to specify:
  1. A Segment; i.e., a pair of points
  2. A Line; i.e., a sequence of connected segments
  3. **A Polygon; i.e., a closed Line**
  4. Assuming that a predicate `intersect` is defined for two segments, write a predicate `linesIntersect` that checks intersection of two lines
  5. Specify an operation `add` that takes a line and a segment and produces a new line where the segment is appended as a last segment; of course the last point of the last segment of the line and the first point of the segment must coincide

Line – signatures and predicates

```
sig Polygon extends Line {
}
{
 ??????????
}
```

Anyone?

Line

- We wish to specify a simple fragment of geometry
- Points are defined by a signature
  - `sig Point {}`
- Starting from Point, you are to specify:
  1. A Segment; i.e., a pair of points
  2. A Line; i.e., a sequence of connected segments
  3. **A Polygon; i.e., a closed Line**
  4. Assuming that a predicate `intersect` is defined for two segments, write a predicate `linesIntersect` that checks intersection of two lines
  5. Specify an operation `add` that takes a line and a segment and produces a new line where the segment is appended as a last segment; of course the last point of the last segment of the line and the first point of the segment must coincide

Line – signatures and predicates

```
sig Polygon extends Line {
}
{
  startSeg.start=endSeg.end
}
```

Line

- We wish to specify a simple fragment of geometry
- Points are defined by a signature
  - `sig Point {}`
- Starting from Point, you are to specify:
  1. A Segment; i.e., a pair of points
  2. A Line; i.e., a sequence of connected segments
  3. A Polygon; i.e., a closed Line
  4. Assuming that a **predicate `intersect`** is defined for two segments, **write a predicate `linesIntersect` that checks intersection of two lines**
  5. Specify an operation `add` that takes a line and a segment and produces a new line where the segment is appended as a last segment; of course the last point of the last segment of the line and the first point of the segment must coincide

Line – signatures and predicates

```
sig Polygon extends Line {
}
{
  startSeg.start=endSeg.end
}

// simplified version of intersect
pred intersect(s1, s2: Segment) {
s1!=s2 and (s1.start=s2.start or s1.start = s2.end or s1.end=s2.end or
     s1.end=s2.start)
}

pred linesIntersect(l1, l2: Line) {
l1!=l2 and (some s1: l1.intermediateSeg, s2:l2.intermediateSeg | intersect[s1,
     s2])
}
```

Can anyone spell these out for me?

Line – more on predicates and facts

```
pred addToLine(l: Line, s: Segment, l': Line) {
// precondition
  l.endSeg.end = s.start

//postcondition
  l'.startSeg=l.startSeg and
  l'.intermediateSeg = l.intermediateSeg + l.endSeg and
  l'.endSeg = s
}

fact noTwoSegWithSameStartEnd {
  no disj x1, x2: Segment | x1.start = x2.start and x1.end=x2.end
}
```

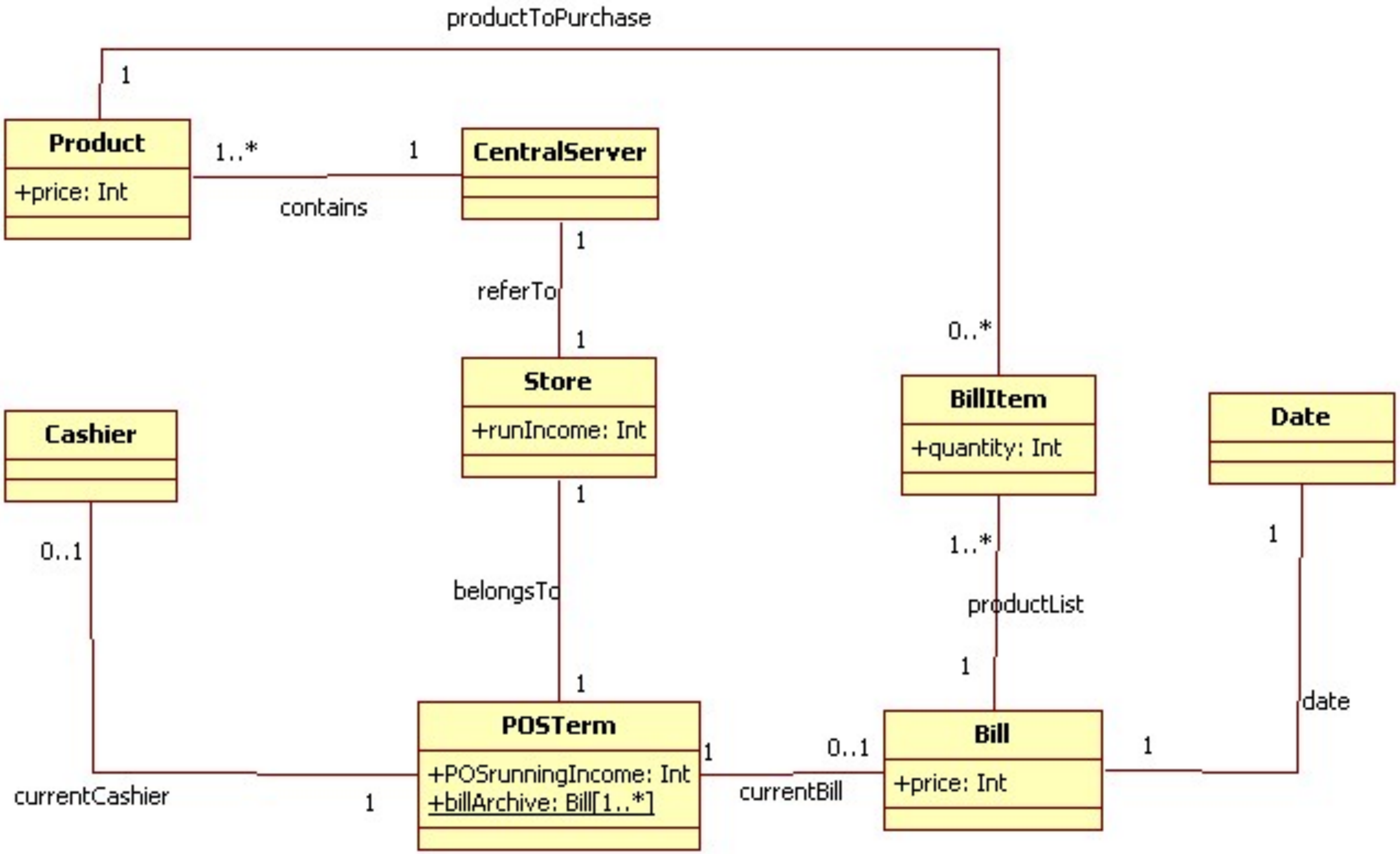
What does this mean?

What does this mean?

POS

- A supermarket wants to build a new POS (Point Of Service) system to support the work of cashier
- 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

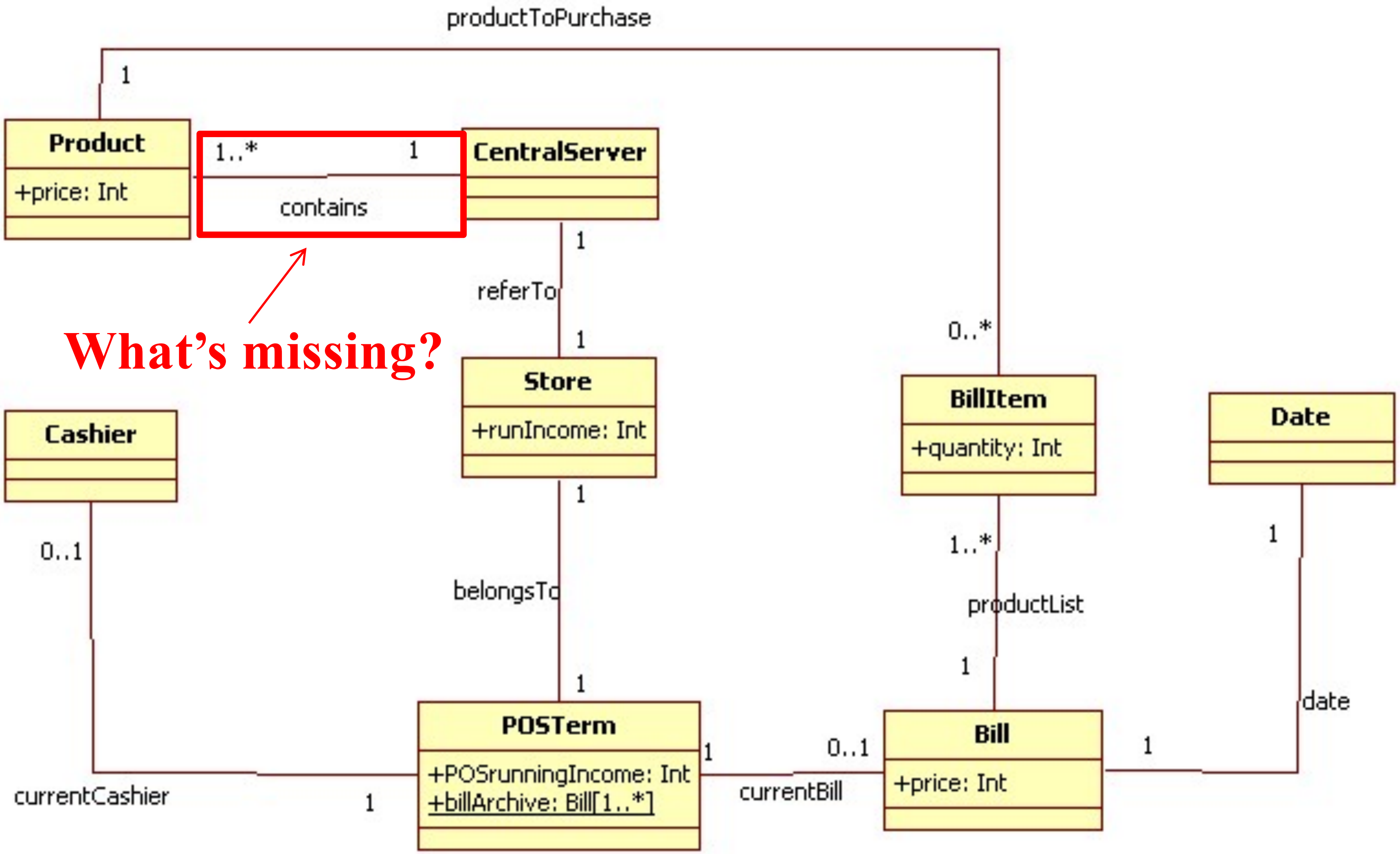


POS - signatures

```
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
}
```

Based on this specification, can you tell me what's wrong with the UML Class diagram?

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
}
```

What is the navigability of the “contains” relation?

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
}
```

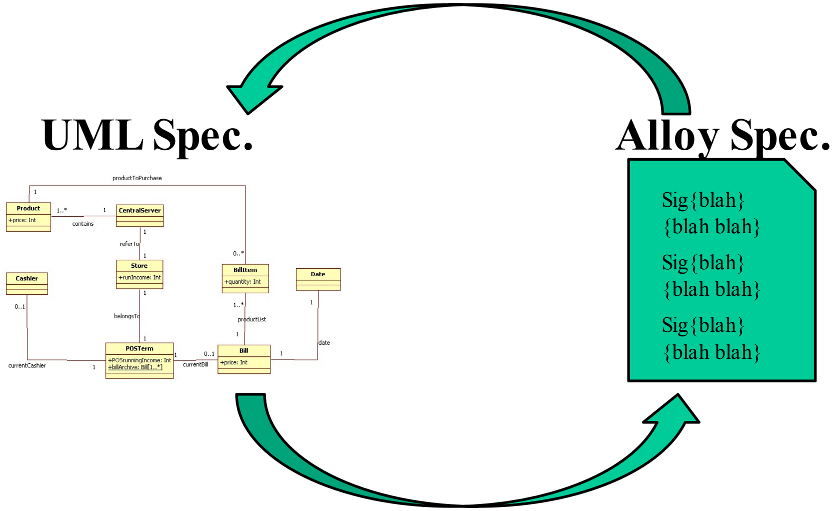
Based on this spec, definitely one or more arrows are missing…

*Alloy allows for **precise** specification*
*Of structural and behavioral **feats**…*
*And these are facts you can **check**! ☺*

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…”



UML Spec.

Alloy Spec.

Sig{blah}
{blah blah}
Sig{blah}
{blah blah}
Sig{blah}
{blah blah}

POS - facts

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

fact billCanBeAssociatedtoOnlyOnePOSTerm {
  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)
}
```

“Correctness check** comes from **checking facts!”

POS - operations

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

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

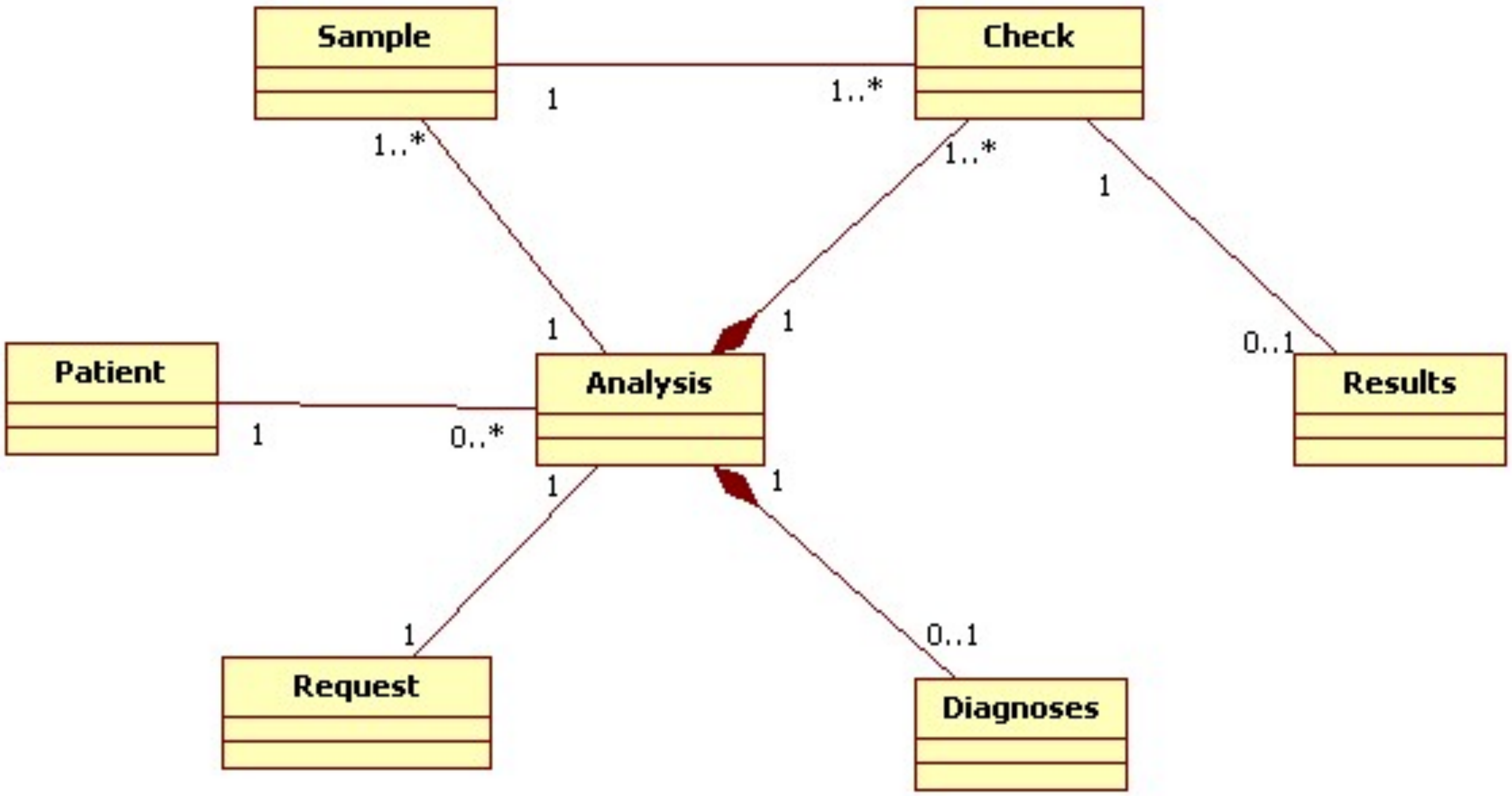
```
pred show(){
 #POSTerm > 1
}

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

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:

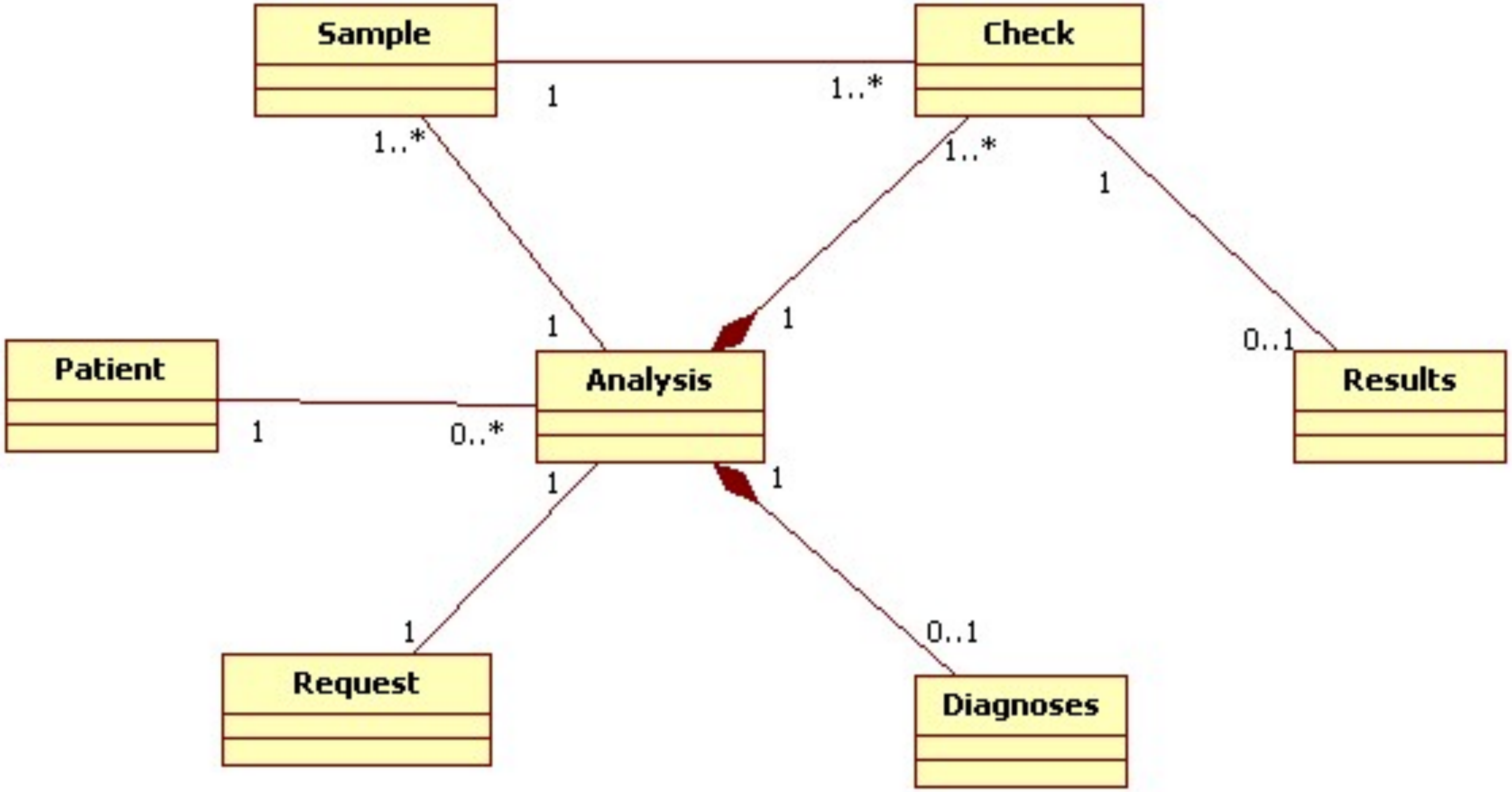
- ***Registration and checkout desk personnel**:* When a patient asks the acceptance desk clerk (ADC) to be accepted for a test, ADC enters in the system the personal data of the patient and the date when the results will be available. ADC is also in charge of delivering the report with the results to 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**.

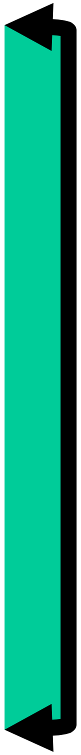
BLAA - signatures

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

First step should always be to figure out which signatures are *atomic*, i.e., they have no further specification beyond the signature.

BLAA - signatures

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

sig Patient {
      healtCard: one Int,
      patientCode: one Int
}{
healtCard > 0
patientCode > 0
}

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



Other signatures usually depend on them.

BLAA – more signatures

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

Note that I need to:

1. **Predicate on the semantics behind the model;**

1. **Specify the expected “response” of every signature with respect to related elements;**

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

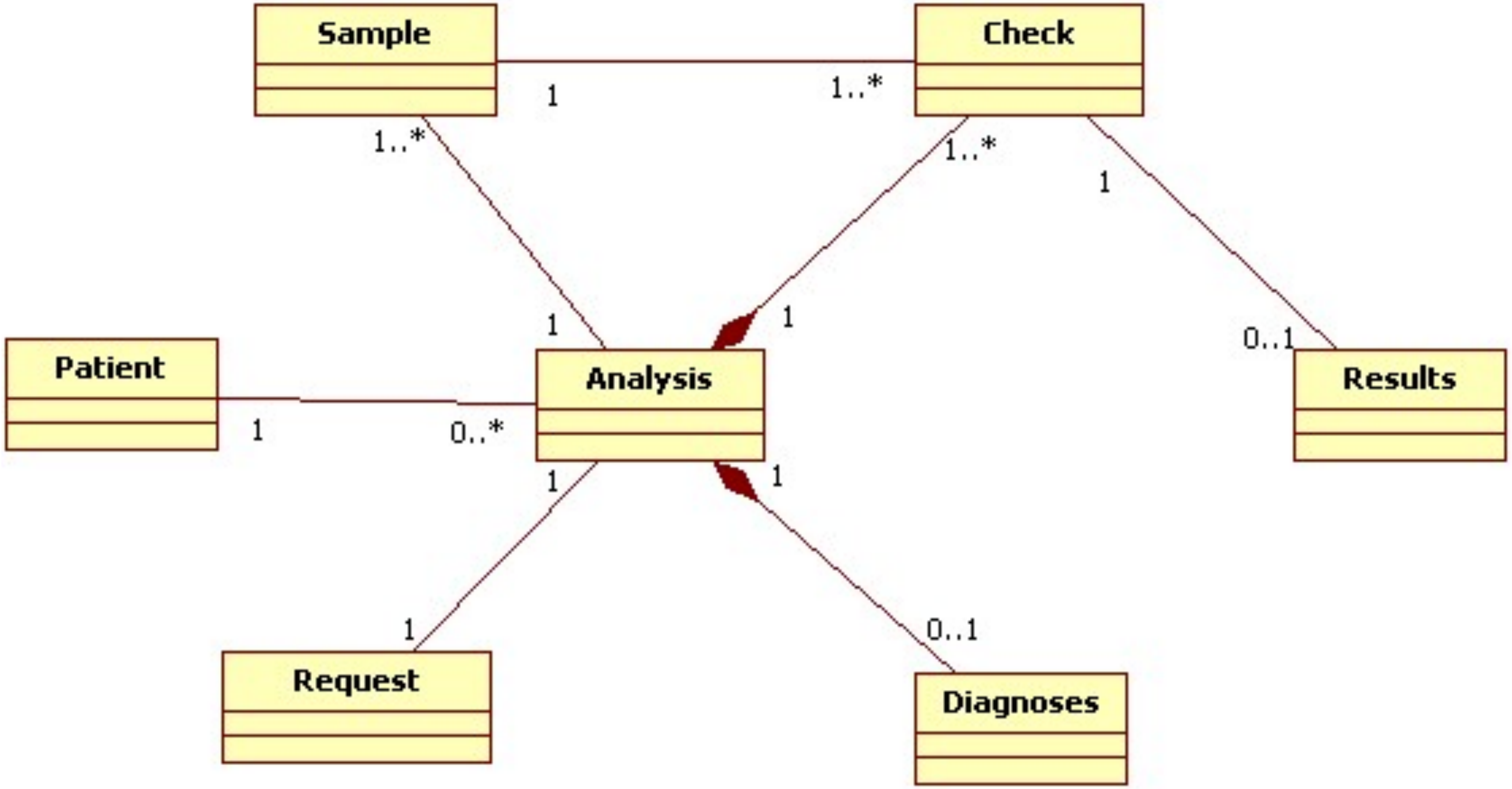
BLAA - facts

```
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?***

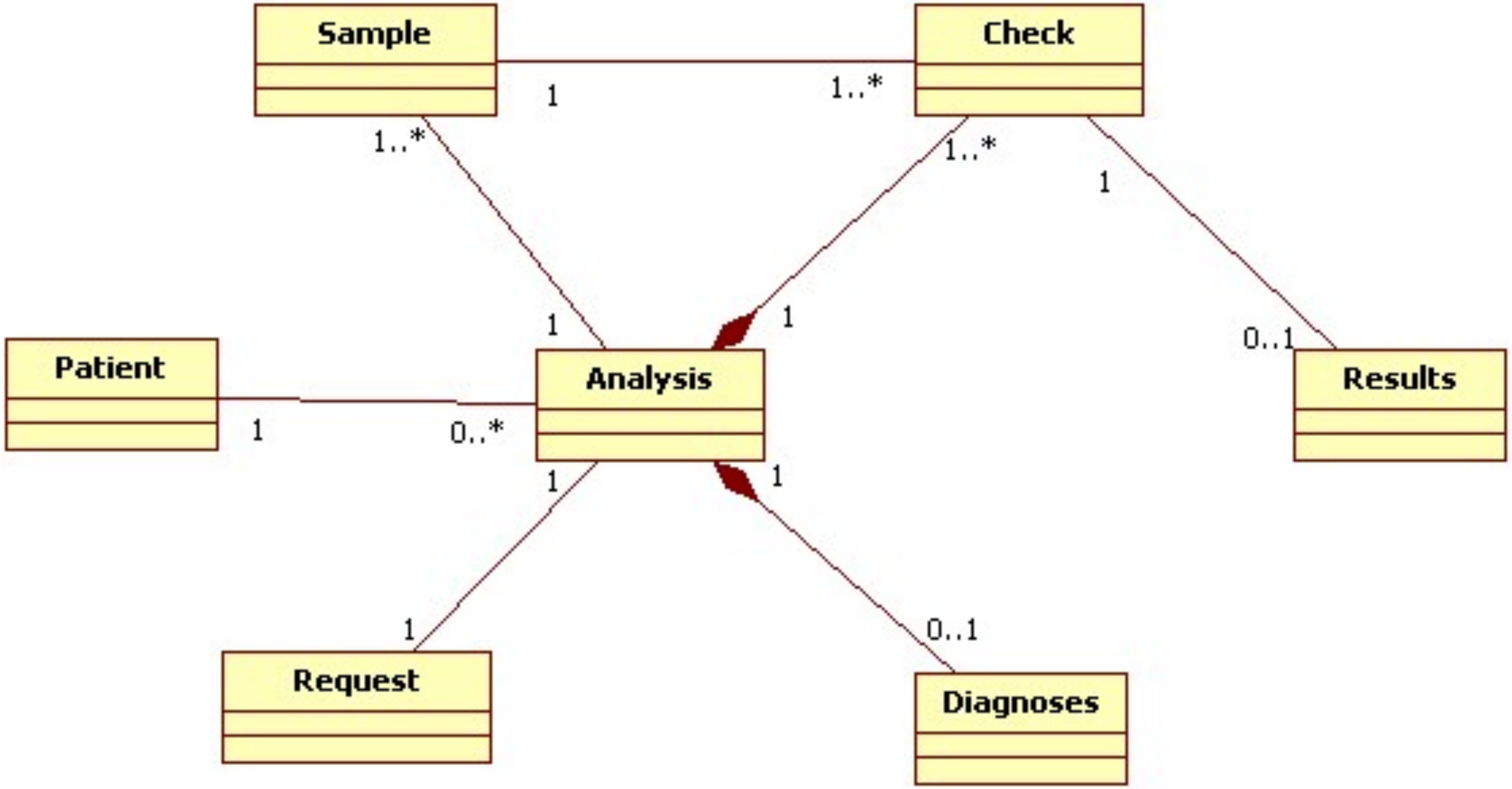
BLAA – class diagram

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 analys***_t:_ 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…**

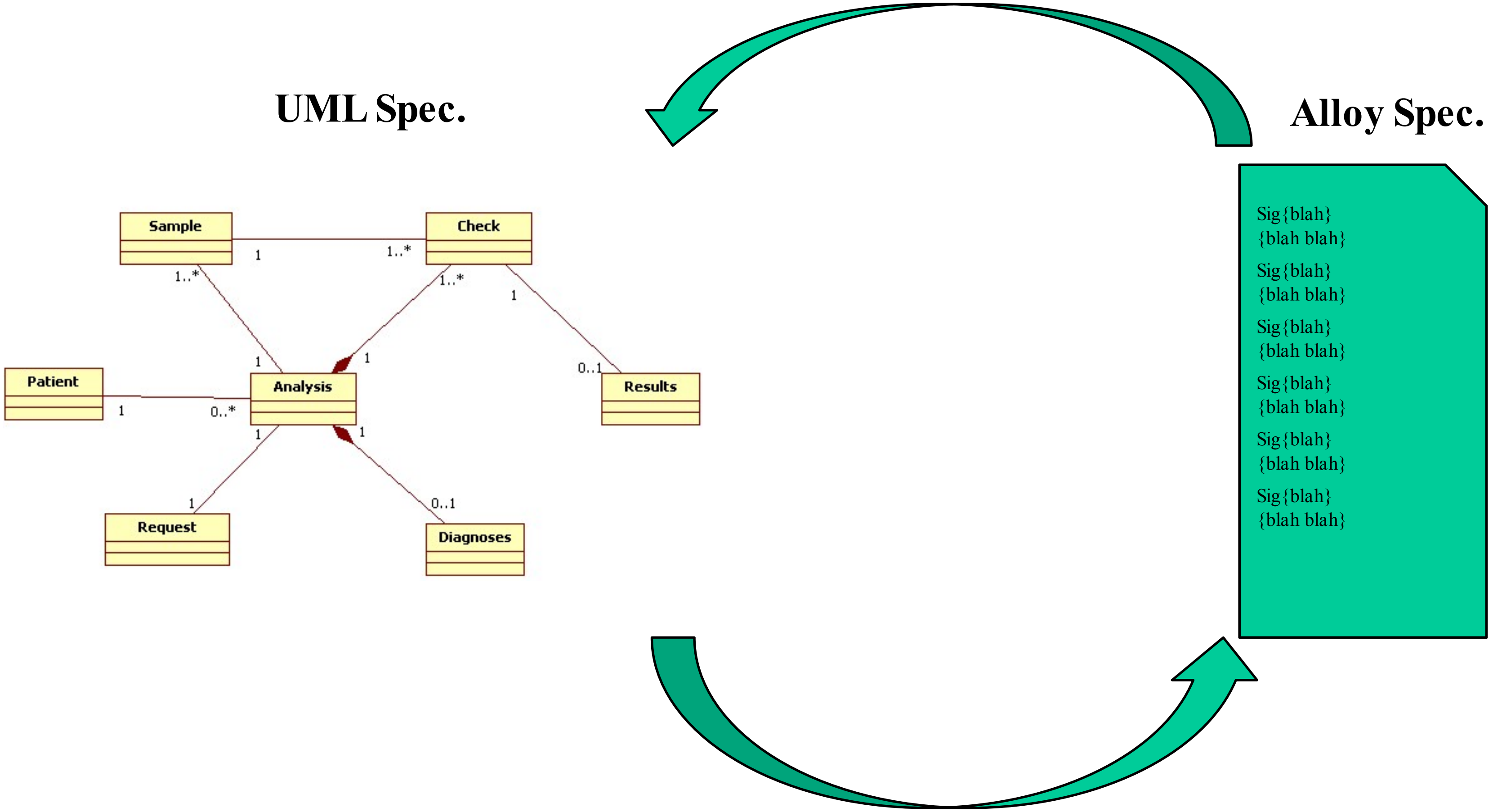
The system, called BLAA, automates the main functions of the lab. These are the main roles involved in the system and the functionalities to support:

- ***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…”



UML Spec.

Alloy Spec.

Sig{blah}
{blah blah}

Sig{blah}
{blah blah}

Sig{blah}
{blah blah}

Sig{blah}
{blah blah}

Sig{blah}
{blah blah}

Sig{blah}
{blah blah}

In summary: Alloy Characteristics

- Finite scope check:
  - The analysis is sound, but incomplete
- Infinite model:
  - Finite checking does not get reflected in your model
- Declarative: first-order relational logic
- Automatic analysis:
  - visualization a big help
- Structured data

In summary: Bad Things

- Sequences are awkward
- Recursive functions hard to express

Hint: do not try to define recursive functions at the exam - It usually turns out into a bloodbath.

Hint: Operator Precedence

Keep in mind operator precedence when writing your specs!!

lowest

```
||
<=>
=>
&&
!
=   !=   in
+   -
++
&
->
<:
:>
[]
.
~   *
```

highest