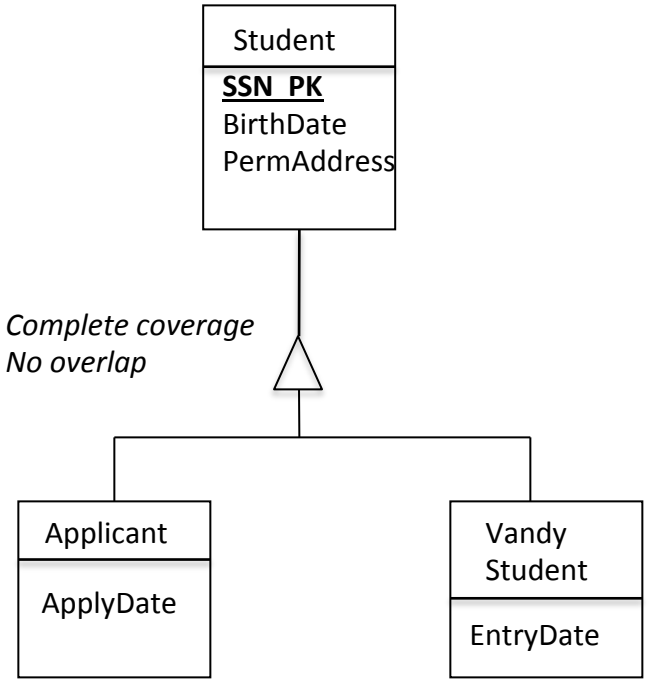
Some Basic Mistakes

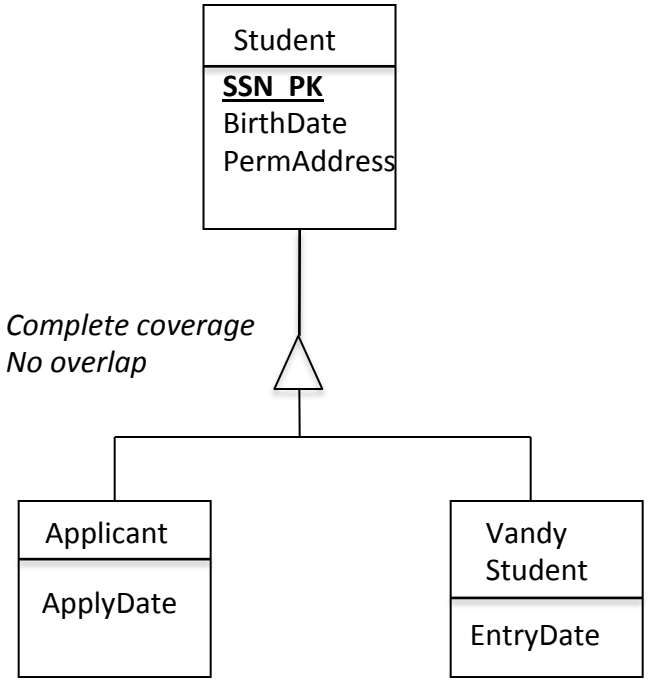
Translate the following into table and constraint definitions

See (partial) answer next page



Translate the following into table and constraint definitions

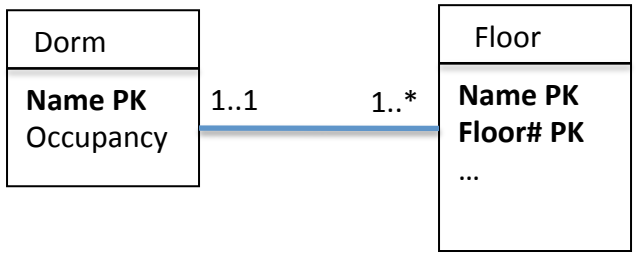
```
CREATE TABLE Student (
  SSN INTEGER PRIMARY KEY,
  BirthDate Date,
  PermAddress VARCHAR(60))

CREATE TABLE Applicant (
  SSN INTEGER PRIMARY KEY,
  ApplyDate Date,
  FOREIGN KEY (SSN) REFERENCES Student
    ON DELETE CASCADE ON UPDATE CASCADE)

CREATE TABLE VandyStudent (
  SSN INTEGER PRIMARY KEY,
  EntryDate Date,
  FOREIGN KEY (SSN) REFERENCES Student
    ON DELETE RESTRICT ON UPDATE RESTRICT)
```



What's missing? Assertions (or alternatively in-table checks and/or triggers) – see the sample dorm energy DB, past exam key and quiz keys for examples of such assertions and intable CHECKS and triggers

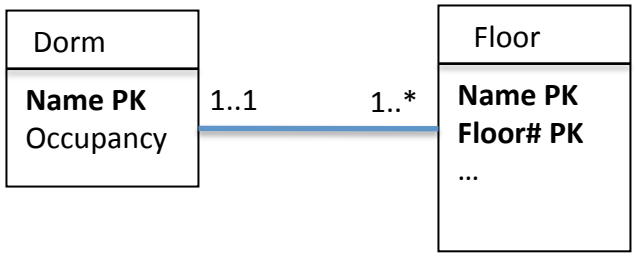
BTW – you don't need special assertions for "partial coverage" allowed or "overlap allowed" conditions

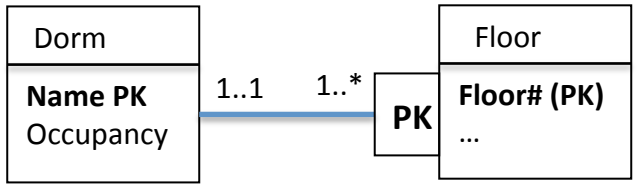
Consider the following buggy UML snippet, in which the **Name** attribute in the Floor table refers to the **Name** attribute in the Dorm Table. Floor# is an integer no greater than 10. Replace it by a correct UML snippet that represents the same database definition.



See answer next page

Consider the following buggy UML snippet, in which the **Name** attribute in the Floor table refers to the **Name** attribute in the Dorm Table. Floor# is an integer no greater than 10. Replace it by a correct UML snippet that represents the same database definition.

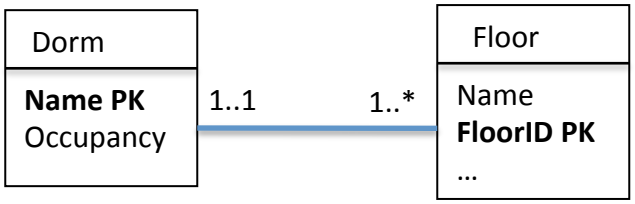




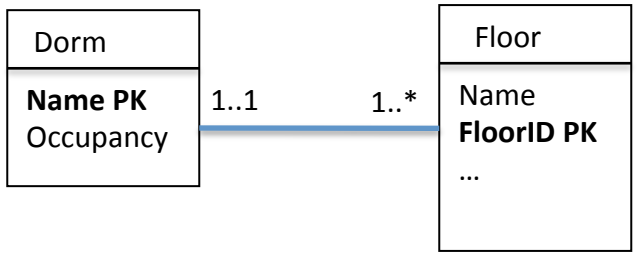
remember that I treat the composition operator as synonymous with 1..1 and the aggregation operator as synonymous with 0..1

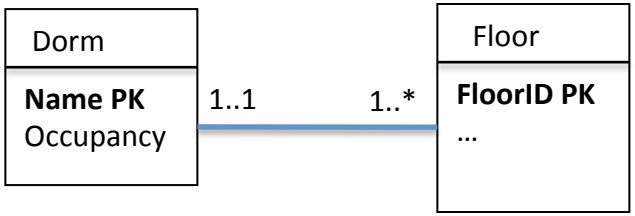
```
CREATE TABLE Floor (
  Floor# INTEGER,
  DormName VARCHAR(60),
  PRIMARY KEY (Floor#, DormName),  /* PK attributes always NOT NULL, whether explitely declared or not */
  FOREIGN KEY (DormName) REFERENCES Dorm(Name)  ....)
```

Consider the following buggy UML snippet, in which the **Name** attribute in the Floor table refers to the **Name** attribute in the Dorm Table. FloorID is an auto-increment PK



Consider the following buggy UML snippet, in which the **Name** attribute in the Floor table refers to the **Name** attribute in the Dorm Table. FloorID is an auto-increment PK

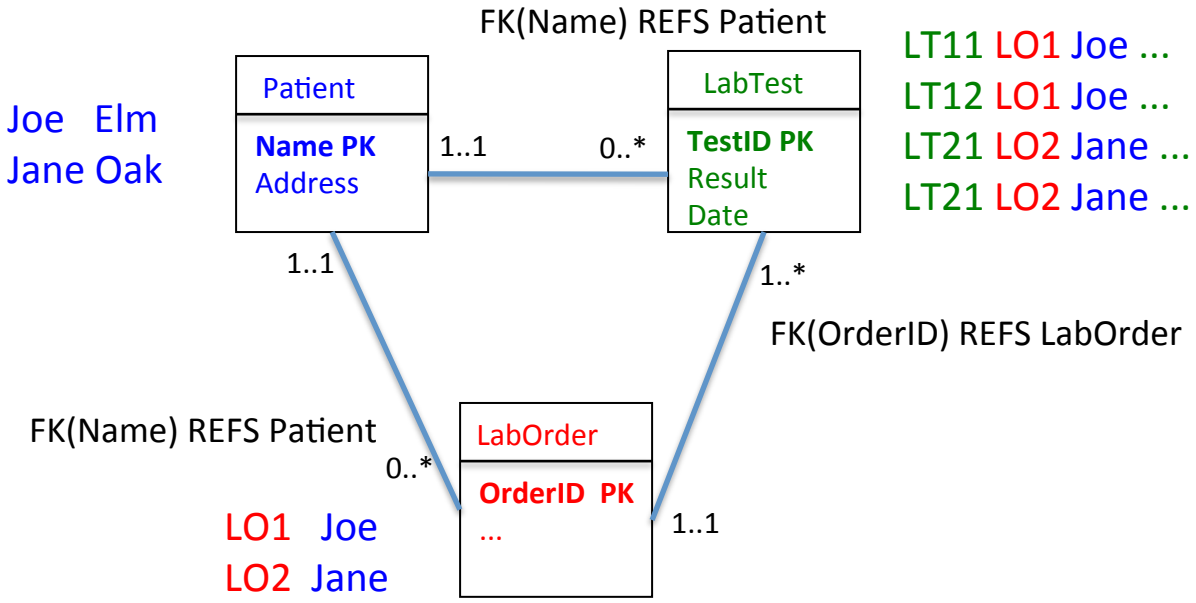





```
CREATE TABLE Floor (
  FloorID INTEGER,
  DormName VARCHAR(60) NOT NULL, /* if NOT NULL forgotten then consistent with 0..1 rather than 1..1
  PRIMARY KEY (FloorID),
  FOREIGN KEY (DormName) REFERENCES Dorm(Name)  ....)
```

Subtle Source of Redundancy and Possibly Inconsistency

Suppose we have three classes. One is a Patient class. When a patient goes to the doctor, a battery of tests can be ordered at one time as represented by the LabOrder class. The individual tests within these packaged orders are represented by the LabTest class.

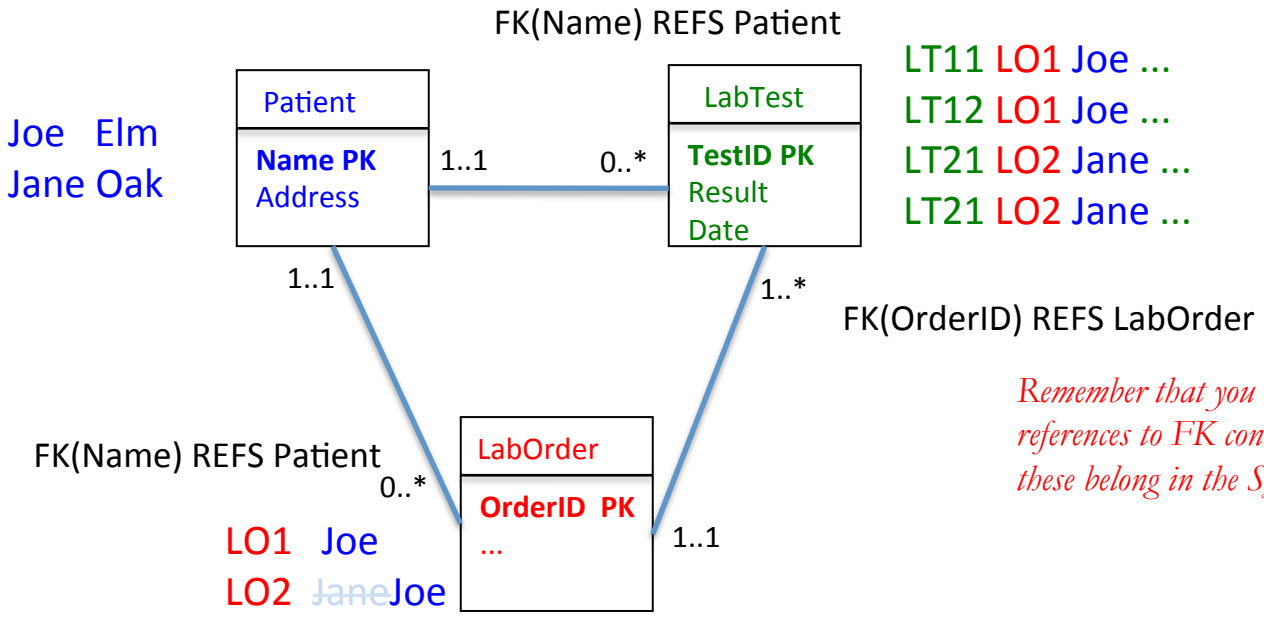


Suppose you have patients Joe and Jane, shown as examples above.

- Joe has lab order LO1 with particular tests LT11 and LT12.
- Jane has lab order LO2 with particular tests LT21 and LT22.

Above is the DB as we would intend it – correct and consistent (i.e., Joe always associated with same tests and orders; Jane always associated with same tests and orders)

But can Joe, for example, be mistakenly associated with LO2 in LabOrder, while Jane is associated with LO2 in LabTest, without causing a constraint violation? YES! "Circularity" can introduce redundancy that enables inconsistency



Remember that you would NOT put explicit references to FK constraints in a UML – rather these belong in the SQL translation of the UML

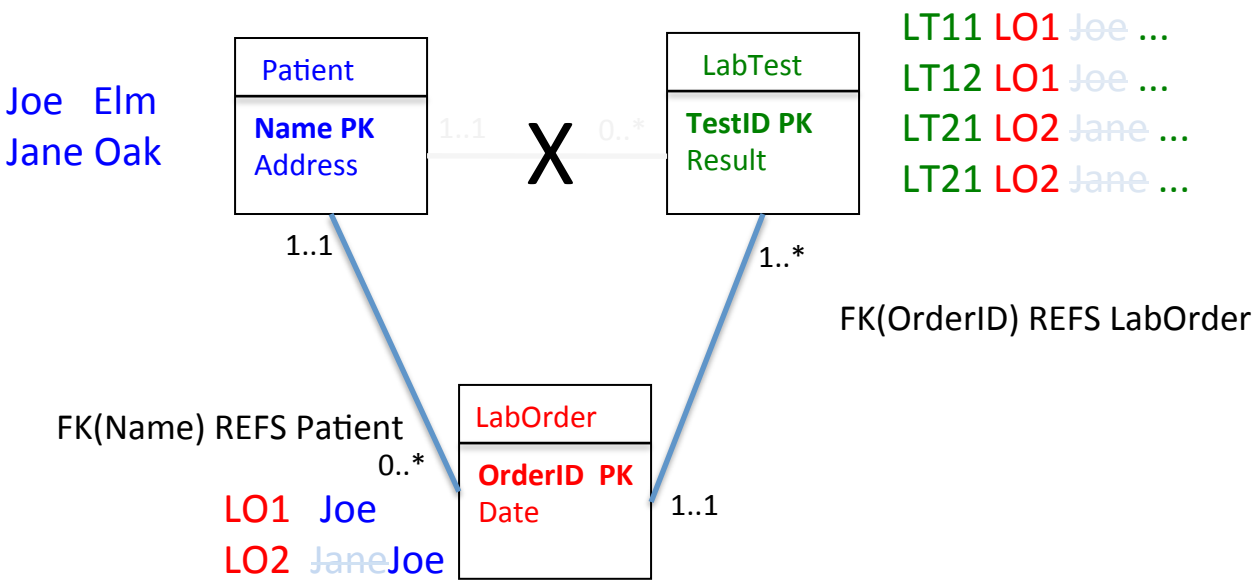
Joe could be erroneously associated with LO2 in LabOrder, and Jane could be associated with LO2 in LabTest, and there would be no constraint violation, of either the FK constraint from LabOrder to Patient or the FK constraint from LabTest to LabOrder.

This example isn't about wrongness per say, its about inconsistency – LO2 is now, in different parts of the database, associated with both Jane and Joe, and still, no constraint violation.

We could write an assertion or trigger to ensure consistency, or do some other design modification to ensure consistency. Some might think that redundancy, for purposes of error checking, can be good ... and it can be, like parity bits, but in this DB case can be expensive (interesting thought: think about how you might use PK box construct(s) to build in redundancy that can be easily checked within table at the cost of a couple of additional fields per tuple).

My point here is just be aware of inconsistency that can arise from "circularity" and in any case.

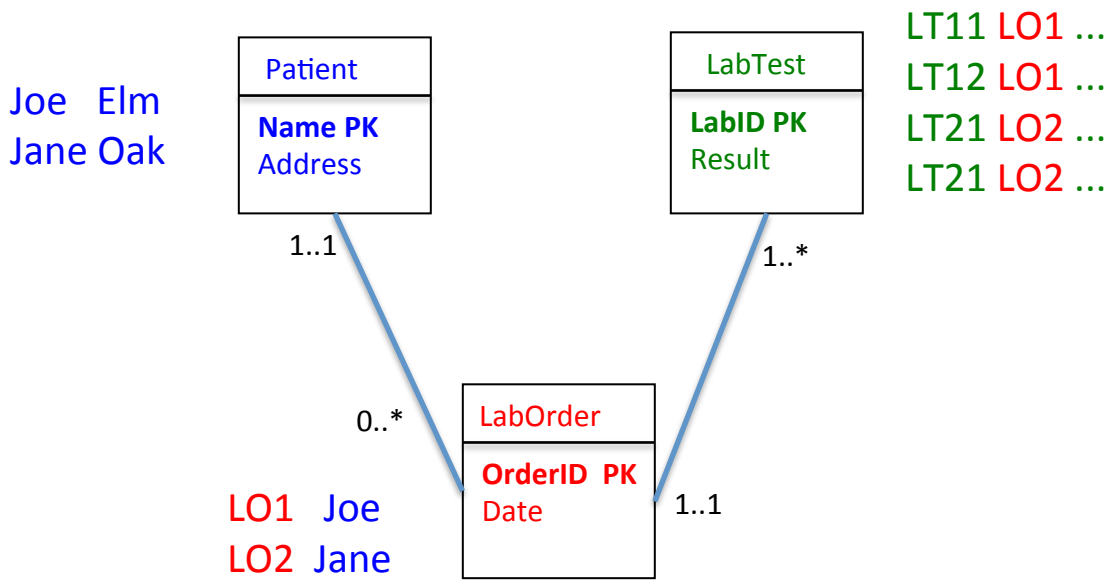
Consider removing the Patient/LabTest link



LO2 is now unambiguously associated with Joe – it still may be incorrect, but not inconsistent!

Alternatively, a design team might think about getting rid of the LabOrder class (or making it an association class) and replacing the LabOrder table with a view, perhaps

Lesson – beware the redundancy that arises from circularity (and otherwise)



We don't need the direct connection between Patient and LabTest to access the individual lab tests of patients:

SELECT P.Name P.Address, LT.LabID, LT.Result
FROM Patient P, LabOrder LO, LabTest LT
WHERE P.Name = LO.Name AND LO.OrderID = LT.OrderID