Question 1 (3 marks)
What is the purpose of a business use-case diagram. How does this differ from a system use-case diagram?

A business (aka essential) use-case diagram is a high-level, simplified, schematic view of the processes and stakeholder interactions of the business under consideration. Such a diagram depicts two kinds of entities:

- **actors** which represent the various classes of stakeholders who interact with the business being modelled and which are depicted using “stick-men”, and
- **use-cases** each one of which denotes a business process or interaction represented as labelled oval connected by lines to the actors who may become embroiled in the execution of that process.

Importantly, each business use-case depicted in this diagram should capture the intentions of its actors in a technology and implementation independent manner. In other words, in a business use-case we are interested in understanding the purpose or intention of an interaction (the what and why of that process) rather than detailing the mechanisms and information systems by which such an interaction might be carried out in practice (the how of that process).

In contrast, a system use-case diagram depicts use-cases that describe the ways in which real-world actors interact with an information system in order to complete a business process. System use-cases elaborate and expand upon business use-cases by making high-level systems design decisions and exploring how those decisions influence the structure of business use-cases.

Once we’ve made some high-level decisions regarding the kind and nature of the information systems we intend to introduce into a business, we are then in a position to construct our system use-case diagram from our business use-case model. This can be done systematically by examining each business use-case in turn and studying how the introduction of our new information systems will impact upon the detail of the interactions and actors involved in that use case. It is highly likely that as we do so each business use-case will split into a family of highly interconnected system use-cases.

Question 2 (3 marks)
Describe the main objectives of logical database design. Identify the main steps associated with the logical database design process.

A logical database model is a model of the information used in an enterprise which is based upon a specific data model (such as the relational or object-relational models) but which is independent of the details of any particular database management system (such as Oracle or Postgres for instance).

Generally, the term logical database design describes the process by which a coherent and consistent logical database model may be derived from a conceptual model of the enterprise for which that model is intended. In this process our primary objectives are to:
derive a family of relations (aka “tables”) that correspond to the entity and relationship types in the original conceptual model,

determine the primary and foreign keys required by the implementation of our original relationship types,

ensure that this family of relations is irredundant and internally self consistent (aka normalised),

check that all of the user transactions identified during the requirements phase may be implemented using the relations and keys identified, and

specify the constraints required to maintain the integrity of the data stored in these relations.

Starting with a conceptual database model, a corresponding logical model (for the relational data model) is often derived in the following steps:

1. **Remove the features of the conceptual model that are incompatible with the relational model.** For instance, if our conceptual model uses features such as many-to-many binary relationships, ternary (or higher) relationships, attributed relationships, multi-valued attributes or extended features (e.g. entity type inheritance) then these need to be eliminated by introducing new entity types to our model.

2. **Derive relations for the logical data model.** This involves constructing a primary key for each weak entity type, creating a relation (table) for each entity type and determining the foreign keys needed to mediate each 1:1 and 1:many relation.

3. **Validate the relations we’ve constructed using normalisation.** We ensure that each relation is at least in Boyce-Codd Normal Form (BCNF) in order to ensure that our logical model is as robust and free of update anomalies as possible. This process forces the designer to understand each attribute completely and leads to a flexible and easily extended logical model.

4. **Validate relations against user transactions.** In principle, this step is as easy as showing that the logical model supports an appropriate SQL query for each user transaction described in the original requirements specification.

5. **Derive integrity constraints.** These should ensure that a database built on our logical model would never become inconsistent. Generally we consider issues such as whether a particular entity (column) can hold nulls, attribute type constraint, entity integrity (ensuring that a table’s primary key cannot hold nulls), referential integrity (ensuring that the tables involved in a primary / foreign key relationship are updated in tandem) and entity constraints (business rules).

6. **Review the logical data model with stakeholders and users.** To double check that the logical data model and its supporting documentation correctly are a true representation of the data model identified in the process of deriving system use-cases.

**Question 3 (6 marks)**
Identify the set of functional dependencies for the data shown in the figure above.

In this record we can identify 13 attributes: PatientNumber, FullName, WardNumber, BedNumber, WardName, DrugNumber, Name, Description, Dosage, MethodOfAdmin, UnitsPerDay, StartDate and FinishDate. Making some pretty straightforward assumptions about the relationship between these attributes in a practical situation, we get the following functional dependencies:

- PatientNumber -> FullName, BedNumber, WardNumber
- WardNumber -> WardName
- DrugNumber -> Name, Dosage, MethodOfAdmin
- Name -> Description
- PatientNumber, DrugNumber, StartDate -> EndDate, UnitsPerDay
- PatientNumber, DrugNumber, EndDate -> StartDate

Notice that while Dosage and MethodOfAdmin are functionally dependent on DrugNumber, they are not functionally dependent on Name, since a named drug such as “Morphine” can be administered in more than one way (IV or Oral for instance) and can have different dosages, which might depend upon, for instance, the condition the drug is being used to treat.

The last couple of functional dependencies don’t fully capture the corresponding “business rule” on the ground, which might state that on a given day a given patient is administered a given drug in a dose equal to a specified number of units. Our rule deals only with ranges of days, which may overlap resulting in two distinct dosages being specified on days in the overlap between two ranges.

Draw a schema based on the data above that is in first normal form (but not fully normalised).
either of the triples of attributes (PatientNumber, DrugNumber, StartDate) or (PatientNumber, DrugNumber, FinishDate). Thus, both of these are suitable candidate keys and we chose the former as our primary key.

So a suitable 1st normal form schema is:

MedicationRecord(  
    PatientNumber, FullName, WardNumber, BedNumber, WardName,  
    DrugNumber, Name, Description, Dosage, MethodOfAdmin  
    StartDate, FinishDate, UnitsPerDay )

where the attributes making up our composite primary key are underlined.

Diagrammatically, we might represent this schema as:

(c) Identify a potential insert anomaly for this schema.

The most important insertion anomaly that arises in this schema is that we cannot insert a patient record unless that patient currently has a medication regime, since the DrugNumber and StartDate columns cannot contain nulls.

(d) Describe and illustrate the process of normalising this schema to second (2NF) and third (3NF). For each stage, identify the primary keys.

To get our schema into 2nd normal form, we need to split our table up into a number of smaller tables in which each non-primary key attribute is fully functionally dependent on the primary key in its table. We say that a functional dependency A -> B is full if the removal of any attribute from A results in the dependency not being sustained any more.

In this case we need to split our relation into 3 pieces, the first two containing patient information and drug information respectively and the third relating them in a medication record, as in the ER diagram below:
To get to 3rd normal form we need to eliminate transitive dependencies between non-primary key attributes and primary keys. In our model, we have two such dependencies: \( \text{DrugNumber} \rightarrow \text{Name} \rightarrow \text{Description} \) and \( \text{PatientNumber} \rightarrow \text{WardNumber} \rightarrow \text{WardName} \). Again these are eliminated by introducing new tables, as in the ER diagram below:
Question 4 (9 marks)

This question assumes a schema `Students(id, name, address, age)`, and is based the following PL/SQL procedure `Proc1`.

```
CREATE OR REPLACE PROCEDURE Proc1 (x VARCHAR2, y VARCHAR2) AS
  reco1 Students%ROWTYPE;
  reco2 Students%ROWTYPE;
  reco3 Students%ROWTYPE;
BEGIN
  SELECT * INTO reco1
  FROM Students
  WHERE Name = x;

  SELECT * INTO reco2
  FROM Students
  WHERE Name = y;

  IF reco1.Age < reco2.Age THEN
    reco3 := reco1;
    reco1 := reco2;
  END IF;
END;
```
reco2 := reco3;
END IF;

DBMS_OUTPUT.PUT_LINE ( reco1.Name || ', ' || Reco2.Name);
END Proc1;
/

(a) Explain what the procedure Proc1 is meant to do. (2 marks)

This procedure simply obtains the records of two named individuals, orders them in age order and prints their names in that order.

(b) It has been brought to your attention that name is not the primary key of the student table. What are some potential errors with the above code. Introduce an error handling mechanism that will output appropriate messages. (2 marks)

The primary problem here is that either of the select statements in the body of this function may return no rows at all or indeed, since Name is not a primary key, multiple rows. To deal with these errors, you should add an exception handler to the body of the procedure to catch NO_DATA_FOUND and TOO_MANY_ROWS errors (details left up to the reader).

If you wanted to be a little more creative and notify the user as to which of the names supplied resulted in no rows or multiple rows being selected, you might like to enclose each SELECT statement in a BEGIN ... EXCEPTION ... END block of its own and handle these exceptions locally to that block itself. Just remember to “re-raise” your exception once you’ve handled it and printed an appropriate message. This avoids the problems associated with control dropping through to the subsequent comparison statement, which would compare variables with null values.

(c) Write some SQL that displays the address, id and name of students where there are two or more students at the same address. Sort this by address. (2 marks)

One way to deal with this is to use a subselect to count the number of students at each address:

```sql
SELECT s.address, s.id, s.name
FROM students as s,
     (SELECT address, COUNT(id) as scount
      FROM students
      GROUP BY address) AS n
WHERE s.address = n.address AND
    n.scount >= 2
ORDER BY s.address;
```

(d) Use this SQL as the basis of a PL/SQL procedure called SameAddress. This will output a report for each address with more than 2 students as follows

address: address
num students : num-students-at-address
student ids : student-ids at address
Solution left up to the reader – this is some pretty routine PL/SQL.

Question 5 – SQL (7 marks)

The relational schema shown below is part of a real estate database.

Branch (branchno, street, city, postcode)
Staff (staffNo, fname, lname, position, sex, dob, annualsalary, branchno)
PropertyForRent (propertyno, street, city, postcode, type, rooms, rent, ownerno, staffno, branchno)
Client (clientno, fname, lname, telno, preftype, maxrent)
Viewing (clientno, propertyno, viewdate, comments)

Formulate the following as SQL statements:

(a) Produce a list of monthly salaries for all staff showing staff number, the first and lastnames and the monthly salary.

SELECT staffNo, fname, lname, annualsalary / 12
FROM Staff;

(b) List the addresses of all branch offices in London or Glasgow.

SELECT street, city, postcode
FROM Branch
WHERE city = 'London' OR
   city = 'Glasgow';

(c) Find the total number of managers and the sum of their salaries.

SELECT COUNT(staffNo), SUM(annualsalary)
FROM Staff
WHERE position = 'Manager';

(d) Find the number of staff working in each branch and the sum of their salaries.

SELECT branchNo, COUNT(staffNo), SUM(annualsalary)
FROM Staff
GROUP BY branchNo;

(e) List the staff number and last name of staff who work in the branch at “13 Main Road”. Use a nested subquery.

SELECT staffNo, lname
FROM Staff
WHERE branchNo = (SELECT branchNo
                   FROM Branch
                   WHERE street = '13 Main Road');
(f) List the branch offices and properties that are not in the same city as the branch office at which that property is registered. It will be sufficient to list the branch number, the branch city, the property number and the city of the property.

SELECT b.branchNo, b.city AS branchCity,
       p.propertyNo, p.city AS propertyCity
FROM Branch AS b,
     PropertyForRent AS p
WHERE b.branchNo = p.branchNo AND
     b.city <> p.city;

(g) Construct a list of all cities where there is both a branch office and a property using nested subqueries.

SELECT DISTINCT city
FROM Branch
WHERE city IN (SELECT DISTINCT city
               FROM PropertyForRent);