8.1

- **Fragmentation transparency**
  The distributed database behaves as if it was centralized. There is no need for the user to know how the data is fragmented, where it is placed or how it has to be accessed.

- **Location Transparency**
  The user must know how data is fragmented (that means he must know which fragment of the database has to be considered to satisfy his information need). He does not have to know where these fragments are located (they could e.g. be replicated at different sites).

- **Local mapping transparency**
  User must know data fragmentation and localization. He need not know details about the access mechanism of the possibly different DBMS types at the different sites.

- **No transparency**
  The user must know exactly how data is fragmented, where it is located and how it is to be accessed (is a relational database that can be queried by SQL statements, is it an XML DBMS that has to be accessed by XQuery,...)
8.2

a) cuboid is subtype of cube

```plaintext
class cube
{
    attribute float length;
    float volume();
    void rotate(in float degree);
}

class cuboid extends cube
{
    attribute float width;
    attribute float height;
    float volume();
    void rotate(in float degree);
}
```

Advantages:
- Each object type has exactly the attributes and methods which are required.

Problems:
- Not much reusability of functions: most functions for cuboid have to be redefined.
- Cuboid is a subtype of cube means that every cuboid is a cube (which is obviously not the case). Therefore, we can pass a cuboid as an argument to a function where a cube is expected. This will lead to unexpected/undefined results.

b) cube is subtype of cuboid

```plaintext
class cuboid
{
    attribute float length;
    attribute float width;
    attribute float height;
    ...
}

class cube extends cuboid
{
    ...
}
```

Advantages:
- Specialization is represented according to the extension of the classes in the real world (i.e. all cubes are also cuboids).
- Operations for cuboid can be reused by cube. However, it might be more efficient to overwrite some operations because they are simpler for a cube (e.g. rotate).

Problems:
- Cube inherits attributes (width + height) which are not necessary
  - Program must maintain consistency (length=width=height).
  - More memory required.
- Cube might inherit operations which are not applicable to a cube (e.g. setWidth)

Conclusion
There is no common solution to such problems. It depends on the context and the requirements of the case. As a database designer, you might want to choose option a) as it requires less memory. As a programmer you might choose option b) as it ensures consistency.
8.3

a)

```plaintext
class Hotel
    ( extent all_hotels
    key hid
    )
    {
        attribute short hid;
        attribute short capacity;
        attribute enum ClassValues{1,2,3,4,5} class;
        attribute struct HotelLocation {
            string city;
            string country;
        } location;
    }
    relationship set<Employee> has_emps
        inverse Employee: works_for;
    relationship set<Room> has_rooms
        inverse Room: belongs_to;
}
class Room
    ( extent all_rooms
    key rid
    )
    {
        attribute short rid;
        attribute float price;
        attribute enum CategoryValues{single, double} category;
    }
    relationship Hotel belongs_to
        inverse Hotel: has_rooms;
    relationship set<Maid> has_maids
        inverse Maid: responsible_for;
    void addMaid(in Maid newMaid);
    void setCategory(in CategoryValues category);
}
class Employee
    ( extent all_employees
    key ssn
    )
    {
        attribute short ssn;
        attribute string name;
        attribute float salary;
    }
    relationship Hotel works_for
        inverse Hotel::has_emps;
}
class Maid extends Employee
    ( extent all_maids
    )
    {
        attribute enum OccupationValues{full_time, part_time} occupation;
    }
    relationship set<Room> responsible_for
        inverse Room: has_maids;
}
class Manager extends Employee
    ( extent all_managers
    )
    {
        attribute string email;
        attribute set<string> telephones;
    }
```
b)

- SELECT m.ssn
  FROM m IN all_maid, r IN m.responsible
  WHERE r.category=single AND r.belongs_to.class=3

- SELECT struct (hid: h.hid, class: h.class,
  max_salary:
    max(SELECT salary
      FROM m IN all_managers
      WHERE m.works_for=h
    )
  )
  FROM h IN all_hotels
a) **Object Types:**

CREATE TYPE address_t AS OBJECT (  
    street varchar2(100),  
    city varchar2(100),  
    zip integer  
);  

CREATE TYPE person_t AS OBJECT (  
    name varchar2(100),  
    has_address REF address_t  
) NOT FINAL;  

CREATE TYPE producer_t UNDER person_t (style varchar2(100));  

CREATE TYPE instruments_t AS TABLE of varchar2(100);  

CREATE TYPE performer_t UNDER person_t (instruments instruments_t);  

CREATE TYPE performers_t AS TABLE OF REF performer_t;  

CREATE TYPE song_t AS OBJECT (  
    title varchar2(100),  
    length integer,  
    performed_by performers_t  
);  

CREATE TYPE songs_t AS TABLE OF REF song_t;  

CREATE TYPE album_t AS OBJECT (  
    genre varchar2(100),  
    title varchar2(100),  
    publication_date date,  
    produced_by REF producer_t,  
    contains songs_t  
);  

**Object Tables:**

CREATE TABLE address OF address_t;  

CREATE TABLE person OF person_t;  
ALTER TABLE person ADD (SCOPE FOR (has_address) IS address);  

CREATE TABLE producer OF producer_t;
CREATE TABLE performer OF performer_t
    NESTED TABLE instruments STORE AS instruments_nt;

CREATE TABLE song OF song_t
    NESTED TABLE performed_by STORE AS performers_nt;
ALTER TABLE performers_nt ADD (SCOPE FOR (COLUMN_VALUE) IS performer);

CREATE TABLE album OF album_t
    NESTED TABLE contains STORE AS songs_nt;
ALTER TABLE album ADD (SCOPE FOR (produced_by) IS producer);
ALTER TABLE songs_nt ADD (SCOPE FOR (COLUMN_VALUE) IS song);

b)

INSERT INTO address VALUES (  
    ‘Black Avenue’,
    ‘New York’,
    ‘70’
);

INSERT INTO producer VALUES (  
    ‘John Smith’,
    (SELECT REF(a)  
     FROM address a  
     WHERE a.street= ’Black Avenue’ AND  
     a.city= ‘New York’ AND  
     a.zip= 70  
    ),  
    ‘Heavy Metal Producer’
);

INSERT INTO address VALUES (  
    ‘Metal Street’,
    ‘Miami’,
    ‘32’
);

INSERT INTO performer VALUES (  
    ‘Metallica’,
    (SELECT REF(a)  
     FROM address a  
     WHERE a.street= ’Metal Street’ AND  
     a.city= ‘Miami’ AND  
     a.zip= 32  
    ),
    instruments_t(‘Guitar’, ‘Drums’, ‘Bass’)
);
INSERT INTO song VALUES (
    ‘One’,
    460,
    performers_t((SELECT REF(p) FROM performer p
        WHERE p.name=’Metallica’))
);

INSERT INTO album VALUES (
    ‘Heavy Metal’,
    ‘And Justice For All’,
    cast(‘4-5-1988’ as date),
    (SELECT REF(p) FROM producer p WHERE p.name= ’John Smith’),
    songs_t((SELECT REF(s) FROM song s WHERE s.title= ’One’))
);

c) Queries:

a. SELECT DEREF(DEREF(a.produced_by).has_address)
    FROM album a
    WHERE a.genre= ’Funk’;

b. SELECT DEREF(p.COLUMN_VALUE).name
    FROM album a, TABLE(a.contains) c,
    TABLE(DEREF(c.COLUMN_VALUE).performed_by) p
    WHERE a.genre= ’Funk’;

c. SELECT p.name
    FROM performer p, TABLE(p.instruments) i
    WHERE i.COLUMN_VALUE= ’Piano’;

d. SELECT DEREF(DEREF(a.produced_by).has_address).city,
    DEREF(a.produced_by).style
    FROM album a,
    TABLE(a.contains) s,
    TABLE(DEREF(s.COLUMN_VALUE).performed_by) per,
    TABLE(DEREF(per.COLUMN_VALUE).instruments) i
    WHERE i.COLUMN_VALUE= ’Guitar’;