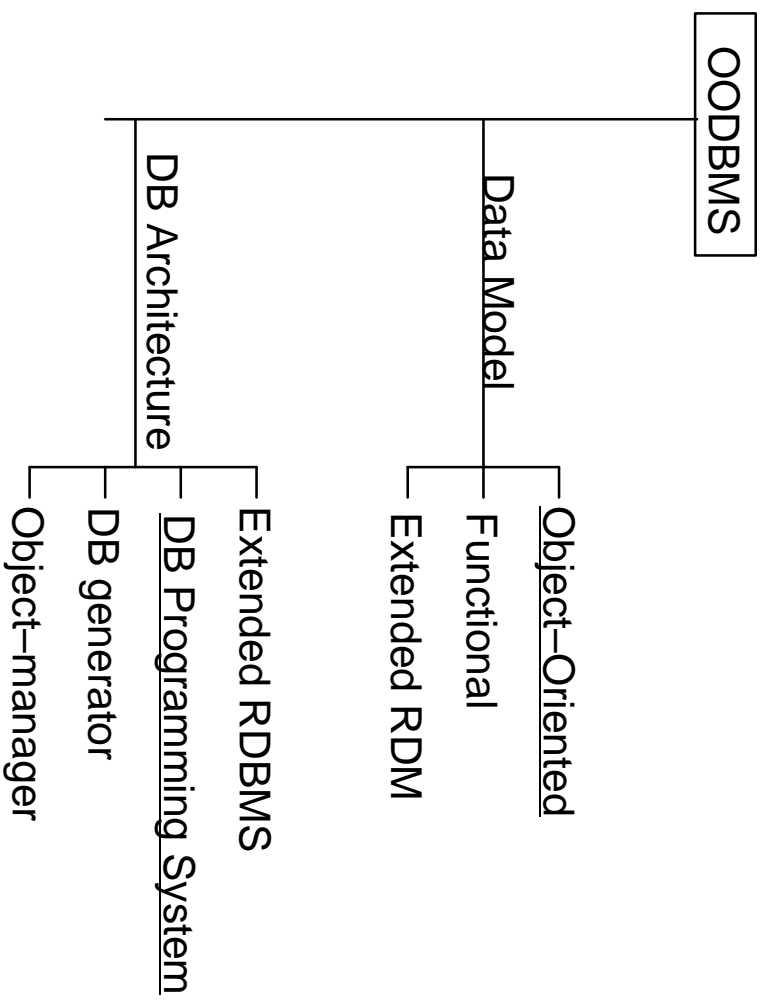


Support of Generalization and Aggregation

```
class Student : public Person {  
    private: int s-id;  
             char * degree;  
};  
class Enrol {  
    private: Student * student;  
            Unit * unit;  
};
```

- Identification: logical-Id vs through content(Key-Attributes).
- Uniqueness: in the system vs in a class hierarchy.
- Inheritance: Object-types vs Relations.
- Maintenance: Who manages it and How manage it.
- Manipulation: Operations vs Relational operations.



Database Programming Language

- DMLs are not computational complete.
- Impedance Mismatch.
- Object-oriented databases allow general purpose programming languages to produce methods



Database Language (DDL, DML)
+ Programming Language

Database Programming Language (Persistence)

Object Identity

- Identity Through Contents
 - A=B always means equality in terms of contents. No possibility to express equality in term of identity.
 - Possibility of changing the contents.
- Identity Through Location
 - Moving of an object from one location to another invalidates the identity.
 - Deletion Problem: reuse the memory which has been allocated for a deleted object.
- Logical Object Identifiers in GOM:
O = (id#, Type, Internal-State)

More about Object: Equality

```
#id-10 : <name: "Mr.Right", age: 20>
#id-11 : <name: "Mr.Right", age: 20>
#id-12 : <name: "Mr.Right", age: 40>
#id-13 : <name: "Mr.Right", age: 20>

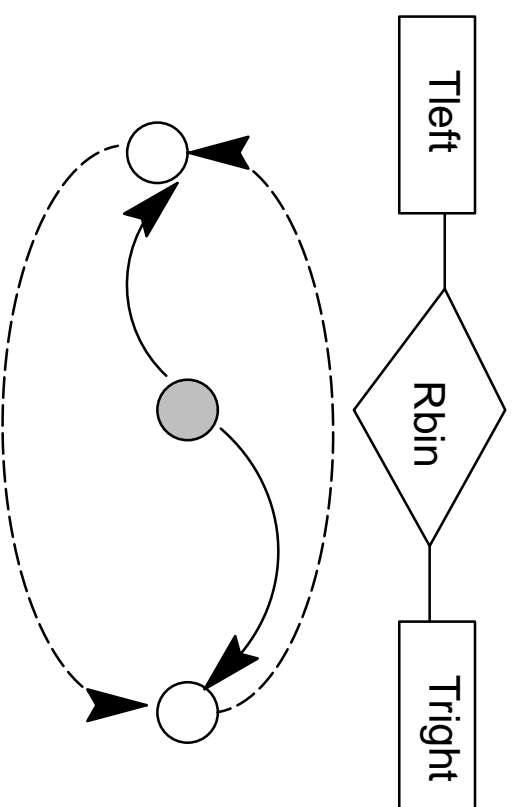
#id-10 = #id-10;          !! equal at depth 0, id-equal
#id-10 ≠ #id-11;
#id-10 =1 #id-11;       !! equal at depth 1
#id-10 ≠1 #id-12;

#id-21 : {#id-10, #id-11};
#id-22 : {#id-10, #id-13};

#id-21 ≠ #id-22;
#id-21 ≠1 #id-22;
#id-21 =2 #id-22;
```

Implementing Relationships

- 1:1 Relationship Type



Entry Points:

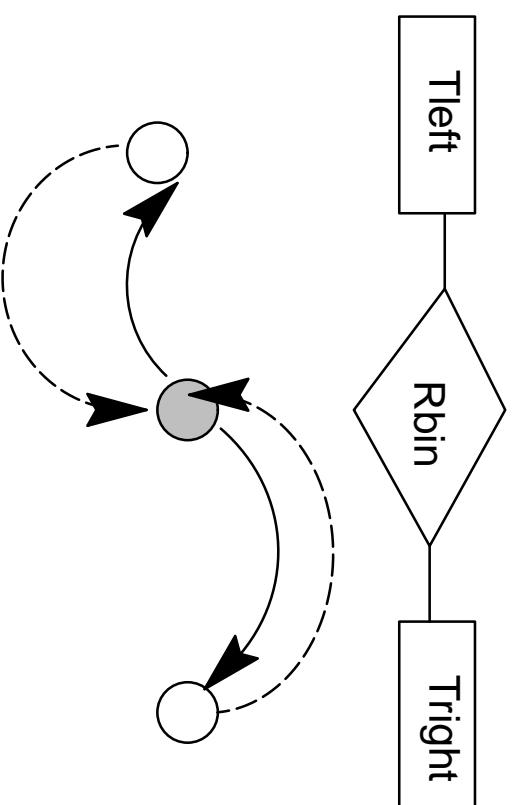
```
class Tleft {  
    private:  
        Tright* l2r;  
    public:  
        void    setRight(Tright *);  
        Tright* getRight();  
};  
  
class Tright {  
    private:  
        Tleft* r2l;  
};  
  
class TR {  
    private:  
        Tleft* left;  
        Tright* right;  
};
```

Allowing multiple entry points in the schema in order to increase the performance of the application programs might result consistency problems.

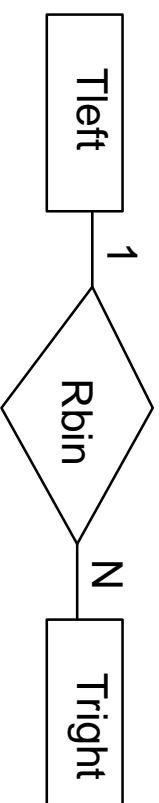
```
class Tleft {
    private:
        Tright* l2r;
    public:
        void setRight(Tright *r) {
            l2r = r; r->setLeft(this);
        };
}

class Tright {
    private:
        Tleft* r2l;
    public:
        void setLeft(Tleft *l) {
            r2l = l; l->setRight(this);
        };
};
```

In some cases, it makes perfect sense to have an explicit type for the relationship and additional attributes in either Tleft, or Tright, or in both.

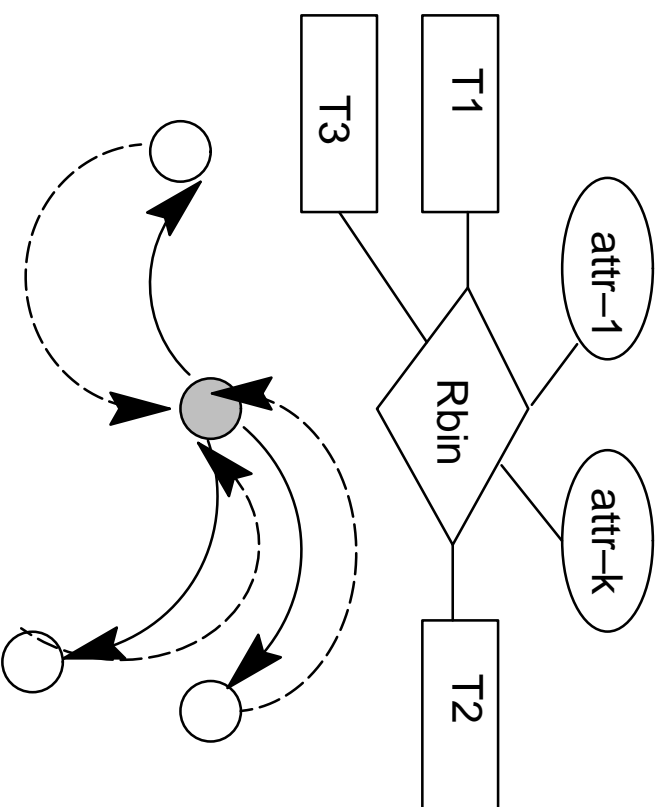


- 1: N Relationship Types: Three alternative representations.

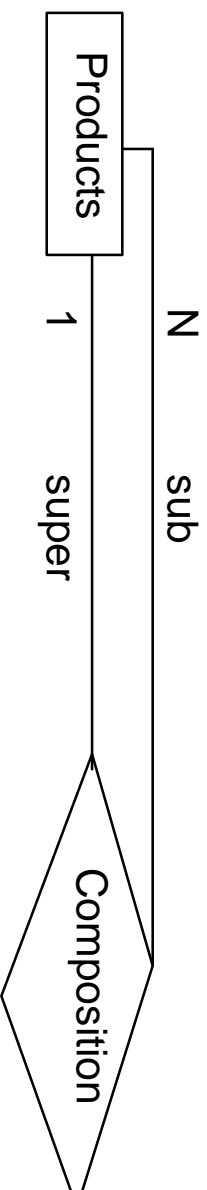


```
class Tleft {  
    private: TrightSet* l2r;  
};  
class TrightSet {  
    private: Tright  many[MAX];  
};  
class Tright {  
    private: Tleft* r2l;  
};  
class TR {  
    private:  
        Tleft* left;  
        Tright* right;  
};
```

Which one is the correct representation?



Recursive 1:N Relationship



Alternative Representation of composition.

```

class ProductSet {
    private:
        Product many[MAX];
};

class Product {
    private:
        Product *super;
    public:
        ProductSet * PartList();
        Boolean isPartof(Product*);
};

(a) super

class Product {
    private:
        ProductSet *sub;
    public:
        ProductSet * PartList();
        Boolean isPartof(Product*);
};

(b) sub
  
```

Suppose that two operations are needed.

- partList, returns the set of product instances of which the Product is composed.
- isPartOf, a Boolean function that determines whether the Product is subpart of the “higher-level” Product passed as a parameter.

For approach (a):

```
ProductSet* Product::partList() {
    ProductSet* resultSet;
    Product* part;
    resultSet = new ProductSet();
    resultSet->insert(this);

    /* Pseudo Code */
    foreach (part in extension(Product))
        if (part->super == this)
            resultSet->setUnion(part->partList());
    return resultSet;
};
```

Very inefficient

For approach (a):

```
Boolean Product::isPartOf(Product* SuperPart) {  
    Product* part;  
  
    part = this;  
    while (part != NULL)  
        if (part == SuperPart)  
            return true;  
        else part = part->super;  
    return false;  
  
};
```

Very efficient

For approach (b)

```
ProductSet* Product::partList() {  
    ProductSet* resultSet;  
    Product* part;  
  
    resultSet = new ProductSet();  
    resultSet->insert(this);  
  
    /* Pseudo Code */  
    foreach (part in this->sub)  
        resultSet->setUnion(part->partList());  
    return resultSet;  
};
```

Very efficient

For approach (b)

```
Boolean Product::isPartOf(Product* SuperPart) {  
    Product* part;  
    Boolean isUsed = false;  
  
    /* Pseudo code */  
    foreach (part in SuperPart->partList())  
        if (part == this)  
            return true;  
    return false;  
end  
};
```

Not efficient.

In summary,

```
class Product {  
    private:  
        Product *super;  
        ProductSet *sub;  
};  
  
class ProductSet {  
    private:  
        Product many[MAX];  
};
```

Inheritance and Information Hiding

```
Person(name, age, spouse);  
Employee(name, age, spouse, ss#, salary, boss);
```

Relational Data Model:

- **Reusability:**
Person(name, age, spouse);
Employee(name, ss#, salary, boss);
- **Flexibility: No Type.**
select P2.name
from Person P1, Person P2
where P1.spouse = P2.name
select P2.name
from Person P1, Employee P2
where P2.spouse = P1.name

OODBMS: reusability and flexibility are good. But How to access private members in a query language?

```
class Person : Object {  
    private:  
        char name[64];  
        nit age;  
        Dog *dog;  
    public:  
        int getage();  
        void setage();  
};  
  
select Person.age  
from Person  
where Person.age > 20;  
  
or  
select Person.getage()  
from Person  
where Person.getage() > 20
```