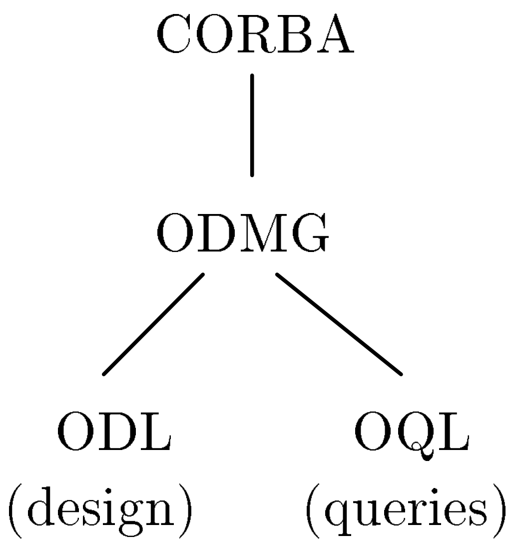
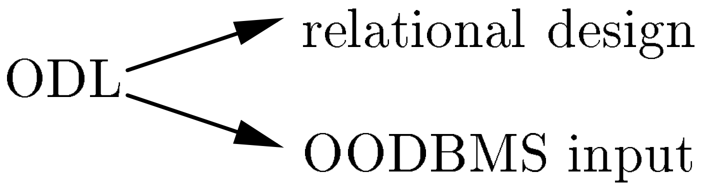
ODL

- Design language derived from the OO community:





- Can be used like E/R as a preliminary design for a relational DB.
- It can also be direct input to some OODBMS's.

ODL Class Declarations

```
interface <name> {
    elements = attributes, relationships,
        methods
}
```

Element Declarations

```
attribute <type> <name>;
relationship <rangetype> <name>;
```

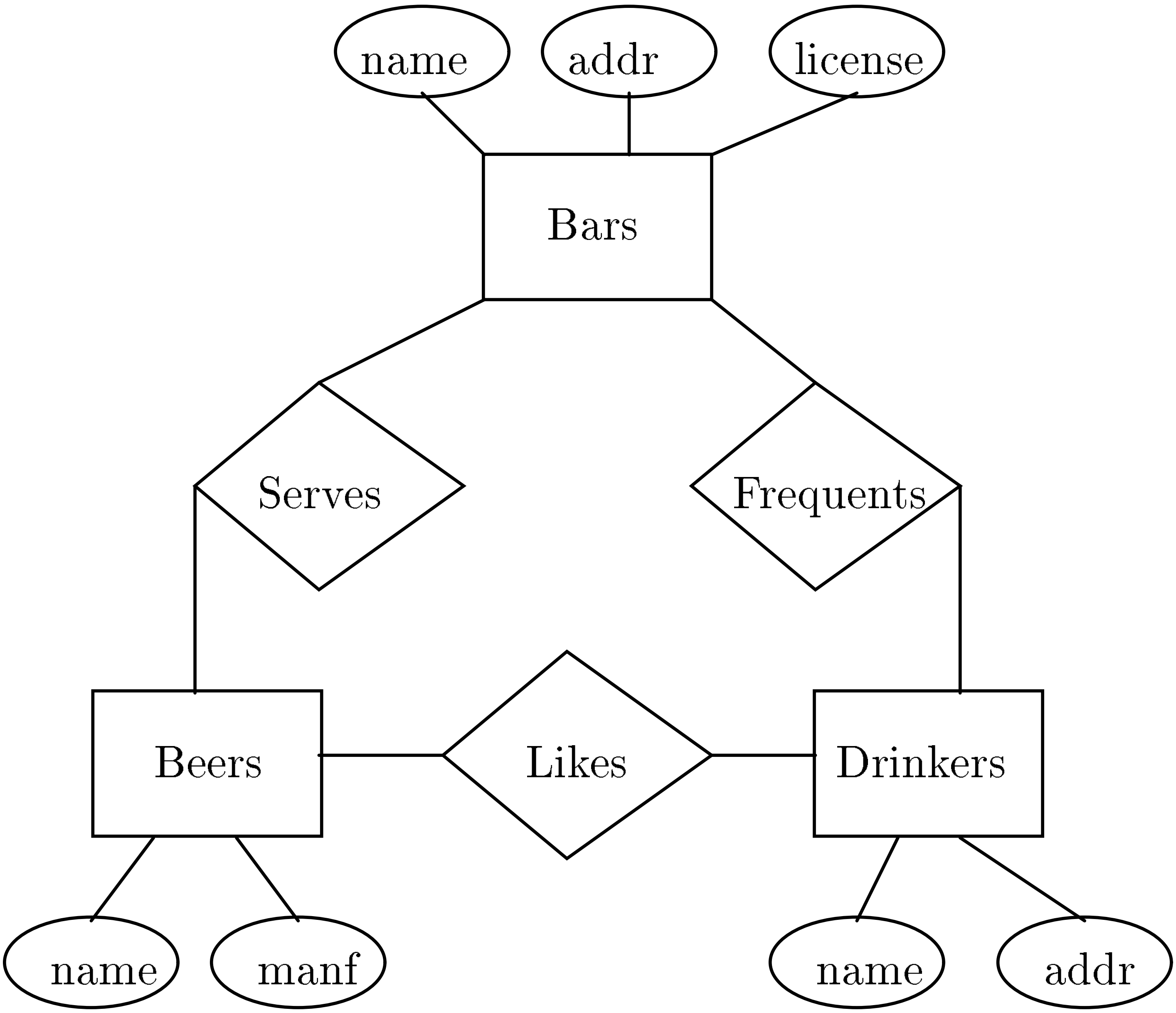
Method Example

```
float gpa(in: Student) raises(noGrades)
```

- `float` = return type.
- `in:` indicates `Student` argument is read-only.
  - Other options: `out`, `inout`.
- `noGrades` is an exception that can be raised by method `gpa`.

Beers-Bars-Drinkers Example

- Our running example for the course.



```
interface Beers {
    attribute string name;
    attribute string manf;
    relationship Set<Bars> servedAt
        inverse Bars::serves;
    relationship Set<Drinkers> fans
        inverse Drinkers::likes;
}
```

- Relationships have inverses.
- An element from another class is indicated by <class>::
- Form a set type with `Set<type>`.

```
interface Bars {
    attribute string name;
    attribute Struct Addr
        {string street, string city, int zip}
            address;
    attribute Enum Lic {full, beer, none}
            licenseType;
    relationship Set<Drinkers> customers
        inverse Drinkers::frequents;
    relationship Set<Beers> serves
        inverse Beers::servedAt;
}
```

- Structured types have names and bracketed lists of field-type pairs.
- Enumerated types have names and bracketed lists of values.

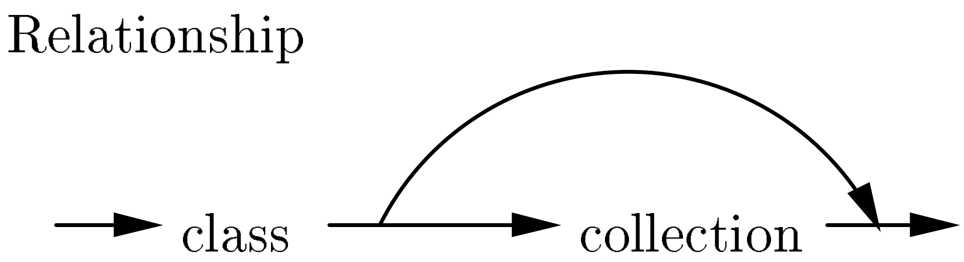
```
interface Drinkers {
    attribute string name;
    attribute Struct Bars::Addr
            address;
    relationship Set<Beers> likes
        inverse Beers::fans;
    relationship Set<Bars> frequents
        inverse Bars::customers;
}
```

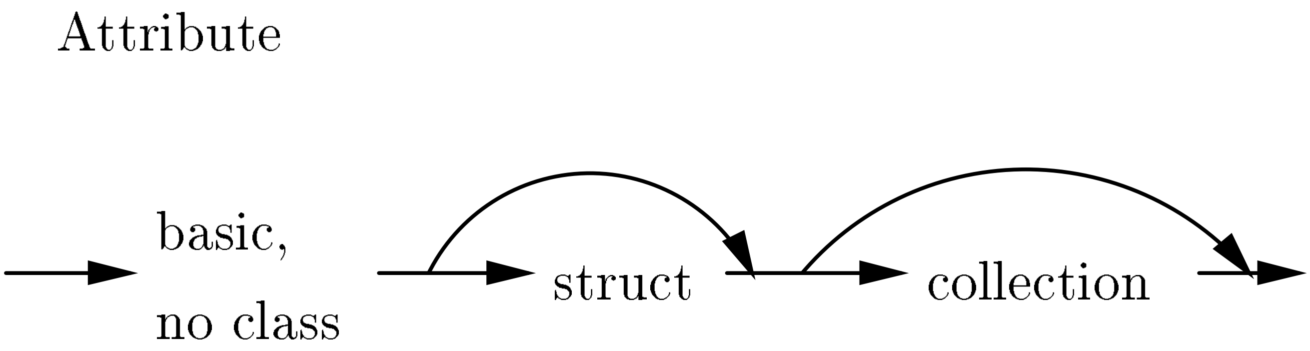
- Note reuse of `Addr` type.

ODL Type System

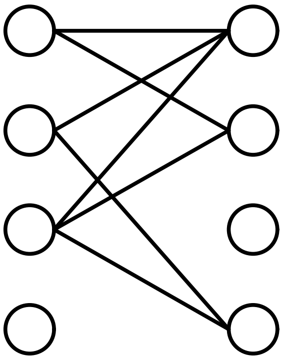
- Basic types: int, real/float, string, enumerated types, and classes.
- Type constructors: `Struct` for structures and four *collection types*: `Set`, `Bag`, `List`, and `Array`.

Limitation on Nesting

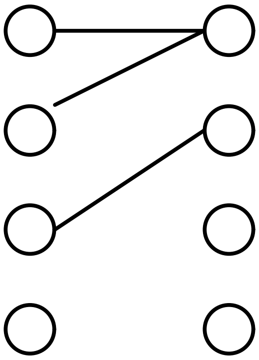




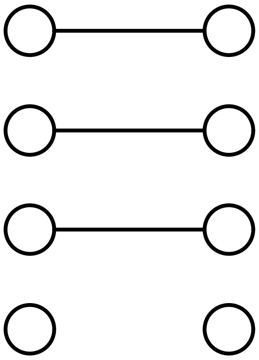
Multiplicity of Relationships



Many-many



Many-one

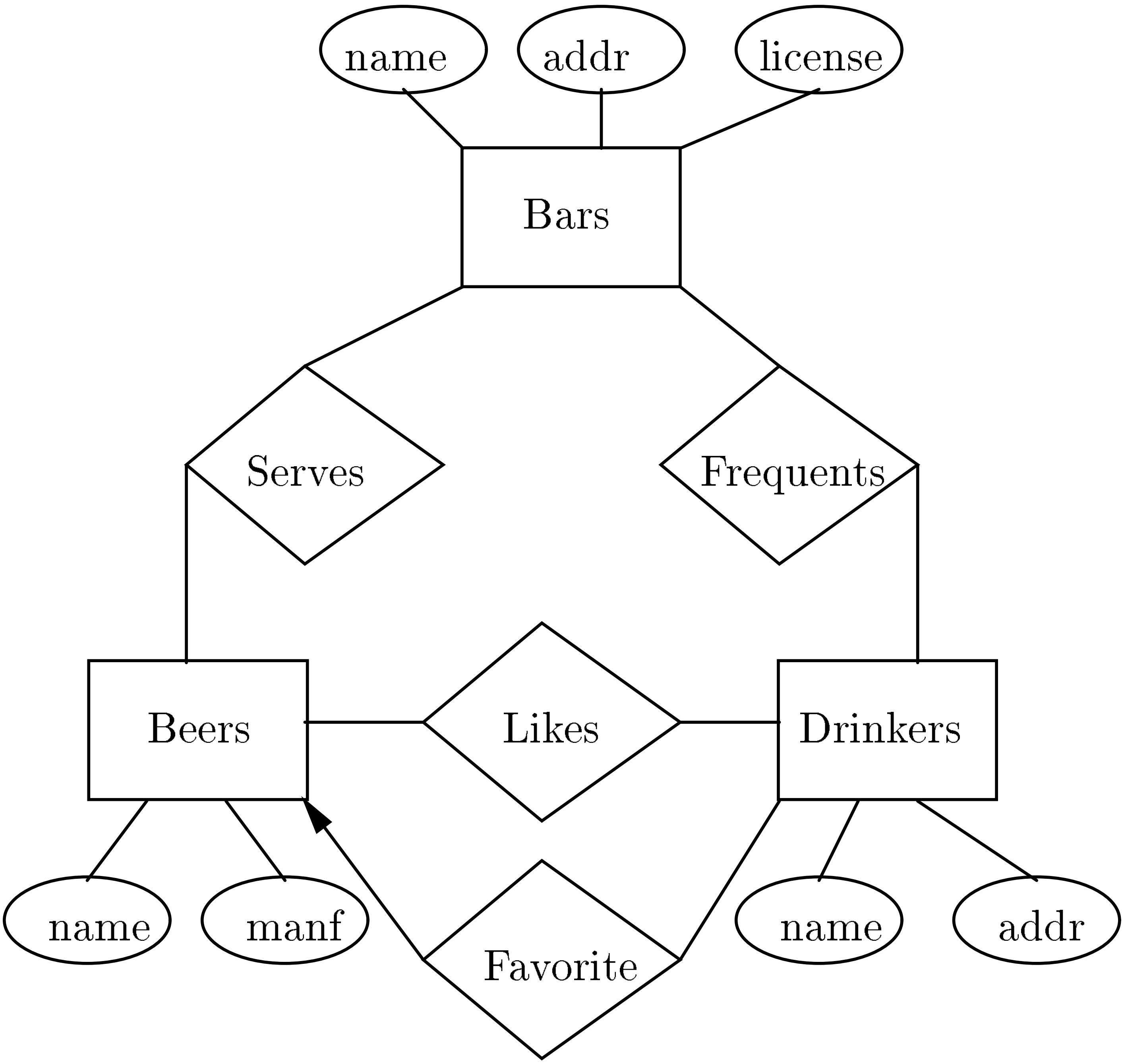


One-one

Representation of Many-One

- E/R: arrow pointing to "one."
  - ✦ Rounded arrow = "exactly one."
- ODL: don't use a collection type for relationship in the "many" class.
  - ✦ Collection type remains in "one."

Example: Drinkers Have Favorite Beers



```
interface Drinkers {
    attribute string name;
    attribute Struct Bars::Addr
            address;
    relationship Set<Beers> likes
        inverse Beers::fans;
    relationship Beers favoriteBeer
        inverse Beers::realFans;
    relationship Set<Bars> frequents
        inverse Bars::customers;
}
```

- Also add to Beers:

```
    relationship Set<Drinkers> realFans
        inverse Drinkers::favoriteBeer;
```

One-One Relationships

- E/R: arrows in both directions.
- ODL: omit collection types in both directions.

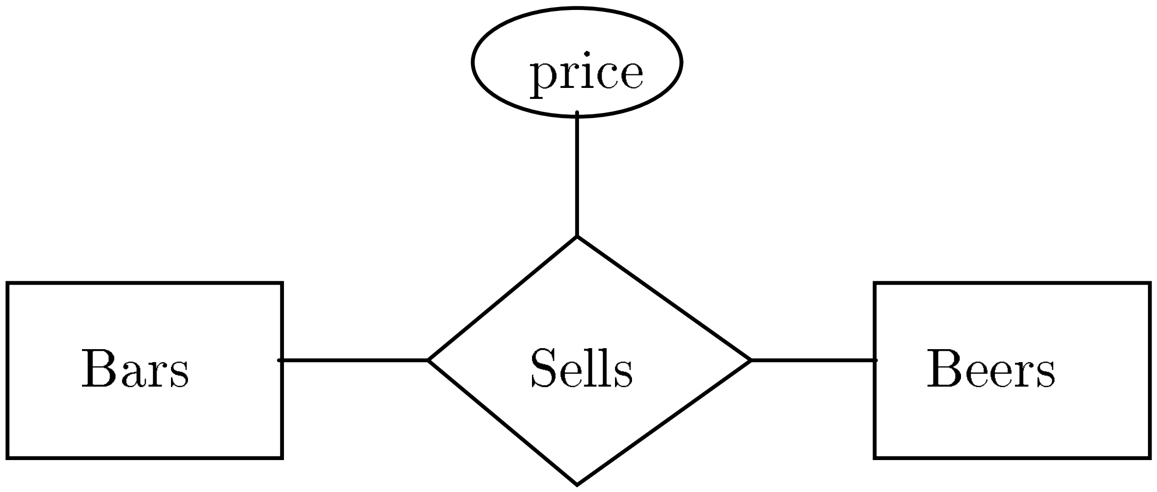
Design Issue:

Is the rounded arrow justified?

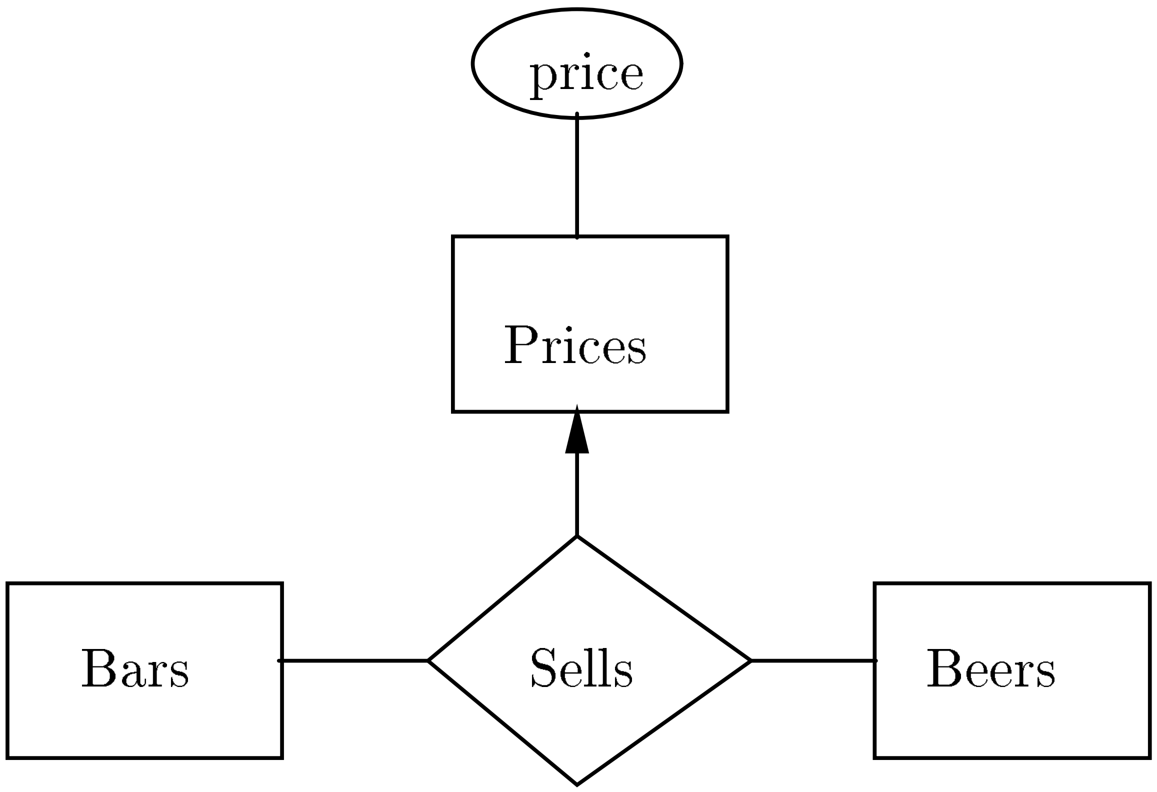
Design Issue:

Here, manufacturer is an E.S.; in earlier diagrams it is an attribute. Which is right?

Attributes on Relationships



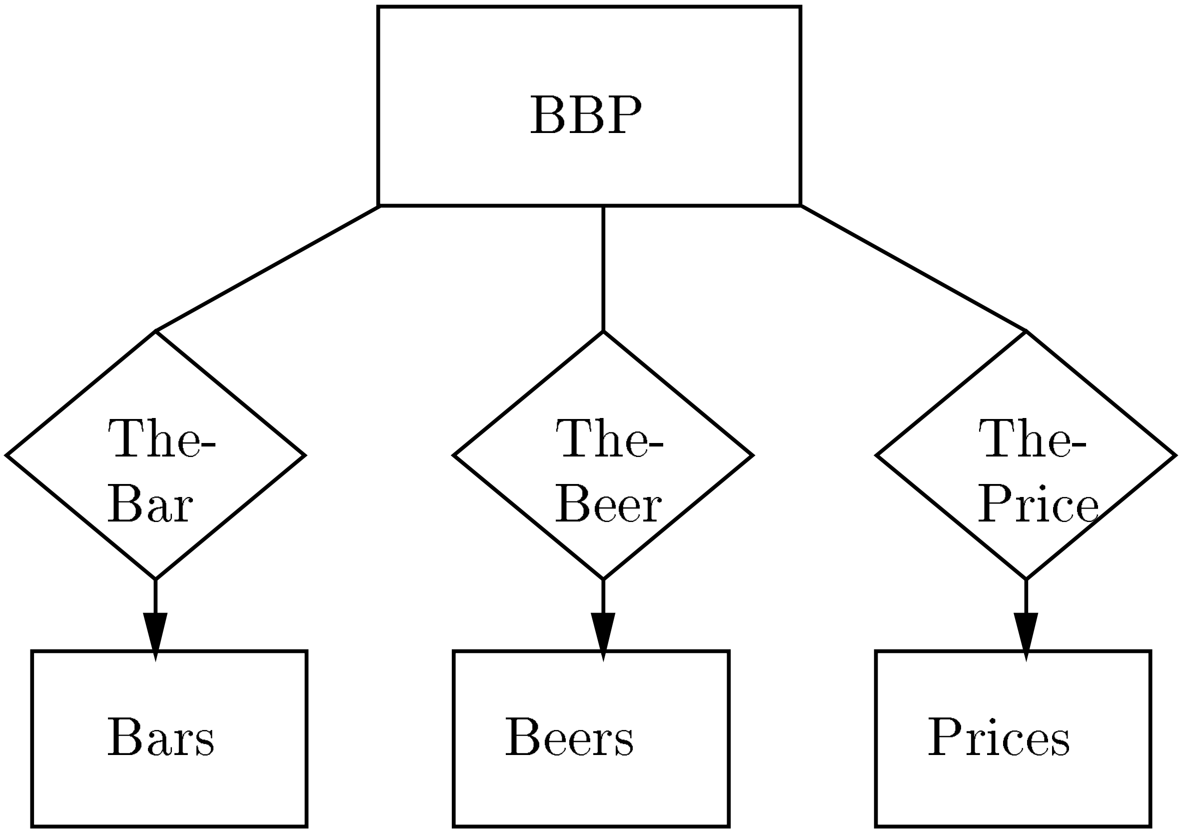
- Shorthand for 3-way relationship:



- A true 3-way relationship.
    - Price depends jointly on beer and bar.
- Notice arrow convention for multiway relationships: "all other E.S. determine one of these."
    - Not sufficiently general to express any possibility.
    - However, if price, say, depended only on the beer, then we could use two 2-way relationships: price-beer and beer-bar.

Converting Multiway to 2-Way

- Baroque in E/R, but necessary in ODL and other models.
- Create a new *connecting* E.S. to represent rows of a relationship set.
  - E.g., (Joe's Bar, Bud, \$2.50) for the *Sells* relationship.
- Many-one relationships from the connecting E.S. to the others.



Multiway in ODL Needs "Connecting" Class

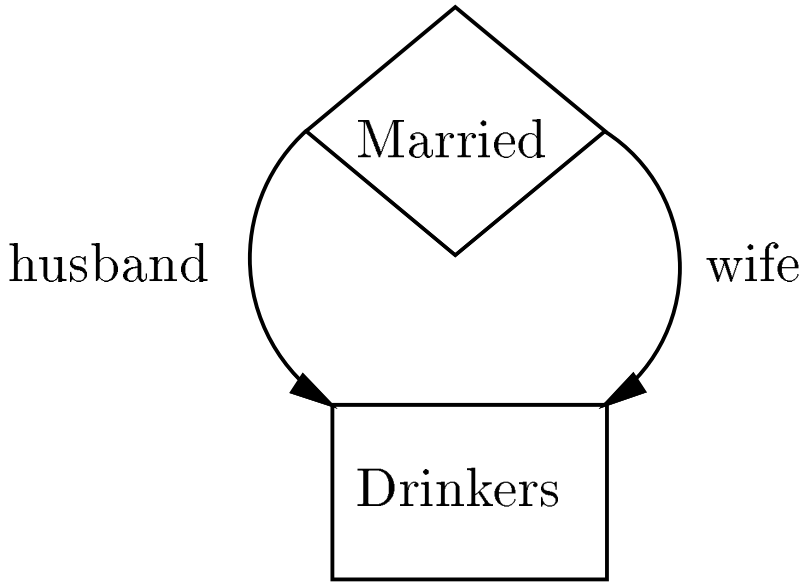
```
interface Prices {
    attribute real price;
    relationship Set<BBP> toBBP
        inverse BBP::thePrice;
}

interface BBP {
    relationship Bars theBar inverse ...
    relationship Beers theBeer inverse ...
    relationship Prices thePrice
        inverse Prices::toBBP;
}
```

- Inverses for `theBar`, `theBeer` must be added to `Bars`, `Beers`.

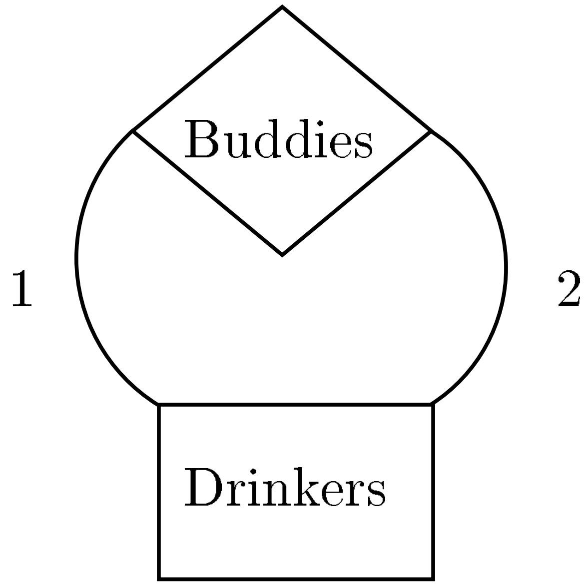
Roles

Sometimes an E.S. participates more than once in a relationship.

- Label edges with *roles* to distinguish.



Husband	Wife
d_1	d_2
d_3	d_4
$\cdots$	$\cdots$



Buddy1	Buddy2
d_1	d_2
d_1	d_3
d_2	d_1
d_2	d_4
...	...

- Notice *Buddies* is symmetric, *Married* not.
  - ✥ No way to say "symmetric" in E/R.
  - ✥ But in ODL, symmetric relations are their own inverse.

Roles in ODL

No problem; names of relationships handle "roles."

```
interface Drinkers {
    attribute string name;
    attribute Struct Bars::Addr
            address;
    relationship Set<Beers> likes
        inverse Beers::fans;
    relationship Set<Bars> frequents
        inverse Bars::customers;
    relationship Drinkers husband
        inverse wife;
    relationship Drinkers wife
        inverse husband;
    relationship Set<Drinkers> buddies
        inverse buddies;
}
```

- Notice that `Drinkers::` is optional when the inverse is a relationship of the same class.

Design Issue

Should we replace `husband` and `wife` by one relationship `spouse`?