Object-Oriented DBMSs

- Standards-based effort
  - ODMG (Object Data Management Group)
- Have produced several important standards so far
- Object Description Language ODL
  - Specifies the kind of schema-construction operations with a similar purpose to what we use in SQL's CREATE TABLE
- Object Query language
  - Provides an interface the attempts to mimic the flexibility and power of SQL, but in an OO framework

The Grand Idea

- ODMG imagines the following scenario:
  - OO-DBMS vendors implement versions of OO languages like C++ or Java
  - These languages have OQL extensions
  - Therefore a programmer could transfer data from the host language to the database (and back) in a seamless manner
- ODL is meant to define **persistent classes**
  - That is, these are objects stored permanently in the database.
  - ODL classes look like Entity sets with binary relationships along with methods.
  - ODL class definitions are part of the extended OO host language

ODL: overview

- Class declarations have several parts.

1. Name of the class
2. Optional key declarations
3. Element declarations
   - An **element** is either an attribute, a relationship, or a method

```
class <name> {
     <list of element declarations, separated by semicolons>
}
```

Declarations: attributes; relationships

- Attributes are (normally) elements with a type that does not involve classes

```
attribute <type> <name>;
```

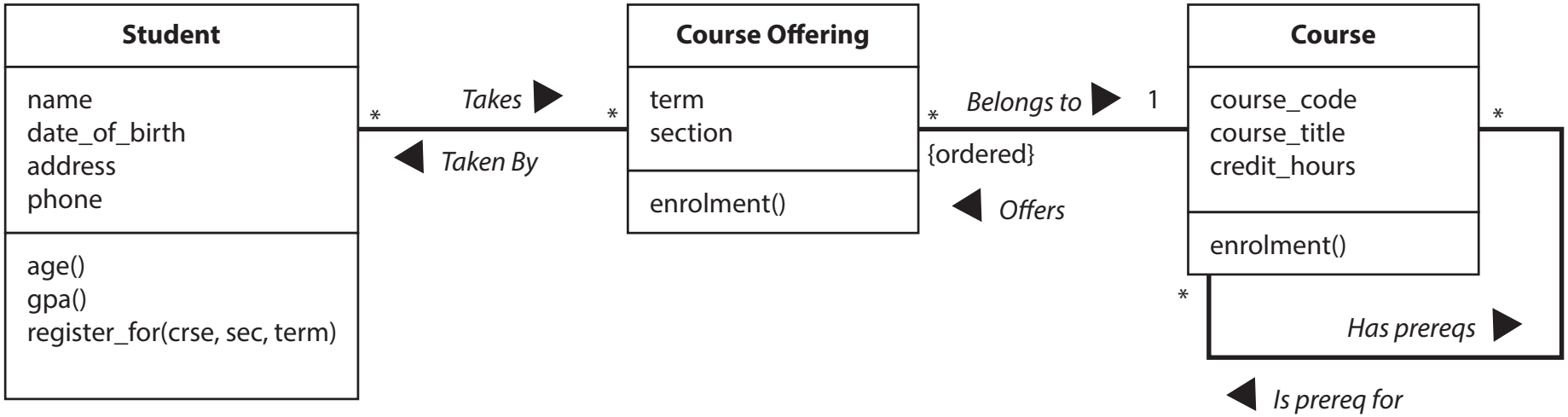
- Relationships connect an object to one or more other objects of one class

```
relationship <type> <name> inverse <relationship>;
```

Inverse relationships

- Given a class **C** that has some relationship **R** to class **D**...
- ... then class **D** must have some relationship **S** to class **C**.
- **R** and **S** must be true inverses.
  - If object **d** is related to object **c** by **R**, then **c** must be related to **d** by **S**.
- ODMG Object model supports only unary and binary relationships

Example: Course system



Example: Pubs and Beers

```
class Pub {
    attribute string name;
    attribute string addr;
    relationship Set<Beer> serves inverse Beer::servedAt;
}


class Beer {
    attribute string name;
    attribute string manf;
    relationship Set<Pub> servedAt inverse Pub::serves;
}
```

The type of relationship **serves** is a set of Beer objects.

The :: operator connects a name on the right to the context containing that name, on the left.

Relationship phrasing

- There are three possibilities.

1. A class, like Pub
   - If so, an object with this relationship can be connected to only one Pub object
2. Set<Pub>
   - The object is connected to a set of Pub objects
3. Bag<Pub>, List<Pub>, Array<Pub>
   - The object is connected to a bag, list, or array of Pub objects respectively.

Multiplicity of relationships

- Recall that all relationships are either binary or unary
- Many-to-many relationships
  - Have Set<...> for the type of relationship
  - Also for its inverse
- Many-to-one relationships
  - Have Set<...> in the relationship of the **one**...
  - ... and just the class for the relationship of the **many**
- One-to-one relationships
  - Have classes as the type in both directions

Example: Multiplicity

```
class Patron {
    attribute ...
    relationship Set<Beer> likes inverse Beer::fans;
    relationship Beer favBeer inverse Beer::superfans;
}


class Beer {
    attribute...
    relationship Set<Patron> fans inverse Patron::likes;
    relationship Set<Patron> superfans
        inverse Patron::favBeer;
}
```

Many-many uses Set<…>
in both directions.

Many-one uses Set<…>
only with the “one.”

Example: Multiplicity

husband and wife are one-one and inverses of each other.

```
class Patron {
    attribute ...
    relationship Patron husband inverse wife;
    relationship Patron wife inverse husband;
    relationship Set<Patron> buddies
        inverse buddies;
}

class PoliticallyCorrectPatron {
    attribute ...
    relationship Patron spouse1 inverse spouse2;
    relationship Patron spouse2 inverse spouse1;
    relationship Set<Patron> specialFriends
        inverse specialFriends;
}
```

buddies is many-many and its own inverse. Note no :: needed if the inverse is in the same class.

Coping with multiway relationships

- In a word: not very well.
- ODL does not support three-way or higher-degree relationships
- We may simulate multiway relationships
  - Use a **connecting class**
  - Objects represent tuples of objects we would like to connect by the multiway relationship

Connecting classes

- Imagine we want to connect classes **F**, **G** and **H** by some relationship **R**.
- To do this, we create a class **C**:
  - Its object instances represent a triple of objects **(f, g, h)**
  - These are, of course, meant to be instances **F**, **G**, and **H**
- We therefore construct three many-to-one relationships.
  - They go from the **(f, g, h)** object to each of **f**, **g**, and **h**.

Example

- Recall our Pub and Beer classes
  - In an E/R diagram we represented the price at which the beer is sold in the pub.
  - Many-to-many relationships in ODL cannot have a price attribute as is possible in the E/R model.
- One solution:
  - Create class Price...
  - ... and a connecting class PBP ...
  - ... to represent a related pub, beer, and the price.

Example

- Unlike E/R diagrams, attributes in ODL can have a structure
  - Just like C
- We declare this by using some new notation:

```
attribute Struct { <details> } <name of attribute>;

attribute Enum { <details> } <name of attribute>;
```

- The <details> portion contains:
  - Field name(s) for a Struct
  - Constant(s) for an Enum

Example: Struct and Enum

```
class Pub {
    attribute string name;

    attribute Struct Addr
        {string street, string city, string postalCode} address;
    attribute Enum Lic
        {FULL, BEER, NONE} license;
    relationships ...
}
```

Names for the structure and enumeration

names of the attributes

Method declarations

- ODL classes may include declarations of methods for the class
  - With this notation, of course, we do specify the method's definition!
- Information for a class:
  - return type (if any)
  - method name
  - argument modes and types (no names!) where modes are **in**, **out** and **inout**
  - any exceptions the method may raise

Example: Methods

```
real gpa(in string) raises (noGrades);
```

1. The method **gpa** returns a **real** number (and we deduce here it is probably some student's GPA)
2. **gpa** accepts a single argument of type **string** (again we deduce it to be the student's name)
   1. The value for the argument will only ever be passed **in**
   2. In a full OO programming language, we would mark the parameter as constant or final
3. **gpa** may raise the exception noGrades

The ODL type systems

- Basic types:
  - int, real/float, string, enumerated types, classes
- Type constructors:
  - Struct for structures
  - Collection types such as Set, Bag, List, Array and Dictionary
- Relationship types can only be either:
  - a class
  - a single collection type applied to a class

Subclassing in ODL

- This is the expected object-oriented notion of subclasses
- We indicate superclass with a colon (':') and the superclass name
  - This notation is borrowed from C++
- A subclass will list only the properties unique to it
  - That is, the subclass will inherit the properties of its superclass (no surprise there)

Example

```
class Ale:Beer {
    attribute string colour;
    attribute int    IBU;
}
```

Keys in ODL

- Unlike E/R you can declare any number of keys for a class
- These keys appear after the class name

```
(key <list of keys> )
```

- A key consisting of more than one attribute needs additional parentheses around the attribute set

Example: Keys

- **name** is the key for Beers

```
class Beer (key name) { ... }
```

- For a Course class:
  - **dept** and **number** form one key
  - **room** and **hours** form another key

```
class Course
   (key (dept, number),
   (key (room, hours)) { ... }
```

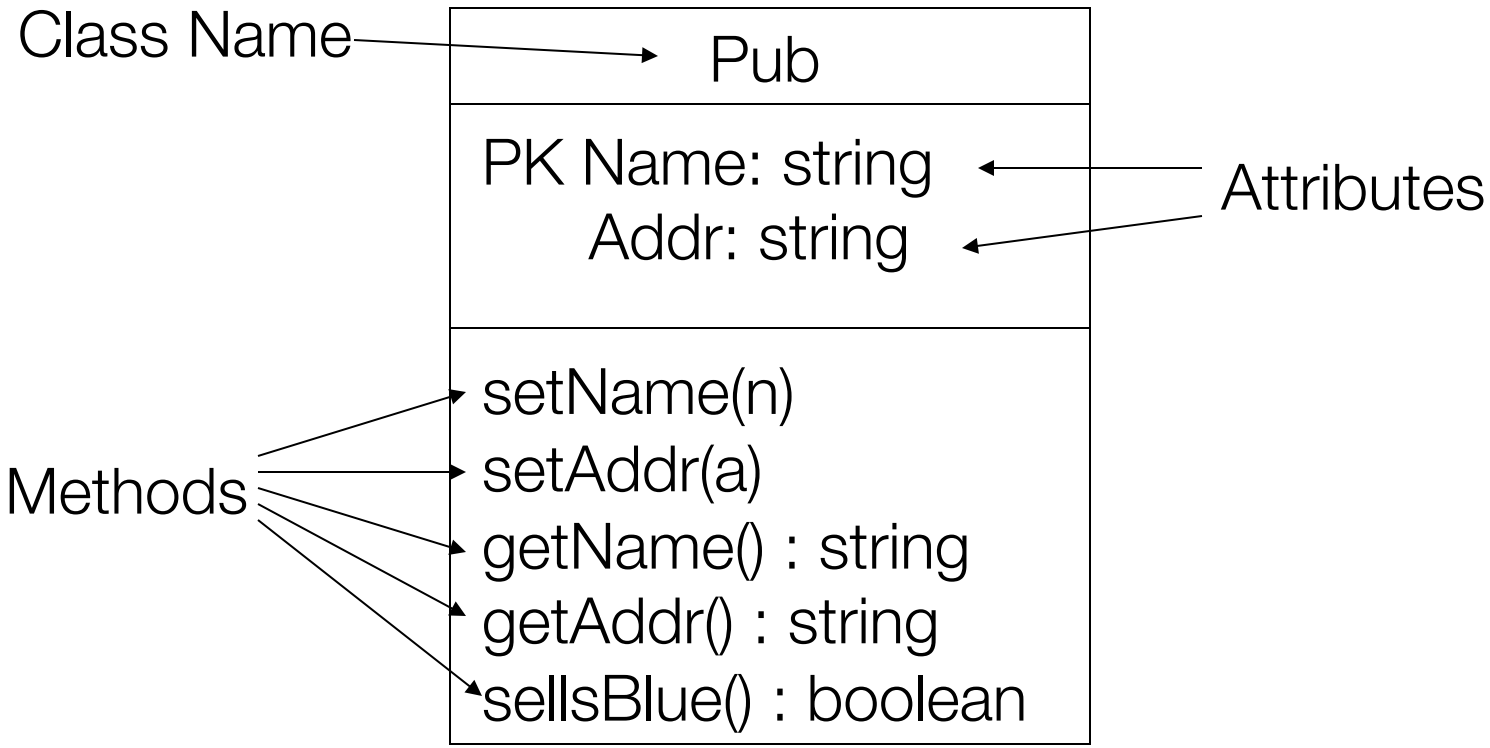
How does UML really fit into all this?

- UML was originally intended for modelling software structure
  - However, it has been adapted as a database modelling notation
- In expressive power, UML lies midway between E/R and ODL
  - No multiway relationships as in E/R
  - Does allow attributes on binary relationships (which ODL does not)
  - Is a graphical notation (unlike ODL)

Classes

- Represent sets of objects
  - Having attributes (i.e., state)
  - Having methods (i.e., behavior)
- Attributes have types
- The notation PK indicates an attributes in the primary key of the object
  - A primary key is optional
- Methods have declarations
  - arguments (if any)
  - return type

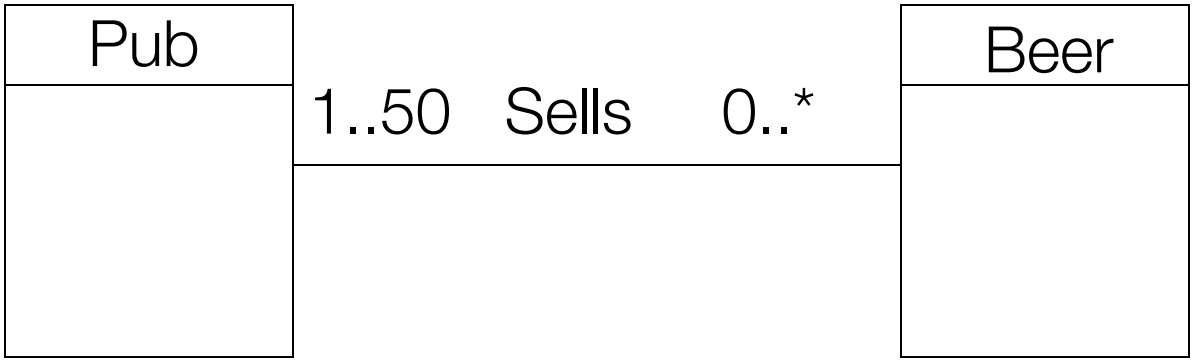
Example: Bar Class



Associations

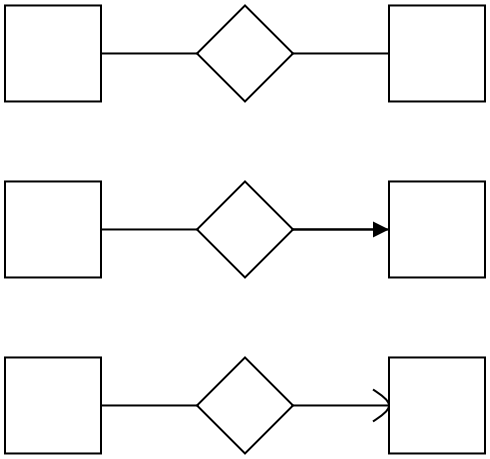
- In UML we already have binary relationships between classes / objects.
- These are represented by named lines
  - There are no diamonds as in E/R
- Multiplicity of the relationship is indicated on either end of the line
  - **m .. n** means from **m** to **n** of these objects **associate with the object on the other end of the line**
  - ***** means **0 to many**
  - **1..*** means **at least one**

Example: Association

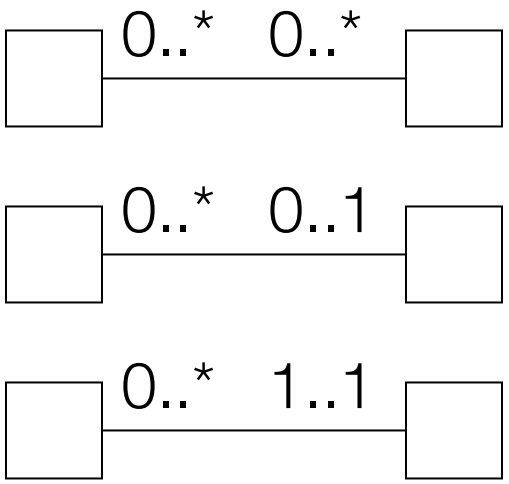


Comparison With E/R Multiplicities

E/R

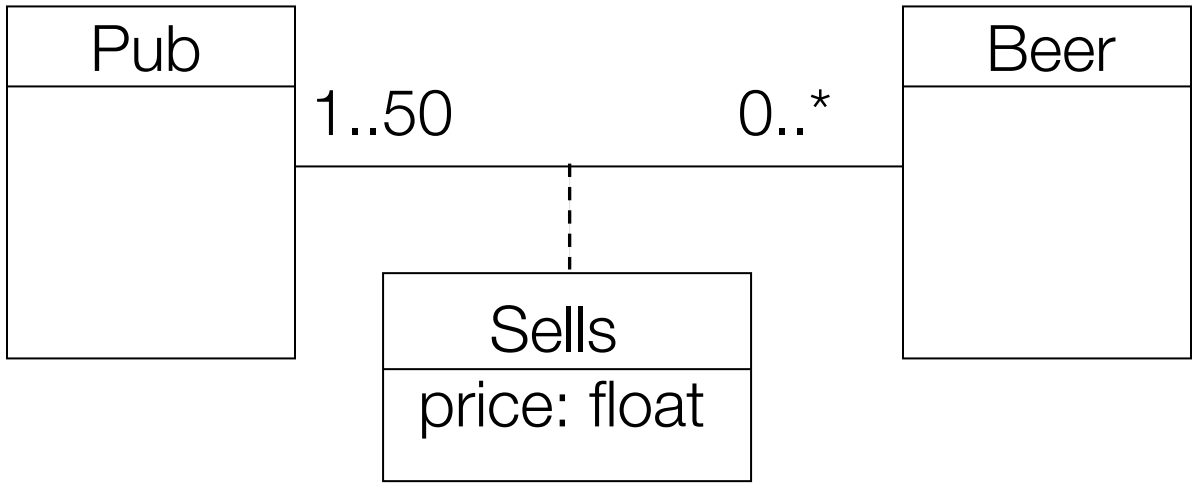


UML



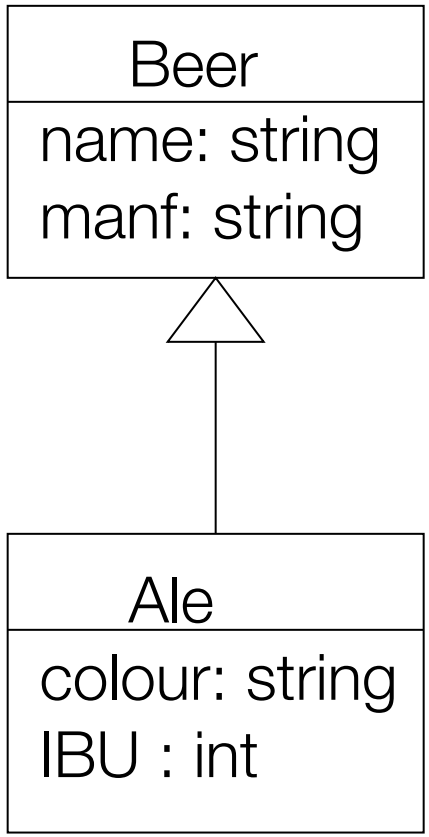
Association Classes

- Attributes on associations are permitted
  - These are **association classes**
  - Similar to attributes on E/R relationships



Subclasses

- Somewhat similar to ISA hiearchies in E/R
  - However, subclass points to superclass with a line ending in a triangle
- Subclasses of a class can be:
  - **Complete** or **partial**
  - **Disjoint** or **overlapping**



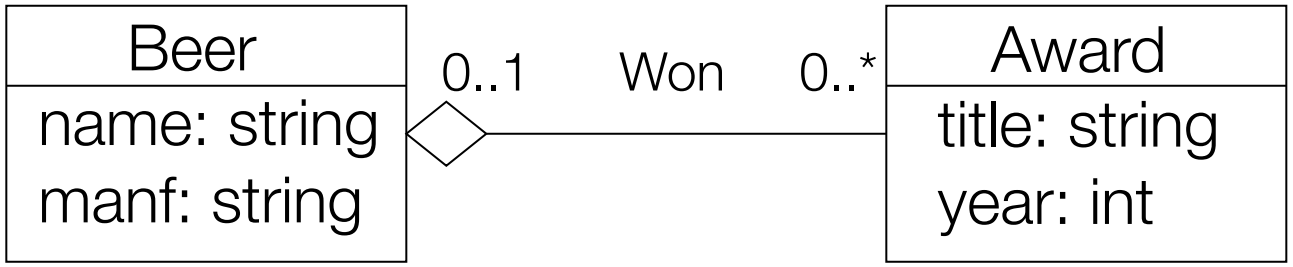
Conversion to relations

- We can use any of the three strategies outlined for E/R for conversion
  - Recall the issue is how we translate an entity and its derived entities as relations

1. E/R style: each subclass' relation stores only its own attribute, plus key.
2. OO-style: relations store attributes of subclass and all superclasses
3. Nulls: One relation, with NULLs as needed

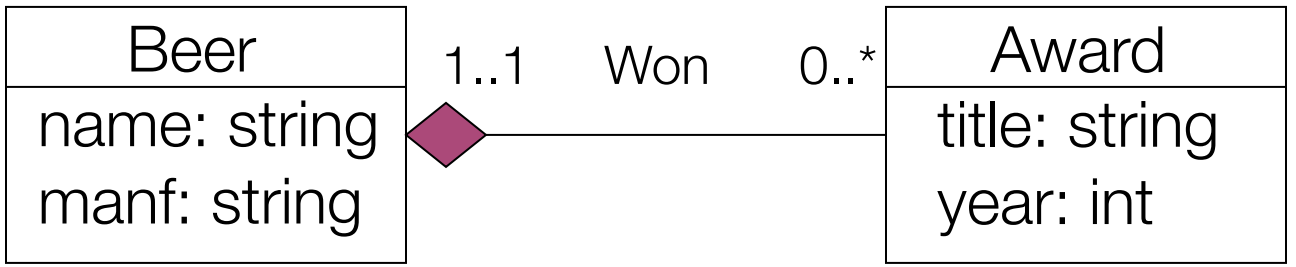
Aggregations

- Relationships in UML have an implication
  - Objects on one side may be owned by (or are part of) objects on the other
- This is represented using the diamond notation
  - Diamond appears at the owner's side of the association.
- Further implication here:
  - In a relational schema, owned objects are part of owner tuples.



Compositions

- Like aggregations, but with a further implication
  - Every object must be owned by some object on the other side
  - In OO terms, also means something about when certain objects come into existence and when they cease to exist
- Solid diamond notation
- Often used for subobjects or structured attributes

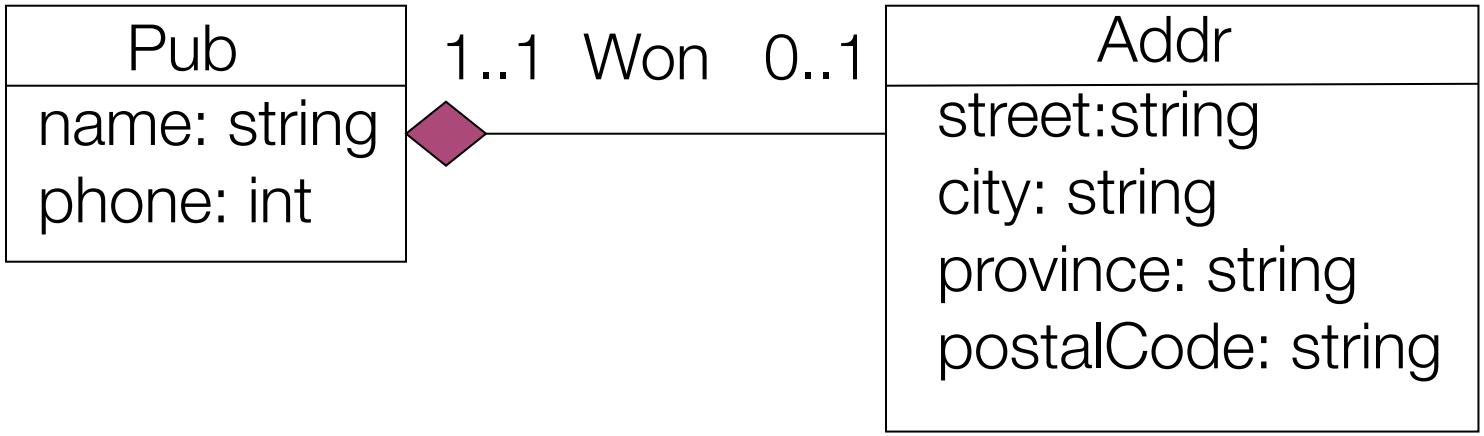


Conversion to relations

- We could store the awards of a beer with the beer tuple itself
- However:
  - This would require an object-relational model (or nested-relation model) for tables.
  - Issue here is no limit exists to the number of awards a beer can win.

Conversion to relations

- Since pub has at most one address...
  - ... it is feasible to add the street, city, province and postal code of Addr to the Pubs relation
- In an object-relational database, Addr can be one attribute of Pubs, with structure



Summary

- E/R model with entities and relationships
- Ternary and higher-order relationships
- Multiplicities
- ISA hierarchies
- Weak entities
- Conversion to relations
- Object Description Language
- Modelling data using UML (and distinctions from ER)

Colophon

- Some slide material is from Stanford CS145 (Jeffrey D. Ullman, Fall 2007)