CS5200
Database Management
Translating to the Relational Model

Ken Baclawski

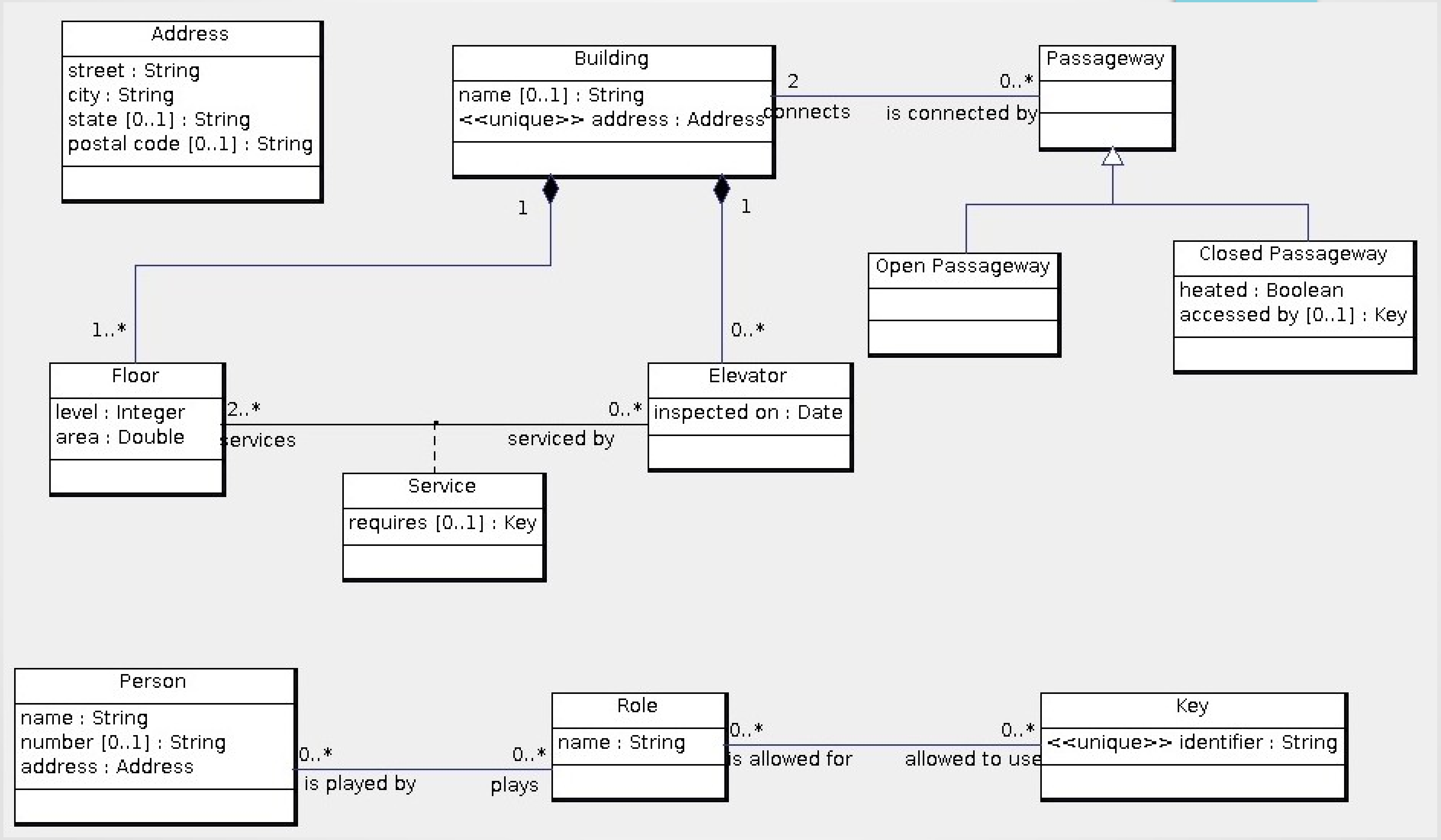
Spring 2017

Outline

- Solution to Assignment #1
- The Relational Model
- Translation from UML to SQL
  - Classes
  - Attributes
  - Associations
  - Subclass Hierarchies
  - Enumerations
- Assignment #2
- Database Design Issues
- The meaning of NULL
- Use Cases

Solution to Assignment #1

Elevator Data Model



The Relational Model

The Relational Model

- Fundamental notion is the *table*
- A table is a set of records (also called rows or tuples)
- A record consists of fields (also called columns or attributes)
- Each field has a name, type and may have constraints
- Every record in a table has the same fields
- A table has a rectangular structure

Relational Table

columns

…

Student Table

unspecified field value

rows

⋮

name	**email**	**phone**	**status**	**gpa**
Fred	fred@neu.edu	555-1212		3.8
Divya	divya@neu.edu	444-3321	enrolled	4.0
…				
Mary	mary@ccs.neu.edu	666-8851	graduated	3.7

Column Types

- Every column must have a type
- The types are always primitive, built-in types
- The types supported by DBMSs vary from vendor to vendor

MySQL Types

- Integral types (tinyint, smallint, mediumint, integer, bigint)
- Floating-point types (real, double, float, decimal, numeric)
- Date and time types (date, time, timestamp, datetime, year)
- String types (char, varchar)
- Binary (bit, binary, varbinary)
- Raw data (tinyblob, blob, mediumblob, longblob)
- Character data (tinytext, text, mediumtext, longtext)
- Enumeration (enum)
- JSON
- Spatial types

Oracle Types

- Numeric types (number, binary_float, binary_double)
- Date and time types (date, timestamp)
- String types (char, nchar, varchar2, nvarchar2)
- Raw data (blob, clob, nclob)
- XML data (xmltype)
- URL (uritype)

SQL Server Types

- Integral types (tinyint, smallint, int, bigint)
- Floating-point types (real, float, decimal, numeric, smallmoney, money)
- Date and time types (date, time, timestamp, datetime, datetime2, smalldatetime, datetimeoffset)
- String types (char, varchar, nchar, nvarchar, text, ntext)
- Binary (bit, binary, varbinary)
- Raw data (image)
- XML (xml)

NULL

- If a field of a record has no value, then it is NULL.
- By default every column allows NULL.
- One can specify that NULL is not allowed.
- A database NULL is not a value!
  - NULL is the absence of a value
- This is discussed in more detail later in this presentation.

Primary Key

- A *primary key* consists of one or more columns
- Every column of a primary key must specify a value
  - NULL is not allowed for any primary key column
  - One can specify NOT NULL for a primary key column, but it is redundant
- No two rows can have the same primary key
- Every table *must* have a primary key
  - Style requirement for this course
  - Required by many database systems
  - Required by many companies
- There can be at most one primary key

SQL Table Syntax

```
create table Student (
  name varchar(200) not null,
  email varchar(200) primary key,
  phone varchar(100),
  status enum ('enrolled', 'graduated'),
  gpa double
)
```

Mathematical meaning of a table

- A relational table is a *set*
  - No two rows are the same
  - Rows have no object identity
  - Modifying a field of a row is logically equivalent to deleting the row and inserting a new one
- Actual database systems relax these requirements
  - Allow two identical rows (*bag* not *set*) if no primary key
  - Each row has a row identifier (rid)
  - However the rid is not normally accessible
    - The rid is an internal identifier
    - The DBMS can change it at any time

Class vs Table

- Instances of a class have attributes
- Attributes can have multiple values (using a set or list structure)
- Instances of a class have object identity
- Attributes of a class can directly reference an object
- A class can be extended to have additional attributes
- A class can be subclassed

- Rows of a table have attributes
- Attributes can have at most one value
- Rows of a table do not have object identity
- Fields can only have primitive, built-in values
- Tables have very limited ability for extension
- No support for inheritance

Translating from UML to SQL

Translation from UML to SQL

- Issues
  - Lack of object identity
  - No multi-valued attributes
  - No support for inheritance
- There is no unique translation from UML to SQL
- Each issue above requires a strategy.
- In the assignment you *must* specify what strategies you are using.

Outline of Translation from UML to SQL

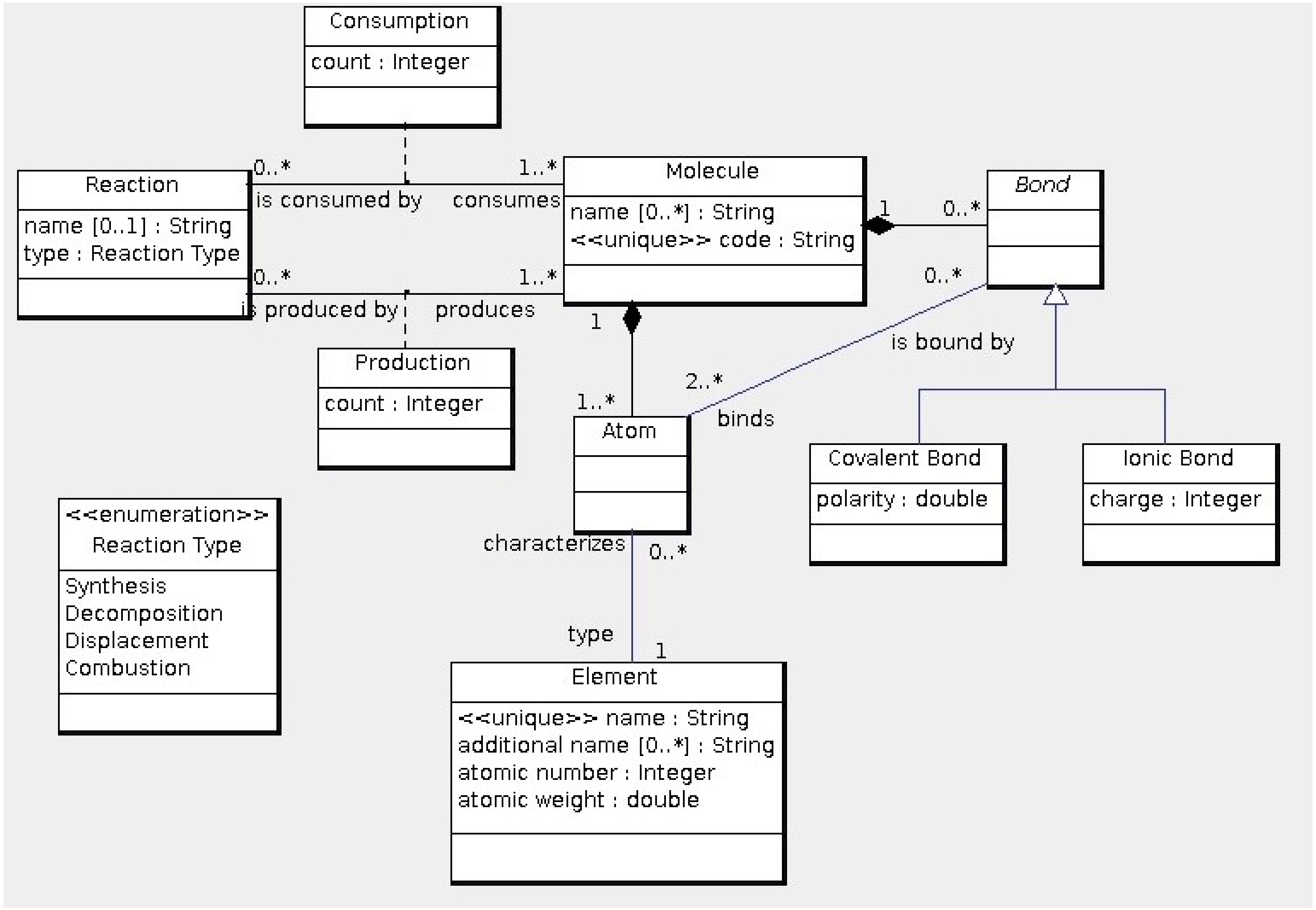
- Classes
- Attributes
- Associations
- Enumerations
- Subclass Hierarchies

Object vs Value

- Object-oriented programming languages support objects
  - Object identity is fundamental
  - Changing a property of an object does not change the object
  - An object need not have any properties at all
  - Only primitive types are values
- Relational databases have no support for objects
  - Every record is a value
  - Changing the value of a field of a record produces a new record
  - A record with no fields is meaningless
  - Object identity must be simulated

Simulating object identity

- Strategy for simulating object identity
  - Add special column to represent the object identity
  - The type of this new column is **int**
  - The special column is the primary key
  - The column will usually be named **id**
- There are other strategies for simulating object identity but this is by far the simplest one
- Remember that this is only a simulation
  - It can be broken by modifying the id column

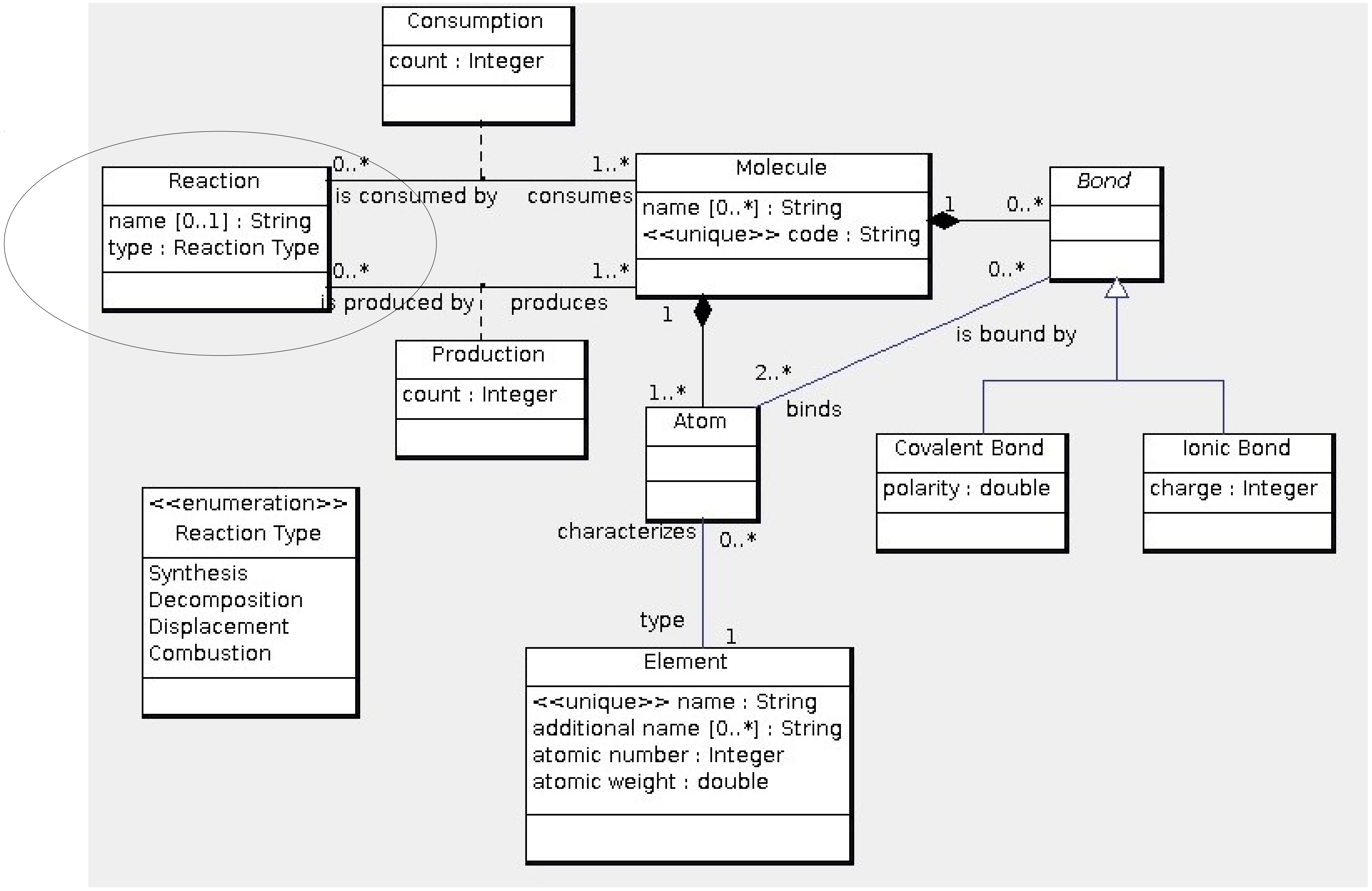


Consumption
count : Integer
Reaction
name [0..1] : String
type : Reaction Type
0..*
is consumed by
1..*
consumes
Molecule
name [0..*] : String
<<unique>> code : String
0..*
is produced by
1..*
produces
Production
count : Integer
Bond
1
0..*
0..*
is bound by
2..*
binds
1
1..*
Atom
Covalent Bond
polarity : double
Ionic Bond
charge : Integer
<<enumeration>>
Reaction Type
Synthesis
Decomposition
Displacement
Combustion
characterizes
0..*
type
1
Element
<<unique>> name : String
additional name [0..*] : String
atomic number : Integer
atomic weight : double

Translating Classes

- Each UML class usually translates to a table
- Inheritance hierarchies are an exception
  - SQL does not support inheritance
  - The translation will depend on the strategy for inheritance

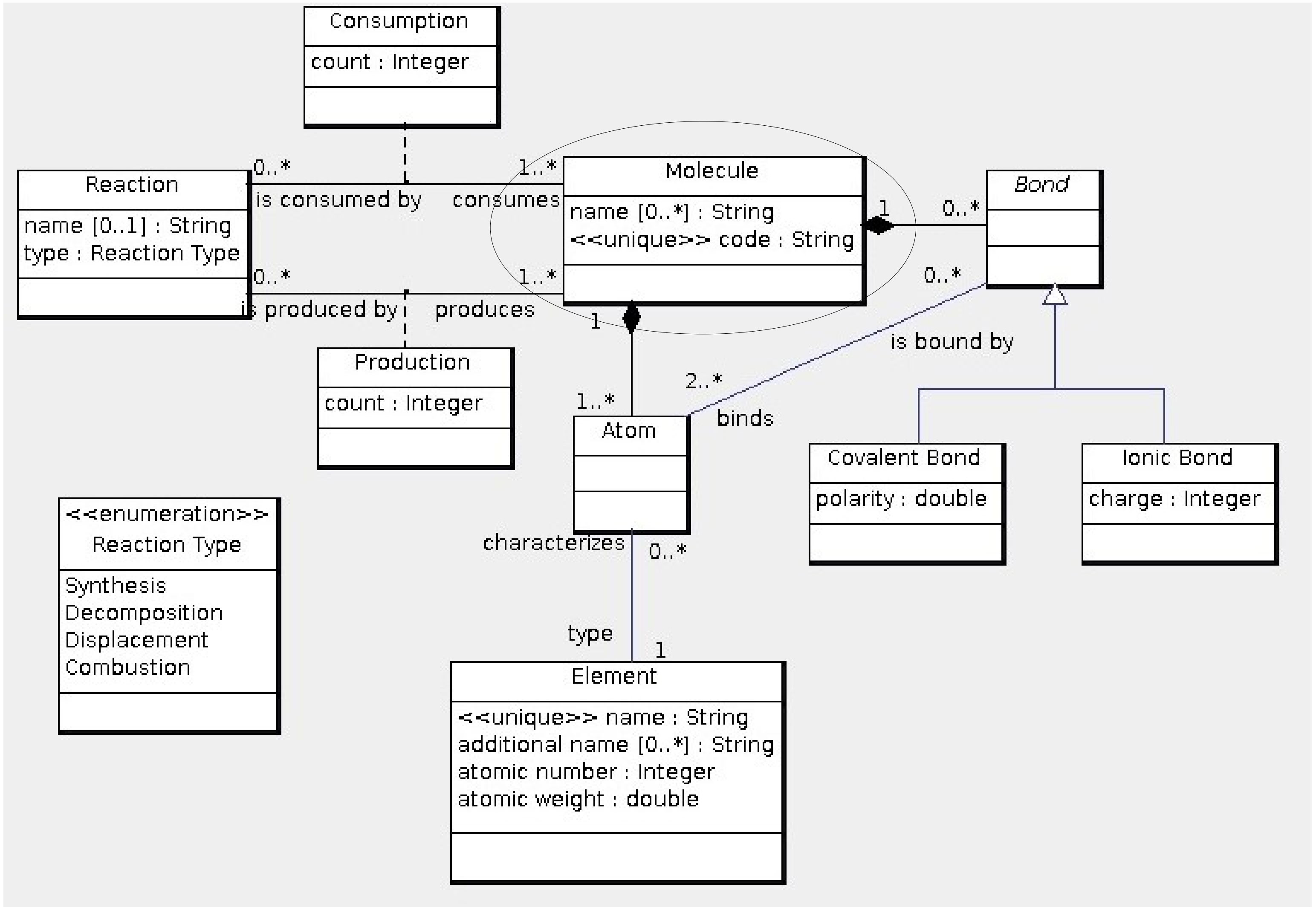
```
create table Reaction (
  id int primary key,
...
);
create table Molecule (
  id int primary key,
...
);
create table Atom (
  id int primary key,
...
);
create table Element (
  id int primary key,
...
);
...
```


Consumption
count : Integer
Reaction
name [0..1] : String
type : Reaction Type
0..*
is consumed by
consumes
1..*
0..*
is produced by
produces
1..*
Molecule
name [0..*] : String
<<unique>> code : String
1
0..*
Bond
0..*
is bound by
2..*
binds
1
1..*
Production
count : Integer
Atom
Covalent Bond
polarity : double
Ionic Bond
charge : Integer
<<enumeration>>
Reaction Type
Synthesis
Decomposition
Displacement
Combustion
characterizes
0..*
type
1
Element
<<unique>> name : String
additional name [0..*] : String
atomic number : Integer
atomic weight : double

Translating Single-Valued Attributes

- Each UML attribute usually translates to an SQL column
- Only single-valued attributes can be translated to SQL columns
- Note differences in the defaults
  - UML default is [`1..1`]
  - SQL default is to allow NULL
- The `200` in `varchar(200)` is the maximum number of bytes allowed in the name field.

```
create table Reaction (
  id int primary key,
  name varchar(200),
  type enum ('Synthesis',
    'Decomposition',
    'Displacement',
    'Combustion') not null
);
```



Consumption
count : Integer
Reaction
name [0..1] : String
type : Reaction Type
0..*
is consumed by
1..*
consumes
Molecule
name [0..*] : String
<<unique>> code : String
0..*
is produced by
1..*
produces
Production
count : Integer
1
0..*
Bond
0..*
is bound by
2..*
binds
1
1..*
Atom
Covalent Bond
polarity : double
Ionic Bond
charge : Integer
<<enumeration>>
Reaction Type
Synthesis
Decomposition
Displacement
Combustion
characterizes
0..*
type
1
Element
<<unique>> name : String
additional name [0..*] : String
atomic number : Integer
atomic weight : double

Unique columns

- Unique means that no two rows can have the same value
- A unique constraint can have more than one column

```
create table Molecule (
  id int primary key,
  code varchar(200) not null,
  unique(code)
);
```

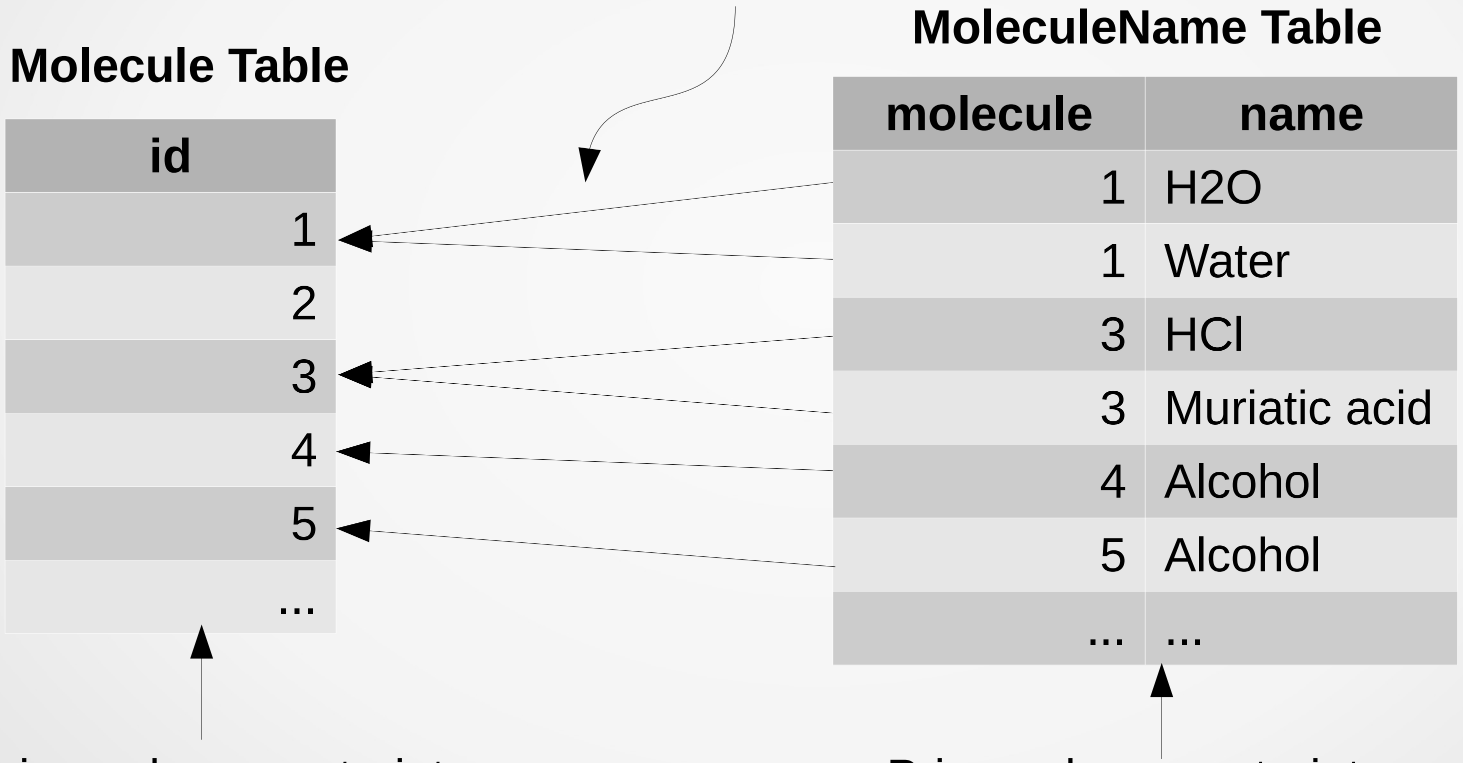
Translating Multi-Valued Attributes

- Each multi-valued attribute requires a separate table
- This separate table does not have object identity
- The primary key has more than one column
- The separate table has a foreign key constraint

```
create table Molecule (
  id int primary key,
  code varchar(200) not null,
  unique(code)
);
create table MoleculeName (
  molecule int not null,
  name varchar(200) not null,
  primary key (molecule, name),
  foreign key(molecule)
    references Molecule(id)
    on update cascade
    on delete cascade
);
```

Primary and Foreign Keys

Foreign key constraint:
Each molecule field value in `MoleculeName` occurs as an id field value in `Molecule`



Molecule Table

id
1
2
3
4
5
...

MoleculeName Table

molecule	name
1	H2O
1	Water
3	HCl
3	Muriatic acid
4	Alcohol
5	Alcohol
...	...

Primary key constraint:
No two rows have same id

Primary key constraint:
No two rows have same id and name

Update and delete behavior

- A foreign key must specify what happens if the constraint is violated
  - The referenced table is the one that is changed (Molecule in this case)
  - The referring table is the one that is affected (MoleculeName)
- Allowed behaviors:
  - `no action` (default)
  - `set null`
  - `set default`
  - `cascade`

```
create table Molecule (
  id int primary key,
  code varchar(200) not null,
  unique(code)
);
create table MoleculeName (
  molecule int not null,
  name varchar(200) not null,
  primary key (molecule, name),
  foreign key(molecule)
    references Molecule(id)
    on update cascade
    on delete cascade
);
```

Effect of a cascading update

Foreign key constraint:
Each molecule field value in `MoleculeName` occurs as an id field value in `Molecule`

Molecule Table

id
1
2
3
4
100
...

MoleculeName Table

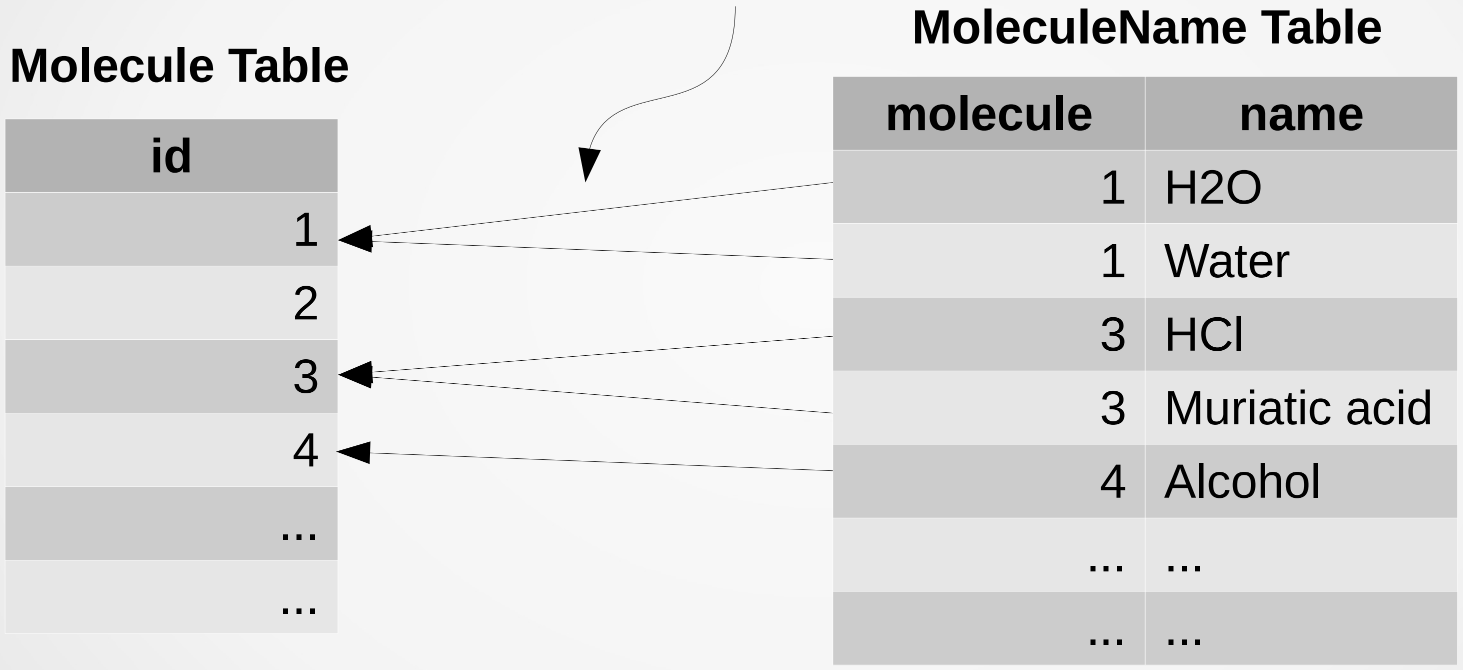
molecule	name
1	H2O
1	Water
3	HCl
3	Muriatic acid
4	Alcohol
100	Alcohol
...	...

User changed 5 to 100

DBMS changed 5 to 100

Effect of a cascading delete

Foreign key constraint:
Each molecule field value in `MoleculeName` occurs as an id field value in `Molecule`



Molecule Table

id
1
2
3
4
...
...

MoleculeName Table

molecule	name
1	H2O
1	Water
3	HCl
3	Muriatic acid
4	Alcohol
...	...
...	...

User deleted record 5

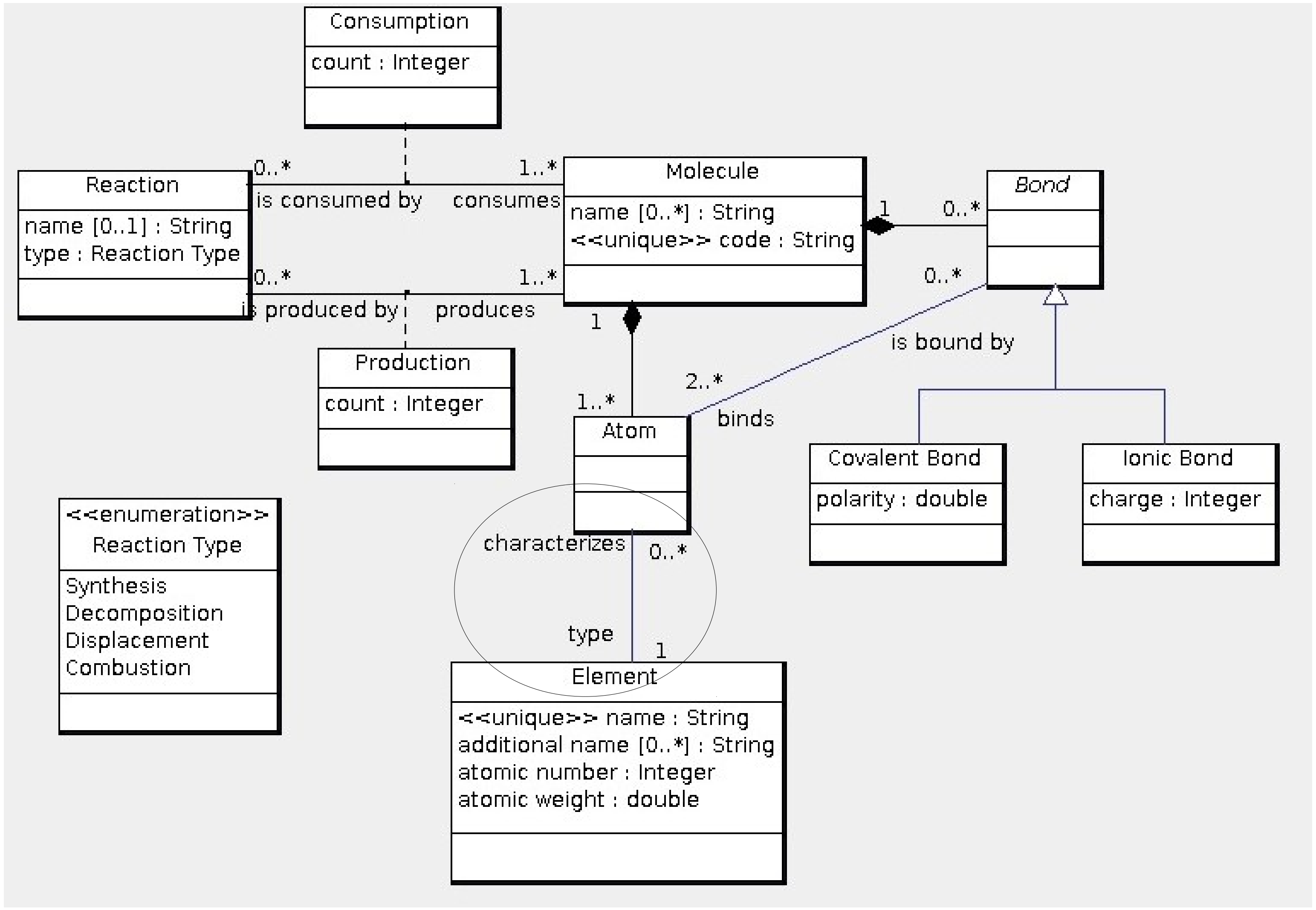
DBMS deleted record [100,'Alcohol']

Other foreign key behavior

- The `no action` behavior
  - If there is a reference to the updated or deleted record then the action is disallowed
  - An exception is thrown and the transaction is rolled back
- The `set null` behavior
  - The foreign key is modified to `NULL`
  - Only allowed if `NULL` is allowed
- The `set default` behavior
  - The foreign key is modified to the default value
  - Only allowed if there is a default value

Translating Associations

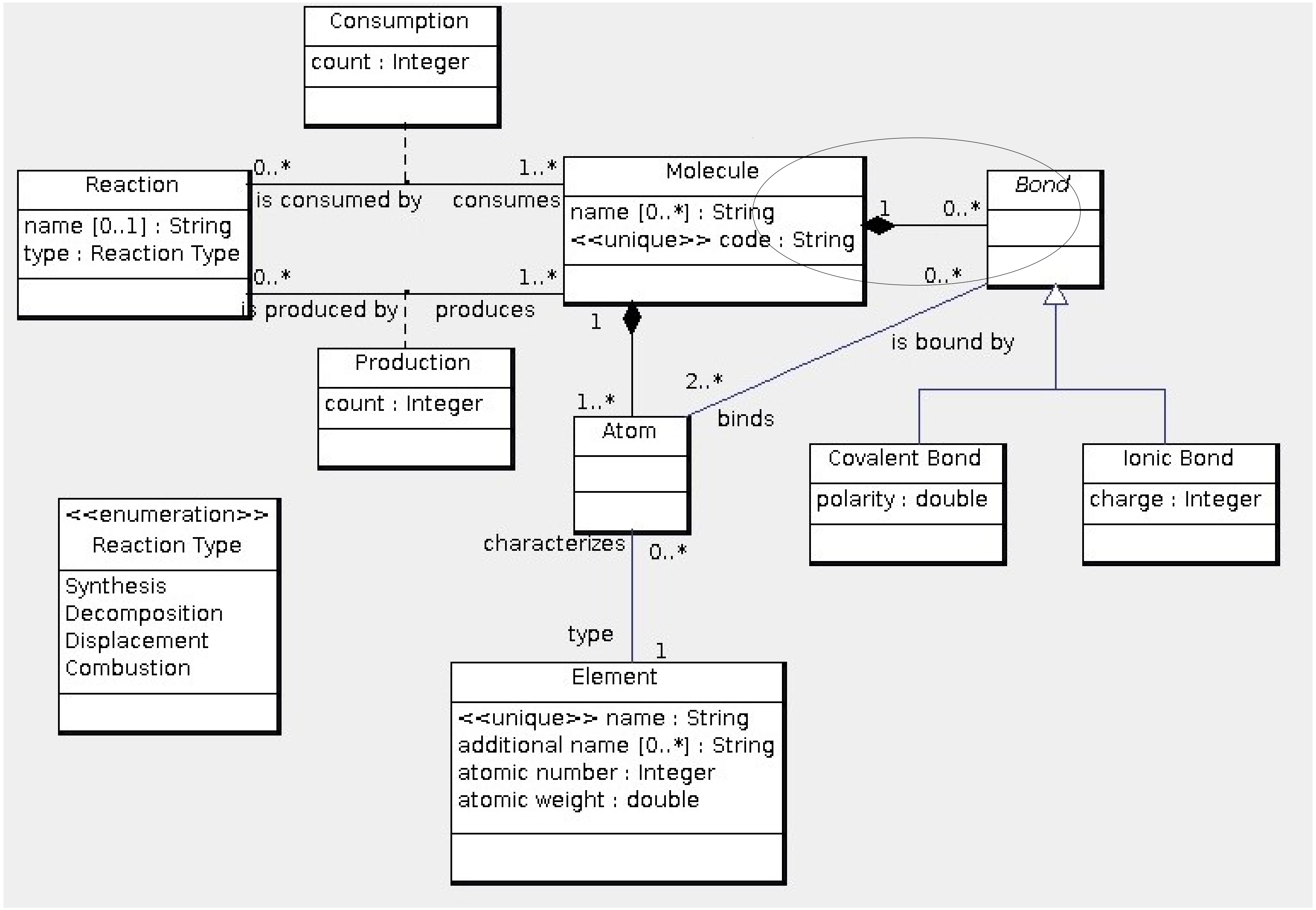
- The translation of an association will depend on the multiplicities
- Many-to-many
  - always requires a separate table
- One-to-many or one-to-one
  - usually a foreign key is sufficient
  - sometimes requires a separate table


Consumption
count : Integer
Reaction
name [0..1] : String
type : Reaction Type
0..*
is consumed by
consumes
1..*
Molecule
name [0..*] : String
<<unique>> code : String
1
0..*
Bond
0..*
is produced by
produces
1..*
1
Production
count : Integer
is bound by
2..*
1..*
Atom
binds
Covalent Bond
polarity : double
Ionic Bond
charge : Integer
<<enumeration>>
Reaction Type
Synthesis
Decomposition
Displacement
Combustion
characterizes
0..*
type
1
Element
<<unique>> name : String
additional name [0..*] : String
atomic number : Integer
atomic weight : double

Translating many-to-one associations

- Many-to-one association may translate to a foreign key column
- If the association is an association class, then this may not be possible
- The on delete behavior must be `no action`
  - Deletion of an `Element` must not affect any `Atom` records
- Compare this with the `MoleculeName` foreign key

```
create table Atom (
  id int primary key,
  type int not null,
  foreign key(type)
    references Element(id)
    on update cascade
    on delete no action
);
```

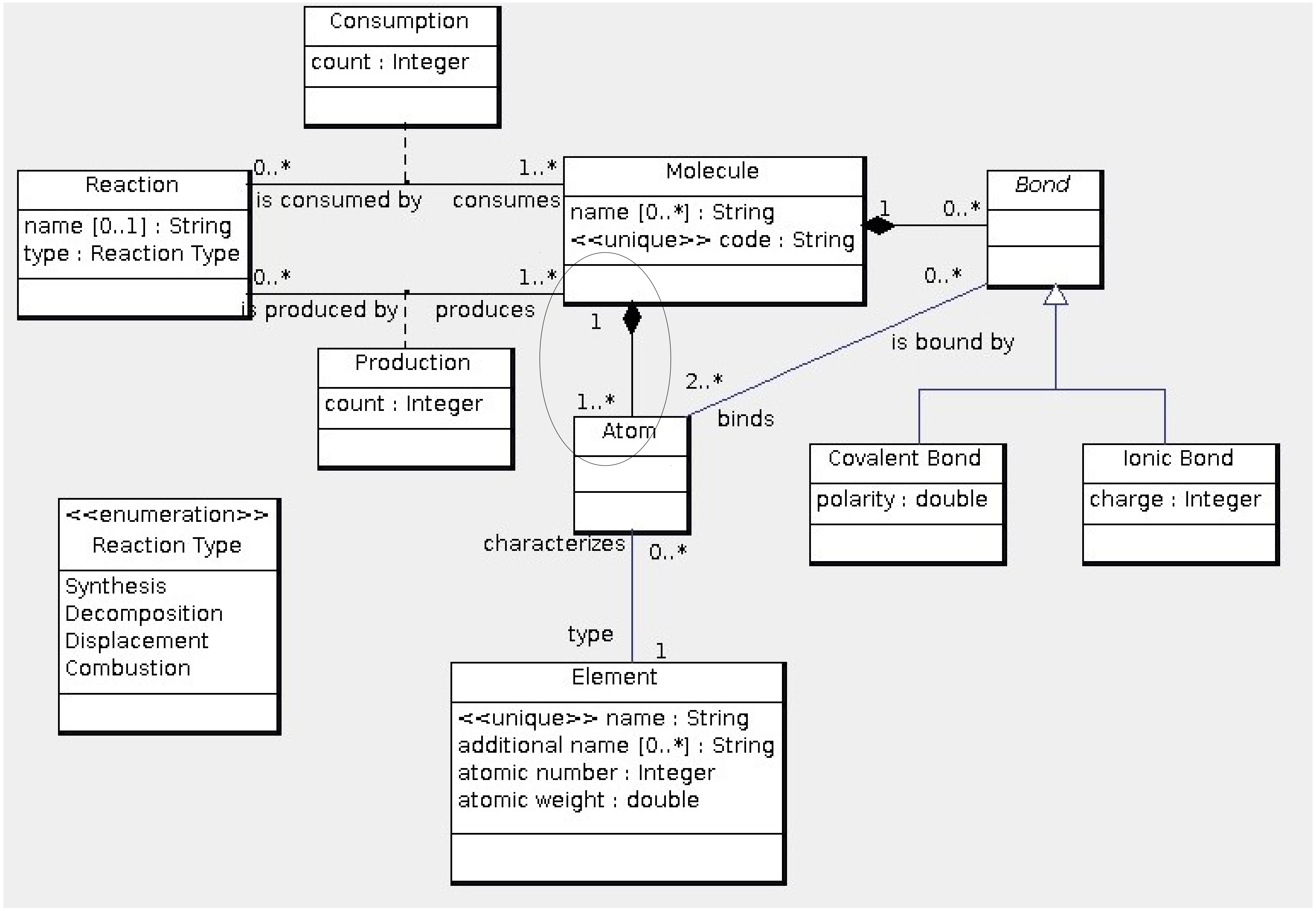


Consumption
count : Integer
Reaction
name [0..1] : String
type : Reaction Type
0..*
is consumed by
1..*
consumes
0..*
is produced by
1..*
produces
Molecule
name [0..*] : String
<<unique>> code : String
1
0..*
Bond
0..*
is bound by
Production
count : Integer
1
1..*
2..*
binds
Atom
Covalent Bond
polarity : double
Ionic Bond
charge : Integer
<<enumeration>>
Reaction Type
Synthesis
Decomposition
Displacement
Combustion
characterizes
0..*
type
1
Element
<<unique>> name : String
additional name [0..*] : String
atomic number : Integer
atomic weight : double

Translating compositions

- A composition translates to a foreign key column
- If the composition is an association class, then this may not be possible, but compositions should not be association classes
- The `on delete` behavior must be `cascade`
  - Deletion of a `Molecule` deletes all of the `Bonds` it contains because a molecule is *composed* of the bonds

```
create table Bond (
  id int primary key,
  partOf int not null,
  foreign key(partOf)
    references Molecule(id)
    on update cascade
    on delete cascade
);
```

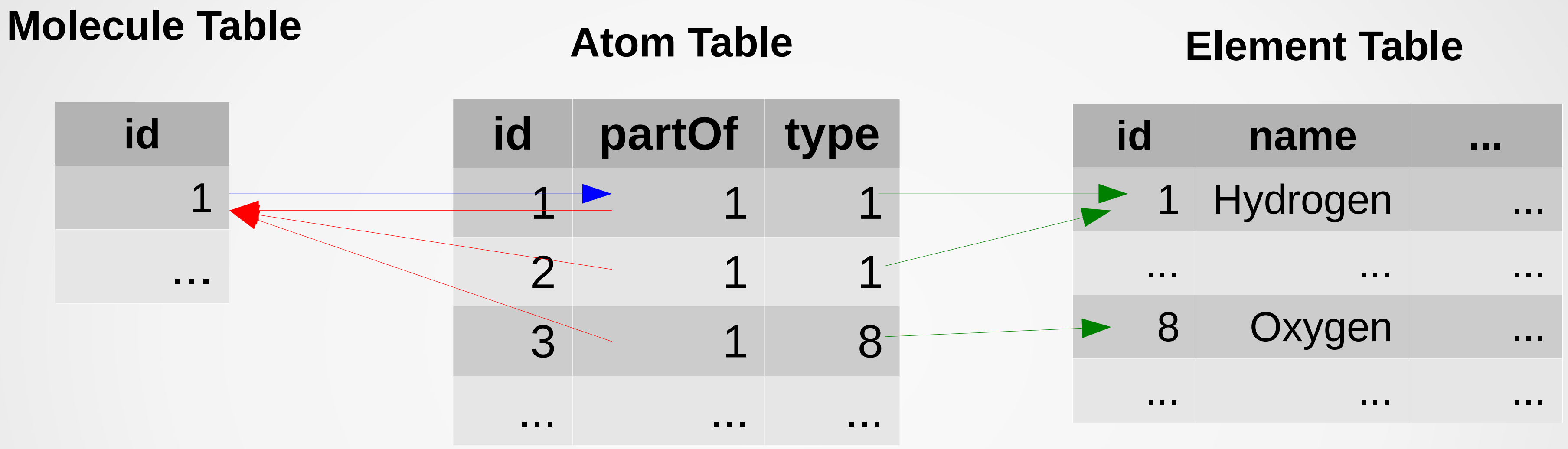

Consumption
count : Integer
Reaction
name [0..1] : String
type : Reaction Type
0..*
is consumed by
consumes
1..*
0..*
is produced by
produces
1..*
Molecule
name [0..*] : String
<<unique>> code : String
1
0..*
Bond
0..*
is bound by
2..*
binds
1
1..*
Atom
Production
count : Integer
Covalent Bond
polarity : double
Ionic Bond
charge : Integer
<<enumeration>>
Reaction Type
Synthesis
Decomposition
Displacement
Combustion
characterizes
0..*
type
1
Element
<<unique>> name : String
additional name [0..*] : String
atomic number : Integer
atomic weight : double

Translating compositions

- The composition between `Atom` and `Molecule` is translated in the same way as the composition between `Bond` and `Molecule`
- However, the other end has multiplicity [`1..*`]
  - Every `Molecule` must contain at least one `Atom`
  - This requires a foreign key constraint in the opposite direction

```
create table Atom (
  id int primary key,
  type int not null,
  foreign key(type)
    references Element(id)
    on update cascade
    on delete no action,
  partOf int not null,
  foreign key(partOf)
    references Molecule(id)
    on update cascade
    on delete cascade
);
```

Enforcing the `1..*` multiplicity



Molecule Table

id
1
...

Atom Table

id	partOf	type
1	1	1
2	1	1
3	1	8
...	...	...

Element Table

id	name	...
1	Hydrogen	...
...	...	...
8	Oxygen	...
...	...	...

```
foreign key(partOf) references Molecule(id)

foreign key(type) references Element(id)

foreign key(id) references Atom(partOf)
```

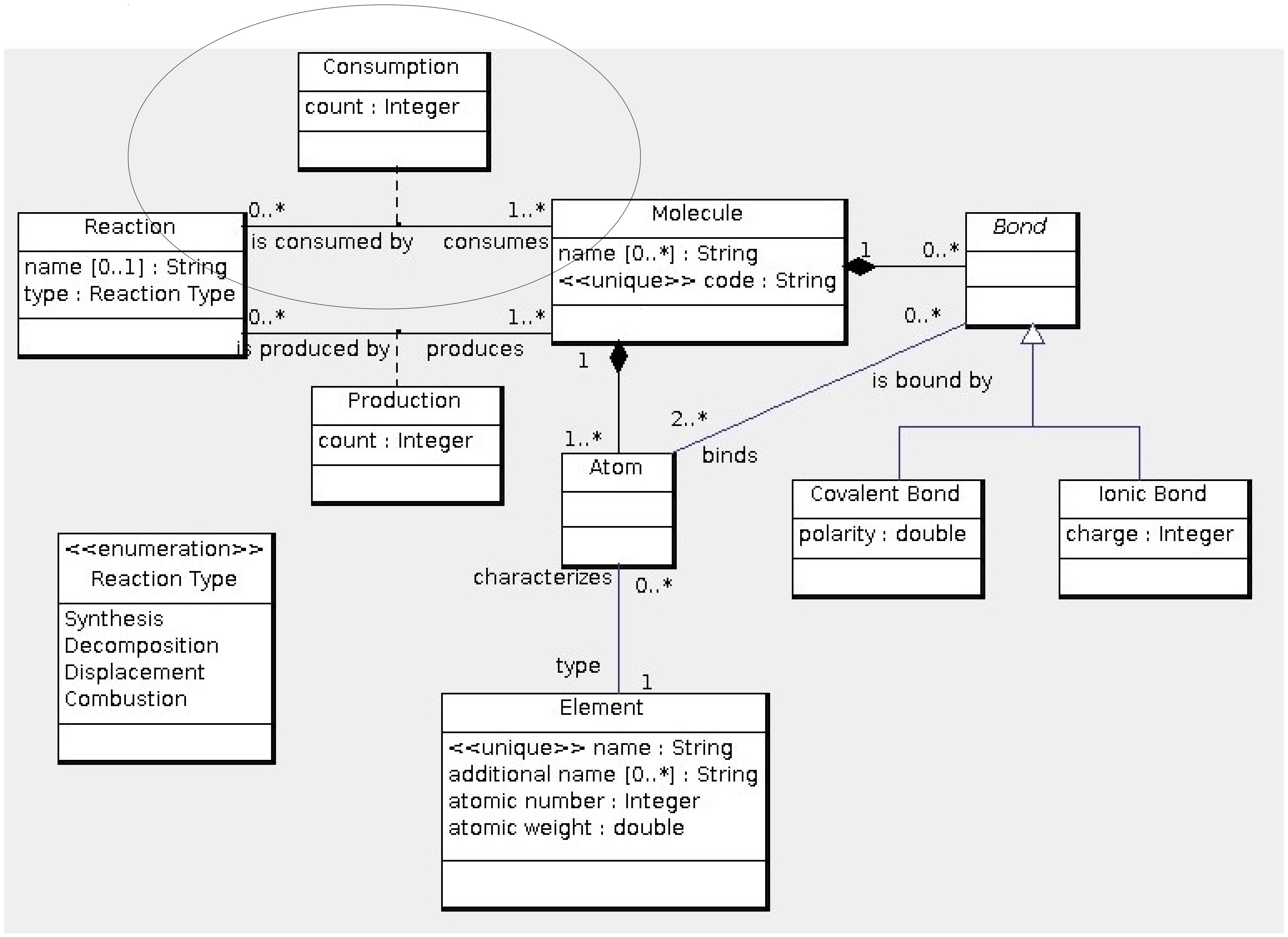
Cyclic Dependencies

- A cycle of dependencies is problematic
  - The foreign key constraints cannot be specified in the tables because the tables do not yet exist
  - One cannot create any of the records until all of them have been created
  - Update or delete could result in an infinite loop in the DBMS
- Solving these problems
  - Add one of the foreign key constraints after the tables have been created
  - This requires a transaction that will be covered later
  - Use `no action` to prevent an infinite loop

```
create table Atom (
  id int primary key,
  type int not null,
  foreign key(type)
    references Element(id)
    on update cascade
    on delete no action,
  partOf int not null,
  foreign key(partOf)
    references Molecule(id)
    on update cascade
    on delete cascade
);
...
alter table Molecule
  add foreign key(id)
    references Atom(partOf)
    on update no action
    on delete no action;
```

Translating aggregations

- An aggregation is the same as an association
- It differs only in the default names and multiplicities
- Translate it like any other association

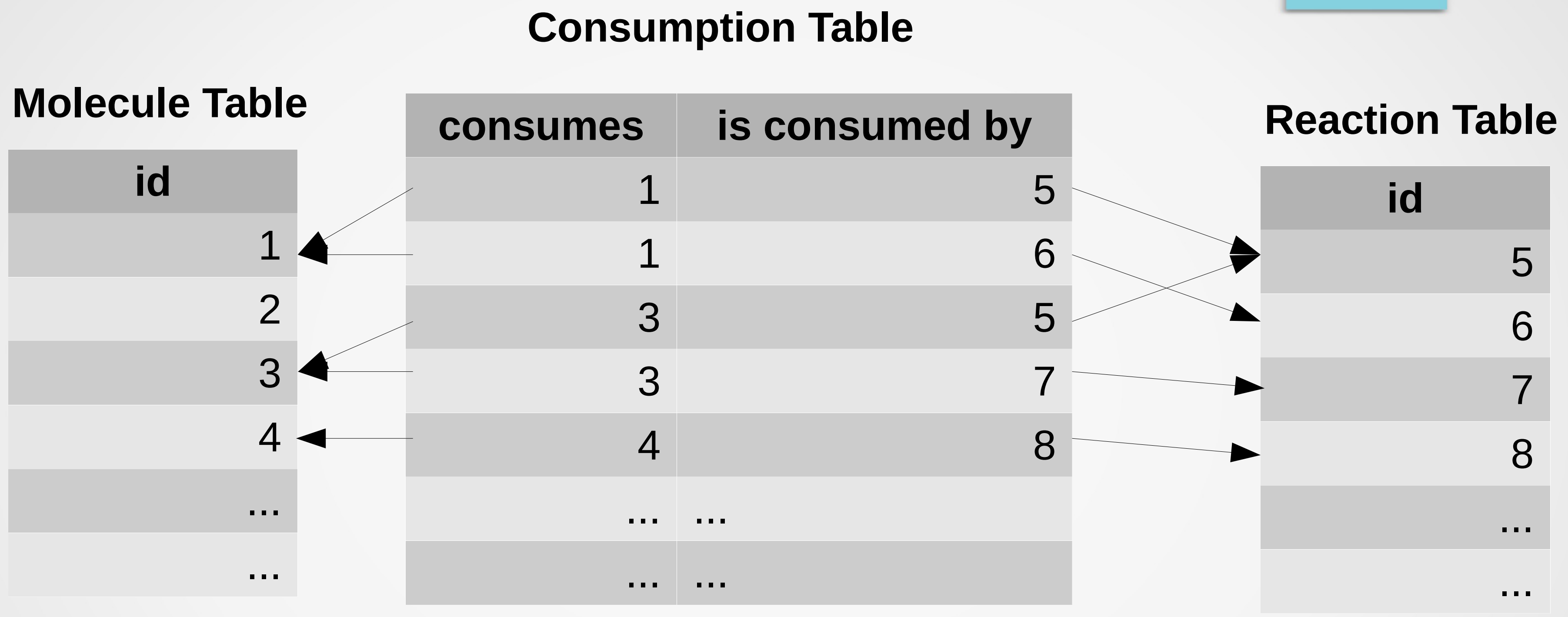

Consumption
count : Integer
Reaction
name [0..1] : String
type : Reaction Type
0..*
is consumed by
consumes
1..*
Molecule
name [0..*] : String
<<unique>> code : String
0..*
is produced by
produces
1..*
Production
count : Integer
1
0..*
Bond
0..*
is bound by
1
1..*
2..*
binds
Atom
Covalent Bond
polarity : double
Ionic Bond
charge : Integer
<<enumeration>>
Reaction Type
Synthesis
Decomposition
Displacement
Combustion
characterizes
0..*
type
1
Element
<<unique>> name : String
additional name [0..*] : String
atomic number : Integer
atomic weight : double

Translating many-to-many associations

- A many-to-many association requires a separate table
- There is one foreign key for each end of the association
- The update and delete behavior is to cascade
- The primary key consists of the two foreign keys
- Additional attributes or associations are added to the table
- No id field because associations do not have object identity

```
create table Consumption (
  consumes int not null,
  foreign key(consumes)
    references Molecule(id)
    on update cascade
    on delete cascade,
  consumedBy int not null,
  foreign key(consumedBy)
    references Reaction(id)
    on update cascade
    on delete cascade,
  primary key(consumes,
    consumedBy),
  count int not null
);
```

Translating many-to-many associations



Consumption Table

consumes	is consumed by
1	5
1	6
3	5
3	7
4	8
...	...
...	...

Molecule Table

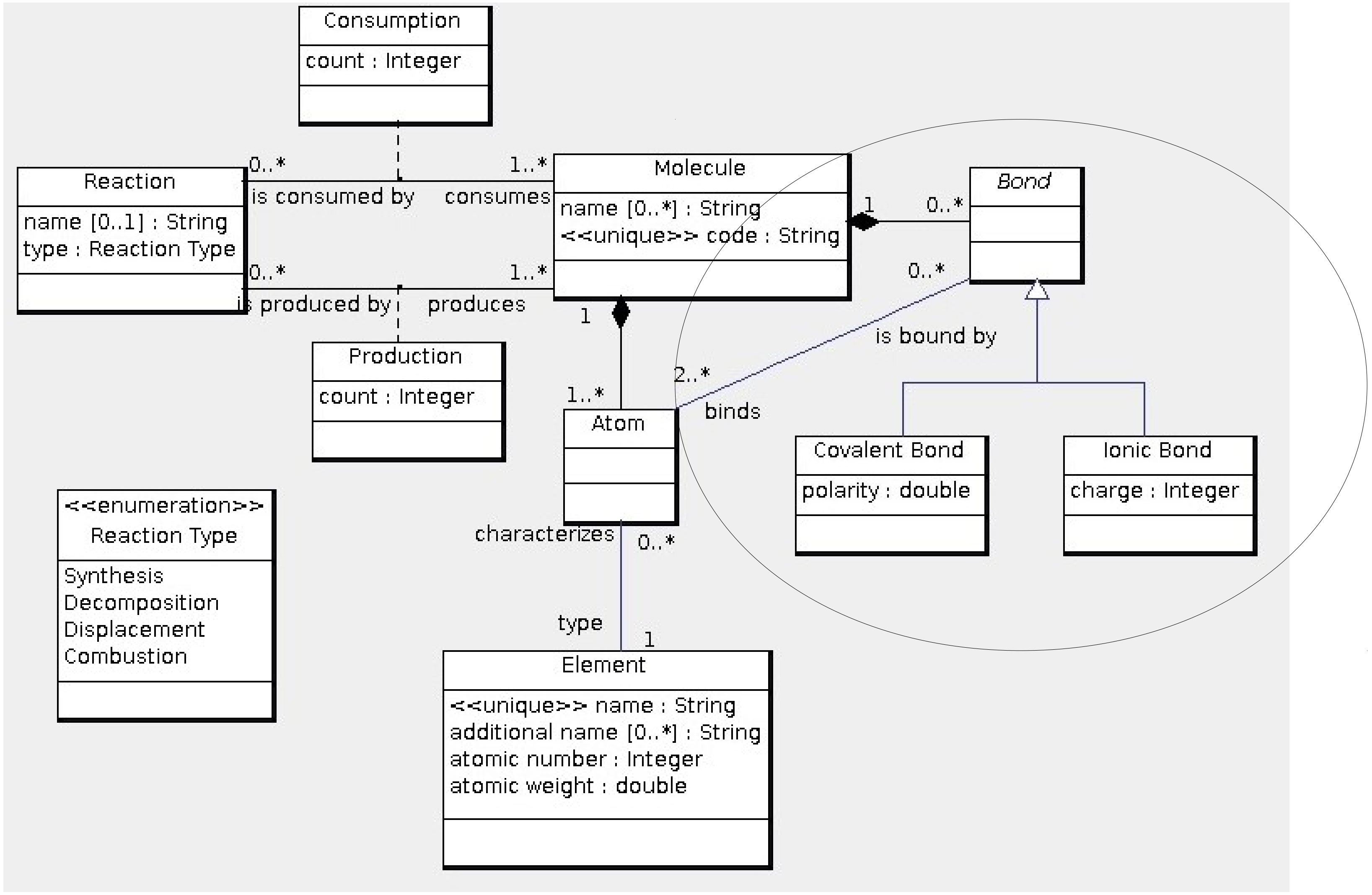
id
1
2
3
4
...
...

Reaction Table

id
5
6
7
8
...
...

Molecule 1 is consumed by reactions 5 and 6
Molecule 3 is consumed by reactions 5 and 7
Molecule 4 is consumed by reaction 8

Reaction 5 consumes molecules 1 and 3
Reaction 6 consumes molecule 1
Reaction 7 consumes molecule 3
Reaction 8 consumes molecule 4


Consumption
count : Integer
Reaction
name [0..1] : String
type : Reaction Type
0..*
is consumed by
1..*
consumes
0..*
is produced by
1..*
produces
Molecule
name [0..*] : String
<<unique>> code : String
1
0..*
Bond
0..*
is bound by
2..*
binds
1
1..*
Production
count : Integer
Atom
Covalent Bond
polarity : double
Ionic Bond
charge : Integer
<<enumeration>>
Reaction Type
Synthesis
Decomposition
Displacement
Combustion
characterizes
0..*
type
1
Element
<<unique>> name : String
additional name [0..*] : String
atomic number : Integer
atomic weight : double

Translating hierarchies

- Main strategies
  - JOINED
  - TABLE_PER_CLASS
  - SINGLE_TABLE
- The simplest is the JOINED strategy.
- A hierarchy can use different strategies in different parts of the hierarchy

JOINED Strategy

- Each class has its own table as usual
- Each subclass specifies that its primary key is a foreign key to its superclass(es)
- Only the attributes local to each class are in each table
- All update and delete behavior is cascaded

```
create table Bond (
  id int primary key,
  partOf int not null,
  foreign key(partOf)
    references Molecule(id)
    on update cascade
    on delete cascade
);
create table CovalentBond (
  id int primary key,
  foreign key(id)
    references Bond(id)
      on update cascade
      on delete cascade,
  polarity double not null
);
create table IonicBond (
  id int primary key,
  foreign key(id)
    references Bond(id)
      on update cascade
      on delete cascade,
  charge int not null
);
```

TABLE_PER_CLASS Strategy

- Each class has its own table as in JOINED
- The id fields are not foreign keys
- Every table has all of its local and inherited attributes

```
create table Bond (
  id int primary key,
  partOf int not null,
  foreign key(partOf)
    references Molecule(id)
    on update cascade
    on delete cascade
);
create table CovalentBond (
  id int primary key,
  partOf int not null,
  foreign key(partOf)
    references Molecule(id)
    on update cascade
    on delete cascade,
  polarity double not null
);
create table IonicBond (
  id int primary key,
  partOf int not null,
  foreign key(partOf)
    references Molecule(id)
    on update cascade
    on delete cascade,
  charge int not null
);
```

SINGLE_TABLE Strategy

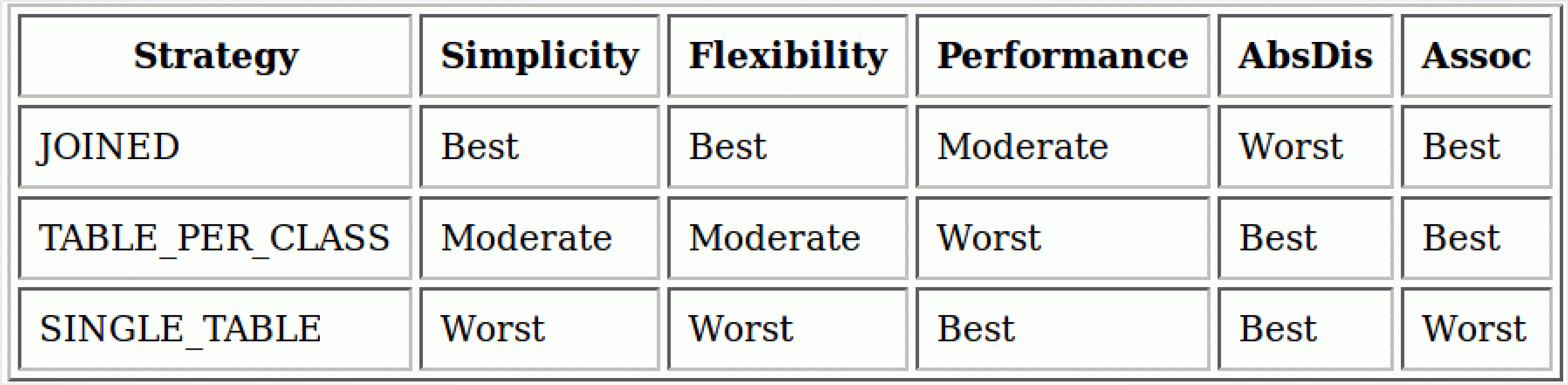
- One table for the entire hierarchy
- An additional discriminator attribute which specifies the class that the record represents
- All attributes of the entire hierarchy are in the table
  - This could result in name clashes
  - It could also result in invalid records such as a covalent bond with a charge
- The constraints that ensure a correct record will be discussed later

```
create table Bond (
  id int primary key,
  partOf int not null,
  foreign key(partOf)
    references Molecule(id)
    on update cascade
    on delete cascade
  polarity double,
  charge int,
  discriminator enum ('Bond',
    'CovalentBond',
    'IonicBond') not null
);
```

Choosing an inheritance strategy

- All of the strategies have advantages and disadvantages
- JOINED is the simplest but queries may not be as fast and some constraints are not feasible
- TABLE_PER_CLASS results in very many large tables if the hierarchy is deep, and queries are much more complicated and slower
- SINGLE_TABLE could have the fastest queries but the constraints are very complicated and may not be feasible

Choosing an inheritance strategy



Strategy	Simplicity	Flexibility	Performance	AbsDis	Assoc
JOINED	Best	Best	Moderate	Worst	Best
TABLE_PER_CLASS	Moderate	Moderate	Worst	Best	Best
SINGLE_TABLE	Worst	Worst	Best	Best	Worst

AbsDis is the ability for the strategy to enforce Abstract class and Disjoint class constraints

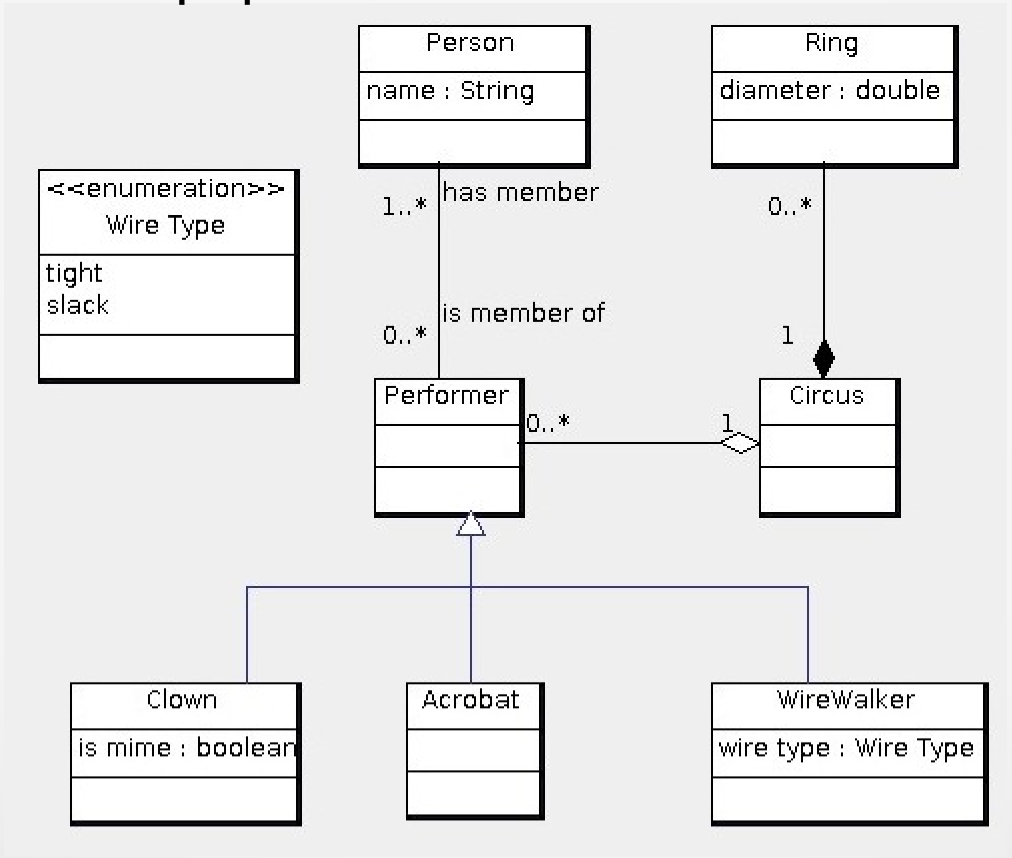
Assoc is the ability to have associations with subclasses in the hierarchy

Enumeration Strategies

- Enumerations can be specified as a MySQL type
- A DBMS without support for enum can synthesize it with a constraint
- However, using an enum is inflexible
  - Very difficult to add or remove an enumerator
  - Cannot add additional attributes to an enumerator
- Other strategies use a separate enumeration table
  - Object enumeration table has a separate id primary key
  - Value enumeration table has the enumerator as the primary key
  - Either of these adds flexibility
  - Object enumeration table is more efficient

Circus Performer Translation Problem

Translate the following UML diagram to an SQL schema. Write your solution on a piece of paper.



Assignment #2

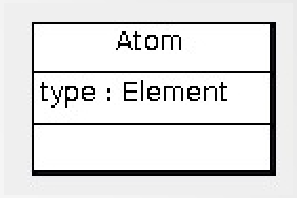
Database Design Issues

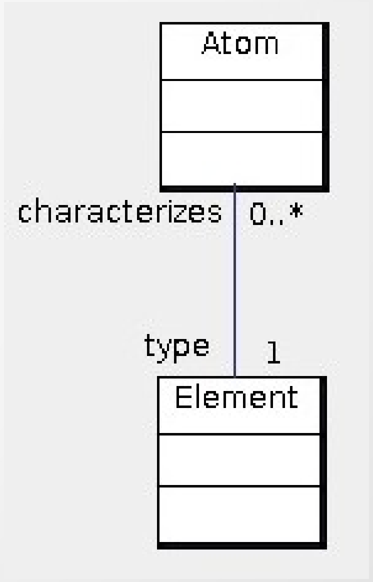
Database Design Issues

- Attribute vs Association
- Attribute vs Class
- Reification

Attribute versus Association

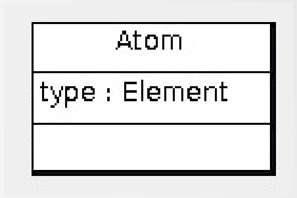
- The relationship between Atom and Element could be either an attribute or association
- Which one is correct?
- Both! They mean the same
- An attribute is an association whose opposite end has multiplicity is [`0..*`]
- Attributes differ from associations in what they support
  - An attribute cannot be an association class
  - An attribute must have [`0..*`] on one end
  - An attribute cannot be an aggregation or composition
  - An attribute is missing one role name

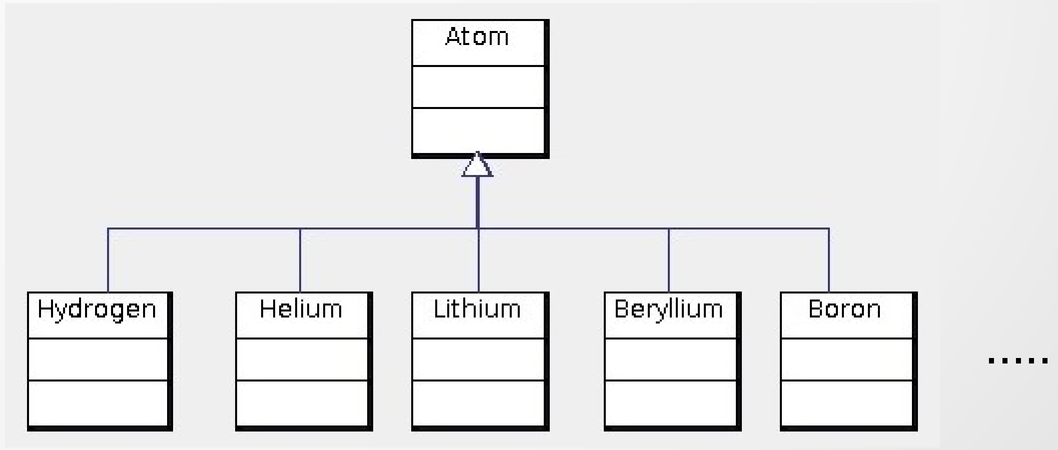




Attribute versus Classes

- Could the type attribute of Atom be represented by a class hierarchy?
- This is appropriate only if the various subclasses have different properties in the data model

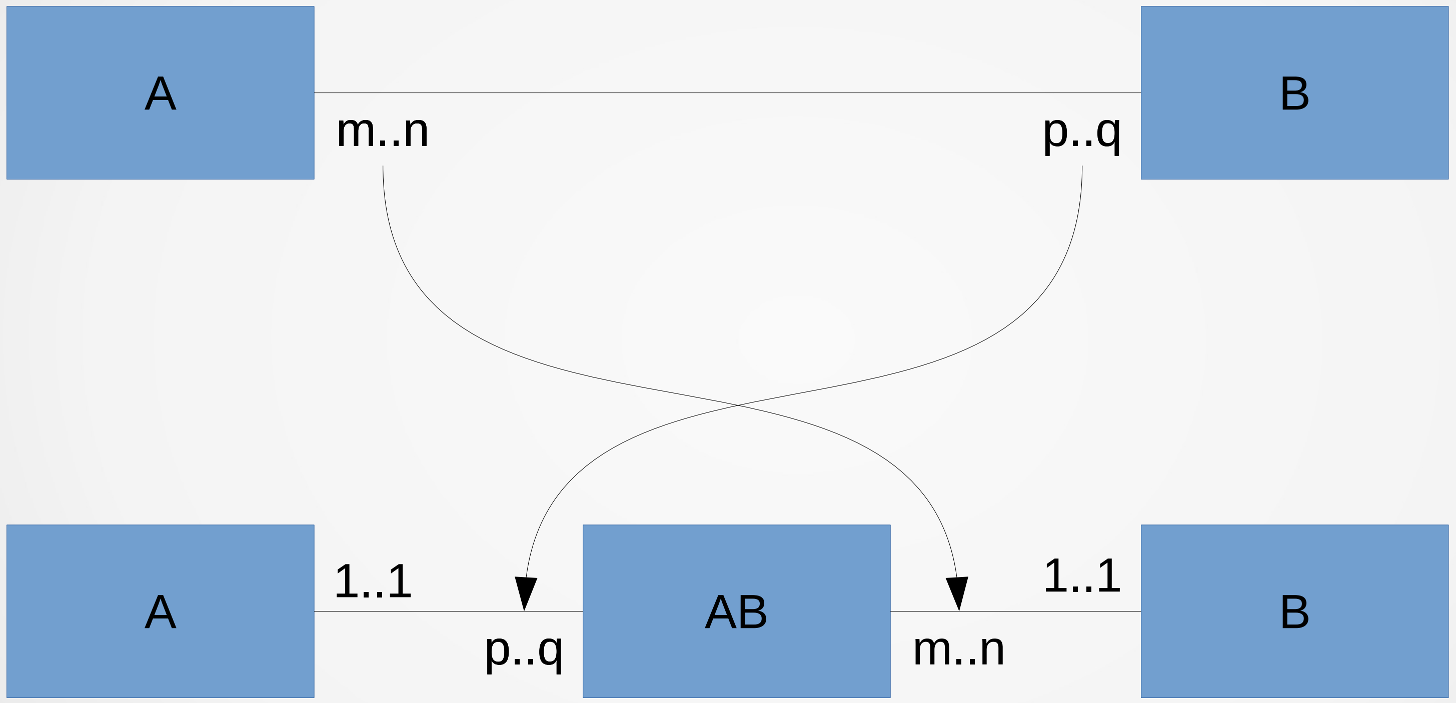




Reification

- This is the design process whereby an entity that is not a “thing” is made into a thing.
- There are many forms of reification, but the most common one is the reification of an association.
- Reification changes the meaning of your design so be certain that it is correct.

Reifying an Association



Reifying an Association

- While the template on the previous slide is useful it is only the “first draft”
- In practice, after reification of an association, one must reconsider the cardinalities.

The Meaning of NULL

Introduction

- What does NULL mean for relational databases?
  - Major source of confusion among database users
  - Nearly all textbooks are wrong
- The main problem: programming languages and databases have different meanings for null
  - For programming languages null is one possible value
  - For databases null is the absence of a value
- More succinctly: PL null **is** a value; DB null **is not** a value

Oxymoron

- For databases, “null value” is self-contradictory (an oxymoron)
- Many textbooks use this phrase repeatedly
  - Sciore [4] has a whole section titled “Null Values”
- Using this phrase leads to major confusions.
- Textbook authors do not help when they use phrases like “strange behavior” and “weird aspect” when explaining null.
- The next few slides examine whether the database null merits such descriptions.

Strange Behavior?

- "Nulls behave strangely in queries." Sciore [4]
- In fact, the strangeness is entirely due to mistakenly believing that database nulls are values.
- When one properly defines database null as the absence of a value, then the strangeness disappears.

Weird Aspect?

- "A weird aspect of nulls is that they are neither equal nor unequal to each other." Sciore [4]
- The weirdness is due to a misunderstanding, not anything intrinsic.
- Since database null is not a value, it is obvious that one cannot compare it with itself.
- Only values can be compared with one another.

Huh?

- "A *null value* denotes a value that does not exist or is unknown." Sciore [4]
- How can something be a value (and therefore exist) yet also not exist?
- Being unknown is very different from not existing, yet Sciore (and many other textbook authors) claim that these are both represented by the database null.

History

- The notion of DB null was introduced by Codd [1]
- DB null was controversial from the start
  - It was not fully specified
  - The specification was inconsistent
- Date and Darwin [3] advocated using PL null
- Codd [2] attempted to fix it by introducing two kinds of null.
  - Too complex
  - Does not fix the problem
  - Never implemented

Possible Meanings of Null

- Potentially infinitely many possible meanings of null
- The next few slides show some of the most common examples
- These examples show how it is hopeless to have a single coherent meaning for null

Possible Meanings of Null

- The field has a value but it isn't known.
  - The value does exist somewhere, but the database does not have it.
    - For example, the taxpayer id of a customer.
  - The value has not yet been determined, but it presumably will be determined at some point.
    - For example, a customer has yet to make a decision about a feature.
  - The value does exist in principle, and it might someday be determined, but there is no guarantee that it will ever be determined.
    - For example, a field might have value 1 if P=NP and 0 if P≠NP.

Possible Meanings of Null

- The field has no value because it isn't applicable. Unlike the meanings above, there is no possibility that such a field could ever have a value.
  - The field is not allowed to have a value.
    - For example, the root node of a tree has no parent node. The fact that the field is null in this case is not due to missing information.
  - The record was produced during an outer join operation.
- A field value could have been inferred but was not inferred, because of an overriding requirement.
  - This can happen for an outer join. The outer join mandates that the additional (padded) columns of an unmatched record be null even when one could infer values for the columns.

Possible Meanings of Null

- The field has a value but it is not within the domain.
  - For example, on a form requesting an ethnic group null (lack of an answer) has many possible meanings some of which are:
    - The choices do not include the one to which a person most closely identifies (usually interpreted as "none of the above" or "other")
    - The person does not wish to answer the question
    - The person does not know the answer.
    - The person has not yet answered the question.
    - The person has neglected to answer the question.

Possible Meanings of Null

- The field value cannot be determined due to an exception.
  - For example, dividing null by 0 is clearly an exception since division by zero is never defined for any number. However, because of the null propagation rules, the result of dividing null by 0 is null rather than a division by zero exception.

References

1. E.F. Codd "A Relational Model of Data for Large Shared Data Banks". Communications of the ACM 13 (6): 377-387. doi:10.1145/362384.362685. (1970).
2. E.F. Codd, The Relational Model for Database Management Version 2 Addison-Wesley, 1990 (ISBN 0201141922).
3. H. Darwen and C.J. Date, Databases, Types, and The Relational Model: The Third Manifesto, 3rd edition, Addison-Wesley, 2006 (ISBN: 0-321-39942-0).
4. E. Sciore, Database Design and Implementation, Wiley, 2008 (ISBN: 978-0-471-75716-0).
5. UML Standard, http://www.omg.org/spec/UML/.
6. CMIS Version 1.1, http://docs.oasis-open.org/cmis/CMIS/v1.1/CMIS-v1.1.html.

Use Cases

Value of Use Cases

- Communication tool
  - Between stakeholders and developers
  - Between designers and implementers
  - Between implementers and testers
- Compact notation
  - Captures infinitely many scenarios
  - Far more useful than large numbers of “user stories”
- Takes advantage of powerful design patterns
  - Inheritance hierarchies of use cases and actors
  - Reuse of frequently used flows and exception use cases
  - Reduces development effort at an early stage

Definitions

- A *scenario* is a narrative or story of an interaction that a system has with its environment
- A *use case* is a collection of scenarios
  - Abstraction similar to that of a class, which is a collection of objects
- Use cases are a powerful and popular technique for specifying the functional requirements of a system to be developed
- Like many techniques, there are many misunderstandings about use cases

Use Case Specification

- The core of the specification of a use case is a list of actions or event steps that define one kind of interaction between an actor and a system
- An *actor* is a role played by a person or another system
- Use cases are formally specified using the Use Case Description Ontology (UCDO)
- A use case does **not** describe:
  - how the interactions are implemented by the system
  - how actors interact with each other
  - the nonfunctional requirements of the system

Example Use Case

Use Case: Update Existing Document

Exposition: An editor updates an existing document submits the modified document

Precondition: Editor has update privilege on the document to be edited
Postcondition: The modified document is stored in the repository

Step-by-step Description:

1. Include: Query Document
2. [Editor] The editor requests that the document be checked out.
3. [System] The system locks the document in the repository
4. [Repository] The repository confirms the lock on the document
5. [Repository] Exception: Document Lock Failure
6. [System] The system provides a template for submitting the modified document.
7. [Editor] Exception: Cancel Document Update
8. [Editor] The editor fills in the template and submits it to the system.
9. [System] The system transfers the template with the modified document to the repository
10. [Repository] The repository stores the modified document, unlocks it, and sends a confirmation to the system.
11. [System] The system sends a confirmation to the editor.

Discussion of the example use case

- The use case *name* is used for identifying the use case
- The *exposition* documents the purpose of the use case
- The *precondition* is a logical expression that must be true before the use case can be invoked
- The *postcondition* is a logical expression that will be true when the use case completes normally
- The step-by-step description defines the interactions of the use case
  - This is the *flow* of the use case
- A use case can *include* another use case
- A use case can invoke an *exception* use case when necessary
  - The UML link for an exception is «extends»
- A use case can have multiple *alternative* flows (not shown in the example)