# CSCI 4333 Section 1 Design of DB Systems

## 2/28/2024 (self - annotation)

**Transforming UML Class Diagrams to Relational Models**

by K. Yue

**1. Transforming OO Model to the Relational Model**

* Once a *conceptual* OO data model is constructed, it needs to be mapped for implementation in the selected *logical* database model. See [IntroDataModeling.html](https://dcm.uhcl.edu/yue/courses/joinDB/Spring2024/notes/model/IntroDataModeling.html).
* If relational DB is used, the mapping will be from OO (classes, attributes, associations, etc.) to relational schema (relations, attributes, keys, etc.)
* Note that the relational model and the OO model are *very different*, even though diagrams representing the two models look similar (there are only finite number of common suitable shapes).
* Computer-Aided Software Engineering (CASE) tools or DB modeling tools may provide varying degrees of facilities for automatic generating relational schema and corresponding SQL statements.
* However, it is important to understand the mechanism a tool uses to generate the relational schema and make adjustment if needed.

**1.1 Model Transformation**

The problem is: Source Model --> Target Model.

* Source Model: UML
  1. Basic elements:
     1. class
     2. Attribute: can be multi-valued.
     3. Association
  2. Secondary elements:
     1. Object
     2. Multiplicity
     3. Data type
     4. Default value
     5. Constraint
     6. Stereotype for RDB extensions: e.g., candidate keys, primary keys, unique, derived, nullability, etc.
     7. ...
* Target Model: Relational Model
  1. Basic elements:
     1. Relation
     2. Attribute (column/field): should be single-valued (atomic).
  2. Secondary elements:
     1. Row (tuple)
     2. Data type
     3. Nullability
     4. Constraint
     5. Candidate key
     6. Primary key
     7. Foreign key
     8. Index
     9. ...
* There are only two basic elements in the targeted model to consider. Data in the relational model can be stoed only in two ways:
  1. relation:
     1. more flexible,
     2. can hold attributes to store a collection of logically related data
     3. more complex.
  2. attribute:
     1. should be single-valued (if good design, i.e., the first normal form, is to be assured.)
     2. less complex.
     3. Thus, if attributes are sufficient in model transformation, they are preferred.
* Different RDBMS provide different features.
  + 1. Thus, the targeted RDB model is not universal.
    2. It is necessary to define vendor-specific transformation rules.

**2. Transformation Rules**

* We present a set of mapping rules below. It is not meant to be complete or universal.
* As alternative examples of transformation rules:
  + A relative simple one: <http://web.fe.up.pt/~ssn/2010/lbaw/slides/lbaw-uml2rel.eng.pdf> a
  + A more elaborated one based on agile methodology: [http://www.agiledata.org/essays/mappingObjects.html](http://www.agiledata.org/essays/mappingObjects.html#Figure2IncludingShadowInformation).
* Do not mechanically follow these rules. Instead, understand the rationale behind the rules and adapt.
* All OO model details should be implemented in the targeted model in some ways.
  + database level: preferred.
  + middle layer level
  + application level

**2.1 Classes**

**C1**. A class C is mapped or transformed to a relation RC.

1. Relations may later be merged and/or reorganized in design refinement and performance tuning.
2. The relation may use the same name as the class.

**Rationale:**

1. A class is a logical unit for encapsulating related data and a relation has the same property.

**2.2 Attributes**

**Basic:**

**ATT1.** Map all single-valued attributes (with simple data types) of a class C as attributes of RC, the relation for the class C.

A diagram of a computer code

Description automatically generated with medium confidence

Transformed to:

A close-up of a computer screen

Description automatically generated

**ATT2**. For each multi-valued attribute A (Hobbies) of the class C (Member), create a *new* relation RCA (Hobby) containing the attribute A (Hobby) and the primary key, RCId (Mid), of the relation RC (Member) (which implements the class C).

1. (RCId, A): (MemberId, Hobby) is a composite candidate key.
2. RCId is a foreign key referencing RC(RCId)
3. A surrogate key, such as RCA\_Id, may be created to serve as the simple primary key.
4. The name of RCA should be meaningfully selected.

***Example:***

Multi-valued attributes: consider the class Member with the following attributes:

1. Member\_Id: <<PK>>
2. Member\_Screen]Name <<unique>>
3. Hobbies[0..\*]
4. Medals[0..\*]

A yellow card with black text

Description automatically generated

RC: Member(MemberId, ScreenName

Example object:

MemberId: M007  
 ScreenName: Bun  
 Hobbies: eating, sleeping, daydreaming  
 Medals: green, red, blue

**Member:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **MemberId** | **ScreenName** | **~~Hobbies?~~** |  |  |
| M007 | Bun | ~~eating, sleeping, daydreaming~~  ~~(not atomic)~~ |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Member2: limited to 3; not very efficient

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **MemberId** | **ScreenName** | **Hobby1** | **Hobby2** | **Hobby3** |
| M007 | Bun | eating | Sleeping | daydreaming |
|  |  |  |  |  |
|  |  |  |  |  |

**RCA: Hobby(MemberId, Hobby, HobbyId)**

|  |  |  |
| --- | --- | --- |
| **MemberId (FK)** | **Hobby** | **HobbyId** |
| M007 | Eating | 1 |
| M007 | Sleeping | 2 |
| M007 | Daydreaming | 3 |
| M117 | Eating | 4 |

**Medal**

|  |  |  |
| --- | --- | --- |
| **MemberId (FK)** | **Medal** | **MedalId** |
| M007 | Green | 1 |
| M007 | Red | 2 |
| M007 | blue | 3 |
| M117 |  | 4 |

Reasonable relation schema: three relations used

1. Member(MemberId, ScreenName):
   1. CK: [1] MemberId, [2] ScreenName
2. Hobby(HobbyId, MemberId, Hobby):
   1. CK: [1] HobbyId, [2] MemberId, Hobby
   2. FK: [1] MemberId references Member(MemberId)
   3. A surrogate key, HobbyId, is created as the primary key.
3. Medal(MedalId, MemberId, Medal):
   1. CK: [1] MedalId, [2] MemberId, Medal.
   2. FK: [1] MemberId references Member(MemberId)
   3. A surrogate key, MedalId, is created as the primary key.

**Secondary:**

**ATT3.** If the multiplicity of an attribute is specified, to handle the case of 0:

1. If 0 is allowed in the UML model (e.g,. 0..1, 0..\* in the UML class diagram), the attribute is nullable. Add the NULL specifier in the column definition in the RDBMS. (NULL is usually the default)
2. If 0 is not allowed, add the NOT NULL specifier in the column definition.

**ATT4.**The default value of an attribute may be directly implemented in SQL DDL.

**ATT5.** Data type mapping should be handled adequately, effectively and consistently.

1. In later modeling phases, one may use SQL data types of the targeted DBMS in the class diagram.
2. If available, consider using user-defined data types in the targeted DBMS.

***Example:***

UML for toyu.student:

A diagram of a student

Description automatically generated

Relational schema (in HW assignment format):

|  |  |
| --- | --- |
| **Relation** | Student(StuId, fname, lname, major, minor, ach, advisor) |
| Candidate Keys | [1] StuId |
| Foreign Keys | [1] major references department(deptCode), [2] minor references department(deptCode), [3] advisor references faculty(facId) |
| Nullable Attributes | major, minor, advisor, ach |
| Non-nullable Attributes | stuId, fname, lname |
| Notes |  |

Relational schema in SQL (with more implementation details):

CREATE TABLE IF NOT EXISTS Student    (  
    stuId        INT NOT NULL,  
    fname        VARCHAR(30) NOT NULL,  
    lname        VARCHAR(30) NOT NULL,  
    major        CHAR(4) NULL,  
    minor        CHAR(4) NULL,  
   -- ach: accumulated credit hours, including transferred credits.  
    ach          INTEGER(3) NULL DEFAULT 0,  
    advisor      INT NULL,  
    CONSTRAINT Student\_stuId\_pk PRIMARY KEY(stuId),  
   -- an artificial example of a CHECK constraint.  
    CONSTRAINT Student\_ach\_cc CHECK ((ach>=0) AND (ach < 250)),  
    CONSTRAINT Student\_major\_fk FOREIGN KEY (major)  
        REFERENCES Department(deptCode) ON DELETE CASCADE,  
    CONSTRAINT Student\_minor\_fk FOREIGN KEY (minor)  
        REFERENCES Department(deptCode) ON DELETE CASCADE,  
    CONSTRAINT Student\_advisor\_fk FOREIGN KEY (advisor)  
        REFERENCES Faculty(facId)  
);

***Example:***

Multi-valued attributes: consider the class Member with the following attributes:

1. Member\_Id: <<PK>>
2. Member\_Screen]Name <<unique>>
3. Hobbies[0..\*]
4. Medals[0..\*]

A yellow card with black text

Description automatically generated

Reasonable relation schema: three relations used

1. Member(MemberId, ScreenName):
   1. CK: [1] MemberId, [2] ScreenName
2. Hobby(HobbyId, MemberId, Hobby):
   1. CK: [1] HobbyId, [2] MemberId, Hobby
   2. FK: [1] MemberId references Member(MemberId)
   3. A surrogate key, HobbyId, is created as the primary key.
3. Medal(MedalId, MemberId, Medal):
   1. CK: [1] MedalId, [2] MemberId, Medal.
   2. FK: [1] MemberId references Member(MemberId)
   3. A surrogate key, MedalId, is created as the primary key.

All columns in the tables above are not nullable.

**ATT6.** A single-valued attribute of a composite data type (such as set, list, array) can be mapped in various ways.

1. If there is an comparable composite data type in the targeted DBMS, it can be implemented as an attribute of that data type in the relation.
   * The relation will no longer be in the first normal form.
   * Care should be taken in handling the difference in data type mapping.
2. Otherwise, regard the attribute as a multi-valued attribute and apply rule ATT2.

**ATT7.** For a derived attribute A: <<derived>> or \

1. Computed or derived values.
2. Not independent.
3. It can be implemented and stored as an attribute of the relation.
   * Mechanisms, such as triggers or stored procedures, should be used to ensure data consistency. That is the derived column should be consistent with the data that derives its value.
4. It may not be stored as a column directly in any relation.
   * Mechanisms, such as virtual columns, views or stored functions, may be used to provide standard access to the derived attributes.

***Example:***

A class Rectangle has three attributes:

* Length
* Width
* \Area or <<derived>>: derived.

A rectangular yellow rectangle with black text

Description automatically generated

What may the relational schema look like?

One solution:

Rectangle(RectangleId, Length, Width)

with a view Rect define as

select RectangleId, Length, Width, Length \* Width as Area  
from Rectangle;

Alternatively, using*virtual* column: a column that is computed in real-time by an expression and not stored.

CREATE or replace TABLE rectangle (  
  width DOUBLE,  
  height DOUBLE,  
  area DOUBLE AS (width \* height) virtual  
);

**2.3 Keys and Constraints**

**KC1.** If a relation R implements a class C or an association (class) A, and C or A has candidate keys K's, set **all** K's as candidate keys of R.

**KC2.**If a relation R implements a class C or an association (class) A, and C or A has no candidate key, create a surrogate primary key K for R.

1. This is needed as every relation must have at least one candidate key.

**KC3.** A candidate key can be implemented by using the 'unique' and non-null constraint together in SQL.

**KC4.** If many CK but no primary key are specified, select a candidate key as the primary key and set it accordingly in the relation.

***Example:***

for the class Department in toyu:

A yellow sign with black text

Description automatically generated

Relational schema (in HW assignment format):

|  |  |
| --- | --- |
| **3** | Department(deptCode, deptName, schoolCode, numStaff) |
| Candidate Keys | [1] deptCode, [2] deptName |
| Foreign Keys | [1] schoolCode references School(schoolCode) |
| Nullable Attributes | schoolCode, numStaff |
| Non-nullable Attributes | deptCode, deptName |
| Notes |  |

Relational schema in SQL (with more implementation details):

CREATE TABLE IF NOT EXISTS Department (  
    deptCode    CHAR(4) NOT NULL,  
    deptName    VARCHAR(30) NOT NULL*,*  
    schoolCode  CHAR(3) NULL,  
    numStaff  TINYINT NULL,  
    CONSTRAINT Department\_deptCode\_pk PRIMARY KEY (deptCode),  
   -- alternate keys: [1] deptName    
    CONSTRAINT Department\_name\_ck UNIQUE (deptName)*,*  
    CONSTRAINT Department\_schoolCode\_fk FOREIGN KEY (schoolCode)  
        REFERENCES School(schoolCode)  
);

**KC5.** For a stereotype:

1. Some may be directly implemented in SQL DDL, e.g., PK, CK, unique, etc.
2. Otherwise, it is necessary to consider where and how it is implemented.

**2.4 Associations**

**A1**. For a many to one association between C1: faculty (the class with the one multiplicity) and Cm: class, add a column R1\_Id (FacId) into the relation Rm, class (which implements Cm).

1. Assume that R1\_Id (FacId) is the primary key of the relation R1, faculty (which implements C1).
2. R1\_Id (facId) is a foreign key of Rm (class) referencing R1(R1\_Id): Faculty(facId).
3. The name R1\_Id may be renamed.
4. R1\_Id is not null in Rm iff (if and only if) 0 is not allowed (i.e., 1..1) for C1.
5. Any single valued attribute of the association is mapped to a column in Rm.
6. If you have composite or multi-valued attributes of the relationship, you should consider promoting the association to an association class or a regular class in your UML class diagram.

***Example:***

For:

A diagram of a diagram

Description automatically generated

We have the three numbered associations implemented by the three foreign keys below.

|  |  |
| --- | --- |
| **4** | Faculty(facId, fname, lname, *deptCode*, rank) |
| Candidate Keys | [1] facId |
| Foreign Keys | [1] deptCode references Department(deptCode) |
| Nullable Attributes | rank, *deptCode* |
| Non-nullable Attributes | facId, fname, lname |
| Notes |  |
| **6** | Class(classId, *courseId*, semester, year, *facId*, room) |
| Candidate Keys | [1] classId |
| Foreign Keys | [1] courseId references Course(courseId), [2] facId references Faculty(facId) |
| Nullable Attributes | room |
| Non-nullable Attributes | classId, *courseId*, semester, year, *facId* |
| Notes |  |

**A2.** For a many to many association (including association classes) between classes CA and CB, create a*new* relation RAB(RA\_Id, RB\_Id).

1. (RA\_Id, RB\_Id) is a candidate key.
2. RA\_Id references RA(RA\_Id) as a foreign key.
3. RB\_Id references RB(RB\_Id) as a foreign key.

A diagram of a group

Description automatically generated

1. An additional surrogate key, such as RAB\_Id, can be created.

***Example:***

For:

A diagram of a program

Description automatically generated

We have:

|  |  |
| --- | --- |
| **8** | Enroll(stuId, classId, grade, n\_alerts) |
| Candidate Keys | [1] stuId, classId |
| Foreign Keys | [1] stuId references Student(stuId), [2] classId references Class(classId), [3] grade references Grade(grade) |
| Nullable Attributes | grade, n\_alerts |
| Non-nullable Attributes | stuId, classId |

**A3.** For a one to one association between classes CA and CB, there are several options:

1. Treat CA as C1 and CB as Cm and apply A1.
2. Treat CA as Cm and CB as C1 and apply A1.
3. Merge the two relations RA and RB into one. (In this case, you may want to refactor the class diagram.)

**A4.** For any n-ary association (n>2), a new relation is needed.

1. You should consider using binary associations instead.
2. A ternary association can be modeled as a class with three binary associations with the participating classes in the ternary association.

***Example:***

Consider the ternary association between the classes Supplier, Part, and Warehouse with an association attribute quantity.

It can reasonably be replaced by a new class and three binary associations.

A diagram of a supply chain

Description automatically generated

Supply(SupplyId, SupplierId, PartId, WarehouseId, Quantity):

1. CK: [1] SupplyId, [2] SupplierId, PartId, WarehouseId
2. FK: [1] SupplierId references Supplier(SupplierId), [2] PartId references Part(PartId), [3] WarehouseId references Warehouse(WarehouseId).
3. All attributes in Supply is not nullable.

***Example:***

Toyu UML diagram:

A diagram of a computer program

Description automatically generated with medium confidence

toyu schema: [toyu\_schema.pdf](https://dcm.uhcl.edu/yue/courses/joinDB/Spring2024/notes/model/toyu_schema.pdf)

toyu SQL: [createtoyu.sql.txt](https://dcm.uhcl.edu/yue/courses/joinDB/Spring2024/notes/query/createtoyu.sql.txt)

**Introduction to UML**

by K. Yue

**1. Introduction to UML**

* UML: A set of graphical notations for object-oriented modeling.
* Wikipedia: "The Unified Modeling Language (UML) offers a way to visualize a system's architectural blueprints in a diagram."
* A standard maintained by OMG: [OMG's UML page](http://www.uml.org/#UML2.0https://www.uml.org/index.htmhttp://www.uml.org/index.htm).
* Two major versions:
  + Version 1.4.2: international standard released in 2005.
  + Version 2.5.1: released in 2017, added nested classifiers and improved behavior models. Specification: <https://www.omg.org/spec/UML>
* Two main types of diagrams:
  + Structure diagrams: model static structures.
  + Behavior diagrams: model dynamic behaviors.
* Version 2.5 has 15 diagrams: 7 structure diagrams and 8 behavior diagrams.
* Some Resources:
  + [OMG UML Resource](https://www.uml.org/resource-hub.htm)
  + [SPARX UML Tutorial](http://www.sparxsystems.com/uml-tutorial.html).
* We will focus on the *class diagram* only.

Class Diagram of UML 2.2 diagram (from Wikipedia):



**2. Class Diagrams (Emphasis on DB applications)**

**2.1 Introduction:**

* A *static* structure diagram in UML.
* "Describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among the classes." -- from Wikipedia.
* Read "class diagram" from Wikipedia: <http://en.wikipedia.org/wiki/Class_diagram>.
* For a significantly better introduction by IBM: <http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/>.
* Two kinds of tools for drawing UML diagrams:
  + Graphical tools: main purposes are drawing diagrams (e.g., MS Visio, draw io, etc.)
  + Computer-Aided Software Engineering (CASE) tools: for software development with some understanding of the semantics of diagram elements (e.g., MagicDraw, IBM Rational Rhapsody, Visual Paradigm, Astah, etc.)
* We use Astah UML Editor
  + We will use *community version* in classroom demonstration, which is now deprecated.
  + Students can use the more powerful *student version* for free: search "astah student license".
* One may also use UML object diagrams to show objects and their associations of a snapshot of the system.

**2.2 A Simple Conceptual Modeling Process**

1. Examine application requirements to gain a good understanding of the problem.
2. Conduct an analysis to extract concepts that may have data requirements.
3. For each concept, design how should it be modeled? Major options are:
   1. by attributes
   2. by a class
   3. by associations between classes (including special associations, such as composition, aggregation, generalization, etc.)
   4. no need to model (as it does not represent data requirement)

These steps are repeated until the model reaches the necessary fidelity, accuracy, and precision.

***Example:***

**Problem**. A used car dealership application's subsystem: information about cars and their manufacturers.

**Specification description**: A car manufacturer has an unique id and name. A car maker may make many cars. For example, Honda, which may have an manufacturer id of 10001, makes Civic and Accord.

**Analysis and Design**

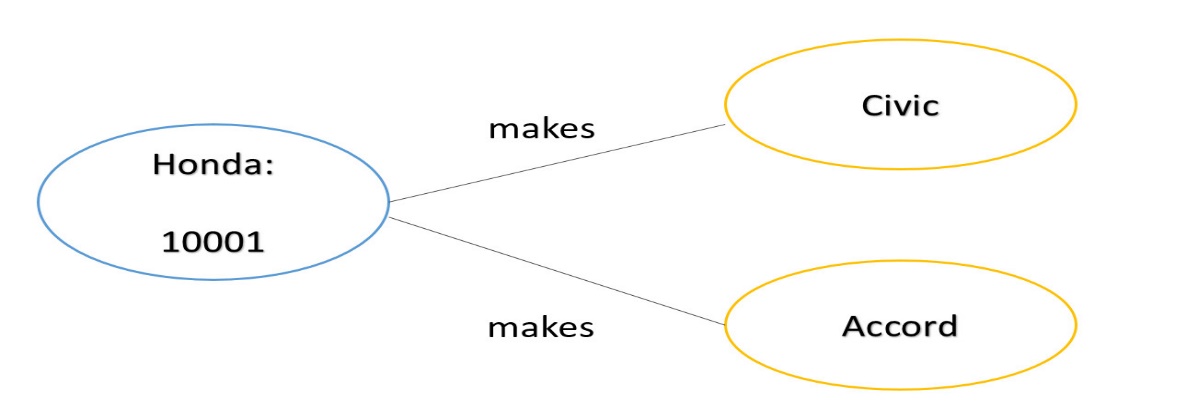
Some observations:

1. Manufacturer: a class (template) that can be used to initiate many manufacturer objects (instances).
2. Honda: an object of the class Manufacturer.
3. Resolve ambiguous terms: e.g., the term "manufacturer" may refer to the manufacturer class, or a particular manufacturer (a manufacturer object such as Honda).
4. Define synonym: manufacturer, car manufacturer, car maker. Different terms can refer to the same concept.
5. "Unique id": may be modeled as an attribute (name), a property of the manufacturer class (modeled as an attribute mid)
   1. Unique: stereotype <<unique>>
6. Make additional assumptions: Every manufacturer object *must* have an unique id.
7. Car:
   1. Definition: Is a product that can be sold by the dealership and bought by customers. E.g. VIN number
      1. Is Civic a car? It is a car model.
   2. A brand.model of a car: e.g., CIVIC
   3. A particular car (VIN: 837937457JD) is a Honda Civic
      1. Class/object: X.
      2. Attribute X: X is a property of Y; X of Y.
      3. Relationship: A (subject) X (verb) B (object)
      4. No la
8. Civic and Accord: modeling choices?
9. 10001: attribute (value) of the id of a manufacturer object.
10. Name: a property of a manufacturer.
11. Additional assumption: Every manufacturer object must have a name.
12. Car: a class, as there may be many *brands* of cars.
13. Prepare questions: Do we need to introduce the concept *model* (e.g., Coupe, Sedan, Si Coupe)?
14. Civic and Accord: object instances of Car.
15. Additional assumption: Every car must have a name as its attribute.
16. Make, or manufacture: a relationship between a manufacturer (object) and a car (object).

**Class Diagram:**



**Object Diagram:**



**2.3 Classes**

1. Drawn as a rectangular box.
2. The class names, attributes, and operations may be specified, with selected details in the name, attribute, and operation *compartments* respectively.
3. Attribute and operation compartments are optional.
4. For DB modeling, the attribute compartment will eventually need to be clearly modeled. The operation compartments may not be needed.
5. The levels of details depend on the phases of modeling. It is a common mistake to specify too much detail in the early modeling phases.
6. As modeling proceeds, more details are added, updated, and refined.

Note that software application modeling and database modeling has different focus.

1. Software modeling: focus on operations (methods, especially public methods).
2. Database modeling: focus on attributes (data).

***Example:*** The following sequence of diagrams of how the modeling of the used car dealership application may proceed.

Initial version: v0.0.1.0:

* Only some major classes, associations, and attributes.

A diagram of a customer

Description automatically generated

Version v0.0.1.1:

* Add a payment class and some attributes.

A diagram of a company structure

Description automatically generated

Version v0.0.1.2:

1. Decided to split the concept 'car' into two concepts 'car model' and 'car'. Adjust associations.
2. Add some type information.

A diagram of a car model

Description automatically generated

Version v0.0.1.3:

1. Add an association between Payment and Car.
2. Add multiplicity of the association "of the model of":
   1. A car must be made of one car model.
   2. There may be many cars made of the same model.
3. Add multiplicity of 1 to the attributes Amount and PayTime of the class Payment. They are mandatory.

A diagram of a model

Description automatically generated

* For example, one may focus on the main classes and their associations in the first model, without worrying about the attribute or operation compartments.
* Most UML editors allow controlling visibility of different elements. For example, in Astah:



* A stereotype (specifying the kind of entities) and a property list with tagged values can be added to any compartment.
* Their flexibility allows for customization and extensibility to fit specific applications.
* Additional *properties*on data members may be specified, such as:
  1. Visibility: + (public: +, protected: #, private: -, etc.)
  2. data types
  3. abstract (in italic) or concrete (as constraints)
  4. class members (underscored) or instance members
  5. default values

**Example:** for software modeling:

A screenshot of a computer program

Description automatically generated

***Example:*** for database modeling.



The class Patron may be identified in the first draft of the UML class diagram.

1. In a subsequent iteration, attributes may be added using settings of the UML tool showing visibility of the attribute members.
2. Data types may be included using predefined data types provided by the tool.
3. In a further iteration, stereotype may be added, such as to identify the primary key <<PK>> and simple candidate key <<unique>>.
4. More specific user-defined types (or implementation types) may be used.
5. Operation members may be added. They are in general less important than data members in data modeling.
6. Multiplicity should eventually be added, as shown in the diagram below.



* Note that multiplicity can be used to depict *nullability* and *multi-valued* attributes. In this example, PatronId is not nullable ([1]), Phone is nullable ([0..1]) and Hobbies can have multiple values ([0..\*]).

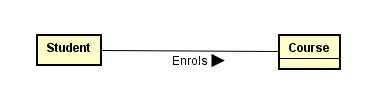
Check out the introductions to class diagrams from [agile modeling](http://www.agilemodeling.com/artifacts/classDiagram.htm) and [wikipedia](http://en.wikipedia.org/wiki/Class_diagram).

* Some *possible* relational database *extensions* on attributes may include:
  1. Multi-valued: \* or by using multiplicity.
  2. Multiplicity can also be used to indicate whether an attribute is nullable.
  3. Derived: <<derived>> using stereotype, \, or using other specific notation
  4. Primary key: <<PK>> as stereotype.
  5. Candidate keys: <<CK>> as stereotype.
  6. Unique field: <<unique>> as stereotype.
  7. Nullability: <<nullable>> or by using multiplicity.
  8. User-defined or system defined SQL data types.
  9. Indexing: <<index>> as stereotype.
* Check with your organizations for UML guidelines on a specific project.
* An example of a database profile for UML: <http://www.agiledata.org/essays/umlDataModelingProfile.html>
  1. may be adapted for uses in later phase of modeling.

**More Properties of Classes**

* A class is a 'first-class citizen.'
  + It has attributes.
  + It can form associations with other classes.
  + Objects of a class can exist by themselves.
  + It has more structures for modeling data requirements.
* As a comparison, an attribute is not a first-class citizen.
  + It does not have sub-attributes.
  + It cannot form associations with other elements.
  + The existence of an attribute depends on the object..
* Objects can be instantiated from classes.

***Example:***



We may have four objects of the student class: S1, S2, S3 and S4. Each student object represents an individual student in a database application.

We may have three objects of the course class: C1, C2, and C3. Each course object represents an individual course in the database application.

**2.4 Associations**

* Binary associations are represented by solid lines.
* Important options include:
  1. Association names with directional arrows (for reading).
  2. Association roles: the role of an object participating in an association.
  3. Multiplicities: the allowed number of associated objects.
  4. Association attributes can usually stored by promoting an association to an *association classes*.
  5. Qualifiers: association attributes to partition the targeted classes.
  6. Navigational requirement: specified by arrows. Usually not used in data modeling.
  7. Dependency constraints: by dotted lines.
* Some modeling questions and decisions:
  1. Should we model something as a class or as an association?
  2. Should we model something as a class or as an attribute?
  3. What kind of association should I use? Binary association, association class, n-ary association?

***Example:*** Note that no attribute is shown in this initial phase.

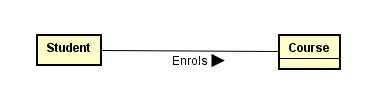
A diagram of a company

Description automatically generated

Note:

1. Job is an association class.
2. The arrow in the association "works-for" shows the direction of the association.
3. The association "manages" is between two job objects.
4. The {or} designation specifies the partition of the account class into two classes: person (account) and corporate (account).

***Example:*** For:



The association Enroll describes the association 'type'. An association is actually between two *objects* (a student object and a course object). Examples:

S1 -- C1: meaning student S1 is enrolled in course C1.  
S1 -- C3: (The associations S1--C2 and S1-C4 do not exist. This means the student S1 has not enrolled in the C2 or C4.)  
S2 -- C1  
S2 -- C2  
S2 -- C4  
S3 -- C3  
S4 -- C1  
S4 -- C4

**2.5 Multiplicity**

* Multiplicity can be specified by a number, the symbol \* (many), a range, or a set. Some example:
  + 0..1: zero or 1
  + 1..1: only 1
  + 1: may be 0..1 or 1..1; usually interpreted as 1..1
  + 0..\*: zero or many
  + 1..\*: 1 or many
  + \*: many; may be 0..\* or 1..\*
  + 1..4: 1 to 4
  + {1, 2, 6}: 1, 2 or 6
  + {1, 3:5, 7:9}: 1, 3, 4, 5, 7, 8, 9
* Multiplicity is a very common source of errors. Please refer to the explanation of the following diagram until you are very clear about it.

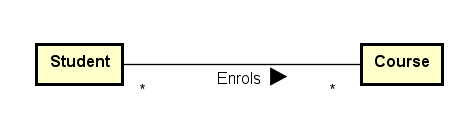


* Meaning:
  + *Every* X object must be associated with *n* Y objects.
  + Every Y object must be associated with m X objects.

***Example***

What do you think about these class diagrams?

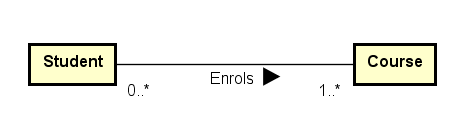
(a)



Assumptions made:

1. A student may take many courses.
2. Not sure whether a student is allowed to take zero course since \* (instead of 0..\* or 1..\* is used).
3. A course may have many students enrolled.
4. Not sure whether a course has no student enrolled since \* (instead of 0..\* or 1..\* is used).

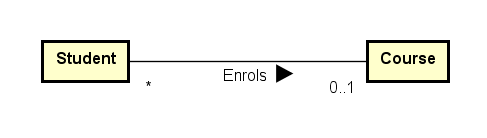
(b)



Assumptions made:

1. A student must be enrolled in one or more courses (may not be a reasonable assumption).
2. A course may have 0 or more students enrolled.

(c)



Assumptions made:

1. A student can only be enrolled in 0 or 1 course only (sound not reasonable).
2. A course may have many students enrolled.
3. Not sure whether a course has no student enrolled since \* (instead of 0..\* or 1..\* is used).

**Aggregation indicator**

1. aggregation (hollow diamond) and composition indicator (solid diamond):
2. Aggregation models the ‘a-part-of’ relationship (whole-part). E.g., car-wheel.
3. Composition is a *strong* form of aggregation: the part's lifecycle is dependent on the whole's lifecycle; e.g. university-department, building-room.
4. They can also be represented by using multiplicity.

***Example:*** Aggregation and Composition

A diagram of a graph

Description automatically generated

What do you think about this composition and aggregation examples in: [http://en.wikipedia.org/wiki/File:Congregationalism](http://en.wikipedia.org/wiki/File:AggregationAndComposition.svg)?

**Ternary Associations**

* N-ary associations are represented using a diamond connecting to participating classes.
  + Not so common.
  + May be modeled as a class instead.
* A ternary association involves three participating objects.

An example from a tutorial:

A diagram of a football team

Description automatically generated

Notes:

* In modeling, a ternary association can reasonably be replaced by a number of binary associations.
* Don't use n-ary associations where n>=3 unless you are sure.

**Generalization and Specialization**

* Generalization is represented by a hollow triangle at the superclass.
  + Generalization models the 'a-kind-of’ association.
  + It is mainly used to
    - manage classes with common data members and methods by put these common members into their superclass.
    - provide inheritance.
    - avoid multiple copies of member definition.
* Some options of generalization includes:
  + discriminator (the name of the partition),
  + powertype (a class in which an instance of it is a subclass of the superclass),
  + constraints (overlapping, disjoint, complete, incomplete and user defined constraints).

***Example:***

A diagram of a vehicle

Description automatically generated

A diagram of a tree

Description automatically generated

* There are many possible options and extensions.

**Constructing class diagrams: some tips**

1. There are many methodologies and best practice tips to construct effective class diagrams.
2. There are many possible modeling options: e.g., classes versus attributes, classes versus associations, multiplicity, etc.
3. Need to fully understand the assumptions and implications when making modeling decisions.
4. Do not model implementation details in earlier modeling phases.

**3. Example: toyu**

A reasonable conceptual model of the toyu database in UML:

A diagram of a computer program

Description automatically generated with medium confidence