**CSCI 4333.2**

9/9/2024

**Database Modeling**

by K. Yue

**1.Modeling and Modeling Transformation**

* Why modeling?
  1. Many DB developers may just directly construct the relational schema without formal data modeling. What are the problems with that?
     1. It may work for very simple problems.
     2. For more complicated problems, it is necessary to capture the detailed requirements before designing a solution.
  2. Initial ideas are usually based on needs. Their definitions can be very coarse and ambiguous. Real-world examples from past UHCL capstone projects:
     1. "I want a system to store and retrieve sale receipts for small business to claim sale tax waiver."
     2. "A mobile phone application to share and manage disaster recovery information."
  3. How do we construct a precise and executable computer solution to a vague real-world problem?

Software development is logical modeling (step [1] in the following diagram): constructing an *executable logical* model (such as using SQL, Java, etc., as the logical language.)

A diagram of a problem solving process

Description automatically generated

Logical modeling spans many phases in the software lifecycle:

1. Analysis
2. Design: architectural and component
3. Implementation (coding)

Furthermore, logical languages are closer to the machine world, and optimized accordingly. They are not designed to describe the real-world or capture real world requirement.

Conceptual modeling is introduced as shown in the diagram below:

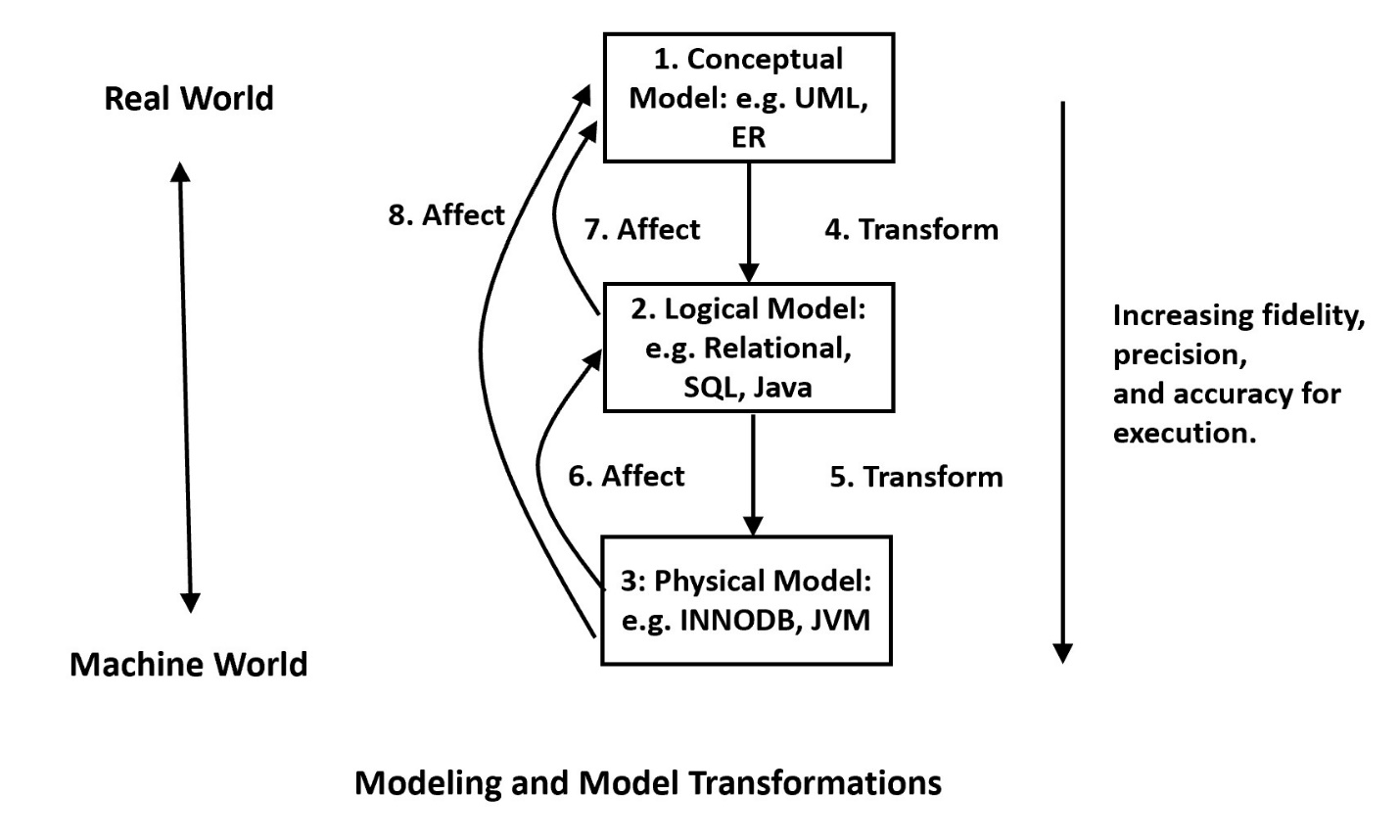
A diagram of a problem solving process

Description automatically generated

Advantages of conceptual modeling:

1. Conceptual modeling languages, such as Unified Modeling Language (UML) or Entity-Relationship (ER) model, are designed to describe real-world problems and can better capture and refine problem requirements.
2. The process is a n-step process, breaking down the complexity.
3. Iterative refinement of the conceptual model provides eventually the necessary fidelity for different software development phases.

In Model-Driven Software Engineering (MDSE), software development is the development of a sequence of models, transforming higher level models to lower-level models in the process.



**2. Motivation for learning modeling**

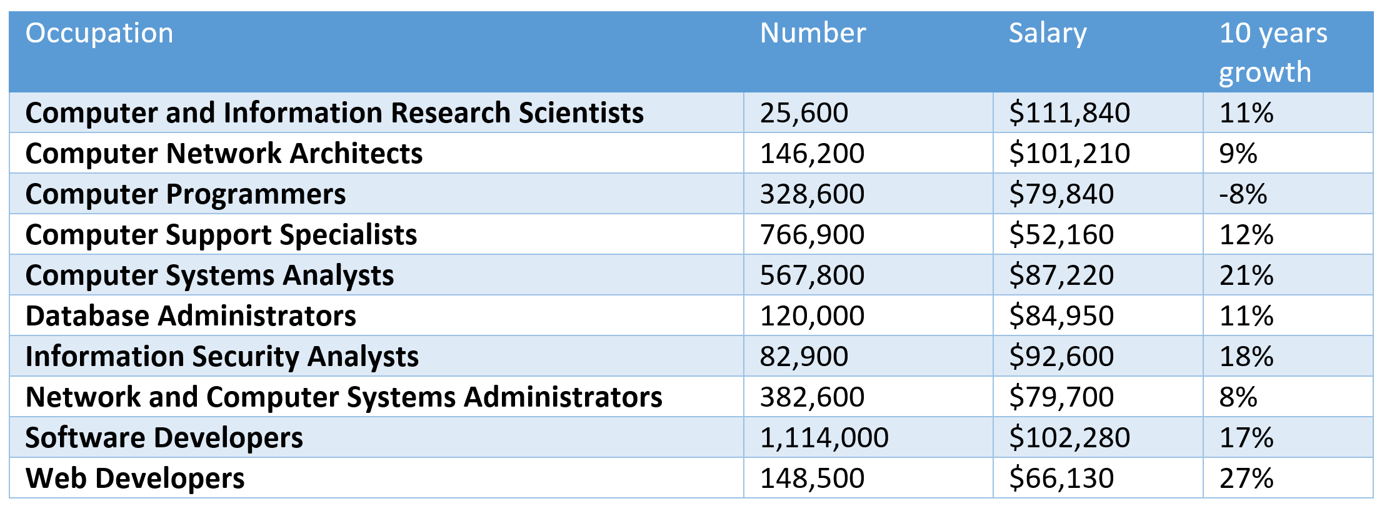
**Bureau of Labor Statistics, Computer and Information Technology Job outlook 2021 to 2031:**

A table of statistics with numbers and text

Description automatically generated

**Older data for comparison:**

Bureau of Labor Statistics, Computer and Information Technology Job outlook 2016 to 2026:



Average annual wage over all jobs in 2017: $37,600. 2016 to 2026 expected overall growth rate is 7%.

* A main difference between "software developers" and "computer programmers" are modeling and model transformation.

**3. DB Modeling**

**What is involved in DB modeling?**

1. Understand, capture, and refine the application requirements until they are clear and precise enough for design and implementation.
2. Communication is crucial. "Do*not* build *your* DB application. Build the DB application for the users."
3. Complexity management.
4. Three main topics in modeling tools and techniques:
   1. Modeling language: e.g., UML, ER (Entity Relationship)
   2. Theory and 'Good' practices: e.g., architectural and design patterns.
   3. Modeling process: what process to follow in modeling, e.g., Agile.

**3.1 Understanding the problem domain**

How do people solve problems?

1. Obtain an initial understanding of the problem.
2. Form a model to capture the understanding.
3. Refine the model to include necessary details.
4. Based on the model, devise a solution.
5. Implement the solution.

A serious problem of many novices is that they do not spend enough time and effort in steps (1) to (3) (i.e., modeling) and jump to (4) and (5) quickly.

* Your supervisor/user/customer/domain expert may not have a clear model themselves!
* In most cases, for novice software developers: what you think the applications should be is NOT as important! Instead, what the domain experts and the users think is crucial.

**What can you do to understand the problem and model it?** Some good practices:

1. Start by *asking a lot of questions*.
2. Collect and analyze as much documentation as possible.
3. Study and use the existing system: find out what is currently being used. Use them alongside the users. Participate in the work process.
4. Model, write, and document!
5. Use your own words and avoid the 'copy and paste' syndrome.
   1. A simple example below.
   2. The copy and paste syndrome in modeling and software development is much more difficult to unearth and costly to fix. A realistic example of an application letter to a Data Science Assistant Professor postion at UHCL.

A close-up of a letter

Description automatically generated

**Some possible generic questions to ask for modeling.**

1. What are the purposes of the application?
2. What problems will the application solve? What value will it provide?
3. What are the boundaries and context of the applications?
4. Who are the users?
5. Who are the domain experts?
6. What documentation is available?
7. How does the existing system work?
8. What are the workflow processes of the current system?

Prepare your own list!

**3.2 Modeling Languages**

**Two important options for database modeling languages:**

1. Entity Relationship (ER) Modeling and Extended ER (EER) Modeling: more relational database specific.
2. UML modeling: more general purpose.

**Some advantages of ER diagrams:**

1. Ease of understanding
2. Simpler
3. More specific to relational database modeling
4. Good collection of theory and best practices
5. Good vendor and tool support

**Some disadvantages of ER diagrams:**

1. Lower expressiveness
2. Not concise
3. Lack of standards
4. many versions that can be confusing
5. Narrow focus on relational databases

**What do we look for in a modeling language?**

1. Completeness: can it capture all needed information?
2. Expressiveness: can it capture information in an efficient and easy way for discussion and communication?
3. Domain support: how well does it support capturing the information in the application domain?
4. Precision: Is it detailed enough?
5. Accuracy: Is it ambiguous?
6. Conciseness: What is the 'information density'?
7. Standardization: Are there many competing versions?
8. Tool support: Are there strong tool support?
9. Community support: Are there a strong community?
10. Extensibility: Can it be extended to satisfy a specific domain?

We will use on UML in this course.

Read: A Badia and D. Lemire. [A call to arms: revisiting database design](http://dl.acm.org/citation.cfm?id=2070750). SIGMOD Record 40, 3 (November 2011), 61-69.

**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. Study 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 any 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 a unique id and name. A car maker may make many cars. For example, Honda, which may have a manufacturer id of 10001, makes Civic and Accord.

**Concepts:**

1. Car:
   1. Class X: have attributes, form relationships; flexibility; exists by themselves. X,
   2. Attributes X: only have values; no attributes; simple; existence depends on other artefacts. X of Y.
   3. Associations X: X associates Y with Z.
   4. No la
2. Manufacturer: class
3. Types of cars?
4. Name
5. Unique Id of manufacturer: attribute (X) of manufacture (Y): UML stereotype
6. Car maker
7. May make many cars
8. Used car: year?
9. Honda: example of manufacturer?
10. 10001
11. Civic
12. Accord
13. Color?
14. What are the users and purposes of the applications?

**Analysis and Design**

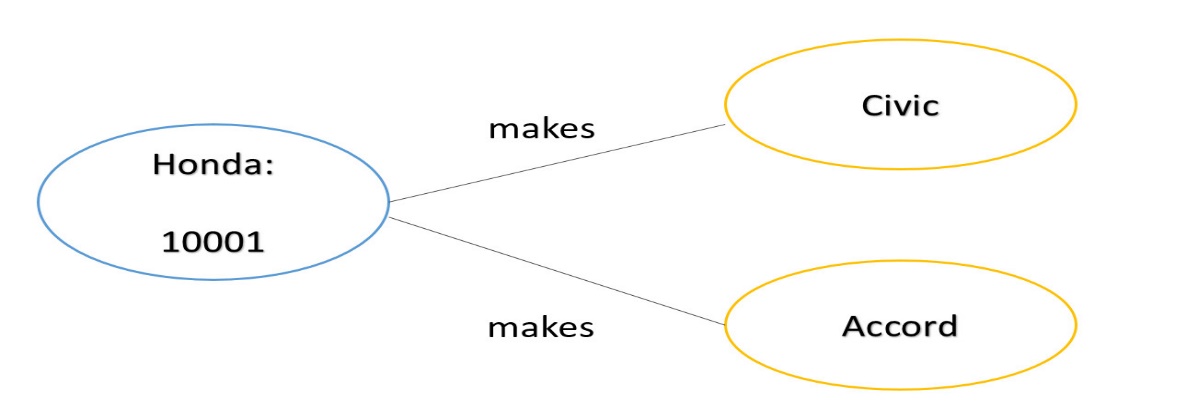
Some observations:

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 (i.e., a manufacturer object such as Honda).
4. Define synonym: manufacturer, car manufacturer and car maker may be the same. Different terms can refer to the same concept.
5. "Unique id": may be modeled as an attribute (name), a property of the manufacturer class.
6. Make additional assumptions: E.g., every manufacturer object *must* have an unique id.
7. 10001: attribute (value) of the id of a manufacturer object.
8. Name: a property of a manufacturer.
9. Another additional assumption: Every manufacturer object must have a name.
10. Car: a class, as there may be many *brands* of cars.
11. Prepare questions: E.g., do we need to introduce the concept *model* (e.g., Coupe, Sedan, Si Coupe)?
12. Civic and Accord: object instances of Car.
13. Additional assumption: Every car must have a name as its attribute.
14. Make, or manufacture: a relationship between a manufacturer (object) and a car (object). A manufactor (object) makes (verb: possible asspication) a car (object).

**Class Diagram:**



**Object Diagram:**



**2.3 Classes**

**2.3.1 Basics**

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,
   1. The attribute compartment will eventually need to be clearly modeled.
   2. The operation compartment 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 have different foci.

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 classes Patron, Member and Department with some attributes may be modeled in the first draft of the UML class diagram. Boxes in the diagram above:

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 for Patron below.



* Note that multiplicity can be used to depict *nullable* 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 notations
  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.

**2.3.2 More Properties of Classes**

* A class is a 'first-class citizen.'
  + It has attributes.
  + It can form associations with other classes.
  + It can have operations.
  + 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 in the following diagram until you are very clear about it.

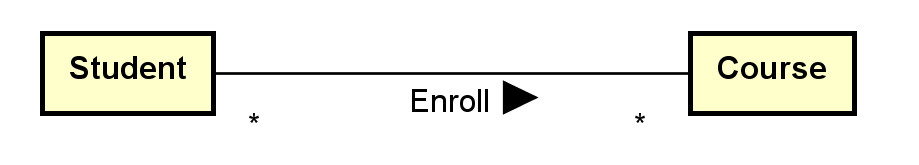


* Meaning:
  + *Every* X object must be associated with *n* Y objects.
  + Every Y object must be associated with m X objects.

***Example***

What do you think about these class diagrams?

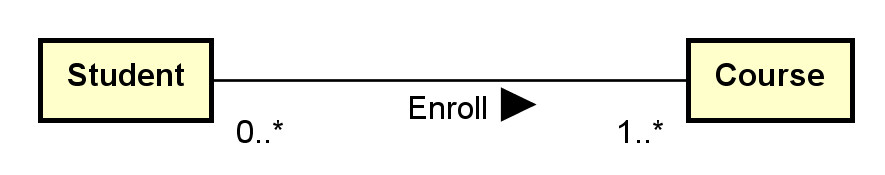
(a)



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).

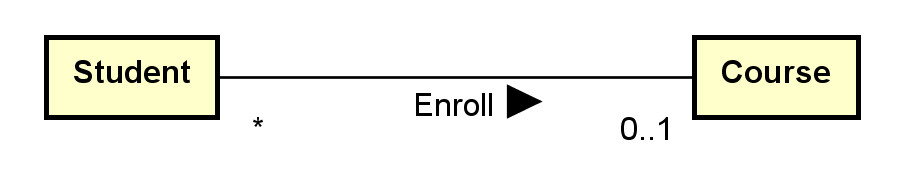
(b)



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 promoting it to a class and add three 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 putting these common members into their superclass.
    - provide inheritance.
    - avoid multiple copies of member definition.
* Some options of generalization include:
  + 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