Designing a Database

Introduction

In this assignment we are going to learn about designing a database. Designing a database consists of determining the structure and relationships of the data we need to store and retrieve. We'll design a database you might use to implement an online blog or social networking application. Consider the data you might find in a typical blogging or social networking application. You'd find users creating blog posts that other users read, like, and comment on. Users socialize by following each other and conversing on various topics.

For this assignment you'll create a UML class diagram that captures data and relationships typical of a social network application. We'll walk through the process giving examples of how to create some of the classes and then ask you to create your own. Look for "***Now you try it***". At the end you'll have a single class diagram that contains all the classes and associations which you will submit as the deliverable.

Learning objectives

- Using UML (Unified Modeling Language) Class Diagrams to represent data and its relationships
- Implementing one to many relationships
- Implementing many to many relationships

UML Class Diagram Tools

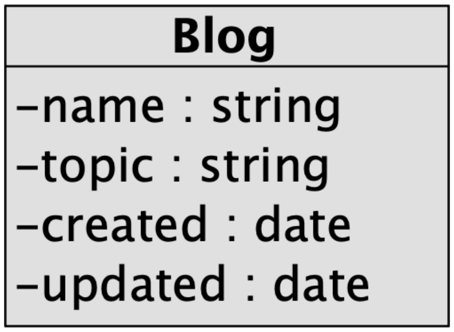
Here are several tools you can use to create UML Class Diagrams:

- Lucid Chart
- Ideal Modeling & Diagramming Tool for Agile Team Collaboration
- Creately: Chart, Diagram & Visual Canvas Software
- Flowchart Maker & Online Diagram Software

Designing a Database with a UML Class Diagram

Classes group related data concepts under a single idea. For instance a ***Blog*** class groups data concepts such as the name of the blog, the main topic of the blog, how many followers, etc. The attributes (or properties) of a ***Blog*** class might look as shown below and represented as a class as shown on here on the right

Attribute	Type
name	string
topic	string
created	date
updated	date



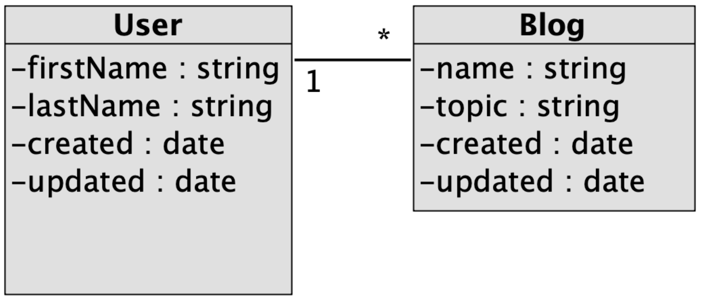
Using the UML tool of your choice, create a new class diagram that contains the ***Blog*** class shown above. To practice this skill on your own, ***now you try it***. A blog ***Post*** class groups data such as the ***title*** of the post, and the actual text content of the ***post***. Using the following attributes, add a new equivalent class called ***Post*** to the class diagram you're working on.

Attribute	Type
title	string
post	string
created	date
updated	date

Designing One to Many Relationships

Users can create many different blogs, i.e., one can be on the topic of space, another on cars, and another on brain to computer interface. We refer to this kind of relationship between one entity related to many instances of some other entity, as a ***one to many*** relationship. In UML class diagrams we capture this relationship with a line between the classes and cardinalities at each end of the line next to the class. The cardinality denotes how many of each class instances participate in the relationship. We put a 1 on the ***one side*** of a ***one to many*** relationship. We put an asterisk ("*") next to the ***many side*** of a ***one to many*** relationship.

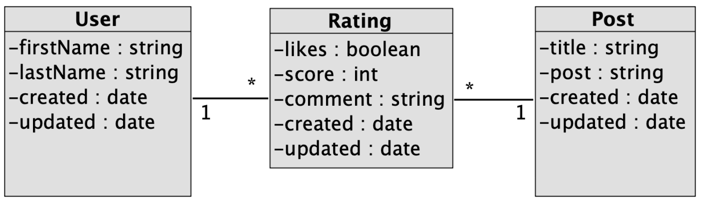
Using the ***User*** class described in an earlier assignment, let's establish a ***one to many*** relationship between the ***User*** class and ***Blog*** class by adding a line between from the ***User*** class to the ***Blog*** class as shown here. We'll set the cardinality on the ***User*** side to 1 and on the ***Blog*** side to * to denote that one ***User*** instance can be related to many ***Blog*** instances. Re-create the classes and associations shown here on the right to the UML class diagram you're working on. ***Now you try it***. Using this example, create a ***one to many*** relationship between the ***Blog*** and ***Post*** classes describing the fact that a ***Blog*** can have many ***Posts***.



Designing Many to Many Relationships

When a user creates a post, other users can rate the post in several ways. They can ***like*** or ***dislike*** a post, they can give a star ***rating*** or ***score***, or they can leave a ***comment*** about the post. A single user can rate ***many*** posts, and a single post can be rated by ***many*** users. We refer to this kind of relationship as a ***many to many*** relationship. In UML class diagrams we can capture this type of relationship with an additional class with several one to many relationships with the other classes it relates to. The cardinalities of the associations are * on the new class, and 1 on the other classes.

In the class diagram you're working on, let's add a ***Rating*** class that captures how a user might rate someone's ***Post***. The ***Rating*** class captures whether a user likes the post, a score, and a comment. The ***Rating*** class has two one to many associations, one with the User class and the other with the ***Post*** class.



When a user creates a post, other users can reply to the post. A single user can reply to many posts, and a single post can be replied to by many users. There is a ***many to many*** relationship between the users and the posts. ***Now you try it***. Using the example of the ***Rating*** class, create a class called ***Reply*** to represent the reply a user might make to a post. A user can reply to many posts, and a post can be replied to by many users. Implement the ***Reply*** class as a ***many to many*** relationship between users and posts. The ***Reply*** class should have the following attributes:

Attribute	**Type**	**Description**
reply	string	a string capturing the reply
created	date	the date when the reply was done
updated	date	the date when the reply was last updated

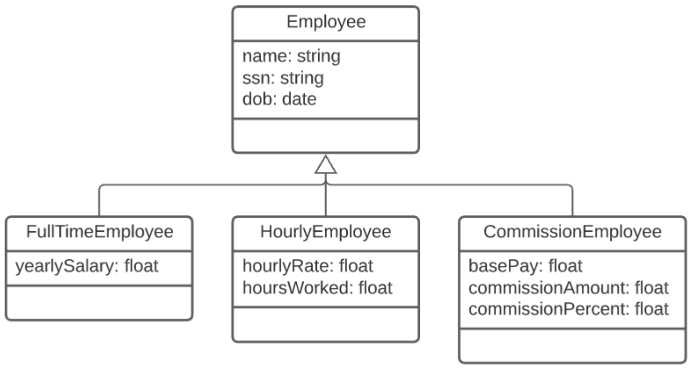
Challenge (required for graduates)

The following are a set of practice sections exploring some additional database design topics. They are optional or extra credit for undergraduate students, but are required for graduate students. For extra credit policy reach out to your instructor.

Inheritance

Inheritance is an object oriented concept for implementing reusability and separation of concern. Inheritance allows implementing hierarchical relations between classes where classes hiegher in the hierarchy declare properties common to classes lower in the hierarchy.

Consider the class diagram here on the right. The ***Employee*** base class defines properties common to its derived classes such as ***name***, ***ssn***, and ***dob***. Class ***FullTimeEmployee*** inherits properties from its base class and adds an additional field ***yearlySalary*** special to its class. HourlyEmployee similarly inherits from its base class and adds two additional fields ***hourlyRate*** and ***hoursWorked***. Finally ***CommissionEmployee*** adds special fields ***basePay***, ***commissionAmount***, and ***commissionPercent***. Recreate the diagram here and include it in your deliverable.

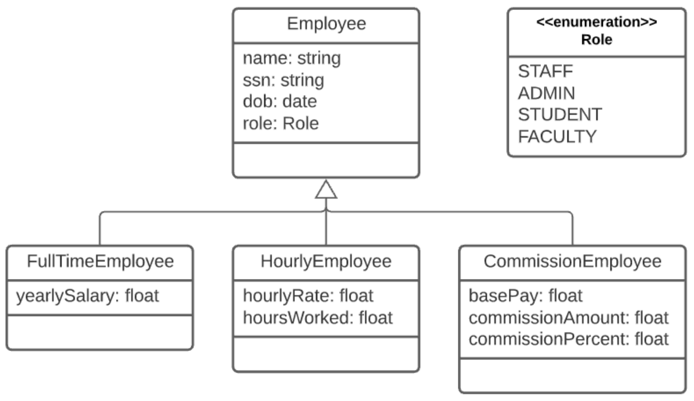


Now you try it. Create a class diagram with base class ***Vehicle***, and derived classes ***Automobile*** and ***Boat***. The ***Vehicle*** class should declare fields common to both ***Automobile*** and ***Boat*** such as ***year***, ***brand***, ***model***, ***length***, and ***weight*** with types ***int***, ***string***, ***string***, ***float***, and ***float***. The Automobile class should declare fields specific to this class such as ***doorNumber***, ***cylinders***, ***vin***. The ***Boat*** class should declare fields specific to this class such as ***beam***, ***draft***, ***length_over_all***, all floats. Include the new diagram as part of your deliverable.

Enumeration

Enumerations are useful for declaring valid values for other fields. This improves the overall quality of the data model and the software written to use the data model. Consider a variable called ***genre*** which we initially declare to be a ***string***. Values we would expect for the ***genre*** field are ***COMEDY***, ***DRAMA***, ***SCIFI***. But what if someone enters something invalid like ***FACULTY*** or ***STAFF*** or maybe misspells a valid value such as ***COMEDI*** or ***DRANA***. A better way to implement the genre field is to constraint its values to only a set of valid, approved values. Enumerations play the role of describing valid values that can constrain a field. For instance we could create an enumeration called ***Genre*** and define the valid values ***COMEDY***, ***DRAMA***, and ***SCIFI***. We can then use this new datatype to declare the ***genre*** field. Now we can be sure that field ***genre*** will only ever have valid values. Let's take a look at an example.

Consider the class diagram here on the right. We've added field ***role*** describing the ***Employee***'s role. We could declare the type of the field as ***string***, but a better solution is to use a strong type such as ***Role*** which declares the valid values of the ***role*** field. This is a better solution because the ***string*** type is too permissive allowing invalid values for the ***role*** field. The ***Role*** datatype constrains the ***role*** field to only the valid values so it's easier to test the application using the database. Recreate the diagram here and include it in your deliverable.



Now you try it. Let's continue with the class ***Vehicle*** diagram from earlier. Create an enumerated datatype called ***Brand*** with values ***HONDA***, ***JEEP***, ***TESLA***, ***AUDI***, ***BMW***, and ***VW***. In the ***Vehicle*** class replace the ***brand*** field's datatype from ***string*** to ***Brand***. This improves the design because ***string*** would allow entering an invalid or non existent brand. Changing it to ***Brand*** constrains the valid values of the ***brand*** field to only have the valid values listed in the ***Brand*** enumeration. Include the new diagram as part of your deliverable.

Realizing relational models

Realizing a relational model consists of creating an SQL equivalent to conceptual models such as those documented as UML diagrams. The relational model would consist of SQL tables that would implement a concrete realization of the abstract conceptual model. The realization of the earlier ***Employee*** class diagram would consist of the relational model described below. The ***roles*** table below implements the enumerated data type ***Role*** described earlier. Its primary key ***role***, guarantees no duplicate roles and referencial integrity from other records. We populate the table with the valid roles STAFF, ADMIN, FACULTY, and STUDENT

```
CREATE TABLE `db_design`.`roles` (
  `role` VARCHAR(10) NOT NULL,
  PRIMARY KEY (`role`));
INSERT INTO `db_design`.`roles` (`role`) VALUES ('STAFF');
INSERT INTO `db_design`.`roles` (`role`) VALUES ('ADMIN');
INSERT INTO `db_design`.`roles` (`role`) VALUES ('STUDENT');
INSERT INTO `db_design`.`roles` (`role`) VALUES ('FACULTY');
```

The table ***employees*** below, implements the ***Employee*** datatype described earlier. It contains the same fields ***name***, ***ssn***, ***dob***, and ***role***. The dataypes of the fields are self explanatory and equivalent to the conceptual model. The ***role*** field is VARCHAR, but it is constrained by referencial integrity to only allow values already in the ***roles*** table implemented earlier.

```
CREATE TABLE `db_design`.`employees` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `name` VARCHAR(45) NULL,
  `ssn` VARCHAR(45) NULL,
  `dob` DATETIME NULL,
  `role` VARCHAR(45) NULL,
  PRIMARY KEY (`id`),
  INDEX `employee_role_idx` (`role` ASC),
  CONSTRAINT `employee_role`
    FOREIGN KEY (`role`)
    REFERENCES `db_design`.`roles` (`role`)
    ON DELETE NO ACTION
    ON UPDATE NO ACTION);
```

To implement the ***FullTimeEmployee*** specialization we use a table that declares an equivalent yearly_salary field, and enforce its primary key to reference the primary key of some related record in the ***employee*** table. This way we make it clear that records in both tables have a one to one correspondence. We also configure the foreign key to cascade a delete command to the parent record, making it clear that it can not exist without its parent record.

```
CREATE TABLE `db_design`.`full_time_employees` (
  `id` INT NOT NULL,
  `yearly_salary` FLOAT NULL,
  PRIMARY KEY (`id`),
  CONSTRAINT `full_time_employee_id`
    FOREIGN KEY (`id`)
    REFERENCES `db_design`.`employees` (`id`)
    ON DELETE CASCADE
    ON UPDATE CASCADE);
```

The implementation of ***HourlyEmployee*** and ***CommissionEmployee*** follow a similar logic as shown below.

```
CREATE TABLE `db_design`.`hourly_employees` (
  `id` INT NOT NULL,
  `hourly_rate` FLOAT NULL,
  `hours_worked` FLOAT NULL,
  PRIMARY KEY (`id`),
  CONSTRAINT `hourly_employee_id`
    FOREIGN KEY (`id`)
    REFERENCES `db_design`.`employees` (`id`)
    ON DELETE CASCADE
```

```
   ON UPDATE CASCADE);
```

```
CREATE TABLE `db_design`.`commissioned_employees` (
  `id` INT NOT NULL,
  `base_pay` FLOAT NULL,
  `commission_amount` FLOAT NULL,
  `commission_percent` FLOAT NULL,
  PRIMARY KEY (`id`),
  CONSTRAINT `commission_employee_id`
    FOREIGN KEY (`id`)
    REFERENCES `db_design`.`employees` (`id`)
    ON DELETE CASCADE
    ON UPDATE CASCADE);
```

No you try it. Realize the conceptual design of the automobile database created earlier into an equivalent relational model. Provide all relevant create and insert statements as part of your deliverable.

Deliverables

On Canvas or Blackboard, submit the class diagrams you developed in this assignment as a PDF. If you are graduate students, also provide the solution to they Challenge section.