Data Modeling & UML

but why data models?

Data modeling

We start with an **enterprise**: a problem to be solved in software, and the real-life **entities** needed to represent that problem.

We then create a **data model** for the enterprise: an abstraction that organizes elements of **data** and how they relate to one another, and to the properties of the real-world entities. We create **classes** that represent the “types” of the entities, identifying the **attributes** of those entities, and the nature of the relationships between the entities.

There are many ways to diagram a data model; we will use **UML**.

Creating a model

1. Identify the **strong entities** (nouns) of the enterprise.
2. For each *type* of entity, create a class to represent those instances.
3. Write a natural-language *description* of each class. A good description:
    a. Is **brief**, just 1-3 sentences.
    b. Focuses on the BIG PICTURE of the class and how it fits into the enterprise.
    c. Does **not** list out class attributes, which will be shown in UML instead.
    d. Limit itself to information relevant to the enterprise requirements.
    e. Skips the **behaviors** of the class, which is part of OOP and not general data modeling.
4. Identify the *natural attributes* of each class, including their types.
5. Indicate the *relationships* between the classes, including their *cardinalities*. (Later.)
6. Diagram the model using an appropriate tool. (UML in this course.)
7. Translate the model to a physical implementation. (Later.)

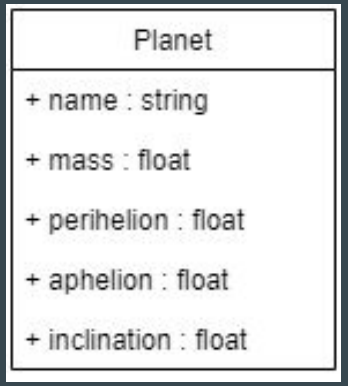
Example model

Enterprise description: I want to build a visualization of our solar system, showing each planet, its moons, and animating their orbits at different time scales. To do that, I must represent each planet that orbits the sun, certain parameters of its orbit, and its natural satellites (moons).

Description:

A **Planet** is an animated object that orbits the sun in an elliptical path. It may have many natural satellites.

Notice: no mention of age, composition (not relevant to this enterprise); no listing of attributes (name, mass, ...). Short and to the point.



UML diagram for Planet class.

We didn't deal with "satellites" at all. More on that later...

Diagramming Tools

Using draw.io for UML (live demo during lecture).
“Automobile” class and “Museum” class.

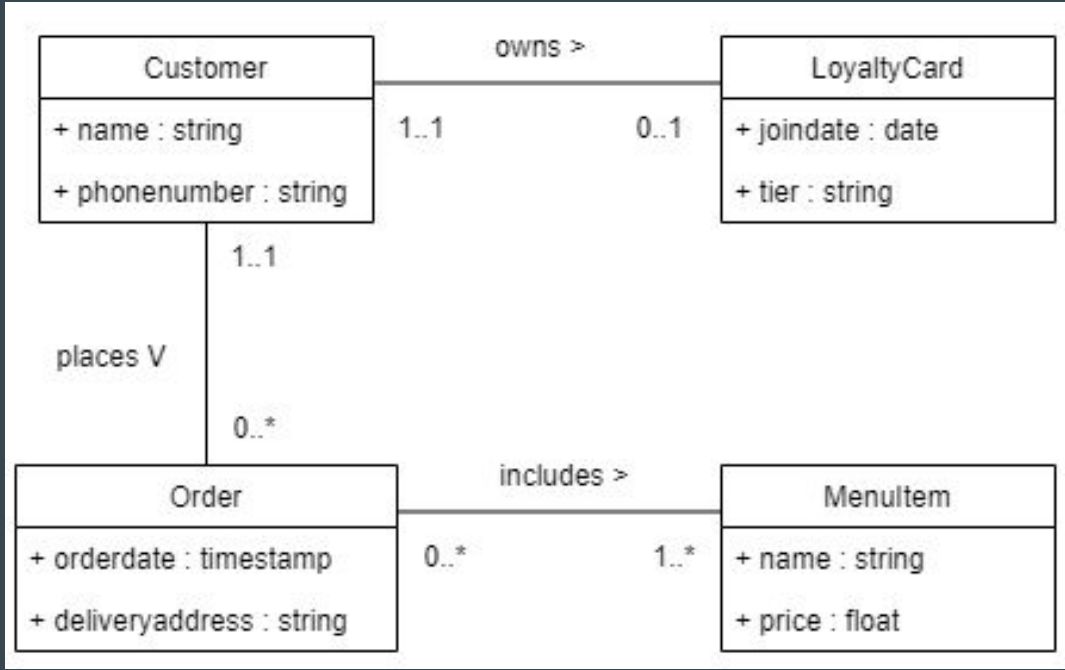
Relationships between entities

When modeling an enterprise, it is almost guaranteed that some entities will have **relationships** with others. A relationship is when one entity class requires another to express one of its fields. Examples:

- An Automobile has a registered Owner who is a person with a name, address, driver's license number, etc. Many different Automobiles can be registered to the same Owner instance.
- A Museum displays many pieces of Art. There can be many Museums, each with its own distinct set of Art pieces.
- A Pizza can be made with (almost) any combination of Ingredients. An Ingredient specifies a serving size and a price that contributes to the Pizza's overall cost.

UML associations

An entity's relationships are modeled in UML using an **association**, a pointer-less line between two classes showing the **cardinalities** of the relationship.



Shown here:

- A 1-to-1 association between Customer and LoyaltyCard.
- A 1-to-many association between Customer and Order.
- A many-to-many association between Order and MenuItem.

Cardinalities of associations

Associations between classes A and B can be qualified as:

- 1-to-1: each object of A is associated with a single object of B, and every B is associated with only one A.

- 1-to-many: each A is associated with many objects of B, but each B is associated with only one A.

- many-to-many: each A is associated with many objects of B, and each B can associated with many objects of A.

Cardinalities practice

Identify the cardinality:

1. Between Fathers and their biological Children.
2. Between library Visitors and the Books they might check out.
3. Between Universities and their Buildings.
4. Between Class Sections and the Students that enroll in them.
5. Between automobile Manufacturers and the Models they produce for sale.
6. Between Residents and their Birth Certificates.

Demo: adding a class

Many-to-many associations

A many-to-many association occurs when each end of the relationship allows multiple objects, either 0..* or 1..*. This represents a relationship in which each object of A relates to many objects of B, and each object of B relates to many objects of A.

Examples:

- An actor can star in many films, and a film can star many actors.
- Each user can vote on many surveys/polls, and each survey can be voted on by many users.
- A student can submit solutions for many assignments, and each assignment can be submitted by many students.
- An Olympic athlete can earn medals in many events, and each event gives medals to many athletes.

Association attributes

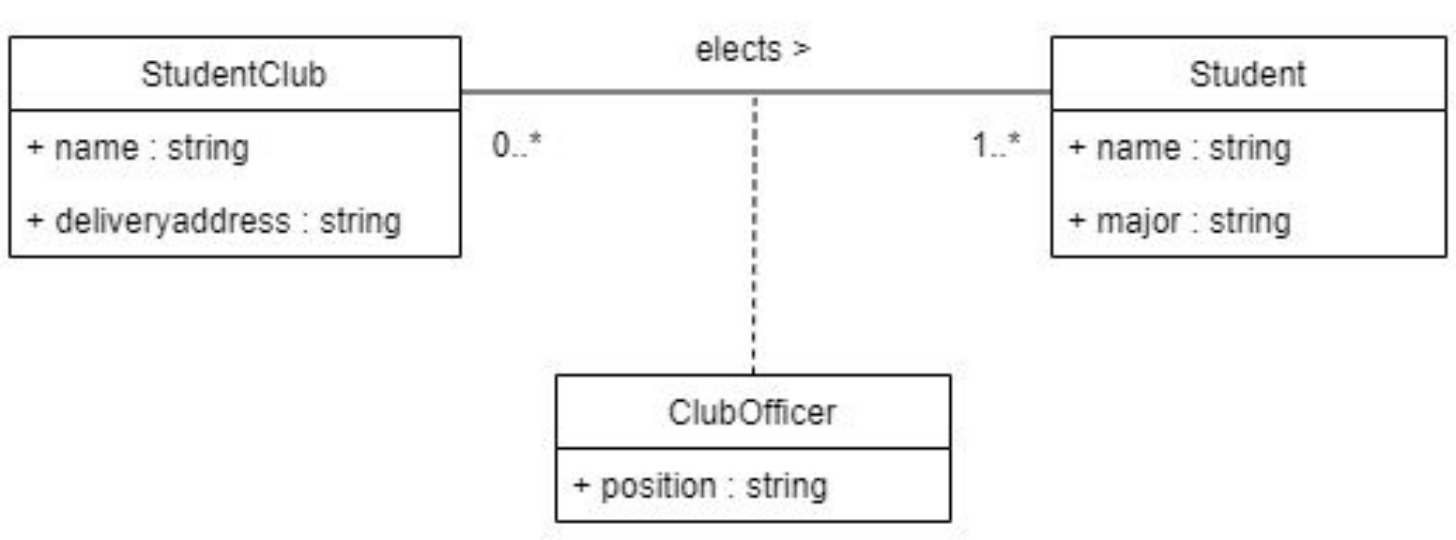
It is often the case that an association has attributes that cannot be part of the related classes.

Examples:

- An actor plays a specific role in a film.
- A user selects a particular option when voting in a poll.
- A student submits their assignment at a particular timestamp.
- An athlete earns a certain placing in their event.

UML many-to-many associations

To model a many-to-many association in UML, we introduce a new **association class** (aka **weak entity**) -- a class designed to represent the association itself. The association class contains any the attributes necessary for the relationship; it is **omitted** if there are no such attributes.



Demo: pick 2 many-to-many examples

Many-to-many with history

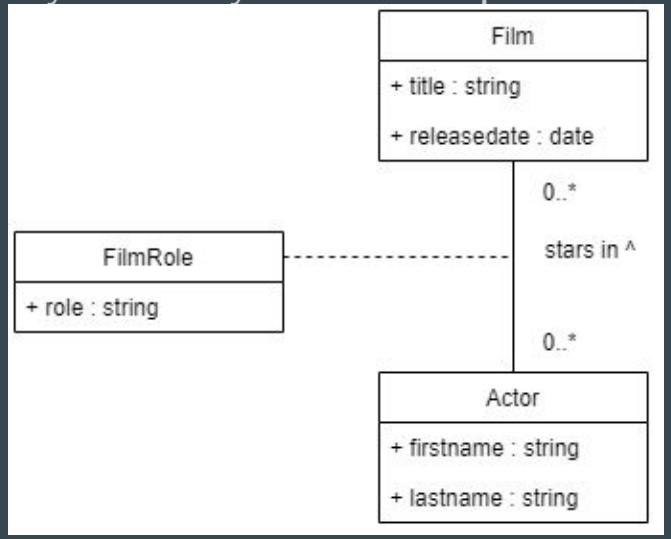
UML's many-to-many notation (with a dashed line) never allows a particular object from A to be associated with a particular object from B *more than once*.

But lots of models require such a "many to many with history"...

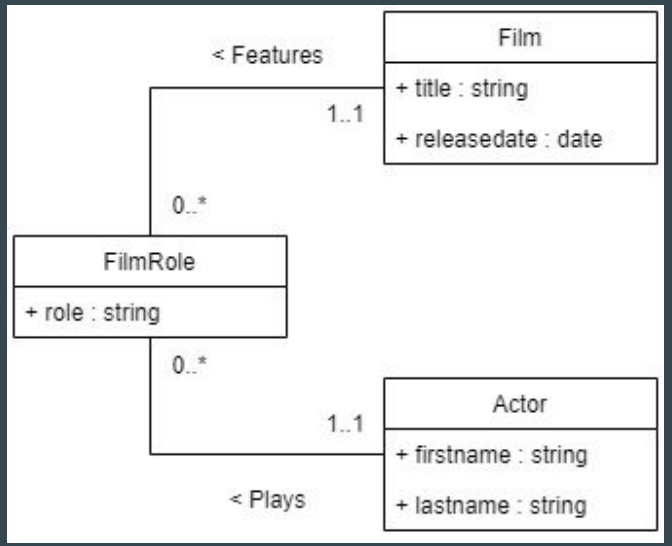
- A library Visitor can check out the same Book more than once.
- A restaurant Order can include the same Menu Item more than once.
- A museum Member can visit the same Museum more than once.
- An Actor can play more than one role in a Film.

Many-to-many with history, part 2

To model a “many to many with history”, we elevate the association class to “full” status. It then becomes part of associations with the two classes that are *actually* in the many-to-many relationship.



Without history: a particular Actor can only play a single role in a specific Film.



With history: an Actor can play any number of Roles in a specific Film.

Demo: the movie showtimes problem

Redundancy

Intro to redundancy

A primary goal for a good data model is **minimizing data redundancy**. Redundant data occurs when the same value is stored many times in a model, and every time that value is stored, it always represents the same quantity/fact from the enterprise.

Challenge: model a baseball ticket, as if we are TicketMaster or StubHub.

```
28.00,Angels,Mariners,Angel Stadium,2017-09-29,C305,H,8
105.00,Angels,Mariners,Angel Stadium,2017-09-29,112,F,1
22.00,Angels,Mariners,Angel Stadium,2017-09-29,C306,FF,18
42.00,Angels,Yankees,Angel Stadium,2017-09-15,530,A,1
42.00,Angels,Yankees,Angel Stadium,2017-09-15,530,A,2
42.00,Angels,Yankees,Angel Stadium,2017-09-15,530,A,3
82.00,Yankees,Angels,Yankee Stadium,2018-07-04,205,EE,4
```

Which of these fields have redundancy problems?

- price
- hometeam
- stadium
- section
- gamedate

Problems with redundancy

So a ticket keeps repeating team names and seat information. What's the big deal?

1. Data size. Each time that "Angels" shows up in the data file is another 12 bytes to store. This can add up over time.

2. Redundant data makes it hard to ensure **data integrity** when creating, updating, and deleting data.
    a. Imagine a copy-paste job that starts with a typo, like "Angles", producing 1,000,000 objects that are misspelled. How many instances have to be fixed?
    b. What if there's a real baseball team named "Angles" in addition to "Angels". How do you fix the typo now?
    c. Does the model prevent a Ticket with a *seat* of -1? Or a *row* of "asdfjkljklasdf"?

Types of redundancy

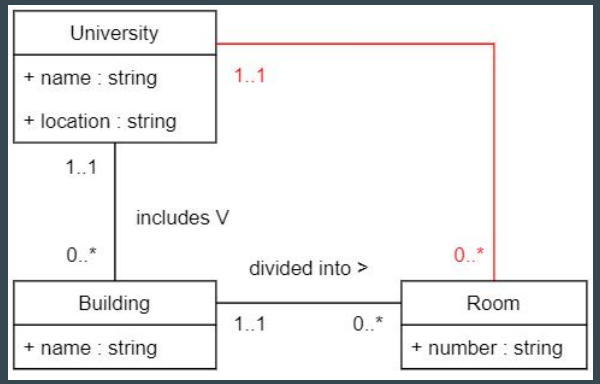
1. Attribute redundancy. Multiple classes have an attribute to represent the exact same quantity from the enterprise.


```
{
    "orderDate" : "2022-09-10T16:52:00",
    "subtotal" : 22.49,
    "pizzas" : [
        {
            "size" : "medium",
            "orderDate" : "2022-09-10T16:52:00"
        },
        {
            "size" : "large",
            "orderDate" : "2022-09-10T16:52:00"
        }
    ]
}
```

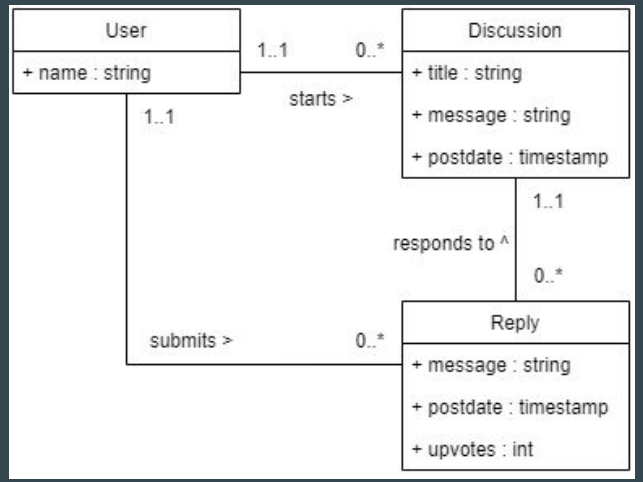
```
{
    "orderDate" : "2022-09-10T16:52:00",
    "subtotal" : 22.49,
    "pizzas" : [
        {
            "size" : "medium",
            "orderDate" : "2022-09-10T16:52:00"
        },
        {
            "size" : "large",
            "orderDate" : "2022-09-17T16:52:00"
        }
    ]
}
```

Types of redundancy

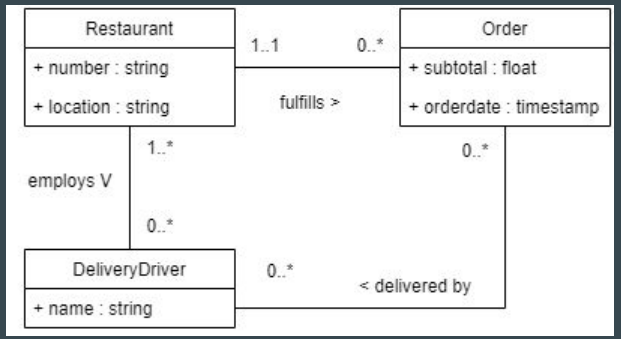
2. Association redundancy. If A is associated with B and B is associated with C, then *sometimes* A does not need to be associated with C.



A university has many buildings, and each building has many rooms. We can determine which university a room is part of by following its association with a building.



The user that starts a discussion is not necessarily the same user that submits a reply to that discussion. **Not** a redundancy!



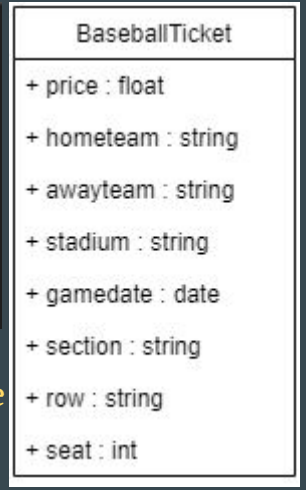
Because a restaurant has many drivers, we can't infer which driver is assigned to an order; we need an explicit link. **Not** a redundancy!!

Types of redundancy

3. Value redundancy. There are many instances of a class, and the same value(s) keep showing up for its attribute(s), and the duplicate values *always* represent the same enterprise information.

```
28.00,Angels,Mariners,Angel Stadium,2017-09-29,C305,H,8
105.00,Angels,Mariners,Angel Stadium,2017-09-29,112,F,1
22.00,Angels,Mariners,Angel Stadium,2017-09-29,C306,FF,18
42.00,Angels,Yankees,Angel Stadium,2017-09-15,530,A,1
42.00,Angels,Yankees,Angel Stadium,2017-09-15,530,A,2
42.00,Angels,Yankees,Angel Stadium,2017-09-15,530,A,3
82.00,Yankees,Angels,Yankee Stadium,2018-07-04,205,EE,4
```

Every time a certain team name occurs, it *always* represents the same thing: that particular real-life team. Likewise for stadium and seat information; “Angel Stadium section C305 row H seat 8” always represents the exact same physical location, no matter the game.

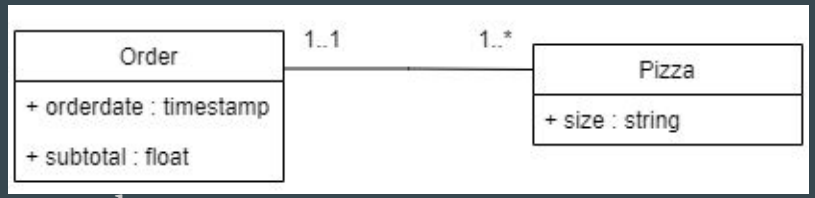



```
[
    {
        "stadium" : "Angel Stadium",
        "gameDate" : "2017-09-29",
        "homeTeam" : "Angels",
        "awayTeam" : "Mariners",
        "price" : 28.0,
        "section" : "C305",
        "row" : "H",
        "seat" : 8
    },
    {
        "stadium" : "Angel Stadium",
        "gameDate" : "2017-09-29",
        "homeTeam" : "Angels",
        "awayTeam" : "Mariners",
        "price" : 105.0,
        "section" : "112",
        "row" : "F",
        "seat" : 1
    }
]
```

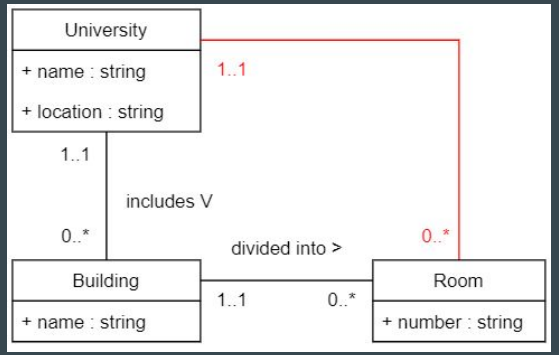
Fixing redundancy

The fixes for *attribute redundancy* and *association redundancy* are easy:

- Attribute redundancy: one of the duplicates is not a *natural attribute* of its class. Eliminate that one, and have one "official" location to store that attribute.



- Association redundancy: look for situations where a "grandchild" links to their "grandparent", and eliminate the link if there's no way the grandchild could have two different grandparents.



Fixing value redundancies

Fixing a value redundancy depends on its nature... is it a **functional dependency**, a **flattened class**, or an **enumerated value**?

Functional dependencies

A **functional dependency** occurs when the value of some attributes of an object can be 100% predicted if you know the value of some other attributes.

Put another way: Z is *functionally dependent* on Y if knowing Y tells you exactly what Z is.

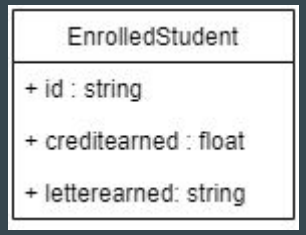
Example:

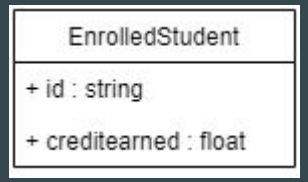
- If I know I earned 85% of credit in a class, then I know I received a B. Letter earned is functionally dependent on credit earned.
- If I know the date and stadium of a baseball game, I know who the home and away teams are. Home and away team are functionally dependent on date and stadium.
- If I know the name of an art piece's artist, I know the birthday of the art piece's artist.

There is never a reason to have Z as an attribute when Y is already one; it may be better off as an association to another class!

Eliminating functional dependencies

Some subkeys are easy to eliminate, like “letter earned”:



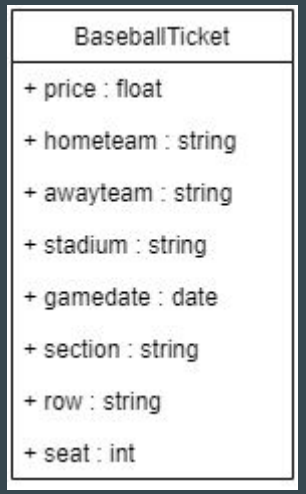


Others are more complicated, and require creating a **new class** for the dependent information...

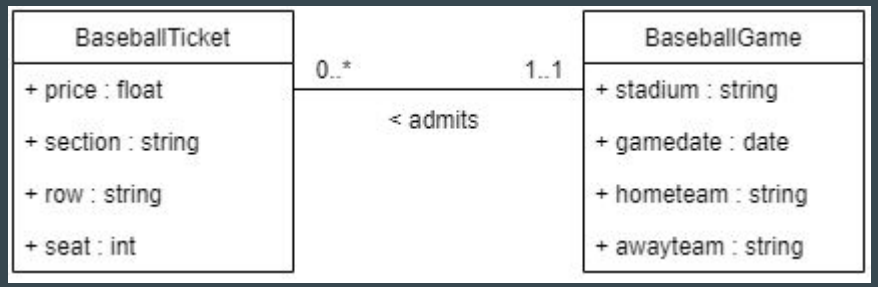
Eliminating other functional dependencies

Suppose a class *T* contains a set of attributes *Z* that is functionally dependent on a set of attributes *Y*. The Z attributes should not be in T, because that creates redundancy.

Instead, we **move** the Y and Z attributes to a new class *U*, eliminating them from T. U then becomes the one-to-many **parent** of T

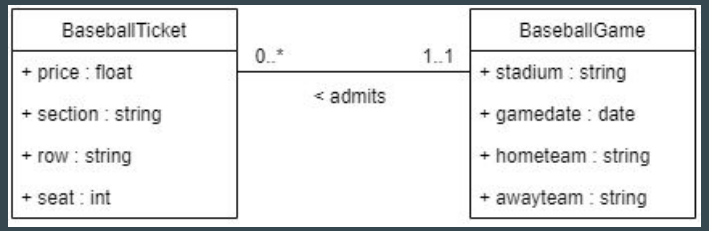


Eliminate the (stadium, date) -> (hometeam, awayteam) redundancy.



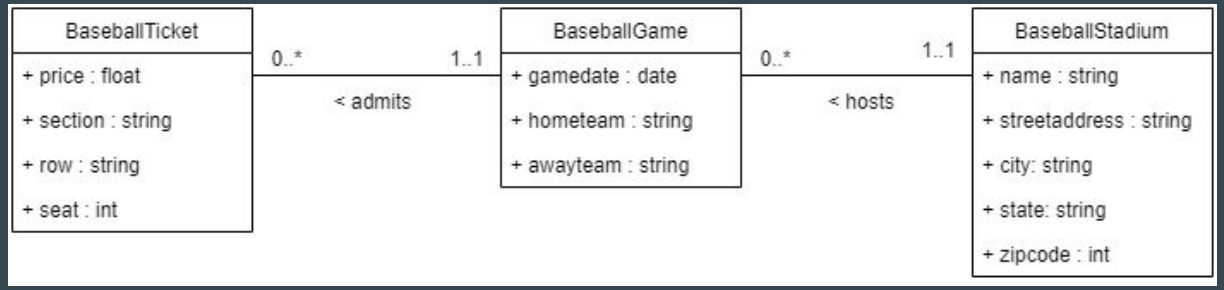

```
{
    "stadium" : "Angel Stadium",
    "gameDate" : "2017-09-29",
    "homeTeam" : "Angels",
    "awayTeam" : "Mariners",
    "tickets" : [
        {
            "price" : 28.0,
            "section" : "C305",
            "row" : "H",
            "seat" : 8
        },
        {
            "price" : 105.0,
            "section" : "112",
            "row" : "f",
            "seat" : 1
        }
    ]
}
```

Flattened classes



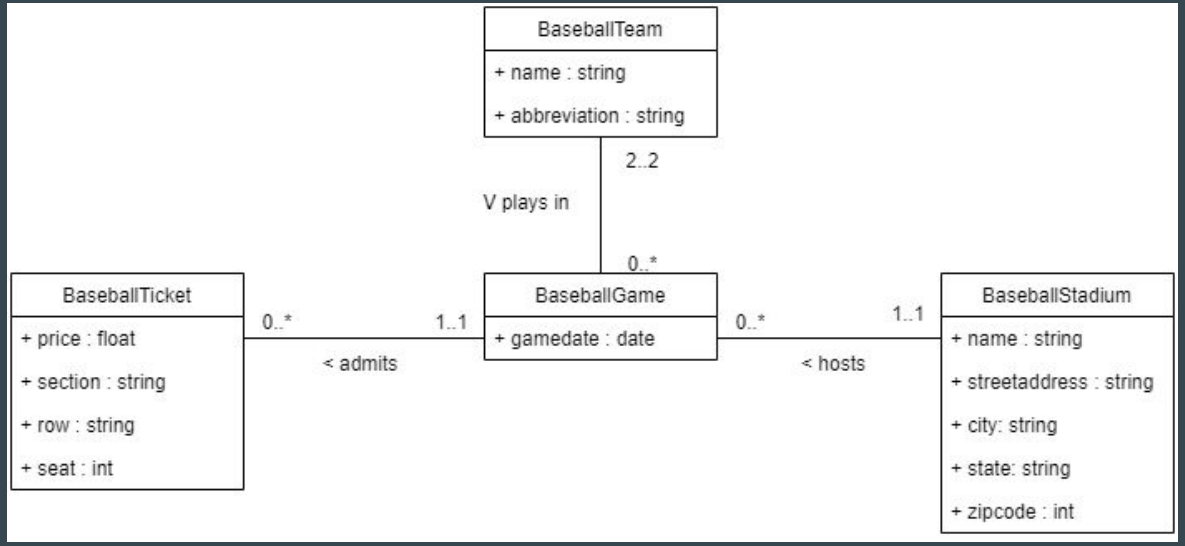
Our BaseballGame class still has a redundancy problem: stadium names are repeated many times in the data. “Angels Stadium” *always* represents the stadium in Anaheim where the Angels play; we should not have to store those 14 bytes of string data once *per ticket* that is sold.

We should recognize a baseball stadium as a *thing*, a *noun*, another **class** from our enterprise. It was a mistake to ***flatten*** the attributes of a stadium into the BaseballGame class. Rather, BaseballStadium should be its own class in the design, having an association with BaseballGame.




```
{
    "stadiumName": "Angel Stadium",
    "street": "2000 E Gene Autry Way",
    "city": "Anaheim",
    "state": "CA",
    "zipCode": "92806",
    "games": [
        {
            "gameDate": "2017-09-29",
            "homeTeam": "Angels",
            "awayTeam": "Mariners",
            "tickets": [
                {
                    "price": 28.0,
                    "section": "C305",
                    "row": "H",
                    "seat": 8
                },
                {
                    "price": 105.0,
                    "section": "112",
                    "row": "f",
                    "seat": 1
                }
            ]
        }
    ]
}
```

We should also recognize that the teams themselves have been flattened into BaseballGame. A BaseballTeam is an obvious noun and should be its own class yet again:



We now have, in effect, a many-to-many w/ history between BaseballTeam and BaseballStadium, with BaseballGame as the association class. It is tricky to represent a many-to-many with JSON – a BaseballGame now needs to be embedded in **both** a BaseballStadium **and** a BaseballTeam – so we will **not** show the JSON implementation for now.

Side challenge: imagine a system for buying tickets to a game, where the system must know which seats have already been sold in order to prevent a double-sale.

```
{
    "stadiumName": "Angel Stadium",
    "street": "2000 E Gene Autry Way",
    "city": "Anaheim",
    "state": "CA",
    "zipCode": "92806",
    "games": [
        {
            "gameDate": "2017-09-29",
            "homeTeam": "Angels",
            "awayTeam": "Mariners",
            "tickets": [
                {
                    "price": 28.0,
                    "section": "C305",
                    "row": "H",
                    "seat": 8
                },
                {
                    "price": 105.0,
                    "section": "112",
                    "row": "f",
                    "seat": 1
                }
            ]
        }
    ]
}
```

Picture this data file having 30,000 <ticket> objects in the <tickets> list for a chosen game. How do we determine if section 115 row A seat 4 is up for sale?

With a linear search!

Here is another reason to avoid flattened classes: they are *expensive to search*, because the data are not organized to support efficient searching.

```xml
<!-- this is all inside a <games> list -->
<baseballgame>
  <!-- game attributes removed for brevity -->
  <sections>
    <section>
      <number>112</number>
      <rows>
        <row>
          <name>A</name>
          <tickets>
            <ticket>
              <price>112.00</price>
              <seat>6</seat>
            </ticket>
            <ticket>
              <price>112.00</price>
              <seat>7</seat>
            </ticket>
            <ticket>
              <price>112.00</price>
              <seat>8</seat>
            </ticket>
          </tickets>
        </row>
      </rows>
    </section>
  </sections>
</baseballgame>
```

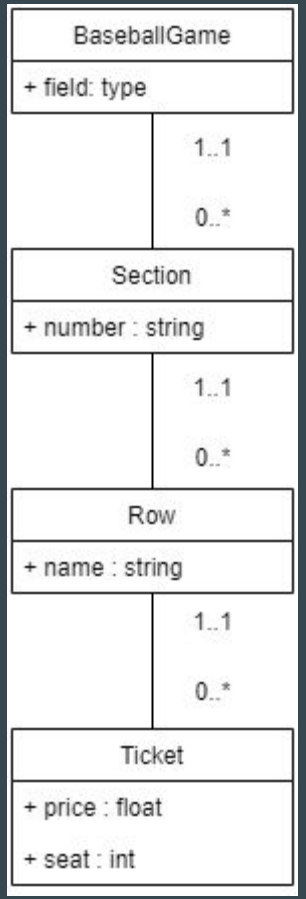
This structure can be searched more easily. We can skim the <section> objects until matching the desired section, then likewise through its <rows>. If those lists are sorted, we can even use a binary search over them!

It looks great, until we build a UML diagram for this XML model...

Object to anything here???

In the goal of making our XML easier to work with, we created a design that does not reflect the enterprise. There has got to be a better way!

Unfortunately the correct solution requires a many-to-many, which is *still* tricky in XML. So we will pass on this for now!



Enumerated domains

Pop quiz: what data type would you use to represent a day of the week, e.g., Monday or Tuesday or...?

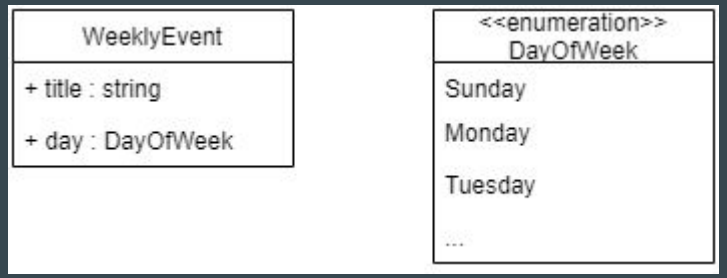
> *Did you say "string"?* ***WRONG.***

If we model "dayOfWeek : string", our model will allow *any string* in that attribute, including "asdfwertas". This is a **data integrity problem**. We also suffer the **redundant data problem**, since we will see the value "Monday" many many times in our data.

Day of the week is really an **enumerated type**: its values come from a finite list of potential values, and no other value is legal for a "day of the week" type. Most programming languages allow you to create an "enum" type; how do we show that in UML?

Enumerations in UML

Fortunately it's easy.



When we implement this model in a programming language or data storage format, our use of an Enumeration will help prevent our data from setting arbitrary string values for the "day" field. The DayOfWeek enumeration type implies that *only* a legal value from the enumeration can be used.

Enumerations vs. flattened classes

It is sometimes unclear if a value redundancy is an enumeration or a flattened class. While both have a similar solution – move the attribute to a new class – there is a semantic difference:

- Enumerations tend to be “hard coded” in the source code of an application. To add or remove a legal enumeration value requires modifying source code and redeploying the application.
- Using a class to represent the type allows the software to instantiate new objects at runtime to represent changes to the legal values. The software does not need to be redeployed. But we get less compile time safety...

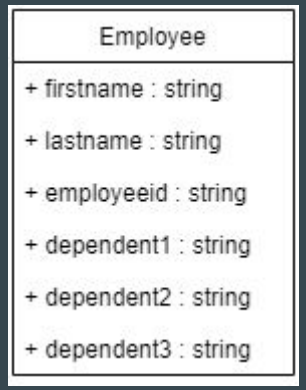
Other integrity issues

Redundant data are not the only integrity issues in a data model. We will briefly introduce another example of an integrity problem: the **repeated attribute problem**.

Repeated- and multi-value attributes

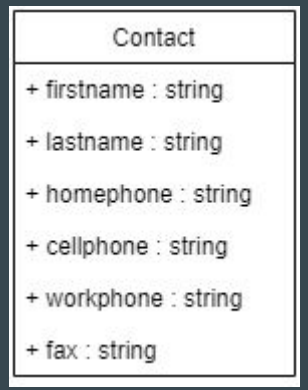
Repeated attributes

Criticize the following data models:



What if an Employee has more than 3 dependents?

What if they have less?



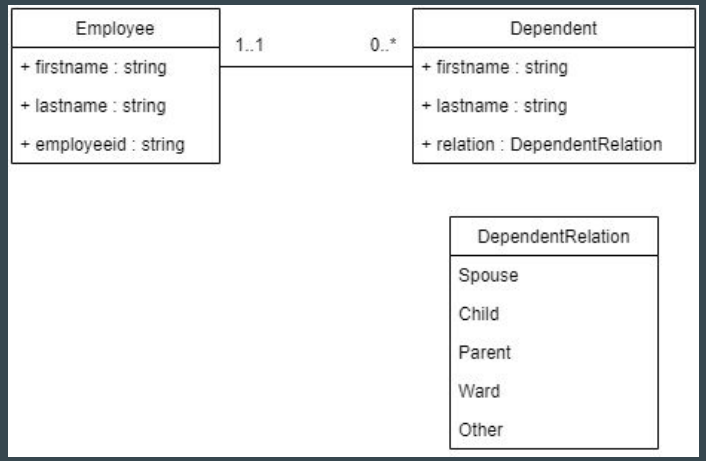
What if a Contact doesn't have a fax or home phone?

What if they have other ways to get in touch?

Both models suffer from the **repeated attribute problem**: instead of using a "list" of values, they use explicit attributes. Attributes cannot be added or removed at run-time, making this a very inflexible design.

Correctly modeling a repeated attribute

To fix this, we recognize that the repeated attributes themselves represent a *noun*, and should be a class in the design associated with the class with the repeated attribute.



"contactmethod" could be made into an enumerated type.

"DependentRelation" is an enumerated type for completeness. Focus on "Dependent" and its many-to-1 association with Employee.

Multi-value attributes

Sometimes laziness leads us to poor decisions. In some designs, there can be a temptation to store a *list* of values as a single *delimited string* instead.

Example: a User of a website has many Favorite Movies. Natural conclusion: “favorite movies” is a list of string movie titles. But we’re feeling lazy... so let’s do a **single string** that stores a **delimited list of movie titles!** This is similar to flattening a class, but instead we are flattening a *list* into a single value.

```
1, Neal Terrell, Office Space|Star Wars: The Empire Strikes Back|High Fidelity
2, Ada Terrell, Dumbo|Moana|Coco|PAW Patrol: The Movie
```

Problems with multi-value attributes

1. Choice of delimiter; can be tricky to choose a character that can't appear in the value domains, and harder to use escape characters.

2. Redundant data: how many times will we store "Star Wars: The Empire Strikes Back"? (Hint: a lot more often than "Star Wars: The Phantom Menace"...)

3. Very difficult to search. What if I want to search favorite films for "Empire"?
    a. String pattern matching is **very expensive.**

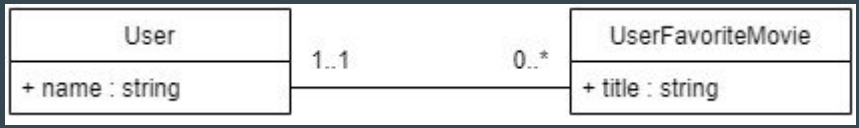
Designing for multi-valued attributes

The correct design for a multi-valued attribute is almost identical to repeated attributes: create a new class to store each value in the attribute, linked back to the “owner” class through an association.

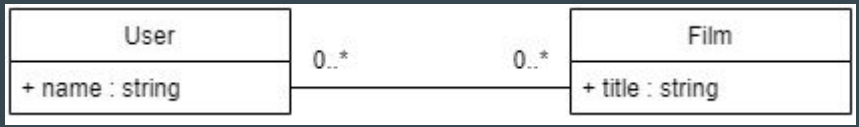

```
{
    "name" : "Neal",
    "favoriteMovies" : [
        {"title" : "Star Wars: The Empire Strikes Back"},
        {"title" : "Office Space"},
        {"title" : "High Fidelity"}
    ]
}
```

Any objections???

We can do better, of course...



Still has a **redundancy problem**... there will be thousands of UserFavoriteMovies objects with a title of “Star Wars: The Empire Strikes Back”...



A proper many-to-many association from Users to Films removes redundant movie titles and allows each user to have a list of favorite movies.

Recursive Associations

Definition

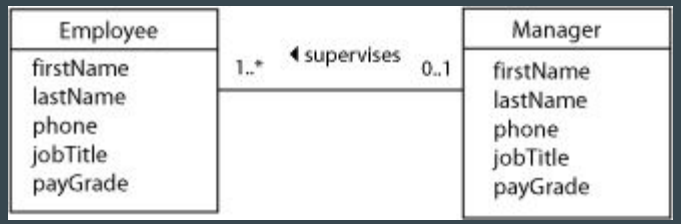
A recursive association connects one class to itself.

Examples:

- An Employee is managed by another Employee.
- A Product is a new model for another Product.
- A university Course has another Course as a prerequisite.

Incorrect models for recursion

An incorrect way to model recursion is to separate the recursive relationship into a new table. An Employee is managed by another employee... so add a Manager table!

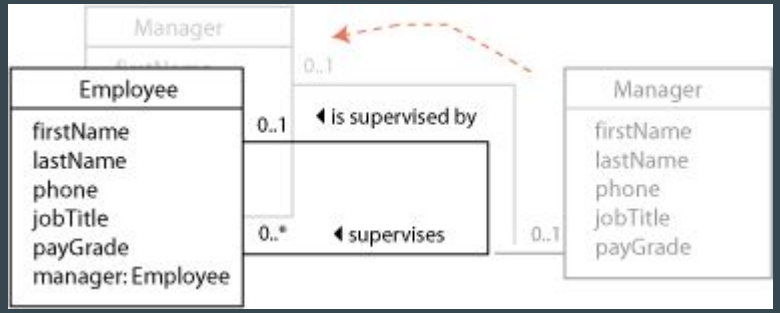


Problem: a Manager is also an Employee, so we have redundant data. (Roberta the Manager has a firstName, etc. in two different classes.)

```
{
    "managers" : [
        {
            "firstName" : "Bob",
            "lastName" : "Roberts",
            "employees" : [
                {
                    "firstName" : "Roberta",
                    "lastName" : "Roberts"
                },
                ...
            ]
        },
        {
            "firstName" : "Roberta",
            "lastName" : "Roberts",
            "employees" : [
                ...
            ]
        }
```

Correct models for recursion

Don't fear the recursion; *embrace it*. A Manager is not a separate type of entity from an Employee (no matter what they think); management is a relationship between two Employees.




```
{
    "employees" : [
        {
            "firstName" : "Bob",
            "lastName" : "Roberts",
            "minions" : [
                {
                    "firstName" : "Roberta",
                    "lastName" : "Roberts",
                    "minions" : [
                        ...
                    ]
                },
            ]
        }
    ]
}
```

Nothing to be afraid of!

Recursive many-to-many associations

One-to-one and one-to-many are not the only possible recursive associations. Suppose an Employee can have *many* managers...